IT Licentiate theses
2000-009

A formalisation of the ITU-T Intelligent Network standard

JAN NYSTRÖM

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA UNIVERSITY

A formalisation of the ITU-T Intelligent Network standard

BY

JAN NYSTRÖM

December 2000

DEPARTMENT OF COMPUTER SYSTEMS
INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Systems
at Uppsala University 2000

A formalisation of the ITU-T Intelligent Network standard

Jan Nyström

Jan.Nystrom@DoCS.UU.SE

Department of Computer Systems
Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

Telecommunication systems are today among the largest and most heterogeneous computer systems that exist. The functionality offered by them is rapidly increasing, by numerous features: call waiting, credit-card billing and call-forwarding to name a few.

The realisation of extra services poses a challenge to implementors, in particular since different services are developed at different times and places, by different people. Not only is the number and complexity of services increasing but some vendors want to enable users to tailor their own services and ultimately design them entirely. This of course calls for rigorous control of the services so that they do not come into conflict with the interest of the vendors, other users or surprise the users with their behaviours.

One way of aiding the service designers is to provide a service creation environment containing a set of well defined building blocks that would be uniform for all features, irrespective of vendor or service. Such an environment also needs tool support for writing, checking, validating and analysing for possible conflicts.

We have constructed a formalism for compactly specifying the interface behaviour of the switching and service logic system underlying a service creation environment, and for specifying the behaviour of components of an environment for service creation. For this formalism we supply tool support in the form of a simulator.

We have further made a formalisation, in our framework, of the ITU-T Intelligent Network model, Capability Set-1. The formalisation concerns both the underlying service architecture, in which service logic is perform by a dedicated Service Control Function, and the component language, in which Service Independent Building Blocks are composed to yield new services.

Acknowledgments

First I would like to thank my supervisor Prof. Bengt Jonsson for accepting me as Ph.D. student and always being a source of inspiration. He has made sure, more often than not by means of a simple question, that I think twice about the why and wherefores of my work.

During the majority of the work I have been accompanied by me fellow Ph.D. student Gustaf Naeser, with whom I have often collaborated. His work on the earliest versions of the model simulator is but the least of the help and support he has given me over the years at, or outside, work.

I would also like to thank Bernhard Steffen and Tiziana Margaria lately of Dortmund, with whom I have cooperated and discussed over the years. We have always been able to rely on them for a different view on matters, something I find invaluable.

How worthwhile your work is is not determined by your coworkers, but your coworkers very much determines how enjoyable tour work was. And my friends and coworkers at the Department of Computer Systems has made this work enjoyable indeed.

In particular I would like to thank Jakob Carlström and Olle Gällmo for their help in my extracurricular studies, of home brewing and playing the Swedish bagpipe, respectively.

As support and inspiration there where two of my contemporary undergraduates students, Kristina Lundqvist Göran Wall, who started their Ph.D. studies before me and have now finished their degrees and left the department.

There has also always been the guiding star, Björn Victor. Björn is a man who will with the same ease advice you on the intricacies of LaTeX and process calculi as that of a good cigar.

My first lasting contact with the department, dating from my first years as a undergraduate student, was Inga-lisa Eriksson who has always proved to be a good friend.

Last but not least I have to thank my family, more so than ever after the intense period finalising the this thesis. They had what I needed most, patience and love.

Publications

Below follows a list of previously published work which is, although somewhat loosely, connected to the present thesis. The publication list is followed by notes on their contents, relation to this thesis and my contribution to them.

[A] A Formalization of Service Independent Building Blocks,
J. Nyström, B. Jonsson,
Proceedings of Advanced Intelligent Networks '96 workshop,
ed. T. Margaria, 1996.

[B] Creation of Dependent Features,
J. Blom, R. Bol, B. Jonsson, J. Nyström,
Proceedings of RadioVetenskap och Kommunikation '96, 1996.

[C] A Case Study in Automated Detection of Service Interactions,
G. Naeser, J. Nyström, B. Jonsson,
Proceedings of RadioVetenskap och Kommunikation '99, 1999.

[D] Building Tools for Creation and Analysis of Telephone Services,
G. Naeser, J. Nyström, B. Jonsson,
Proceedings of RadioVetenskap och Kommunikation '99, 1999.

[E] On Modelling Feature Interactions in Telecommunications,
B. Jonsson, T. Margaria, G. Naeser, J. Nyström, B. Steffen,
Proceedings of Nordic Workshop on Programming Theory,
eds. B. Victor and W. Yi., 1999.

[F] Incremental Requirement Specification for Evolving Systems,
B. Jonsson, T. Margaria, G. Naeser, J. Nyström, B. Steffen,
Feature Interactions in Telecommunications and Software Systems VI,
eds.M. Calder and E. Magill, ISO Press, 2000.

Paper [A] This paper is a work in progress paper describing one of the first versions of the present work. I did the major part of this paper.

Paper [B] This paper presented the current research at the feature interaction group which at that time consisted of the authors of the paper. I supplied the text for the part concerning my work, which was heavily based on Paper [A].

Paper [C] is an early description of the model for analysis of the services presented in the Feature Interaction Detection Contest[FI:Griffeth98]. My contribution consisted of part of the implementation and testing.

Paper [D] This paper presents briefly, for the first time the model of this thesis in its present shape. Beside the continued collaboration of my supervisor Bengt Jonsson, I had advisory and implementation help on this paper by Gustaf Naeser. I wrote the majority of the paper.

Paper [E] is published in a collection of extended abstracts. The paper is about using requirements when detecting feature interactions. I was part of the initial discussions and participated in implementing and testing the techniques. I also contributed to the final stages of writing.

Paper [F] is a paper related to Paper [E] focusing on how requirements through a taxonomy of predicates may adopt to changes in both the specification and implementation of the model. My contributions to the paper where the same is in Paper [E].

Contents

List of Tables

List of Figures

Part I

Getting started

CHAPTER 1

INTRODUCTION

This introductory chapter we will present the background, the goals and the contributions of this thesis as well as a description of the design decisions made along the way to the goals. Finally an overview of the rest of the thesis is given.

1.1 BACKGROUND

This thesis is focused on features and system made out of features. Our notion of a feature stems from the field of telephony systems where the features[1] add services to the basic call handling of the telephony system. Taking a more general perspective, a feature is any part of the system having a self contained functional role. In fact, many systems can be viewed as consisting of a base service, in itself perhaps quite useless, with a set of features added.

The upside of viewing or constructing a system in terms of a simple base service with additional features is that it makes it easy to tailor a particular instance of the system to the exact needs of a user by choosing a set of features to add. The downside is that when designing one feature one has to take into account all the other features that may be present in the actual system. The concept of a feature, in the sense we use the term, emerged in the telecommunication field and it is within this field we will look for concrete problems.

The reason to choose telephone services as our field of study is that telecommunication systems are today among the largest and most heterogeneous computer systems that exist. The functionality offered by them is rapidly increasing[2]; earlier the main purpose of telecommunication systems was to manage connections between telephones, but now their functionality has been extended by numerous features: call waiting, credit-card billing and call-forwarding to name a few. The realisation of extra services poses a challenge to implementors, in particular since different services are developed at different times and places, by different people.

Not only is the number and complexity of services increasing but some vendors want to enable the users to tailor their own services and ultimately design them

[1] In this thesis we will use the terms feature and service almost interchangeably. A feature may be a service or only part of one or more services, but the feature will always have a self contained functional role.

[2] In 1992 it was reported [NorthernTelecom92] that the DMS–100 switch supported over 800 different features.

entirely. This of course calls for rigorous control of the services so that they do not come into conflict with the interest of the vendors, other users or surprise the users with their behaviours. It is of course vital also that no conflicts or anything unwanted/unforeseeable occur. Staggering as this problem seems today, the problem will become even worse with the growing flexibility. Hitherto one could stick one's head in the sand and thus avoid some of the problem, e.g., by only allowing one service at a time to be active for each user, but that is hardly satisfactory and will not meet the demands of the users in the near future.

It is the abovementioned problem we want to address. One way of writing more structured features is to use a set of well defined building blocks that would be uniform for all features, irrespective of vendor or service. The advantage of writing services as combinations of building blocks is that it not only makes the possible conflicts more easy to find, by highlighting the structure of the features, but that it also makes it easier to write the services.

When designing a service, not only should it be possible to build services using simple and understandable building blocks, but there should also be the possibility to execute the service from its design specification in order to validate the service. After validating the service by running it in isolation and checking that it behaves as intended, one can test it in an environment with other services present.

Together with the validation it is also important to be able to test and verify the service against the requirements of the service, of other services and of the system. This analysis, although interesting, is out of scope of the present thesis but has been pursued in other work[3] and it would be interesting to apply the result of that work to the model presented in the present thesis.

1.2 QUESTIONS AND ANSWERS

In this section we will try to present the questions we have posed when pursuing the research presented in this thesis and how we intended to answer them. Finally we will present an example of the result.

1.2.1 Questions

Here we will summarise the issues, found in the design of telecommunication system, that we wish to address.
The issues are:

- how do we design features and services in a structured way;
- how do we assure ourselves that the design of a feature follows our intentions;
- how do we know that a feature does not come in conflict with existing features[4];

[3] See [FI:Naeser99, FI:Jonsson99, FI:Jonsson00] for work by the author on analysis.

[4] See [FI:Bowen88] for an introduction to the notion of feature interactions, and see [FI:Cameron93b, FI:Cameron94] for a categorisation of causes of interactions in telecommunication systems.

- if a feature comes into conflict with another feature, how do we find the source of the conflict;
- if we know the source of a conflict between features, how do we resolve the conflict.

1.2.2 Answers

We will describe how we view these questions and how we intend to go about answering them.

structured design

Our answer to structured design is the block diagram style, where each type of block has a simple functionality, e.g., to translate according to a given table.

The connections in the diagram denotes flow of control between the blocks, where an additional diagram defines the flow of data.

New blocks can be constructed hierarchically from already defined blocks, in the same way we have procedures in a general purpose programming language. This in order that one may group related issues and frequent substructures of the features may be reused.

As important, as how we structure the services, is how we incorporate the services into the base system. In order that it should clear where the services are invoked, the services are initiated from a set of well defined points in the normal call handling. A typical point of invocation would be after the user has dialed a number and it has been analysed by the switch, or at the end of a call.

fidelity of design

The fidelity of the design to the intentions of its designer can either be assured through simulation and testing or through analysis of formal requirements of the design. Simulation and testing requires that the design will be executable, whereas formal analysis requires that the design be rigorous. We want to be able to test, as well as to perform formal verification, thus we need a formal specification that is executable.

finding conflicts

The problem of handling conflicts can be grouped into detection, resolution and avoidance according to Cameron and Velthuijsen[FI:Cameron93a]. All of these three important parts of conflict handling can either be performed offline or online, i.e., either before the system is deployed or in the actual running system.

Avoidance entails measures taken to ensure that conflicts do not arise; we believe that this can be achieved best offline through structured design with clear links to the requirements. Further is it important to be able to validate design decisions as early as possible, through, e.g., executable designs.

Detection, the actual finding of conflicts, should to the greatest possible extent be performed offline; achieving predictable and satisfactory resolution for all conflicts in a running system seems hardly possible.

We have not addressed the issue of detection further in this thesis. These thoughts on detection, as well as those on diagnosis and resolution, have been incorporated into the thesis since our views on these issues has influenced the thesis.

conflict diagnosis

The main support for diagnosis of conflicts between features we envision is the animation of traces of the system, highlighting the parts that went wrong. Traces have been shown to be a valuable diagnostic tool in for example UPPAAL[5] and SPIN[6]. Diagnostic traces have also been suggested as a means to identify detected conflicts online in order to trigger resolution mechanisms by Aggoun and Combes in [FI:Aggoun97].

conflict resolution

In existing systems, conflicts are often resolved by the redesign of one or both of the features involved. We wish to take a more structured approach where one does not actually change the design/implementation of the features but simply decides what cookbook recipe should be used. An example of such a recipe, which is already employed in existing systems, is the assignment of priorities to the services. This will of course require support in the system for resolution methods, such as a mechanism for setting priorities.

1.3 OBJECTIVES

Our objective is make a formal model of a service creation environment containing a set of well defined building blocks that would be uniform for all features, and the interface behaviour of the switching and service logic system underlying the service creation environment.

The service creation environment we have chosen is defined by the ITU-T Intelligent Network standard, which describes how telephone switching systems may be expanded with additional services.

The model should handle the key issues in the following way:

structured design

There should be tool support for the construction of features from simple building blocks graphically in a hierarchical manner.

[5] This is briefly presented in [Larsen95].

[6] In [Holzmann98] Holzmann and Smith describes how they through inspection of traces, using support in the SPIN tool, debug a prototype model of an Industrial project, and goes on to find faults in the specification.

fidelity of design

Firstly the model of the base system should mirror the IN–standard in such a direct manner and using simple notation so that confidence in the model can be gained by inspection. Furthermore the model should be executable so that one can validate the model, through simulations and test runs.

Secondly the design of actual features can be validated through testing in the executable model of their design.

Finally, it should be possible to check that formal requirements of the system and features are satisfied in the test scenarios. It should also be possible to formally verify the requirements.

finding conflicts

Conflicts should be found when[7]:

- different features try to perform actions that are inconsistent with each other;
- features try to perform actions that are inconsistent with the present system state;
- when formal requirements of a feature or the base system are voilated.

conflict diagnosis

The conflict finding part of the tool support should when a conflict is found, animate the trace of actions that led to the conflict situation, with the possibility to highlight the parts relevant to the conflicting feature(s).

conflict resolution

The tool service design should incorporate a number of resolution techniques, so that when a conflict is reported and diagnosed an appropriate method could be selected and the tool applies the necessary changes to the feature.

In the end we intend to have the structure in Figure 1.1 where, based on a model of the base system, new services can be designed, and the services can be simulated as well as analysed. The analysis should, if problems are found, produce a trace such that the simulator can animate the sequence of events leading to the source of the problems.

1.4 DESIGN DECISIONS

The present thesis was originally inspired by earlier work at the department in the area of feature interaction presented in [FI:Blom94, FI:Blom95] which dealt with services at a very abstract level. In these papers the services were modeled in first

[7] A thorough description of the different of types conflicts mentioned above, and an implementation to find them in a process model intended for the Feature Interaction Detection Contest [FI:Griffeth98] is found in [FI:Naeser00]

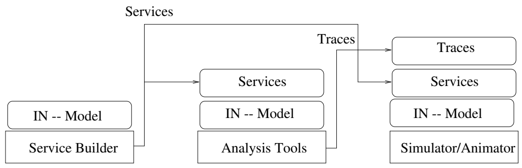


Figure 1.1: The solution structure.

order linear temporal logic, and further expanded with a modular specification style in [FI:Blom97].

We wanted to get closer to the actual designs of the services and choose the ITU-T Intelligent Network (IN) standard as the target for our efforts. There are several reasons for choosing the IN standard:

- Getting proprietary information from the companies involved in service design was hard, and if obtainable heavily restricted with regards to dissemination.

- The IN standard is a result from the standardisation work with participation from many of the major international companies and organisations concerned with telecommunication, thus making our results of wider interest.

- There was already great interest in the IN standard from other research groups [IN:Klusener91, IN:Verweij93, FI:Cameron93b, FI:Cameron94, IN:Csurgay94, IN:Tsang94, IN:Kang96], making the comparison of our works with others in the same area easier.

- The notion of service independent building blocks found in the standard seemed a suitable candidate for formalisation, having many of the properties of structured design we desired.

1.4.1 Standard

We have chosen to center our attention on the level of detail below that of services. In the terminology of IN standard this level of abstraction is called the Global Functional Plane. In the Global Functional Plane services are constructed out of service independent building blocks.

In order to understand the nature of the individual building blocks, we had to look a level of abstraction further down, at the Distributed Functional Plane. The Distributed Functional Plane defines what functional entities are needed in the network to realise the services and make them work together with the normal call handling[8], and the communication patterns between these entities.

[8]Normal call handling, or call control, is often referred to as the Plain Old Telephone System (POTS) in the literature.

In the approach we had chosen we needed to model the execution environment of the building blocks, as well as the mechanisms for activating services and the normal call control they affect. The reason to model the normal call control is that we knew that many of the interactions would manifest itself in the interaction between the services and the normal call control.

In order to model the framework of the services and normal call control we chose to model the relevant functional entities identified in the Distributed Functional Plane. It seemed natural to model the functional entities and the services as separate processes.

1.4.2 Formalism

An important issue was what formalism to use when modeling the IN standard. Our first effort started with TLA$^+$[Lamport95]; we found a language that is merely an extension of a linear temporal logic with Zermelo-Frenkels set theory appealing. TLA$^+$ however did not give the support for a hierarchical and structured model we needed and the formalisation tended cumbersome and opaque. This came in sharp contrast to our objective that the model should mirror the standard in a direct manner.

After this first start we continued with a formalisation in Z[Diller94, Nicholls95, Spivey87, Wordsworth92] which was presented in [IN:Nyström96] at the AIN'96 workshop. The Z notation however is highly declarative and since the standard is procedural in its structure our formalisations of the elements seemed strained. There was a further problem with the support for structuring the model, which seemed to have a possible solution the use of the object-oriented extension of Z called Object-Z[Duke91, Smith94]. Object-Z however lacked the tool support that existed for Z and the semantics [Griffiths94, Griffiths95a, Griffiths95b] was still immature and far too complex to inspire confidence.

We then turned to the formalism presented by van Hee [vanHee94], which combines Coloured Petri Nets [Jensen92, Jensen95] and Z (really a subset of Z). In this formalism the design of services could be made graphically with a given semantics. Furthermore the object and agent frameworks support better structuring of the specifications. Introduced to the notion of Coloured Petri Nets by van Hee we found the tool Design/CPN[Jensen91] which has a graphical editor with animation of the models and analysis tools. The Design/CPN tool has a notion of high level and parameterised nets, which however to a large extent is realised by several copies of the nets which makes it intractable in large model.

In the end we decided that the ease of modeling and the closeness between the model and the standard was the most important consideration and we chose to use the style of operational semantics[9] for programming languages. The style

[9] The author has not been able to find any text that thoroughly and in one place presents operational semantics; the best descriptions was found in works devoted to other related subjects: Stoy's book on denotational semantics [Stoy77] and de Bakker's book on program correctness [de Bakker80] or more recently Winskel book on semantics for programming lan-

was already familiar to the author and lends itself to easy implementation of a simulator. The ease of modeling is important since, as Capellmann et al. points out [FI:Capellmann97], more time is spent modeling than in analysis of the models.

Each functional entity is modeled as a separate process and the processes are described by a number of simple transformation rules of the form.

$$\frac{Guard}{State \xrightarrow{Actions} State'}$$

Here the *Guard* decides when the rule is applicable, in which case the state changes from *State* to *State'* while performing *Actions*. Where an operational semantic would define what the abstract machine should do when it encounters an operation in the code, our model defines what the processes should do when encountering an event in the input queue.

Modeling using this formalism was first presented in [IN:Naeser99].

1.4.3 Tool support

When it comes to tool support the downside of our choice of formalism becomes apparent: we have to supply tool support ourselves. We had from the beginning decided that the model should be executable and we can either translate the formalised model into something that can be executed or interpret them as they are.

An advantage of translating or compiling the model would be that then one has to formally define the syntax and can perform various syntactical checks to ensure that the model is well-formed. As what should be incorporated into the formalism has been decided as we have gone along, this would have meant constant rework of the syntax and the translating/checking tools.

We have chosen instead to interpret the formalised model. When new additions to the vocabulary of the formalism have been made, this has in most cases been accommodated by similar additions to the interpreter. In some cases, changes to the interpreter could not be avoided, but they have been comparatively few.

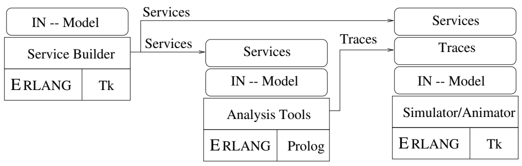


Figure 1.2: The concrete solution structure.

guages [Winskel89]. Many texts centered round a particular machine may be found: G-machine [Peyton87], SECD-machine [Landin64, Henderson80, Peyton87] for functional languages, and not to forget the probably most successful, WAM (Warren's Abstract Machine) [Warren83, Aït-Kaci90] for PROLOG.

We have chosen to write the interpreter in ERLANG[10]. We can today simulate the system, but the interface is textual and based on scripts to describe the scenarios. The ERLANG however is a modern languages and has libraries for graphics linking to Tk.

The intended solution structure presented in Figure 1.1 can now be made concrete as in Figure 1.2. In the concrete version of our goals:

Service builder has been written in ERLANG using ERLANG/Tk to implement the user interface, although only a first prototype.

Analysis tools will be written in ERLANG, PROLOG or even C, with the interface to the other parts and user interface written in ERLANG/Tk.

Simulator/Animator has been written in ERLANG/Tk.

1.5 CONTRIBUTIONS

We have formally modeled the ITU-T Intelligent Network standard. The model handles the key issues described in Section 1.2.1 in the following way:

structured design
The graphical building blocks style described in the standard, extended with a mechanism to express data flow, enables a structured design.

fidelity of design
The abstract machine style of operational semantics with simple mathematical notation gives an operational mode of modeling. This way of modeling makes the step from the informal standard to the formal model, considering the circumstances, easy.

The model is executable and a tool for executing the model has been written in ERLANG.

finding conflicts
Nothing has been done towards the detection of conflicts.

conflict diagnosis
Since no conflicts can be detected other as incorrect or unexpected behaviour of the simulations no diagnosis support has been added.

conflict resolution
The only built in support for conflict resolution is the possibility to give priorities to the services.

[10]See section 4.3 for a more thorough description of ERLANG.

1.5.1 Example design scenario

To illustrate the definition of services in the IN–model we present an example definition of a service, where the basic building blocks defined in IN are used. The building blocks of the IN–model are called Service Independent Building Blocks or SIBs for short. The service definition is presented using the block diagrams suggested in the standard, augmented with a diagram describing the the dynamic data flow of the service and the static instantiation of the building blocks.

The Service is the Automatic Alternative Billing service (AAB) that allows a subscriber to charge an alternative account when placing a call. The service is activated prior to dialing a destination number. The subscriber is queried for the account to which he wants to redirect the charge, and for the appropriate PIN code.

Control

We will first describe the control flow of AAB, shown in diagram in Figure 1.3. Each SIB is presented as a rectangular box with the mnemonic name of its instance in **bold** and the SIB type in normal font below. In the two first SIBs in the chain, the SIB type name is surrounded by brackets to indicate that they are hierarchical SIBs, defined in a similar way to a service. From each SIB rectangle emanates a number of arcs representing the possible outcomes of the SIB's execution (these are logical ends in the vocabulary of the IN standard). The SIB rectangle in which the arc ends, is the next SIB in the flow of control.

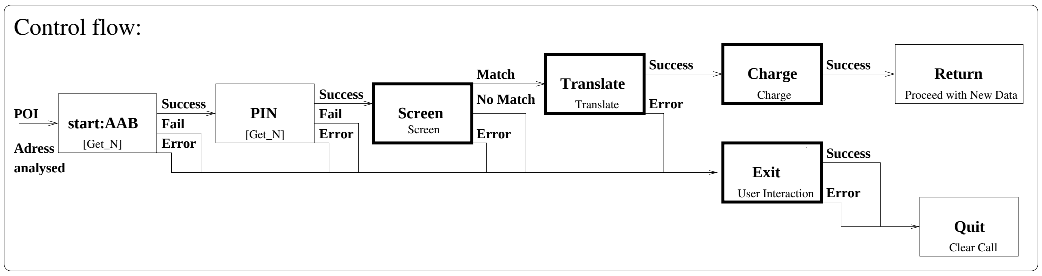


Figure 1.3: Example: Control flow in the Automatic alternative billing service

The leftmost arc that does not emanate from a SIB is annotated with the point in the call (Point of Initiation – POI) at which the service is invoked. The rightmost rectangular boxes lack outgoing arcs and represent Points of Return (POR), which like the SIBs have a mnemonic in **bold** and the type below in normal font.

- The POI for the AAB service is at the point of the call where the address of the call has been analysed; in this case it is not an address to another subscriber but a request for the AAB service.

- The First SIB is called **start:AAB** and of SIB type Get_N, which is a hierarchical SIB that collects an account number from the subscriber, giving him a number of tries to supply an account in the correct format. The SIB has three logical ends: **success**, **fail** or **error**.
- The **PIN** SIB is of SIB type Get_N as well, and collects the PIN from the user.
- The **Screen** SIB is of SIB type screen and checks whether the data supplied is found in its screening list, the logical ends are **match**, **nomatch** and **error**. The logical end **error** signifies that some internal error has occurred, such that the screening list does not exist.
- The **Translate** SIB is of SIB type translate converts the account number given by the subscriber to the actual account which can be an external account, e.g., a bank account, to be charged. The translate SIBs logical ends are **success** or **error**, where **error** would typically be the result when there is no correct conversion to be had.
- The **Charge** SIB of type charge signals the billing system that the charge shall be handled by the service. The only logical end is **success** and the normal call handling is resumed but with new data.
- The **Exit** SIB is of type user interaction and informs the subscriber that the service has failed. The service has two logical ends, **success** and **error**, but in both cases the call is cleared.

Data

To show the flow of data in the service as seen in Figure 1.3, the SIBs are once again presented as rectangles in the same manner as for the control flow but here the arcs (dotted so that one can superimpose the two diagrams without giving rise to ambiguity) represent the flow of dynamic data between the SIBs. The endpoints of the flows are annotated with the name of the dynamic data, or call instance data (CID) in IN terminology.

In some cases the dynamic data is composed of parts received from several preceding SIBs, as is the case with screen data of the **Screen** SIB. The $\langle\ ,\ \rangle$ conjoining data from the **start:AAB** and **PIN** SIBs, the two data flows indicate that the screen data is the tuple constructed from these.
Below the SIBs there is a list of statically instantiated data, or service support data (SSD) in IN terminology.

The statically instantiated data is instantiations of the SIBs performed at the definition of the service; specialising the SIBs for their role in the service. One SIB may have several different instantiations in the same service, e.g., the Get_N SIB which has two different instances in AAB.

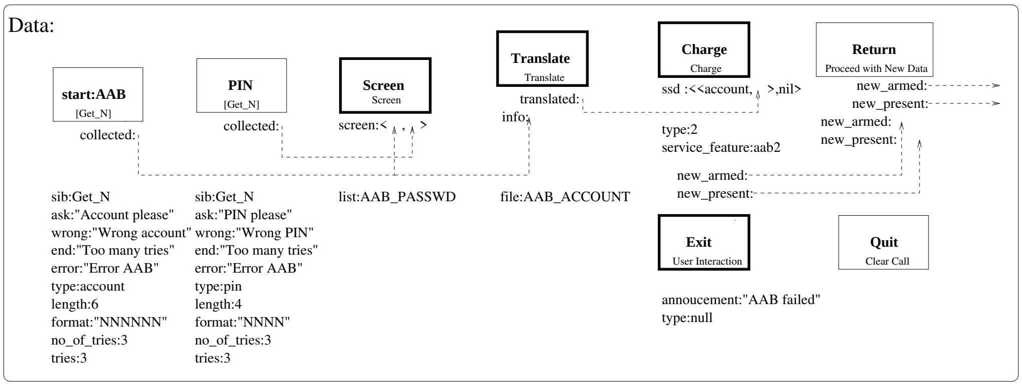


Figure 1.4: Example: Data flow in the Automatic alternative billing service

Below is explained the SSD of the SIBs in AAB:

- The **start:AAB** SIB being a hierarchical Get_N SIB has the following static data:

  sib The name of the hierarchical sib, in this case Get_N.

  ask The question posed to the user.

  wrong The answer given a when a string on the wrong format is supplied by the user.

  end The answer given to the user when he has made the maximum number of tries.

  error The answer given to the user should an internal error occur.

  type The type of string expected from the user.

  length The length of the string expected from the user.

  format The format string expected from the user, e.g. in this case six digits.

  number_of_tries The maximum number of tries allowed.

  tries This is the place of the internal counter that keeps track of how many tries have been performed so far.

- The **Screen** SIB has the following static data:

  list The screening list to be used.

- The **Translate** SIB has the following static data:

  file The translation list to be used.

- The **Charge** SIB has the following static data:

type The type of charging performed.

service_feature The service feature taking over the charging, as we have modeled the normal charging procedure as a service feature.

- The **Exit** of type user interaction SIB has the following static data:

announcement The announcement to be made to the user.

type What type of user interaction that is to occur.

There are two dataflows from the "proceed with new data" POR that leaves at the right hand side of the figure, these are the values that are returned from the service to the initiator when the service terminates so that the switch can be updated.

Similarly any information supplied by the invoker when invoking of the service would have been listed at the left of the first SIB in the chain, e.g., see Figure 14.1 depicting the Normal billing service feature.

The meaning of the two returned dataflows are:

new_armed are the new value of the "armed" information of the call, which determines what services may be started and under what condition they should start.

new_present are the new value of the "present" information of the call, which determines when events should spontaneously be generated by the IN capabilities of the switch. An example of event to be generated by the switch would be logging information to a statistic service.

1.6 RELATED WORK

Overview of the field

If one wishes to find a general overview of the field of feature interaction and models of the IN standard there is a survey by D.O. Keck and P.J.Kuehn in [FI:Keck98b].

Level of abstraction

We have several times stressed the importance we put on modeling at the right level of abstraction, this has been mentioned also by other authors such as M. Calder in his paper with the provocative title: *What Use are Formal Design and Analysis Methods to Telecommunications Systems* [FI:Calder98]. The level of abstraction will to a large extent determine what we can possibly analyse and how relevant the model is to a particular framework, such as the ITU-IN standard.

For example G. Naeser[FI:Naeser00] does not believe that it is realistic to perform feature interaction analysis on a full model of the ITU-IN standard using the methods currently available.

In the sections below we will see more concretely what influence the level of abstraction may have.

Separation of concerns

In some cases the high level of abstraction will give the results a more general application since they are valid in all framework, a good example is P. Zaves's study[FI:Zave95] of how the incorrect separation of concerns can lead to interactions in the services. In this paper is showed that a service that encapsulates several complementary notions, in the particular case of call forwarding two notions, we can not supply one realisation and have the expectations of both notions satisfied. In this case the difference of the two notions is not apparent when studying the service in itself, it only becomes evident in the presence of other services or several instances of call forwarding.

Temporal logic and constraints

One example of others that have taken a very high level approach is Blom et al [FI:Blom94, FI:Blom95], which used only first order linear temporal logic[Manna92] under a frame assumption to describe the services. The advantage is that the formalism is simple and relatively well understood by now, and the requirements of the system are described in the same formalism as the specification itself.

But the simplicity is double-edged in that whereas the simplicity makes ease to understand the individual notions used it does not support any structure in it self. The result is somewhat like the differences of using a high or low level language: what all instructions do in the low level assembly language can easy to understand but if care is not taken and program disciplined and with good structure the overall programme can be wholly incomprehensible. That is not to say it can not be done, I only suggest that the disciplines and structures used are of as great a importance as the formalism itself.

As the work Blom et al found that it helped the writing and understanding of the models a great deal to add a modular specification style[FI:Blom97].

Another approach where the same formalism is used both to write the model and the formulation of requirements is presented by Accorsi et al. in [FI:Accorsi00]. The formalism used is constraint programming with stable semantics as a basis for implementation.

The notion of constraint programming goes well in hand with the non-monotonic properties of feature addition. The adding off features can easily be viewed as adding further constraints to the system. The main disadvantage is that in order to find interaction one has to specify what properties would be violated.

The black box

A rather extreme view is taken by D. Marples and E.H. Magill in [FI:Marples98] where they detect and handle interactions between features online without any knowledge of the inner workings of the features. This is made possible by a mediator through which all communication to and fro the features passes. The mediator will examine the responses produced by features and if they conflict a resolution is triggered. The mediator will act as a wrapper of all the services and change their observable behaviour without changing the actual features.

The resolution will have the possibility to test what will happen as a result of the different feature responses. The responses are issued to the system and the result is observed and the "transaction" is then rolled back and the system reset to the previous state and the next response tested. When a response is chosen that is issued and "committed". The method resembles the rollback-commit strategies used in databases.

The reason for this black box approach is that twofold: first that it may be impossible to make changes in legacy systems, and secondly that features we will naturally not have access to the design of features provided by other vendors.

Further work to handle more complex combinations of input and output by the features has been presented in [FI:Reiff00] by S. Reiff. This work has been

This approach has the weakness that it is impossible to handle all the intentional interactions since the tool does not have the information needed to understand the combined intentions, these cases would have to be handled manually offline.

Another black box model, of in this case AIN 0.1 framework, is presented by Lin et al. in [FI:Lin98]. Here the features are once again black boxes, but their interface is well defined. the work is both detailed and at the same it treats many parts very abstractly, considering mainly what data is shared between the different features.

The examination of data flows and sharing of data is a good way find a lot of interactions depending on features having a conflicting views on shared data, or just being unaware that the data is shared.

Scenarios

In the papers [FI:Keck96, FI:Keck98a] D.O. Keck has combined modeling of the IN standard with a high level approach. Keck looks at scenarios of connection between SIBs and tries to determine which scenarios may give rise to interactions. What role the features play, if it is in an originating or terminating side of a call, and data usage is used to determine if the scenario is likely to contain interactions.

Focusing on the scenarios that may give problems and the elimination of symmetries considerably simplifies the task of finding interactions since only a small part of all the possible scenarios has to be taken examined.

A similar approach was used by K.H. Braithwaite and J.M. Atlee in [FI:Braithwaite94], where they used layered state transition machines to model the telephony system and services.

Specification and description language (SDL) of more detail

There has been made a few models of the IN standard of note in SDL[Z.100] formalism. Csurgay has in [IN:Csurgay94] presented an IN-laboratory, i.e., a highly detailed model of large part of the IN standard including distribution. The work by Csurgay is not only a prototype implementation as there simpler analysis available through the SDL tools. The services defined seem somewhat too simplistic and it would have been interesting indeed to see what analysis could be performed on realistic services.

An interesting idea of checking for possible interactions offline so that they can be identified online has been pursued by I. Aggoun and P. Combes, as presented in [FI:Aggoun97].

The model of the system is written in SDL, and in which the analysis is performed in the service creation environment and some interactions handled, but most importantly for each detected interaction a solutions strategy is chosen.

The features interaction are detected using passive observer, that are encodings of the service requirements, that run in parallel with the services in a functional model of the network.

When one of the interaction identified in the service creation environment is observed in the service execution environment the selected solution is applied and the intervention noted. The logging of interventions actually confronted in the service execution environment is a vital diagnostic tool to maintain the services and for the construction of new services.

Petri Nets

A work similar to the ideas we had about using Coloured Petri Nets is found in [IN:Capellmann98] by C. Capellmann, S. Christensen, and U. Herzog, which models the IN standard at BCSM level and analyse the behaviour through inspection of different visualisations. The tool they use for the simulation of the model is Design/CPN[Jensen91], which has support for highly customisable visualisation. This has enabled them to strictly model the BCSMs and the rest of the system, and yet having a visualisation that is understandable to people not used to formal methods.

Their main problem is that the model in Design/CPN will not scale to anything but relatively small scenarios, due to the problems we mentioned in Section 1.4.2 of copies of the nets when reusing subnets.

Process algebra

Klusener et al. pressents a model of the IN standard wholly based on the Global Functional Plane in [IN:Klusener91, IN:Verweij93], and formalised in Process Specification Formalism (PSF[Mauw91]). The purpose of the effort was to test the formalism as much as to get to grips with IN.

When only taking the view presented by the Global Functional Plane, as done by Klusener et al., some SIBs seem strange. For example the Translate SIB which only translates values according to a given translation table, why do we not simply translate the value according to the table in situ instead? Looking at the Distributed Functional Plane one realises that the Translate SIB has a valid reason to exist: the intension of the SIB is that it will perform a database lookup to make the translation. Unfortunately they decide to treat the SIBs that does not seem to have a dynamic behaviour different, called data SIBs, from the others.

The environment of the services that we have modeled with some of the functional entities from the Distributed Functional Plane are by Klusener et al. reduced to: a database and a clock. This choice greatly simplifies their model, although it does not feel as relevant.

The real strength of the paper is in the discussions concerning what parts of the SIBs should be omitted, such as parameters, and we have found inspiration in it when trying to make similar decision.

Service creation environment

One of the few complete service creation environments are the one presented by Steffen et al. in [IN:Steffen96]. This service creation environment for IN services has a graphical editor and supports animation and verification of consistency properties.

So within the service creation environment embedded in the METAFRAME [Steffen95], an environment for management of large repositories of tools, one can develop, configure, test and verify new services.

One of the strengths of this tool is the thematic views, presenting the user with only the parts relevant to a particular view. An example of a view is that of a service engineer specialised in billing, his view would only show those SIBs directly concerned with billing and yet he can change, animate and verify the service.

The verification is model checking of global properties expressed in a linear time variant of μ-calculus[Kozen83]. Should errors be found during verification a error view is presented with only the SIBs relevant to detection of the error. One can of course correct the error in the error view.

1.7 CONCLUSIONS

We have made a model of the ITU-T Intelligent Network standard together with simulator of the model.

From the work with the modeling we can conclude that the style of the modeling formalism is of great importance, if one are to capture the notions involved in a successful manner. In our case the different entities involved and the highly procedural description suggested the style of abstract machines used to describe the operational semantics of programming languages.

Many of the questions that arose during the modeling are the same as will arise during an realisation of the standard, and from that we can conclude that making a formal model of a standard at fairly early stage would be of great help to ensure that the different realisations conform.

1.8 FURTHER WORK

There a large number of things still to do in order to achieve all our original intentions. First and foremost on the agenda would be completing the service builder so that new services may be written in a graphical editor for services. Related to that are tools to setup scenarios and run them in the simulator.

The next burning issue would be to perform analysis on the services, there various techniques could be used. First the simulation and checking of requirements in the same manner employed in previous work [FI:Naeser99, FI:Jonsson99, FI:Jonsson00], but it would also be interesting to try parameterising the system.

The model in itself is undecidable but using various abstractions should make it possible to attack the problem piecemeal, and verify different aspect one at a time. It is possible that one can not verify the model or services in entirety, but for the debugging purposes it could prove powerful.

Finally one would have to look into resolution of conflict both offline and online. That would entail both designing useful methods as well as devising mechanisms for realising them in a structured manner in the IN standard model.

1.9 OVERVIEW

The thesis is organised as follows:

Part I: Getting Started
: This part contains introductions and general discussions as well as descriptions needed to understand the rest of the thesis.

Chapter 1: Introduction
: This is a introductory chapter containing general discussions such as the contribution and background to the thesis.

Chapter 2: Intelligent Networks
: This chapter contains a brief description of the Intelligent Networks standard and our use of it.

Chapter 3: Formalism
: Gives a description of the notation used in this thesis, as well as some definitions common to the later chapters.

Chapter 4: Tool
: This chapter describes the simulator we made for the model.

Part II: Framework

This part contains the model of the framework in which the services may be defined.

Chapter 5: Service switching function (SSF)

This chapter describes the service switching function that provides all call processing and control, thus embodying the normal switch.

Chapter 6: Basic call process (BCP)

Herein is modeled the Basic call process, a specialised SIB conceptualising the ordinary phone call.

Chapter 7: Service control function (SCF)

This chapter describes the service control function which commands all call control functions in the calling of an IN service.

Chapter 8: Service data function (SDF)

This chapter describes the service data function which is modeled a database.

Chapter 9: Specialized resources function (SRF)

This chapter describes the specialized resources function which handles all extra functionality needed by the services to communicate with the user.

Part III: Building Blocks

This part describes the building blocks out of which service are built, together with the framework in which to build them.

Chapter 10: Service independent building blocks (SIBs)

This chapter describes the service independent building blocks and the service instance.

Chapter 11: Global service logic (GSL)

This chapter describes the global service logic used to "glue" building blocks together into a service and the service unto the system.

Chapter 12: Hierarchical Service Independent Building Blocks (HSIBs)

In this chapter a number of hierarchical utility SIBs are defined.

Part IV: Services

In this part the services, service features are described.

Chapter 13: Introduction

This chapter describes the organisation of the this part of the thesis and defines a number of constants common to the services.

Chapter 14: Service features

This chapter models a number of service features.

Chapter 15: Services
This chapter models a number of services.

Part V: Appendices
This part contains the appendices.

Appendix A: Abbreviations
Here is gathered the acronyms used, what they stand for and where the concept is defined.

Appendix B: Syntax
This appendix contains a grammar describing the syntax used in definitions in this thesis.

Part VI: Bibliographies
The part contain the bibliographies group according to concern.

Bibliography A: Miscellaneous
The general bibliography.

Bibliography B: Intelligent Network
The bibliography of publications centered on Intelligent Network.

Bibliography C: Feature Interaction
The bibliography of publications centered on feature interaction.

Part VII: Indices
This part contains a number of indices of definitions.

Index A: Domain
This index contains the definitions of domains.

Index B: Function
This index contains the definitions of functions.

Index C: Constant
This index contains the definitions of constants.

Index D: External References
This index contains notes on where domains, functions and constants are referenced outside their chapter of definition.

CHAPTER 2

INTELLIGENT NETWORK

In this chapter we will shortly present the ITU-T IN standard and what in that standard we have focused on in our attempt to make a formal model of the standard.

2.1 THE STANDARD

The ITU-T[1] standard for Intelligent Networks (IN)[2] is an effort to present a coherent framework for the realisation of services in telecommunication systems. The IN standard proposes a way to divide functionality among components, e.g., to allow components from different suppliers to coexist. These components are called *Service Independent Building Blocks* (SIBs). The standard also makes it possible to define services as combinations of predefined functionalities, and thereby also paves way for the analysis of services and interactions of services prior to implementation.

More specifically the IN standard[3] is intended as an architectural model for the creation and provision of telecommunication services. The IN standard has two primary goals:

1. to enable specification and realisation of services manageable in a multi vendor environment.

2. to supply a coherent framework in which existing and new services can easily be specified an integrated with each other.

The IN standard is currently evolving through a sequence of steps. Each step defines a so-called Capability Set (CS), which extends the CS of the previous step. Presently, the two first capability sets, CS-1 and CS-2, has been defined. A major restriction of the CS-1 is that it only considers single point of call services, i.e., services which can be realised by specific actions of one of the parties in a call. Examples of services that are excluded are Multi-way calling and Meet-me conference. This is partly remedied in CS-2 which includes extensions for creation

[1] The telecommunication standardization sector of the International Telecommunication Union.
[2] See [IN:Duran92, IN:Faynberg97] for an overlook of IN.
[3] The ITU-T standard for IN is found in the Q.1200 series.

of service processes and communications amongst these. The CS-2 service process is also multi threaded although only one can be executing at any time.

The IN model consists of four layers (called *planes*), each representing services and network functions at a certain level of abstraction. The purpose of a plane is as usual to separate a piece of functionality from its implementation. Figure 2.1 shows the 3 uppermost planes.

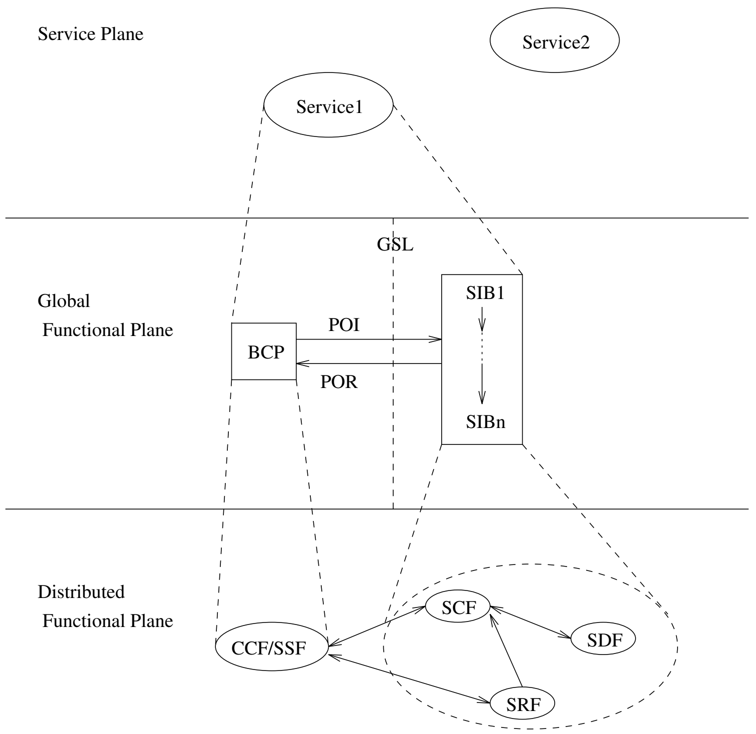


Figure 2.1: Intelligent network layers

Service plane: This plane represents the view of a user or manager. The elements of this plane are services, such as Call forwarding, Free phone and Mass calling. No attention is paid to how realisations of these services are to be structured. Attention can be paid to interactions and other relationships between services.

Global functional plane (GFP): This plane represents the view of a service provider. The purpose of this plane is to provide a level of abstraction at which services can be realised as the combination of more elementary functionalities, but still hiding the aspects that pertain to network distribution. The elements of this plane are so-called service independent building blocks (SIBs), each of which represents a certain functionality. E.g., the *Verify* SIB that verifies that user supplied information conforms to a specified format. Services are realised by connecting a suitable set of SIBs, using so-called Global service logic (GSL). In other words, SIBs can be thought of as "off the shelf" components to be used for assembling services.

Distributed functional plane (DFP): This plane illustrates a network operator's view of the system, represented as functional entities[4]. To give an example, the Service database function, which models systemwide databases.

The functional entities, onto which the SIBs and GSL of the global functional plane are mapped, are listed below:

CCAF: The *call control agent* function is the interface between user and network control functions.

CCF: The *call control function* provides call/connection processing and control.

SSF: The *service switching function* provides functions for interaction between the CCF and SCF.

SCF: The *service control function* commands call control functions in the processing of IN provided services.

SDF: The *service data function* contains customer and network data for real time use of the SCF.

SRF: The *specialized resources function* provides the specialized resources needed in the execution of IN services.

SMF: The *service management function* manages the service provisioning and deployment.

Physical plane: This plane models the physical network with different physical nodes and protocols. We will not address this plane since it is out of scope of our investigation.

2.2 LEVEL OF ABSTRACTION

In this section we will described how our fits in the structure of the IN standard.

2.2.1 Documents

As we have described in Section 1.4 we have focused in the Global Functional Plane described in [IN:Q.1213] for Capability Set-1 and the draft [IN:Q.1223] for Capability Set-2.

In order to understand the intentions behind the we have had to refer to the Distributed Functional Plane described in [IN:Q.1214], where amongst other things the suggested control flow patterns have been most instructive. We have however

[4]See Figure 2.2 for the relations between the functional entities

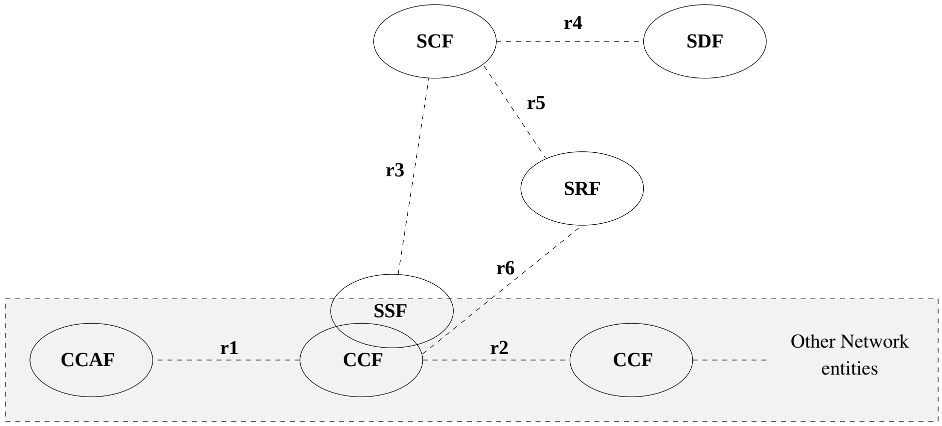


Figure 2.2: Figure 5-1 of [IN:Q.1214], depicting the relationships between functional entities

decided not to implement the control flows defined as we have intended to model the Global Functional Plane and only those parts of the lower level strictly necessary.

We have also studied other documents in the ITU-T Q.1200-series such as [IN:Q.1204, IN:Q.1211, IN:Q.1215, IN:Q.1218] where the [IN:Q.1204, IN:Q.1211] have a general and more abstract and the [IN:Q.1215, IN:Q.1218] details the Physical Plane and the Interfaces in the standard.

2.2.2 The service environment

Serving as an environment to the services are the normal call handling and the added functionality that supports the services. Examples of added functionality in normal call handling is mechanisms to start services and the mechanisms whereby the services can influence the normal call handling. Further there is mix of other support needed, such as databases, synthesized speech etc.

We have chosen to use the functional units identified, in the Distributed Functional Plane, as needed for normal call handling and service realisation and support. We have not modeled these in detail from the standard since we wish to have a more abstract we then presented in the standard.

Of the normal call handling only the `CCF` and `SSF` is modeled and they are treated as one unit, and will in the remainder of the thesis simply be referred to as `SSF`.

When dealing with the functional entities of the Distributed Functional Plane will not consider the distributed aspects, consequently there is only one representative of each type: `SSF`, `SCF`, `SDF` and `SRF` in our model.

2.2.3 Interpretation

The ITU-T standard is vague and seldom specific. It presents a framework supplying enough flexibility to be accepted by a large community of developers. This vagueness needed to ensure the flexibility is not desired when implementing a tool simulating the standard since they are not computable. So there have to be a trade-off between on one hand adherence to the standard and what a formal analysis can yield. We have therefore tried to minimise the amount of interpretation.

The level of abstraction we have chosen, where we for example fully model the BCSMs, seems to give us the granularity that we will need when we want to start analysing the behaviour of services. The alternative would have been to create a model describing the mechanics and behaviour the BCP SIB would have, but that would in essence have been the BCSM.

By choosing to design the model at the Global Functional and by modeling most of Distributed Functional Plane's entities, we create the relevance we deem necessary. There have however been some modifications as well as some additions made to the standard.

Modifications of the standard

There are some aspects of the standard that seem awkward, both from an implementors view, but even more so for our aim to later analyse the model. One example is Normal Billing. Since this functionality is integrated in POTS, it is also an integrated part of the standard. We have however chosen not to have it this way. There are several conceivable test and analyse scenarios which benefit from not having to consider the superfluous complexity added by Normal Billing so we have therefore chosen to implement Normal Billing as a service rather than a built in functionality of the model. Examination of the behaviour of the functionality shows that it behaves like a service, i.e. it is activated and executed in the manner of a service, so the modification should not pose a problem.

The notion of triggers set in detection points, used in Section 6.1.4 and defines points where the need for IN intervention may be detected, is simplified. Where the IN standard will differentiate between dynamically set detection points triggers and statically set triggers, and whether they will result in the invocation of a new service or just communicate with an existing one, we will simply group them all together. We have made this simplification since the added distinction brings nothing to our model.

Furthermore the details of errors, such as their type, have largely been omitted. We do not think that this is a severe restriction as error handling is hardly specified at all in the Q.1200-series.

Simplified Functional Entities

We have chosen not to fully model the SDF. The SDF is described in the standard a service database and a common interface to external database that need to be accessed. To fully model the SDF as described one will create a model of a general and rather powerful database manager, which is not the aim of this paper. Since the SDF's internal behaviour does not affect the overall workings of IN, and for most cases a rudimentary database will suffice we have modeled the bare essentials.

Another entity which we have greatly simplified is the SRF since much of its internal behaviour is of not interest to our model of the IN standard. The SRF is modeled as a black box that transfers information to and fro the user from the services. Should we however which to capture possible interactions in the user interface, such as different interpretations by services of dial events, we would have to model the SRF in greater detail.

CHAPTER 3

FORMALISM

In this chapter will present the overview of the model and notation needed to understand the following chapters, whereupon follows some generic definitions common the following process descriptions.

3.1 THE MODEL

The model of the IN standard will consist of a number of concurrent processes (automata) that execute independently of each other and communicate asynchronously.

First there is a fixed but unspecified number of terminals (phones) connected to the system via the `SSF`.

The IN system itself consists of four permanent processes `SSF`/`SCF`/`SDF`/`SRF` which model functionality existing in IN. Furthermore the `BCSM`s are dynamically created by the `SSF` on a per call basis. Normally there are two `BCSM`s per established connection. Finally there are the `Service instance processes` created by the `SCF` to execute services.

A picture of the processes in the model can be seen below in Figure 3.1 where processes that may communicate with each other are connected by arrows.

3.2 NOTATION

In this section we will describe the notation used to define the model processes in the following chapters. We will mostly use examples and no strict or complete definition of the syntax will be presented in this section although such a description may be found in Appendix B.

Each process will be informally introduced and our interpretation of the process will be presented. Then the process is formally defined by the following four parts:

Domains in which a set of domain equations that define the possible states of the process is given.

Constants where the constants used in the rules and functions are defined.

Functions where a set of utility functions are defined.

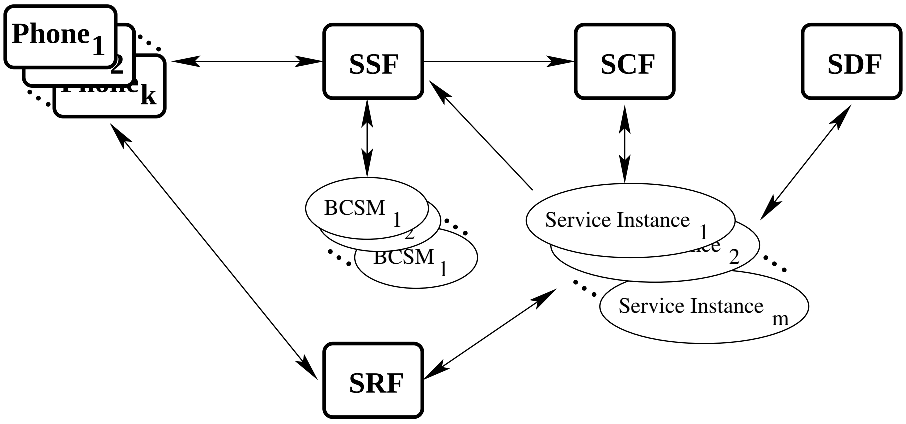


Figure 3.1: The Model

Rules where the rules defining the behaviour of the process are defined.

The are a few notational conventions used. Variables are written in "*lower case italics*" where the anonymous variable used in matchings for parts in which we have no interest is written as "_". Constant values, constants and function names are written in "sans-serif". The domain names begin with a capital letter and are in "*italics*", with the exception of parameterised domains which are written in "sans-serif".

When a domain, constant or function is used outside the chapter which it was defined in it is annotated with what chapter it was defined in. An example is the head function hd defined in the list part Shared Definitions Section 3.3, which is referred to as list: hd outside the Shared Definitions Section.

3.2.1 Domains

The domains defines the set of possible states of the process, expressed by a set of domain equations[1].

Each process has a state consisting of four parts: *mode*, *data*, *input queue* and *database*. The state is formally defined as a four–tuple made up of the four parts, e.g., the SSF's state in Equation 3.1.

$$SSFState \equiv SSFMode \times SSFData \times \mathsf{event}\colon EventQueue \times SSFDataBase \tag{3.1}$$

The *mode* is the control state of the process, and is normally defined by explicitly giving the set of control states. For instance the SSF's, mode which is defined in Equation 3.2, only has the control state running.

$$SSFMode \equiv \{\mathsf{running}\} \tag{3.2}$$

[1] To read more on the use of domains to describe data structures and their theory see [Stoy77].

The *data* constitutes the dynamic data of the process that changes over time. An example definition of a *data* part is that of the SSF found in Equation 3.3. The *SSFData* contains the current connections between phones and BCSMs in the *Current* part as well as the connections between BCSMs in the *Paths* part.

$$SSFData \equiv Current \times Paths \tag{3.3}$$

The *input queue* (event: *EventQueue*) which is defined in Section 3.3.6 later in this chapter contains the events that have been sent to this process which has not yet been processed. The variable σ is mostly used for this part of the state.

The *database* is the static data of the process, which is given initially and provide setup and definitions needed by the process. An example of a database is that of the SSF, found in Equation 3.4, which consists of information regarding what user is associated with a phone (*DefaultUser*) and what service settings the users have (*UserProfile*).

$$SSFDataBase \equiv DefaultUser \times UserProfile \tag{3.4}$$

The expressions are constructed from names of domains and set operations:

$\times$
: *products* as in Equation 3.3 where the SSF domain is defined as the product of *Current* and *Paths*;

$\rightarrow$
: *mappings* as in Equation 3.5 where the SSF domain *Paths* is defined as mappings between the identities of BCSMs;

$\mathbb{P}(X)$
: *powerset* as in Equation 3.6 where the SSF domain is defined as containing sets of BCSM identities;

$\mathsf{Domain}(\mathcal{X})$
: *parameterised* as in Equation 3.7 where the domain List is instantiated with $\{\mathbb{N}, \#, *\}$ to construct lists of dialable by the user.

  A special case is the parameterised domain id, which is only instantiated with singleton sets. In this case we have simplified the notation so that only the element of the singleton set is used in the instantiation instead of the singleton set. Examples of this is given in Equation 3.5 and Equation 3.6 where the domain Id is instantiated with bcsm to construct the domain of BCSM identities.

$$Paths \equiv \mathsf{Id}(\mathsf{bcsm}) \rightarrow \mathsf{Id}(\mathsf{bcsm}) \tag{3.5}$$

$$Active \equiv \mathbb{P}(\mathsf{Id}(\mathsf{bcsm})) \tag{3.6}$$

$$ServiceId \equiv \mathsf{List}(\{\mathbb{N}, \#, *\}) \cup \mathsf{services}\colon ServiceName \tag{3.7}$$

3.2.2 Constants

Constant definitions are written as "constant name $\stackrel{\text{def}}{=}$ expression" where the expression can be as simple as in the normal billing constant Equation 3.8 which consists of a one element set.

$$\mathsf{Normal\_Billing\_lookup} \stackrel{\text{def}}{=} \{\langle \mathsf{normal\_billing}, \langle \mathsf{Normal\_Billing}, \emptyset\rangle\rangle\} \tag{3.8}$$

A rather more complex expressions defining a constant may be found in the normal billing Equation 3.9 where a set union expression over the services constant Billing_DPs. Not only is the expression more complex but it also has as "**where**" clause that provide auxiliary definitions of shorthand nature, i.e., the defined names can be interchanged with their definition in verbatim.

$$\begin{array}{rcl} \mathsf{Normal\_Billing\_armed} & \stackrel{\text{def}}{=} & \bigcup_{dp \in \mathsf{services:Billing\_DPs}} \{\langle dp, template\rangle\} \\ \textbf{where} & & template = \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{normal\_billing}\rangle \end{array} \tag{3.9}$$

3.2.3 Functions

Functions are written basically on the same format with the addition of the arguments. An example of the simplest functions is the SIB function presented in Equation 3.10 below. The success function takes two arguments, the dynamic data of the service instance and the database and call the logical_end function with the additional argument success which indicates what logical_end should follow the SIB.

$$\mathsf{success}(data, db) \stackrel{\text{def}}{=} \mathsf{logical\_end}(\mathsf{success}, data, db) \tag{3.10}$$

A slightly more complex function is the SDF function external_reply presented in Equation 3.11 below. This function chooses the result depending on whether: the input queue argument (σ) is empty; or the sender of the first element in the input queue is the same as the expected *extern*; or otherwise if none of the above is true. The selection always chooses the first statement with a true guard, and the otherwise guard is always true.

This function also has auxiliary definition in a **where** clause which works i the same manner as, as mentioned above in Section 3.2.2, for constant definitions.

$$\begin{array}{rcl} \mathsf{external\_reply}(\sigma, extern) & \stackrel{\text{def}}{=} & \begin{cases} \bot & \mathsf{if\ list:null}(\sigma) \\ \mathsf{list:hd}(\sigma) & \mathsf{if}\ sender = extern \\ rest & \mathsf{otherwise} \end{cases} \\ \textbf{where} & & sender = \mathsf{event:sender}(\mathsf{list:hd}(\sigma)) \\ & & rest = \mathsf{external\_reply}(\mathsf{list:tl}(\sigma), extern) \end{array} \tag{3.11}$$

A further construct is used in the SSF function presented in the Equation 3.12 below. Here the second argument to the current_bcsms function is the tuple

$\langle current, paths\rangle$ which is matched against the actual arguments when calling the function.

The function application is well formed only if the formal parameters (the parameters used in the definition) match the actual (the values used in the call). Should the function call not be well formed, the result is $\perp$. The formal parameters may only contain constants, variables, tuple- and set-constructs formed out of these.

When matching there is a special anonymous variable which is written "_" and signifies that we are not interested in the value of this parameter. The anonymous variable can be used, e.g., in a selector for an abstract data type where we only match the parts we are selecting with named variables.

$$\mathsf{current\_bcsms}(phone, \langle current, paths\rangle) \stackrel{\text{def}}{=} current(phone) \tag{3.12}$$

The final way of constructing functions are through the explicit definition of them as sets, as shown in the BCSM function num1 which is presented in Equation 3.14 below.

$$\begin{aligned} \mathsf{num1} \stackrel{\text{def}}{=} \{ & \langle \mathsf{O\_Null}, 1\rangle, \langle \mathsf{Collect\_Info}, 3\rangle, \\ & \langle \mathsf{Analyse\_Info}, 2\rangle, \langle \mathsf{Routing\&Alerting}, 5\rangle, \\ & \langle \mathsf{O\_Active}, 3\rangle, \langle \mathsf{O\_Exception}, 1\rangle, \\ & \langle \mathsf{T\_Null}, 1\rangle, \langle \mathsf{Select\_Facility\&Present\_Call}, 4\rangle, \\ & \langle \mathsf{T\_Alerting}, 3\rangle, \langle \mathsf{T\_Active}, 3\rangle, \\ & \langle \mathsf{T\_Exception}, 1\rangle \} \end{aligned} \tag{3.13}$$

3.2.4 Rules

The rules in general have the format presented below in Equation 3.14 with a *Guard*, a pre-state, a post-state, an *Action* sequence and **where** clauses. The pre- and post-states are structured as 4-tuples with $Mode$, $Data$, $Queue$ and DB as described in Section 3.2.1. The *Action* sequence and **where** clauses are optional.

$$\frac{Guard}{\langle Mode, Data, Queue, DB\,\rangle \xrightarrow{Action} \langle Mode', Data', Queue', DB\,\rangle} \tag{3.14}$$

$$\begin{array}{rrcl} \textbf{where} & V & = & F(\ldots) \\ & \vdots & & \vdots \\ & V & = & F(\ldots) \end{array}$$

The *Guard* is a boolean expression which depends on the pre-state of the rule. When applying a rule, the actual pre-state of the process is matched against the pre-state tuple of the rule in the same manner as actual parameters are matched against formal parameters in functions as mentioned in Section 3.2.3. The rule is applicable only if the actual pre-state matches the pre-state of the rule and the *Guard* evaluates to true. The matching of the pre-state takes place before the evaluation of the *Guard* since if the actual pre-state does not match the rule's pre-state the *Guard* may be ill-formed.

If the rule is applicable and chosen, the actions are performed in order before the actual state of the process is set to the post-state. The definition of the post-state in the rule may contain expressions that depend on the pre-state and values returned by the actions. The expressions defining the actual parameters to an action may depend on the pre-state as well as values returned by actions executed before this action. The actions are:

send (*Message*, *To*)
: This action places the a tuple of $\langle Message,\ id \rangle$ at the end of the process *To*'s input queue, where *To* is the identity of one of the processes in the system and *id* is the identity of the sending process.

broadcast (*Set of Message/Receiver Pairs*)
: This action takes as argument a set of pairs on the form $\langle Message,\ To \rangle$ where for each pair the tuple $\langle Message,\ id \rangle$ is placed at the end of the process *To*'s input queue, where *To* is the identity of one of the processes in the system and *id* is the identity of the broadcasting process. Should the set be empty this action does nothing.

create(*Process Type, Initial State*)
: This action creates a new process of type *Process Type*, e.g., a bcsm, with the initial state *Initial State*.

Variable := create(*Process Type, Initial State*)
: This action creates a new process of type *Process Type*, e.g., bcsm, with the initial state *Initial State*, and returns the identity of the created process into the variable *Variable*.

The only rules that would be allowed to change the *DB* is the rules handling service deployment and subscriptions in a running system. The deployment would be handled by the SMF and the subscriptions either by the SMF or as services, but since we do not model this the *DB* will be unchanged.

The **where** clauses works in the same manner as for constant definitions in Section 3.2.2. The expressions in the clauses may depend on the pre-state and values returned by actions. The limitation is that if a variable depends on a value returned by an action it can only be used to construct the post-state and parameters for actions taking place after the action on which the variable depends.

Below in Equation 5.1 is an example rule from Chapter 5 defining SSF. The rule handles the case where the SSF has received a notification that an idle phone has gone offhook.

$$\text{(5.1)}\quad \frac{\begin{array}{l}\mathsf{event\colon notification}(\mathsf{phone}, \mathsf{offhook}, \mathsf{list\colon hd}(\sigma)) \wedge_{\perp} \\ \mathsf{list\colon null}(\mathsf{incoming}(phone, data))\end{array}}{\langle\, \mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{id:=\mathsf{create}(\mathsf{bcsm}, state)} \langle\, \mathsf{running},\ data',\ \mathsf{list\colon tl}(\sigma),\ db\,\rangle}$$

$$\begin{array}{rrcl} \textbf{where} & phone & = & \mathsf{event\colon sender}(\mathsf{list\colon hd}(\sigma)) \\ & state & = & \mathsf{bcsm} : \mathsf{new}(\mathsf{O\_Null}, \perp, phone, \mathsf{true}, db) \\ & data' & = & \mathsf{add\_active}(id, phone, data) \end{array}$$

The pre-state has the mode running and all the other parts are variables, so any actual state with the mode running will match. The guard states that the first element in the input queue (σ) must be a phone offhook event and that the set of incoming calls for that phone must be empty. A new BCSM is created and the identity of the created BCSM is entered into the set of active calls for that phone.

3.3 SHARED DEFINITIONS

In this section we are going to present a number of definitions that does not belong to a specific process definition. These utility functions and domains will, when used, not be annotated with their origin. This differs from the case when a function or domain belonging to a specific process is used outside its chapter. We will however annotate the functions and domains of list and queue sections since there would otherwise be a large number of name clashes.

3.3.1 Lifted domains

If A is a domain then $A_{\perp}$ is the domain where a smallest element $\perp$ has ben added to the domain A.

3.3.2 Select and Test

In this section we presented the general polymorphic select and test functions.

Tuple selectors

π_i is a family of selector functions that selects the i:th element from a tuple contained in a product domain.

$$\pi_m\colon Type_1 \times \ldots \times Type_m \times \ldots \times Type_n \rightarrow Type_m$$

Tests

The if, ifdef and ifundef are classical conditional functions. Where if is the general if that selects between argument two and three depending on whether argument one is true or false.

The ifdef and ifundef functions test whether if argument one is defined of undefined respectively. Undefined in this case means the value $\perp$ which is added a smallest element to lifted domains, and all other values are considered as defined.

$$\begin{array}{lcl} \mathsf{if} : (\mathbb{B} \times Type \times Type) & \rightarrow & Type \\ \mathsf{ifdef} : (Type_1 \times Type_2 \times Type_2) & \rightarrow & Type_2 \\ \mathsf{ifundef} : (Type_1 \times Type_2 \times Type_2) & \rightarrow & Type_2 \end{array}$$

$$\begin{array}{lcl} \mathsf{if}(test, alt_1, alt_2) & \stackrel{\text{def}}{=} & \begin{cases} alt_1 & \mathsf{if}\ test \\ alt_2 & \mathsf{otherwise} \end{cases} \\ \mathsf{ifdef}(x, alt_1, alt_2) & \stackrel{\text{def}}{=} & \mathsf{if}(x \neq \perp, alt_1, alt_2) \\ \mathsf{ifundef}(x, alt_1, alt_2) & \stackrel{\text{def}}{=} & \mathsf{if}(x = \perp, alt_1, alt_2) \end{array}$$

The $a \wedge_{\perp} b$ relation is an expansion of the common and ($\wedge$) relation in the sense that if a or b is defined it is identical to $\wedge$. The expansion is that it is well defined if either a or b is undefined, i.e., $\perp$, then the relation is false.

$$\wedge_{\perp} : \mathbb{B} \times \mathbb{B}_{\perp} \rightarrow \mathbb{B}$$

$$\wedge_{\perp}(a, b) \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \mathsf{if}\ a = \mathsf{true} \wedge b = \mathsf{true} \\ \mathsf{false} & \mathsf{otherwise} \end{cases}$$

3.3.3 Identities

This is a utility domain consisting of identifiers that are used as identities of processes. The notation of the parameterised domain id has been simplified since it is only instantiated with singleton sets the element is used in the instantiation instead of the singleton set.

$$Id \equiv [\mathsf{a} - \mathsf{zA} - \mathsf{Z0} - \mathsf{9\_-}]+$$

$$\mathsf{Id}(\mathcal{X}) \equiv \{\mathcal{X}\} \times \mathbb{N}$$

3.3.4 Data

This utility domain encapsulates what is considered as general data by the processes.

$$Data \equiv \mathbb{N} \cup \mathbb{R} \cup Id \cup Data \times Data \cup Data \rightarrow Data \cup \mathsf{scf} : ServiceId$$

3.3.5 List

The list are common programming language lists that are constructed out of tuples with the empty list denoted nil.

Some examples to illustrate how the lists are constructed:

one element list $\langle element, \mathsf{nil} \rangle$

two element list $\langle element_1, \langle element_2, \mathsf{nil} \rangle \rangle$

DOMAINS

$$\mathsf{List}(\mathcal{X}) \quad \equiv \quad \{\mathsf{nil}\} \cup \mathcal{X} \times \mathsf{List}(\mathcal{X})$$

FUNCTIONS

The null and member functions are predicates that tells whether a list is empty or an element is a member of a list respectively.

$$\mathsf{null}\colon \mathsf{List}(\mathcal{X}) \to \mathbb{B}$$

$$\mathsf{null}(list) \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathsf{true} & \text{if } list = \mathsf{nil} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{member}\colon (\mathcal{X} \times \mathsf{List}(\mathcal{X})) \to \mathbb{B}$$

$$\mathsf{member}(elt, list) \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathsf{member}(elt, tl) & \text{if } (list = \langle hd, tl\rangle) \wedge (hd \neq elt) \\ \mathsf{true} & \text{if } (list = \langle hd, \_\rangle) \wedge (hd = elt) \\ \mathsf{false} & \text{if } list = \mathsf{nil} \end{cases}$$

The hd and tl functions selects the head and the tail of the list, should the argument not be a tuple $\perp$ is returned.

$$\mathsf{hd}\colon \mathsf{List}(\mathcal{X}) \to \mathcal{X}$$

$$\mathsf{hd}(list) \quad \stackrel{\text{def}}{=} \quad \begin{cases} hd & \text{if } list = \langle hd, \_\rangle \\ \perp & \text{otherwise} \end{cases}$$

$$\mathsf{tl}\colon \mathsf{List}(\mathcal{X}) \to \mathsf{List}(\mathcal{X})$$

$$\mathsf{tl}(list) \quad \stackrel{\text{def}}{=} \quad \begin{cases} tl & \text{if } list = \langle \_, tl\rangle \\ \perp & \text{otherwise} \end{cases}$$

The add function adds an element last to the list, i.e., furthest in the structure. Adding an element first is done by constructing a tuple of the element and a the list: $\langle element, list\rangle$.

$$\mathsf{add}\colon (\mathcal{X} \times \mathsf{List}(\mathcal{X})) \to \mathsf{List}(\mathcal{X})$$

$$\mathsf{add}(elt, list) \quad \stackrel{\text{def}}{=} \quad \begin{cases} \langle hd, \mathsf{add}(elt, tl)\rangle & \text{if } list = \langle hd, tl\rangle \\ \langle elt, \mathsf{nil}\rangle & \text{if } list = \mathsf{nil} \\ \perp & \text{otherwise} \end{cases}$$

The remove function removes all occurrences of the element from the list, should no occurrences of the element be present the remove operation returns the original list.

$$\mathsf{remove}\colon (\mathcal{X} \times \mathsf{List}(\mathcal{X})) \rightarrow \mathsf{List}(\mathcal{X})$$

$$\mathsf{remove}(elt, list) \stackrel{\text{def}}{=} \begin{cases} \langle hd, \mathsf{remove}(elt, tl)\rangle & \mathsf{if}\ (list = \langle hd, tl\rangle) \wedge (hd \neq elt) \\ \mathsf{remove}(elt, tl) & \mathsf{if}\ (list = \langle hd, tl\rangle) \wedge (hd = e) \\ \mathsf{nil} & \mathsf{if}\ list = \mathsf{nil} \end{cases}$$

3.3.6 Event

Defined in this section are the domains and functions related to the messages passed to and fro the processes. The messages are called henceforth events.

DOMAINS

The Event Queue which is a part of all processes state is realised by a list of Event Queue Elements. The elements consists of pairs of Event, Originator Ids.

The events have three parts:

the type
identifies what class of event it is and what type of process it comes from;

the name
identifies exactly what event it is;

data
which dynamic is data supplied with the event and may contain anything.

The Event Template is just an event without the data part.

$$\begin{array}{lcl} \mathit{EventQueue} & \equiv & \mathsf{List}(\mathit{EventQueueElement}) \\ \mathit{EventQueueElement} & \equiv & \mathit{Event} \times \mathit{OriginatorId} \\ \mathit{Event} & \equiv & \mathit{EventType} \times \mathit{EventName} \times \mathit{Data} \\ \mathit{EventTemplate} & \equiv & \mathit{EventType} \times \mathit{EventName} \end{array}$$

The type of the event consists of the Originator and the Type Name. The Originator is the type of process that has sent the event and the Type Name is one of three:

notification
which is an event notifying that something has occurred, to this event no reply is required;

request
which is an event that request that the receiving process take some action, and responds with a **response** event;

response
which is an event responding to a **request** event.

$$\begin{array}{lcl} \mathit{EventType} & \equiv & \mathit{Originator} \times \mathit{EventTypeName} \\ \mathit{Originator} & \equiv & \{\mathsf{phone}, \mathsf{bcsm}, \mathsf{si}, \mathsf{ssf}, \mathsf{scf}, \mathsf{sdf}, \mathsf{srf}\} \\ \mathit{EventTypeName} & \equiv & \{\mathsf{notification}, \mathsf{request}, \mathsf{response}\} \end{array}$$

The Event Name is defined by the different process types.

$$\begin{array}{lcl} \mathit{EventName} & \equiv & \mathsf{ssf}\colon \mathit{SSFEventName} \cup \mathsf{bcsm}\colon \mathit{BCSMEventName} \cup \\ & & \mathsf{scf}\colon \mathit{SCFEventName} \cup \mathsf{si}\colon \mathit{SIEventName} \cup \\ & & \mathsf{sdf}\colon \mathit{SDFEventName} \cup \mathsf{srf}\colon \mathit{SRFEventName} \end{array}$$

The Originator Id is the identity of a sending process.

$$\mathit{OriginatorId} \equiv \mathsf{Id}(\mathsf{phone}) \cup \mathsf{Id}(\mathsf{bcsm}) \cup \mathsf{Id}(\mathsf{si}) \cup \{\mathsf{ssf}, \mathsf{scf}, \mathsf{sdf}, \mathsf{srf}\}$$

FUNCTIONS

The functions type, name, data and sender all selects that part of an Event Queue Element.

$$\mathsf{type}\colon EventQueueElement \rightarrow EventType$$

$$\mathsf{type}(\langle\langle type, \_, \_\rangle, \_\rangle) \stackrel{\text{def}}{=} type$$

$$\mathsf{name}\colon EventQueueElement \rightarrow EventName$$

$$\mathsf{name}(\langle\langle \_, name, \_\rangle, \_\rangle) \stackrel{\text{def}}{=} name$$

$$\mathsf{data}\colon EventQueueElement \rightarrow Data$$

$$\mathsf{data}(\langle\langle \_, \_, data\rangle, \_\rangle) \stackrel{\text{def}}{=} data$$

$$\mathsf{sender}\colon EventQueueElement \rightarrow OriginatorId$$

$$\mathsf{sender}(\langle\langle \_, \_, \_\rangle, originator\rangle) \stackrel{\text{def}}{=} originator$$

The $\mathsf{template}$ function constructs an Event Template out of an Event Queue Element by removing the sender information and data.

$$\mathsf{template}\colon EventQueueElement \rightarrow EventTemplate$$

$$\mathsf{template}(elt) \stackrel{\text{def}}{=} \langle \mathsf{type}(elt), \mathsf{name}(elt)\rangle$$

The $\mathsf{notification}$, $\mathsf{request}$ and $\mathsf{response}$ functions tests whether an Event Queue Element is a notification, request or response respectively. Further they check that the type of process sending the event was *orig* and that the Event Name was *name*.

The event_test function is a utility function used by the notification, request and response functions. The function tests whether the Event Template supplied is the same as that of the Event Queue Element.

$$*\colon Originator \times EventName \times EventQueueElement_{\perp} \rightarrow \mathbb{B}$$

$$\begin{aligned}
\mathsf{notification}(orig, name, elt) &\stackrel{\text{def}}{=} \mathsf{event_test}(\langle\langle orig, \mathsf{notification}\rangle, name\rangle, elt)\\
\mathsf{request}(orig, name, elt) &\stackrel{\text{def}}{=} \mathsf{event_test}(\langle\langle orig, \mathsf{request}\rangle, name\rangle, elt)\\
\mathsf{response}(orig, name, elt) &\stackrel{\text{def}}{=} \mathsf{event_test}(\langle\langle orig, \mathsf{response}\rangle, name\rangle, elt)
\end{aligned}$$

$$\mathsf{event\_test}\colon EventTemplate \times EventQueueElement_{\perp} \rightarrow \mathbb{B}$$

$$\mathsf{event\_test}(template, elt) \stackrel{\text{def}}{=} \begin{cases} \mathsf{false} & \mathsf{if}\ elt = \perp \\ \mathsf{true} & \mathsf{if\ template}(elt) = template \\ \mathsf{false} & \mathsf{otherwise} \end{cases}$$

CHAPTER 4

TOOL

We implemented a simulator and service creation tool in ERLANG using the library connection to Tk handle the graphics.

The service creation tool is so far only a prototype in which one can graphically design the services as was described in the design example in section Section1.5.1, with the exception that connections for the dynamic has to be given textually.

4.1 SIMULATOR

We started writing the simulator when we had already decided on formalism and had made the first draft of the model. Initially we simply implemented the functional entities in ERLANG, making a prototype to see whether ERLANG suitable as language.

The choice of language was suitable, however direct implementation made changes difficult when the model evolved. To make the handling of changes in the model easier we decided to state the model as an ERLANG term and interpret the term. We then realised that to get a truly flexible system we would break down the abstract data types and other structures, such as conditionals, that are used to construct the model into modules.

Luckily the ERLANG runtime system has a good support for modules, with the application functions that accept both the module and function as argument.

We now had a system composed of the core engine performing a very simple top-loop, as shown simplified in Figure 4.1, calling on the module of the functional entity it is simulating to get the actual details.

```
schedule := process type:schedule()
rules := process type:rules()

loop(schedule, rules, state)
 rule := schedule(schedule, rules, state);
 state := evaluate(rule, state);
end loop
```

Figure 4.1: The engine core of the simulator

The terms describing the functional entities are built up using constructors defined in the different modules. Say for example that we want to build a list composed of the one element, which is the head of the variable `format`. Then we would write: `list:mklist([list:head(env:var(format))])` where the notation `list:mklist` signifies that `mklist` is a functions in module `list`.

The constructed term contains the information that is a list constructed out of the subterm which is a function call, but also contain information as to what module constructed the term. When the term is interpreted the responsibility of this term falls on the module that constructed the term.

In this way we have gained great flexibility, since in most cases when we introduce a new construct of abstract data type we do not have to change any of the existing ones. Should we on the other hand find that an existing module should be incorrect or inadequate we can just change or augment it without changes the other modules.

When adding an analysis tool one could in the same manner break down the problem, for example for calculating the weakest precondition.

4.2 MODULES

In this section is presented a list of the modules that make up the simulator with some short comments on their role:

- Engine
  The modules that directly support the engine.
  - engine
    The core engine module.
  - selection
    The different strategies for order of selecting rules. The strategies can be hierarchally ordered, say first choose stochastically from the first set and if none of them are applicable choose stochastically from the next set.
    - ordered
      Just enumerate the rules either as they are ordered in the file or following some given order.
    - stochastic
      Select among a set of rules according to some stochastic distribution.
  - library
    - state
      Functions for handling the four parts of the state, selectors constructors etc., the format of the states are described in Section 3.2.1.

  * env
    An environment used to save values of variables used by rules and functions, the values of the functions are lazily evaluated.

- Libraries
  General libraries for data structures and functions etc.
  - arith
    Arithmetic functions.
  - base
    Constructors of base types: integers, identifiers etc.
  - control
    Conditional constructs.
  - function
    Constructs for functions, their format is described in Section 3.2.3.
  - lambda
    Constructs for nameless lambda functions.
  - list
    Functions for the list data structure. The list format and the utility functions are described in Section 3.3.5.
  - logic
    Logical operators.
  - rule
    The constructors, selectors etc. for the process rules. Their format is described in Section 3.2.4.
  - set
    Functions for the set data structure.
  - tuple
    Functions for the tuple data structure.
- IN
  The model specific modules.
  - Automata
    The processes used to model the different functional entities of the IN standard, each with its scheduling, functions, constants and rules.
    * SSF
      The SERVICE SWITCHING FUNCTION described in Chapter 5.
    * SCF
      The SERVICE CONTROL FUNCTION described in Chapter 7.
    * SDF
      The SERVICE DATA FUNCTION described in Chapter 8.

  * SRF
    The SPECIALIZED RESOURCES FUNCTION described in Chapter 9.
  * BCSM
    The BASIC CALL STATE MODEL described in Section 6.1.4.
  * SI
    The SERVICE INSTANCE described in Section 10.2.
  * phone
    A model of the users as perceived by the system, it is intended to use with stochastic choices.
- Services
  The definitions of the services.
  * AAB
    The AUTOMATIC ALTERNATIVE BILLING service described in Section 15.2.
  * Call Forwading
    The Call Forwading service described in Section 15.3.
  * Call Forwading On Busy Line
    The Call forwarding On Busy Line service described in Section 15.4.
- Features
  The definitions of the service features.
  * Normal Billing
    The Normal Billing service feature described in Section 14.2.
  * Normal Charging
    The Normal Charging service feature described in Section 14.3.
  * Call Forwading
    The Call forward service feature described in Section 14.4.
- HSIBs
  The definitions of the hierarchical SIBs.
  * Null
    The Null hierarchical SIB described in Section 12.3.
  * Get_N
    The Get_N hierarchical SIB described in Section 12.4.
- util
  Model specific functions used by several modules.
  * side_effect
    The side effects, or actions as we have described them, are described in Section 3.2.4
  * event
    The event format and the utility functions are described in Section 3.3.6.

4.3 ERLANG

The ERLANG language is a functional process oriented language.

The Language is functional with simple syntax a few (compared to other functional languages, e.g. Common Lisp) language constructs and built in functions. The additional functions needed, besides the fundamental functions are supplied in library modules. The notion of process oriented entails that process are as basic a notion in the language as that of functions, the language supports lightweight processes.

The processes in ERLANG communicates via asynchronous message passing where the messages are placed an input queue, with the messages from the same sender in sending order otherwise unspecified. The messages may be retrieved random access using matching.

The ERLANG language is compiled into byte code for an abstract machine[1] that is emulated. The code is maintained at module level each module residing is a separate file and complied individually and loaded on demand unless the programme explicitly loads a module into the run time system. The per module treatment goes even further a: new version of a module may be loaded into the system while it is executing and consequent calls to functions in that module will be made to the new version.

The language run time system supports distribution both as several operating system processes on the same machine or on several machines. The distribution is stated explicitly but the main portion of a programme may be coded without taking distribution into account.

The simple syntax and the fact that ERLANG is untyped together with the per module compilation and loading makes ERLANG and excellent language for rapid prototyping, even of distributed systems.

The ERLANG language seemed almost tailored to our needs[2] with the properties of a good prototyping language with support for lightweight processes, and unlike JAVA they have well defined scheduling. The language also has support for distribution over several machines makes it all the more interesting since then one can run larger scenarios and get real complications that entails in a simulation.

[1] [Armstrong96]

[2] The ERLANG was not tailored specifically to our needs but well to the needs of telecommunication systems being originally developed by the Ericsson Computer Science Laboratory for that expressed purpose and used in among other projects as the main programming language in the new Ericsson back bone ATM switches AXD 301[Blau99] and by One 2 One in implementing IN services[IN:Hinde00].

Part II

Framework

CHAPTER 5

SSF - SERVICE SWITCHING FUNCTION

In this chapter we will describe the service switching function, first informally its role and the scope of our model. Thereafter follows in Section 5.2 a more rigorous annotated description.

5.1 DESCRIPTION

The SSF is the intermediary between the normal call control of the CCF and the IN-capabilities of the SCF. It should not only manage communication between the CCF and SCF but also enable the CCF to detect the need of IN-capabilities. According to [IN:Q.1204, p.3]:

> It
>
> a) extends the logic of the CCF to include recognition of service control triggers and to interact with the SCF;
> b) manages signalling between the CCF and SCF;
> c) modifies call/connection processing functions (in the CCF) as required to process requests for IN provided service usage under the control of the SCF;
> d) is managed, updated and/or otherwise by a SMF.

Since we are interested in the IN aspects of call processing we will only model the SSF in the CCF/SSF pair. However, we still need the normal call connection handling and control which will be modeled as integral to the SSF rather than by explicitly modeling the CCF.

Consequently, the role of our model of the SSF is to handle the global aspects of normal call connection and control, together with the communication with the service logic of the SCF. The local handling of individual calls, such as, the detection of "service control triggers" mentioned in the quote above, will be delegated to the BCSMs described in Chapter 6. The SSF will only have the global view of the system, with connection paths and the association between BCSMs and the terminal units.

Examples of the communication during the connection setup is shown in Figures 5.1 and 5.2. In Figure 5.1 the message sequence chart shows a connection being set up between two phones, the steps of the setup being:

- Phone1 sends an offhook message to the SSF, whereupon an originating BCSM is created since Phone1 is not engaged in any call, rule 5.1.
- The originating BCSM sends an authorised message to the SSF notifying it that the call can proceed, this message is relayed to Phone1, rule 5.8.
- Phone1 sends a request to be connected to Phone2, the SSF relays the request to the originating BCSM, rule 5.4.
- The originating BCSM sends a connect request to the SSF which creates a terminating BCSM, rule 5.6.
- The terminating BCSM sends a selectfacility request to the SSF which responds with a facility available message since Phone2 is free, rule 5.9.
- Phone2 sends an offhook message to the SSF, the SSF relays the offhook message to the terminating BCSM, rule 5.2.
- The terminating BCSM sends a connection message to the SSF, which the SSF relays to Phone2 and sends as an offhook message to the originating BCSM, rule 5.10.
- The originating BCSM sends a connection message to the SSF, which the SSF relays to Phone1, rule 5.10.

In Figure 5.2 the message sequence chart shows a the attempt to establish a connection between two phones when called phone is busy, the steps of the setup attempt being:

- Phone1 sends an offhook message to the SSF, whereupon an originating BCSM is created since Phone1 is not engaged in any call, rule 5.1.
- The originating BCSM sends an authorised message to the SSF notifying it that the call can proceed, this message is relayed to Phone1, rule 5.8.
- Phone1 sends a request to be connected to Phone2, the SSF relays the request to the originating BCSM, rule 5.4.
- The originating BCSM sends a connect request to the SSF which creates a terminating BCSM, rule 5.6.
- The terminating BCSM sends a selectfacility request to the SSF which responds with a busy message since Phone2 is busy, rule 5.9.

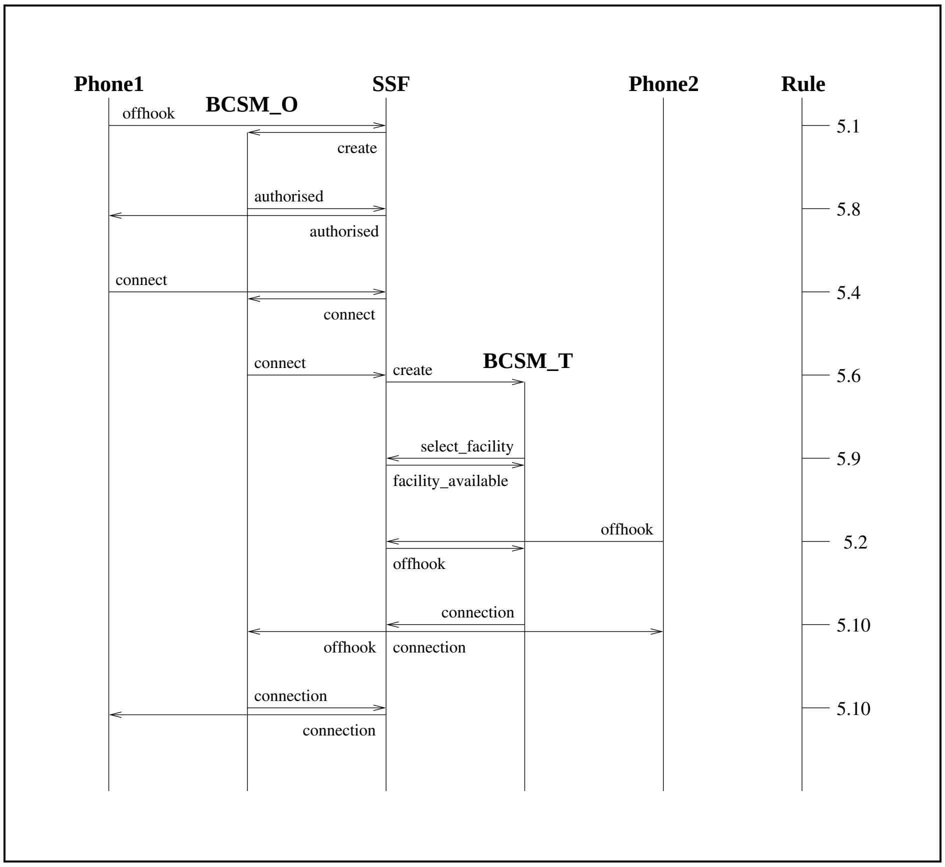


Figure 5.1: Example 1: communication pattern of the SSF in call setup

- The terminating BCSM sends a busy message to the SSF and terminates, the SSF relays the message to the originating BCSM, rule 5.11.
- The originating BCSM sends a busy message to the SSF and terminates, the SSF relays the message Phone1, rule 5.11.

5.2 DOMAINS

The state of the SSF contains the global information for both call handling, such as associations between BCSMs, and the subscription of services.

$$SSFState \equiv SSFMode \times SSFData \times \mathsf{event}\colon EventQueue \times SSFDataBase$$

We model the SSF without any control state, hence the *SSF_Mode*, of the SSF is always the constant running. The dynamic data of the SSF contains the status

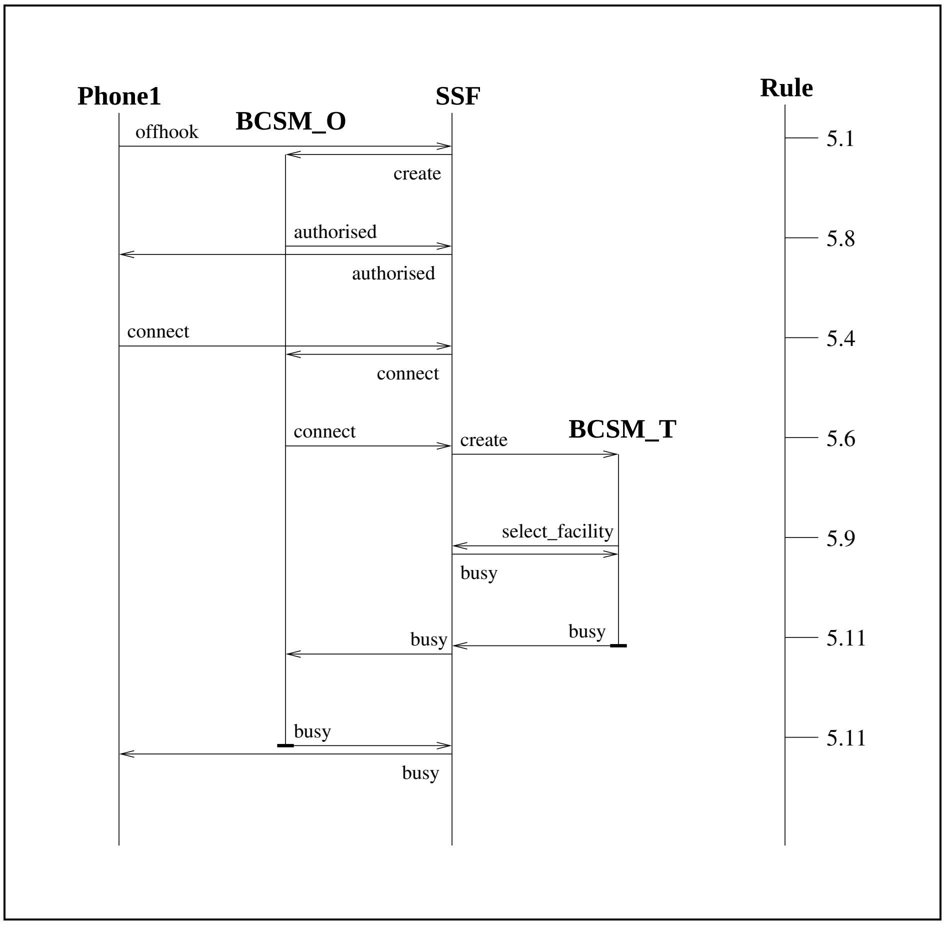


Figure 5.2: Example 2: communication pattern of the SSF when busy condition occurs in call setup

of the connections and the paths of connections through the system. The static database contains information saying what terminal unit a user is associated with, as well as the individual service profiles of each user.

$$\begin{array}{lcl} \mathit{SSFMode} & \equiv & \{\mathsf{running}\} \\ \mathit{SSFData} & \equiv & \mathit{Current} \times \mathit{Paths} \\ \mathit{SSFDataBase} & \equiv & \mathit{DefaultUser} \times \mathit{UserProfile} \end{array}$$

The *Current* part of the dynamic data records the connections of each terminal unit. It maps a terminal unit to three groups of BCSMs:

Active – the set of BCSMs now active, i.e., having a voice-path to the terminal unit;

Passive – the set of BCSMs that have been put "on hold";

Incoming – a queue of BCSMs trying to the terminal unit.

The elements of *Active* and *Passive* are simply sets of BCSMs, whereas *Incoming* BCSMs are queued up in the order they arrived. The elements of *Paths* are partial injective mappings on the set of BCSM id:s, defining the single connections in the paths of the system.

$$\begin{array}{lcl} \mathit{Current} & \equiv & \mathsf{Id(phone)} \to \mathit{Active} \times \mathit{Passive} \times \mathit{Incoming} \\ \mathit{Paths} & \equiv & \mathsf{Id(bcsm)} \to \mathsf{Id(bcsm)} \\ \\ \mathit{Active} & \equiv & \mathbb{P}\,(\mathsf{Id(bcsm)}) \\ \mathit{Passive} & \equiv & \mathbb{P}\,(\mathsf{Id(bcsm)}) \\ \mathit{Incoming} & \equiv & \mathsf{Queue(Id(bcsm))} \end{array}$$

The domain *DefaultUser* contains a mapping from each terminal unit to a particular subscriber, whereas the *UserProfile* consists of active and subscribed services and how to charge his/her call. *Armed* decides what events triggers services. The *Present* functions decide, given a *Position*, what events should be issued by subscribed services to this BCSM. The *Lookup* functions will, given a service id, return the name of a service together with statically set data to the service, such as forwarding number for a forwarding service. The *Accounting* is a list of account and percentage pairs that decide what accounts should carry the charge of the call and what fraction of the charge they should carry. Typically the *Accounting* list will only be single pair of one account and 1, i.e., that this account will carry the whole charge.

$$\begin{array}{lcl} \mathit{DefaultUser} & \equiv & \mathsf{Id(phone)} \to \mathsf{Id(user)} \\ \mathit{UserProfile} & \equiv & \mathsf{Id(user)} \to \mathsf{bcsm}\!: \mathit{Armed} \times \mathsf{bcsm}\!: \mathit{Present} \times \\ & & \qquad\qquad \mathsf{bcsm}\!: \mathit{Lookup} \times \mathsf{bcsm}\!: \mathit{Accounting} \times \\ & & \qquad\qquad \mathit{Activated} \times \mathit{Subscribed} \\ \mathit{Activated} & \equiv & \mathbb{P}\,(\mathsf{services}\!: \mathit{ServiceName}) \\ \mathit{Subscribed} & \equiv & \mathbb{P}\,(\mathsf{services}\!: \mathit{ServiceName}) \end{array}$$

The *SSFEventName* domain are the names of those events accepted or issued by the SSF.

$$\begin{array}{lcl} \mathit{SSFEventName} & \equiv & \{\mathsf{answer}, \mathsf{authorised}, \mathsf{busy}, \mathsf{connect}, \mathsf{connection}, \\ & & \ \mathsf{facility\_available}, \mathsf{initiate\_call}, \mathsf{no\_answer}, \mathsf{offhook}, \\ & & \ \mathsf{onhook}, \mathsf{report}, \mathsf{request}, \mathsf{select\_facility}, \mathsf{select\_fail}, \\ & & \ \mathsf{service}, \mathsf{terminate}\} \end{array}$$

5.3 FUNCTIONS

The profile function retrieves the user profile of the user associated with the *phone* from the *database*, which is a pair $\langle default\_user, profile \rangle$.

$$\mathsf{profile}(phone, \langle default\_user, profile \rangle) \stackrel{\text{def}}{=} profile(default\_user(phone))$$

The bcsm:new function, defined in the BCP Chapter 6, initiates a new BCSM. The arguments to the functions are:

pic the point in call,

party what party it is connected to (id of the other BCSM),

phone The phone which it is associated with and used to determine the user profile,

connected if the BCSM is connected to the phone,

db the database in which the user profile is found.

$$\mathsf{bcsm} : \mathsf{new}(pic, party, phone, connected, db)$$

The current_bcsms function retrieves the current BCSMs connected to the *phone* according to the data and update_current returns new data which has the current connections updated. The data consists of the pair $\langle current, paths \rangle$.

$$\begin{array}{lcl} \mathsf{current\_bcsms}(phone, \langle current, paths \rangle) & \stackrel{\text{def}}{=} & current(phone) \\ \mathsf{update\_current}(update, phone, \langle current, paths \rangle) & \stackrel{\text{def}}{=} & \langle current \oplus \\ & & \langle phone, update \rangle, \\ & & paths \rangle \end{array}$$

The functions active and incoming are selectors that given a *phone* selects from the current BCSMs the active set and incoming queue respectively.

$$\begin{array}{lcl} \mathsf{active}(phone, data) & \stackrel{\text{def}}{=} & \pi_1\,(\mathsf{current\_bcsms}(phone, data)) \\ \mathsf{incoming}(phone, data) & \stackrel{\text{def}}{=} & \pi_3\,(\mathsf{current\_bcsms}(phone, data)) \end{array}$$

The utility functions enqueue_incoming, dequeue_incoming, clear_active_passive and add_active updates the current connections for *phone* in *data*. The function add_active adds the *bcsm_id* to the set of active BCSMs for *phone*. The functions enqueue_incoming and dequeue_incoming, add and remove an element from the incoming queue of *phone*, and when a dequeue is performed the dequeued element is added to the active set for the *phone*. The function clear_active_passive empties the active and passive sets of *phone*.

$$
\begin{array}{rcl}
\mathsf{add\_active}(bcsm\_id, phone, data) & \stackrel{\text{def}}{=} & \mathsf{update\_current}(\langle a', p, i\rangle, phone, data) \\
\textbf{where} & & \langle a, p, i\rangle = \mathsf{current\_bcsms}(phone, data) \\
& & a' = a \cup \{bcsm\_id\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{enqueue\_incoming}(bcsm\_id, phone, data) & \stackrel{\text{def}}{=} & \mathsf{update\_current}(\langle a, p, i'\rangle, phone, data) \\
\textbf{where} & & \langle a, p, i\rangle = \mathsf{current\_bcsms}(phone, data) \\
& & i' = \mathsf{list\colon add}(bcsm\_id, i)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{dequeue\_incoming}(phone, data) & \stackrel{\text{def}}{=} & \mathsf{update\_current}(\langle a', p, i'\rangle, phone, data) \\
\textbf{where} & & \langle a, p, i\rangle = \mathsf{current\_bcsms}(phone, data) \\
& & a' = a \cup \{\mathsf{list\colon hd}(i)\} \\
& & i' = \mathsf{list\colon tl}(i)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{clear\_active\_passive}(phone, data) & \stackrel{\text{def}}{=} & \mathsf{update\_current}(\langle \emptyset, \emptyset, i'\rangle, phone, data) \\
\textbf{where} & & i' = \mathsf{list\colon add}(\mathsf{incoming}(phone, data), \mathsf{nil})
\end{array}
$$

The purge function removes all occurrences of the *bcsm_id* from the current connections of *phone*.

$$
\begin{array}{rcl}
\mathsf{purge}(bcsm\_id, phone, data) & \stackrel{\text{def}}{=} & \mathsf{ifdef}(phone, \\
& & \quad \mathsf{update\_current}(\{\langle a', p', i' \rangle, phone, data), \\
& & \quad data) \\
\textbf{where} & & \langle a, p, i \rangle = \mathsf{current\_bcsms}(phone, data) \\
& & a' = a \setminus \{bcsm\_id\} \\
& & p' = p \setminus \{bcsm\_id\} \\
& & i' = \mathsf{list\colon remove}(bcsm\_id, i)
\end{array}
$$

The paths selection functions looks up successor (next) or predecessor (previous) of a *bcsm_id* in a connection path. The add_path function adds a link between two BCSMs (*path*) to the path. The remove_path function takes a *path* between two BCSMs and removes it from the set of paths in the *data*.

$$
\begin{array}{lcl}
\mathsf{next\_bcsm}(bcsm\_id, \langle current, paths\rangle) & \stackrel{\text{def}}{=} & paths\,(bcsm\_id) \\
\mathsf{previous\_bcsm}(bcsm\_id, \langle current, paths\rangle) & \stackrel{\text{def}}{=} & paths^{-1}\,(bcsm\_id) \\
\mathsf{add\_path}(path, \langle current, paths\rangle) & \stackrel{\text{def}}{=} & \langle current, \{path\} \cup paths\rangle \\
\mathsf{remove\_path}(path, \langle current, paths\rangle) & \stackrel{\text{def}}{=} & \langle current, paths \setminus \{path\}\rangle
\end{array}
$$

The notify functions return a set of messages to be sent, for each element in a set. The first function notify returns an event message for all entities found in the *list*, and the second function notify_phone returns the event if the *phone* is defined and the *bcsm_id* is one of the active BCSMs of the *phone*.

$$
\begin{array}{lcl}
\mathsf{notify}(list, name) & \stackrel{\text{def}}{=} & \bigcup_{elt \in list} \mathsf{ifdef}(elt, \\
 & & \qquad\qquad \{\langle\langle\langle \mathsf{ssf}, \mathsf{notification}\rangle, name, \bot\rangle, elt\rangle\}, \\
 & & \qquad\qquad \emptyset) \\
\mathsf{notify\_phone}(phone, & \stackrel{\text{def}}{=} & \mathsf{if}((phone \neq \bot) \wedge_\bot (bcsm\_id \in \mathsf{active}(phone, data)), \\
\qquad data, & & \quad \{\langle\langle\langle \mathsf{ssf}, \mathsf{notification}\rangle, name, \bot\rangle, phone\rangle\}, \\
\qquad bcsm\_id, & & \quad \emptyset) \\
\qquad name) & &
\end{array}
$$

5.4 RULES

PHONE EVENT RULES

The following rule is applicable when an offhook event has been received from a phone and there are no incoming calls for that phone. A new originating BCSM is initiated and is added to the set of "Active" BCSMs for this phone.

$$
(5.1)\ \frac{\begin{array}{l}\mathsf{event\colon notification(phone, offhook, list\colon hd}(\sigma)) \wedge_\bot \\ \mathsf{list\colon null(incoming}(phone, data))\end{array}}{\langle\, \mathsf{running},\, data,\, \sigma,\, db \,\rangle \xrightarrow{id := \mathsf{create}(\mathsf{bcsm}, state)} \langle\, \mathsf{running},\, data',\, \mathsf{list\colon tl}(\sigma),\, db \,\rangle}
$$

$$
\begin{array}{llcl}
\textbf{where} & phone & = & \mathsf{event\colon sender(list\colon hd}(\sigma)) \\
 & state & = & \mathsf{bcsm : new(O\_Null}, \bot, phone, \mathsf{true}, db) \\
 & data' & = & \mathsf{add\_active}(id, phone, data)
\end{array}
$$

The following rule is applicable when an offhook event has been received from a phone and there are incoming calls for that phone. The first of the *Incoming* BCSMs is chosen and is added to the *Active* set as well as notified that an offhook has occured (a message is sent to that BCSM).

$$
(5.2)\ \frac{\begin{array}{l}\mathsf{event\colon notification(phone, offhook, list\colon hd}(\sigma)) \wedge_\bot \\ \neg\mathsf{list\colon null(incoming}(phone, data))\end{array}}{\langle\, \mathsf{running},\, data,\, \sigma,\, db \,\rangle \xrightarrow{\mathsf{send}\ (m,\, bcsm\_id)} \langle\, \mathsf{running},\, data',\, \mathsf{list\colon tl}(\sigma),\, db \,\rangle}
$$

$$
\begin{array}{llcl}
\textbf{where} & phone & = & \mathsf{event\colon sender(list\colon hd}(\sigma)) \\
 & data' & = & \mathsf{dequeue\_incoming}(phone, data) \\
 & bcsm\_id & = & \mathsf{list\colon hd(incoming}(phone, data)) \\
 & m & = & \langle\langle \mathsf{ssf}, \mathsf{notification}\rangle, \mathsf{offhook}, phone\rangle
\end{array}
$$

The following rule is applicable when an onhook event has been sent from a phone. Then the SSF sends an onhook message to all BCSMs in the *Active* and *Passive* sets of this phone. The *Active* and *Passive* sets for this phone are then emptied.

$$\frac{\textsf{event: notification}(\textsf{phone}, \textsf{onhook}, \textsf{list: hd}(\sigma))}{\langle\, \textsf{running},\, data,\, \sigma,\, db\,\rangle \xrightarrow{\textsf{broadcast}\,(m)} \langle\, \textsf{running},\, data',\, \textsf{list: tl}(\sigma),\, db\,\rangle} \tag{5.3}$$

$$\begin{array}{lrcl} \textbf{where} & phone & = & \textsf{event: sender}(\textsf{list: hd}(\sigma)) \\ & \langle a, p, i\rangle & = & \textsf{current\_bcsms}(phone, data) \\ & data' & = & \textsf{clear\_active\_passive}(phone, data) \\ & m & = & \bigcup_{to \in a \cup p}\{\langle\langle\langle \textsf{ssf}, \textsf{notification}\rangle, \textsf{onhook}, \perp\rangle, to\rangle\} \end{array}$$

The following rule is applicable when a connect message or service request from the phone has been sent to the SSF. The message is relayed to all active BCSMs for this phone.

$$\frac{\begin{array}{l}\textsf{event: notification}(\textsf{phone}, \textsf{connect}, \textsf{list: hd}(\sigma)) \vee \\ \textsf{event: notification}(\textsf{phone}, \textsf{service}, \textsf{list: hd}(\sigma))\end{array}}{\langle\, \textsf{running},\, data,\, \sigma,\, db\,\rangle \xrightarrow{\textsf{broadcast}\,(m)} \langle\, \textsf{running},\, data,\, \textsf{list: tl}(\sigma),\, db\,\rangle} \tag{5.4}$$

$$\begin{array}{lrcl} \textbf{where} & m & = & \bigcup_{bcsm\_id \in active}\{\langle\langle\langle \textsf{ssf}, \textsf{request}\rangle, \\ & & & \qquad\qquad \textsf{event: name}(\textsf{list: hd}(\sigma)) \\ & & & \qquad\qquad \textsf{event: data}(\textsf{list: hd}(\sigma))\rangle, \\ & & & \qquad\quad bcsm\_id\rangle\} \\ & active & = & \textsf{active}(\textsf{event: sender}(\textsf{list: hd}(\sigma)), data) \end{array}$$

CONNECTION HANDLING RULES

The following rule is applicable when a termination event has been sent from a BCSM. For each path to and from this BCSM the path is removed and the BCSMs at the other end are notified of the termination. The terminated BCSM is removed from the *Active*, *Passive* sets and from the *Incoming* queue.

$$(5.5)\quad \frac{\textsf{event: notification}(\textsf{bcsm}, \textsf{terminate}, \textsf{list: hd}(\sigma))}{\langle\, \textsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\textsf{broadcast}\ (m_1 \cup m_2)} \langle\, \textsf{running},\ data^3,\ \textsf{list: tl}(\sigma),\ db\,\rangle}$$

$$\begin{array}{lrcl}
\textbf{where} & from & = & \textsf{event: sender}(\textsf{list: hd}(\sigma)) \\
& phone & = & \textsf{event: data}(\textsf{list: hd}(\sigma)) \\
& prev & = & \textsf{previous_bcsm}(from, data) \\
& next & = & \textsf{next_bcsm}(from, data) \\
& data^1 & = & \textsf{purge}(from, phone, data) \\
& data^2 & = & \textsf{remove_path}(\langle prev, from\rangle, data^1) \\
& data^3 & = & \textsf{remove_path}(\langle from, next\rangle, data^2) \\
& m_1 & = & \textsf{notify}(\{prev, next\}, \textsf{onhook}) \\
& m_2 & = & \textsf{notify_phone}(phone, data, from, \textsf{terminate})
\end{array}$$

The following rule is applicable when a connection request from a BCSM has been sent. A new terminating BCSM is initiated and a path is set up with the BCSM requesting a connection.

$$(5.6)\quad \frac{\textsf{event: request}(\textsf{bcsm}, \textsf{connect}, \textsf{list: hd}(\sigma))}{\langle\, \textsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{id:=\textsf{create}(\textsf{bcsm}, state)} \langle\, \textsf{running},\ data',\ \textsf{list: tl}(\sigma),\ db\,\rangle}$$

$$\begin{array}{lrcl}
\textbf{where} & phone & = & \textsf{event: data}(\textsf{list: hd}(\sigma)) \\
& from & = & \textsf{event: sender}(\textsf{list: hd}(\sigma)) \\
& state & = & \textsf{bcsm : new}(\textsf{T_Null}, from, phone, \textsf{true}, db) \\
& data' & = & \textsf{add_path}(\langle from, id\rangle, data)
\end{array}$$

The following rule is applicable when a connection request from a BCSM has been sent and the facility for connection is not available. This case corresponds to a failure or congestion in the switch. A message is sent to the requesting BCSM informing it of the failure.

$$(5.7)\quad \frac{\textsf{event: request}(\textsf{bcsm}, \textsf{connect}, \textsf{list: hd}(\sigma))}{\langle\, \textsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\textsf{send}\ (m, from)} \langle\, \textsf{running},\ data,\ \textsf{list: tl}(\sigma),\ db\,\rangle}$$

$$\begin{array}{lrcl}
\textbf{where} & phone & = & \textsf{event: data}(\textsf{list: hd}(\sigma)) \\
& from & = & \textsf{event: sender}(\textsf{list: hd}(\sigma)) \\
& m & = & \langle\langle \textsf{ssf}, \textsf{notification}\rangle, \textsf{select_fail}, phone\rangle
\end{array}$$

The following rule is applicable when an authorisation notification has been sent from a BCSM. This signifies that the tests that a user may begin a call has been successful. An example of a test is when the user subscribes to teenline, then the service will during a specified part of the day require that the user supplies a password in order to place an outgoing call. The phone connected to the BCSM, if any, is notified.

$$\frac{\mathsf{event{:}\,notification(bcsm, authorised, list{:}\,hd(}\sigma\mathsf{))}}{\langle\, \mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{broadcast}\ (m)} \langle\, \mathsf{running},\ data,\ \mathsf{list{:}\,tl}(\sigma),\ db\,\rangle} \tag{5.8}$$

$$\begin{array}{rrcl} \textbf{where} & phone & = & \mathsf{event{:}\,data(list{:}\,hd(}\sigma\mathsf{))} \\ & m & = & \mathsf{notify}(\{phone\}, \mathsf{authorised}) \end{array}$$

The following rule is applicable when a select_facility request has been sent from a BCSM. If the phone is not busy (i.e. the *Active*, *Passive* sets and the *Incoming* queue are empty) a facility_available notification. If on the other hand the phone is busy the BCSM is sent a busy notification. Finally the BCSM is entered into the phone's *Incoming* queue.

$$\frac{\mathsf{event{:}\,request(bcsm, select\_facility, list{:}\,hd(}\sigma\mathsf{))}}{\langle\, \mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\, from)} \langle\, \mathsf{running},\ data',\ \mathsf{list{:}\,tl}(\sigma),\ db\,\rangle} \tag{5.9}$$

$$\begin{array}{rrcl} \textbf{where} & phone & = & \mathsf{event{:}\,data(list{:}\,hd(}\sigma\mathsf{))} \\ & from & = & \mathsf{event{:}\,sender(list{:}\,hd(}\sigma\mathsf{))} \\ & \langle a,p,i\rangle & = & \mathsf{current\_bcsms}(phone, data) \\ & facility & = & \begin{cases} \mathsf{facility\_available} & \mathsf{if}\ (a \cup p = \emptyset) \wedge \mathsf{list{:}\,null}(i) \\ \mathsf{busy} & \mathsf{otherwise} \end{cases} \\ & data' & = & \mathsf{enqueue\_incoming}(from, phone, data) \\ & m & = & \langle\langle \mathsf{ssf}, \mathsf{notification}\rangle, facility, \perp\rangle \end{array}$$

The following rule is applicable when a connection notification has been sent from a BCSM. If the BCSM is connected to a phone it is notified. An offhook notification is propagated backwards in the path.

$$\frac{\mathsf{event{:}\,notification(bcsm, connection, list{:}\,hd(}\sigma\mathsf{))}}{\langle\, \mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{broadcast}\ (m_1 \cup\, m_2)} \langle\, \mathsf{running},\ data,\ \mathsf{list{:}\,tl}(\sigma),\ db\,\rangle} \tag{5.10}$$

$$\begin{array}{rrcl} \textbf{where} & from & = & \mathsf{event{:}\,sender(list{:}\,hd(}\sigma\mathsf{))} \\ & phone & = & \mathsf{event{:}\,data(list{:}\,hd(}\sigma\mathsf{))} \\ & prev & = & \mathsf{previous\_bcsm}(from, data) \\ & m_1 & = & \mathsf{notify}(\{prev\}, \mathsf{offhook}) \\ & m_2 & = & \mathsf{notify\_phone}(phone, data, from, \mathsf{connection}) \end{array}$$

The following rule is applicable when a busy notification has been sent from a BCSM. The BCSM is removed from the *Active*, *Passive* sets and the *Incoming* queue. The BCSM is removed from the path and a busy notification is propagated backwards in the path. If any phone is connected it is also notified.

$$(5.11)\quad \frac{\textsf{event: notification}(\textsf{bcsm}, \textsf{busy}, \textsf{list: hd}(\sigma))}{\langle\, \textsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\textsf{broadcast}\ (m_1 \cup m_2)} \langle\, \textsf{running},\ data'',\ \textsf{list: tl}(\sigma),\ db\,\rangle}$$

$$\begin{array}{rrcl}
\textbf{where} & from & = & \textsf{event: sender}(\textsf{list: hd}(\sigma)) \\
 & phone & = & \textsf{event: data}(\textsf{list: hd}(\sigma)) \\
 & prev & = & \textsf{previous_bcsm}(from, data) \\
 & data' & = & \textsf{purge}(from, phone, data) \\
 & data'' & = & \textsf{remove_path}(\langle prev, from\rangle, data') \\
 & m_1 & = & \textsf{notify}(\{prev\}, \textsf{busy}) \\
 & m_2 & = & \textsf{notify_phone}(phone, data, from, \textsf{busy})
\end{array}$$

The following rule is applicable when a no_answer notification has been sent from a BCSM. The BCSM is removed from the *Active*, *Passive* sets and the *Incoming* queue. The BCSM is removed from the path and a no_answer notification is propagated backwards and forwards in the path. If any phone is connected it is also notified.

$$(5.12)\quad \frac{\textsf{event: notification}(\textsf{bcsm}, \textsf{no\_answer}, \textsf{list: hd}(\sigma))}{\langle\, \textsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\textsf{broadcast}\ (msgs)} \langle\, \textsf{running},\ data^3,\ \textsf{list: tl}(\sigma),\ db\,\rangle}$$

$$\begin{array}{rrcl}
\textbf{where} & from & = & \textsf{event: sender}(\textsf{list: hd}(\sigma)) \\
 & phone & = & \textsf{event: data}(\textsf{list: hd}(\sigma)) \\
 & prev & = & \textsf{previous_bcsm}(from, data) \\
 & next & = & \textsf{next_bcsm}(from, data) \\
 & data^1 & = & \textsf{purge}(from, phone, data) \\
 & data^2 & = & \textsf{remove_path}(\langle prev, from\rangle, data^1) \\
 & data^3 & = & \textsf{remove_path}(\langle from, next\rangle, data^2) \\
 & m_1 & = & \textsf{notify}(\{prev\}, \textsf{no_answer}) \\
 & m_2 & = & \textsf{notify}(\{next\}, \textsf{onhook}) \\
 & m_3 & = & \textsf{notify_phone}(phone, data, from, \textsf{no_answer}) \\
 & msgs & = & m_1 \cup m_2 \cup m_3
\end{array}$$

SCF COMMUNICATION RULES

The following rule is applicable when a service request event or report request has been sent from a BCSM. The request is relayed to the SCF.

$$\frac{\begin{array}{c}\mathsf{event{:}\,request(bcsm, request, list{:}\,hd(}\sigma\mathsf{))} \vee \\ \mathsf{event{:}\,request(bcsm, report, list{:}\,hd(}\sigma\mathsf{))}\end{array}}{\langle\, \mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\, \mathsf{scf})} \langle\, \mathsf{running},\ data,\ \mathsf{list{:}\,tl}(\sigma),\ db\,\rangle} \tag{5.13}$$

$$\textbf{where} \quad m \;=\; \begin{array}{l}\langle\langle \mathsf{ssf}, \mathsf{request}\rangle \\ \ \mathsf{event{:}\,name(list{:}\,hd(}\sigma\mathsf{))} \\ \ \langle \mathsf{event{:}\,data(list{:}\,hd(}\sigma\mathsf{)), event{:}\,sender(list{:}\,hd(}\sigma\mathsf{))}\rangle\rangle\end{array}$$

The following rule is applicable when a service report response event has been sent from the SCF. The report is relayed to the BCSM that requested the service.

$$\frac{\mathsf{event{:}\,response(scf, report, list{:}\,hd(}\sigma\mathsf{))}}{\langle\, \mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\ (report,\, to)} \langle\, \mathsf{running},\ data,\ \mathsf{list{:}\,tl}(\sigma),\ db\,\rangle} \tag{5.14}$$

$$\textbf{where} \quad \langle to, report\rangle \;=\; \mathsf{event{:}\,data(list{:}\,hd(}\sigma\mathsf{))}$$

The following rule is applicable when an initiate call point of return[1] has been sent from the SCF. A call is initiated by initiating a new originating BCSM, which if the call according to the message is to be active, replaces the *Current Active* of the phone and swaps the active to passive. If the call will not be active the requesting BCSM will be removed from the phone's *Current*. Finally a initiate call point of return message is sent to the requesting BCSM.

$$\frac{\textsf{event: request}(\textsf{scf}, \textsf{initiate\_call}, \textsf{list: hd}(\sigma))}{\langle\, \textsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow[]{\substack{id:=\textsf{create}(\textsf{bcsm}, state);\\ \textsf{broadcast}\ (m)}} \langle\, \textsf{running},\ data'',\ \textsf{list: tl}(\sigma),\ db\,\rangle} \tag{5.15}$$

$$
\begin{array}{rrcl}
\textbf{where} & \langle to, ph, type, & & \\
 & armed, nr\rangle & = & \textsf{event: data}(\textsf{list: hd}(\sigma)) \\
 & \langle a, p, i\rangle & = & \textsf{current\_bcms}(ph, data) \\
 & connected & = & \textsf{if}(type = \textsf{active}, \textsf{true}, \textsf{false}) \\
 & state & = & \textsf{bcsm : new}(\textsf{O\_Null}, to, ph, connected, db) \\
 & data' & = & \textsf{if}(type = \textsf{active} \\
 & & & \quad \textsf{update\_current}(\{\langle\{id\}, a \cup p, i\rangle\}, ph, data), \\
 & & & \quad \textsf{purge}(to, ph, data)) \\
 & data'' & = & \textsf{add\_path}(\langle to, id\rangle, data') \\
 & m & = & \{\langle\langle\langle \textsf{scf}, \textsf{por}\rangle, \textsf{initiate\_call}, \langle type, armed\rangle\rangle, to\rangle, \\
 & & & \ \langle nr, id\rangle\}
\end{array}
$$

[1] For explanation of point of return see following Chapter 6.

Chapter 6

BCP - Basic Call Process

In this chapter we will describe the Basic Call Process SIB and how this will be modeled using the Basic Call State Model. Thereafter a formal model is given of the Basic Call State Model.

6.1 DESCRIPTION

The BCP SIB shall model the functionality of the normal call capabilities together with the ability to decide when IN functionality is needed. This model is rather coarse since the intentions are according to [IN:Q.1213, p.29–30]:

> The basic call process (BCP) has been defined in 4.0/Q.1203 as a specialized SIB which provides the basic call capabilities.
>
> These capabilities enable the use of GSL as well as other SIBs to completely describe CS-1 services and service features.
>
> It is not necessary or intended to fully model the Basic Call Process on the Global Functional Plane in CS-1, but rather to insure that the functionality of the Basic Call Process that is required on the Global Functional Plane in conjunction with SIBs be available to fully support CS-1 Service/Service features.

6.1.1 BCP

The Basic call process at the global functional plane is described only by how it interacts with the Intelligent Network service in the form of sequences of SIBs. This is accomplished by describing the interface between the BCP and the GSL which is used to construct the SIB sequences. The interface consists of Points of initiation (POI) and Points of return (POR).

6.1.2 POI - Point Of Initiation

The Points of initiation are the points in the BCP's execution where the Basic call process may detect that Intelligent Network functionality is required; whereupon the BCP is suspended and control is handed over to the GSL which then initiates a sequence of SIBs. Below is a list of the POIs.

- **Call originated**
  This POI identifies that the user has made a service request without yet specifying a destination address.

- **Address Collected**
  This POI identifies that the address input has been received from the user.

- **Address analysed**
  This POI identifies that the address input has been analysed to determine characteristics of the address.

- **Prepared to complete call**
  This POI identifies that the network as prepared to attempt completion of the call to the terminating party.

- **Busy**
  This POI identifies that the call is destined for a user who is currently busy.

- **No Answer**
  This POI identifies that the call has been offered to a user who has not answered.

- **Call acceptance**
  This POI identifies that the call is active but the connection between the calling and called parties is not established.

- **Active state**
  This POI identifies that the call is active and the condition between the calling and called parties is established.

- **End of call**
  This POI identifies that a call party has disconnected.

6.1.3 POR - Point Of Return

The Points of return denote the different types of return that may occur from a sequence of SIBs by way of the GSL to the BCP SIB. Below follows a list of the PORs.

- **Continue with existing data**
  This POR identifies that the BCP should continue call processing with no modification.

- **Proceed with new data**
  This POR identifies that the BCP should proceed with call processing with only a data modification.

- ***Handle as transit***
  This POR identifies that the BCP should treat the call as if it had just arrived.

- **Clear call**
  This POR identifies that the BCP should clear the call.

- ***Enable call party handling***
  This POR identifies that the BCP should perform functions to enable call control for individual parties.

- **Initiate call**
  This POR identifies that a new call should be initiated. This may be independent of an existing call, or may be in the context of an existing call.

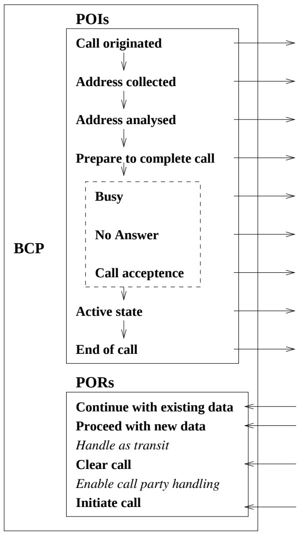


Figure 6.1: Basic Call Process SIB

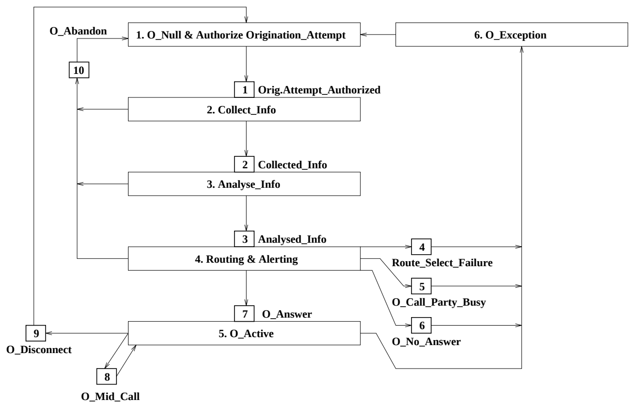
Of the above mentioned PORs the PORs Handle as transit and Enable call party handling have not been modeled. The reason for not modeling these PORs was that they were not needed in the services we have modeled so far. Further the portent of Enable call party handling PORs is not entirely clear to the author.

The rather limited view of normal call handling presented by the BCP SIB above is not sufficient when trying to model the SIBs[1]. The normal call handling has to be considered at greater detail and we have used as a starting point the Basic call state model (BCSM)found in [IN:Q.1214] admirably suited for this purpose.

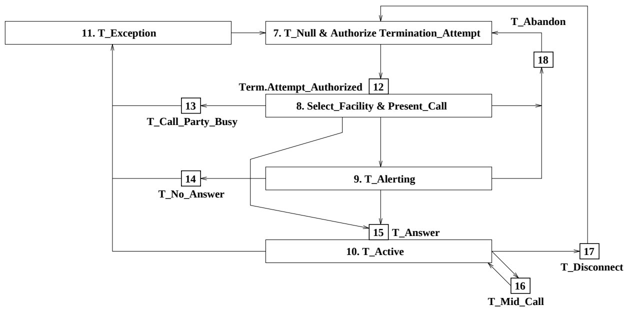
[1] A graphical depiction of the BCP SIB is shown in Figure 6.1.

6.1.4 BCSM - Basic Call State Model

The Basic Call State Model is partitioned into an originating and terminating BCSM depending on what part of the call that is modeled.[2]



(a) Originating BCSM



(b) Terminating BCSM

Figure 6.2: Figures 4-3 and 4-4 of [IN:Q.1214], originating/terminating BCSM for CS-1

The BCSM consists of Points in call (PICs) which reflects normal call handling. Detection points (DPs) are points where the need for IN functionality can be identified and the basic call handling suspended while the call control is transfered to the IN functionality in question. The PICs are represented as rectangular boxes in the figure and the DPs as smaller squares.

The DPs are related to the POIs in that one or several DPs in the originating and terminating BCSM “implement” one POI. The POR on the other hand can

[2]A picture of the originating and terminating BCSM are given in Figure 6.2.

not be so simply identified in the BCSM, since they determine not only whether normal call handling is resumed at the same point as where it was suspended or elsewhere, but also if any changes should be made to the data associated with the call. In Table 6.1 the POIs are related to the DPs. The information in the table is also shown in the Figure 6.3.

Point of initiation	Originating BCSM DP	Terminating BCSM DP
Call originated	Orig.Attempt_Authorized	
Address collected	Collected_Info	
Address analysed	Analysed_Info	
Prepare to complete call		Term.Attempt_Authorized
Busy	O_Call_Party_Busy	T_Call_Party_Busy
No answer	O_No_Answer	T_No_Answer
Call acceptance	O_Answer	T_Answer
Active state	O_Mid_Call	T_Mid_Call
End of call	O_Disconnect	T_Disconnect

Table 6.1: Relating the BCP SIB POIs to BCSM DPs

6.2 DOMAINS

The BCSM state contains the call control information of one side of a normal call, together with service information. The service information is what services are subscribed and what triggers to listen for.

$$\begin{aligned} \mathit{BCSMState} \;\equiv\;& \mathit{BCSMMode} \times \mathit{BCSMData} \times \\ & \mathsf{event}\colon \mathit{EventQueue} \times \mathit{BCSMDataBase} \end{aligned}$$

The mode of the BCSM is either normal call handling (pic), checking whether a service has been triggered (dp) or suspended while the call is controlled by the SCF (stalled). The dynamic data of the BCSM consists of a position in the control graph of the BCSM, a queue of incoming events, and the party to which this BCSM is connected. The BCSMs database consists of setting for services and charging together with the accumulated cost and the phone the BCSM is connected to.

$$\begin{aligned} \mathit{BCSMMode} \;&\equiv\; \{\mathsf{pic}, \mathsf{dp}, \mathsf{stalled}\} \\ \mathit{BCSMData} \;&\equiv\; \mathit{Position} \times \mathit{PriorityQueue} \times \mathit{Party} \\ \mathit{BCSMDataBase} \;&\equiv\; \mathit{Armed} \times \mathit{Present} \times \mathit{Lookup} \times \\ & \quad\; \mathit{Accounting} \times \mathit{Cost} \times \mathsf{Id}(\mathsf{phone})_{\perp} \end{aligned}$$

The *Position* is the name of the point in call or detection point at which the state is. The *PriorityQueue* is the priority sorted queue of untreated events. The

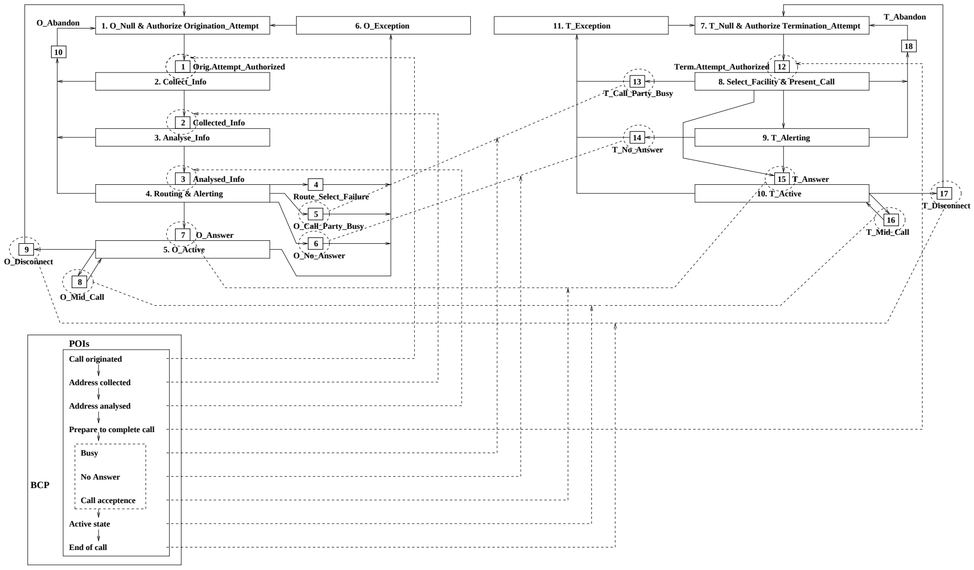


Figure 6.3: Relating the BCP SIB POIs to BCSM DPs

Party is the identity the of the BCSM that this BCSM is connected to if any. If the BCSM is not yet connected to another BCSM the party is $\perp$.

$$
\begin{array}{lcl}
Position & \equiv & O\_PICs \cup T\_PICs \cup O\_DPs \cup T\_DPs \\
PriorityQueue & \equiv & \mathsf{event}\colon EventQueue \\
Party & \equiv & \mathsf{Id}(\mathsf{bcsm})_{\perp}
\end{array}
$$

What is triggered on is decided on by *Armed*, which contains pairs of a *Position* and an *EventTemplate* (type and name of the event). The *Present* functions decide, given a *Position*, what events should be issued by subscribed services to this BCSM. The *Lookup* functions will given a service id return the name of a service together with statically set data to the service, such as forwarding number for a forwarding service.

$$
\begin{array}{lcl}
Armed & \equiv & Position \times \mathsf{event}\colon EventTemplate \\
Present & \equiv & Position \to \mathbb{P}\,(\mathsf{event}\colon EventQueueElement) \\
Lookup & \equiv & \mathsf{scf}\colon ServiceId \to (\mathsf{services}\colon ServiceName \times \mathsf{scf}\colon SSD)
\end{array}
$$

Accounting is a list of ID:s and the percentage they should bear of the *Cost*. The *Cost* is a set of real-valued cost-tokens identifying where the cost arose (*CostType*).

$$
\begin{array}{lcl}
Accounting & \equiv & \mathsf{List}(\mathsf{Id}(\mathsf{account}) \times \mathbb{R}) \\
Cost & \equiv & \mathbb{P}\,(CostType \times \mathbb{R}) \\
CostType & \equiv & \{\mathsf{normal}\} \cup \mathsf{services}\colon ServiceName
\end{array}
$$

The $BCSMEventName$ domain are the names of those events accepted or issued by the BCSM.

$$BCSMEventName \equiv \{\mathsf{answer}, \mathsf{authorised}, \mathsf{busy}, \mathsf{clear\_call}, \mathsf{connect}, \mathsf{connection}, \mathsf{continue\_with\_existing\_data}, \mathsf{exception}, \mathsf{facility\_available}, \mathsf{initiate\_call}, \mathsf{no\_answer}, \mathsf{offhook}, \mathsf{onhook}, \mathsf{proceed\_with\_new\_data}, \mathsf{report}, \mathsf{request}, \mathsf{select\_facility}, \mathsf{select\_fail}, \mathsf{service}, \mathsf{terminate}\}$$

The O_PICsm, T_PICs, O_DPs and T_DPs are positions in the control graph of the BCSM.

$$O\_PICs \equiv \{\mathsf{O\_Null}, \mathsf{Collect\_info}, \mathsf{Analyse\_info}, \mathsf{Routing\&Alerting}, \mathsf{O\_Active}, \mathsf{O\_Exception}\}$$

$$T\_PICs \equiv \{\mathsf{T\_Null}, \mathsf{Select\_Facility\&Present\_Call}, \mathsf{T\_Alerting}, \mathsf{T\_Active}, \mathsf{T\_Exception}\}$$

$$O\_DPs \equiv \{\mathsf{Orig.Attempt\_Authorized}, \mathsf{Collected\_Info}, \mathsf{Analysed\_Info}, \mathsf{Route\_Select\_Failure}, \mathsf{O\_Call\_Party\_Busy}, \mathsf{O\_No\_Answer}, \mathsf{O\_Answer}, \mathsf{O\_Mid\_Call}, \mathsf{O\_Disconnect}, \mathsf{O\_Abandon}, \mathsf{Collected\_Service\_Info}\}$$

$$T\_DPs \equiv \{\mathsf{Term.Attempt\_Authorized}, \mathsf{T\_Call\_Party\_Busy}, \mathsf{T\_No\_Answer}, \mathsf{T\_Answer}, \mathsf{T\_Mid\_Call}, \mathsf{T\_Disconnect}, \mathsf{T\_Abandon}\}$$

6.3 CONSTANTS

The constant standard_profile used in this chapter is defined in Chapter 13. The standard_profile defines the service setup of the standard user.

6.4 FUNCTIONS

These are utility functions that selects a part of the dynamic data or database.

$$\begin{aligned}
\mathsf{pos}(data) &\stackrel{\text{def}}{=} \pi_1\,(data)\\
\mathsf{queue}(data) &\stackrel{\text{def}}{=} \pi_2\,(data)\\
\mathsf{party}(data) &\stackrel{\text{def}}{=} \pi_3\,(data)
\end{aligned}$$

$$\begin{aligned}
\mathsf{armed}(db) &\stackrel{\text{def}}{=} \pi_1\,(db)\\
\mathsf{present}(db) &\stackrel{\text{def}}{=} \pi_2\,(db)
\end{aligned}$$

The `new` function initiates a `BCSM` state with the the point in call (pic) and what *party* its connected to. The phone (ph) which it is associated to is used to determine the user profile. The phone can be *connected* or, if it is just a link in a connection chain, not connected. Finally the database (db) is used to lookup the profile in.

$$\begin{array}{rcll}
\mathsf{new}(pic, party, ph, & & & \\
connected, db) & \stackrel{\text{def}}{=} & \langle \mathsf{pic}, & \\
 & & \langle pic, \mathsf{nil}, party\rangle, & \\
 & & \mathsf{nil}, & \\
 & & \langle armed^2, present^2, lookup^2, accounting, \emptyset, cph\rangle\rangle & \\
 & \textbf{where} & \langle armed, & \\
 & & \ present, & \\
 & & \ lookup\rangle & = \mathsf{services: standard_profile}\\
 & & \langle armed^1, present^1, & \\
 & & \ lookup^1, & \\
 & & \ accounting, _, _\rangle & = \mathsf{ssf: profile}(ph, db)\\
 & & armed^2 & = armed \oplus armed^1\\
 & & present^2 & = present \oplus present^1\\
 & & lookup^2 & = lookup \oplus lookup^1\\
 & & cph & = \mathsf{if}(connected, ph, \bot)
\end{array}$$

The arcs function describes the graphs of the `BCSM` annotated with *guards* and sent messages found in the Figure 6.4, 6.5. The functions takes a *Position* and the number of the outgoing arc returns guard message to be sent and the resulting *Position*.

arcs: $(Position \times \mathbb{N} \to (EventTemplate_\epsilon \times EventTemplate_\epsilon \times Position)$

arcs =

{

⟨O_Null, 1, ⟨ϵ, ϵ, Orig.Attempt_Authorized⟩⟩,
⟨Collect_Info, 1, ⟨⟨⟨ssf, request⟩, connect⟩, ϵ, Collected_Info⟩⟩,
⟨Collect_Info, 2, ⟨⟨⟨ssf, notification⟩, onhook⟩, ϵ, O_Abandon⟩⟩,
⟨Collect_Info, 3, ⟨⟨⟨ssf, request⟩, service⟩, ϵ, Collected_Service_Info⟩⟩,
⟨Analyse_Info, 1, ⟨ϵ, ϵ, Analysed_Info⟩⟩,
⟨Analyse_Info, 2, ⟨⟨⟨ssf, notification⟩, onhook⟩, ϵ, O_Abandon⟩⟩,
⟨Routing&Alerting, 1, ⟨⟨⟨ssf, notification⟩, offhook⟩, ϵ, O_Answer⟩⟩,
⟨Routing&Alerting, 2, ⟨⟨⟨ssf, notification⟩, select_fail⟩, ϵ, Route_Select_Failure⟩⟩,
⟨Routing&Alerting, 3, ⟨⟨⟨ssf, notification⟩, busy⟩, ϵ, O_Call_Party_Busy⟩⟩,
⟨Routing&Alerting, 4, ⟨⟨⟨ssf, notification⟩, no_answer⟩, ϵ, O_No_Answer⟩⟩,
⟨Routing&Alerting, 5, ⟨⟨⟨ssf, notification⟩, onhook⟩, ϵ, O_Abandon⟩⟩,
⟨O_Active, 1, ⟨⟨⟨ssf, request⟩, service⟩, ϵ, O_Mid_Call⟩⟩,
⟨O_Active, 2, ⟨⟨⟨ssf, notification⟩, onhook⟩, ϵ, O_Disconnect⟩⟩,
⟨O_Active, 3, ⟨⟨⟨ssf, notification⟩, exception⟩, ⟨⟨bcsm, notification⟩, terminate⟩, O_Exception⟩⟩,
⟨O_Exception, 1, ⟨ϵ, ϵ, O_Null⟩⟩,
⟨Orig.Attempt_Authorized, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, authorised⟩, Collect_Info⟩⟩,
⟨Collected_Service_Info, 1, ⟨ϵ, ϵ, Analyse_Info⟩⟩,
⟨Collected_Info, 1, ⟨ϵ, ϵ, Analyse_Info⟩⟩,
⟨Analysed_Info, 1, ⟨ϵ, ⟨⟨bcsm, request⟩, connect⟩, Routing&Alerting⟩⟩,
⟨Route_Select_Failure, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, terminate⟩, O_Exception⟩⟩,
⟨O_Call_Party_Busy, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, busy⟩, O_Exception⟩⟩,
⟨O_No_Answer, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, no_answer⟩, O_Exception⟩⟩,
⟨O_Answer, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, connection⟩, O_Active⟩⟩,
⟨O_Mid_Call, 1, ⟨ϵ, ϵ, O_Active⟩⟩,
⟨O_Disconnect, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, terminate⟩, O_Null⟩⟩,
⟨O_Abandon, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, terminate⟩, O_Null⟩⟩,
⟨T_Null, 1, ⟨ϵ, ϵ, Term.Attempt_Authorized⟩⟩,
⟨Select_Facility&Present_Call, 1, ⟨⟨⟨ssf, notification⟩, facility_available⟩, ϵ, T_Term_Seized⟩⟩,
⟨Select_Facility&Present_Call, 2, ⟨⟨⟨ssf, notification⟩, offhook⟩, ϵ, T_Answer⟩⟩,
⟨Select_Facility&Present_Call, 3, ⟨⟨⟨ssf, notification⟩, busy⟩, ϵ, T_Call_Party_Busy⟩⟩,
⟨Select_Facility&Present_Call, 4, ⟨⟨⟨ssf, notification⟩, onhook⟩, ϵ, T_Abandon⟩⟩,
⟨T_Alerting, 1, ⟨⟨⟨ssf, notification⟩, no_answer⟩, ϵ, T_No_Answer⟩⟩,
⟨T_Alerting, 2, ⟨⟨⟨ssf, notification⟩, onhook⟩, ϵ, T_Abandon⟩⟩,
⟨T_Alerting, 3, ⟨⟨⟨ssf, notification⟩, offhook⟩, ϵ, T_Answer⟩⟩,
⟨T_Active, 1, ⟨⟨⟨ssf, request⟩, service⟩, ϵ, T_Mid_Call⟩⟩,
⟨T_Active, 2, ⟨⟨⟨ssf, notification⟩, onhook⟩, ϵ, T_Disconnect⟩⟩,
⟨T_Active, 3, ⟨⟨⟨ssf, notification⟩, exception⟩, ⟨⟨bcsm, notification⟩, terminate⟩, T_Exception⟩⟩,
⟨T_Exception, 1, ⟨ϵ, ϵ, T_Null⟩⟩,
⟨Term.Attempt_Authorized, 1, ⟨ϵ, ⟨⟨bcsm, request⟩, select_facility⟩, Select_Facility&Present_Call⟩⟩,
⟨T_Call_Party_Busy, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, busy⟩, T_Exception⟩⟩,
⟨T_Term_Seized, 1, ⟨ϵ, ϵ, T_Alerting⟩⟩,
⟨T_No_Answer, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, no_answer⟩, T_Exception⟩⟩,
⟨T_Answer, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, connection⟩, T_Active⟩⟩,
⟨T_Mid_Call, 1, ⟨ϵ, ϵ, T_Active⟩⟩,
⟨T_Disconnect, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, terminate⟩, T_Null⟩⟩,
⟨T_Abandon, 1, ⟨ϵ, ⟨⟨bcsm, notification⟩, terminate⟩, T_Null⟩⟩

}

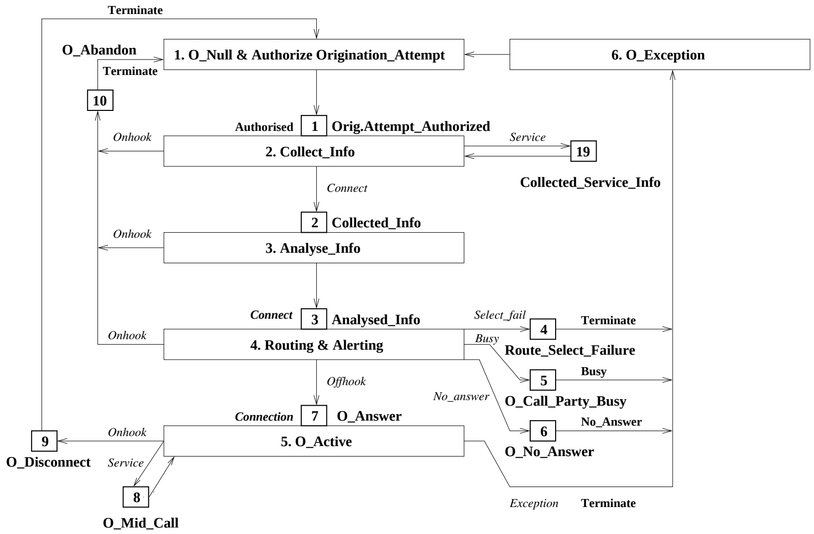


Figure 6.4: Figure 4-3 of [IN:Q.1214], annotated originating BCSM for CS-1

The guard, next and message functions handles the different aspects of an outgoing arc from the current *Position*. The guard function returns the guard of the n:th outgoing arc, whereas the next returns the new data if one takes the n:th arc. Finally the message function returns the events to be emitted,if any, should the BCSM follow the n:th arc.

$$
\begin{array}{lcl}
\mathsf{guard}(n, data) & \stackrel{\text{def}}{=} & \pi_1\left(\mathsf{arcs}(\mathsf{pos}(data), n)\right) \\
\mathsf{next}(n, data) & \stackrel{\text{def}}{=} & \langle \pi_3\left(\mathsf{arcs}(\mathsf{pos}(data), n)\right), \mathsf{queue}(data), \mathsf{party}(data) \rangle \\
\mathsf{message}(n, data, db) & \stackrel{\text{def}}{=} & \begin{cases} \epsilon & \mathsf{if}\ template = \epsilon \\ \langle \pi_1\left(template\right), \pi_2\left(template\right), d \rangle & \mathsf{otherwise} \end{cases} \\
 & \textbf{where} & template = \pi_2\left(\mathsf{arcs}(\mathsf{pos}(data), n)\right) \\
 & & d = \begin{cases} \mathsf{party}(data) & \mathsf{if}\ \mathsf{pos}(data) = \mathsf{Analysed\_Info} \\ \mathsf{phone}(db) & \mathsf{if}\ \mathsf{pos}(data) \in O\_PICs \cup \\ & \qquad T\_PICs \\ \bot & \mathsf{otherwise} \end{cases}
\end{array}
$$

The insert function inserts the *event* into the priority queue (pq) of events. The relation $\prec$ defines the order of priority and will be one of means of resolving feature

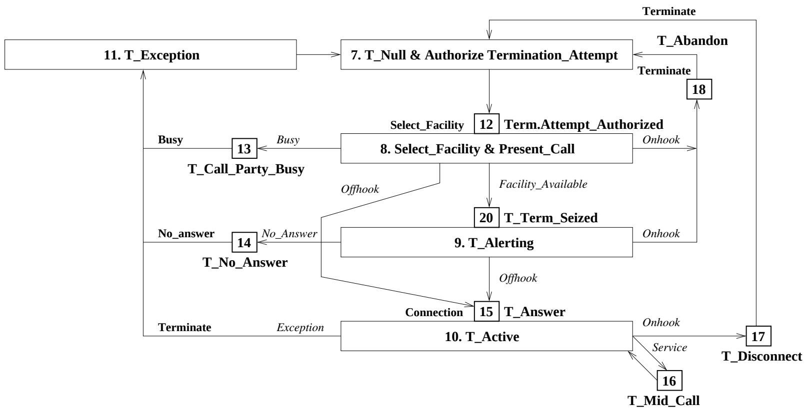


Figure 6.5: Figure 4-4 of [IN:Q.1214], annotated terminating BCSM for CS-1

interactions. Since we have as yet not tackled the problems of feature interactions no priorities are given, and consequently the $\prec$ relation is empty.

$$
\begin{aligned}
\mathsf{insert}(event, data) &\stackrel{\text{def}}{=} \langle \mathsf{pos}(data), \mathsf{insert1}(event, \mathsf{queue}(data)), \mathsf{party}(data)\rangle \\
\mathsf{insert1}(event, pq) &\stackrel{\text{def}}{=} \begin{cases} \langle event, \mathsf{nil}\rangle & \text{if } pq = \mathsf{nil} \\ \langle event, pq\rangle & \text{if } event \prec \mathsf{list\colon hd}(pq) \\ \langle \mathsf{list\colon hd}(pq), & \text{otherwise} \\ \quad \mathsf{insert1}(event, \mathsf{list\colon tl}(pq))\rangle & \end{cases} \\
\prec &\stackrel{\text{def}}{=} \emptyset
\end{aligned}
$$

The num function returns the number of outgoing arcs there are from a *position*.

$$
\begin{aligned}
\mathsf{num}(position) &\stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } position \in DPs \\ \mathsf{num1}(position) & \text{otherwise} \end{cases} \\
\mathsf{num1} &\stackrel{\text{def}}{=} \{\langle \mathsf{O\_Null}, 1\rangle, \langle \mathsf{Collect\_Info}, 3\rangle, \\
&\quad \langle \mathsf{Analyse\_Info}, 2\rangle, \langle \mathsf{Routing\&Alerting}, 5\rangle, \\
&\quad \langle \mathsf{O\_Active}, 3\rangle, \langle \mathsf{O\_Exception}, 1\rangle, \\
&\quad \langle \mathsf{T\_Null}, 1\rangle, \langle \mathsf{Select\_Facility\&Present\_Call}, 4\rangle, \\
&\quad \langle \mathsf{T\_Alerting}, 3\rangle, \langle \mathsf{T\_Active}, 3\rangle, \\
&\quad \langle \mathsf{T\_Exception}, 1\rangle\}
\end{aligned}
$$

The add_present function adds the events generated on entering this *position*, by the subscribed services.

$$\begin{aligned}
\mathsf{add_present}(data, db) &\stackrel{\text{def}}{=} \langle \mathsf{pos}(data),\\
&\quad \mathsf{add_present1}((\mathsf{present}(db))(\mathsf{pos}(data))),\\
&\quad \mathsf{party}(data)\rangle
\end{aligned}$$

$$\begin{aligned}
\mathsf{add_present1}(new, queue) &\stackrel{\text{def}}{=} \mathsf{if}(new = \mathsf{nil}\\
&\qquad queue,\\
&\qquad \mathsf{add_present1}(\mathsf{list{:}\,tl}(new),\\
&\qquad\qquad \mathsf{insert}(\mathsf{list{:}\,hd}(new), queue))
\end{aligned}$$

The active function searches the priority queue of events and looks if any event matches any of the guards of the outgoing arcs from the present point in call. If it finds an event matching a guard a tuple of that guard and the number of the arc is returned. If no matching event is found the function returns the tuple consisting of $\perp$ and 0.

$$\mathsf{active}(data) \stackrel{\text{def}}{=} \begin{cases} \langle \perp, 0\rangle & \mathsf{if}\ (guard = \epsilon)\ \wedge \\ & \quad (\mathsf{pos}(data) \in active)\ \wedge \\ & \quad (\mathsf{queue}(data) = \emptyset) \\ \mathsf{active}'(\mathsf{num}(\mathsf{pos}(data)), & \mathsf{otherwise} \\ \quad data) & \end{cases}$$
$$\textbf{where}\quad guard = \pi_1\,(\mathsf{active}'(\mathsf{num}(\mathsf{pos}(data)), data))$$
$$active = \{\mathsf{O\_Active}, \mathsf{T\_Active}\}$$

$$\mathsf{active}'(n, data) \stackrel{\text{def}}{=} \begin{cases} \langle \perp, 0\rangle & \mathsf{if}\ n = 0 \\ \mathsf{active}'(n-1, data) & \mathsf{if}\ guard = \perp \\ \langle guard, n\rangle & \mathsf{otherwise} \end{cases}$$
$$\textbf{where}\quad guard = \mathsf{find\_guard}(\mathsf{guard}(n, data)\mathsf{queue}(data))$$

$$\begin{aligned}\mathsf{find\_guard}(&guard,\\ &queue)\end{aligned} \stackrel{\text{def}}{=} \begin{cases} \epsilon & \mathsf{if}\ guard = \epsilon \\ \perp & \mathsf{if}\ \mathsf{list{:}\,null}(queue) \\ \mathsf{list{:}\,hd}(queue) & \mathsf{if}\ guard = template \\ \mathsf{find\_guard}(guard, & \mathsf{otherwise} \\ \qquad \mathsf{list{:}\,tl}(queue)) & \end{cases}$$
$$\textbf{where}\quad template = \mathsf{event{:}\,template}(\mathsf{list{:}\,hd}(queue))$$

The delete_guard function deletes the *guard* from the priority event queue, unless it is a service request. If it is a service request it has to be retained in the queue to be handled by the reached detection point.

$$\begin{aligned}
\mathsf{delete_guard}(event, data) &\stackrel{\text{def}}{=} \mathsf{if}((event = \epsilon) \vee \mathsf{event{:}\,request}(\mathsf{ssf}, \mathsf{service}, event),\\
&\qquad data,\\
&\qquad \langle \mathsf{pos}(data),\\
&\qquad\ \mathsf{list{:}\,remove}(event, \mathsf{queue}(data)),\\
&\qquad\ \mathsf{party}(data)\rangle)
\end{aligned}$$

The set_party function sets the party in the state if the *event* is a SSF connection request.

$$\begin{aligned}\mathsf{set\_party}(event, data) \;&\stackrel{\text{def}}{=}\; \mathsf{if}(\pi_1\,(event) = \langle\langle \mathsf{ssf}, \mathsf{request}\rangle, \mathsf{connect}, party\rangle,\\ &\qquad \langle \mathsf{pos}(data), \mathsf{queue}(data), party\rangle,\\ &\qquad data)\end{aligned}$$

The triggered function searches the priority queue for any event that is *armed*. If any armed event is found the this is returned otherwise $\perp$ is returned.

$$\mathsf{triggered}(data, db) \stackrel{\text{def}}{=} \mathsf{triggered1}(\mathsf{pos}(data), \mathsf{queue}(data), \mathsf{armed}(db))$$

$$\begin{aligned}\mathsf{triggered1}(&position,\\ &queue,\\ &armed) \stackrel{\text{def}}{=} \begin{cases} \perp & \mathsf{if\ list{:}\ null}(queue)\\ \mathsf{list{:}\ hd}(queue) & \mathsf{if}\ current \in armed\\ \mathsf{triggered1}(position, \mathsf{list{:}\ tl}(queue), armed) & \mathsf{otherwise}\end{cases}\\ &\textbf{where}\;\; current = \langle position, \mathsf{service\_template}(\mathsf{list{:}\ hd}(queue))\rangle\end{aligned}$$

The service_template function takes an *event* and if it is a service request, it constructs the service template.

$$\mathsf{service\_template}(event) \stackrel{\text{def}}{=} \langle \mathsf{event{:}\ type}(event), \pi_1\,(\mathsf{event{:}\ data}(event))\rangle$$

The report and request functions tests whether an event is a report or request event respectively.

$$\begin{aligned}\mathsf{report}(event\_template) &\stackrel{\text{def}}{=} \pi_2\,(\pi_1\,(event\_template)) = \mathsf{report}\\ \mathsf{request}(event\_template) &\stackrel{\text{def}}{=} \pi_2\,(\pi_1\,(event\_template)) = \mathsf{request}\end{aligned}$$

The pack function constructs a service request message to the SSF for delivery to the SCF, incorporating the service specific information as well the parts of the BCSM state needed.

$$\begin{aligned}\mathsf{pack}(&type, name,\\ &ssd, data,\\ &\langle armed, p, l, a, c, ph\rangle) \stackrel{\text{def}}{=} \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle,\\ &\quad type,\\ &\quad \langle \pi_1\,(l(name)),\\ &\quad\; \pi_2\,(l(name)) \cup\\ &\quad\; ssd \cup\\ &\quad\; \{\langle \mathsf{armed}, armed\rangle, \langle \mathsf{lookup}, l\rangle, \langle \mathsf{present}, p\rangle,\\ &\quad\;\; \langle \mathsf{accounting}, a\rangle, \langle \mathsf{cost}, c\rangle,\\ &\quad\;\; \langle \mathsf{party}, \mathsf{party}(data)\rangle, \langle \mathsf{phone}, ph\rangle\}\rangle\rangle\end{aligned}$$

The clean function removes untreated events from the priority queue. Some of the events are not removed, such as a number of SSF notifications and in some positions service request.

$$\mathsf{clean}(data) \stackrel{\text{def}}{=} \langle \mathsf{pos}(data), \mathsf{clean1}(\mathsf{pos}(data), \mathsf{queue}(data)), \mathsf{party}(data)\rangle$$

$$\mathsf{clean1}(pos, queue) \stackrel{\text{def}}{=} \begin{cases} \mathsf{nil} & \text{if } queue = \mathsf{nil} \\ \langle \mathsf{list{:}\,hd}(queue), & \text{if } template \in stay \\ \quad \mathsf{clean1}(pos, \mathsf{list{:}\,tl}(queue))\rangle & \\ \mathsf{clean1}(pos, \mathsf{list{:}\,tl}(queue)) & \text{otherwise} \end{cases}$$

$$\begin{aligned} \textbf{where}\quad & template = \mathsf{event{:}\,template}(\mathsf{list{:}\,hd}(queue)) \\ & stay = phone \cup service \\ & phone = \{\langle\langle \mathsf{ssf}, \mathsf{notification}\rangle, \mathsf{offhook}\rangle \\ & \qquad \langle\langle \mathsf{ssf}, \mathsf{notification}\rangle, \mathsf{onhook}\rangle \\ & \qquad \langle\langle \mathsf{ssf}, \mathsf{request}\rangle, \mathsf{connect}\rangle \\ & \qquad \langle\langle \mathsf{ssf}, \mathsf{notification}\rangle, \mathsf{facility\_available}\rangle \\ & \qquad \langle\langle \mathsf{ssf}, \mathsf{notification}\rangle, \mathsf{select\_fail}\rangle \\ & \qquad \langle\langle \mathsf{ssf}, \mathsf{notification}\rangle, \mathsf{busy}\rangle \\ & \qquad \langle\langle \mathsf{ssf}, \mathsf{notification}\rangle, \mathsf{no\_answer}\rangle\} \\ & service = \mathsf{if}(pos \in \{\mathsf{Collected\_Service\_Info}, \\ & \qquad\qquad \mathsf{O\_Mid\_Call}, \mathsf{T\_Mid\_Call}\}, \\ & \qquad \emptyset, \\ & \qquad \{\langle\langle \mathsf{ssf}, \mathsf{request}\rangle, \mathsf{service}\rangle\}) \end{aligned}$$

The por function returns the first point of return event found in the priority queue, if none is present it will return $\perp$.

$$\mathsf{por}(data) \stackrel{\text{def}}{=} \mathsf{por1}(\mathsf{queue}(data))$$

$$\mathsf{por1}(queue) \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } queue = \mathsf{nil} \\ \mathsf{list{:}\,hd}(queue) & \text{if } type = \langle \mathsf{scf}, \mathsf{response}\rangle \\ \mathsf{por1}(\mathsf{list{:}\,tl}(queue)) & \text{otherwise} \end{cases}$$

$$\textbf{where}\quad type = \mathsf{event{:}\,type}(\mathsf{list{:}\,hd}(queue))$$

6.5 RULES

EVENT RULE

The following rule is applicable when there is a message in the event queue. The BCSM removes the first message in the event queue and inserts it into the priority queue. Since this rule is the first to be applied the remaining rules to look at the priority queue will always attend to the highest priority event first.

$$(6.1)\quad \frac{\sigma \neq \mathsf{nil}}{\langle\, mode,\ data,\ \sigma,\ db\,\rangle \ \longrightarrow\ \langle\, mode,\ \mathsf{insert}(\mathsf{list\colon hd}(\sigma), data),\ \mathsf{list\colon tl}(\sigma),\ db\,\rangle}$$

PIC RULE

The following rule is applicable if the BCSM is in a point in call and the the guard of arc n is true. The BCSM follows arc n to the next detection point and adds the events corresponding the subscriptions using add_present. The message is sent and the set_party may set the *party*. If there according to arcs is any message to send, a message is sent. message is

$$(6.2)\quad \frac{guard \neq \bot}{\langle\, \mathsf{pic},\ data,\ \mathsf{nil},\ db\,\rangle \xrightarrow{\mathsf{broadcast}\ (m)} \langle\, \mathsf{dp},\ data^2,\ \mathsf{nil},\ db\,\rangle}$$

$$\begin{array}{lrcl}
\textbf{where} & \langle guard, n\rangle & = & \mathsf{active}(data) \\
& data^1 & = & \mathsf{set_party}(guard, \mathsf{next}(n, data)) \\
& data^2 & = & \mathsf{add_present}(\mathsf{delete_guard}(guard, data^1), db) \\
& message & = & \mathsf{message}(n, data, db) \\
& m & = & \begin{cases} \{\langle message, \mathsf{ssf}\rangle\} & \mathsf{if}\ message \neq \epsilon \\ \emptyset & \mathsf{otherwise} \end{cases}
\end{array}$$

DP RULES

The following rule is applicable if the BCSM is at a detection point and there is a triggered event. If it is a request event the BCSM is stalled otherwise it continue running in the detection point. The report or request event is sent to the SSF.

$$\frac{triggered \neq \bot}{\langle\, \mathsf{dp},\ data,\ \mathsf{nil},\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\,\mathsf{ssf})} \langle\, mode,\ data',\ \mathsf{nil},\ db\,\rangle} \tag{6.3}$$

$$\textbf{where}\quad \begin{aligned} triggered &= \mathsf{triggered}(data, db)\\ data' &= \langle \mathsf{pos}(data),\\ &\quad\ \mathsf{list}\colon \mathsf{remove}(triggered, \mathsf{queue}(data)),\\ &\quad\ \mathsf{party}(data)\rangle\\ \langle mode, type\rangle &= \begin{cases} \langle \mathsf{dp}, \mathsf{report}\rangle & \mathsf{if\ report}(triggered)\\ \langle \mathsf{stalled}, \mathsf{request}\rangle & \mathsf{if\ request}(triggered)\end{cases}\\ m &= \mathsf{pack}(type, \mathsf{event}\colon \mathsf{name}(triggered),\\ &\qquad\quad \mathsf{event}\colon \mathsf{data}(triggered), data', db)\end{aligned}$$

The following rule is applicable if the BCSM is at a detection point and there are no triggered events. The priority event queue is cleared using the clean function and normal call handling is resumed at the next point in call. Should according to the arcs function any message be sent, then a message is sent.

$$\frac{\mathsf{triggered}(data, db) = \bot}{\langle\, \mathsf{dp},\ data,\ \mathsf{nil},\ db\,\rangle \xrightarrow{\mathsf{broadcast}\ (m)} \langle\, \mathsf{pic},\ \mathsf{next}(1, \mathsf{clean}(data)),\ \mathsf{nil},\ db\,\rangle} \tag{6.4}$$

$$\textbf{where}\quad m = \begin{cases} \{\langle \mathsf{message}(1, data, db), \mathsf{ssf}\rangle\} & \mathsf{if\ message}(1, data, db) \neq \epsilon\\ \emptyset & \mathsf{otherwise}\end{cases}$$

POR RULES

The following rule is applicable if the BCSM is stalled at a detection point and there is a $\mathsf{continue\_with\_existing\_data}$ point of return event in the priority event queue. Operation is resumed at the current detection point.

$$\frac{\mathsf{event{:}\,response}(\mathsf{scf}, \mathsf{continue\_with\_existing\_data}, por)}{\langle\, \mathsf{stalled},\ data,\ \mathsf{nil},\ db\,\rangle \ \longrightarrow\ \langle\, \mathsf{dp},\ data',\ \mathsf{nil},\ db\,\rangle} \tag{6.5}$$

$$\begin{aligned} \textbf{where}\quad por &= \mathsf{por}(data) \\ data' &= \langle \mathsf{pos}(data), \mathsf{list{:}\,remove}(por, \mathsf{queue}(data)), \mathsf{party}(data)\rangle \end{aligned}$$

The following rule is applicable if the BCSM is stalled at a detection point and there is a $\mathsf{proceed\_with\_new\_data}$ point of return event in the priority event queue. The changed data is updated (undefined values means that the data should remain unchanged), and operation is resumed at the current detection point.

$$\frac{\mathsf{event{:}\,response}(\mathsf{scf}, \mathsf{proceed\_with\_new\_data}, por)}{\langle\, \mathsf{stalled},\ data,\ \mathsf{nil},\ db\,\rangle \ \longrightarrow\ \langle\, mode,\ data',\ \mathsf{nil},\ db'\,\rangle} \tag{6.6}$$

$$\begin{aligned} \textbf{where}\quad por &= \mathsf{por}(data) \\ \langle pos, queue, party\rangle &= data \\ \langle armed, p, l, a, c, ph\rangle &= db \\ \langle pc', armed', p', l', a', c'\rangle &= \mathsf{event{:}\,data}(por) \\ \langle mode, pos'\rangle &= \mathsf{ifdef}(pc', pc', \langle \mathsf{dp}, pos\rangle) \\ armed'' &= \mathsf{ifdef}(armed', armed', armed) \\ l'' &= \mathsf{ifdef}(l', l', l) \\ p'' &= \mathsf{ifdef}(p', p', p) \\ a'' &= \mathsf{ifdef}(a', a', a) \\ c'' &= \mathsf{ifdef}(c', c + c', c) \\ db' &= \langle armed'', p'', l'', a'', c'', ph\rangle \\ queue' &= \mathsf{list{:}\,remove}(por, queue) \\ data' &= \langle pos', queue', party\rangle \\ data'' &= \mathsf{ifundef}(pc', \\ &\qquad\quad data', \\ &\qquad\quad \mathsf{if}(mode = \mathsf{pic}, \\ &\qquad\qquad \mathsf{clean}(data'), \\ &\qquad\qquad \mathsf{add\_present}(\mathsf{clean}(data'), db))) \end{aligned}$$

The following rule is applicable if the BCSM is stalled at a detection point and there is a clear_call point of return event in the priority event queue. The priority event queue is cleared and the call terminated.

$$\frac{\textsf{event: response}(\textsf{scf}, \textsf{clear\_call}, por)}{\langle\, \textsf{stalled},\ data,\ \textsf{nil},\ db\,\rangle \xrightarrow{\textsf{send}\ (m,\textsf{ssf})} \langle\, \textsf{pic},\ data',\ \textsf{nil},\ db\,\rangle} \tag{6.7}$$

where

$$\begin{array}{rcl}
por & = & \textsf{por}(data) \\
m & = & \langle\langle \textsf{bcsm}, \textsf{notification}\rangle, \textsf{terminate}, \pi_6\,(db)\rangle \\
pos & = & \begin{cases} \textsf{O_Exception} & \textsf{if pos}(data) \in o_exception \\ \textsf{O_Null} & \textsf{if pos}(data) \in o_null \\ \textsf{T_Exception} & \textsf{if pos}(data) \in t_exception \\ \textsf{T_Null} & \textsf{if pos}(data) \in t_null \end{cases} \\
o_exception & = & \{\textsf{Orig.Attempt_Authorized}, \textsf{Collected_Service_Info}, \\
 & & \textsf{Collected_Info}, \textsf{Analysed_Info}, \textsf{Route_Select_Failure}, \\
 & & \textsf{O_Call_Party_Busy}, \textsf{O_No_Answer}, \textsf{O_Answer}, \\
 & & \textsf{O_Mid_Call}\} \\
o_null & = & \{\textsf{O_Disconnect}, \textsf{O_Abandon}\} \\
t_exception & = & \{\textsf{Term.Attempt_Authorized}, \textsf{T_Term_Seized}, \\
 & & \textsf{T_Call_Party_Busy}, \textsf{T_No_Answer}, \\
 & & \textsf{T_Answer}, \textsf{T_Mid_Call}\} \\
t_null & = & \{\textsf{T_Disconnect}, \textsf{T_Abandon}\} \\
data' & = & \langle pos', \textsf{nil}, \perp\rangle
\end{array}$$

The following rule is applicable if the BCSM is stalled at a detection point and there is a initiate_call point of return event in the priority event queue. If the event is not of *type* set party to $\bot$, and update *armed*. Normal call handling is resumed in the alerting point in call (in the case of an originating BCSM at Routing&Alerting, and in the case of a terminating BCSM at T_Alerting).

$$\frac{\mathsf{event\!:response}(\mathsf{scf}, \mathsf{initiate\_call}, por)}{\langle\, \mathsf{stalled},\ data,\ \mathsf{nil},\ db\,\rangle\ \longrightarrow\ \langle\, \mathsf{pic},\ data',\ \mathsf{nil},\ db'\,\rangle} \tag{6.8}$$

$$\textbf{where}\quad \begin{aligned} por &= \mathsf{por}(data) \\ \langle type, armed' \rangle &= \mathsf{event\!:data}(por) \\ \langle armed, p, l, a, c, ph \rangle &= db \\ pos &= \begin{cases} \mathsf{Routing\&Alerting} & \mathsf{if}\ pc \in O\_DPs \\ \mathsf{T\_Alerting} & \mathsf{otherwise} \end{cases} \\ data' &= \langle pos, \\ &\quad\ \mathsf{list\!:remove}(por, \mathsf{queue}(data)), \\ &\quad\ \mathsf{party}(data) \rangle \\ armed'' &= \mathsf{ifdef}(armed', armed', armed) \\ db' &= \langle armed'', p, l, a, c, \mathsf{if}(type = \mathsf{active}, ph, \bot) \rangle \end{aligned}$$

CHAPTER 7

SCF - SERVICE CONTROL FUNCTION

In this chapter we will describe the service control function, first informally its role and the scope of the model. Thereafter follows a more rigorous although annotated description.

7.1 DESCRIPTION

The SCF should contain all the extra control logic necessary to provide the IN services, or gain use of such through communication with the other functional entities, viz. SDF and SRF. The SCF takes a controlling and monitoring relationships towards the SSF. Its functionality is briefly defined in [IN:Q.1204, pp.3–4] as:

> The SCF is a function that commands call control functions in the processing of IN provided and/or custom service requests. The SCF may interact with other functional entities to access additional logic or to obtain information (service or user data) required to process a call/service instance. It
>
> a) interfaces and interacts with service switching function/call control function, specialized resource function (SRF) and service data function (SDF) functional entities;
>
> b) contains the logic and processing capability required to handle IN provided service attempts;
>
> c) interfaces and interacts with other SCFs, if necessary;
>
> d) is managed, updated and/or otherwise administered by a SMF.

In this thesis the SCF gets much the role of a communications bridge between the Service instance processes and the SSF. It passes communications between the Service instance processes and the SSF and initiates new Service instance processes when requested by the SSF or a Service instance process. Much of the intended functionality of the SCF is embodied by the individual Service instance processes.

7.2 DOMAINS

The state of the of SCF consists of an *EventQueue* of untreated incoming messages and descriptions of the *Services*.

$$\begin{aligned} SCFState &\equiv SCFMode \times SCFData \times \\ &\quad \mathsf{event}\colon EventQueue \times SCFDataBase \end{aligned}$$

$$\begin{aligned} SCFMode &\equiv \{\mathsf{running}\} \\ SCFData &\equiv \{\bot\} \end{aligned}$$

The *SCFDDatabase* contains mappings from the logical names of the services (since a service may have several names) to their *ServiceDescriptions*. The *ServiceDescriptions* contains mappings from names of each building block of the service to its *SIBInstance*.

$$\begin{aligned} SCFDataBase &\equiv ServiceLogicName \to ServiceDescription \\ ServiceLogicName &\equiv \mathsf{hsibs}\colon HighLevelSIB \cup \\ &\quad \mathsf{service\ features}\colon ServiceFeature \cup \\ &\quad \mathsf{services}\colon ServiceName \\ ServiceDescription &\equiv Id \to \mathsf{si}\colon SIBInstance \end{aligned}$$

Each *SIBInstance* consists of the name to the SIB and the mapping of names of service support data to the actual values.

$$\begin{aligned} SIBInstance &\equiv SIBId \times SSD \\ SSD &\equiv Id \to Data \end{aligned}$$

The *ServiceId* contains the *ServiceName*s and digit-, special character-sequences.

$$ServiceId \equiv \mathsf{List}(\{\mathbb{N}, \#, *\}) \cup \mathsf{services}\colon ServiceName$$

The *SCFEventName* domain are the names of those events accepted or issued by the SCF.

$$SCFEventName \equiv \{\mathsf{poi}, \mathsf{report}, \mathsf{request}\}$$

7.3 RULES

DELIVER RULE

The following rule is applicable when a report request is at the head of the *EventQueue*. The report is sent to the service instance which is addressed.

$$\frac{\mathsf{event}\colon \mathsf{request}(\mathsf{ssf}, \mathsf{report}, \mathsf{list}\colon \mathsf{hd}(\sigma))}{\langle\, \mathsf{running}\,, \bot\,, \sigma\,, db\,\rangle \xrightarrow{\mathsf{send}\ (report, to)} \langle\, \mathsf{running}\,, \bot\,, \mathsf{list}\colon \mathsf{tl}(\sigma)\,, db\,\rangle} \tag{7.1}$$

$$\textbf{where} \quad \langle to, report\rangle = \mathsf{event}\colon \mathsf{data}(\mathsf{list}\colon \mathsf{hd}(\sigma))$$

INITIATE SERVICE INSTANCE RULES

The following rule is applicable when a service request from either a BCSM via the SSF is at the head of the *EventQueue*. The name of the requested service instance together with the data to the service is extracted from the message. Then a Service instance process is created.

$$\frac{\mathsf{event:request(ssf, request, list: hd}(\sigma))}{\langle\, \mathsf{running}\,, \perp, \sigma\,, db\,\rangle \xrightarrow{\mathsf{create(si,}state)} \langle\, \mathsf{running}\,, \perp, \mathsf{list: tl}(\sigma)\,, db\,\rangle} \tag{7.2}$$

$$\begin{array}{lrcl}
\textbf{where} & \langle\langle name, ssd\rangle, from\rangle & = & \mathsf{event: data(list: hd}(\sigma)) \\
& state & = & \langle \mathsf{running}, \\
& & & \quad \langle\langle name, \mathsf{start} : name\rangle, \emptyset, \langle\langle \perp, ssd\rangle, \mathsf{nil}\rangle\rangle, \\
& & & \quad \mathsf{nil}, \\
& & & \quad \langle db, from\rangle\rangle
\end{array}$$

The following rule is applicable when a request for a Service instance process initiation from a Service instance process is at the head of the *EventQueue*. The name of the requested service instance together with the data is extracted from the message and a Service instance process is created. If it is a point of initiation (poi) request then the *service_id* of the initiated Service instance process is sent to the requesting Service instance process.

$$\frac{\mathsf{event: request(si, poi, list: hd}(\sigma)) \vee \mathsf{event: request(si, request, list: hd}(\sigma))}{\langle\, \mathsf{running}\,, \perp, \sigma\,, db\,\rangle \xrightarrow[]{\substack{id{:=}\mathsf{create(si,}state);\\ \mathsf{broadcast}\ (m)}} \langle\, \mathsf{running}\,, \perp, \mathsf{list: tl}(\sigma)\,, db\,\rangle} \tag{7.3}$$

$$\begin{array}{lrcl}
\textbf{where} & \langle name, ssd\rangle & = & \mathsf{event: data(list: hd}(\sigma)) \\
& from & = & \mathsf{event: sender(list: hd}(\sigma)) \\
& state & = & \langle \mathsf{running}, \\
& & & \quad \langle\langle name, \mathsf{start} : name\rangle, \emptyset, \langle\langle \perp, ssd\rangle, \mathsf{nil}\rangle\rangle, \\
& & & \quad \mathsf{nil}, \\
& & & \quad \langle db, from\rangle\rangle \\
& type & = & \mathsf{event: name(list: hd}(\sigma)) \\
& m & = & \begin{cases} \{\langle\langle\langle \mathsf{scf, response}\rangle, & \text{if } type = \mathsf{poi} \\ \quad \mathsf{report}, & \\ \quad id\rangle, & \\ \quad from\rangle\} & \\ \emptyset & \text{if } type = \mathsf{request} \end{cases}
\end{array}$$

CHAPTER 8

SDF - SERVICE DATA FUNCTION

In this chapter we will describe the service data function, first informally it role and the scope of the model. Thereafter follows a more rigorous although annotated description.

8.1 DESCRIPTION

The SDF s functions as a global database where data pertaining to the IN services may be stored and looked up. The SDF also acts as a uniform interface against external database systems visible to the IN. The SDF is defined in [IN:Q.1204, p.4]:

> The SDF contains customer and network data for real time asccess by the SCF in the execution of an IN provided service. It
>
> a) interfaces and interacts with the SCFs as required;
>
> b) interfaces and interacts with other SDFs, if necessary;
>
> c) is managed, updated and/or otherwise administered by a SMF.
> NOTE – The SDF contains data relating directly to the provision or operation of IN provided services. Thus it does not necessarily encompass data provided by a third party such as credit information, but may provide access to these data.

The databases of the SDF are either stored in the SDF itself or references to other external databases. The model of the SDF include the framework for calling external databases although no external databases has yet been modeled. This preparation for external databases give rise to a number of definitions that at this time may seem without use.

8.2 DOMAINS

The state of the SDF consists of an *EventQueue* of untreated incoming messages. The dynamic data of the SDF contains the databases, but also should it be waiting for an outstanding request to an external database, the source and destination of an external database request.

The static data of the SDF is a table over the external database sources.

$$\begin{aligned} \mathit{SDFState} \;\equiv\;& \mathit{SDFMode} \times \mathit{SDFData} \times \\ & \mathsf{event}{:}\,\mathit{EventQueue} \times \mathit{SDFDataBase} \end{aligned}$$

The mode of SDF consists of the two constants running and waiting and signifies respectively: that the SDF is prepared to handle the next message or that it is wearing for an outstanding answer to an external request.

$$SDFMode \equiv \{\mathsf{running}, \mathsf{waiting}\}$$

The *SDFData* contains the information concerning external requests, and the local databases. The part of the data concerning an external request in progress, *Waiting*, consists of three parts: to whom the request has been forwarded, who made the request and what action was requested.

The local databases consists of partial functional mapping from *id*:s to any legally formed *Data* or set of legally formed *Data*.

The *SDFAction* domain contains the names of the applicable actions which is most easily understood reading their respective rules below. The *SDFFile* domain contains the names of the databases, at this time they are the accounts database used by the billing system, the AAB_PASSWD and AAB_ACCOUNT databases used by the Automatic Alternative Billing service used as an example in this thesis. Further there is the CF_TABLE and CFB_TABLE used by the call forwarding and call forwarding on busy line services respectively.

$$\begin{array}{lcl}
SDFData & \equiv & Waiting \times DataBases \\
Waiting & \equiv & (\mathsf{event}\colon OriginatorId \times \mathsf{event}\colon OriginatorId \times SDFAction)_{\perp} \\
DataBases & \equiv & SDFFile \to DataBase \\
DataBase & \equiv & (Id \to Data) \cup \mathbb{P}\,(Data) \\
 & & \\
SDFAction & \equiv & \{\mathsf{lookup}, \mathsf{retrieve}, \mathsf{add}, \mathsf{remove}, \mathsf{replace}, \mathsf{replace_insert}, \\
 & & \ \mathsf{increment}, \mathsf{decrement}\} \\
SDFFile & \equiv & \{\mathsf{accounts}, \\
 & & \ \mathsf{AAB_PASSWD}, \mathsf{AAB_ACCOUNT}, \\
 & & \ \mathsf{CF_TABLE}, \mathsf{CFB_TABLE}\}
\end{array}$$

The *SDFDataBase* contains partial functional mappings from pairs of database names and actions to the names and databases and actions of external database sources. The intention is that if there is a mapping from the database name and actions requested the request must be forwarded to the external database. Since the external databases may not have the same vocabulary as SDF the mapping includes an external action as well as name of the database in question.

Since the no external databases have be incorporated into the model so far the domains *ExternalFile* and *ExternalAction* are empty.

$$\begin{array}{lcl}
SDFDataBase & \equiv & (SDFFile \times SDFAction) \to \\
 & & (\mathsf{event}\colon OriginatorId \times \\
 & & \ ExternalFile \times ExternalAction)_{\perp} \\
ExternalFile & \equiv & \emptyset \\
ExternalAction & \equiv & \emptyset
\end{array}$$

The $SDFEventName$ domain are the names of those events accepted or issued by the SDF.

$$SDFEventName \equiv SDFAction \cup ExternalAction$$

8.3 FUNCTIONS

The waiting and database selects the information concerning an outstanding external database request and the databases, from the dynamic data, respectively.

$$\begin{aligned}
\mathsf{waiting}(data) &\stackrel{\text{def}}{=} \pi_1\,(data) \\
\mathsf{database}(data) &\stackrel{\text{def}}{=} \pi_2\,(data)
\end{aligned}$$

The wait_for function adds the information to the dynamic data concerning an outstanding external database request, the originator of the request ($requester$) what the action requested was ($action$) and to $whom$ the request was forwarded.

The update_db function updates the databases with the $update$.

$$\begin{aligned}
\mathsf{wait_for}(&whom, \\ &requester, \\ &action, \\ &data) &\stackrel{\text{def}}{=} \langle\langle whom, requester, action\rangle, \mathsf{database}(data)\rangle \\
\mathsf{update_db}(&update, \\ &data) &\stackrel{\text{def}}{=} \langle \mathsf{waiting}(data), \mathsf{database}(data) \oplus update\rangle
\end{aligned}$$

The file function selects the aproriate database named *file* from the dynamic data. The elt selects the *element* from the database *file* in the dynamic data.

$$\begin{aligned}
\mathsf{file}(file, data) &\stackrel{\text{def}}{=} (\mathsf{database}(data))(file) \\
\mathsf{elt}(file, element, data) &\stackrel{\text{def}}{=} (\mathsf{file}(file, data))(element)
\end{aligned}$$

The action function tests whether the name of the first element in the event queue is $action$.

$$\mathsf{action}(action, \sigma) \stackrel{\text{def}}{=} action = \mathsf{event}\colon \mathsf{name}(\mathsf{list}\colon \mathsf{hd}(\sigma))$$

These functions tests whether the requested database file exists and whether the file exists and the element requested, respectively.

$$\begin{aligned}
\mathsf{exist}(file, data) &\stackrel{\text{def}}{=} \mathsf{file}(file, data) \neq \bot \\
\mathsf{exist}(file, item, data) &\stackrel{\text{def}}{=} \mathsf{exist}(file, data) \wedge_\bot (\mathsf{elt}(file, item, data) \neq \bot)
\end{aligned}$$

The $\mathsf{file\_element\_missing}$ function tests whether the database (*file*) or, depending on the type of action, *element* requested is missing.

$$
\begin{array}{rcl}
\mathsf{file\_element\_missing}(data, \sigma) & \stackrel{\text{def}}{=} & (action \in type_1 \wedge \neg \mathsf{exist}(file, data)) \vee \\
& & (action \in type_2 \wedge \neg \mathsf{exist}(file, item, data)) \\
\textbf{where} & & \begin{array}{lcl}
action & = & \mathsf{event: name(list: hd}(\sigma)) \\
\langle file, & & \\
\ item, & & \\
\ action\rangle & = & \mathsf{event: data(list: hd}(\sigma)) \\
type_1 & = & \{\mathsf{lookup, add, replace\_insert}\} \\
type_2 & = & \{\mathsf{retrieve, replace,} \\
& & \ \mathsf{increment, decrement}\}
\end{array}
\end{array}
$$

The $\mathsf{external\_reply}$ traverses the event queue and checks whether any message has been received from the *external* database which the `SDF` is waiting for. If there is a message this is returned otherwise $\perp$ is returned.

$$
\begin{array}{rcl}
\mathsf{external\_request}(\sigma, db) & \stackrel{\text{def}}{=} & db(\langle file, action\rangle) \neq \perp \\
\textbf{where} & & action = \mathsf{event: name(list: hd}(\sigma)) \\
& & \langle file, \_, \_\rangle = \mathsf{event: data(list: hd}(\sigma)) \\
\mathsf{external\_reply}(\sigma, extern) & \stackrel{\text{def}}{=} & \begin{cases} \perp & \mathsf{if\ list: null}(\sigma) \\ \mathsf{list: hd}(\sigma) & \mathsf{if}\ sender = extern \\ rest & \mathsf{otherwise} \end{cases} \\
\textbf{where} & & sender = \mathsf{event: sender(list: hd}(\sigma)) \\
& & rest = \mathsf{external\_reply(list: tl}(\sigma), extern)
\end{array}
$$

8.4 RULES

EXTERNAL DATABASE RULES

The following rule is applicable when the first element of the event queue is a request for an external database. The received request is translated into the external database name (*file'*) and action (*action'*). The translated requested is then forwarded to the external database, and the SDF is suspended waiting for an answer. It is also noted from whom an answer is expected.

$$\frac{\mathsf{external\_request}(\sigma)}{\langle\,\mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\,to)} \langle\,\mathsf{waiting},\ data',\ \mathsf{list\!:tl}(\sigma),\ db\,\rangle} \tag{8.1}$$

$$\begin{array}{llcl}
\textbf{where} & \langle file, item, v\rangle & = & \mathsf{event\!: data(list\!: hd}(\sigma)) \\
 & action & = & \mathsf{event\!: name(list\!: hd}(\sigma)) \\
 & \langle to, file', action'\rangle & = & db(\langle file, action\rangle) \\
 & data' & = & \mathsf{wait_for}(to, \\
 & & & \quad \mathsf{event\!: sender(list\!: hd}(\sigma)), \\
 & & & \quad action, \\
 & & & \quad data) \\
 & m & = & \langle\langle \mathsf{sdf}, \mathsf{request}\rangle, action', \langle file', item, v\rangle\rangle
\end{array}$$

The following rule is applicable when the SDF is suspended waiting for an outstanding external database request, and the answer to the request is now in the event queue. The answer is translated back to the original and sent to the originator of the request.

$$\frac{reply \neq \bot}{\langle\,\mathsf{waiting},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\,to)} \langle\,\mathsf{running},\ data,\ \mathsf{list\!: remove}(reply, \sigma),\ db\,\rangle} \tag{8.2}$$

$$\begin{array}{llcl}
\textbf{where} & \langle external, to, action\rangle & = & \mathsf{waiting}(data) \\
 & reply & = & \mathsf{external_reply}(\sigma, external) \\
 & m & = & \langle\langle \mathsf{sdf}, \mathsf{report}\rangle, action, \mathsf{event\!: data(reply)}\rangle
\end{array}$$

ERROR RULE

The following rule is applicable when the first element of event queue is a request where the database or database entry queried for does not exist. An error message is sent back to the originator of the request, and the request discarded.

$$\frac{\mathsf{file\_element\_missing}(data, \sigma)}{\langle\,\mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\,sender)} \langle\,\mathsf{running},\ data,\ \mathsf{list\!: tl}(\sigma),\ db\,\rangle} \tag{8.3}$$

$$\begin{array}{llcl}
\textbf{where} & m & = & \langle\langle \mathsf{sdf}, \mathsf{report}\rangle, \mathsf{event\!: name}(\sigma), \langle \mathsf{fail}, \bot\rangle\rangle \\
 & sender & = & \mathsf{event\!: sender(list\!: hd}(\sigma))
\end{array}$$

Lookup rule

The following rule is applicable when the requested operation is lookup. If the *item* look for is present in the database *file* the result is yes otherwise no.

$$\frac{\mathsf{action}(\mathsf{lookup},\sigma)}{\langle\,\mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\,(m,\,sender)} \langle\,\mathsf{running},\ data,\ \mathsf{list\!:tl}(\sigma),\ db\,\rangle} \tag{8.4}$$

$$\begin{array}{rrcl}
\textbf{where} & \langle file, item, _\rangle & = & \mathsf{event\!:data}(\mathsf{list\!:hd}(\sigma)) \\
 & result & = & \begin{cases} \mathsf{yes} & \mathsf{if}\ item \in \mathsf{file}(file, data) \\ \mathsf{no} & \mathsf{otherwise} \end{cases} \\
 & m & = & \langle\langle\mathsf{sdf},\mathsf{report}\rangle, \mathsf{lookup}, \langle\mathsf{success}, result\rangle\rangle \\
 & sender & = & \mathsf{event\!:sender}(\mathsf{list\!:hd}(\sigma))
\end{array}$$

Retrieve rule

The following rule is applicable when the requested operation is $\mathsf{retrieve}$. The element mapped to by *item* database (*file*) is returned to the originator of the request.

$$\frac{\mathsf{action}(\mathsf{retrieve},\sigma)}{\langle\,\mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\,(m,\,sender)} \langle\,\mathsf{running},\ data,\ \mathsf{list\!:tl}(\sigma),\ db\,\rangle} \tag{8.5}$$

$$\begin{array}{rrcl}
\textbf{where} & \langle file, item, _\rangle & = & \mathsf{event\!:data}(\mathsf{list\!:hd}(\sigma)) \\
 & m & = & \langle\langle\mathsf{sdf},\mathsf{report}\rangle, \mathsf{retrieve}, \langle\mathsf{success}, \mathsf{elt}(\mathit{file}, item, data)\rangle\rangle \\
 & sender & = & \mathsf{event\!:sender}(\mathsf{list\!:hd}(\sigma))
\end{array}$$

Modification rules

The following rule is applicable when the requested operation is add. The *item* is added to the set of database (*file*). And a message reporting $\mathsf{success}$ to the requester.

$$\frac{\mathsf{action}(\mathsf{add},\sigma)}{\langle\,\mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\,(m,\,sender)} \langle\,\mathsf{running},\ data',\ \mathsf{list\!:tl}(\sigma),\ db\,\rangle} \tag{8.6}$$

$$\begin{array}{rrcl}
\textbf{where} & \langle file, item, _\rangle & = & \mathsf{event\!:data}(\mathsf{list\!:hd}(\sigma)) \\
 & data' & = & \mathsf{update_db}(\langle file, \{item\} \cup \mathsf{file}(file, data)\rangle, data) \\
 & m & = & \langle\langle\mathsf{sdf},\mathsf{report}\rangle, \mathsf{add}, \langle\mathsf{success}, \perp\rangle\rangle \\
 & sender & = & \mathsf{event\!:sender}(\mathsf{list\!:hd}(\sigma))
\end{array}$$

The following rule is applicable when the requested operation is remove. The *item* is removed from the set of database (*file*). And a message reporting $\mathsf{success}$ to the requester.

$$\frac{\mathsf{action}(\mathsf{remove},\sigma)}{\langle\,\mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\ sender)} \langle\,\mathsf{running},\ data',\ \mathsf{list}\colon\mathsf{tl}(\sigma),\ db\,\rangle} \tag{8.7}$$

$$\begin{array}{rrcl}
\textbf{where} & \langle file, item, _\rangle & = & \mathsf{event}\colon\mathsf{data}(\mathsf{list}\colon\mathsf{hd}(\sigma)) \\
& data' & = & \mathsf{update_db}(\langle file, \\
& & & \quad\quad \{element \mid \\
& & & \quad\quad\ \ element \in \mathsf{file}(file, data) \wedge \\
& & & \quad\quad\ \ (element \neq item)\}\rangle, \\
& & & \quad\quad data) \\
& m & = & \langle\langle\mathsf{sdf},\mathsf{report}\rangle, \mathsf{remove}, \langle\mathsf{success},\perp\rangle\rangle \\
& sender & = & \mathsf{event}\colon\mathsf{sender}(\mathsf{list}\colon\mathsf{hd}(\sigma))
\end{array}$$

The following rule is applicable when the requested operation is $\mathsf{replace}$. The indicated element is replaced with received one, and $\mathsf{successis}$ reported to the requester.

$$\frac{\mathsf{action}(\mathsf{replace},\sigma)}{\langle\,\mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\ sender)} \langle\,\mathsf{running},\ data',\ \mathsf{list}\colon\mathsf{tl}(\sigma),\ db\,\rangle} \tag{8.8}$$

$$\begin{array}{rrcl}
\textbf{where} & \langle file, item, v\rangle & = & \mathsf{event}\colon\mathsf{data}(\mathsf{list}\colon\mathsf{hd}(\sigma)) \\
& data' & = & \mathsf{update_db}(\langle file, \mathsf{file}(file, data) \oplus \langle item, v\rangle\rangle, data) \\
& m & = & \langle\langle\mathsf{sdf},\mathsf{report}\rangle, \mathsf{replace}, \langle\mathsf{success},\perp\rangle\rangle \\
& sender & = & \mathsf{event}\colon\mathsf{sender}(\mathsf{list}\colon\mathsf{hd}(\sigma))
\end{array}$$

ARITHMETIC RULES

The following rule is applicable when the requested operation is $\mathsf{increment}$. The indicated element is incremented with received amount, and success is reported to the requester.

$$\frac{\mathsf{action}(\mathsf{increment},\sigma)}{\langle\,\mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\ sender)} \langle\,\mathsf{running},\ data',\ \mathsf{list}\colon\mathsf{tl}(\sigma),\ db\,\rangle} \tag{8.9}$$

$$\begin{array}{rrcl}
\textbf{where} & \langle file, item, v\rangle & = & \mathsf{event}\colon\mathsf{data}(\mathsf{list}\colon\mathsf{hd}(\sigma)) \\
& file' & = & \mathsf{file}(file, data) \oplus \langle item, \mathsf{elt}(file, item, data) + v\rangle \\
& data' & = & \mathsf{update_db}(\langle file, file'\rangle, data) \\
& m & = & \langle\langle\mathsf{sdf},\mathsf{report}\rangle, \mathsf{increment}, \langle\mathsf{success},\perp\rangle\rangle \\
& sender & = & \mathsf{event}\colon\mathsf{sender}(\mathsf{list}\colon\mathsf{hd}(\sigma))
\end{array}$$

The following rule is applicable when the requested operation is decrement. The indicated element is decremented with received amount, and success is reported to the requester.

$$\frac{\mathsf{action}(\mathsf{decrement}, \sigma)}{\langle\, \mathsf{running},\ data,\ \sigma,\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\ sender)} \langle\, \mathsf{running},\ data',\ \mathsf{list}\colon \mathsf{tl}(\sigma),\ db\,\rangle} \tag{8.10}$$

$$\begin{array}{rrcl}
\textbf{where} & \langle file, item, v\rangle & = & \mathsf{event}\colon \mathsf{data}(\mathsf{list}\colon \mathsf{hd}(\sigma)) \\
 & file' & = & \mathsf{file}(file, data) \oplus \langle item, \mathsf{elt}(file, item, data) - v\rangle \\
 & data' & = & \mathsf{update_db}(\langle file, file'\rangle, data) \\
 & m & = & \langle\langle \mathsf{sdf}, \mathsf{report}\rangle, \mathsf{decrement}, \langle \mathsf{success}, \perp\rangle\rangle \\
 & sender & = & \mathsf{event}\colon \mathsf{sender}(\mathsf{list}\colon \mathsf{hd}(\sigma))
\end{array}$$

Chapter 9

SRF - Specialized Resources Function

In this chapter we will describe the specialized resources function, first informally it role and the scope of the model. Thereafter follows a more rigorous although annotated description.

9.1 DESCRIPTION

The SRF embodies all the extra functionality to communicate with the user to provide IN services. The SRF is defined in [IN:Q.1204, p.4]:

> The SRF provides the specialized resources required for the execution of IN provided services (e.g. digit receivers, announcements, conference bridges, etc.).It
>
> a) interfaces and interacts with the SCF SSF (and with the CCF);
>
> b) is managed, updated and/or otherwise administered by a SMF;
>
> c) may contain logic and processing capability to receive/send and convert information received from users;
>
> d) may contain functionality similar to the CCF to manage bearer connections to the specialized resources.

In this thesis the SRF is viewed as an additional source of communication with the user, that is sent requests of announcements and collects messages from the user. This is a huge simplification of the arbitrarily complex SRF but captures the properties we are concerned with in this investigation.

9.2 DOMAINS

The SRF has a simple state the *SRFMode* contain the two constants running and waiting where the waiting state is when the SRF is waiting for an answer. The *SRFData* contains the identity of the originators of requests that demands replies.

$$
\begin{array}{lcl}
\mathit{SRFState} & \equiv & \mathit{SRFMode} \times \mathit{SRFData} \times \\
 & & \mathsf{event}\colon \mathit{EventQueue} \times \mathit{SRFDataBase} \\
 & & \\
\mathit{SRFMode} & \equiv & \{\mathsf{running}, \mathsf{waiting}\} \\
\mathit{SRFData} & \equiv & \mathsf{event}\colon \mathit{OriginatorId}_{\perp} \\
\mathit{SRFDataBase} & \equiv & \{\perp\}
\end{array}
$$

The $SRFEventName$ domain are the names of those events accepted or issued by the SRF.

$$
\mathit{SRFEventName} \equiv \{\mathsf{announcement}, \mathsf{announce\&reply}, \mathsf{reply}\}
$$

9.3 FUNCTIONS

The reply function traverses the event queue and checks whether there is any reply in the queue. If there is a reply it returned otherwise $\perp$ is returned.

$$
\mathsf{get\_reply}(\sigma) \stackrel{\mathrm{def}}{=} \begin{cases} \perp & \mathsf{if\ list\colon null}(\sigma) \\ \mathsf{list\colon hd}(\sigma) & \mathsf{if}\ \mathit{notification} \\ \mathsf{reply}(\mathsf{list\colon tl}(\sigma)) & \mathsf{otherwise} \end{cases}
$$
$$
\textbf{where}\quad \mathit{notification} = \mathsf{event\colon notification}(\mathsf{phone}, \mathsf{reply}, \mathsf{list\colon hd}(\sigma))
$$

9.4 RULES

The following rule is applicable when a request for an announcement has been received. The *message* is sent to the appropriate *phone*.

$$(9.1)\quad \frac{\mathsf{event\colon request(scf, announcement, list\colon hd(}\sigma\mathsf{))}}{\langle \mathsf{running},\ \bot,\ \sigma,\ \bot\rangle \xrightarrow{\mathsf{send}\ (m,\, phone)} \langle \mathsf{running},\ \bot,\ \mathsf{list\colon tl}(\sigma),\ \bot\rangle}$$

$$\begin{array}{lrcl} \textbf{where} & \langle phone, message\rangle & = & \mathsf{event\colon data(list\colon hd(}\sigma\mathsf{))} \\ & m & = & \langle\langle \mathsf{srf, notification}\rangle, \mathsf{announcement}, message\rangle \end{array}$$

The following rule is applicable when a request for an announcement and reply has been received. The *message* is sent to the *phone* and the mode changes from running to waiting. The data is set to the originator of the request in order to be able to relay the reply.

$$(9.2)\quad \frac{\mathsf{event\colon request(scf, announce\&reply, list\colon hd(}\sigma\mathsf{))}}{\langle \mathsf{running},\ \bot,\ \sigma,\ \bot\rangle \xrightarrow{\mathsf{send}\ (m,\, phone)} \langle \mathsf{waiting},\ sender,\ \mathsf{list\colon tl}(\sigma),\ \bot\rangle}$$

$$\begin{array}{lrcl} \textbf{where} & \langle phone, message, & & \\ & type, min, max\rangle & = & \mathsf{event\colon data(list\colon hd(}\sigma\mathsf{))} \\ & m & = & \langle\langle \mathsf{srf, notification}\rangle \\ & & & \ \mathsf{announce\&reply} \\ & & & \ \langle message, type, min, max\rangle\rangle \\ & sender & = & \mathsf{event\colon sender(list\colon hd(}\sigma\mathsf{))} \end{array}$$

The following rule is applicable when the mode is waiting after the handling of an announcement and reply request has been initiated and a reply is present in the queue. The reply message is relayed to the originator of the request and the mode is once again set to running.

$$(9.3)\quad \frac{reply \neq \bot}{\langle \mathsf{waiting},\ to,\ \sigma,\ \bot\rangle \xrightarrow{\mathsf{send}\ (m,\, to)} \langle \mathsf{running},\ \bot,\ \mathsf{list\colon remove}(reply, \sigma),\ \bot\rangle}$$

$$\begin{array}{lrcl} \textbf{where} & reply & = & \mathsf{get\_reply}(\sigma) \\ & m & = & \langle\langle \mathsf{srf, response}\rangle, \mathsf{announce\&reply}, \mathsf{event\colon data}(reply)\rangle \end{array}$$

Part III

Building Blocks

SIBs - Service Independent Building Blocks

In this chapter we will describe the Service independent building blocks and the framework we use to build them into services: the service instance.

10.1 DESCRIPTION

The SIBs are the building stones from which one constructs services, they are meant to encapsulate the functionalities needed to implement the IN services. The SIBs are defined in [IN:Q.1213, p.1] as:

> SIBs are abstract representations of network capabilities that exist in an IN structured network. Their definition enforces the concept of service and technology independence by decoupling the service from technology on which services are provided.

The normal call control is modeled as a special SIB, namely the BCP SIB already described in Chapter 6, which also has the capability to decide when IN functionality is needed.

The Services are realised by sequences of SIBs that, when IN functionality is needed, takes over control from the BCP SIB and returns control to the BCP SIB when the SIB sequence terminates. The immediate execution environment is here called the *service instance* which will be treated later in Section 10.2 when we have described the overall structure of the SIBs.

10.1.1 The structure of the SIBs

A SIB is described through its inputs, outputs and the operation performed on the inputs and outputs (with the exception of BCP SIB. The inputs and outputs can be divided into three distinct classes:

Control Each SIB has one logical start indicating the start of execution, and several logical ends which always include the `Error` logical end. The logical ends serve to indicate the nature of the termination and allow for branching on different outcomes. E.g., Verify SIB[1] has two non-erroneous logical ends: pass and fail, which indicate success or failure to pass the verification, respectively.

Service Support Data (SSD) SSD is static information supplied when instantiating the SIB. The SSD contains information about the input arguments of the SIB, such as their type and the location where they are stored in the local environment. The locations and types of the resulting output are also given as SSD. Constants may also be supplied, such as the format that the Verify SIB checks against.

Call Instance Data (CID) CID is the dynamic data, and will be accessible to the SIB during its execution. Where the dynamic data is to be found is given by static pointers to the local environment. An example of dynamic data used is the information checked by the Verify SIB.

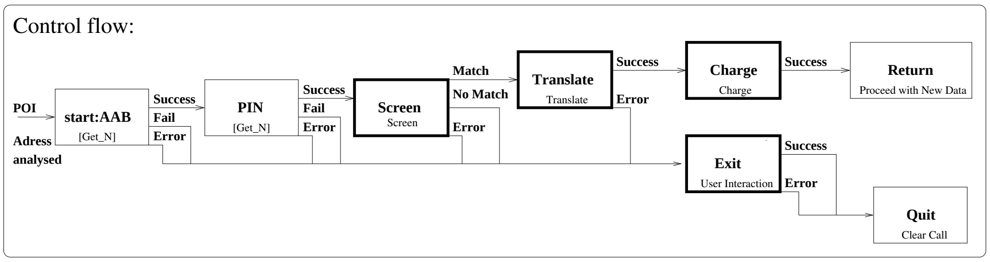
Call Instance Data Field Pointer (CIDFP) CIDFP is SSD that contains the information where the CID is placed in the local store of the service process. So if there is a CID called value then there will also be one called CIDFP − value that tells where to find value.

The control flow between SIBs is specified by the connection of logical ends and starts, and the data is passed through a SIB chain using common locations for input parameters and corresponding output. Below in Figure 10.1 this is exemplified graphically. The figure depicts the Automatic Alternative billing service (AAB). The AAB service transfers the charging of a call to some other account than the default account. First the user is queried for the account to charge, and then for a personal identification number (PIN). The account is screened against the PIN to verify that the transfer is allowed. Then the actual system account is found using the Translate SIB. The system account, together with the information that this account should be charged with the whole cost, is supplied to the Charge SIB which makes the necessary steps.

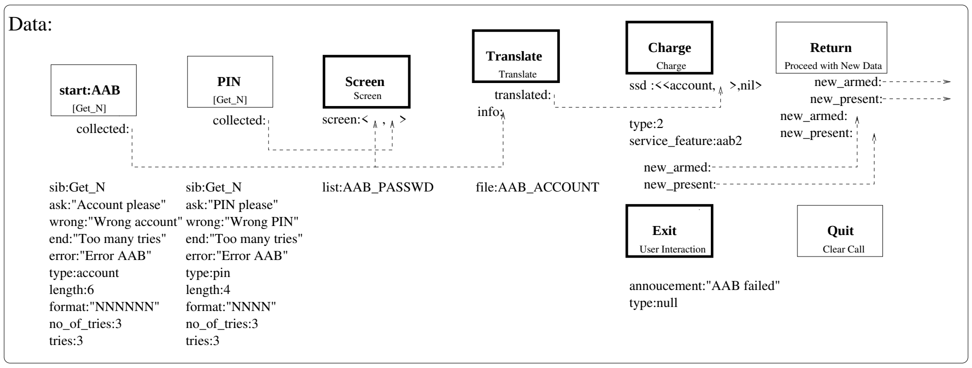
The flow of control is indicated by arrows connecting the SIBs. The outgoing control is labeled with the logical end (name of that control transfer).

The static data is presented the lower part figure of the figure where the SSD is listed below the SIBs which is belongs to as a list of “name : value” pairs. The dynamic CID is annotated through the dotted arcs between the SIBs, e.g., *collected* that is output by the **start:AAB** and input by **Translate** as its *info* CID. The dynamic data may also be denoted by the value of the static information associated with each CID namely the call instance data field pointer (CIDFP). The

[1] Verify SIB will be used as example SIB, this SIB is used to confirm that received information has the correct syntactical format.



(a) Control



(b) Data

Figure 10.1: Example: Control flow in the Automatic alternative billing service

CIDFP notes what place in the local store of the service instance this data will be placed.

There are two dataflows from the "proceed with new data" POR that leaves at the right hand side of the figure, these are the values that are returned from the service to the initiator when the service terminates so that the switch can be updated.

Similarly any information supplied by the invoker when invoking of the service would have been listed at the left of the first SIB in the chain, e.g., see Figure 14.1 depicting the Normal billing service feature.

10.1.2 Interpretation

When formalising the SIB we had to interpret the descriptions rather than just formalise them. This is due to two problems: overspecification and underspecification.

In order that the CS-1 description should not hem in further development it has been made deliberately vague; this is of course a problem when trying to formalise the notions therein. One example of underspecification is the Charge SIB which is defined in [IN:Q.1213] as:

> Determine special charging treatment for the call, where special refers to any charging in addition to that normally performed by the basic process.

In the example above one has to identify some typical cases and yet try to be general enough in the description to handle most sensible cases that may arise. But it is obviously impossible to capture the [IN:Q.1213] definition entirely. Some help to identify typical cases has been gained by studying the description of the Charge SIB at the level of the distributed functional plane in [IN:Q.1214].

As an example of overspecification with respect to analysis, we find that the definition of the Verify SIB given in [IN:Q.1213] gives an explicit coding of the format to be checked against. For purposes of analysis, such an explicit coding is unnecessary and even a source of complication, although perhaps necessary for simulation depending on the level of abstraction. We have in this case chosen to limit the model of SIBs to their essential functionality.

10.1.3 Capability Set - 1

The CS-1 SIBs are:

Algorithm: Applies a mathematical algorithm to data to produce data result.

Basic call process: A specialized SIB which provides the basic call properties.

Charge: Determine special charging treatment for the call.

Compare: Performs a comparison of an identifier against a specified reference value.

Distribution: Distribute calls to different logical ends of the SIB based on user specified parameters.

Limit: Limit the number of calls to IN provided service features based on user specified parameters.

Log call information: Log detailed information for each call into a file.

Queue: Providing sequencing of IN calls to be completed to a call party.

Screen: Perform a comparison of an identifier against a list to determine whether the identifier has been found in the list.

Service data management: Enables end user specific data to be replaced, retrieved, incremented or decremented.

Status notification: Provide the capability of inquiring about the status and/or status changes of network resources.

Translate: Determine output information from input information.

User interaction: Allows information to be exchanged between the network and a call party.

Verify: Provide confirmation that information received is syntactically consistent with the expected form of such information.

Of these, only **Log call informationl**, **Queue** and **Status notification** has not been implemented. The **Log call information** SIB did not seem necessary at this time and the **Queue** SIB we considered to be too unclear and complex. The **Status notification** SIB is highly complex and not very relevant to user services.

10.1.4 Capability Set - 2

The CS-2 SIBs are:

Attach: Attach previously detached resource.

Authenticate: Provides authentication functionality for a service.

Create Service Process: Creates a new service process.

Detach: Detach resource.

Service Filter: Filters the number of calls related to IN provided service features.

End: Terminates a service process.

Join: Join the voice path.

Send: Send a message to another Service process.

Split: Split the voice path.

Wait Wait for and receive a message from another Service process.

Of these, only **Create service process** and **end** has been implemented for want of time.

10.2 THE SERVICE INSTANCE

The Service instance is the context in which the service and consequently the SIBs execute; it defines the state of the "service process". Various functions to access and manipulate the state is defined, and also the functions modeling the global service logic needed to construct services out of the individual SIBs.

10.2.1 Domains

The state of the Service instance contains the mode, dynamic data, event queue and database. The mode of the Service instance is either the normal running or waiting for the reply of some other functional entity or stopped when the Service instance has terminated.

The dynamic data consists of the *ServicePC* that keeps track of where in the execution of the service the Service instance is, and the *Frame* that contain the local data. The Dump makes up the final part and is used as a stack to store the current state when entering a hierarchical SIB.

The database contains definitions of all the services, service features and hierarchical SIBs. Furthermore the database contains the id of the BCSM that originated the Service instance.

$$
\begin{array}{lcl}
\mathit{ServiceInstanceState} & \equiv & \mathit{ServiceInstanceMode} \times \\
 & & \mathit{ServiceInstanceData} \times \\
 & & \mathsf{event}{:}\, \mathit{EventQueue} \times \\
 & & \mathit{ServiceInstanceDataBase} \\
 & & \\
\mathit{ServiceInstanceMode} & \equiv & \{\mathsf{running}, \mathsf{waiting}, \mathsf{stopped}\} \\
\mathit{ServiceInstanceData} & \equiv & \mathit{ServicePC} \times \mathit{Frame} \times \mathit{Dump} \\
\mathit{ServiceInstanceDataBase} & \equiv & (\mathsf{scf}{:}\, \mathit{ServiceLogicName} \rightarrow \\
 & & \;\mathsf{scf}{:}\, \mathit{ServiceDescription}) \times \\
 & & \mathsf{Id}(\mathsf{bcsm})
\end{array}
$$

The *ServicePC* contains the name of the service together with the name of the SIB currently executing in the service. The *Frame* is simply a functional mapping from the natural numbers to some legally formed *Data*, the simplest of store. The *Dump* is a list of names of services together with the static data that was supplied to them, the *Dump* is used as a stack.

$$
\begin{array}{lcl}
\mathit{ServicePC} & \equiv & \mathsf{scf}{:}\, \mathit{ServiceLogicName} \times \mathit{Id} \\
\mathit{Frame} & \equiv & \mathbb{N} \rightarrow \mathit{Data} \\
\mathit{Dump} & \equiv & \mathsf{List}(\mathsf{scf}{:}\, \mathit{ServiceLogicName} \times \mathsf{scf}{:}\, \mathit{SSD})
\end{array}
$$

The $SIBId$ contains the names of the types of SIBs.

$$\begin{array}{rcl} SIBId & \equiv & \{\mathsf{algorithm}, \mathsf{charge}, \mathsf{compare}, \mathsf{create}, \mathsf{distribution}, \\ & & \mathsf{end}, \mathsf{limit}, \mathsf{screen}, \mathsf{sdm}, \mathsf{translate}, \mathsf{verify}, \\ & & \mathsf{continue\_with\_existing\_data}, \mathsf{proceed\_with\_new\_data}, \\ & & \mathsf{call\_clear}, \mathsf{initiate\_call}, \mathsf{hierarchy}, \mathsf{unhierarchy}\} \end{array}$$

The $SIEventName$ domain are the names of those events accepted or issued by the Service instances.

$$\begin{array}{rcl} SIEventName & \equiv & \{\mathsf{clear\_call}, \mathsf{continue\_with\_existing\_data}, \mathsf{initiate\_call}, \mathsf{poi}, \\ & & \mathsf{proceed\_with\_new\_data}, \mathsf{report}, \mathsf{request}\} \cup \\ & & \mathsf{sdf}\colon SDFAction \cup \mathsf{srf}\colon SRFEventName \end{array}$$

10.2.2 Constants

Defined in Chapter 13 Services is the constant Billing_DPs used in this chapter. The Billing_DPs defines the detection points where normal billing takes place. The constant is used by the charge SIB when overriding normal billing.

10.2.3 Functions

The current functions and the frame, dump and lookup functions are selection functions.

The current function selects the service pc from the *data*. The current_service selects the name of the service being run by the Service Instance from *data*. The current_sib_name selects the name of the sib being run by the Service Instance from *data*.

The current_instance looks up in the database (*db*) what the current SIB instance is of the service being run by the Service Instance using the name found in *data*. The instance contains the name of type of the SIB and the SSD of the SIB. The current_sib function looks up the name of the type of the current SIB instance, and the current_ssd looks up the SSD of the current SIB instance

The frame function selects the frame from *data*. The lookup function looks up the element (*loc*) in the frame of *data*.

Finally the dump function selects the dump from *data*.

$$
\begin{array}{lll}
\mathsf{current}(data) & \stackrel{\text{def}}{=} & \pi_1\,(data) \\
\mathsf{current\_service}(data) & \stackrel{\text{def}}{=} & \pi_1\,(\mathsf{current}(data)) \\
\mathsf{current\_sib\_name}(data) & \stackrel{\text{def}}{=} & \pi_2\,(\mathsf{current}(data)) \\
\mathsf{current\_instance}(data, db) & \stackrel{\text{def}}{=} & service\_def(\mathsf{current\_service\_name}(data)) \\
 & \textbf{where} & service\_def = \mathsf{desc}(\mathsf{current\_service}(data), db) \\
\mathsf{current\_sib}(data, db) & \stackrel{\text{def}}{=} & \pi_1\,(\mathsf{current\_instance}(data, db)) \\
\mathsf{current\_ssd}(data, db) & \stackrel{\text{def}}{=} & \pi_2\,(\mathsf{current\_instance}(data, db)) \\
 & & \\
\mathsf{frame}(data) & \stackrel{\text{def}}{=} & \pi_2\,(data) \\
\mathsf{lookup}(loc, data) & \stackrel{\text{def}}{=} & (\mathsf{frame}(data))(loc) \\
 & & \\
\mathsf{dump}(data) & \stackrel{\text{def}}{=} & \pi_3\,(data)
\end{array}
$$

The desc function looks up the definition of service with *name* in the database (db). The initiator function looks up who initiated this Service Instance.

$$
\begin{array}{lll}
\mathsf{desc}(name, db) & \stackrel{\text{def}}{=} & (\pi_1\,(db))(name) \\
\mathsf{initiator}(db) & \stackrel{\text{def}}{=} & \pi_2\,(db)
\end{array}
$$

The sib function tests whether the current SIB is of type *sib*.

$$
\mathsf{sib}(sib, data, db) \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \mathsf{if\ current\_sib}(data, db) = sib \\ \mathsf{false} & \mathsf{otherwise} \end{cases}
$$

GLOBAL SERVICE LOGIC

The logical_end function updates the dynamic data with a new service pc, having the same service name but the id of the next SIB in the chain along logical end *name*. The id of the next SIB in the chain is looked up in the SSD using the ssd function. The functions success and error is the special cases of logical_end where the *name* is success and error respectively.

$$
\begin{array}{lll}
\mathsf{logical\_end}(name, data, db) & \stackrel{\text{def}}{=} & \langle\langle \mathsf{current\_service}(data), \mathsf{ssd}(name, data, db)\rangle, \\
 & & \ \mathsf{frame}(data), \\
 & & \ \mathsf{dump}(data)\rangle \\
\mathsf{success}(data, db) & \stackrel{\text{def}}{=} & \mathsf{logical\_end}(\mathsf{success}, data, db) \\
\mathsf{error}(data, db) & \stackrel{\text{def}}{=} & \mathsf{logical\_end}(\mathsf{error}, data, db)
\end{array}
$$

The update_frame produces new dynamic data for the Service Instance with the frame updated according to update (U). The two store functions change the frame and associates new *value* with the locations. The first store function associates loc with value and the second does similarly with a list (*updates*) of location value pairs.

$$\begin{aligned}
\mathsf{update_frame}(U, data) &\stackrel{\text{def}}{=} \langle \mathsf{current}(data), \mathsf{frame}(data) \oplus U, \mathsf{dump}(data)\rangle \\
\mathsf{store}(loc, value, data, db) &\stackrel{\text{def}}{=} \mathsf{update_frame}(\{\langle place, value\rangle\}, data) \\
\textbf{where} &\quad place = \mathsf{location}(loc, data, db) \\
\mathsf{store}(updates, data, db) &\stackrel{\text{def}}{=} \mathsf{update_frame}(new, data) \\
\textbf{where} &\quad new = \textstyle\bigcup_{\langle l,v\rangle \in updates}\{\langle \mathsf{location}(l, data, db), v\rangle\}
\end{aligned}$$

The function location and help function location1 looks up the location of SSD *name* in the current frame.

$$\begin{aligned}
\mathsf{location}(name, data) &\stackrel{\text{def}}{=} \mathsf{location1}(name, \\
&\qquad\qquad \mathsf{current_ssd}(data, db), \\
&\qquad\qquad \mathsf{dump}(data))
\end{aligned}$$

$$\begin{aligned}
\mathsf{location1}(name, ssd, dump) &\stackrel{\text{def}}{=} \begin{cases} out & \text{if } parameter \\ loc & \text{if } ssd(name) = \langle \mathsf{location}, loc\rangle \\ \perp & \text{otherwise} \end{cases} \\
\textbf{where} &\quad out = \mathsf{location1}(n, \\
&\qquad\qquad \pi_2\,(\mathsf{list\colon hd}(dump)), \\
&\qquad\qquad \mathsf{list\colon tl}(dump)) \\
&\quad parameter = (ssd(name) = \langle \mathsf{parameter}, n\rangle) \wedge \\
&\qquad\qquad (dump \neq \mathsf{nil})
\end{aligned}$$

The ssd function and help function ssd1 selects the service support data named *name* from the database, possibly traversing to the SSDs of outer executions if it is a hierarchical SIB. If there is no SSD named *name*, $\perp$ is returned.

$$\mathsf{ssd}(name, data, db) \stackrel{\text{def}}{=} \mathsf{ssd1}(name, \mathsf{current\_ssd}(data, db), \mathsf{dump}(data))$$

$$\begin{aligned}
\mathsf{ssd1}(name, ssd, dump) &\stackrel{\text{def}}{=} \begin{cases} out_1 & \text{if } (ssd(name) = \perp) \wedge (dump \neq \mathsf{nil}) \\ out_2 & \text{if } (ssd(name) = \langle \mathsf{parameter}, n\rangle) \wedge \\ & \quad (dump \neq \mathsf{nil}) \\ value & \text{if } ssd(name) = \langle \mathsf{immediate}, value\rangle \\ \perp & \text{otherwise} \end{cases} \\
\textbf{where} &\quad out_1 = \mathsf{ssd1}(name, \\
&\qquad\qquad \pi_2\,(\mathsf{list\colon hd}(dump)), \\
&\qquad\qquad \mathsf{list\colon tl}(dump)) \\
&\quad out_2 = \mathsf{ssd1}(n, \pi_2\,(\mathsf{list\colon hd}(dump)), \mathsf{list\colon tl}(dump))
\end{aligned}$$

The cid function and help function cid1 select the call instance data named *name* and returns the CID and the updated dynamic data. The functions may traverse the frames of outer executions if it is a hierarchical SIB.

The cid function can construct the value it retrieves out of several places of the frame. The cid function may also return part of the value, such as the head of a list.

Not only can the cid function construct values, but it can also have side effects, such as the pop which returns the head of the list and sets the value of that frame position to the tail of the list. This possibility of changing the values in the frame is the reason that the cid function returns data, that may have a updated frame.

If there is no CID named *name* the current dynamic data and $\perp$ is returned.

$$\mathsf{cid}(name, data, db) \stackrel{\text{def}}{=} \begin{cases} \langle \perp, data \rangle & \text{if } cid = \perp \\ \langle val, data \rangle & \text{if } cid = \langle \mathsf{immediate}, val \rangle \\ \langle \mathsf{lookup}(loc, data), data \rangle & \text{if } cid = \langle \mathsf{location}, loc \rangle \\ \mathsf{cons}(cons, data, db) & \text{if } cid = \langle \mathsf{cons}, cons \rangle \\ \mathsf{car}(cons, data, db) & \text{if } cid = \langle \mathsf{car}, cons \rangle \\ \mathsf{cdr}(cons, data, db) & \text{if } cid = \langle \mathsf{cdr}, cons \rangle \\ \mathsf{pop}(cons, data, db) & \text{if } cid = \langle \mathsf{pop}, cons \rangle \end{cases}$$
$$\textbf{where} \quad cid = \mathsf{cid1}(name, \mathsf{current\_ssd}(data, db), \mathsf{dump}(data))$$

$$\mathsf{cid1}(name, ssd, h) \stackrel{\text{def}}{=} \begin{cases} out & \text{if } (ssd(name) = \langle \mathsf{parameter}, n \rangle) \wedge \\ & \quad (h \neq \mathsf{nil}) \\ ssd(name) & \text{if } ssd(name) \neq \perp \\ \perp & \text{otherwise} \end{cases}$$
$$\textbf{where} \quad out = \mathsf{cid1}(n, \pi_2\,(\mathsf{hd}(h)), \mathsf{tl}(h))$$

The following functions are help functions to the cid function and handles combination and selection from the data. The cons function concatenates two values from different locations. The car function selects the first in a pair found in the location or $\perp$ should it not be pair, likewise the cdr function selects the second of in a pair.

The pop function selects the first from a pair much like the car function but also changes the dynamic data by saving the second part of the pair back in the location.

$$
\begin{array}{lll}
\mathsf{cons}(\langle car, cdr\rangle, data, db) & \stackrel{\text{def}}{=} & \begin{cases} \langle \bot, data\rangle & \text{if } (r1 = \bot) \vee (r2 = \bot) \\ \langle\langle r1, r2\rangle, data''\rangle & \text{otherwise} \end{cases} \\
& \textbf{where} & \langle r1, data'\rangle = \mathsf{cid}(car, data, db) \\
& & \langle r2, data''\rangle = \mathsf{cid}(cdr, data', db)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{car}(cons, data, db) & \stackrel{\text{def}}{=} & \begin{cases} \langle car, data'\rangle & \text{if } r = \langle car, \_\rangle \\ \langle \bot, data\rangle & \text{otherwise} \end{cases} \\
& \textbf{where} & \langle r, data'\rangle = \mathsf{cid}(cons, data, db)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{cdr}(cons, data, db) & \stackrel{\text{def}}{=} & \begin{cases} \langle cdr, data'\rangle & \text{if } r = \langle \_, cdr\rangle \\ \langle \bot, data\rangle & \text{otherwise} \end{cases} \\
& \textbf{where} & \langle r, data'\rangle = \mathsf{cid}(cons, data, db)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{pop}(cons, data, db) & \stackrel{\text{def}}{=} & \begin{cases} \langle car, data'\rangle & \text{if } \mathsf{lookup}(loc, data) = \langle car, cdr\rangle \\ \langle \bot, data\rangle & \text{otherwise} \end{cases} \\
& \textbf{where} & loc = \mathsf{location}(cons, data, db) \\
& & \quad data' = \mathsf{update\_frame}(\{loc, cdr\}, data)
\end{array}
$$

The function arm adds the *event* at detection points *dps* to *armed*. The function disarm removes the *event* at detection points *dps* to *armed*.

$$
\begin{array}{lll}
\mathsf{arm}(dps, event, armed) & \stackrel{\text{def}}{=} & armed \cup (\bigcup_{dp \in dps} \{\langle dp, event\rangle\}) \\
\mathsf{disarm}(dps, event, armed) & \stackrel{\text{def}}{=} & armed \setminus \bigcup_{dp \in dps} \{\langle dp, event\rangle\}
\end{array}
$$

10.3 ALGORITHM

Description

This SIB description is based on [IN:Q.1213, section 2.1, pp.3]:

> Applies a mathematical algorithm to data to produce a data result.

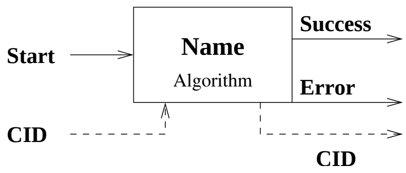


Figure 10.2: Algorithm SIB

Input

Logical Start

Service Support Data

- **Type** The type of algorithm applied, in CS-1 two have been identified:
  a) increment;
  b) decrement.

  We use a nameless function, i.e. a λ-expression.
- **Value** Specifies the amount to be used when applying the SIB.
- **CIDFP-Algorithm** Specifies the location of the data to which the algorithm is to be applied.
- **CIDFP-Identifier** Specifies the location to which the resulting value should be stored.

Call Instance Data

- **Algorithm** The data to which the algorithm is to be applied.

Output

Logical End

- Success
- Error

Call Instance Data

- **Identifier** Contains the resulting value after the SIB has completed.

A graphical representation of the Algorithm SIB is given in Figure 10.2.

RULE

$$\frac{\mathsf{sib}(\mathsf{algorithm}, data, db)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db \,\rangle \ \longrightarrow\ \langle\, \mathsf{running},\ data'',\ \mathsf{nil},\ db \,\rangle} \tag{10.1}$$

$$\begin{array}{lrcl} \textbf{where} & function & = & \mathsf{ssd}(\mathsf{type}, data, db) \\ & amount & = & \mathsf{ssd}(\mathsf{value}, data, db) \\ & \langle args, data' \rangle & = & \mathsf{cid}(\mathsf{cidfp{-}algorithm}, data, db) \\ & result & = & function(amount)(args) \\ & data'' & = & \mathsf{ifundef}(result, \mathsf{error}(data, db), success) \\ & success & = & \mathsf{success}(\mathsf{store}(\mathsf{cidfp{-}identifier}, result, data', db), db) \end{array}$$

10.4 CHARGE

Description

This SIB description is based on [IN:Q.1213, section 2.2, pp.4]:

> Determine special charging treatment for the call, where special refers to any charging in addition to that normally performed by the basic call process
>
> In general, this involves identifying
>
> - the resources for which charging is to occur;
> - to where the charges are to be directed.
>
> It should be noted that this SIB is not responsible for the subscriber billing process.

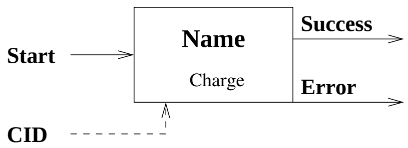


Figure 10.3: Charge SIB

The charge SIB can handle three different scenarios:

One The special charging can be determined immediately and information is simply furnished to the basic call process. A charge is added or the accounts to be charged for the cost of the call is changed.

Two: the special charging requires information regarding the normal charging to determine correct procedure. The account to be charged for the cost of the call is set, and the normal billing feature is superseded by another service.

Three: this is a scenario *Two* with the exception that the normal billing is not superseded but supplemented by the new service.

In order to be able to specify what scenario is treated the static parameters have been augmented with a Type parameter. Further the static parameters have been augmented with the feature that will be added in scenario two and three, as well as the SSD of the feature.

Furthermore four new output CIDs have been added, the new_cost, new_armed, new_present and new_accounting used by our service activation and billing model.

Input

Logical Start

Service Support Data

- **Type** The type of charge performed.
- **Cost** The cost to be charged.
- **Accounts** Each account is specified by two parameters.
  a) CIDFP-Account or Account
     Account to charge
  b) Description
     Is it an account given, Fixed or is it a CIDFP.
  c) Percent(%)
     How much this account is charged(the sum must always be 100 %.)
- **Service_Feature** The new service feature that either supersedes or supplements the normal billing.
- **CIDFP-SSD** The location of SSD of the service feature.
- **CIDFP-New_Cost** The location that the new cost is stored in.
- **CIDFP-New_Armed** The location that the new arming of services is stored in.
- **CIDFP-New_Present** The location that the new present function is stored in.
- **CIDFP-New_Accounting** The location that the new accounting list is stored in.

Call Instance Data

- **Accounts** The accounts to charge, only present if the SSD Accounts is CIDFPs.
- **SSD** The SSD of the service feature used.

Output

Logical End

- Success
- Error

Call Instance Data

- **New_Cost** The new cost which is added to the cost of the call.
- **New_Armed** The new arming of services.
- **New_Present** The present function.
- **New_Accounting** The new accounting list

In the

A graphical representation of the Charge SIB is given in Figure 10.3.

RULES

$$\frac{\mathsf{sib}(\mathsf{charge}, data, db) \wedge_{\perp} (type = 1)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \ \longrightarrow\ \langle\, \mathsf{running},\ \mathsf{success}(data', db),\ \mathsf{nil},\ db\,\rangle} \tag{10.2}$$

$$\begin{array}{lrcl}
\textbf{where} & type & = & \mathsf{ssd}(\mathsf{type}, data, db) \\
& cost & = & \mathsf{ssd}(\mathsf{cost}, data, db) \\
& ac & = & \mathsf{ssd}(\mathsf{accounts}, data, db) \\
& new & = & \mathsf{ifundef}(ac, \perp, new') \\
& new' & = & \cup_{\langle p, desc, a\rangle \in ac}\{\langle \mathsf{if}(desc = \mathsf{Fixed}, a, \pi_1\,(\mathsf{cid}(a, data, db))), p\rangle\} \\
& data' & = & \mathsf{store}(\{\langle \mathsf{new_cost}, cost\rangle, \langle \mathsf{new_accounting}, new\rangle\}, data, db)
\end{array}$$

$$\frac{\mathsf{sib}(\mathsf{charge}, data, db) \wedge_{\perp} (type = 2)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \ \longrightarrow\ \langle\, \mathsf{running},\ \mathsf{success}(data'', db),\ \mathsf{nil},\ db\,\rangle} \tag{10.3}$$

$$\begin{array}{lrcl}
\textbf{where} & type & = & \mathsf{ssd}(\mathsf{type}, data, db) \\
& armed & = & \mathsf{ssd}(\mathsf{armed}, data, db) \\
& present & = & \mathsf{ssd}(\mathsf{present}, data, db) \\
& feature & = & \mathsf{ssd}(\mathsf{service_feature}, data, db) \\
& \langle ssd, data' \rangle & = & \mathsf{cid}(\mathsf{cidfp{-}ssd}, data, db) \\
& armed' & = & \mathsf{disarm}(\mathsf{services{:}\ Billing_DPs}, \\
& & & \qquad\quad \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{normal_billing}\rangle, \\
& & & \qquad\quad armed) \\
& armed'' & = & \mathsf{arm}(\mathsf{services{:}\ Billing_DPs}, \\
& & & \qquad \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, feature\rangle, \\
& & & \qquad armed') \\
& present' & = & \mathsf{disarm}(\mathsf{services{:}\ Billing_DPs}, \\
& & & \qquad\quad \langle event, \mathsf{bcsm}\rangle, \\
& & & \qquad\quad present) \\
& event & = & \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{service}, \langle \mathsf{normal_billing}, \emptyset\rangle\rangle \\
& present'' & = & \mathsf{arm}(\mathsf{services{:}\ Billing_DPs}, \\
& & & \qquad \langle event', \mathsf{bcsm}\rangle, \\
& & & \qquad present') \\
& event' & = & \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{service}, \langle feature, ssd\rangle\rangle \\
& data'' & = & \mathsf{store}(\{\langle \mathsf{new_armed}, armed''\rangle, \\
& & & \qquad\quad \langle \mathsf{new_present}, present''\rangle, \\
& & & \qquad\quad \langle \mathsf{new_accounting}, \mathsf{nil}\rangle\}, \\
& & & \qquad data', \\
& & & \qquad db)
\end{array}$$

$$\text{(10.4)} \quad \frac{\mathsf{sib}(\mathsf{charge}, data, db) \wedge_{\perp} (type = 3)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \ \longrightarrow\ \langle\, \mathsf{running},\ \mathsf{success}(data'', db),\ \mathsf{nil},\ db\,\rangle}$$

$$\begin{array}{lrcl}
\textbf{where} & type & = & \mathsf{ssd}(\mathsf{type}, data, db) \\
& armed & = & \mathsf{ssd}(\mathsf{armed}, data, db) \\
& present & = & \mathsf{ssd}(\mathsf{present}, data, db) \\
& feature & = & \mathsf{ssd}(\mathsf{service_feature}, data) \\
& \langle ssd, data' \rangle & = & \mathsf{cid}(\mathsf{cidfp}{-}\mathsf{ssd}, data) \\
& armed' & = & \mathsf{arm}(\mathsf{services{:}\,Billing_DPs}, \\
& & & \quad \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, feature\rangle, \\
& & & \quad armed) \\
& present' & = & \mathsf{arm}(\mathsf{services{:}\,Billing_DPs}, \\
& & & \quad \langle\langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{service}, \langle feature, ssd\rangle\rangle, \mathsf{bcsm}\rangle, \\
& & & \quad present) \\
& data'' & = & \mathsf{store}(\{\langle \mathsf{new_armed}, armed'\rangle, \\
& & & \qquad \langle \mathsf{new_present}, present'\rangle, \\
& & & \qquad \langle \mathsf{new_accounting}, \mathsf{nil}\rangle\}, \\
& & & \quad\ data', \\
& & & \quad\ db)
\end{array}$$

10.5 COMPARE

Description

This SIB description is based on [IN:Q.1213, section 2.3, pp.7]:

> Performs a comparison of an identifier against a specified reference value. Three result are possible:
>
> - identifier is GREATER than the value;
> - identifier is LESS than the value;
> - identifier is EQUAL TO the value.

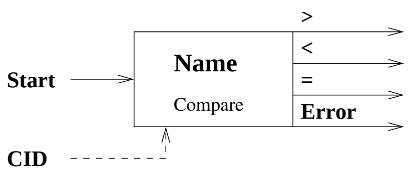


Figure 10.4: Compare SIB

Input

Logical Start

Service Support Data

- **Type** Specifies the type of comparison to be performed. The following types have been identified:

  a) Identifier value – compare the identifier against reference value.

  b) Time – compare network time to reference value.

  We use a nameless function, i.e. λ-expression.

- **CIDFP-Compare** Specifies the location of the data to compare with.
- **Value** Specifies the reference value against which the comparison will be made.

Call Instance Data

- **Compare** The data which will be compared against reference value.

Output

Logical End

- GREATER THAN the value
- LESS THAN the value
- EQUAL TO the value
- Error

A graphical representation of the Compare SIB is given in Figure 10.4.

RULE

$$\frac{\mathsf{sib}(\mathsf{compare}, data, db)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \ \longrightarrow\ \langle\, \mathsf{running},\ data'',\ \mathsf{nil},\ db\,\rangle} \tag{10.5}$$

$$\begin{aligned}
\textbf{where}\quad comparison &= \mathsf{ssd}(\mathsf{type}, data, db)\\
value &= \mathsf{ssd}(\mathsf{value}, data, db)\\
\langle args, data'\rangle &= \mathsf{cid}(\mathsf{cidfp{-}compare}, data, db)\\
res &= comparison(value, args)\\
data'' &= \begin{cases} \mathsf{logical_end}(\mathsf{greater}, data', db) & \mathsf{if}\ res = \mathsf{greater}\\ \mathsf{logical_end}(\mathsf{less}, data', db) & \mathsf{if}\ res = \mathsf{less}\\ \mathsf{logical_end}(\mathsf{equal}, data', db) & \mathsf{if}\ res = \mathsf{equal}\\ \mathsf{error}(data, db) & \mathsf{otherwise}\end{cases}
\end{aligned}$$

10.6 CREATE SERVICE PROCESS

Description

This SIB description is based on [IN:Q.1223, section 5.17, pp.26]:

> Creates a new service process instance to provide parallel processing.

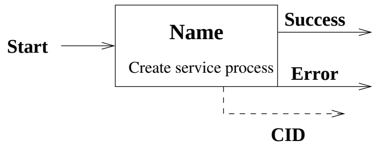


Figure 10.5: Create Service Process SIB

Input

Logical Start

Service Support Data

- **Feature** What service process to start.
- **SSD** Service support data for the created process.
- **CIDFP-PID** What place to store the process id in.

Output

Logical End

- Success
- Error

Call Instance Data

- **PID** The process id of the created process.

We have augmented the static parameters with the SSD of the feature to be executed. The Error logical end would indicate that we have failed to create a new service instance, presently that possibility in not considered.

A graphical representation of the Create Service Process SIB is given in Figure 10.5.

RULES

(10.6)
$$\frac{\mathsf{sib}(\mathsf{create}, data, db)}{\langle\, \mathsf{running},\, data,\, \mathsf{nil},\, db\,\rangle \xrightarrow{\mathsf{send}\ (m,\, \mathsf{scf})} \langle\, mode,\, data,\, \mathsf{nil},\, db\,\rangle}$$

where
$$\begin{array}{rcl} feature & = & \mathsf{ssd}(\mathsf{feature}, data, db) \\ ssd & = & \mathsf{ssd}(\mathsf{ssd}, data, db) \\ loc & = & \mathsf{ssd}(\mathsf{pid}, data, db) \\ m_1 & = & \langle\langle \mathsf{si}, \mathsf{request}\rangle, \mathsf{request}, \langle feature, ssd\rangle\rangle \\ m_2 & = & \langle\langle \mathsf{si}, \mathsf{request}\rangle, \mathsf{poi}, \langle feature, ssd\rangle\rangle \\ \langle mode, data', m\rangle & = & \begin{cases} \langle \mathsf{running}, \mathsf{success}(data, db), m_1\rangle & \text{if } loc = \perp \\ \langle \mathsf{waiting}, data, m_2\rangle & \text{otherwise} \end{cases} \end{array}$$

(10.7)
$$\frac{\mathsf{sib}(\mathsf{create}, data, db)}{\langle\, \mathsf{waiting},\, data,\, \langle m, \mathsf{nil}\rangle,\, db\,\rangle \;\longrightarrow\; \langle\, \mathsf{running},\, data',\, \mathsf{nil},\, db\,\rangle}$$

where
$$\begin{array}{rcl} m & = & \langle\langle\langle \mathsf{scf}, \mathsf{response}\rangle, \mathsf{report}, service\rangle, \mathsf{scf}\rangle \\ data' & = & \mathsf{success}(\mathsf{store}(\mathsf{pid}, service, data, db), db) \end{array}$$

10.7 DISTRIBUTION

Description

This SIB description is based on [IN:Q.1213, section 2.4, pp.8]:

> Distribute calls to different logical ends of the SIB based on user specified parameters.

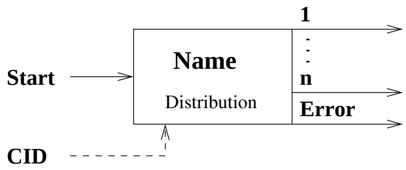


Figure 10.6: Distribution SIB

Input

Logical Start

Service Support Data

- **Algorithm type** Specifies the on what criteria distribution is made:
    a) percentage;

b) sequential;
c) time of day;
d) day of week.

We use a nameless function, i.e. λ-expression.

- **Number of logical ends** This changes the structure of the GSL.
- **Parameters** Depending on the type of algorithm used this will different types of values.
- **Distribution** Which maps the number of the distribution to the name of logical end.

Call Instance Data

- **None**

Output

Logical End

- **Distribution**(1)
- **Distribution**(2)
- ...
- **Distribution**(N)[where N = Number of logical ends]
- Error

We have augmented the SSDs by distribution so that the logical ends can have more informative names then just plain numbers. A graphical representation of the Distribution SIB is given in Figure 10.6.

RULE

(10.8)
$$\frac{\mathsf{sib}(\mathsf{distribution}, data, db)}{\langle\,\mathsf{running},\, data,\, \mathsf{nil},\, db\,\rangle \;\longrightarrow\; \langle\,\mathsf{running},\, data',\, \mathsf{nil},\, db\,\rangle}$$

$$\textbf{where}\quad \begin{array}{rcl} algorithm & = & \mathsf{ssd}(\mathsf{type}, data, db) \\ number & = & \mathsf{ssd}(\mathsf{number}, data, db) \\ parameters & = & \mathsf{ssd}(\mathsf{parameters}, data, db) \\ distribution & = & \mathsf{ssd}(\mathsf{distribution}, data) \\ result & = & algorithm(parameters) \\ data' & = & \mathsf{if}(0 < result \leq number, \\ & & \quad \mathsf{logical\_end}(distribution(result)), data, db), \\ & & \quad \mathsf{error}(data, db)) \end{array}$$

10.8 END

Description

This SIB description is based on [IN:Q.1223, section 5.20, pp.33]:

> Indicates the normal end of an executing service process, or part of a service process in case of multiple threads.

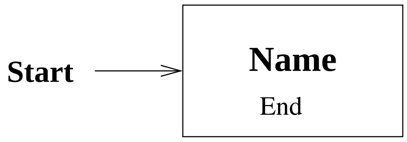


Figure 10.7: End SIB

Input

Logical start

Service Support Data

- **Clear POSs** Items to be removed from the POS (Points of Synchronization)-buffer.

Output

Inter Process Communication

- POR - End(Optional)Identifies the termination of a service process.

The notion of POS mentioned in the SSDs is a part of the extensions in CS-2 to enable multithreaded services, enabling them to synchronise on specific points in the execution. The POS SSD is not required since we do not have multithreaded services. We not needed to use the optional End point of return. A graphical representation of the End SIB is given in Figure 10.7.

RULE

$$\frac{\mathsf{sib}(\mathsf{end}, data, db)}{\langle\, \mathsf{running},\, data,\, \mathsf{nil},\, db \,\rangle \;\longrightarrow\; \langle\, \mathsf{stopped},\, data,\, \mathsf{nil},\, db \,\rangle} \tag{10.9}$$

10.9 LIMIT

Description

This SIB description is based on [IN:Q.1213, section 2.5, pp.10]:

> Limit the number of calls related to IN provided service features. Such limiting will be based on user specified parameters.

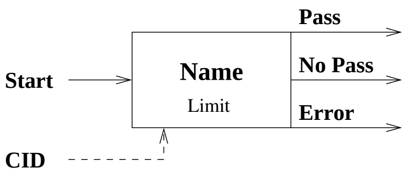


Figure 10.8: Limit SIB

Input

Logical Start

Service Support Data

- **Type** Specifies the type of limiting to be performed. The following types have been identified:

  a) pass calls for (S) seconds out of every (Q) seconds;

  d) pass (N) calls out of (P) calls.

  We use a nameless function, i.e. λ-expression.

- **Parameters** Depending on type.

- **CIDFP-File** This specifies where the limit count is located

Call Instance Data

- **File** Specifies the current count of calls.

Output

Logical End

- Pass

- No pass

- Error

A graphical representation of the Limit SIB is given in Figure 10.8.

RULE

$$\frac{\mathsf{sib}(\mathsf{limit}, data, db)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \ \longrightarrow\ \langle\, \mathsf{running},\ data'',\ \mathsf{nil},\ db\,\rangle} \tag{10.10}$$

$$\begin{array}{lrcl}
\textbf{where} & limit & = & \mathsf{ssd}(\mathsf{type}, data, db) \\
& parameters & = & \mathsf{ssd}(\mathsf{parameters}, data, db) \\
& \langle file, data' \rangle & = & \mathsf{cid}(\mathsf{cidfp{-}file}, data, db) \\
& result & = & limit(parameters)(file) \\
& data'' & = & \begin{cases} \mathsf{logical_end}(\mathsf{pass}, data', db) & \text{if } result = \mathsf{pass} \\ \mathsf{logical_end}(\mathsf{nopass}, data', db) & \text{if } result = \mathsf{nopass} \\ \mathsf{error}(data, db) & \text{otherwise} \end{cases}
\end{array}$$

10.10 SCREEN

Description

This SIB description is based on [IN:Q.1213, section 2.8, pp.17]:

> Perform a comparison of an identifier against a list to determine whether the identifier has been found in the list.

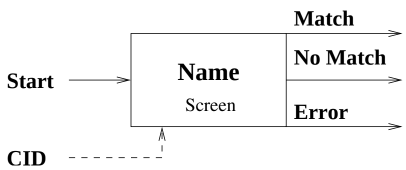


Figure 10.9: Screen SIB

Input

Logical Start

Service Support Data

- **Screen list indicator** Identifies the screen list in the SDF to be used.
- **CIDFP-Screen** Where the data to be screened is to be found.

Call Instance Data

- **Screen** The data to be screened.

Output

Logical End

- Match [= on the list]
- No Match
- Error

A graphical representation of the Screen SIB is given in Figure 10.9.

RULES

$$\text{(10.11)} \qquad \frac{\mathsf{sib}(\mathsf{screen}, data, db)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \xrightarrow{\mathsf{broadcast}\,(m')} \langle\, mode,\ data'',\ \mathsf{nil},\ db\,\rangle}$$

$$\textbf{where} \quad \begin{array}{rcl} list & = & \mathsf{ssd}(\mathsf{list}, data, db) \\ \langle screen, data' \rangle & = & \mathsf{cid}(\mathsf{cidfp{-}screen}, data, db) \\ m & = & \{\langle\langle\langle \mathsf{scf}, \mathsf{request}\rangle, \mathsf{lookup}, \langle list, screen, \bot\rangle\rangle, \mathsf{sdf}\rangle\} \\ \langle mode, data'', m' \rangle & = & \begin{cases} \langle \mathsf{running}, \mathsf{error}(data', db), \emptyset\rangle & \text{if } screen = \bot \\ \langle \mathsf{waiting}, data', m\rangle & \text{otherwise} \end{cases} \end{array}$$

$$\text{(10.12)} \qquad \frac{\mathsf{sib}(\mathsf{screen}, data, db)}{\langle\, \mathsf{waiting},\ data,\ \langle m, \mathsf{nil}\rangle,\ db\,\rangle \longrightarrow \langle\, \mathsf{running},\ data',\ \mathsf{nil},\ db\,\rangle}$$

$$\textbf{where} \quad \begin{array}{rcl} m & = & \langle\langle\langle \mathsf{sdf}, \mathsf{report}\rangle, \mathsf{lookup}, result\rangle, \mathsf{sdf}\rangle \\ data' & = & \begin{cases} \mathsf{logical\_end}(\mathsf{match}, data, db) & \text{if } result = \langle \mathsf{success}, \mathsf{yes}\rangle \\ \mathsf{logical\_end}(\mathsf{nomatch}, data, db) & \text{if } result = \langle \mathsf{success}, \mathsf{no}\rangle \\ \mathsf{error}(data, db) & \text{otherwise} \end{cases} \end{array}$$

10.11 SDM - SERVICE DATA MANAGEMENT

Description

This SIB description is based on [IN:Q.1213, section 2.9, pp.18]:

> Enables end user specific data to be replaced, retrieved, incremented or decremented.

Input

Logical Start

Service Support Data

- **Type** The type of operation.
- **CIDFP-Data** Specific data for the operation.
- **CIDFP-Retrieved** Specifies the location to store retrieved data in.

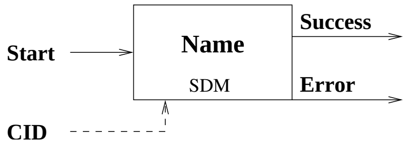


Figure 10.10: Service Data Management SIB

Call Instance Data

- **data** To be used in data management.

Output

Logical End

- Success
- Error

Call Instance Data

- **Retrieved** The retrieved data.

We have added a further static data identifying what database file and the dynamic data specifying what element that is being addressed. There is also the dynamic data value that may be used to change the element, e.g., the amount to decrement the element with. A graphical representation of the SDM SIB is given in Figure 10.10.

RULES

(10.13)
$$\frac{\mathsf{sib}(\mathsf{sdm}, data, db)}{\langle\, \mathsf{running},\, data,\, \mathsf{nil},\, db\,\rangle \xrightarrow{\mathsf{broadcast}\ (m')} \langle\, mode,\, data^3,\, \mathsf{nil},\, db\,\rangle}$$

where

$$\begin{array}{rcl}
file & = & \mathsf{ssd}(\mathsf{file}, data, db) \\
action & = & \mathsf{ssd}(\mathsf{action}, data, db) \\
\langle value, data' \rangle & = & \mathsf{cid}(\mathsf{cidfp{-}value}, data, db) \\
\langle element, data'' \rangle & = & \mathsf{cid}(\mathsf{cidfp{-}element}, data', db) \\
m & = & \{\langle\langle\langle \mathsf{scf}, \mathsf{request}\rangle, action, \langle file, element, value\rangle\rangle, \\
 & & \ \mathsf{sdf}\rangle\} \\
\langle mode, data^3, m' \rangle & = & \mathsf{ifundef}(element, \\
 & & \qquad\qquad \langle \mathsf{running}, \mathsf{error}(data, db), \emptyset\rangle, \\
 & & \qquad\qquad \langle \mathsf{waiting}, data'', m\rangle)
\end{array}$$

$$\text{(10.14)} \qquad \frac{\mathsf{sib}(\mathsf{sdm}, data, db)}{\langle\, \mathsf{waiting},\ data,\ \langle m, \mathsf{nil}\rangle,\ db\,\rangle \ \longrightarrow\ \langle\, \mathsf{running},\ data'',\ \mathsf{nil},\ db\,\rangle}$$

$$\begin{array}{lrcl}
\textbf{where} & m & = & \langle\langle\langle \mathsf{sdf}, \mathsf{report}\rangle, action, report\rangle, \mathsf{sdf}\rangle \\
& \langle result, answer\rangle & = & report \\
& data' & = & \mathsf{ifdef}(\mathsf{ssd}(\mathsf{retrieved}, data), \\
& & & \qquad \mathsf{store}(\mathsf{retrieved}, answer, data, db), \\
& & & \qquad data) \\
& data'' & = & \mathsf{if}(result = \mathsf{success}, \\
& & & \quad \mathsf{success}(data', db), \\
& & & \quad \mathsf{error}(data, db))
\end{array}$$

10.12 TRANSLATE

Description

This SIB description is based on [IN:Q.1213, section 2.11, pp.23]:

> Determines output information from input information.

This somewhat cryptic description means that a simple translation is performed using a translation table.

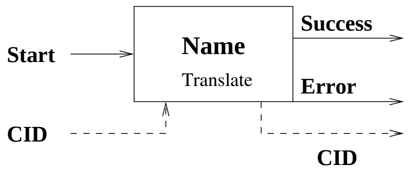


Figure 10.11: Translate SIB

Input

Logical Start

Service Support Data

- **File** What table in the SDF to use for the translation.
- **CIDFP-Info** Where the info is.
- **CIDFP-Translated** Where to store result.

Call Instance Data

- **Info** To be translated.

Output

Logical End

- Success
- Error

Call Instance Data

- **Translated** The translated data.

A graphical representation of the Translate SIB is given in Figure 10.11.

RULES

$$\text{(10.15)}\quad \frac{\mathsf{sib}(\mathsf{translate}, data, db)}{\langle\, \mathsf{running},\, data,\, \mathsf{nil},\, db\,\rangle \xrightarrow{\mathsf{broadcast}\,(m')} \langle\, mode,\, data'',\, \mathsf{nil},\, db\,\rangle}$$

$$\textbf{where}\quad \begin{aligned} file &= \mathsf{ssd}(\mathsf{file}, data, db) \\ \langle info, data'\rangle &= \mathsf{cid}(\mathsf{cidfp{-}info}, data, db) \\ m &= \{\langle\langle\langle \mathsf{scf}, \mathsf{request}\rangle, \mathsf{retrieve}, \langle file, info, \bot\rangle\rangle, \mathsf{sdf}\rangle\} \\ \langle mode, data'', m'\rangle &= \mathsf{ifundef}(info, \\ &\qquad\qquad \langle \mathsf{running}, \mathsf{error}(data, db), \emptyset\rangle, \\ &\qquad\qquad \langle \mathsf{waiting}, data', m\rangle) \end{aligned}$$

$$\text{(10.16)}\quad \frac{\mathsf{sib}(\mathsf{translate}, data, db)}{\langle\, \mathsf{waiting},\, data,\, \langle m, \mathsf{nil}\rangle,\, db\,\rangle \longrightarrow \langle\, \mathsf{running},\, data',\, \mathsf{nil},\, db\,\rangle}$$

$$\textbf{where}\quad \begin{aligned} m &= \langle\langle\langle \mathsf{sdf}, \mathsf{report}\rangle, \mathsf{retrieve}, report\rangle, \mathsf{sdf}\rangle \\ \langle result, answer\rangle &= report \\ data' &= \mathsf{if}(result = \mathsf{success}, \\ &\qquad \mathsf{success}(\mathsf{store}(\mathsf{translated}, answer, data, db), db), \\ &\qquad \mathsf{error}(data, db)) \end{aligned}$$

10.13 UI - USER INTERACTION

Description

This SIB description is based on [IN:Q.1213, section 2.12, pp.25]:

> Allows information to be exchanged between the network and a call party, where a call party can be either a calling or a called party.

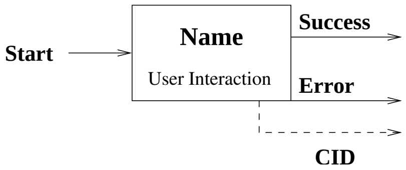


Figure 10.12: User Interaction SIB

Input

Logical Start

Service Support Data

- **Type** Specify the of user information to collected:
  a) account; a account designator
  b) pin; personal identification number
  c) null. no information is to be gathered.
- **Announcement** The annoucement to play to the user.
- **Min** The minimal size of the collected message.
- **Max** The maximal size of the collected message.
- **CIDFP-Collected** Where to store collected information.

Output

Logical End

- Success
- Error

Call Instance Data

- **Collected** The collected data.

A graphical representation of the UI SIB is given in Figure 10.12.

RULES

$$\frac{\mathsf{sib}(\mathsf{ui}, data, db) \wedge_{\perp} (phone = \perp)}{\langle\, \mathsf{running},\, data,\, \mathsf{nil},\, db\,\rangle \;\longrightarrow\; \langle\, \mathsf{running},\, \mathsf{error}(data, db),\, \mathsf{nil},\, db\,\rangle} \tag{10.17}$$

$$\textbf{where} \quad phone \;=\; \mathsf{ssd}(\mathsf{phone}, data, db)$$

$$\frac{\mathsf{sib}(\mathsf{ui}, data, db) \wedge_{\perp} (type = \mathsf{null})}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\mathsf{srf})} \langle\, \mathsf{running},\ data',\ \mathsf{nil},\ db\,\rangle} \tag{10.18}$$

$$\begin{array}{rrcl}
\textbf{where} & message & = & \mathsf{ssd}(\mathsf{announcement}, data, db) \\
& type & = & \mathsf{ssd}(\mathsf{type}, data, db) \\
& phone & = & \mathsf{ssd}(\mathsf{phone}, data, db) \\
& data' & = & \mathsf{success}(data, db) \\
& m & = & \langle\langle \mathsf{scf}, \mathsf{request}\rangle, \mathsf{announcement}, \langle phone, message\rangle\rangle
\end{array}$$

$$\frac{\mathsf{sib}(\mathsf{ui}, data, db) \wedge_{\perp} (type \neq \mathsf{null})}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\mathsf{srf})} \langle\, \mathsf{waiting},\ data,\ \mathsf{nil},\ db\,\rangle} \tag{10.19}$$

$$\begin{array}{rrcl}
\textbf{where} & message & = & \mathsf{ssd}(\mathsf{announcement}, data, db) \\
& type & = & \mathsf{ssd}(\mathsf{type}, data, db) \\
& min & = & \mathsf{ssd}(\mathsf{min}, data, db) \\
& max & = & \mathsf{ssd}(\mathsf{max}, data, db) \\
& phone & = & \mathsf{ssd}(\mathsf{phone}, data, db) \\
& m & = & \langle\langle \mathsf{scf}, \mathsf{request}\rangle \\
& & & \ \mathsf{announce\&reply} \\
& & & \ \langle phone, message, type, min, max\rangle\rangle
\end{array}$$

$$\frac{\mathsf{sib}(\mathsf{ui}, data, db)}{\langle\, \mathsf{waiting},\ data,\ \langle m, \mathsf{nil}\rangle,\ db\,\rangle \ \longrightarrow\ \langle\, \mathsf{running},\ data'',\ \mathsf{nil},\ db\,\rangle} \tag{10.20}$$

$$\begin{array}{rrcl}
\textbf{where} & m & = & \langle\langle\langle \mathsf{srf}, \mathsf{report}\rangle, \mathsf{announce\&reply}, \langle result, input\rangle\rangle, \mathsf{srf}\rangle \\
& data' & = & \mathsf{store}(\mathsf{collected}, input, data, db) \\
& data'' & = & \begin{cases} \mathsf{success}(data', db) & \mathsf{if}\ result = \mathsf{success} \\ \mathsf{error}(data, db) & \mathsf{otherwise} \end{cases}
\end{array}$$

10.14 VERIFY

Description

This SIB description is based on [IN:Q.1213, section 2.13, pp.27]:

> Provide confirmation that information received is syntactically consistent with the expected form of such information.

The purpose of this SIB is typically to check that some particular input data satisfies some restriction on its format, e.g., that a received PIN code consists of four digits. The parameters of the SIBS are

Input

Logical Start

Service Support Data

- **Format** Specifies the format using a language similar to regular expressions.
- **Min** Specifies the minimum length the data may have.
- **Max** Specifies the maximum length the data may have.
- **CIDFP-Data** Where the data to be verified is found.

Call Instance Data

- **Data** The data to be verified.

Output

Logical End

- Pass
- Fail
- Error

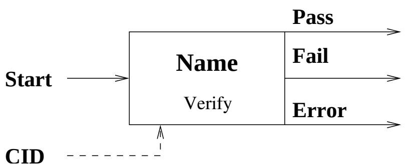


Figure 10.13: Verify SIB

A graphical representation of the Verify SIB is given in Figure 10.13.

RULE

(10.21)
$$\frac{\mathsf{sib}(\mathsf{verify}, data, db)}{\langle\,\mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \ \longrightarrow\ \langle\,\mathsf{running},\ data'',\ \mathsf{nil},\ db\,\rangle}$$

where
$$\begin{aligned}
min &= \mathsf{ssd}(\mathsf{min}, data, db)\\
max &= \mathsf{ssd}(\mathsf{max}, data, db)\\
format &= \mathsf{ssd}(\mathsf{format}, data, db)\\
\langle subject, data'\rangle &= \mathsf{cid}(\mathsf{cidfp{-}data}, data, db)\\
result &= \mathsf{verify}(min, max, format, subject)\\
data'' &= \begin{cases} \mathsf{logical_end}(\mathsf{pass}, data', db) & \mathsf{if}\ result = \mathsf{pass}\\ \mathsf{logical_end}(\mathsf{fail}, data', db) & \mathsf{if}\ result = \mathsf{fail}\\ \mathsf{error}(data, db) & \mathsf{otherwise}\end{cases}
\end{aligned}$$

CHAPTER 11

GSL - GLOBAL SERVICE LOGIC

11.1 DESCRIPTION

From [IN:Q.1213, chapter 4, p.33] we have the following on GSL:

> Global service logic(GSL) has been defined in 5/I.329/Q.1203 as the "glue" that describes the order in which SIBs can be chained together to accomplish services.
>
> For a given CS-1 service/service feature, global service logic is used to describe:
>
> i) A specific POI which will define the functional launching point from the BCP to the SIB chain.
> ii) A specific set of PORs where the SIB chain can logically return to the BCP.
> iii) The pattern and order of SIBs which are to be chained together. This pattern begins at the POI defined in i) and ends at the set of PORs defined in ii).
> iv) Data parameters (SSD and CID) for each SIB in the SIB chain.

The POIs are in our case represented through the detection points in the BCSMs which are described in Section 6.1.2. The PORs are described in Section 6.1.3 and we have so far only modeled **Clear call**, **Continue with existing data**, **Proceed with new data** and **Initiate call**.

Much of the gluing of the dynamic data is modeled by the cid function with help functions found in Section 10.2.3. This family of functions makes it possible to select and construct new dynamic data, from the dynamic data produced by previous SIBs.

11.2 PORS -POINTS OF RETURN

11.2.1 Clear call

The clear call point of return constructs and sends a clear call por message to the SSF for delivery to the relevant BCSM. The service instance then changes the mode to stopped.

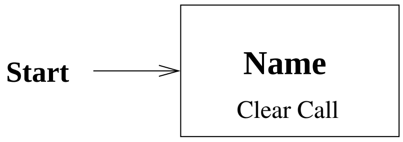


Figure 11.1: Clear Call POR

A graphical representation of the Clear Call POR is given in Figure 11.1.

$$\frac{\mathsf{sib}(\mathsf{clear\_call}, data, db)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\, \mathsf{ssf})} \langle\, \mathsf{stopped},\ data,\ \mathsf{nil},\ db\,\rangle} \tag{11.1}$$

$$\textbf{where}\quad m \;=\; \langle\langle \mathsf{scf}, \mathsf{response}\rangle,\ \mathsf{report}\ \langle \mathsf{initiator}(db), \langle\langle \mathsf{scf}, \mathsf{response}\rangle, \mathsf{clear\_call}, \perp\rangle\rangle$$

11.2.2 Continue with existing data

The continue with existing data point of return constructs and sends a The continue with existing data por message to the SSF for delivery to the relevant BCSM. The service instance then changes the mode to stopped.

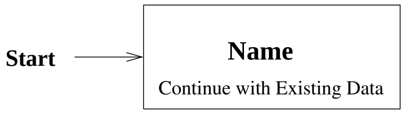


Figure 11.2: Continue with Existing Data POR

A graphical representation of the Continue with Existing Data POR is given in Figure 11.2.

$$\frac{\mathsf{sib}(\mathsf{continue\_with\_existing\_data}, data, db)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \xrightarrow{\mathsf{send}\ (m,\, \mathsf{ssf})} \langle\, \mathsf{stopped},\ data,\ \mathsf{nil},\ db\,\rangle} \tag{11.2}$$

$$\textbf{where}\quad m \;=\; \langle\langle \mathsf{scf}, \mathsf{response}\rangle,\ \mathsf{report},\ \langle \mathsf{initiator}(db), \langle\langle \mathsf{scf}, \mathsf{response}\rangle, \mathsf{continue\_with\_existing\_data}, \perp\rangle\rangle$$

11.2.3 Proceed with new data

The proceed with new data point of return collects the potential new data:

pc the position in the bcsm;

armed the arming of detection points;

present the generation of events by the services;

lookup what service is called and with what SSD;

accounting who is charged what part of the call;

cost the cost of the call.

The data is packed in a proceed with new data por message and sent to the SSF for delivery to the relevant BCSM. The mode is then set to stopped.

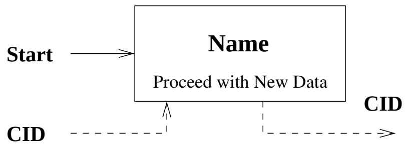


Figure 11.3: Proceed with New Data POR

A graphical representation of the Proceed with New Data POR is given in Figure 11.3.

$$\frac{\mathsf{sib}(\mathsf{proceed\_with\_new\_data}, data, db)}{\langle\, \mathsf{running},\, data,\, \mathsf{nil},\, db\,\rangle \xrightarrow{\mathsf{send}\ (m,\, \mathsf{ssf})} \langle\, \mathsf{stopped},\, data,\, \mathsf{nil},\, db\,\rangle} \tag{11.3}$$

$$\begin{aligned}
\textbf{where} \qquad pc &= \mathsf{ssd}(\mathsf{new_pc}, data, db)\\
armed &= \mathsf{cid}(\mathsf{new_armed}, data, db)\\
present &= \mathsf{cid}(\mathsf{new_present}, data, db)\\
lookup &= \mathsf{cid}(\mathsf{new_lookup}, data, db)\\
accouting &= \mathsf{cid}(\mathsf{new_accounting}, data, db)\\
cost &= \mathsf{cid}(\mathsf{new_cost}, data, db)\\
m &= \langle\langle \mathsf{scf}, \mathsf{response}\rangle,\\
&\quad\ \mathsf{report},\\
&\quad\ \langle \mathsf{initiator}(db),\\
&\quad\quad \langle\langle \mathsf{scf}, \mathsf{response}\rangle,\\
&\quad\quad\ \mathsf{proceed_with_new_data},\\
&\quad\quad\ \langle pc, armed, present, lookup, accounting, cost\rangle\rangle\rangle\rangle
\end{aligned}$$

11.2.4 Initiate call

The initiate call point of return indicates that a new call should be initiated either within an existing call, e.g., in call forwarding. The call can also be outside any existing call when a service has been triggered but there is not presently a call, e.g., call completion of call to busy subscriber that sets up a call when the busy subscriber becomes idle once more. The type SSD indicates whether the created connection leg set up is connected to a phone directly. The mode is then set to stopped.

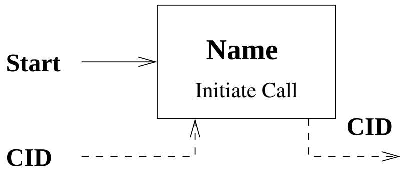


Figure 11.4: Initiate Call POR

A graphical representation of the Initiate Call POR is given in Figure 11.4.

$$\frac{\mathsf{sib}(\mathsf{initiate\_call}, data, db)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \xrightarrow{\mathsf{send}\ (m, \mathsf{ssf})} \langle\, \mathsf{stopped},\ data,\ \mathsf{nil},\ db\,\rangle} \tag{11.4}$$

$$\begin{array}{rrcl} \textbf{where} & phone & = & \mathsf{ssd}(\mathsf{phone}, data, db) \\ & type & = & \mathsf{ssd}(\mathsf{type}, data, db) \\ & armed & = & \mathsf{cid}(\mathsf{new\_armed}, data, db) \\ & no & = & \mathsf{cid}(\mathsf{cidfp{-}no}, data, db) \\ & event & = & \mathsf{ifundef}(no, \perp, \langle\langle \mathsf{phone}, \mathsf{notification}\rangle, \mathsf{connect}, no\rangle) \\ & m & = & \langle\langle \mathsf{scf}, \mathsf{request}\rangle \\ & & & \quad \mathsf{initiate\_call} \\ & & & \quad \langle \mathsf{initiator}(db), phone, type, armed, event\rangle\rangle \end{array}$$

11.3 HIERARCHY

The concept of hierarchical SIBs stems from the CS-2 draft. The concept enables a better structuring of the services, in that common parts may be reused and the SIBs of the service may be logically partitioned. We have chosen to model the hierarchical SIBs in a similar way to the services, with the difference that they lack a POI and PORs. In order to call and return from the execution of hierarchical SIBs we have added two new SIBs apart from the standard. The new SIBs hierarchy and unhierarchy acts similarly to the call and return to and from a process call in a programming language.

The hierarchy SIB sets the current service to be the hierarchical SIB invoked with the start : SIB as the current running SIB. The state is saved in the dump, the third part of the service instance' dynamic data. For resumption only the name of the service need bee recorded but in order that parameter SSD may be resolved the current SSD is also saved in the dump.

The unhierarchy SIB need only restore the state from the dump and decide what SIB in the service the control was returned to b y the hierarchical SIB.

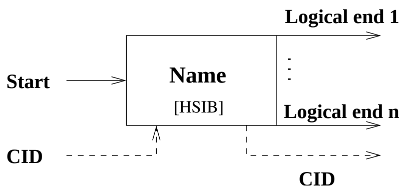


Figure 11.5: Hierarchy SIB

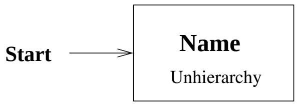


Figure 11.6: Unhierarchy SIB

A graphical representation of the Hierarchy SIB is given in Figure 11.5. A graphical representation of the Unhierarchy SIB is given in Figure 11.6.

11.3.1 RULES

(11.5)
$$\frac{\mathsf{sib}(\mathsf{hierarchy}, data, db)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \ \longrightarrow\ \langle\, \mathsf{running},\ data',\ \mathsf{nil},\ db\,\rangle}$$

$$\begin{array}{lrcl} \textbf{where} & hsib & = & \mathsf{ssd}(\mathsf{sib}, data) \\ & data' & = & \langle\langle hsib, \mathsf{start} : hsib\rangle \\ & & & \ \mathsf{frame}(data), \\ & & & \ \langle\langle \mathsf{current\_service}(data), \mathsf{current\_ssd}(data, db)\rangle, \mathsf{dump}(data)\rangle\rangle \end{array}$$

(11.6)
$$\frac{\mathsf{sib}(\mathsf{unhierarchy}, data, db)}{\langle\, \mathsf{running},\ data,\ \mathsf{nil},\ db\,\rangle \ \longrightarrow\ \langle\, \mathsf{running},\ data',\ \mathsf{nil},\ db\,\rangle}$$

$$\begin{array}{lrcl} \textbf{where} & sib & = & \mathsf{ssd}(\mathsf{exit}, data) \\ & service & = & \pi_1\,(\mathsf{list\!:hd}(\mathsf{dump}(data))) \\ & data' & = & \langle\langle service, sib\rangle, \mathsf{frame}(data), \mathsf{list\!:tl}(\mathsf{dump}(data))\rangle \end{array}$$

CHAPTER 12

HIERARCHICAL SIBs

12.1 DESCRIPTION

In this chapter will be modeled two hierarchical SIBs that are of the utility kind. The first SIB Null simply checks that a list is empty. This is a kind of low level utility SIBs which one may gather to handle abstract data types.

The second SIB is a utility input SIB that gathers information from the user, a string identifying the user in some way and a password for that user. This can be used by any resource that requires a password.

12.2 DOMAINS

$$HighLevelSIB \equiv \{\mathsf{Null}, \mathsf{Get\_N}\}$$

12.3 NULL

The Null hierarchical SIB tests whether a list is empty and the logical end reflects the result. This exemplifies the type of abstract data type functions needed when trying to get a manageable structure to the services. The abstract data type functions will as always be selectors, constructors and tests, the null hierarchical SIB implementing a testing function.

Input

Logical Start

Service Support Data

- **CIDFP-list** Specifies place of the list.
- **true** The SIB to which the control will be returned to when returning on the True logical end.
- **false** The SIB to which the control will be returned to when returning on the False logical end.
- **error** The SIB to which the control will be returned to when returning on the Error logical end.

Call Instance Data

- **list** The list to be checked.

Output

Logical End

- True
- False
- Error

The Null SIBs defined graphically in Figure 12.2 and the graphical notation for it in Figure 12.1.

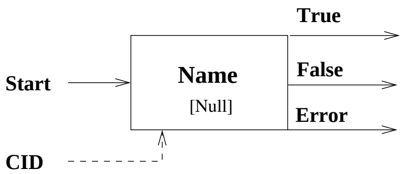
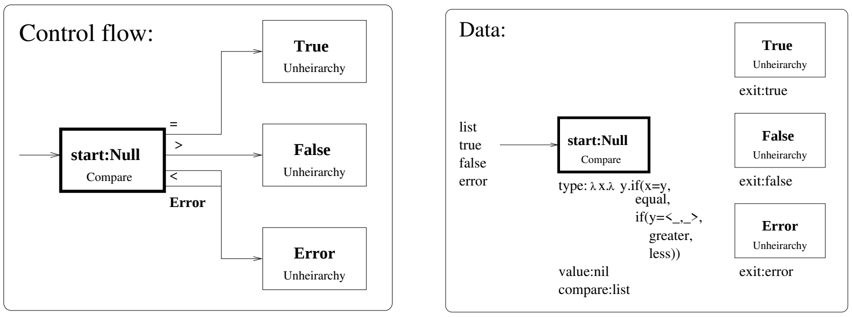


Figure 12.1: Null HSIB

$$
\begin{aligned}
\mathsf{Null} = \{ \\
&\langle \mathsf{start} : \mathsf{Null}, \\
&\ \langle \mathsf{compare}, \\
&\ \ \{\langle \mathsf{type}, \langle \mathsf{immediate}, \lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{if}(\mathsf{x} = \mathsf{y}, \mathsf{equal}, \mathsf{if}(\mathsf{y} = \langle \_, \_\rangle, \mathsf{greater}, \mathsf{less}))\rangle\rangle, \\
&\ \ \ \langle \mathsf{value}, \langle \mathsf{immediate}, \mathsf{nil}\rangle\rangle,\ \langle \mathsf{cidfp{-}compare}, \langle \mathsf{parameter}, \mathsf{list}\rangle\rangle, \\
&\ \ \ \langle \mathsf{equal}, \langle \mathsf{immediate}, \mathsf{True}\rangle\rangle,\ \langle \mathsf{greater}, \langle \mathsf{immediate}, \mathsf{False}\rangle\rangle, \\
&\ \ \ \langle \mathsf{less}, \langle \mathsf{immediate}, \mathsf{Error}\rangle\rangle,\ \langle \mathsf{error}, \langle \mathsf{immediate}, \mathsf{Error}\rangle\rangle\}\rangle\rangle, \\
&\langle \mathsf{True}, \\
&\ \langle \mathsf{unhierarchy}, \\
&\ \ \{\langle \mathsf{exit}, \langle \mathsf{parameter}, \mathsf{true}\rangle\rangle\}\rangle\rangle, \\
&\langle \mathsf{False}, \\
&\ \langle \mathsf{unhierarchy}, \\
&\ \ \{\langle \mathsf{exit}, \langle \mathsf{parameter}, \mathsf{false}\rangle\rangle\}\rangle\rangle, \\
&\langle \mathsf{Error}, \\
&\ \langle \mathsf{unhierarchy}, \\
&\ \ \{\langle \mathsf{exit}, \langle \mathsf{parameter}, \mathsf{error}\rangle\rangle\}\rangle\rangle, \\
&\}
\end{aligned}
$$



(a) Control (b) Data

Figure 12.2: Null hierarchical SIB

12.4 GET_N

The Get_N hierarchical SIB queries the user for information which the user has "N" tries to supply. The data supplied by the user is verified to see that it has the correct format. Should the data supplied by the user not conform to the required format he is queried once more, until he runs out of tries. This hierarchical SIB exemplifies how normal constructs, such as in this case getting additional information from the user, can be encapsulated using the hierarchical SIB concept.

Input

Logical Start

Service Support Data

- **type** The type of information to be collected from the user.
- **length** The length of the data to be collected from the user.
- **format** The format expected of the data collected from the user.
- **ask** The message informing the user what is expected of him.
- **wrong** The message informing the user that the data supplied by the user was incorrect.
- **end** The message informing the user that he is not given any more tries to supply the requested data.
- **errormsg** The message informing the user that a internal error has occurred within the service and that the service will be aborted.
- **no_of_tries** The maximal number of tries allowed the user to supply the requested data.

- **CIDFP-tries** The place in which it is recorded the number of tries the user has made to supply the requested data.
- **CIDFP-collected** The place in which the information supplied by the user is to be stored.
- **success** The SIB to which the control will be returned to when returning on the Success logical end.
- **fail** The SIB to which the control will be returned to when returning on the Fail logical end.
- **error** The SIB to which the control will be returned to when returning on the Error logical end.

Call Instance Data

- **tries** The number of tries the user has made to supply the requested informing.

Output

Logical End

- Success
- Fail
- Error

Call Instance Data

- **collected** The information supplied by the user.

The Get_N SIBs defined graphically in Figure 12.4 and the graphical notation for it in Figure 12.3.

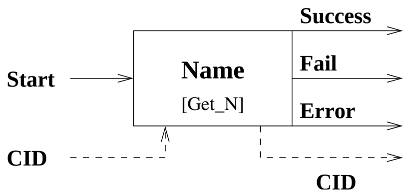
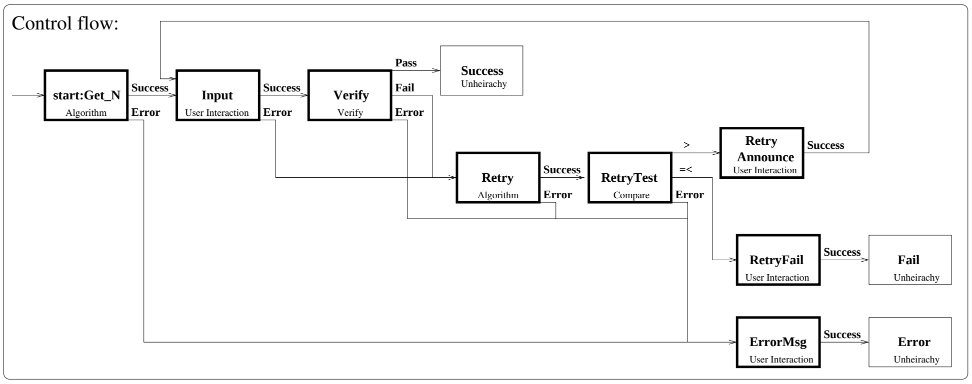
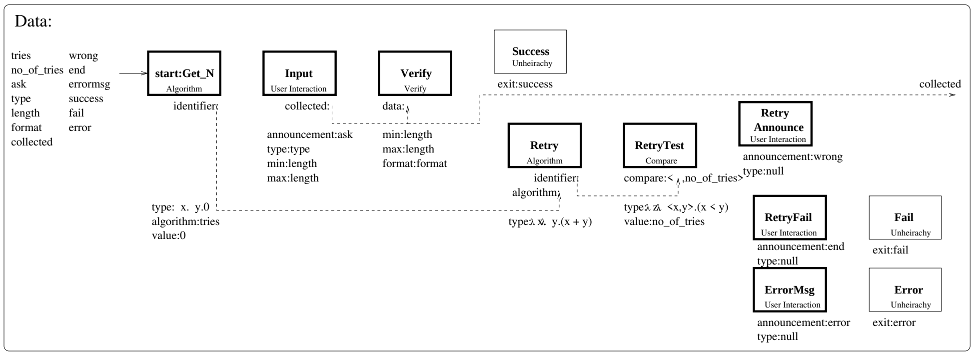


Figure 12.3: Get_N HSIB



(a) Control



(b) Data

Figure 12.4: Get_N hierarchical SIB

$$
\begin{array}{l}
\text{Get\_N} = \{ \\
\qquad \langle \text{start : Get\_N}, \\
\qquad\; \langle \text{algorithm}, \\
\qquad\;\; \{\langle \text{type}, \langle \text{immediate}, \lambda x.\lambda x.0\rangle\rangle,\;\; \langle \text{value}, \langle \text{immediate}, 0\rangle\rangle, \\
\qquad\;\;\; \langle \text{cidfp}-\text{algorithm}, \langle \text{parameter}, \text{tries}\rangle\rangle,\;\; \langle \text{cidfp}-\text{identifier}, \langle \text{parameter}, \text{tries}\rangle\rangle, \\
\qquad\;\;\; \langle \text{success}, \langle \text{immediate}, \text{Input}\rangle\rangle,\;\; \langle \text{error}, \langle \text{immediate}, \text{Error}\rangle\rangle\}\rangle\rangle, \\
\qquad \langle \text{Input}, \\
\qquad\; \langle \text{ui}, \\
\qquad\;\; \{\langle \text{annoucement}, \langle \text{parameter}, \text{ask}\rangle\rangle,\;\; \langle \text{type}, \langle \text{parameter}, \text{type}\rangle\rangle, \\
\qquad\;\;\; \langle \text{min}, \langle \text{parameter}, \text{length}\rangle\rangle,\;\; \langle \text{max}, \langle \text{parameter}, \text{length}\rangle\rangle, \\
\qquad\;\;\; \langle \text{collected}, \langle \text{parameter}, \text{collected}\rangle\rangle, \\
\qquad\;\;\; \langle \text{success}, \langle \text{immediate}, \text{Verify}\rangle\rangle,\;\; \langle \text{error}, \langle \text{immediate}, \text{Retry}\rangle\rangle\}\rangle\rangle, \\
\qquad \langle \text{Verify}, \\
\qquad\; \langle \text{verify}, \\
\qquad\;\; \{\langle \text{min}, \langle \text{parameter}, \text{length}\rangle\rangle,\;\; \langle \text{max}, \langle \text{parameter}, \text{length}\rangle\rangle, \\
\qquad\;\;\; \langle \text{format}, \langle \text{parameter}, \text{format}\rangle\rangle,\;\; \langle \text{cidfp}-\text{data}, \langle \text{parameter}, \text{collected}\rangle\rangle, \\
\qquad\;\;\; \langle \text{pass}, \langle \text{immediate}, \text{Success}\rangle\rangle,\;\; \langle \text{fail}, \langle \text{immediate}, \text{Retry}\rangle\rangle,\;\; \langle \text{error}, \langle \text{immediate}, \text{Error}\rangle\rangle\}\rangle\rangle, \\
\qquad \langle \text{Retry}, \\
\qquad\; \langle \text{algorithm}, \\
\qquad\;\; \{\langle \text{type}, \langle \text{immediate}, \lambda x.\lambda y.x + y\rangle\rangle,\;\; \langle \text{value}, \langle \text{immediate}, 1\rangle\rangle, \\
\qquad\;\;\; \langle \text{cidfp}-\text{algorithm}, \langle \text{parameter}, \text{tries}\rangle\rangle,\;\; \langle \text{cidfp}-\text{identifier}, \langle \text{parameter}, \text{tries}\rangle\rangle, \\
\qquad\;\;\; \langle \text{success}, \langle \text{immediate}, \text{RetryTest}\rangle\rangle,\;\; \langle \text{error}, \langle \text{immediate}, \text{Error}\rangle\rangle\}\rangle\rangle, \\
\qquad \langle \text{RetryTest}, \\
\qquad\; \langle \text{compare}, \\
\qquad\;\; \{\langle \text{type}, \langle \text{immediate}, \lambda z\lambda\langle x, y\rangle.x < y\rangle\rangle,\;\; \langle \text{value}, \langle \text{parameter}, \text{no\_of\_tries}\rangle\rangle, \\
\qquad\;\;\; \langle \text{cidfp}-\text{compare}, \langle \text{cons}, \langle\langle \text{parameter}, \text{tries}\rangle, \langle \text{parameter}, \text{no\_of\_tries}\rangle\rangle\rangle\rangle, \\
\qquad\;\;\; \langle \text{greater}, \langle \text{immediate}, \text{RetryAnnounce}\rangle\rangle,\;\; \langle \text{less}, \langle \text{immediate}, \text{RetryFail}\rangle\rangle, \\
\qquad\;\;\; \langle \text{equal}, \langle \text{immediate}, \text{RetryFail}\rangle\rangle,\;\; \langle \text{error}, \langle \text{immediate}, \text{Error}\rangle\rangle\}\rangle\rangle, \\
\qquad \langle \text{RetryAnnounce}, \\
\qquad\; \langle \text{ui}, \\
\qquad\;\; \{\langle \text{annoucement}, \langle \text{parameter}, \text{wrong}\rangle\rangle,\;\; \langle \text{type}, \langle \text{immediate}, \text{null}\rangle\rangle, \\
\qquad\;\;\; \langle \text{success}, \langle \text{immediate}, \text{Input}\rangle\rangle\}\rangle\rangle, \\
\qquad \langle \text{RetryFail}, \\
\qquad\; \langle \text{ui}, \\
\qquad\;\; \{\langle \text{annoucement}, \langle \text{parameter}, \text{end}\rangle\rangle,\;\; \langle \text{type}, \langle \text{immediate}, \text{null}\rangle\rangle, \\
\qquad\;\;\; \langle \text{success}, \langle \text{immediate}, \text{Fail}\rangle\rangle\}\rangle\rangle, \\
\qquad \langle \text{ErrorMsg}, \\
\qquad\; \langle \text{ui}, \\
\qquad\;\; \{\langle \text{annoucement}, \langle \text{parameter}, \text{errormsg}\rangle\rangle,\;\; \langle \text{type}, \langle \text{immediate}, \text{null}\rangle\rangle, \\
\qquad\;\;\; \langle \text{success}, \langle \text{immediate}, \text{Error}\rangle\rangle\}\rangle\rangle, \\
\qquad \langle \text{Success}, \\
\qquad\; \langle \text{unhierarchy}, \\
\qquad\;\; \{\langle \text{exit}, \langle \text{parameter}, \text{success}\rangle\rangle\}\rangle\rangle, \\
\qquad \langle \text{Fail}, \\
\qquad\; \langle \text{unhierarchy}, \\
\qquad\;\; \{\langle \text{exit}, \langle \text{parameter}, \text{fail}\rangle\rangle\}\rangle\rangle, \\
\qquad \langle \text{Error}, \\
\qquad\; \langle \text{unhierarchy}, \\
\qquad\;\; \{\langle \text{exit}, \langle \text{parameter}, \text{error}\rangle\rangle\}\rangle\rangle, \\
\qquad \}
\end{array}
$$

Part IV

Services

Introduction

The services and service features are presented as sequences of SIBs in graphical form as described in Section 1.5.1. The services and service are also presented in textual form. Further the constants defining the initiation points and the what the name is of the SIB chain defining the service.

All in all three service features are and three services are presented.

Two service features concern the billing of normal call handling which we have chosen to break out of the call handling and make into service features since they does not seems a natural integral part of call control. We feel that the model becomes simpler and less ad hoc when the billing is realised as separate features, one adding costs to the call and the other charging the accounts when the call is completed. This is of course, by necessity, a much simplified view of the charging system.

13.1 CONSTANTS

The "Billing_DPs" are the points in a call where billing is to occur.

$$\begin{aligned}\mathsf{Billing\_DPs} \stackrel{\text{def}}{=} \{&\mathsf{Route\_Select\_Failure}, \mathsf{O\_Call\_Party\_Busy},\\ &\mathsf{O\_No\_Answer}, \mathsf{O\_Disconnect}, \mathsf{O\_Abandon},\\ &\mathsf{T\_Call\_Party\_Busy}, \mathsf{T\_No\_Answer}, \mathsf{T\_Disconnect}, \mathsf{T\_Abandon}\}\end{aligned}$$

The "standard_profile" is definition of active and subscribed services the are assumed for all user by default, and the actual profile of a subscriber is gained by overriding these defaults.

The default service subscription consists of normal charging service that sets up the call so that normal billing takes place. The normal billing service that for each user noted as payer exacts it share of the cost for the call.

$$\begin{aligned}\mathsf{standard\_profile} \stackrel{\text{def}}{=} \langle&\mathsf{Normal\_Charging\_armed} \cup \mathsf{Normal\_Billing\_armed},\\ &\mathsf{Normal\_Charging\_present} \cup \mathsf{Normal\_Billing\_present},\\ &\mathsf{Normal\_Charging\_lookup} \cup \mathsf{Normal\_Billing\_lookup}\rangle\end{aligned}$$

Chapter 14

Service Features

14.1 DOMAINS

$$ServiceFeature \equiv \{\mathsf{Normal\_Billing}, \mathsf{Normal\_Charging}, \mathsf{CF}\}$$

14.2 NORMAL BILLING

a The normal billing service feature adds the cost of the call to the accounts of the user according to the information residing in the BCSM that invoked the service. The service is supplied with the accounts to charge and the portion of the cost they shall carry for the call as well as the cost.

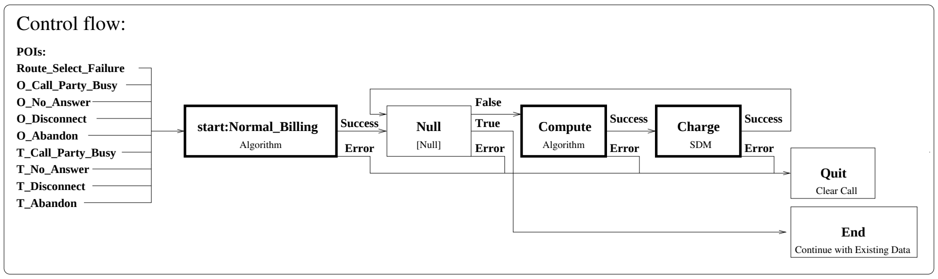
The service feature is invoked at all detection points that terminates the call. If the feature succeeds making the charging the accounts the service will return control to the same point in the call that is was invoked from. If an error occurs the call is cleared.

The service feature first loads the description of the accounts to use from the SSD into the dynamic data, using a no-operation algorithm SIB, in order to be able to change the list of accounts. The accounts are then processed one by one until the list of accounts are empty. Each account are charged according to each share in the cost using a SDM SIB.

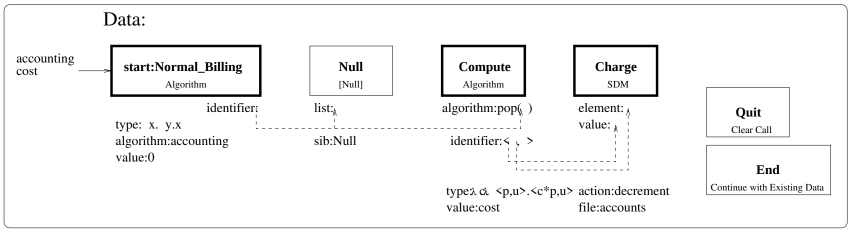
Figure 14.1 gives a presentation of the service feature as a sequence of SIBs.

14.2.1 Constants

$$\begin{array}{rcl}
\mathsf{Normal_Billing_armed} & \stackrel{\text{def}}{=} & \bigcup_{dp \in \mathsf{services:Billing_DPs}} \{\langle dp, template\rangle\} \\
\textbf{where} & & template = \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{normal_billing}\rangle \\
\mathsf{Normal_Billing_present} & \stackrel{\text{def}}{=} & \bigcup_{dp \in \mathsf{services:Billing_DPs}} \{\langle dp, \langle event, \mathsf{bcsm}\rangle\rangle\} \\
\textbf{where} & & event = \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{service}, \langle \mathsf{normal_billing}, \emptyset\rangle\rangle \\
\mathsf{Normal_Billing_lookup} & \stackrel{\text{def}}{=} & \{\langle \mathsf{normal_billing}, \langle \mathsf{Normal_Billing}, \emptyset\rangle\rangle\}
\end{array}$$



(a) Control



(b) Data

Figure 14.1: Normal billing service feature

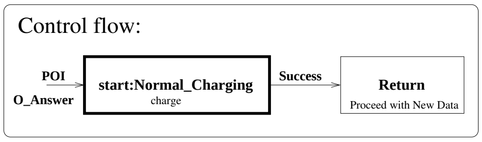
14.2.2 Definition

$$
\begin{aligned}
\mathsf{Normal\_Billing} = \{ & \\
& \langle \mathsf{start} : \mathsf{Normal\_Billing}, \\
& \ \langle \mathsf{algorithm}, \\
& \ \ \{\langle \mathsf{type}, \langle \mathsf{immediate}, \lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x}\rangle\rangle,\ \langle \mathsf{value}, \langle \mathsf{immediate}, 0\rangle\rangle, \\
& \ \ \ \langle \mathsf{cidfp-algorithm}, \langle \mathsf{parameter}, \mathsf{accounting}\rangle\rangle, \\
& \ \ \ \langle \mathsf{cidfp-identifier}, \langle \mathsf{location}, 1\rangle\rangle, \\
& \ \ \ \langle \mathsf{success}, \langle \mathsf{immediate}, \mathsf{Null}\rangle\rangle,\ \langle \mathsf{error}, \langle \mathsf{immediate}, \mathsf{Quit}\rangle\rangle\}\rangle\rangle, \\
& \langle \mathsf{Null}, \\
& \ \langle \mathsf{hierarchy}, \\
& \ \ \{\langle \mathsf{sib}, \langle \mathsf{immediate}, \mathsf{Null}\rangle\rangle,\ \langle \mathsf{list}, \langle \mathsf{location}, 1\rangle\rangle, \\
& \ \ \ \langle \mathsf{error}, \langle \mathsf{immediate}, \mathsf{Quit}\rangle\rangle,\ \langle \mathsf{true}, \langle \mathsf{immediate}, \mathsf{End}\rangle\rangle, \\
& \ \ \ \langle \mathsf{false}, \langle \mathsf{immediate}, \mathsf{Compute}\rangle\rangle\}\rangle\rangle, \\
& \langle \mathsf{Compute}, \\
& \ \langle \mathsf{algorithm}, \\
& \ \ \{\langle \mathsf{type}, \langle \mathsf{immediate}, \lambda \mathsf{c}.\lambda \mathsf{w}.\langle \mathsf{c} * \pi_2\,(\mathsf{w}), \pi_1\,(\mathsf{w})\rangle\rangle\rangle, \\
& \ \ \ \langle \mathsf{value}, \langle \mathsf{parameter}, \mathsf{cost}\rangle\rangle, \\
& \ \ \ \langle \mathsf{cidfp-algorithm}, \langle \mathsf{pop}, \langle \mathsf{location}, 1\rangle\rangle\rangle, \\
& \ \ \ \langle \mathsf{cidfp-identifier}, \langle \mathsf{location}, 2\rangle\rangle, \\
& \ \ \ \langle \mathsf{success}, \langle \mathsf{immediate}, \mathsf{Charge}\rangle\rangle,\ \langle \mathsf{error}, \langle \mathsf{immediate}, \mathsf{Quit}\rangle\rangle\}\rangle\rangle, \\
& \langle \mathsf{Charge}, \\
& \ \langle \mathsf{sdm}, \\
& \ \ \{\langle \mathsf{action}, \langle \mathsf{immediate}, \mathsf{decrement}\rangle\rangle, \\
& \ \ \ \langle \mathsf{file}, \langle \mathsf{immediate}, \mathsf{accounts}\rangle\rangle, \\
& \ \ \ \langle \mathsf{cidfp-value}, \langle \mathsf{car}, \langle \mathsf{location}, 2\rangle\rangle\rangle, \\
& \ \ \ \langle \mathsf{cidfp-element}, \langle \mathsf{cdr}, \langle \mathsf{location}, 2\rangle\rangle\rangle, \\
& \ \ \ \langle \mathsf{success}, \langle \mathsf{immediate}, \mathsf{Null}\rangle\rangle,\ \langle \mathsf{error}, \langle \mathsf{immediate}, \mathsf{Quit}\rangle\rangle\}\rangle\rangle, \\
& \langle \mathsf{End}, \langle \mathsf{continue\_with\_existing\_data}, \emptyset\rangle\rangle, \\
& \langle \mathsf{Quit}, \langle \mathsf{clear\_call}, \emptyset\rangle\rangle \\
\}&
\end{aligned}
$$

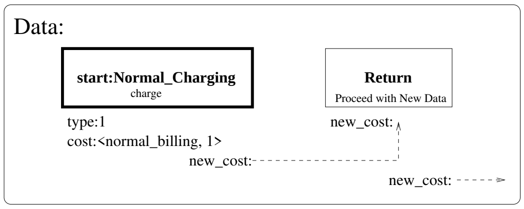
14.3 NORMAL CHARGING

The normal charge service feature adds the cost of normal call to the cost of the call. The feature is invoked at the O_Answer detection point when a normal call has been established.

Figure 14.2 gives a presentation of the service feature as a sequence of SIBs.



(a) Control



(b) Data

Figure 14.2: Normal charging service feature

14.3.1 Constants

$$
\begin{aligned}
\mathsf{Normal\_Charging\_armed} &\stackrel{\text{def}}{=} \{\langle \mathsf{O\_Answer}, \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{normal\_charging}\rangle\rangle\} \\
\mathsf{Normal\_Charging\_present} &\stackrel{\text{def}}{=} \{\langle \mathsf{O\_Answer}, \langle event, \mathsf{bcsm}\rangle\rangle\} \\
\textbf{where} \quad & event = \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle \\
& \qquad \mathsf{service} \\
& \qquad \langle \mathsf{normal\_charging}, \emptyset\rangle\rangle \\
\mathsf{Normal\_Charging\_lookup} &\stackrel{\text{def}}{=} \{\langle \mathsf{normal\_charging}, \langle \mathsf{Normal\_Charging}, \emptyset\rangle\rangle\}
\end{aligned}
$$

14.3.2 Definition

$$
\begin{aligned}
\mathsf{Normal\_Charging} = \{ \\
& \langle \mathsf{start} : \mathsf{Normal\_Charging}, \\
& \ \langle \mathsf{charge1}, \\
& \quad \{\langle \mathsf{type}, \langle \mathsf{location}, 1\rangle\rangle,\ \langle \mathsf{new\_cost}, \langle \mathsf{immediate}, 1\rangle\rangle, \\
& \quad\ \langle \mathsf{cost}, \langle \mathsf{immediate}, \langle \mathsf{normal\_billing}, 1\rangle\rangle\rangle, \\
& \quad\ \langle \mathsf{success}, \langle \mathsf{immediate}, \mathsf{Return}\rangle\rangle\}\rangle\rangle, \\
& \langle \mathsf{Return}, \\
& \ \langle \mathsf{proceed\_with\_new\_data}, \\
& \quad \{\langle \mathsf{new\_cost}, \langle \mathsf{location}, 1\rangle\rangle\}\rangle\rangle \\
& \}
\end{aligned}
$$

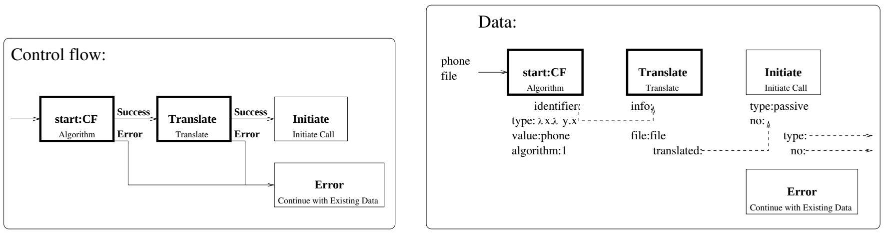
14.4 CALL FORWARDING

A description of The Call forwarding feature is found in [IN:Q.1211, appendix B.2.7, p.26], which is incorporated verbatim below:

> This service feature allows the user to have his incoming calls addressed to another number, no matter what the called party line status may be.

The call forwarding service feature looks up the destination forwarded to in a table. The name of the table is a parameter of the feature and must be set in the context of the service of which it constitutes a part. A new connection leg is established between the party forwarding and the forwarded to party, using the initiate call point of return.

Figure 14.3 gives a presentation of the service feature as a sequence of SIBs.



(a) Control (b) Data

Figure 14.3: Call Forward service feature

14.4.1 Definition

$$
\begin{aligned}
&\mathsf{CF} = \{\\
&\quad \langle \mathsf{start} : \mathsf{CF},\\
&\quad\ \langle \mathsf{algorithm},\\
&\quad\quad \{\langle \mathsf{type}, \langle \mathsf{immediate}, \lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x}\rangle\rangle,\ \langle \mathsf{value}, \langle \mathsf{parameter}, \mathsf{phone}\rangle\rangle,\\
&\quad\quad\ \langle \mathsf{cidfp{-}algorithm}, \langle \mathsf{location}, 1\rangle\rangle,\ \langle \mathsf{cidfp{-}identifier}, \langle \mathsf{location}, 1\rangle\rangle,\\
&\quad\quad\ \langle \mathsf{success}, \langle \mathsf{immediate}, \mathsf{Translate}\rangle\rangle,\ \langle \mathsf{error}, \langle \mathsf{immediate}, \mathsf{Error}\rangle\rangle\}\rangle\rangle,\\
&\quad \langle \mathsf{Translate},\\
&\quad\ \langle \mathsf{translate},\\
&\quad\quad \{\langle \mathsf{file}, \langle \mathsf{parameter}, \mathsf{file}\rangle\rangle,\ \langle \mathsf{cidfp{-}info}, \langle \mathsf{location}, 1\rangle\rangle,\\
&\quad\quad\ \langle \mathsf{translated}, \langle \mathsf{location}, 1\rangle\rangle,\\
&\quad\quad\ \langle \mathsf{success}, \langle \mathsf{immediate}, \mathsf{Initiate}\rangle\rangle,\ \langle \mathsf{error}, \langle \mathsf{immediate}, \mathsf{Error}\rangle\rangle\}\rangle\rangle,\\
&\quad \langle \mathsf{Initiate},\\
&\quad\ \langle \mathsf{initiate\_call},\\
&\quad\quad \{\langle \mathsf{type}, \langle \mathsf{immediate}, \mathsf{passive}\rangle\rangle,\ \langle \mathsf{cidfp{-}no}, \langle \mathsf{location}, 1\rangle\rangle\}\rangle\rangle,\\
&\quad \langle \mathsf{Error}, \langle \mathsf{continue\_with\_existing\_data}, \emptyset\rangle\rangle,\\
&\ \}
\end{aligned}
$$

Chapter 15

Services

15.1 DOMAINS

$$ServiceName \equiv \{\mathsf{AAB}, \mathsf{AAB2}, \mathsf{Call\_Forwarding}, \mathsf{Call\_Forwarding\_On\_Busy}\}$$

15.2 AUTOMATIC ALTERNATIVE BILLING SERVICE(AAB)

A description of The Automatic alternative billing service is found in [IN:Q.1211, appendix B.1.3, p.18], which is incorporated verbatim below:

> **Description No. 1**
>
> The AAB service enables the user to make a call from any telephone and for the call charge to be billed to the user's account which is specific to this service, and which does not refer to either the calling line or the called line.
>
> An account code and PIN are allocated to a user by the service management procedure.
>
> To invoke the service, the user dials an access code as a free call. Different access codes could be used to identify the language to be used. The user then receives announcement asking him to dial his account code and PIN. The account code and PIN are validated, and a check could be made for expired credit limits.
>
> NOTE - Account/credit card calling is similar, with the account No being supplied using a card wipe.
>
> **Description No. 2**
>
> This service allows a user to call another user and ask him to receive the call at his expenses. Two steps may be defined: the calling party is welcomed to record a brief message giving the callers' name and explaining the call reason, then the called party is alerted, receives the recorded message and is asked to accept to the charge for that call.

We choose to ignore **Description No 2**, and concentrate on **Description No 1**. A more elaborate and detailed description of this service is found in [IN:Klusener91], we have also relied on this description although we have chosen, as will be described in Subsection 15.2, to model it as two sequences of Service independent building blocks rather then one.

Figure 15.1 and Figure 15.2 gives a presentation of the service as a sequence of SIBs, whereas

The Figure 15.3 presents the chains in context of the BCP. The dashed lines indicate the return if an error should occur.

The instantiations of static parameters of the SIBs are as follows:

start:AAB hierarchy

- *The SIB to execute*: **sib** = Get_N
- *Question to user*: **ask** = "Account please"
- *On wrong input*: **wrong** = "Wrong account"
- *On to many tries*: **end** = "To many tries"
- *On error*: **errormsg** = "Error AAB"
- *Input type*: **type** = account
- *Length of input*: **length** = 6
- *Format of input*: **format** = "NNNNNN"
- *Number of allowed tries*: **no_of_tries** = 3
- *Where to store number of tries*: **tries** = location 3
- *Temporary for comparison*: **comparison** = location 4
- *Where to store the input*: **collected** = location 1

PIN: hierarchy

- *The SIB to execute*: **sib** = Get_N
- *Question to user*: **ask** = "PIN please"
- *On wrong input*: **wrong** = "Wrong PIN"
- *On to many tries*: **end** = "To many tries"
- *On error*: **errormsg** = "Error AAB"
- *Input type*: **type** = pin
- *Length of input*: **length** = 4
- *Format of input*: **format** = "NNNN"
- *Number of allowed tries*: **no_of_tries** = 3
- *Where to store number of tries*: **tries** = location 3
- *Temporary for comparison*: **comparison** = location 4
- *Where to store the input*: **collected** = location 2

Screen: screen

- *Screen list indicator*: **list** = AAB_PASSWD
- *The element to match*: **cidfp–screen** = $\langle$location 1, location 2$\rangle$

Translate: translate

- *Table for translation*: **Table** = AAB_ACCOUNT
- *Where to find information to translate*: **cidfp–Info** = location 1
- *Where to store translated result*: **cidfp–Translated** = location 1

Charge: charge

- *The type of charge*: **type** = 2
- *Where to find new armings*: **new_armed** = location 2
- *Where to find new present*: **new_present** = location 3
- *Where to find new accounting*: **new_accounting** = location 4
- *The service feature that supersedes Normal_Billing*: **cidfp–feature** = aab2
- *The ssd of the superseding feature*: **cidfp–ssd** = $\langle \mathsf{account}, \text{location } 1\rangle$

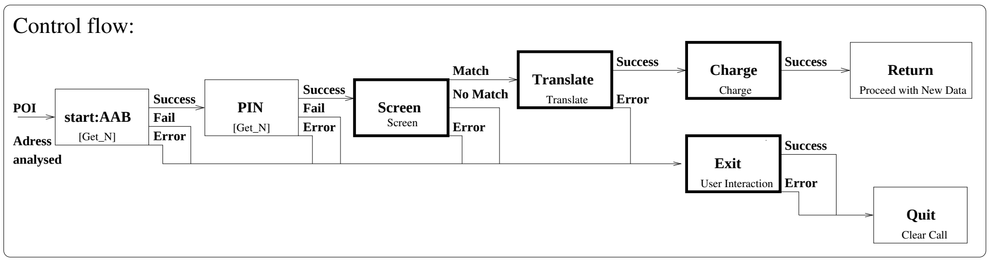
start:AAB2 sdm

- *The action to bee performed*: **action** = decrement
- *What database file to change*: **file** = accounts
- *The value used to change the elemnt*: **value** = parameter cost
- *Where to find the name of the element changed*: **cidfp–element** = parameter account

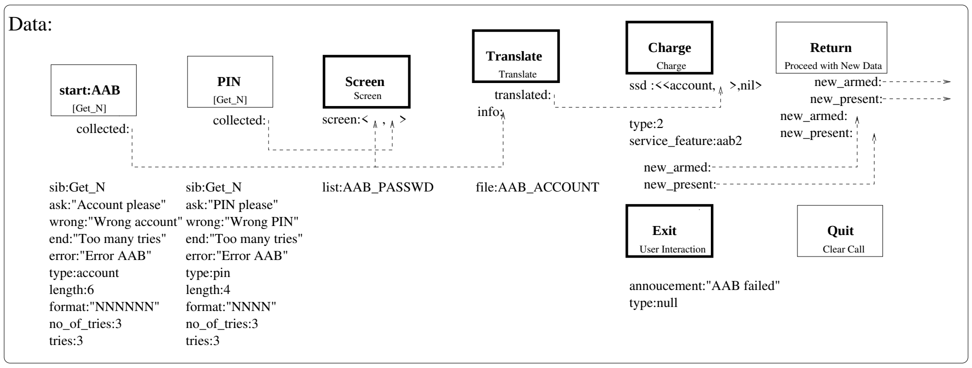
15.2.1 Constants

The activation point is Collected_Service_Info which is when it has been established an outgoing call may be placed and the BCSM is waiting for a connection request from the phone, and a AAB service request has been placed.

$$
\begin{aligned}
\mathsf{AAB\_armed} &\stackrel{\text{def}}{=} \{\langle \mathsf{Collected\_Service\_Info}, \langle\langle \mathsf{ssf}, \mathsf{request}\rangle, \mathsf{aab}\rangle\rangle\} \\
\mathsf{AAB\_present} &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{AAB\_lookup} &\stackrel{\text{def}}{=} \{\langle \mathsf{aab}, \langle \mathsf{AAB}, \emptyset\rangle\rangle\}
\end{aligned}
$$

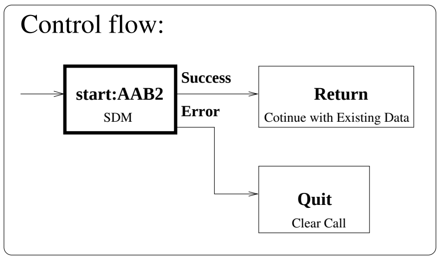


(a) Control

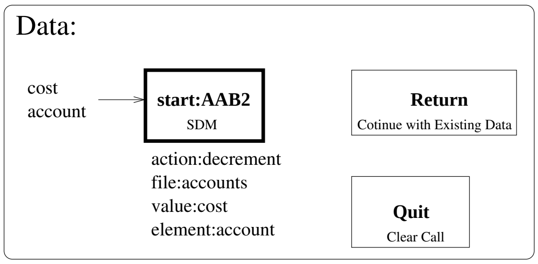


(b) Data

Figure 15.1: Automatic alternative billing service



(a) Control



(b) Data

Figure 15.2: Automatic alternative billing service (chain 2)

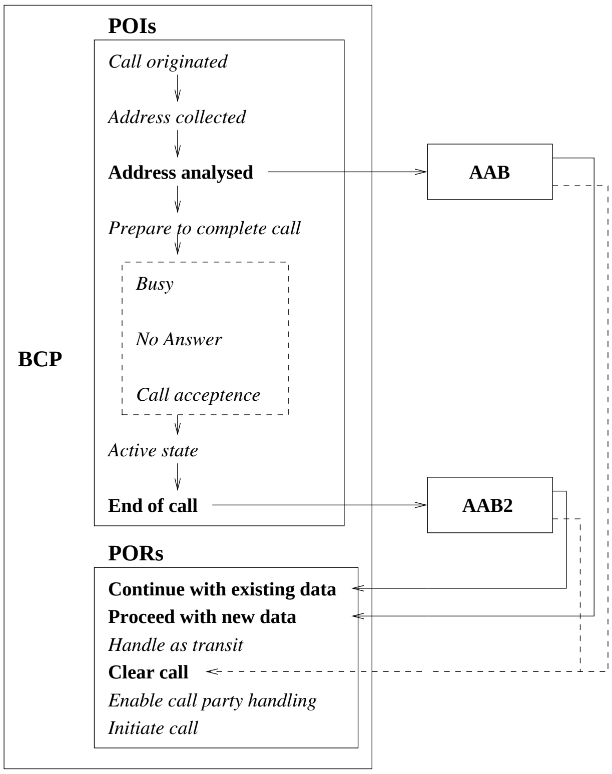


Figure 15.3: Automatic alternative billing service, visa vi the BCP

15.2.2 Definitions

AAB = {
⟨start : AAB,
⟨hierarchy,
{⟨sib, ⟨immediate, Get_N⟩⟩,
⟨ask, ⟨immediate, "Account please"⟩⟩, ⟨wrong, ⟨immediate, "Wrong account"⟩⟩,
⟨end, ⟨immediate, "To many tries"⟩⟩, ⟨errormsg, ⟨immediate, "Error AAB"⟩⟩,
⟨type, ⟨immediate, account⟩⟩, ⟨length, ⟨immediate, 6⟩⟩,
⟨format, ⟨immediate, "NNNNNN"⟩⟩,
⟨no_of_tries, ⟨immediate, 3⟩⟩, ⟨tries, ⟨location, 3⟩⟩, ⟨collected, ⟨location, 1⟩⟩,
⟨success, ⟨immediate, PIN⟩⟩, ⟨fail, ⟨immediate, Quit⟩⟩, ⟨error, ⟨immediate, Quit⟩⟩}⟩⟩,

⟨PIN,
⟨hierarchy,
{⟨sib, ⟨immediate, Get_N⟩⟩,
⟨ask, ⟨immediate, "PIN please"⟩⟩, ⟨wrong, ⟨immediate, "Wrong PIN"⟩⟩,
⟨end, ⟨immediate, "To many tries"⟩⟩, ⟨errormsg, ⟨immediate, "Error AAB"⟩⟩,
⟨type, ⟨immediate, pin⟩⟩, ⟨length, ⟨immediate, 4⟩⟩,
⟨format, ⟨immediate, "NNNN"⟩⟩,
⟨no_of_tries, ⟨immediate, 3⟩⟩, ⟨tries, ⟨location, 3⟩⟩, ⟨collected, ⟨location, 2⟩⟩,
⟨success, ⟨immediate, Screen⟩⟩, ⟨fail, ⟨immediate, Quit⟩⟩, ⟨error, ⟨immediate, Quit⟩⟩}⟩⟩,

⟨Screen,
⟨screen,
{⟨list, ⟨immediate, AAB_PASSWD⟩⟩, ⟨cidfp−screen, ⟨cons, ⟨⟨location, 1⟩, ⟨location, 2⟩⟩⟩⟩,
⟨match, ⟨immediate, Translate⟩⟩, ⟨nomatch, ⟨immediate, Exit⟩⟩,
⟨error, ⟨immediate, Quit⟩⟩}⟩⟩,

⟨Translate,
⟨translate,
{⟨file, ⟨immediate, AAB_ACCOUNT⟩⟩, ⟨cidfp−info, ⟨location, 1⟩⟩,
⟨translated, ⟨location, 1⟩⟩,
⟨success, ⟨immediate, Charge⟩⟩, ⟨error, ⟨immediate, Quit⟩⟩}⟩⟩,

⟨Charge,
⟨charge,
{⟨type, ⟨immediate, 2⟩⟩, ⟨new_armed, ⟨location, 2⟩⟩, ⟨new_present, ⟨location, 3⟩⟩,
⟨new_accounting, ⟨location, 4⟩⟩, ⟨service_feature, ⟨immediate, aab2⟩⟩,
⟨cidfp−ssd, ⟨cons, ⟨⟨cons, ⟨⟨immediate, account⟩, ⟨location, 1⟩⟩⟩, nil⟩⟩⟩,
⟨success, ⟨immediate, Return⟩⟩}⟩⟩,

⟨Return,
⟨proceed_with_new_data,
{⟨new_armed, ⟨location, 2⟩⟩, ⟨new_present, ⟨location, 3⟩⟩,
⟨new_accounting, ⟨location, 4⟩⟩}⟩⟩,

⟨Exit,
⟨ui,
{⟨annoucement, "AAB failed"⟩, ⟨type, ⟨immediate, null⟩⟩, ⟨success, Quit⟩}⟩⟩,

⟨Quit, ⟨clear_call, ∅⟩⟩
}

$$
\begin{aligned}
\mathsf{AAB2} = \{ \\
&\langle \mathsf{start} : \mathsf{AAB2}, \\
&\ \langle \mathsf{sdm}, \\
&\quad \{\langle \mathsf{action}, \langle \mathsf{immediate}, \mathsf{decrement}\rangle\rangle,\ \langle \mathsf{file}, \langle \mathsf{immediate}, \mathsf{accounts}\rangle\rangle, \\
&\quad\ \langle \mathsf{value}, \langle \mathsf{parameter}, \mathsf{cost}\rangle\rangle, \\
&\quad\ \langle \mathsf{cidfp{-}element}, \langle \mathsf{parameter}, \mathsf{account}\rangle\rangle, \\
&\quad\ \langle \mathsf{success}, \langle \mathsf{immediate}, \mathsf{Return}\rangle\rangle,\ \langle \mathsf{error}, \langle \mathsf{immediate}, \mathsf{Quit}\rangle\rangle\}\rangle\rangle, \\
&\langle \mathsf{Return}, \langle \mathsf{continue\_with\_existing\_data}, \emptyset\rangle\rangle \\
&\langle \mathsf{Quit}, \langle \mathsf{clear\_call}, \emptyset\rangle\rangle \\
\}
\end{aligned}
$$

15.3 CALL FORWARDING

A description of The Call forwarding service is found in [IN:Q.1211, appendix B.1.5, p.19], which is incorporated verbatim below:

> Call forwarding allows the user to forward calls to another telephone number when this service is activated. With this service, all calls destined to the subscriber's number are redirected to the new telephone number.
>
> This service is under the control of the subscriber and can de activated/deactivated by the subscriber.
>
> When this service is activated, the subscriber's line will receive an alerting ring, "reminder ring", to indicate the service is activated.

We have chosen to ignore the last paragraph and the subscriber is not reminded by a "ring", this is in line with the forwarding services the author is used to. As the call forwarding service uses the call forwarding feature defined earlier in Section 14.4 the only definition needed to conclude the service is the activation conditions and the setup necessary.

The activation point is Term.Attempt_Authorized which is when it has been established that a call can be, and is allowed, to be setup. The translating table used to determine the forwarded to number is determined by the SSD file which is set to CF_Table on invocation of the service.

The Figure 15.4 presents the Call forwarding services in context of the BCP. The CF chain of SIBs originating in the Prepare to complete call POI is the Call forwarding service. The dashed lines indicate the return if an error should occur.

15.3.1 Constants

$$
\begin{aligned}
\mathsf{Call\_Forwarding\_armed} &\stackrel{\text{def}}{=} \{\langle \mathsf{Term.Attempt\_Authorized}, \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{cf}\rangle\rangle\} \\
\mathsf{Call\_Forwarding\_present} &\stackrel{\text{def}}{=} \{\langle \mathsf{Term.Attempt\_Authorized}, \langle event, \mathsf{bcsm}\rangle\rangle\} \\
\textbf{where}\ & event = \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{service}, \langle \mathsf{cf}, \emptyset\rangle\rangle \\
\mathsf{Call\_Forwarding\_lookup} &\stackrel{\text{def}}{=} \{\langle \mathsf{cf}, \langle \mathsf{CF}, \{\langle \mathsf{file}, \mathsf{CF\_TABLE}\rangle\}\rangle\rangle\}
\end{aligned}
$$

15.4 CALL FORWARDING ON BUSY

A description of The Call forwarding on busy line service is found in [IN:Q.1211, appendix B.1.20, p.23], which is incorporated verbatim below:

> This service permits the user to have its incoming calls addressed to another number if they encounter a busy condition. The user's originating service is unaffected, even for charging.

As the call forwarding service uses the call forwarding feature defined earlier in Section 14.4 the only definition needed to conclude the service is the activation conditions and the setup necessary.

The activation point is T_Call_Party_Busy which is when it has been established that a busy condition has indeed been encountered. The translating table used to determine the forwarded to number is determined by the SSD file which is set to CFB_Table on invocation of the service.

The Figure 15.4 presents the Call forwarding services in context of the BCP. The CF chain of SIBs originating in the Busy POI is the Call forwarding on busy line service. The dashed lines indicate the return if an error should occur.

15.4.1 Constants

$$
\begin{aligned}
\mathsf{Call\_Forwarding\_On\_Busy\_armed} &\stackrel{\text{def}}{=} \{\langle \mathsf{T\_Call\_Party\_Busy}, \mathit{template}\}\\
\textbf{where}\ & \mathit{template} = \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{cfb}\rangle\rangle\\
\mathsf{Call\_Forwarding\_On\_Busy\_present} &\stackrel{\text{def}}{=} \{\langle \mathsf{T\_Call\_Party\_Busy}, \langle \mathit{event}, \mathsf{bcsm}\rangle\rangle\}\\
\textbf{where}\ & \mathit{event} = \langle\langle \mathsf{bcsm}, \mathsf{request}\rangle, \mathsf{service}, \langle \mathsf{cfb}, \emptyset\rangle\rangle\\
\mathsf{Call\_Forwarding\_On\_Busy\_lookup} &\stackrel{\text{def}}{=} \{\langle \mathsf{cfb}, \langle \mathsf{CF}, \{\langle \mathsf{file}, \mathsf{CFB\_TABLE}\rangle\}\rangle\rangle\}
\end{aligned}
$$

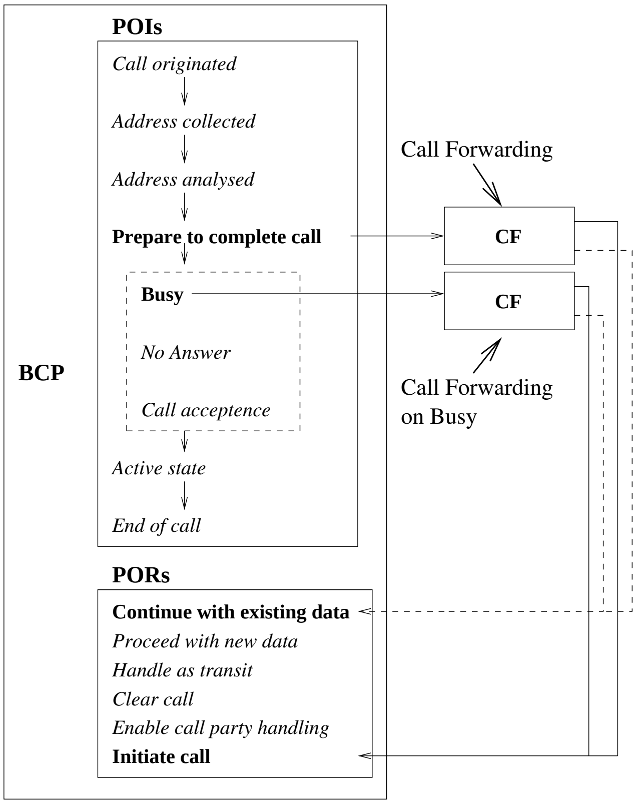


Figure 15.4: Call Forwarding services, visa vi the BCP

Part V

Appendices

Appendix A

Abbreviations

AAB	Alternative billing service	Section 15.2	p. 153
BCP	Basic call process	Section 6.1	p. 61
BCSM	Basic call state model	Section 6.1.4	p. 64
BNF	Backus Nauer form	Appendix B	p. 167
CCAF	Call control agent function	Section 2.1	p. 23
CCF	Call control function	Section 2.1	p. 23
CID	Call instance data	Section 10.1.1	p. 100
CIDFP	Call instance data field pointer	Section 10.1.1	p. 100
CS-1	Capability set one	Section 2.1	p. 21
CS-2	Capability set two	Section 2.1	p. 21
DFP	Distributed functional plane	Section 2.1	p. 23
DP	Detection point	Section 6.1.4	p. 64
GFP	Global functional plane	Chapter 2.1	p. 22
GSL	Global service logic	Section 11.1	p. 131
IN	Intelligent network	Chapter 2	p. 21
ITU-T	International Telecommunication Union - Telecommunications group	Section 2.1	p. 21
PIC	Point in call	Section 6.1.4	p. 64
PIN	Personal identification number	Section 10.1.1	p. 100
POI	Point of initiation	Section 6.1.2	p. 61
POR	Point of return	Section 6.1.3	p. 62
POS	Point of synchronisation	Section 10.8	p. 120
POTS	Plain old telephone system	Section 1.4.1	p. 6
PSF	Process specification formalism	Section 1.6	p. 16
SCF	Service control function	Section 7.1	p. 81
SDF	Service data function	Section 8.1	p. 85
SDM	Service data management	Section 10.11	p. 123
SIB	Service independent building blocks	Section 10.1	p. 99
SMF	Service management function	Section 2.1	p. 23
SRF	Specialized resources function	Section 9.1	p. 93
SSD	Service support data	Section 10.1.1	p. 100
SSF	Service switching function	Section 5.1	p. 47
UI	User interaction	Section 10.13	p. 126

Appendix B

Syntax

The following BNF[1] grammar describes the syntax of the definitions of the various processes.

In the BNF grammars the following type-face conventions are used: terminal tokens are in "sans serif", non terminals in *italics*. The ϵ stands for the empty string.

B.1 MACHINE

$$
\begin{array}{rcl}
\mathit{Machine} & ::= & \begin{array}{l}\mathit{MachineName}\\ \mathit{DomainDefs}\\ \mathit{ConstantDefs}\\ \mathit{FunctionDefs}\\ \mathit{RuleDefs}\end{array}\\[1em]
\mathit{MachineName} & ::= & \mathit{Identifier}\\[1em]
\mathit{DomainDefs} & ::= & \epsilon \mid \begin{array}{l}\mathit{DomainDef}\\ \mathit{DomainDefs}\end{array}\\[1em]
\mathit{ConstantDefs} & ::= & \epsilon \mid \begin{array}{l}\mathit{ConstantDef}\\ \mathit{ConstantDefs}\end{array}\\[1em]
\mathit{FunctionDefs} & ::= & \epsilon \mid \begin{array}{l}\mathit{FunctionDef}\\ \mathit{FunctionDefs}\end{array}\\[1em]
\mathit{RuleDefs} & ::= & \epsilon \mid \begin{array}{l}\mathit{RuleDef}\\ \mathit{RuleDefs}\end{array}
\end{array}
$$

[1] Backus Nauer Form, see [Aho86] for description.

B.2 DOMAINS

$$
\begin{array}{rcl}
\mathit{DomainDef} & ::= & \mathit{DomainDefName}\ \text{"}\equiv\text{"}\ \mathit{DomainExpression} \\
\\
\mathit{DomainDefName} & ::= & \mathit{DomainIdentifier} \\
 & & \mid \mathit{Identifier}\ \text{"("}\ \mathit{DomainVariable}\ \text{")"} \\
\mathit{DomainName} & ::= & \mathit{DomainIdentifier} \\
 & & \mid \mathit{Identifier}\ \text{"("}\ \mathit{Identifier}\ \text{")"} \\
 & & \mid \mathit{Identifier}\ \text{"("}\ \mathit{DomainExpression}\ \text{")"} \\
\mathit{DomainExpression} & ::= & \mathit{MachineDesignator}\ \mathit{DomainName} \\
 & & \mid \text{"}\emptyset\text{"} \\
 & & \mid \text{"}\{\text{"}\ \mathit{Identifier}\ \mathit{DomainCont}\ \text{"}\}\text{"} \\
 & & \mid \mathit{DomainExpression}\ \text{"}\cup\text{"}\ \mathit{DomainExpression} \\
 & & \mid \mathit{DomainExpression}\ \text{"}\times\text{"}\ \mathit{DomainExpression} \\
 & & \mid \mathit{DomainExpression}\ \text{"}\rightarrow\text{"}\ \mathit{DomainExpression} \\
 & & \mid \text{"}\mathbb{P}\text{"}\ \text{"("}\ \mathit{DomainExpression}\ \text{")"} \\
\mathit{DomainCont} & ::= & \epsilon \mid \text{","}\ \mathit{Identifier}\ \mathit{DomainCont}
\end{array}
$$

B.3 CONSTANTS

$$
\begin{array}{rcl}
\mathit{ConstantDef} & ::= & \begin{array}{l} \mathit{ConstantName}\ \text{"}\stackrel{\text{def}}{=}\text{"}\ \mathit{RHS} \\ \mathit{Where} \end{array} \\
\mathit{ConstantName} & ::= & \mathit{Identifier}
\end{array}
$$

B.4 FUNCTIONS

$$
\begin{array}{rcl}
\mathit{FunctionDef} & ::= & \begin{array}{l} \mathit{FunctionName}\ \text{"("}\ \mathit{Args}\ \text{")"}\ \text{"}\stackrel{\text{def}}{=}\text{"}\ \mathit{RHS} \\ \mathit{Where} \end{array} \\
\mathit{FunctionName} & ::= & \mathit{Identifier} \\
\mathit{Args} & ::= & \mathit{LHSList}
\end{array}
$$

B.5 RULES

$$
\begin{array}{rcl}
\mathit{RuleDef} & ::= & \begin{array}{l} \dfrac{\mathit{Guard}}{\mathit{PreState} \xrightarrow{\mathit{Actions}} \mathit{PostState}} \\ \mathit{Where} \end{array}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Guard} & ::= & \mathit{BooleanExpression} \\
\mathit{PreState} & ::= & \text{"}\langle\text{"}\ \mathit{LHS}\ \text{","}\ \mathit{LHS}\ \text{","}\ \mathit{LHS}\ \text{","}\ \mathit{LHS}\ \text{"}\rangle\text{"} \\
\mathit{PostState} & ::= & \text{"}\langle\text{"}\ \mathit{RHS}\ \text{","}\ \mathit{RHS}\ \text{","}\ \mathit{RHS}\ \text{","}\ \mathit{RHS}\ \text{"}\rangle\text{"} \\
\mathit{Actions} & ::= & \epsilon \mid \mathit{Action}\ \mathit{ActionList} \\
\mathit{ActionList} & ::= & \epsilon \mid \begin{array}{ll} & \text{";"} \\ \mathit{Action} & \mathit{ActionList} \end{array} \\
\mathit{Action} & ::= & \text{"}\mathsf{create}\text{"}\ \text{"("}\ \mathit{RHS}\ \text{","}\ \mathit{RHS}\ \text{")"} \\
 & & \mid \mathit{Variable}\ \text{":="}\ \text{"}\mathsf{create}\text{"}\ \text{"("}\ \mathit{RHS}\ \text{","}\ \mathit{RHS}\ \text{")"} \\
 & & \mid \text{"}\mathsf{broadcast}\text{"}\ \text{"("}\ \mathit{RHS}\ \text{")"} \\
 & & \mid \text{"}\mathsf{send}\text{"}\ \text{"("}\ \mathit{RHS}\ \text{","}\ \mathit{RHS}\ \text{")"}
\end{array}
$$

B.6 UTILITY

B.6.1 Let definitions

$$
\begin{array}{rcl}
\mathit{Where} & ::= & \epsilon \mid \begin{array}{ll} \text{"}\mathbf{where}\text{"} & \mathit{LetDef} \\ & \mathit{LetDefs} \end{array} \\
\mathit{LetDefs} & ::= & \epsilon \mid \begin{array}{l} \mathit{LetDef} \\ \mathit{LetDefs} \end{array} \\
\mathit{LetDef} & ::= & \mathit{LHS}\ \text{"="}\ \mathit{RHS}
\end{array}
$$

B.6.2 Left hand side expression – LHS

$$
\begin{array}{rcl}
\mathit{LHSList} & ::= & \epsilon \mid \mathit{LHS}\ \mathit{LHSListCont} \\
\mathit{LHSListCont} & ::= & \epsilon \mid \text{","}\ \mathit{LHS}\ \mathit{LHSListCont}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{LHS} & ::= & \text{"\_"} \\
 & & \mid \mathit{Variable} \\
 & & \mid \mathit{Constant} \\
 & & \mid \text{"}\langle\text{"}\ \mathit{LHS}\ \text{","}\ \mathit{LHSList}\ \text{"}\rangle\text{"} \\
 & & \mid \text{"}\{\text{"}\ \mathit{LHSList}\ \text{"}\}\text{"}
\end{array}
$$

B.6.3 Right hand side expression – RHS

$$
\begin{aligned}
RHSList &::= \epsilon \mid RHS\ RHSListCont \\
RHSListCont &::= \epsilon \mid \text{","}\ RHS\ RHSListCont \\
\\
RHS &::= Variable \\
&\mid Constant \\
&\mid BooleanExpression \\
&\mid NumberExpression \\
&\mid SetExpression \\
&\mid TupleExpression \\
&\mid FunctionExpression \\
&\mid IFExpression
\end{aligned}
$$

B.6.4 Constants

$$
\begin{aligned}
ConstantList &::= \epsilon \mid Constant\ ConstantListCont \\
ConstantListCont &::= \epsilon \mid \text{","}\ Constant\ ConstantListCont \\
\\
Constant &::= \text{"}\bot\text{"} \\
&\mid Identifier \\
&\mid BooleanConstant \\
&\mid NumberConstant \\
&\mid TupleConstant \\
&\mid SetConstant \\
\\
BooleanConstant &::= \text{"true"} \mid \text{"false"} \\
NumberConstant &::= \mathbb{N} \mid \mathrm{Z} \mid \mathbb{R} \\
TupleConstant &::= \text{"}\langle\text{"}\ ConstantList \text{"}\rangle\text{"} \\
SetConstant &::= \text{"}\emptyset\text{"} \mid \text{"}\{\text{"}\ ConstantList \text{"}\}\text{"}
\end{aligned}
$$

B.6.5 Expressions

Boolean

$$
\begin{aligned}
BooleanExpression &::= BooleanUnOp\ RHS \\
&\mid RHS\ BooleanBinOp\ RHS \\
BooleanUnOp &::= \text{"}\neg\text{"} \\
BooleanBinOp &::= \text{"}\wedge\text{"} \mid \text{"}\vee\text{"} \mid \text{"}\rightarrow\text{"} \mid \text{"}\leftrightarrow\text{"} \mid \text{"}=\text{"} \mid \text{"}\neq\text{"}
\end{aligned}
$$

Number

$$
\begin{aligned}
NumberExpression &::= NumberUnOp\ RHS \\
&\mid RHS\ NumberBinOp\ RHS \\
NumberUnOp &::= \text{"}+\text{"} \mid \text{"}-\text{"} \\
NumberBinOp &::= \text{"}+\text{"} \mid \text{"}-\text{"} \mid \text{"}*\text{"} \mid \text{"}/\text{"}
\end{aligned}
$$

Tuple

$$
\begin{array}{rcl}
\textit{TupleExpression} & ::= & \epsilon \mid \text{"}\langle\text{"}\ \textit{RHS}\ \ \textit{TupleExpressionCont}\ \text{"}\rangle\text{"} \\
\textit{TupleExpressionCont} & ::= & \epsilon \mid \text{","}\ \textit{RHS}\ \ \textit{TupleExpressionCont}
\end{array}
$$

Set

$$
\begin{array}{rcl}
\textit{SetExpression} & ::= & \textit{RHS SetBinOp RHS} \\
 & & \mid \textit{SetFilter} \\
 & & \mid \textit{SetMap} \\
\textit{SetBinOp} & ::= & \text{"}\in\text{"} \mid \text{"}\cup\text{"} \mid \text{"}\cap\text{"} \mid \text{"}\backslash\text{"} \mid \text{"}\oplus\text{"} \\
\textit{SetFilter} & ::= & \text{"}\{\text{"}\ \textit{LHS}\ \text{"}|\text{"}\ \textit{BooleanExpression}\ \text{"}\}\text{"} \\
\textit{SetMap} & ::= & \text{"}\bigcup\text{"}_{\textit{LHS}\ \text{"}\in\text{"}\ \textit{RHS}}\ \textit{RHS}
\end{array}
$$

Functions

$$
\begin{array}{rcl}
\textit{FunctionExpression} & ::= & \textit{Function}\ \text{"("}\ \textit{RHSList}\ \text{")"} \\
 & & \\
\textit{Function} & ::= & \textit{MachineDesignator}\ \ \textit{FunctionName} \\
 & & \mid \textit{FunctionName}^{\text{"}-1\text{"}} \\
 & & \mid \textit{Variable} \\
 & & \mid \textit{FunctionExpression} \\
 & & \mid \textit{LambdaExpr} \\
 & & \\
\textit{LambdaExpr} & ::= & \text{"}\lambda\text{"}\ \textit{Variable}\ \text{"."}\ \textit{RHS} \\
 & & \mid \text{"}\lambda\text{"}\ \textit{Variable}\ \text{"."}\ \textit{LambdaExpr}
\end{array}
$$

IF

$$
\begin{array}{rcl}
\textit{IFExpression} & ::= & \left\{ \begin{array}{l} \textit{IFClause} \\ \textit{CaseClause} \end{array} \right. \\
\textit{CaseClause} & ::= & \epsilon \mid \begin{array}{l} \textit{IFClause} \\ \textit{CaseClause} \end{array} \mid \textit{OtherwiseClause} \\
\textit{IFClause} & ::= & \textit{RHS}\ \text{"if"}\ \textit{BooleanExpression} \\
\textit{OtherwiseClause} & ::= & \textit{RHS}\ \text{"otherwise"}
\end{array}
$$

B.6.6 Variables & Identifiers

$$
\begin{array}{rcl}
\mathit{MachineDesignator} & ::= & \epsilon \mid \mathit{MachineName}\ \text{"}:\text{"} \\
\mathit{Variable} & ::= & \text{"}\sigma\text{"} \mid [a - Z]* \\
\mathit{DomainVariable} & ::= & [\mathcal{A} - \mathcal{Z}] \\
\mathit{DomainIdentifier} & ::= & [A - Z][a - Z]* \\
\mathit{Identifier} & ::= & [\mathsf{a} - \mathsf{Z}][\mathsf{a} - \mathsf{Z}1 - 9]*
\end{array}
$$

Part VI

Bibliographies

Miscellaneous

[Aho86] Aho, A.V., Sethi, R., Ullman, J.D., 1986: *Compilers Principles, Techniques and Tools,* Addison-Wesley, Reading, Mass. 1986. {167}

[Aït-Kaci90] Aït-Kaci, H., 1990: *The WAM: A (Real) Tutorial,* Digital PRL 1990. {8}

[Armstrong96] Armstrong, J., Virding, R., Wikström, C., Williams, M., 1996: *Concurrent Programming in Erlang,* Prentice Hall, 1996. {43}

[de Bakker80] de Bakker, J., 1980: *Mathematical Theory of Program Correctness,* Prentice-Hall International Series in Computer Science, Prentice-Hall Inc., London, 1980. {7}

[Blau99] Blau, S., Rooth, J., Axell, J., Hellstrand, F., Buhrgard, M., Westin, T., Wicklund, G., 1999: *AXD 310: A new generation ATM switching system,* Computer Networks, No.31, pp.559–582, Elsevier Science B.V., 1999. {43}

[Diller94] Diller, A., 1994: *Z: an introduction to formal methods,* 2nd ed., John Wiley & Sons Ltd., Baffin Lane, Chichester, West Sussex PO19 1UD, England, 1994. {7}

[Duke91] Duke, R., King, P., Rose, G., Smith, G., 1991: *The Object-Z Specification Language: Version 1,* Technical report No.91-1, Software Verification Center, Department of Computer Science, University of Queensland, Australia, 1991. {7}

[Griffiths94] Griffiths, A., Rose, G., 1994: *A semantic foundation for object identity in formal specification,* Technical report No.94-21, Software Verification Center, Department of Computer Science, University of Queensland, Australia, 1994. {7}

[Griffiths95a] Griffiths, A., Rose, G., 1995: *A Semantic Foundation for Object Identity in Formal Specification,* Technical report No.95-38, Software Verification Center, Department of Computer Science, University of Queensland, Australia, 1995. {7}

[Griffiths95b] Griffiths, A., 1995: *An Extended Semantic Foundation For Object-Z,* Technical report No.95-39, Software Verification Center, Department of Computer Science, University of Queensland, Australia, 1995. {7}

[vanHee94] van Hee, K.M., 1994: *Information systems engineering: A formal approach,* Press Syndicate of the University of Cambridge, The Pitt Building, Trumping Street, Cambridge CB2 1RP, 1994. {7}

[Henderson80] Henderson, P., 1980: *Functional Programming: application and implementation,* Prentice-Hall, Englewood Cliffs, N.J., 1980. {8}

[Holzmann98] Holzmann, J.G., Smith, M.H., 1998: *Interval Reduction through Requirement Analysis,* Bell Labs Technical Journal, Vol.3, No.2, pp.22–31, 1998. {4}

[Jensen91] Jensen, K., 1991: Design/slash CPN Manual, Meta Software Corporation, Cambridge, USA, 1991. {7, 16}

[Jensen92] Jensen, K., 1992: *Coloured Petri Nets: Basic concepts, analysis methods and practical use,* Vol.1, 2nd ed., Springer-Verlag, 1996. {7}

[Jensen95] Jensen, K., 1995: *Coloured Petri Nets: Basic concepts, analysis methods and practical use,* Vol.2, 2nd ed., Springer-Verlag, 1995. {7}

[Kozen83] Kozen, D., 1983: *Results on the Propositional μ-Calculus,* Theoretical Computer Science, Vol.27, pp.333–354, 1983. {17}

[Lamport95] Lamport, L., 1995: *TLA^+, Technical report, DEC/SRC, Jul 1995,* . {7}

[Larsen95] Larsen, K.G., Pettersson, P., Yi, W., 1995: *Diagnostic Model-Checking for Real-Time Systems,* In Proceedings of Workshop on Verification and Control of Hybrid Systems III, eds. R. Alur, T.A. Henzinger and E.D. Sontag, LNCS Vol.1066, pp.575–586, 1995. {4}

[Landin64] Landin, P.J., 1964: *The mechanical evaluation of expressions,* Computer Journal, Vol.6, pp.308–320, 1964. {8}

[Manna92] Manna, Z., Pnueli, A, 1992: *The Temporal Logic of Reactive and Concurrent Systems,* Springer-Verlag: Heidelberg, Germany, 1992. {14}

[Mauw91] Mauw, S., 1991: *PSF, A Process Specification Formalism,* Ph.D. thesis, University of Amsterdam, The Netherlands, 1991. {16}

[Nicholls95] Nicholls, J., 1995: *Z base standard: Version 1.2,* Programming Research Group, Oxford University Computing Laboratory, Sep 1995. {7}

[NorthernTelecom92] Northern Telecom, 1992: *DMS–100,* Meridian Digital Centrex Library, 50039.08/12–92 issue 1 edition, 1992. {1}

[Peyton87] Peyton Jones, S.L., 1987: *The Implementation of Functional Programming Languages,* Prentice-Hall, Englewood Cliffs, 1987. {8}

[Smith94] Smith, G., 1994: *A Logic for Object-Z,* Technical report, No.94-48, Software Verification Center, Department of Computer Science, University of Queensland, Australia, Dec 1994. {7}

[Spivey87] Spivey, J.M., 1987: *The Z Notation : a reference manual,* Prentice Hall International Ltd, 66 Wood Lane End, Hemel Hampstead, Hertfordshire, HP2 4RG, 1989. {7}

[Steffen95] Steffen, B., Claßen, A., Margaria, T., 1995: *The* METAFRAME*: An Environment for Flexible Tool Management,* Proceedings of TAPSOFT'95: Theory and Practice of Software Development, eds. Mosses, Nielsen and Schwartzbach, LNCS Vol.915, pp.791–792, 1995. {17}

[Stoy77] Stoy, J.E., 1977: *Denotational Semantics: The Scott-Strachy approach to Programming Language Theory,* The MIT Press, Cambridge, Mass., 1987. {7, 28}

[Winskel89] Winskel, G., 1989: *The Formal Semantics of Programming Languages,* The MIT Press, cambridge, Mass. 02142, 1993. {8}

[Warren83] Warren, D.H.D., 1983: *An Abstract Prolog Instruction Set,* SRI Technical Note 309, SRI, Menlo Park, 1983. {8}

[Wordsworth92] Wordsworth, J.B., 1992: *Software development with Z: a practical approach to formal methods in software engineering ,* Addison-Wesley Publishing Company Inc., 1992. {7}

[Z.100] ITU-T, 1993: *Specification and description language,* Recommendation Z100, Mar 1993. {16}

Intelligent Network

[IN:Capellmann98] Capellmann, C., Christensen, S., Herzog, U., 1998: *Visualising the Behaviour of Intelligent Networks,* Proceedings of VISUAL'98, Services and Visualization: Towards User-Friendly Design, eds. T. Margaria, B. Steffen, R. Rückert and J. Posegga, LNCS Vol.1385, pp.174–189, 1998. {16}

[IN:Csurgay94] Csurgay, P., 1994: *Service Creation and Deployment in an Intelligent Network Laboratory,* Master's Thesis, Norwegian Institute of Thechnology, Inst. for Datateknikk og Telematik, University of Trondhiem, 1994. {6, 16}

[IN:Duran92] Duran, J.M., Visser, J., 1992: *International Standards for Intelligent Networks,* IEEE Communications Magazine, Feb 1992. {21}

[IN:Faynberg97] Faynberg, I., Gabuzda, R.L., Kaplan, M.P., Shah, N.J, 1997: *The Intelligent Network standards: Their application to services,* McGraw Hill Series on Telecommunications, McGraw Hill, 11 West 19th Street, New York, 1997. {21}

[IN:Hinde00] Hinde, S., 2000: *Use of Erlang/OTP as a Service Creation Tool for IN Services,* Proceedings of Sixth International Erlang/OTP Users Conference, `http://www.erlang.se/euc/00/one2one.pdf`, 2000. {43}

[IN:Kang96] Kang, M., Kown, T., Yang, C., Song, J., 1996: *The Design and Verification of Service Feature Interaction Manager on AIN Using Z Schema,* Proceedings of COMPSAC'96, pp.477–482, 1996. {6}

[IN:Klusener91] Klusener, A.S., Van Vlijmen, B., Van Waveren, A., 1991: *Service Independent Building Blocks-I; Concepts, Examples and Formal Specifications,* Report CS-R9326, Stichting Mathematisch Centrum, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, Apr 1993. {6, 16, 153}

[IN:Naeser99] Naeser, G., Nyström, J., Jonsson, B., 1999: *Building Tools for Creation and Analysis of Telephone Services,* Proceedings of RadioVetenskap och Kommunikation '99, 1999. {8}

[IN:Nyström96] Nyström, J., Jonsson, B., 1996: *A Formalization of Service Independent Building Blocks,* Proceedings of Advanced Intelligent Networks '96 workshop, ed. T. Margaria T, pp.1–14, 1996. {7}

[IN:Q.1204] ITU-T, 1993: *Intelligent network Distributed functional plane architecture,* General recommendations on telephone switching and signaling, Intelligent network, Mar 1993. {24, 47, 81, 85, 93}

[IN:Q.1211] ITU-T, 1993: *Introduction to Intelligent network Capability set 1,* General recommendations on telephone switching and signaling, Intelligent network, Mar 1993. {24, 150, 153, 160, 161}

[IN:Q.1213] ITU-T, 1993: *Global functional plane for intelligent network CS-1,* General recommendations on telephone switching and signaling, Intelligent network, Mar 1993. {23, 61, 99, 102, 110, 111, 115, 118, 120, 122, 123, 125, 126, 128, 131}

[IN:Q.1214] ITU-T, 1993: *Distributed functional plane for intelligent network CS-1,* General recommendations on telephone switching and signaling, Intelligent network, Mar 1993. {xv, 23, 24, 63, 64, 70, 71, 102}

[IN:Q.1215] ITU-T, 1993: *Physical plane for intelligent network CS-1,* General recommendations on telephone switching and signaling, Intelligent network, Mar 1993. {24}

[IN:Q.1218] ITU-T, 1993: *Interface recommendation for intelligent network CS-1,* General recommendations on telephone switching and signaling, Intelligent network, Mar 1993. {24}

[IN:Q.1223] ITU-T, 1995: *Global functional plane for intelligent network CS-2 (Draft),* General recommendations on telephone switching and signaling, Intelligent network, Apr 1995. {23, 117, 120}

[IN:Steffen96] Steffen, B., Margaria, T., Claßen, A., Braun, V., Reitenspieß, M., 1996: *An Environment for the Creation of Intelligent Network Services,* Proceedings of Advanced Intelligent Networks '96 workshop, ed. T. Margaria T, pp.117–136, 1996. {17}

[IN:Tsang94] Tsang, S., Magill, E.H., 1994: *Detecting feature interactions in the Intelligent Network,* Feature Interactions in Telecommunication Networks, eds. Bouma L.G. and Velthuijsen H., pp.236–248, IOS Press, 1994. {6}

[IN:Verweij93] Verweij, H., 1993: *Service Independent Building Blocks II (DRAFT),* Unpublished report—private communications, University of Amsterdam, Programming Research Group, P.O. Box 41882, 1009 DB Amsterdam, The Netherlands, Jun 1993. {6, 16}

Feature Interaction

[FI:Accorsi00] Accorsi, R., Areces, C., Bouma, W., de Rijke, M., 2000: *Features as Constraints,* Feature Interactions in Telecommunications and Software Systems VI, eds. Calder M. and Magill E., pp.210–225, ISO Press, 2000. {14}

[FI:Aggoun97] Aggoun, I., Combes, P., 1997: *Observers in the SCE and the SEE to Detect and Resolve Service Interactions,* Feature Interactions in Telecommunication Networks IV, eds Dini P., Boutaba R. and Logrippo L., pp.198–212, IOS Press, 1997. {4, 16}

[FI:Blom94] Blom, J., Jonsson, B., Kempe, L., 1994: *Using Temporal Logic for Modular Specification of Telephone Services,* Feature Interactions in Telecommunication Networks, eds. Bouma L.G. and Velthuijsen H., pp.197–216, IOS Press, 1994. {5, 14}

[FI:Blom95] Blom, J., Bol, R., Kempe, L., 1995: *Automatic Detection of Feature Interactions in Temporal Logic,* Feature Interactions in Telecommunication Networks III, eds. Cheng K.E. and Ohta T., pp.1–19, IOS Press, 1995. {5, 14}

[FI:Blom97] Blom, J., 1997: *Formalization of Requirements with Emphasis on Feature Interaction Detection,* Feature Interactions in Telecommunication Networks IV, eds Dini P., Boutaba R. and Logrippo L., pp.61–77, IOS Press, 1997. {6, 14}

[FI:Bowen88] Bowen, T.F., 1988: *Views on the feature interaction problem,* Technical Report TM-ARH-012849, Bellcore Inc., 1988. {2}

[FI:Braithwaite94] Braithwaite, K.H., Atlee, J.M., 1994: *Towards Automated Detection of Feature Interactions,* Feature Interactions in Telecommunication Networks, eds. Bouma L.G. and Velthuijsen H., pp.36–59, IOS Press, 1994. {15}

[FI:Calder98] Calder, M., 1998: *What Use are Formal Design and Analysis Methods to Telecommunications Systems,* Feature Interactions in Telecommunications and Software Systems V, eds. Kimbler K. and Bouma L.G., pp.23–31, IOS Press, 1998. {13}

[FI:Cameron93a] Cameron, E.J., Velthuijsen ,H., 1993: *Feature Interactions in Telecommunications Systems,* IEEE, Communications Magazine, Vol.31, No.8, pp.18–23, 1993. {3}

[FI:Cameron93b] Cameron, E.J., Griffeth, N., Lin, Y.-J., Nilson, M.E., Schnure, W.K., Velthuijsen, H., 1993: *A Feature-Interaction Benchmark for IN and Beyond,* IEEE Communications Magazine, Vol.31, No.3, pp.64–69, 1993. {2, 6}

[FI:Cameron94] Cameron, E.J., Griffeth, N., Lin, Y.-J., Nilson, M.E., Schnure, W.K., Velthuijsen, H., 1994: *A Feature-Interaction Benchmark for IN and Beyond,* Feature Interactions in Telecommunication Networks, eds. Bouma L.G. and Velthuijsen H., pp.1–23, IOS Press, 1994. {2, 6}

[FI:Capellmann97] Capellmann, C., Combes, P., Pettersson, J., Renard, B., Ruiz, J.L., 1997: *Consistent Interaction Detection - A Comprehensive Approach Integrated with Service Creation,* Feature Interactions in Telecommunication Networks IV, eds Dini P., Boutaba R. and Logrippo L., pp.183–197, IOS Press, 1997. {8}

[FI:Griffeth98] Griffeth, N., Blumenthal, R., Gregoire, J.-C., Ohta, T., 1998: *Feature Interaction Detection Contest,* Feature Interactions in Telecommunications and Software Systems V, eds. Kimbler K. and Bouma L.G., pp.237–259, IOS Press, 1998. {vi, 5}

[FI:Jonsson99] Jonsson, B., Margaria, T., Naeser, G., Nyström, J., Steffen, B., 1999: *On Modelling Feature Interactions in Telecommunications,* Proceedings of Nordic Workshop on Programming Theory, eds. Victor B. and Yi. W., 1999. {2, 18}

[FI:Jonsson00] Jonsson, B., Margaria, T., Naeser, G., Nyström, J., Steffen, B., 1999: *Incremental Requirement Specification for Evolving Systems,* Feature Interactions in Telecommunications and Software Systems VI, eds. Calder M. and Magill E., pp.145–162, ISO Press, 2000. {2, 18}

[FI:Keck96] Keck., D.O., 1996: *Identification of Call Scenarios with potential Feature Interactions,* Proceedings of Advanced Intelligent

Networks '96 workshop, ed. T. Margaria T, pp.42–55, 1996. {15}

[FI:Keck98a] Keck., D.O., 1998: *A Tool for the Identification of Interaction-Prone Call Scenarios,* Feature Interactions in Telecommunications and Software Systems V, eds. Kimbler K. and Bouma L.G., pp.276–290, IOS Press, 1998. {15}

[FI:Keck98b] Keck., D.O., Kuehn, P.J., 1998: *The Feature and Service Interaction Problem in Telecommunication Systems: A Survey,* IEEE Transactions on Software Engineering, Vol.24, No.10, pp.779–796, 1998. {13}

[FI:Lin98] Lin, F.L., Liu, H., Ghosh. A., 1998: *A Methodology for Feature Interaction Detection in the AIN 0.1 Framework,* IEEE Transactions on Software Engineering, Vol.24, No.10, pp.797–817, 1998. {15}

[FI:Marples98] Marples, D., Magill, E.H., 1998: *The use of Rollback to prevent incorrect operation of Features in Intelligent Network Based Systems,* Feature Interactions in Telecommunications and Software Systems V, eds. Kimbler K. and Bouma L.G., pp.115–134, IOS Press, 1998. {14}

[FI:Naeser99] Naeser, G., Nyström, J., Jonsson, B., 1999: *A Case Study in Automated Detection of Service Interactions,* Proceedings of RadioVetenskap och Kommunikation '99, 1999. {2, 18}

[FI:Naeser00] Naeser, G., 2000: *A Flexible Framework for Detection of Feature Interactions in Telecommunication Systems,* Licentiate thesis, Department of Information Technology, Uppsala University, Sweden, 2000. {5, 13}

[FI:Zave95] Zave, P., 1995: *Secrets of call forwarding: A specification case study,* Proceedings of FORTE'95, pp.153–168, Chapman & Hall, 1995. {14}

[FI:Reiff00] Reiff, S., 2000: *Identifying Resolution Choices for an Online Feature Manager,* Feature Interactions in Telecommunications and Software Systems VI, eds. Calder M. and Magill E., pp.113–128, ISO Press, 2000. {15}

Part VII

Indices

Domain Index

Function Index

Constant Index

External References Index

Licentiate theses from the Department of Information Technology

2000-001 Katarina Boman: *Low-Angle Estimation: Models, Methods and Bounds*

2000-002 Susanne Remle: *Modeling and Parameter Estimation of the Diffusion Equation*

2000-003 Fredrik Larsson: *Efficient Implementation of Model-Checkers for Networks of Timed Automata*

2000-004 Anders Wall: *A Formal Approach to Analysis of Software Architectures for Real-Time Systems*

2000-005 Fredrik Edelvik: *Finite Volume Solver for the Maxwell Equations in Time Domain*

2000-006 Gustaf Naeser: *A Flexible Framework for Detection of Feature Interactions in Telecommunication Systems*

2000-007 Magnus Larsson: *Applying Configuration Management Techniques to Component-Based Systems*

2000-008 Marcus Nilsson: *Regular Model Checking*

2000-009 Jan Nyström: *A formalisation of the ITU-T Intelligent Network standard*

2000-010 *Markus Lindgren: Measurement and Simulation Based Real-Time System Analysis*

UPPSALA
UNIVERSITY

Department of Information Technology, Uppsala University, Sweden