Service Independent Building Blocks-I; Concepts, Examples and Formal Specifications

Steven Klusener

CWI

P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

e-mail: stevenk@cwi.nl

Bas van Vlijmen & Arjan van Waveren

University of Amsterdam, Programming Research Group

P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

e-mail: vlijmen@fwi.uva.nl & waveren@fwi.uva.nl

Abstract

In this paper we present descriptions of some Service Independent Building Blocks and related concepts like Global Service Logic, which form part of the conceptual model for Intelligent Networks. These descriptions correspond with the formal specifications that are given in the appendix, however they differ slightly from the definitions in the CCITT Recommendation. The differences occur at points at which the Recommendation was not clear to us and they are discussed in this paper as well.

A formal specification is useful for a thorough understanding of the Service Independent Building Blocks. Furthermore, it enables simulation of services and it can serve as a basis for studies of the correctness of implementations.

1991 Mathematics Subject Classification: 68Q60, 68Q65
1991 CR Categories: 69D21,69J13
Keywords & Phrases: Intelligent Networks, CCITT Recommendation Q.1200, Formal Specification, PSF
Note: The contribution of the first author is sponsored by the RACE project BOOST.

Introduction

Over the last ten years the Telecommunication Industry has been working on increasing the number of services supplied by Telephone Networks. Examples of these services are *Abbreviated Dialling*, *Freephone*, *Call Forwarding* and *Automatic Alternative Billing*. The latter service enables the user to charge his/her account by giving the account code and the associated pin code. Intelligent Networks are Telephone Networks which provide these non-trivial services.

The CCITT has formulated a collection of services which are to be covered initially. Together with a conceptual model and an architecture this is described in the *Recommendation*, [CCITT92]. The contents of this Recommendation are also known as *Capability Set 1*. Other references for Intelligent Networks are [KD92],[Sod],[dB90].

The design and implementation of these services and networks are comprised by the field of Service Engineering. So, we inherit the issue of the control of the design, implementation and maintenance of these services. Recent developments in Software Engineering have shown us that it is worth to spend a considerable amount of effort on the specification phase to find inconsistencies, ambiguities in the designed model as soon as possible. If these errors are found in a later phase then much more effort is needed to solve them. To support clear specifications, formal specification languages are necessary.

In this paper we study mainly the *Global Functional Plane* of the conceptual model for Intelligent Networks, which contains four planes in total. The *Global Functional Plane* consists of a collection of *Service Independent Building Blocks* from which the services are constructed.

We have given a specification of some Service Independent Building Blocks, the Basic Call Process, the Global Service Logic and The Automatic Alternative Billing Service in the formal specification language PSF (Process Specification Formalism) [MV90],[Mau91]. These formal specifications can be found in the appendix. While trying to give a formal specification which is equivalent, as far as possible, with the Recommendation we encountered several points at which the Recommendation was unclear. We have resolved these and, hence, our definitions differ slightly from the Recommendation. In this paper we give the informal definitions, which are consistent with the formal specifications, and we discuss how they differ from the definitions in the Recommendation.

It is not at all our intention to propose new definitions for these Service Independent Building Blocks. We hope to get feedback from experts in Intelligent Networks; they can tell us whether our point of view corresponds with their intuition or not. If not, we will adapt our formal specifications. In this way we hopefully obtain a formal specification of the Service Independent Building Blocks (and related concepts like the Basic Call Process and the Global Service Logic) which correspond with the intuition of the experts. Such a formal specification has the following advantages:

- From the formal specification we can again extract an informal one which is probably easier to read than the Recommendation, especially for those without a background in Telecommunication.

- Due to the modularity of the formal specification it is probably easier to see what can be reused in case of extensions of the Intelligent Network model, such as Capability Set 2.

- The PSF toolset contains a *simulator* which enables one to "run" a service. In this way it is easier to test whether a specification of a service corresponds with the intuition; there is less need to build a prototype on an existing network.

- Several phenomenons can be studied more concretely. For example, *feature interaction* is the problem that features of different services may "clash". For more information of feature interaction and the corresponding need of formal methods we refer to [KD92].

- In the conceptual model the plane below the Global Functional Plane is called *Distributed Functional Plane* consisting of *Functional Entities*. If also these Functional Entities are formally defined then the relations between these planes can be studied in more detail.

We will consider some Service Independent Building Blocks as processes and others as functions over abstract data types. Therefore, we have used the language PSF, since it is built on the process specification language ACP ([BW90]) and the abstract data type specification language ASF ([BHK89]). Moreover, PSF features several powerful modularization constructs which enable modular design. In all these aspects PSF resembles LOTOS [ISO87]. We have chosen for PSF for the simple reason that it is developed at the University of Amsterdam. For us it is an interesting test case whether PSF and its toolset are suited for this specification study on Service Independent Building Blocks. We expect that a similar study can be done in LOTOS as well.

The RACE Projects BOOST and SCORE

Both in the RACE project BOOST (Broadband Object Oriented Service Technology, RACE R2076) and in the RACE project SCORE (Service Creation in Object Oriented Reuse Environment, RACE R2017), Intelligent Networks are a central topic of study (see also [Cam92]). For documentation on BOOST and Intelligent Networks we refer to [CET92].

At the Dutch PTT research laboratory Wiet Bouma & Han Zuidweg are working on a specification of the Basic Call Process in LOTOS ([BZ93]), which is done in the context of SCORE. We will exchange experiences and results, hoping that in that way we will mutually benefit from each others work.

Acknowledgements

The authors would like to thank Wiet Bouma, Kees Middelburg and Han Zuidweg of the Dutch PTT Research Centre in Leidschendam, for providing us with the literature on Intelligent Networks and for having fruitful discussions on earlier versions of this paper.

Contents

1 An introduction to the Intelligent Networks model

1.1 The four layer model

An Intelligent Network is a collection of connected *physical entities*, such as telephone switches, databases and computers, which provides services to the end user of the Network and the system manager which controls the Network.

So, there is a high level of Services and a low level of physical entities upon which these services are implemented. To guide and structure the reasoning over these services two more levels have been identified in the Recommendation:

- The *Global Functional Plane*
  This level consists of *Service Independent Building Blocks* (SIB's). Different SIB's do not have functionalities in common and each service can be constructed by *chaining* several SIB's together. In this paper we will consider two types of SIB's; *process* SIB's and *data* SIB's. This distinction does not occur in the Recommendation.

  Furthermore, this level consists of the *Basic Call Process* and the *Global Service Logic*. The Basic Call Process handles all calling for which no additional services are required. As soon as it receives a request for a service it delivers the request to the Global Service Logic which is responsible for the initiation of the right service.

- The *Distributed Functional Plane*
  For each physical entity several (logical) *functions*, or *Functional Entities* (FE's), can be identified. One such Functional Entity can be provided by different physical entities. The Distributed Functional Plane is the collection of all these Functional Entities.

These two planes together with the *Service Plane* and the *Physical Plane* give the four plane model as shown in figure 1. In this paper we will not consider the Distributed Functional Plane and the Physical Plane anymore.

In the Recommendation, Services are subdivided in *Service Features*, which are implemented by Service Independent Building Blocks. In this paper we will not apply the concept of Service Features.

1.2 The Basic Call Process and the Global Service Logic

The Basic Call Process (BCP) is, according to the Recommendation, a special SIB that resides at the Global Functional Plane. It is however not clear to what extend the Basic Call Process can be viewed as a Service Independent Building Block. We consider the Basic Call Process as the process at the Global Functional Plane that is able to handle a call which does not require any services. If on the other hand a service is requested during a call, the Global Service Logic is invoked by the Basic Call Process to initiate that service.

The invocation of the Global Service Logic happens on a so called Point Of Initiation (POI). Control is returned from the Global Service Logic to the Basic Call Process at a so called Point Of Return (POR). There can be more than one Point Of Return in the

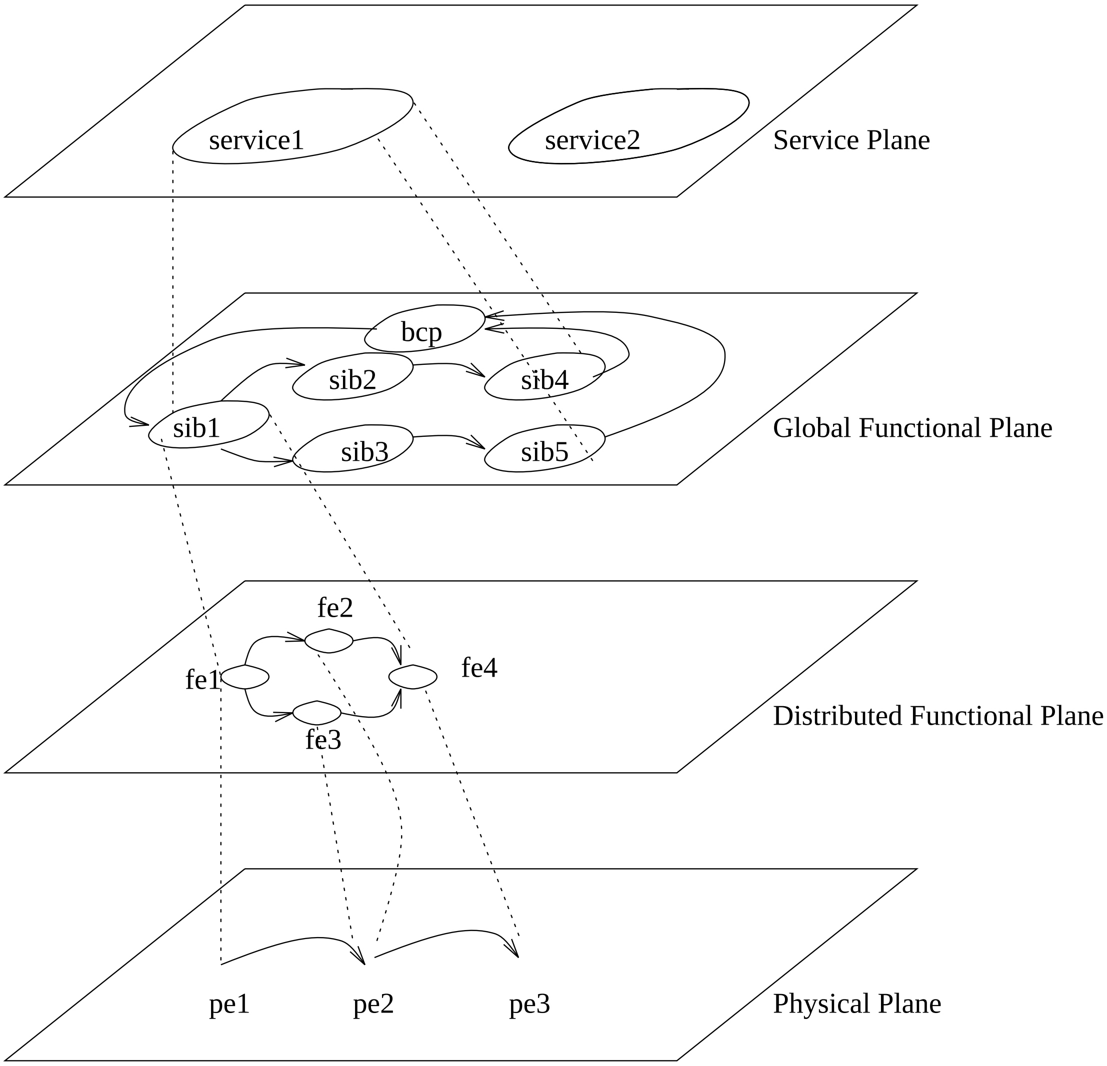

Figure 1: Graphical Representation of the four layer IN model

Basic Call Process at which the Global Service Logic can transfer control again to the Basic Call Process. As a consequence the Global Service Logic can influence the flow of control in the Basic Call Process.

1.3 The SIB's

In this section we discuss the general format of the other SIB's, it corresponds with the Recommendation (Q1203).

1.3.1 The Parameters

A SIB has two kinds of input parameters, *Service Support Data* and *Call Instance Data*. Service Support Data are those parameters which are static for each service in which the SIB is used. Furthermore, there are parameters which depend on the specific run of a service, they are dynamically bound, and these parameters are the Call Instance Data.

Take for example the SIB Verify. It has two strings as parameters, say s and f. The parameter f denotes a format and Verify checks whether the string s fits into the format defined by f. This SIB can be used in a service which must read a pincode from a user. In this case f will be NNNN, which denotes four digits, so f is static with respect to this service and therefore f is a Service Support Data. On the other hand, s is the pincode given by the user during the run of the service, hence s is a Call Instance Data.

1.3.2 The Endpoints and their Output

A process SIB may have different points of termination, which are called *endpoints*. The output, which is sent by the SIB to the next SIB in the chain, depends on the specific endpoint.

1.3.3 The Interface Diagram

When we chain SIB's together to build a service we have to know the interface of each of the SIB's. That is, its Service Support Data (SSD), Call Instance Data (CID) and, in case of a process SIB, the endpoints and the type of output for each endpoint. In the sequel we will present the interface of each SIB by a so called *interface diagram*.

Assume we have a process SIB, *P-Check*, which has as Service Support Data the parameters min, max of sort NAT, the sort of *natural numbers*. Furthermore, as Call Instance Data it has a parameter of sort STRING, the sort of *strings*. In case of succesfull termination it outputs an element of sort BOOL, the sort of the *booleans*. Finally, there are two cases of error; the input string may be to small or to large. The name 'P-Check' in the left upper corner denotes the name of the SIB and the name 'p-check' in the middle of the diagram denotes the name of the process which is offered by the SIB. It may seem redundant to present both names. In the sequel we will see that if we instantiate one SIB more than once in the same context (with different Service Support Data) then we can rename the names of the processes, which are offered by the instances, to distinguish them. In that case it remains clear from which instance of a SIB these processes originate. The interface diagram of P-Check is given in Figure 2.

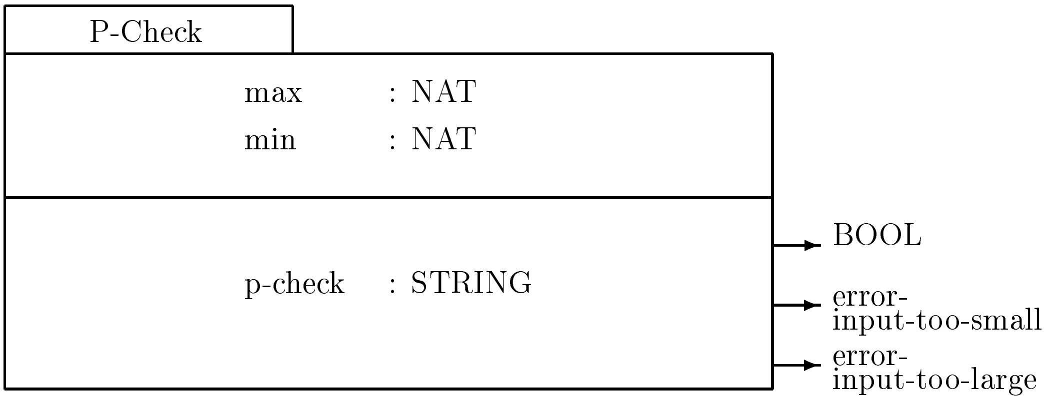

Figure 2: Interface Diagram of P-Check

A data SIB differs from a process SIB since it does not have a dynamic behaviour; it delivers an element of some sort depending on its input arguments. Assume a SIB, *D-Check*, which has the same Service Support Data and Call Instance Data as P-Check. Furthermore, it delivers a BOOL. Hence, the SIB D-Check offers a function d-check of type STRING -> BOOL. Its interface diagram is given in Figure 3.

1.4 Building a service from SIB's

Recall that a service is constructed by chaining SIB's together. If a SIB occurs in a certain chain the output of this SIB must be read by the chain. Since this SIB can be active in different chains at the same time we have to be careful that the output is sent to the right chain. Therefore, we give a *job number* to each chain. Moreover, every process SIB will have the job number as parameter. A process SIB will send his output together with the current job number. Hence it is guaranteed that the chain which has started that process SIB will read its output as well. We will not discuss these job nummers anymore in the paper itself (they will not be shown in the interface diagrams) but they

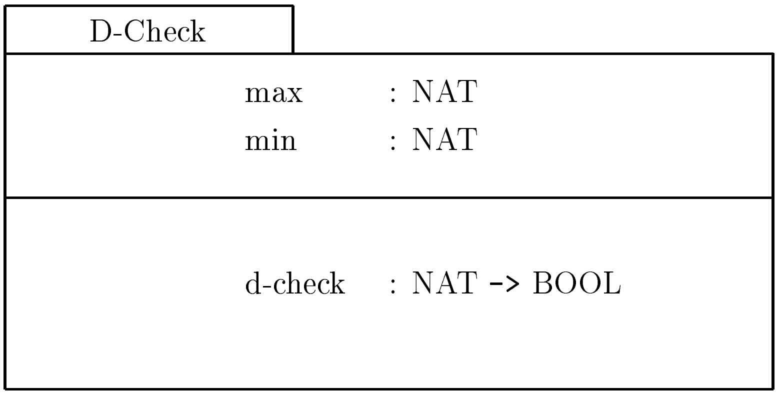

Figure 3: Interface Diagram of D-Check

will play a role in the formal specification.

1.5 The Environment; the Clock, the Dbase and some Telephones

In this paragraph we introduce three processes that are necessary for the specification of the Basic Call Process, the other SIB's and the Global Service Logic.

The behaviour of the service may be time dependent, for example if the user waits to long to give his pincode after the service has been started then an error message must be sent. Therefore, constructs like *time-outs* must be available when we design our SIB's and thus we have to include a notion of a *Clock* in our specification.

Furthermore, the costs of a service are charged to some account. So, we need a database in which the accounts and their chargings are stored.

With Telephones an user annex telephone is modelled. This is not strictly necessary for technical reasons, it just makes the whole more vivid. A Telephone can play an active or a passive role in a call. In the active role a Telephone calls another Telephone, in the passive role a Telephone is the one that is called.

2 Some SIB's

In this section we introduce all the SIB's which are necessary to construct the *Automatic Alternative Billing* service and discuss their informal definition and their interface. During the formal specification of these SIB's several indistinctions in the Recommendation appeared to us. Therefore, we have applied a slightly different point of view on most SIB's, in order to give a consistent formal specification. The formal specifications of these SIB's will be given in Appendix B. The differences with the Recommendation will be discussed as well in this Section.

In the Recommendation for each Call Instance Data of an arbitrary SIB there occurs a so called *Field Pointer* in the Service Support Data as well. For the time being we consider these Field Pointers as irrelevant implementation details, hence we will not consider them any more in the paper. If they are really relevant then their necessity will appear at the point where we cannot continue neglecting them.

Remember that it is not our intention to propose a different point of view on Intelligent Networks in general and on SIB's in particular. The differences with the Recommendation show the points at which the Recommendation was not clear to us. That means that by discussing these differences with experts from the area of Intelligent Networks either the experts can clarify these points by which we can adapt our definitions and formal specifications, or we might show these experts where the Recommendation can be changed.

2.1 Charge

2.1.1 Its informal definition and interface

The SIB Charge treats the charging of an account for the use of a service and a resource.

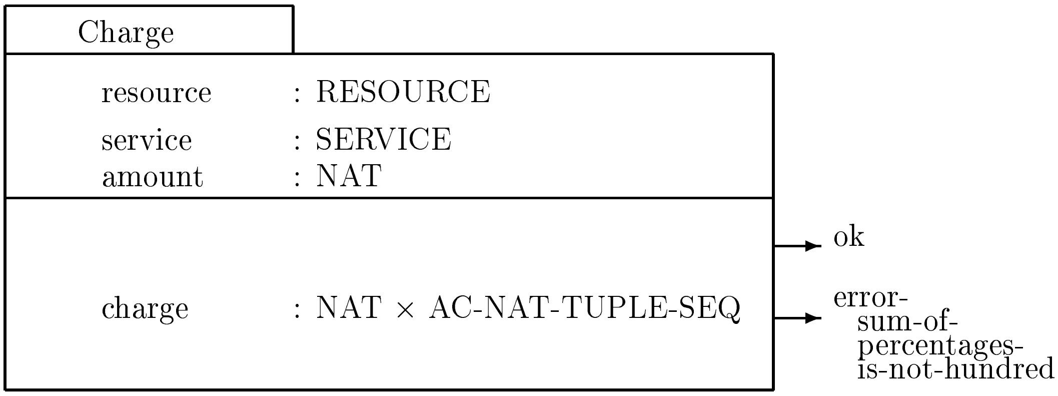

Figure 4: Interface Diagram of Charge

Its Service Support Data are a resource, and a service. As Call Instance Data it has an amount and a sequence of pairs, each pair consists of an account and a percentage (a natural number). If the sum of all percentages equals hundred, then for each pair (`ac`,`n`) in the sequence the account `ac` is charged for `n` percent of the amount. Thus, for each pair in the list an update is sent to the database. Finally the message `ok` is outputted. If the sum of all percentages does not equal hundred, then the output is an error message.

The data type of pairs of accounts and naturals is denoted by `AC-NAT-TUPLE` and the data type of sequences over these pairs is denoted by `AC-NAT-TUPLE-SEQ`. The interface diagram of Charge is given in Figure 4.

2.1.2 Correspondence with Recommendation (Q.1213-2.2)

We have simplified the definition of Charge as given in the Recommendation a little. In the Recommendation also a line number to be charged can be given. We assume that for each line number the associated account can be found. In that case the line number is redundant and so we can leave it out as argument.

Furthermore, in the Service Support Data we have left out the *Units*, and in the Call Instance Data we have left out the *Pulse Metering*. We assume that the first Call Instance Data of Charge, that is the amount to be charged, is expressed in terms of these units and the pulse metering. Hence, it is not necessary to have them as seperate parameters.

In the Recommendation it is stated explicitly that the standard charging is done by the Basic Call Process. However, it is not clear to us why Charge could not handle this standard charging as well.

2.2 Screen

2.2.1 Its informal definition and interface

Screen checks whether an item is in a list. The list is part of the Service Support Data and the item is part of Call Instance Data. Screen is parameterised by the data type of the item. Its interface diagram is given in Figure 5.

```
Screen
  SCREEN-ITEM
  screen-list   : SCREEN-ITEM-SEQ
  ---------------------------------
  screen        : SCREEN-ITEM -> BOOL
```

Figure 5: Interface Diagram of Screen

2.2.2 Correspondence with Recommendation (Q.1213-2.8)

In the Recommendation nothing is said about the parameterisation of Screen by the data type `SCREEN-ITEM`. In fact the parameter mechanisms of Service Support Data and Call Instance Data are only suited for passing constants but not data types.

2.3 Translate

2.3.1 Its informal definition and interface

Translate supplies some functions which transform telephone numbers or account numbers to accounts. It does not have any Service Support Data. As Call Instance Data it has a telephone number or an account number, which is to be translated. Its interface diagram is given in Figure 6.

2.3.2 Correspondence with Recommendation (Q.1213-2.11)

In the Recommendation the translating functions are part of the Service Support Data. Translate itself does nothing more than applying these functions. In case the function is not defined on the telephone number or account number an error message is output.

In our approach we distinguish process SIB's and data SIB's. The functions which are supplied by the data SIB's can be applied when the process SIB's are chained together;

```
Translate

  ---------------------------------------------
  translate   : TELNR          -> ACCOUNT
  translate   : ACCOUNT-CODE   -> ACCOUNT
```

Figure 6: Interface Diagram of Translate

in other words, if we would add these translate functions to the Service Support Data the SIB itself wouldn't do anything, except exporting what it imports. So, either this SIB vanishes or it defines the translating functions itself, which it can supply to the services.

2.4 User-interaction

2.4.1 Its informal definition and interface

User-interaction is responsible for the exchange of information between the user and the network. The network sends announcements to the user while the user sends strings, as sequences of characters, to the network. Its definition is quite complicated, due to the different mechanisms and timing constructs upon which it is based.

Basically, User-interaction sends several announcements (*beeps*) to the user expressing that it is ready for input. After a beep it takes some time until the next beep, this is the SSD-parameter `rep-interv`. The maximal number of beeps can depend on the time (in case `rep-type` is `time`) in which case it sends beeps until a time-out is reached. Or the maximal number of beeps is fixed to a constant (in case `rep-type` is `count`). Moreover, it can also be the case that no repetition of beeps is needed, therefore we have a boolean SSD-parameter `rep-requested`.

If the user has given input then no beeps occur anymore. There is a maximal time interval in between the two inputs of consecutive characters, the SSD-parameter `inter-char-interv`. If the user does not give the next character within this interval the error `error-char-interv-elapsed` is encountered.

The two SSD-parameters `min` and `max` determine the minimal and maximal length of the input string. The SSD-parameter `end-char` determines the character by which the input string must be closed, if this parameter is `nil` then the maximal number of characters is read.

Finally, if the maximal number of beeps is given to the user then User-interaction waits for input until the time-out `initial-input-waiting` is reached by which the error `error-time-expired` is set. This time-out starts after the last beep has been sent.

2.4.2 Correspondence with the Recommendation (Q1203-2.12)

We introduce a data type `ANNOUNCEMENT` which contains several constants, each for a special type of message such as DTMF tones etc..

Furthermore, we have simplified the possible input of the user to strings. Hence, we do not have the parameters *Voice Feedback* and *Type* in our Service Support Data, as is the case in the Recommendation.

The SSD parameter `total-counter` which represents the repetition number in case `rep-type` is `count` is not given in the list of Service Support Data in the Recommendation.

The definition of User-interaction is rather complicated, there is also a great deal of redundancy in the Service Support Data. For example if `rep-type` equals `time` then the parameter `total-counter` is not used at all, and vice versa; either `total-counter` or `total-time` is not used. Is it really the case that the services in which this SIB is used require such a complicated behaviour, i.e., is it not possible to accept a slightly different behaviour which enables a much simpler definition?

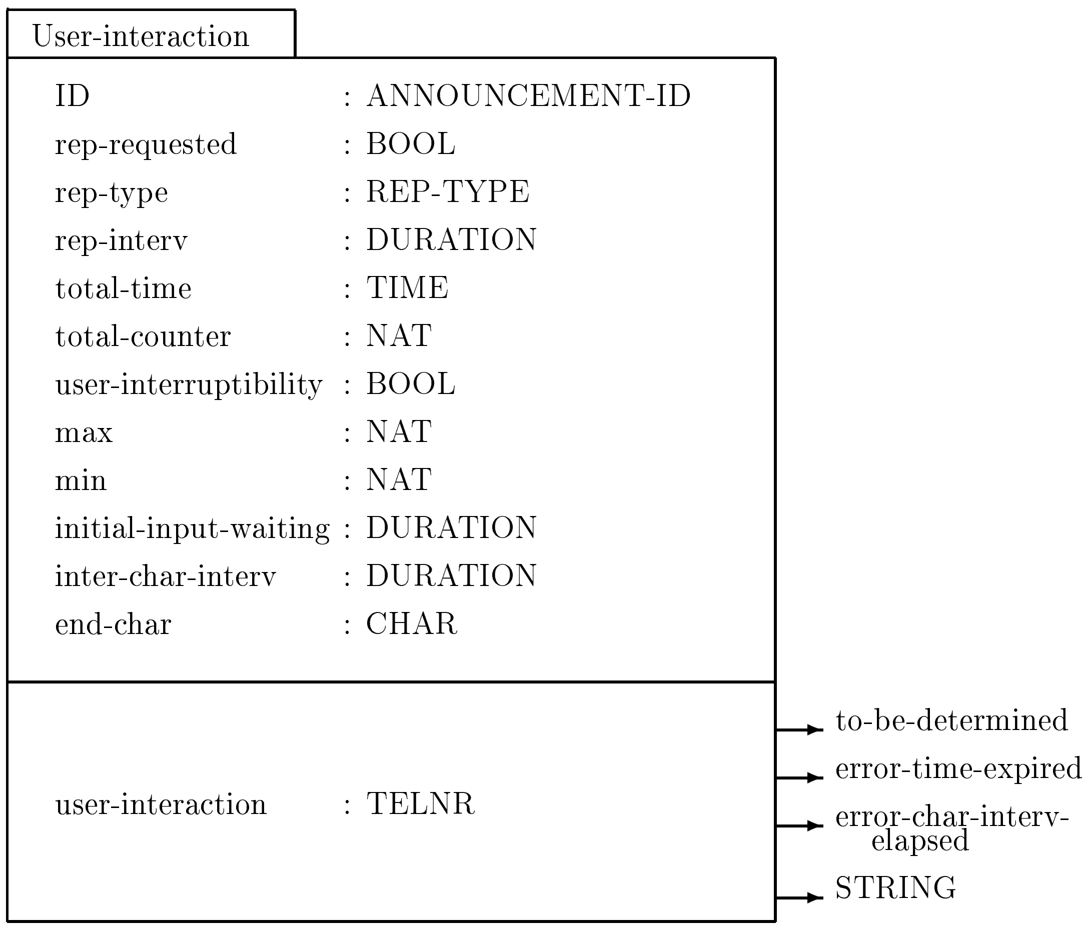

Figure 7: Interface Diagram of User-interaction

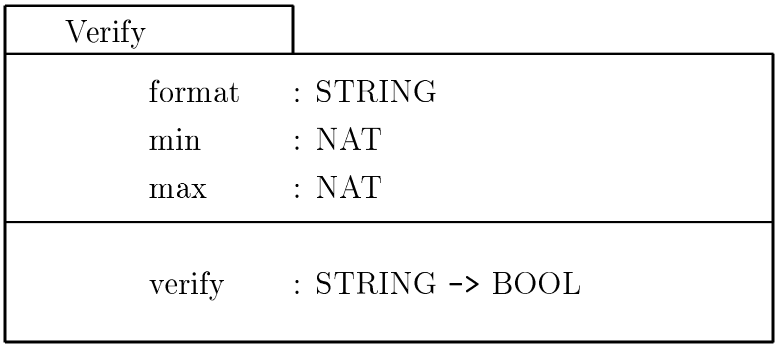

Figure 8: Interface Diagram of Verify

In the diagram of the construction of the Automatic Alternative Billing Service (Recommendation, Q.1219, page 45) it is suggested that the SIB User-interaction can be used as well to send only an announcement to the user without expecting any input. We could not find a reasonable instantiation of the process User-interaction. Therefore, we have introduced another process User-interaction-beep which sends one beep only.

From the Recommendation it is not clear whether User-interaction can be used in the Basic Call Process or whether the Basic Call Process has his own mechanism for exchanging information with the user. However, as in the case of Charge, we do not see any reason why the Basic Call Process should not use the SIB User-interaction for its communications with the user. (We don't use User-interaction in the Basic Call Process as yet.)

2.5 Verify

2.5.1 Its informal definition and interface

Verify checks whether a string matches a certain format. The format itself is a string as well, the following coding has been used:

x	Any character
L	Any letter (a-z,A-Z)
A	Upper case letters only (A-Z)
a	Lower case letters only (a-z)
D	Any digit (0-9) or delineator (#,*)
N	Any digit (0-9)
n	Any digit, except 0 (1-9)
[x]	Optional character, where x represents one of the above codes
\ s \	Character occurring in string s required. (e.g. \ 012 \ indicates that a 0, 1 or 2 must be present at the designated position.)

2.5.2 Correspondence with the Recommendation (Q.1213-2.13)

The above definition is equivalent with the one in the Recommendation.

3 The Basic Call Process

3.1 Point Of Initiation and Point Of Return

Control is transferred from the Basic Call Process to the Global Service Logic at a Point Of Initiation. Control is transferred back to the Basic Call Process at a Point Of Return. In Recommendation (Q1213-3.0) we find the following list of suggested Points Of Initiation and Points Of Return.

Points Of Initiation:

- Call Originated
- Address Collected
- Address Analyzed
- Prepared To Complete Call
- Busy
- No Answer
- Call Acceptance
- Active State
- End Of Call

Points Of Return:

- Continue With Existing Data
- Proceed With New Data
- Handle as Transit
- Clear Call
- Enable Call Party Handling
- Initiate Call

Not clear to us are Prepared To Complete Call, Handle As Transit, Enable Call Party Handling and Initiate Call; so we have not used them in the specification of the Basic Call Process, the Global Service Logic or the services. The other Points Of Initiation and Points of Return have a place in the Basic Call Process, the Global Service Logic or the services. But, as one will see in the specification, there is mostly no real follow-up by the Global Service Logic after control is transferred to it, i.e., no service is invoked and the Global Service Logic just transfers control back to the Basic Call Process. See Figure 9 for a pictorial representation of the Basic Call Process, the Global Service Logic and the Points Of Initiation and Points Of Return used.

In this figure the order from top to bottom suggest an ordering of the Points Of Initiation and Points Of Return. However, the Global Service Logic is able to handle the Points of Initiation in all possible orders; when it is ready to handle a Point of Initiation it can handle all Points of Initiation, independently of the previous ones.

The figure the dots (●) denote unsuccesfull termination of the Basic Call Process and the squares (■) denote succesfull termination.

3.2 Informal definition of the Basic Call Process

Every time the Basic Call Process has collected new data it passes control to the GSL at a point of initiation. The Global Service Logic checks the data (which is the call instance data) to see if a service is needed. If not, it returns control to the Basic Call Process at a Point of Return. If so, the Global Service Logic starts a service, i.e. chain of SIB's. After its completion the chain of SIB's returns control to the Basic Call Process at a Point of Return. A Basic Call Process is initiated when a Telephone is unhooked, the only call instance data is the so called calling line identification. In the processing of a call the following extra information can be gathered: dialled line identification, access code, account code, pin code and account. After every Point of Initiation control is returned to the Basic Call Process at one of the (two) Points of Return. This is however not obligatory, the Recommendation does not specify the number of Points Of Return, it just happens in our specification that there are two Points Of Return. One of the Points Of Return signifies that an error has occurred, the call will now be cleared. The other signifies that further processing of the call by the Basic Call Process is allowed. In the sequel the 'normal' processing of the Basic Call Process is discussed, i.e. no errors occur.

After its initiation the Basic Call Process passes control and the calling line identification to the Global Service Logic at the Call Originated Point Of Initiation. In the specification the Global Service Logic here immediately returns control, but one can imagine that the calling line identification alone already triggers a special service. Now the Basic Call Process expects input from the Telephone. This information consists of a dialled line identification or an access code followed by a dialled line identification. In both cases control is passed to the Global Service Logic. In the former case control is directly returned to the Basic Call Process, in the latter the Global Service Logic invokes the Automatic Alternative Billing service which on its turn returns control to the Basic Call Process. (In the specification in this paper there happens to be only one access code and it corresponds with Automatic Alternative Billing.) When control is returned to the Basic Call Process it will try to establish a connection between the caller and called. At this point there are three possibilities: the called Telephone is busy, the call is not answered or the call is answered. In the latter a conversation will follow, and hereafter an account is charged. This is the account assigned to account-code in case Automatic Alternative Billing is used, otherwise the account is the account assigned to the calling line identification.

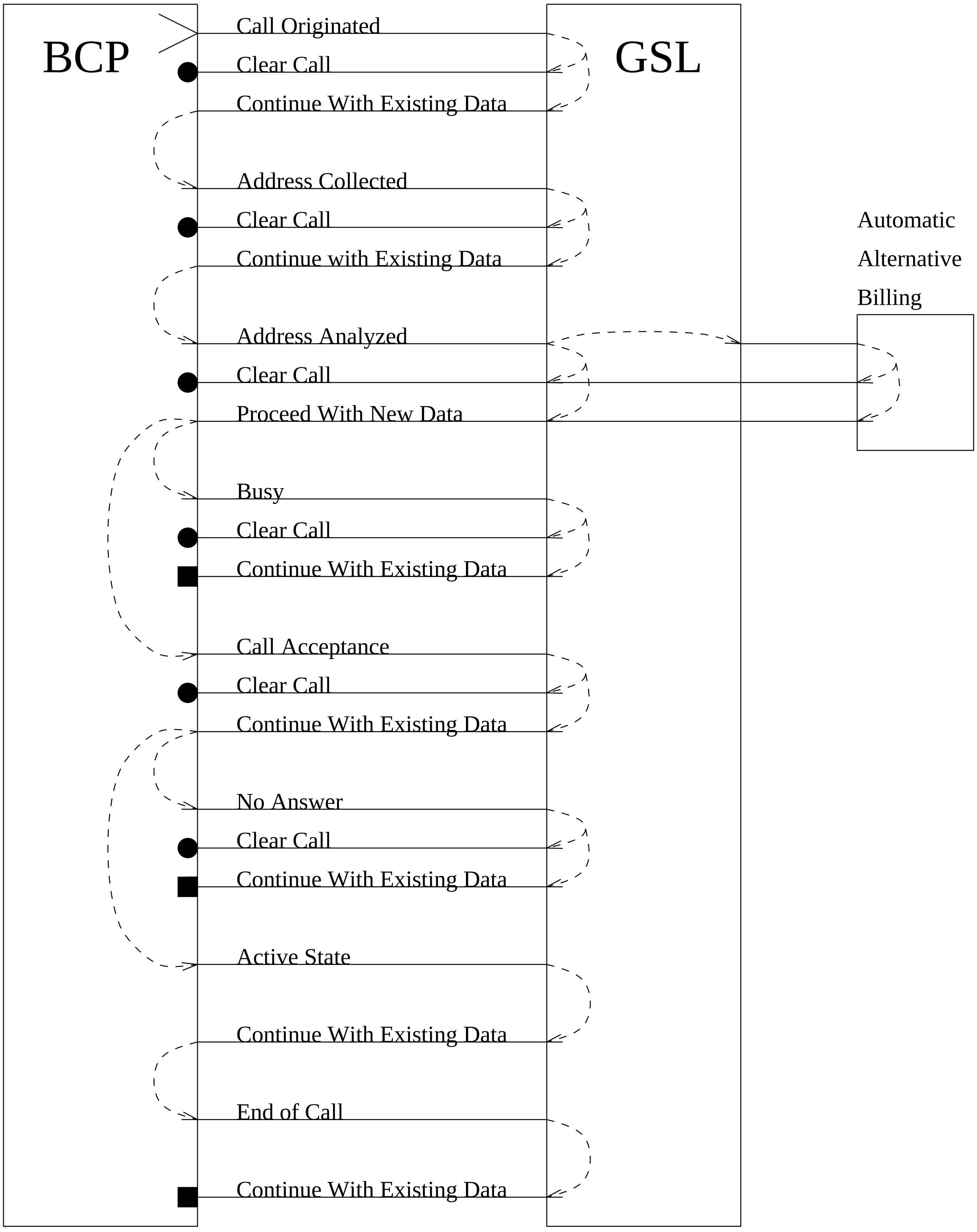

Figure 9: The Interaction among the Basic Call Process and Global Service Logic

3.3 Correspondence with the Recommendation (Q1203, Q1213)

In this paper the Basic Call Process is considered as the main controlling process in the Global Functional Plane, i.e. we do not consider the Basic Call Process as a SIB as is done in the Recommendation (Q1213). Considering the Basic Call Process as a SIB is, as we think, artificial. For instance, if the Basic Call Process is considered a SIB it should execute only as part of a service, but that is not the case: the Basic Call Process asks the Global Service Logic to start services.

The Service Support Data and Call Instance Data as defined in the Recommendation (Q1213-3.0) are not present in our specification of the Basic Call Process. Some Points of Initiation and some Points of Return are used, but are not regarded as Service Support Data of the Basic Call Process. The point of initiation Prepared To Complete Call is not used and so are the points of return Handle As Transit, Enable Call Party Handling and Initiate Call. It is not clear whether the Basic Call Process can use SIB's directly, instead of by means of the Global Service Logic. The Recommendation (Q1213-3.1) suggests that direct use of SIB's from the Basic Call Process is possible. The Basic Call Process specified in this paper does however use a SIB, i.e., Charge.

The Recommendation is not explicit in what way control is transferred from the Basic Call Process to the Global Service Logic. In our interpretation the Basic Call Process stops when it transfers control to the Global Service Logic at a Point Of Initiation and resumes processing after control is transferred to it at a Point Of Return. It might however be possible that they operate in parallel, or it might even be possible that part of a chain of SIB's or the Global Service Logic transfers control to the Basic Call Process and an other part executes in parallel with the Basic Call Process. This last possibility can be useful in the following. The Automatic Alternative Billing service reads a pin code and an account code. If the pin code and the account code are correct it charges the account associated with the read account code. Subsequently, the Basic Call Process tries to establish a connection. If the Basic Call Process does not succeed in establishing a connection the account is still charged for using the Alternative Automatic Billing service. If Charge was executing in parallel with the Basic Call Process it could check whether a connection is really established and charge the account subsequently.

4 The Global Service Logic

4.1 Its informal definition

The Global Service Logic can be said to be distributed among the module Global Service Logic and the modules specifying the services (only Automatic Alternative Billing as yet). The former is a server that selects and launches a service after the Basic Call Process has passed control. The structuring of the latter is what may be called the 'logic'. Two constructs are used to chain SIB's together. We have a form of sequential composition such that a SIB B can be put after a SIB A where the output of A is the input of B. Moreover we have a choice construct (*if-then-else*). See section 4.3 for more on this.

After a service has terminated it directly transfers control to the Basic Call Process, the Global Service Logic is not involved in this return of control. That is however in principle, in the specification of the Global Service Logic in this paper there is no service attached to all Points Of Initiation except for the Point Of Initiation called Address Analyzed. So, in the specification the Global Service Logic transfers control back to the Basic Call Process if there is no associated service.

4.2 Correspondence with the Recommendation (Q1203, Q1213)

The Recommendation is not explicit in its description of the Global Service Logic. In the Recommendation (Q1203-2.0) is stated: *The process of how the Global Service Logic is described through the Service Creation Environment using the Application Programming Interface is an area of further study.* However, the specification differs from the Recommendation in the sense that Global Service Logic launches services and not service features. The notion of service feature is not considered in this paper. Furthermore the Point Of Initiation with the name Prepared To Complete Call is not used and so are the Points Of Return with respectively the names Handle As Transit, Enable Call Party Handling and Initiate Call.

4.3 Remarks on Global Service Logic

In this section we show how the SIB's will be instantiated and chained together. First we give an example in Figure 10 of the data SIB Verify; its Service Support Data has been instantiated for verifying pin codes and the function which it offers, "verify", has been renamed to "verify-pin-code" to distinguish it from the functions which are offered by other instantiations. Next, we give an example of the process SIB User-interaction.

Verify	
format	= format-pin-code
min	= 4
max	= 4
verify-pin-code	STRING -> BOOL

Figure 10: The instantiations of Verify for verifying pin codes

If it ouputs a string, then the context has to read it in a variable, for example "pin-str". This is shown in Figure 11, where it is shown as well how a boolean function can be used in a choice construct. The error outputs have been omitted in this figure. In the next section we will give a more exhaustive example.

5 The Automatic Alternative Billing Service

First we instantiate the data modules, as given in Figure 12 and Figure 13. The data SIB Translate (see Section 2.3) is not shown here as it has no parameters to instantiate. In Figure 14, a pictorial representation is given of the Automatic Alternative Billing service constructed by chaining together the process SIB's User-interaction and Charge. The

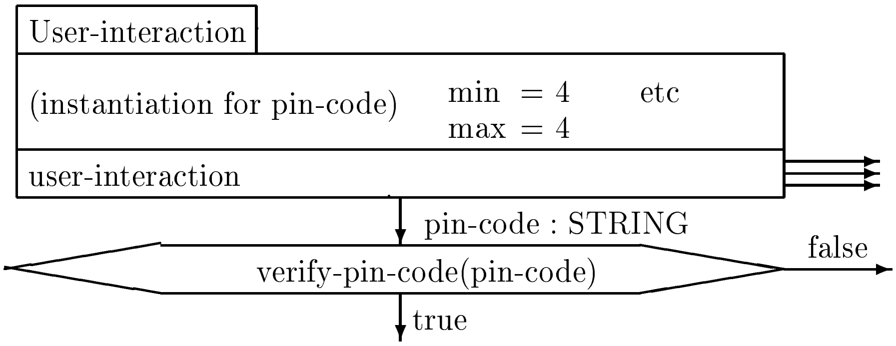

Figure 11: The instantiation of User-interaction and a choice construct

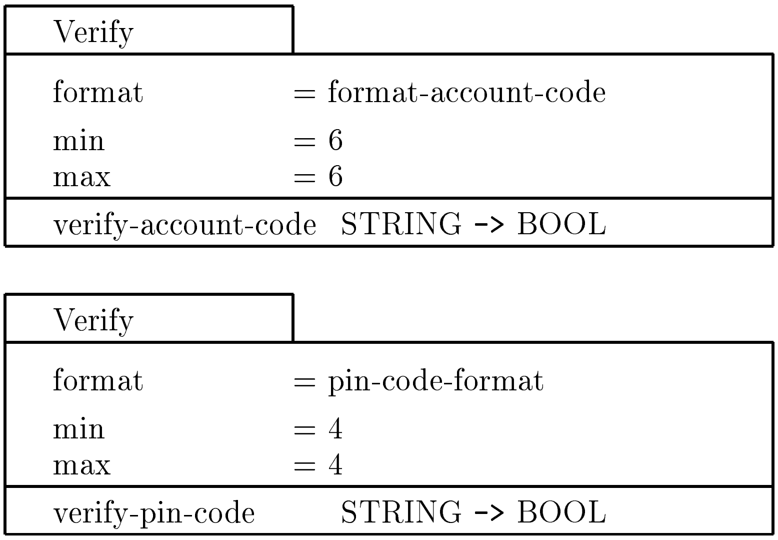

Figure 12: The instantiation of Verify for pin codes and account codes

functions offered by the instantiations of the data modules Verify and Screen occur in the choices and the data module Translate occurs in the argument of the charge-process.

Note that these figures contain much more information of the Automatic Alternative Billing service than the corresponding diagram in the Recommendation Q1219 (page 45). That figure does not give any information on the values of the parameters at all and it is not clear how they can be added easily.

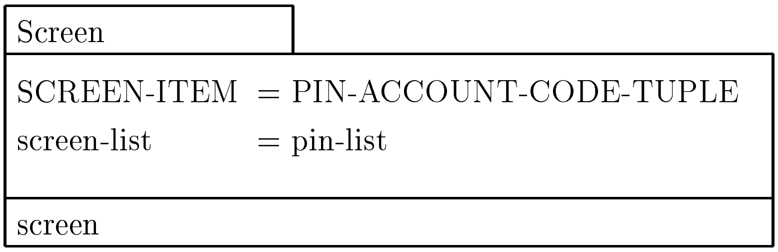

Figure 13: The instantiation of Screen

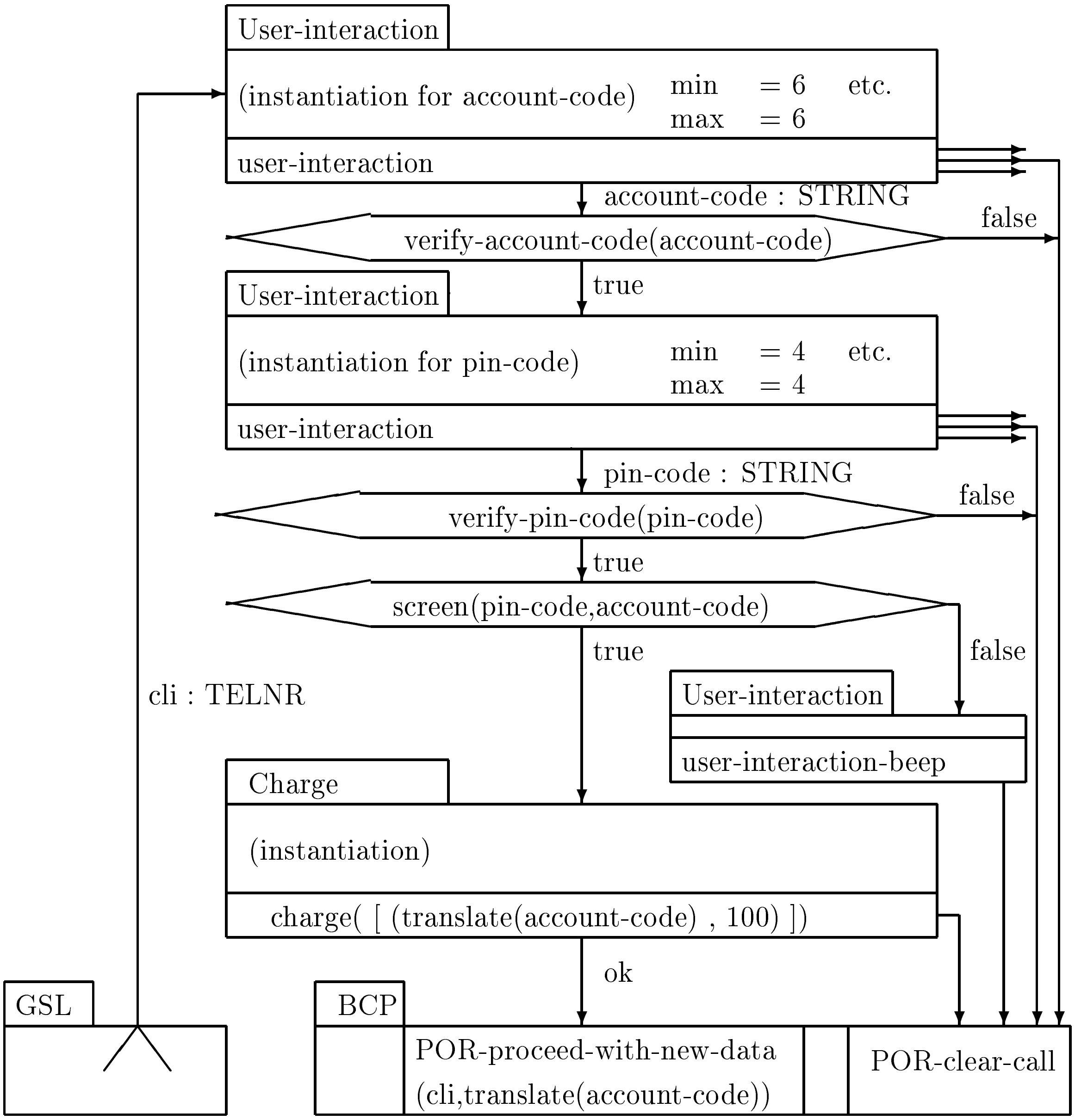

Figure 14: The AAB Chain

6 Conclusions

In this paper we have presented an initial version of a specification of the Global Functional Plane of the Conceptual Model for Intelligent Networks as formulated in the CCITT Recommendation [CCITT92]. We have given informal definitions, which are consistent with the formal specifications given in the appendix, of notions as Service Independent Building Blocks, Global Service Logic, and Basic Call Process. Furthermore we have considered the Service Plane as well by giving an example of the Automatic Alternative Billing service. We have encountered various points at which the Recommendation was unclear. Hence, our definitions deviate slightly from the Recommendation as is discussed as well in the paper.

Our long term goal is to obtain a specification which covers the complete Conceptual Model. This specification must have a formal version such that implementations can be proven correct and techniques like model checking are applicable for studying feature interaction. Moreover, if we use a language like PSF we can simulate the services by which expensive prototyping is prevented. Furthermore, this specification must have an informal version in English such that people can gradually get familiar with the Conceptual Model; the present description in the Recommendation contains a lot of technical details and some people complain that it is not easy readable.

However, an important prerequisite is the acceptance of this specification by the Intelligent Network community. Hence, we can succeed only if we interact with Intelligent Network experts; they can tell us whether our point of view is reasonable and how the present specification can be improved.

A lot of work still has to be done. First of all we want to study the "architecture" of the Conceptual Model in more detail. For example, in the Recommendation the Basic Call Process is considered as a special Service Independent Building Block. Moreover, it is supposed to be responsible for its own user interaction and charging while there are other Service Independent Building Blocks which are specially designed for that. But, on the other hand, Service Independent Building Blocks are supposed to be *orthogonal*, i.e. without overlap in their functionality. For us it is not clear yet whether the Basic Call Process must be considered as a Service Independent Building Block. Other questions are related to the status of the Basic Call Process and the Global Service Logic; for example whether they must indeed be considered as two seperate processes.

After we have gained insight in the architecture of the model we can continue our work by specifying all the Service Independent Building Blocks and the services as defined in the Capability Set 1. We can also go "downwards" in the Conceptual Model by taking the Distributed Functional Plane into account. We can do similar for the Physical Plane. The relation between the Global and the Distributed Functional Plane is not clear, since there are several points of view. One point of view is that each action, which is atomic with respect to the Global Functional Plane, is elaborated further on the Distributed Functional Plane. Another point of view is to implement the services on the Distributed Functional Plane by which for each service its implementation must be proven equal with its specification, i.e., its description in terms of Service Independent Building Blocks.

Another interesting question is whether feature interactions are dependent on the plane; some of them can be recognized already on the Global Functional Plane but it is possible as well that other feature interactions depend on implementation details at the

Distributed Functional Plane or even at the Physical Plane.

References

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press frontier series. Addison Wesley, 1989.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[BZ93] W. Bouma and H. Zuidweg. Formal analysis of feature interactions by model checking. Draft document, Dutch PTT Research, Leidschendam, The Netherlands, 1993.

[Cam92] M. Campolargo. Service engineering in RACE. Technical report, Commission of the European Communities, DGXIII/F - RACE Programme, 1992.

[dB90] N.C.J.M. de Beer. Intelligente netten. In *Handboek Informatica*, 1990. In dutch.

[ISO87] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.

[KD92] M. Kooij and M. Dauphin. Using formal methods in determining feature interaction. Internal document, Centre d'Etudes et Recherches, Compagnie IBM France, 06610 La Gaude, France, 1992.

[Mau91] S. Mauw. *PSF, A Process Specification Formalism.* PhD thesis, University of Amsterdam, Amsterdam, 1991.

[MV90] S. Mauw and G.J. Veltink. A Process Specification Formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In J.W. de Bakker at al., editor, *Proceedings of the REX Workshop "Real-Time :Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 526–548. Springer-Verlag, 1991.

[Sod] L. Soderberg. Architecture for intelligent networks, pages 13–22.

[CET92] CET BOOST Team. Service engineering - an intelligent networks point of view. BOOST deliverable BO/CE/0002/W/1.0/C, CET-Centro de Estudos de Telecomunicacoes, Rua Eng. Ferreira Pinto Basto, P-3800 Aveiro, Portugal, 1992.

[Wan91] Y. Wang. *A Calculus of Real Time Systems.* PhD thesis, Chalmers University of Technology, Göteborg, 1991.

[CCITT92] Study Group XI (WP XI/4). *New Recommendation Q.1200- Q Series Intelligent Network Recommendation Structure.* CCITT, Geneva, 10-17 march 1992.

A Some remarks on PSF

The appendices of this paper contain the formal specifications of the SIB's, the GSL, BCP and associated concepts. The language which is used here is the specification language PSF. For an introduction in PSF we refer to [MV90] and [Mau91].

The specification which is presented in this appendix can be simulated, i.e., it can be run by the PSF simulator tool. However, this tool requires that all sets which are used in the sum construct are finite. Therefore we define for every sort an associated finite set. For example, the finite set of natural numbers is denoted by `sNAT` and the finite set of characters is denoted by `sCHAR`.

In the specification we assume that internal actions will not be delayed. This will be discussed in more detail in Section H.3.

B The Specifications of the SIB's

B.1 The process module Charge

First we have to define the data type `AC-NAT-TUPLE` by instantiating the parameterised datatype `TUPLE` by taking `ACCOUNT` for `ITEM1` and `NAT` for `ITEM2`.

```
data module Ac-nat-tuple
begin
imports
Tuple
{ Item1 bound by [ITEM1 -> ACCOUNT         ,
                  eq    -> eq              ,
                  nil1  -> account0        ] to Account
  Item2 bound by [ITEM2 -> NAT             ,
                  eq    -> eq              ,
                  nil2  -> 0               ] to Nat
 renamed by      [TUPLE -> AC-NAT-TUPLE    ,
                  nil   -> nil-ac-nat-tuple]
}
end Ac-nat-tuple
```

Next we define the data type `AC-NAT-TUPLE-SEQ` by instantiating the parameterised data type `SEQ` with `AC-NAT-TUPLE`. Moreover, we define the function `totalsum` which computes the sum of all natural numbers of all the pairs in the sequence.

```
data module Ac-nat-tuple-seq
```

```
begin
exports
begin
functions
  totalsum : AC-NAT-TUPLE-SEQ -> NAT
end
imports
Seq
{ Item bound by [ITEM  -> AC-NAT-TUPLE          ,
                 eq    -> eq                    ,
                 nil   -> nil-ac-nat-tuple      ] to Ac-nat-tuple
 renamed by     [SEQ   -> AC-NAT-TUPLE-SEQ      ,
                 empty -> empty-ac-nat-tuple-seq ]
}
variables
  s :-> AC-NAT-TUPLE-SEQ
  a :-> ACCOUNT
  n :-> NAT
equations
[TS1] totalsum(empty-ac-nat-tuple-seq) = 0
[TS2] totalsum(add( tuple(a,n), s))    = n + totalsum(s)
end Ac-nat-tuple-seq
```

Now we are ready to give the definition of `Charge`. The process `charge(s)` (where `s :-> AC-NAT-TUPLE-SEQ`) updates, for each tuple in the sequence `s`, the database.

```
process module Charge
begin
  parameters
    SSD begin
      functions
        resource :-> RESOURCE
        service  :-> SERVICE
    end  SSD
  exports
    begin
      processes
        charge : JOBNR # NAT# AC-NAT-TUPLE-SEQ
    end
  imports
    Ac-nat-tuple-seq, Resource, Service, Nat,
    Communication, Dbase-atoms, Jobnr
  processes
    send-to-dbase : NAT # AC-NAT-TUPLE-SEQ
  variables
```

```
    s   :-> AC-NAT-TUPLE-SEQ
    n,p :-> NAT
    a   :-> ACCOUNT
    j   :-> JOBNR

definitions

charge(j,n,s) =
( [ eq(totalsum(s),100) = true ]
   -> send-to-dbase(n,s) . send-chain(j,ok)
+ [ eq(totalsum(s),100) = false ]
   -> send-chain(j,error-sum-of-percentages-is-not-100)
)

send-to-dbase(n,empty-ac-nat-tuple-seq) = skip

send-to-dbase(n,add( tuple(a,p), s)) =
  send-dbase(update)
. send-dbase(a)
. send-dbase(resource)
. send-dbase(service)
. send-dbase(div( times(p,n), 100))
. send-to-dbase(n,s)
end Charge
```

B.2 The data module Screen

The Service Support Data section of the Screen is quite complicated, see Figure 5 for the interface diagram. Screen is parameterised by the data type `SCREEN-ITEM` while also the data type `SCREEN-ITEM-SEQ` occurs in Service Support Data. This means that the context which instantiates `SCREEN-ITEM` with a data type, say `S`, is also responsible for the construction of sequences over `S`. Therefore, we have two data types in the SSD parameter section of the data module `Screen`; `SCREEN-ITEM` and `SCREEN-ITEM-SEQ`. Moreover, the functions `screen-list` and the `is-in` must occur in this parameter section.

In fact we do not require formally that the sort `SCREEN-ITEM-SEQ` and the function `is-in` are what they are supposed to be; this is not expressible in PSF. Therefore, we have to be careful when we instantiate this module; we can for example instantiate both sorts arbitrarily and we can instantiate the function `is-in` by a function which delivers always `true` (or always `false`). In that case we have obtained an instantiation which has nothing to do any more with the informal definition of Screen.

```
data module Screen
begin
  parameters
```

```
    SSD begin
      sorts
        SCREEN-ITEM,
        SCREEN-ITEM-SEQ
      functions
        screen-list : -> SCREEN-ITEM-SEQ
        is-in       : SCREEN-ITEM # SCREEN-ITEM-SEQ -> BOOL
    end SSD
  exports
    begin
      functions
        screen      : SCREEN-ITEM                   -> BOOL
    end
  imports   Bool
  variables i:-> SCREEN-ITEM
  equations
    [S] screen(i) = is-in(i,screen-list)
end Screen
```

B.3 The process module User-interaction

We define two auxiliary processes, one which sends the announcement ID over a certain period, for in case rep-type is time (send-announcement-time), and one which sends it a fixed amount (send-announcement-count). Moreover, we define a process user-input(cli,j) (where cli is a TELNR and j a JOBNR) which is always ready to treat the input from the user. As soon as the user has given his first character c this process evolves in user-input(cli,j,empty-string,c,t). Here, t is the time at which the character c is read, we must keep track of the time since it must be guaranteed that there is a maximal idle time between the input of two characters. The process user-input(cli,j,s,c,t) (where s is a STRING, which is a sequence of characters), checks whether c is not equal to nil and whether it is ready with input. If so, then either the string is sent locally within the process User-interaction, or an error is detected. If not, then the next character is read and the process is called recursively.

```
process module User-interaction
begin
  parameters
    SSD begin
      functions
        ID                      :-> ANNOUNCEMENT
        rep-requested           :-> BOOL
        rep-type                :-> REP-TYPE
        rep-interv              :-> DURATION
        total-time              :-> TIME
```

```
      total-counter         :-> NAT
      user-interruptibility :-> BOOL
      max                   :-> NAT
      min                   :-> NAT
      initial-input-waiting :-> DURATION
      inter-char-interv     :-> DURATION
      end-char              :-> CHAR
  end SSD
exports
  begin
    processes
      user-interaction      : TELNR # JOBNR
      user-interaction-beep : TELNR
  end
imports
  Bool,Nat,Time,Announcement,Rep-type,String,Message,
  Communication, Clock, Basic-sort, Jobnr, Telnr
atoms
  send-local,read-local,comm-local : JOBNR # STRING
  send-local,read-local,comm-local : JOBNR # MESSAGE
processes
  user-input              : TELNR # JOBNR
  user-input              : TELNR # JOBNR # STRING # CHAR # TIME
  send-announcement-time  : TELNR # JOBNR # TIME
  send-announcement-count : TELNR # JOBNR # TIME # NAT
  output                  : JOBNR
sets of atoms
  H-local-s = { send-local(j,s),read-local(j,s)
              | j in JOBNR, s in STRING  }
  H-local-e = { send-local(j,m),read-local(j,m)
              | j in JOBNR, m in MESSAGE }
  I-local-s = { comm-local(j,s)
              | j in JOBNR, s in STRING  }
  I-local-e = { comm-local(j,m)
              | j in JOBNR, m in MESSAGE }
communications
  send-local(j,s) | read-local(j,s)
   = comm-local(j,s) for j in JOBNR, s in STRING
  send-local(j,m) | read-local(j,m)
   = comm-local(j,m) for j in JOBNR, m in MESSAGE
variables
  n   :-> NAT
  t   :-> TIME
  c   :-> CHAR
  s   :-> STRING
  j   :-> JOBNR
```

```
    cli :-> TELNR

definitions

user-input(cli,j) =
sum(c in sCHAR,
    read-user(cli,c) .
    sum(t' in sTIME, read-time(t') .
        user-input(cli,j,empty-string,c,t')
       )
   )

user-input(cli,j,s,c,t) =
( [ true  =( eq(c,nil) .or. gt(min,max) .or.
             eq(min,0) .or. eq(max,0)      )
  ]  -> send-local(j,to-be-determined)
+ [ false =( eq(c,nil) .or. gt(min,max) .or.
             eq(min,0) .or. eq(max,0)      )
  ] ->
   ( [ true = eq(c,end-char) ]
      ->
      ( [ true = lte(min,len(s)) ]
         -> send-local(j,s)
      + [ false= lte(min,len(s)) ]
         -> send-local(j,to-be-determined)
      )
   + [ false= eq(c,end-char) ]
      ->
      ( [ true =  lt(len(s),max) ]
         ->
          ( [ true = eq(len(s),max - 1) .and. eq(end-char,nil) ]
             -> send-local(j,add(c,s))
          + [ false= eq(len(s),max - 1) .and. eq(end-char,nil) ]
             ->
             ( read-time(t + inter-char-interv)
                 . send-local(j,error-char-interv-elapsed)
             + sum(c' in sCHAR, read-user(cli,c')
                 . sum(t' in sTIME, read-time(t')
                 . user-input(cli,j,add(c,s),c', t' )
                       )
                   )
             )
          )
      + [ false=  lt(len(s),max) ]
         -> send-local(j,to-be-determined)
      )
```

```
   )
)

send-announcement-time(cli,j,t) =
send-user(cli,ID) .
( [ true = gt(total-time,t + rep-interv) .and. rep-requested ]
   ->
   ( read-time(t + rep-interv)
      . send-announcement-time(cli,j,t + rep-interv)
   + [ user-interruptibility = true ]
      -> user-input(cli,j)
   )
+ [ false = gt(total-time,t + rep-interv) .and. rep-requested ]
   -> send-local(j,ready-without-input)
)

send-announcement-count(cli,j,t,n) =
send-user(cli,ID) .
([ true = gt(total-counter,n + 1) .and. rep-requested ]
  ->
   ( read-time(t + rep-interv)
      . send-announcement-count(cli,j,t + rep-interv,n + 1)
   + [ user-interruptibility = true ]
      -> user-input(cli,j)
   )
+ [ false = gt(total-counter,n + 1) .and. rep-requested ]
   -> send-local(j,ready-without-input)
)

user-interaction(cli,j) =
hide(I-local-s, hide(I-local-e, encaps(H-local-s, encaps(H-local-e,
( sum(t' in sTIME, read-time(t') .
      ( [rep-type=rep-time]  -> send-announcement-time(cli,j,t')
      + [rep-type=rep-count] -> send-announcement-count(cli,j,t',0)
      )
     )
  ||
  ( read-local(j,error-char-interv-elapsed)
     . send-chain(j,error-char-interv-elapsed)
  + read-local(j,to-be-determined)
     . send-chain(j,to-be-determined)
  + read-local(j,ready-without-input)
     . sum(t in sTIME, read-time(t)
            . ( ( user-input(cli,j)
                || sum(s in sSTRING, read-local(j,s)
                               . send-chain(j,s)
```

```
                        )
                    )
                  + read-time(t + initial-input-waiting)
                     . send-chain(j,error-time-expired)
                  )
              )
  + sum(s in sSTRING, read-local(j,s) . send-chain(j,s))
  )
)
))))

user-interaction-beep(cli) = send-user(cli,ID)
end User-interaction
```

B.4 The data module Verify

The SSD parameter section of the data module `Verify` consists of the constants `format`, `min` and `max`.

First we check whether the constant `format` is indeed a format. Therefore we have two local boolean functions `is-format`, one with a string as argument and one with a character as argument.

If it is indeed a format then we apply the function `match(s,format)`. The application `match(s,f)` checks whether the string `s` matches the format `f` by comparing repeatedly the first character of `s` with that of `f`.

A little complication is caused by a format like \012\NNN. As soon as the first backslash is encountered the rest of the sequence is split into two parts; before the next backslash and after that backslash. For example, as soon as the first backslash of \012\NNN is reached, the rest of the string is split into the strings 012 and NNN. A string `add(c,s)` matches the format \012\NNN if `c` occurs in the string 012 and `s` matches the format NNN.

Hence, we need the data type of tuples of strings (`STRING-STRING-TUPLE`) and a function `break : STRING -> STRING-STRING-TUPLE`.

```
data module Verify
begin
  parameters
    SSD begin
      functions
        format          :  -> STRING
        min             :  -> NAT
        max             :  -> NAT
    end SSD
  exports
    begin
```

```
      functions
        verify          : STRING  -> BOOL
    end
  imports
    Char, String, Bool,
    Tuple { Item1 bound by [ITEM1 -> STRING       ,
                            eq    -> eq           ,
                            nil1  -> empty-string ] to String
            Item2 bound by [ITEM2 -> STRING       ,
                            eq    -> eq           ,
                            nil2  -> empty-string ] to String
            renamed by     [TUPLE -> STRING-STRING-TUPLE      ,
                             nil  -> nil-string-string-tuple ]
          }
  functions
    match        : STRING # STRING -> BOOL
    is-format    : STRING          -> BOOL
    is-format    : CHAR            -> BOOL
    break        : STRING          -> STRING-STRING-TUPLE
    add-to-first : CHAR # STRING-STRING-TUPLE
                                   -> STRING-STRING-TUPLE
  variables
    f,s          :                 -> STRING
    z,z1,z2,z3,c :                 -> CHAR
    t            :                 -> STRING-STRING-TUPLE
  equations
[M0]  match(empty-string,empty-string)= true
[M1]  match(empty-string,add(c,s))    = false
[M2]  match(add(z,f),empty-string)    = false

[M3]  match(add('x',f),add(c,s))
      = match(f,s)
[M4]  match(add('L',f),s)
      = match(add('A',f),s) .or. match(add('a',f),s)
[M5]  match(add('A',f),add(c,s))
      = lte(ord('A'),ord(c)) .and. (lte(ord(c),ord('Z'))
                             .and. match(f,s))
[M6]  match(add('a',f),add(c,s))
      = lte(ord('a'),ord(c)) .and. (lte(ord(c),ord('z'))
                             .and. match(f,s))
[M7]  match(add('N',f),add(c,s))
      = lte(ord('0'),ord(c)) .and. (lte(ord(c),ord('9'))
                             .and. match(f,s))
[M8]  match(add('D',f),add(c,s))
      = match(add('N',f),add(c,s)) .or.
         ((eq(c,fence) .or. eq(c,star)) .and. match(f,s))
```

```
[M9]  match(add('n',f),add(c,s))
      = lte(ord('1'),ord(c)) .and. lte(ord(c),ord('9'))
                             .and. match(f,s)
[M10] match(add(sqbracket-open,add(z,add(sqbracket-close,f))),s)
      = match(add(z,f),s) .or. match(f,s)
[M11] match(add(backslash,f),add(c,s))
      = is-in(c,proj1(break(f))) .and. match(proj2(break(f)),s)

[M12] is-format(f) = false ==> match(f,s) = false

[B0]  break(empty-string) = nil-string-string-tuple
[B1]  eq(c,backslash) = true  ==>
         break(add(c,s)) = tuple(empty-string,s)
[B2]  eq(c,backslash) = false ==>
         break(add(c,s)) = add-to-first(c,break(s))

[A1]  add-to-first(c,t)  = tuple(add(c,proj1(t)),proj2(t))

[F0]  is-format(z) =
      ( eq(z,'x') .or. eq(z,'L') .or. eq(z,'A') .or. eq(z,'a')
                  .or. eq(z,'D') .or. eq(z,'N') .or. eq(z,'n')
      )

[F1]  is-format(empty-string)
      = true
[F2]  is-format(add(z1,empty-string))
      = is-format(z1)
[F3]  is-format(add(z1,add(z2,empty-string)))
      = is-format(z1) .and. is-format(z2)
[F2]  is-format(add(z1,add(z2,add(z3,f))))
      =    ( is-format(z1)  .and. is-format(add(z2,add(z3,f)))  )
      .or. ( eq(z1,sqbracket-open)  .and. is-format(z2)  .and.
             eq(z3,sqbracket-close) .and. is-format(f)          )
      .or. ( eq(z1,backslash) .and.
                  is-format(proj2(break(add(z2,add(z3,f)))))    )

[V]   verify(s)
      = gte(len(s),min) .and. lte(len(s),max)
                        .and. match(format,s)
end Verify
```

C The process module Gsl

The service Automatic Alternative Billing is imported in `Gsl` and bound to specific SSD parameters. Other services should also get imported and bound to SSD parameters in a similar way.

The process module `Gsl` is specified as a server for services, i.e., it comes to action when it reads a Point Of Initiation and then invokes the appropriate service. This is the idea we try to put forward. However, in the specification below one will find only one service that is invokable: the Automatic Alternative Billing being invoked in response to reading Point Of Initiation Address Analyzed (`read-poi-address-analyzed(....,aab)`). The other 7 Points Of Initiation have no follow-up with a service as yet, so the process `Gsl` just returns control with either `send-por-clear-call(....)` or `send-por-continue-with-existing-data(....)`. The former signalling that for one reason or the other the call should be cleared by the Basic Call Process. The latter signalling that the Basic Call Process should continue normal handling of the call.

```
process module Gsl
begin
  exports
    begin
      processes
        GSL
    end
  imports
    Poi-and-por-communication, Telnr,
    AAB { BCP-INFO bound by [used-resource -> resource1,
                             used-service -> service1,
                             used-amount -> 1 ] to AAB-data }
definitions

GSL =
sum(j in sJOBNR, sum(cli in sTELNR,
       read-poi-call-originated(j, cli) .
    (( send-por-clear-call(j,to-be-determined)
       +
       send-por-continue-with-existing-data(j, cli)
       -- or other not implemented por's --
     )
    ||
    GSL
    )
   ))
+
sum(j in sJOBNR, sum(cli in sTELNR, sum(dli in sTELNR,
    read-poi-address-collected(j, cli, dli) .
    ((send-por-clear-call(j,to-be-determined)
```

```
      +
      send-por-continue-with-existing-data(j, cli, dli)
      -- or other not implemented por's --
     )
     ||
     GSL
    )
   )))
+
sum(j in sJOBNR, sum(cli in sTELNR, sum(dli in sTELNR,
    read-poi-address-collected(j, cli, dli, aab) .
    ((send-por-clear-call(j,to-be-determined)
      +
      send-por-continue-with-existing-data(j, cli, dli, aab)
      -- or other not implemented por's --
     )
     ||
     GSL
    )
   )))
+
sum(j in sJOBNR, sum(cli in sTELNR, sum(dli in sTELNR,
    read-poi-address-analyzed(j, cli, dli, aab) .
    (AAB(j, cli, dli)
     ||
     GSL
    )
   )))
+
sum(j in sJOBNR, sum(cli in sTELNR, sum(dli in sTELNR,
    read-poi-address-analyzed(j, cli, dli) .
    ((send-por-clear-call(j,to-be-determined)
      +
      send-por-continue-with-existing-data(j, cli, dli)
      -- or other not implemented por's --
     )
     ||
     GSL
    )
   )))
+
sum(j in sJOBNR, sum(cli in sTELNR,
sum(dli in sTELNR, sum(account in sACCOUNT,
    read-poi-busy(j, cli, dli,account) .
    ((send-por-clear-call(j,to-be-determined)
      +
```

```
      send-por-continue-with-existing-data(j,cli,dli,account)
      -- or other not implemented por's --
     )
     ||
     GSL
    )
   ))))
+
sum(j in sJOBNR, sum(cli in sTELNR,
sum(dli in sTELNR, sum(account in sACCOUNT,
    read-poi-call-acceptance(j, cli, dli,account) .
    ((send-por-clear-call(j,to-be-determined)
      +
      send-por-continue-with-existing-data(j,cli,dli,account)
      -- or other not implemented por's --
     )
     ||
     GSL
    )
   ))))
+
sum(j in sJOBNR, sum(cli in sTELNR,
sum(dli in sTELNR, sum(account in sACCOUNT,
    read-poi-no-answer(j, cli, dli,account) .
    ((send-por-clear-call(j,to-be-determined)
      +
      send-por-continue-with-existing-data(j,cli,dli,account)
      -- or other not implemented por's --
     )
     ||
     GSL
    )
   ))))
+
sum(j in sJOBNR, sum(cli in sTELNR,
sum(dli in sTELNR, sum(account in sACCOUNT,
    read-poi-active-state(j, cli, dli,account) .
    (send-por-continue-with-existing-data(j,cli,dli,account)
     -- or other not implemented por's --
     ||
     GSL
    )
   ))))
+
sum(j in sJOBNR, sum(cli in sTELNR,
sum(dli in sTELNR, sum(account in sACCOUNT,
```

```
    read-poi-end-call(j, cli, dli,account) .
    (send-por-continue-with-existing-data(j,cli,dli,account)
     -- or other not implemented por's --
     ||
     GSL
    )
   ))))
end Gsl
```

D The process module Bcp

The Basic Call Process is specified with process module `Bcp`. `Bcp` has only one SSD parameter. (This parameter is not found in the Recommendation.) Notice that `Bcp` imports `Charge`, `Charge` is used to charge the account for the call charge.

Now we will discuss the structure of process `BCP`. `BCP` is actually specified with a number of smaller processes: `bcp`, `bcp-what-service`, `bcp-try-connect`, `bcp-ring`, `bcp-end-of-call`, `bcp-charge` and `bcp-error`. These processes, roughly put, do some action in the handling of the call, invoke the Global Service Logic, wait for return of control and then invoke an other process to handle the rest of the call. So `bcp` calls `bcp-what-service`, `bcp-what-service` on its turn calls `bcp-try-connect` etc., that is, when no errors occur. When an error occurs this error is handled by a process `bcp-error` that clears the call. The process `bcp-what-service` does most of the interaction with the user. It reads the telephone number of the called party and reads the access codes that is used by the Global Service Logic to select for instance a service as Automatic Alternative Billing.

```
process module Bcp
begin
  parameters
    SSD begin
      functions
        max-ring-repeat : -> NAT
    end SSD
  exports
    begin
      atoms
        handle-bcp-error : JOBNR # MESSAGE
      processes
        BCP      : TELNR
    end
  imports
    Nat, Poi-and-por-communication, Telnr, Account, Communication,
    Translate, Message, Jobnr-management,
```

```
    Charge {SSD bound by [resource -> resource0,
                          service  -> service0 ]
                to Bcp-data
           }
  processes
    bcp               : JOBNR # TELNR
    bcp-what-service : JOBNR # TELNR
    bcp-try-connect  : JOBNR # TELNR # TELNR # ACCOUNT
    bcp-ring          : JOBNR # TELNR # TELNR # ACCOUNT # NAT
    bcp-end-call      : JOBNR # TELNR # TELNR # ACCOUNT
    bcp-charge        : JOBNR # TELNR # TELNR # ACCOUNT
    bcp-error         : JOBNR # MESSAGE
    bcp-error         : TELNR # MESSAGE
  variables
    cli, dli   : -> TELNR
    j          : -> JOBNR
    account    : -> ACCOUNT
    err        : -> MESSAGE
    rest-rings : -> NAT

definitions

BCP(cli) =
sum(j in sJOBNR,
       read-jobnr(j) . bcp(j, cli) . send-back-jobnr(j)
   )

bcp(j, cli) =
send-poi-call-originated(j, cli) .
(read-por-clear-call(j,to-be-determined) .
 bcp-error(cli,to-be-determined)
 +
 read-por-continue-with-existing-data(j, cli) .
 bcp-what-service(j, cli)
 -- or other not implemented por's --
)

bcp-what-service(j, cli) =
sum(dli in sTELNR,
    read-number(cli, dli) .
    send-poi-address-collected(j, cli, dli) .
    (read-por-clear-call(j,to-be-determined) .
     bcp-error(cli,to-be-determined)
     +
     read-por-continue-with-existing-data(j, cli, dli) .
     (send-poi-address-analyzed(j, cli, dli) .
```

```
      (read-por-clear-call(j,to-be-determined) .
       bcp-error(cli,to-be-determined)
      +
       read-por-continue-with-existing-data(j, cli, dli) .
       bcp-try-connect(j,cli,dli,translate(cli))
       -- or other not implemented por's --
      )
     )
     -- or other not implemented por's --
    )
   )
+
sum(ac in sACCESS-CODE,
    read-access-code(cli, ac) .
    sum(dli in sTELNR,
        read-number(cli, dli) .
         send-poi-address-collected(j, cli, dli, ac) .
         (read-por-clear-call(j, to-be-determined) .
          bcp-error(cli,to-be-determined)
          +
          read-por-continue-with-existing-data(j, cli, dli, ac) .
          send-poi-address-analyzed(j, cli, dli, ac) .
          (sum(m in sMESSAGE, read-por-clear-call(j,m)) .
           bcp-error(cli, to-be-determined)
           +
           sum(account in sACCOUNT,
               read-por-proceed-with-new-data(j,cli, dli,account) .
               bcp-try-connect(j, cli, dli, account)
              )
           -- or other not implemented por's --
          )
          -- or other not implemented por's --
         )
       )
   )

bcp-try-connect(j, cli, dli, account) =
read-busy(dli) .
 send-poi-busy(j,cli,dli,account) .
 (read-por-clear-call(j,to-be-determined) .
  bcp-error(cli,to-be-determined)
 +
  read-por-continue-with-existing-data(j, cli, dli, account) .
  send-user(cli,busy-tone)
  -- or other not implemented por's --
 )
```

```
+
send-connect(cli, dli) .
 send-poi-call-acceptance(j,cli,dli,account) .
 (read-por-clear-call(j,to-be-determined) .
  bcp-error(cli,to-be-determined)
 +
  read-por-continue-with-existing-data(j, cli, dli, account) .
  bcp-ring(j, cli, dli, account,max-ring-repeat)
  -- or other not implemented por's --
 )

bcp-ring(j, cli, dli, account,0) =
send-poi-no-answer(j,cli,dli,account) .
( read-por-clear-call(j,to-be-determined)
+ read-por-continue-with-existing-data(j,cli,dli,account)
  -- or other not implemented por's --
) .  send-quit-ring(cli,dli)

bcp-ring(j, cli, dli, account,suc(rest-rings)) =
send-ring(cli, dli) .
((read-is-answered(cli, dli) .
  send-poi-active-state(j,cli,dli,account) .
  read-por-continue-with-existing-data(j,cli,dli,account)
  -- or other not implemented por's --
 || read-hook(cli)
 || read-hook(dli)
 ) .  bcp-end-call(j,cli,dli,account)
+ bcp-ring(j, cli, dli, account,rest-rings)
)

bcp-end-call(j,cli,dli,account) =
send-poi-end-call(j,cli,dli,account) .
read-por-continue-with-existing-data(j,cli,dli,account) .
-- or other not implemented por's --
bcp-charge(j,cli,dli,account)

bcp-charge(j,cli,dli,account) =
charge(j,1, add(tuple(account,100), empty-ac-nat-tuple-seq))
||
(read-chain(j,error-sum-of-percentages-is-not-100) .
 bcp-error(j,error-sum-of-percentages-is-not-100)
 +
 read-chain(j,ok)
)

bcp-error(j,err) =
```

```
handle-bcp-error(j,err)

bcp-error(cli,err) =
send-user(cli,error-beep)
end Bcp
```

E The process module Bcp-server

The process `BCP-SERVER` is a process that brings Basic Call Processes in existence when a telephone is unhooked. Furthermore is binds the parameter `max-ring-repeat` of module `Bcp`.

```
process module Bcp-server
begin
  exports
    begin
      processes
        BCP-SERVER
    end
  imports
    Bcp { SSD bound by [max-ring-repeat -> 4] to Nat}, Communication
  definitions
    BCP-SERVER = sum(cli in sTELNR,
                     read-unhook(cli) . (BCP(cli) || BCP-SERVER))
end Bcp-server
```

F The process module Intelligent-Network

The process module Intelligent-Network specifies the process IN that just puts the processes `GSL`, `TELEPHONES`, `BCP-SERVER`, `JOB`, `CLOCK` and `DBASE` in parallel. Furthermore the actions in H are encapsulated and the actions in I is abstracted from.

```
process module Intelligent-Network
begin
  exports
    begin
      atoms terminate
      processes
        IN
        Test test user-input
    end
```

```
imports
  Telephones, Gsl, Bcp-server,
  Clock, Dbase, Poi-and-por-communication, Communication
sets
  of atoms
    I = I-poi-por      +
        I-user-char    +
        I-user-id      +
        I-string       +
        I-error        +
        I-account-code +
        I-busy         +
        I-stop         +
        I-connect      +
        I-ring         +
        I-number       +
        I-is-answered  +
        I-jobnr        +
        I-back-jobnr   +
        I-dbase-a      +
        I-dbase-r      +
        I-dbase-s      +
        I-dbase-n      +
        I-dbase-d      +
        I-clock        +
        I-quit-ring
    H = H-poi-por      +
        H-user-char    +
        H-user-id      +
        H-string       +
        H-error        +
        H-access-code  +
        H-account-code +
        H-unhook       +
        H-hook         +
        H-busy         +
        H-stop         +
        H-connect      +
        H-ring         +
        H-number       +
        H-is-answered  +
        H-talk         +
        H-back-jobnr   +
        H-jobnr        +
        H-dbase-a      +
        H-dbase-r      +
```

```
          H-dbase-s      +
          H-dbase-n      +
          H-dbase-d      +
          H-clock        +
          H-quit-ring
  definitions
    IN = hide(I, encaps(H,
         GSL || TELEPHONES || BCP-SERVER || JOB || CLOCK || DBASE))
end Intelligent-Network
```

G The process module AAB

First we define the auxiliary data module `AAB-data` in which all constants are defined.

```
data module AAB-data
begin
  exports
    begin
      functions
        duration5           : -> DURATION
        NNNNc               : -> STRING
        format-account-code : -> STRING
        format-pin          : -> STRING
        pin-list            : -> PIN-ACCOUNT-CODE-TUPLE-SEQ
    end
  imports
    Bool,Char,Nat,String,
    Account, Announcement, Rep-type, Resource,Service,
    Time, String-seq, Pin-account-code-tuple,
    Pin-account-code-tuple-seq
  variables
    c :-> CHAR
    s :-> STRING

equations
[D5]  duration5 = duration(5)
[N]   NNNNc     =
       add(backslash,add('c',add(backslash,
       add('N',add('N',add('N',add('N',empty-string)))) )))
[FAC] format-account-code =
--     add('N',add('N',add('N',
--     add('N',add('N',add('N',empty-string))))))
       add('N',empty-string)
```

```
[FP]  format-pin =
--     add('N',add('N',add('N',add('N',empty-string))))
       add('N',empty-string)
[SL]  pin-list=
       add( tuple(pin2,account-code2),
       add( tuple(pin2,account-code1),
       add( tuple(pin2,account-code0),
              empty-pin-account-code-tuple-seq )))
end AAB-data
```

Second we define the process module `AAB`. The import section corresponds with the SSD parts of the interface diagrams of Figure 12, 13 and 14. The process definition corresponds with the control structure of Figure 14; chaining is represented by a parallel composition.

Take for example the SIB `User-interaction` which offers the process `user-interaction : JOBNR×TELNR`. This process outputs a `STRING` or an error message (see Figure 7). The output is done by an action `send-chain()` such that the context can read the output by means of a `read-chain()`. The `JOBNR` is necessary to guarantee that an output of a SIB is read by the right chain. If we put this in a context and we represent chaining by parallel composition then we obtain in PSF:

```
( user-interaction(cli,j)
|| ( read-chain(j,to-be-determined).  .......
   + read-chain(j,error-char-interv-elapsed) .  .......
   + read-chain(j,error-time-expired)         .  .......
   + sum(s in STRING, read-chain(j,s)         .  .......
         )
   )
)
```

In Figure 14 we see a choice construct as well. The choice *if a then P1 else P2* is in PSF represented by

```
( [ a = true  ] -> P1
+ [ a = false ] -> P2
)
```

Using these two mechanisms we can construct the service `AAB` as follows.

```
process module AAB
begin
  parameters
    BCP-INFO begin
      functions
        used-resource :-> RESOURCE
        used-service  :-> SERVICE
        used-amount   :-> NAT
```

```
  end BCP-INFO
exports
  begin
    processes
      AAB : JOBNR # TELNR # TELNR
  end
imports
  Jobnr, Telnr,
  User-interaction
  { SSD bound by
     [ID                     -> beep                  ,
      rep-requested          -> true                  ,
      rep-type               -> rep-count             ,
      rep-interv             -> tick-duration         ,
      total-time             -> now                   ,
      total-counter          -> 4                     ,
      user-interruptibility -> true                   ,
      max                    -> 6                     ,
      min                    -> 1                     ,
      initial-input-waiting -> duration5              ,
      inter-char-interv      -> duration5             ,
      end-char               -> close                 ]
     to AAB-data
  },
  Verify
  { SSD bound by
     [format                 -> format-account-code   ,
      min                    -> 1                     ,
      max                    -> 1                     ]
     to AAB-data
    renamed by
     [verify                 -> verify-account-code   ]
  },
  Verify
  { SSD bound by
     [format                 -> format-pin            ,
      min                    -> 1                     ,
      max                    -> 1                     ]
     to AAB-data
    renamed by
     [verify                 -> verify-pin ]
  },
  Screen
  { SSD bound by
     [SCREEN-ITEM            -> PIN-ACCOUNT-CODE-TUPLE    ,
      SCREEN-ITEM-SEQ        -> PIN-ACCOUNT-CODE-TUPLE-SEQ,
```

```
        screen-list           -> pin-list                    ,
        is-in                 -> is-in                       ]
       to AAB-data
    },
    Charge
    { SSD bound by
       [resource              -> used-resource               ,
        service               -> used-service                ]
      to AAB-data
    },
    Translate,
    Dbase-atoms, AAB-data, Basic-sort,
    Poi-and-por-communication
  variables
    j            :-> JOBNR
    cli, dli     :-> TELNR
    account-code :-> ACCOUNT-CODE

definitions

AAB(j, cli, dli) =
hide(I-string, hide(I-error, encaps(H-string, encaps(H-error,
user-interaction(cli,j)
||
( read-chain(j,to-be-determined) .
  send-por-clear-call(j,to-be-determined)
+ read-chain(j,error-char-interv-elapsed).
  send-por-clear-call(j,error-char-interv-elapsed)
+ read-chain(j,error-time-expired).
  send-por-clear-call(j,error-time-expired)
+ sum(acc-str in sSTRING,
      read-chain(j,acc-str) .
      ([ verify-account-code(acc-str) = false ]
        -> send-por-clear-call(j,error-input-doesn't-fit-format)
      +[ verify-account-code(acc-str) = true  ]
        ->
        ( user-interaction(cli,j)
        ||
         ( read-chain(j,to-be-determined).
           send-por-clear-call(j,to-be-determined)
         + read-chain(j,error-char-interv-elapsed) .
           send-por-clear-call(j,error-char-interv-elapsed)
         + read-chain(j,error-time-expired) .
           send-por-clear-call(j,error-time-expired)
         + sum(pin-str in sSTRING,
               read-chain(j,pin-str)    .
```

```
              ([ verify-pin(pin-str) = false ]
                -> send-por-clear-call(j, error-input-doesn't-fit-format)
              +[ verify-pin(pin-str) = true ]
                ->
                ([ screen(tuple(pin(pin-str)
                         ,account-code(acc-str)) ) = false
                 ]-> user-interaction-beep(cli) .
                         send-por-clear-call(j, to-be-determined)
                +[ screen(tuple(pin(pin-str)
                         ,account-code(acc-str)) ) = true
                 ]->
                  ( charge(j,used-amount,
                           add(tuple(
                                 translate(account-code(acc-str)),
                                 100),
                               empty-ac-nat-tuple-seq     )
                          )
                  ||(read-chain(j,
                             error-sum-of-percentages-is-not-100) .
                     user-interaction-beep(cli) .
                     send-por-clear-call(j,
                             error-sum-of-percentages-is-not-100)
                    +
                     read-chain(j,ok) .
                     send-por-proceed-with-new-data            (
                             j,cli,dli                         ,
                             translate(account-code(acc-str)) )
                    )
                  )
                )
              )
             )
        )
       )
      )
     )
)
))))
end AAB
```

G.1 Standard Datatypes and IN Specific Datatypes

PSF does not contain any datatypes a priori. Therefore, we have developed a library of standard datatypes such as the booleans, naturals and characters. This library contains also *parameterised* data types such as the data type of sequences over an arbitrary sort of items. We will not give the modules for these data types in detail, the readers who are interested in the details of these modules are advised to contact one of the authors of this paper.

The only modules we discuss shortly are `Seq`, `Tuple` and `Sort-0-9`.

G.1.1 The parameterised data type Seq

Below we give the parameter and export section of the module `Seq`. The parameter section states that as argument of the module `Seq` it assumes a module `Item` which contains the sort `ITEM`. Moreover, it requires an equality predicate over `ITEM`'s which is necessary for defining the equality over sequences of `ITEM`'s. Finally, it requires a `nil` element in the sort `ITEM`. In the defining equations we need `nil` for putting

`hd(empty) = nil`

The names of the exported functions are "explained" by comment. The equations for `Seq` are organized such that the constant `empty` and the function `add : ITEM` $\times$ `SEQ -> SEQ` are the so called constructors. That means that PSF, and its tools, can always reduce a term of sort `SEQ` to a term which is constructed by `empty` and `add` only.

```
data module Seq
begin
  parameters
    Item begin
      sorts
        ITEM
      functions
        eq   : ITEM # ITEM -> BOOL
        nil  : -> ITEM
    end Item
  exports
    begin
      sorts
        SEQ
      functions
        empty :              -> SEQ
        add   : ITEM # SEQ  -> SEQ
        hd    : SEQ         -> ITEM
        tl    : SEQ         -> SEQ
        len   : SEQ         -> NAT
        cat   : SEQ # SEQ   -> SEQ
        is-in : ITEM # SEQ  -> BOOL
        eq    : SEQ # SEQ   -> BOOL
    end
  imports
    Nat, Bool
  variables
```

```
    i,j : -> ITEM
    s,s': -> SEQ
  equations
    [H2] hd(empty)      = nil
    [H4] hd(add(i,s))   = i

    [T2] tl(empty)      = empty
    [T4] tl(add(i,s))   = s

    [L2] len(empty)     = 0
    [L3] len(add(i,s))  = suc(len(s))

    [C1] cat(empty,s)     = s
    [C2] cat(s,empty)     = s
    [C3] cat(add(i,s),s') = add(i,cat(s,s'))

    [I1] is-in(i,empty)     = false
    [I2] is-in(i,add(j,s))  = eq(i,j) .or. is-in(i,s)

    [E1] eq(empty,empty)    = true
    [E2] eq(empty,add(i,s)) = false
    [E3] eq(add(i,s),empty) = false
    [E4] eq(add(i,s),add(j,s')) = eq(i,j) .and. eq(s,s')
end Seq
```

We will apply the convention that if we have a sort `S` and we construct the sort of sequences over `S` then this latter sort will be called `S-SEQ` and we rename the `empty` sequence to `empty-s-seq`. In this way it is easy to detect the type of for example `add(exp,empty-s-seq)`, even if `exp` is a rather complicated expression. The disadvantage, however, is that it can introduce rather large identifiers.

There is one exception on this convention, the set of strings. A string is a sequence of characters, since strings are rather standard we will use the name `STRING` in stead of `CHAR-SEQ`. We need also the set of sequences over strings, which will be called `STRING-SEQ`:

```
data module String-seq
begin
  imports
    Seq { Item bound by [ITEM  -> STRING           ,
                         eq    -> eq               ,
                         nil   -> empty-string     ] to String
          renamed by    [SEQ   -> STRING-SEQ       ,
                         empty -> empty-string-seq ]
        }
end String-seq
```

G.2 The parameterised data type Tuple

In certain cases we want to construct from two sorts the *cartesian product*, in other words the set of tuples. Therefore we have the parameterised data type `Tuple`. Its parameter section consists of two parts; one for each of the sorts. Again we require an equality predicate and a `nil` element over these sorts.

```
data module Tuple
begin
  parameters
    Item1 begin
      sorts     ITEM1
      functions
        eq   : ITEM1 # ITEM1    -> BOOL
        nil1 :                  -> ITEM1
    end Item1,
    Item2 begin
      sorts     ITEM2
      functions
        eq   : ITEM2 # ITEM2    -> BOOL
        nil2 :                  -> ITEM2
    end Item2
  exports
    begin
      sorts
        TUPLE
      functions
        tuple : ITEM1 # ITEM2   -> TUPLE
        proj1 : TUPLE           -> ITEM1
        proj2 : TUPLE           -> ITEM2
        eq    : TUPLE # TUPLE   -> BOOL
        nil   :                 -> TUPLE
    end
  imports Bool
  variables
    i1,i1' :-> ITEM1
    i2,i2' :-> ITEM2
  equations
    [P1] proj1(tuple(i1,i2)) = i1
    [P2] proj2(tuple(i1,i2)) = i2

    [E]  eq(tuple(i1,i2),tuple(i1',i2'))
         = eq(i1,i1') .and. eq(i2,i2')
```

```
    [N]  nil = tuple(nil1,nil2)
end Tuple
```

Now we can define easily the set of tuples of ACCOUNT's and NAT's. Following our convention the resulting sort must be called ACCOUNT-NAT-TUPLE, but we abbreviate it slightly to AC-NAT-TUPLE.

```
data module Ac-nat-tuple
begin
imports
Tuple
{ Item1 bound by [ITEM1 -> ACCOUNT         ,
                  eq    -> eq              ,
                  nil1  -> account0        ] to Account
  Item2 bound by [ITEM2 -> NAT             ,
                  eq    -> eq              ,
                  nil2  -> 0               ] to Nat
 renamed by      [TUPLE -> AC-NAT-TUPLE    ,
                  nil   -> nil-ac-nat-tuple]
}
end Ac-nat-tuple
```

We need sequences over this sort as well:

```
data module Ac-nat-tuple-seq
begin
exports
begin
functions
  totalsum : AC-NAT-TUPLE-SEQ -> NAT
end
imports
Seq
{ Item bound by [ITEM  -> AC-NAT-TUPLE           ,
                 eq    -> eq                     ,
                 nil   -> nil-ac-nat-tuple       ] to Ac-nat-tuple
 renamed by     [SEQ   -> AC-NAT-TUPLE-SEQ       ,
                 empty -> empty-ac-nat-tuple-seq ]
}
variables
  s :-> AC-NAT-TUPLE-SEQ
  a :-> ACCOUNT
  n :-> NAT
```

```
equations
[TS1] totalsum(empty-ac-nat-tuple-seq) = 0
[TS2] totalsum(add( tuple(a,n), s))    = n + totalsum(s)
end Ac-nat-tuple-seq
```

We define an additional function `totalsum` which computes the sum of all naturals occurring in the sequence of these pairs.

G.3 The data type Sort-0-9

Several data types, such as the data type of Accounts, Services, etc. are nothing more than a finite set of constants without any further structure; that is, without any further functions such as addition etc. Therefore we define the module `Sort-0-9` which defines the sort `SORT` and ten constants `s0`,...,`s9`. Furthermore we define an equality predicate.

```
data module Sort-0-9
begin
  exports
    begin
      sorts
        SORT
      functions
        s0 :-> SORT
        s1 :-> SORT
        s2 :-> SORT
        s3 :-> SORT
        s4 :-> SORT
        s5 :-> SORT
        s6 :-> SORT
        s7 :-> SORT
        s8 :-> SORT
        s9 :-> SORT
        eq : SORT # SORT -> BOOL
    end
  imports
    Bool,Nat
  functions
    I    : SORT -> NAT
  variables
    s,s' :       -> SORT
  equations
    [E0] I(s0) = 0
    [E1] I(s1) = suc(I(s0))
    [E2] I(s2) = suc(I(s1))
    [E3] I(s3) = suc(I(s2))
```

```
    [E4] I(s4) = suc(I(s3))
    [E5] I(s5) = suc(I(s4))
    [E6] I(s6) = suc(I(s5))
    [E7] I(s7) = suc(I(s6))
    [E8] I(s8) = suc(I(s7))
    [E9] I(s9) = suc(I(s8))

    [EQ] eq(s,s') = eq(I(s),I(s'))
end Sort-0-9
```

We can define now the data type ACCOUNT as follows

```
data module Account
begin

imports
Sort-0-9
{ renamed by [SORT -> ACCOUNT ,
              s0   -> account0,
              s1   -> account1,
              s2   -> account2,
              s3   -> account3,
              s4   -> account4,
              s5   -> account5,
              s6   -> account6,
              s7   -> account7,
              s8   -> account8,
              s9   -> account9]
}
end Account
```

Other data types which are obtained similarly are JOBNR, TELNR, RESOURCE etc.

The data type PIN is defined by a renaming of Sort-0-9 as well but it has an additional function pin : STRING -> PIN which defines for each STRING a certain pin0,..., pin9. The data type ACCOUNT-CODE is defined likewise.

Some data types have meaningful constants. For example ANNOUNCEMENT has constants like error-beep, accept-tone etc. We obtain this sort by renaming of Sort-0-9; however, s0 is not renamed into announcement0 but to error-beep. Also MESSAGE is defined in this way.

H The Clock

H.1 The data module Time

First we give a specification of the datatypes `TIME` and `DURATION`. For each element n of sort `NAT` there is an element in `TIME`, denoted by `time(n)`, and an element in `DURATION`, denoted by `duration(n)`. We define some boolean functions over `TIME`, such as `gt` (greater than), `gte` (greater than or equal) etc.

We can add a `DURATION` to a `TIME` which gives a `TIME` again. Furthermore, we can subtract two `TIME`'s which gives a `DURATION`.

Finally, we have some constants of which their names speak for themselves.

```
data module Time
begin
  exports
    begin
      sorts
        TIME, DURATION
      functions
        initial-time  :                     -> TIME
        tick-duration :                     -> DURATION
        gt            : TIME # TIME         -> BOOL
        gte           : TIME # TIME         -> BOOL
        lt            : TIME # TIME         -> BOOL
        lte           : TIME # TIME         -> BOOL
        eq            : TIME # TIME         -> BOOL
        _ + _         : TIME # DURATION     -> TIME
        _ - _         : TIME # TIME         -> DURATION
       time           : NAT                 -> TIME
       duration       : NAT                 -> DURATION
       now            :                     -> TIME
       immediate      :                     -> DURATION
    end
  imports Nat,Bool
  variables
    n,m : -> NAT
  equations
    [1]  initial-time         = time(0)
    [2]  tick-duration        = duration(suc(0))
    [3]  gt(time(n),time(m))  = gt(n,m)
    [4]  gte(time(n),time(m)) = gte(n,m)
    [5]  lt(time(n),time(m))  = lt(n,m)
    [6]  lte(time(n),time(m)) = lte(n,m)
    [7]  eq(time(n),time(m))  = eq(n,m)
    [8]  time(n) + duration(m)= time(n + m)
    [9]  time(n) - time(m)    = duration(n - m)
```

```
    [10] now                  = time(0)
    [11] immediate            = duration(0)
end Time
```

H.2 The process module Clock

To define a `CLOCK` we import the module `Time` and we define the atom (atomic action) `tick`, for expressing the ticking of the `CLOCK`, and a local process `CLOCK:TIME`. We export the process `CLOCK` (the one without argument) and three atomic actions, e.g. `read-time`, `send-time` and `comm-time`, all of them have a parameter `TIME`.

The `CLOCK` starts at `initial-time`. The process `CLOCK(t)` expresses the clock at time t, it can either `tick` and increase the time with one `tick-duration` or it does a `send-time(t)`, by which the time doesn't change. The atom `send-time(t)` can communicate with the atom `read-time(t)`, resulting in the action `comm-time(t)`.

H.3 An Example

As example of the use of the process `CLOCK` we give a process which executes every 5 time units the atom `a`. Moreover, at every moment it can execute the atom `b` after which it takes 10 time units to execute another `b`.

```
process module Example
begin

exports
begin
processes example
end

import    Clock
atoms     a,b
processes A :   TIME
          B
variables t :-> TIME
definitions
A(t) =
read-time(t+duration(5)) . a . A(t+duration(5))

B =
sum(s in TIME, read-time(s) . b . read-time(s + duration(10))

example =
abstr(I-clock, encaps(H-clock, A(initial-time) || B || Clock ))
end Example
```

If we run the simulator, then at every moment the process `CLOCK` can do `tick`, also an `a`, `b` or `skip` (resulting from the hiding of the action `comm-time`) are possible.

Consider `A(0)`, initially it can not do anything, since the action `read-time(5)` is encapsulated and only the clock can tick. After the clock has ticked five times the internal action `comm-time(5)`, the communication between `read-time(5)` and `send-time(5)`, becomes enabled and we assume that this internal action will be executed before the next clock tick. For if we allow the clock to execute another tick before this internal action `comm-time(5)`, then the process `A` would not be able to execute any action anymore. We have to assume as well that the actions `a` and `b` are not delayed. For if the action `a` is delayed more than five clock ticks, after the internal action `comm-time(5)`, then `A(5)` can never pass its next action `read-time(10)` since the corresponding action `send-time(10)` is not possible anymore.

Therefore, we need the assumption that certain actions are not delayed. In the literature this assumption is known as *action urgency* [NS91], and in case only internal actions may not be delayed it is called *maximal progress* [Wan91].

In [Mau91] PSF has been extended with priorities of actions over other actions by which the above assumption can be expressed. However, the present implementation of PSF does not support priorities of actions which means that while simulating the specification we have to do all choices ourselves and, hence, we cannot use the random simulator.

I The Database

The database is represented by a process, called `DBASE`. Locally we have a process `DBASE:DBASE-CONF` where the data type `DBASE-CONF` contains all possible configurations of the database.

Before we can give the definition of `DBASE` we have to define the data type `DBASE-CONF` in the data module `DBASE-data` and the necessary atoms in the process module `DBASE-atoms`.

I.1 The data module DBASE-data

The database gives for each tuple (`a`,`s`,`r`), where `a:ACCOUNT`, `s:SERVICE` and `r:RESOURCE`, an element of sort `NAT`, denoting the amount of money which is charged to the account `a` for the use of the service `s` and the resource `r`. In other words the database is a mapping from these three-tuples into the natural numbers. So, we construct the data type `DBASE-CONF` from the parameterised datatypes `3TUPLE` and `MAP`.

We instantiate the parameterised datatype `3TUPLE` by the datatypes `ACCOUNT`, `SERVICE` and `RESOURCE`. Since this will be considered as the *domain* of the database we call this instantiated three-tuple `DBASE-CONF-DOM`:

```
data module Dbase-dom
begin
imports
```

```
3Tuple
{ Item1 bound by [ITEM1 -> ACCOUNT         ,
                  eq    -> eq              ,
                  nil1  -> account0        ] to Account
  Item2 bound by [ITEM2 -> RESOURCE        ,
                  eq    -> eq              ,
                  nil2  -> resource0       ] to Resource
  Item3 bound by [ITEM3 -> SERVICE         ,
                  eq    -> eq              ,
                  nil3  -> service0        ] to Service
 renamed by     [3TUPLE -> DBASE-DOM       ,
                 nil    -> nil-dbase-dom   ]
}
end Dbase-dom
```

Next, we instantiate the parameterised data type `MAP` by taking `DBASE-CONF-DOM` as domain and `NAT` as range. Finally, we rename this instantiated sort to `DBASE-CONF` and we rename the `empty` instantiated `MAP` to `empty-dbase`.

```
data module Dbase-data
begin
imports
Map
{ Dom bound by [DOM   -> DBASE-DOM    ,
                eq    -> eq           ] to Dbase-dom
  Ran bound by [RAN   -> NAT          ,
                eq    -> eq           ,
                nil   -> 0            ,
                _ + _ -> _ + _        ] to Nat
 renamed by    [MAP   -> DBASE        ,
                eq    -> eq           ,
                empty -> empty-dbase  ]
        }
end Dbase-data
```

I.2 The process module DBASE-atoms

The process `DBASE` communicates with the SIB's for exchanging `ACCOUNT`'s, `RESOURCE`'s, `SERVICE`'s and `NAT`'s. Moreover, before the process `DBASE` exchanges information with a SIB it must be communicated whether it is an *update* or a *request*. Therefore, we take the module Sort-0-1 which exports the sort `SORT` with two constants, `s0` and `s1` which we rename to `update` and `request`.

We define atoms `read-dbase:ACCOUNT`, `send-dbase:ACCOUNT` and `comm-dbase:ACCOUNT`. We define for each `ac` of sort `ACCOUNT` that `read-dbase(ac)` can communicate with `send-dbase(ac)` resulting in `comm-dbase(ac)`. Finally, we define the encapsulation set `H-dbase-ac` and the hide set `I-dbase-ac`. We do similarly for the other sorts.

```
process module Dbase-atoms
begin

exports
begin

atoms

read-dbase, send-dbase, comm-dbase : ACCOUNT
read-dbase, send-dbase, comm-dbase : RESOURCE
read-dbase, send-dbase, comm-dbase : SERVICE
read-dbase, send-dbase, comm-dbase : NAT
read-dbase, send-dbase, comm-dbase : DBASE-COMM

sets of atoms
H-dbase-a = { read-dbase(a),send-dbase(a) | a in ACCOUNT    }
H-dbase-r = { read-dbase(r),send-dbase(r) | r in RESOURCE   }
H-dbase-s = { read-dbase(s),send-dbase(s) | s in SERVICE    }
H-dbase-n = { read-dbase(n),send-dbase(n) | n in NAT        }
H-dbase-d = { read-dbase(d),send-dbase(d) | d in DBASE-COMM }

I-dbase-a = { comm-dbase(a) | a in ACCOUNT                  }
I-dbase-r = { comm-dbase(r) | r in RESOURCE                 }
I-dbase-s = { comm-dbase(s) | s in SERVICE                  }
I-dbase-n = { comm-dbase(n) | n in NAT                      }
I-dbase-d = { comm-dbase(d) | d in DBASE-COMM               }

end

imports
Dbase-data { renamed by [empty-dbase -> initial-dbase       ]
           },
Sort-0-1   { renamed by [SORT        -> DBASE-COMM          ,
                         s0          -> update              ,
                         s1          -> request             ]
           }

communications
read-dbase(a) | send-dbase(a) = comm-dbase(a) for a in ACCOUNT
read-dbase(r) | send-dbase(r) = comm-dbase(r) for r in RESOURCE
read-dbase(s) | send-dbase(s) = comm-dbase(s) for s in SERVICE
```

```
read-dbase(n) | send-dbase(n) = comm-dbase(n) for n in NAT
read-dbase(d) | send-dbase(d) = comm-dbase(d) for d in DBASE-COMM
end Dbase-atoms
```

I.3 The process module DBASE

The process `dbase` is a database containing the constant `initial-dbase`. If a database is in configuration `db` it can accept an `update` or a `request`. Remember that the complementary action of `read-dbase` is `send-dbase`, which occurs in a SIB like Charge.

In both cases a three-tuple of an `ACCOUNT`, `RESOURCE` and a `SERVICE` is read item for item. In case of an update also a `NAT` is read (which the additional charging) and finally the database is updated by storing the new configuration `add(tuple(ac,res,s),n,db)`. In case of a request the charged amount of money is obtained by applying the three-tuple to `db`, i.e., `app(tuple(ac,res,s),db)`, which value is sent back (to the SIB).

```
process module Dbase
begin
  exports
    begin
      processes
        DBASE
    end
  imports
    Dbase-atoms, Basic-sort
  processes
    dbase : DBASE
  variables
    db : -> DBASE
  definitions
    DBASE = dbase(initial-dbase)

    dbase(db)
    = read-dbase(update) .
      sum(a in sACCOUNT,
          read-dbase(a) .
          sum(r in sRESOURCE,
              read-dbase(r) .
              sum(s in sSERVICE,
                  read-dbase(s) .
                  sum(n in sNAT,
                      read-dbase(n) .
                      dbase(add(tuple(a,r,s),n,db))
         ))))
      +
```

```
    read-dbase(request) .
    sum(a in sACCOUNT,
        read-dbase(a) .
        sum(r in sRESOURCE,
            read-dbase(r) .
            sum(s in sSERVICE,
                read-dbase(s) .
                send-dbase(app(tuple(a,r,s),db)) .
                dbase(db)
       )))
end Dbase
```

J The Telephone

For every line identification a telephone process is started by defining:

```
TELEPHONES = merge(li in sTELNR, telephone(li))
```

J.1 The process module Telephones

As said earlier a telephone can play an active or a passive role in a call. This is shown by the process `telephone(cli)`, where `cli` is the calling line identification of the active telephone:

```
telephone(cli) = (active(cli) + passive(cli)) . telephone(cli)
```

First a Telephone in an active role is described, and second a telephone in a passive role.

In an active role a telephone first unhooks, see action `send-unhook(..)` in the specification. All actions executed by a telephone are read or send actions because they are communications with other processes, i.e., the BCP. Next, the active telephone may receive an error message sent by the BCP, or it may start a process called `busy(cli)` in parallel with a process `choose-action(cli)`. The process `busy(cli)` tells the world that the telephone, with calling line identification `cli`, is busy. The process `active(cli)` and `busy(cli)` are defined as follows:

```
active(cli) = send-unhook(cli) .
               ( read-user(cli, error-beep)
               + (busy(cli) || choose-action(cli))
               )

busy(cli)    =   send-busy(cli) . busy(cli)
               + read-stop(cli)
```

In the process `choose-action(cli)` a choice can be made between dialling a number, i.e., executing a standard call, or first giving an access code and then dialling a number, i.e., using the Automatic Alternative Billing service (AAB), because AAB is the only service considered in this specification. The processes `choose-action(cli)` and `conversation-with-service(cli, aab)`, where `aab` stands for AAB, deal with these alternatives:

```
choose-action(cli)
= (sum(dli in sTELNR, send-number(cli, dli) .
         ( read-user(cli, error-beep)
         + wait-for-connection(cli, dli)))
         + sum(ac in sACCESS-CODE,
                     send-access-code(cli, ac) .
                      conversation-with-service(cli, ac))
         ) .  send-stop(cli)

conversation-with-service(cli, aab)
= sum(dli in sTELNR, send-number(cli, dli) .
           conversation-with-aab(cli, dli))
```

In the first alternative, i.e., after only dialling a number, an error may be received from the BCP, or the process continues as defined by the process `wait-for-connection(cli, dli)`. In the second alternative, defined by the process `conversation-with-aab(cli, dli)`, an account code and a pin code is sent to the AAB service. Next again an error may be received, or the process `wait-for-connection(cli, dli)` is called. The process `wait-for-connection(cli, dli)` may receive a busy tone from the BCP, which means that the dialled telephone is busy itself, or a conversation may follow given by the action `talk-caller(cli, dli)`. After the conversation the telephone is hooked.

```
conversation-with-aab(cli, dli)
= read-user(cli, beep) .
  send-account-code(cli) .
  ( read-user(cli, error-beep)
  + read-user(cli, beep) .
      send-pin-code(cli) .
      ( read-user(cli, error-beep)
      + wait-for-connection(cli, dli)
      )
  )

wait-for-connection(cli, dli)
=   read-user(cli, busy-tone)
  + talk-caller(cli, dli) . send-hook(cli)
```

The processes `send-account-code(cli)` and `send-pin-code(cli)` are not specified here for they simply contain the sending of a number.

In a passive role a telephone first receives a connect from the BCP and then two processes are called in parallel, namely the `busy(li)` process defined above and the process `passive-ringing(li, cli)`. Here `li` is the line identification of the passive telephone and `cli` is the calling line identification, i.e., identifying the telephone that calls. The process `passive-ringing(li, cli)` lets the telephone ring and next it may answer the call or let it ring again. If the call is answered a conversation follows and is ended by hooking the telephone. The processes `passive(li)` and `passive-ringing(li, cli)` are defined as follows:

```
passive(li)
= sum(cli in sTELNR, read-connect(cli, li) .
          (passive-ringing(li, cli) || busy(li)))

passive-ringing(li, cli)
= read-ring(cli, li) .
   ( passive-ringing(li, cli)
   + send-is-answered(cli, li) .
       talk-called(cli, li) .
       send-hook(li) .
       send-stop(li)
   )
```