

An Algebraic Specification of Process Algebra, including two examples

S. Mauw

Programming Research Group, University of Amsterdam,
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands.

Abstract A study is made of the possibilities to describe process algebra as an algebraic specification. Two examples from the field of the specification of communication protocols are discussed to analyse the adequacy of this approach in practical situations.

Note: This work was sponsored in part by ESPRIT contract nr. 432, Meteor.

1. Introduction

Both process algebra and algebraic specifications can be viewed as specification formalisms. Process algebra can be used to specify the behaviour of (concurrent) processes, while algebraic specifications can be used to specify abstract data types. It is an interesting question whether the combination of the two forms a more powerful tool for specification than each of them separately.

Process algebra is one of the theories developed in the last decade to describe concurrency (see e.g. [8]). Some alternative theories are CCS [19], CSP [17] and Petri-nets [20]. An important and growing topic of interest in the field of process algebra is the development of applications to significant and practical cases. These cases include e.g. the specification and verification of communication protocols and CAM-architectures, or the specification of programming languages. When applying process algebra new features are developed and added to process algebra, in order to make specification and verification easier. So process-algebra can be considered as a dynamic and modular theory.

Because the growth of specifications in practical cases leads to an increasing chance of being imprecise and making errors, software tools for the development of specifications and proofs are needed. This paper is meant to be a first step from the stage of manual specification in process algebra towards Computer Integrated Specification. This step is done by using algebraic specifications as a method to describe process algebra. When using algebraic specifications a specification is forced to be precise. The specification is written in a subset of the COLD language (see [15] and [18]). It was translated by hand from the ASF-language, which is described in [6] and [7]. Some features from ASF are added to COLD, in order to make the translation as simple as possible. The subset of COLD which is used is roughly equal to the complete ASF-language. ASF offers the possibility to define abstract data types, consisting of sorts, elements of these sorts and functions on these elements. The intended meaning of a function or a constant is given by a set of algebraic equations. As a semantics of algebraic specifications we will use the Initial Algebra semantics (see e.g. [14]). The question whether algebraic specification methods, with initial algebra semantics, are flexible and expressive enough to describe process algebra is discussed in this paper.

The combination of algebraic specifications with a concurrency theory is not a new

approach. The ISO specification language LOTOS [12] is a combination of ACT ONE and CCS. In LOTOS the concurrency theory is part of the language, while in this paper process algebra is described by means of the specification language.

This paper contains the following sections: After a brief description of the specification language, the basic modules `BOOLEANS`, `SEQUENCES`, `SETS` and `SUMMATION` are defined. Some alternative specifications of Sets are considered, and after that, the notion of an implicit definition is introduced. In the chapter about process algebra some building blocks of the theory are described, followed by their algebraic specification. At the end of this chapter some problematic notions are considered. The section about the protocols contains the description of the PAR and the OBSW protocol. Each protocol is specified by defining the appropriate set of atoms, providing the definition of the "parameters" and specifying all extra processes. In the final section various observations about the specification of process algebra are made.

I would like to express my thanks to Jos Baeten, Jan Bergstra, Wiet Bouma, Paul Klint, Pum Walters and Freek Wiedijk for their useful comments.

2. The Specification Language

The specifications are written in an enhanced subset of the COLD language (see [15] and [18]). The COLD constructs dealing with algebraic specification and modularization are all used. New features from ASF are listed below.

- We assume existence of a polymorphic function "if", with three arguments. The first is of sort `BOOL`, and the other two have the same unspecified type. The intended meaning of a term starting with the "if" is obvious: if the `BOOL` equals true, then the term equals its second argument; if the term equals false then it equals its third argument.

- We adopt the possibility to use infix and prefix operator-symbols. They of course can be viewed as shorthand for normal functional notation, but are needed because process algebra expression would be hard to read without them.

- All functions are considered to be total, so all terms are defined.

The intended semantics of the specification is the initial algebra semantics.

More about algebraic specifications, specification languages and initial algebra semantics can be found in [13, 14, 16].

3. Basic Modules

3.1. Booleans

The module `Booleans` describes the sort `BOOL`, with values `true` and `false`, and some boolean functions.

```

LET Booleans :=
CLASS
  SORT BOOL
  FUNC true  :          -> BOOL
  FUNC false :          -> BOOL
  FUNC _|_   : BOOL # BOOL -> BOOL      %or
  FUNC _&_   : BOOL # BOOL -> BOOL      %and
  FUNC not   : BOOL      -> BOOL
  FUNC eq    : BOOL # BOOL -> BOOL
AXIOM
  FORALL b:BOOL, b1:BOOL, b2:BOOL (
{Bo1}  true | b      = true;
{Bo2}  false | b     = b;
```

```

{Bo3}  true  & b      = b;
{Bo4}  false & b      = false;
{Bo5}  not (true)     = false;
{Bo6}  not (false)    = true;
{Bo7}  eq(b1,b2)      = (b1 & b2) | (not(b1) & not(b2)) )
END;

```

3.2. Sequences

In the specification of the OBSW-protocol a communication-channel is described which acts as a FIFO-queue. This can be modeled by indexing the process-variable with the sequence of items that is contained in the queue. The module `sequences` has a parameter for the sequenced items. The constant `eps` denotes the empty sequence, `seq` transforms an item into a one-item sequence and `_+_` concatenates two sequences.

```

LET Items :=
CLASS
  SORT ITEM FREE
END;
LET Sequences :=
LAMBDA Items_Parameter: Items OF
  IMPORT Items_Parameter INTO
  CLASS
    SORT SEQ
    FUNC eps :          -> SEQ
    FUNC seq : ITEM     -> SEQ
    FUNC _+_ : SEQ # SEQ -> SEQ
  AXIOM
    FORALL q:SEQ, q1:SEQ, q2:SEQ, q3:SEQ (
      {Sq1}  eps + q      = q;
      {Sq2}  q + eps     = q;
      {Sq3}  (q1+q2)+q3 = q1+(q2+q3) )
  END;

```

3.3. Sets

A notion which is frequently used in specifications is the notion of a set. Examples are the set of data-items, which can be read in, and the set of atoms to be encapsulated. The easiest way to form a set is just summing up all its elements. The constructor functions `∅` (the empty set) and `insert` (add an element to a set) can easily be specified. With these constructors, other set-operators can be specified using recursion. This approach has been used in e.g. [12]. The next specification is derived from the LOTOS standard library of data types (see section A.5.2 of [12]). The specification is rewritten in COLD. Functions that are of no interest are omitted, while some other functions are renamed. Note again that all functions are total.

```

LET Items :=
  IMPORT Booleans INTO
  CLASS
    SORT ITEM FREE
    FUNC eq : ITEM # ITEM -> BOOL
  END;
LET sets :=
LAMBDA Items_Parameter: Items OF
  IMPORT Items_Parameter INTO
  CLASS
    SORT SET

```



```

FUNC null-set:          -> SET
FUNC is-in   : SET # ITEM -> BOOL
FUNC delete  : SET # ITEM -> SET
FUNC insert  : SET # ITEM -> SET
FUNC union   : SET # SET  -> SET
FUNC inters  : SET # SET  -> SET
FUNC diff    : SET # SET  -> SET
FUNC incl    : SET # SET  -> BOOL
FUNC eq      : SET # SET  -> BOOL
AXIOM
FORALL s:SET, s1:SET, s2:SET, s3:SET, i:ITEM, j:ITEM (
  insert(insert(s, i), i)      = insert(s, i);
  insert(insert(s, i), j)      = insert(insert(s, j), i);
  is-in(s, i) = false =>
    delete(insert(s, i), i)    = s;
  is-in(s, i) = false =>
    delete(s, i)               = s;
  union(null-set, s)           = s;
  union(insert(s1, i), s2)     = insert(union(s1, s2), i);
  inters(null-set, s)          = null-set;
  inters(s1, s2)               = inters(s2, s1);
  is-in(s2, i) = true =>
    inters(insert(s1, i), s2) = insert(inters(s1, s2), i);
  is-in(s2, i) = false =>
    inters(insert(s1, i), s2) = inters(s1, s2);
  diff(s, null-set)           = s;
  diff(s1, s2)                 = diff(s1, inters(s1, s2));
  diff(s1, insert(s2, i))      = diff(delete(s1, i), delete(s2, i));
  is-in(null-set, i)           = false;
  is-in(insert(s, j), i)       = eq(i, j) | is-in(s, i)
  diff(s2, s1) = null-set =>
    incl(s1, s2)               = true;
  diff(s2, s1) = insert(s3, i) =>
    incl(s1, s2)               = false;
  eq(s1, s2)                   = incl(s1, s2) & incl(s2, s1) )
END;

```

It is easy to verify that all elements of the initial algebra can be written in the form:

$$\text{insert}(\text{insert}(\dots \text{insert}(\emptyset, x_1), \dots, x_{n-1}), x_n) \quad [n \geq 0]$$

Moreover, this specification forces every instance of a set to be specified using the \emptyset and `insert` constructor. Of course `union` and other operators may be used, but all elements of a set are eventually summed up one by one. This introduces difficulties when specifying large sets, or sets over sorts which are imported as a parameter. For example consider the set of all `Data`-elements, where the elements of the sort `Data` are not individually known. It is not possible to specify this set using an expression with repeated inserts, nor is there an alternative.

So the need for a stronger mechanism for constructing sets is apparent. Suppose the parameter `ITEM` is bound to the sort `Data`. If we consider the set

$$T = \{x \in \text{Data} \mid x = d_0 \vee x = d_1\}$$

then it should be equal to the set

$$S = \text{insert}(\text{insert}(\emptyset, d_0), d_1).$$

A specification in which the first form is supported should not only involve the Zermelo-Fraenkel axioms, but also a specification of first-order logic, including the notions of variables, formulas, etc. This would take several pages of specifications.

Another way to specify the set T is as follows:

$$\text{is-in}(T, x) = \text{eq}(x, d_0) \mid \text{eq}(x, d_1)$$

(where x is a variable of sort `Data`.)

Now the initial algebra of `Sets` contains an extra element T , which is not equal to the element S . The set T is defined implicitly, giving the value of some special function (`is-in`) on T . To find an equation which forces S and T to be equal, the notion of an implicit definition is investigated.

Given some existing specification, a new constant is said to be specified implicitly if instead of giving its intended meaning, only some characterizing properties of the constant are given. Some examples can clarify this definition.

Consider the sort N with successor (s) and zero (0). Without any equations the initial algebra of this signature equals \mathbb{N} , the nonnegative integers. Suppose a constant `one` is defined with equation

$$s(\text{one}) = s(s(0))$$

Now the *intended meaning* of `one` is the term $s(0)$, but in the initial algebra it is a new element, unequal to $s(0)$. To force these two elements to be equal the next equation should be included:

$$s(x) = s(y) \Rightarrow x = y$$

This equation justifies all implicit definitions of the form $s(\text{constant}) = \text{some_term}$. In general the equation

$$f(x) = f(y) \Rightarrow x = y$$

for some function f (or a composition of functions), makes implicit definitions of the form

$$f(\text{constant}) = \text{some_term}$$

possible. Notice that the function f has to be an injection and that the righthand term has to be in the range of f (relative to the initial algebra without the implicitly defined constant). Consider e.g. an extension of the previous specification with the predecessor-function, which is not injective:

$$\begin{aligned} p(0) &= 0 \\ p(s(x)) &= x. \end{aligned}$$

Now adding

$$p(x) = p(y) \Rightarrow x = y$$

results in $p(s(0)) = 0 = p(0)$, so $s(0) = 0$. The predecessor function can not be specified

implicitly.

Now return to sets. An implicit definition of a set looks like:

$$\text{is-in}(T, i) = \text{eq}(i, d_0) \mid \text{eq}(i, d_1)$$

Notice that in contrast to the previous example a variable i is needed, so the equation to add would look like:

$$\forall_i [(i \in X) = (i \in Y)] \Rightarrow X=Y.$$

Adding this equation to the specification is not possible if we want to use initial algebra semantics. This is because in general a specification with universal quantification has no initial algebra. Now there are two approaches to simulate this equation. The first one is by decrementing the range of the variable x to the class of closed terms. An initial algebra still can be defined, but COLD doesn't support this kind of expressions.

The second one is to recursively check the condition $(i \in X) = (i \in Y)$ for all elements in the sort, and conclude $X=Y$ if this succeeds. This is only possible if the sort is finite and some mechanism for summing up all elements is provided. A possible way is to indicate the first element and define for all elements a next element, yielding a kind of linked list. Define the predicate `last` so that it indicates if its argument is the last in the list. Now the equation can be simulated by:

$$\begin{aligned} \text{eq}(X, Y) &= \text{eq2}(X, Y, \text{firstItem}) \\ \text{eq2}(X, Y, i) &= \text{if}(\text{last}(i), \\ &\quad (i \in X) \Leftrightarrow (i \in Y), \\ &\quad \text{if}((i \in X) \Leftrightarrow (i \in Y), \\ &\quad \quad \text{eq2}(X, Y, \text{next}(i)), \\ &\quad \quad \text{false})) \\ \text{eq}(X, Y) &= \text{true} \Rightarrow X = Y \end{aligned}$$

When these equations are added, the sets T and S are equal. This is easily checked by looking at the definition of $\text{eq}(T, S)$.

So an equation of the form

$$\forall_i [f(i, X) = f(i, Y)] \Rightarrow X=Y$$

in general justifies implicit definitions like

$$f(i, \text{const}) = \text{some_term}(i).$$

Again the function f has to meet some injectivity criterion

$$\forall_{X \neq Y} \exists_i f(i, X) \neq f(i, Y)$$

and $\text{some_term}(i)$ has to be an element of the "range" of f

$$\exists_X \forall_i f(i, X) = \text{some_term}(i).$$

This technique can be used when dealing with datatypes, whose elements can be

defined in more then one way. When unifying these different ways, a user can choose the one which is the most appropriate for his purpose. So e.g. in working with matrices, the definition of the unity could be written as

```
U = mat(row(1,0,0,0),
        row(0,1,0,0),
        row(0,0,1,0),
        row(0,0,0,1))
```

or, using an implicit definition:

```
elt(U,i,j) = if(eq(i,j), 1, 0).
```

Now a simulation of the following equation has to be added:

$$\forall i,j [\text{elt}(X,i,j) = \text{elt}(Y,i,j)] \Rightarrow X=Y.$$

For sets the resulting specification looks like the following. Notice that the parameter `Items` must contain a `firstItem` and an enumerating function, called `next`. The last element of `Items` is determined as the unique element which is invariant under the `next`-function. Notice also that the definition of the various set operators is much more close to their mathematical definition then it would be when using a recursive definition, as used in the previous example from the LOTOS document.

[illegible]


```

eq2(s1, s2, next(i)),
false));
{S8}    eq(s1, s2) = true => s1=s2 )
END;
```

It is worth mentioning that in contrast to the specification without implicit definitions, this specification only works for sets over a finite domain. In the infinite case, the universal quantifier would range over an infinite domain, resulting in infinitely many conditions that have to be satisfied. Such a quantification can not be simulated, as the next example will show. Suppose the set A, with infinite domain D, is defined as:

```
(i ∈ A) = false.
```

Then a proof that in the initial algebra A equals the empty set, uses finitely many instances of this equation. Because D is infinite one can find some element d0 which is not encountered in these finite equations. Now define the set B as:

```
(i ∈ B) = if(i=d0, true, false).
```

Now the same proof can be used to prove that B also equals the empty set.

3.4. Summation

In process algebra summation over an indexed set of processes is frequently used, e.g. when reading in data, or defining the state-operator. Summation always takes place over a finite index set. If the index set is empty, the summation yields deadlock ($i(i(\delta))$), which is the neutral element for summation. The parameter *Items* is inherited from the imported sort *Sets*, and provides the sort *ITEM* with the known first/next structure as the sort of indexes. To define the operation of indexing a process, a kind of function space is required. A new sort (*FuncsToP*) consisting of functions from elements of the index-set to processes has to be defined. The most natural definition would look like:

```
SORT          FuncsToP : ITEM -> process.
```

Unfortunately this construction is not allowed. To solve this problem a new, initially empty sort *FuncsToP* has to be introduced, together with an application function from *FuncsToP* # *ITEM* to processes. If one wishes to construct some summation, a constant of sort *FuncsToP* has to be introduced, and for all elements of sort *ITEM*, the application of the function on this element must be specified. For examples of the use of *Sum*, see the module *SumsOverData*.

Notice that the definition of the function *Sum* resembles the definition of the function *eq* in the module *Sets*. Addition of processes and the special process $i(i(\delta))$ are provided by the imported module *Base*.

```

LET Sum :=                               %Summation of indexed processes
LAMBDA Items_Parameter2: Items OF
  IMPORT APPLY Sets TO Items_Parameter2 INTO
  IMPORT Base INTO
  EXPORT
    SORT FuncsToP,                       %Functions from ITEMs to processes
    FUNC Sum : SET # FuncsToP -> process, %summation
    FUNC app : FuncsToP # ITEM -> process  %application
FROM
```



```

CLASS
  SORT FuncsToP                                %Functions from ITEMS to processes
  FUNC Sum : SET # FuncsToP -> process          %summation
  FUNC app : FuncsToP # ITEM -> process          %application
  FUNC Sum2 : SET # FuncsToP # ITEM -> process

AXIOM
  FORALL set:SET, it:ITEM, f:FuncsToP (
    {Sul} Sum(set,f) = Sum2(set,f,firstItem);
    {Su2} Sum2(set,f,it) = if(eq(next(it),it),
                              if(is-in(set,it), app(f,it), i(i(delta))),
                              if(is-in(set,it),
                                  app(f,it) + Sum2(set,f,next(it)),
                                  Sum2(set,f,next(it)))) )

END;

```

4. Process Algebra

In this section some topics in process algebra will briefly be introduced and, if possible, an algebraic specification will be given. Process algebra is the study of processes, as described in e.g. [8, 11]. A process can be viewed as a list of (possible) activities that some actor (a computer e.g.) can perform. Each action is thought to be an indivisible unit, called atom.

4.1. Base

Let a finite set of atomic actions be given. Every atom is a process (injection i), and new processes can be created by application of the choice-operator ($_ + _$) and the sequencing-operator ($_ \cdot _$).

```

LET Base :=                                %Definition of the basic operations
  IMPORT Atoms INTO
CLASS
  SORT process
  FUNC i : Atoms -> process %embed Atoms in process
  FUNC  $\_ \cdot \_$  : process # process -> process
  FUNC  $\_ + \_$  : process # process -> process
END;

```

4.2. BPA

The axiom system BPA (Basic Process Algebra) provides some mathematical laws for processes.

```

A1     $x+y = y+x$ 
A2     $x+(y+z) = (x+y)+z$ 
A3     $x+x = x$ 
A4     $(x+y)z = xz+yz$ 
A5     $(xy)z = x(yz)$ 

```

```

LET BPA :=                                %Axioms for the system BPA
  IMPORT Atoms INTO
  IMPORT Base INTO
CLASS
AXIOM
  FORALL x:process, y:process, z:process (
    {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) )
END;

```

4.3. Delta

A new Atom `delta` can be introduced to denote the machine that is in a deadlock, unable to do anything at all. Equation A6 states that deadlock will be avoided if there are some alternatives left. Equation A7 shows that after a deadlock has occurred, nothing more can happen. The system BPA_{δ} can be obtained by importing both the modules `BPA` and `Delta`.

```

A6      x+δ = x
A7      δx = δ

```

In the section about the definition of the atoms the function `delta` will be declared. The atom δ will be denoted by `i(delta)`, so the process δ will be denoted by `i(i(delta))`.

```

LET Delta :=                                %Axioms for dead-lock
    IMPORT Base INTO
CLASS
AXIOM
    FORALL x:process (
{A6}    x+i(i(delta)) = x;
{A7}    i(i(delta)).x = i(i(delta)) )
END;

```

4.4. Encapsulation

With the encapsulation operator it is possible to control what atoms can be performed by a process. If the machine that the process is running on, lacks the possibility to do certain actions, this can be expressed using the encapsulation operator. It takes as its input a process and a set of atoms that should be encapsulated. Each atom from this set will be substituted by `delta`, so the process is forced to make an alternative choice if possible. The imported module `encapsset` can contain various sets of atoms. An example will be given in the sequel.

```

D1      ∂H(a) = a   if a ∉ H
D2      ∂H(a) = δ   if a ∈ H
D3      ∂H(x+y) = ∂H(x) + ∂H(y)
D4      ∂H(xy) = ∂H(x) · ∂H(y)

```

```

LET Encaps :=                                %Axioms for the encapsulation-operator
    IMPORT Booleans INTO
    IMPORT Atoms INTO
    IMPORT Base INTO
    IMPORT Delta INTO
    IMPORT encapsset INTO
CLASS
    FUNC d : SetsAtoms # process -> process
AXIOM
    FORALL a:Atoms, x:process, y:process, H:SetsAtoms (
{D1-2}  d(H,i(a)) = if(is-in(H,a), i(i(delta)), i(a));

```

```

{D3}    d(H,x+y) = d(H,x) + d(H,y);
{D4}    d(H,x.y) = d(H,x) . d(H,y)
END;

```

4.5. ACP

Two processes can run simultaneously. The new process created is called the merge of these processes, which is denoted by the operator \parallel . This operator is defined in terms of the leftmerge (\ll , in the specification denoted by $\backslash\backslash$) and the communication-operator ($|$). The first action of the leftmerge is the first atom of its left operand. Then the merge of the rest follows. Two processes can communicate if their first actions have the possibility to communicate. The communication of two atoms results in a new atom. For every application the resulting atom has to be defined separately. This must be done in the module *Commerge*, where a communication-function between atoms must be defined. In the first equation (CM0) of the module *Comm*, this function is extended to processes.

```

CM1    x||y = x||y + y||x + x|y
CM2    a||x = ax
CM3    ax||y = a(x||y)
CM4    (x+y)||z = x||z + y||z
CM5    (ax)|b = (a|b)x
CM6    a|(bx) = (a|b)x
CM7    (ax)|(by) = (a|b)(x||y)
CM8    (x+y)|z = x|z + y|z
CM9    x|(y+z) = x|y + x|z
C1     a|b = b|a
C2     (a|b)|c = a|(b|c)
C3     δ|a = δ

```

```

LET Comm := %Communication axioms (used in ACP)
IMPORT Atoms INTO
IMPORT Base INTO
IMPORT Delta INTO
IMPORT Commerge INTO
CLASS
  FUNC _||_ : process # process -> process %merge
  FUNC _\_\_ : process # process -> process %leftmerge
  FUNC _|_ : process # process -> process %Communication merge on processes
AXIOM
  FORALL a:Atoms, b:Atoms, d:Atoms, x:process, y:process, z:process (
{CM0}    i(a)|i(b) = i(a|b); %identify overloaded _|_
{CM1}    x||y = (x||y) + (y||x) + (x|y);
{CM2}    i(a)||x = i(a).x;
{CM3}    (i(a).x)||y = i(a).(x||y);
{CM4}    (x+y)||z = (x||z) + (y||z);
{CM5}    (i(a).x)|i(b) = (i(a)|i(b)).x;
{CM6}    i(a)|(i(b).x) = (i(a)|i(b)).x;
{CM7}    (i(a).x)|(i(b).y) = (i(a)|i(b)).(x|y);

```



```

{CM8}      (x+y) | z = (x | z) + (y | z);
{CM9}      x | (y+z) = (x | y) + (x | z);
{C1}       i(a) | i(b) = i(b) | i(a);
{C2}       (i(a) | i(b)) | i(d) = i(a) | i(b) | i(d));
{C3}       i(i(delta)) | i(a) = i(i(delta)) }
END;

```

The system ACP (Algebra of communicating processes, see e.g. [11]) is constructed from BPA, Delta, Encapsulation and Communication.

```

LET ACP :=                                %Axioms for the system ACP
    IMPORT BPA INTO
    IMPORT Delta INTO
    IMPORT Encaps INTO
        Comm
END;

```

4.6. ACP_{τ}

The special process `tau` denotes the silent step (see e.g. [9, 19]). It can be used to model internal actions. `Tau` is not an atom.

$$\begin{array}{ll} \text{T1} & x\tau = x \\ \text{T2} & \tau x + x = \tau x \\ \text{T3} & a(\tau x + y) = a(\tau x + y) + ax \end{array}$$

```

LET Tau :=                                %Axioms for the silent step
    IMPORT Atoms INTO
    IMPORT Base INTO
CLASS
    FUNC tau : -> process
AXIOM
    FORALL a:Atoms, x:process, y:process (
{T1}      x.tau = x;
{T2}      (tau.x) + x = tau.x;
{T3}      i(a).((tau.x)+y) = (i(a).((tau.x)+y)) + (i(a).x) )
END:

```

If one is only interested in some external actions of a process, the internal actions must be hidden. The abstraction-operator has as a parameter a set of atoms that are declared to be invisible. All these atoms are changed into τ .

A module `abstrset` must be provided to indicate what sets of atoms can be abstracted from.

$$\begin{array}{ll} \text{TI1} & \tau_I(\tau) = \tau \\ \text{TI2} & \tau_I(a) = a \quad \text{if } a \in I \\ \text{TI3} & \tau_I(a) = \tau \quad \text{if } a \in I \\ \text{TI4} & \tau_I(x+y) = \tau_I(x) + \tau_I(y) \\ \text{TI5} & \tau_I(xy) = \tau_I(x) \cdot \tau_I(y) \end{array}$$

```
LET Abstr := %Axioms for the abstraction-operator
```

```

IMPORT SetsAtoms INTO
IMPORT Tau INTO
IMPORT abstrset INTO
CLASS
  FUNC abstr : SetsAtoms # process -> process
AXIOM
  FORALL a:Atoms, x:process, y:process, I:SetsAtoms (
{TI1}   abstr(I,tau) = tau;
{TI2-3} abstr(I,i(a)) = if(is-in(I,a), tau, i(a));
{TI4}   abstr(I,x+y) = abstr(I,x) + abstr(I,y);
{TI5}   abstr(I,x.y) = abstr(I,x) . abstr(I,y) )
END;
```

The system ACP_τ (see e.g. [9]) combines ACP with abstraction.

```

TM1    $\tau \parallel x = \tau x$ 
TM2    $(\tau x) \parallel y = \tau(x \parallel y)$ 
TC1    $\tau | x = \delta$ 
TC2    $x | \tau = \delta$ 
TC3    $(\tau x) | y = x | y$ 
TC4    $x | (\tau y) = x | y$ 
DT     $\partial_H(\tau) = \tau$ 
```

```

LET ACP-Tau :=                                     %Axioms for the system ACP-Tau
  IMPORT ACP INTO
  IMPORT Tau INTO
  IMPORT Abstr INTO
CLASS
AXIOM
  FORALL x:process, y:process, H:SetsAtoms (
{TM1}   tau \ x = tau.x;
{TM2}   (tau.x) \ y = tau.(x \ y);
{TC1}   tau | x = i(i(delta));
{TC2}   x | tau = i(i(delta));
{TC3}   (tau.x) | y = x | y;
{TC4}   x | (tau.y) = x | y;
{DT}    d(H,tau) = tau )
END;
```

4.7. Standard Concurrency

In ACP_τ often some extra laws are assumed. These laws are called Standard Concurrency (see [9]).

```

SC1    $(x \parallel y) \parallel z = x \parallel (y \parallel z)$ 
SC2    $(x | ay) \parallel z = x | (ay \parallel z)$ 
SC3    $x | y = y | x$ 
SC4    $x \parallel y = y \parallel x$ 
SC5    $x | (y | z) = (x | y) | z$ 
```

$$\text{SC6} \quad x \parallel (y \parallel z) = (x \parallel y) \parallel z$$

```

LET SC :=                                     %standard concurrency for ACP-Tau
IMPORT Atoms INTO
IMPORT ACP-Tau INTO
CLASS
AXIOM
  FORALL x:process, y:process, z:process, a:Atoms (
{SC1}   (x\y)\z = x\ (y\z);
{SC2}   (x|(i(a).y))\z = x|((i(a).y)\z);
{SC3}   x|y = y|x;
{SC4}   x||y = y||x;
{SC5}   x|(y|z) = (x|y)|z;
{SC6}   x||(y||z) = (x||y)||z )
END;

```

4.8. ACP_θ

Instead of adding the silent step to ACP, forming ACP_τ , it is also possible to add priorities, forming the system ACP_θ (see e.g. [4]). Relative to a given partial order on the atoms, the operator θ determines which actions should be enabled or disabled. The smallest atoms in this partial order ($<$) have the lowest priority. The following equations define a partial order on the atoms, requiring that deadlock has the lowest priority of all atoms.

1. $\neg(a < a)$
2. $a < b \Rightarrow \neg(b < a)$
3. $a < b \wedge b < c \Rightarrow a < c$
4. $\delta < a$ (if $a \neq \delta$)

These laws are not included in the algebraic specification. When defining some specific partial order, one just has to meet these requirements.

Now, introducing one auxiliary operator (\triangleleft , in the specification denoted by $\$$), the priority operator can be defined. This \triangleleft operator deadlocks if its lefthand-side has lower priority than its righthand-side, else yields its lefthand-side. Now the priority operator can easily be defined.

- P1 $a \triangleleft b = a$ if $\neg(a < b)$
- P2 $a \triangleleft b = \delta$ if $a < b$
- P3 $x \triangleleft yz = x \triangleleft y$
- P4 $x \triangleleft (y+z) = (x \triangleleft y) \triangleleft z$
- P5 $xy \triangleleft z = (x \triangleleft z)y$
- P6 $(x+y) \triangleleft z = x \triangleleft z + y \triangleleft z$
- TH1 $\theta(a) = a$
- TH2 $\theta(xy) = \theta(x) \cdot \theta(y)$
- TH3 $\theta(x+y) = \theta(x) \triangleleft y + \theta(y) \triangleleft x$


```

LET Theta :=
  IMPORT Base INTO
  IMPORT PO INTO
  IMPORT Booleans INTO
  CLASS
    FUNC theta : process -> process
    FUNC _$_ : process # process -> process
  AXIOM
    FORALL a:Atoms, b:Atoms, x:process, y:process, z:process (
      {P1-2} i(a)$i(b) = if(sm(a,b), i(i(delta)), i(a));
      {P3} x $(y.z) = x $ y;
      {P4} x $(y+z) = (x$y)$z;
      {P5} (x.y) $ z = (x$z).y;
      {P6} (x+y) $ z = (x$z) + (y$z);
      {TH1} theta(i(a)) = i(a);
      {TH2} theta(x.y) = theta(x).theta(y);
      {TH3} theta(x+y) = (theta(x)$y) + (theta(y)$x)
    )
  END;

LET ACP-Theta :=
  IMPORT ACP INTO
  Theta
END;

```

4.9. State-Operator

The state operator (see e.g. [22]) can very well be used to describe a system with some kind of memory. This operator is indexed by an object. To each object a state has been assigned, which can change depending on the atomic action that is being performed. The effect of every action on an arbitrary state of an arbitrary object is defined by the effect-function. When encountering an atom in a certain state, the action-function determines which actions possibly can be executed. So this function yields a set of atoms. The resulting action that is actually chosen influences the outcome of the effect-function. The following equations define the state-operator, supposing that the set of Objects, the set of States and two functions are given which satisfy:

$$\text{action} : A \times \text{Objects} \times \text{States} \rightarrow \text{Pow}(A_\tau)$$

$$\text{effect} : A \times A_\tau \times \text{Objects} \times \text{States} \rightarrow \text{States}$$

$$\text{L1} \quad \Lambda_\sigma^m(\delta) = \delta$$

$$\text{L2} \quad \Lambda_\sigma^m(\tau) = \tau$$

$$\text{L3} \quad \Lambda_\sigma^m(\tau x) = \tau \cdot \Lambda_\sigma^m(x)$$

$$\text{L4} \quad \Lambda_\sigma^m(ax) = \sum_{b \in \text{action}(a, m, \sigma)} b \cdot \Lambda_{\text{effect}(a, b, m, \sigma)}^m(x)$$

$$\text{L5} \quad \Lambda_\sigma^m(x+y) = \Lambda_\sigma^m(x) + \Lambda_\sigma^m(y)$$

The imported modules Objects, States and ACTION-EFFECT can be viewed as parameters, determined by the application. Because summation is needed over the set of all atoms enriched with the process τ , a new (finite) sort Atoms-Tau is introduced. In this sort the atom pre-tau is defined, which, when considered as a process, should equal the process tau. See the definition of the atoms for the OBSW-protocol for more

information about the structure of Atoms-Tau.

```

LET Atoms-Tau :=
  IMPORT Booleans INTO
  IMPORT Tau INTO
CLASS
  SORT Atoms-Tau
  FUNC j      : Atoms      -> Atoms-Tau
  FUNC pre-tau :           -> Atoms-Tau
  FUNC i      : Atoms-Tau -> process

  FUNC eq      : Atoms-Tau # Atoms-Tau -> BOOL
  FUNC firstAT :           -> Atoms-Tau
  FUNC next    : Atoms-Tau   -> Atoms-Tau
  FUNC penultimateAT :       -> Atoms-Tau
AXIOM
  FORALL a:Atoms, x:Atoms-Tau, y:Atoms-Tau (
{AT1} i(j(a)) = i(a);           %identify identical processes
{AT2} i(pre-tau) = tau;

{ATf} firstAT = pre-tau;
{ATn1} next(pre-tau) = j(firstAtom);
{ATn2} next(j(a)) = j(next(a));
{ATp} penultimateAT = j(penultimateAtom);

{ATeq1} eq(penultimateAT, next(penultimateAT)) = false;
{ATeq2} eq(next(penultimateAT), penultimateAT) = false;
{ATeq3} eq(x, y) = false when eq(next(x), next(y)) = false;
{ATeq4} eq(x, x) = true )
END;

LET SumsOverAtoms-Tau :=
RENAME
  SORT SET      -> SetsOfAtoms-Tau,
  FUNC null-set -> AtomsT-null-set,
  FUNC FuncsToP -> FuncsAtoms-TauToP
IN
APPLY
  RENAME
    SORT ITEM      -> Atoms-Tau,
    FUNC eq        -> eq,
    FUNC firstItem -> firstAT,
    FUNC next      -> next
  IN Sum
TO
  IMPORT Atoms-Tau INTO
CLASS
  FUNC tau-set : -> SetsOfAtoms-Tau
AXIOM
  FORALL d:Data
{SuAT1} tau-set = insert (AtomsT-null-set, pre-tau)
END;

LET lambda := %Axioms for the extended state-operator
  IMPORT Objects INTO
  IMPORT States INTO
  IMPORT ACTION-EFFECT INTO
  IMPORT SumsOverAtoms-Tau INTO
CLASS

```

```

FUNC lambda : Objects # States # process -> process %state operator
FUNC FL      : Atoms # Objects # States # process -> FuncsAtoms-TauToP

AXIOM
  FORALL a:Atoms, at:Atoms-Tau, x:process, y:process, m:Objects, st:States(
{LA0}  app(FL(a, m, st, x), at) = i(at).lambda(m, EFFECT(a, at, m, st), x);
{LA1}  lambda(m, st, i(i(delta))) = i(i(delta));
{LA2}  lambda(m, st, tau)         = tau;
{LA3}  lambda(m, st, tau.x)       = tau.lambda(m, st, x);
{LA4}  lambda(m, st, i(a).x)      = Sum(ACTION(a, m, st), FL(a, m, st, x));
{LA5}  lambda(m, st, x+y)         = lambda(m, st, x) + lambda(m, st, y) )
END;

```

4.10. Some Problems

In this section some topics in process algebra are elaborated, which (seem to) have no algebraically specifiable counterpart. This implies that too few identifications are made, so the initial algebra of the resulting specification is too large.

The first kind of problems arises from the fact that all previously encountered notions worked on all processes. In the following notions the way in which a process is defined is taken into account. So some extra information is needed to identify a process, for example whether it is the solution of some guarded recursive equation. A new sort, the sort of guarded equations, has to be introduced and some way to form a process from an equation. An equation should be an object, instead of being an expression in the specification language. About such an object we must be able to determine if it is guarded, and if it contains abstraction. It is obvious that this approach leads to a counterintuitive specification. It violates the elegant thought that an axiom of process algebra can easily be represented by an algebraic equation.

The Recursive Specification Principle (RSP) is a rule that uses the way in which a process is defined (see e.g. [3]). It states that two processes that are both solutions of the same guarded recursive equation without abstraction, are equal. The notation $E(x, -)$ indicates that x is a solution of equation E .

$$\text{RSP} \quad \frac{E(x, -) \quad E(y, -)}{x=y} \quad (E \text{ guarded, no abstraction})$$

The only way to model this equation as an algebraic specification is to define some mechanism that deals with the notion of a guarded equation.

The Recursive Definition Principle (RDP) states that all guarded recursive equations without abstraction have a solution.

$$\text{RDP} \quad \frac{E \text{ guarded, no abstraction}}{\exists x \ E(x, -)}$$

The approach, of defining a process by introducing a new constant and giving some equation over this constant, satisfies this principle. In the initial algebra every recursively defined process exists as a (new) constant.

Using the Approximation Induction Principle (AIP) it is possible to identify two processes, if their projections are equal. The projection of a process is a simple notion

that can be specified easily.

$$\text{AIP} \quad \frac{\forall_{n \geq 1} \pi_n(x) = \pi_n(y) \quad E(x, -)}{x=y} \quad (E \text{ guarded, no abstraction})$$

This equation not only uses guardedness, it also has an infinite number of premises. This is not algebraically expressible. An attempt has been made to reduce the number of premises, obtaining a constructive form of AIP.

Often a set of Conditional Axioms (CA) is used to verify equalities in process algebra (see e.g. [2]). These axioms all depend on the alphabet function, which determines all atoms "present" in a process. On finite processes it is defined by:

1. $\alpha(\delta) = \emptyset$
2. $\alpha(\tau) = \emptyset$
3. $\alpha(\tau x) = \alpha(x)$
4. $\alpha(ax) = \{a\} \cup \alpha(x) \quad (a \in A)$
5. $\alpha(x+y) = \alpha(x) \cup \alpha(y)$

On infinite processes it is defined by:

6.
$$\frac{E(x, -)}{\alpha(x) = \bigcup_{n=1}^{\infty} \alpha(\pi_n(x))} \quad (E \text{ guarded, no abstraction})$$
7.
$$\frac{E(x, -)}{\alpha(\tau_I(x)) = \alpha(x) - I} \quad (E \text{ guarded, no abstraction})$$

The problem arises from the infinite union, which can not be modelled using an algebraic specification. Though the resulting set is finite, in general the alphabet of a process is undecidable. In [2] some results about alphabets are gathered.

If the alphabet function is presumed to exist, the Conditional Axioms can easily be specified. They look like:

$\text{CA1} \quad \frac{\alpha(x) \mid (\alpha(y) \cap H) \subseteq H}{\partial_H(x \parallel y) = \partial_H(x \parallel \partial_H(y))}$	$\text{CA2} \quad \frac{\alpha(x) \mid (\alpha(y) \cap I) = \emptyset}{\tau_I(x \parallel y) = \tau_I(x \parallel \tau_I(y))}$
$\text{CA3} \quad \frac{\alpha(x) \cap H = \emptyset}{\partial_H(x) = x}$	$\text{CA4} \quad \frac{\alpha(x) \cap I = \emptyset}{\tau_I(x) = x}$

$$\text{CA5} \quad \frac{H=H_1 \cup H_2}{\partial_H(x) = \partial_{H_1} \circ \partial_{H_2}(x)}$$

$$\text{CA6} \quad \frac{I=I_1 \cup I_2}{\tau_I(x) = \tau_{I_1} \circ \tau_{I_2}(x)}$$

$$\text{CA7} \quad \frac{H \cap I = \emptyset}{\tau_I \circ \partial_H(x) = \partial_H \circ \tau_I(x)}$$

Because it uses guardedness, the Cluster Fair Abstraction Rule (CFAR, see [22]) is not easily transformed to an algebraic specification. This will be stated without going into detail. Another function on processes, which is defined using an infinite union is the trace function.

This section will end by indicating how to solve some of these problems.

As mentioned some problems can be solved by introducing a new sort `equation`, with predicates `guarded`, `no_abstraction` and `has_solution(x)`. This approach would be less natural than the one used. Instead of transforming an equation in process algebra into an equation in COLD, it introduces the higher level objects "equation" and "variable".

Another possibility is adding all needed instances of some problematic laws by hand. This would be the same as rewriting (parts of) a known proof in the specification. All identifications that are not generated by the specified features should be added. This method seems not to add extra value to the specification. On the other hand it can serve as a check on type-correctness of the proof.

One way to solve these problems within the field of process algebra is to find more constructive counterparts of the mentioned laws. If this is not possible, due to e.g. undecidability, maybe some restricted form could be obtained, which only holds for simple (regular) processes.

5. The PAR-Protocol

5.1. Global Description

As an example of the use of the algebraic specification of process algebra two specifications of communication protocols are transferred to COLD. These specifications, and also their verifications, can be found in [22]. The first protocol, Positive Acknowledgement with Retransmission, as described in [21], consists of four components: a Sender (S), a Data transmission channel (K), a Receiver (R) and an Acknowledgement transmission channel (L).

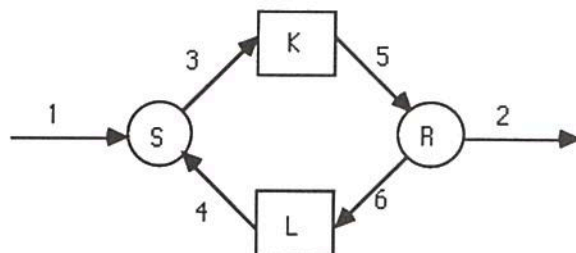


Fig. 1

The intended behaviour is that every data-element offered at port 1 eventually arrives, in the right order, at port 2, using the two unreliable channels K and L.

5.2. Atoms

Before specifying the process PAR, first the construction of the Atoms and auxiliary sorts must be given (See Fig. 2). The first sort is the sort of Ports. It consists of the elements p_1, \dots, p_6 . At every port an interaction between the two connected processes takes place. Therefore the sort Interaction-Types contains send (s), receive (r) and communicate (c).

The different types of data that can be transmitted over the channels are gathered in the sort D. It embeds the sorts Data and DB. The sort Data contains all data-elements that can arrive at port p_1 , and should be transmitted to port p_2 . This sort of elementary Data items should be provided by the environment and can be viewed as a "parameter" of the specified protocol.

Over the ports p_3 and p_5 frames (of sort DB) are communicated. These frames consist of a data-element and a boolean, which can be seen as an indicator for retransmission. At ports p_4 and p_6 , acknowledgements (ac) are communicated. To describe a mutilated message, a checksum-error (ce) can be communicated at ports p_4 and p_5 . This completes the description of all "interaction atoms", which together form sort D. There are some other atomic actions, all gathered in the sort Events: the internal actions i and j , the time-out action (tio) and the deadlock (delta). Now all atoms are either an event ($i(i), i(j), i(tio), i(delta)$) or have the form $do(int_type, port, d)$. All atoms can be viewed as processes, using the injection function i , which was defined in the module Base. The expression $i(i(i))$ is correct and denotes the internal process i . The function-symbol i is overloaded.

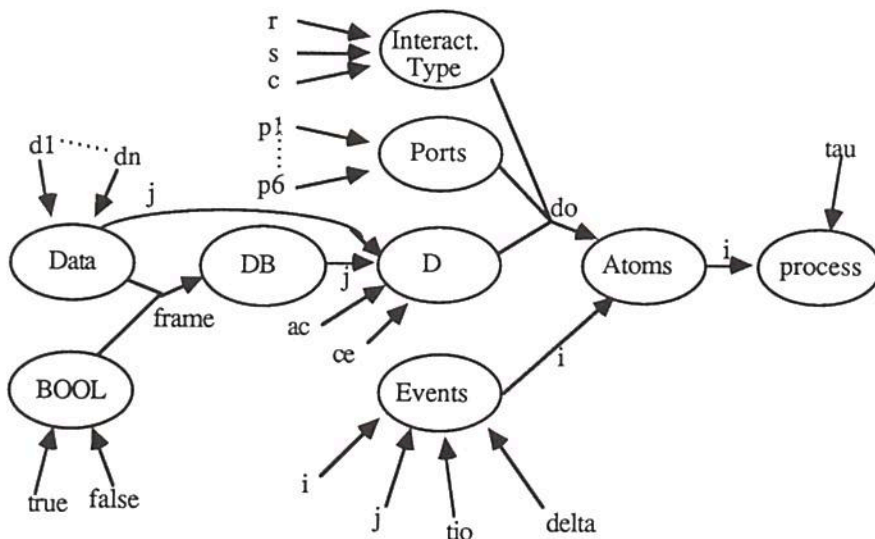


Fig. 2

Notice that for a proper definition of equality on the Atoms, as indicated in the section about sets, all auxiliary sorts are provided with the same first/next/penultimate structure.


```

LET Data :=                %Definition of the set of data to be transmitted
    IMPORT Booleans INTO
CLASS
    SORT Data
    FUNC d1 : -> Data          %three sample data-elements
    FUNC d2 : -> Data
    FUNC d3 : -> Data

    FUNC eq                : Data # Data -> BOOL
    FUNC firstDatum        :          -> Data
    FUNC next               : Data      -> Data
    FUNC penultimateDatum  :          -> Data
AXIOM
    FORALL x:Data, y:Data (
{Daf}   firstDatum = d1;
{Dan1}  next(d1) = d2;
{Dan2}  next(d2) = d3;
{Dan3}  next(d3) = d3;
{Dap}   penultimateDatum = d2;

{Daeq1} eq(penultimateDatum, next(penultimateDatum)) = false;
{Daeq2} eq(next(penultimateDatum), penultimateDatum) = false;
{Daeq3} eq(next(x), next(y)) = false => eq(x, y) = false;
{Daeq4} eq(x, x) = true )
END;

LET DB :=
    IMPORT Data INTO
CLASS
    SORT DB
    FUNC frame          : Data # BOOL -> DB
    FUNC eq              : DB # DB    -> BOOL
    FUNC firstDB         :          -> DB
    FUNC next            : DB         -> DB
AXIOM
    FORALL x:DB, y:DB, d:Data, n:BOOL (
{DBf}   firstDB = frame(firstDatum, false);
{DBn1}  next(frame(d,n)) = if(not(eq(next(d),d)) & eq(n,true),
                                frame(next(d),false),
                                frame(d,true));
{DBp}   penultimateDB = frame(next(penultimateDatum), false);

{DBeq1} eq(penultimateDB, next(penultimateDB)) = false;
{DBeq2} eq(next(penultimateDB), penultimateDB) = false;
{DBeq3} eq(next(x), next(y)) = false => eq(x, y) = false;
{DBeq4} eq(x, x) = true )
END;

LET Events :=
    IMPORT Booleans INTO
CLASS
    SORT Events
    FUNC delta : -> Events          %deadlock
    FUNC tio   : -> Events          %timeout
    FUNC i     : -> Events          %internal action
    FUNC j     : -> Events          %internal action

    FUNC eq                : Events # Events -> BOOL
    FUNC firstEvent        :          -> Events
    FUNC next              : Events      -> Events

```

```

FUNC penultimateEvent :                -> Events
AXIOM
  FORALL x:Events, y:Events (
    {Evf}   firstEvent = delta;
    {Evn1}  next(delta) = tio;
    {Evn2}  next(tio) = i;
    {Evn3}  next(i) = j;
    {Evn4}  next(j) = j;
    {Evp}   penultimateEvent = i;

    {Eveq1} eq(penultimateEvent, next(penultimateEvent)) = false;
    {Eveq2} eq(next(penultimateEvent), penultimateEvent) = false;
    {Eveq3} eq(next(x), next(y)) = false => eq(x, y) = false;
    {Eveq4} eq(x, x) = true )
  END;

LET InteractionType :=
  IMPORT Booleans INTO
  CLASS
    FUNC r : -> IntType                %receive
    FUNC s : -> IntType                %send
    FUNC c : -> IntType                %communicate

    FUNC eq          : IntType # IntType -> BOOL
    FUNC firstType   :                -> IntType
    FUNC next        : IntType          -> IntType
    FUNC penultimateIntType :          -> IntType
  AXIOM
    FORALL x:IntType, y:IntType (
      {ITf}   firstType = r;
      {ITn1}  next(r) = s;
      {ITn2}  next(s) = c;
      {ITn3}  next(c) = c;
      {ITp}   penultimateIntType = s;

      {ITeq1} eq(penultimateIntType, next(penultimateIntType)) = false;
      {ITeq2} eq(next(penultimateIntType), penultimateIntType) = false;
      {ITeq3} eq(next(x), next(y)) = false => eq(x, y) = false;
      {ITeq4} eq(x, x) = true )
    END;

LET Ports :=
  IMPORT Booleans INTO
  CLASS
    SORT Ports
    FUNC p1:Ports
    FUNC p2:Ports
    FUNC p3:Ports
    FUNC p4:Ports
    FUNC p5:Ports
    FUNC p6:Ports
    FUNC internal : Ports -> BOOL

    FUNC eq          : Ports # Ports -> BOOL
    FUNC firstP      :                -> Ports
    FUNC next        : Ports          -> Ports
    FUNC penultimateP :                -> Ports
  AXIOM
    FORALL x:Ports, y:Ports (
      {Po1}   internal(x) = not(eq(x,p1) | eq(x,p2));

```

```

f)   firstP   = p1;
n1)  next(p1) = p2;
n2)  next(p2) = p3;
n3)  next(p3) = p4;
n4)  next(p4) = p5;
n5)  next(p5) = p6;
n6)  next(p6) = p6;
op)  penultimateP = p5;

oeq1) eq(penultimateP, next(penultimateP)) = false;
oeq2) eq(next(penultimateP), penultimateP) = false;
oeq3) eq(next(x), next(y)) = false => eq(x, y) = false;
'oeq4) eq(x, x) = true )
ID;

ET D :=
  IMPORT Booleans INTO
  IMPORT DB INTO
  CLASS
    SORT D
    FUNC ac :      -> D          %acknowledge
    FUNC ce :      -> D          %checksum error
    FUNC j  : Data -> D          %embed Dataset
    FUNC j  : DB   -> D          %embed DB

    FUNC eq      : D # D -> BOOL
    FUNC firstD  :      -> D
    FUNC next    : D      -> D
    FUNC penultimateD :    -> D
  AXIOM
    FORALL x:D, y:D, d:Data, f:DB (
  {Df}   firstD = ac;
  {Dn1}  next(ac) = ce;
  {Dn2}  next(ce) = j(firstDatum);
  {Dn3}  next(j(d)) = if(eq(next(d), d), j(firstDB), j(next(d)));
  {Dn4}  next(j(f)) = j(next(f));
  {Dp}   penultimateD = j(penultimateDB);

  {Deq1} eq(penultimateD, next(penultimateD)) = false;
  {Deq2} eq(next(penultimateD), penultimateD) = false;
  {Deq3} eq(next(x), next(y)) = false => eq(x, y) = false;
  {Deq4} eq(x, x) = true )
  END;

LET Atoms :=          %Definition of the Atoms
  IMPORT Events INTO
  IMPORT InteractionType INTO
  IMPORT Ports INTO
  IMPORT D INTO
  CLASS
    SORT Atoms
    FUNC i      : Events          -> Atoms %embed Events
    FUNC do     : IntType # Ports # D -> Atoms
    FUNC has-type : IntType # Atoms -> BOOL
    FUNC port    : Atoms          -> Ports %what port is involved?
    FUNC datum   : Atoms          -> D    %and what datum?

    FUNC eq      : Atoms # Atoms -> BOOL
    FUNC firstAtom :      -> Atoms

```

```

FUNC next          : Atoms          -> Atoms
FUNC penultimateAtom :              -> Atoms
AXIOM
  FORALL x:Atoms, y:Atoms, e:Events, t:IntType, t1:IntType, t2:IntType,
    p:Ports, d:D (
{At1}   has-type(t,i(e)) = false;
{At2}   has-type(t1,do(t2,p,d)) = eq(t1,t2);

{At3}   port(i(e)) = firstP;          %default value
{At4}   port(do(t,p,d)) = p;
{At5}   datum(i(e)) = firstD;         %default value
{At6}   datum(do(t,p,d)) = d;

{Atf}   firstAtom      = i(firstEvent);
{Atn1}  next(i(e))      = if(eq(next(e),e),
                             do(firstType, firstP, firstD),
                             i(next(e)));
{Atn2}  next(do(t,p,d)) = if(not(eq(next(t),t)),
                             do(next(t),p,d),
                             if(not(eq(next(p),p)),
                                 do(firstType,next(p),d),
                                 if(not(eq(next(d),d)),
                                     do(firstType,firstP,next(d)),
                                     do(t,p,d) )));
{Atp}   penultimateAtom = do(penultimateIntType, next(penultimateP),
                             next(penultimateD));

{Ateq1} eq(penultimateAtom, next(penultimateAtom)) = false;
{Ateq2} eq(next(penultimateAtom), penultimateAtom) = false;
{Ateq3} eq(next(x), next(y)) = false => eq(x, y) = false;
{Ateq4} eq(x, x) = true )
END;

```

5.3. Sets and Summations

The following sets and summations are needed:

```

LET SetsAtoms :=          %Sets of Atoms
RENAME
  SORT SET      -> SetsAtoms,
  FUNC null-set -> Atoms-null-set
IN
APPLY
  RENAME
    SORT ITEM    -> Atoms,
    FUNC eq      -> eq,
    FUNC firstItem -> firstAtom,
    FUNC next     -> next
  IN Sets
TO Atoms

```

```

LET SumsOverData :=
RENAME
  SORT SET      -> SetsOfData,
  FUNC null-set -> Data-null-set,
  FUNC FuncsToP -> FuncsDataToP
IN
APPLY
  RENAME

```



```

        SORT ITEM      -> Data,
        FUNC eq        -> eq,
        FUNC firstItem -> firstDatum,
        FUNC next      -> next
    IN Sum
TO
    IMPORT Data INTO
CLASS
    FUNC DataSet : -> SetsOfData      %The set of all Data-elements
AXIOM
    FORALL d:Data
    {SuDal} is-in(DataSet,d) = true
END;

LET SumsOverDB :=
RENAME
    SORT SET      -> SetsOfDB,
    FUNC null-set -> DB-null-set,
    FUNC FuncsToP -> FuncsDBToP
IN
APPLY
    RENAME
        SORT ITEM      -> DB,
        FUNC eq        -> eq,
        FUNC firstItem -> firstDB,
        FUNC next      -> next
    IN Sum
TO
    IMPORT DB INTO
CLASS
    FUNC DBSet : -> SetsOfDB      %The set of all DBB-elements
AXIOM
    FORALL d:DB
    {SuDB1} is-in(DBSet,d) = true
END;

```

5.4. PAR

Now the processes K, L, S and R are defined:

The channel K waits for input of a frame at port p3. Then three things can happen:

- (i) The frame is sent on correctly.
- (ii) The frame is damaged, and a checksum-error is communicated at port p5
- (iii) The frame is lost, indicated by the occurrence of the internal action i.

This leads to the following equations (cf. equations {PAR5, 6} in the module PARPart1):

$$\begin{aligned}
 K &= \sum_{f \in DB} r3(f) \cdot K^f \\
 K^f &= (s5(f) + s5(ce) + i) \cdot K \quad (f \in DB)
 \end{aligned}$$

The specification for channel L is almost identical, except that the frames are replaced by the acknowledgement atom (cf. equations {PAR7, 8} in the module PARPart1):

$$\begin{aligned}
 L &= r6(ac) \cdot L^{ac} \\
 L^{ac} &= (s4(ac) + s4(ce) + j) \cdot K
 \end{aligned}$$

The sender S is specified using some auxiliary functions. It reads a data-element (d) from port $p1$, and sends this element, enriched with some boolean information (n) at port $p3$. This extra bit enables the receiver to deal with retransmissions, it flips to distinguish successive messages. The process RH^n reads a message (d) from the host. The process SF^{dn} sends frame (d, n) at port $p3$. The process WS^{dn} waits for something to happen. It can receive an acknowledgement, and after that the sequence can start all over. It can receive a checksum-error, which indicates a failure in the communication, and should be followed by a retransmission. Or, if none of these two events are offered, a time-out occurs, also followed by a retransmission.

$$\begin{aligned} S &= RH^0 \\ RH^n &= \sum_{d \in D} r1(d) \cdot SF^{dn} \\ SF^{dn} &= s3(dn) \cdot WS^{dn} & (d \in \text{Data}, n \in \{0, 1\}) \\ WS^{dn} &= r4(ac) \cdot RH^{1-n} + (r4(ce) + tio) \cdot SF^{dn} \end{aligned}$$

The receiver R is also defined using extra variables. The process WF^n waits for the arrival of a frame at port $p5$. If a new frame arrives (as indicated by the extra bit n), the data-element has to be transmitted to the host at port $p2$ (SH^{dn}), followed by the transmission of an acknowledgement at port $p6$. There is also a possibility that, due to malfunction of the acknowledgement channel L , a retransmission of the previously arrived frame occurs. Then an acknowledgement has to be transmitted again. If a checksum-error arrives, the receiver just waits until the timer of the sender elapses, resulting in a retransmission.

$$\begin{aligned} R &= WF^0 \\ WF^n &= r5(ce) \cdot WF^n + \sum_{d \in D} r5(d, 1-n) \cdot SA^n + \sum_{d \in D} r5(d, n) \cdot SH^{dn} \\ SA^n &= s6(ac) \cdot WF^n \\ SH^{dn} &= s2(d) \cdot SA^{1-n} & (d \in \text{Data}, n \in \{0, 1\}) \end{aligned}$$

The communication function is defined by:

$$st(f) | rt(f) = ct(f) \text{ for } t \in \{3, 4, 5, 6\}, f \in D$$

```

LET Commerge :=          %Definition of the communication-merge function
  IMPORT Atoms INTO
  CLASS
    FUNC _|_ : Atoms # Atoms -> Atoms          %communication merge on Atoms
  AXIOM
    FORALL a:Atoms, b:Atoms
      {Com1} a|b = if((has-type(s,a) & has-type(r,b))
                     | (has-type(r,a) & has-type(s,b)),
                     if(eq(port(a),port(b)) & eq(datum(a),datum(b))
                       & internal(port(a)),
                       do(c, port(a), datum(a)),
                       i(delta)),
                     i(delta))
END;
```

All unsuccessful communications are encapsulated:

$$H_0 = \{st(f), rt(f) \mid t \in \{3, 4, 5, 6\}, f \in D\}$$

```

LET encapsset :=
  IMPORT SetsAtoms INTO
  CLASS
    FUNC H0 : -> SetsAtoms
  AXIOM
    FORALL a:Atoms
      {encl} is-in(H0, a) = if(has-type(r,a) | has-type(s,a),
                              internal(port(a)),
                              false)
END;

```

A priority is defined, to manage the time-outs. By giving time-out the lowest priority, it only occurs when no alternatives are offered, so successful communication is not timed-out.

$$\delta < a \quad \text{for } a \in A - \{\delta\}$$

$$tio < a \quad \text{for } a \in A - \{tio, \delta\}$$

```

LET PO :=                                %partial order on Atoms
  IMPORT Atoms INTO
  CLASS
    FUNC sm : Atoms # Atoms -> BOOL      %smaller
  AXIOM
    FORALL a:Atoms, b:Atoms
      {PO1} sm(a, b) = (eq(a, i(delta)) & not(eq(b, i(delta))))
                       | (eq(a, i(tio)) & not(eq(b, i(delta))) & not(eq(b, i(tio))))
END;

```

Furthermore only the external behaviour at ports p1 and p2 is of interest. We abstract from the internal actions:

$$I_0 = \{ct(f) \mid t \in \{3, 4, 5, 6\}, f \in D\} \cup \{tio, i, j\}$$

```

LET abstrset :=
  IMPORT SetsAtoms INTO
  CLASS
  AXIOM
    FORALL a:Atoms
      {abs1} is-in(I0, a) = eq(a, i(tio)) | eq(a, i(i)) | eq(a, i(j)) |
                           if(has-type(c,a), internal(port(a)), false)
END;

```

Now the PAR-protocol is described by:

$$PAR = \tau_{I_0} \circ \theta \circ \partial_{H_0} (S \parallel K \parallel R \parallel L)$$

Because the systems ACP_t and ACP_θ are not yet integrated into one single system, application of the theta-operator and of the abstraction-operator should be separated in two different modules. In the first module the theta-operator is applied, and in the second module the abstraction-operator. When importing the first module in the second, the theta-operator is hidden.

```

LET PARPart1 :=
EXPORT
  FUNC pre-PAR :      -> process
FROM
  IMPORT Atoms INTO
  IMPORT Base INTO
  IMPORT SumsOverData INTO
  IMPORT SumsOverDB INTO
  IMPORT ACP-Theta INTO
CLASS
  FUNC K      :      -> process
  FUNC K      : DB   -> process
  FUNC L      :      -> process
  FUNC L-ac   :      -> process
  FUNC S      :      -> process
  FUNC RH     : BOOL -> process
  FUNC SF     : DB   -> process
  FUNC WS     : DB   -> process
  FUNC R      :      -> process
  FUNC WF     : BOOL -> process
  FUNC SA     : BOOL -> process
  FUNC SH     : DB   -> process
  FUNC pre-PAR :      -> process
  FUNC FunK    :      -> FuncsDBToP
  FUNC FunRH   : BOOL -> FuncsDataToP
  FUNC FunWFa  : BOOL -> FuncsDataToP
  FUNC FunWFb  : BOOL -> FuncsDataToP
AXIOM
  FORALL f:DB, d:Data, n:BOOL (
{PAR1}  app(FunK,f)      = i(do(r,p3,j(f))).K(f);
{PAR2}  app(FunRH(n),d)  = i(do(r,p1,j(d))).SF(frame(d,n));
{PAR3}  app(FunWFa(n),d) = i(do(r,p5,j(frame(d,not(n))))).SA(n);
{PAR4}  app(FunWFb(n),d) = i(do(r,p5,j(frame(d,n))))).SH(frame(d,n));

{PAR5}  K                = Sum(DBSet, FunK);
{PAR6}  K(f)             = (i(do(s,p5,j(f))) + i(do(s,p5,ce)) + i(i(i))) . K;

{PAR7}  L                = i(do(r,p6,ac)) . L-ac;
{PAR8}  L-ac            = (i(do(s,p4,ac)) + i(do(s,p4,ce)) + i(i(j))) . L;

{PAR9}  S                = RH(false);
{PAR10} RH(n)            = Sum(DataSet, FunRH(n));
{PAR11} SF(f)           = i(do(s,p3,j(f))) . WS(f);
{PAR12} WS(frame(d,n))  = i(do(r,p4,ac)).RH(not(n)) +
                        (i(do(r,p4,ce))+i(i(tio))).SF(frame(d,n));

{PAR13} R                = WF(false);
{PAR14} WF(n)            = i(do(r,p5,ce)).WF(n) +
                        Sum(DataSet, FunWFa(n)) +
                        Sum(DataSet, FunWFb(n));
{PAR15} SA(n)           = i(do(s,p6,ac)).WF(n);
{PAR16} SH(frame(d,n))  = i(do(s,p2,j(d))).SA(not(n));

{PAR17} pre-PAR         = theta(d(H0, (S||K||R||L))) )
END;

LET PARPart2 :=
  IMPORT ACP-Tau INTO
  IMPORT PARPart1 INTO

```



```

CLASS
  FUNC PAR : -> process
AXIOM
{PAR18} PAR = abstr(I0, pre-PAR)
END;

```

6. The OBSW-protocol

6.1. Global Description

The One Bit Sliding Window protocol is a bit more complicated. In contrast to the PAR-protocol, it allows communications in two directions. The process algebra specification is derived by Vaandrager in [22] from a computer program, described in [21]. Using the State Operator, all constructions in the program can be translated to process algebra.

The structure of the system can be visualized as follows:

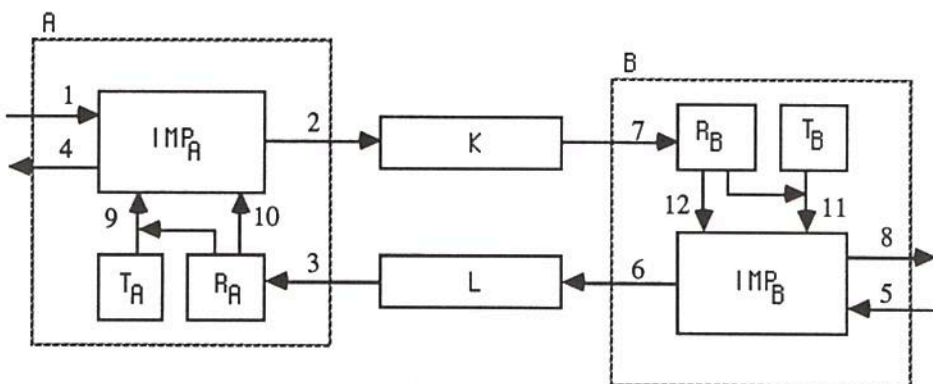


Fig. 3

The two systems A and B communicate using the two channels K and L. Both systems consist of a Timer (T), a Receiver (R) and an Interface Message Processor (IMP), which is the process implementing the computer program.

6.2. Atoms

The description of the way the atoms are built up is analogous to that of the PAR-protocol. Again the interaction-atoms are constructed using three sorts: an Interaction Type (r, s, c), Ports (p_1, \dots, p_{12}), and a sort D, containing the items that are communicated. Such an item can be a single Data-element (at ports p_1, p_4, p_5, p_8), a data-frame (from DBB), which is some Data-element enriched with two bits of auxiliary information (at ports $p_2, p_3, p_6, p_7, p_{10}, p_{12}$), or it can be one of the events (time-out and frame-arrival at ports p_9 and p_{11} , checksum-error at ports $p_9, p_{10}, p_{11}, p_{12}$). Again there is the sort of Simple Atoms, which do not communicate. It contains the internal actions i and j , the deadlock (δ), and for each "action" in the computer program a corresponding atom, which will be interpreted by the State-operator. Because the three events (ce, tio and fa) are also atomic actions in the computer program, the sort Events is also embedded in the sort Simple Atoms.

The introduction of the extra sort Atoms-Tau is due to the State-Operator, which uses summation over this set. So instead of using the infinite sort *process*, a new finite

sort has to be defined.

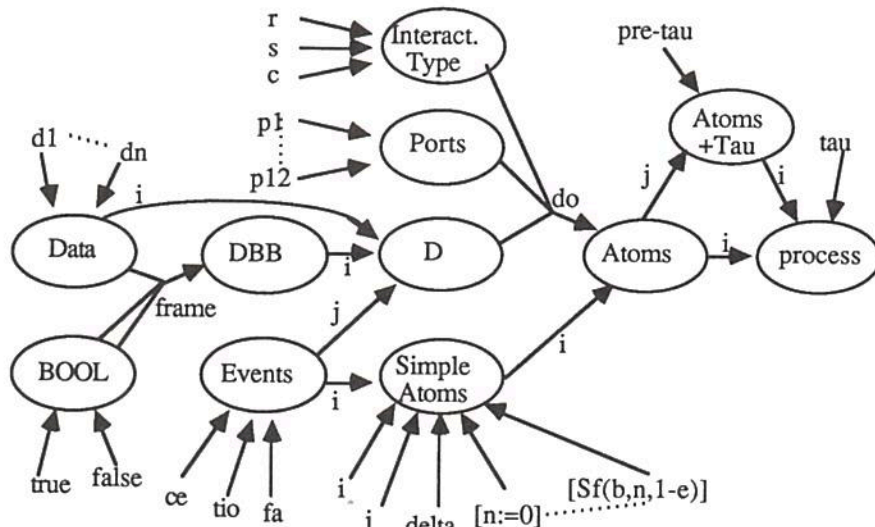


Fig. 4

The modules *Data* and *IntType* are equal to the ones defined for the PAR-protocol, so they are not copied.

```

LET DBB :=                %Definition of the set of data to be transmitted
  IMPORT Data INTO
  CLASS
    SORT DBB
    FUNC frame : Data # BOOL # BOOL -> DBB

    FUNC eq      : DBB # DBB -> BOOL
    FUNC firstDBB :      -> DBB
    FUNC next     : DBB   -> DBB
    FUNC penultimateDBB : -> DBB

  AXIOM
    FORALL x:DBB, y:DBB, d:Data, n1:BOOL, n2:BOOL (
      {DBf} firstDBB = frame(firstDatum, false, false);
      {DBn1} next(frame(d, n1, n2)) = if(eq(n2, false),
                                         frame(d, n1, true),
                                         if(eq(n1, false),
                                             frame(d, true, false),
                                             frame(next(d), false, false)));
      {DBp} penultimateDBB = frame(next(penultimateDatum), false, true);

      {DBeq1} eq(penultimateDBB, next(penultimateDBB)) = false;
      {DBeq2} eq(next(penultimateDBB), penultimateDBB) = false;
      {DBeq3} eq(next(x), next(y)) = false => eq(x, y) = false;
      {DBeq4} eq(x, x) = true )
    END;

LET Events :=
  IMPORT Booleans INTO
  CLASS
    SORT Events

```

```

FUNC tio : -> Events           %timeout
FUNC ce  : -> Events           %checksum error
FUNC fa  : -> Events           %frame arrival

FUNC eq      : Events # Events -> BOOL
FUNC firstEvent :           -> Events
FUNC next    : Events       -> Events
FUNC penultimateEvent :     -> Events
AXIOM
  FORALL x:Events, y:Events (
    {Evf} firstEvent = tio;
    {Evn1} next(tio) = ce;
    {Evn2} next(ce) = fa;
    {Evn3} next(fa) = fa;
    {Evp} penultimateEvent = ce;

    {Eveq1} eq(penultimateEvent, next(penultimateEvent)) = false;
    {Eveq2} eq(next(penultimateEvent), penultimateEvent) = false;
    {Eveq3} eq(next(x), next(y)) = false => eq(x, y) = false;
    {Eveq4} eq(x, x) = true )
  END;

LET Ports := %Definition of the set of Dataports
  IMPORT Booleans INTO
  CLASS
    SORT Ports
    FUNC p1:Ports
    FUNC p2:Ports
    FUNC p3:Ports
    FUNC p4:Ports
    FUNC p5:Ports
    FUNC p6:Ports
    FUNC p7:Ports
    FUNC p8:Ports
    FUNC p9:Ports
    FUNC p10:Ports
    FUNC p11:Ports
    FUNC p12:Ports

    FUNC eq      : Ports # Ports -> BOOL
    FUNC firstP  :           -> Ports
    FUNC next    : Ports     -> Ports
    FUNC penultimateP :     -> Ports
  AXIOM
    FORALL x:Ports, y:Ports (
      {Pof} firstP = p1;
      {Pon1} next(p1) = p2;
      {Pon2} next(p2) = p3;
      {Pon3} next(p3) = p4;
      {Pon4} next(p4) = p5;
      {Pon5} next(p5) = p6;
      {Pon6} next(p6) = p7;
      {Pon7} next(p7) = p8;
      {Pon8} next(p8) = p9;
      {Pon9} next(p9) = p10;
      {Pon10} next(p10) = p11;
      {Pon11} next(p11) = p12;
      {Pon12} next(p12) = p12;
      {Pop} penultimateP = p11;
    )
  END

```

```

{Poeq1} eq(penultimateP, next(penultimateP)) = false;
{Poeq2} eq(next(penultimateP), penultimateP) = false;
{Poeq3} eq(next(x), next(y)) = false => eq(x, y) = false ;
{Poeq4} eq(x, x) = true )
END;

```

```
LET D :=
```

```

    IMPORT DBB INTO
    IMPORT Events INTO

```

```
CLASS
```

```
    SORT D
```

```

    FUNC j          : Events -> D          %embed Events
    FUNC i          : DBB    -> D          %embed DBB
    FUNC i          : Data   -> D          %embed Dataset

```

```

    FUNC originDBB : D      -> BOOL        %Is its origin DBB?
    FUNC originData : D     -> BOOL        %Is its origin Data?

```

```

    FUNC eq        : D # D -> BOOL
    FUNC firstD    :      -> D
    FUNC next      : D     -> D
    FUNC penultimateD :    -> D

```

```
AXIOM
```

```

    FORALL x:D, y:D, e:Events, dbb:DBB, d:Data (
{Dor1} originDBB(i(dbb)) = true;
{Dor2} originDBB(i(d))   = false;
{Dor3} originDBB(j(e))   = false;
{Dor4} originData(i(dbb)) = false;
{Dor5} originData(i(d))   = true;
{Dor6} originData(j(e))   = false;

{Df}   firstD          = j(firstEvent);
{Dn1}  next(j(e))      = if(eq(next(e),e), i(firstDBB), j(next(e)));
{Dn2}  next(i(dbb))    = if(eq(next(dbb),dbb), i(firstDatum), i(next(dbb)));
{Dn3}  next(i(d))      = i(next(d));
{Dp}   penultimateD    = i(penultimateDatum);

{Deq1} eq(penultimateD, next(penultimateD)) = false;
{Deq2} eq(next(penultimateD), penultimateD) = false;
{Deq3} eq(next(x), next(y)) = false => eq(x, y) = false;
{Deq4} eq(x, x) = true )
END;

```

```
LET SimpleAtoms :=
```

```
    IMPORT Events INTO
```

```
CLASS
```

```
    SORT SimpleAtoms
```

```

    FUNC delta      :      -> SimpleAtoms    %deadlock
    FUNC i          :      -> SimpleAtoms    %internal action
    FUNC j          :      -> SimpleAtoms    %internal action

```

```

    FUNC n-bec-0    :      -> SimpleAtoms    %[n:=0]
    FUNC e-bec-0    :      -> SimpleAtoms    %[e:=0]
    FUNC n-bec-notn :      -> SimpleAtoms    %[n:=1-n]
    FUNC e-bec-note :      -> SimpleAtoms    %[e:=1-e]
    FUNC nn-eq-e    :      -> SimpleAtoms    %[nn=e]
    FUNC ee-eq-n    :      -> SimpleAtoms    %[ee=n]
    FUNC nn-neq-e   :      -> SimpleAtoms    %[nn<>e]
    FUNC ee-neq-n   :      -> SimpleAtoms    %[ee<>n]
    FUNC Fh         :      -> SimpleAtoms    %[Fh(b)]

```



```

FUNC Sf      :      -> SimpleAtoms      %[Sf(b,n,1-e)]
FUNC Gf      :      -> SimpleAtoms      %[Gf(bb,nn,ee)]
FUNC Th      :      -> SimpleAtoms      %[Th(bb)]
FUNC i       : Events -> SimpleAtoms      %embed Events

FUNC eq      : SimpleAtoms # SimpleAtoms -> BOOL
FUNC firstS  :      -> SimpleAtoms
FUNC next    : SimpleAtoms      -> SimpleAtoms
FUNC penultimateS :      -> SimpleAtoms

AXIOM
  FORALL x:SimpleAtoms, y:SimpleAtoms, e:Events (
    {Sif} firstS      = delta;
    {Sin1} next(delta) = i;
    {Sin2} next(i)     = j;
    {Sin3} next(j)     = n-bec-0;
    {Sin4} next(n-bec-0) = e-bec-0;
    {Sin5} next(e-bec-0) = n-bec-notn;
    {Sin6} next(n-bec-notn) = e-bec-note;
    {Sin7} next(e-bec-note) = nn-eq-e;
    {Sin8} next(nn-eq-e) = ee-eq-n;
    {Sin9} next(ee-eq-n) = nn-neq-e;
    {Sin10} next(nn-neq-e) = ee-neq-n;
    {Sin11} next(ee-neq-n) = Fh;
    {Sin12} next(Fh) = Sf;
    {Sin13} next(Sf) = Gf;
    {Sin14} next(Gf) = Th;
    {Sin15} next(Th) = i(firstEvent);
    {Sin16} next(i(e)) = i(next(e));
    {Sip} penultimateS = i(penultimateEvent);

    {Sieq1} eq(penultimateS, next(penultimateS)) = false;
    {Sieq2} eq(next(penultimateS), penultimateS) = false;
    {Sieq3} e eq(next(x), next(y)) = false => q(x, y) = false;
    {Sieq4} eq(x, x) = true )
  END;

LET Atoms := %Definition of the Atoms
  IMPORT SimpleAtoms INTO
  IMPORT InteractionType INTO
  IMPORT Ports INTO
  IMPORT D INTO
CLASS
  SORT Atoms
  FUNC do      : IntType # Ports # D -> Atoms
  FUNC i       : SimpleAtoms      -> Atoms %embed SimpleAtoms
  FUNC has-type : IntType # Atoms  -> BOOL
  FUNC port    : Atoms            -> Ports %what port is involved?
  FUNC datum   : Atoms            -> D    %and what datum?

  FUNC eq      : Atoms # Atoms -> BOOL
  FUNC firstAtom :      -> Atoms
  FUNC next     : Atoms      -> Atoms
  FUNC penultimateAtom :      -> Atoms

AXIOM
  FORALL x:Atoms, y:Atoms, e:SimpleAtoms,
    t IntType, t1:IntType, t2:IntType, p:Ports, d:D (
    {At1} has-type(t,i(e)) = false;
    {At2} has-type(t1,do(t2,p,d)) = eq(t1,t2);

    {At3} port(i(e)) = firstP; %default value

```

```

{At4}  port(do(t,p,d)) = p;
{At5}  datum(i(e)) = firstD;      %default value
{At6}  datum(do(t,p,d)) = d;

{Atf}  firstAtom = i(firstS);
{Atn1} next(i(e)) = if(eq(next(e),e),
                      do(firstType, firstP, firstD),
                      i(next(e)));
{Atn1} next(do(t,p,d)) = if(not(eq(next(t),t)),
                           do(next(t),p,d),
                           if(not(eq(next(p),p)),
                               do(firstType,next(p),d),
                               if(not(eq(next(d),d)),
                                   do(firstType,firstP,next(d)),
                                   do(t,p,d) )));
{Atn}  penultimateAtom = do(penultimateIntType, next(penultimateP),
                           next(penultimateD));

{Ateq1} eq(penultimateAtom, next(penultimateAtom)) = false;
{Ateq2} eq(next(penultimateAtom), penultimateAtom) = false;
{Ateq3} eq(next(x), next(y)) = false => eq(x, y) = false;
{Ateq4} eq(x, x) = true )
END;

```

6.3. Sets and Summations

The following sets, summations and sequences are needed:

```

LET SetsAtoms :=          %Sets of Atoms
RENAME
  SORT SET      -> SetsAtoms,
  FUNC null-set -> Atoms-null-set
IN
APPLY
  RENAME
    SORT ITEM      -> Atoms,
    FUNC eq        -> eq,
    FUNC firstItem -> firstAtom,
    FUNC next      -> next
  IN Sets
TO Atoms

LET SetsD :=          %Sets of elements from D
RENAME
  SORT SET      -> SetsD,
  FUNC null-set -> D-null-set
IN
APPLY
  RENAME
    SORT ITEM      -> D,
    FUNC eq        -> eq,
    FUNC firstItem -> firstD,
    FUNC next      -> next
  IN Sets
TO
  IMPORT D INTO
CLASS
  FUNC origin-Data      : -> SetsD
  FUNC origin-DBB       : -> SetsD
  FUNC origin-DBB-ce    : -> SetsD
  FUNC origin-DBB-ce-tio-fa : -> SetsD

```

```

AXIOM
  FORALL dbb:DBB, d:Data, e:Events (
{SD1}  is-in(origin-Data, i(dbb)) = false;
{SD2}  is-in(origin-Data, i(d))   = true;
{SD3}  is-in(origin-Data, j(e))   = false;
{SD4}  is-in(origin-DBB, i(dbb))  = true;
{SD5}  is-in(origin-DBB, i(d))    = false;
{SD6}  is-in(origin-DBB, j(e))    = false;
{SD7}  origin-DBB-ce              = insert(origin-DBB, j(ce));
{SD8}  origin-DBB-ce-tio-fa       = insert(insert(origin-DBB-ce,
                                                    j(tio)), j(fa)))
  )
END;

LET SetsPorts :=      %Sets of elements from Ports
RENAME
  SORT SET      -> SetsPorts,
  FUNC null-set -> Ports-null-set
IN
APPLY
  RENAME
    SORT ITEM      -> Ports,
    FUNC eq        -> eq,
    FUNC firstItem -> firstP,
    FUNC next      -> next
  IN Sets
TO
  IMPORT Ports INTO
CLASS
  FUNC pset1      : -> SetsPorts
  FUNC pset2      : -> SetsPorts
  FUNC internalports: -> SetsPorts
AXIOM (
{SP1}  pset1      = insert(insert(Ports-null-set, p3),p7);
{SP2}  pset2      = insert(insert(insert(insert(insert(Ports-null-set,
                                                    p2),p6),p9),p10),p11),p12);
{SP3}  internalports = union(pset1, pset2) )
END;

LET SumsOverDBB :=
RENAME
  SORT SET      -> SetsOfDBB,
  FUNC null-set -> DBB-null-set,
  FUNC FuncsToP -> FuncsDBBToP
IN
APPLY
  RENAME
    SORT ITEM      -> DBB,
    FUNC eq        -> eq,
    FUNC firstItem -> firstDBB,
    FUNC next      -> next
  IN Sum
TO
  IMPORT DBB INTO
CLASS
  FUNC DBBSet : -> SetsOfDBB
AXIOM
  FORALL d:DBB
{SDBB1} is-in(DBBSet,d) = true
END;

```

```

LET DBBSeq := %sequences over DBB
RENAME
  SORT SEQ -> DBBSeq
IN
APPLY
  RENAME
    FUNC ITEM -> DBB
  IN Sequences
TO DBB

```

6.4. OBSW

Now a brief discussion on the specification of the OBSW-protocol will be given. The easiest components are the two timers T_A and T_B . They just offer time-outs at ports p9 and p11.

$$T_A = s9(tio) \cdot T_A$$

$$T_B = s11(tio) \cdot T_B$$

The communication channels K and L are modeled as FIFO-queues with unbounded capacity. The process variables are indexed by the contents of the queue. Frames are received and subsequently communicated correctly, damaged or lost completely.

$$K = K^e = \sum_{f \in DBB} r2(f) \cdot K^f$$

$$K^{\sigma^*f} = (s7(f) + s7(ce) + i) \cdot K^\sigma + \sum_{g \in DBB} r2(g) \cdot K^{g^*\sigma^*f}$$

$$L = L^e = \sum_{f \in DBB} r6(f) \cdot L^f$$

$$L^{\sigma^*f} = (s3(f) + s3(ce) + j) \cdot L^\sigma + \sum_{g \in DBB} r6(g) \cdot L^{g^*\sigma^*f}$$

$$f \in DBB; \sigma \in (DBB)^*$$

Receiver R_A serves as an intermediate process, which accepts a frame from the communication channel L, and offers this frame to IMP_A via port p10, while signaling this offer to the IMP by sending a frame-arrival message at port p9. Incoming check-sum errors are also signaled at port p9, without using p10.

$$R_A = \sum_{f \in DBB} r3(f) \cdot R_A^f + r3(ce) \cdot R_A^{ce}$$

$$R_A^f = s9(fa) \cdot s10(f) \cdot R_A \quad (f \in DBB)$$

$$R_A^{ce} = s9(ce) \cdot R_A$$

Receiver R_B has the same structure:

$$R_B = \sum_{f \in DBB} r7(f) \cdot R_B^f + r7(ce) \cdot R_B^{ce}$$

$$R_B^f = s11(fa) \cdot s12(f) \cdot R_B \quad (f \in DBB)$$

$$R_B^{ce} = s11(ce) \cdot R_B$$

The definition of the IMP's follows from the computer program, from [21]. Every IMP takes care of two boolean variables: NextFrameToSend (n) and FrameExpected (e) and one datum-variable DatumToTransmit (b). These three items are packed in a frame, and transmitted over the communication channel. If the other IMP receives a frame, it is unpacked and the items are stored in the variables \underline{n} , \underline{e} and \underline{b} , and are examined. (In the specification \underline{n} , \underline{e} and \underline{b} are denoted by nn , ee and bb)

The program for IMP_A starts with PA . The variables n and e are initialized at 0, while b is initialised by the procedure $FromHost(Fh(b))$, which accepts a Data-element at port $p1$. The next state IMP_A enters is SF (SendFrame), in which the three items are packed and transmitted using the procedure $Sf(b, n, 1-e)$. Then we Wait for Something to happen (WS). Either a time-out (tio) or a checksum-error (ce) occurs, or a frame arrives (fa). The first two possibilities indicate some malfunction of one of the communication ports, resulting in a retransmission (SF). The frame arrival indicates that a frame must be accepted and examined (GF). The first test ($T1$) checks whether this frame has been accepted earlier or not. If not, then the datum-element is offered to the host ($Th(\underline{b})$) and the expected frame bit is flipped ($e := 1-e$). In the second test ($T2$) the need for a retransmission of the previous frame is examined. If our previous frame was received undamaged ($\underline{e}=n$), then a new data-element is fetched from the host, the n -bit is flipped and this new frame is sent. In the other case the old frame is retransmitted.

$$\begin{aligned} PA &= [n:=0] \cdot [e:=0] \cdot [Fh(b)] \cdot SF \\ SF &= [Sf(b, n, 1-e)] \cdot WS \\ WS &= tio \cdot SF + ce \cdot SF + fa \cdot GF \\ GF &= [Gf(\underline{b}, \underline{n}, \underline{e})] \cdot T1 \\ T1 &= [\underline{n}=e] \cdot Th(\underline{b}) \cdot [e:=1-e] \cdot T2 + [\underline{n} \neq e] \cdot T2 \\ T2 &= [\underline{e}=n] \cdot Fh(b) \cdot [n:=1-n] \cdot SF + [\underline{e} \neq n] \cdot SF \end{aligned}$$

The program for IMP_B looks the same as the one for IMP_A , except that IMP_B is unable to send a frame before an undamaged frame has arrived from A.

$$\begin{aligned} PB &= [n:=0] \cdot [e:=0] \cdot [Fh(b)] \cdot WF \\ WF &= ce \cdot WF + fa \cdot GF \\ SF &= [Sf(b, n, 1-e)] \cdot WS \\ WS &= tio \cdot SF + ce \cdot SF + fa \cdot GF \\ GF &= [Gf(\underline{b}, \underline{n}, \underline{e})] \cdot T1 \\ T1 &= [\underline{n}=e] \cdot Th(\underline{b}) \cdot [e:=1-e] \cdot T2 + [\underline{n} \neq e] \cdot T2 \\ T2 &= [\underline{e}=n] \cdot Fh(b) \cdot [n:=1-n] \cdot SF + [\underline{e} \neq n] \cdot SF \end{aligned}$$

Now we can transform these sequences of atoms into two meaningful processes IMP_A and IMP_B by applying the state-operator to PA and PB .

6.5. Action-Effect

Consider two objects A and B (see the module `Objects`). With each of them some state is associated, which consists of the values of the six variables $n, e, b, \underline{n}, \underline{e}, \underline{b}$ (see the module `States`). Now for each argument the functions action and effect have to be defined. The following definition is too informal, and has to be converted into a more explicit form (see the module `ACTION-EFFECT`). All relevant instances of the State Operator are given. The notation $\sigma[0/n]$ is used to denote the state that is derived from state σ by substituting 0 for variable n .

For all states σ , all objects m and all events e :

1. $\Lambda_{\sigma}^m([n:=0] \cdot x) = \tau \cdot \Lambda_{\sigma[0/n]}^m(x)$
 $\Lambda_{\sigma}^m([e:=0] \cdot x) = \tau \cdot \Lambda_{\sigma[0/e]}^m(x)$
 $\Lambda_{\sigma}^m([n:=1-n] \cdot x) = \tau \cdot \Lambda_{\sigma[1-\sigma(n)/n]}^m(x)$
 $\Lambda_{\sigma}^m([e:=1-e] \cdot x) = \tau \cdot \Lambda_{\sigma[1-\sigma(e)/e]}^m(x)$
2. $\Lambda_{\sigma}^A([Fh(b)] \cdot x) = \sum_{d \in D} r1(d) \cdot \Lambda_{\sigma[d/b]}^A(x)$
 $\Lambda_{\sigma}^B([Fh(b)] \cdot x) = \sum_{d \in D} r5(d) \cdot \Lambda_{\sigma[d/b]}^B(x)$
3. $\Lambda_{\sigma}^A([Th(\underline{b})] \cdot x) = s4(\sigma(\underline{b})) \cdot \Lambda_{\sigma}^A(x)$
 $\Lambda_{\sigma}^B([Th(\underline{b})] \cdot x) = s8(\sigma(\underline{b})) \cdot \Lambda_{\sigma}^B(x)$
4. $\Lambda_{\sigma}^A([Gf(\underline{b}, \underline{n}, \underline{e})] \cdot x) = \sum_{d \in D} \sum_{p, q \in B} r10(d, p, q) \cdot \Lambda_{\sigma[d/\underline{b}][p/\underline{n}][q/\underline{e}]}^A(x)$
 $\Lambda_{\sigma}^B([Gf(\underline{b}, \underline{n}, \underline{e})] \cdot x) = \sum_{d \in D} \sum_{p, q \in B} r12(d, p, q) \cdot \Lambda_{\sigma[d/\underline{b}][p/\underline{n}][q/\underline{e}]}^B(x)$
5. $\Lambda_{\sigma}^A(e \cdot x) = r9(e) \cdot \Lambda_{\sigma}^A(x)$
 $\Lambda_{\sigma}^B(e \cdot x) = r11(e) \cdot \Lambda_{\sigma}^B(x)$
6. $\Lambda_{\sigma}^m([n=e] \cdot x) = \begin{matrix} \tau \cdot \Lambda_{\sigma}^m(x) & \text{if } \sigma(\underline{n}) = \sigma(e) \\ \delta & \text{otherwise} \end{matrix}$
 $\Lambda_{\sigma}^m([n \neq e] \cdot x) = \begin{matrix} \tau \cdot \Lambda_{\sigma}^m(x) & \text{if } \sigma(\underline{n}) \neq \sigma(e) \\ \delta & \text{otherwise} \end{matrix}$
 $\Lambda_{\sigma}^m([\underline{e}=n] \cdot x) = \begin{matrix} \tau \cdot \Lambda_{\sigma}^m(x) & \text{if } \sigma(\underline{e}) = \sigma(n) \\ \delta & \text{otherwise} \end{matrix}$
 $\Lambda_{\sigma}^m([\underline{e} \neq n] \cdot x) = \begin{matrix} \tau \cdot \Lambda_{\sigma}^m(x) & \text{if } \sigma(\underline{e}) \neq \sigma(n) \end{matrix}$

δ otherwise

$$7. \quad \Lambda_{\sigma}^A([Sf(b,n,1-e)] \cdot x) = s2(\sigma(b), \sigma(n), 1-\sigma(e)) \cdot \Lambda_{\sigma}^A(x)$$

$$\Lambda_{\sigma}^B([Sf(b,n,1-e)] \cdot x) = s6(\sigma(b), \sigma(n), 1-\sigma(e)) \cdot \Lambda_{\sigma}^B(x)$$

```

LET Objects :=                %Set of objects to be used with the state-operator
CLASS
  SORT Objects
  FUNC A : -> Objects
  FUNC B : -> Objects
END;

LET States :=                %Definition of the set of states
  IMPORT D INTO
  CLASS
    SORT States
    FUNC st: BOOL # BOOL # D # BOOL # BOOL # D -> States
    FUNC zero-state : -> States

    FUNC subst1 : BOOL # States -> States
    FUNC subst2 : BOOL # States -> States
    FUNC subst3 : D # States -> States
    FUNC subst4 : BOOL # States -> States
    FUNC subst5 : BOOL # States -> States
    FUNC subst6 : D # States -> States

    FUNC proj1 : States -> BOOL
    FUNC proj2 : States -> BOOL
    FUNC proj3 : States -> D
    FUNC proj4 : States -> BOOL
    FUNC proj5 : States -> BOOL
    FUNC proj6 : States -> D
  AXIOM
    FORALL x1:BOOL, x2:BOOL, x4:BOOL, x5:BOOL, b:BOOL,
      x3:D, x6:D, d:D (
      {St1} zero-state = st(false,false,firstD,false,false,firstD);

      {St2} subst1(b, st(x1,x2,x3,x4,x5,x6)) = st(b,x2,x3,x4,x5,x6);
      {St3} subst2(b, st(x1,x2,x3,x4,x5,x6)) = st(x1,b,x3,x4,x5,x6);
      {St4} subst3(d, st(x1,x2,x3,x4,x5,x6)) = st(x1,x2,d,x4,x5,x6);
      {St5} subst4(b, st(x1,x2,x3,x4,x5,x6)) = st(x1,x2,x3,b,x5,x6);
      {St6} subst5(b, st(x1,x2,x3,x4,x5,x6)) = st(x1,x2,x3,x4,b,x6);
      {St7} subst6(d, st(x1,x2,x3,x4,x5,x6)) = st(x1,x2,x3,x4,x5,d);

      {St8} proj1(st(x1,x2,x3,x4,x5,x6)) = x1;
      {St9} proj2(st(x1,x2,x3,x4,x5,x6)) = x2;
      {St10} proj3(st(x1,x2,x3,x4,x5,x6)) = x3;
      {St11} proj4(st(x1,x2,x3,x4,x5,x6)) = x4;
      {St12} proj5(st(x1,x2,x3,x4,x5,x6)) = x5;
      {St13} proj6(st(x1,x2,x3,x4,x5,x6)) = x6 )
    END;

  LET ACTION-EFFECT :=      %definition of the action- and effectfunctions
    IMPORT Objects INTO
    IMPORT States INTO
    IMPORT SetsD INTO
    IMPORT SumsOverAtoms-Tau INTO

```

CLASS

FUNC ACTION : Atoms # Objects # States -> SetsOfAtoms-Tau

FUNC EFFECT : Atoms # Atoms-Tau # Objects # States -> States

AXIOM

FORALL a:Atoms, at:Atoms-Tau, sa:SimpleAtoms, m:Objects, st:States,
e:Events, t:IntType, p:Ports, da:Data, d:D, dbb:DBB,
b1:BOOL, b2:BOOL (

```
{Ae1a} ACTION(i(n-bec-0), m, st) = tau-set;
{Ae1b} ACTION(i(e-bec-0), m, st) = tau-set;
{Ae1c} ACTION(i(n-bec-notn), m, st) = tau-set;
{Ae1d} ACTION(i(e-bec-note), m, st) = tau-set;

{Ae2a} is-in(ACTION(i(Fh), A, st), j(a)) =
  has-type(r,a) & eq(port(a),p1) & is-in(origin-Data,datum(a));
{Ae2b} is-in(ACTION(i(Fh), B, st), j(a)) =
  has-type(r,a) & eq(port(a),p5) & is-in(origin-Data,datum(a));
{Ae2c} is-in(ACTION(i(Fh), m, st), pre-tau) = false;

{Ae3a} ACTION(i(Th), A, st) =
  insert(AtomsT-null-set, j(do(s,p4,proj6(st))));
{Ae3b} ACTION(i(Th), B, st) =
  insert(AtomsT-null-set, j(do(s,p8,proj6(st))));

{Ae4a} is-in(ACTION(i(Gf), A, st), j(a)) =
  has-type(r,a) & eq(port(a),p10) & is-in(origin-DBB,datum(a));
{Ae4b} is-in(ACTION(i(Gf), B, st), j(a)) =
  has-type(r,a) & eq(port(a),p12) & is-in(origin-DBB,datum(a));
{Ae4c} is-in(ACTION(i(Gf), m, st), pre-tau) = false;

{Ae5a} ACTION(i(i(e)), A, st) = insert(AtomsT-null-set, j(do(r,p9,j(e))));
{Ae5b} ACTION(i(i(e)), B, st) = insert(AtomsT-null-set, j(do(r,p11,j(e))));

{Ae6a} ACTION(i(nn-eq-e), m, st) = if(eq(proj4(st),proj2(st)),
  tau-set,
  AtomsT-null-set);
{Ae6b} ACTION(i(nn-neq-e), m, st) = if(not(eq(proj4(st),proj2(st))),
  tau-set,
  AtomsT-null-set);
{Ae6c} ACTION(i(ee-eq-n), m, st) = if(eq(proj5(st),proj1(st)),
  tau-set,
  AtomsT-null-set);
{Ae6d} ACTION(i(ee-neq-n), m, st) = if(not(eq(proj5(st),proj1(st))),
  tau-set,
  AtomsT-null-set);

{Ae7a} i(da) = proj3(st) =>
  ACTION(i(Sf), A, st) =
    insert(AtomsT-null-set,
      j(do(s,p2,i(frame(da,proj1(st),not(proj2(st)))))));
{Ae7b} i(da) = proj3(st) =>
  ACTION(i(Sf), B, st) =
    insert(AtomsT-null-set,
      j(do(s,p6,i(frame(da,proj1(st),not(proj2(st)))))));

{Ae8a} ACTION(do(t,p,d), m, st) = insert(AtomsT-null-set, j(do(t,p,d)));
{Ae8b} ACTION(i(i), m, st) = insert(AtomsT-null-set, j(i(i)));
{Ae8c} ACTION(i(j), m, st) = insert(AtomsT-null-set, j(i(j)));
{Ae8d} ACTION(i(delta), m, st) = insert(AtomsT-null-set, j(i(delta)));
```



```

{aE1a} EFFECT(i(n-bec-0), at, m, st) = subst1(false, st);
{aE1b} EFFECT(i(e-bec-0), at, m, st) = subst2(false, st);
{aE1c} EFFECT(i(n-bec-notn), at, m, st) = subst1(not(proj1(st)), st);
{aE1d} EFFECT(i(e-bec-note), at, m, st) = subst2(not(proj2(st)), st);

{aE2a} EFFECT(i(Fh), j(do(t,p,i(da))), m, st) = subst3(d,st);
{aE2b} EFFECT(i(Fh), j(do(t,p,i(dbb))), m, st) = st;
{aE2c} EFFECT(i(Fh), j(do(t,p,j(e))), m, st) = st;
{aE2d} EFFECT(i(Fh), j(i(sa)), m, st) = st;
{aE2e} EFFECT(i(Fh), pre-tau, m, st) = st;

{aE3} EFFECT(i(Th), at, m, st) = st;

{aE4a} EFFECT(i(Gf), j(do(t,p,i(frame(da,b1,b2)))), m, st)
      = subst6(d, subst4(b1, subst5(b2,st)));
{aE4b} EFFECT(i(Gf), j(do(t,p,i(da))), m, st) = st;
{aE4c} EFFECT(i(Gf), j(do(t,p,j(e))), m, st) = st;
{aE4d} EFFECT(i(Gf), j(i(sa)), m, st) = st;
{aE4e} EFFECT(i(Gf), pre-tau, m, st) = st;

{aE5} EFFECT(i(i(e)), at, m, st) = st;

{aE6a} EFFECT(i(nn-eq-e), at, m, st) = st;
{aE6b} EFFECT(i(nn-neq-e), at, m, st) = st;
{aE6c} EFFECT(i(ee-eq-n), at, m, st) = st;
{aE6d} EFFECT(i(ee-neq-n), at, m, st) = st;

{aE7} EFFECT(i(Sf), at, m, st) = st;

{aE8a} EFFECT(do(t,p,d), at, m, st) = st;
{aE8b} EFFECT(i(i), at, m, st) = st;
{aE8c} EFFECT(i(j), at, m, st) = st;
{aE8d} EFFECT(i(delta), at, m, st) = st )

END;

```

6.6. OBSW

Now, when σ_0 denotes an arbitrary initial state, we can define:

$$\text{IMP}_A = \Lambda_{\sigma_0}^A(\text{PA})$$

$$\text{IMP}_B = \Lambda_{\sigma_0}^B(\text{PB})$$

The communication function (see module Commerge) is defined as:

$$\text{st}(f) \mid \text{rt}(f) = \text{ct}(f) \\ \text{for } t \in \{2, 3, 6, 7, 9, 10, 11, 12\}, f \in \text{DBB} \cup \{\text{ce}, \text{tio}, \text{fa}\}$$

```

LET Commerge := %Definition of the communication-merge function
IMPORT Atoms INTO
IMPORT SetsPorts INTO
IMPORT SetsD INTO
CLASS
FUNC _|_ : Atoms # Atoms -> Atoms %communication merge
AXIOM
FORALL a:Atoms, b:Atoms (

```

```

{Com1} a|b = if( (has-type(s,a) & has-type(r,b)) |
                (has-type(r,a) & has-type(s,b)))
    & eq(port(a), port(b))
    & eq(datum(a), datum(b))
    & is-in(internalports, port(a))
    & is-in(origin-DBB-ce-tio-fa, datum(a)),

    do(c, port(a), datum(a)),
    i(delta)) )
END;

```

The message passing mechanism between L and R_A , respectively K and R_B is modeled as described in [5]. It is possible that before passing the previous frame to IMP_A the receiver R_A is offered a new frame ($s3(f)$) from the communication channel. This frame can get lost. To guarantee that whenever the receiver is able to accept an offer ($r3(f)$), it will not be lost, the following priority is defined (see module PO).

$$\begin{array}{ll} \delta < a & \text{for } a \in A \\ s3(f) < c3(f) & \text{for } f \in DBB \cup \{ce\} \\ s7(f) < c7(f) & \text{for } f \in DBB \cup \{ce\} \end{array}$$

```

LET PO :=                %partial order on Atoms
  IMPORT Atoms INTO
  IMPORT SetsD INTO
CLASS
  FUNC sm : Atoms # Atoms -> BOOL          %smaller
AXIOM
  FORALL a:Atoms, b:Atoms (
{PO1}  sm(a, b) = (eq(a, i(delta)) & not(eq(b, i(delta))))
        | ( has-type(s,a)
            & has-type(c,b)
            & eq(datum(a), datum(b))
            & is-in(origin-DBB-ce, datum(a))
            & ((eq(port(a), p3) & eq(port(b), p3)) |
                (eq(port(a), p7) & eq(port(b), p7)))
        ) )
END;

```

After defining (see module encapsset)

$$\begin{aligned} H1 &= \{r3(f) \mid f \in DBB \cup \{ce\}\} \\ H2 &= \{r7(f) \mid f \in DBB \cup \{ce\}\} \end{aligned}$$

the following two systems can be defined.

$$\begin{aligned} \theta \circ \partial_{H1} (L \parallel R_A) \\ \theta \circ \partial_{H2} (K \parallel R_B) . \end{aligned}$$

Now let the sets H_0 and I_0 be defined by (see modules encapsset and abstrset)

$$H_0 = \{st(f), rt(f) \mid t \in \{2, 6, 9, 10, 11, 12\}, f \in DBB \cup \{ce, tio, fa\}\}$$

$$I_0 = \{ct(f) \mid t \in \{2, 3, 6, 7, 9, 10, 11, 12\}, f \in DBB \cup \{ce, tio, fa\}\} \\ \cup \{st(f) \mid t \in \{3, 7\}, f \in DBB \cup \{ce\}\} \cup \{i, j\}$$

```

LET encapsset :=
  IMPORT SetsAtoms INTO
  IMPORT SetsPorts INTO
  IMPORT SetsD INTO
CLASS
  FUNC H0 : -> SetsAtoms
  FUNC H1 : -> SetsAtoms
  FUNC H2 : -> SetsAtoms
AXIOM
  FORALL a:Atoms (
    {e1} is-in(H0, a) = (has-type(r,a) | has-type(s,a)) &
                                is-in(pset2, port(a)) &
    is-in(origin-DBB-ce-tio-fa, datum(a));
    {e2} is-in(H1, a) = has-type(r,a) & eq(port(a), p3) &
                                is-in(origin-DBB-ce, datum(a));
    {e3} is-in(H2, a) = has-type(r,a) & eq(port(a), p7) &
                                is-in(origin-DBB-ce, datum(a)) )
  END;

LET abstrset :=
  IMPORT SetsAtoms INTO
  IMPORT SetsPorts INTO
  IMPORT SetsD INTO
CLASS
  SORT I0 : -> SetsAtoms
AXIOM
  FORALL a : -> Atoms (
    {abs1} is-in(I0, a) = eq(a, i(i))
                                | eq(a, i(j))
                                | (has-type(c,a) &
                                    is-in(internalports, port(a)) &
                                    is-in(origin-DBB-ce-tio-fa, datum(a)))
                                | (has-type(s,a) &
                                    is-in(pset1, port(a)) &
                                    is-in(origin-DBB-ce, datum(a))) )
  END;

```

Then finally the process OBSW is defined by

$$OBSW = \tau_{I_0} \circ \partial_{H_0} (IMP_A \parallel T_A \parallel \theta \circ \partial_{H_2} (K \parallel R_B) \parallel IMP_B \parallel T_B \parallel \theta \circ \partial_{H_1} (L \parallel R_A))$$

```

LET OBSWPart1 :=
EXPORT
  FUNC pre-OBSW : -> process
FROM
  IMPORT SumsOverDBB INTO
  IMPORT ACP-Theta INTO
  IMPORT lambda INTO
  IMPORT DBBSeq INTO
CLASS
  FUNC PA : -> process
  FUNC SF : -> process
  FUNC WS : -> process

```

```

FUNC GF      :      -> process
FUNC T1      :      -> process
FUNC T2      :      -> process
FUNC PB      :      -> process
FUNC WF      :      -> process
FUNC Ta      :      -> process
FUNC Tb      :      -> process
FUNC Ra      :      -> process
FUNC Rb      :      -> process
FUNC Ra      : DBB   -> process
FUNC Rb      : DBB   -> process
FUNC Ra-ce   :      -> process
FUNC Rb-ce   :      -> process
FUNC K       :      -> process
FUNC L       :      -> process
FUNC K       : DBBSeq -> process
FUNC L       : DBBSeq -> process
FUNC IMPa    :      -> process
FUNC IMPb    :      -> process
FUNC pre-OBSW :      -> process

FUNC FunRa   :      -> FuncsDBBToP
FUNC FunRb   :      -> FuncsDBBToP
FUNC FunK-eps :      -> FuncsDBBToP
FUNC FunL-eps :      -> FuncsDBBToP
FUNC FunK-s   : DBBSeq -> FuncsDBBToP
FUNC FunL-s   : DBBSeq -> FuncsDBBToP
AXIOM
  FORALL f:DBB, q:DBBSeq (
{OBS1} app(FunRa, f)      = i(do(r, p3, i(f))).Ra(f);
{OBS2} app(FunRb, f)      = i(do(r, p7, i(f))).Rb(f);
{OBS3} app(FunK-eps, f)   = i(do(r, p2, i(f))).K(seq(f));
{OBS5} app(FunL-eps, f)   = i(do(r, p6, i(f))).L(seq(f));
{OBS6} app(FunK-s(q), f)  = i(do(r, p2, i(f))).K(seq(f)+q);
{OBS7} app(FunL-s(q), f)  = i(do(r, p6, i(f))).L(seq(f)+q);

{OBS8} PA                = i(i(n-bec-0)).i(i(e-bec-0)).i(i(Fh)).SF;
{OBS9} SF                = i(i(Sf)).WS;
{OBS11} WS               = (i(i(i(tio))).SF) + (i(i(i(ce))).SF) + (i(i(i(fa))).GF);
{OBS12} GF               = i(i(Gf)).T1;
{OBS13} T1               = (i(i(nn-eq-e)).i(i(Th)).i(i(e-bec-note)).T2) +
                          (i(i(nn-neq-e)).T2);
{OBS14} T2               = (i(i(ee-eq-n)).i(i(Fh)).i(i(n-bec-notn)).SF) +
                          (i(i(ee-neq-n)).SF);
{OBS15} PB               = i(i(n-bec-0)).i(i(e-bec-0)).i(i(Fh)).WF;
{OBS16} WF               = (i(i(i(ce))).WF) + (i(i(i(fa))).GF);

{OBS17} Ta               = i(do(s, p9, j(tio))) . Ta;
{OBS18} Tb               = i(do(s, p11, j(tio))) . Tb;

{OBS19} Ra               = Sum(DBBSet, FunRa) + (i(do(r, p3, j(ce))).Ra-ce);
{OBS20} Ra(f)            = i(do(s, p9, j(fa))) . i(do(s, p10, i(f))) . Ra;
{OBS21} Ra-ce            = i(do(s, p9, j(ce))) . Ra;

{OBS22} Rb               = Sum(DBBSet, FunRb) + (i(do(r, p7, j(ce))).Rb-ce);
{OBS23} Rb(f)            = i(do(s, p11, j(fa))) . i(do(s, p12, i(f))) . Rb;
{OBS24} Rb-ce            = i(do(s, p11, j(ce))) . Rb;

{OBS25} K                 = K(eps);
{OBS26} K(eps)            = Sum(DBBSet, FunK-eps);

```



```

{OBS27} K(q+seq(f)) = (i(do(s,p7,i(f))) + i(do(s,p7,j(ce))) + i(i(i))) . K(q);
               + Sum(DBBSet, FunK-s(q+seq(f)));

{OBS28} L          = L(eps);
{OBS29} L(eps)     = Sum(DBBSet, FunL-eps);
{OBS30} L(q+seq(f)) = (i(do(s,p3,i(f))) + i(do(s,p3,j(ce))) + i(i(j))) . L(q)
               + Sum(DBBSet, FunL-s(q+seq(f)));

{OBS31} IMPa       = lambda(A, zero-state, PA);
{OBS32} IMPb       = lambda(B, zero-state, PB);

{OBS33} pre-OBSW   = d(H0, IMPa || Ta || theta(d(H2, K||Rb)) ||
               IMPb || Tb || theta(d(H1, L||Ra)) ) )

END;

LET OBSWPart2 :=
  IMPORT ACP-Tau INTO
  IMPORT SC INTO
  IMPORT SC OBSWPart1 INTO
CLASS
  FUNC OBSW : -> process
AXIOM
{OBS34} OBSW = abstr(I0, pre-OBSW)
END;

```

7. Final Remarks

7.1. Execution and implementation

One reason for making this algebraic specification is given by the wish to 'execute' or 'implement' systems that are described in terms of process algebra. Two different approaches are possible. The first one is the automatic generation of proofs and the second one is prototyping the specified system.

The former approach originates from the fact that, in many cases, large parts of proofs about specifications in process algebra (e.g. two specification being equal for an external observer) turn out to be straightforward, but long and tedious. The reason for this can be found in the character of the axioms constituting process algebra. Most of them can be viewed, and are often used, as rewrite rules. However, when a proof involves some human ingenuity, which exceeds just using some rewrite rules, an automatic prover may not succeed. This shortcoming is unavoidable, due to the fact that the theory is undecidable. So in the verification of specifications some interaction between man and machine is needed. The computer can be used as an aid in proving. The user can just point out what (sub)term should be expanded or what new process variable should be introduced. Then, after some computations, an equivalent system is displayed.

The latter approach to executing specifications consists of the simulation of the (external) behaviour of the specified system. In interaction with the computer a user is able to check if the specified behaviour conforms to his expectations or intentions. For this purpose the computer should report all (relevant) actions to the user, making appropriate internal choices (e.g. interleaving actions in a merge), and offer external choices (e.g. summation over data-values) to the user. The computer produces a possible trace. Of course the real-time behaviour of the specified system will not be simulated, but it would be a low-cost prototyping technique.

Both approaches explicitly depend on term rewrite systems (TRS's). This requires that all equations can be interpreted as rewrite rules. Equations that are used in only one

direction are simple to deal with, but some equations, like the commutativity and associativity of alternative composition, are more difficult. The TRS resulting by adding these rules as a rewrite rule from left to right and from right to left would not be strongly normalizing.

Some equations that could cause problems when being interpreted as rewrite rules are mentioned in the sequel.

In the modules that define the atoms there are no problems, but since the eq -function only tests for syntactical equality, this function could be implemented easier.

In the module Sets only the last equation $S8 (s1=s2 \text{ when } eq(s1, s2)=true)$ is not a rewrite rule. The solution is to delete this equation and substitute in the following modules all occurrences of $s1=s2$ by $eq(s1, s2)=true$.

In the module BPA, there are three equations that must be implemented directly, not by using rewrite rules. Rule A1 ($x+y=y+x$) is dealt with by using a "commutative" rewriting system. For rule A2 ($(x+y)+z=x+(y+z)$) one needs an "associative" system and for rule A3 ($x+x=x$) an "idempotent" system is needed. Together these three demands state that a process can be represented by the set of its summands. The elements of a set are not ordered (A1 and A2), and a set contains no duplicate elements (A3). Now addition of processes is represented by the union of their defining sets, while removing duplicate summands.

The equations for the communication function C1 ($alb=bla$), C2 ($((alb)lc=al(blc))$) and C3 ($(\delta la=\delta)$) simply define some restrictions on its definition. So if the definition is right, these rules can be deleted.

In [1] rewriting systems are given for ACP_τ and ACP_θ . In ACP_τ the rules T2 ($\tau x+x=\tau x$) and T3 ($a(\tau x+y)=a(\tau x+y)+ax$) are deleted because they are used in two directions. Moreover a rule RT2 ($x(\tau y) \rightarrow xy$) is added. In ACP_θ two rules are

added: RP8 ($(x \triangleleft y) \triangleleft y \rightarrow x \triangleleft y$) and RTH4 ($\theta(x) \triangleleft x \rightarrow \theta(x)$)

The equations for Standard Concurrency SC1-SC6 cause some problems that are not simple to deal with.

The most natural way to implement the resulting TRS's seems to be a functional or a logic programming language. Both PROLOG and LISP are candidates. Some programs are already being developed: a PROLOG program that calculates the normal form of finite processes and a PASCAL program that determines the equality of regular processes. There is also an implementation of the sets module in PROLOG.

7.2. Relation to LOTOS

There is some resemblance between the specification language LOTOS [12] and the combination of (the subset of) COLD with the specification of process algebra. LOTOS consists of two parts: the data-definition language (ACT-ONE) (see [13]) and the process specification language (based on Milner's CCS, see [19]). It appears that the COLD-subset and ACT-ONE are much the same. Their syntax is interchangeable and they both use initial algebra semantics.

Some differences between the process specification part and process algebra arise from the fact that in LOTOS it is part of the specification language, while in this approach it is defined by means of the specification language.

1. LOTOS has a fixed number of primitives, whereas the specification of process algebra could be extended with new primitives any time.
2. LOTOS supports conditional expressions and variables in a natural way, while in process algebra this is implemented with the State Operator.
3. LOTOS can deal with (simple) abstraction and encapsulation sets. In this specification a complex sort SETS was needed.
4. The semantics of LOTOS processes is defined separately from the semantics

of ACT-ONE, while in the specification of process algebra the initial algebra semantics is used.

5. In process algebra a wide range of proof techniques is being developed, while LOTOS is merely used for specification purposes.
6. The information about what process is allowed to use what communication channel is in LOTOS part of the specification. In process algebra this has to be stated in an informal way.

7.3. About the Specification Language

After adding some features from ASF to COLD, the language becomes quite suitable. There are still more features needed to improve the specification of process algebra. A point of comment about the operators is the lack of a mechanism to define the precedence of operators. In the present formalism every equation with some ambiguity has to be expanded with an overkill of parentheses. A method to express the priority of operators would add to the readability of specifications.

The injection functions also produce a lot of brackets. A way to avoid this is the introduction of a new notion: *subsorts*. If some sort A is stated to be a subsort of sort B, then B inherits all elements of sort A. All functions and operators on sort A then become partial functions and operators on sort B. If this construction is considered as a purely syntactical abbreviation, no semantical problems occur. A definition of a subsort could semantically be expanded to a definition of an injection function from sort A to sort B. With these enrichments of the specification language e.g. the equation for timer one would not look like :

$$T1 = (i(i(nn-eq-e)) . i(i(Th)) . i(i(e-bec-note)) . T2) + (i(i(nn-neq-e)) . T2)$$

but like:

$$T1 = nn-eq-e.Th.e-bec-note.T2 + nn-neq-e.T2$$

7.4. Conclusion

Now we can make some concluding remarks. Transforming process algebra into an algebraic specification is quite easy. The definitions of most operators and functions are in an algebraic form. On account of the modularization it is easy to pick out some modules and join them to form the desired axiom-system. Also adding new operators or processes is easy.

To transform an application of process algebra (e.g. a specification of a communication protocol) into an algebraic specification takes more effort. This is due to the fact that the atoms are often specified quite informally. Giving an algebraic specification forces the specification to be more exact. To facilitate a proper definition of the atoms and other parameters a strong concept of set has to be introduced.

If one wishes to transform a verification into an algebraic specification, problems arise when the proof involves some advanced process algebra techniques. Research in process algebra has to be done to make an algebraic specification possible. This also influences the way to implement or execute a given specification. The most natural way to implement this would be in a term rewriting system, which itself could be implemented in e.g. PROLOG.

Finally the specification formalism could use some more extensions

References

- [1] J.C.M. Baeten, *Procesalgebra*, [in Dutch], Kluwer, 1986.
- [2] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *Conditional axioms and α/β calculus in process algebra*, Proc. IFIP Conf. on Formal Description of Programming Concepts - III, Ebberup 1986, (M. Wirsing, ed.), North-Holland Amsterdam, pp.53-75, 1987.
- [3] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *On the consistency of Koomen's Fair Abstraction Rule*, Report CS-R8504, Centre for Math. and Comp. Sci., Amsterdam 1985, to appear in TCS 51 (1/2), 1987.
- [4] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *Syntax and defining equations for an interrupt mechanism in process algebra*, Fund. Inf. IX (2), pp. 127-168, 1986.
- [5] J.A. Bergstra, *Put and get, primitives for synchronous unreliable message passing*, CIF report LGPS 3, State University of Utrecht, 1985.
- [6] J.A. Bergstra, J. Heering & P. Klint, *Algebraic definition of a simple programming language*, Report CS-R8504, Centre for Math. and Comp. Sci., Amsterdam 1985.
- [7] J.A. Bergstra, J. Heering & P. Klint, *ASF - An algebraic specification formalism*, report CS-R8705, Centre for Math. and Comp. Sci., Amsterdam 1987.
- [8] J.A. Bergstra & J.W. Klop, *Algebra of communicating processes*, Proc. CWI Symp. Math. & Comp. Sci. (J.W. de Bakker, M. Hazewinkel & J.K. Lenstra, eds.), pp. 89-138, North-Holland, 1986.
- [9] J.A. Bergstra & J.W.Klop, *Algebra of communicating processes with abstraction*, TCS 37 (1), pp. 77-121, 1985.
- [10] J.A. Bergstra & J.W.Klop, *Conditional Rewrite Rules: Confluence and Termination*, JCSS 32, pp. 323-362, 1986.
- [11] J.A. Bergstra & J.W.Klop, *Process algebra for synchronous communication*, Inf. & Control 60 (1/3), pp. 109-137, 1984.
- [12] E. Brinksma (ed.), *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO DIS 8807, 1987.
- [13] H. Ehrig, W. Fey & H. Hansen, *ACT ONE, an algebraic specification language with two levels of semantics*, TU Berlin, FB 20, Techn. Report 83-03, 1983.
- [14] H. Ehrig & B. Mahr, *Fundamentals of Algebraic Specification 1*, Springer verlag, 1985.
- [15] L.M.G. Feijs, H.B.M. Jonkers, C.P.J. Koymans & G.R. Renardel de Lavalette, *Formal definition of the design language COLD-K*, Technical Report, ESPRIT project 432, Doc.Nr. METEOR/t7/PRLE/7, 1987.
- [16] J.V. Guttag & J.J. Horning, *The algebraic specification of abstract datatypes*, Acta Informatica 10, pp. 27-52, 1978.
- [17] C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall International, 1985.
- [18] H.B.M. Jonkers, *A concrete syntax for COLD-K*, Philips Research Laboratories Eindhoven, 1988.
- [19] R. Milner, *A calculus of communicating systems*, Springer LNCS 92, 1980.
- [20] W. Reisig, *Petrinetze*, Springer-Verlag, 1982.
- [21] A.S. Tanenbaum, *Computer Networks*, Prentice Hall, 1981.
- [22] F.W. Vaandrager, *Verification of two communication protocols by means of process algebra*, Report CS-R8608, Centre for Math. and Comp. Sci., Amsterdam 1986.
- [23] H.R. Walters, *An annotated algebraic specification of the static semantics of POOL*, FVI report 86-20, University of Amsterdam, 1986.