# Mixing Formal Specifications

# Using ICL

# (InterConnection Language)

Author

Margaret O'Donnell B.Sc.

Supervisor

Prof. J. A. Moynihan

Submitted to

The School of Computer Applications

Dublin City University

for the degree of

Master of Science

September 1994

**Declaration**

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science in Computer Applications, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: M O'Donnell Date : 21-9-94

Margaret O'Donnell

## *Acknowledgements*

*I would like to thank my supervisor, Prof. Tony Moynihan, whose patience and guidance throughout this work proved invaluable. I would also like to thank my family, all my friends and fellow postgraduates for their encouragement and intriguing distractions. Finally, sincere and special thanks to Ken for his patience and support and especially for proof-reading this thesis.*

**Mixing Formal Specifications Using ICL (InterConnection Language)**

**Margaret O'Donnell** **September,1994**

## ABSTRACT

There is an increasing need and desire to develop systems by combining components that are written in different languages and/or run on different kinds of machines. Success largely depends on the ability of their components to communicate and work together despite their differing backgrounds.

This thesis addresses the problem of mixing two formal specification languages, SDL and LOTOS. Various approaches to mixing specification languages are examined including the SPECS approach which is presented in more detail. A unique feature of the SPECS approach is the support of multiple specification languages, including the ability to mix specifications languages within a given system design. This area of research investigates the SPECS specific mixing language ICL (InterConnection Language). The thesis looks at two formal languages, one of an asynchronous nature (SDL) and one of a synchronous nature (LOTOS), which can be combined using the InterConnection Language. Also a set of rules are given to produce this formal mixing specification from less formal descriptions. These rules use a range of informal representations and rigorous models of the required system to produce of the ICL specification. An application of these rules is presented.

This research work was carried out as part of DCU's contribution to the SPECS (Specification and Programming Environment for Communication Software) project, part of the RACE program of the EC. SPECS's aim was to, as much as possible, automate the software development process by using formal languages. An overview of the SPECS project is presented in chapter 1 of this thesis.

# Chapter 1 An Introduction to SPECS and the Mixing of Formal Languages

## 1.1 Introduction

This chapter gives an introduction to the mixing of formal specification languages and describes the research work that I contributed to the *SPECS*[1] project, part of the *RACE*[2] programme of the EC. The language, for mixing formal specification languages, on which I focused my research is the **"InterConnection Language"** *ICL*[3]. It is the mixing language used by SPECS to mix the system with its environment and its sub-components that are specified in a different language.

The formal specification languages used by the SPECS project are *SDL*[4] and *LOTOS*[5]. An ICL specification allows specifications written in SDL and LOTOS to interact. A system can be divided into sub-components, specified in a language best suited to represent its behaviour and combined into one specified system, via an ICL specification. A set of rules, the "ICL Production Rules" which I devised, is given to derive an ICL specification from the products of each of the development processes used in SPECS for producing formal specifications. The description of these rules and their application to a "pilot case study" is the core of my dissertation and also my personal contribution to SPECS. An overview of the mixed specified system is given in Figure 1.1.

SPECS is one of four RACE software technology projects. The RACE SPECS project covers the tasks pertaining to the techniques for the specification, design,

[1] SPECS Specification and Programming Environment for Communications

[2] *RACE* Research and Development in Advanced Technologies in Europe

[3] InterConnection Language

[4] Specification and Description Language

[5] Language Of Temporal Ordering Specification

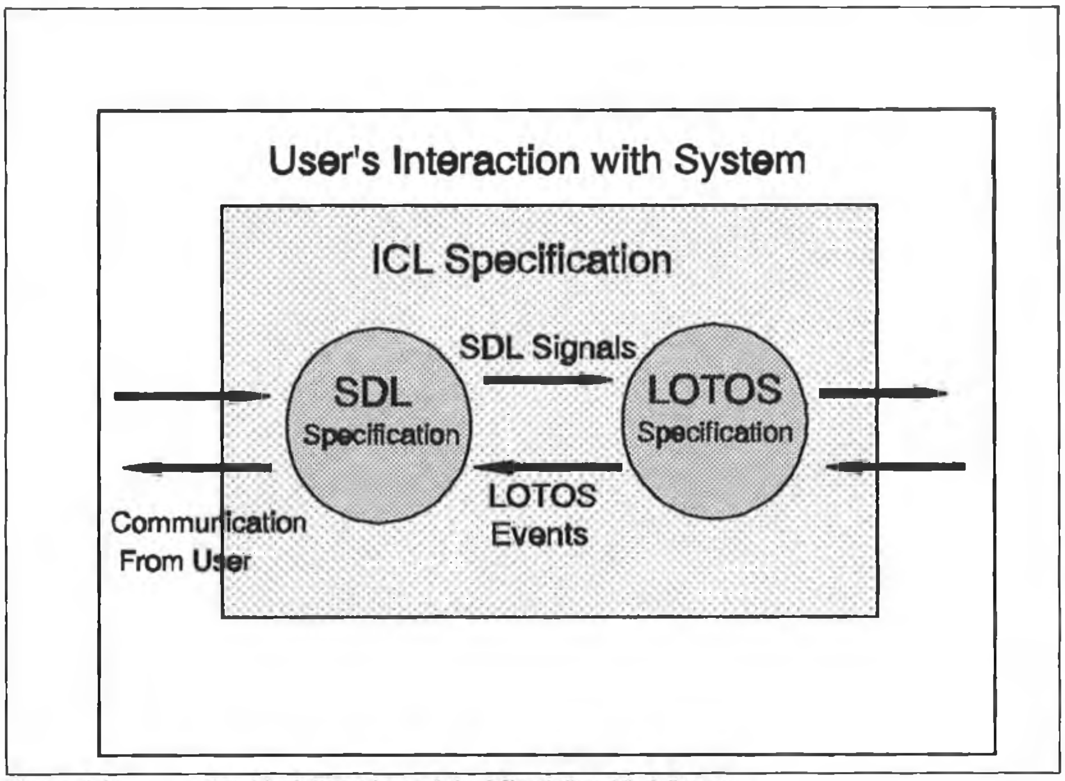

Figure 1.1 Graphical Overview of the Mixed Specified System.

implementation, testing and reuse of *IBC*[6] software. The primary aim of the SPECS project is to provide maximum automation and optimisation of the software engineering of IBC software within its software development life-cycle.

The project was organised into nine workpackages in which each workpackage was assigned an aspect within this life-cycle. The aspect of the development process life-cycle which DCU was assigned was the production of Formal Specifications. The formal specifications LOTOS and SDL with the ICL specification to glue these two components together, when completed, would be passed on to the workpackage which was assigned the next step in the software development life-cycle. This development process life-cycle is described in this chapter in section 1.3.2. From these formal specifications a high-level programming language is produced from SPECS code generation tools, thus achieving SPECS aim to automate and optimise the software engineering of IBC software.

[6] *IBC* Integrated Broadband Communications (Unifying the various means of transmission (terrestrial, mobile satellite..) and various kinds of services offered on this telecommunications network.

## 1.2 How Mixed Specifications are used in SPECS

A unique feature of the SPECS environment is the ability to handle several specification languages separately or even co-existing within a particular specification. It was the conclusion of the "definition phase" of the SPECS project that none of the current specification languages such as SDL, LOTOS and ESTELLE supports all the needs for a specification of telecommunication systems, but that most desirable features are available somewhere amongst the languages. However, a compromise has to be made to some extent where two languages like SDL and LOTOS have different paradigms. LOTOS assumes synchronous communication whilst in SDL it is asynchronous. Although both FDTs[7] can be handled by the semantic layer of the tools developed by SPECS, an additional language to define the inter-connection between them has to be designed and implemented to enable the handling of the complete mixed FDT systems. Thus SPECS has defined a scheme for combining specifications in LOTOS and SDL, bridging the worlds of synchronous and asynchronous specification languages. This mixing scheme is ICL "*InterConnection Language*". This language maps the outputs of LOTOS events to incoming queues in SDL and also maps SDL outgoing signals directly to gates to activate LOTOS events. This offers the possibility to specify parts of large systems in the best suited language and to reuse existing specifications.

## 1.3 The SPECS Project

### 1.3.1 SPECS's Goals and Achievements

The global objective of SPECS was to define a methodology that would *promote the use of formal description languages throughout the methodology to enable the engineering of IBC software to be of better quality, quicker and cheaper.* The software development life-cycle from informal requirements to executable and tested code is

[7] FDT Formal Descriptive Techniques

covered. The goals included in this are:

- methods for the analysis of informal specifications and the generation of formal specifications in LOTOS and SDL.

- methods for the analysis of formal specifications, based on their semantics, to detect errors as soon as possible in the development process.

- methods for the generation of implementations from formal specifications.

- methods for the generation of TTCN[8] test cases from formal specifications and the verification and the execution of these test cases

- methods for the reuse of components in the various development activities.

SPECS has tried to adapt the possibilities offered by the achieved goals to practical applications in industrial software development. A Pilot Case Study which offered many of the characteristics of an industrial development was carried out to test the SPECS methodology. The case study has validated several aspects of the SPECS methodology and the "*ICL Production Rules*" which I defined. The feedback provided was beneficial and taken into account in the final project deliverables. A detailed account of the Pilot Case Study can be found in Chapter 5.

To distribute the architecture and methodology of the project, SPECS produced **"SPECS - THE BOOK: Synopsis"**[SPECS 93]. The book was designed for readers who are engineers or information technologists and are working in the field of telecommunications, or readers who are interested in the use of FDTs. The book itself is not an evaluation of FDTs but provides an overview of the detailed project work and shows results of the SPECS approach.

[8] Tree and Tabular Combined Notation. See [ISO 90].

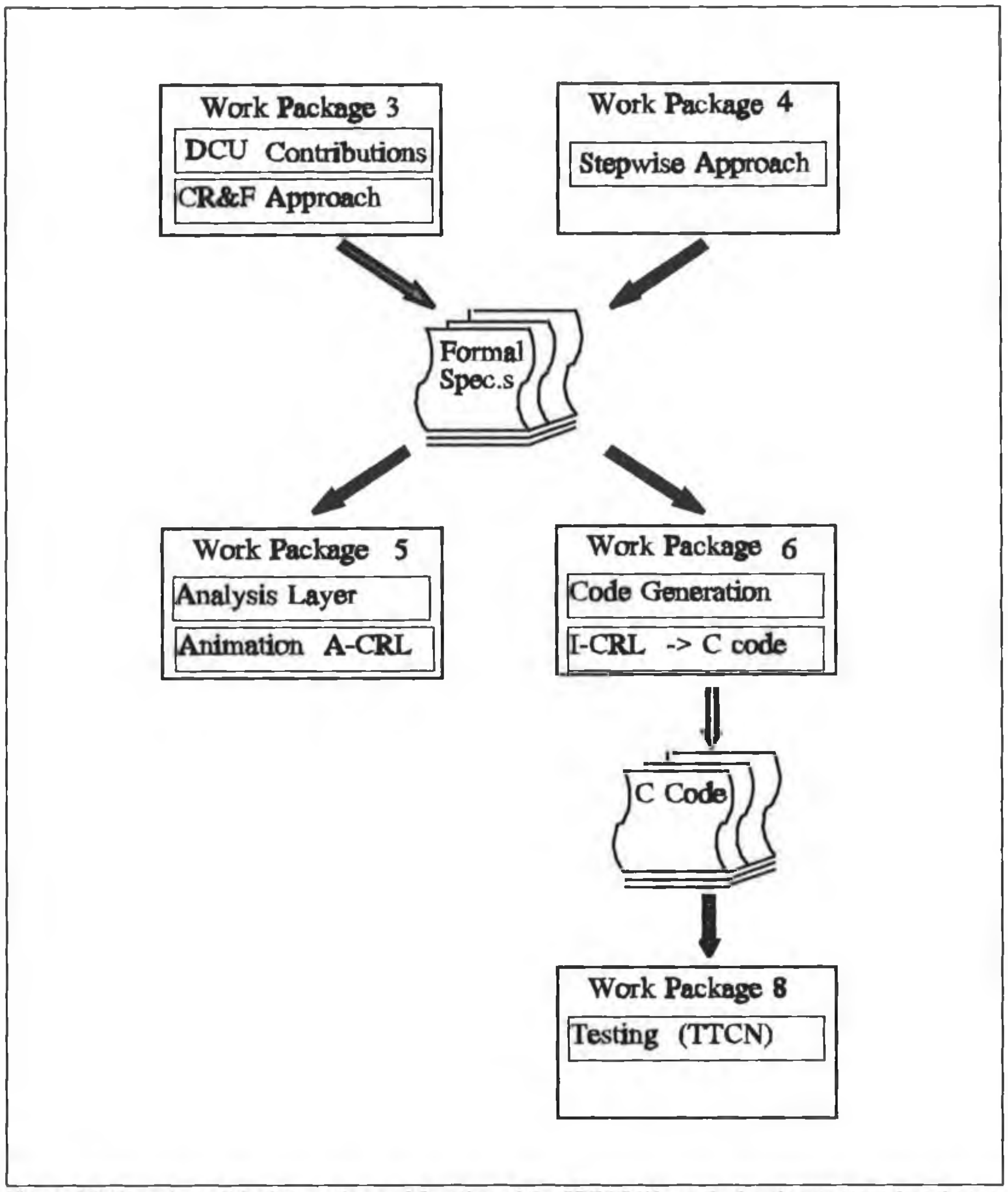

Figure 1.2 Overview of the interaction and function of the SPECS life-cycle development workpackages.

### 1.3.2 Software Development Life-cycle Work Packages

The following are the related workpackages to the software Development life-cycle:

Work Package 3 **SPECS-Specification Generation**. This workpackage has been concerned with the development and refinement of a methodology and of tools for generating formal specifications from informal ones. This methodology is knows as the CR&F[9] process. DCU provided its contributions to this work package.

Work Package 4 **SPECS-Specification Handling.** The role of this workpackage has been the development and maintenance of the SPECS tower languages, methods and tools to handle formal specifications (SDL and LOTOS). The tower languages are the main languages that SPECS focused on and are shown in figure 1.3. The functions of this package was the refinement of methodologies for the stepwise production of formal specifications (which have been embedded in the CR&F approach), the support for refinement of specifications towards implementation and the development of proprietary languages for LOTOS and SDL.

Work Package 5 **SPECS-Semantics and Analysis.** Each of the tower languages of SPECS can be translated into a common internal representation, called MR/CRL[10]. The MR/CRL consists of a mathematical representation model (MR) and the concept of a machine interpretable common representation language (CRL). The MR/CRL is split into analysis A-CRL and implementation I-CRL. This work package developed tools and techniques for the structural and dynamic analysis of A-CRL specifications and defined a generic scheme for the properties of specifications on both the tower and A-CRL level.

Work Package 6 **SPECS-Implementation Generation.** This workpackage has been based on the development of the implementation CRL (I-CRL) to which formal specifications are translated and from which implementation code[11] is derived. The generated code relies on a Run-Time Environment (*RTE*), also developed by WP6, which provides the environment to execute the generated code. This work package developed the tools to translate the formal specifications into I-CRL, then translate the I-CRL into a programming language (in this case 'C'), plus generating a portable Run-

[9] CR&F Classification, Rigorisation and Formalisation
(See Section 1.8 for a brief explanation on each)

[10] Mathematical Representation and its Common Representation Language

[11] The C programming language is the current implementation language.

Time Environment. These tools have been tested by the *Racebank* case study [DWP3.8 92] and used by the *Pilot* case study (Appendix A), two case studies that DCU contributed to the SPECS project. For the *Racebank*, apart from generating the core code of the system, the environment code was written also, which establishes the connection between the signals and events of the system and the external world. The ICL specification which would define these connections was contributed by DCU.

Work Package 8 **SPECS-Testing.** This workpackage was involved in identifying the functional and non-functional properties of IBC systems for testing purposes. This work package developed a test language comprising of extensions to the ISO standard test language TTCN. The responsibility of this work package was the refinement of methods for automatic generation of tests from specifications and the definition of a test facility to manage test sessions.

## 1.4 Tools used and developed in SPECS.

SPECS proposes methods and tools applicable to the entire IBC software development life cycle. SPECS methods and tools provide facilities for the handling of specifications in SDL and LOTOS, particularly: mixing of languages, use of SPECS data types, help tools, support for SDL-92, reporting back from the CRL to the user at the LOTOS/SDL level. The tools were installed and run on a UNIX[12] based workstation with X Windows[13] and the Concerto[14] development environment. The style of the screen layout is based on the use of overlapping windows appearing on the screen, menus (permanent and popup) for access to the tools, and interaction through either the keyboard or the mouse.

[12] UNIX is a trademark of AT&T Bell Laboratories

[13] X Windows is a trademark of M.I.T.

[14] Concerto is a trademark of Sema Group

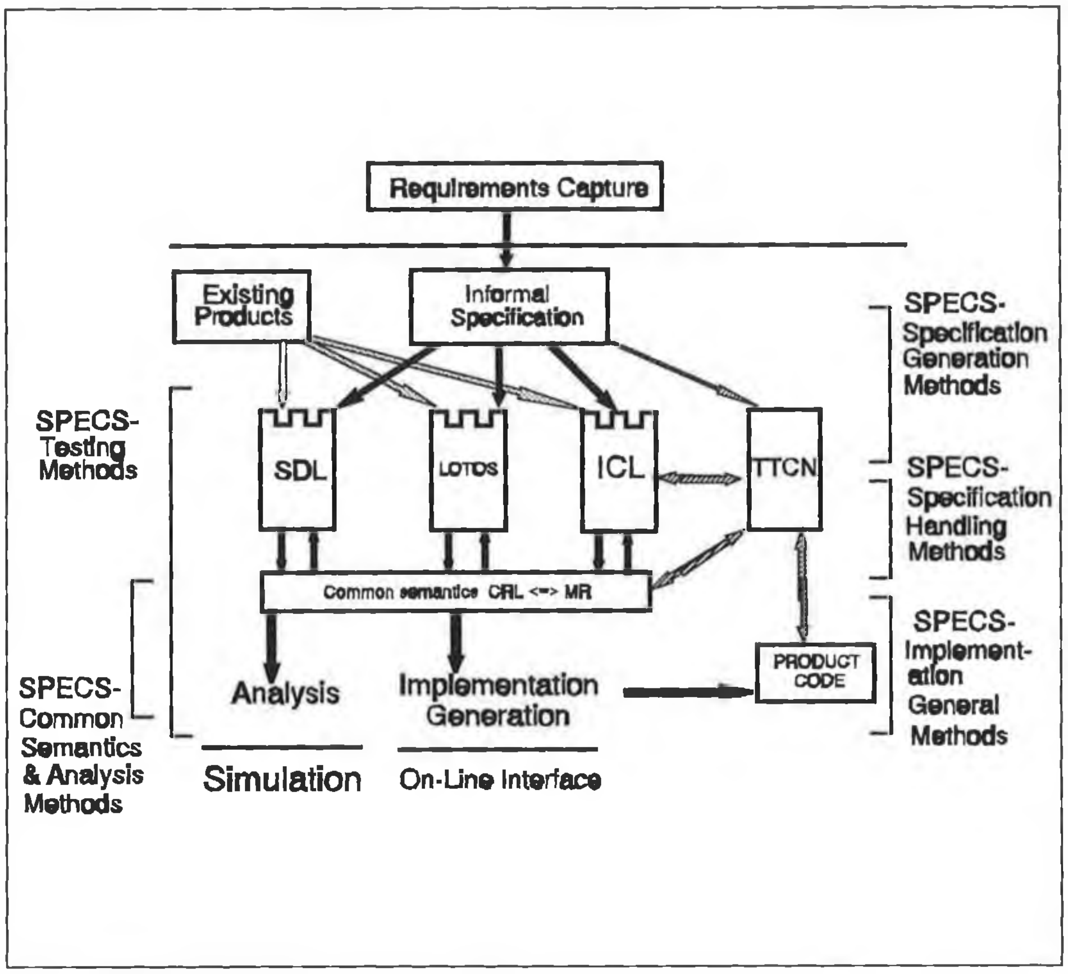
Requirements Capture
Existing Products
Informal Specification
SPECS-Specification Generation Methods
SPECS-Testing Methods
SDL
LOTOS
ICL
TTCN
SPECS-Specification Handling Methods
Common semantics CRL <=> MR
SPECS-Implementation General Methods
PRODUCT CODE
SPECS-Common Semantics & Analysis Methods
Analysis
Implementation Generation
Simulation
On-Line Interface

Figure 1.3 The SPECS architecture

## 1.5 SPECS Architecture for handling Specifications

The SPECS architecture, adapted from generic compiler architectures, allows towers to be built for each specification language, a tower having a translator to a common internal representation. Central in the architecture is this internal representation. An abstract formalisation, MR/CRL, has been developed that has an expressive power exceeding that of the standard specification languages, and that is capable of being extended with many other concepts. MR/CRL serves as target for translators from all specification languages used, and as source for compilers to implementation languages and operating systems interfaces, as well as to representations suited for simulation and analysis. Thus the architecture is "open" to allow a number of targets for the generation of code. An important feature resulting from this architecture is the ability to mix specification languages within a given specification. Parts of a given specification can each be done in the most appropriate specification language, yet, because of the common internal representation, such mixed specifications can be analysed, animated, phototyped, automatically implemented and tested. The SPECS architecture can be seen in figure 1.3.

## 1.6 DCU's Role in software development life-cycle of SPECS

SPECS was organised into nine workpackages. Each workpackage was assigned an aspect of the development life-cycle in SPECS (section 1.3.2). The aim of the workpackage to which DCU contributed was the development of formal specifications using a defined methodology and various methods and tools developed within SPECS. The Work Package is known as "Work Package 3" or "WP3" and DCU contributed its deliverables to this work package.

WP3 was responsible for the development of methods and tools to help produce formal specifications from a purely informal description of what the user intended but may include parts that are already in a more rigorous form. The input to this process is a mixed informal document whose content is expressed in some combination of natural language, of expressions such as data flow diagrams or entity relationship diagrams coming from particular analysis techniques, and of formal languages. The methods and tools developed in WP3, rely on a "divide and conquer" strategy to cope with the complexity of the problem. The method divides the production of the formal specification into three generation processes, the classification, rigorisation and the formalisation processes.

The **classification process** (based on a conceptual model of the application) where all aspects of the desired system are identified and are defined as a set of application concepts. The **rigorisation process** (based on general purpose techniques e.g. data flow diagrams, message sequence charts and entity relationship diagrams) where the behaviour of the system is modelled. The **formalisation process** (based on the formal languages SDL and LOTOS) where the system is specified formally using the products of the previous two processes. These development processes are collectively known as the **CR&F methodology** (Classification, Rigorisation and Formalisation). From the specifications produced by these three processes I defined rules and guidelines to produce the ICL specification. This rules are given in detail in chapter 3.

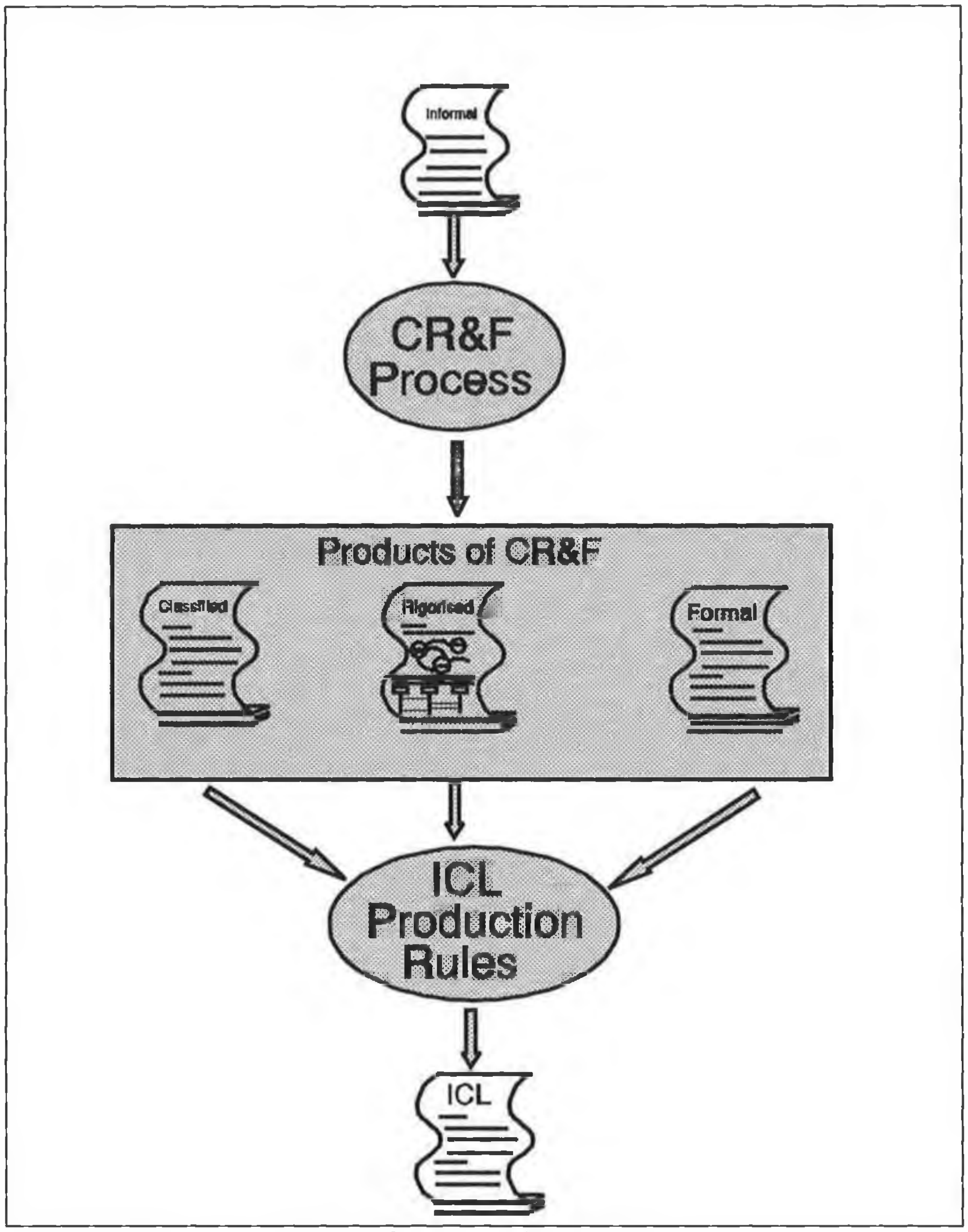

Figure 1.4 Overview of ICL Production Rules in Relation to CR&F Methodology

## 1.7 My Role In SPECS.

Work in SPECS involved the production of methods and tools to derive formal specifications from informal ones by using the CR&F methodology developed by SPECS. From the products of the CR&F process a mixed specification was produced to mix these formal specifications. The area chosen for my research were the methods used to produce mixed specifications. A set of rules, the *"ICL Production Rules"*, was defined to automate the production of an ICL specification using the products of the CR&F methodology. These products are the Classification specification, Rigorous specification and the Formal specification. Figure 1.4 illustrates an overview of the *"ICL Production Rules"* in relation to the CR&F Methodology. The production of the ICL specification was standardised to follow a structure similar to the CR&F process. These rules assume no detailed knowledge of the InterConnection Language used to mix the specifications. Constructing these rules involved producing the CR&F products to give correct and concise formal specifications, and defining steps where information is extracted from each specification to construct the mixing specification. The results of my work were contributed to the deliverables produced by DCU [DWP3.8 92] and to the book by SPECS [SPECS 93].

## 1.8 Overview of the thesis

This chapter described the goals and achievements of the SPECS project, its work structure, integration of work packages and the architecture chosen to achieve its aim. Also an account of my role in the SPECS research project and its relevance to my area of study is presented.

Chapter 2 describes the two formal specification languages, SDL and LOTOS. It presents a description of each language, a comparison of the two, and their limitations. Chapter 3 looks at other approaches to allow different specification or programming languages to be mixed with each other. It also describes the InterConnection Language, which is designed to support the mixing of SDL and LOTOS. Transformation mappings between each language are given and also an account of how the mixing works between a SDL specification and a LOTOS specification via an ICL specification. Chapter 4 presents an approach to tackle the problems of combining formal specification languages. This approach involves combining the two formal languages SDL and LOTOS by following the set of rules, which I defined, to produce the necessary ICL specification. These rules are known as the "*ICL Production Rules*". The SPECS Pilot Case Study given in Chapter 5 is based on one of the ISDN[15] supplementary services. The case study was taken from CET[16] ELDIS[17] switch project. ELDIS is an ISDN rural exchange targeted at the Portuguese market. The *ICL Production Rules* are applied to this case study as a worked example. An informal specification of the case study is given in Appendix A. The products of the software life-cycle development that DCU implemented for the Pilot Case Study, including the ICL specification are given in the rest of the Appendices. Chapter 6 gives the conclusions to my work.

## 1.9 Summary

This chapter described the RACE Project SPECS and its primary aim - *to automate as far as possible the software development process*. The automation process of software development incorporated two formal specification languages, SDL and LOTOS. The ability to mix these specification languages would prove beneficial to an analyst wishing to specify different behaviour aspects of a system in the most appropriate specification language. The SPECS software development life-cycle is presented, showing how DCU functioned in this life-cycle. Also an overview of my research and my contributions to SPECS are given.

[15] Integrated Services Digital Network

[16] Centro de Estudos de Telecomunicacoes

[17] Estacao Local Digital com Integracao de Servicos

# Chapter 2 The Formal Specification Languages to be Mixed(SDL & LOTOS)

## 2.1 Introduction.

This chapter describes the two formal specification languages used within SPECS to represent the behaviour of systems. The formal specification languages SDL and LOTOS are presented in turn. At the end of this chapter a comparison of the two languages is given. The InterConnection Language (ICL) allows the mixing of these two languages.

## 2.2 SDL Specification and Description Language.

### 2.2.1 SDL - A Brief Overview.

#### 2.2.1.1 History

SDL is a standard language for the specification and description of systems. CCITT (International Telegraph and Telephone Consultative Committee) has developed and standardised it. It was first defined by Recommendations Z.101 to Z.103 in 1976 and later extended in Recommendations Z.101 to Z.104 in 1980. It was further extended and reorganised in 1984 [BELI 88]. To date SDL has been extended and harmonised to a single mathematical definition.

#### 2.2.1.2 Application Area

As the language was developed for the purpose of specifying telecommunication systems including data communications, it also can be used in all real time and interactive systems. Systems can be developed and understood one part at a time, which is essential for distributed systems. SDL may be used to represent, at various levels of detail, the functional properties of a system. If a system can be effectively modelled by communicating Extended Finite-State Machines then this system's behaviour could successfully be described in SDL eg. functions for telephone, telex, data switching.

### 2.2.1.3 System Description

The system is what the SDL description specifies, an abstract machine communicating with its environment, and contains everything the specification is trying to define. It communicates with the environment via channels. A system diagram contains the following elements:

- **system name**. (Identification of the system).
- **signal description**. (This element contains a signal name and the types of values conveyed by the signal).
- **channel description**. (This element contains a channel name, a list of signal names for signals that can be transported by the channel, and the identification of the endpoints of the channel - block or environment).
- **data type description**. (Description of the Data types used within the system e.g Booleans, natural).
- **block description**. (A block is a part of the system that can be treated in various respects (development, description, understanding, etc) as a self-contained object).

## 2.2.2 Behaviour Description

The behaviour of a system depends on the combined behaviour of a number of processes in the system. A process is an extended finite state machine, that works autonomously and concurrently with other processes [BELI 88]. Communication between processes is performed asynchronously by discrete messages, called signals. Signals can be received and sent by the processes to/from the environment of the system (See Figure 2.1). Its reaction to external stimuli (in the form of signals) can be predicted, and is in accordance with its description. Each process has:

- unique address (PId). Each signal has the address of the sending and the receiving process, along with possible data values. Thus the receiving process is always aware of the address of the sending process.

- ability to store variables, along with the state information.

- a infinite input queue for the queuing of incoming signals. When a signal arrives a transition occurs then the signal is removed from the input queue (Figure 2.2). In a transition, variables can be manipulated, or created; also new processes can be created and signals can be sent.

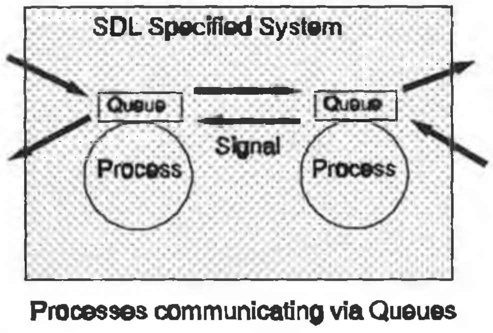

Figure 2.1 Communication between processes is performed by signals.

### 2.2.2.1 Use of Blocks in SDL

Systems that are specified by a formal specification language are usually quite complex so it is necessary to have a structuring concept to break down the complexity of the system. In basic SDL, a system description is structured into block descriptions and process descriptions as shown in Figure 2.3. An SDL system would contain one or more blocks, interconnected with each other and with the boundary of the system by channels. Within a block, processes can communicate with one another either by

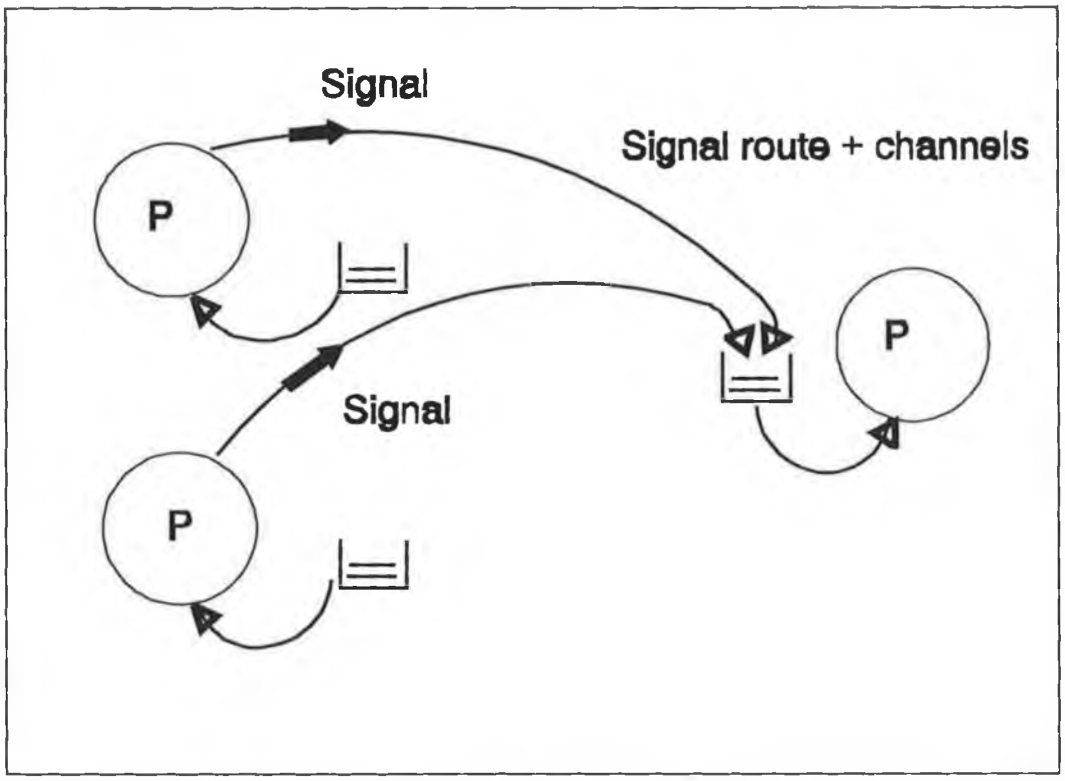

Figure 2.2 Queuing of incoming signals via channels

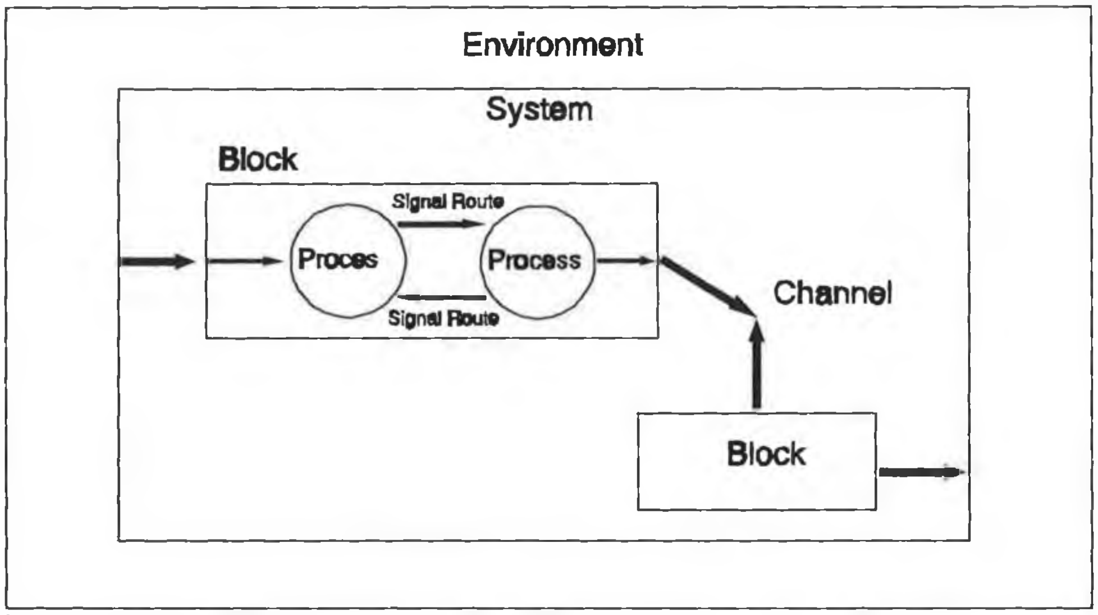

Figure 2.3 System Behaviour

signals or shared values. Blocks provide:

- a convenient mechanism for grouping processes.
- a boundary for the visibility of data.

Thus the grouping of processes within a block is a reasonable functional grouping. When designing a system using blocks, break the system into functional units, then assign processes to these units. A block can be partitioned into (sub)blocks and channels, similarly to the partitioning of the system according to figure 2.3. Repeated block partitioning results in a block tree structure (with the system as the root block) [HOGR 88]. Signal routes are communication paths between processes within the block or between processes within the boundary of the block. A block diagram is similar to a system diagram.

#### 2.2.2.2 Processes in SDL

A Process is an extended finite state machine which defines the dynamic behaviour of a system. Before a process receives a signal it is in an awaiting state. The process then responds by performing the specific actions that are specified for the receiving signal.

The different states of a process allow it to perform different actions in response to receiving a signal. With each state, the memory of actions performed are provided. When all of the actions associated with that receiving signal have occurred, the next state is entered and the process returns to its awaiting state. Processes can be created or terminated by other processes. Several instances of the same process type may be created and exist at the same time. They can also be executed independently and concurrently.

Should a process perform a stop action and there are pending signals that were sent to it and not yet received, then such signals are discarded.

#### 2.2.2.3 Data Handling

##### 2.2.2.3.1 Abstract Data Types

In order to describe data types in SDL, we use the abstract data type (ADT) approach.

This means that all data types are defined in an implementation independent way, in terms of their properties alone. The definition of an abstract data type has three components.

- set of values
- set of operations
- set of axioms defining the operations

```
SORTS  bool
OPNS   true     :        ->bool
       false    :        ->bool
       not      : bool   ->bool

EQNS
       not(true)       =      false
       not(false)      =      true
```

Table 2 ADT example

In the most simple case, an ADT is a set of objects hereafter called *sort*, together with a number of operators on this set. Standard operators may have arguments of different sorts, and the results of the operators are always in one of the sorts. Thus an ADT = *sorts* + *operators*. SDL abstract data types provide a powerful means of specifying data by describing their behaviour rather than their implementation [BARR 85]. There are three aspects to describing ADTs, which are the *signature*, *terms* and *equations*. The names of the *sorts* and *operators*, together with the definition of domains and codomains of the operators, are called the *signature* of the ADT. A *term* is the combination of applications of the operators to the elements of a *sort*. Equations between *terms* specify which *terms* represent the same element of a given *sort*. For example not(false) = true specifies that both terms represent the same value. Table 2 shows an ADT which has the sorts "boolean" and its various operators. Use of these data types is shown in the SDL specification for the pilot case study in Appendix D. Abstract data types can be described only once in the system description.

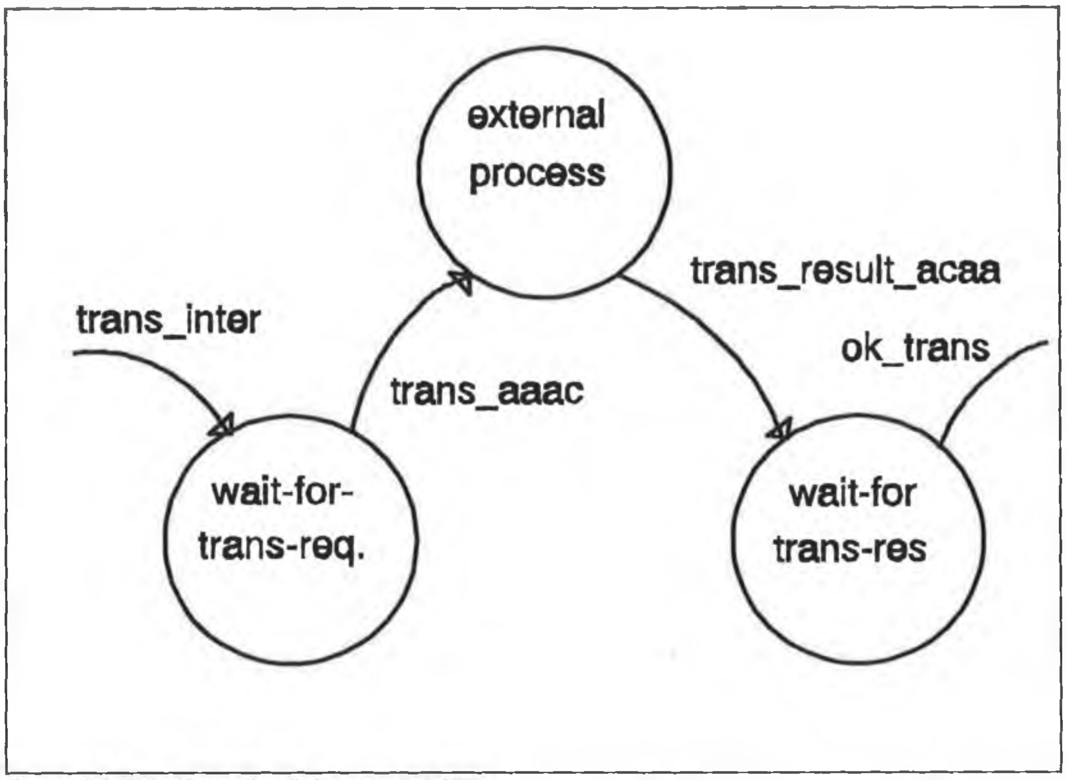

Figure 2.4 Behaviour of a finite state machine.

### 2.2.2.4. States and Transitions

Process behaviour may be specified either by a single monolithic state transition diagram or by a set of partial state transition diagrams. Figure 2.4 shows the behaviour of the system defined by a directed graph. The behaviour of this system is to validate a users transaction [D.WP3.8 92].

The system inputs the users identification and necessary details from the incoming signal *trans_inter* and checks this information with an external process. The external process acts as a data source which returns TRUE or FALSE, to which our system then responds. Two states *wait_for_trans_req* (wait for the user request) and *wait_for_trans_res* (wait for the result) are involved . If the machine is in state *wait_for_trans_req* it can perform a transition to state *wait_for_trans_res* initiated by an output *trans_aaac* to the external process. During the transition an input signal from the external process is received. The output of this signal will result in a transition to the starting state.

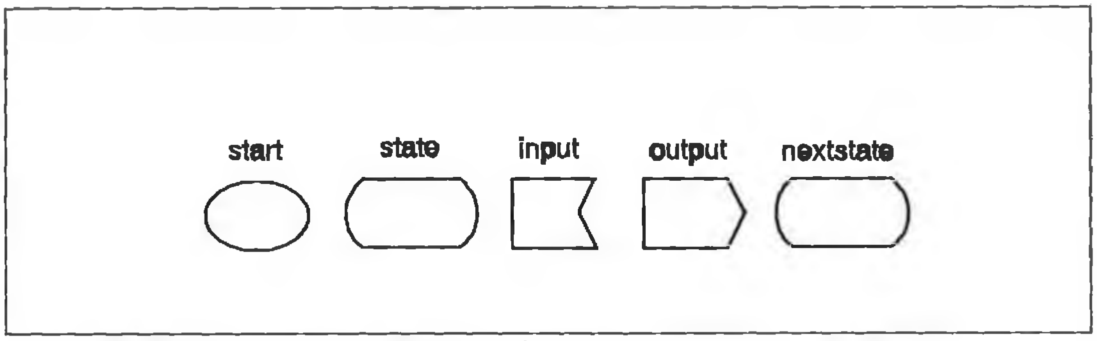

Figure 2.4a Basic constructs for the description of a process.

Using the five basic constructs for the description of a process : start, state, input, output, and nextstate as shown in figure 2.4a, a description of our system by a process diagram is given in figure 2.4b. From the process diagram the SDL specification code can be produced.

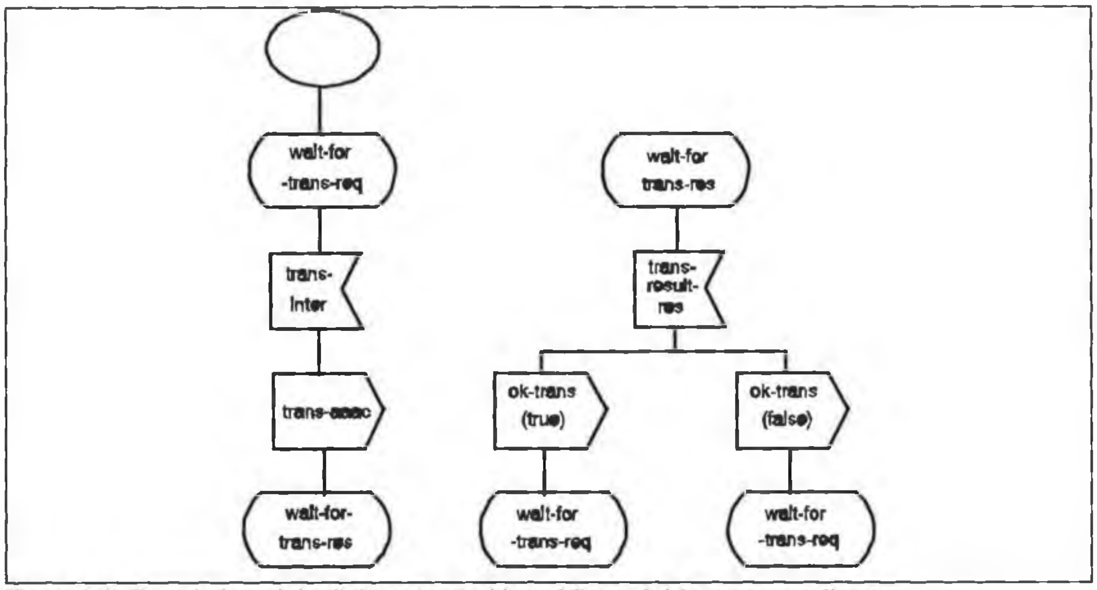

Figure 2.4b Description of the finite state machine of figure 2.4 by a process diagram.

### 2.2.3 Conclusion on SDL.

From this chapter we see that every process is associated with an input queue, which acts like a FIFO queue. Any signal arriving at the process and belonging to its so called 'complete valid input signal set' is put into the input queue. (see figure 2.2 Process communication). If in a given state the input queue is not empty, the first signal (in FIFO order) is removed from the queue (it is consumed). It is checked whether this signal can initiate a transition. If so, the transition is performed; if not, the signal is discarded. Sometimes the discarding of a signal is not required, thus the signal should be saved for future use. SDL has the save construct for this. The queuing aspect of SDL is taken into consideration when mixing the specification with another.

A comparison is given at the end of this chapter between SDL and LOTOS. SPECS takes these characteristics of SDL into consideration when using the interconnection language to mix specifications.

## 2.3 LOTOS Language Of Temporal Ordering Specification

### 2.3.1 LOTOS - A Brief Overview

#### 2.3.1.1 History

LOTOS has been standardised as ISO[18] 88077 by the joint ISO and IEC committee JTC1/SC21 on OSI[19]. It was developed by FDT experts from ISO/TC97/SC21/WG1 ad hoc group on FDT/Subgroup C during the years 1981-86. LOTOS is intended for formally describing OSI Standards, but has much wider applicability. As a FDT it is generally applicable to distributed, concurrent, information processing systems. LOTOS has been extensively applied to the description of OSI Standards in Layers 2 to 7. However it is a general-purpose, formal specification language for describing concurrent and distributed systems.

The basic idea from which LOTOS was developed was that systems can be specified by defining the temporal relation among the interactions that constitute the externally observable behaviour of a system. Contrary to what the name seems to suggest, this description technique is not related to temporal logic, but is based on process algebraic methods [LUIG 90]. The formal mathematical model of LOTOS is based on a mixture of the principles of Milner's Calculus of Communicating Systems (CCS) [MILN 85]

[18] International Organisation for Standardisation

[19] Open Systems Interconnection

and Hoare's Communicating Sequential Process (CSP) [HOAR 85] .

### 2.3.1.2 Structure of the Language

There are two components to the language, the first one deals with the description of process behaviours and interactions known as the control component. Most of the theoretical framework of the control component and especially the concept of internal action are based on Milner's work. Non-determinism is modelled by internal actions as in [MILN 85]. The rendezvous semantics follow Hoare's "multi-way rendezvous" concept, by which all processes that share a gate must participate in a rendezvous on that gate. The second component deals with the description of data structures and value expressions, the data type component. This part is based on the formal theory of abstract data types and equational specification of data types.

## 2.3.2 Behaviour Description

Dynamic behaviour is described in terms of processes which interact synchronously by events. A black-box view of dynamic behaviour is taken, namely that one should specify only the relative order of externally-visible events, not the detailed machinery which produces the required behaviour. In LOTOS, a distributed, concurrent system is seen as a process, possibly consisting of several sub-processes.

A sub-process is a process in itself, so that in general a LOTOS specification describes a system via a hierarchy of process definitions. A process is an entity able to perform internal, unobservable actions, and to interact with other processes, which form its environment. Thus the use of LOTOS is to describe systems in terms of their capability to interact with their environment.

The mixing language captures and specifies these interactions of the LOTOS specification with its environment and other specifications. For LOTOS, interactions between processes are through units of synchronisation known as events (or atomic interactions, or simply actions) which occur at the interaction point (see figure 2.5).

Events are atomic in the sense that they occur instantaneously, without consuming time. An event is thought of as occurring at an interaction point or gate, and in the case of synchronisation without data exchange, the event name and the gate name coincide. There are three methods of inter-process communication:

*pure synchronisation*, where no values are exchanged

*value establishment*, where one or more processes supply a specific value which is acceptable to the other processes

*value negotiation*, where two or more processes agree on a set of values. Process communication is discussed in more detail later on in this chapter.

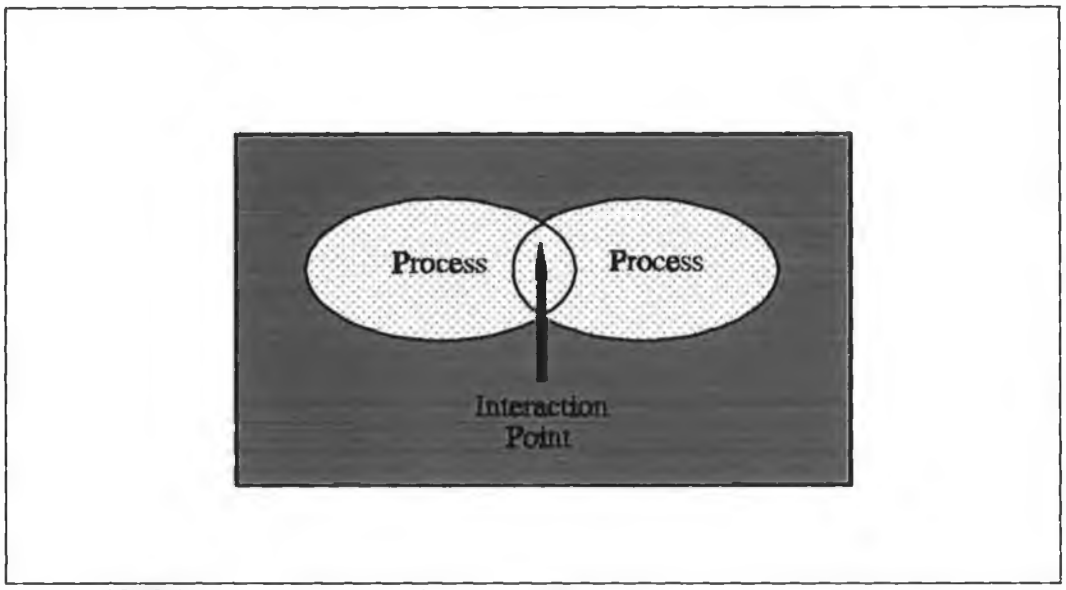

Figure 2.5 LOTOS Process Interaction Point

### 2.3.2.1 How to define a system's behaviour in LOTOS.

This section uses the operators in LOTOS to show how a system can be formally specified. A process is specified in LOTOS, by what is known as a behaviour expression. These behaviour expressions are built out of actions and operators. They define the externally visible behaviour of a process in terms of sequences of events in which it may participate. The following is a list of the more commonly used operators to express the behaviour of a system.

**2.3.2.1.1 Sequencing.** B is the behaviour of a box that first has to synchronise on "a", and later will behave as "B1". The operator ';' is known as the sequential composition operator.

*B ; a ; B1*

It is used to prefix a behaviour expression with an event called an "action prefix". Example : *connect_request; connect_confirm; data; DISCONNECT*

**2.3.2.1.2 Internal actions.** Internal actions or events are represented by 'i', and represent non-determinism since the environment may not influence them. B will behave as B1 after some undefined, but finite, time has gone.

*B := i ; B1*

Example : *WORK [] ( i; GO_TO_BED ).* This indicates that GO_TO_BED may happen without work being an option. ( *'[]'* is the choice operator which is described in 2.3.2.1.7).

**Nondeterministic choice.** Internally, in an uncontrollable manner, a demon will perform one or the other internal events. So, B may behave as B1 or as B2, where the actual events which happen may be influenced by the environment of the process.

*B := i ; B1 [] i ; B2*

**2.3.2.1.3 Parallelism.**

**General Case.** Every action performed by B1 on gate "g" must wait for a matching action of B2 on "g". Also actions performed on any other gate may be freely interleaved. B1 and B2 are synchronised on "g" and only on "g".

*B := B1 |[ g ]| B2*

Example : *(off_hook; dial; answer; speak; on_hook; TELEPHONE)*

*|[ dial ]|*

*(find_number; dial; engage_brain; speak; CALL)*

Synchronise will only happen on the dial event and will allow "speak" in the first behaviour expression before "answer" and after "on_hook" in the second behaviour expression.

**Pure interleaving.** This operator is used to allow behaviours to unfold completely independently in parallel. Given the expression B1 ||| B2, if both B1 and B2 are ready for some action (say actions b1 and b2 respectively), then both action orderings (b1 before b2, b2 before b1) are possible. Notice that b1 and b2 may even be the same. Since B1 ||| B2 transforms, after an action, into an expression still involving the "|||" operator, we conclude that this case of parallel composition expresses nothing but an interleaving of the actions of B1 with the actions of B2.

Example : *( data_in; data_out; BUFFER)*

||| *(read; mark; digest; BOOK)*

The behaviour of this expression could either be :

a) data_in, read, mark, data_out, digest .....

or b) read, mark, digest, data_in .....

**Full synchronisation.** The synchronising parallel composition operator "||" is used where there are events that need to be synchronised. A typical example of the use of this parallel operator is when the capabilities of a process are determined by two or more of its subprocesses. The behaviour of the example below would be bang, start, finish.

Example : *( bang; start; finish; ATHLETE )* || *(bang; start; finish; STARTER )*

### 2.3.2.1.4 Terminating a process.

**Successful termination.** Exit is a process whose purpose is solely that of performing the successful termination action δ, after which it transforms into the process stop.
exit -> δ -> stop.

*B := exit*

Example : *clock_in; clock_out; exit;*

**Inaction.** This is usually used to represent inaction or deadlock.

*B := stop*

Example : *born; live; die; stop.*

**2.3.2.1.5 Process enabling.** B1 enables B2 upon its successful termination. If B1 does not terminate successfully, B2 will not be initiated.

*B := B1 >> B2*

Example : *SHOP process*

*visit_shop; buy_food; come_home; exit*

*EAT process*

*cook_food; eat_food; stop*

*DINE process*

*SHOP >> EAT*

If the left-hand behaviour expression does not terminate successfully, the right-hand behaviour expression will not apply.

**2.3.2.1.6 Process disabling.** Behaviour expression B1 may be disabled by the initial event of B2 at any point of its execution, with the subsequent behaviour that of B2. If B1 terminates successfully, then B2 disappears. Frequently used in specification when there is a need to specify behaviour which may be interrupted by something else.

*B := B1 [> B2*

Example : *(send_data; reset_timer; receive_ack; exit)*

*[> (time_expired; sound_alarm; stop )*

This expression may terminate successfully if an acknowledgement is received to a message, but may sound an alarm if no acknowledgement is received within some time period. Thus the disable operator allows the right-hand behaviour expression to interrupt the left-hand behaviour expression. When it happens, the future behaviour is that of the right-hand behaviour expression.

**2.3.2.1.7 Choice.** In the expression below B is able to perform "a" or "b". If "a", then behaves as B1; if "b", as B2. Decision is external to B. To put it in other words, B offers both "a" and "b"; the environment will choose one of them. If the environment offers both, the decision is nondeterministic. The choice between the alternatives is resolved by the environment of the process.

*B := a ; B1 [] b ; B2*

Example : *(lift_arrive; ENTER) [] (lift_broke; USE_STAIRS)*

**2.3.2.1.8 Guard.** If the condition yields TRUE, B behaves as B1. Otherwise, as "stop".

*B := [ cond ]-> B1*

Example : *[door_open] -> enter;*

**2.3.2.1.9 Hiding.** Allows one to transform some observable actions of a process into unobservable ones. Gate "g" is not known outside B1. That means there is no need to wait for synchronisation from outside B1. In the example below the observable behaviour is begin, end, δ.

*B := hide g in B1*

Example : *hide middle in (begin; middle; exit) |[ middle]| (middle; end;exit)*

### 2.3.2.2 Process Communication

Communication between processes takes place at interaction points called gates. A gate is an abstract entity that is shared by two or more processes. Points of interaction between a system and the environment or a process are known as event gates. The system offers events at these gates. This would be defined as follows:

An event offer is written as

*g!E*

where *g* is the event gate

and *E* stands for an expression defining

the value to be offered.

If two processes (the environment is also seen as a process) offer the same values at the same gate, then the event may occur which in this case results in a synchronisation of the participating processes. The basic element of the control part of a LOTOS specification is the action offered, where a process declares itself ready to synchronise with other processes and establish one or more values. Behaviour expressions may be abstracted in processes that are later instantiated much like conventional procedures. They may be thought of as parameterised processes. The header of a process neatly distinguishes between gate and value parameters.

```
process B[g] (x:integer) : exit :=
g ! E; exit
endproc
when instantiated
B [m] (4)
```

The above expression shows the process B offering the integer value "4" on gate "m".

#### 2.3.2.2.1 Basic LOTOS

This is the subset of LOTOS where processes interact with each other by pure synchronisation, without exchanging values. In basic LOTOS we can appreciate the expressiveness of all the LOTOS process constructors (operators) without being distracted by interprocess value communication.

#### 2.3.2.2.2 Full LOTOS

In full LOTOS, processes may exchange values or be parameterised by them. If a process B1 offers g!v for all values v of a data type t, then this is written as g?x:t . Say we have another process B2 offering the value E of type t, then B1 may synchronise with process B2. This can be interpreted as a value passing from B2 to B1. Values can also be created. A process B3 offering g?y:t would be creating a value y of type t. A non-empty list of event offers is allowed at one gate.

Events may only occur if the event offers of both participating processes 'match'. While in basic LOTOS an observable action coincides with a gate name, in full LOTOS it is formed by a gate name followed by a list of zero or more values offered at that gate: g<v1...vn>.

For example, g!3 states that the process is offering to synchronise on the specific value 3 with other processes, on gate g. g?x:integer states instead that the process is ready to synchronise on any integer value with other processes, on gate g. Thus, two processes containing these two complementary action offers may be able to synchronise, and if they do, the second process gets the value 3 for the variable x.

"Multiple" and "bidirectional" action offers are also possible, such as g!3?x:integer, where the process declares itself ready to simultaneously offer a value and receive one on gate g. Selection predicates can establish conditions for when an action may occur.

```
specification:
        Specification typical_spec [ gate list ] ( parameter list ) : functionality
                type definitions
        behaviour
                behaviour expression
        where
                type definitions
                process definitions
        endspec


process definition:
        process typical_proc [gate list] (parameter list) : functionality :=
                behaviour expression
        where
                type definitions
                process definitions
        endspec
```

Figure 2.6 Typical structures of specification and process definition

### 2.3.2.3 LOTOS Structure Specification.

The structure of a LOTOS specification is shown in figure 2.6. Process and type definitions appear in the "*where*" clause of a specification or process definition, in either order or even interleaved. It clearly appears that a specification and a process definition have similar structure. A minor difference is that the behaviour expression is preceded by the keyword "*behaviour*" in the first case, and by the definition symbol ":=" in the second case. A more significant difference is that some type definitions may appear before the behaviour expression of a specification, whereas this is not allowed in a process definition.

### 2.3.2.4 The use of algebra data type in LOTOS

The LOTOS language is based on process algebras and equational specifications of abstract data types. The most basic form of data type specification in LOTOS consists of a signature and, possibly, a list of equations. Data types are described in terms of their components and the effects of operations on them. They are described abstractly in the sense that implementation concerns (eg. order and length of fields) are deliberately avoided.

ACT ONE allows complex data types to be built out of simple ones, and allows data types to be parameterised. The data typing part of a LOTOS description defines the meaning of values and expressions which appear as parameters in the behaviour part of the description.

### 2.3.2.5 The SPECS Data Types for LOTOS.

The data part of LOTOS is based on the specification language ACT ONE which gives a powerful notation for defining abstract data types [EHRI 85]. Experience shows that rather simple types (eg. records or enumerated types) occur frequently in specifications. Due to the initial algebra semantics, lots of equations have to be written in a schematic way. This is a tedious and error-prone task.

SPECS LOTOS offers the predefined simple types Boolean, Integer, Rational and

Character. The first three of them have the usual meaning. The type Character gives a representation of the ASCII character set by nullary functions (operations without arguments in LOTOS terminology). The simple types can be used in a specification after declaring them in a library statement.(e.g. library Boolean, Integer, endlib ). A number of predefined parameterised data types also exist which take as parameters a number of item data types. The parameterised data types are list, set and map. The data types list and set have their usual functions. The data type map has as parameters a *domain* and *codomain* data type, and the values of a map data type are finite mappings (functions) from domain values to codomain values. A complete list of predefined SPECS data types and operators can be found in [4.17 92].

The data type approach presented by SPECS was driven by the following requirements:

- No writing of equations for "standard" operations like selecting a record component or equality between values of a given sort.

- No change of LOTOS syntax.

- The resulting specification must be statically correct LOTOS. In addition it must be possible to expand a specification to pure LOTOS with the intended semantics.

- Generation of efficient code must be possible.

- Finally, there is a quite specific SPECS requirement: the approach must support the mixing of several specifications written in LOTOS and/or SDL.

SPECS first considered only a restricted set of datatypes that are sufficient for practical purposes. SPECS have standard scalar types (boolean, numbers, characters) and generic types (like sets, arrays etc.) together with the usual operations on them. To every sort declaration SPECS have attached a special comment (called "pragma") to indicate its intuitive meaning e.g. "this is a list of integers". From the point of view of the syntax of the specification language these pragmas are comments, but they have a specific format that make them recognisable by the SPECS translators. All operations are declared together with a pragma, stating e.g. "this operation selects the first element of the list". For example, a LOTOS message primitive is defined and declared in table 2.1.

Table 2.1 LOTOS message primitive example

```
type Primitive is Address, Module
sorts Primitive (*$ rec $*)
opns CON_RQ (*$ mkrec 3 $*): Address,Address, Module -> Primitive
        calling_of (*$ sel 1 $*)
        called_of  (*$ sel 2 $*) : Primitive -> Address
        module_of  (*$ sel 3 $*) : Primitive -> Module
...
endtype
```

The pragma **rec** defines the message primitive as a record. In the pragma **mkrec**, the constant 3 denotes the number of components the record Primitive has. Note that the pragmas can handle the case where two or more components belong to the same sort. In table 2.1 two selector operations have the same signature but they are distinguished by the integers in their respective pragmas. The compiler interprets the meaning of these equations via the pragmas [KARN,91].

SPECS proposal considerably shortens the LOTOS declaration of widely used data types. It supports both rewrite techniques and generation of efficient code, since SPECS aims at the automatic code generation from such specifications. Moreover, there is no change to LOTOS syntax and static semantics rules. The number of axioms to be written by the user is reduced significantly. To support the development of the few remaining user-defined equations, methodological guidelines are given. These guidelines assume only minimum ADT knowledge. The formal semantics of the approach are well-defined by giving a mapping to standard LOTOS. The same mapping can be used to transform a SPECS LOTOS specification in a way such that standard LOTOS tools can be applied to it.

```
Specification Large_of_[in1, in2, in3, out] : noexit

type integer is
          sorts int
          opns
                    zero : -> int
                    succ : int -> int
                    largest : int, int -> int
          eqns forall X,Y : int ofsort int
                    largest (zero, X) = X;
                    largest (X, zero) = X;
                    largest ( succ(X), succ(Y)) = succ( largest(X,Y) );
          endtype

behaviour
          hide centre in
          (
          Large_of_ [in1, in2, centre]
          |[centre]|
          Large_of_ [centre,in3,out]
          )

where
          process Large_of_ [val1,val2,max] : noexit :=
                    (
                     val1?X:int; exit (X, any int)
                    |||
                     val2?Y:int; exit (any int, Y)
                    )
                    >>
                    accept V: int, W, int in
                              max!largest(V,W); stop
          endproc
endspec
```
Figure 2.7 LOTOS specification of a system which accepts three natural numbers and stops after printing the largest of them.

Also the algebraic data types in SDL and LOTOS are in general difficult to handle by semantic tools. In order for SPECS semantic tools to be able to handle data, the predefined data types are used, which means that both animation and code generation can be done automatically. SPECS LOTOS tools directly interpret the SPECS data types, for example when animating a specification or when generating C code.

### 2.3.2.6 LOTOS Example

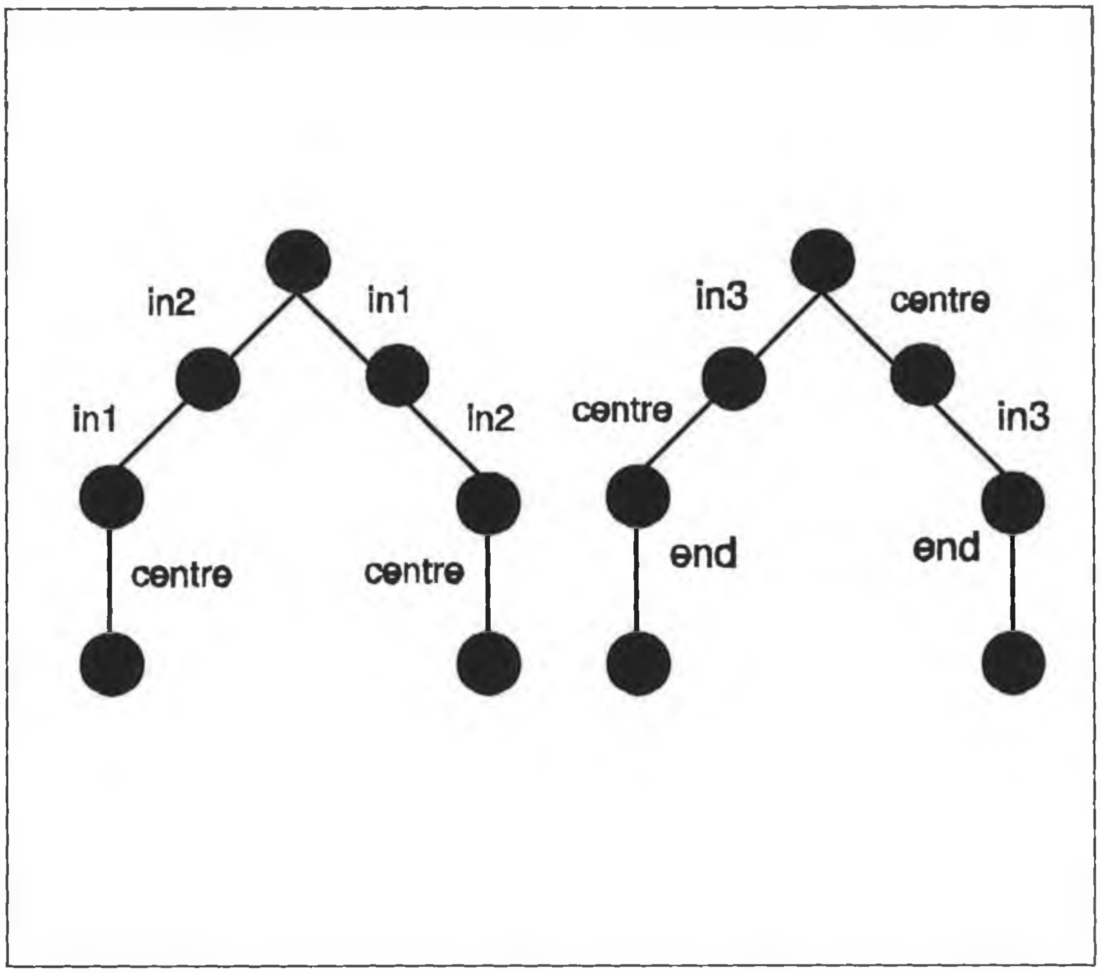

Figure 2.8 Action trees

The temporal ordering of actions can be shown by an action tree. The following specification is modelled by an action tree to show the possible sequence of events and highlight the use of the operators explained earlier. The example is a simple LOTOS specification for an entity which is able to accept three natural numbers in any order and stops after printing the largest of them (See Figure 2.7). The action tree of each process in shown in Figure 2.8. The action tree of the specification is shown in Figure 2.9.

The behaviours part of the specification describes the top structure of the specification, which consists of two instantiations of process *Large_of_*. This process finds the largest of two values read in on the gates *val1* and *val2* and gives the result on gate *larger*. As the two copies of *Large_of_* are instantiated, their gates are relabelled respectively as: *in1, in2, in3, end.* Thus the output value (largest of two natural

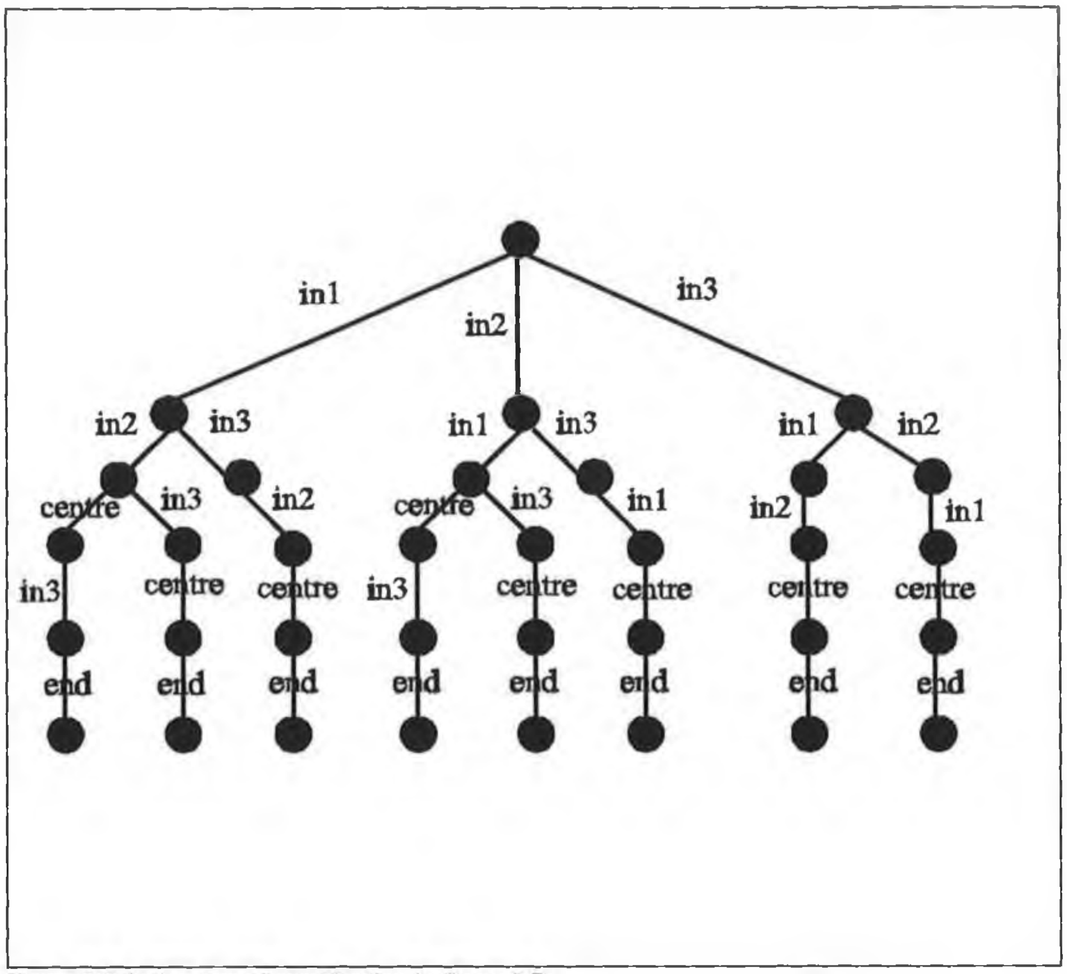

Figure 2.9 Action tree of specification in figure 2.7

numbers) is passed from one instantiation to the other via gate *centre.*

### 2.3.3 LOTOS in Practice.

A methodology for using LOTOS specifications in the implementation phase is still a subject of research. SPECS have produced such a methodology which successfully produces C code for a LOTOS specification.

Specifications of real-life systems of thousands of lines have been written in LOTOS. Some of these are on their way towards becoming part of ISO International Standards. For example: several OSI layers (Network, Transport, Session), specifications of telephone systems [FACI 88], etc. (in addition of course to all the best known "text-book" examples such as the Alternating Bit Protocol, the Dining Philosopher's problem, etc.). Several such examples are included in [VVDE 89]. The language is starting to be used in industrial environments, especially in the UK where British Telecom and Hewlett-Packard have substantial LOTOS groups.

## 2.4 Conclusion (LOTOS vs SDL).

LOTOS has synchronous communication ports and SDL has asynchronous communication ports. Synchronous communication is characterised by the fact that the communicating partners are involved in an interaction at the same time. An example of this is shaking hands. Asynchronous communication can be described by introducing a context which hides a system from its environment. The system only communicates indirectly via its context with its environment. An example is writing a letter: the moments of writing and reading are independent; the postal service constitutes the context.

The nature of SDL signals and LOTOS events are quite different since an event carries only a list of data values whereas signals have additional knowledge about their receiver and sender addresses. In synchronous communication there is no real notion of send and receive, only of interaction. In asynchronous communication the partners perceive the occurrence of communication at different moments in time. The one that starts the communication is called the sender, the other the receiver.

If a LOTOS event occurs then this "signals" to the specification that the environment has participated in that event and the environment also "knows" that the specification has accepted that event. In SDL the situation is quite different. There are no direct means for the environment to detect whether a signal has been "accepted" by SDL or not. It is even not defined what "accepted" means in terms of SDL. It could mean that the signal has been removed from the input queue of the receiver or that the processing triggered by the signal is in a certain state. This is also the same for signals from the SDL specification to its environment. It is the very nature of asychronous communication. These problems are discussed in chapter 3 in mixing SDL and LOTOS.

The data type approach for these two languages differs substantially from the usual approach in programming languages, where a user has to define a new data type by means of already defined data types, ultimately by means of the predefined data types of the language.The data type concept in SDL and LOTOS is based on the abstract data types (ADT), and is conceptually based on the specification language ACT ONE [KARN 91]. ACT ONE is an algebraic specification method and it can be used for unparameterised as well as parameterised ADT specifications. Structure and properties of the system's data structures are describes using this method [SMIT 89]. This gives a powerful notation for defining abstract data types. Although the notations used in SDL and LOTOS differ for historical reasons, the equivalence of the concept is underlined by the fact that ISO and CCITT were able to agree on a common document for both [BELI 91].

Unlike SDL, LOTOS descriptions are generally more abstract, hence there is usually a bigger step in producing implementations. The other major difference from SDL is that LOTOS has a fully formal definition : there can be no dispute over the meaning of a LOTOS description, and rigorous analysis is possible. On the debit side, LOTOS requires deeper study and more discipline if good descriptions are to be written, but the results simply repay the time spent.

LOTOS emerges favourably over SDL and other similar languages such as Estelle. The detailed differences between the two languages were shown to lead to significant weaknesses in SDL descriptions [VISS, 86]. SDL lies somewhere between a specification language and an implementation language, and does not quite manage to be either. Although claiming to be a specification language, SDL is really a high-level implementation language.

The main strength of the language LOTOS is its precision of expression, its implementation-independence, and its ability to analyse systems which makes it more suitable for definitive descriptions of OSI standards. SDL has other important roles to play. In particular, it has a useful part to play in giving guidance on implementation issues and in providing reference implementations. LOTOS is really a general-purpose language for formally specifying concurrent and distributed systems. LOTOS may thus be used in a wider context than SDL.

## 2.5 Summary

In this chapter we looked at the two formal specification languages SDL and LOTOS, and compared their differences. Although LOTOS appears to be most favourable language, SDL does have its advantages in being more reliable and of course best suited to specifying the behaviour of asynchronous systems.

Chapter 3 looks at the reasons for mixing formal specification languages and presents the InterConnection Language which enables these languages to be mixed. Other approaches to combining languages are also examined.

# Chapter 3 A Description of the InterConnection Language (ICL)

## 3.1 Introduction

This chapter mainly concentrates on the mixing of SDL and LOTOS using the mixing language ICL. The Interconnection Language (ICL) is used within SPECS to describe the combination of complete specifications written in LOTOS and/or SDL. This chapter brings forward advantages to mixing specifications, examines the InterConnection Langauge and gives a detailed explanation of the mapping rules of the language. Chapter 4 gives a step by step technique to produce the ICL mixing specification using the specifications produced from the CR&F methodology. This technique is known as the "*ICL Production Rules*" and is applied to the SPECS pilot case study in chapter 5.

This chapter also looks at another type of mixing. An approach to mixing specification languages adopted by [WILE 90] is presented. He explains that the mixing of different languages can be placed into two categories which are mentioned in this chapter. A model given to illustrate one of these categories, the "Specification Level Interoperability", - which [WILE 90] defined to be the ability of the components to communicate and work together despite their differing paradigms.

## 3.2 Why mix specification languages?

The need for the interaction of different languages, despite their background arises in many contexts. Generally, the desire to combine programs written in different languages springs from the availability of specific capabilities in some particular language, processor or existing program. Thus, for example, the number-crunching power of a vector processor and the availability of a particular numerical analysis routine in Fortran might entice a programmer to attempt to have a LISP program running on a workstation interacting with Fortran code running on the Vector processor. The reasons for mixing formal specification languages are :

- **Re-use existing specifications :** Allow system designers to extend the behaviour of a specified system by combining new specifications with previously existing specifications.

- **Flexibility :** Specifications written by a set of people with different backgrounds can be combined. Mixing allows the re-use of existing specifications even if they are written in a language different from the one used for the new specification.

- **Better specification of a system :** One formal language may have better facilities to represent a certain aspect of the behaviour of a given system than another. Depending on the problem at hand, it may be the case that some of its subproblems are easily expressed in LOTOS, whereas SDL is better suited for other parts. These "modules" are then combined into one single system which solves the initial problem.

- **For test cases :** One language can be used to specify test cases for specifications written in the other language.

## 3.3 Overview of Two Alternative Mixing Approaches

### 3.3.1 Introduction

This section presents two approaches to mixing languages. The first approach describes the work carried out in the Esprit Project 2565 ATMOSPHERE on method integration of LOTOS and SDL through service definitions. The second approach describes different levels that systems or subsystems can communicate, despite having been written in different languages.

### 3.3.2 LOTOS-SDL Integration via Service Definitions

This section presents a description of the 'Service Definitions' used in Esprit Project

2565 ATMOSPHERE for LOTOS-SDL integration and gives an example showing how service definitions can be used in the specification of an alarm activation service.

ATMOSPHERE's LOTOS-SDL integration used the behaviour expressions of LOTOS to specify signalling interfaces in SDL descriptions. These interface descriptions are called 'Service Definitions'. This was achieved by adjusting the LOTOS notation to meet the requirements of SDL interfaces. The internal modules of the overall structure of the SDL system are viewed as black boxes. Conceptually these internal modules would be SDL blocks. For any other process to communicate with these 'black boxes', the only knowledge required is 'how' the communication occurs, i.e. how to behave according to the service definition.

The service definition language is similar to the syntax of LOTOS and describes the flow of inputs and outputs of each module. The following concepts comprise the service definition language *(their informal semantics are given in italics)* [KRON 93]:

Action prefix **;** - *sequential operator*
Value declaration **g!v** - *sending of signals*
Variable declaration **g?x:type** - *receiving of signals*
Choice **[]** - *conditionality*
Interleaving **|||** - *arbitrary mutual temporal ordering of events belonging to different event sequences*
Enabling **>>** - *sequentiality, dependent on successful termination preceding sequence of events*
Disabling **[>** - *means to interrupt the 'ordinary' sequence of events*
**stop** - *the non-event*
**exit** - *signifies successful termination*

These are then translated into SDL state machines. This is done by hand as there are many aspects of design that have to be elaborated at the SDL level. In writing the SDL state machines for each module a semantic interpretation can be given to the interconnections of each module.

An example of a service definition is given below using an alarm service activation process. This service accepts from the user a valid address and time for which an alarm is required. The user is informed of the success or failure of the operation. The service definition of the alarm activation would be defined as follows:

```
alarm_activation := ?address ?alarm_time;
                    (( !ack; exit [] !nack; stop )
                         >> ( alarm_set [] alarm_not_set )) [> break


alarm_set := !success; exit
alarm_not_set := !failure; exit
break := !system_break; exit [] ?user_break; exit
```

ATMOSPHERE developed tools to verify and validate the SDL state machines against their service definitions. The service definition approach provides a powerful conceptual tool for the early phases of the SDL based design process and gives the analyst a chance to apply the method in the later stages, such as testing and maintenance.

### 3.3.3 Mixing at different levels

The mixing of different languages can be categorised into two levels. These two levels are

- Representation Level Interoperability

- Specification Level Interoperability

Interoperability means the ability of two or more systems or subsystems to communicate or work together despite having been written in different languages. A central issue in supporting interoperability is achieving "type correspondence" so that entities, such as data objects or procedures used in one system can be shared by another program that may be written in a different language or running on a different kind of processor. While most approaches to mixing provide support at the representation level, my research concentrates on the specification level of interoperability. Sections 3.3.3.1 and 3.3.3.2 describes each of these two levels.

### 3.3.3.1 Representation Level Interoperability (RLI)

This level is concerned with how the representation of simple types, such as integers, floating-point numbers or characters, differ in different languages or on different processors and focuses on ways to map between those different representations. It provides a means of overcoming differences in the ways that different machines or programming languages implement simple types. Most approaches to supporting interoperability have been based on establishing correspondence of data types at the representation level.

The earliest form of this approach involved interoperation through ASCII representations of data, where the data communicated between the "interoperating" program via files. This required the programs themselves to translate the data either into or out of the ASCII representation.

The UNIX operating system supports interoperability via pipes (untyped byte streams) through which two interoperating programs can communicate. The byte stream can encode any type of data. So, as long as the interoperating programs agree on how to interpret the bytes, they can share data of any type. Again, this requires the programs at either end of the pipe to translate the shared data from its actual type into the byte stream representation and back again.

### 3.3.3.2 Specification Level Interoperability (SLI)

Specification level interoperability extends representation level interoperability by hiding the differences on data types. For example, where RLI[20] would hide the byte orders of array elements used to represent a stack object, SLI[21] would hide the fact that the stack was represented as an array. Thus the representation of the stack as an array or as a linked list or both is made irrelevant to the different programs sharing the stack.

By increasing the degree of information hiding, the extent to which interoperating programs depend on low-level details of each other's data representations is reduced. SLI allows much greater flexibility in implementation approaches and thus more opportunities for optimisation. SLI also increases the range of languages and types that can participate within an interoperating system.

### 3.3.4 Interoperability Mixing Model

#### 3.3.4.1 Introduction

To achieve the specification level interoperability, [WILE 90] developed a model to support mixing systems that allow data objects or procedures to be shared. This model consists of four components:

- Unified Type Model
- Language Bindings
- Underlying Implementations
- Automatic Assistance

A diagram of this model is shown in figure 3.1.

[20] RLI Representation Level Interoperability

[21] SLI Specification Level Interoperability

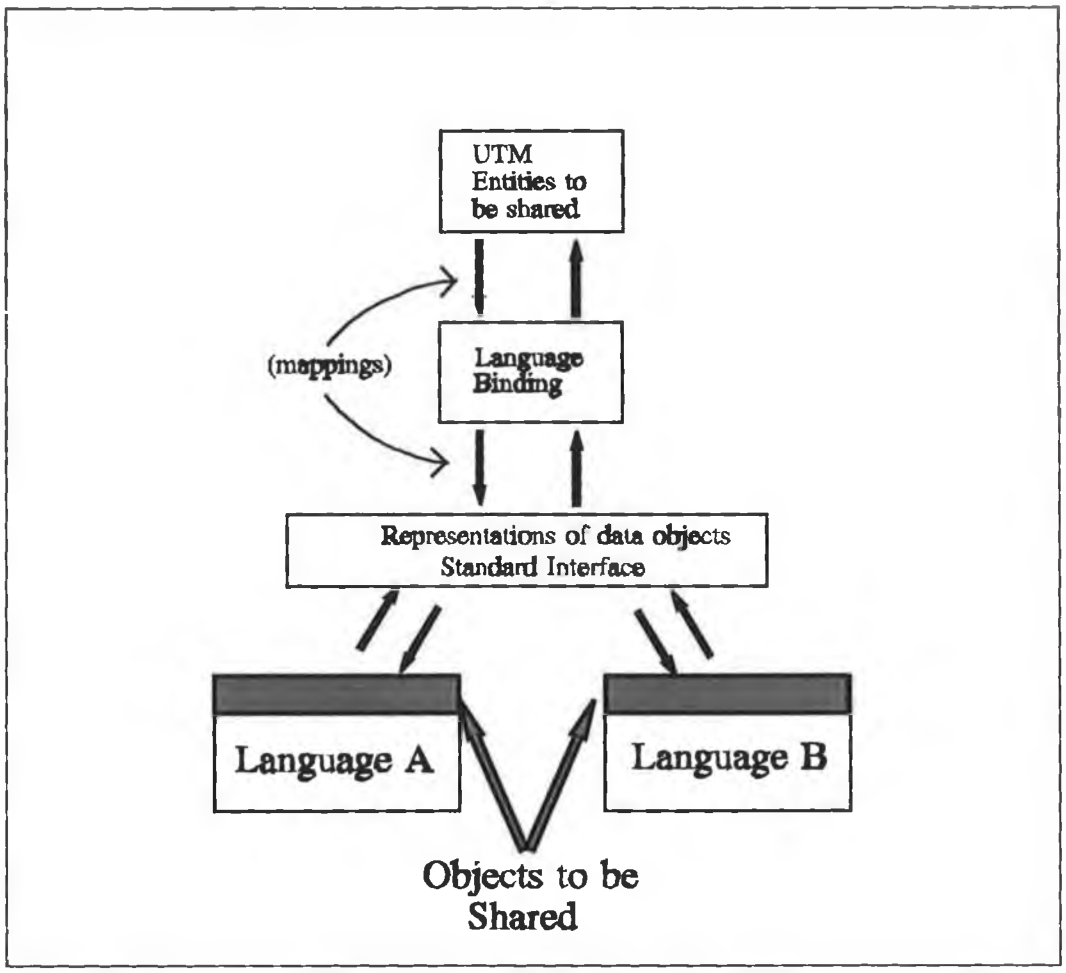

Figure 3.1 Interoperability model for supporting mixed systems [WILE 90]

### 3.3.4.2 Description of the model

- **A unified type model (UTM)**, which is a notation for describing the entities to be shared by interoperating programs (different programs that will communicate with each other). UTM type definitions supplement but do not replace the type definitions for the shared entities that are expressed in the language(s) in which the interoperating programs are written. A UTM should be capable of expressing high-level, abstract descriptions of the properties of a broad range of types, but need not adhere too closely to the syntax or type definition style of any particular programming language.

- **language bindings**, which connect the type models of the languages to the

Unified Type Model. Given a Unified Type Model and a particular programming language, there must be a way to relate the relevant parts of a type definition as given in the language to a definition as given in the Unified Type Model. Each such mapping between a UTM and a particular language is referred to as a *language binding*. Not all aspects of a UTM must be mappable to a given language, but only those that are relevant to the programs in that language. A set of different bindings could be defined for a given language, each providing mappings for only those UTM aspects relevant to a particular interoperating program written in that language.

- **underlying implementations**, which realise the types used by the different interoperating programs. The combination of a UTM type definition and a language binding induces an interface through which an interoperating program written in that language can manipulate instances of that entity type. Underneath the interface will be one or more representations for data objects and code to implement procedures (i.e., operations) that the interface provides for manipulating the data objects.

- **automated assistance**, which eases the task of combining components into an interoperable whole. The creation of a UTM definition would be greatly increased through automatic support by providing a library of pre-existing UTM type definitions, language bindings and underlying implementations, plus a browser for exploring that library. An automatic generation tool would also be valuable. Such a tool would, for example, take a UTM type definition, plus specification for the desired language binding and underlying implementation and generate the corresponding interface.

## 3.4 Mixing LOTOS and SDL Specifications

This section describes an overview of the functionality of the ICL mixing language. The main aim of this mixing is to transform the LOTOS events into SDL signals and vice versa which makes it possible that one specification becomes part of the environment of the other. It is necessary that the specifications to be mixed follow the data type conventions defined by SPECS. In order to allow mixing, the external interfaces of the specifications to be mixed must be designed with the same conceptual ideas in mind. So, when re-using existing specifications it is possible to take an existing specification and "connect" new ones to it. The new ones must adopt the interface concepts of the old ones in the sense that there must be a suitable correspondence between gates and/or signals. When combining specifications which have been built independently it is likely that some changes will be necessary to achieve this. Specifications developed via the SPECS methodology (CR&F Approach) allow less room for such changes.

There must be a clear correspondence between certain events and signals. In case of LOTOS/LOTOS and SDL/SDL mixing, this is the correspondence of the list of data carried by connected signals or gates (the signal/gate names can be different). For the mixing of different specification languages (LOTOS/SDL mixing) a more subtle approach is applied, where only some values have to match, whereas other values are used to carry specific routing information. The connections between the various specifications can be defined by a special purpose Interconnection Language (ICL). At the final stage of the formal specification design, the LOTOS component, the SDL component and the ICL component can be jointly compiled into C.

## 3.5 The Interconnection Language ICL

ICL allows us to describe the transformation in a concise way in allowing single statements to cover a large number of transformations by means of "wildcard-like i.e. (*)" statements. It was originally designed with the idea of allowing the mixing between SDL and LOTOS specifications. Therefore the central parts of the ICL are the statements (also called rules) that allow expression of this mixing. Additionally there are constructs for specifying the system interface, for instantiating subsystems and for LOTOS/LOTOS and SDL/SDL communication.

When mixing different specifications one specification becomes part of the environment of the other. Since both languages expect from the environment a behaviour in terms of their own language, some intermediate translation is needed. An SDL environment sends signals to the SDL system and receives signals from the SDL system. We will denote SDL signals with name *s.* valuelist *vl.* via channel *ch.* from sender *send.* to receiver *rec.* as <*s.vl.ch.send.rec.*>. A LOTOS environment synchronises with the LOTOS system over certain events. LOTOS events are noted over gate *g.* with valuelist *vl.* as <*g.vl*>. [SSH 92].

### 3.5.1 ICL Syntax Rules

#### 3.5.1.1 Transformation mappings from SDL to LOTOS

When describing mixed specifications it is necessary to give a definition of mappings between SDL signals and LOTOS events. One problem when defining such mappings is the difference in information contents of events and signals. LOTOS events are more abstract in the sense that they do not contain sender and receiver information ( events have no direction at all ). Furthermore there is no channel concept in LOTOS. When studying LOTOS specifications it can clearly be observed that LOTOS gates sometimes directly express "signals" and sometimes are used like "channels" on which several signals are exchanged. In the second case a further qualification is needed (by the means of data values ) to identify the signal. This technique is called subgating.

In SDL process instances are identified by so called PId ( process instance identifiers ) values that are automatically available. In practice they are often used to maintain the communication between specific processes.

From the notation above, the transformation form is <*s,vl,ch,send,rec*> -> <*g,vl'*>. We assume that *vl* is some representation that is the same for LOTOS and SDL. *vl'* will always be augmented with at most two values which are added at the beginning and/or end of the list. The problem is tackled by having a number of predefined data types like integer, record, set. etc (For SPECS data types see chapter 2 section 2.3.2.5). They are equivalent for SDL and LOTOS and can be mapped to the same representation on CRL level.

Signals from the SDL channels contain more information than a LOTOS event, so it is necessary sometimes to decode that additional information into a valuelist. According to [SARI 91], mapping rules are given to generate mappings for a large class of signals and still to allow a distinguished treatment that takes the semantic differences into account. These generic mappings are given in table 3.1. On this table fixed names are denoted by quoted strings i.e. *"subgate"*. Unquoted variable names i.e. *vl* are used to indicate that they are copied from the signal to the event. To indicate that the value does not matter a *"*"* is used. This means that the transformation can be applied for arbitrary values of the specific field.

The idea behind those rules is to generate transformations for a large class of signals and still to allow a distinguished treatment that takes the semantics difference into account. The rules of Table 3.1 handle a large number of signal to event transformations. In fact each rule adds several elements into the relation *Signal-to-Event*, where Event is the set of all LOTOS events and Signal is the set of all SDL signals. If the value of a signal *"s"* consists of one boolean value the rule 1.2 adds:

< "s",<true>,*,*,"LPid"> -> <"g","subgate"^<true>>

< "s",<false>,*,*,"LPid"> -> <"g","subgate"^<false>>

#### 3.5.1.2 Transformation mappings from LOTOS to SDL

Table 3.2 below shows the generic mapping rules for LOTOS to SDL. The individual rules of table 3.1 share similar interpretations to the ones in table 3.2 which are explained in the next section.

### 3.5.2 Explanation of the Rules

> **Rule (1.1)** means : If a signal with name *"s"* comes from the SDL system on any channel from any sender and it is directed to LOTOS (receiver PId is

Table 3.1 Generic Mapping Rules for SDL to LOTOS

| | | | |
|---|---|---|---|
| (1.1) | < "s".vl.* .* ."LPid" > | - | < "g".vl > |
| (1.2) | < "s".vl.* .* ."LPid" > | - | < "g"."subgate"^vl > |
| (1.3) | < "s".vl.* .pidS."Lpid" > | - | < "g".vl^pidL > |
| (1.4) | < "s".vl.* .pidS."Lpid" > | - | < "g"."subgate"^vl^pidL > |
| (1.5) | < "s".vl."c".* ."Lpid" > | - | < "g".vl > |
| (1.6) | < "s".vl."c".* ."Lpid" > | - | < "g"."subgate"^vl > |
| (1.7) | < "s".vl."c".pidS."Lpid" > | - | < "g".vl^pidL > |
| (1.8) | < "s".vl."c".pidS."Lpid" > | - | < "g"."subgate"^vl^pidL > |

Table 3.2 Generic Mapping Rules from events to signals

| | | | |
|---|---|---|---|
| (2.1) | < "g".vl > | - | < "s".vl.* ."LPid".* > |
| (2.2) | < "g"."subgate"^vl > | - | < "s".vl.* ."LPid".* > |
| (2.3) | < "g".vl^pidL > | - | < "s".vl.* ."LPid".pidS > |
| (2.4) | < "g"."subgate"^vl^pidL > | - | < "s".vl.* ."LPid".pidS > |
| (2.5) | < "g".vl > | - | < "s".vl."c"."LPid".* > |
| (2.6) | < "g"."subgate"^vl > | - | < "s".vl."c"."LPid".* > |
| (2.7) | < "g".vl^pidL > | - | < "s".vl."c"."LPid".pidS > |
| (2.8) | < "g"."subgate"^vl^pidL > | - | < "s".vl."c"."LPid".pidS > |

*"LPid"*) it may be mapped to a LOTOS event over gate *"g"* with valuelist *vl*. Note that the simple copying of value lists is only possible because we have assumed that LOTOS and SDL values have been mapped to the same representation.

**Rule (1.2)** means : In this case, signal *"s"* is transformed to an event over *"g"* where the value list starts with the specific value *"subgate"*.

**Rule (1.3)** means : The sender Pid is appended to the valuelist of the generated LOTOS event. We use the operator "^" to denote that *pidS* refers to the SDL representation of the Pid value and *pidL* refers to the equivalent LOTOS representation. There must be a suitable correspondence between values of the two different representations. Since SDL *pidS* are isomorphic to the natural numbers it is rather natural to use only similar sorts for *pidL*.

**Rule (1.4)** means : A combination of rule (1.2) and (1.3).

**Rule (1.5)** only signals *"s"* arriving on channel *"c"* are mapped. The fact that the signal has arrived on a specific channel is mapped to the LOTOS value *"lval"* and prefixed to the value list of the generated LOTOS event.

**Rule (1.7-8)** Similar to rules (1.2-1.4) with the exception that only signals *"s"* arriving on channel *"c"* are transformed.

This is the foundation of the SPECS mixing scheme which is a mapping between LOTOS events and SDL signals and vice versa. These rules are not very intuitive for the inexperienced analyst. Thus I produced another set of rules, the *ICL Production Rules*, which aid the analyst in producing the ICL specification. ( Chapter 4 section 4.3).

Table 3.3 shows the essential ICL syntax to express SDL/LOTOS mixing that corresponds to the generic mappings defined in tables 3.1 and 3.2. Brackets ("[]") denote optional constructs and terminal symbols (keywords) are quoted.

Within this section we will clarify the semantics of our LOTOS and SDL mixing scheme by making use of an example (Figure 3.3). In this simple example, a signal *sig(comm,91)* could be mapped to an event *sig!comm!91*. This example takes a piece

Table 3.3 The essential ICL syntax

| | |
|---|---|
| *<ICL>* | *:= { < ICLstatement > }* |
| *<ICLstatement>* | *:= <signal> '->' <action>* |
| | *\| <action> '->' <signal>* |
| *<signal>* | *:= 'SIGNAL' <signalname> '(*)'['VIA'<channelname>]* |
| *<action>* | *:= 'ACTION' <gatename> ['!'<value>]'!*'['!PID']* |
| *<signalname>* | *:= SDL signal identifier* |
| *<gatename>* | *:= LOTOS gate identifier* |
| *<channelname>* | *:= SDL channel identifier* |
| *<value>* | *:= value being exchanged* |

of LOTOS that is at any time able to perform one event of the form *g!0.g!1.g!2...* (in our notation *<"g"."0">...*) or an event *h!91*. The LOTOS operator *"[]"* denotes a choice, the *";"* denotes sequence and *"g?n:nat"* expresses the ability to perform any *"g"*-action with a natural number value.

The SDL part is a one-state process that waits for the reception of a signal *"in"* that carries a data value of type natural number (in our notation *<"in".n.***>* ) and that responds by sending a signal *"out"* with value *"17"* to LOTOS. We resolve the indeterminism at the LOTOS side (*"g?n:nat"*) by receiving a value from SDL and the *"INPUT"* statement of SDL will be fed with the value of the *"h"*-action. This is shown in Figure 3.3.

When the system starts, both the queue between SDL and LOTOS (QSDL-LOTOS) and the channels of the SDL systems (*Cin-Cout*) are empty. Since SDL waits for the reception of *in(n)* it can do nothing. So it is LOTOS's task to take the initiative.

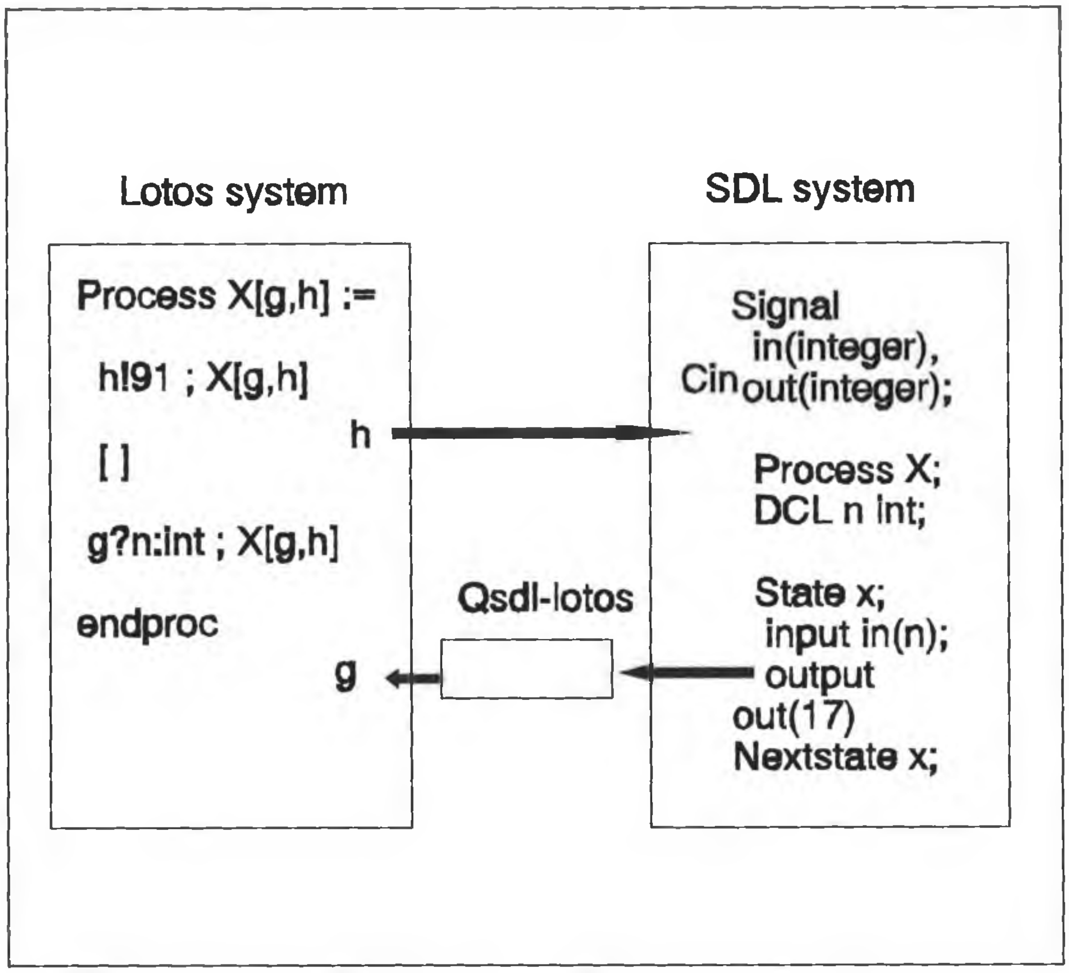

Figure 3.3 Interconnection Example
ICL : action h!* -> signal in(*) /*1*/
signal out(*) -> action g!* /*2*/

Event *h!91* will take place and will be mapped to signal *in(91)* and sent to SDL. After reception of this signal, *out(17)* will be sent, mapped to *g!17* and put in to QSDL-LOTOS. Now the LOTOS action *"g?n:nat"* may synchronise with *g!17* and the event is removed from QSDL-LOTOS. LOTOS has also the possibility to perform another *h* event to be mapped to an SDL signal *"in"*.

Rule (1) states that every event l.h!n will be transformed to a signal sdl.in(n) and sent to SDL. Rule (2) states that every sdl.out(n) signal sent from SDL to its environment will be transformed to a l.g!n event and put into the queue QSDL->LOTOS. LOTOS then synchronises with that queue in such a way that it is always possible to "consume" the first element of the queue.

## 3.6 How does the mixing work

- The ICL language allows one to describe the transformation in a concise way. Single statements are able to cover a large number of transformations by means of "wildcard-like" statements. In general an infinite number of transformations are needed. This essentially is caused by the parameterisation of gates and signals. Furthermore it has been taken into account that signals and events differ in their information content. Events carry only a list of data values whereas signals have additional knowledge about their receiver and sender addresses.

- The gap between synchronous LOTOS and asynchronous SDL communication has been closed by queuing signals from SDL to LOTOS.

- Pure LOTOS/LOTOS mixing is described by giving a list of gates from different LOTOS subsystems that have to synchronise. This allows the description of the same combinations that can be represented using LOTOS parallel operators (|||, ||, |[..]|) and hide.

- Pure SDL/SDL mixing is realised by allowing that signals are exchanged (and renamed) between SDL subsystems.

- In order to distinguish between gates or signals from different systems, their names are qualified with the system name. The names of all systems involved in a mixed system have to be different.

- Direct transformation of a LOTOS gate name to a SDL signal name where data are copied as they are. Copying of data is possible because we allow only certain data types( like e.g. integers, records, sets,..) which are equivalent in LOTOS and SDL and which are mapped to the same lower level representation. These are the so-called SPECS data types.

- Transformation of a LOTOS gate name to an SDL channel name, where one element of the LOTOS value list is used to identify a specific signal. As an example we can transform event *in_port!on_hook!1* to signal *on_hook(1)* on channel *c_in*, or signal *off_hook(1)* on channel *c_out* to event *off_port!off_hook!1*. This is expressed by the rules in table 3.4. (The star indicates an arbitrary list of data items. )

- Usage of one element of the value list of a LOTOS event as PId value for SDL. This makes it possible that a LOTOS process remembers the sender process of a signal and can direct further communication to the same process.

Table 3.4 Example of subgating.

| | | |
|---|---|---|
| *ACTION* | | *SIGNAL* |
| *in_port!on_hook!** | -> | *on_hook(*) VIA c_in* |
| *SIGNAL* | | *ACTION* |
| *off_hook(*) VIA c_out* | -> | *out_port!off_hook!** |

## 3.7 Mixing Results

Mixed specifications are translated into the internal Common Representation Language (A-CRL, I-CRL) and therefore all the CRL methods and tools offered by SPECS are available, for instance code generation and simulation.

The compilation to C code and necessary runtime environment sub-routines deliver a routine with the interfaces described in the ICL. This routine can be embedded into the target application. It offers the following interfaces:

- Synchronisation events to be shared between the mixed system and its environment. Environment refers to the threads that communicate with the mixed system via RTE primitives.

- Signals sent to specific environment threads.

- Reception of signals from the environment.

In order to use the generated code, the routine representing the mixed system has to be started by a piece of C code to be provided by the system developer. It is also necessary to provide some environment code that interacts with the system and performs the embedding into the application. For example the environment code may display some message if a specific signal is received or it may offer an event on user request.

## 3.8 Summary of the ICL language

ICL is designed to express the connection between LOTOS and SDL systems. An ICL specification can link an arbitrary number of these specifications to form one system. This chapter presents the syntax of the ICL language and the generic mapping rules for correct syntax. Chapter four presents another set of rules, the "ICL Production Rules" which guide the specifier to easily create ICL specifications using the outputs generated by the CR&F Methodology.

# Chapter 4 Mixing SDL and LOTOS using the ICL Production Rules

## 4.1 Introduction

SPECS has defined a methodology (CR&F methodology) for producing the SDL and LOTOS specification but no steps or guidelines were given for producing the ICL specification. From working with the DCU team on the CR&F methodology, the production of the formal specification LOTOS is found to be efficient and quite automatic. Using the specifications produced from each process, I defined a set of rules to also automate the construction of the ICL specification. This chapter looks at this set of rules, the *ICL Production Rules* for generating the ICL specification, which adhere to the ICL generic mapping rules in chapter 3 tables 3.1 and 3.2. The ICL Production Rules use the outputs of the CR&F methodology. An overview of the CR&F methodology is given below. These rules aid in the production of a concise and correct ICL specification.

## 4.2 Products of the CR&F Methodology.

The methodology is composed of three processes:

- **Classification**, for getting a first understanding of the input specification by identifying entities in the informal specification and re-expressing it in terms of concepts in the application domain.

- **Rigorisation**, for increasing such understanding through analysis and building of different views using different paradigms.

- **Formalisation**, for actually producing the formal (SDL/LOTOS) specification using all the knowledge acquired in the previous processes.

The processes, collectively called the **CR&F-process**, aim is to establish an increasing understanding of the application domain, of the input specification and of the requirements of the system under development, and to express this understanding as formal specifications.

The methodology was named after the above processes names, as **CR&F methodology**. The ICL Production Rules use the products of these processes to produce the ICL statements. These are the:

- *CR&F Classified Specification.* An example is shown is Appendix B.
- *CR&F Rigorous Specification.* An example is shown in Appendix C.
- *CR&F Formal Specification.* (Consisting of 2 or more systems expressed in SDL and/or LOTOS). An example is shown in Appendix D for SDL and Appendix E for LOTOS.

## 4.3 ICL Production Rules

This section presents a set of rules on using the products of the CR&F methodology to produce the ICL specification. Each rule consists of an explicit statement of the rule followed by a justification. A detailed knowledge and understanding of the ICL mixing language is not necessary.

### 4.3.1 Using the Classified Specification

**Rule 1:** From each of the classified components, the *interface aspect* gives the complete set of communications for that component. It details the external interface, indication and response, and the internal interface. Interface aspects are defined in terms of inputs and outputs. List all entities of the interface aspect for each component.

The Classification process is a process of identifying and defining application concepts, then re-expressing the informal specification in terms of these concepts.

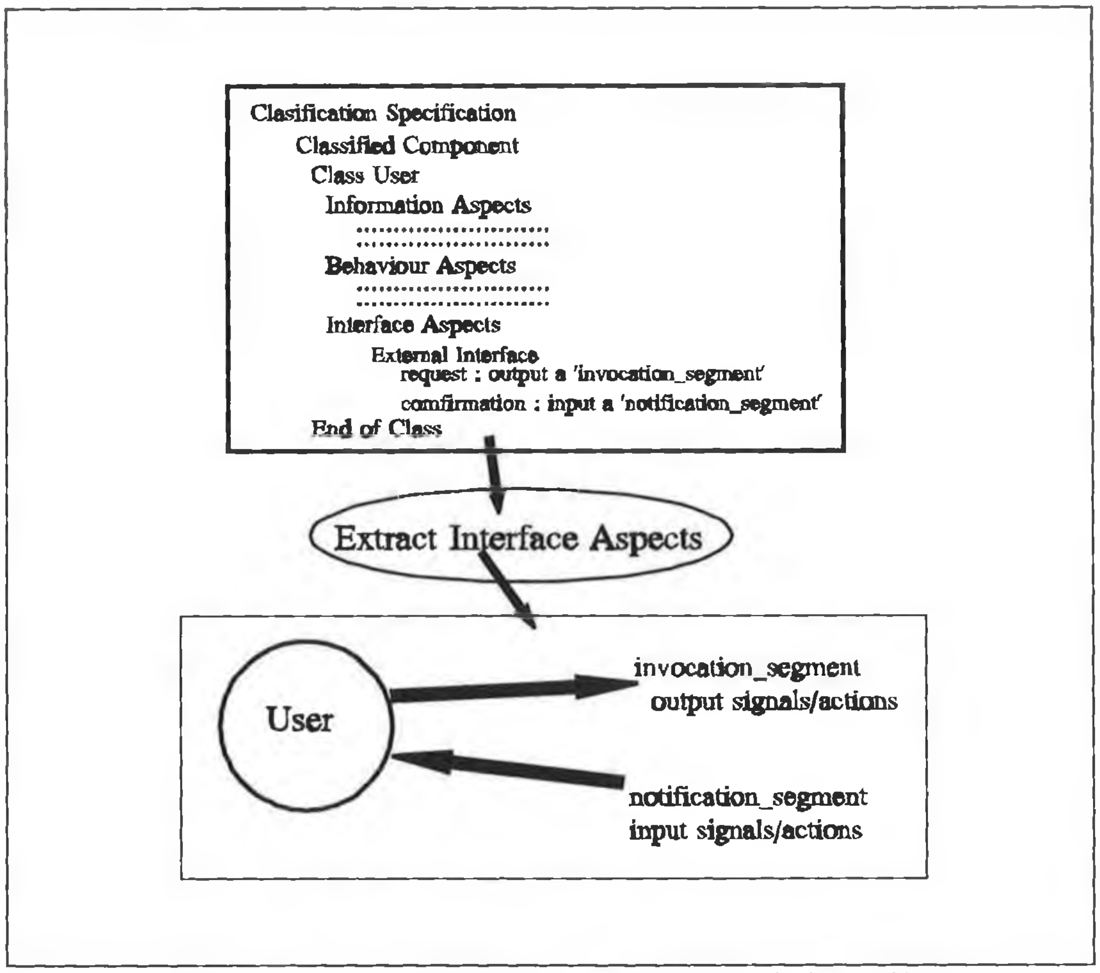

Figure 4.1: Interface Aspects of the Classified Specification gives signals/actions a object

A Classified component consists of three aspects :

- information aspect
- behaviour aspect
- interface aspect

The **information aspect** describes how the properties of the objects of the class are represented. The **behaviour aspect** describes the sequence of actions performed by the objects of the class. The **interface aspect** describes the object's ability to cooperate with other components in its environment [BELS 92]. This aspect is the main source of information for the ICL specification (see figure 4.1).

The classification process will record information about the target system and the development process in a set of *classified-components* and *development-components*. An application concept is described as a class contained in the body of a *classified-component*. The class is described by its behaviour, information, interface and intrinsic miscellaneous aspects. A section of a classified specification based on a pilot case study is given in *Appendix B*.

Defining communications between the subsystem and its environment for the generation of ICL statements is a very specific activity, thus only one aspect of the classified specification is necessary, i.e. the interface aspect. Using the conceptual structure of the classified specification and the interface aspects, an overall understanding of the needs and the requirements of the target system is attained.

### 4.3.2 Using the Rigorous Specification

The rigorous specification gives a good understanding of the functionality of the system. It is possible to structure the statements for each of the two systems from well detailed rigorous models. At this point the decision as to which section of the system is to be represented by which specific tower language should have been made. The steps below show how to interpret each of the rigorous models to obtain the information necessary for the production of the ICL statements.

#### 4.3.2.1 Using The DFD's

> **Rule 2:** The data flows between the process in the context diagram are to be specified in the header of the ICL specification. The data flows between the subsystems in Level 0 DFD constitute the ICL mapping statements

From the context diagram the system's communications with its environment are defined. The external communication, that the system has with its environment, is easily identifiable from this rigorous model. These data flows are to be specified in the header of the ICL specification, as they represent the systems external communication.

The subsystems which represent the parts of the system to be defined in the different tower languages are identified. The data flows between these subsystems, as noted in the rigorous model's Level 0 DFD, represents the main communications between the two tower languages which will be mapped to ICL statements. This gives only an overview of the communication between the tower languages and between the system and its environment. Each data flow arrow may have more than one value being passed but each data flow must be represented by at least one ICL statement.

Any new inter-process communication from this stage on, will be hidden, as they are not externally visible. Level 0 DFD identifies the main processes in the system. The behaviour of these processes may be specified either by LOTOS or SDL.

The system is partitioned into a number of processes which synchronise according to a given architecture. These subprocesses are expanded in DFDs at lower levels.

#### 4.3.2.2 Using the Message Sequence Charts (MSC)

> **Rule 3:** Communication with external processes on an MSC may be defined by one ICL statement. Disregard communications between processes which are internal to a particular subsystem.

The Message Sequence Charts (MSC) give a graphical and concise description of the messages being passed. This rigorous model is vital for understanding the relationship between each subsystem as it clearly defines the reason for each message. The same communication may be present in two or more MSCs as a different sequence of events may cause it to occur.

#### 4.3.2.3 Using ASN.1 and PSpecs

> **Rule 4:** Use the ASN.1 descriptions of the data structures and types to identify

the exact contexts of each information flow. An enumerated type used in a communication flow to another processes may denote more than 1 ICL statement.

ASN.1 gives a detailed description of the data structure of the messages being passed. The Process Specifications (PSpecs), describe the behaviour of each process. Using these two models, the specifier can accurately state the data type of information (i.e. integer/boolean) being passed and the reason why, before looking at the formal specification. The ASN.1 describes the exact contents of each flow entering and leaving the system.

### 4.3.3 Using the Formal Specification

**Rule 5:** Read carefully each of the formal specifications noting in each specification the data being received/sent externally. Ensure that the data type declaration for the value(s) being passed is consistent in both specifications (i.e. same data type is used). From the information obtained in Rule 3 from the MSC, identify the SDL and LOTOS statements where points of external communication between specifications occur.

Due to the limitations of ICL it is important that the formal specifications are of a consistent style. Direct transformation of a LOTOS gate name to an SDL signal name is possible by the use of certain data types (e.g. integers, records, sets, etc.) which are equivalent in LOTOS and SDL and which are mapped to the same lower representation. These are the SPECS data types. With ICL the handling of data values is successful when the formal specifications are restricted to using SPECS data types.

Transformation of a LOTOS gate name to an SDL signal name, where one element of the LOTOS value list is used to identify a specific signal, is known as subgating. It is only allowed with enumerated types. In ICL, it is not possible to map booleans that are being passed between two systems. The solution is to introduce an enumerated type for these data values in both specifications.

Table 4.1 Syntax template for LOTOS to/from SDL mappings

**6.a From a LOTOS event to a SDL signal:**

```
action lotos-subsystem-ID . gate-name(s)!*
->
signal sdl-subsystem-ID . signal-name(*)
        via sdl-subsystem-ID . channel-name;
```

**6.b From a SDL signal to a LOTOS event:**

```
signal sdl-subsystem-ID . signal-name(*)
        via sdl-subsystem-ID . channel-name;
->
action lotos-subsystem-ID . gate-name(s)!*
```

Table 4.2 Syntax template for mappings to the border of the system.

**6.c Communication flow from SDL to system border:**

```
signal sdl-subsystem-ID . signal-name(*)
->
signal icl-system-ID . signal-name(*);
```

**6.d Communication flow from system border to SDL:**

```
signal icl-system-ID . signal-name(*)
->
signal sdl-subsystem-ID . signal-name(*);
```

**6.e Communication flow to/from LOTOS event with system border:**

```
action icl-system-ID . gate-name,
        lotos-subsystem-ID . gate-name!*;
```

## 4.3.4 Constructing the ICL statements

**Rule 6:** Construct the ICL statements individually, using the syntax template in table 4.1 and 4.2. Using the following system identifier names.

| | |
|---|---|
| *lotos-subsystem-ID* | *= system identifier for LOTOS spec.,* |
| *sdl-subsystem-ID* | *= system identifier for SDL spec.,* |
| *icl-system-ID* | *= system identifier for ICL spec.* |

From the rigorous specification, a clear correspondence between events and signals would have been established from Rule 5. These correspondences can be defined using the InterConnection Language (ICL). From analysing the general construct of the language, I have developed five types of statements which would support all mappings. These syntax templates for constructing ICL statements are shown in table 4.1 and table 4.2.

For an SDL signal connected directly to the border of the system, the direction of the signal is important. In table 4.2 the flow of communication is specified for SDL mappings.

#### 4.3.4.1 ICL Specification Header

> **Rule 7:** Construct the ICL Specification Header, from the syntax template in table 4.3, using the systems external communication flows identified in Rule 1 and Rule 2.

The header of an ICL specification indicates which variables are accessible to the environment of the complete system. The header is used to produce the signature files which are necessary for testing the interface of the final system. Each ICL specification has a header which consists of a formal parameter list. This contains a list of LOTOS gates communicating with the environment of the system. For SDL it contains in/out signals and in/out channels for communication at the border of the system.

The external-gate-names is a list of gates where the LOTOS specification communicates with its environment. The inchannel and outchannel refer to those channels on which the SDL specification would communicate with its environment. The insignal and outsignal refer to the signals on these channels.

Table 4.3 ICL specification header syntax header

**7.a ICL Specification Header Template**

```
icl-system-ID [ external-gate-names]
{
inchannel SDL-channel;
outchannel SDL-channel;
insignal signal-name;
outsignal signal-name;
}
using
lotos-subsystem-ID, SDL-subsystem-ID
where

/* then follow the actual ICL statements */

end system
```

## 4.4 Conclusion and Implementation of the Rules

Using the products of the CR&F methodology enables the specifier to obtain an overall understanding of the system. The *ICL Production Rules* allows a software engineer to produce an accurate ICL statement without a detailed knowledge of the language being necessary. Chapter 5 shows how these mixing rules are used to produce the ICL specification for the SPECS Pilot Case Study.

The ASN.1, PSpecs and DFD can be used to ensure that all the data, or communication flows, referred to in these techniques have been identified in the ICL specification.

## 4.5 Summary

The rules presented above are easily followed, self explanatory and easily applied. Although these rules depend on the CR&F methodology being followed to produce the formal specifications, the rules produce a reliable ICL mixing specification and leave little room for error. Insight is gained on the system behaviour from the classified, rigorous specifications thus aiding the specification production process.

In the next chapter the methodology described in the previous section is applied to an example in the SPECS Pilot Case study. The informal Specification of the problem, as defined in [ARISE, 92] is contained in Appendix A.

# Chapter 5 Application of the ICL Production Rules to the SPECS Pilot Case Study

## 5.1 Introduction

In this chapter the rules for generating an ICL specification, described in the previous chapter, are applied to a case study. The case study was based on a service provided by the national Portuguese ISDN standard for the user of supplementary services. Due to the vast amounts of informational documents generated by the CR&F processes in this case study, the information has been condensed in this chapter. The complete documents are given in the appendices. The complete informal Specification of the problem can be found in *Appendix A*. The system is classified into concepts in *Appendix B*. Its behaviour is modelled in *Appendix C* using software development models i.e. Data flow diagrams, Message Sequence Charts. The Pilot Case Study's behaviour is formally specified in *Appendix C* (SDL specification) and in *Appendix D* (LOTOS specification). These formal specifications were produced from the CR&F Methodology. The mixing specification which is the result of the *"ICL Production Rules"* is given in *Appendix E*.

## 5.2 Overview of the SPECS Pilot Case Study.

The informal specification was provided by CET (Centro de Estudos de Telecomunicacose), a Portugese member of the ARISE project. It is a section of the CET's ELDIS project. ELDIS is an ISDN rural telephone exchange used widely in Portugal. ELDIS utilises distributed computing, a relational database and high level design.

The case study concerns the provision of supplementary services to the users of an ELDIS exchange. At present ELDIS users have access to eight such services :

- Closed User Group
- Call Forwarding

- Advice of Charge
- Malicious Call Identification
- Outgoing Calls Barring Service
- Incoming Calls Barring Service
- All Calls Barred Service
- Alarm Call Service

Due to pressure of time and manpower, *SPECS-Specification Generation* was unable to develop all of these functions. It was decided to concentrate on the Alarm Call Service as it does not rely on other services for its performance. Also it is not necessary that all of the calls be modelled to successfully test the SPECS methodology and ICL Production Rules.

In Figure 5.1, a block diagram of the Alarm Call service shows the two main processes, the USER and the Call Control Coordinator, and their interactions with each other. The figure shows that the Call Control Coordinator contains two sub processes that interact internally to the system.

## 5.3 Description of the Alarm Call Service

The following description of the Alarm Call Service is taken from [ARIS 92]:

### 5.3.1 Introduction

The Alarm Call Service is activated by inputs from the subscriber. This service provides four functions. These functions are invoked by the user sending a specific service code via a keypad to the Call Control Coordinator.

### 5.3.2 Functionality of the Alarm Call Service.

#### 5.3.2.1 Activation

For activation of Alarm Call service, the user invokes the call control coordinator

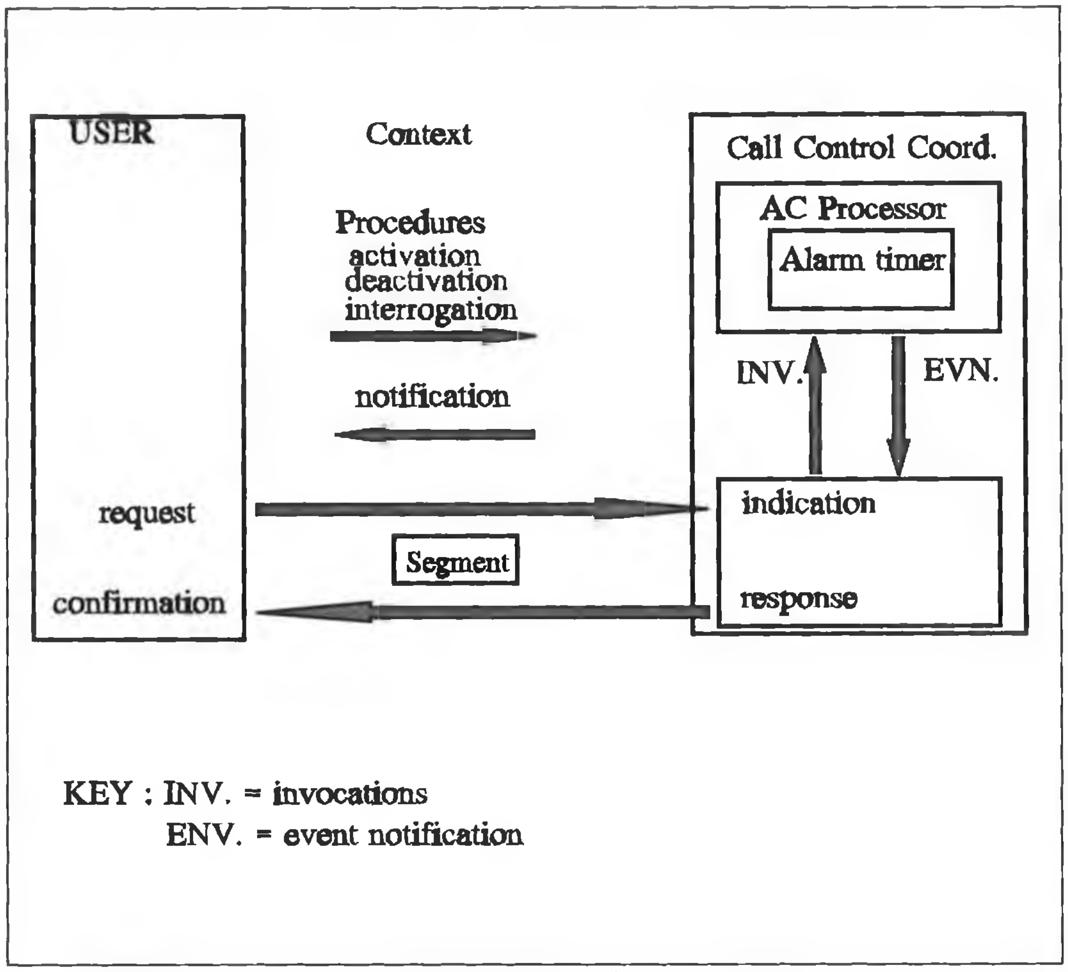

Figure 5.1: Abstract "realistic" view of the considered problem context

(CCC) with a request of an alarm time. The (CCC) checks the users validity and responds to the request. At the appropriate time an alarm is activated and the message is sent to the user.

#### 5.3.2.3 Deactivation

The user sends a message to the CCC requesting the deactivation of his/her alarm call.

#### 5.3.2.4 Interrogation

The user may check if his requested service has been established or if the correct alarm time has been received by the CCC.

#### 5.3.2.5 Notification

A list of the responses from the CCC. These responses inform the user on the status of his/her request.

Notification on Activation
Notification on Deactivation
Notification on Interrogation

A complete description of the informal specification, giving the service code for each function, can be found in Appendix A.

## 5.4 Alarm Timer Process

Although it was not specifically required, it was decided to design a process which initiated alarm calls. This was deemed to be necessary as otherwise comprehensive testing and demonstration of the case study would be impossible. This extra process was called the *Alarm-Timer*. It receives the current time and checks to see which users require a call at this time.

The rest of this chapter shows the production of the ICL statements necessary for the pilot case study, using the ICL Production Rules in Chapter 4.

## 5.5 Generating the ICL specification.

### 5.5.1 Using the Classification Specification

The initial input to the classification process was the informal specification provided by CET in Appendix A. The following classified components constitute the classified specification of the problem considered. (The Full classified specification of the pilot case study is given in *Appendix B*.)

- Context Interface Aspects
- User Interface Aspects
- Call Control Coordinator Interface Aspects
- Analyser Interface Aspects
- AC Processor Interface Aspects
- Alarm Timer Interface Aspects.

From the interface aspects of these components, we receive an understanding of their interaction with other components and their environment. Using **Rule 1**, the following is an extract from the classification specification explaining each of their interface aspects. The Interface Aspects of these Components are as follows;

#### 5.5.1.1 The Context Interface Aspects

*interface aspects*

*internal interface*

*'user' 'request' interfaces to call_control_coordinator' 'indication';*

*'call_control_coordinator' 'response' interfaces to 'user' 'confirmation'.*

These communications are shown in Fig A.1 of appendix A. Thus we have two external communication flows. The user "requests" to the system (indication) and receives a "response" (confirmation). The request is known to the system as "invocation-segment and the confirmation as "notification segment" as shown in the **User's** interface aspects.

#### 5.5.1.2 The User

*interface aspects*

*external interface*

*request: output a 'invocation_segment';*

*confirmation: input a 'notification_segment'.*

These interface aspects are the same as the 'Context Interface Aspects'.

#### 5.5.1.3 The Call Control Coordinator

*interface aspects*

*external interface*

*indication: input a 'invocation_segment';*

*response: output a 'notification_segment'. /* from/to User*/*

*internal interface*

*'analyser' 'AC_activation' interfaces to 'ac_processor' 'activation';*

*'analyser' 'AC_deactivation' interfaces to 'ac_processor' 'deactivation';*

*'analyser' 'AC_interrogation' interfaces to 'ac_processor' 'interrogation';*

*'ac_processor' 'notification' interfaces to 'analyser' 'AC_notification';*

*'ac_processor' 'alarm' interfaces to 'call_control_coordinator' 'alarm'.*

These seven communications are shown in Figure C.2 of appendix C. Again we have the same two external communication flows and four new internal communication flows. The alarm interface would be the alarm time from the alarm timer process. This would be an ICL statement from the call control coordinator to the border of the system. The four internal communications are expanded below in sections 5.5.1.4 and 5.5.1.5.

#### 5.5.1.4 The Analyser

*interface aspects*

*external interface*

*indication: input a 'Segment';*

*response: output a 'Segment'; /* information for the user*/*

*AC_activation: output invoking activation of AC for a 'User', with "address", "hour" and "minute" extracted from the 'Segment' received as 'indication';*

*AC_deactivation: output invoking deactivation of AC for a 'User' with "address" extracted from the 'Segment' received as 'indication';*

*AC_interrogation: output invoking interrogation of AC for a 'User' with "address" extracted from the 'Segment' received as 'indication';*

*AC_notification: input notification information about an AC_invocation.*
*/* four information flows from/to the AC-Processor */*

### 5.5.1.5 The AC Processor

*interface aspects*

*external interface*

*activation: input the "hour", "minute" and 'User' "address" for AC invocation;*

*deactivation: input the 'User' "address" for the supplementary service deactivation;*
*interrogation: input the 'User' "address" for inquiring from the network about the AC service;*

*notification: output notification information about an AC invocation;*
*alarm_timeout: input a 'User' "address";*
*alarm: output a 'User' "address".*

### 5.5.1.6 The Alarm Timer

The alarm timer process checks the time value of all the user's records in the data base against the current time and sets off an alarm if necessary. It receives the current time from a database called "Clock" and checks each user's entry in the Alarm-Database. If the time in the user's record matches with the time stored in the "Clock" database, then the address of that user is sent out on the ALARM flow to the Alarm

Handler process. See Figure C.3 in Appendix. Its interface aspects are given below.

*interface aspects*

*external interface*

*set_alarm: input an Hour and a Minute;*

*turn_off: input;*

*ask_on_off: input;*

*current_on_off: output a Boolean;*

*ask_setting: input;*

*current_setting: output an Hour and a Minute;*

*alarm_timeout: output a 'User' 'address'.*

### 5.5.2 Using the Rigorous Specification

#### 5.5.2.1 Using the DFDs.

**Communication with the Environment:** Using **Rule 2**, the Context Diagram shown in *Appendix C Figure C.1*, shows two data flows between the environment,the USER, and the system, the CALL CONTROL COORDINATOR. These communications will be represented by at least 2 ICL statements.

The CALL CONTROL COORDINATOR consists of two independent systems, the AC-PROCESSOR and the ANALYSER. The AC-PROCESSOR was specified in LOTOS and the ANALYSER was specified in SDL. The data flows between these subsystems, as noted in the rigorous model Level 0 DFD *Appendix C Figure C.2*, represent the main communications between the two tower languages which can be mapped to ICL statements. This shows the minimal number of ICL statements necessary. In this case there are seven data flows; *Invocation Segment , Notification Segment, AC-Activation, AC-Deactivation, AC-Interrogation, AC-Notification, Alarm.*

#### 5.5.2.2 Using the MSC

From the Message Sequence Charts, identify the external process and take these communication lines only. Disregard the process communication between internal processes. The same communication may be present in two MSC charts, as a different sequence of events may cause it to occur.

Following **Rule 3** for each MCS gives is a list of the exact ICL statements.

| **No. of ICL statements** | **Description** | **Reference** |
|---|---|---|
| 2 | From user to analyser. | Appx.C Fig C.14 |
| 2 | Alarm Activation + notification success | Appx.C Fig C.4 |
| 1 | Alarm Activation + notification failure | Appx.C Fig C.5 |
| 2 | Alarm Deactivation + notification success | Appx.C Fig C.6 |
| 1 | Alarm Deactivation + notification failure | Appx.C Fig C.7 & C.8 |
| 2 | Alarm Interrogation + notification success | Appx.C Fig C.9 |
| 1 | Alarm Interrogation + notification failure | Appx.C Fig C.10 & C.11 |
| 1 | Alarm from Alarm Timer to Alarm Handler. | Appx.C Fig C.12 |
| 1 | Time from Clock to Alarm Timer | Appx.C Fig C.13 |

(13 ICL statements)

### 5.5.2.3 Using ASN.1+PSpec

Enumerated type denotes more than 1 ICL statement as stated in **Rule 4**. This allows the use of subgating as shown in chapter 3 table 3.4. AC_Notification is enumerated from:

| | |
|---|---|
| successful_activation | - facilidade Aceite |
| unsuccessful_activation | - erro de activacao |
| successful_deactivation | - facilidade desactivada |
| unsuccessful_deactivation | - error de desactivacao |
| interrogation_result | - desperatar <hour:time> |
| unsuccessful_interrogation | - error na consulta |

These messages will be passed from the AC-processor to the Analyser. They can be passed as strings but since strings are difficult to manage in LOTOS, subgating is used instead by using the gate name and the message being passed.

### 5.5.3 Using the Formal Specification

As the formal specifications were developed using the SPEC's data types and tools, they would have the correct requirements for mixing using ICL. From **Rule 5**, the actual lines from the formal specification where the external communication is sent/received are given below. The SDL specification is given in *Appendix D* and the LOTOS specification in *Appendix E*.

**For Mapping SIGNALS to EVENTS**

a) From *OUTPUT AC_ACTIVATION ( address, time) ;* to

*AC_ACTIVATION ? address : address_type ? time : time_type;*

b) From *OUTPUT AC_DEACTIVATE ( address );* to

*AC_DEACTIVATION ? address : address_type;*

c) From *OUTPUT AC_INTERROGATE ( address );* to

*AC_INTERROGATION ! ERRO_DE_DESACTIVACAO ! address;*

**For Mapping EVENTS to SIGNALS**

d) From *AC_NOTIFICATION ! FACILIDADE_ACEITE ! address;*
to *INPUT AC_ACTIVATE_SUCCESS ( address ) ;*

e) From *AC_NOTIFICATION ! ERRO_DE_ACTIVACAO ! address;*
to *INPUT AC_ACTIVATE_UNSUCCESS ( address );*

f) From *AC_NOTIFICATION ! FACILIDADE_DESACTIVADA ! address;*
to *INPUT AC_DEACTIVATE_SUCCESS ( address );*

g) From *AC_NOTIFICATION ! ERRO_DE_DESACTIVACAO ! address;*
to *INPUT AC_DEACTIVATE_UNSUCCESS ( address );*

h) From *AC_NOTIFICATION ! DESPERTAR ! time ! address;*
to *INPUT AC_INTERROGATE_SUCCESS ( time , address );*

i) From *AC_NOTIFICATION ! ERRO_NA_CONSULTA ! address;*
to *INPUT AC_INTERROGATE_UNSUCCESS (address );*

**For Mapping SIGNALS to the border of the system.**

j) From *OUTPUT NOTIFY ( address, <service message>, time );* to the environment.

k) From the environment to

*INPUT INVOKE ( address, service_code, invocation, time );*

signal-name = *AC_DEACTIVATE_SUCCESS*

channel-name = *ANALYSER_AC_PROCESSOR*

this gives the following ICL statement;

*action AC_PROCESSOR . AC_NOTIFICATION ! FACILIDADE_DESACTIVADA !* -> signal ANALYSER . AC_DEACTIVATE_SUCCESS (*) via ANALYSER . ANALYSER_AC_PROCESSOR;*

**Syntax template 6.b From an SDL signal to a LOTOS event:**

For signal *OUTPUT AC_ACTIVATION ( address, time) ;* to be mapped to event *AC_ACTIVATION ? address : address_type ? time : time_type;*

Using syntax template 6.b in table 4.1 chapter 4:

```
    signal sdl-subsystem-ID . signal-name(*)
            via sdl-subsystem-ID . gate-name(s)
    ->
    action lotos-subsystem-ID . gate-names(s) ! * ;
```

where;

lotos-subsystem-ID = *AC_PROCESSOR*

sdl-subsystem-ID = *ANALYSER*

gate-name(s) = *AC_ACTIVATION*

signal-name = *ACTIVATION*

channel-name = *ANALYSER_AC_PROCESSOR*

this gives the following ICL statement;

```
    signal ANALYSER . AC_ACTIVATE (*)
            via ANALYSER . ANALYSER_AC_PROCESSOR
    ->
    action AC_PROCESSOR . AC_Activation !* ;
```

**Syntax template 6.c For Signals from System Border to SDL**

From the environment to a SDL process *INPUT invoke(address,service_code, invocation, time);*. Using the syntax template 6.c in table 4.2 chapter 4, where the new parameter is;

icl-system-ID = *CALL_CONTROL_COORDINATOR*

gives the following ICL statement;

```
signal CALL_CONTROL_COORDINATOR.INVOKE (*)
        via  CALL_CONTROL_COORDINATOR.USER_ANALYSER
->
signal ANALYSER.INVOKE (*) via  ANALYSER.USER_ANALYSER;
```

**Syntax template 6.d For Signals from SDL to System Border.**

From the SDL process OUTPUT notify() to the environment using syntax template 6.d in table 4.2 Chapter 4 gives:

```
signal ANALYSER.NOTIFY (*) via ANALYSER.USER_ANALYSER
->
signal  CALL_CONTROL_COORDINATOR.NOTIFY (*)
        via CALL_CONTROL_COORDINATOR. USER_ANALYSER;
```

**Syntax template 6.e For Events to/from the Border of the System.**

From the environment, the event *ALARM* occurs, which would synchronise with the event *ALARM* of the *AC_PROCESSOR* specification. Using syntax template 6.e in table 4.2 of chapter 4 would give the following ICL statement:

*action CALL_CONTROL_COORDINATOR . ALARM, AC_PROCESSOR . ALARM !**
In both these cases the border of the System is identified by the icl-system-ID.

### 5.5.4.1 ICL Specification Header

The header of an ICL specification indicates which variables are accessible to the environment of the complete system. The header is used to produce the signature files which are necessary for testing the interface of the final system.

Using the header template 7.a from **Rule 7**, we construct the ICL specification header as follows: From the LOTOS specification we have two external communication flows. The *ALARM* from the process *Alarm Timer* (internal process of the system) to the *Alarm Handler*(part of the environment) and the *Time* from the *Clock*(part of the environment) to the *Alarm Timer*. These external-gate-names must be included in the header. Also on the SDL side, the external interfacing parameters were two signals, *NOTIFY* and *INVOKE*. These will also be part of the header. Using the header template 7.a in table 4.3 in chapter 4, the ICL specification header is defined.

```
system    call_control_coordinator
 [   ALARM, TIME, ADD_USERS  ]
{
inchannel user_analyser ;
outchannel user_analyser;
outsignal notify ;
insignal invoke
}
using
/*  System Identifier for the LOTOS  */
AC_PROCESSOR,
/*  System Identifier for the SDL     */
ANALYSER
where
```

```
/* Actual ICL statements */

endsystem
```

## 5.6 Conclusion

This chapter demonstrated the application of the "ICL Production Rules" to the SPECS pilot case study. These rules have proven to be efficient and accurate in producing an ICL specification for systems developed using the SPECS CR&F methodology.

# Chapter 6 Conclusions

## 6.1 Introduction

This research was carried out as part of DCU's contribution to the SPECS project, part of the RACE program of the EC. This thesis addresses the problem of combining specifications written in different FDTs. To my knowledge, the most practical approach for mixing formal specification languages is the SPECS method. SPECS has developed a mixing language and support tools to implement the combination of LOTOS and SDL specifications. The rules SPECS devised towards producing the ICL specification were quite "ad-hoc" with no adequate structure. Thus as part of my research and work with SPECS, I defined a set of rules to automate the generation of the ICL specification.

The aim of the SPECS project was to automate the software development process as much as possible. Deriving a methodology for producing formal languages was an intermediate goal towards achieving this aim. Code generation tools, developed by the SPECS project, take the formal specifications as input and automatically translate them into a suite of programs, in the programming language C, thus reducing the time that normally would be required to develop software. Chapter 1 gives a description of my role in SPECS and describes the SPECS project work structure, integration of work packages and architecture to achieve its aim.

The two formal languages on which SPECS concentrated were SDL and LOTOS. These are described in detail in chapter 2 and a comparison of these two languages is given.

Other approaches to mixing languages are described in chapter 3, including the SPECS approach. A description of how the mixing works between SDL and LOTOS specifications via an ICL specification is presented. Also a brief description of the CR&F methodology is given, where the outputs of each process are used to produce the ICL specification. A set of rules, referred to as the "*ICL Production Rules*", are given in chapter 4. These rules aim to automate the production of the ICL Specification. A worked example, the SPECS Pilot Case Study, is presented in chapter 5, to illustrate the use of the *ICL Production Rules*. The results of the Pilot Case Study are given in this chapter.

The results of my work were incorporated in the deliverables produced by DCU and in the book by SPECS "*SPECS - THE BOOK:Synopsis*" [SPECS 93]. The two main deliverables are *"CR&F Specification of the SPECS Pilot Case Study"* [I.WP3.9 92] which describes the pilot case study undertaken SPECS in its final year and *"Final Methods and Tools for the Generation of Specifications"* [DWP3.8 92] which describes in detail the work done by workpackage 3 in SPECS final year.

## 6.2 Results of the Pilot Case Study

The set of rules presented in chapter 4 are a revised version of those which are documented in the SPECS project deliverables. The products of the pilot case study are shown in the appendices. INESC produced an SDL specification and DCU produced the LOTOS specification and the ICL specification which linked these two specifications together, creating a system called "Call Control Coordinator". The Pilot Case Study provided an excellent environment to test the "*ICL Production Rules*". One aspect of the rules was the speed in producing the ICL specification once the SDL and LOTOS specifications were completed and tested separately. In comparison to the other formal languages, it took less than a third of the total formalisation time.

The results of the Pilot Case Study showed advantages in using formal specifications for developing software. When the final executable code was being intensively tested, a number of errors and bugs were uncovered. These were small aspects of the system which were overlooked by the system specifier. These errors were easily traced back to the relevant parts of the SDL and LOTOS specifications, and amended. These amendments filtered through to the ICL specification. In comparison to amending C programs, this was not a difficult task as formal specifications are concise and it is easier to detect errors in them. When the formal specifications were corrected, new versions of the executable code were produced within minutes by the SPECS tools. This demonstrated the ease with which amendments can be made to C-code produced using the SPECS methodology.

## 6.3 Strengths and Weaknesses of the ICL Production Rules.

### 6.3.1 Advantages to using the Rules.

The "*ICL Production Rules*" defined in chapter 4 provide the system specifier with an insight into, and understanding of the system being developed. These rules are intuitive and easy to follow. The specifier does not have to work through listings of formal languages to extract the required information between the specifications to be mixed. A range of different rigorous modules is used to capture as much information about the intended system as possible. Thus the technique for developing the ICL specification is semi-automated. The rules would appear to have been designed for inexperienced system developers but they efficiently produce a correct mixing specification.

### 6.3.2 Limitations of the Rules.

The system specifier has the extra burden of familiarising him/herself with the specifications produced by the CR&F methodology. Since the ICL language is restricted to mixing the specifications produced by this methodology it would be good advice to follow the methodology, not only because of the ICL limitations but also, because it aids the production of correct formal specifications.

A number of tools have been developed for the production, static semantics checking and translating of the ICL specification. These tools are only applicable to SPECS and may not be generally suitable for other systems.

The extent to which the rules were tested was restricted. Although the rules were used in another case study performed in SPECS, RACEBANK, the only major application was the Pilot Case Study. The rules were satisfactory and produced a fundamentally correct ICL specification with minor errors.

## 6.4 Possible Enhancements to ICL

The handling of value-lists is not yet flexible enough. The values passed by the ICL are indicated by "(*)" for an SDL signal and "!*" for a LOTOS action. This stands for the number of passed values. No checks on compatibility of those values are performed. The ICL specification header for LOTOS does not contain the information as regards which gate may be used to pass which kind of value and how many values. ICL does not handle a one-to-many mapping. If an action on the LOTOS side is used in more than one ICL ( action->signal ) statement then the SDL system always has to be the same. Also as in most languages, there is a specific ordering to the mapping of values. Subgates have to be at the first position in the value list and PId values have to be at the last position. All other values have to be in the same order on the LOTOS and SDL side. For instance it is not possible to map the gate *<"g",<"O","true">>* to signal *<"s",<"true","O">,*,*,*>*. In this case a more sophisticated notation which does not depend on subgating would be useful.

LOTOS and SDL share the same representation of their data values, that is Abstract Data Types (ADTs). This made it easier for the SPECS tools to validate the ICL specification against the formal specifications. ICL may also be used to combine an SDL specification with an ESTELLE specification as this would be similar to combining two SDL specifications. This would involve modifications to the tools or development of new tools to check the correctness of the ICL specification for SDL/ESTELLE combination.

### 6.4.1. Refinement of the Data Type Approach

The formalism used by the tower languages SDL and LOTOS to define data types was found to be quite difficult to understand. This is because the task of correctly and completely specifying a datatype in an axiomatic way ( by means of signatures and equations), is not an easy one, and often underestimated even by experts. Thus there is room for improvement in the support for development of datatypes. For example, one such improvement would be adding constructs to the language that are oriented towards a more imperative style of datatype specification, e.g. enumerated types, records. Another improvement would be to decorate signature definitions with special annotations, called pragmas, which could be used to generate implemented datatypes in the common representation languages (CRL). A more long-term improvement would be the use of these pragmas to generate equations automatically and so align this approach with the axiomatic one. [SSH 92].

### 6.4.2 Nested Mixing

The SPECS mixing approach was originally only intended to show the feasibility of interconnecting LOTOS and SDL specifications by making use of an intermediate language. Therefore it was not a design goal to provide support for the combination of mixed systems. Nevertheless it turned out that ICL is powerful enough to express such a "nested" mixing. ICL can handle LOTOS gate parameters, LOTOS value parameters, SDL channels and SDL signals. As this list covers the possible external interfaces of the "ICL system" (i.e. a system that is not described in terms of SDL or LOTOS specification language but by means of ICL itself), there is no theoretical difficulty in allowing ICL systems as subsystems. This will support the assembly of large systems from smaller parts which are complete, self-contained specifications and can therefore be understood and analysed separately.

## 6.5 Summary.

This dissertation has provided rules to mix SDL and LOTOS specifications. This is achieved by viewing the specification in one langauge as part of the other language's environment. These rules, "*ICL Production Rules*" are presented in chapter 4. An application of these rules to an industrial example is given in Chapter 5. The feasibility of such intermixing increases the possibility of reusing specifications as sub-systems in a larger context and allows the use of the most appropriate specification language for a given task.

## Bibliography

[ARIS 92] ARISE Document ARCE0048.MSW, "Specification and Requirements Document for CET Case Study", version 1.0, 6th January 1992.

[BARR 85] S. Barra, O. Ghisio, M. Modesti "Mapping SDL data tyes in CHILL", CSELT Technical reports - Vol XII- No3, 1985.

[BELI 88] Ferenc Belina, "The CCITT-Specification and Description Language SDL", University of Hamburg, Hamburg. 1987.

[BELI 91] Ferenc Belina, "SDL with Applications from Protocol Specification", Prentice Hall International (UK) Ltd, 1992. Chapter 7 pg 136.

[BELS 92] Dag Belsnes, "The Classification Process", NCC-Norweigan Computer Centre, Blindern, 0314 Oslo 3, Norway, 1992.

[BIEM 86] Frank Biemans and Pieter Blonk, "On the Formal Specification and Verification of CIM Architectures Using LOTOS", Philips CAM Centre, The Netherlands, 1986.

[BIND 91] Carl Binding, Heinz Saria and Heinrich Nirschl, "Mixing LOTOS and SDL Specifications", IBM Research Division, Zurich Research Laboratory, 8803 Ruschlikon, Switzerland, 1991.

[BOLO 89] Tommaso Bolognesi, "Introduction to the ISO Specification Language LOTOS", Elsevier Science Publishers B.V. (North-Holland), 1989.

[BOLO 90] Tommaso Bolognesi, "On the Soundness of Graphical Representations of Interconnected Processes in LOTOS", C.N.R.-CNUCE, 36, Via S. Maria - 56100 Pisa - ITALY, 1990.

[BRIN 87] Ed Brinksma, Giuseppe Scollo and Chris A.Vissers, "Experience with and Future of LOTOS as a Specification Language", University of Twente, The Netherlands, 1987.

[DUAP 91] Michel Dauphin, "SPECS: Formal Methods and Techniques for Telecommunications Software Development", Centre d'Etudes et Recherches IBM France, 1991 page 1-7.

[DWP3.8 92] D.WP3.8, "Final Methods and Tools for the Generation of Specifications", 46/SPE/WP3/DS/A/008/al, RACE 1046, 1992

[D4.17 92] "Final Methods and Tools for the handling of SPECS tower Languages", SPECS- Specification Handling, SPECS internal document id: 46/SPE/WP4/DS/A/017/b1 ver. 6, October 1992.

[ECHA, 86] W.F. Chan K. Turner, "The Daemon Game in ESTELLE, LOTOS and SDL" Draft Example, Joint Meeting ISO/CCITT (ISO/TC97/SC21/WG1/FDT-CCITTX/3), Turin, December 15-19, 1986.

[EHRI 85] H.Ehrig and B.Mahr, "Fundamentals of Algebraic Specification 1", Springer Verlag, 1985.

[EIJK 90] Peter Van Eijk, "Tools for LOTOS Specification Style Transformation", University of Twente, The Netherlands, 1990.

[FAER 92] Ove Faergemand, "Stepwise production of an SDL Specification", TFL, Lyngso Allé 2, DK-2970 Horsholm, Denmark, 1992.

[FACI 88] Faci, M., Logrippo, L., and Stepien, B. "Formal Specification of Telephone Systems in LOTOS", Brinksma, E., Scollo, G., and Vissers, C. (eds.) Protocol specification, Testing, and Verification VII, North-holland, 1988, 399-410.

[GUST 92] J.Gustafsson, E.Mumprecht, "LOTOS ADT Based Random Testing",, IBM Research Divison, Zurich Research Laboratory, 8803 Ruschlikon, Switerland, 22 January 1992.

[GUTT 85] John V. Guttag, James J. Horning, Jeannette M. Wing, "The Larch Family of Specification Languages", Massachusetts Institute of Technology, IEEE software 0740-7459/85/0900/0024, September 1985.

[HALL 90] Anthony Hall, "Seven Myths of Formal Methods", IEEE Software 0740-7459/90/0900/0011, 1990.

[HOAR 85] Hoare, C.A.R. "Communicating Sequential Process", Prentice-Hall, 1985.

[HOGR 88] Dieter Hogrefe, Sebastiano Trigla, Ferenc Belina, "Modelling OSI in SDL", University of Hamburg, Fondazione Ugo Bordoni Rome, Telelogic Malmo, 1988 pg 1-4.

[ID35 91] SPECS-Specification Generation, "Prototype Methods and Tools for the Generation of Specifications, SPECS-id: 46/SPE/WP3/PI/C/005/a0, December 1991.

[ISO 90] "Conformance Testing Methodology and Framework - Part 3: the tree and tabular combined notation (TTCN)", ISO, DIS 9646-3, March 1990.

[I.WP3.9 92] I.WP3.9, "CR&F Specification of the SPECS Pilot Case Study", SPECS Specification Generation, CEC Id: 46/SPE/WP3/PI/C/009/a3, ver. 3, 7 December 1992.

[I.WP613 92] I.WP6.13, "Use of SPECS for implementation", SPECS-Implementation Generation, IBM France, 12 October 1992.

[KARN 91] Georg Karner, Heinz Saria, and Bo Bichel Noraek, "Simple ADTs in LOTOS", Alactel Austria - ELIN Research Centre, Ruthnergasse 1-7, A-1210 Wien, Austria, Telecom Research Laboratory, Lyngso Alle, DK-2970 Horsholm, Denmark, 1991.

[KISH 91] Yoshinori Kishimoto, "SOFTON:A Flexible Software Construction Model by Interface Mediation", Systems Development Laboratory, Hitachi, Ltd., IEEE Software, 0730/3157/91/0000/0479, 1991.

[KOIC 91] Yoshinori Kishimoto, Koichi Yamano, "Softon: A flexible Software Construction Model by Interface Mediation", Systems Development Laboratory, Hitachi, 1099 Ohzenji, Asao-Ku, Kawasaki-shi, 215 Japan.

[KRON 93] Kronlof Klaus (Editor), "Method Integration, Concepts and Case Studies", John Wiley & Sons Ltd., 1993.

[LEDU 87] Leduc, G.J. "The Intertwining of Data Types and Processes in LOTOS." In: H.Rudin and C.H. West (eds.) Protocol Specification, testing, and Verification, VII. North-Holland, 1987, 123-136.

[LUIG 90] Luigi Logrippo, University of Ottawa, Protocols Research Group, Canada K1N 9B4; Tim Melanchuk, Advanced Development Group, Gandalf Data Ltd., 130 Colonnade Rd. S., Canada K2E 7J5; Robert J.Du Wors, Connected Systems Group, 61 Reaney Court, Canada K2K 1W7; "The Algebraic Specification Language LOTOS: An Industrial Experience", Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development, Napa, California, May 9-11, 1990

[MIGU 88] Tomas de Miguel and Jose A.Mahas, "An Implemtation Architecture for LOTOS", Dpto:Ingenieria Telematica (ETSIT) ETSI Telecomunicacion, UPM, Madrid, SPAIN, 1988

[MILN 85] R.Milner, "A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol.92, Springer-Verlag, 1985.

[MURP 91] S. Murphy, "Experiences with Estelle, LOTOS and SDL: a protocol implementation experiment, Per Guinningberg and J.PJ.Kelly, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA.

[NCC 86] National Computer Centre, "STARTS- Software Tools for Application to large Real-Time Systems - SDL", ISBN 0 850012 5774, 1986.

[PUEN 86] J.A. de la Puente, A Crespo and T.Perez, "Formal specification of Real-Time Software Systems. An Industrial Example", IFAC Real Time Programming, Lake Balton, Hungary 1986.

[QUEM 89] J. Quemada and A. Azcorra, "A Constraint Oriented Specification of Al's Node", Department of Telematics Engineering, Madrid University of Technology, SPAIN, 1989.

[Q.931 92] ISDN User-Network Interface Layer 3 Specification for Basic Call Control, 355 pages, CCITT Blue Book, volume VI, fascicle 11, recommendation Q.931 (I.451).

[REED 92] Rick Reed, Michel Dauphin, John Evans, "Methods for Quality and Automation", 1992.

[SCOL 87] Giuseppe Scollo and Marten Van Sinderen, "On the architectural design of the formal specification of the session standards in LOTOS", Twente University of Technology, Department of Computer Science, The Netherlands, 1987.

[SARI 91] Heinz Saria, "Definition of ICL", SPECS ID. A.A41.ELIHS354, March

1991.

[SPECS 92] SPECS RACE Ref:1046, "The Project SPECS", The SPECS Consortium, 1992, page 5.

[SPECS 93] SPECS The Consortium, "Specification and programming Environment for Communication Software", North-Holland, 1993.

[SSH 92] SPECS-Specification Handling (WP4), "Final common methods and tools for the handling of SPECS tower languages", ed. ANV-ELIN, August 1992.

[TRET 89] Jan Tretmans. "Hippo : A LOTOS simulator". In Peter H.J. van Eijk, Chris A.Vissers, and Michel Diaz, editors, "The Formal Description Technique LOTOS", pages 391-396, Elsevier Science Publishers B. V. (North Holland), 1989.

[TRET 92] J.Tretmans and L.Verharrd, "A Queue Model Relating Synchronous and Asynchronous Communication". Internal Report TFL RR 1992-1, TFL, Horsholm, Denmark, 1992. Memorandum INF-92-04, University of Twente, Enschede, The Netherlands, 1992.

[TURN 87] Kenneth J. Turner, "LOTOS : a practical formal description technique for OSI, University of Stirling, Scotland, 1987.

[TURN 89] Kenneth J. Turner, "The Formal Specification Language LOTOS", Department of Computing Science, University of Stirling, Stirling FK9 4LA, Scotland, 8th August, 1989.

[VALE 90] A.Valenzano, R.Sisto and L.Cimminiera, "An abstract execution model for basic LOTOS", Software Engineering Journal, November 1990.

[VVD 89] van Eijk, P., Vissers, C.A., and Diaz, M. "The formal Description Technique LOTOS". North-Holland, 1989.

[VISS 86] Chris A. Vissers, Giuseppe Scollo, and Antonello Di Stefano, "LOTOS in Practice", Department of Computer Science, Twente University of Technology, The Netherlands, 1986.

[VISS 87] Chris A. Vissers, "LOTOS Background", University of TWENTE, Dept. Informatics, 7500, AE Enschede, The Netherlands, 1989.

[VISS 88] Vissers, C., Scollo, G., Alderden, R.B., Schot, J., and Ferreira-Pires, L. "The Architecture of Interaction Systems", Lecture Notes, Twente University of Technology (NL), 1989.

[WILE 90] Jack C. Wileden, Alexander L. Wolf, Willan R. Rosenblatt, Peri L.Tarr, "Specification Level Interoperability", AT&T Bell Laboratories Murray Hill New Jersey, Computer and Information Science Department, University of Massachusetts, 1990.

[WING 87] Jeannette M. Wing, "Writing Larch Interface Language Specifications", Carnegie-Mellon University, ACM transactions on Programming Languages and Systems, Vol.9 No.1, January 1987, Pages 1-24.

[X.208 87] "Specification of Abstract Syntax Notation One (ASN.1)", recommendation X.208, CCITT, 1987.

# Appendix A The Informal Specification of the Alarm Service

The informal specification was provided by CET, a Portuguese member of the ARISE project. It is a section of CET's ELDIS project. ELDIS is an ISDN rural telephone exchange used widely in Portugal. The case study is concerned with the provision of supplementary services to the users of an ELDIS exchange. One such service is the Alarm Call Service which is used in the Pilot Case Study.

## A.1 Description of the Alarm Call Service

The following description of the Alarm Call Service is taken from [ARIS 92]:

### A.1.1 Introduction

The Alarm Call Service is an ELDIS proprietary service and is activated by inputs of the subscriber.

### A.1.2 Procedures

**General**

The operation of Alarm Call in Keypad mode makes use of the KEYPAD and DISPLAY information elements inserted in adequate messages of the call basic control. The invocation procedures use the * and # characters with the following meaning:

* - *Start of invocation/split of fields*

*# - End of invocation*

**Activation/Deactivation/Interrogation**

**Activation**

For activation of AC service, the user shall send a SETUP message with the KEYPAD FACILITY information element with the following coding:

* <Service-Code> * <Hour> * <Minute> #

*Service-Code = 313*

*Hour = A two digit number in the range 0..23*

*Minute = A two digit number in the range 0..59*

**Deactivation**

For deactivation of the AC service the user shall send a SETUP message with the KEYPAD FACILITY information with the following code

* <Service-Code> #

*Service-Code = 313*

**Interrogation**

To interrogate the network about the AC service the user shall send a SETUP message with the KEYPAD FACILITY information element with the following code:

* # <Service-Code> #

*Service-Code = 313*

### A.1.3 Notification

**Notification on Activation**

When activating the AC service, the network shall answer in the case of unsuccessful activation with the following code:

*ERRO DE ACTIVACAO*

and, if successful, the display shall be the following:

*FACILIDADE ACEITE*

**Notification on deactivation**

When deactivating the AC service, the network shall answer in the case of success with the following code:

*FACILIDADE DESACTIVADA*

and if unsucessful, the display will be:

*ERRO DE DESACTIVACAO*

**Notification on interrogation**

When interrogating the AC service, the network shall answer with DISC, REL or REL COMP message, including the DISPLAY information element with the following code:

*DESPERTAR HH:MM*

being HH the hour and MM the minutes field. When interrogating the AC service, the network shall answer in the case of unsucessful activation with the following code:

*ERRO NA CONSULTA*

As the task of working with this informal specification proceeded, a number of communications, by telephone and meetings, with CET took place to elicit further information. Elsewhere in the informal specification there was a brief description of an Analyser which took the invocation segments from the User and passed them onto the relevant supplementary service. When this system was specified by SPECS-Specification Generation, the *Analyser* was expressed using SDL and the supplementary service processor (*AC-processor*) was expressed in LOTOS. For the purpose of this example, it is the *Analyser* which sends the user's commands to the AC-processor. The Analyser performed some initial parsing on the user's inputs which made them more easily processable by the *AC-Processor*.

**Alarm Timer**

Although it was not specifically required, it was decided to design a process which initiated alarm calls. This was deemed to be necessary as otherwise comprehensive testing and demonstration of the case study would be impossible. This extra process was call the *Alarm-Timer*. It receives the current time and checks to see which users required a call at this time.

# Appendix B

## The Classified specification for the Pilot case study

### B.1 Conceptual structure of the classified specification

A classified specification is a set of "*classified components*". Figure B.1 shows the conceptual structure of the classified specification of the Pilot Case Study. Each "*classified component*" is represented by a rounded box, labelled with the name of the component.
The (unidirectional) "*conceptual structure links*" between classified components are denoted by arrows. Each arrow is labelled with the type of link, according to their definition in [ID35 91].
The dashed horizontal lines suggest interaction, through messages, between objects of the "classes" connected by each line.
A "*class*" describing such a message can be highlighted by crossing the dashed line with a short line attached to the "*class*" (e.g. 'Segment').

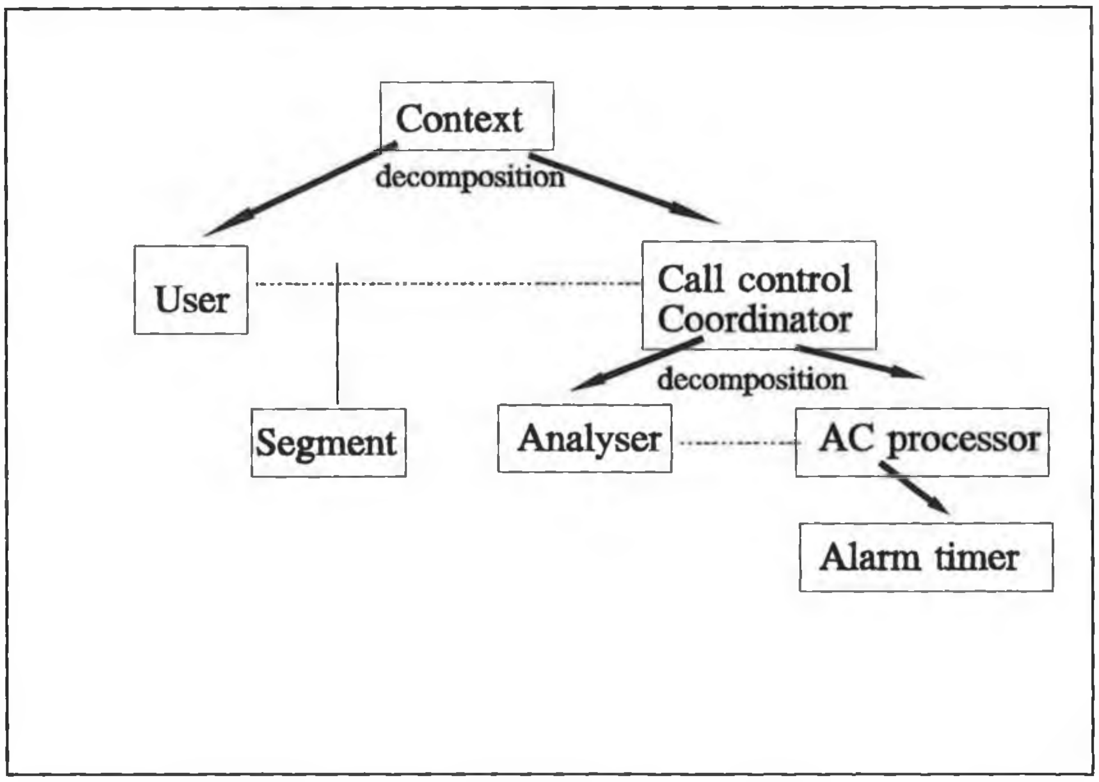

Figure B.1: Conceptual structure of the classified specification.

### B.2 Conventions in the Textual Classified Specification

Class names begin with an Uppercase letter e.g. Context
Class (interface, information and behaviour) aspects appear in lowercase characters (e.g. address).
Quoted class names and aspects, defined in some class, appear between single quotes (e.g. 'Segment' class or 'indication' event).
External interface aspects are defined in terms of inputs and outputs. A colon (:) separates the name of each input or output from the respective definition.
The following data types (classes) are assumed to be predefined:

Boolean - true or false logical type.
Integer - zero or positive number.
Hour - integer range from 0 to 23.
Minute - integer range from 0 to 59.
(IA5) character - 7 bits ASCII symbol.
(IA5) character string - chain of characters.

### B.2.1 The "Context" Classified Component

Description : system context in the considered problem.
Interface : (this component sets the boundary of the system context; only an internal interface between the part objects **user** and **call_control_coordinator** is classified.)

```
CLASSIFIED COMPONENT Context

 concept structure

     decomposition links

       --> * Call_control_coordinator
       --> * User
       --> * Segment

 class Context

  information aspects CLOSED

      part objects

       * part object user CLOSED of
           CLOSED * class User

       * part object call_control_coordinator CLOSED of
           CLOSED * class Call_control_coordinator

  interface aspects

      internal interface
      'user' 'request' interfaces to
      'call_control_coordinator' 'indication';

      'call_control_coordinator' 'response'
      interfaces to 'user' 'confirmation'.

 end of class

END OF CLASSIFIED COMPONENT
```

### B.2.2 The "User" Classified Component

Description: Abstract and generic user (terminal) of an ELDIS ISDN local exchange. Abstract, insofar as an active peer entity of ISDN local layer 3 protocol - DSS.1 LAPD [Q.931 92] - is reduced to a dummy peer entity of "layer 4" virtual communication. And generic, because *address* is meant to be a parameter of class *User*, though a single *user* is considered.
Interface: conventional primitives **request** and **confirmation** of an OSI layer. The data types *Integer* and *IA5 character string* are assumed to be pre-defined.

```
CLASSIFIED COMPONENT User

 class User

  information aspects CLOSED

     part objects

       * part object address CLOSED of
          CLOSED * class Integer

     relationships

       * relation CLOSED invocation_segment CLOSED
        sequence of
          CLOSED * class Segment

       * relation CLOSED notification_segment CLOSED
        sequence of
          CLOSED * class Segment

  behaviour aspects CLOSED

     behaviour description

        * CLOSED on internal event service_invocation_ready
```

```
    do
       atomic action sending of an object
       'invocation_segment' (through the 'request'
       output) conveying the 'User' 'Address' and a
       'keypad_information_element' invoking a
       supplementary service in "keypad mode".

    * CLOSED on external event confirmation do
       atomic action
        display the notification of an invoked
        supplementary service, conveyed by the
        received 'notification_segment' in a
        'display_information_element'.

  interface aspects

     external interface

      request: output a 'invocation_segment';

      confirmation: input a 'notification_segment'.

 end of class

END OF CLASSIFIED COMPONENT
```

## B.2.3 The "Segment" Classified Component

Description: abstraction of a "segment array". An abstract object which conveys the relevant information of a segment array is assumed to be common to both peer sides. So it is also regarded as a "layer 4" virtual message between both peer sides, thus bypassing the actual ELDIS ISDN local protocol (layers 1 to 3). Requirements on physical formats for part of a *Segment* are attached as the intrinsic miscellaneous aspects *Keypad coding* and *Display coding*
Interface: the data types *Integer, Hour* and *Minute* are assumed to be pre-defined.

```
CLASSIFIED COMPONENT Segment

 class Segment

  information aspects CLOSED

     part objects

      * part object address CLOSED of
         CLOSED * class Integer
      * part object keypad_information_element CLOSED of
          CLOSED * class IA5_character_string

      * part object display_information_element CLOSED of
          CLOSED * class IA5_character_string

  miscellaneous aspects

    --> * Keypad_coding
    --> * Display_coding

 end of class

END OF CLASSIFIED COMPONENT
```

### B.2.3.1 The ''Keypad coding'' Intrinsic Miscellaneous Aspect

*DESIGN CONSTRAINT Aspect: Software Design Constraint*

*The following information codes are strings of "IA5" (7 bits ASCII) characters such as, namely, the ones available in a phone keypad: *, # and the decimal digits (0 to 9).*

```
  1  Keypad Information Element Coding for Alarm Call
     (AC) Supplementary Service Invocation Procedures

     Service_Code = 313


     1.1  AC_activation_coding

        *<Service_Code>*<Hour>*<Minute>#

        Hour = A two digit number in the range 0..23

        Minute = A two digit number in the range 0..59


     1.2  AC_deactivation_coding

        #<Service_Code>#


     1.3  AC_interrogation_coding
```

*#<Service_Code>#

### B.2.3.2 The "Display coding" Intrinsic Miscellaneous Aspect

*DESIGN CONSTRAINT Aspect: Software Design Constraint*

*The following information codes are strings of "IA5" (7 bits ASCII) characters.*

*1 Display Information Element Coding for Alarm Call (AC) Supplementary Service Notification Procedures*

*In the next situations the network shall answer with the (DISC, REL or) REL COMP message which shall include the DISPLAY information element with the proper notification of the following ones.*

*1.1 AC_activation_notification*

*1.1.1 AC_activation_success_notification*

*FACILIDADE ACEITE*

*1.1.2 AC_activation_unsuccess_notification*

*ERRO DE ACTIVACAO*

*1.2 AC_deactivation_notification*

*1.2.1 AC_deactivation_success_notification*

*FACILIDADE DESACTIVADA*

*1.2.2 AC_deactivation_unsuccess_notification*

*ERRO DE DESACTIVACAO*

*1.3 AC_interrogation_notification*

*1.3.1 AC_interrogation_success_notification*

*DESPERTAR <HH>:<MM>*

*where HH are two digits representing the hour and MM denote the minute.*

*1.3.2 AC_interrogation_unsuccess_notification*

*ERRO NA CONSULTA*

## B.2.4 The "Call control coordinator" Classified Component

Description: the actual "system" under development. Restricted to the functionality of the AC supplementary service and without calls modelling.
Interface: external (virtual) interface through Segments (the real interface would be to a ELDIS local layer 3 protocol control module.). The external interface connects internally to *Analyser* which still interfaces to *AC processor.*

```
CLASSIFIED COMPONENT Call_control_coordinator

 concept structure

    decomposition links

      --> * Analyser
      --> * AC_processor

 class Call_control_coordinator

  information aspects CLOSED

     part objects

      * part object analyser CLOSED of
          CLOSED * class Analyser

      * part object ac_processor CLOSED of
          CLOSED * class AC_processor

     relationships
      * relation CLOSED invocation_segment CLOSED
       sequence of
          CLOSED * class Segment

      * relation CLOSED notification_segment CLOSED
       sequence of
          CLOSED * class Segment
```

*behaviour aspects CLOSED*

*behaviour description*

** CLOSED on external event indication do
atomic action sending of an object
'Segment' (the received 'invocation_segment')
to 'analyser' 'indication'.*

** CLOSED on internal event response do
atomic action sending of an object
'Segment' (the 'notification_segment' coming
from 'analyser' 'response') conveying a
'display_information_element'.*

** CLOSED on internal event alarm do
atomic action
- TO BE CONSIDERED ONLY IF MODELLING CALLS -
place an alarm call to the local 'User' whose
'address' was assigned to the triggered timer.*

*interface aspects*

*external interface*

*indication: input a 'invocation_segment';*

*response: output a 'notification_segment'.*

*internal interface*

*'analyser' 'AC_activation' interfaces to
'ac_processor' 'activation';*

*'analyser' 'AC_deactivation' interfaces to
'ac_processor' 'deactivation';*

*'analyser' 'AC_interrogation' interfaces to
'ac_processor' 'interrogation';*

*'ac_processor' 'notification' interfaces
to 'analyser' 'AC_notification';*

*'ac_processor' 'alarm' interfaces to
''call_control_coordinator' 'alarm'.*

*end of class*

*END OF CLASSIFIED COMPONENT*

### B.2.5 The "Analyser" Classified Component

Description: "demultiplexer" of invocations of the AC supplementary service.
Interface: two separate external interfaces, respectively, with *Call control coordinator* and with *AC processor.*

```
CLASSIFIED COMPONENT Analyser

 class Analyser

  behaviour aspects CLOSED

      behaviour description

        * CLOSED on external event indication do

            case
              selector is 'keypad_information_element'
              of (the received) 'Segment'

            * when AC_activation_coding
              do
              atomic action sending of an object
              with "hour", "minute" and 'User' "address"
              through 'AC_activation' output.

            * when AC_deactivation_coding
              do
              atomic action sending of an object
              with 'User' "address" through
             'AC_deactivation' output.

            * when AC_interrogation_coding
              do
              atomic action sending of an object
              with 'User' "address"  through
              'AC_interrogation' output.

            end of case

        * CLOSED on external event AC_notification do

            atomic action sending of an object
            'Segment' (through 'response' output) to
```

*'Call_control_coordinator' 'response', conveying, in a 'display_information_element', an eventual notification about an AC supplementary service invocation (received through 'AC_notification' input).*

*interface aspects*

*external interface*

*indication: input a 'Segment';*

*response: output a 'Segment';*

*AC_activation: output invoking activation of AC for an 'User', with "address", "hour" and "minute" extracted from the 'Segment' received as 'indication';*

*AC_deactivation: output invoking deactivation of AC for an 'User' with "address" extracted from the 'Segment' received as 'indication';*

*AC_interrogation: output invoking interrogation of AC for an 'User' with "address" extracted from the 'Segment' received as 'indication';*

*AC_notification: input notification information about an AC invocation.*

*end of class*

*END OF CLASSIFIED COMPONENT*

**B.2.6 The ''AC processor'' Classified Component**

Description: processor for invocations of the AC supplementary service.
Interface: two separate external interfaces: one, explicit, with *AC processor* and another, implicit, with *Call control coordinator* for placing an eventual "alarm call" to the respective user.

*CLASSIFIED COMPONENT AC_processor*

*concept structure*

*layering links*

```
      --> * Alarm_timer

class AC_processor

 information aspects CLOSED

    relationships

     * relation CLOSED alarm_database CLOSED set of
        CLOSED * class Alarm_timer

 behaviour aspects CLOSED

    action definition

      * CLOSED action authorization_check is
          atomic action
           Check in 'alarm_database' if there is
           any 'alarm_timer' assigned to the user
           identified by its "address".
           If so then the result is "success".
           Otherwise it means "unsuccess".

      * CLOSED action hour_minute_check is
          atomic action
           Check the validity of the given "hour"
           and "minute" values (respectively belonging
           to the ranges 0 to 23 and 0 to 59).
           If so then the result is "success".
           Otherwise it means "unsuccess".

      * CLOSED action on_check is
          atomic action
           'ask_on_off' of the 'alarm_database'
           'Alarm_timer' given by 'User' 'address' of
           'activation'. If it returns 'true' (Alarm
           timer on) then the result is "success".
           Otherwise it means "unsuccess".

      * CLOSED action activation_checks is
          atomic action
           perform 'authorization_check'.
           If successful, perform 'hour_minute_check'.
           If the previous conditions hold then the
           result is "success".
           Otherwise it means "unsuccess".

      * CLOSED action
    deactivation_or_interrogation_checks
  is
    atomic action
     perform 'authorization_check'.
     If successful, perform 'on_check'.
     If the previous conditions hold then the
     result is "success".
     Otherwise it means "unsuccess".

behaviour description

  * CLOSED on external event activation do

    sequence of

    * action activation_checks

    * case selector is 'activation_checks' result

      * when success
        do
        sequence of
        * atomic action sending of an object
          to 'set_alarm' of the 'alarm_database'
          'Alarm_timer' given by 'User' "address"
          of 'activation', with "hour" and "minute"
          of 'activation'.
        * atomic action sending of an object
          through 'notification' output, with
          the information in the
          'AC_activation_success_notification'
          format.
        end of sequence

      otherwise

       atomic action sending of an object
       through 'notification' output, with the
       information in the
       'AC_activation_unsuccess_notification'
       format.

      end of case

    end of sequence

  * CLOSED on external event deactivation do
```

```
   sequence of

   * action deactivation_or_interrogation_checks

   * case
     selector is
      'deactivation_or_interrogation_checks'
      result

    * when success
      do
      sequence of
      * atomic action sending of an object
        to 'turn_off' of the 'alarm_database'
        'Alarm_timer' given by 'User' "address"
         of 'activation'.
      * atomic action sending of an object
        through 'notification' output, with
        the information in the
        'AC_deactivation_success_notification'
        format.
      end of sequence

    otherwise

     atomic action sending of an object
     through 'notification' output, with the
     information in the
     'AC_deactivation_unsuccess_notification'
     format.

    end of case

   end of sequence

* CLOSED on external event interrogation do

   sequence of

   * action deactivation_or_interrogation_checks

   * case
     selector is
      'deactivation_or_interrogation_checks'
      result

    * when success
      do
```

```
                sequence of
                * inclusion of partial sequence
                   'ask_setting' of the 'alarm_database'
                   'Alarm_timer' given by 'User'
                   "address" of 'activation' (returning
                   the 'Alarm_timer' 'hour' and 'minute').
                * atomic action sending of an object
                  through 'notification' output, with
                  the information in the
                  'AC_interrogation_success_notification'
                  format.
                end of sequence

              otherwise

               atomic action sending of an object
               through 'notification' output, with the
               information in the
               'AC_interrogation_unsuccess_notification'
               format.

              end of case

            end of sequence

        * CLOSED on internal event alarm_timeout do
            atomic action sending of an object
            (through 'alarm' output) with the 'User'
            "address" of 'alarm_timeout'.

interface aspects

    external interface

     activation: input the "hour", "minute" and
     'User' "address" for AC invocation;

     deactivation: input the 'User' "address" for the
     supplementary service deactivation;

     interrogation: input the 'User' "address"
     for inquiring the network about the AC service;

     notification: output notification information
     about an AC invocation;

     alarm_timeout: input an 'User' "address";
```

*alarm: output an 'User' "address".*

*end of class*

*END OF CLASSIFIED COMPONENT*

**B.2.7 The "Alarm timer" Classified Component**

Description: functionality of a generic alarm clock for a AC supplementary service. Interface: the external interface (below) assumes the data types *Boolean, Integer, Hour* and *Minute* to be pre-defined.

*CLASSIFIED COMPONENT Alarm_timer*

*class Alarm_timer*

*information aspects CLOSED*

*part objects*

** part object address CLOSED of CLOSED * class Integer*

** part object on_off CLOSED of CLOSED * class Boolean*

** part object hour CLOSED of CLOSED * class Hour*

** part object minute CLOSED of CLOSED * class Minute*

*behaviour aspects CLOSED*

*behaviour description*

** CLOSED on external event set_alarm do*

*sequence of*

** atomic action state change set 'on_off' to true.*

** atomic action object value modification "hour" and "minute" of 'set_alarm' are assigned to, respectively, 'hour' and 'minute'.*

```
        end of sequence

      * CLOSED on external event turn_off do
        atomic action state change
         set 'on_off' to false.

      * CLOSED on external event ask_on_off do
        atomic action sending of an object
        Boolean (through 'current_on_off' output)
        with the 'on_off' state value.

      * CLOSED on external event ask_setting do
        atomic action sending of an object
        (through 'current_setting' output to
        'AC_processor') with the values of 'hour'
        and 'minute'.

      * CLOSED on internal event alarm_timeout do
        atomic action sending of an object
        (through 'alarm_timeout' output to
        'AC_processor') with the 'User' 'address'.

interface aspects

    external interface

     set_alarm: input an Hour and a Minute;

     turn_off: input;

     ask_on_off: input;

     current_on_off: output a Boolean;

     ask_setting: input;

     current_setting: output an Hour and a Minute;

     alarm_timeout: output an 'User' 'address'.

end of class

END OF CLASSIFIED COMPONENT
```

# Appendix C

## The Rigorous Specification for the Pilot Case Study

### C.1 Introduction

This appendix gives the rigorous models used to describe the functionality and behaviour of the supplementary services described in appendix A and conceptualised in appendix B. From these models a new process was introduced to the system which would check the time recorded in the user's records against the current time and return an alarm call to the user. This new process, called Alarm-Timer, is shown in Level 0 DFD and a description of its behaviour is given in section C.2.4.6.

### C.2 Rigorous Models

The Rigorous step of the CR&F methodology uses 5 different models to describe the system under examination. These are :

Data/Control Flow Diagrams (DDFDs)
Message Sequence Charts (MSCs)
Abstract Syntax Notation 1 (ASN.1)
Process Specification (PSPECs)
Entity-Relationship Diagrams (ERDs)

#### C.2.1 Data Flow Diagrams

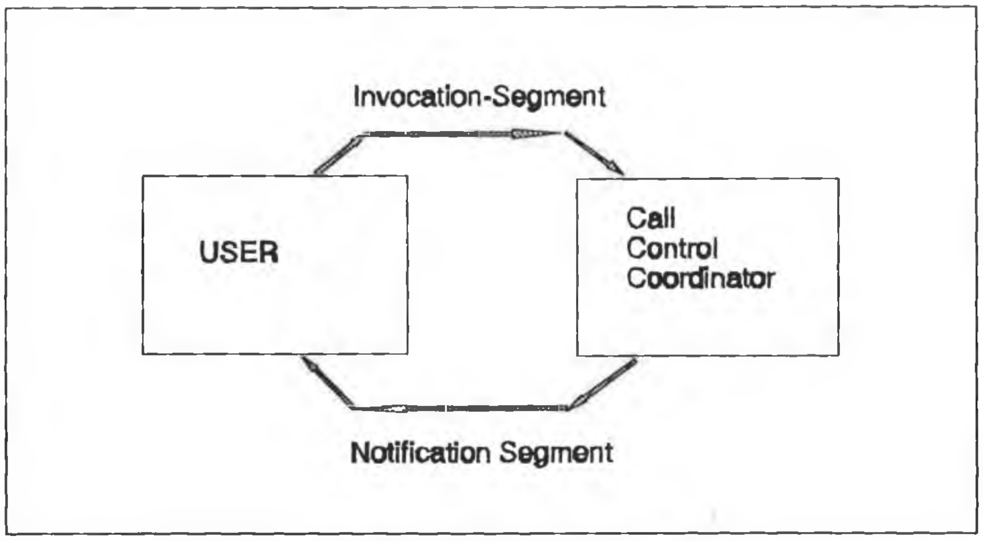

Figure C.1 Context Data Flow Diagram of the Pilot Case Study

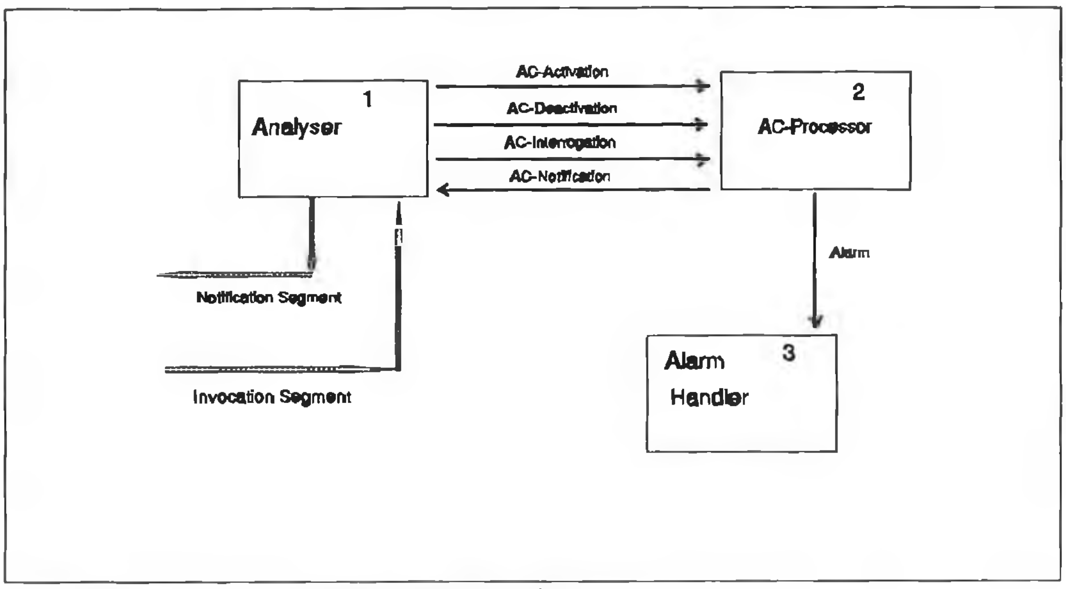

Figure C.2 Level 0 DFD of the Call control controller

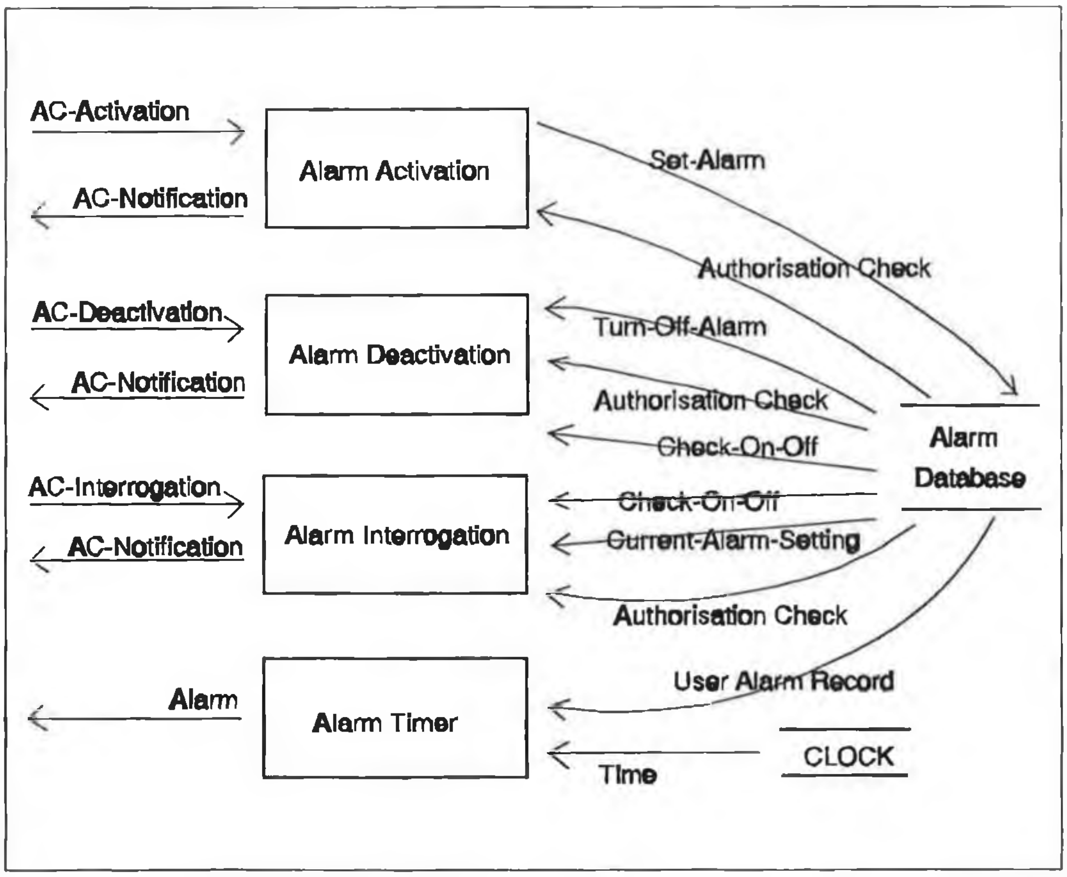

Figure C.3 Level 1 DFD of the AC-Processor

### C.2.2 Message Sequence Charts

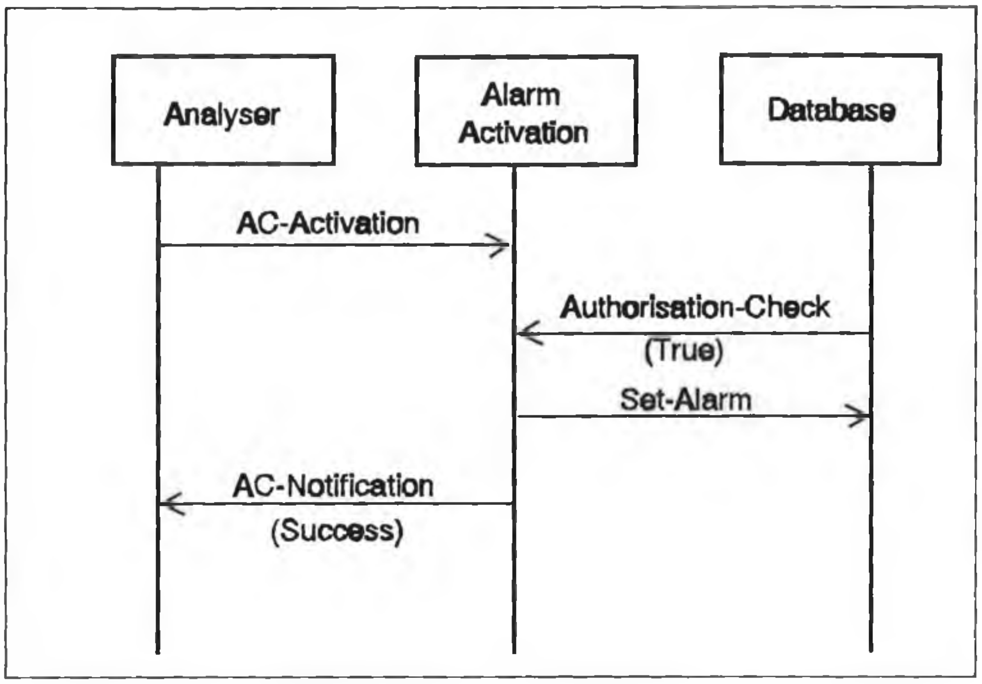

Figure C.4 : MSC for Successful Alarm Activation

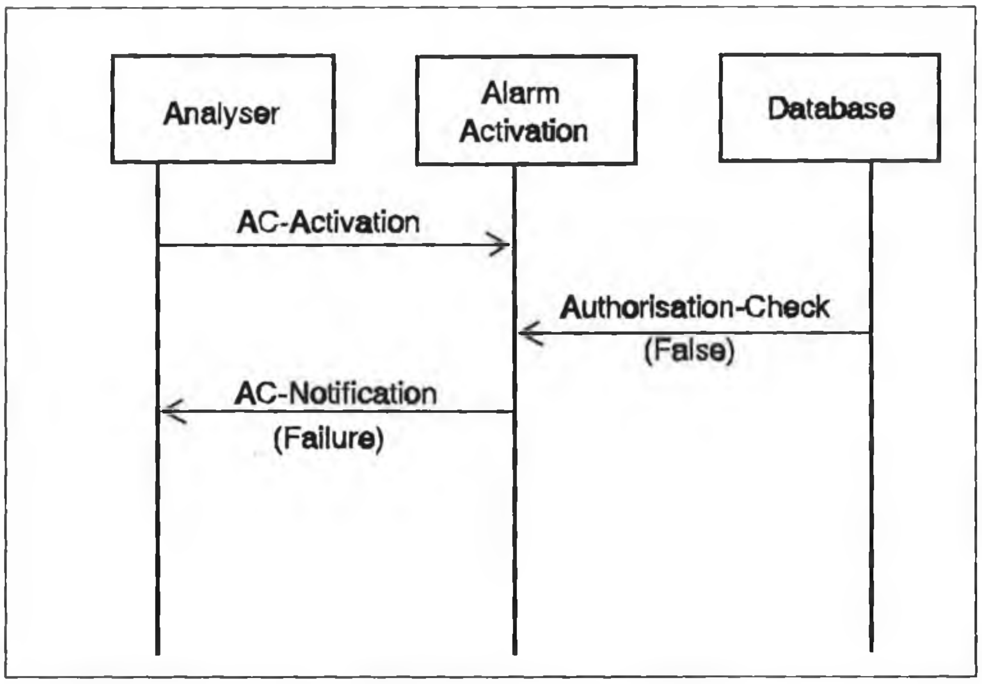

Figure C.5 MSC for an Unsuccessful Alarm Activation

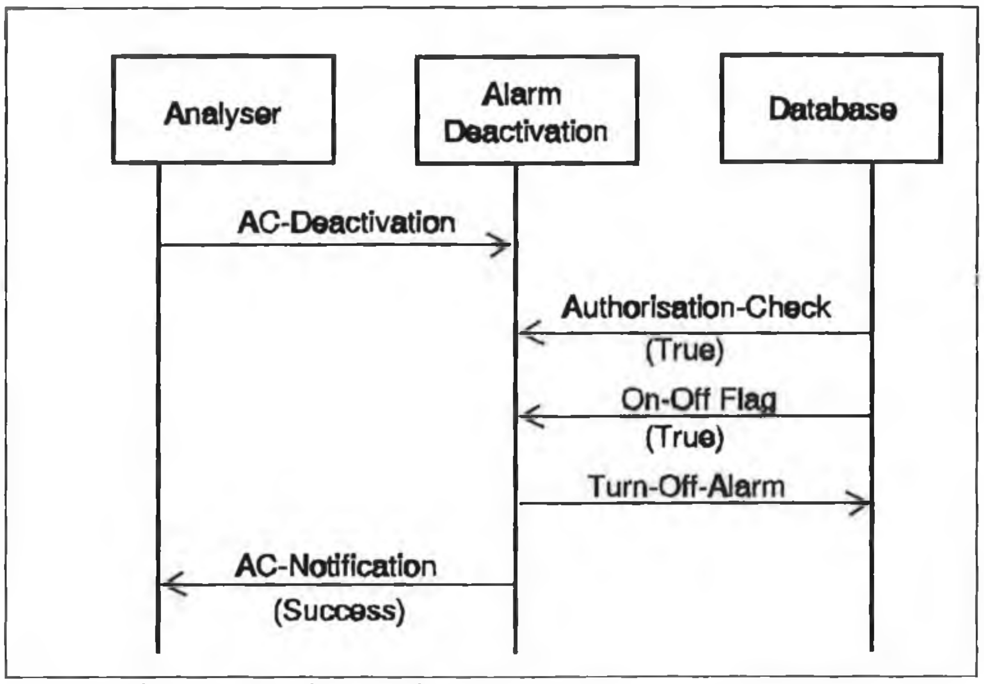

Figure C.6 MSC for Successful Alarm Deactivation

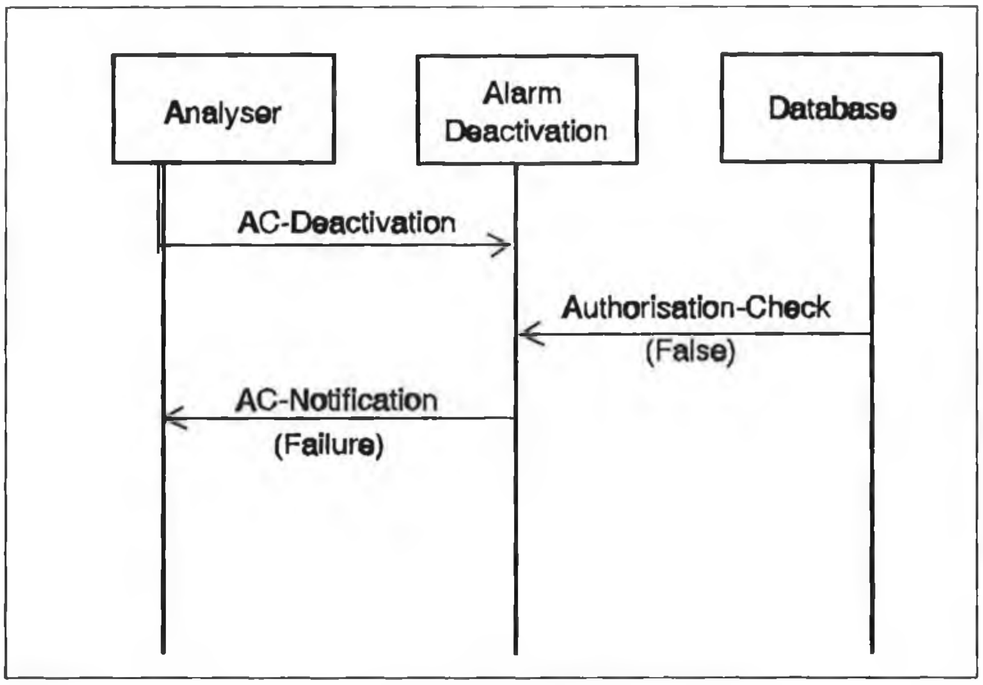

Figure C.7 MSC for Unsuccessful Alarm Deactivation (No. 1)

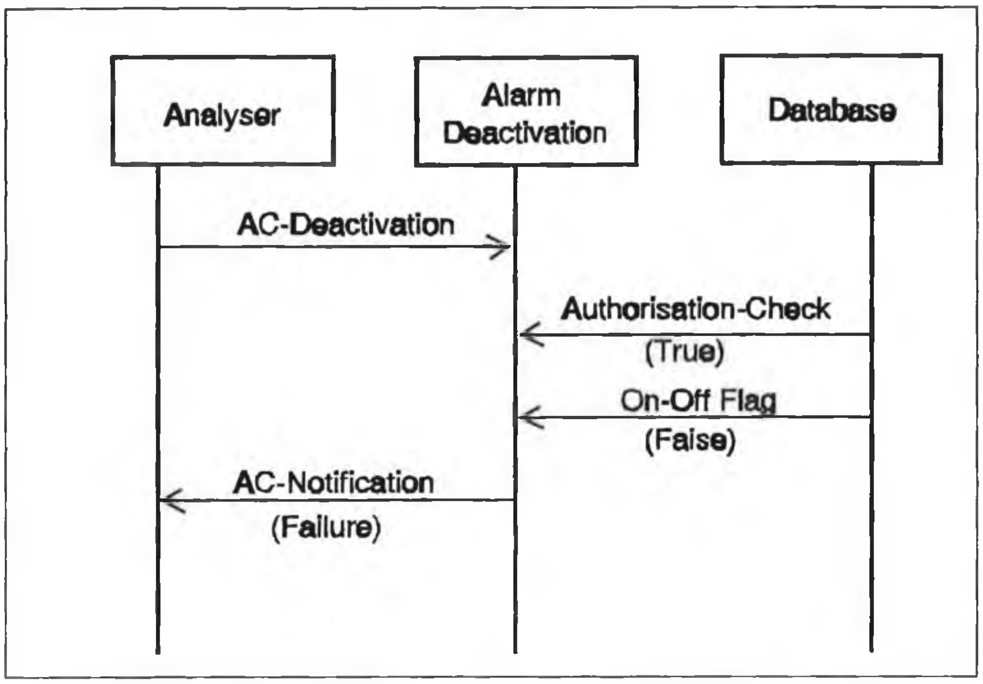

Figure C.8 MSC for Unsuccessful Alarm Deactivation (No.2)

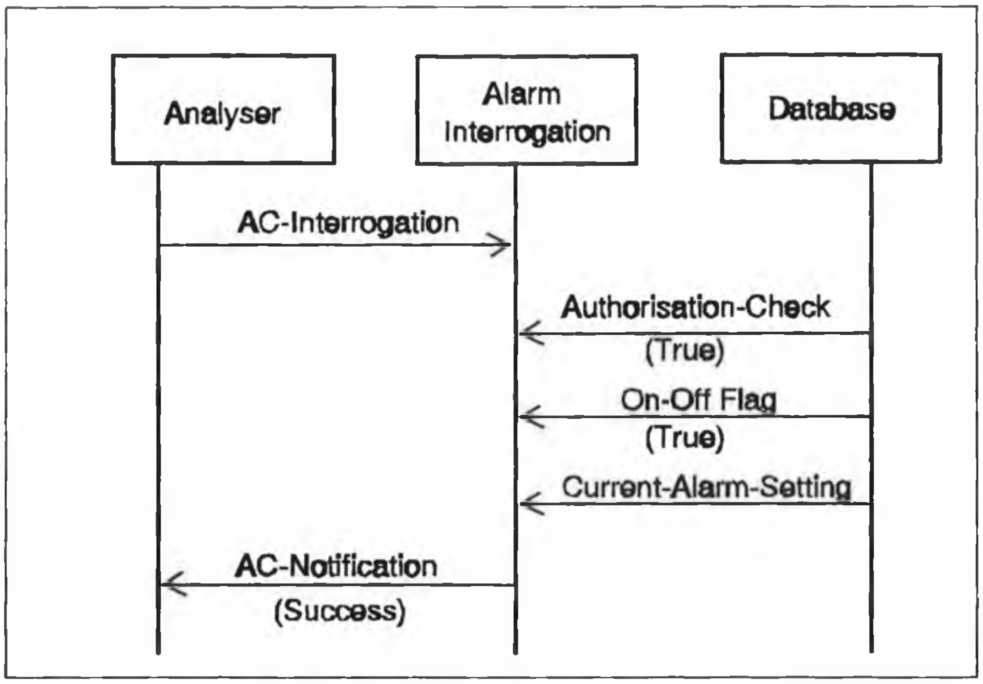

Figure C.9 MSC for Successful Alarm Interrogation

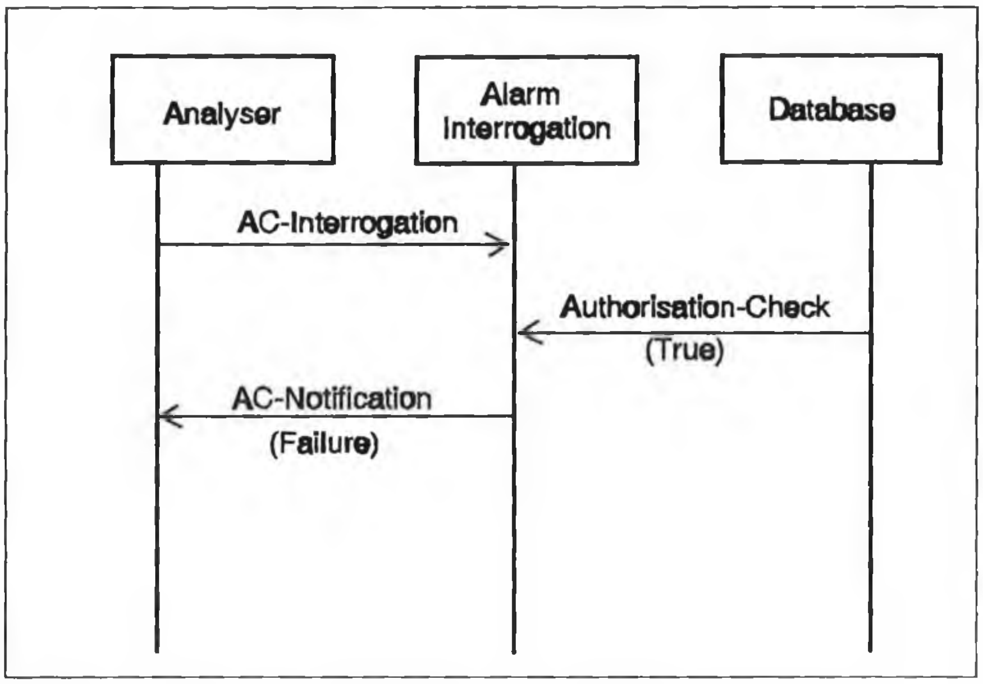

Figure C.10 MSC for Unsuccessful Alarm Interrogation (No.1)

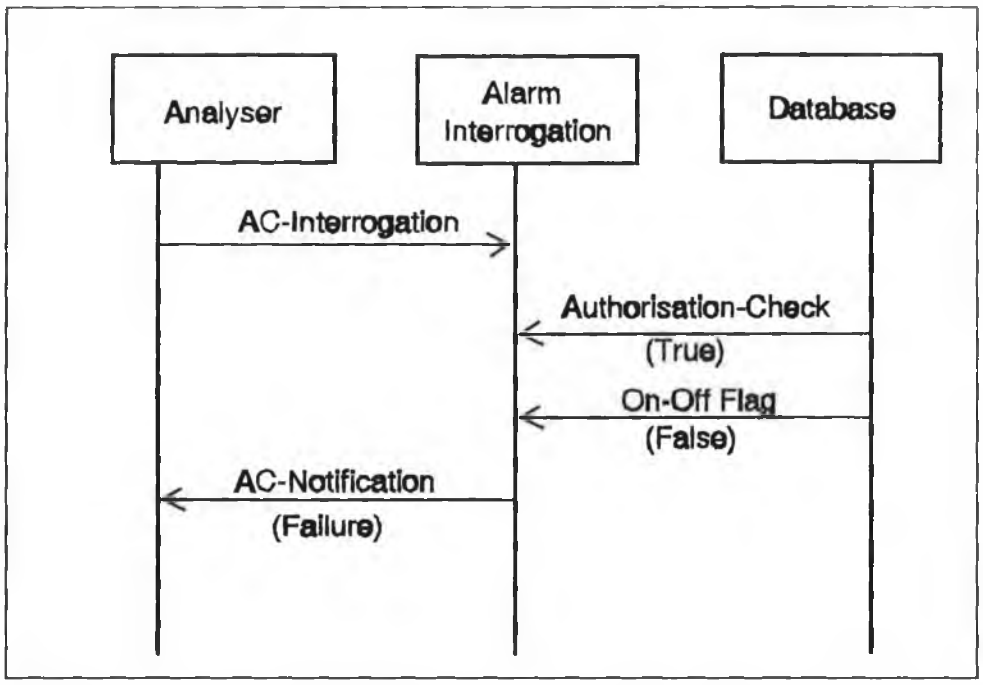

Figure C.11 MSC for Unsuccessful Alarm Interrogation (No.2)

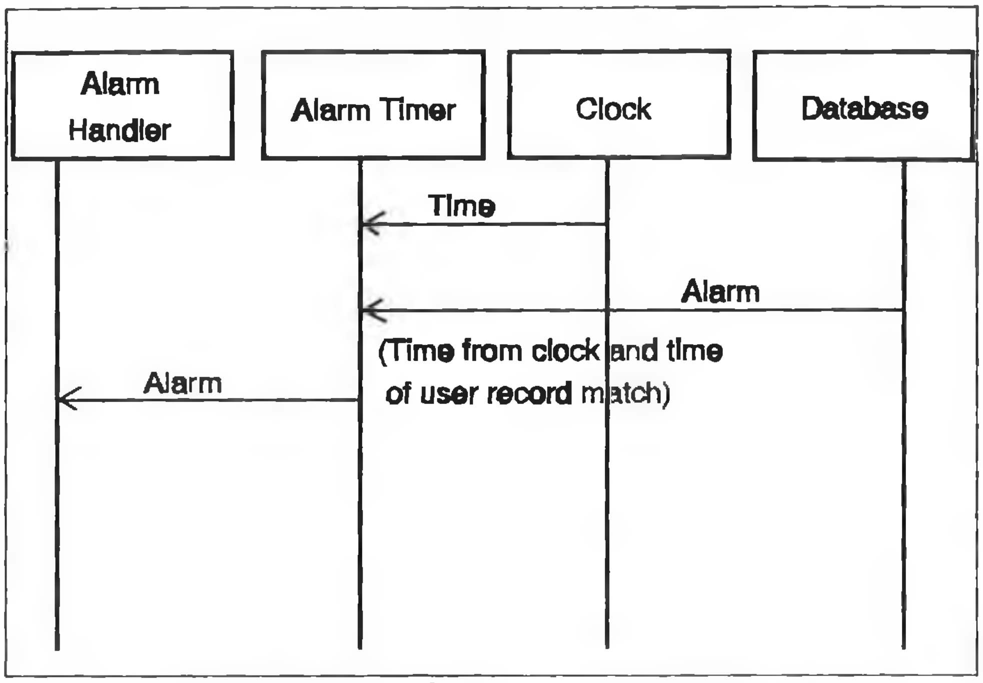

Figure C.12 MSC for Successful Alarm Timer

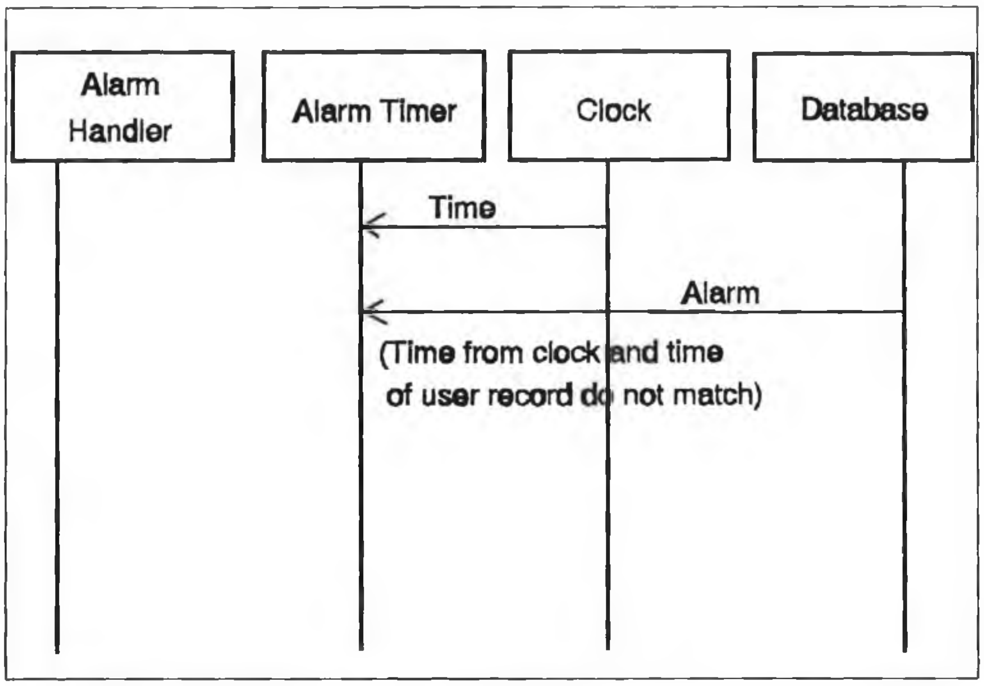

Figure C.13 MSC for Unsuccessful Alarm Timer

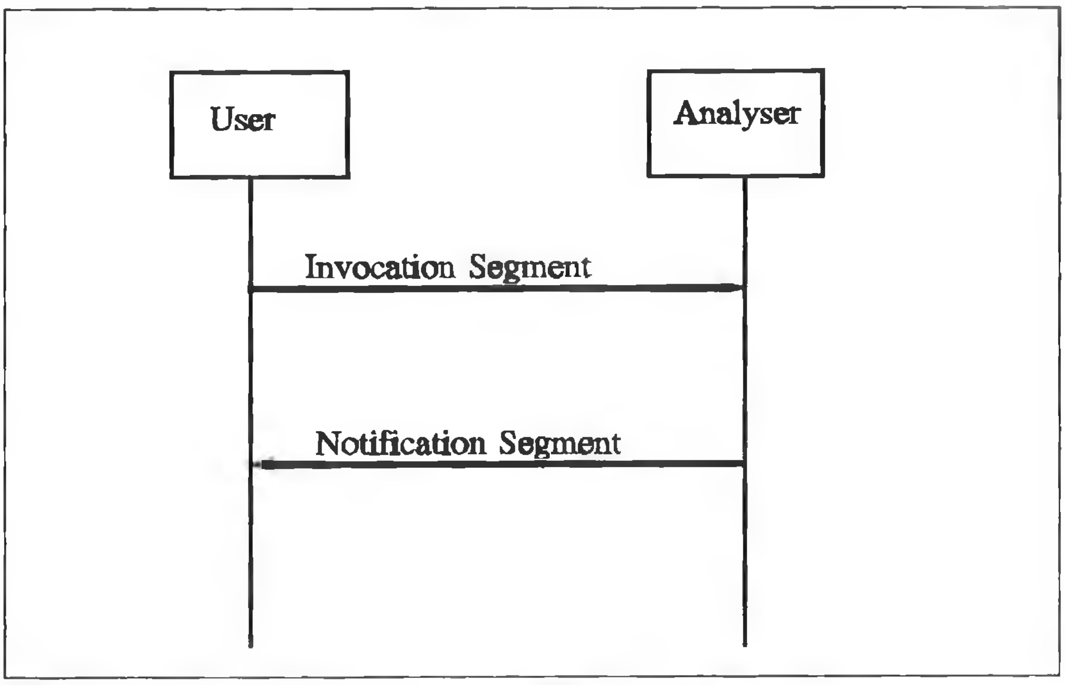

Figure C.15 MSC for User to Analyser.

### C.2.3 Abstract Syntax Notation 1 (ASN.1)

This section describes the data structures, data types and messages of variables used within the system by using ASN.1 notation. The syntax used is that described in [X.208 87].

```
ALARM-Database ::=
SET {
        address       INTEGER
        hour          INTEGER (0..23)
        minute        INTEGER (0..59)
        on_off        BOOLEAN
        authorization BOOLEAN
}

CLOCK ::=
SET {
        hour     INTEGER (0..23)
        minute   INTEGER (0..59)
    }
```

All the Data Flows contained in the DCFDs are described here using ASN.1 :

```
Invocation_Segment ::=
SET {
        address INTEGER
        keypad_coding IA5STRING ::=
        keypad_request
        display_coding IA5STRING empty
   }

keypad_request ::= ENUMERATED
{AC_Activation_coding(0), AC_Deactivation_coding(1),
AC_Interrogation_coding(2)}

AC_Activation_coding IA5STRING ::=
"*service_code*hour*minute#"

AC_Deactivation_coding IA5STRING ::=
"#service_code#"

AC_Interrogation_coding IA5STRING ::=
"*#service_code#"

service_code INTEGER ::= 313

hour   INTEGER (0..23)
minute INTEGER (0..59)

Notification_Segment ::=
SET {
        address INTEGER
        keypad_coding  IA5STRING empty
        display_coding IA5STRING ::=
        AC_Notification
   }

AC_Activation ::=
SET {
        address INTEGER
        hour    INTEGER
        minute  INTEGER
   }

AC_Deactivation ::=
SET {
        address INTEGER
   }

AC_Interrogation ::=
SET {
        address INTEGER
   }
```

```
AC_Notification ::= ENUMERATED
{successful_activation (0), unsuccessful_activation (1),
successful_deactivation (2), unsuccessful_deactivation (3),
interrogation_result (4), unsuccessful_interrogation (5)}

successful_activation IA5STRING ::=
"FACILIDADE ACEITE"

unsuccessful_activation IA5STRING ::=
"ERRO DE ACTIVACAO"

successful_deactivation IA5STRING ::=
"FACILIDADE DESACTIVADA"

unsuccessful_deactivation IA5STRING ::=
"ERRO DE DESACTIVACAO"

interrogation_result IA5STRING ::=
"DESPERTAR hour:minute"

unsuccessful_result IA5STRING ::=
"ERRO NA CONSULTA"

Set_Alarm ::=
SET {
        address INTEGER
        hour    INTEGER (0..23)
        minute  INTEGER (0..59)
        on_off  BOOLEAN ::= TRUE
   }

Turn_Off_Alarm ::=
SET {
        address INTEGER
        on_off  BOOLEAN ::= FALSE
   }

Current_Alarm_Setting ::=
SET {
        hour    INTEGER (0..23)
        minute  INTEGER (0..59)
   }

User_Alarm_Record ::=
SET {
        address INTEGER
        hour    INTEGER (0..23)
        minute  INTEGER (0..59)
```

```
          on_off   BOOLEAN
    }

Check_On_Off ::=
SET {
          on_off   BOOLEAN
    }

TIME ::=
SET {
          hour      INTEGER (0..23)
          minute    INTEGER (0..59)
    }

ALARM ::=
SET {
           address INTEGER
    }
```

### C.2.4 Process Specifications (PSpecs)

This section shows the behaviour of the processes of the system.

#### C.2.4.1 PSpec of Analyser

```
START
    NEXTSTATE wait for segment
STATE wait for segment
    INPUT Invocation_Segment /* from User */
    DECISION service_code = 313
        (TRUE):
            DECISION is AC_Activation_coding
              (TRUE):
                  OUTPUT hour, minute, address
                  /* AC_Activation to AC_processor */;
                  NEXTSTATE wait for Notification;
            ENDDECISION
            DECISION is AC_Deactivation_coding
              (TRUE):
                  OUTPUT address
                  /*AC_Deactivation to AC_processor*/;
                  NEXTSTATE wait for Notification;
            ENDDECISION
            DECISION is AC_Interrogation_coding
              (TRUE):
                  OUTPUT address
                  /*AC_Interrogation to AC_processor*/;
                  NEXTSTATE wait for Notification;
            ENDDECISION
        (FALSE):
            /* No description of what happens if service
               code is not 313 in informal or Classified
               specifications */
    ENDDECISION
STATE wait for Notification;
    INPUT AC_Notification;
    /* from AC-processor */
    OUTPUT Notification_Segment;
    /* created using AC_Notification and address read in
       above - output to the User */
    NEXTSTATE wait for segment;
```

#### C.2.4.2 PSPEC of Alarm Handler

This process will not be described here as no informal specification for it currently exists. Similarly no attempt will be made to formalise this into SDL or LOTOS.

**C.2.4.3 PSPEC of Alarm Activation**

```
START
    NEXTSTATE wait for AC_Activation;
STATE wait for AC_Activation
    INPUT AC_Activation /* from analyser */;
    DECISION((Hour >= 0) and (hour <= 23)
            and (minute >= 0) and (minute <= 59))
       (TRUE):
            NEXTSTATE check authorization;
       (FALSE):
            NEXTSTATE send error message;
    ENDDECISION
STATE check authorization
    INPUT authorization_check;
            /* from database */
    DECISION (authorization_check = TRUE)
       (TRUE):
           Set_Alarm.address = AC_Activation.address;
           Set_Alarm.hour = AC_Activation.hour;
           Set_Alarm.minute = AC_Activation.minute;
           Set_Alarm.on_off = TRUE;
           OUTPUT Set_Alarm
           /* to database */;
           OUTPUT successful_activation
           /* to Analyser as AC_Notification */;
           NEXTSTATE wait for AC_Activation;
       (FALSE):
           NEXTSTATE send error message
     ENDDECISION
STATE send error message
     OUTPUT unsuccessful_activation
    /* to Analyser as AC_Notification */;
     NEXTSTATE wait for AC_Activation
```

**C.2.4.4 SPEC of Alarm Deactivation**

```
START
    NEXTSTATE wait for AC_Deactivation;
STATE wait for AC_Deactivation
    INPUT AC_Deactivation;
    INPUT authorization_check;
            /* from database */
    DECISION (authorization_check = TRUE)
       (TRUE):
              NEXTSTATE check On_Off_Flag;
          (FALSE):
              NEXTSTATE send error message
        ENDDECISION
STATE check On_Off_Flag
        INPUT on_off
        /* from database */;
        DECISION (on_off = TRUE)
          (TRUE):
              turn_off_alarm.address =
                           AC_Deactivation.address;
              turn_off_alarm.on_off = FALSE;
              OUTPUT Set_Alarm
              /* to database */;
              OUTPUT successful_deactivation
              /* to Analyser as AC_Notification */;
              NEXTSTATE wait for AC_Deactivation;
          (FALSE):
              NEXTSTATE send error message
        ENDDECISION
STATE send error message
        OUTPUT unsuccessful_deactivation
        /* to Analyser as AC_Notification */;
        NEXTSTATE wait for AC_Deactivation
```

### C.2.4.5 PSPEC of Alarm Interrogation

```
START
      NEXTSTATE wait for AC_Interrogation;
STATE wait for AC_Interrogation
      INPUT AC_Interrogation;
      INPUT authorization_check;
              /* from database */
      DECISION (authorization_check = TRUE)
          (TRUE):
              NEXTSTATE check On_Off_Flag;
          (FALSE):
              NEXTSTATE send error message
        ENDDECISION
STATE check On_Off_Flag
        INPUT on_off
        /* from database */;
        DECISION (on_off = TRUE)
          (TRUE):
              INPUT current-alarm-setting
              /* from database */
              AC_Notification := interrogation_result
              + current-alarm-setting.hour + ':'
```

```
                + current-alarm-setting.minute
                OUTPUT AC_Notification
                /* to Analyser */;
                NEXTSTATE wait for AC_Interrogation;
            (FALSE):
                NEXTSTATE send error message
        ENDDECISION
STATE send error message
        OUTPUT unsuccessful_interrogation
        /* to Analyser as AC_Notification */;
        NEXTSTATE wait for AC_Interrogation
```

### C.2.4.6 PSpec of Alarm-Timer

/* Checks the user's entry in the alarm-database with the current time. When a match is found an alarm is sent to the user. */

```
START
        NEXTSTATE initialise old-time;
STATE initialise old-time
        old-time.hour := 24;
        old-time.minute := 24; /* set old-time to some invalid value */
        NEXTSTATE get time
STATE get time
        INPUT time /* from Clock */
        DECISION (old-time = time)
            (TRUE):
                NEXTSTATE initialise old-time;
            (FALSE):
                NEXTSTATE search user database;
        ENDDECISION
STATE search user database
        INPUT User-Alarm-Record; /* get Record for first User in database */
        NEXTSTATE check not end of list;
STATE check not end of list
        DECISION (valid(User-Alarm-Record))
            (TRUE):
                NEXTSTATE check users time;
            (FALSE):
                old-time := time;  /* end of list reached, start again */
                NEXTSTATE get time;
        ENDDECISION
STATE check users time
        DECISION ((User-Alarm-Record.hour = time.hour) and
                (User-Alarm-Record.minute = time.minute))
            (TRUE):
                NEXTSTATE check on_off;
```

```
      (FALSE):
          NEXTSTATE get next user record;
    ENDDECISION
STATE check on_off;
    DECISION (User-Alarm-Record.on_off = true)
      (TRUE):
          OUTPUT User-Alarm-Record.address; /* on Alarm flow */
          NEXTSTATE get next user record;
      (FALSE):
          NEXTSTATE get next user record;
    ENDDECISION
STATE get next user record
    INPUT User-Alarm-Record; /* get Record for next User in database */
    NEXTSTATE check not end of list
```

### C.2.5 Entity Relationship Diagram

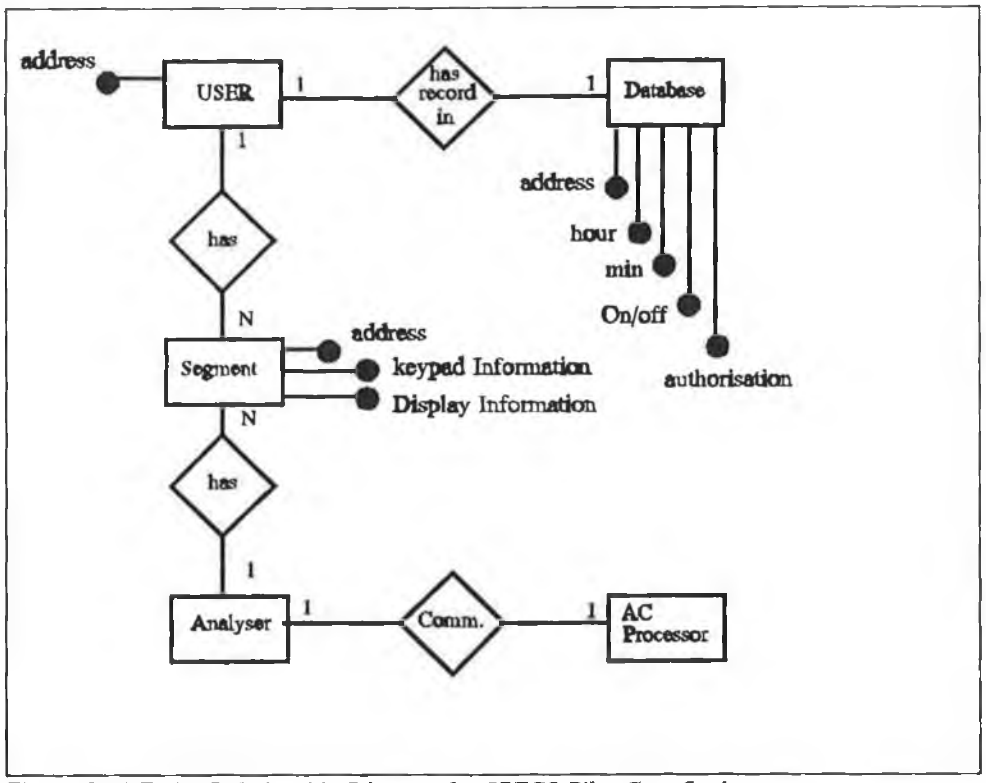

Figure C.14 Entity-Relationship Diagram for SPECS Pilot Case Study.

# Appendix D

## The SDL specification for the Pilot Case Study

```
        /* SPECS SDL component; version 1.3;
        on 92/05/14;
        translated to ACRL apparently with success.
        Minimum comments, to avoid odd observed problems.
        */

system analyser ;             /*  Parameters and Constants  */
 synonym max_calls            /* natural */
integer = 1 ;

                              /* Data Types */

 newtype service_code_type enum /* 313 */ ac_service_code
  endnewtype ;

 newtype invocation_type
  enum ac_activation , ac_deactivation , ac_interrogation
  endnewtype ;

 newtype ac_display_type
  enum
   facilidade_aceite , erro_de_activacao ,
    facilidade_desactivada , erro_de_desactivacao ,
    despertar , erro_na_consulta
   endnewtype ;

 syntype address_type = /* natural */ integer endsyntype ;

 syntype hour_type = /* -1 to 23 */ integer endsyntype ;

 syntype minute_type = /* -1 to 59 */ integer endsyntype ;

 newtype time_type struct hour hour_type ; minute minute_type
  literals dummy_time
   axioms dummy_time == make ! ( - 1 , - 1 ) endnewtype ;

               /* Signal interface to "User" (in environment) */
 signal
  invoke
   (
    address_type , service_code_type , invocation_type ,
    time_type
   ), notify ( address_type , ac_display_type , time_type ) ;

/* Signal interface with "AC processor" (in environment) */

signal
 ac_activate ( address_type , time_type ) ,
  ac_deactivate ( address_type ) ,
  ac_interrogate ( address_type ) ,
  ac_activate_success ( address_type ) ,
  ac_activate_unsuccess ( address_type ) ,
  ac_deactivate_success ( address_type ) ,
  ac_deactivate_unsuccess ( address_type ) ,
  ac_interrogate_success ( address_type , time_type ) ,
  ac_interrogate_unsuccess ( address_type ) ;

signallist ac_invocations =
 ac_activate , ac_deactivate , ac_interrogate ;

signallist ac_notifications =
 ac_activate_success , ac_activate_unsuccess ,
  ac_deactivate_success , ac_deactivate_unsuccess ,
  ac_interrogate_success , ac_interrogate_unsuccess  ;

/* Structure */

block ac_analyser referenced ;

channel user_analyser
 from env to ac_analyser with invoke ;
  from ac_analyser to env with notify ;
 endchannel ;

channel analyser_ac_processor
 from env to ac_analyser with ( ac_notifications ) ;
  from ac_analyser to env with ( ac_invocations ) ;
 endchannel ;
endsystem ;

block ac_analyser ;
process ac_analysis referenced ;

signalroute front1 from env to ac_analysis with invoke ;

signalroute front2 from ac_analysis to env with notify ;

connect user_analyser and front1 , front2 ;
```

```
signalroute ac_back1
 from env to ac_analysis with ( ac_notifications ) ;

signalroute ac_back2
 from ac_analysis to env with ( ac_invocations ) ;

connect analyser_ac_processor and ac_back1 , ac_back2 ;
endblock ;

/* Behaviour */

process ac_analysis ( 1 , max_calls ) ;
 dcl address address_type
     ,
     invocation invocation_type
     ,
     service_code service_code_type
     ,
     time time_type ;
 start ;
 nextstate wait_for_invocation ;
  state wait_for_invocation ;
   input
    invoke ( address , service_code , invocation , time ) ;
    decision service_code ;
     ( ac_service_code ) :
      decision invocation ;
       ( ac_activation ) :
        output ac_activate ( address , time ) ;
         nextstate wait_for_notification ;
       ( ac_deactivation ) :
        output ac_deactivate ( address ) ;
         nextstate wait_for_notification ;
       ( ac_interrogation ) :
        output ac_interrogate ( address ) ;
         nextstate wait_for_notification ;
       else : nextstate wait_for_invocation ; enddecision ;
     else : nextstate wait_for_invocation ; enddecision ;
   endstate wait_for_invocation ;

  state wait_for_notification ;
   input ac_activate_success ( address ) ;
    output
     notify ( address , facilidade_aceite , dummy_time ) ;
     nextstate wait_for_invocation ;

   input ac_activate_unsuccess ( address ) ;
    output
    notify ( address , erro_de_activacao , dummy_time ) ;
    nextstate wait_for_invocation ;

  input ac_deactivate_success ( address ) ;
   output
    notify ( address , facilidade_desactivada , dummy_time
     ) ;
    nextstate wait_for_invocation ;

  input ac_deactivate_unsuccess ( address ) ;
   output
    notify ( address , erro_de_desactivacao , dummy_time ) ;
    nextstate wait_for_invocation ;

  input ac_interrogate_success ( address , time ) ;
   output notify ( address , despertar , time ) ;
    nextstate wait_for_invocation ;

  input ac_interrogate_unsuccess ( address ) ;
   output
    notify ( address , erro_na_consulta , dummy_time ) ;
    nextstate wait_for_invocation ;
  endstate wait_for_notification ;
endprocess ;
```

Appendix E

The LOTOS Specification for the Pilot Case Study.

```
(*   Version : 1.2     *)

specification
        AC_PROCESSOR[AC_ACTIVATION, AC_DEACTIVATION,
        AC_INTERROGATION,
        AC_NOTIFICATION,ADD_USERS, ALARM,TIME] :  noexit

library   Integer, Boolean
endlib

type
        EnrichedInt
is
        Integer
        opns
        1       (*$ userdefined $*):  ->      int
        2       (*$ userdefined $*):  ->      int
        3       (*$ userdefined $*):  ->      int
        4       (*$ userdefined $*):  ->      int
        5       (*$ userdefined $*):  ->      int
        6       (*$ userdefined $*):  ->      int
        7       (*$ userdefined $*):  ->      int
        8       (*$ userdefined $*):  ->      int
        9       (*$ userdefined $*):  ->      int
        10      (*$ userdefined $*):  ->      int
        23      (*$ userdefined $*):  ->      int
        25      (*$ userdefined $*):  ->      int
        59      (*$ userdefined $*):  ->      int
        61      (*$ userdefined $*):  ->      int

eqns
 ofsort int
       1 = succ(0);
       2 = succ(succ(0));
       3 = succ(succ(succ(0)));
       4 = succ(succ(succ(succ(0))));
       5 = succ(succ(succ(succ(succ(0)))));
       6 = succ(succ(succ(succ(succ(succ(0))))));
       7 = succ(succ(succ(succ(succ(succ(succ(0)))))));
       8 = succ(succ(succ(succ(succ(succ(succ(succ(0))))))));
       9 = succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))));
      10 = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))));
```

```
     23 = 10 + 10 + 3;
     25 = 23 + 2 ;
     59 = (25*2) + 9;
     61 = 59+2
endtype

type address_type is EnrichedInt, Boolean
    sorts address_type (*$ rec $*)
    opns
      invalid_address (*$ userdefined $*) : -> address_type
      mk_address (*$ mkrec 1 $*): int -> address_type
      _eq_ (*$ equal $*),
      _ne_ (*$ notequal $*) : address_type, address_type -> Bool
      _eq_ (*$ userdefined $*),
      _ne_ (*$ userdefined $*) : address_type, int -> Bool
    eqns
      forall
        address : address_type,
        n : int
     ofsort address_type
        invalid_address = mk_address(pred(0));
     ofsort bool

        address eq n = address eq mk_address(n);
        address ne n = address ne mk_address(n);

endtype

type time_type is EnrichedInt, Boolean
    sorts time_type (*$ rec $*)
    opns
      valid_time (*$ userdefined $*) : time_type -> Bool
      invalid_time (*$ userdefined $*) : -> time_type
      mk_time (*$ mkrec 2 $*) : int, int -> time_type
      hour_of (*$ sel 1 $*) : time_type -> int
      minute_of (*$ sel 2 $*) : time_type -> int
      _eq_ (*$ equal $*) : time_type, time_type -> Bool

    eqns
      forall times : time_type

    ofsort time_type
      invalid_time = mk_time(pred(0),pred(0));

    ofsort bool

       ( (hour_of(times) ge 0) and (hour_of(times) le 23)
        and (minute_of(times) ge 0) and (minute_of(times) le 59) )
        => valid_time(times) = true;

        (((hour_of(times) le 0) or (hour_of(times) ge 23))
        or ((minute_of(times) le 0) or (minute_of(times) ge 59)))
        => valid_time(times)  = false;

endtype


 type
    AC_NOTIFICATION
 is
   sorts    AC_NOTIFICATION (*$ enum 6 $*)

   opns
      FACILIDADE_ACEITE,
      ERRO_DE_ACTIVACAO,
      FACILIDADE_DESACTIVADA,
      ERRO_DE_DESACTIVACAO,
      DESPERTAR,
      ERRO_NA_CONSULTA:
     -> AC_NOTIFICATION
 endtype

 type
    UserListElement (* LINK : R-component store description
          and C-component Alarm-timer information aspects *)

 is address_type, time_type, Boolean
   sorts    user (*$ rec $*)

   opns
   (* represents the information stored about each user in
       * the Supplementary Services Database, i.e. User Address,
       * hour and minute of alarm call, on/off flag for alarm
       * service and authorization flag for alarm service.       *)
    mk_user (*$ mkrec 4 $*):   address_type, time_type, bool, bool-> user
   (* mk_user(address, time , on_off, authorised)
       * means that the user whose address in the ISDN system
       * is address has its alarm time set for time,
       * its on_off flag is set to the boolean value on_off and
       * its authorization flag set to the boolean value
       * authorised.                                         *)
      address_of (*$ sel 1 $*) : user -> address_type
      alarm_time_of (*$ sel 2 $*): user -> time_type
      on_off_flag_of (*$ sel 3 $*),
      authorization_flag_of (*$ sel 4 $*):
```

```
      user-> bool
       _ eq _  (*$  equal $*),    _ ne _  (*$  notequal $*):
      user, user-> bool
endtype

(* User_list is a list of all users that the Supplementary
      * Services Database is aware of and stores information on.
      * For each user it contains a record with the user's
      * address on the system, the time that their
      * alarm call is set for, a flag to indicate whether the
      * alarm is on or off and a flag to indicate whether or
      * not the user is authorised to use the Alarm Call service *)
type
   UserList
is
   EnrichedInt, UserListElement,address_type, time_type, Boolean
  sorts     user_list (*$ list( user) $*)

  opns
   remove_user (*$ userdefined $*):   address_type, user_list-> user_list
  (* change_user(address,t,o,a,users) changes for the user
       * with address address, time time, on_off
       * flag to o and authorization flag to a. *)
   change_user (*$ userdefined $*):
     address_type, time_type, bool, bool, user_list-> user_list
   make_user (*$ userdefined $*):
     address_type, time_type, bool, bool, user_list-> user_list
   valid_user (*$ userdefined $*):   address_type, user_list-> bool
   alarm_time_of (*$ userdefined $*): address_type, user_list -> time_type
   on_off_flag_of (*$ userdefined $*):   address_type, user_list-> bool
   authorization_flag_of (*$ userdefined $*):  address_type, user_list-> bool
   _eq_ (*$ equal $*), _ne_ (*$ notequal $*): user_list, user_list ->bool
   <> (*$  empty $*):  -> user_list
   mklist (*$  $*):   user-> user_list
   conc (*$  $*):   user_list, user_list-> user_list
   first (*$  $*):   user_list-> user
   rest (*$  $*):   user_list-> user_list
   sublist (*$  $*):   user_list, int, int-> user_list
   length (*$  $*):   user_list-> int

  eqns
    forall
       address1, address2  :  address_type,
       t                   : time_type,
       o, a                :   bool,
       us                  :   user_list

     ofsort user_list
```

```
( us eq <>)  =>
  remove_user (  address1 , us )  =  <> ;

(
  ( us ne <> )  and
  (  address1  eq  address_of (  first (  us ) ) )
  )
  =>
  remove_user (  address1 , us )  =
    remove_user (  address1 , rest (  us ) ) ;

(
  ( us ne <> )  and
  (  address1  ne  address_of (  first (  us ) ) )
  )
  =>
  remove_user (  address1 , us )  =
    conc
     (
       mklist (  first (  us ) ) ,
       remove_user (  address1 , rest (  us ) )
      ) ;


( us eq <> )  =>
  change_user (  address1 , t, o , a , us )  =
   <> ;

(
  ( us ne <>  )  and
  (  address1  eq  address_of (  first (  us ) ) )
  )
  =>
  change_user (  address1 , t, o , a , us )  =
    conc
     (
       mklist
        (  mk_user (  address1 , t, o , a ) )
      ,
       remove_user (  address1 , rest (  us ) )
      ) ;

(
  ( us ne <> )  and
  (  address1  ne  address_of (  first (  us ) ) )
  )
  =>
  change_user (  address1 , t , o , a , us )  =
          conc
           (
              mklist (   first  (   us  ) ) ,
              change_user  (   address1  ,  t ,  o ,  a ,  rest(us)  )
             ) ;


      (  us eq <> )    =>
         make_user  (   address1  ,  t,  o ,  a ,  us  )   =
           mklist
            (   mk_user  (   address1  ,  t ,  o ,  a  ) ) ;

      (  us ne <>)    =>
         make_user  (   address1  ,  t,  o ,  a ,  us  )   =
           conc
            (
               mklist
                (   mk_user  (   address1  ,  t ,  o ,  a  ) )
               ,
               us
              )


ofsort time_type
      (
        (  us ne <> )    and
        (   address1    eq   address_of  (   first  (   us  ) ) )
        )
         =>
         alarm_time_of  (   address1  ,  us  )   =
           alarm_time_of  (   first  (   us  ) ) ;


      (
        (  us ne <> )    and
        (   address1    ne   address_of  (   first  (   us  ) ) )
        )
         =>
         alarm_time_of  (   address1  ,  us  )   =
           alarm_time_of  (   address1  ,  rest  (   us  ) ) ;


ofsort bool
      (  us eq <>  )    =>
         valid_user  (   address1  ,  us  )   =   false ;

      (
        (  us ne <> )    and
```

```
 (   address1    eq   address_of  (    first  (    us  )  )  )
 )
  =>
  valid_user  (    address1 ,  us  )   =    true ;

(
 (  us ne <> )   and
 (   address1    ne   address_of  (    first  (    us  )  )  )
 )
  =>
  valid_user  (    address1 ,  us  )   =
    valid_user  (    address1 ,  rest  (    us  )  ) ;

(   us eq <>  )   =>
  on_off_flag_of  (    address1 ,  us  )   =    false ;

(
 (  us ne <> )   and
 (   address1    eq   address_of  (    first  (    us  )  )  )
 )
  =>
  on_off_flag_of  (    address1 ,  us  )   =
    on_off_flag_of  (    first  (    us  )  ) ;
(
 (  us ne <> )   and
 (   address1    ne   address_of  (    first  (    us  )  )  )
 )
  =>
  on_off_flag_of  (    address1 ,  us  )   =
    on_off_flag_of  (    address1 ,  rest  (    us  )  ) ;

(   us eq <>  )   =>
  authorization_flag_of  (    address1 ,  us  )   =    false ;

(
 (  us ne <> )   and
 (   address1    eq   address_of  (    first  (    us  )  )  )
 )
  =>
  authorization_flag_of  (    address1 ,  us  )   =
    authorization_flag_of  (    first  (    us  )  ) ;

(
 (  us ne <> )   and
 (   address1    ne   address_of  (    first  (    us  )  )  )
 )
  =>
  authorization_flag_of  (    address1 ,  us  )   =
          authorization_flag_of  (   address1  ,  rest  (    us  )  )
 endtype

\end{verbatim}
\begin{verbatim}

 behaviour

  hide
    QUERY_USERS, CHANGE_USERS, NEXT_USER
    in
    START_AC_PROCESSOR[AC_ACTIVATION, AC_DEACTIVATION,
    AC_INTERROGATION, AC_NOTIFICATION, ALARM, TIME,
    QUERY_USERS, CHANGE_USERS, ADD_USERS, NEXT_USER]

    |[QUERY_USERS,CHANGE_USERS, ADD_USERS, NEXT_USER]|

     USERS_MANAGER
     [QUERY_USERS, CHANGE_USERS, ADD_USERS,NEXT_USER] ( <> )

  where

  process USERS_MANAGER [QUERY_USERS, CHANGE_USERS,
       ADD_USERS,NEXT_USER] (users : user_list) : noexit :=
  (
      QUERY_USERS ?    address   :   address_type;
      (
          (  [valid_user(address,users) eq true]->
             (* Query user's details (i.e. alarm time,
                authorization and on_off flags *)
             (
               QUERY_USERS ! alarm_time_of(address,users)
                           !  on_off_flag_of(address,users)
                           !  authorization_flag_of(address,users);
               USERS_MANAGER[QUERY_USERS, CHANGE_USERS,
                           ADD_USERS, NEXT_USER] (users)
             ) (* end valid users eq true *)
            []
            [valid_user(address,users) eq false]->
            (
                QUERY_USERS!false;
                USERS_MANAGER[QUERY_USERS, CHANGE_USERS,
                         ADD_USERS, NEXT_USER](users)
            ) (* end valid users eq  false *)
          ) (* end valid user options *)
      ) (* end query users block *)
      []
      (* Amend a user record *)
    CHANGE_USERS ?address : address_type
    ?alarm_time  :  time_type ? on_off, authorization : bool;
    (
     USERS_MANAGER[ QUERY_USERS, CHANGE_USERS,
     ADD_USERS,NEXT_USER]
     (change_user(address, alarm_time, on_off ,authorization,users))
    )
    []
    (* Add a new user *)
    ADD_USERS ?address : address_type
              ? alarm_time : time_type
              ? on_off, authorization  : bool;
    (
       USERS_MANAGER [QUERY_USERS, CHANGE_USERS,
                   ADD_USERS, NEXT_USER ]
       (make_user(address ,  alarm_time,  on_off ,
                        authorization ,users ) )
    )
    []
    NEXT_USER  ?    user_offset  :   int;
    (* Given the position of a user in the list return their address *)
    (
       let temp_user_list : user_list
          =  sublist(users, user_offset, length(users))
       in
       (
          [temp_user_list ne <>] ->
          (
             NEXT_USER  !address_of(first(temp_user_list));
             exit
          )
          []
          [temp_user_list  eq <> ] ->
          (
             NEXT_USER !invalid_address;
             exit
          )
        )
        >>USERS_MANAGER[QUERY_USERS, CHANGE_USERS,
             ADD_USERS,NEXT_USER] (users) )

 )


endproc

(* LINK : R-component Level 1 DFD AC-Processor and
         C-component AC-Processor *)
```

```
process START_AC_PROCESSOR [AC_ACTIVATION, AC_DEACTIVATION,
        AC_INTERROGATION, AC_NOTIFICATION, ALARM, TIME,
        QUERY_USERS, CHANGE_USERS, ADD_USERS,
        NEXT_USER] : noexit :=
(
    ALARM_ACTIVATION [AC_ACTIVATION, AC_NOTIFICATION,
                            CHANGE_USERS,QUERY_USERS]

    |||

    ALARM_DEACTIVATION [AC_DEACTIVATION, AC_NOTIFICATION,
                            QUERY_USERS, CHANGE_USERS]

    |||

    ALARM_INTERROGATION [AC_INTERROGATION, AC_NOTIFICATION,
                                        QUERY_USERS]

    |||

    ALARM_TIMER [ ALARM, TIME, QUERY_USERS, CHANGE_USERS,
                                        NEXT_USER]
)
endproc


(* LINK : R-component Alarm Activation PSpec and
         R-components System Communication 2 & 3
         and C-component AC-Processor behaviour aspects *)

  process    ALARM_ACTIVATION
    [ AC_ACTIVATION, AC_NOTIFICATION, CHANGE_USERS,
      QUERY_USERS ]
    :  noexit
    :=
  (* ALARM ACTIVATION process accepts the user's address and the time
   * for which the alarm is required on the AC_ACTICATION
   * gate. After checking that the time values are within the
   * required range, and that the user is authorised to use the alarm
   * facility (by reading the value of the authorization flag in the
   * user's database entry) the user's database entry is updated to
   * record the alarm time and the on_off flag is set to true. The user
   * is informed of the success or failure of the operation on the
   * AC_NOTIFICATION gate.
   *)
  (
     AC_ACTIVATION  ? address  :  address_type ? time :time_type;
    ( [valid_time(time) eq false]->
      ( (* User's proposed time is invalid *)
        AC_NOTIFICATION! ERRO_DE_ACTIVACAO !address;
        exit
      ) (* end val time eq false *)
     []
       [valid_time(time) eq true]->
       ( (* time in valid range *)
         QUERY_USERS !address ?stored_time:time_type ?on_off:Bool
                 ?authorization:Bool;
         ( [authorization eq true]-> (* User is authorised *)
            (
              CHANGE_USERS !address !time !true !true;
              AC_NOTIFICATION  !FACILIDADE_ACEITE !address; exit
            ) (* end auth eq true *)
            []
            [authorization eq false]->
            (  (* User is not authorised *)
               AC_NOTIFICATION !ERRO_DE_ACTIVACAO !address; exit
            ) (* auth eq false *)
          ) (* end authorization checks *)
         []
         QUERY_USERS!address !false;
         (  (* User is not valid (i.e. does not exist *)
            AC_NOTIFICATION  !ERRO_DE_ACTIVACAO !address; exit
         ) (* end false *)
       )  (* val time eq true *)
    ) (* end time checks *)
    >>
    ALARM_ACTIVATION[AC_ACTIVATION, AC_NOTIFICATION,
                     CHANGE_USERS, QUERY_USERS]
  )
  endproc


(* LINK : R-component Alarm Deactivation PSpec and
        R-components System Communication 4,5 & 6
        and C-component AC-Processor behaviour aspects *)

 process ALARM_DEACTIVATION [  AC_DEACTIVATION, AC_NOTIFICATION,
                     QUERY_USERS, CHANGE_USERS ] :  noexit :=
  (* ALARM DEACTIVATION process accepts the address of the user who wishes
   * to have their alarm turned off on the AC_DEACTIVATION gate. After
   * checking that the user is authorised to use the Alarm Call facility
   * and that the alarm is turned on, the on_off flag is set to false. The
   * user is informed of the success or failure of the operation on the
   * AC_NOTIFICATION gate.
  *)
(
   AC_DEACTIVATION  ? address  :   address_type;
   (QUERY_USERS !address ?time:time_type ?on_off:Bool
                  ?authorization:Bool;
   (   [(authorization eq true) and (on_off eq true)]->
       (* User is authorised and alarm is set *)
       (
          CHANGE_USERS !address !time !false !true;
          AC_NOTIFICATION !FACILIDADE_DESACTIVADA !address; exit
       )
       []
       [(authorization eq false) or (on_off eq false)]->
       (
          (* User is not authorised or alarm is off*)
          AC_NOTIFICATION  !ERRO_DE_DESACTIVACAO !address; exit
       )
   )
   []
   QUERY_USERS!address !false;
   (
      (* User is not valid (i.e. does not exist *)
      AC_NOTIFICATION  !ERRO_DE_DESACTIVACAO !address;
      exit
   ))
   >>
   ALARM_DEACTIVATION  [ AC_DEACTIVATION, AC_NOTIFICATION,
                         QUERY_USERS, CHANGE_USERS ]
)
endproc


(* LINK : R-component Alarm Interrogation PSpec and
        R-components System Communication 7,8 & 9
        and C-component AC-Processor behaviour aspects *)

process ALARM_INTERROGATION [ AC_INTERROGATION, AC_NOTIFICATION,
                              QUERY_USERS ] :  noexit :=
  (* ALARM INTERROGATION process accepts the address of the user who wishes
   * to query their alarm time on the AC_INTERROGATION gate. After checking
   * that the user is authorised to use the Alarm Call service and that the
   * alarm is turned on the process returns a string containing the
   * hour and minute that the alarm has been set to on the AC_NOTIFICATION
   * gate (or a message indicating that the operation failed).
   *)

(
   AC_INTERROGATION  ? address  :   address_type;
  (QUERY_USERS !address ?time:time_type ?on_off:Bool
                     ?authorization:Bool;
  (    [(authorization eq true) and (on_off eq true)]->
              (* User is authorised and alarm is set *)
       (
          AC_NOTIFICATION !DESPERTAR !time !address;
          exit
       )
       []
       [(authorization eq false) or (on_off eq false)]->
       (   (* User is not authorised or alarm is off*)
          AC_NOTIFICATION !    ERRO_NA_CONSULTA !address;
          exit
       )
  )
  []
  QUERY_USERS!address !false;
  (    (* User is not valid (i.e. does not exist *)
      AC_NOTIFICATION !    ERRO_NA_CONSULTA  !address;
      exit
  ))
  >>
  ALARM_INTERROGATION [ AC_INTERROGATION,
                    AC_NOTIFICATION, QUERY_USERS]
)
endproc

(* LINK : R-components System Communication 10 & 11
        and C-component Alarm-Timer *)

process  ALARM_TIMER [ ALARM, TIME, QUERY_USERS,
                  CHANGE_USERS, NEXT_USER]:noexit :=
  (* ALARM_TIMER receives the current time on the
   * TIME gate. It calls the CHECK_FOR_ALARM process to see if any
   * user has the current time set as its alarm time. If such a user
   * exists their adddress is written on the ALARM gate.
   *)
  (
     TIME  ?current_time:time_type;
     CHECK_FOR_ALARM[ALARM, QUERY_USERS, CHANGE_USERS,
                      NEXT_USER] ( current_time, 1  )
     >> ALARM_TIMER[ALARM, TIME, QUERY_USERS,
                      CHANGE_USERS, NEXT_USER]
  )

where

   process CHECK_FOR_ALARM[ALARM, QUERY_USERS,
                                    CHANGE_USERS, NEXT_USER]
          (current_time : time_type, user_offset : Int): exit :=
        (* CHECK_FOR_ALARM receives the current time and
         * the list of user records. If the user list is empty then the process
         * exits giving a user address of zero and a boolean value of false.
         * If there is a match between the inputted time and the time to which
         * a user has their alarm set, then the process exits, returning
         * that user's address and a boolean value of true. Otherwise the process
         * recursively calls itself, passing the time and the rest of the list
         * of users' records as parameters.
         *)
         (
            NEXT_USER  !   user_offset ? address  :   address_type;
           ([address ne invalid_address]->
              (
                QUERY_USERS!address ?time:time_type ?on_off:Bool
                       ?authorization:Bool;

                ([(time eq current_time) and (on_off eq true)
                                and (authorization eq true)]->
                  (        (* User's alarm is on *)
                    ALARM  !address;
                    CHANGE_USERS  !address !time !false !true; exit
                    (* Turn on/off flag off after raising alarm *)

                  )  (* end time eq current etc. *)
                 ) (* end time checks etc. *)
                 >>
                 CHECK_FOR_ALARM [ALARM, QUERY_USERS,
                                 CHANGE_USERS, NEXT_USER]
                              (current_time ,succ(user_offset))
               ) (* address ne invalid *)
               []
               [address eq invalid_address]->
               (    (* Finished searching list *)
                    exit
               ) (* address eq valid *)
            ) (* end address options *)
         )  (* end process *)
         endproc
      endproc

    endspec
    ^Z
```

# Appendix F

## The ICL specification for the Pilot Case Study

```
system    call_control_coordinator

 [   ALARM, TIME, ADD_USERS   ]


{
  inchannel user_analyser ;
  outchannel user_analyser;
  outsignal notify ;
  insignal invoke

}

using
 /*  System Identifier for the LOTOS   */
  AC_PROCESSOR,
 /*  System Identifier for the SDL     */
  ANALYSER

where

  signal
     ANALYSER  .  AC_ACTIVATE
     (*)  via   ANALYSER  .  ANALYSER_AC_PROCESSOR
    ->
    action    AC_PROCESSOR  .  AC_Activation   !* ;

  signal
     ANALYSER  .  AC_DEACTIVATE
     (*)  via   ANALYSER  .  ANALYSER_AC_PROCESSOR
    ->
    action    AC_PROCESSOR  .  AC_Deactivation   !* ;

  signal
     ANALYSER  .  AC_INTERROGATE
     (*)  via   ANALYSER  .  ANALYSER_AC_PROCESSOR
    ->
    action    AC_PROCESSOR  .  AC_Interrogation   !* ;

  action    AC_PROCESSOR  .  AC_Notification
    !  FACILIDADE_ACEITE   !*
    ->
  signal
     ANALYSER  .   AC_ACTIVATE_SUCCESS
     (*)   via   ANALYSER  .   ANALYSER_AC_PROCESSOR;

action    AC_PROCESSOR  .   AC_Notification
  !   ERRO_DE_ACTIVACAO    !*
  ->
  signal
     ANALYSER  .   AC_ACTIVATE_UNSUCCESS
     (*)   via   ANALYSER  .   ANALYSER_AC_PROCESSOR;

action    AC_PROCESSOR  .   AC_NOTIFICATION
  !   FACILIDADE_DESACTIVADA    !*
  ->
  signal
     ANALYSER  .   AC_DEACTIVATE_SUCCESS
     (*)   via   ANALYSER  .   ANALYSER_AC_PROCESSOR;

action    AC_PROCESSOR  .   AC_NOTIFICATION
  !   ERRO_DE_DESACTIVACAO    !*
  ->
  signal
     ANALYSER  .   AC_DEACTIVATE_UNSUCCESS
     (*)   via   ANALYSER  .   ANALYSER_AC_PROCESSOR;

action    AC_PROCESSOR  .   AC_NOTIFICATION
  !   DESPERTAR    !*
  ->
  signal
     ANALYSER  .   AC_INTERROGATE_SUCCESS
     (*)   via   ANALYSER  .   ANALYSER_AC_PROCESSOR;

action    AC_PROCESSOR  .   AC_NOTIFICATION
  !   ERRO_NA_CONSULTA    !*
  ->
  signal
     ANALYSER  .   AC_INTERROGATE_UNSUCCESS
     (*)   via   ANALYSER  .   ANALYSER_AC_PROCESSOR;


/*  LOTOS gates that are 'directly connected' to the
        border of the system   */

action   CALL_CONTROL_COORDINATOR  .   ALARM,
         AC_PROCESSOR  .   ALARM !* ;

action   CALL_CONTROL_COORDINATOR  .   TIME,
         AC_PROCESSOR  .   TIME  !* ;
```

```
/* SDL channels that are directly connected to the
        border of the system.
        There are two messages on this channel.
        Invocation and notification Segment.   */


/* signal from the user to the analyser */

signal
   CALL_CONTROL_COORDINATOR  .   INVOKE
     (*) via  CALL_CONTROL_COORDINATOR . USER_ANALYSER
  ->
signal ANALYSER  .   INVOKE
     (*) via  ANALYSER  .  USER_ANALYSER ;



/* signal from the analyser to the user */
signal
   ANALYSER  .   NOTIFY
   (*)  via  ANALYSER . USER_ANALYSER
 ->
   signal CALL_CONTROL_COORDINATOR .  NOTIFY
   (*)  via  CALL_CONTROL_COORDINATOR . USER_ANALYSER ;


endsystem
```