ALGEBRAIC SPECIFICATION OF SYNCHRONISATION AND ERRORS: A TELEPHONIC EXAMPLE

Brigitte Biebow (*) and Jacques Hagelstein (**)

(*) Laboratoires de Marcoussis
C.R.C.G.E.
Route de Nozay
F-91460 Marcoussis
France

(**) Philips Research Laboratory
Avenue van Becelaere, 2
B-1170 Brussels
Belgium

ABSTRACT

This paper presents an algebraic specification of the switching module, a component of a telephone switching system. This module exhibits interesting synchronisation properties which lead to consider it as a process. The specification is first presented without error handling, and then refined to include a non trivial error recovery strategy. Thus, we additionally show how error handling, which often obscures specifications, may be postponed and become a systematic refinement of a simpler specification.

1. Introduction

This paper aims at presenting an algebraic specification for a component of a telephonic exchange called a 'switching module', exhibiting synchronisation properties and possible erroneous behaviours. These two aspects admittedly raise problems in the abstract data type framework. We show, on this example, how these problems may be overcome. Incidentally, the switching module is usually implemented in hardware. This paper shows that a common specification language between hardware and software is by no means impossible.

Specification methods able to describe the behaviour of processes cover various levels of abstraction, from operational (e.g. Petri nets) to axiomatic ones (e.g. ACP). In this paper, we are specifically interested in specifying processes by using the classical algebraic approach to data type specification.

An algebra is formed by a collection of 'sorts' -- sets of objects -- and a collection of functions over the sorts. It may be defined by providing the signature of the functions and a set of axioms, thus enabling an axiomatic style of specification. Classically [GM 84], the sorts are used to model data types in programming languages, whereas the functions over the sorts model the functions defined in the programming language. Several approaches have been proposed to use the algebraic framework for the specification of processes.

ACP([BK 83]) provides operators to combine atomic actions into individual processes and sets of cooperating ones. These operators are defined axiomatically and form an 'algebra of communicating processes' in which specifications can be written. Currently, ACP suffers from a lack of integration with classical algebraic specifications of data types.

Another approach describes processes by means of auxiliary data types, specified in the algebraic style. In [Jul 83], processes communicate by applying the functions 'produce' (for the sender) and 'consume' (for the receiver) to a shared object belonging to a 'communication type'. This object is described axiomatically, but the specification has to include an operational part, described by means of a variant of weakest preconditions.

In this article, we propose an algebraic specification technique for processes which is more similar to the one used for abstract data types: in the same way as stacks belong to some sort in the algebra of stacks, processes will belong to some sort in some algebra. The elements of the sort will be the various possible states in which the process may be, all along its life. The functions applicable to these objects will correspond to the various events in which the process may be involved. These ideas will be developed in Section 3.

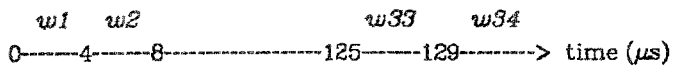
As proned in [Gog 77], the specification of a program should include all exceptional behaviours. All information necessary to the handling of the exceptional state, and only that information, should be kept in case of error. To follow these principles without obscuring the specification with too many details, we propose to provide the specification in three steps: first, just describing the circumstances of errors, secondly, providing error diagnostics, and thirdly, specifying the chosen error handling. These ideas will be developed in Section 4.

In the next section, we present an informal description of the switching module.

2. The switching module

The switching module is a component of a 'time multiplexed digital telephonic system'. A telephonic system is digital when the conversations are transmitted in digital form. To achieve this, the analogic signal delivered by the microphone is sampled 8000 times per second, i.e. every 125 microseconds. A sample takes slightly less than 4 microseconds to be transmitted. It is therefore possible for a line to transmit repeatedly the samples of 32 different conversations (figure 1). This technique is called "time multiplexing". The periodical interval of time allocated to a conversation is called a "channel", and there are thus 32 channels on a line. A channel is allocated to a user for the duration of his talk.

Still, a conversation is not transmitted on one single channel: it follows a path formed by several physical lines connected through nodes. It may be the case that it uses the channel 7 before a node and the channel 30 after it. Therefore, the nodes consist of



w1 = 1st sample of conversation 1
w2 = 1st sample of conversation 2
. . .
w33 = 2nd sample of conversation 1
w34 = 2nd sample of conversation 2

figure 1

'switching modules' performing the switching of the channels. We assume that the network is synchronous (at least around a switching module), i.e. when it is channel 7's time before a node, it is channel 7's time after it too. Therefore, the switching module can only move the samples from one channel to another by delaying them. Of course, the amount of delay depends on the two channels which must be connected: if incoming channel 7 should be switched to 30, all samples reaching the switching module on channel 7 should be delayed by 23 channels, i.e around 90 μs.

Suppose the unit of time is 125/32 μs. Channel 1 is then handled at time 1, 33, 65, etc; channel 7 is handled at time 7, 39, 71, etc. Consider the voice sample or 'word' w received in the switching module at time 7, when channel 7 is handled. At the same time, an irrelevant word (for the considered conversation) goes out, sent on the outgoing channel 7. At time 30, an irrelevant word is received on the incoming channel 30, and the word w is sent on the outgoing channel 30.

Apart from the regularly incoming samples, there are two kinds of commands addressed to the switching module: 'connect' and 'disconnect' specifying two channels to connect or disconnect. Indeed, the channel connections are established and broken dynamically. An input channel may be connected to several output channels, but the reverse is not allowed. An output channel connected to no input channel will contain garbage.

The delicate points in the specification below come from synchronisation and error handling aspects:

* At the same time a word is received and a word is sent: input and output are synchronous.
* Errors happen when trying to disconnect two channels not connected together, or to connect an output channel which is already connected. This should not prevent the module to continue the transmission of words.

In the sequel, we make some simplifying assumptions, to avoid obscuring the specification with useless details. For instance, we assume that the connect and disconnect commands are immediately effective, whereas this could require some time. These simplifications can be removed without any problem, except the increasing size of the specification.

3. Specification of synchronisation in the switching module

3.1. Algebraic specification of processes

Algebraic techniques apply nicely to the specification of objects entirely characterised by the functions that one may invoke to create or modify them. Similarly, the state of a process is entirely characterised by the events that affected the process in the past, i.e. the information that was input and output, and, if relevant, the identification of the originators and destinators. In the spirit of the algebraic approach, we will thus consider process states as objects modified by the application of functions modeling the various events.

Example:
Consider a process which may receive from the outside the messages 'start', 'stop' and 'busy?'. It may issue the messages 'busy!' and 'notbusy!'. The signature of the algebra modeling the process's behaviour will thus include the following functions:

start:	Proc	-->	Proc
stop:	Proc	-->	Proc
busy?:	Proc	-->	Proc
busy!:	Proc	-->	Proc
notbusy!:	Proc	-->	Proc

In addition, the function 'new' produces a process that no event has affected yet.

new:	-->	Proc

An expression such as 'stop(start(new))' describes the state of a process which was created, received the message 'start', and then received the message 'stop'.

The behaviour of the process will be described by stating which sequences of events lead to identical states. As usually, this is achieved by equating expressions formed with the available functions.

In the example above, the equation

stop(new) = new

expresses that a message 'stop' has no effect on a newly created process.

The behaviour of the process is also described by stating which sequences of events are not allowed. For the process above, one may want to state that a stopped process may not issue 'busy!'. A discussion about how to state such facts is given in Section 4.

In the simple example above, the functions take no arguments in addition to the process itself. There is thus a function for each possible event. In general, a function will model a whole family of events, distinguished by the value of possible arguments of the function. It is up to the specifier to cluster the events in an appropriate way, but the problem is generally quite clear. For example, to specify a process able to receive a certain request accompanied by varying data, there would be a function identifying this kind of event and taking the data as argument.

Although it is not mandatory, we found it natural to let incoming messages appear as

arguments of the function modeling the reception, and outgoing messages appear as additional values produced by the function modeling the emission. Consider for instance, a process of type 'Connector' which may receive the message 'connect' asking for the connection of two channels, and will, when the connection is completed, issue the message 'connected' specifying again the two channels. Its signature will include the functions

connect: Connector * Channel * Channel --> Connector
connected: Connector --> Connector * Channel * Channel

When several receptions or emissions are guaranteed to be synchronous, they form one single event and are thus modeled by one single function. This will be illustrated in the specification of the switching module.

3.2. Formal specification of the switching module

In the sequel, we will use the algebraic specification language PLUSS ([BBGGG 84]). It is based upon a set of specification-building operators derived from those of ASL ([Wir 83]) which allow to describe structured abstract data types specifications of realistic size. It includes the notion of multi-target algebras ([BGP 83]) which we will use to specify exception handling in algebraic data types. Errors will however only be considered in Section 4.

The algebraic model of the switching module is an object that records the history of events that have affected it until a certain moment. Those events, modeled by functions, are of the following kinds:

* creation of a new switching module that no event has affected yet
* switching of a channel, at regular intervals of time; this consists in the reception of a word belonging to that channel upstream from the switching module and the simultaneous emission of a word on the same channel downstream
* reception of a command to connect two channels
* reception of a command to disconnect two channels

The channel that is switched by the event of the second kind needs not be specified because the channels are handled in sequence: the nth switching handles the channel number n modulo 32.

The constructors of the sort Sm (for 'switching module') correspond to the events above. Note that the switching operation will be modeled by one single constructor, as the reception of one word and the emission of another one are simultaneous. The data that is received appears as argument while the data being produced appears as result. The constructors are the following ones:

* *init* produces an initial state,
* *inout(s,w)* models the switching of a certain channel on which w is received; the channel depends on the number of times 'inout' has been applied to produce the Sm s; the operation produces two things: a new Sm s' and a word w'; s' is identical to s, except that one more switching event has taken place, during which the word w has been received and w' has been sent; we shall note *inout1(s,w)* and *inout2(s,w)*, the first and

second value of *inout(s,w)*,

- *connect(s,i,j)* produces a Sm *s'*, identical to *s* except that the input channel *i* is now connected to the output channel *j*,
- *disconnect(s,i,j)* produces an Sm *s'*, identical to *s* except that the input channel *i* is now disconnected from the output channel *j*.

We provide also some observers :

- *channel(s)* is the channel to which the next switching of the Sm *s* will refer,
- *identin(s,c)* is the input channel connected to the output channel *c* in the Sm *s*,
- *lastin(s,c)* is the last word input on the channel connected to the output channel *c* in the Sm *s*.

The specification of Sm uses the predefined specifications CHAN (integers from 1 to 32, with addition modulo 32) and WORD. The functions *is* and *isnot* are defined in CHAN and denote the equality and non-equality among objects of the sort.

```
specif SM =
  enrich WORD, CHAN by
  sorts Sm;
  functions
    init:                          -->  Sm;
    inout:        Sm * Word        -->  Sm * Word;
    connect:      Sm * Chan * Chan -->  Sm;
    disconnect:   Sm * Chan * Chan -->  Sm;
    channel:      Sm               -->  Chan;
    identin:      Sm * Chan        -->  Chan;
    lastin:       Sm * Chan        -->  Word;
  variables
    s,s' : Sm;
    w : Word;
    i,j,k,l : Chan;
% i and k are used for input channels, j and l for output channels. %

  axioms
% The following axiom is the heart of the specification. It states which word is output each
time a switching takes place (i.e. at regular intervals). This word, the second value of
inout, is the last word that entered the input channel to which the current output chan-
nel is connected. %

{Word}        inout2(s,w) = lastin(s,channel(s));

% The following axioms describe channel. They express that a new channel is handled
each time inout is invoked. Note that channel(s) denotes the channel that the next inout
will handle. %

{Chan}    channel(init) = 1;
```

channel(inout1(s,w)) = channel(s)+1;
channel(connect(s,i,j)) = channel(s);
channel(disconnect(s,i,j)) = channel(s);

% The following axioms describe *lastin*. *lastin(s,j)* is the last word entered on the input channel connected to j, i.e. on *identin(s,j)*. %

{Word} channel(s) is identin(s,j) ==> lastin(inout1(s,w),j) = w;
channel(s) isnot identin(s,j) ==> lastin(inout1(s,w),j) = lastin(s,j);
lastin(connect(s,k,l),j) = lastin(s,j);
lastin(disconnect(s,k,l),j) = lastin(s,j);

% The following axioms describe *identin*. %

{Chan} j is l ==> identin(connect(s,i,j),l) = i;
j isnot l ==> identin(connect(s,i,j),l) = identin(s,l);
j isnot l ==> identin(disconnect(s,i,j),l) = identin(s,l);
identin(inout1(s,w),j) = identin(s,j);
end SM;

Note that the words arriving on a channel connected to itself are not transmitted instantaneously, but delayed by 32 channels. Instantaneous transmission would have been allowed by replacing the first axiom by:

{Word} inout2(s,w) = lastin(inout1(s,w),channel(s));

4. Error handling

When defining an abstract data type, one has to consider so called error situations. These are produced by operations which are meaningless with the given arguments, for instance popping an empty stack.

Errors raise difficulties in the algebraic framework ([BG 83] for a survey, [Bid 84]), and it is tempting to just exclude them. This can be done by providing a conventional valid object as result of a meaningless operation. For instance, the following equation specifies that popping the empty stack has no effect:

pop(empty-stack) = empty-stack

This strategy has two drawbacks. First, it violates the principle of separation of concerns: both the reader and the writer of a specification will gain in considering first the normal cases, leaving the error handling for later. Second, it provides only a very specific way to handle errors, i.e. invisible recovery.

Other error handlings may be desirable: it may be needed to recognise in the value produced by a function, that an error just happened; this is not the case if the error produces a valid object. While doing so, it may also be needed to distinguish various errors

one from the other. A classical example requiring a more flexible error handling than invisible recovery is a tolerant stack ([BBGGG 84]). It may be popped when empty, but only once before being pushed again. Popping the empty stack twice should lead to an unrecoverable error. This behaviour is described very easily if 'pop(empty-stack)' is an error object which behaves like 'empty-stack' when passed as argument to 'push' (error recovery), but produces a new error when passed to 'pop' (error propagation).

The second goal of this paper is to propose a method for the description of general error handlings. This strategy proceeds in three steps and is based on the definition of disjoints sorts, separating the valid objects from the erroneous ones.

4.1. Multi-target algebras

The simultaneous definition of several sorts is based on multi-target algebras ([BGP 83]), in which functions may produce values in several sorts:

pop: Stack --> Stack ∪ Stack-err;

When multi-target operators occur in a term, it is necessary to state the sort of the term, in a unique way. This is expressed by declarations, such as

{Stack-err} pop(empty-stack);

stating that 'pop(empty-stack)' belongs to the sort 'Stack-err'. The axioms and declarations are extended to positive conditional ones ([Bie 84]). Their general form is:

[{<sort>}] [<condition> ==>] <term> [= <term>];

with the optional parts between brackets. Such an expression states that, [when the condition is verified], the left term belongs to the specified sort [and is equal to the right term]. The sort associated with an axiom indicates the sort on which the property expressed by the axiom is valid. This sort may be omitted if it is the same as the one of the previous axiom.

4.2. The method

This idea of multi-target algebras is used as follows to specify exceptions :

* In a first step, the domains of the functions include only valid objects, while the ranges may contain error objects. Declarations specify when errors take place, and when valid terms are produced. The handling of errors is thus not considered; only their production is specified.
* The second step still does not consider error handling: it provides error diagnostic, i.e. it specifies the constructors of the error sorts. Appropriate arguments are given to these functions and new axioms associate them with the error cases previously defined. These axioms replace the declarations introduced during step 1.
* In a last step, the desired error handling is described. The signature is modified to extend the domains of functions which can now accept erroneous arguments. The ranges of these functions may have to be extended too, depending on the chosen error handling. The axiom set has to be altered for two reasons: (1) to specify the behaviour of functions with arguments in the extension of the domain (specifying at the same

time the type of the result, in case of extended range), and (2) to adapt previous axioms relying on smaller domains or ranges. This will be illustrated in the sequel. New error objects and sorts may need to be defined, as a result of error propagation.

Note that the specification produced in the second step does not preclude any error handling if the term describing an error records any information that could be used for the error handling. This can be done by choosing appropriate arguments for the constructor of the error object.

Consider, for instance, the function 'def-array' which creates an array with given upper and lower bounds. It is an error to define an array with its upper bound inferior to its lower bound. Therefore, the first step of the error specification would include the signature:

```
def-array:    Nat * Nat    -->    Array ∪ Array-err;
```

and the declarations:

```
{Array}       y >= x  ==>  def-array(x,y);
{Array-err}   y < x   ==>  def-array(x,y);
```

In the second step, we would add the function:

```
error-def-array:    Nat * Nat    -->    Array-err;
```

intended to describe the error raised when 'def-array' is invoked with inappropriate arguments. The following axiom would replace the second one above:

```
{Array-err}   y < x  ==>  def-array(x,y) = error-def-array(x,y);
```

The term 'error-def-array(x,y)' records all information available about the error: the axiom associates the name 'error-def-array' with an improper use of 'def-array', while the arguments tells what indices produced the error. Certain error handlings may indeed require to know the value of x and y. For instance 'error-def-array(x,y)' could be handled as 'def-array(x,x)', i.e. the operation would define an array of one element, with x as index.

4.3. The tolerant stack

As a larger example, let us consider the stack that may be popped once when empty. We will define the sorts 'Stack' and 'Stack-err' for erroneous stacks. The first specification describes the valid behaviour and specifies when valid arguments lead to erroneous values. It includes the following functions (we omitted 'top'):

```
empty-stack:                 -->    Stack;
push:   Stack * Elem         -->    Stack;
pop:    Stack                -->    Stack ∪ Stack-err;
```

No function may be applied to erroneous stacks, and only 'pop' may produce such stacks. The first set of axioms gives the usual property of stacks and tells when the value of 'pop' belongs to Stack or Stack-err (s ∈ Stack; x ∈ Elem) :

{Stack} pop(push(s,x)) = s;
{Stack-err} pop(empty-stack);

In a second step, we introduce constructors for the error sorts. There is only one declaration for erroneous objects. Thus, we need only introduce one function:

bad-pop: --> Stack-err;

Bad-pop does not take any argument, because there is no need to distinguish the various circumstances in which the error may take place (there is only one: popping the empty stack). The second declaration is replaced by the following axiom:

{Stack-err} pop(empty-stack) = bad-pop;

In a third step, we want to specify how 'push' and 'pop' handle the error object 'bad-pop'. The two domains are extended to include 'Stack-err'. As 'push' will recover the error, its range is still limited to 'Stack'. However, 'pop' will produce an unrecoverable error, of the new sort 'Stack-err1', when given 'bad-pop' as argument. The arity of the functions thus becomes:

push: (Stack ∪ Stack-err) * Elem --> Stack;
pop: Stack ∪ Stack-err --> Stack ∪ Stack-err ∪ Stack-err1;

New axioms are added to specify the behaviour of the functions on error arguments:

{Stack} push(bad-pop,x) = push(empty-stack,x);
{Stack-err1} pop(bad-pop);

Note that the second axiom completes the specification of the type of 'pop' for all kinds of arguments.

As Stack-err1 does not belong to the domain of 'pop' and 'push', terms such as 'pop(pop(bad-pop))' or 'push(pop(bad-pop),x)' simply do no exist in the described model. This is part of the error handling we chose to specify: no further computation is allowed after popping the empty stack twice. Another possibility would have been to allow such computations, and to continuously obtain the same object in 'Stack-err1'.

4.4. Error handling in the switching module

Let us now apply the previous strategy to the 'switching module' example. Errors may happen in the following cases:

- when disconnecting two channels which are not connected together,
- when connecting an output channel which is already connected (an output channel may only be connected to one input, although an input may be connected to several outputs).
- when determining the input channel connected to a non connected output channel,
- when determining the word coming out of channel *j* when there is no input channel connected to it.

For each sort in the range of 'connect', 'disconnect', 'identin' and 'lastin', we have to define a corresponding error sort. Let *Sm-err*, *Chan-err* and *Word-err* denote the error

sorts corresponding respectively to *Sm*, *Chan* and *Word*. The functions 'is' and 'isnot' are defined in *Chan* ∪ *Chan-err* and denote the equality and non-equality among objects of the sorts.

We introduce a new observer, *free*, to ease the expression of error cases: *free(s,c)* is true if and only if the output channel *c* is not connected to any input channel in the Sm *s*. We thus use the predefined specification BOOL in addition to WORD and CHAN.

In the first step, we just define when errors take place (for brevity, we will denote $X \cup X\text{-}err$ by means of *Xer*):

```
functions
  init:                          -->  Sm;
  inout:        Sm * Word        -->  Sm * Worder;
  connect:      Sm * Chan * Chan -->  Smer;
  disconnect:   Sm * Chan * Chan -->  Smer;
  channel:      Sm               -->  Chan;
  free:         Sm * Chan        -->  Bool;
  identin:      Sm * Chan        -->  Chaner;
  lastin:       Sm * Chan        -->  Worder;
variables
  s,s' : Sm;
  w : Word;
  i,j,k,l : Chan;

axioms
```

% The word output at each switching may be an erroneous one. %

{Worder} inout2(s,w) = lastin(s,channel(s));

% *channel* is never applied to an erroneous *Sm*. Its axioms are left unchanged %

```
{Chan}  channel(init) = 1;
        channel(inout1(s,w)) = channel(s)+1;
        channel(connect(s,i,j)) = channel(s);
        channel(disconnect(s,i,j)) = channel(s);
```

% The following axioms define *free*. %

```
{Bool}  free(init,j) = true;
        free(inout1(s,w),j) = free(s,j);
        j is l  ==>  free(connect(s,i,j),l) = false;
        j isnot l  ==>  free(connect(s,i,j),l) = free(s,l);
        j is l  ==>  free(disconnect(s,i,j),l) = true;
        j isnot l  ==>  free(disconnect(s,i,j),l) = free(s,l);
```

% lastin(s,j), the last word entered on the input channel connected to *j*, is erroneous if *j* is free or if its connection to an input channel is too recent and no word arrived yet on this channel. %

```
{Word}      channel(s) is identin(s,j) ==> lastin(inout1(s,w),j) = w;
{Word-err}  l is j ==> lastin(connect(s,k,l),j);
            free(s,j) ==> lastin(s,j);
{Worder}    channel(s) isnot identin(s,j) ==> lastin(inout1(s,w),j) = lastin(s,j);
            l isnot j ==> lastin(connect(s,k,l),j) = lastin(s,j);
            l isnot j ==> lastin(disconnect(s,k,l),j) = lastin(s,j);
```

% identin(s,c) is erroneous if no input channel is connected to *c* in *s*. %

```
{Chan}      j is l ==> identin(connect(s,i,j),l) = i;
{Chan-err}  free(s,j) ==> identin(s,j);
{Chaner}    j isnot l ==> identin(connect(s,i,j),l) = identin(s,l);
            j isnot l ==> identin(disconnect(s,i,j),l) = identin(s,l);
            identin(inout1(s,w),j) = identin(s,j);
```

% The following declarations specify the type of *connect* and *disconnect*. %

```
{Sm}        free(s,j)) ==> connect(s,i,j);
            identin(s,j) is i ==> disconnect(s,i,j);
{Sm-err}    not(free(s,j)) ==> connect(s,i,j);
            identin(s,j) isnot i ==> disconnect(s,i,j);
```

In a second step, we define, for each declaration of an error sort, a constructor associated with the corresponding error. For brevity, we will limit ourselves to the constructors of Sm-err:

```
bad-connect:      Sm * Chan * Chan    -->    Sm-err;
bad-disconnect:   Sm * Chan * Chan    -->    Sm-err;
```

The following axioms replace the two last declarations above:

```
{Sm-err}    not(free(s,j)) ==> connect(s,i,j) = bad-connect(s,i,j);
            identin(s,j) isnot i ==> disconnect(s,i,j) = bad-disconnect(s,i,j);
```

In the third step, we will specify a non trivial error handling. As suggested in Section 2, bad connections or disconnections are without effect on the transmission of words but will prevent the effect of any other operation (i.e. further connect or disconnect). It is up to the controller of the switching module to reinitialise it in such cases. This is an example of partial recovery of errors.

To describe this behaviour, the second value of 'inout' will not be affected by the error, but its first value will belong to *Sm-err* if an error object is given as argument. Both

'connect' and 'disconnect' also produce an element of *Sm-err* when taking one as argument. The new arity of the functions is as follows (the domain of *channel* has not been extended to *Sm-err* as it is never applied to error objects):

```
init:                                  -->  Sm;
inout:          Smer * Word            -->  Smer * Worder;
connect:        Smer * Chan * Chan     -->  Smer;
disconnect:     Smer * Chan * Chan     -->  Smer;
channel:        Sm                     -->  Chan;
free:           Smer * Chan            -->  Bool;
identin:        Smer * Chan            -->  Chaner;
lastin:         Smer * Chan            -->  Worder;
bad-connect:    Sm * Chan * Chan       -->  Sm-err;
bad-disconnect:    Sm * Chan * Chan    -->  Sm-err;
```

The axiom given for *inout* in the first step specifies its behaviour when its first argument belongs to *Sm* (the variable *s* is typed). The following ones are concerned with arguments in *Sm-err*. They specify a partial error recovery: words continue to be output as if no error had arisen. %

```
{Worder}    inout2(bad-disconnect(s,i,j),w) = inout2(s,w);
            inout2(bad-connect(s,i,j),w) = inout2(s,w);
```

On the other hand, the errors are propagated by *connect*, *disconnect* and the first value of *inout*:

```
{Sm-err}    inout1(bad-disconnect(s,i,j),w) = bad-disconnect(inout1(s,w),i,j);
            connect(bad-disconnect(s,i,j),k,l) = bad-disconnect(s,i,j);
            disconnect(bad-disconnect(s,i,j),k,l) = bad-disconnect(s,i,j);
```

The auxiliary functions *channel*, *free*, *lastin*, and *identin* are only used to define the word output by inout. As this word is not affected by connection errors, the functions will recover the error. We add the following axioms:

```
{Bool}      free(bad-disconnect(s,i,j),l) = free(s,l);
{Worder}    lastin(bad-disconnect(s,i,j),l) = lastin(s,l);
{Chaner}    identin(bad-disconnect(s,i,j),l) = identin(s,l);
```

Note that most axioms given for *free*, *lastin*, and *identin* in the first step rely on the assumption of a smaller domain. For instance,

```
{Bool}    j is l  ==>  free(disconnect(s,i,j),l) = true;
```

was only valid because *free* did only take elements of *Sm* as argument, and thus

disconnect(s,i,j) was assumed not to be an error object. This is not the case any more. The condition of this axiom has to be strengthened to guarantee that 'disconnect(s,i,j)' is a valid object. We replace it by:

{Bool} (identin(s,j) is i) and (j is l) ==> free(disconnect(s,i,j),l) = true;

A similar transformation is applied to the other axioms, when needed. This completes the specification of the switching module, with error handling.

5. Conclusions

The first goal of the paper was to show that processes may, to a certain extend, be specified in the algebraic style, in the same way as passive objects. The key idea is in the appropriate interpretation of the meaning of a function. The advantage of this approach is to provide a uniform framework for the description of parallel and sequential parts of system. However, this goal is not yet reached. What we have illustrated is the local specification of the behaviour of a process, and not yet the specification of the interaction of several ones. This work is currently going on.

Another limitation of this formalism is that it does not, in its current form, allow for real-time specifications, i.e. specifications involving absolute values of time. For instance, we cannot say that 'inout' takes place every 125 μs. This limitation is common to most other specification techniques.

The second goal of the paper was to propose a systematic approach to the specification of error handling. This contribution is more of a methodological nature. The idea to distinguish the specification of normal cases and error situations is an illustration of the principle of separation of concern. Considering the size of an actual specification, its writing, reading or modification would nearly be possible without a clever application of this principle. But it is also important to introduce early the error cases. What we propose is to distinguish at the very beginning error cases from normal ones in an economic way, and to postpone the real choices of error handling.

6. Acknowledgements

We thank Professor Marie-Claude Gaudel for her helpful suggestions.

7. Bibliography

[BBGGG 84] M. Bidoit, B. Biebow, M-C. Gaudel, G. Guiho, C. Gresse, "Exception Handling: Formal Specification and Systematic Program Construction", International Conference on Software Engineering, Orlando, Florida, March 1984.

[Bid 84] M. Bidoit, "Algebraic specification of exception handling and error recovery by means of declarations and equations", Proceedings ICALP 84, LNCS 172.

[Bie 84] B. Biebow, "Application d'un langage de spécification algébrique à des exemples téléphoniques", 3rd cycle thesis of the University of Paris 6, Paris, France, February 1984.

[BG 83] M. Bidoit, M-C. Gaudel, "Etude des méthodes de spécification des cas d'exceptions dans les types abstraits algébriques", Actes du Séminaire d'Informatique Théorique du LITP 1982-1983, Paris 6, Paris, France.

[BGP 83] F. Boisson, G. Guiho, D. Pavot, "Algèbres à Opérateurs Multicibles", LRI report, Orsay, France, June 1983.

[BK 83] J.A. Bergstra, J.W. Klop, "Process Algebra for Communication and Mutual Exclusion", Report IW 218/83, Mathematisch Centrum, Amsterdam.

[GM 84] J.A. Goguen, J. Meseguer, "An initiality primer", SRI International, Computer Science Laboratory, Menlo Park CA 94025, USA.

[Gog 77] J.A. Goguen, "Abstract errors for abstract data types", Description of Programming Concepts, E.J.Neuhol Ed., North Holland, New York, 1977.

[Jul 83] J. Julliand, "Spécification algébrique de la communication entre processus parallèles", Technique et Science Informatiques, Vol. 2 Nr 4, 1983.

[Wir 83] M. Wirsing, "Structured Algebraic Specifications: A Kernel Language", Technische Universität München, 1983.