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 then 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;
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 \texttt{eps} denotes the empty sequence, \texttt{seq} transforms an item into a one-item sequence and \texttt{+} concatenates two sequences.

\begin{verbatim}
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;
\end{verbatim}

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 \texttt{Ø} (the empty set) and \texttt{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.

\begin{verbatim}
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
      END;
\end{verbatim}
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;  
      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) = inters(insert(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(s1, 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:

    insert(insert(... insert(Ø, x₁), ... ,xₙ₋₁), xₙ) [n≥0]

Moreover, this specification forces every instance of a set to be specified using the Ø 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∈Data | x=d₀ ∨ x=d₁}

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:

\[
\text{is-in}(T, x) = \text{eq}(x, d_0) \lor \text{eq}(x, d_1)
\]

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

Now the initial algebra of \( \text{Sets} \) 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 \( \text{is-in} \) 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 \( N \), the nonnegative integers. Suppose a constant \( \text{one} \) is defined with equation

\[
\text{s} (\text{one}) = s(s(0))
\]

Now the intended meaning of \( \text{one} \) 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:

\[
p(0) = 0
\]
\[
p(s(x)) = x.
\]

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:

\[
is\text{-}\text{in}(T, i) = eq(i, d_0) \lor eq(i, d_l)
\]

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:

\[
eq(X, Y) = \text{eq2}(X, Y, \text{firstItem})
eq2(X, Y, i) = \begin{cases} 
\text{last}(i), \\
(i \in X) \Leftrightarrow (i \in Y), \\
\text{if}( (i \in X) \Leftrightarrow (i \in Y), \\
\text{eq2}(X, Y, \text{next}(i)), \\
\text{false}) 
\end{cases}
\]

\[
eq(X, Y) = \text{true} \Rightarrow X = Y
\]

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 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 than 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

\[
U = \text{mat}(\text{row}(1, 0, 0, 0), \\
\quad \text{row}(0, 1, 0, 0), \\
\quad \text{row}(0, 0, 1, 0), \\
\quad \text{row}(0, 0, 0, 1))
\]

or, using an implicit definition:

\[
elt(U, i, j) = \text{if}(\text{eq}(i, j), 1, 0).
\]

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
\text{Items} must contain a first\text{Item} and an enumerating function, called next. The
last element of \text{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.

\begin{verbatim}
LET Items :=
  IMPORT Booleans INTO
CLASS
  SORT ITEM FREE
  FUNC eq : ITEM # ITEM -> BOOL
  FUNC firstItem : -> ITEM
  FUNC next : ITEM -> ITEM
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 eq : SET # SET -> BOOL
  FUNC eq2 : SET # SET # ITEM -> BOOL
AXIOM
  FORALL s:SET, s1:SET, s2:SET, i:ITEM, j:ITEM (\begin{align*}
  (S1) & \text{is-in}(\text{null-set}, j) = \text{false}; \\
  (S2) & \text{is-in}(\text{delete}(s, i), j) = \text{is-in}(s, j) \land \text{not(eq}(i, j)); \\
  (S3) & \text{is-in}(\text{insert}(s, i), j) = \text{is-in}(s, j) \land \text{eq}(i, j); \\
  (S4) & \text{is-in}(\text{union}(s1, s2), i) = \text{is-in}(s1, i) \lor \text{is-in}(s2, i); \\
  (S5) & \text{is-in}(\text{inters}(s1, s2), i) = \text{is-in}(s1, i) \land \text{is-in}(s2, i); \\
  (S6) & \text{eq}(s1, s2) = \text{eq2}(s1, s2, \text{firstItem}); \\
  (S7) & \text{eq2}(s1, s2, i) = \text{if}(\text{eq}(\text{next}(i), i), \\
  & \text{eq}(\text{is-in}(s1, i), \text{is-in}(s2, i)), \\
  & \text{if}(\text{eq}(\text{is-in}(s1, i), \text{is-in}(s2, i)),)
\end{align*}
\end{verbatim}
eq2(s1, s2, next(i)),
false));

\{S8\} \text{eq}(s1, s2) = \text{true} \implies 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:

\[(i \in A) = \text{false}.\]

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:

\[(i \in B) = \text{if}(i = d0, \text{true}, \text{false}).\]

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 (\( \text{i}(\text{i}(\text{delta})) \)), which is the neutral element for summation. The parameter \( \text{Items} \) is inherited from the imported sort \( \text{Sets} \), and provides the sort \( \text{ITEM} \) 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 \( \text{FuncsToP} \) consisting of functions from elements of the index-set to processes has to be defined. The most natural definition would look like:

\[
\text{SORT} \quad \text{FuncsToP} : \text{ITEM} \rightarrow \text{process}.
\]

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

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

\[
\text{LET Sum :=} \quad \%\text{Summation of indexed processes}
\]
\[
\text{LAMBDA Items_Parameter2: Items OF}
\]
\[
\text{IMPORT APPLY Sets TO Items_Parameter2 INTO}
\]
\[
\text{IMPORT Base INTO}
\]
\[
\text{EXPORT}
\]
\[
\text{SORT FuncsToP,} \quad \%\text{Functions from ITEMS to processes}
\]
\[
\text{FUNC Sum : SET \# FuncsToP \rightarrow process,} \quad \%\text{summation}
\]
\[
\text{FUNC app : FuncsToP \# ITEM \rightarrow process} \quad \%\text{application}
\]
\[
\text{FROM} \]
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 \( (\_\_\_) \).

\[
\text{LET Base :=} \quad \text{\%Definition of the basic operations} \\
\text{IMPORT Atoms INTO} \\
\text{CLASS} \\
\text{SORT \( \text{process} \)} \\
\text{FUNC \( i \) : Atoms -> process \%embed Atoms in process} \\
\text{FUNC \( \_+\_ \) : process \# process -> process} \\
\text{FUNC \( \_\_\_ \) : process \# process -> process} \\
\text{END;}
\]

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

\[
\begin{align*}
A_1 & \quad x+y = y+x \\
A_2 & \quad x+(y+z) = (x+y)+z \\
A_3 & \quad x+x = x \\
A_4 & \quad (x+y)z = xz+yz \\
A_5 & \quad (xy)z = x(yz)
\end{align*}
\]

\[
\text{LET BPA :=} \quad \text{\%Axioms for the system BPA} \\
\text{IMPORT Atoms INTO} \\
\text{IMPORT Base INTO} \\
\text{CLASS} \\
\text{AXIOM} \\
\text{FORALL x:process, y:process, z:process (} \\
\{A1\} & \quad x+y = y+x; \\
\{A2\} & \quad (x+y)+z = x+(y+z); \\
\{A3\} & \quad x+x = x;
\}
\{A4\} \quad (x+y).z = (x.z) + (y.z);
\{A5\} \quad (x.y).z = x.(y.z) \)
END;

4.3. Delta
A new Atom \texttt{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$_\delta$ can be obtained by importing both the modules BPA and Delta.

\begin{align*}
A6 \quad x + \delta &= x \\
A7 \quad \delta x &= \delta
\end{align*}

In the section about the definition of the atoms the function \texttt{delta} will be declared. The atom \(\delta\) will be denoted by \texttt{i(delta)}, so the process \(\delta\) will be denoted by \texttt{i(i(delta))}.

\begin{verbatim}
LET Delta := %Axioms for deadlock
  IMPORT Base INTO
  CLASS
  AXION
    FORALL x:process (
      \{A6\} \quad x+i(i(delta)) = x;
      \{A7\} \quad i(i(delta)).x = i(i(delta))
    )
END;
\end{verbatim}

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 \texttt{delta}, so the process is forced to make an alternative choice if possible. The imported module \texttt{encapsset} can contain various sets of atoms. An example will be given in the sequel.

\begin{align*}
D1 \quad \partial_H(a) &= a \quad \text{if } a \notin H \\
D2 \quad \partial_H(a) &= \delta \quad \text{if } a \notin H \\
D3 \quad \partial_H(x+y) &= \partial_H(x) + \partial_H(y) \\
D4 \quad \partial_H(xy) &= \partial_H(x) \cdot \partial_H(y)
\end{align*}

\begin{verbatim}
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
  AXION
    FORALL a:Atoms, x:process, y:process, H:SetsAtoms (\{D1-2\} \quad d(H,i(a)) = if(is-in(H,a), i(i(delta)), i(a));
\end{verbatim}
\{D3\} \quad d(H, x+y) = d(H, x) + d(H, y);
\{D4\} \quad d(H, x.y) = d(H, x) \cdot 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 $\setminus$) 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 $\text{Commmerge}$, where a communication-function between atoms must be defined. In the first equation ($\text{CM0}$) of the module $\text{Comm}$, this function is extended to processes.

\[
\text{CM1} \quad x\parallel y = x\ll y + y\ll x + x\parallel y
\]
\[
\text{CM2} \quad a\ll x = ax
\]
\[
\text{CM3} \quad ax\ll y = a(x\parallel y)
\]
\[
\text{CM4} \quad (x+y)\ll z = x\ll z + y\ll z
\]
\[
\text{CM5} \quad (ax)\parallel b = (a\parallel b)x
\]
\[
\text{CM6} \quad a\parallel (bx) = (a\parallel b)x
\]
\[
\text{CM7} \quad (ax)\parallel (by) = (a\parallel b)(x\parallel y)
\]
\[
\text{CM8} \quad (x+y)\parallel z = x\parallel z + y\parallel z
\]
\[
\text{CM9} \quad x\parallel (y\parallel z) = x\parallel y + x\parallel z
\]
\[
\text{C1} \quad a\parallel b = b\parallel a
\]
\[
\text{C2} \quad (a\parallel b)c = a\parallel (b\parallel c)
\]
\[
\text{C3} \quad \delta\parallel a = \delta
\]

LET Comm := %Communication axioms (used in ACP)
IMPORT Atoms INTO
IMPORT Base INTO
IMPORT Delta INTO
IMPORT Commmerge INTO

CLASS
FUNC $\ll$ : process $\parallel$ process $\rightarrow$ process $\parallel$merge
FUNC $\setminus$ : process $\parallel$ process $\rightarrow$ process $\parallel$leftmerge
FUNC $\parallel$ : process $\parallel$ process $\rightarrow$ process $\parallel$Communication merge on processes

AXIOM
FORALL a:Atoms, b:Atoms, d:Atoms, x:process, y:process, z:process ( 
\{CM0\} \quad i(a)\parallel i(b) = i(a\parallel b); %identify overloaded $\parallel$
\{CM1\} \quad x\parallel y = (x\ll y) + (y\ll x) + (x\parallel y);
\{CM2\} \quad i(a)\ll x = i(a).x;
\{CM3\} \quad (i(a).x)\ll y = i(a).x\parallel y);
\{CM4\} \quad (x+y)\ll z = (x\ll z) + (y\ll z);
\{CM5\} \quad (i(a).x)\parallel i(b) = (i(a)\parallel i(b)).x;
\{CM6\} \quad i(a)\parallel (i(b).x) = (i(a)\parallel i(b)).x;
\{CM7\} \quad (i(a).x)\parallel (i(b).y) = (i(a)\parallel i(b))\ll (x\parallel y);

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

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

\text{LET ACP :=} \quad \% \text{Axioms for the system ACP}
\text{IMPORT BPA INTO} \\
\text{IMPORT Delta INTO} \\
\text{IMPORT Encaps INTO} \\
\text{Comm}
\text{END;}

\text{4.6. ACP}_t

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.

\[
\begin{align*}
T_1 & \quad x\tau = x \\
T_2 & \quad \tau x + x = \tau x \\
T_3 & \quad a(\tau x+y) = a(\tau x+y)+ax
\end{align*}
\]

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

A module \text{abstrset} must be provided to indicate what sets of atoms can be abstracted from.

\[
\begin{align*}
\text{TI1} & \quad \tau_I(\tau) = \tau \\
\text{TI2} & \quad \tau_I(a) = a \quad \text{if } a \notin I \\
\text{TI3} & \quad \tau_I(a) = \tau \quad \text{if } a \in I \\
\text{TI4} & \quad \tau_I(x+y) = \tau_I(x) + \tau_I(y) \\
\text{TI5} & \quad \tau_I(xy) = \tau_I(x) \cdot \tau_I(y)
\end{align*}
\]

\text{LET Abstr :=} \quad \% \text{Axioms for the abstraction-operator}
IMPORT SetsAtoms INTO
IMPORT Tau INTO
IMPORT abstrset INTO
CLASS
   FUNC abstr : SetsAtoms # process -> process
AXIOM
   FOR ALL 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 $ACP_\tau$ (see e.g. [9]) combines ACP with abstraction.

\begin{align*}
\text{TM1} & \quad \tau \parallel x = \tau x \\
\text{TM2} & \quad (\tau x) \parallel y = \tau (x \parallel y) \\
\text{TC1} & \quad \tau | x = \delta \\
\text{TC2} & \quad x | \tau = \delta \\
\text{TC3} & \quad (\tau x) | y = x | y \\
\text{TC4} & \quad x | (\tau y) = x | y \\
\text{DT} & \quad \partial_H(\tau) = \tau
\end{align*}

\text{LET } ACP-Tau := \%Axioms for the system ACP-Tau

IMPORT ACP INTO
IMPORT Tau INTO
IMPORT Abstr INTO
CLASS
AXIOM
   FOR ALL x:process, y:process, h:SetsAtoms (  
      [TM1] tau \parallel x = tau . x; \\
      [TM2] (tau . x) \parallel y = tau . (x \parallel 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 $ACP_\tau$ often some extra laws are assumed. These laws are called Standard Concurrency (see [9]).

\begin{align*}
\text{SC1} & \quad (x \parallel y) \parallel z = x \parallel (y \parallel z) \\
\text{SC2} & \quad (x | ay) \parallel z = x | (ay \parallel z) \\
\text{SC3} & \quad x | y = y | x \\
\text{SC4} & \quad x \parallel y = y \parallel x \\
\text{SC5} & \quad x | (y | z) = (x | y) | z
\end{align*}
SC6  \[ x \parallel (y \parallel z) = (x \parallel y) \parallel z \]

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 \parallel y) \parallel z = x \parallel (y \parallel z);
    {SC2}  (x | (i(a) . y)) \parallel z = x | ((i(a) . y) \parallel z);
    {SC3}  x \parallel y = y \parallel x;
    {SC4}  x \parallel y = y \parallel x;
    {SC5}  x \parallel (y \parallel z) = (x \parallel y) \parallel z;
    {SC6}  x \parallel (y \parallel z) = (x \parallel y) \parallel z
  )
END;

4.8. ACP0

Instead of adding the silent step to ACP, forming ACP\(\tau\), it is also possible to add
priorities, forming the system ACP0 (see e.g. [4]). Relative to a given partial order on
the atoms, the operator \(\text{ht}e\)t\(\text{a}\) determines which actions should be enabled or disabled.
The smallest atoms in this partial order \(\prec\) 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.  \(- (a \prec a)\)
2.  \(a \prec b \Rightarrow \neg (b \prec a)\)
3.  \(a \prec b \land b \prec c \Rightarrow a \prec c\)
4.  \(\delta \prec a \quad \text{(if} \ a \neq \delta\text{)}\)

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 \(\prec\), in the specification denoted by \(\prec\), the
priority operator can be defined. This \(\prec\) operator deadlocks if its lefthand-side has
lower priority then its righthand-side, else yields its lefthand-side. Now the priority
operator can easily be defined.

P1  \(a \prec b = a\) if \(- (a \prec b)\)
P2  \(a \prec b = \delta\) if \(a \prec b\)
P3  \(x \prec yz = x \prec y\)
P4  \(x \prec (y + z) = (x \prec y) \prec z\)
P5  \(xy \prec z = (x \prec z) y\)
P6  \((x + y) \prec z = x \prec z + y \prec z\)
TH1 \(\theta(a) = a\)
TH2 \(\theta(xy) = \theta(x) \cdot \theta(y)\)
TH3 \(\theta(x + y) = \theta(x) \prec y + \theta(y) \prec x\)
LET Theta :=
IMPORT Base INTO
IMPORT PO INTO
IMPORT Booleans INTO
CLASS
  FUNC theta : process -> process
  FUNC _$ _ : process # process -> process
AXIOM
  { 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:

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

\[
\text{effect} : A \times \dot{A}_{\tau} \times \text{Objects} \times \text{States} \rightarrow \text{States}
\]

\[
\begin{align*}
L1 & \quad \Lambda_{m}^{\sigma}(\delta) = \delta \\
L2 & \quad \Lambda_{m}^{\sigma}(\tau) = \tau \\
L3 & \quad \Lambda_{m}^{\sigma}(tx) = \tau \cdot \Lambda_{m}^{\sigma}(x) \\
L4 & \quad \Lambda_{m}^{\sigma}(ax) = \sum_{b \in \text{action}(a,m,\sigma)} b \cdot \Lambda_{m}^{\sigma}(\text{effect}(a,b,m,\sigma))(x) \\
L5 & \quad \Lambda_{m}^{\sigma}(x+y) = \Lambda_{m}^{\sigma}(x) + \Lambda_{m}^{\sigma}(y)
\end{align*}
\]

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 $\tau$, 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 \[ i(pre-tau) = tau; \]

  ATf1 \[ firstAT = pre-tau; \]
  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
4.10. Some Problems
In this section some topics in process algebra are elaborated, which (seem to) have no
globally specified or identifiable 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
work 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
form 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.

\[
\text{RDP} \quad \exists x \quad E(x, -)
\]

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.

\[
\forall n \geq 1 \quad \pi_n(x) = \pi_n(y) \quad E(x, -) \quad (E \text{ guarded, no abstraction})
\]

\[
x = y
\]

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. \[ \alpha(a x) = \{a\} \cup \alpha(x) \quad (a \in A) \]
5. \[ \alpha(x + y) = \alpha(x) \cup \alpha(y) \]

On infinite processes it is defined by:

6. \[ E(x, -) \quad (E \text{ guarded, no abstraction}) \]

\[
\alpha(x) = \bigcup_{n=1}^{\infty} \alpha(\pi_n(x))
\]

7. \[ E(x, -) \quad (E \text{ guarded, no abstraction}) \]

\[
\alpha(\tau(x)) = \alpha(x) - I
\]

The problem arises from the infinite union, which cannot 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:

CA1 \[ \alpha(x) \mid (\alpha(y) \cap H) \subset H \]

\[
\partial_H(x \parallel y) = \partial_H(x \parallel \partial_H(y))
\]

CA2 \[ \alpha(x) \mid (\alpha(y) \cap I) = \emptyset \]

\[
\tau_I(x \parallel y) = \tau_I(x \parallel \tau_I(y))
\]

CA3 \[ \alpha(x) \cap H = \emptyset \]

\[
\partial_H(x) = x
\]

CA4 \[ \alpha(x) \cap I = \emptyset \]

\[
\tau_I(x) = x
\]
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](image-url)
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 p1, ..., p6. 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 p1, and should be transmitted to port p2. 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 p3 and p5 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 p4 and p6, acknowledgements (ac) are communicated. To describe a mutilated message, a checksum-error (ce) can be communicated at ports p4 and p5. 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.

![Diagram](image)

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 :
FUNC penultimateIntType : -> IntType

AXIOM
FORALL x:IntType, y:IntType (
(IFn) firstType = r;
(IN1) next(r) = s;
(IN2) next(s) = c;
(IN3) next(c) = c;
(IN) penultimateIntType = s;

(IN1e) eq(penultimateIntType, next(penultimateIntType)) = false;
(IN2e) eq(next(penultimateIntType), penultimateIntType) = false;
(IN3e) eq(next(x), next(y)) = false -> eq(x, y) = false;
(IN4e) 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 :

AXIOM
FORALL x:Ports, y:Ports (PO) internal(x) = not(eq(x,p1) | eq(x,p2));
f1) 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;
sp) 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

\[
\exists D \vdash
\]
IMPORT Booleans INTO
IMPORT DB INTO

\[
\text{LASS}
\]
SORT D
FUNC ac : D -> D %acknowlegde
FUNC ce : D -> D %checksum error
FUNC j : Data -> D %embed Dataset
FUNC j : DB -> D %embed DB

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

\[
\text{AXIOM}
\]
\[
\text{FORALL x:D, y:D, d:Data, f:DB (}
\]
\[
\text{[Df]} \quad \text{firstD = ac;}
\]
\[
\text{[Dn1]} \quad \text{next(ac) = ce;}
\]
\[
\text{[Dn2]} \quad \text{next(ce) = j(firstDatum);}
\]
\[
\text{[Dn3]} \quad \text{next(j(d)) = if(eq(next(d), d), j(firstDB), j(next(d)));
}[Dn4] \quad \text{next(j(f)) = j(next(f));}
\]
\[
\text{(Dp)} \quad \text{penultimateD = j(penultimateDB);}
\]
\[
\text{(Deq1)} \quad \text{eq(penultimateD, next(penultimateD)) = false;}
\]
\[
\text{(Deq2)} \quad \text{eq(next(penultimateD), penultimateD) = false;}
\]
\[
\text{(Deq3)} \quad \text{eq(next(x), next(y)) = false => eq(x, y) = false;}
\]
\[
\text{(Deq4)} \quad \text{eq(x, x) = true}
\]

\[
\text{END;
}\]

\[
\text{LET Atoms :=}
\]
%Definition of the Atoms
IMPORT Events INTO
IMPORT InteractionType INTO
IMPORT Ports INTO
IMPORT D INTO

\[
\text{CLASS}
\]
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 -> 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 -> SetsOfDay, 
   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
{SuDa1} 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):

\[ K = \sum_{f \in DB} r3(f) \cdot K_f \]
\[ K_f = (s5(f) + s5(ce) + i) \cdot K \]  \hspace{2cm} (f \in DB)

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):

\[ L = r6(ac) \cdot L^{ac} \]
\[ L^{ac} = (s4(ac) + s4(ce) + j) \cdot K \]
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.

$$S = RH^0$$
$$RH^n = \sum_{d \in D} r1(d) \cdot SF^{dn}$$
$$SF^{dn} = s3(dn) \cdot WS^{dn} \quad (d \in Data, \ n \in \{0,1\})$$
$$WS^{dn} = r4(aci) \cdot RH^{1-n} + (r4(ce) + tio) \cdot SF^{dn}$$

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.

$$R = WF^0$$
$$WF^n = r5(ce) \cdot WF^n + \sum_{d \in D} r5(d, 1-n) \cdot SA^n + \sum_{d \in D} r5(d, n) \cdot SH^{dn}$$
$$SA^n = s6(aci) \cdot WF^n$$
$$SH^{dn} = s2(d) \cdot SA^{1-n} \quad (d \in Data, \ n \in \{0,1\})$$

The communication function is defined by:

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

LET Commerce := %Definition of the communication-merge function
IMPORT Atoms INTO
CLASS
  FUNC _|_ : Atoms x 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:
\[ H_0 = \{ \text{st}(f), \text{rt}(f) \mid t \in \{3, 4, 5, 6\}, f \in D \} \]

LET encapset :=
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.

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

LET po :=
IMPORT Atoms INTO
CLASS
FUNC sm : Atoms \& Atoms -> BOOL
% smaller
AXIOM
FORALL a:Atoms, b:Atoms
\{P01\} 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:

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

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

Now the PAR-protocol is described by:

\[ \text{PAR} = \tau_{I_0} \theta_0 \delta_0 (S||R||L) \]

Because the systems ACP_t and ACP_\theta 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 PARR :=

IMPORT Atoms INTO
IMPORT Base INTO
IMPORT SumsOverData INTO
IMPORT SumsOverDB INTO
IMPORT ACP-Theta INTO

CLASS

FUNC K : DB -> process
FUNC K : DB -> process
FUNC L : -> process
FUNC L-ac : -> process
FUNC S : -> process
FUNC RH : -> process
FUNC SF : -> process
FUNC WS : -> process
FUNC R : -> process
FUNC WF : -> process
FUNC SA : -> process
FUNC SH : -> process
FUNC pre-PAR : -> process
FUNC FunK : -> FuncsDBTop
FUNC FunRH : -> FuncsDataTop
FUNC FunWFa : -> FuncsDataTop
FUNC FunWFb : -> 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(iio))).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 PARR :=

IMPORT ACP-Tau INTO
IMPORT PARR :=

INTO
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:

![Diagram](image)

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 (p1, ..., p12), and a sort D, containing the items that are communicated. Such an item can be a single Data-element (at ports p1, p4, p5, p8), a data-frame (from DBB), which is some Data-element enriched with two bits of auxiliary information (at ports p2, p3, p6, p7, p10, p12), or it can be one of the events (time-out and frame-arrival at ports p9 and p11, checksum-error at ports p9, p10, p11, p12). Again there is the sort of Simple Atoms, which do not communicate. It contains the internal actions i and j, the deadlock (delta), 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.

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

```plaintext
LET DBB := %Definition of the set of data to be transmitted
    IMPORT Data INTO
    CLASS
        SORT DBB
        FUNC frame : Data # BOOL # BOOL -> DBB
    ENDCLASS
    FUNC eq : DBB # DBB -> BOOL
    FUNC firstDBB : -> DBB
    FUNC next : DBB -> DBB
    FUNC penultimateDBB : -> DBB
    AXION
        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
FUNC ce  : -> Events
FUNC fa : -> Events

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 :=

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;
  {Por1} next(p1) = p2;
  {Por2} next(p2) = p3;
  {Por3} next(p3) = p4;
  {Por4} next(p4) = p5;
  {Por5} next(p5) = p6;
  {Por6} next(p6) = p7;
  {Por7} next(p7) = p8;
  {Por8} next(p8) = p9;
  {Por9} next(p9) = p10;
  {Por10} next(p10) = p11;
  {Por11} next(p11) = p12;
  {Por12} next(p12) = p12;
  {Pop} penultimateP = p11;
(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(d)) = true;
(Dor2) originDBB(i(d)) = false;
(Dor3) originDBB(j(e)) = false;
(Dor4) originData(i(d)) = false;
(Dor5) originData(i(d)) = true;
(Dor6) originData(j(e)) = false;
(Df1) firstD = j(firstEvent);
(Dn1) next(j(e)) = if(eq(next(e), e), i(firstDBB), j(next(e)));
(Dn2) next(l(dbb)) = if(eq(next(d(bb), dbb), i(firstDatum), i(next(d(bb))));
(Dn3) next(l(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 ee-eq-e : -> SimpleAtoms [%ee=e]
FUNC nn-eq-e : -> SimpleAtoms [%nn=e]
FUNC ee-neq-n : -> SimpleAtoms [%ee>n]
FUNC Fh : -> SimpleAtoms [%Fh(x)]
FUNCTIONS

\begin{align*}
\text{FUNC} \ Sf & : \rightarrow \text{SimpleAtoms} \quad \%[Sf(b,n,l-e)] \\
\text{FUNC} \ Gf & : \rightarrow \text{SimpleAtoms} \quad \%[Gf(bb,nn,ee)] \\
\text{FUNC} \ Th & : \rightarrow \text{SimpleAtoms} \quad \%[Th(bb)] \\
\text{FUNC} \ i & : \text{Events} \rightarrow \text{SimpleAtoms} \quad \%\text{embed Events} \\
\text{FUNC} \ eq & : \text{SimpleAtoms} \# \text{SimpleAtoms} \rightarrow \text{BOOL} \\
\text{FUNC} \ firstS & : \rightarrow \text{SimpleAtoms} \\
\text{FUNC} \ next & : \text{SimpleAtoms} \rightarrow \text{SimpleAtoms} \\
\text{FUNC} \ penultimateS & : \rightarrow \text{SimpleAtoms}
\end{align*}

AXIOM

\begin{align*}
\text{FORALL} \ x: \text{SimpleAtoms},\ y: \text{SimpleAtoms},\ e: \text{Events} \ ( \\
\{\text{Sin1}\} \ next(delta) & = i; \\
\{\text{Sin2}\} \ next(i) & = j; \\
\{\text{Sin3}\} \ next(j) & = n\text{-bec-0}; \\
\{\text{Sin4}\} \ next(n\text{-bec-0}) & = e\text{-bec-0}; \\
\{\text{Sin5}\} \ next(e\text{-bec-0}) & = n\text{-bec\text{-}notn}; \\
\{\text{Sin6}\} \ next(n\text{-bec\text{-}notn}) & = e\text{-bec\text{-}note}; \\
\{\text{Sin7}\} \ next(e\text{-bec\text{-}note}) & = n\text{-eq}\text{-}e; \\
\{\text{Sin8}\} \ next(n\text{-eq}\text{-}e) & = e\text{-eq-}n; \\
\{\text{Sin9}\} \ next(e\text{-eq-}n) & = n\text{-eq-}e; \\
\{\text{Sin10}\} \ next(n\text{-eq-}e) & = e\text{-eq-}n; \\
\{\text{Sin11}\} \ next(e\text{-eq-}n) & = Fh; \\
\{\text{Sin12}\} \ next(Fh) & = Sf; \\
\{\text{Sin13}\} \ next(Sf) & = Gf; \\
\{\text{Sin14}\} \ next(Gf) & = Th; \\
\{\text{Sin15}\} \ next(Th) & = i(\text{firstEvent}); \\
\{\text{Sin16}\} \ next(i(e)) & = i(\text{next}(e)); \\
\{\text{Sin17}\} \ penultimateS & = i(\text{penultimateEvent}); \\
\{\text{Sinv1}\} \ eq(\text{penultimateS}, \ next(\text{penultimateS})) & = \text{false}; \\
\{\text{Sinv2}\} \ eq(\text{next}(\text{penultimateS}), \ penultimateS) & = \text{false}; \\
\{\text{Sinv3}\} \ e \ eq(\text{next}(x), \ next(y)) & = \text{false} \Rightarrow q(x, y) = \text{false}; \\
\{\text{Sinv4}\} \ eq(x, x) & = \text{true})
\end{align*}

END;

LET

\begin{align*}
\text{Atoms} := \quad \%\text{Definition of the Atoms} \\
\text{IMPORT} \ \text{SimpleAtoms} \ \text{INTO} \\
\text{IMPORT} \ \text{InteractionType} \ \text{INTO} \\
\text{IMPORT} \ \text{Ports} \ \text{INTO} \\
\text{IMPORT} \ D \ \text{INTO}
\end{align*}

CLASS

SORT

\begin{align*}
\text{Atoms} \\
\text{IntType} & \# \text{Ports} \# D \rightarrow \text{Atoms} \\
\text{SimpleAtoms} & \rightarrow \text{Atoms} \quad \%\text{embed SimpleAtoms} \\
\text{BOOL} \\
\text{what port is involved?} \\
\text{and what datum?}
\end{align*}

AXIOM

\begin{align*}
\text{FORALL} \ x: \text{Atoms},\ y: \text{Atoms},\ e: \text{SimpleAtoms}, \\
\ t: \text{IntType}, \ t1: \text{IntType}, \ t2: \text{IntType}, \ p: \text{Ports}, \ d: D \ ( \\
\{\text{At1}\} \ \text{has-type}(t, i(e)) & = \text{false}; \\
\{\text{At2}\} \ \text{has-type}(t1, do(t2, p, d)) & = eq(t1, t2); \\
\{\text{At3}\} \ \text{port}(i(e)) & = \text{firstP}; \quad \%\text{default value}
\end{align*}
{At5}  datum(i(e)) = firstD;   %default value
{At6}  datum(do(t,p,d)) = d;

{Atf}  firstAtom = i(firstS);
{Atnl} next(i(e)) = if(eq(next(e),e),
        do(firstType, firstP, firstD),
        i(next(e)));
{Atnl} 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 ))));

{Anl}  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 d:DBB, i:Data, e:Events ( 
{SD1} is-in(origin-Data, i(db)) = false; 
{SD2} is-in(origin-Data, i(d)) = true; 
{SD3} is-in(origin-Data, j(e)) = false; 
{SD4} is-in(origin-DBB, i(db)) = 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
RENAMED
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(Ports-null-set, 
p2), p6), p9), p10), p11), p12);
{SP3} internalports = union(pset1, pset2)
END;

LET SumsOverDBB :=
RENAMED
SORT SET -> SumsOfDBB,
   FUNC null-set -> DBB-null-set,
   FUNC FuncaToP -> FuncaDDBToP
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. OBSDW
Now a brief discussion on the specification of the OBSDW-protocol will be given. The easiest components are the two timers \( T_A \) and \( T_B \). They just offer time-outs at ports \( p^9 \) and \( p^{11} \).

\[
T_A = s^9(tio).T_A \\
T_B = s^{11}(tio).T_B
\]

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.

\[
K = K^e = \sum_{f \in DBB} r^2(f) \cdot K^e \\
K^g^{*e} = (s^7(f) + s^7(ce) + i) \cdot K^g + \sum_{g \in DBB} r^2(g) \cdot K^g^{*g^{*f}} \\
L = L^e = \sum_{f \in DBB} r^6(f) \cdot L^e \\
L^g^{*e} = (s^3(f) + s^3(ce) + j) \cdot L^g + \sum_{g \in DBB} r^6(g) \cdot L^g^{*g^{*f}}
\]

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 \( p^{10} \), while signaling this offer to the \( IMP \) by sending a frame-arrival message at port \( p^9 \). Incoming check-sum errors are also signaled at port \( p^9 \), without using \( p^{10} \).

\[
R_A = \sum_{f \in DBB} r^3(f) \cdot R_A^f + r^3(ce) \cdot R_A^{ce} \\
R_A^f = s^9(fa) \cdot s^{10}(f) \cdot R_A \quad (f \in DBB) \\
R_A^{ce} = s^9(ce) \cdot R_A
\]

Receiver \( R_B \) has the same structure:

\[
R_B = \sum_{f \in DBB} r^7(f) \cdot R_B^f + r^7(ce) \cdot R_B^{ce}
\]
\[ R_B^f = s_{11}(fa) \cdot s_{12}(f) \cdot R_B \quad (f \in \text{DBB}) \]

\[ R_B^{ce} = s_{11}(ce) \cdot R_B \]

The definition of the IMP's follows from the computer program, from [21]. Every IMP takes care of two boolean variables: NextFrameToSent (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 n, e and b, and are examined. (In the specification n, e and 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 (UF). 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 (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 (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.

\[ PA = [n:=0] \cdot [e:=0] \cdot [Fh(b)] \cdot SF \]
\[ SF = [Sf(b, n, 1-e)] \cdot WS \]
\[ WS = tio \cdot SF + ce \cdot SF + fa \cdot GF \]
\[ GF = [Gf(b, n, e)] \cdot T1 \]
\[ T1 = [n=e] \cdot Th(b) \cdot [e:=1-e] \cdot T2 + [n \neq e] \cdot T2 \]
\[ T2 = [e=n] \cdot Fh(b) \cdot [n:=1-n] \cdot SF + [e \neq n] \cdot SF \]

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.

\[ PB = [n:=0] \cdot [e:=0] \cdot [Fh(b)] \cdot WF \]
\[ WF = ce \cdot WF + fa \cdot GF \]
\[ SF = [Sf(b, n, 1-e)] \cdot WS \]
\[ WS = tio \cdot SF + ce \cdot SF + fa \cdot GF \]
\[ GF = [Gf(b, n, e)] \cdot T1 \]
\[ T1 = [n=e] \cdot Th(b) \cdot [e:=1-e] \cdot T2 + [n \neq e] \cdot T2 \]
\[ T2 = [e=n] \cdot Fh(b) \cdot [n:=1-n] \cdot SF + [e \neq n] \cdot SF \]
Now we can transform these sequences of atoms into two meaningful processes \( \text{IMP}_A \) and \( \text{IMP}_B \) by applying the state-operator to \( \text{PA} \) and \( \text{PB} \).

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 \( n, e, b, \alpha, \beta, \delta \) (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 \( \sigma \) by substituting \( 0 \) for variable \( n \).

For all states \( \sigma \), all objects \( m \) and all events \( e \):

1. \[ \Lambda^m_\sigma([n:=0] \cdot x) = \tau \cdot \Lambda^m_\sigma[0/n] (x) \]
   \[ \Lambda^m_\sigma([e:=0] \cdot x) = \tau \cdot \Lambda^m_\sigma[0/e] (x) \]
   \[ \Lambda^m_\sigma([n:=1-n] \cdot x) = \tau \cdot \Lambda^m_\sigma[1-\sigma(n)/n] (x) \]
   \[ \Lambda^m_\sigma([e:=1-e] \cdot x) = \tau \cdot \Lambda^m_\sigma[1-\sigma(e)/e] (x) \]

2. \[ \Lambda^A_\sigma([\text{Fh}(b)] \cdot x) = \sum_{d \in D} r1 (d) \cdot \Lambda^A_\sigma[d/b] (x) \]
   \[ \Lambda^B_\sigma([\text{Fh}(b)] \cdot x) = \sum_{d \in D} r5 (d) \cdot \Lambda^B_\sigma[d/b] (x) \]

3. \[ \Lambda^A_\sigma([\text{Th}(\alpha)] \cdot x) = \sum_{d \in D} r4 (\sigma(\alpha)) \cdot \Lambda^A_\sigma(\alpha) (x) \]
   \[ \Lambda^B_\sigma([\text{Th}(\alpha)] \cdot x) = \sum_{d \in D} r8 (\sigma(\alpha)) \cdot \Lambda^B_\sigma(\alpha) (x) \]

4. \[ \Lambda^A_\sigma([\text{Gf}(\alpha, n, \beta)] \cdot x) = \sum_{d \in D} r10 (d, p, q) \cdot \Lambda^A_\sigma[d/b] \cdot [p/n] \cdot [q/\beta] (x) \]
   \[ \Lambda^B_\sigma([\text{Gf}(\alpha, n, \beta)] \cdot x) = \sum_{d \in D} r12 (d, p, q) \cdot \Lambda^B_\sigma[d/b] \cdot [p/n] \cdot [q/\beta] (x) \]

5. \[ \Lambda^A_\sigma(e \cdot x) = r9 (e) \cdot \Lambda^A_\sigma(x) \]
   \[ \Lambda^B_\sigma(e \cdot x) = r11 (e) \cdot \Lambda^B_\sigma(x) \]

6. \[ \Lambda^m_\sigma([\alpha=n] \cdot x) = \tau \cdot \Lambda^m_\sigma(x) \quad \text{if } \sigma(\alpha)=\sigma(n) \]
   \[ \delta \quad \text{otherwise} \]

\[ \Lambda^m_\sigma([\alpha\neq n] \cdot x) = \tau \cdot \Lambda^m_\sigma(x) \quad \text{if } \sigma(\alpha)\neq \sigma(n) \]

\[ \delta \quad \text{otherwise} \]
\[ \Lambda^{A}_{\sigma}([Sf(b,n,1-e)] \cdot x) = s2(\sigma(b),\sigma(n),1-\sigma(e)) \cdot \Lambda^{A}_{\sigma}(x) \]

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

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,x3,4,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,x3,4,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 effect functions
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

bl: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-notn), 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(Atoms-T-null-set, j(do(s,p4,proj6(st))));
{Ae3b} ACTION(i(Th), B, st) =
    insert(Atoms-T-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(Atoms-T-null-set, j(do(r,p9,j(e))));
{Ae5b} ACTION(i(i(e)), B, st) = insert(Atoms-T-null-set, j(do(r,p11,j(e))));

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

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

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

6.6. OBSW

Now, when $\sigma_0$ denotes an arbitrary initial state, we can define:

\[
\begin{align*}
\text{IMP}_A &= \lambda^A_{\sigma_0}(PA) \\
\text{IMP}_B &= \lambda^B_{\sigma_0}(PB)
\end{align*}
\]

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

\[
\text{st}(f) \mid \text{rt}(f) = \text{ct}(f)
\]

for \( t \in \{2, 3, 6, 7, 9, 10, 11, 12\}, f \in \text{DBBU}\{ce, tio, fa\} \)

\[
\text{LET Commerge :=} \\
\text{IMPORT Atoms INTO} \\
\text{IMPORT SetsPorts INTO} \\
\text{IMPORT SetsS INTO} \\
\text{CLASS} \\
\text{FUNC _|_: Atoms \# Atoms -> Atoms} \\
\text{AXIOM} \\
\text{FORALL a:Atoms, b:Atoms (}
\]

\[
\begin{align*}
\text{IMP}_A &= \lambda^A_{\sigma_0}(PA) \\
\text{IMP}_B &= \lambda^B_{\sigma_0}(PB)
\end{align*}
\]
\{Com1\} \ a \ \mid b = \text{if} ( (\text{has-type}(s,a) \ \& \ \text{has-type}(r,b)) \ \mid \\
\text{has-type}(r,a) \ \& \ \text{has-type}(s,b)) \\
\& \ \text{eq}(\text{port}(a), \ \text{port}(b)) \\
\& \ \text{eq} (\text{datum}(a), \ \text{datum}(b)) \\
\& \ \text{is-in} (\text{internal-ports}, \ \text{port}(a)) \\
\& \ \text{is-in} (\text{origin-DBB-ce-tio-fa}, \ \text{datum}(a)), \\
\text{do}(c, \ \text{port}(a), \ \text{datum}(a)), \\
i(\text{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 (s_3(f)) from the communication channel. This frame can get lost. To guarantee that whenever the receiver is able to accept an offer (r_3(f)), it will not be lost, the following priority is defined (see module PO).

$$\delta a \ a \in A \hspace{1cm} \text{for } a \in A$$

$$s_3(f) < c_3(f) \hspace{1cm} \text{for } f \in \text{DBBU} \cup \{ce\}$$

$$s_7(f) < c_7(f) \hspace{1cm} \text{for } f \in \text{DBBU} \cup \{ce\}$$

\text{LET PO :=}
\text{IMPORT Atoms INTO}
\text{IMPORT SetsD INTO}
\text{CLASS}
\text{FUNC sm : Atoms } \rightarrow \text{ Atoms } \rightarrow \text{ BOOL } \hspace{1cm} \text{%smaller}
\text{AXIOM}
\text{FORALL a:Atoms, b:Atoms (}
\text{PO1) \ sm(a, b) = (eq(a, i(delta)) \ \& \ \text{not} (eq(b, i(delta)))) \\
\text{\mid (\text{has-type}(s,a) \\
\text{\& \ \text{has-type}(c,b) \\
\text{\& \ \text{is-in}(\text{origin-DBB-ce}, \ \text{datum}(a)) \\
\text{\& \ ((eq(port(a), p3) \ \& \ eq(port(b), p3)) \\
\text{\mid (eq(port(a), p7) \ \& \ eq(port(b), p7))})) \\
\text{)})}
\text{END:}

After defining (see module encapsset)

$$H_1 = \{ r_3(f) \mid f \in \text{DBBU} \cup \{ce\} \}$$

$$H_2 = \{ r_7(f) \mid f \in \text{DBBU} \cup \{ce\} \}$$

the following two systems can be defined.

$$\theta_0 \partial_{H_1} (L \parallel R_A)$$

$$\theta_0 \partial_{H_2} (K \parallel R_B).$$

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

$$H_0 = \{ s(t(f), r(t(f)) \mid t \in \{2, 6, 9, 10, 11, 12\}, f \in \text{DBBU} \cup \{ce, tio, fa\} \}$$
\[ I_0 = \{ ct(f) \mid te \{ 2, 3, 6, 7, 9, 10, 11, 12 \}, fe \& DB\{ ce, tio, fa \} \} \\
\cup \{ st(f) \mid te \{ 3, 7 \}, fe \& DB\{ ce \} \} \cup \{ i, j \} \]

\begin{verbatim}
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(origin-DBB-ctio-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-ctio-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

\[ OBSW = t_{i0} \circ \partial_{H_0}(\text{IMP}_A \| T_A \| \theta_0 \circ \partial_{H_2}(K \| R_B) \| \text{IMP}_B \| T_B \| \theta_0 \circ \partial_{H_1}(L \| R_A)) \]

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
\end{verbatim}
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-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));
  [OBS4] app(FunL-eps,f) = i(do(r,p6,i(f))).L(seq(f));
  [OBS5] app(FunK-s,q,f) = i(do(r,p2,i(f))).K(seq(f)+q);
  [OBS6] app(FunL-s-q,f) = i(do(r,p6,i(f))).L(seq(f)+q);

  [OBS7] 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(Ph)).i(i(e-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\} \quad K(q+seq(f)) = (i(do(s,p7,i(f))) + i(do(s,p7,j(co))) + i(i(i)) \cdot K(q)
+ \text{Sum\{DBBSet,Func\}-s\{q+seq(f)\}});

\{OBS28\} \quad L(e) = L(eps);
\{OBS29\} \quad L(eps) = \text{Sum\{DBBSet,Func\}-eps};
\{OBS30\} \quad L(q+seq(f)) = (i(do(s,p3,i(f))) + i(do(s,p3,j(co))) + i(i(j)) \cdot L(q)
+ \text{Sum\{DBBSet,Func\}-s\{q+seq(f)\}});

\{OBS31\} \quad \text{IMPa} = \lambda(A, \text{zero-state}, FA);
\{OBS32\} \quad \text{IMPb} = \lambda(A, \text{zero-state}, FB);

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

\text{END};

\text{LET OBSWPart2 :=}
\quad \text{IMPORT ACP-Tau INTO}
\quad \text{IMPORT SC INTO}
\quad \text{IMPORT SC OBSWPart1 INTO}
\quad \text{CLASS}
\quad \text{FUNC OBSW : \to process}
\quad \text{AXIOM}
\quad \{OBS34\} \quad \text{OBSW = abstr(I0, pre-OBSW)}
\quad \text{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 $S8 \ (s1=s2 \ when \ eq(s1,s2)=true))$ 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 $C1 \ (alb=bla), \ C2 \ ((alb)c=al(bgc))$ and $C3 \ (\delta al=\delta)$ 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 $ACP_{\tau}$ and $ACP_0$. In $ACP_{\tau}$ 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 $ACP_0$ 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:

\[ T_1 = (i(i(nn-eq-e)).i(i(Th)).i(i(e-bec-note)).T_2) + (i(i(nn-neq-e)).T_2) \]

but like:

\[ T_1 = nn-eq-e.Th.e-bec-note.T_2 + nn-neq-e.T_2 \]

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


