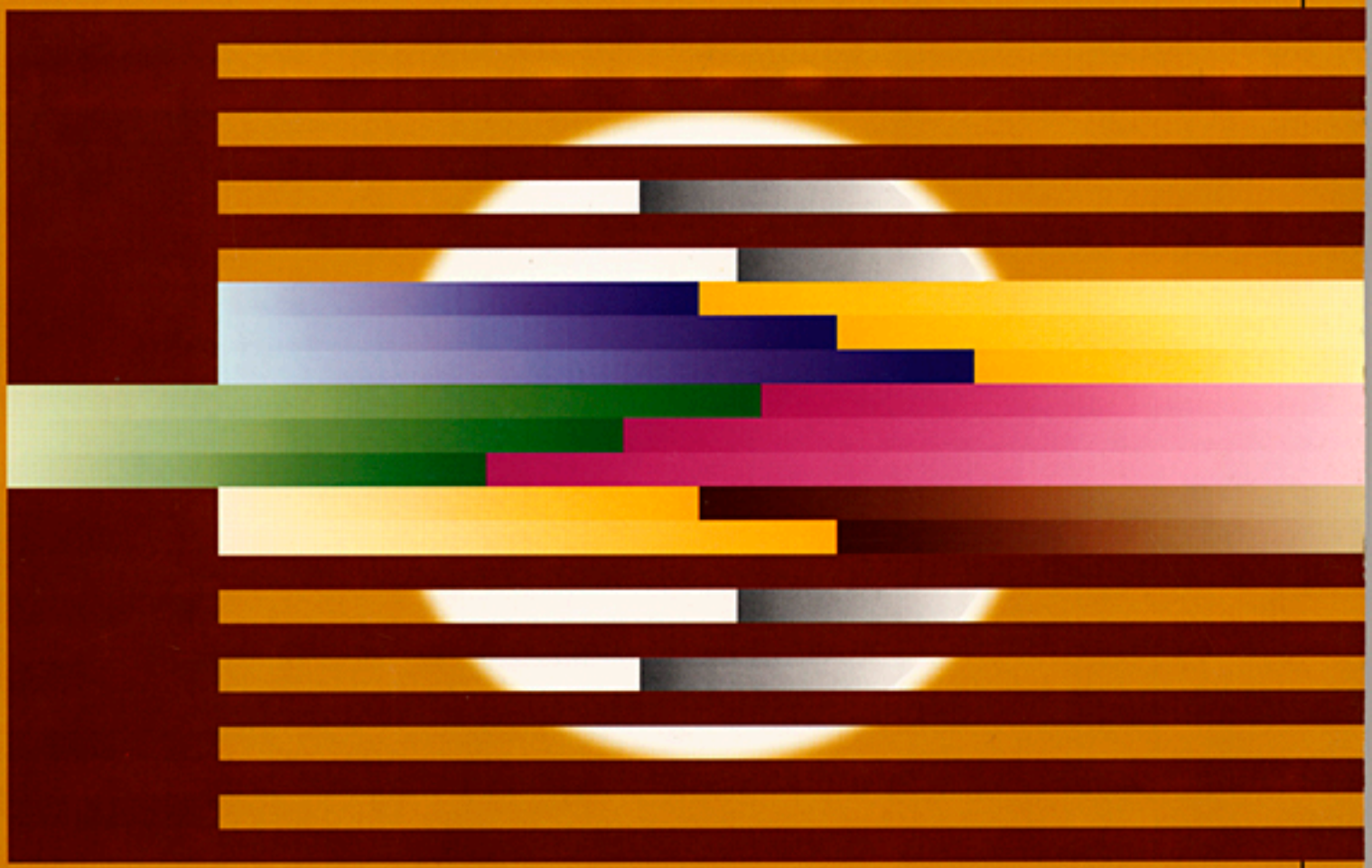
Telektronikk 2/3.93

Information Systems



Contents

Telektronikk

Volume 89 No. 2/3 - 1993
ISSN 0085-7130

Editor:
Ola Espvik
Tel. + 47 63 80 98 83

Feature editor:
Arve Meisingset
Tel. + 47 63 80 91 15

Editorial assistant:
Gunhild Luke
Tel. + 47 63 80 91 52

Editorial office:
Telektronikk
Norwegian Telecom Research
P.O. Box 83
N-2007 Kjeller, Norway

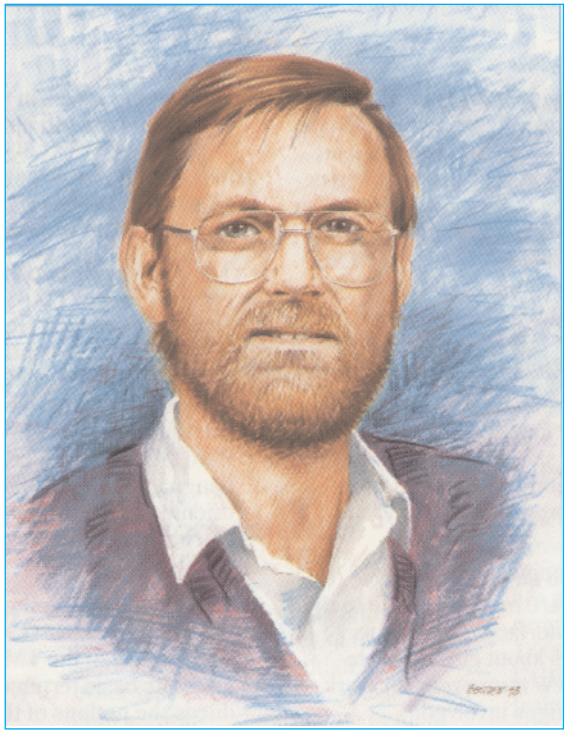
Editorial board:
Ole P Håkonsen, Senior Executive Vice President
Karl Klingsheim, Vice President, Research
Bjørn Løken, Vice President, Market and Product Strategies

Graphic design:
Design Consult AS

Layout and illustrations:
Gunhild Luke, Britt Kjus, Åse Aardal
Norwegian Telecom Research

Guest editorial

BY TOR M JANSEN



This issue of Telektronikk is concerned with software for telecommunications applications. Since the early sixties, it has been clear to all involved in both the computer industry and the telecommunications industry that computer technology will inevitably become the basis of telecommunications equipment and services. What was not foreseen was the enormous challenge of making this happen.

There are three main areas where computer technology is now applied. The first is in the network elements. The second area is operation and maintenance of the network and the services. Thirdly, customer support and customer service is an area of great competitive concern.

In the very early days of telecommunications, the limited number of subscribers and the limited number and simplicity of the services, i.e. telephone and telegraph, made it possible for an operator to handle most problems that emerged, such as network management and special services to the customer. The size of the operation was such that it could easily be handled manually. Today, in a world with many and complex and evolving services, hundreds of millions of customers, booming traffic, and fiercely increasing competition in new and old markets, the situation has changed dramatically. Computer technology, especially software, at present seems to be the only way to cope with the problems and challenges.

The network itself is a good example of the role of computers and software. Previously, all functions relevant to communications resided inside the network elements, be it for maintenance or service provision. The network elements, i.e. switches and transmission equipment, limited by their design, controlled what could be offered to the customer. The cost and lifetime of telecom systems were such that introduction of new services and improvement of old services by necessity had to be slow. The first “improvements” came with digital switching and digital transmission in the sixties and seventies. When computer programmers tried to design communications systems, and telecom engineers tried to program computers, the results were invariably bad. After these first experiences, one realised that the problem area had to be approached much more systematically and proactively.

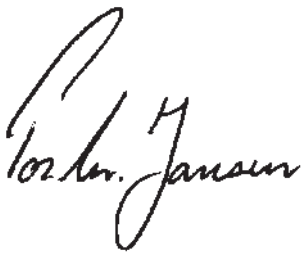
It is often said that in the future, there will be fewer telecom equipment and systems suppliers world-wide than the scarce dozen we have today, because of the cost and complexity of developing new generations of systems. My view is different. With increased standardisation, the hardware will be bought from vendors who can manufacture most effectively, and thus sell to the lowest prices. The same will happen to basic software, which will be modularised with standard interfaces. This will lead to the same situation we have in the computer industry today, with hundreds and hundreds of competing and, relatively speaking, small companies. The challenge is to plan the network and implement the services in a cost effective way. We will at the same time see more international co-operation, and also a more competitive and closed attitude of the actors in the business.

For Norwegian Telecom Research, it is important to have a long term perspective, a medium term view, and also produce results that are useful in the present situation. This means that we have to watch the whole area of informatics, but concentrate our work in selected areas.

Presently, we give special attention to

- Strategies for the use of information technology
- Languages, methods, and tools, both for system design and human interface design
- Database technology, especially aspects of real time performance, reliability, ease of application design, distribution and maintenance
- Standardisation of methods and languages for the use in Telecommunication Management Network (TMN) and also harmonisation with service creation and provisioning (IN)
- Implementation of computer based systems for optimal network planning.

As a conclusion to this introduction, the importance of being competent in informatics is becoming more and more obvious. This issue addresses fields that are being worked on by Norwegian Telecom Research now. However, we are aiming at a moving target. Awareness of the main areas of importance is essential, and future needs will be the guideline for the prioritising.



Overview

BY ARVE MEISINGSET

This issue of Telektronikk will provide both *basic and new knowledge* about information systems. We bring knowledge which we believe is important, but we do not aspire to give a complete or representative overview of all relevant topics.

The magazine is organised into two sections; an overview section introducing central topics, and a technical section going into more detail about various topics. The numbers used refer to the article numbers in the contents list.

The *overview section* addresses the following issues:

- (1) In the first article, 'the father of CHILL', Kristen Rekdahl, puts software development for telecommunication into perspective. He surveys the status and history of the CHILL programming language and gives a short introduction to the language and its programming environment.
- (2) Software is only valuable in so far as it is valuable to its users. The software and data are made available to the users via the Human-Machine Interface (HMI). The author puts forward some doubts about current metaphors for HMIs, he presents a new framework for work on HMIs and outlines the challenges involved.
- (3) Even if current software systems are becoming very large, current development techniques are mainly addressing programming on a small scale rather than programming of very large systems. In fact, the area of architecture for information systems is poorly developed. Therefore, we present this introductory article on reference models for software systems.
- (4) At the core of automatic information systems lies the notion of formal languages. Formal languages are used to instruct the computer what to do, they define the interfaces to the users, other subsystems and the outside world. Formal languages are also used to specify – maybe in a non-executable way – the entire information system. The article presents the development from early programming languages to modern specification languages and provides some examples of these.
- (5) Programming language theory has traditionally been concerned with how to organise and state programs to be efficiently executed in the computer. However, new information systems and their development and maintenance will become increasingly data centred. Therefore, we present this introductory article to database theory. Also, current development and some challenges are outlined.
- (6) How to carry out and control software development is addressed by development methods and life-cycle models. Different views on the software development process, characteristics of conventional and non-conventional software development and interests to be supported, are some of the issues in this overview article on software development methods and life-cycle models.

The *technical section* of this magazine goes into more technical detail than the articles of the overview section. The technical section is divided into three subsections:

- The first subsection presents work on *formal languages going on in the ITU* – the International Telecommunication Union – Study Group 10 (Languages and Methods). (7) The first article provides a theoretical background for and extensions to the HMI reference model. (8) The second article introduces and explains the formalism of the Data Oriented HMI Specification Technique. (9) The third article introduces the Specification and Description Language, SDL. (10) The fourth article presents its object oriented extension OSDL. We are proud to observe that Norway has given key contributions to many areas of SG10 work.
- The second subsection presents *Telecommunications Network Management* (TMN). (11) The first article provides an overview of the TMN functional architecture and introduces its basic terminology. (12) The second article introduces the notions of the OSI (Open Systems Interconnection) Management information model. (13) The third article provides an overview of most network support systems in Norwegian Telecom. (14) The last article presents key database applications for central network management in Norwegian Telecom. TMN recommendations are becoming increasingly important and are expected to gradually change the outlook of support systems used by the operators.
- The third subsection presents some *tools for database application development and usage* in Norwegian Telecom. (15) The first article presents the Datran and Diman tools for application development and graphical presentations. (16) The second article presents the Dibas tool for distributed database management. (17) The third article presents a specification of (a database application for advanced) Access Control Administration; this exemplifies the usage of the HMI formalism and provides some discussion of alternatives.

We hope that the information provided in this issue of Telektronikk is found to be valuable to its readers, and encourage readers to give feedback to the editor of the magazine.

CHILL – the International Standard Language for Telecommunications Programming

BY KRISTEN REKDAL

681.3.04

Abstract

Computer controlled telecommunication systems have proven to contain some of the largest and most complex pieces of software ever constructed. To master this complexity requires the use of powerful methods and tools. The CCITT High Level programming Language – CHILL – is an enabling technology that has contributed to making such systems possible. This paper surveys the status and history of CHILL and gives a short introduction to the language. The Norwegian and Nordic activities supporting the CCITT work are described. Finally the development of the CHIPSY toolset for CHILL programming is summarised.

1 Software is important in telecom

With the increasing use of software in telecommunication systems, programming has become an important enabling technology for telecommunication services. The proportion of software cost over hardware cost in the development and production of telecom systems has been shifting rapidly. In 1970 the software cost was practically 0 % of the total, in 1980 it was around 50 %, while presently it has risen to around 85 %. More and more the services earning money for the operating companies are implemented in software.

This shift towards software in telecom was realised by some pioneers already in the late 1960's when the CCITT made initial investigations into the impact of computerised telephone exchanges. However, even when work on the CHILL language started in 1975, nobody dared to predict the present extent and pervasiveness of software in telecom systems.

Most of the basic technical problems to be faced in telecom programming were already at that time fairly well understood, and programming languages had been developed or were being developed to cope with these problems. One may therefore ask why CCITT should engage in making yet another programming language, adding to the already abundant flora of such languages. There were three major reasons:

- to remedy weaknesses of existing languages when applied to telecom systems, because such systems have special needs
- to consolidate, in one language, features only found in a variety of other languages
- to provide one standard language covering all producers and users of computerised telecom systems.

The standardisation aspect was motivated by the fact that the telecommunications community was a very large one and distributed all over the world. Most producers and operators of such equipment would somehow come into contact with software. Thus, the benefits of standardisation also in the field of software were likely to be substantial.

2 Telecom software is special

Experience has clearly taught us that there are many features of telecom software which differentiate it from other types of software. Typically telecom software exhibits some or all of the following characteristics:

- Real-time, concurrent behaviour, time critical response
- Very large size, e.g. 100,000 – 10,000,000 lines of source code
- High complexity
- High performance requirements
- High quality requirements, e.g. reliability, availability, fault tolerance
- Use of multiple, heterogeneous, networked computers, e.g. host and target computers
- Long lifetime, e.g. 3 – 20 years
- Continuous development during the lifetime, spanning several technology generations in both software and hardware
- Large development teams, the same personnel rarely stays with the system for its lifetime
- No single person has full understanding of the whole system
- Geographic separation of development activities
- Delivered in many copies in several variants and generations
- Total lifetime cost much higher than initial development cost
- Strong influence from international standards, e.g. CCITT, ISO.

CHILL was designed to meet the challenges of such software. In retrospect it was clearly justified to develop a special language for telecom programming.

3 CHILL is a viable language

Since its inception in 1975, CHILL has grown to become a major programming language within telecom. There are now at least 12,000 – 15,000 CHILL programmers in the world. More than 1,000 man-years have been invested in CHILL compilers, support tools and training. Many of the most successful telecom switching systems on the world market have been engineered in CHILL. See Table 1.

Two other ways of illustrating the extent of CHILL usage are shown in the graphs below. Figure 1 shows that six of the top ten world telecom equipment manufacturers are using CHILL for their major switching products. The manufacturers are shown according to turnover in 1989 for telecom products. The major CHILL users are shown in black columns. Non-CHILL users are shown in white columns. Partial users are shown in shaded columns.

Figure 2 shows which programming languages are actually most used in the telecom industry for public switching systems. The volume measure is the percentage of the total number of installed public digital local lines per year, for the years 1985 to 1990. The figure for 1990 is an estimate. The graph shows that the usage of CHILL has been steadily increasing. 45 % of the digital local lines installed in 1990 were supported by software written in CHILL. This is up from 0 % in 1980. CHILL is the only programming language common to more than one of the major public telecom switching systems. The CHILL column is the sum of the installed lines for EWSD, E10, D70, and System 12.

The second most used telecom programming language is Protel used by Northern Telecom, while C, used by AT&T, is in third place.

Table 1 Telecommunication switching systems programmed in CHILL

System name	System type	Manufacturer	Country
EWSD	Public exchange	Siemens	Germany
PRXD	Public exchange	Philips	Netherlands
System 12	Public exchange	Alcatel	Belgium/Germany
E10	Public exchange	Alcatel	France
D70	Public exchange	NTT, NEC, Hitachi, Fujitsu, Oki	Japan
KBD70	Public exchange	Oki Electric	Japan
LINEA UT	Public exchange	Italtel	Italy
RN64	Cross connect	Telettra	Italy
	PABX	PKI	Germany
TDX-10	Public exchange	Daewoo Telecom	Korea
Tropico	Public exchange	Telebras	Brazil
PXAJ-500/2000	Public exchange	10th Research Institute	China
Levent	Rural exchange	Teletas	Turkey
Nodal Switch	PABX	Alcatel Telecom	Norway
HICOM	PABX	Siemens	Germany
Saturn	PABX	Siemens	USA
Amanda	PABX	Alcatel	Austria
Sopho	PABX	Philips	Netherlands
Focus	PABX	Fujitsu	Japan
TTCF	Teletex	Ascom Hasler	Switzerland

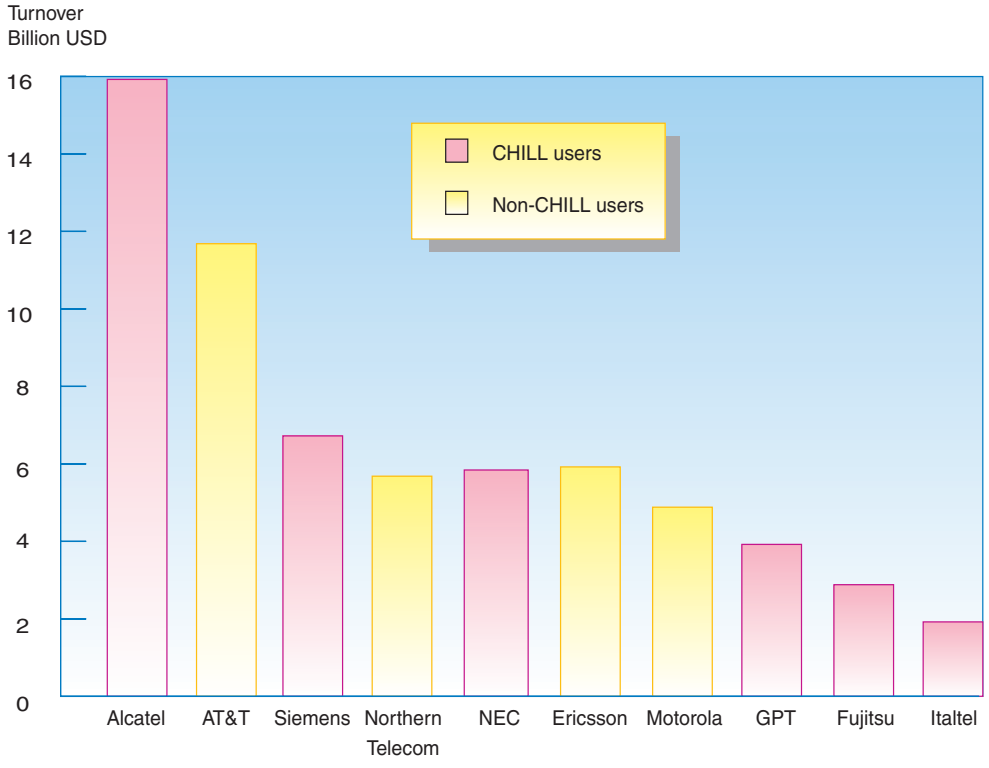


Figure 1 CHILL usage among major telecom manufacturers

4 CHILL development started in 1975

The groundwork for the development of a high-level telecom programming language started already in 1966. The CCITT plenary assembly in 1968 in Mar del Plata, Argentina, decided that such development should be handled by the CCITT. The necessity of standardising a language of this type for switching functions was recognised, as was the need for a unified language for both delivery and technical specifications.

In the study period between 1968 and 1972 the CCITT mainly concerned itself with the definition of the material to which the standards were to apply. Recommendations for software specifications were looked for, as were unified descriptions of software-controlled systems (or SPC – Stored Program Control – systems, as they were called at the time) so that the different telephone switching systems could more easily be compared with each other.

The new language to be developed was required to be easy to read and learn while at the same time being open-ended and system-independent with regard to technical innovations.

The outcome of the discussions was that it was deemed necessary to standardise not only one, but three types of language:

- a *specification and description language,* later to become SDL, for defining descriptions of features, specifications and system designs
- a *high-level programming language,* later to become CHILL, for actual coding, and also
- a *man-machine command language,* later to become MML, for operation and maintenance by the user of SPC switching systems.

In the 1973-1976 Study Period, specific work was started to create this family of languages for SPC use, CHILL, SDL and MML. SDL stands for the CCITT Specification and Description Language (2). MML stands for the CCITT Man-Machine Language (3).

As the responsibility of CCITT was confined to public (telephone) switching systems at the time, it is important to note that the scope of the CCITT standards, including the languages, was confined to that application domain.

The work of CCITT in 1973 started with an investigation and evaluation of 27 existing languages. From this set a short-list of six languages was made. They were:

- DPL, made by NTT, Japan
- ESPL1, made by ITT (now Alcatel), USA and Belgium
- Mary, made by SINTEF/RUNIT, Norway
- PAPE, made by France Telecom/ CNET
- PLEX, made by Ericsson, Sweden
- RTL2, made by the University of Essex, UK.

The conclusion of this study was, however, that none of the languages were satisfactory for the intended application area. In 1975 an ad hoc group of eight people called "The Team of Specialists" was formed to handle the development of a new language. The team had representatives from

- Philips, Netherlands
- NTT, Japan
- Nordic telecom administrations
- Siemens, Germany
- Ellemtel, Sweden
- ITT (now Alcatel), USA
- British Telecom
- Swiss PTT.

A preliminary proposal for a new language was ready in 1976. The language was named *CHILL* – the CCITT High Level Language.

In the following Study Period, starting in 1977, it was decided that there was a need for an evaluation of this proposal by practical experience in order to complete the language. For this purpose the Team of Specialists was replaced by "The Implementors Forum". Its task was to encourage and gather experience from trial implementations of CHILL. By the end of 1979 a language proposal was completed. The CCITT Plenary Assembly approved the CHILL definition in November 1980 to become CCITT Recommendation Z.200 (1).

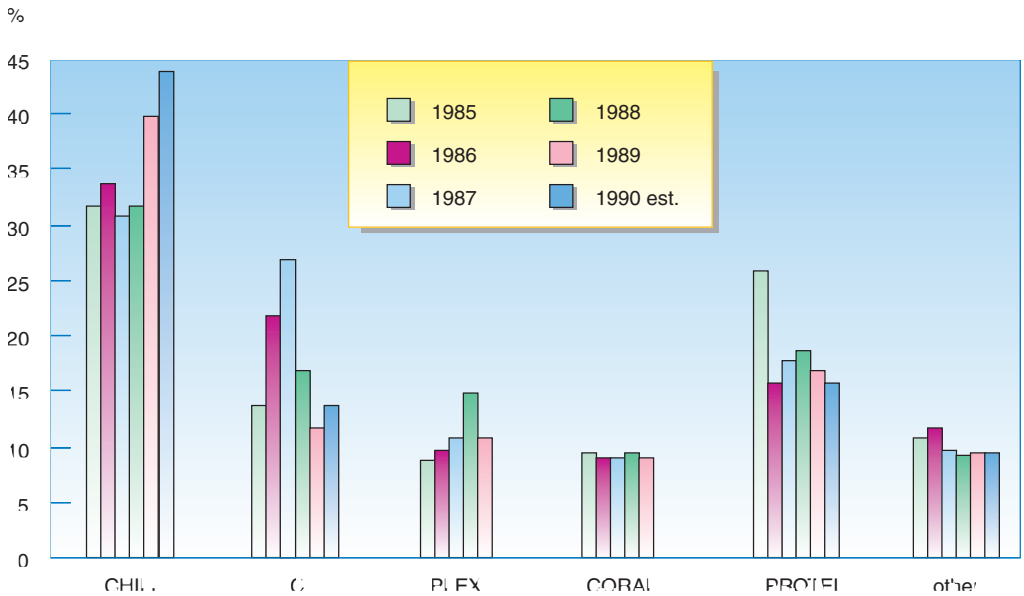


Figure 2 CHILL usage in terms of installed digital local lines per year

CHILL inherited most of its traits form other high-level programming languages, drawing upon the best of the language developments of the 1970's and 1980's. CHILL became a truly state-of-the-art language. Box one below shows the essential groups of constructs in CHILL while box two shows a program fragment giving a flavour of the appearance of the language.

From 1977 many companies and organisations took up the challenge of constructing CHILL compilers. Until 1987 more than 30 compilers had been made for many different computers. Several of the compilers were put to use in industrial product development. Most of the systems listed in Table 1 were programmed by means of compilers developed in-house.

Since 1980 the CCITT has continued to support and maintain CHILL and upgrade the language according to needs from the industry and new technical developments. The main extensions to the language have been:

1981-84	Piecewise programming, name qualification, input/ output
1985-88	Time values and timing operations
1991-92	Data objects for real numbers
1993-96	Work on object orientation is planned.

In 1989 CHILL was recognised as ISO/IEC Standard 9496.

5 Norwegian Telecom was actively involved

Norwegian Telecom had the foresight to enter into the CCITT language work at an early point in time. When the CCITT had decided, in late 1974, that a new language had to be developed, Norwegian Telecom Research (NTR) took an interest to participate actively. NTR managed to enlist the telecom administrations of Denmark, Finland, Norway, and Sweden in a concerted Nordic effort to sponsor an expert from the SINTEF research institute to participate in the CHILL Team of Specialists.

This Nordic representative became the vice chairman of the group from 1976 to 1980 and the chairman of CHILL development from 1980 to 1984. Thus the Nordic effort in this development was highly visible.

In addition it was decided to establish a Nordic support group covering the activities of all the three CCITT languages. This group had representatives from the four countries mentioned above plus Iceland. Later also British Telecom joined. The group was very active and met several times a year from 1975 to 1984.

6 CHIPSY – the CHILL Integrated Programming System

One specific outcome initiated by the Nordic co-operation on CHILL has been *CHIPSY* – the CHILL Integrated Pro-

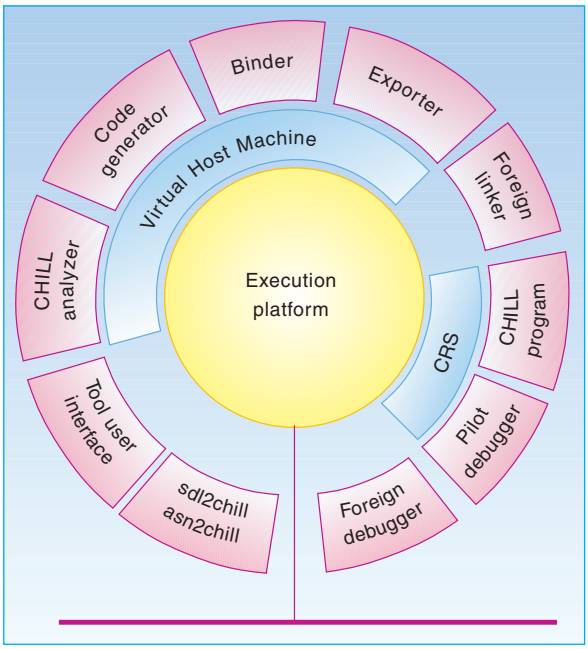


Figure 3 The CHIPSY concept

Box one – Sample CHILL program

```
line_allocator:
MODULE
    SEIZE line_process, line, occupied, unoccupied,
        search,connect, accepted;
    GRANT line_allocator_process;

    line_allocator_process:
    PROCESS ();
        NEWMODE states = SET(free, busy);
        DCL next_state states := free, lno INT := 0;
        line(lno) := START line_process(lno);
        DO FOR EVER;
            CASE next_state OF
            (free): RECEIVE CASE SET sender;
                (occupied):
                    next_state := busy;
                (search):
                    SEND connect(sender) TO line(lno);
                    SEND accepted TO sender;
                    next_state := busy;
                (else): -- Consume any other signal
                ESAC;
            (busy): RECEIVE CASE SET sender;
                (unoccupied):
                    next_state := free;
                (search):
                    SEND rejected TO sender;
                (else): -- Consume any other signal
                ESAC;
            ESAC;
        OD;
    END
    line_allocator_process;
END
line_allocator;
```

gramming System. CHIPSY is an open-ended support environment for programming in CHILL. CHIPSY is oriented towards the development and maintenance of real-time software, especially in the area of telecommunications.

CHIPSY is a family of productivity tools for telecom software development. The toolset comprises compilation tools, associated real-time operating systems and source level test and debug tools. See Figure 3.

The CHIPSY tools cover the design (partly), programming, execution and testing phases of the software development cycle.

Where convenient and useful, CHIPSY interfaces to products or components produced by other companies, e.g. the text editor Emacs (7) or the SDL tool SDT made by TeleLOGIC Malmö AB (6). CHIPSY has been in regular use for the development of major industrial telecom software since 1980, comprising several million lines of CHILL source code. CHIPSY's reliability has been proven by the fact that there are now in operation several hundred digital switching units world wide programmed with CHIPSY. Several units are operating in high load, mission critical (e.g. military) applications.

6.1 Development of CHIPSY since 1977

The CHIPSY development started in early 1977 when the telecom administrations of Denmark, Finland, Norway and Sweden decided to sponsor a CHILL compiler implementation project at SINTEF. Later also British Telecom (now BT) joined in funding the project.

All the development work was carried out by SINTEF until 1984. Then CHIPSY was taken out of SINTEF to establish a new company, later called KVATRO A/S, for the purpose of commercial exploitation of the research results.

The major milestones in the history of CHIPSY have been:

1974 The Mary language was shortlisted by CCITT as one of six candidate languages for a CCITT telecom programming language. Mary is a machine oriented high level programming language developed by SINTEF from 1971 to 1973.

1975 SINTEF is sponsored by the Nordic administrations to join the CCITT Team of Specialists to design the CHILL language.

1977 Start of CHIPSY compiler development project hosted on the ND-100 16-bit minicomputer, initially targeted to the Ericsson APZ210 switching processor, later changed to Intel 8086.

1980 First industrial contract. CHIPSY licenses were sold to Alcatel Telecom Norway for use in developing a digital military communication system.

1982 CHIPSY was sold to Ascom Hasler AG. Hasler signs a contract with SINTEF to implement a CHILL level debugger called CHILLscope.

1983 Code generators implemented for the bare Intel 80286 microprocessor and the ND-100 computer with the SINTRAN III operating system. A new, portable CRS – CHIPSY Real-time Operating System – written in CHILL for both the 80286 and the ND-100/SINTRAN III.

1984 CHIPSY sold to 10th Research Institute of the Ministry of Posts and Telecommunications of China. A new company, later known as KVATRO A/S, was founded in order to commercialise CHIPSY. KVATRO was given the rights to market and further develop CHIPSY by the Nordic telecom administrations.

1985 Rehosting of CHIPSY cross compilers to VAX/VMS was completed.

1978 CHIPSY native compiler and new generation CRS ready for VAX/VMS.

1990 Completion of rehosting and retargeting to several UNIX platforms. CHIPSY becomes the first CHILL compiler available on a 386 PC platform. CHIPSY licenses sold to Nippon Telegraph and Telephone Software Laboratory.

1991 Extending CHIPSY to cover distributed processing. CHIPSY becomes the first CHILL compiler available on a laptop computer.

1992 Implementation of an ASN.1 to CHILL translator completed. Implementation of an SDL to CHILL translator completed. First implementation of the Pilot debugger for testing and debugging in distributed real-time systems.

1993 Full implementation of the Pilot debugger. CHIPSY is ported to SPARC workstations.

6.2 CHIPSY is the outcome of a co-operative effort

Until 1993 more than 100 million NOK (15 million USD) have been invested in the development and maintenance of CHIPSY. The investments constitute co-operative efforts with various partners. The main contributors to this development have been:

- Telecom administrations of Denmark, Finland, Norway, Sweden and United Kingdom acting in concert
- Telecom Norway
- Alcatel Telecom Norway
- Ascom Hasler
- SINTEF
- 10th Research Institute of MPT, China
- Norwegian Industrial Fund
- Nippon Telegraph and Telephone
- KVATRO A/S.

7 Conclusions

CHILL is unique because it is a *standardised* programming language catering for the needs of the telecommunications community. It has already gained a solid acceptance within the industry world wide, and has been used for the construction of some of the world's most successful switching systems.

Because of the longevity of telecom systems and the persistence of programming languages, CHILL will continue to have a strong impact in the world of telecommunications well beyond the year 2000.

It is safe to say that CHILL has largely achieved its original objective of becoming a standard language for the programming of public telecom switching systems.

Box two – CHILL Overview

CHILL is a strongly typed, block structured language. A CHILL program essentially consists of data objects, actions performed upon the data objects and description of program structure.

Data modes

The categories of data modes (data types) provided in CHILL are discrete modes, real modes, powerset modes, reference modes, composite modes, procedure modes, instance modes, synchronisation modes, input-output modes and timing modes. Some modes may have run-time defined parameters. In addition to the CHILL standard modes, a user may define other modes.

Data objects

The data objects of CHILL are values and locations (variables). Locations may be created by declaration or by built-in storage allocators. The properties of locations and values are language defined to a detailed level and subject to extensive consistency checking in a given context.

Sequential actions

Actions constitute the algorithmic part of a CHILL program. To control the sequential action flow, CHILL provides if action, case action, do action, exit action, goto action, and cause action. Expressions are formed from operators and operands. The assignment action stores a value into one or more locations. Procedure call, result and return actions are provided.

Concurrent execution

CHILL allows for concurrent execution of program units (processes). Creation and termination of processes are controlled by the start and stop actions. Multiple processes may execute concurrently. Events, buffers and signals are provided for synchronisation and communication. To control the concurrent action flow, CHILL provides the start, stop, delay, continue, send, delay case and receive case actions, and receive and start expressions.

Exception handling

During run-time the violation of a dynamic condition causes an exception. When an exception occurs the control is transferred to an associated user-defined handler.

Time supervision

Time supervision facilities of CHILL provide means to react to the elapsed time of the external world.

Input and output

The input and output facilities of CHILL provide the means to communicate with a variety of devices of the outside world.

Program structure

The program structuring statements are the begin-end block, module, procedure, process and region. These statements provide the means for controlling lifetime of locations and visibility of names. Modules are provided to restrict visibility in order to protect names against unauthorised usage. Processes and regions provide the means by which a structure of concurrent executions can be achieved. A complete CHILL program is a list of modules or regions that is surrounded by a (imaginary outermost) process.

Piecewise programming

This facility allows large programs to be broken down into smaller, separate handling units.

CHILL has also spread beyond public switching and has been used for a number of other telecom products like PABXs, non-voice equipment, etc.

CHILL has stood the test of time. Even though CHILL is now almost 20 years old, it has not been outdated. On the contrary it is flexible enough to fit into new technological contexts. For example, experience has shown that:

- CHILL is well suited as a target language for translations from SDL (5) and ASN.1 (8)
- CHILL concurrency concepts are well suited for implementing distributed systems (4).

CHILL has proven to be powerful enough for implementing modern telecom systems which are much larger and more sophisticated than could have been foreseen in the 1970's, e.g. ISDN, IN and mobile communication systems.

Finally, the CHILL work has provided a foundation for commercial product developments.

References

1 ITU. *CCITT High Level Language (CHILL).* Geneva, 1980-92. (Recommendation Z.200.)

2 ITU. *Functional Specification and Description Language (SDL).* Geneva, 1976-92. (Recommendation Z.100.)

3 ITU. *Man-Machine Language (MML).* Geneva, 1976-92. (Recommendation Z.300.)

4 *CHIPSY Reference Manual.* Trondheim, KVATRO A/S, 1993.

5 Botnevik, H. Developing Telecom Software with SDL and CHILL. *Telecommunications,* 25(9), 126-132, 139, 1991.

6 *SDT Reference Manual.* Malmö, TeleLOGIC Malmö, June 1992.

7 *GNU Emacs Manual.* Free Software Foundation, 1986.

8 ITU. *Specification of Abstract Syntax Notation One (ASN.1).* Geneva, 1988. (Recommendation X.208.)

Human-Machine Interface design for large systems

BY ARVE MEISINGSET

681.327.2

Abstract

To read a good book on painting will not make you become a Michelangelo. Neither will the reading of this paper make you become a good Human Machine Interface (HMI) designer. But, if the reading makes you become a wiser HMI designer, then the objective of writing this paper is met.

The paper identifies challenges imposed by design and use of large systems and outlines solutions to these problems. An enlarged scope and alternative perspectives on HMIs are proposed. The scope of HMI is proposed to contain a large portion of the system specifications, which have traditionally been considered to be the restricted domain of the developer. The end user needs this information for on-line help and as a menu and tutorial to the system. Also, end user access to the specifications puts strong requirements on how specifications are presented, formulated, their structure, terminology used, and grammar. This leads the HMI developer into issues on deep structure language design and fundamental questions about the role of languages and methods for their design. An encyclopaedia metaphor is proposed for organising and managing data of large organisations. This leads to thoughts about what competence is needed to develop and manage future information systems.

Norwegian Telecom Research has contributed significantly to the new draft CCITT Recommendations on the Data Oriented Human-Machine Interface Specification Technique (1). The major experience for making these contributions arrives from development of the DATRAN and DIMAN tools, which are also presented in this magazine.

Scope and objectives

The design of Human-Machine Interfaces, HMIs, has great consequences for their users. The HMI, in the broad sense, is what matters when designing information systems. It has consequences for efficiency, flexibility and acceptability. Therefore, the internal efficiency and external sales of services of any Telecommunication operator will be dependent on the HMI provided.

As applications are becoming increasingly data centred, this paper is focusing on HMI design for large database applications. The term 'Human-Machine Interfaces' has been introduced by the Consultative Committee for International Telegraph and Telephone, CCITT, Study Group X Working Party X/1 (2),(3), to replace the older term 'Man-Machine Communication'. Except from replacing the term 'Man' by the more general term 'Human', the new word is also intended to indicate an enlarged scope of HMIs, as will be explained in a subsequent section.

The term 'Interface' indicates a narrowing in of the scope to the interface itself, without any concern about the situation and tasks of the communicating partners, as indicated by the term 'Communication'. This delimitation is not intended. However, this paper and the current work of CCITT are focusing on the formal aspects of the 'languages' used on the interface itself.

Most work on HMIs have, so far, been concerned with the design of 'small systems'. The interface typically comprises the handset of a telephone or some screen pictures for a specific user group carrying out a limited well defined task. However, users of computer systems are nowadays not only carrying out routine work through their HMI. Users are often using several divergent applications simultaneously, and one application can comprise several hundred screen pictures, many of which are often used, others are hardly known. This usage of large systems imposes new requirements to and new objectives of HMI designs.

The focus is no longer on enabling the carrying out of one specific task in the most efficient and user friendly way. The concern will be to carry out a total set of divergent tasks in an efficient and user friendly way. Hence, tailoring of HMIs to a specific task can be counterproductive to the overall accessibility and productivity goals.

To achieve the overall goals, we have to enforce harmonisation of HMIs across large application areas. The central concern will not be on 'style guidelines', but on linguistics – to ensure a common 'terminology and grammar' of all HMIs across one or more application areas. This co-ordination is much more fundamental than providing guidelines for usage of 'windows, icons and fonts'. The new issue can be likened with choosing and designing 'French' or 'Norwegian' terms and word order. In a large system the user must be able to recognise and interpret the information when he sees it. It does not help to be 'user friendly' in the narrow task-oriented sense, if the users do not recognise how data are structured and related to each other. In large systems, the users want to be able to easily recognise the 'language' used when the same data are presented manipulated on various screens.

In the quest for solutions to our very appropriate needs, we will have to address approaches and techniques which are not yet 'state of the art' for HMI design. We will have to address questions on architecture of HMIs towards information systems and formal language aspects of the HMIs. Before doing so, we will provide an overview of what is HMI and what perspectives on HMIs can be taken. What then, is HMI?

HMI comprises all communication between human users and computer systems.

Some authors claim that the computer is just an intermediate medium for communication between humans. This metaphor, to consider a database application to be a multi-sender multi-receiver electronic mail system between humans, provides a powerful perspective on what an automated information system is and can be. In a real electronic mail system, however, we distinguish and record the individual senders and receivers. In a database application usually the data only are of interest, while the user groups are only discriminated by the access control system.

In some applications this perspective of mediating communication between humans makes little sense. If data are automatically collected and recorded in a database, e.g. alarms from the telecommunications network, it is of little use to claim that the equipment in the very beginning was installed or designed by humans. Therefore, we will consider the computer system to be a full-blown communicating partner in its own right.

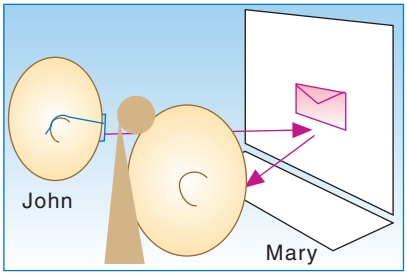


Figure 1 The computer system can be considered being a multi-user medium for human-to-human communication. This is a powerful metaphor for system design and supports a good tool perspective on HMIs

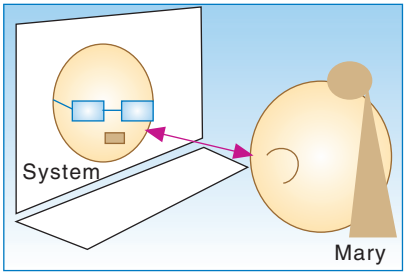


Figure 2 The computer system can be considered being a communicator in its own right. This is the most general metaphor, but can easily mislead the HMI designer to mimic human communication

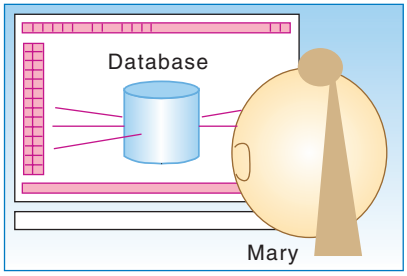


Figure 3 The tool perspective provides the user with full control over the systems he is using. In this perspective the user is considered being outside the system and is not a part of it

To consider the computer to be a communicating partner does not mean to require that the computer system should act like or mimic a human. We all share both good and bad experiences from trying to communicate with humans. The experience is often that the more 'intelligence' possessed by the partner, the more screwed up the communication can become. A simple example is a copying machine. When you were using the old machines, you pressed the key for A4 format and you got A4 copies. However, modern machines observe the size of whatever you put on the glass plate and provides you with what it thinks you 'need'. If you put on a large book, it provides you with A3 copies, even if you only want A4 of one page. When using 'intelligent' user interfaces, your problem easily becomes how to bypass the 'intelligence'. Therefore, we want 'dumb' interfaces, where the information system serves you as a tool. This does not mean that the interface should be poor and impoverished. We want interfaces that provide powerful features and applications that undertakes complex analysis and enforcement of data. The point made here about 'intelligence' is that we want the user to be able to predict how the system will react, that the system behaves 'consistently' and does not change behaviour as time passes. Also, we do not want the system to react very differently on the same or a similar command in a different state. This issue, to enforce consistency of behaviour over a large application area is a separate research subject within formal HMI (4).

The tool perspective on HMIs is important, because it provides an alternative approach to information systems design. Many existing methods, more precisely the system theoretic school (5), consider information systems to be complex factories, made up of humans and machines, for producing and maintaining information. Also, most existing software development tools are based on this paradigm. Tasks are separated between humans and computers according to maximum fitness for the task. The slogan for this can be 'to tailor the system to the tasks'. However, the end result can easily become 'to tailor the human to the task'. This tradition goes back to Taylor's Scientific management (1911). In some restricted applications this approach can be appropriate. However, in the 'large systems' context it comes too short – due to the diverse needs of different tasks to be supported simultaneously.

An alternative to the system theoretic approach is to consider the information system to be a tool for recording, organising, enforcing, deriving and presenting information. The 'production' primarily takes place inside the software system. The end user supplies and uses data from the system, and he can control the processing and information handling, which takes place inside the system. The end user is not considered to be a part of the system, but he is considered to control and use the system as a tool.

Unfortunately, the current practising system developer, or more specifically the HMI designer, is often not aware of which approach he is using, which perspective he is imposing and thereby, which work environment he is creating. Therefore, more reflection on implicit effects of alternative system development methods is needed.

Terminals

Hardware design is a compromise between technology, price, functionality, ergonomics, aesthetics, history, knowledge, performance, security, and other factors.

Hardware design is outside the scope of interest and scope of freedom of most readers of this paper. However, there is one hardware factor which must be addressed by every HMI designer: The number one priority is availability of the terminal equipment to the users you want to reach.

In most cases you cannot require that the users acquire a new terminal when introducing a new service. Most often you have to accept existing terminals and communication networks, or only require modest extensions to these. On more rare occasions you are responsible for purchasing and evaluating terminals – seldom for one service only, but more often for general use of several services.

This consideration has too often been neglected in expensive development projects. It is of little help to develop an advanced electronic directory system requiring powerful UNIX workstations, when most potential directory users have access only to a telephone handset. This problem is illustrated by the French Minitel service. The terminals are simple and cheap, and hence, affordable and widely spread.

The alternatives to Minitel have provided good graphics, require special communication, are costly, and hence, unaffordable and hardly in use. The situation is illustrated by an intelligent guess about the current population of terminal equipment in Norway:

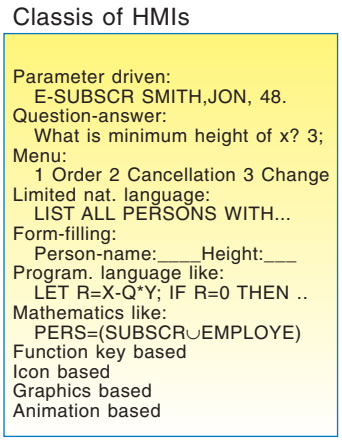


Figure 4 The choice of a dialogue technique is more fundamental than the choice of style guidelines. Note that the techniques can be combined

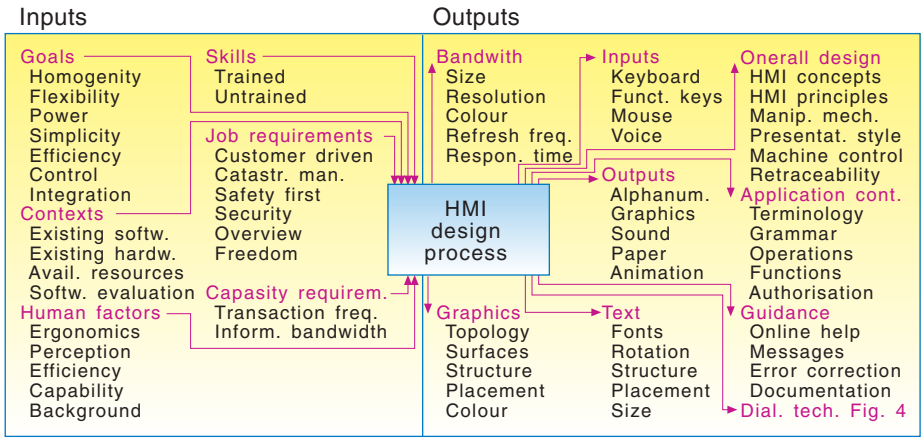


Figure 5 The HMI designer has to consider many factors and has many design choices. Therefore, the novice designer often requests detailed design techniques. However, a good development technique may not lead to a good design. The expert designer has knowledge about priorities and is more concerned with the final result

Telephones	3,000,000
Radios	4,000,000
TV sets	2,000,000
PCs	300,000
Workstations	10,000

These figures have to be correlated to what communication facilities are available, e.g. to PCs, use of supplementary equipment, such as a modem, tape recorder, video recorder, etc., and how the equipment is available to the actual user group you want to reach. No doubt, the most widespread electronic directory service is achieved if a practical service can be offered on an ordinary telephone or a simple extension of this equipment. The competitors on other equipment will just get a small fringe of the total market.

However, there are pitfalls. Many existing telephone services, e.g. Centrex services, are hardly in use, due to a complex, difficult to learn, hard to overview, and tiring in use 'human-machine interface'. Therefore, an alternative strategy can be to introduce a service for a small and advanced user group. Also, the choice of strategy can be dependent on whether you want to provide a service or sell equipment. Or maybe the service offered is just a means for promoting other services, for example to promote a new basic communication service. Also, the terminal and communication environment may be settled by another and more important service in an unrelated area.

However, there are many challenges left to be addressed by the future hardware designers. The most obvious challenge is the design of the PC. The laptop is more ergonomic in transport, but not in use. You definitely want a 'glass plate' terminal with which you can lay down in a sofa and read, like when reading a book. And, you definitely do not want to have to use a keyboard when just reading. The pen based terminals make up a move in the right direction, but they are not the end. In the future, you are likely to see 'walkman terminals', 'mobile phone terminals', 'wall board terminals', 'magic stick terminals', etc. appearing under labels like the 'Telescreen', the 'Telestick' and the 'Telegnome'. With the coming technology, there is no reason why you should sit on an excavator to dig a ditch. Rather you could sit in a nearby or remote control room, controlling the grab by joy sticks, observe the digging by video cameras and remote sensing. Most of the work could even be carried out automatically, while you supervise the process and are consulted when something has to be decided. And, also the excavator can be designed very differently, when it is no more designed for a human driver. The excavator becomes your 'little busy gnome'. Tunnel construction, mining and other application environments where you prefer not to have humans, are obvious areas for the introduction of this kind of services. Remote control of mini submarines is an early, already existing, application of this technology. In the telecommunication area, the technicians are not typically replaced by robots. Rather, control, repair, and switching functions are built into the telecommunication equipment. The equipment is administrered and its functions are activated as if managing a database.

Style guidelines

Both industry and standardisation bodies have done a lot on providing guidelines for layout design on screens and keyboards. The International Standardisation Organisation, ISO, is currently issuing a 17 piece standard ISO 9241 (6) on HMI design, of which 3 has reached the status of a Draft International Standard, DIS. The industry has provided products like MS Windows (7) and Motif implementations (8).

Obviously there are many good advises in these guidelines. However, suppose you compare these guidelines with that of designing the layout and editing of a book. A Motif form looking like a front panel of an electronic instrument, where each field is put in a separate reading window, may not be so good after all. The push buttons look like that of a video recorder – which you definitely are not able to use. All the windows popping up on the screen look like an unordered heap of reminder notes in metallic frames. The frames and push buttons are filling up most of the screen. The ones you need are often under those you have on the top. Each window is so small that it can hardly present information, and you have to click a lot of buttons in order to get what you want. In fact, it is more likely that a book layout designer will be horrified rather than satisfied by the current use of style guidelines.

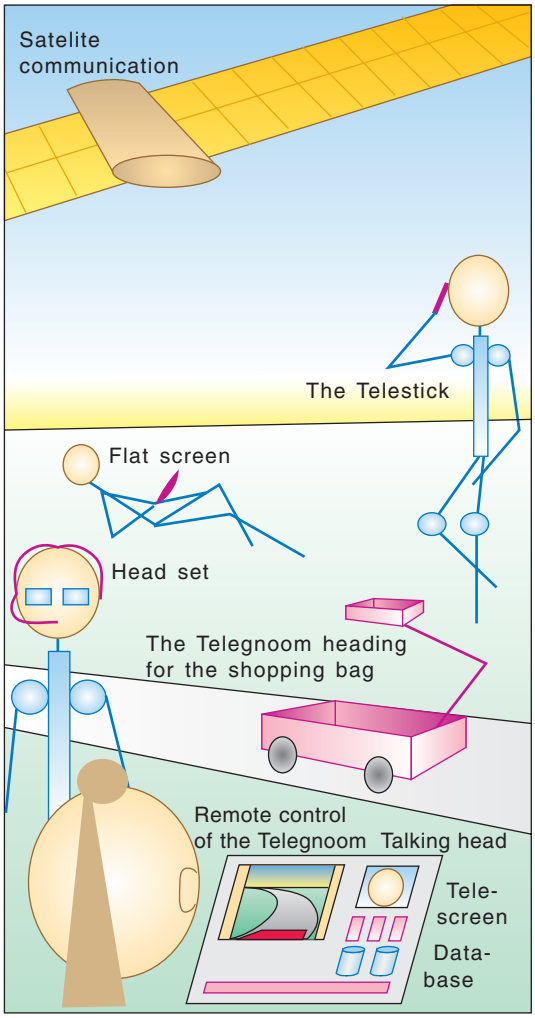


Figure 6 The number one priority of hardware design is the availability of the terminals to the users. Too many projects have failed because the design has been made for workstations not available to the users. Much work remains to make terminals as ergonomic as a book; this concerns physiognomy, perception, controllability, etc. Also, new kinds of terminals are likely to appear for existing and new applications

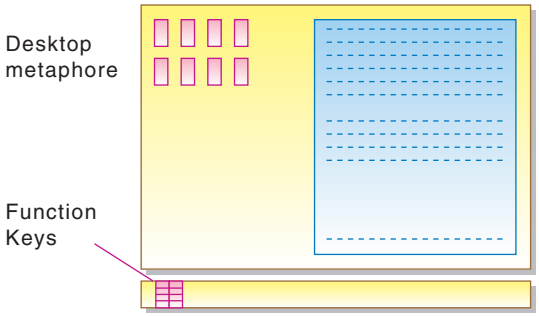


Figure 7 The Xerox Star workstation has been the model for all desktop metaphors. The Star had a fixed set of function keys which applied to all kinds of objects. This way it provided a homogeneity not equalled by other products

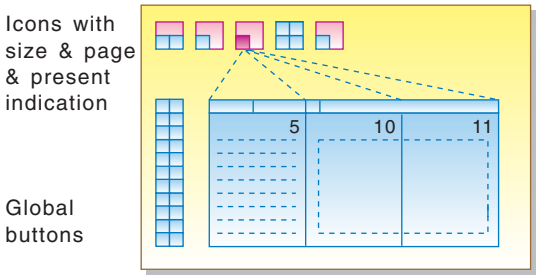


Figure 8 The current desktop metaphor can be improved to provide better control to the user. Here sub-icons represent different presentation forms of a document. Several pages are shown from the same document. A global slidebar indicates open pages

What about printing? If the form is put into a Motif panel, it is likely that this is not suited for printing on paper. You will have to design another form for print out. However, this violates one of your more basic requirements – to recognise the form and the data in different environments. The recognition of data is achieved by having persistence of presentation forms between different media. In order to get a clean work space for information, which the screen is for, you should move the tools – the push buttons – out to the sides and arrange them in a fixed order there – much like that of the early drawing packages. Arrangements of buttons around each window and in separate push button windows may not be the best. Also, the use of tiled windows distorts the whole view. Maybe the fixed arrangement of windows, like in the early Xerox Star, was a better choice? Use of shading, 'perspective', and colours are often a total misuse of effects.

There are several shortcomings of the current desktop metaphors. For example, you may want to see the relative sizes of the documents and files when you see their icons. You may want to open the icons on any page or place without always having to see the first page; you may want to use slide bars on the icons and not only inside the windows. You may want to open several pages simultaneously inside the same document. You may want to have a common overall indication of which pages you have opened. In short, you want to have the same control as when holding and paging through a book. You want to copy some pages to print when others are opened. And, you want all desktop tools to be available on the desktop simultaneously, and not to pop up as dialogue boxes in certain states. Also, you want a clear distinction between a book (an icon for its contents), alternative presentation forms (of the same contents) and finally windows to present the contents in a form. Therefore, iconification of windows is what you least need.

We have all seen the bad results of producing papers by the current desktop publishing products. The individual author lacks the competence of the professional layout designer, but is provided

Interface engineering	Development phase	Software engineering
User and task analysis	Requirements analysis	Application design
Human versus machine	Requirements allocation	Hardware versus software
Dialogue design	Preliminary design	Architectural design
Screen design	Detailed design	Logical design
Coding	Implementation	Coding
Useability lab	Implementation testing	Unit and integration test
Contextual observation	System testing	System testing
Human performance	Optimisation	Machine performance

Figure 9 Curtis and Hefley (9) present this as 'A potential high-level mapping scheme for integrating phases of the interface- and software-engineering processes.'

As development tools become more high level, we should design systems as seen from the outside only. This implies a move towards the leftmost column

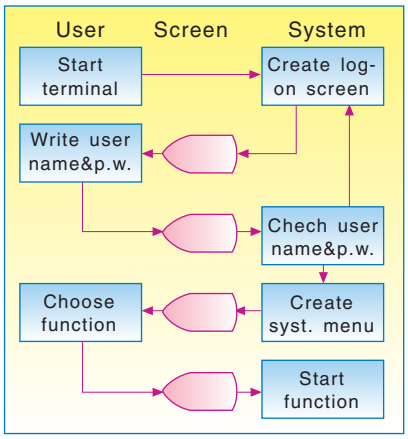


Figure 10 In the function oriented approach, tasks are decomposed and finally grouped to become manual or automatic processes. Screens and reports constitute the HMI between these processes

with all the freedom he is not able to use appropriately. The provisioning of freedom requires the practising of responsibility. Therefore, the good designer is often puritanical.

Our database applications are often very expensive 'publications'. We spend large sums to develop them and large amounts to support and use them. For example a database containing information about resources and connections in the telecommunication network costs more to develop, maintain and use than most existing paper publications. Also, the database is used by hundreds and thousands of users daily. We usually put much more emphasis on using qualified personnel when designing a paper publication than when designing a database application. Database applications are most often designed by programmers having no layout competence. We need personnel to design large and complex applications which can appear systematic, comprehensible and typograhically pleasant to the end user. Therefore, we should spend just as much money on using qualified personnel on HMI layout design to database applications as we spend on paper publications of comparable costs.

This definitely implies that the system development departments have to be upgraded with the appropriate typographic competence. Current technology is no longer a limitation to the use of this competence. A graphic terminal is becoming the standard equipment of most office workers. Currently we automatically produce some simple graphs of the telecommunication network. However, the challenge is to illustrate this very complex network in such a way that this provides overview and insight. Investment in competent personnel to design systematic and illustrative graphs of the entire network for the different user groups can be paid back manifolds.

The challenge does not stop with turning database applications into high quality publications of text and graphics. New technology will provide a multi media environment. The future database application designer will have to become a multi art artist. He will create an artificial 'cyber space' of his database, for example for 'travelling through' the telecommunication network database. And, he will integrate artificial speech, recorded sound and video. These features have their own strengths and possibilities, weaknesses and pitfalls. The use and integration of all this will require somewhat more than programming competence.

Contents

To design the layout of the data without bothering about the contents is like making a typographic design of a book without knowing what it is about and how it is structured. And, does it consist of tables, charts, free text or other? How can these be arranged to best present the information?

Contents design of a screen picture comprises: What information should be included in the picture? What statements are used to express this information? Which arrangement of the statements is chosen? Also, the contents design comprises: What pictures should exist? How are pictures grouped together to provide overview? What links, if any, should exist between pictures or groups of pictures? We see that contents design corresponds to organising a book into sections and writing the contents of each section. This topic is covered by the 'contents structure' of the Open Document Architecture ISO/CCITT (10), while style guidelines correspond to the 'layout structure' of ODA.

When designing the contents structure of a picture, the HMI designer also has to decide what operations (commands or directives) are permissible on the selected data items in this context.

Several design methods exist for contents design. The system theoretic school has been dominating. In this school the designer analyses the tasks undertaken by the user. The tasks are identified in data flow graphs. Some tasks are automatised, others are kept manual. The HMI is identified on the boundary between manual and automated tasks. One screen picture is designed to include the data flowing one way between the two tasks. The operations are limited to those needed for this data exchange and the transition to the needed picture for the next exchange of data. This way, the HMI is tailored to the tasks and routines of an organisation.

The system theoretic approach can be appropriate for undertaking well defined or routine tasks. However, the approach is not flexible for alternative ways of undertaking the same tasks or different tasks. Also, the approach does not address the harmonisation of the dialogue across many tasks for a large application domain. For this a modeless dialogue is wanted. This means that all screen pictures should be permissible and easily accessible in every conceivable sequence. Also, the user should be allowed to

- Cursor qualities
  - Easy to find
  - Easy to track
  - Do not interfere
  - Do not distract
  - Be unique
  - Be stable
- Presentation
  - Label information clearly
  - Keep display simple
  - Moderate highlighting
  - Moderate the use of colours
  - Keep layout consistent
  - Presentation of data
    - Use upper and lower case letters
    - Minimise use of codes
    - Maximum 5 characters in a group
    - Use standardised formats
    - Place related items together
    - Place figures in columns
    - Adjust integers to the right
    - Adjust figures around the decimal point
    - Adjust text to the left
  - Make only input fields accessible
  - Distinguish
    - Input fields
    - System response
    - Status information
    - User guidance
    - Menus
- Response time
  - Psychologically acceptable
  - Uniform
- User guidance
  - Keep guidance up-to-date
  - Ensure consistent presentation
  - Avoid abbreviations
  - Use action oriented sentences
  - Make help messages clear
  - Make help contextual
  - Do not guess what the user wants
  - Provide help for help
  - Use identical commands to that of the application
- Errors
  - Where are they
  - What error
  - How to recover
  - Make error messages clear and polite
  - Report all errors together, prioritise presentation
  - Keep error information until corrected
- Dialogue
  - Use direct information entry for experienced users
  - Use menus for causal users
- Menus and forms
  - Keep the menu hierarchy low
  - Provide direct addressing of menus
  - Keep the presentation consistent

Figure 11 Extracted guidelines from extended MML for visual display terminals (3)

- General principles
  - The meta-language
- Basic syntax
  - Introduction
  - Basic format layout
  - The character set
  - Input command language
  - Output language
  - Dialogue procedures
    - Use of SDL
- Extended MML
  - Capabilities
  - Interaction
    - Dialogue procedures
    - Windows
- Specification
  - Methodology
  - Tools and methods
    - Glossary
    - Procedure description
    - Backus Naur Form

Figure 12 Man-Machine Language. Summarised contents of the older CCITT Blue book (3) on MML

perform all permissible interrogations of data in all pictures in any sequence. This means that each picture should allow the end user to perform all operations he is authorised to do in all pictures containing the appropriate data. This implies that the pictures are no more restricted to certain tasks and operations needed in these tasks. Each picture can contain data and provide operations which apply for many tasks. To some extent, data will still have to be grouped into pictures appropriate for certain tasks or groups of tasks. However, there is no more any strict correspondence between tasks and the contents of screen pictures. We will call this alternative approach a Data oriented approach to HMI design, while the previously described approach we will call Function oriented.

The data oriented approach implies a need for a new technique to design programs, such that all conceivable and permissible operations are provided

- Introduction
- Scope, Approach and Reference Model
  - Scope
  - Approach
  - Reference Model
  - Guidelines for HMI developers
    - Introduction
    - Method
    - Data design
    - Summary of Requirements
    - Relationship to TMN
- Formalism and Documentation
  - Introduction
  - Formalism
  - Documentation
  - Guidelines for HMI developers
    - Examples
    - Handling of Time
    - Explanation and Usage
    - Extensions
    - End User's Access to Specifications
    - Access Control Administration

Figure 13 The data oriented Human-Machine Interface specification technique. Summarised contents of new draft CCITT Recommendation (1) on HMI

- Be consistent
- Provide feedback
- Minimise error attributes
- Provide error recovery
- Accommodate multiple skill levels
- Minimise memorisation

Figure 14 Important design considerations (11)

simultaneously. Also, the approach implies a need to provide the user with powerful means to select and project data in run time for divergent needs, which the application programmer may not be aware of. This means that forms can no more be fixed panels, but must be dynamically changeable to user needs. The details of the 'HMI design' are moved from the programmer to the end user himself. For this use, the end user has to be provided with a general, power-

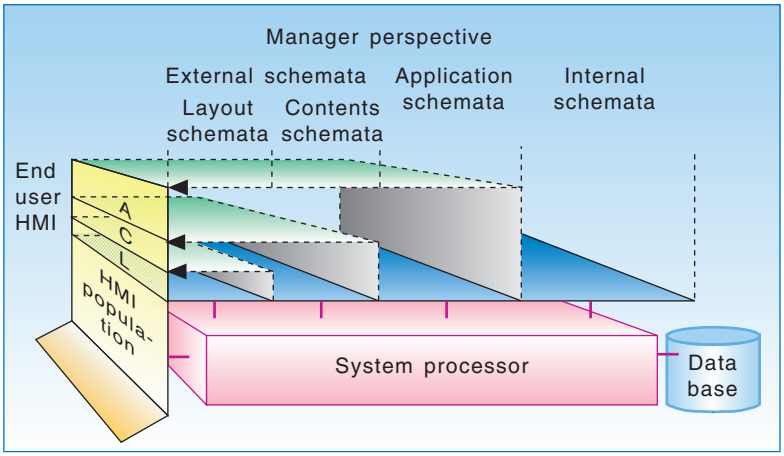


Figure 15 If we take a black box approach to system design, we realise that the end user needs access to much the same information as the system developer.

The application schema (A) prescribes the terminology and grammar of the application. The external schemata prescribes contents (C) and layout(L)

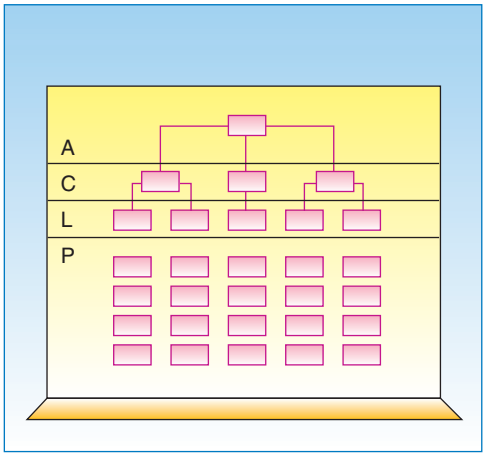


Figure 16 When designing data in the application schema (A), the HMI designer has to foresee permissible external contents (C), layouts (L) and instantiations (P)

ful and simple to use editor of the form and contents of all his screen pictures.

Preferably, this editor should allow alternative presentations of the same data, e.g. both alphanumeric and graphic presentations. However, this is not easily achieved in practice. For example, in addition to the pure contents, a graph of a telecommunication network will require the provisioning of appropriate icons for the classes of data, e.g. for stations and circuit groups. Also, information is needed about where to place the icons, scaling, rotation, etc. And, if editing is allowed, information must be provided about the topology of data, e.g. how circuit groups are linked to stations. Also, these items have to be linked by appropriate routines. All this information may not be explicitly available in an alphanumeric screen picture, which the end user intuitively believes contains the same information. Therefore, meaningful graphs may not be easily derived from pictures not prepared for graphic presentations. Rather, the user needs a seamless integration of graphic and alphanumeric dialogues, in a way which appears to be integrated to the end user.

In the Function oriented approach the end user is considered to sit in the stream of information flow, entering information from his environment or retrieving information to the environment. In the Data oriented approach the information flow can be considered to take place inside the computer system, not to or from it through the terminal. The end user supervises and controls this flow. He can retrieve data – from the flow –, insert new data and redirect the flow. This way, 'register handling', 'information flow', as well as 'process control' can be undertaken by the Data oriented approach to HMI design.

Applications

To have a good contents structure of a publication is a must to get easy access to information. However, if the information is not expressed in a consistent and understandable language, then the end user will be lost. A typical problem in many current application designs is that headings and icons are inconsistently and uncoordinatedly used in different pictures. Therefore, in large systems the designer needs methods and tools to ensure the harmonisation of a common terminology and grammar across large application areas. By the term grammar we mean the sequence in which data are presented. The issue here is that if data are presented in arbitrary order, for example by interchangeable columns in a table, the user can misinterpret the relationships between items in the different columns. To achieve unambiguous interpretation of data, the end user grammar has to be defined by the relationships between the data.

To achieve the wanted harmonisation, the terminology and grammar have to be expressed only once for each application. This central definition is called the Application schema of the system. The Application schema will also contain all constraints on and derivations from the data.

The Application schema is not only the focus of the developer. Once defined, the definition of the common terms of a system is what the expert users need in order to understand what the system is about. In addition, they need this information to validate and approve the design prior to implementation. The ordinary users need access to these specifications for on-line help, as a tutorial and a menu to the system. Therefore, the Application schema is considered to be inside the scope of interest of the end user's HMI. Also, the end user has to know which presentations are available and which operations are permissible in these presentations. Therefore, the specification of external contents and layout of each picture is inside the scope of the end user's HMI. This way, a large portion of the specification data are included in the scope of the HMI. The impact of this, is that these specifications have to be presented in a way which is accessible to the end user, using the end user's own terminology and grammar. Hence, the developer is not free to specify the system in a way found convenient only for the programmer and the computer, but he has to observe end user needs both concerning what is specified and how this is specified.

There are many approaches to application design. Many schools take contents schemata as their starting point and merge these into one Application schema. So-called 'normalisation' belongs to this group of schools. There are several problems with this approach: it presupposes that (1) contents schemata can be defined prior to having a unified Application schema, (2) all data are appropriately used in all contents

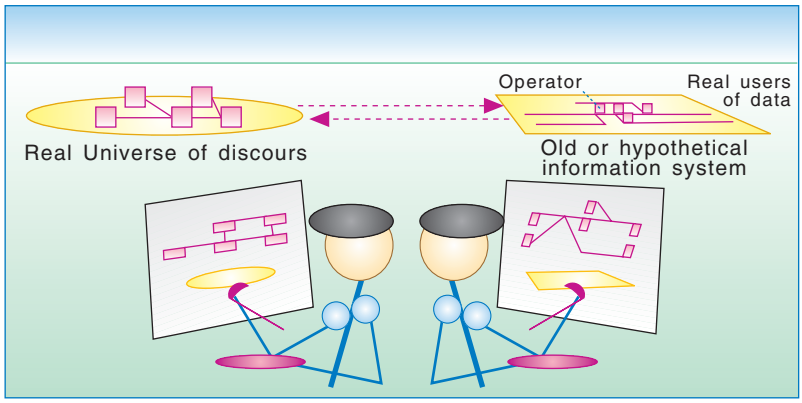


Figure 17 Designers regularly confuse universes of discourse. Specification techniques often prescribe analysis of the information system rather than the UoD administered by this system. This way, they prescribe the copying and conservation of previous designs, rather than contributing to creating alternative designs.

Also, user groups can be confused. Often the operators at the terminals are just communicators of data, while the real users of the data are outside the information system being analysed. Creative data design should focus on the needs and usages of the real users

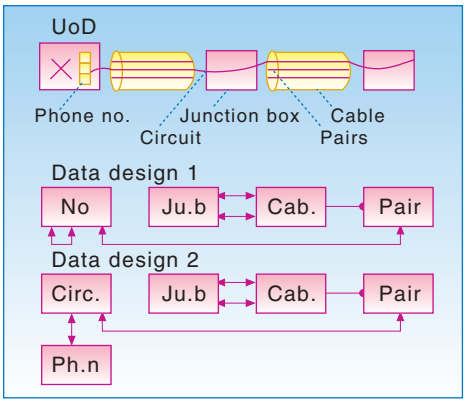


Figure 18 Depiction of alternative data designs to what was believed to be about the same UoD. The first design stems from 'normalising' existing data. The second design takes a fresh look at the structure of the real UoD. Alternative data designs have great consequences for their users

schemata, and (3) all data are conveniently and consistently defined in all contents schemata.

Experience with large systems design shows that these assumptions are regularly violated. In fact, to define a consistent and efficient terminology and grammar for an application area should be the focus of all HMI design, while most current design techniques take current data designs for granted. However, no measures seem to have such a profound effect on overall efficiency and usability as data design. Therefore, CCITT Recommendation Z.352 contains a separate Appendix on data design for HMIs (1), (2).

Many current application design techniques are naivistically conceptualistic. This means that they disregard the form in which data are presented to the end user. However, for the end user to recognise and understand his data, the forms of the data have to appear persistent throughout the system. Experience shows that 'conceptualistic' designs, where the external forms are not taken into account from the very start, have to be redesigned when the external aspects are taken into consideration. Therefore, HMI design is intrinsically concerned with the definition of the form of data across large application areas (2).

The design of a harmonised and efficient terminology for a large application area is a complex task, requiring a lot of experience. The designer has to foresee all conceivable presentations, i.e. contents and layouts, of his data designs and also foresee all conceivable instantiations of these presentation forms. Therefore, data design requires good skills to generalise and systematise. The end users are often not capable to foresee all consequences of the data design. Therefore, prototyping of the HMI is needed to validate the design. This prototyping must be realistic, i.e. allow the end users to use and report judgements about the design.

The end user has several subtle requirements on the form of his data that have profound implications (1).
Firstly, labels should be allowed to be reused in different contexts having different value sets and different meanings. For example, the label Group could mean an organisation unit in an administration, or a circuit group within the transmission network, or other. A label which is reused for different purposes in different contexts is called a local label to the label of the context item. Secondly, identical labels can be reused within the same context to mean different entities. For example, identical icons can be used to represent different exchanges in an electronic map, and the end user may never see unique labels for each station. Therefore, the treatment of significant duplicates is needed. Lastly, specifications have to be homomorphic to, i.e. have the same form as, the instantiated data. The last requirement allows the end user to recognise data classes for given data instances and vice versa. Without obeying this requirement, the end user will easily get lost when traversing between specifications in the schemata and corresponding instances in the populations.

The consequence of the above requirements is that the specification language needed for end user access to data, using the end user's own terminology, will be a context-sensitive language (12, 13, 14), having much in common with the context-sensitivity of natural languages. The required language will have features which are distinctively different from widespread existing specification languages.

Data are often considered to describe some Universe of Discourse. Maybe surprising to the reader, most designers have great difficulties in identifying the right UoD. Most designers take an existing or hypothetical information system, e.g. of a Telecom operator, as their starting point and extract and organise the data from this. However, the information system usually administrates information about some other system, e.g. the telecommunication network. A designer should study this network when designing an information system to manage it.

However, the designer should be warned that data often do not describe physical entities in the naivistic sense. More often data are invented to provide overview of other data, which provide overview of

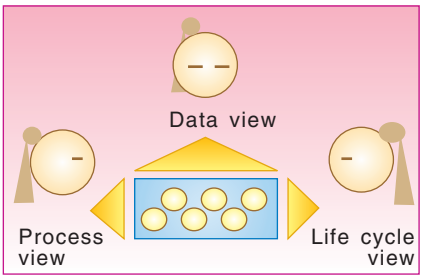


Figure 19 The current machine metaphor of information systems comprises a data view, a processing view and a life cycle view. This metaphor seems not to provide the needed overview and control to users and managers. The metaphor introduces too much consistency to be controlled by the developer and too little systematisation. Rather than developing separate specifications, which have to be checked for consistency, a centralised specification (i.e. the application schema) should be developed, from which all other specifications (i.e. the external and internal schemata) can be derived

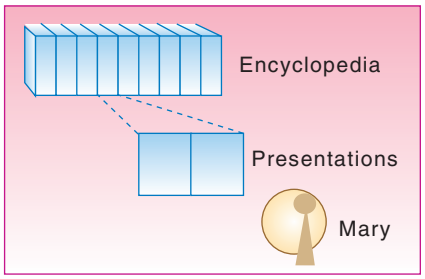


Figure 20 The encyclopaedia metaphor can provide simple overview and control to the user. Each book corresponds to a system. This metaphor can provide manageability to the manager. However, the metaphor is not complete, as it disregards communication between systems. But the details provided seem right for discussions with user departments about ownership of and responsibility for which system

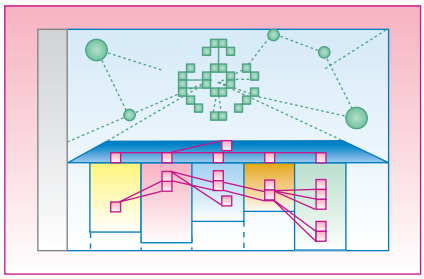


Figure 21 An advanced approach to providing overview. Directory information is depicted by the stellar map metaphor in the top window. Object class icons make up the system stars. The user can zoom in the wanted ones. The corona depicts external screen pictures. The communication links between systems are indicated by dotted lines. The space platform in perspective indicates the work station. Its icons indicate the extracted data types. Data instances are depicted in the bottom window. Colours indicate selected icons. The heights of the rectangles indicate sizes of the files. The icon instances are placed in their appropriate places according to an alphabetical sort

still other data, etc. Also, designers have been inclined to believe that the terminal operator is the user of the data. This belief may not be true. Quite often the operators are just communicators of data to the real users, who can be customers, planners, managers, etc. Therefore, identification of UoDs and users are important, not trivial tasks.

Architecture

We realise that the complexity of large systems implies a need for alternative approaches to HMI design. While current form-filling dialogues have been appropriate for presenting and manipulating small groups of individual data items, they are not appropriate to provide overview of large systems. The user wants to know what features are available, what data types can be interrogated, how large are the populations, where is this item in this population, etc.? The challenges for future HMI designers are to provide easy to use and easy to recognise features for this.

Current metaphors for system design have been considering the information system to be a huge machine consisting of processes which exchange data by data flow. As systems grow, this metaphor has become too complex to cope with. Therefore alternative metaphors are sought after. The encyclopaedia metaphor can be a convenient alternative. The totality of information systems can be considered being a library. The information systems for one organisation is an encyclopaedia. Each system is a book, each function is a section, etc. A directory can provide an overview of the totality. This metaphor can constitute the needed overview and control to the user. The metaphor does neither exclude references to be maintained across several books, nor that information is exchanged between books. However, normally information is selected from several books to one user, while exchange between books is minimised.

Some schools of data design put up a ‘corporate database’ as the ideal goal, where the users see no distinction between books or systems. However, this can become like having a large house rather than several small ones. The goal may contradict the users’ wish of having several smaller systems, which can be wanted both for overview and controllability. ‘Megastructures’ may not be the ideal architecture.

Layering of systems can be another means to provide overview and controllability. The already introduced application, contents and layout structures are examples of this. If functionality is sorted into layers, the data flow between the layers also becomes more comprehensible. The flow between the layers is concerned with undertaking one step in the communication of data between two peer media, while the internal flow inside a layer is related to constraint enforcement and derivations. This data flow architecture can be a very powerful alternative to the control structures of ordinary programs.

Current thinking about information systems has been focusing on the details of the HMI or the details of programming statements. Architecture of information systems in large has not evolved to become a separate discipline. Such a discipline, however, is highly needed.

Education

From the discussion in this paper, it should be evident that current teaching on HMI design and information systems design in general, have distinct shortcomings: 1) HMI designers have been focusing too much on the surface layer style guidelines of HMI design and have missed most aspects of the deep language structure data design. 2) The architectural aspects of information systems in large need to be addressed. 3) Development methods for HMIs have to be reconsidered in light of large systems design. 4) The scope of HMIs has to be broadened, and this will require an integration of end user and developer perspectives. This will have significant impact on formal languages for the specification of information systems.

Pelle Ehn (15) quotes from Simon a proposal of a curriculum to redirect the focus of information systems development:

- Social systems and design methods
- Theory of designing computer artefacts
- Process-oriented software engineering and prototyping
- History of design methodology
- Architectural and industrial design as paradigm examples
- Philosophy of design
- Practical design.

Teaching about HMIs can be undertaken in the perspectives of end users, developers or scientists. This article has focused on the perspective of the developer. However, the reader should be warned that the three perspectives can be entirely different. An HMI can be easy to use, while it can be very difficult to design, e.g. the data designer has to foresee all implications for all conceivable presentations and instantiations of his designs. The tasks of the scientist can be entirely different from those of the developer. The scientist should be able to categorise the approach used, e.g. according to which philosophy of science. He may identify the organisation theory supported, the design principles used, the underlying philosophy of art, etc. Occasionally he may also engage in developing tools and methods for HMI design and HMI usage.

References

1 CCITT. *Draft Recommendations Z.35x and Appendices to draft Recommendations,* 1992. (COM X-R 12-E.)

2 CCITT. *Recommendations Z.35x and Appendices to Recommendations,* 1992. (COM X-R 24-E.)

3 CCITT. *Man-Machine Language (MML).* (Blue book. Recommendations Z.301-Z.341.)

4 Harrison, M, Thimbelby, H. *Formal methods in Human-Computer Interaction.* Cambridge, Mass., Cambridge University Press, 1990.

5 Bansler, J. System development in Scandinavia: Three theoretical schools. *Scandinavian Journal of Information Systems,* 1, 3-20, 1989.

6 ISO. *Visual display terminals (VDTs) used for office tasks – Ergonomic requirements. Part 3: Visual requirements.* 1989. (Draft International Standard ISO/DIS 9241-3.)

7 *Brukerhåndbok Microsoft Windows. Versjon 3.1.* Microsoft corporation, 1991.

8 *Open Software Foundation. OSF/Motif Style Guide. Rev. 1.2.* Englewood Cliffs, N.J., Prentice Hall, 1991.

9 Curtis, B, Hefley, B. Defining a place for interface engineering. *IEEE Software,* 9(2), 84-86, 1992.

10 ISO. *Information processing – Text and office system – Office Document Architecture (ODA) and interchange format,* 1989. (ISO 8613).

11 Foley, J D et al. *Computer Graphics: Principles and Practice.* IBM Systems Programming Series, second edition. Reading, Mass., Addison-Wesley, 1990.

12 Meisingset, A. *Perspectives on the CCITT Data Oriented Human-Machine Interface Specification Technique.* SDL Forum, Glasgow, 1991. Kjeller, Norwegian Telecom Research, 1991. (TF lecture F10/91.)

13 Meisingset, A. *The CCITT Data Oriented Human-Machine Interface Specification Technique.* SETSS Conference, Florence, 1992. Kjeller, Norwegian Telecom Research, 1992. (TF lecture F24/92.)

14 Meisingset, A. *Specification languages and environments.* Kjeller, University Studies at Kjeller, Norway, 1991 (Version 3.0).

15 Ehn, P. The art and science of designing computer artefacts. *Scandinavian Journal of Information Systems,* 1, 21-42, 1989.

Reference models

BY SIGRID STEINHOLT BYGDÅS AND VIGDIS HOUMB

681.3.01

Abstract

This paper explains what a reference model is and gives some examples of reference models. The examples are the ISA framework, different database architectures like the 3-schema architecture, the ISO OSI reference model and the client-server architecture. Telecommunications operators, like other large organisations with administrative and technical information systems, need reference models. This is mainly due to the increasing complexity of software systems and the need to provide interoperability between software systems. Another aspect is the necessity of organising the data of the organisation in an effective way.

Some of the important challenges in the area of reference models are to unify related reference models, integrate reference models across fields, make the standards precise and make the information technology community aware of the benefits of widespread use of standardised reference models. It is a hope that in the long run unified, integrated and standardised reference models will simplify education and systems development.

Introduction

The notion *reference models* is often encountered in standards and other literature on information systems. In this paper we want to focus on what a reference model actually is, where and why it is needed and illustrate these 'what', 'where' and 'why' of reference models through some examples of existing reference models.

The term reference model is defined in the literature (1) as a conceptual framework for a subject area, i.e. "a conceptual framework whose purpose is to divide standardisation work into manageable pieces, and to show, at a general level, how these pieces are related to one another. It provides a common basis for the co-ordination of standards development, while allowing existing standards to be placed into perspective within the overall reference model."

For all practical purposes, however, a reference model can be thought of as an idealised architectural model of a system (2). We thus consider the term reference model in the area of information systems to be synonymous to the terms architecture, reference architecture and framework of information systems. These terms are all used throughout the literature. Unfortunately, there is no common understanding of either of these terms.

Reference models are nevertheless a means to decompose and structure the whole or aspects of information systems. As size, complexity and costs of information systems are increasing, reference models are needed to handle the complexity of large information systems.

The field of information systems is quite young compared to house building, which has existed for thousands of years. Lots of experience has been accumulated throughout these years, and it would only be natural if the field of information systems could gain from studying the process of building houses, towns, or even aeroplanes. Zachman (3) has observed the processes of building houses and aeroplanes and found that a generalisation of these processes would be appropriate for the process of building information systems as well.

Existing reference models

The currently best known reference model within the field of information systems is perhaps the reference model for open systems interconnection, the OSI reference model, standardised by ISO. There are, however, several other important areas of information systems for which reference models have been, or currently are being, developed.

According to Zachman (3) interpretation of the term architecture depends on the viewer. This is the case for reference models as well, i.e. interpretation of the term reference model depends on the viewer. Different users have different viewpoints, or different perspectives, on a reference model. These different perspectives may be incorporated into a single reference model or may result in quite different reference models, each serving one particular perspective. A typical example of different perspectives that might be incorporated into a single reference model or in several reference models is development, use and management of an information system.

In the area of *data management,* i.e. definition and manipulation of an enterprise's information, several reference models are already being standardised. Examples are the ANSI/SPARC three-schema architecture, the ANSI and ISO IRDS (Information Resource Dictionary System), and a reference model for data management being standardised by ISO. The three-schema architecture will be presented later in this paper.

The OSI reference model has already been mentioned as a standardised reference model for *communication.* This reference model defines means for communication between two or more separate systems. Two aspects of communication that especially influence the co-operation between systems and system parts are *distribution* and *interoperability.* A distributed system is a system that supports distributed processing or distribution of data. Distributed processing may be done according to ODP (Open Distributed Processing) or the client-server model, and data may be distributed according to an extended three-schema architecture. Interoperability means that systems or system components co-operate to perform a common task. Interoperation is essential e.g. for database applications requesting information from each other.

In addition to the general reference models already mentioned, there exist several reference models for specific application areas. Three application oriented reference models currently being developed in the area of telecommunications are reference models for Intelligent Networks (IN) (4), Telecommunications Management Network (TMN) (5), and Human Machine Interface (HMI) (6). They are all developed by CCITT.

Zachman's framework for information systems architecture (ISA)

As we have already mentioned, Zachman has observed house building to see if the experiences made there can be applied to the process of building information systems (3). The observations he made resulted in a framework for information systems architecture called the *ISA framework.*

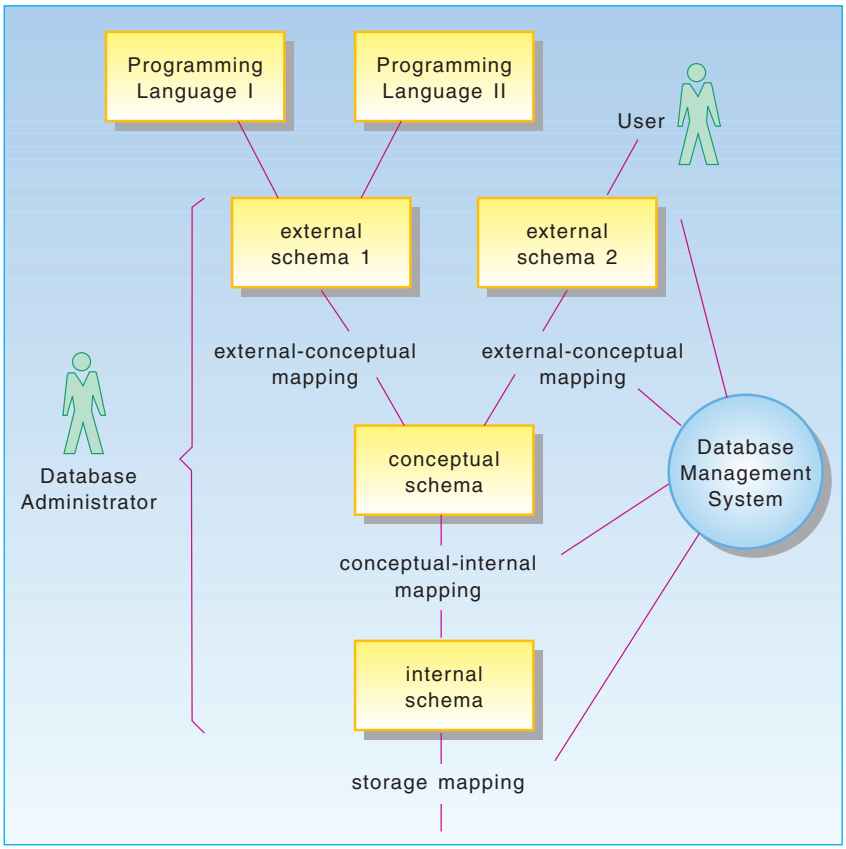


Figure 1 Three-schema architecture (9)

The three-schema architecture consists of three levels:
- *the exernal level*
- *the conceptual level, being the central level of the architecture defining the universe of discourse*
- *the internal level*

The ISA framework is based on the following ideas:

- There is a set of architectural representations of the information system representing the different perspectives of the different participants.
- The same product can be described, for different purposes, in different ways, resulting in different types of descriptions.

The framework assumes that there exist five architectural representations, one for each "player in the game", i.e. the planner, the owner, the designer, the builder, and the subcontractor. Each of these representations represents a different perspective.

Different types of descriptions are oriented to different aspects of an object

Table 1 The ISA framework
The cells are filled with examples of the kind of representation that can be used by the combination of perspective and type of description

	Data (what)	Function (how)	Network (where)	People (who)	Time (when)	Motivation (why)
Scope	List of things important to the business	List of processes the business performs	List of locations in which the business operates	List of organisations/agents important to the business	List of events significant to the business	List of business goals/strategy
Enterprise model	Entity/relationship diagram	Process flow diagram	Logistics network	Organisation chart	Master schedule	Business plan
System model	Data model	Data flow diagram	Distributed system architecture	Human interface architecture	Processing structure	Knowledge architecture
Technology model	Data design	Structure chart	System architecture	Human/technology interface	Control structure	Knowledge design
Components	Data definition description	Program	Network architecture	Security architecture	Timing definition	Knowledge definition
Functioning system	Data	Function	Network	Organisation	Schedule	Strategy

being described. The aspects are data (what), function (how), network (where), people (who), time (when) and motivation (why).

The ISA framework combines the five perspectives with the different types of descriptions into a matrix consisting of 30 cells (7). The framework, shown in Table 1, is thus a two-dimensional matrix where

- the columns state what aspect of an information system is described
- the rows state what perspective is taken
- the cells are filled with examples of the kind of representation which can be used by that particular combination of perspective and type of description.

According to the ISA framework each cell of the matrix is different from the others, being one particular abstraction of reality. There are, however, dependencies between cells; between cells in the same row and between adjacent cells in the same column. The cells are considered as information systems architectures where the notion of architecture is viewed upon as relative, depending on perspective and type of description.

Zachman's matrix provides a valuable overview of techniques used for systems development and the relationships between such techniques. We believe, however, that a separate technique is not needed for each cell in the matrix.

Database architectures

Traditional database management systems (DBMSs) envision a two-level organisation of data: the data as seen by the system and the data as seen by the user, i.e. an end user or a programmer. The definition of data at the first level is often termed (internal) schema and prescribes the organisation of the entire database. The definition of data at the second level is often termed subschema or view and specifies the subset and presentation form of the data to the user.

ANSI/SPARC introduced the *three-schema architecture* for database management systems. This standardisation effort started in 1972. The first framework was published in 1978 (8) and a second report based on this framework was published in 1986 (9). A report from ISO closely related to the ANSI/SPARC framework is the conceptual schema report (10).

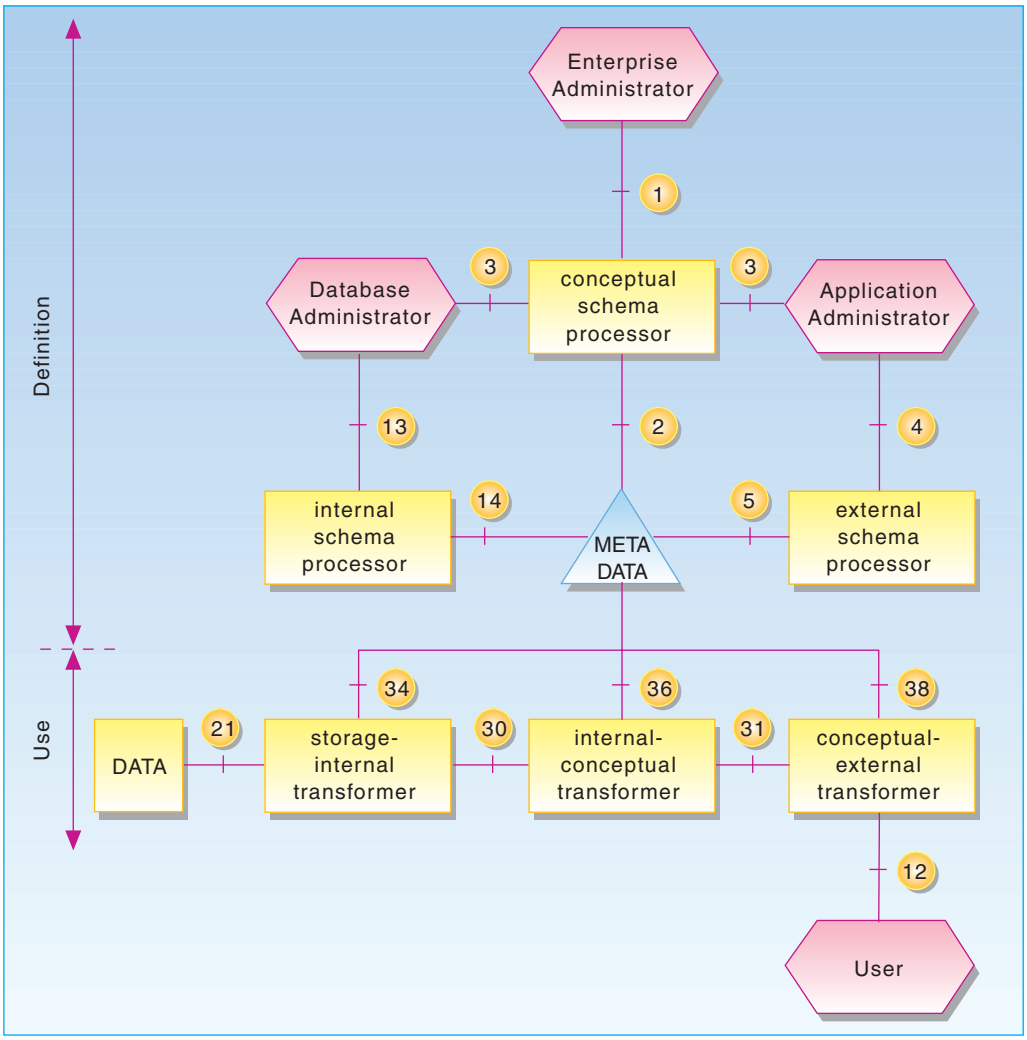


Figure 2 The original ANSI/SPARC three-schema architecture (9)

This figure is a part of the original three-schema architecture from ANSI/SPARC (9), illustrating the relations between the different roles, processors, interfaces, schemata, and data. The external, conceptual, and internal schemata are part of the META DATA. The enterprise administrator, database administrator, and application administrator manage META DATA through schema processors over specific interfaces. The user accesses DATA

ANSI/SPARC focused its work on interfaces, personal roles, processing functions and information flows within the DBMS. In this framework the requirements that must be satisfied by the interfaces, are specified, not how components are to work.

In the three-schema architecture the two levels of organisation of data corresponding to the traditional (internal) schema, and subschema or view, are termed *internal* and *external,* respectively. In addition, a third level is introduced, the *conceptual* level. The conceptual level represents the enterprise's description of the information of interest. This description is recorded in what is called the *conceptual schema.*

The conceptual schema is the core of the three-schema architecture. This schema describes all relevant general static and dynamic aspects, i.e. all rules and constraints, of the universe of discourse. It describes only conceptual relevant aspects. These can among other things be

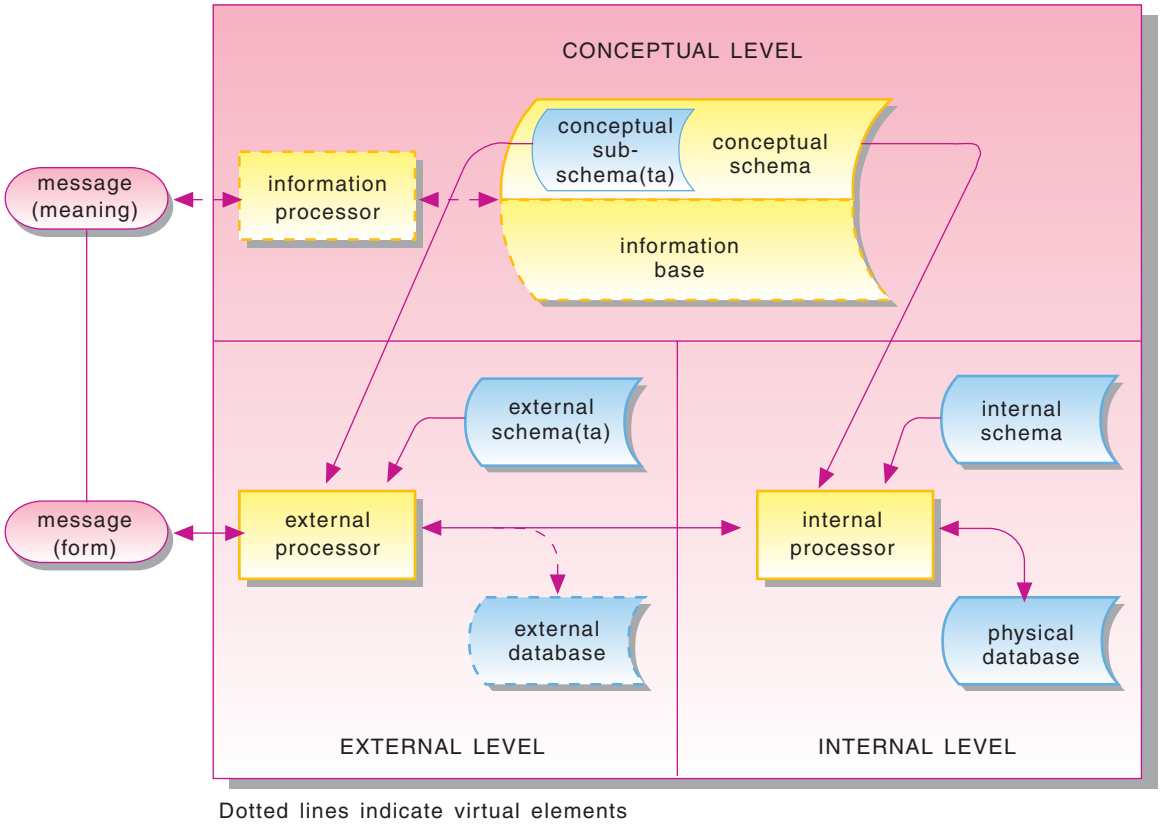


Figure 3 The ISO three-schema architecture (10)

This figure illustrates that each level consists of schema(ta), processor(s), and population(s). A schema prescribes permissible data instances and all rules and constraints concerning these instances. A population contains data instances which satisfy the structure and constraints specified in the corresponding schema. A processor is responsible for enforcing the schema on the population. The populations are here called the information base, the external database and the physical database for the conceptual, external and internal populations, respectively. Only the physical database is a real database.

The conceptual schema comprises all application dependent logic, in opposition to the ANSI/SPARC three-schema architecture where the application logic can be specified outside the external level. The ISO three-schema architecture does not state what is outside the external level

categorised as entities, properties, and relationships.

Several external schemata, representing different user views, will often exist for one single conceptual schema. Both the internal and the external schemata must be consistent with and mappable to the conceptual schema. The three-schema architecture is shown in Figure 1. The original three-schema architecture from ANSI/SPARC is shown in Figure 2, while the ISO three-schema architecture is illustrated in Figure 3.

Introducing the conceptual schema gives some major benefits to database management:

- the concepts are documented only once, in the central conceptual schema
- data independence, i.e. both external views of the data and physical storage of data can be changed without necessarily having to change the conceptual schema.

A schema is itself a collection of data (also called meta data) which can be recorded in a database. Figure 4 shows how the three-schema architecture can be used during both system development and use of database applications. A generalisation of the use of meta data can be found in the newer framework from ANSI/SPARC (9).

The OSI reference model

ISO's reference model for open systems interconnection is called *the OSI reference model* (11). This reference model provides a framework for standardising communication between computer systems. It provides computer users with several communications-based services. Computers from different vendors, conforming to OSI, should be able to communicate with each other.

OSI is a reference model according to the definition given earlier in this paper, i.e. a conceptual framework for understanding communication between computers. The problem of computer communication is partitioned into manageable pieces called layers. Each layer has been subject to standardisation activities, but these standards are not, strictly speaking, part of the reference model (12).

The OSI reference model consists of seven layers. All the tasks or functions involved in communication are distributed between these layers. For communication to take place, functions of several layers have to be combined. Each layer therefore provides services (through functions) to the next higher layer. A service definition specifies the services of a layer and the rules governing access to these services. The service definitions constitute the interfaces between adjacent layers of the OSI reference model.

In (13) two communicating computer systems are called *end systems.* An implementation of services of a layer on an end system is called a *layer entity.* The functionality of a layer is achieved by the exchange of messages between co-operating layer entities in co-operating end systems. The functions of a layer and associated messages are defined in a layer protocol specification. The actions to be taken by a layer entity when receiv-

ing a certain message are defined here, too. A layer protocol constitutes the interface between layer entities of the same layer.

Figure 5 illustrates communication according to OSI.

The client-server architecture

The client-server architecture allows several clients to use a common server, see Figure 6.

A *client* is an active module which has a high degree of autonomy. Nevertheless, it needs to communicate and share resources like information, complex services or expensive hardware. A client often interfaces a human user. A typical example of a client is a workstation attached to a local area network using software located on a central computer called server.

The *server* is a module providing services to one or more clients. The services can be management services as for directory and authentication servers, control of shared resources as for database servers or file servers or user oriented services like electronic mail (14). A server can provide more than one type of service.

The client-server relationship may be viewed as a "master-slave" relationship. The client has the role of the master, requesting services from the obeying slave. Clients may, however, be servers for other clients, and servers may be clients of other servers.

The interface between a client and a server is defined by a communication protocol.

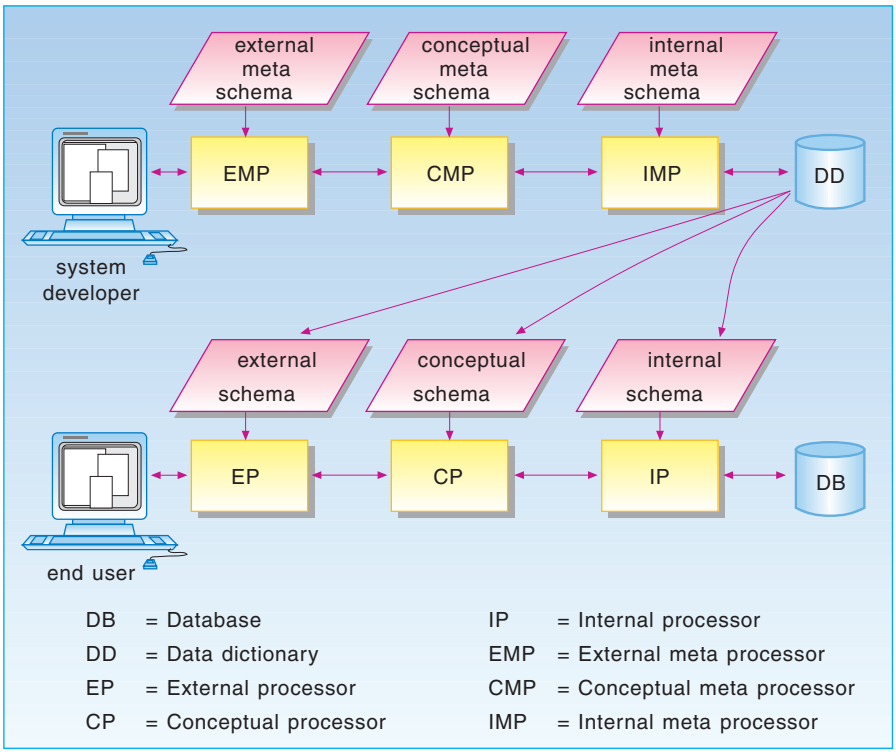


Figure 4 Three-schema architecture used both during development and use of database applications

This figure shows how use and development of database applications can be supported by the three-schema architecture.

The conceptual meta schema specifies all permissible data that can be found in the data dictionary (DD). Internal and external meta schema define the storage form and the presentation form of these data respectively. The system developer works towards the external form of the dictionary data when he specifies schemata for the database application.

Schemata for the final database application are thus stored as ordinary data in the data dictionary. These data may act as schemata for the database application directly or after being compiled into a form appropriate for execution

General features of reference models

The examples show that reference models are useful when dealing with complex problems in the area of information systems. The ISA framework shows that architectures (or reference models) may represent different perspectives of an information system according to the perspectives of the different actors in the system development process.

Reference models help splitting complex problems into smaller parts that are easier to understand and handle. These smaller parts can be layers or components. The parts together make up the total system. The interoperation between parts takes place over interfaces. It is favourable to standardise such interfaces (preferably internationally), because customers will gain flexibility owing to an increasing amount of plug-compatible system components.

Standards often include conformance criteria. These criteria usually indicate the minimum set of services or functions that must be provided at particular interfaces by an implementation of the reference model, in order for the implementation to conform to the standard. In addition a reference model might include performance criteria. The internal implementation of each part of a system is considered a black box, providing the outward behaviour of the system is as required by the reference model.

Why do telecommunications operators need reference models?

Telecommunications operators are usually big organisations, having large administrative and technical information systems. Within both areas the operators will need reference models.

The telecommunications market is now getting more and more exposed to competition. In this market it is important that the operators are not bound to vendor specific systems. The operators will need the possibility to replace com-

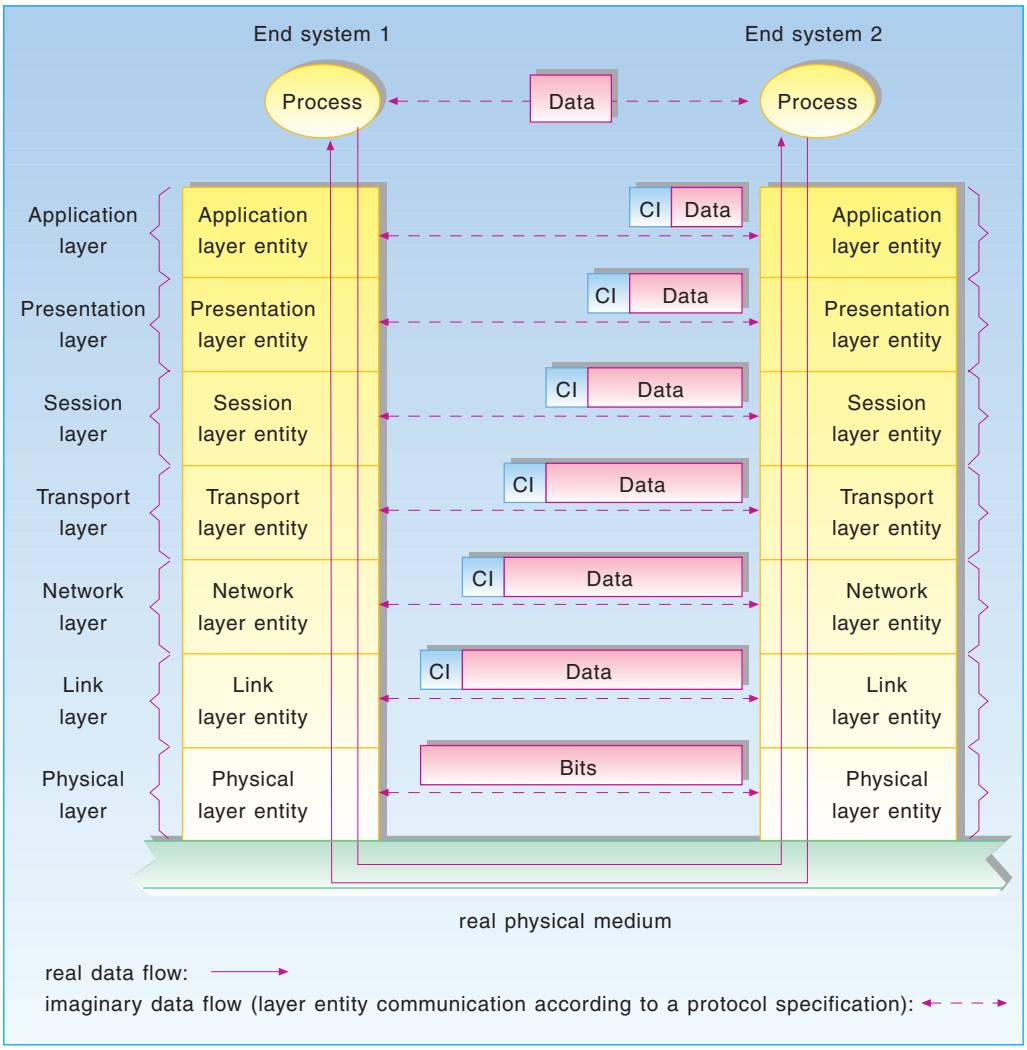


Figure 5 Process communication according to the OSI reference model

The brackets ({ and }) in the figure illustrate that the functionality of a layer is achieved by co-operating layer entities in the same layer. A layer entity is an implementation of a layer's service on an end system (13).

Human users have access to the application layer only. When an end user requests an application layer service, the application layer entity has to co-operate with a corresponding application layer entity in another end system (a peer entity) to carry out the service requested. To initiate this co-operation it transmits a message to this peer entity. To send the message, the application entity uses services provided by its next lower layer, the presentation layer. The local presentation layer entity adds some control information (CI) for use by its peer entity, and uses the services provided by the next lower layer, to send the expanded message. This process continues until the message reaches the physical layer. This is the only layer of the OSI reference model interfacing a real physical medium. Here the message is transmitted to the actual end system. When the message arrives at the remote system, each layer entity in turn peels off the control information from its peer entity to read and act on it. The rest of the message is delivered to the next higher layer

ponents from one vendor with components of another vendor if this vendor has a better component. This is only possible if the different components comply to a common framework, i.e. a reference model.

A reference model can thus be a means for users to build an information system from plug-compatible components possibly purchased from several vendors. The reference model will aid in comparing and selecting information systems and system components, and in reviewing, changing and introducing components into the organisation. Other important benefits gained from using reference models are increased interoperation and data interchange between systems and system components.

Human productivity and training costs will be improved due to reduced training requirements and potentially reduced costs of products. The training costs may be reduced because applications made according to the same reference model will have a number of common features, thereby reducing what has to be learned for each new application.

Generally, telecommunications operators will need, and to some degree are already using, reference models for most of the areas mentioned in this paper. Telecommunications operators have particular needs in finding good reference models for intelligent networks, and operations and maintenance of telecommunications networks and services.

Another important task for telecommunications operators to consider, is how to organise the total amount of data within the organisation. Possible ways of organising the data is according to function (e.g. operations and maintenance), subject (e.g. personnel), chronology (e.g. planning vs. operations systems) or organisation (e.g. divisions), or perhaps a combination of some of these. The issue of organising corporate data is discussed in (15).

Activities on reference models within Norwegian Telecom Research

Norwegian Telecom Research (NTR) has been, and still is, participating in several standardisation organisations in the area of reference models for information systems. In addition, NTR is doing research on reference models and is making products based on existing reference models.

Within the database management systems area the institute has participated in ISO preparing the conceptual schema report (10). Recently NTR prepared a reference model for interoperability for ETIS (16). NTR is also participating in CCITT developing specification techniques. This includes a reference model for human machine interfaces (6).

NTR early developed a tool based on the three-schema architecture. This tool is called DATRAN and is described in (17).

A research project within NTR is preparing a reference model for database applications. This reference model will include three perspectives of database applications: use, development and interoperability. The reference model is based on the three-schema architecture.

NTR early made contributions to the development of reference models for TMN (Telecommunications Management Network). NTR proposed an interface between the TMN system and the network components as early as in 1983 in NT-DV. This proposal has been incorporated in various contributions from NT-DV via CEPT/ ETSI to CCITT. Now this interface is known as Q3.

In connection with the work on TMN done at NTR, a prototype distributed database management system with a four-schema architecture has been developed. This DBMS is called TelSQL (18). A current project is developing another distributed database management system, called Dibas (19), with a more modest approach to distribution than TelSQL.

Challenges and perspectives

Reference models for information systems comprise several challenges for research and standardisation organisations. Reference models exist in several fields, and some fields have many reference models. Some of the reference models belonging to the same field are overlapping, but still different. An example is the IRDS and the 3-schema architecture in the field of data management. Unifying such reference models is a big challenge.

Another challenge is to integrate reference models across fields. Reference models for distributed database systems, for instance, will incorporate elements from reference models for communications and data management. Technical information systems in the areas of Intelligent Networks and Telecommunications Management Network have their own reference models. Integration of reference models from these areas with reference models for administrative database applications and that of commercial CASE tools is another example. Making use of, or taking into account, existing reference models when developing new ones, is another aspect of reference model integration. In an ongoing EURESCOM project, the ODP reference model is examined in order to find out if and how it can be useful for the development of the TMN reference model.

A challenge for standardisation organisations is to make their reference models precise. Due to different views and objectives of the participants of standardisation committees, standards tend to be rather general. A general reference model may lead to different interpretations and thereby incompatibility of components of different vendors, even if the components confirm to the same standard.

Unified, integrated and standardised reference models will simplify education and systems development in the long run. When a reference model for the system to be developed is chosen, everyone involved immediately gets a lot of information and has a common understanding of the systems architecture, or at least parts of it. Standardised reference models will lead to standardised ways of developing systems as well, and will thereby simplify education and training of system developers.

To make reference models the valuable tool they can be, another challenge has to be dealt with. This is the challenge of making the information technology communities aware of all the benefits of standardised reference models, and to make them use such reference models in a greater extent than today.

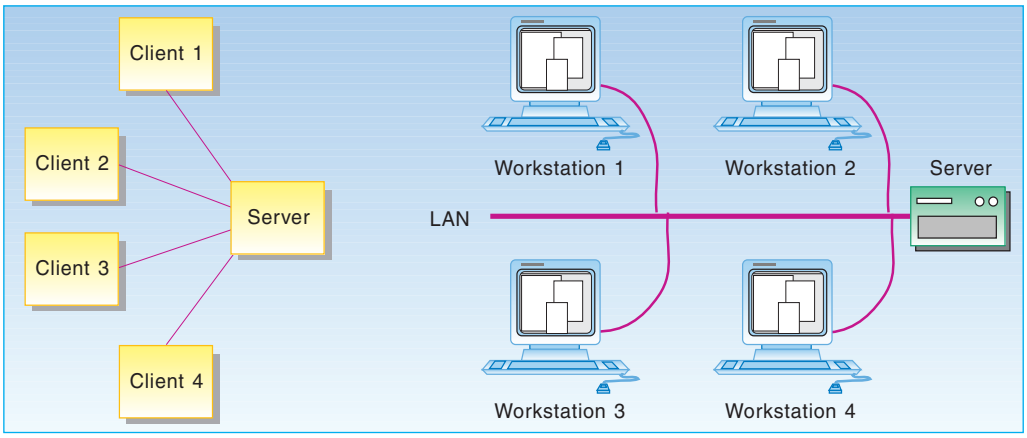


Figure 6 The client-server architecture

A logical model of the client-server architecture (left) and a possible physical implementation of this architecture in an office environment (right). The example shows workstations as clients supported by a server. The clients communicate with the server over a local area network (LAN)

Abbreviations

ANSI/SPARC
: American National Standards Institute / Standards Planning and Requirements Committee

CCITT
: The International Telegraph and Telephone Consultative Committee

CEPT/ETSI
: The European Conference of Postal and Telecommunications Administrations / European Telecommunications Standardisation Institute

ETIS
: European Telecommunications Informatics Services

EURESCOM
: European institute for research and strategic studies in telecommunications Gmbh.

ISO
: International Standardisation Organisation

NT-DV
: Nordtel O&M, i.e. the Nordic tele conference – operations and maintenance

References

1 Fong, E N, Jefferson, D K. Reference models for standardization. *Computer standards & interfaces,* 5, 93-98, 1986.

2 Ozsu, M, Valduriez, P. *Principles of distributed database systems.* Englewood Cliffs, Prentice-Hall, 1991. ISBN 0-13-715681-2.

3 Zachman, J A. A framework for information systems architecture. *IBM systems journal,* 26, 276-292, 1987.

4 Skolt, E. Standardisation activities in the intelligent network area. *Telektronikk,* 88 (2), 43-51, 1992.

5 Wolland, S. An introduction to TMN. *Telektronikk,* 89 (2/3), 84-89, 1993 (this issue).

6 Meisingset, A. The draft CCITT formalism for specifying human-machine interfaces. *Telektronikk,* 89 (2/3), 60-66, 1993 (this issue).

7 Sowa, J F, Zachman, J A. Extending and formalizing the framework for information systems architecture. *IBM systems journal,* 31, 590-616, 1992.

8 Tsichritzis, D, Klug, A (eds). The ANSI/X3/SPARC DBMS framework. *Information systems,* 3, 173-191, 1978.

9 Burns, T et al. Reference model for DBMS standardization. *SIGMOD RECORD,* 15 (1), 19-58, 1986.

10 Griethuysen, J J van (ed). *Concepts and terminology for the conceptual schema and the information base.* ISO/TC97, 1982. (ISO/TC97/SC5 - N 695.)

11 ISO. *Information processing systems – open systems interconnection – basic reference model.* Geneva 1984. (ISO 7498.)

12 Tanenbaum, A S. *Computer networks.* 2nd ed. Englewood Cliffs, N.J., Prentice-Hall, 1989. ISBN 0-13-166836-6.

13 Henshall, J, Shaw, S. *OSI Explained, end-to-end computer communication standards.* New York, Wiley, 1988. ISBN 13-643404-5.

14 Svobodova L. Client/server model of distributed processing. In: *GI/NTG-Fachtagung 'Distributed database systems'.* Heidelberg, Springer Verlag, 1985 (Informatik Fachberichte, 95) 485-498.

15 Inmon, W H. *Data architecture: the information paradigm.* 2nd ed. Wellesley, QED Information Sciences, 1992. ISBN 0-89435-358-6.

16 Meisingset, A. A data flow approach to interoperability. *Telektronikk,* 89 (2/3), 52-59, 1993 (this issue).

17 Nordlund, C. The DATRAN and DIMAN tools. *Telektronikk,* 89 (2/3), 104-109, 1993 (this issue).

18 Risnes, O et al. *TelSQL. A DBMS platform for TMN applications.* Kjeller, Norwegian Telecom Research, 1991. (Internal project note.)

19 Dahle, E, Berg, H. Dibas – a management system for distributed databases. *Telektronikk,* 89 (2/3), 110-120, 1993 (this issue).

Formal languages

BY ASTRID NYENG

681.3.04

Abstract

This paper gives an introduction to formal languages used to develop, maintain and operate computer based information systems. Historically, computer languages have developed from low level machine dependent languages to high level specification languages, where the system developer is freed from the worry of how things are executed by the computer. More emphasis is put on how to make programming feasible for non programmers, for instance by the introduction of visual languages. As systems grow in complexity, languages become central tools to manage this complexity, to provide overview and reduce the amount of programming needed. All languages are based on certain ideas that will influence what the developed system will be like. Research is needed to establish more knowledge about how different kinds of languages influence the finally developed system. Also, there is a need for a framework to be used when comparing and evaluating languages. This framework should give a rationale for selection of evaluation criteria as well.

1 What is a formal language?

The term language is often encountered in the field of software engineering. How the term is defined depends on the context in which it is used. In IEEE's Standard Glossary of Software Engineering Terminology (1) a language is defined as "A systematic means of communicating ideas by the use of conventional signs, sounds, gestures, or marks and rules for the formation of admissible expression." This definition does not state whether the communication takes place between human beings, between human beings and computers, or between computer programs. In the same glossary the term computer language is defined as "a language designed to enable humans to communicate with computers". The first definition is closer to what is often called natural languages, i.e. languages like Norwegian and English, while the second definition is closer to the notion of a programming language or a man-machine language. This sharp distinction between natural languages and computer languages has, however, been blurred. It is now possible to communicate with a computer in a fairly "natural" way, either through natural written languages or even by means of speech. To call a language formal, however, we require that the language has a defined syntax and semantics. By syntax we mean the grammatic rules, which determine whether a sentence is a permissible sentence in the language. Semantics determine how one interprets these well-formed sentences, i. e. the semantics define how an (abstract) machine will execute and represent the sentences written in the language.

In this paper we will concentrate on formal languages used to develop, maintain and operate a computer based information system.

The reader should note that this is only an introduction to formal languages. The theme is a very broad one, and the literature on the topic is overwhelming.

2 History and fundamentals

All computer languages are based on more or less explicitly stated models of the computer. These models influence the language constructs offered to instruct the machine. Figure 1 shows how the machine can be viewed as a black box where input data and a program are given to the machine and the machine produces output data. In this chapter we will mention some of the important machine models, which influence the design of programming languages.

An initial attempt at modelling a computer begins with the observation that at any time the machine is in a certain state and only changes this state as it processes input (2). This is the so-called finite state machine. Based on the current state and input, the machine produces a new state and possibly an output. The specification language SDL (3) assumes this machine model.

The basic von-Neuman machine is the model for the overwhelming majority of computers in use (4). This computer model comprises a processor which executes instructions and a memory which holds instructions and data. One instruction is executed at a time, in sequence. The von-Neuman model has influenced most of the current programming languages, like Pascal and C.

Languages based on logic offer the user a quite different machine model. The machine (the inference engine) can systematically search for a solution of a set of logical equations. The programmer does not have to state how a solution shall be found. He only states the requirements to the solutions. PROLOG is a language which assumes this machine model (5).

Today people program their computers differently from how they used to several years ago. This development is of course influenced by the development of machine models and the ability to abstract away from how the computer works. The Figures 2a-2f illustrate steps in the development. In the mid-1950's the computers had to be instructed by means of *assembler code,* i.e. a kind of programming language which corresponds closely to the instruction set of a given computer, but allows symbolic naming of operations and addresses, see Figure 2a.

The so-called *high level programming languages* (3 GL) emerged in the early 1960's. These languages require little knowledge of the computer on which a program will run and can be translated into several different assembler or machine languages. Fortran and Pascal are examples of high level languages. See Figure 2b.

Functional programming emerged in the late 1960's. The idea was to free the programmer even further from the worry of the steps a computer has to perform. The essence of functional programming is to combine functions in expressions to pro-

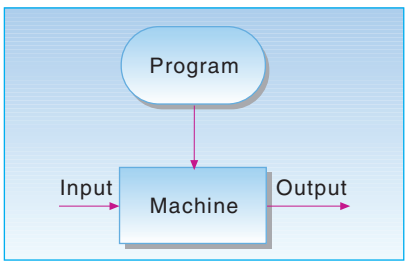


Figure 1 Abstract machine
The computer is a black box which provides output from given input and instructions (program)

```
READ R1, [BP].DATE1        % read from stack
READ R2, [BP].DATE2
SUB R1, R2                 % compare the dates
CMP R1, 10
JG FOUND                   % DATE1 - DATE2 > 10?
                           % if greater, jump to FOUND
.....
FOUND: .....
```

*Figure 2a Mid 1950s, **assembler code**; talking to a computer about registers and jump addresses*

```
( defstruct    (PATIENT)
               (NAME nil) (NO nil)
               (DATE-DISCHARGED nil)
               (DATE-ADMITTED nil))
( setq DATABASE (list      (make-patient ...)
                           (make-patient...)...))
(defun select-patient (database)
 (cond ((null database); empty
       nil ); result if empty
       ((>             ; match
        (-
         (patient-date-discharged (car database))
         (patient-date-admitted (car database)))
        10)
        (cons          ; result if match
         (patient-no (car database))
         (select-patient (cdr database )))
        )
       (t              ; otherwise
       ; result if no match
       (select-patient (cdr database ))))
```

*Figure 2c Late 1960s **functional programming,** using LISP; programs are written by constructing complex functions from more elementary ones*

```
%% The database is defined to be a set of facts,
%% PATIENT (NAME, PATIENTNO,
            DATE-ADMITTED, DATE-DISCHARGED)
person (johnson, 1001, 930115, 930126).
person (smith, 1002, 930116, 930128). %% and so on ...
all(X) :- output(X), fail.
list X,Y :- Y, all(X). % for all solutions of Y, output X.
:-list PATIENTNO, person(NAME, PATIENTNO, DATE1,
            DATE2),
       DATE2 - DATE1 > 10.
```

*Figure 2d Late 1960s, **logic programming**; rules and queries are stated as a set of facts (logical equations)*

```
database:   RECORD
            nrentries: 0 ... max
            patients.ARRAY [1...max] OF
            RECORD
            name: nametype;
            patientno: notype;
            dateadmitted: data;
            datedischarged;
            END;
            END;
WITH database DO
   REPEAT
      WITH patients[index] do
         IFdatedischarged - dataadmitted > 10 THEN
         WRITELN (patientno);
         index := index + 1;
   UNTIL index > nrentries;
```

*Figure 2b Early 1960s, **high level programming language** (PASCAL) allowing data structures and operations to be expressed in a more machine independent way*

```
CREATE TABLE PATIENT
   ( NAME CHAR(30),
   PATIENT-NO INTEGER, ...)
SELECT PATIENT-NO
   FROM PATIENT
   WHERE    (DATA-DISCHARGED
            - DATE-ADMITTED > 10)
```

*Figure 2e 1970s, **Fourth Generation Language** (4 GL); SQL is used to query the database in a nonprocedural way*

```
NEWTYPE Patient
   STRUCT
            Name: CharString;
            No: Integer;
            DateDischarged: Integer;
            DateAdmitted: Integer;
ENDNEWTYPE;
NEWTYPE PatientDatabase;
   ARRAY (Patient, Integer);
   ADDING
   OPERATORS
      moreThanTenDays:PatientDatabase->Boolean;
   AXIOMS
      FOR ALL p IN PatientDatabase
         moreThanTenDays(p) ==
            p!DateDischarged - p!DateAdmitted > 10;
ENDNEWTYPE;
```

*Figure 2f 1980s, **Formal Description Techniques;** SDL Abstract Data Types expressing database operations as operators on abstract types*

Figure 2 Steps in the history of computer languages

vide more powerful functions. See Figure 2c.

In the 1970's the idea of abstracting from the underlying, low level machine architecture was further developed in *logic languages.* The idea was to allow the programmer to use languages from mathematical logic to state the problem and then ask for a solution. See Figure 2d.

In the late 1970's the so-called *Fourth generation languages* (4 GL) emerged. A fourth generation language is primarily designed to improve the productivity and to make programming feasible for non-programmers. Features typically include a database management system, query languages (see Figure 2e), report generators, and screen definition facilities. Actually, it is the user interface to these tools that are called fourth generation languages. This means that all tools that support the developer in one or another way in the development and hiding the programming languages (3 GLs) from the developer, can be said to have a fourth generation language.

All languages introduced so far are traditionally used in the implementation phase of the development of a system. *Specification languages,* which emerged in the 1970's, are most often used in the analysis and design phase of the system development. They are used to produce requirements and design specifications without going into detail on how things will be implemented.

In the late 1970's and especially in the 1980's a new term appeared; the *Formal Description Techniques* (FDTs). An FDT is a special kind of specification language. The idea behind these languages was to enable specifications of systems which were unambiguous, clear and concise. These specifications can form a basis for transformations into lower level languages and implementation. Examples of languages counted to belong to FDTs are SDL (3), LOTOS (6), and Z (7).

3 Classification of languages

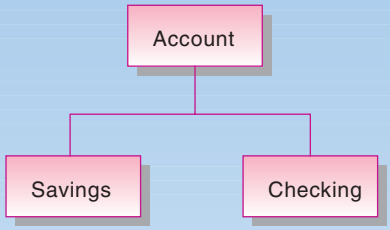
The historical perspective presented above is one possible way of classifying languages, that is, by time of appearance (generations). (8) suggests alternative criteria for classifying programming languages according to:

- the degree to which they are dependent on the underlying hardware configuration (low level vs. high level)
- the degree to which programs written in that language are process-oriented (procedural vs. declarative)
- the data types that are assumed or that the language handles most easily (lists, functions, strings, etc.)
- the problem area which it is designed for (special purpose vs. general purpose)
- the degree to which the language is extensible
- the degree (and kind) of interaction of the program with its environment
- the programming paradigm most naturally supported (style of problem solving; imperative, functional, object oriented, etc.).

It is important to note that the above classification scheme only covers programming languages. (9) suggests a different categorisation of languages which covers a broader scope:

- machine languages (assembler)
- system programming languages (3 GLs)
- languages for analysis and simulation
- languages for communication between end users and computers
- professional languages of end users (4 GLs).

(9) says that "if an analysis and simulation language is formalised to the extent that upon translation to a systems programming language ambiguity and errors are unlikely, we say that it is a specification language". It is important to note that such languages can make the choice of programming language less important as far as systems development is concerned. The specification language can be the only language seen by the developer. This should be taken into account when discussing what categories of languages it is important to focus on in the future.



Class Hierarchy for accounts

```
class name Account
superclass Object
instance variable names accountNumber accountOwner
                        balance rate
instance methods
deposit: amount
        balance:=balance+amount
withdraw:amount
        .....
check: amount
        self withdraw: amount


class name Savings
superclass Account
instance variable names <none>
check: amount
        error: "You  can't write checks against this account"
```

Figure 3 Object oriented programming (SMALLTALK)
Special kinds of accounts are specified as subclasses of the general class Account. The class Savings inherits all the instance variables and methods (often called services) from its superclass Account. The method 'check' is redefined to cover what is special for a Saving account

4 Object oriented languages

The last years many existing languages have been enhanced with object oriented features. Although object orientation is regarded as a rather new area, the first object oriented language Simula (10) was developed at the University of Oslo as early as 1967. The basic idea behind object oriented languages is that programs/specifications are organised into classes of objects which can be arranged in a hierarchy, the most general ones on the top, the more specific ones on the leaves. The specialised classes of objects inherit properties from the more general classes of objects. An object has an interface which specifies what services this object can offer other objects; the

internal aspects of an object are hidden or encapsulated from the other objects. See Figure 3. In the beginning, object orientation was only applied in programming languages. Today we see that specification languages (see below) are enhanced with object oriented features.

5 Specification languages

The focus in language research has shifted from programming languages to what is often called specification languages. There is no exact border between these two categories of languages. However, it is often said that a specification language is used to define a system at a higher abstraction level. A programming language is executable by a machine, while this is not always the case for specification languages. A specification language is primarily used in the analysis or design phase of the system's life cycle.

Specification languages can be divided into two groups, informal specification languages and formal specification languages. The informal ones have a long history in the field of software engineering. They are mainly graphical techniques with a precise syntax such that they can be handled by a tool, but most often have loosely defined semantics, i.e. it is not clear how a program should work based on these specifications. Two diagram types are often used; entity-relationship diagrams and data flow diagrams.

Entity relationship diagram (ER-diagram) is a popular graphic notation for showing data entities, their attributes, and the relations which connect them. They are suitable for showing certain kinds of constraints, but usually they cannot show all constraints, and they ignore the operations performed by and on the entities. ER-diagrams cannot be used to represent actual instances of data in the database (11). The diagram in Figure 4 expresses that persons belong to organisations, one cannot express that a specific person is an employee at a specific organisation.

Data flow diagrams show the processes, data stores, and data flows into and out of the processes and the system, see Figure 5.

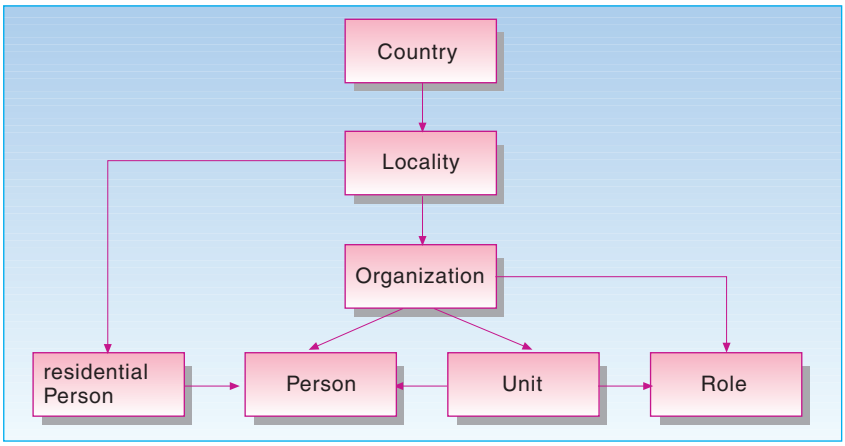


Figure 4 Graphical specifications of data definitions (ER-diagram)
The example specifies the data model for a small directory. The graph depicts classes of entities (boxes) and relations (arrows) between them. The notation is a variant of the original ER notation

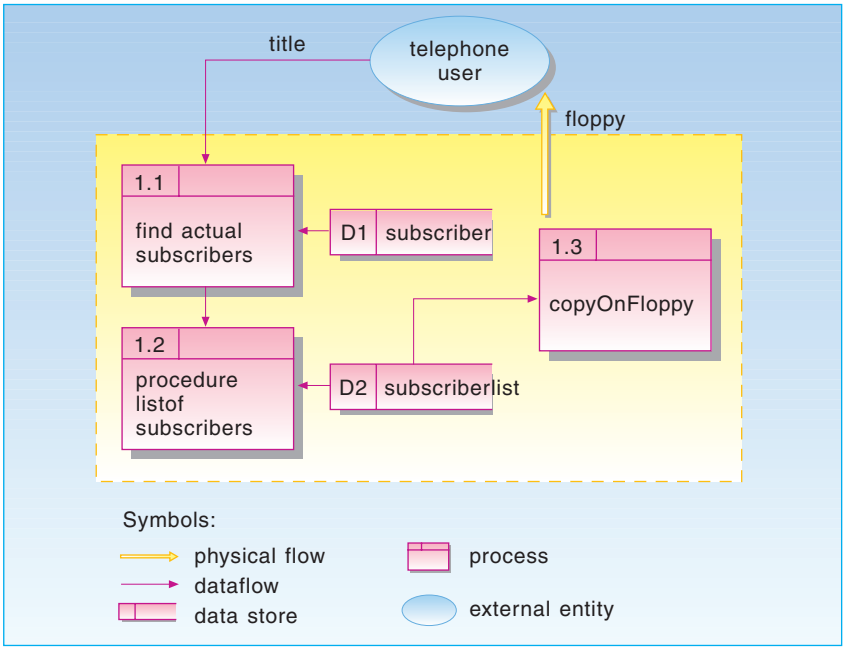


Figure 5 Data flow diagram
An example of a data flow diagram for a directory system. The graph shows how data flows between processes, data stores and external entities. The dashed line is the system border

6 Formal specification languages

Formal specification languages are specification languages whose vocabulary, syntax, and semantics are formally defined (12). The vocabulary gives the finite set of symbols that can be used to specify sentences according to the rules of combining the symbols (syntax rules).

Possible advantages of using a formal language for specifying the design of a system are (12):

- it may be possible to prove that a program conforms to its specification
- it may be possible to animate the behaviour of a formal system specification to provide a prototype system

```
<textual block reference> ::=
    block <block name> referenced
<end>
<end> ::= ;
```

Figure 6 Example use of BNF
The BNF meta language allows strings to be replaced by strings in a hierarchy. The finally produced strings are called terminals, the rest are non terminals. Non terminals are represented as strings surrounded by '<' and '>', e.g. <end>. A production rule for a non terminal symbol consists of the non terminal symbol on the left hand side of the symbol '::=', and one or more non terminals/terminals on the right hand side. Above, block *and* referenced *are terminal symbols*

```
Newtype Bool;
    literals true, false;
    operators
        not: Bool -> Bool;
    axioms
        not (true) == false;
        not (false) == true;
Endnewtype;
```

Figure 7 An SDL ADT
Abstract data types define the result of operations on data objects without constraining how this result is obtained.

Bool is a new sort with true and false as values. The operator part gives the signature of the not operator. In the axioms part the relations between not and the literals true and false (which are operators without arguments) are defined

- formal software specifications are mathematical entities and can be studied and analysed using mathematical methods
- the specification may be used as a guide to identify test cases.

Formal specifications should be regarded as complementary to more informal specification techniques. End users will probably have problems when reading formal specifications and need other means of getting an idea of whether the specified system will meet their needs or not. Formal specifications are more important for other user groups, for instance those who are going to implement the system. They need to know exactly what are the requirements to the system in order to ensure that they implement the right system.

It is convenient to decompose a language definition into several parts (13):

- the definition of the syntax rules
- the definition of the static semantic rules (so-called well-formedness conditions) such as which names are allowed at a given place, which kind of values are allowed to be assigned to variables, etc.
- the definition of the semantics of the constructs in the language when they are interpreted (the dynamic semantics, defining the abstract behaviour of the computer).

In the definition of a formal specification language all parts are specified in another language, a meta language (in contrast to informal languages where the semantic is only stated in natural language). A meta language is a language suited for defining other languages. An example of such a language is the BNF (The Backus Naur Form). Figure 6 illustrates how BNF is used to specify a part of the textual grammar of SDL.

As far as definition of semantics is concerned, there are different approaches. It is far beyond the scope of this paper to treat this subject in any detail. However, we will briefly mention three approaches (14):

- *operational semantics,* where the meaning of language constructs are specified by the computation it induces when it is executed on a machine
- *denotational semantics,* where the meanings are modelled by mathematical objects that represent the effect of executing the constructs
- *axiomatic semantics,* where specific properties of the effect of executing the constructs are expressed as assertions.

The formal definition of the static and dynamic semantics of the CCITT specification language SDL is written in Meta-IV (15) and defines a denotational semantics.

There are several categories of formal specification languages. Here we will only introduce two kinds; Algebraic Specification and Model based Specification.

Algebraic specification is a technique whereby an object is specified in terms of the relationships between operations that act on that object (12). The relationships between operations that act on the types are stated by equations. The equations are used to indicate which terms denote the same value. A term is an application of an operator on an argument. In the example in Figure 7 it is stated that the application of not true produces the same value as false. Without these equations, all terms would be regarded as different values of the type.

An algebraic specification consists normally of four parts, see Figure 7:

- the sort of the entity being specified (Bool in Figure 7)
- an informal description of the sort and its operations
- a signature part, where the names of the operations on that object and the sort of the parameters are provided (operators in Figure 7)
- an axiom part, where the relationships between the operations are defined (axioms in Figure 7).

An algebraic specification is regarded hard to read for non mathematicians, and even harder to write. The main problem is to know when you have given enough axioms to be sure that all operations are properly defined.

The CCITT language SDL has adopted the algebraic approach for its data types. However, for reasons indicated above, this part of the language is not much used.

Model based specification is another approach chosen in some languages. Model based specification is a technique that relies on formulating a model of the system, using well understood mathematical entities such as sets and functions. Operations are specified by defining how they affect the overall system model (12).

One of the best known model-based techniques is Z (7), which is based on set theory.

7 Visual languages

While formal specifications are regarded to be suitable for highly trained people, the idea behind *visual* languages is to make formal languages feasible for non programmers. (16) gives a good tutorial on visual programming. Here we will only briefly introduce the basic notions.

A visual language can be defined as "any system that allows the user to specify a program in a two- (or more) dimensional fashion" (16).

In (16) four possible positive properties of visual languages are listed, which may explain why these languages are more attractive to non programmers than traditional programming:

- pictures can convey more information in a more concise unit of expression
- pictures can help understanding and remembering
- pictures can make programming more interesting
- when properly designed, pictures can be understood by people regardless of what language they speak.

(16) introduces three broad categories of visual programming languages, *diagrammatic* programming languages, *iconic* programming languages, and *form oriented* programming languages.

Diagrammatic programming languages are based on the idea of making charts executable. They are most often a supplement to a textual representation. The specification language SDL can be characterised as a diagrammatic language. SDL used at a low level of abstraction is executable. It is possible to use the graphical syntax as a kind of a flow chart for the code. SDL also has a textual syntax. However, the graphical form is the main reason why the language has attracted large user groups.

Iconic languages use icons to represent objects and actions. Icons are specifically designed to be essential language elements playing a central role. The Xerox Star system (17) is an example of an iconic system where icons and pointing devices are used as a means to communicate with the computer. User interfaces using icons are now commonplace in many systems.

Iconic programming languages go a step further than diagrammatic languages in making programming visual. Programs are written by selecting icons denoting the various operations needed and placing them in the program area of the screen. These icons are then connected by paths to denote the flow of control. Iconic programming appears to be more exciting for novice programmers than text based programming. It is, however, doubtful whether it will increase productivity in long terms.

Form oriented visual programming languages are more to be compared with graphical user interfaces where the user "program" the computer by filling in forms. They are designed for people who have no desire to learn how to program. Database query is one application area where the form oriented approach is used. QBE (Query By Example) is a form language supporting relational databases (18). Spreadsheet is another example.

8 Standardisation of languages

Much work is being done on standardisation of languages and their environments. Standardised languages may attract more users and make it worthwhile for vendors to develop tools supporting the languages. There are, however many examples of languages which have not been successful despite the fact that they have been issued as an international standard. Here, we will give an overview of the language work in CCITT (International Telegraph and Telephone Consultative Committee). CCITT is recommending languages in the area of telecommunications.

CCITT Study Group X has been assigned the responsibility for methods and languages. This group should support the other groups with needed expertise on a few recommended languages. CHILL (CCITT High Level Language) is recommended as a high level programming language, SDL as a specification language (FDT). SDL is used to specify recommendations in certain application areas, e.g. ISDN. Even if SDL provides a high level of formality, in practice most recommendations use it at a rather low level of formality, as a means of communication between humans. SDL has over the last four years been enhanced with object oriented features. Also, SG X has developed a draft recommendation on a Data Oriented Human-Machine Interface Specification Technique.

During the last study period, several languages or notations were developed outside Study Group X. Many groups felt it was needed to develop techniques specially suited for their applications. In the work on Message Handling Systems (X.400, electronic mail) the developers felt a need for a more precise and unambiguous way of defining the data to be exchanged. As a consequence of this need, they came up with ASN.1, Abstract Syntax Notation (19). ISO (International Standardisations Organisation) and CCITT jointly defined the OSI Management Formalism (20), to be used in the field of network management, TMN (Telecommunications Management Network). This formalism is based on using templates for defining managed objects and using ASN.1 to define the attributes. This formalism is more or less directly adopted by those working on electronic directories (X.500) with some adjustments for their specific needs. However, the directory technique has been further developed such that it has become a language on its own.

There is no sign that the work on languages in the standardisation bodies will unify to one common language for all needs. The new application areas such as ODP (Open Distributed Processing)(21) and IN (Intelligent Network) (22) have their specific requirements on languages, although much work is done to investigate whether existing languages meet these requirements. In both areas the developers are asking for object oriented techniques.

9 Why is work on languages needed?

The research activities on languages have not come to an end. The concern is how to develop and use new languages on new application areas. The complexity of applications is rapidly growing and so are the costs of development and maintenance. Languages are central tools to manage the complexity; the system developers need powerful languages that can help them to express different aspects of the system in a straightforward way. Also, the chosen language(s) will influence what the developed system will

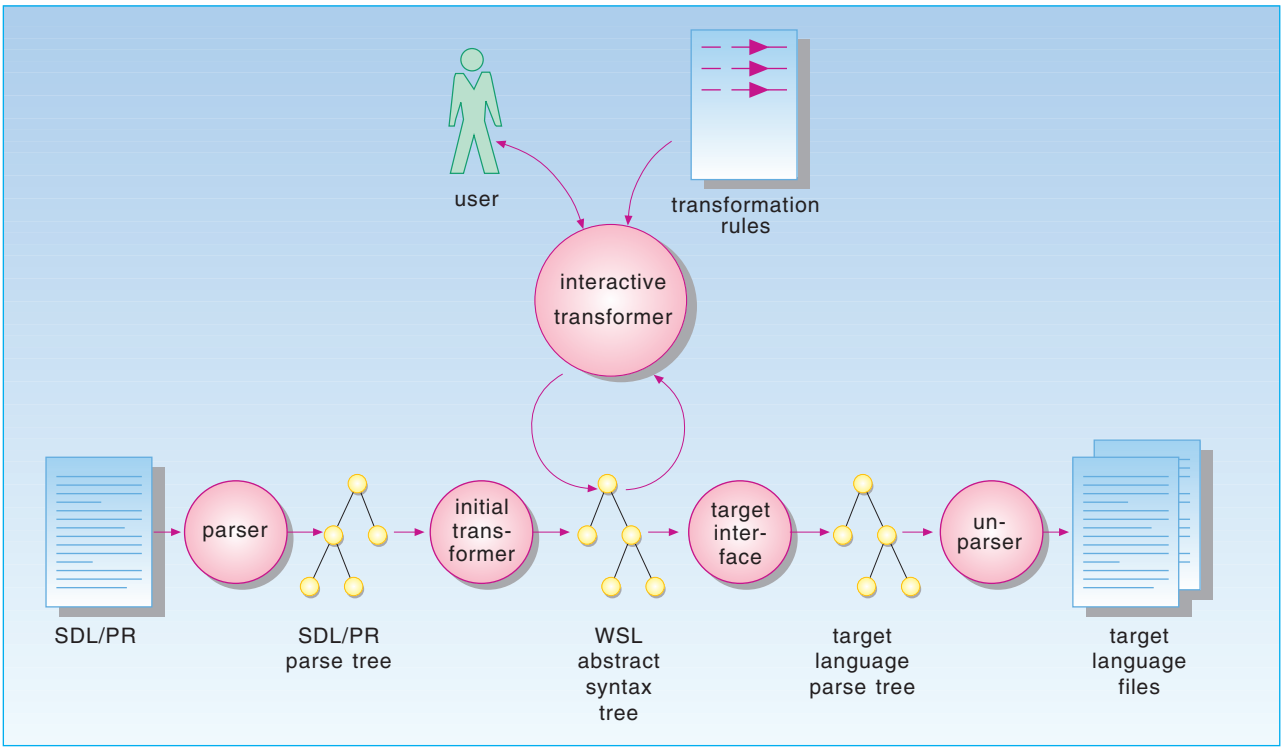


Figure 8 The architecture of the SDL translator
The translator consists of an input part that reads SDL specification providing SDL/PR form and transforms this to an internal form (a wide spectrum language, WSL, not tied to a specific language). The transformation part supports the application of transformation rules to this internal form. An output part dresses the target program in the proper external syntax (C)

be like. All languages are based on certain ideas, which reflect the language designers' point of view on systems and systems development. The usage of a process-oriented language (a language focusing on functions/processes to be performed in the system) and a data-oriented language (focusing on the data to be handled by the system) will most often lead to different systems, even when the requirements from the users are the same. The resulting system will probably be differently structured, have different user interfaces, and different properties as far as maintenance is concerned. The importance of this observation is that system developers need to be conscious about the choice of language, and also research is needed to establish more knowledge about how different kinds of languages influence the resulting system.

The work on languages is tightly connected to the work on reference models. A reference model is a means of splitting up the system into manageable parts where different people can look at the system from different points of view (interfaces). The different interfaces can require different features to be expressed and will put specific requirements on the languages used to define the interfaces.

10 Research activities at Norwegian Telecom Research

Norwegian Telecom Research (NTR) has much experience in using, evaluating and developing techniques for design of data oriented information systems and their human-machine interfaces (HMIs). This work has resulted in the development of the HMI specification technique (23). This technique allows the specification of the data encountered on the screens, using the end-user's own terminology.

NTR has for many years participated in the standardisation of SDL. During the period from 1988 to 1992 a prototype of a translator from SDL to C has been developed. The objectives of the project were both to experiment with new technology, especially transformational programming, and also to develop a useful tool for SDL users. When the project started, there was a rather big group specifying ISDN-services in SDL and implementing them using C. The implementation of the specifications was based on a runtime support system which implemented the basic SDL constructs of finite states machines. SDL was applied in a low level of abstraction, such that translation to C was rather straightforward and possible to automate. It was felt that a code generator could free the developers from much uncreative work. A tool called REFINE (24), was used to develop a prototype code generator. Unlike most code generators, which are more or less like an ordinary compiler, the SDLTranslator offered the possibility to generate different code where different aspects where taken into account. Optimisation on execution speed can lead to a different implementation than if storage is the main concern. REFINE offers a very flexible environment. For instance an SDL grammar and a C grammar could be developed in short time and be used to parse and print SDL specifications and C code, respectively. The parsed SDL

specification was given an internal representation in REFINE's knowledge base and step by step transformed to a C representation. For small specifications the ideas were promising, for large scale projects, however, the performance turned out to be unsatisfactory. Figure 8 shows the architecture of the SDLTranslator.

The 1992-version of SDL has incorporated object oriented extensions to the language. A small project has been working on applying these new features in the specification of Intelligent Network Services. The results showed that types and subtyping made it possible to compose services out of reusable components. The reusable components as defined in the IN-recommendations were, however, not very well designed. The work also illustrated the importance of formal specification of a recommendation. A lot of unclear points where found during the specification work.

In 1992 a project called Systemutvikling 2000 (Systems Development 2000) was initiated by the Information Technology Department. The project was asked to come up with the needed knowledge for making decisions on strategies for what kind of languages, tools, and reference models should be used for systems development in the future (year 2000). So far, the work has focused on how to put requirements on languages. It is felt that existing lists of such requirements (found in for instance recommendations) are more or less unmotivated. There is a lack of rationale behind the selection of evaluation criteria. A central objective for the work is therefore to develop a framework for comparing languages.

References

1 IEEE. *Standard Glossary of Software Engineering Terminology.* New York, IEEE, 1990.

2 Raymond-Smith, V J. Automata Theory. In: *Software Engineer's Reference Book* (ed: J McDermid). Butterworth-Heinemann, 1991, 9/3 – 9/15. ISBN 0-750-61040-9.

3 CCITT. *CCITT Specification and Description Language*, March 1988. (Blue Book, Fascicle X.1-5, Recommendation Z.100.)

4 Ghezzi, C. Modern non-conventional programming language concepts. In: *Software Engineer's Reference Book* (ed: J McDermid). Butterworth-Heinemann, 1991, 44/3 – 44/16. ISBN 0-750-61040-9.

5 Clocksin, W F, Mellish, C S. *Programming in Prolog.* Second Edition. Berlin, Springer Verlag, 1984. ISBN 0-387-15011-0.

6 ISO. *LOTOS – A formal description technique based on the temporal ordering of observational behaviour,* 1988. (ISO 8807.)

7 Spivey, J M. *The Z notation: A reference manual.* Englewood Cliffs, N.J., Prentice Hall, 1987. (International Series in Computer Science.)

8 Weiser Friedman, L. *Comparative Programming Languages: generalizing the programming function.* Prentice-Hall, 1991. ISBN 0-13-155714-9.

9 Schubert, W, Jungklaussen, H. Characteristics of programming languages and trends in development. *Programming and computer software,* 16, 196-201, 1990.

10 Kirkerud, B. *Object-oriented programming with SIMULA.* Reading, Mass., Addison-Wesley, 1989. ISBN 0-201-17574-6.

11 Sowa, J F, Zachman, J A. Extending and formalising the framework for information systems architecture. *IBM Systems Journal,* 31(3), 1992.

12 Sommerville, I. *Software engineering,* third edition. Reading, Mass., Addison-Wesley, 1990. ISBN 0-201-17568-1.

13 CCITT. *CCITT Specification and Description Language, SDL formal definition. Introduction.* March 1988. (Blue Book, Fascicle X.1-5, Recommendation Z.100 – Annex F.1.)

14 Nielson, H R, Nielson, F. *Semantics with applications, a formal introduction.* New York, Wiley, 1992. ISBN 0-471-92980-8.

15 Bjørner, D, Jones, C B. *Formal specification and software development.* Englewood Cliffs, N.J., Prentice-Hall, 1992.

16 Shu, N C. Visual Programming: Perspectives and approaches, *IBM Systems Journal,* 28(4), 1989.

17 Smith, D C et al. The Star user interface: An overview. *Proceedings of the National Computer Conference,* 515-528, 1982.

18 Zloof, M M. Query-by-example, *AFIPS Conference Proceedings, National Computer Conference,* 431-438, 1975.

19 CCITT. *Information technology – Open Systems Interconnection – Abstract Syntax Notation One (ASN.1),* March 1988. (Blue Book, Fascicle VIII.4, Recommendation X.208.)

20 ISO. *Final Text of DIS 10165-4, Information Technology – Open Systems Interconnection – Structure of Management Information – Part 4: Guidelines for the Definition of Managed Objects,* 1991. (ISO/IEC JTC 1/SC 21/ N 63009.)

21 ISO. *Working Draft – Basic Reference Model of Open Distributed Processing – Part 1: Overview and Guide to Use,* 1992. (ISO/IEC JTC 1/SC 21/N 7053.)

22 ETSI. *Intelligent network: framework.* Draft Technical Report NA-6001, Version 2, 14 September 1990.

23 Meisingset, A. The draft CCITT formalism for specifying Human-Machine Interfaces, *Telektronikk,* 89(2/3), 60-66, 1993 (this issue).

24 *REFINE™ User's Guide.* Palo Alto, CA., Reasoning Systems Inc., 1990.

Introduction to database systems

BY OLE JØRGEN ANFINDSEN

1 Introduction

Database technology is currently becoming increasingly important in a large number of areas where computers are used. The goal of this article is to introduce the reader to databases and database management systems; what they are, how they can be used, and why they are important to telecommunication operators.

2 What is a database?

The meaning of this term varies with the context. Any collection of data, such as e.g. a file – electronic or otherwise – or a set of files, could possibly be called a database, see Figure 1. In the context of computers, however, the term has a fairly specific meaning. The following is a possible definition: A database is a persistent collection of data managed by a *database management system* (DBMS). Loosely speaking, a DBMS is an agent whose job it is to manage a database; i.e. handle all database access and make sure the database is never corrupted, see Figure 2. The following section gives a formal definition of what a DBMS is. Unfortunately, no single definition has been accepted as *the* definition of DBMS. The one used below seems to be more precise than any other, though.

3 What is a database management system?

Think of a database *transaction,* or just transaction, as a collection of read and/or write operations against one or more databases. The so-called ACID properties (9) are a set of rules which transactions must obey, and it is the responsibility of the DBMS to enforce those rules – or else it is not a DBMS. One could also say that ACID specifies a certain *behaviour* for transactions. In outline, a DBMS must guarantee that:

- Any not yet completed transaction can be undone
- No transaction is able to violate any integrity constraints defined for the database
- Concurrent transactions cannot interfere with each other
- The effects of a completed transaction will indeed be reflected by the database, even if a hardware, software, or other failure should occur "immediately" after completion.

A more accurate description of the ACID properties follows:

- The "A" in ACID stands for *atomicity.* This provides users with the ability to change their minds at some point *during* transaction execution, and effectively tell the DBMS to "forget" the current transaction, in which case the database will be left as if the transaction never executed at all. A transaction must be atomic in the sense that the DBMS must either accept everything a transaction does, or the DBMS must reject everything; leaving the database unchanged. In other words; a transaction must be an all or nothing proposition. A classic example is a transaction to transfer money from one bank account to another: It involves, among other things, (1) reducing the balance of one account and (2) increasing the balance of the other by the same amount. If only one of the two operations takes place, the customer and/or the bank will be unhappy; the DBMS must ensure that both or no operations take effect.
- The "C" in ACID stands for *consistency.* A DBMS must facilitate (declarative) definition of integrity constraints, and guarantee that they are not violated. That is, a transaction must not be allowed to bring a database into an inconsistent state – i.e. a state where some integrity constraint is violated. This means that any transaction that attempts to do something illegal, must be prevented from doing so. For example, a department must never be shown as being managed by a non-existent employee.
- The "I" in ACID stands for *isolation.* Any two transactions that execute concurrently under the control of a DBMS, must be isolated from each other such that the net result is as if they had been executed one after the other. This is also known as *serialisability,* and is closely related to *concurrency control* mechanisms; the means by which serialisability/isolation is achieved. If transactions were not isolated from each other, results would be unpredictable; e.g. work performed by one user could accidentally be nullified by another concurrent user.
- The "D" in ACID stands for *durability.* When a transaction completes successfully, the DBMS must ensure that all its changes to the database are durable – even in case of power failures and similar events. In other words, a DBMS must make sure that data committed to it, is available until explicitly removed (deleted).

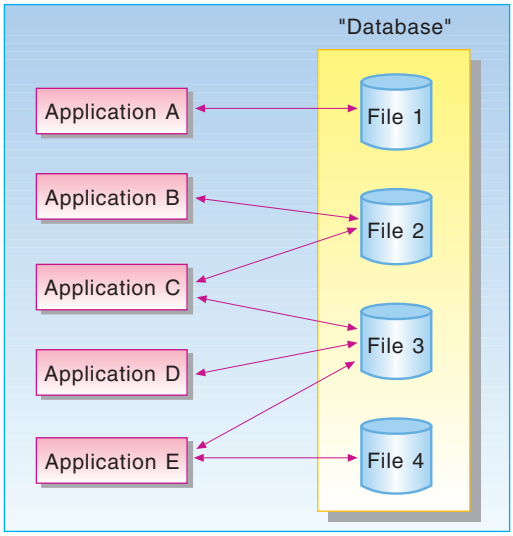


Figure 1 Without a database management system (DBMS) applications access files directly. This typically leads to duplication of data and/or concurrency problems and/or inconsistent data. Applications are not protected from changes in file structures, i.e. little or no physical data independence *is provided. The "database" is just an arbitrary collection of files with no centralised control*

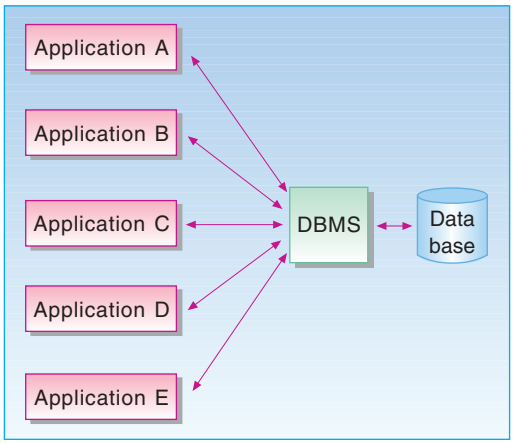


Figure 2 The database is managed by a DBMS. No application is allowed to access the database except through the DBMS. This facilitates data sharing among applications, provides a certain degree of physical data independence, and guarantees ACID properties for all database transactions (this concept is explained in the text). Different applications may have different views *of the database, see* ***figure 3***

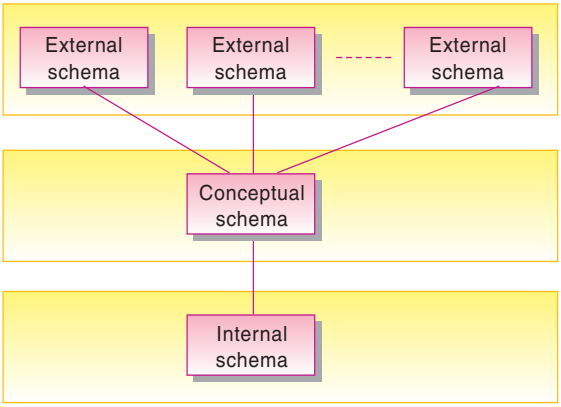


Figure 3 The three-schema architecture. The conceptual schema may be thought of as the *database schema, because it is a* logical *description of the entire database. The internal schema may be thought of as a description of how the conceptual schema is to be* realised *or* implemented, *because it defines the physical properties of the database. The external schemas are* views *through which different applications or users may access the database. An external view could be identical to the conceptual schema, it could be a proper subset of the conceptual schema, it could provide aggregation of data, it could join data instances and present them as a single instance, etc*

Definition: a DBMS is an information storage and retrieval system which provides ACID properties for all database transactions.

This can be summed up – less formally – as follows. A database that is controlled by, and therefore accessed through, a DBMS, is a database that will *behave properly*. Even though the above explanation of the ACID properties might be hard to understand for people who are not familiar with database systems, they define behaviour that is intuitively and obviously desirable.

Any decent DBMS product will have additional features, such as e.g. authorisation control, buffer pool management, parameter tuning, and trace facilities, as well as utilities for back-up, recovery, load, unload, reorganisation, and more.

DBMSs also provide physical data independence and makes data sharing among applications feasible; two extremely significant properties. An important term in this context is *schema,* see Figure 3. Although most people seem to agree that the three-schema architecture is a good idea, few if any commercial DBMS products fully support it; they typically do not clearly separate the internal and conceptual schemas, which has given rise to the ironic term *two-and-a-half-schema* architecture. For the remainder of this article, *schema* should be taken to mean conceptual schema.

4 Current usage of database systems

In this section we mention some of the most common uses of database systems. The term "database system" is often used to denote a DBMS; a DBMS and its database; a DBMS, its database, and the applications that manipulate the database; or even a collection of database systems working together. Thus, the reader should be aware that the exact meaning of the term may vary depending on the context.

Database systems were first developed in the 1960's. They were then mostly used for business applications with large amounts of structured data, typically in the banking, insurance, and airline industries. Today, virtually all large corporations use database systems to keep track of customers, suppliers, reservations, orders, deliveries, invoices, employees, etc.

As database systems became more versatile, powerful, and user friendly, their use proliferated into a growing number of areas. For example management information systems (MIS), decision support systems (DSS), ad hoc query systems, inventory control systems, point of sale systems (POS), and more.

Norwegian Telecom is a large user of database systems. We have been actively involved with database technology for more than twenty years, our first major database system (Telsis) went into production in 1975, and we currently have database systems that cover most of the areas mentioned in the previous paragraphs. Also, information about our telecommunications networks is stored in database systems. And our modern public switches use database systems to process calls, define customer services, store billing information, produce input data to the invoice system, etc. As can be seen from the following section, database technology will become even more important in the future.

5 Emerging areas for database systems

In principle, any system that needs to store and retrieve data could benefit from using a DBMS. This is beneficial because the ACID properties (described above) are always an advantage. It is beneficial because a DBMS has a well defined interface – thus making applications more portable. And it is beneficial because a DBMS, a modern one at least, will allow users to combine data in almost any conceivable way, thus enabling discovery of previously unknown correlations (roaming about in a database looking for "new" information, is frequently called *database mining)*. There are also other advantages of using a DBMS. However, the functionality and/or the performance characteristics offered by current DBMSs, are inadequate for a number of application areas.

Research and industry are currently working hard to provide database systems that will meet the needs of the following application areas: Computer aided design (CAD), computer aided manufacturing (CAM), computer aided engineering (CAE), computer aided software engineering (CASE), computer integrated manufacturing (CIM), geographical information systems (GIS), multimedia systems, process control systems, text search systems, word processors, office support systems, program development environments, and more. In particular, these application areas need support for:

- User defined types, e.g. Employee, Vehicle, or any other data type describing entities within an application domain
- Complex objects, i.e. objects that are nested inside other objects, to an arbitrary depth
- Versioning, i.e. the ability to let several versions of given objects exist simultaneously, and co-ordinate their use
- Long transactions, i.e. transactions that may last for days, weeks, or months, rather than seconds or minutes.

Other challenges come from new application areas that will depend on database technology. Of particular interest to telecommunication operators, are intel-

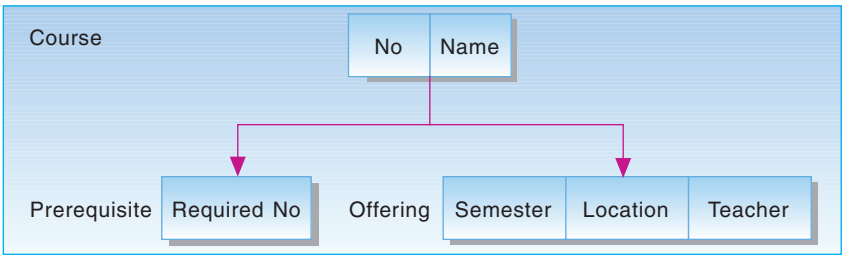


Figure 4 The course-prerequisite-offering-teacher structure represented by a single hierarchy. The two arrows indicate pointers from Course segments to Prerequisite and Offering segments. This means that it is possible to navigate from a given Course to all its dependent Prerequisites and Offerings. However, this does not preclude the presence of other pointers, e.g. a pointer from every Offering to its parent Course

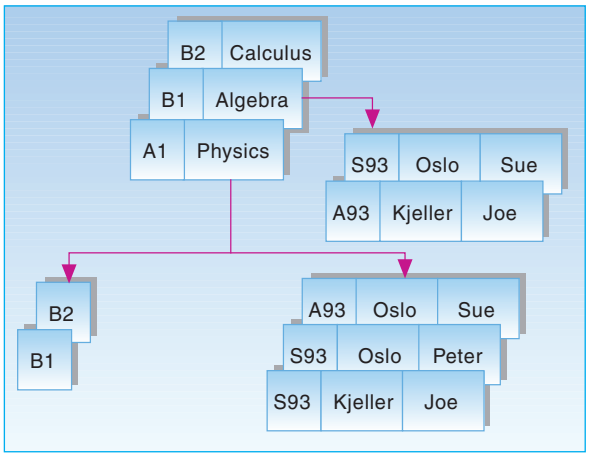


*Figure 5 Occurrence diagram for the schema of **figure 4***

ligent networks (IN) and telecommunications management networks (TMN). Database systems that are to be integrated with public networks, must have scalable capacity, low response times, and high availability – way beyond that of today's commercially available systems. The HypRa project, primarily sponsored by Norwegian Telecom Research and carried out at the Norwegian Institute of Technology (NTH) and SINTEF/
DELAB, has produced interesting results in these areas. Further, it is conceivable that new telecommunications services will evolve that rely on database systems. These could be run by the public network operator (PNO) or by some third party. They could contain information that users would want to access, or they could contain information that would enable the service in question.

Yet another research field is that of *distributed* database systems, i.e. where two or more DBMSs co-operate to form a single database system. The DBMSs in question may run on separate computers, possible geographically remote. See the article on Dibas elsewhere in this issue.

6 Kinds of DBMS

Traditional DBMSs are usually given one of the following three labels: hierarchic, network, or relational. These are not the only ones, but they are by far the best known. Two new kinds of DBMS that are beginning to have an impact on the marketplace, have the label object oriented and/or extended relational.

6.1 Hierarchic

A hierarchic – or hierarchical – DBMS (HDBMS) stores data in hierarchic structures (also known as trees), with root nodes and dependent nodes. A root node may have zero or more children, while a dependent node has exactly one parent. The trees may be arbitrarily deep, i.e. children may have children, and so forth. Also, trees need not be balanced, i.e. siblings can be root nodes in unequally sized subtrees. The archetype of an HDBMS is clearly the IBM product IMS, also known as IMS/DB, and sometimes even referred to as DL/1. The latter stands for Data Language 1, which is the language used to access IMS/DB. The product has been around since 1968, and used to be *the* top mainframe DBMS both in terms of number of installations and also in terms of the number and size of applications that depend on it. It is probably fair to say that most major corporations in the world – those who where major during the 1970's at least – use IMS/DB.

For illustration purposes, a classic database example of courses, prerequisites, offerings, and teachers will now be used. A possible hierarchic schema for such a database is shown in Figure 4. The root node in this tree structure is Course, while Prerequisite and Offering are sibling dependent nodes. Figure 5 contains an occurrence diagram.

An obvious weakness with the chosen structure is that teacher name appears as a field inside the Offering segments. This has disadvantages, in particular:

- Storing redundant data. Assuming that a teacher may be involved in more than one course offering in a given semester, and that offerings from several semesters are stored in the database at any time, the same teacher names will occur multiple times. If the database is to contain not only the *names* of teachers, but also their employee numbers, addresses, phone numbers, birth dates, etc., then such a schema soon becomes unacceptable.

To solve this problem one needs to create a *separate hierarchy* with Teacher as the root node. The schema would then be as shown in Figure 6. Carefully note that the relationship between Teacher and Offering is different from that of Course

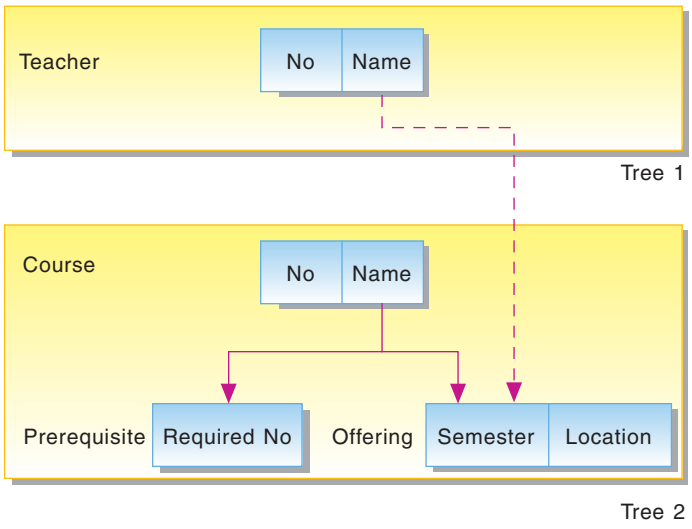


*Figure 6 An alternative to the schema of **figure 4**, with Teacher and Course as root nodes in two separate hierarchies and with Offering as a dependant of Course*

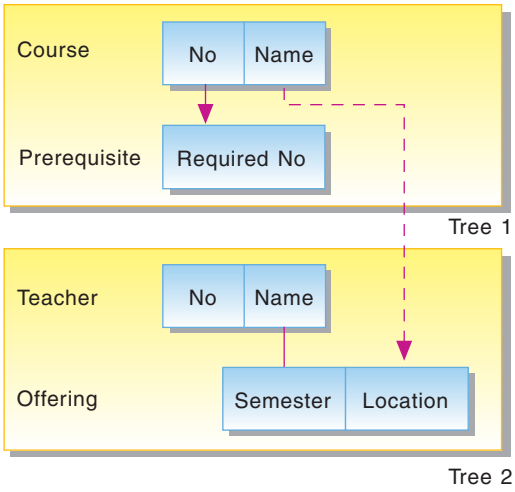


*Figure 7 An alternative to the schema of **figure 6**, still with Teacher and Course as root nodes in two separate hierarchies but with Offering as a dependant of Teacher instead of Course*

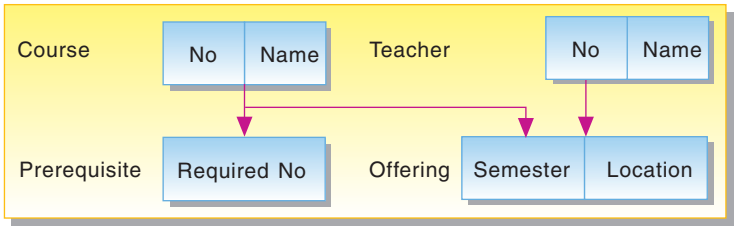


Figure 8 A network schema for the course-prerequisite-offering-teacher database. As with the hierarchic schema, arrows point from parent to child nodes but Offering is now a child of both Course and Teacher

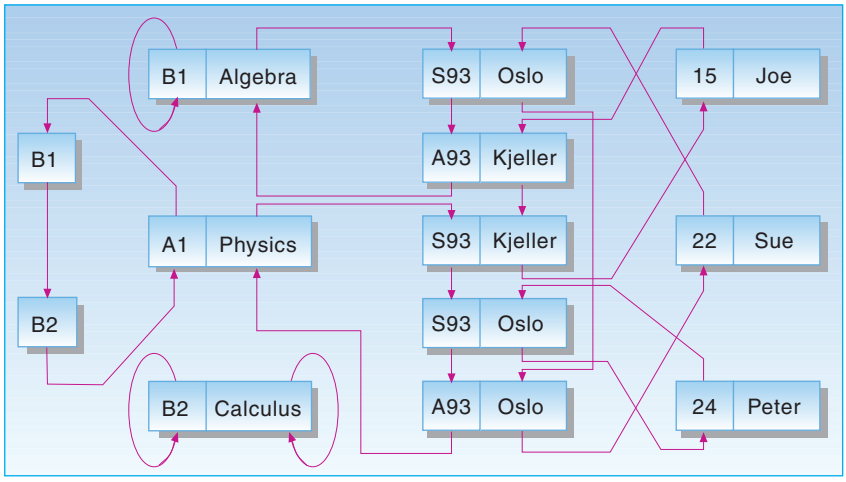


*Figure 9 Occurrence diagram for the schema of **figure 8**. This figure also serves to illustrate the term* link *(also known as* co-set*), which is central for NDBMSs. A link definition involves a* parent *(or* owner*) node and a* child *(or* member*) node type. This schema has three link types; one with Teachers as parent and two with Courses as parent. As indicated by the arrows, Joe is the parent and A93-Kjeller and S93-Kjeller are the children in one such link type occurrence. Note that links are circular structures*

with Offering. One could say that the network structure needed in this case has been achieved by combining two tree structures. It would also have been possible to define Offering as a dependant of Teacher, as shown in Figure 7. The two schemas will have different performance characteristics.

6.2 Network

The data structures supported by network DBMSs (NDBMSs), also known as CODASYL-systems (see next paragraph), are an extension of those supported by HDBMSs; a dependent node in an NDBMS data structure may have more than one parent, i.e. it may be part of more than one parent-child relationship type. A well known example of an NDBMS is IDMS from Computer Associates.

The Database Task Group (DBTG) of the Programming Language Committee (also known as the Cobol committee) of the *Conference on Data Systems Languages* (CODASYL) produced a report in 1971 describing how the language interface of an NDBMS should be. Hence, DBMSs that implement these proposals, more or less, are often called CODASYL-systems.

A network schema for the example database could be as shown in Figure 8. The Offering segment now has *two* parent segments, Course and Teacher, and the two parent-child relationships are equivalent. We now have a *directed graph* instead of one or two tree structures. The occurrence diagram is shown in Figure 9.

Navigation between two or more tree structures in an HDBMS is limited to a *predefined* number of iterations, while no such limitation applies to an NDBMS.

Another feature that distinguishes NDBMSs from HDBMSs is the ability to define a schema such that segments of type C, say, should be dependent *either* on a segment of type A *or* a segment of type B, but not both. For example, a Child could be a dependant of a Mother or a Father. This is claimed to be a useful feature by NDBMS proponents.

6.3 Relational

Relational DBMSs (RDBMSs) are radically different from the two foregoing kinds of DBMS. Rather than performing explicit navigation through tree or network structures, an application must submit search criteria to an RDBMS which will then return all data items that satisfy those criteria. This is known as query access. It is common to say that one specifies *what* one wants, not *how* to get it. The latter is figured out by an RDBMS component called the optimiser. Another important difference is that H- and NDBMSs limit access to a single node at a time, while RDBMSs are set-oriented.

The following is an **informal definition** of relational DBMS: the only data structure supported at the user level by an RDBMS, is tables, also known as relations – hence *relational.*

Most people find tables easy to understand; they have a fixed number of columns and zero or more rows. This simplicity is both a strength and a weakness of the relational model of data; it provides ease of use, but also imposes certain drawbacks, such as e.g. having to represent complex data structures by a large number of tables – resulting in performance degradation.

The relational model was published by E.F. Codd – then of IBM – in 1970, for which he later received the Turing

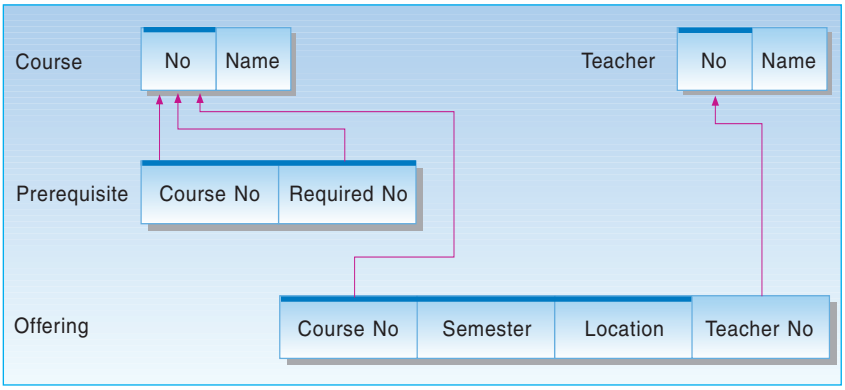


Figure 10 A relational schema for the course-prerequisite-offering-teacher database. See discussion in text

Award. The DBMS marketplace is currently dominated by relational products, of which some of the more well known are Oracle, Sybase, Ingres, Informix, RDB, and DB2.

A relational schema for the example database, could be as shown in Figure 10. The relational model requires that all tables have a *primary key,* i.e. one or more columns that contain enough information to uniquely identify an individual row. Primary keys are indicated in Figure 10 by bold lines over the column names in question. While relationships between segments in HDBMSs and NDBMSs are typically represented by *physical pointers,* i.e. explicit references to the physical addresses where the segments in question reside, no such construct is allowed to be exposed at the programming interface of an RDBMS.

All references between two rows, whether they belong to the same table or not, must be associative, i.e. based on column values only. The relational mechanism for inter-row references is called *foreign* key. There are four foreign keys in the schema of Figure 10, each represented by an arrow. For example, the Offering.TeacherNo is a foreign key whose *values* point to Teacher.No, the primary key of the Teacher table. This means that every value of Offering.TeacherNo must be present in Teacher.No, or else it would be possible for an offering to reference a non-existent teacher, in which case the database would be inconsistent. As pointed out in the above discussion of ACID properties, it is the responsibility of the DBMS to ensure consistency.

But what if one needs to register an offering for the next semester, say, and it is not yet known who will teach; how can this be handled? Well, the relational model allows the use of *nil marks* – also known as null values – indicating e.g. "value not known". Unless prohibited for Offering.TeacherNo, we can use a nil mark in that column to avoid referential constraint enforcement from the DBMS when necessary. The use of nil marks is somewhat controversial among database researchers, see e.g. (4).

Course

No	Name
B1	Algebra
B2	Calculus
A1	Physics

Prerequisite

Course No	Required No
A1	B1
A1	B2

Teacher

No	Name
22	Sue
15	Joe
24	Peter

Offering

Course No	Semester	Location	Teacher No
A1	S93	Kjeller	15
A1	S93	Oslo	24
A1	A93	Oslo	22
B1	S93	Oslo	22
B1	A93	Kjeller	15

Figure 11 Occurrence diagram for the schema of ***figure 10***

The de facto standard language for relational data access is SQL. Sub- and supersets of this language are also being used by non-RDBMS products or prototypes. Its vast importance is summed up in the famous words "For better or worse, SQL is intergalactic dataspeak" (13). This is of course an exaggeration, but at least the language is worth having a brief look at in this article. This will be done by means of some examples, using the database of Figure 11, which is the relational schema of Figure 10 populated with now familiar values.

Example 1: SQL expression that will find all courses:

```
SELECT   *
FROM     Course
```

Result:

```
No       Name
-----------------

B1       Algebra
B2       Calculus
A1       Physics
```

Example 2: SQL expression that will find all offerings taught by teacher number 15:

```
SELECT   CourseNo,
         Semester,
         Location
FROM     Offering
WHERE    TeacherNo = 15
```

Result:

```
   CourseNo Semester
Location
   ------------------------
--

   A1       S93       Kjeller
   B1       A93       Kjeller
```

Example 3: SQL expression that will find names of courses taught at Kjeller:

```
   SELECT A.Name
   FROM   Course A, Offering
B
   WHERE  A.No=B.CourseNo
   AND
B.Location='Kjeller'
```

Result:

```
   Name
   -------

   Physics
   Algebra
```

Explanation: the latter SQL expression is known as a *join query;* it joins information from tables Course and Offering. Conceptually, the DBMS will scan through the Offering table, and for every row whose Location is Kjeller it will use its CourseNo value to access the Course table and find the Name of the course whose No value is equal to the current CourseNo value.

6.4 Object oriented

Object oriented DBMSs (OODBMS or just ODBMS) have only been commercially available since the late 1980's, and are therefore quite immature compared to the other kinds of DBMS. ODBMSs have their roots in the *object oriented programming paradigm,* a certain way of organising computer programs, first supported by the programming language Simula – invented in Norway during the 1960's – and later by Smalltalk, Objective-C, C++, Eiffel, and many more. Distinctive features of object oriented programming languages (OOPLs) include support for classes, objects, inheritance, messages, methods, and encapsulation; which may informally be outlined as follows:

- A **class** is a description of the properties of objects of that class. For example, class Beagle could describe the properties of all beagles.
- An **object** is an *instance* (or occurrence) of a class. Snoopy and Fido could e.g. be instances of class Beagle.
- If class Beagle is defined as a *subclass* of Dog, Beagle will **inherit** all the properties of Dog. This is very natural, since every beagle is a dog.
- All objects understand certain **messages.** The set of messages that objects of a given class understand, is called its protocol or interface. Upon receipt of a protocol message, a corresponding **method** will be executed by the receiver. Thus, a message is a special kind of procedure call, and a method is a procedure that is local to an object.
- An object must be accessed through its protocol and will – in general – hide its internal structure from its surroundings. This data hiding is known as **encapsulation.** If e.g. the message "age" is in the protocol of a given class and that class encapsulates its data, users cannot know whether age values are explicitly stored inside objects or calculated during execution – e.g. as the difference between current date and birth date.

It is common to say that objects are data that exhibit a certain *behaviour,* and the reader has probably guessed by now that object oriented programming is focused on defining the behaviour and implementation of classes of objects, as well as manipulating objects during program execution. But what is an ODBMS? Unfortunately, there is no universal consensus on that question. Very informally, one could say that an ODBMS is a DBMS which is able to store and manipulate arbitrarily complex objects, or an agent that provides persistence for objects.

In other words, and OODBMS is a DBMS that "understands objects". *The Object-Oriented Database System Manifesto* (1) "attempts to define an object-oriented database system (and) describes the main features and characteristics that a system must have to qualify as an object-oriented system". This manifesto separates the characteristics into three groups.

Quote:

- ***Mandatory,*** the ones the system must satisfy in order to be termed an object-oriented database system. These are complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extendibility, computational completeness, persistence, secondary storage management, concurrency, recovery, and an *ad hoc* query facility.
- ***Optional,*** the ones that can be added to make the system better, but which are not mandatory. These are multiple inheritance, type checking and inferencing, distribution, design transactions, and versions.
- ***Open,*** the points where the designer can make a number of choices. These are the programming paradigm, the representation system, the type system, and uniformity.

Unquote.

6.5 Extended relational

Several extensions to the relational model of data has been proposed since it was published in 1970. This article will only briefly mention object oriented extensions, which are currently receiving considerable attention from the ISO and ANSI committees responsible for the emerging SQL3-standard. Recall that SQL is the de facto standard language for relational data access. Several RDBMS vendors claim that future releases of their products will have "complex object support", which obviously must include the ability to store arbitrarily complex objects "as they are", and facilities for traversal of object structures. Presumably, an extended RDBMS product may also involve e.g. inheritance for table definitions, support for one or more persistent languages, message passing to and among stored objects, and other things. The important point is that this approach attempts to provide facilities requested by object oriented applications, while retaining the relational model of data.

The authors of The Third-generation Data Base System Manifesto (13) "present the three basic tenets that should guide the development of third generation systems (and) indicate 13 propositions which discuss more detailed requirements for such systems". This paper classifies network and hierarchic DBMSs as first generation, classifies relational DBMSs as second generation, and uses third generation to denote future DBMSs.

Quote:

- Tenet 1: Besides traditional data management services, third generation DBMSs will provide support for richer object structures and rules.
- Tenet 2: Third generation DBMSs must subsume second generation DBMSs.
- Tenet 3: Third generation DBMSs must be open to other subsystems.

Unquote.

Based on these tenets, thirteen propositions are indicated (carefully note that the first part of any proposition number below indicates the tenet to which it belongs).

Quote:

- 1.1: A third generation DBMS must have a rich type system.
- 1.2: Inheritance is a good idea.

- 1.3: Functions, including database procedures and methods, and encapsulation are a good idea.
- 1.4: Unique Identifiers (UIDs) for records should be assigned by the DBMS only if a user-defined primary key is not available.
- 1.5: Rules (triggers, constraints) will become a major feature in future systems. They should not be associated with a specific function or collection.
- 2.1: All programmatic access to a database should be through a non-procedural, high-level access language.
- 2.2: There should be at least two ways to specify collections, one using enumeration of members and one using the query language to specify membership.
- 2.3: Updatable views are essential.
- 2.4: Performance indicators have almost nothing to do with data models and must not appear in them.
- 3.1: Third generation DBMSs must be accessible from multiple high-level languages.
- 3.2: Persistent X for a variety of Xs is a good idea. They will all be supported on top of a single DBMS by compiler extensions and a (more or less) complex run time system.
- 3.3: For better or worse, SQL is intergalactic dataspeak.
- 3.4: Queries and their resulting answers should be the lowest level of communication between a client and a server.

Unquote.

6.6 Persistent languages

A concept closely related to that of ODBMS and ERDBMS, is *persistent languages.* In programs written in traditional OOPLs, objects – like ordinary program variables – cease to exist once program execution terminates. If objects are to be used by several (invocations of) programs, they must be explicitly stored in and retrieved from a database or a file system. If, on the other hand, the language is persistent, all objects created during execution may more or less automatically be stored in the database. Thus, the program's address space and the database are conceived by the programmer as a single, contiguous object space. This is a very powerful concept and a major vehicle for overcoming the so-called *impedance mismatch* between the two data spaces, which has to do with the different data models they support. There seems to be a broad consensus in the database community that persistent languages are useful; see e.g. proposition 3.2 in the previous section.

7 Conclusions

Database systems are important today, and are likely to become even more so in the future, as an increasing number of application areas require data sharing, ACID properties, and other good things offered by DBMSs. The research and development efforts in the DBMS area are considerable and growing (12). Database systems will be a vital part of future telecommunication systems. One way or the other, next generation DBMS products will support complex objects (1, 13).

8 Further reading

For readers interested in a deeper understanding of topics introduced in this article, the following reading is suggested:

- General database theory and related topics: introductory (4); advanced (7) and (9)
- Relational database theory: (2), (4) and (7)
- Object oriented programming: (11)
- Object data management: introductory: (3); advanced (1), (6), (8), (13), and (14)
- The new SQL standards: (5), (8), and (10)
- Current and future importance of database systems: (12).

9 References

1 Atkinson, M et al. The object-oriented database system manifesto. *Deductive and object-oriented databases,* Amsterdam, Elsevier Science Publishers, 1990.

2 Atzeni, P, De Antonellis, V. *Relational database theory.* Redwood City, Calif., Benjamin/Cummings, 1993.

3 Cattell, R G G. *Object data management.* Reading, Mass., Addison-Wesley, 1991.

4 Date, C J. *An introduction to database systems, volume 1*. Fifth edition. Reading, Mass., Addison-Wesley, 1990.

5 Date, C J, Darwen, H. *A guide to the SQL standard.* Reading, Mass., Addison-Wesley, 1993.

6 Dittrich, K R, Dayal, U, Buchmann, A P (eds). *On object-oriented database systems.* Berlin, Springer-Verlag, 1991.

7 Elmasri, R, Navathe, S B. *Fundamentals of database systems.* Redwood City, Calif., Benjamin/Cummings, 1989.

8 Gallagher, L. Object SQL: Language extensions for object data management. In: *Proceedings of the first international conference on information and knowledge management,* Baltimore, M.D., November 1992.

9 Gray, J, Reuter, A. *Transaction processing: concepts and techniques.* San Mateo, Calif., Morgan Kaufmann, 1993.

10 Melton, J, Simon, A R. *Understanding the new SQL: a complete guide.* San Mateo, Calif., Morgan Kaufmann, 1993.

11 Meyer, B. *Object-oriented software construction.* London, Prentice Hall, 1988.

12 Silberschatz et al. Database systems: achievements and opportunities. *Communications of the ACM,* 34(10), 1991.

13 Stonebraker M et al (the committee for advanced DBMS function). Third-generation database system manifesto. *ACM SIGMOD Record,* 19, (3), 1990.

14 Zdonic, S B, Maier, D (eds). *Readings in object-oriented database systems.* San Mateo, Calif., Morgan Kaufmann, 1990.

Software development methods and life cycle models

BY SIGRID STEINHOLT BYGDÅS AND MAGNE JØRGENSEN

681.3.06

Abstract

Different software development methods and life cycle models are based on different views on software development. This paper discusses some of these views and describes several software development methods and life cycle models in context of these views. The methods and models described are Structured Systems Analysis and Design Method (SSADM), the ESA Waterfall model, Coad and Yourdon's Object Oriented Analysis (OOA), verification oriented software development and two evolutionary life cycle models. In addition software development by Syntax Description, Operational Software development and the spiral model are briefly described. More research on the effects of using different software development methods and life cycle models is needed. The advantages and disadvantages of two different research strategies are therefore briefly discussed.

1 Introduction

"No methodology – no hope", from Making CASE work (1)

The earliest view of software development was the *programming* view. This view had the implicit model of software development being a transcription from a mental representation of a problem to an executable program. The development approach of this view was: code & run (and, of course, debug and run again). The programmers were artists and programming was artistic work.

Later on, when the programming view had failed in large software development projects, software development was viewed as a *sequential* process from problem understanding to executable code, i.e. analogous to the rather sequential process of building a house. The sequence of the steps was prescribed and the result of each step was validated and verified. This view has resulted in for instance the waterfall model and top-down design. Most of the models and methods belonging to this view are in widely use today.

After some years, the sequential and top-down views of the software development were seriously questioned. The new view, software development as an *exploratory activity* emerged. Object oriented methods, evolutionary models and prototyping are results of this view. This view considers software development to be rather opportunistic and emphasises reuse and evolution. Software development based on exploration is typically less controllable than sequential development. Methods and life cycle models based on this view are currently not very wide-spread.

Independent of development view, software projects often overspend and deliver software too late with too low quality. Also, the impact of the alternative development views are not well understood. Thus, the change of development view throughout the history has probably not been based on knowledge of change in productivity and quality. It is more probable that the change has been driven by the lack of success of the previous views.

Methodology: 1. the system of methods and principles used in a particular discipline. 2. the branch of philosophy concerned with the science of method or procedure.

Method: 1. a way of proceeding or doing something, esp. a systematic or regular one. 2. orderliness of thought, action, etc. 3. (often pl.) the techniques or arrangement of work for a particular field or subject ...

Model: 1. a. a representation, usually on a smaller scale, of a device, structure, etc. ... 2. a. a standard to be imitated ... 9. a simplified representation or description of a system or complex entity.

Technique: 1. a practical method, skill, or art applied to a particular task.

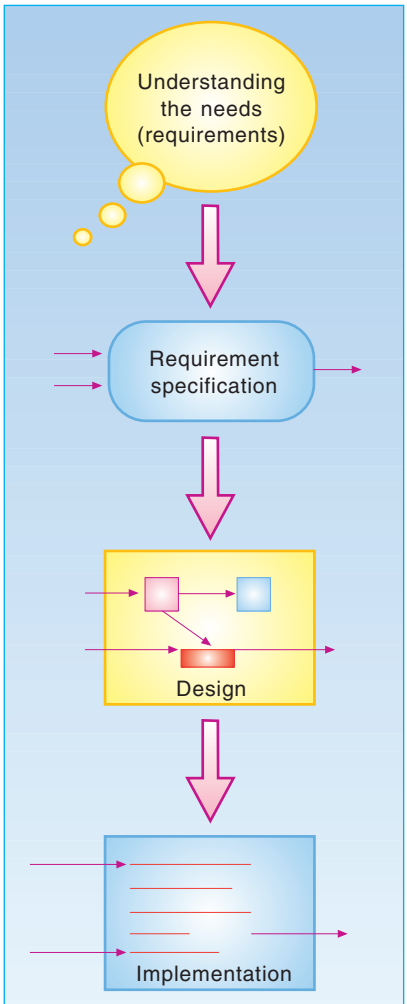
Technology: 1. the application of practical or mechanical sciences to industry or commerce. 2. the methods, theory, and practices governing such application. 3. the total knowledge and skills available to any human society for industry, art, science, etc.

Figure 1 Definitions from (2)

2 Definitions

The term method is often confused with methodology, model, technique, and technology. The definitions in Figure 1, taken from Collins dictionary (2), give some clues in what the differences are.

This paper uses the underlined definitions in Figure 1. The relationship between methods and life cycle models is assumed to lay mainly in the scope and the focus on needs. Life cycle models cover the whole life cycle and have focus on project control, while methods sometimes only cover parts of the life cycle and focus other needs than project control.



Key: The horizontal arrows illustrates inputs and outputs to the system, the vertical arrows show the sequential progression from phase to phase.

Figure 2 Conventional development

3 Conventional methods and life cycle models

Figure 2 gives an overall picture of conventional software development from the software developer's point of view. The figure illustrates some typical characteristics of conventional software development:

- The process of understanding is a rather undisciplined and "fuzzy" process (illustrated by the "bubbles" in the figure.) The output is not documented, but consist in increased understanding of the needs.
- Development of the requirements specification use "black box" methods, i.e. for use of methods describing the external behaviour ("what" in opposition to "how") of the system (illustrated by not drawing the arrows inside the Requirement specification box).
- Development of the design specification makes use of "white box" methods, i.e. for methods describing the internal behaviour of the system (illustrated by drawing the arrows inside the Design box).
- The implementation methods focus on structuredness and modularity.
- Life cycle models are rather sequential (like the water fall model), document driven and phase-oriented.
- Maintenance considerations in the early phases are lacking.
- The methods used are top-down and use a hierarchical decomposition.

It is not likely that real software development ever can or should be totally sequential and purely top-down. However, methods and life cycle models based on these assumptions have been rather successful. A reason for this may be that the activities of the methods and life cycle models are not really carried out as prescribed, they just act as guidelines together with common sense. This view is defended in the famous paper "A Rational Design Process: How and Why to Fake it" (3). Another reason for the success may be that the method enables a rather high control of the software development process.

Table 1 gives an overview of the phases and notations of some conventional methods. The rest of this chapter gives a description of the software development method SSADM and a variant of the "Waterfall model", suggested by ESA (European Space Agency).

3.1 SSADM

SSADM is widely used, especially in the UK. It is the UK government's standard method for developing information technology systems. A modified version of the method is used by Norwegian Telecom.

In this article we give a coarse overview of the original method based on the descriptions in (4) and (5).

The method covers the systems analysis and systems design phases of the software life cycle. In addition, it includes feasibility studies as a first phase. It is an iterative method, i.e. it is allowed to redo phases or tasks in a phase, and to correct deliveries of a phase. Some activities can be performed in parallel.

The method contains the following stages:

- Problem definition
- Project identification
- Analysis of systems operation and current problems

Table 1 Some conventional methods

Methods	Life cycle phases	Notations
SA/SD	Analysis Design Implementation	Data flow diagrams and structured English
SADT	Analysis Design	Data and activity diagrams
ISAC	Feasibility study Analysis Design	Activity graphs, information precedence graphs, component relation graphs, design graphs and equipment graphs
JSD	Design Implementation	Control flow, process structure and data flow diagrams
Sysdoc	Design Implementation	Entity relationship diagrams and a process structure language
SSADM	Feasibility study Analysis Design	Data flow diagrams, logical data structures, entity life histories, logical dialogue outlines, relational data analysis, and composite logical data design, among others

1) Examine each entry on the problem/requirement list. If the solution is to be provided by the selected business system option ensure that any necessary changes are made to the logical data structures and entity descriptions.

2) Complete the entity descriptions and ensure that they include volumes, key attributes and other known attributes.

3) Update the datastore/entity cross reference.

4) Update the data dictionary.

Figure 3 This figure illustrates the size of the tasks in SSADM. It shows the four tasks of SSADM step 240, Create required data structure, which is step four of stage 2; Specification of requirements. Stage 2 belongs to phase 2; Systems analysis

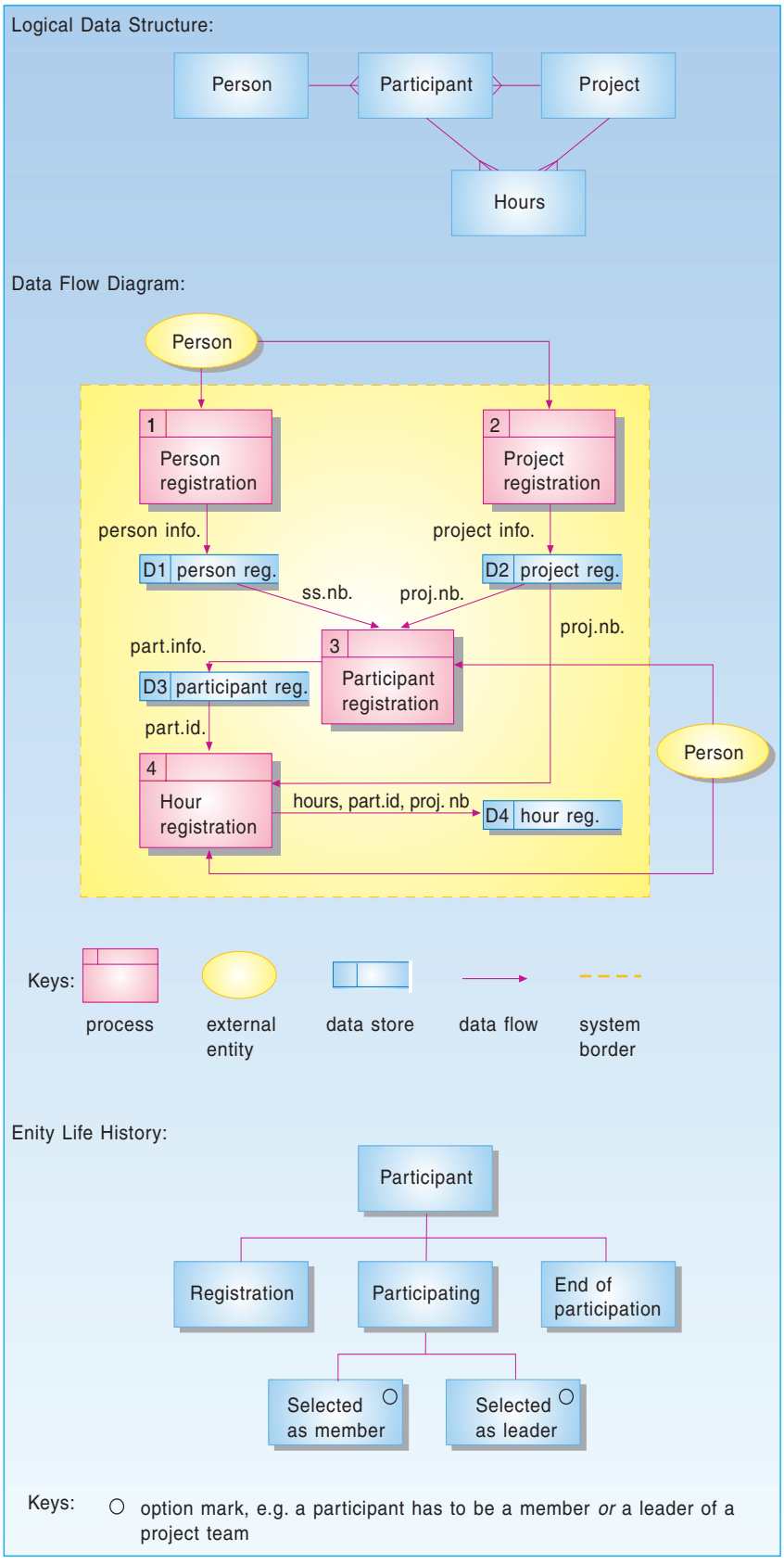


Figure 4 Examples of the three main notations in SSADM

- Specification of requirements
- Selection of technical options
- Data design
- Process design
- Physical design.

The first two stages belong to the feasibility study, the next three to the systems analysis phase and the last three to the systems design phase.

Each stage is divided into steps and each step is divided into tasks. It is claimed in (4) that this results in higher productivity because the software developers always handle tasks of the "right" size. An example of a step is "create required data structure". This step involves the four tasks shown in Figure 3.

SSADM is characterised as data and function oriented. This is due to the three main notations used during the feasibility study and systems analysis. These notations are data flow diagrams (function oriented), logical data structures (data oriented) and entity life history diagrams. Figure 4 gives an example of a logical data structure, a high-level data flow diagram and a diagram of one of the entity's life histories.

A high-level data flow diagram and a logical data structure are developed in parallel, or in arbitrary order. For every entity in the data structure an entity life history diagram is developed. Such diagrams are used to explore the processing requirements of each entity and to identify state change transactions, errors and special cases. Examination of entity life history diagrams may result in changes of both the data flow diagrams and the data structure.

The notations mentioned above are not the only ones used when developing systems according to SSADM. Other notations are used when

- defining user options
- designing dialogues
- performing relational data analysis (normalising)
- designing a composite logical data model
- making process outlines
- making first cut data design
- making first cut program design
- defining physical design control.

There exists a micro version of the method for the development of smaller systems. A special maintenance version also exists.

3.2 ESA Waterfall model

The best known life cycle model is undoubtedly the "Waterfall model". It was originally described in (6), but has since been modified. A variant of the "Waterfall model" is described in the ESA (European Space Agency) software engineering standard, (7). This standard describes the phases and activities which must occur in any software project at ESA, and suggests how the standard can be interpreted in a "Waterfall model". It is not a pure waterfall model, since iterations within a phase are allowed.

Figure 5 pictures the ESA variant of the Waterfall model and the verification approach suggested. The figure illustrates the sequence and the phases of system development. The verification arrows are directed on the activity from which the results are to be verified. The phases are described below:

Problem Definition:
- determination of operational environment
- identification of user requirements

Software Requirement:
- construction of logical model
- identification of software requirements

Architectural design:
- construction of physical model
- definition of major components

Detailed design:
- module design
- coding
- unit tests
- integration tests
- system tests

Transfer:
- installation
- provisional acceptance tests

Operations and Maintenance:
- final acceptance tests
- operations
- maintenance of code and documentation.

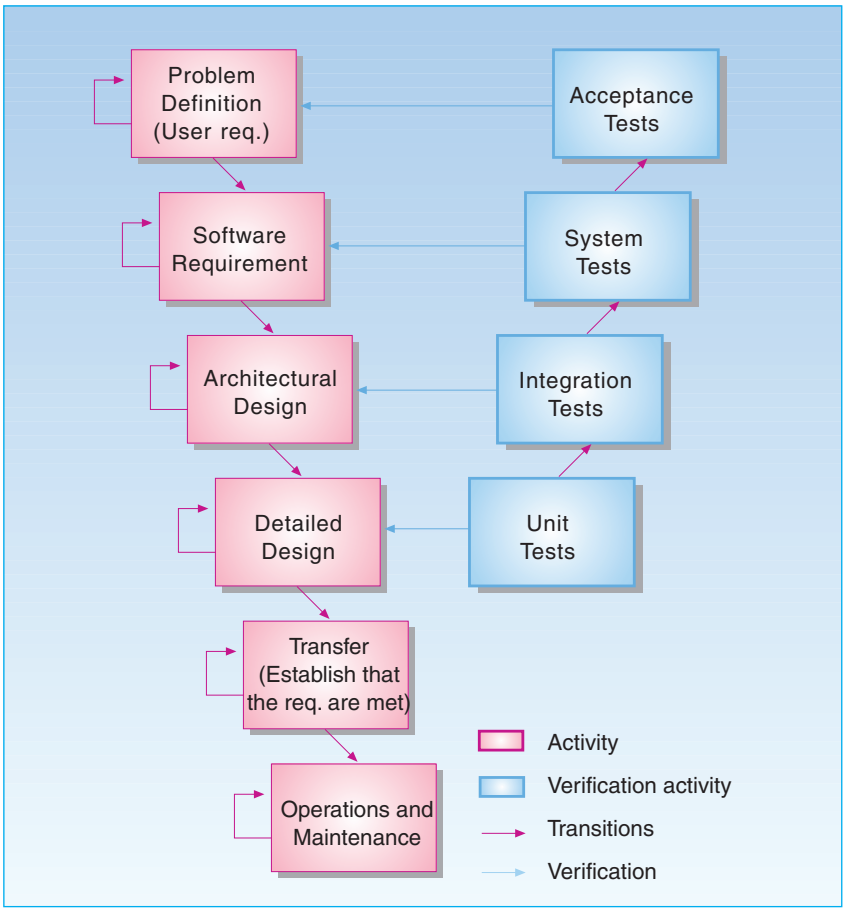


Figure 5 ESA Waterfall model and verification approach

The unit, integration and system tests are *executed* in the detailed design phase and the acceptance tests in the transfer and operation phase. The standard requires that the acceptance tests are *developed* in the problem definition phase, the system tests in the software requirement phase, the integration tests in the architectural design phase and the unit tests in the detailed design phase.

The software development activities standardised by ESA focus on reviews. Reviewing is a phase-independent method of discovering errors, incompleteness, inconsistencies, etc., in specifications, programs and other documents. The method of reviewing documents was introduced by Fagan (8) and is based on human reading of documents. It is believed that different errors, incompleteness, inconsistencies, etc., can be found more easily using this method than by testing. Testing and reviews do, thus, not exclude each other.

4 Non-conventional methods and life cycle models

It is not easy for "new" methods and life cycle models to become commonly accepted. The reasons for this may be the conservativeness of software managers and the large costs of teaching employees a new method or a new life cycle model. In addition, the usefulness of many non-conventional methods and life cycle models has not been sufficiently validated. In the following, we will give a description of some non-conventional software development methods and life cycle models; an object oriented method, a verification oriented method and an evolutionary life cycle model. Finally we will briefly characterise a selection of other non-conventional system development methods and life cycle models.

4.1 Object Oriented Analysis (OOA)

"The objects are just there for the picking" (9)

Object oriented programming was introduced in the late 1960's by Simula. Most of the principles used by current OOA methods were known already then. While the focus of object oriented programming is on *specifying the solution* of a problem, the focus of OOA methods is on *understanding and specifying* the problem.

The following method, the Coad-Yourdon Object Oriented Analysis (10), is probably one of the best known OOA-methods. The method aims at development of an object oriented description. The method does only cover the analysis phase of software development and has the following five steps:

1 Identify Classes and Objects

2 Identify Structures

3 Identify Subjects

4 Define Attributes

5 Define Services.

The steps are carried out in parallel, i.e. based on the view of software development being an exploratory activity. The method has not a well defined description of the steps and sequences to be performed. Instead, it offers guidelines, i.e. common sense rules of "where to look", "what to look for", and "what to consider and challenge". The problem domain is explored through identification of classes and objects, structures (relations between classes or objects), subjects (i.e. partitioning of the object model), attributes and services (i.e. activities of the classes). The notations used together with the method are simple and the semantics of the constructs are rather informal.

The resulting problem specification, the object oriented description, is presented in five transparent layers, see Figure 6.

4.2 Verification oriented software development

The view taken in verification oriented software development is that programs can be treated as mathematical objects. This way, mathematical proofs can be applied to verify that the programs are "correct". Program verification can be used to prove that a program is correct relative to specification of the state before and after program execution. However, if the specification is incorrect or incomplete, the proof may be of little value.

Mathematical proofs on already developed programs can be hard to develop. An easier approach seems to be to develop the program and the proof hand-in-hand. The method described below illustrates what a verification oriented development of a loop subprogram can look like. The method below is taken from (11), although not explicitly formulated there.

The method:

1 Develop a general description of the task the program is to perform.

2 Develop a specification – consisting of pre-conditions (V) and post-conditions (P), formulated as logical algebraic expressions. Pre-conditions and post-conditions are conditions, e.g. $x > 6$, describing the state before and after the execution of the program.

3 Decide on the basic structures of the program; in this case a loop structure

4 Develop the loop invariant (I), i.e. conditions that will not be affected through the execution of the program. Through the development of the invariant, the initialisation part of the program is determined.

5 Develop the loop condition, based on the difference between the post-condition and the loop invariant.

6 Using the loop invariant as a check list, design the body of the loop.

7 Prove that the program is partially correct, i.e. correct if it terminates.

8 Prove that the program terminates.

A more general method is described in (12).

The proof of partial correctness, step 7, will reflect closely the design steps, using the same pre- and post-conditions and invariant. It is not in general possible to prove that a program will terminate, i.e. step 8, although in most cases it is relatively simple, according to (11).

In Figure 7 the first four steps of the method are illustrated in the development of a while loop subprogram. The example is extracted from an example in (11). The explanation is mainly ours.

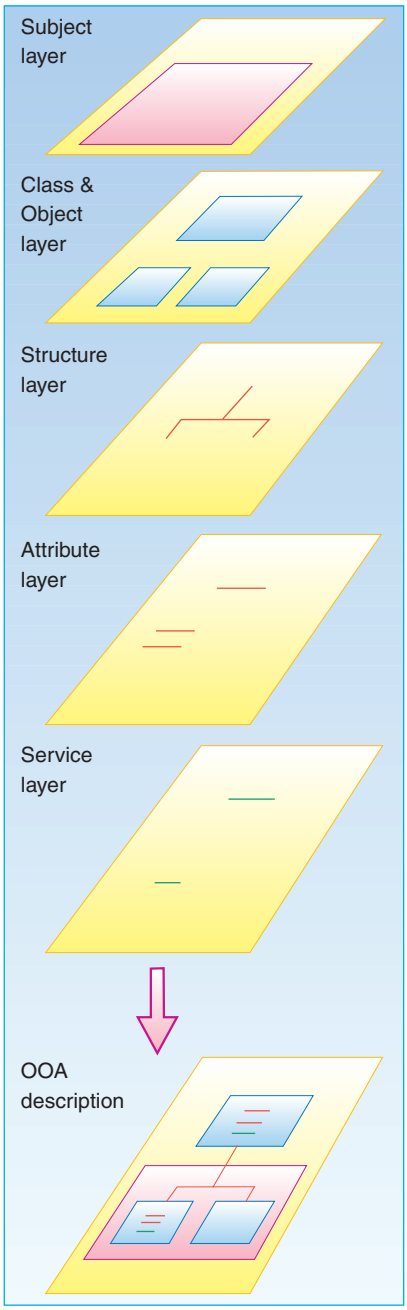


Figure 6 The five layers and an OOA description as seen through the transparent layers

4.3 Evolutionary life cycle models

The evolutionary life cycle models are based on the experience that often it is impossible to get a full survey of the user

Linear search – verification oriented development

Step 1: Informal description of the task

The variable n, the array a(1), a(2), ... , a(n) and the variable x are given. The program to be designed is to determine whether the value of x is present in the array A and if so, where it first occurs. The value of the result variable k should indicate which element of A was found to be equal to x. If x is not equal to any element in A, k should be assigned the value n + 1.

Step 2: Specification

The task description leads to the following preconditions (V) and postconditions (P):
V: $n \in Z \wedge 0 \leq n$

P: $(k \in Z \wedge 1 \leq k \leq n+1)$
$\wedge (\forall i < k \mid A(i) \neq x)$ [all elements before the k-th $\neq x$, i = {1,2, ...}]
$\wedge ((k \leq n \wedge A(k) = x) \vee (k = n+1))$
(The values and indexes of the array A, should not be modified.)

Step 3: Decision on basic structure

The task requires that a repeated test is carried out on whether x is equal to an element in A. For this purpose a while loop is used. The while loop has the general form:

initialisation; while B do S endwhile

where B is the loop condition and S the loop body (statements). The loop condition should have no side effects.

Step 4: Deciding on invariant (I) and initialisation

The invariant is developed through generalisation (weakening) of the postconditions. Choosing the initialisation k := 1, the two first parenthesised and-terms in the postconditions (see step 2) will always be true, i.e. true before entering the loop for the first time and true after terminating the loop. The two first parantesised and-terms in the postconditions are thus chosen as the invariant I.

I: $(k \in Z \wedge 1 \leq k \leq n+1)$
$\wedge (\forall i < k \mid A(i) \neq x)$

Figure 7 Some of the steps in a verification oriented development method

requirements before the system is presented to the users. This may be due to the nature of the system or be due to the fact that the users' opinions and requirements change as more knowledge is acquired. In order to build the right product, it becomes necessary to get responses from the users early in and throughout the software development process.

The evolutionary model can be described as follows:

- Deliver something to the users
- Receive and record user responses
- Adjust design and objectives based on the responses.

Evolutionary prototyping and incremental development are two examples of evolutionary models.

Prototyping is often used in conventional software development, during feasibility studies or requirement analysis. This kind of prototyping is often called *rapid prototyping*. In *evolutionary prototyping,* however, the prototyping technique is the basis of the total system development process (13). By applying this approach to development, it should be possible to incorporate all types of changes that are requested both during development and during system operation. This way maintenance becomes an integrated part of the software life cycle, i.e. there is no significant difference between software development and maintenance. Figure 8 shows what the software life cycle can look like if evolutionary prototyping is used.

The goal of incremental development is to manage risk by developing small parts of the system at a time. A system skeleton covering the basic functions of the system is developed first. Additional functionality is then implemented in several partly overlapping sub-projects. During the system development process, users will have an executable system most of the time, and be able to give early feedback to the developers.

In literature it is often argued that evolutionary methods can give flexible solutions, incorporate maintenance in the software life cycle and simplify software reuse. In spite of these claims, there is a strong disagreement about the claims e.g. in (14) and (15), that these methods reduce software development costs.

4.4 Other non-conventional methods and life cycle models

Software development by syntax description is described in (16) and aims at the development of syntax-directed programs. Syntax directed software is software where the syntax of the input plays a central role, e.g. text formatters and command interpreters. The main characteristics of this method is the development of a formal grammar of the

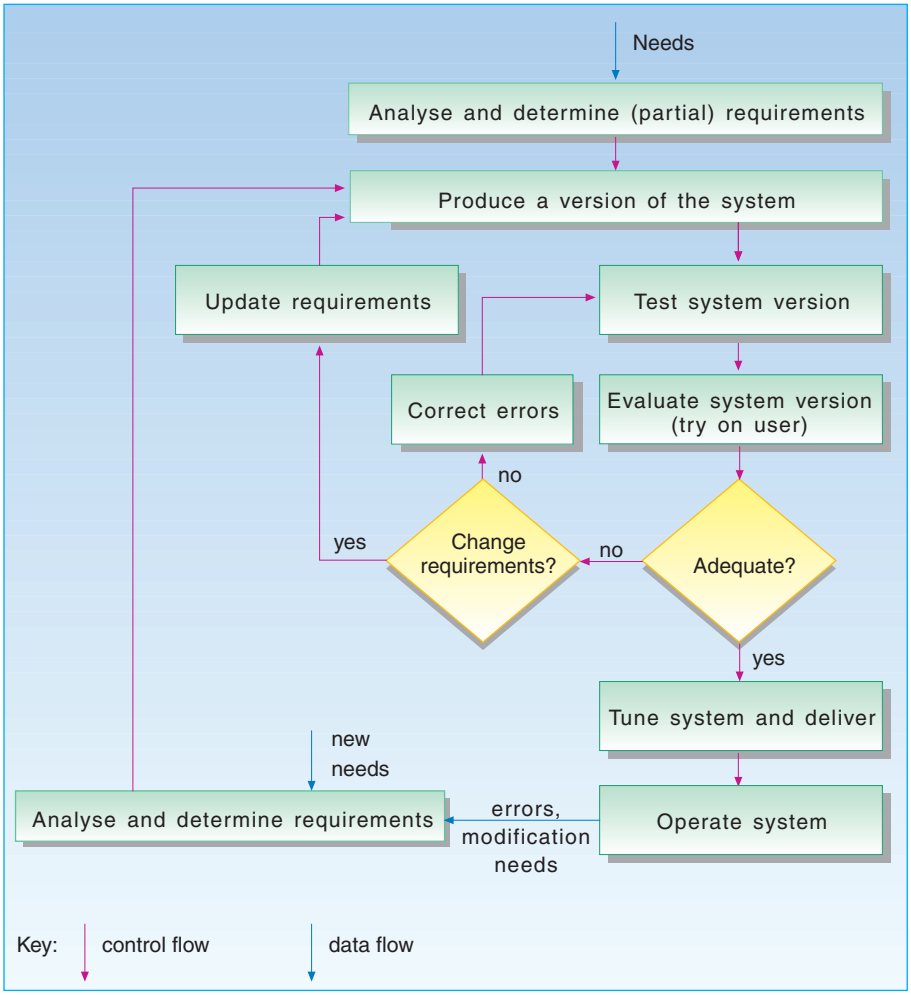


Figure 8 A possible evolutionary prototyping life cycle model

possible inputs to a system. When a grammar for the input language is developed, a translation from input to output (translation syntax) is developed. Developing software using the UNIX tool YACC (Yet Another Compiler Compiler) can be considered a software development by syntax description.

As far as we know, there is no documentation of the benefits of separating focus on "what to do" from "how to do it" into two different phases. The life cycle model described in (17), *Operational software development,* breaks away from this commonly accepted separation. In the analysis phase of operational software development a complete and executable (by a suitable interpreter) operational representation of the system is developed. The operational representation mixes "what to do" and the "how to do" of the system specification.

Adding more design and implementation information, the specification is then transformed into executable code.

This life cycle model is based on two "modern principles": executable specifications and transformation of specifications.

The spiral model (18), accommodates several other life cycle models. The "instantiation" of the spiral model in a particular model depends on the risk involved in the development. Each cycle in the spiral consists of four steps:

1 Determine objectives, alternative solutions (e.g. buy vs. develop product) and constraints

2 Evaluate alternatives, identify risk areas

3 Analyse the risk and develop system (or a prototype of the system)

4 Review the software and decide whether a new cycle is needed.

If a new cycle is needed, plan for the next cycle and go to step 1.

If the user and software requirements are understood reasonably well, one cycle will be sufficient, i.e. the spiral model equals the waterfall model. In projects with less understood requirements several cycles are necessary, i.e. the spiral model is similar to an evolutionary model.

The main characteristics of the spiral model is the focus on risk analysis and the cyclic approach. Instead of deciding on a life cycle model already when the development projects starts, repeating risk analysis is used to decide which type of model is needed, the number of cycles needed and whether for instance prototyping is needed.

5 Challenges

Software development starts with more or less complete and articulated needs and sometimes ends with a specification, doing something useful – not necessarily meeting the original needs. During the development process needs from many different groups and of many kinds occur, like for instance:

- end users need user friendly systems with sufficient functionality
- software maintainers need maintainable software
- software operators need reliable software
- software managers need security of data used by the software, correctness of software, control of the development process, co-ordination of programmers and managers, low costs and high productivity
- software designers need a complete requirement description
- software implementors need a complete, precise and readable design and efficient tools.

Facing this diversity of needs, some stated exact and complete, some vague, some that may be changed soon, some not articulated but implicitly assumed, and most not measurable, the software development participants need support from methods and life cycle models.

Selecting the appropriate methods and life cycle models should be an important issue. More research on the effects of

using different software development methods and life cycle models is therefore needed.

There are two main strategies for studying the effects of development methods and life cycle models: experiments and observation.

Experimental results are often, due to the cost and impracticality of realistic environments, not very applicable to real software development. Typically, software systems developed in experiments are small, students are used as developers and the developers are novice users of the methods and life cycle models. (19) gives some evidence that use of methods leads to more efficient programming than "ad hoc" approaches. However, it is not possible to apply the results of (19) on large software project with professional and experienced developers unless it is assumed that the benefits of using methods on these projects are at least as high as on small scale projects using students with little experience.

Observational studies of implications of methods and life cycle models are often hard to apply on real software development, as well. This is mainly due to the lack of control of external variables. (20) gives an example illustrating this. This study found that the size of the floor space in the office of a developer was more correlated with the development productivity than use of a particular CASE tool. Important to note is that observational studies normally focus on correlations, not so much on cause-effect relationships. (21) gives an example of an observational study where use of structured methods correlate with increased productivity. Application of this result on real software development makes it necessary to assume that the structured methods did not only correlate with productivity, but was the cause of the increased productivity.

There are no evidence that a generally "best" method or a generally "best" life cycle model exists, independently of development environment. This suggests that methods and life cycle models should be evaluated relative to at least the developers, end users and type of system to be developed. However, this way the results may not be sufficiently general to be of any use.

The view of software development as exploration may imply that the development methods and life cycle models should be rather pragmatic, support reusability and incremental change. It is probable that object oriented methods and evolutionary life cycle models support exploration better than conventional methods and life cycle models. However, the need of a controllable software development process seems to be in opposition to the view of software development as exploration. Can we hope for a next generation of methods and life cycle models supporting both control and exploration?

References

1 Parkinson, J. Making CASE work. *Proc. from CAiSE 1990, Stockholm.* Springer Verlag, 21-41, 1990. ISBN 3-540-52625-0.

2 *Collins dictionary,* Oxford, 1981. ISBN 0-00-433078-1.

3 Parnas, D L, Clements, P C. A rational design process: How and why to fake it. *IEEE Trans. on Software Engineering,* 12, 251-257, 1986.

4 Downs, E, Clare, P, Coe, I. *Structured Systems Analysis and Design Method – application and context.* Englewood Cliffs, N.J., Prentice Hall, 1988. ISBN 0-13-854324-0.

5 Avison, D E, Fitzgerald, G. *Information systems Development – Methodologies, Techniques and Tools.* Blackwell Scientific Publications Ltd., 1988. ISBN 0-632-01645-0.

6 Royce, W W. Managing the Development of Large Software Systems: Concepts and Techniques. *Wescon Proc.,* 1970.

7 ESA. *ESA software engineering standard.* February 1991. (ESA PSS-05-0 Issue 2.)

8 Fagan, M E. Design and code inspection to reduce errors in program development, *IBM Systems Journal,* 3(15), 182-211, 1976.

9 Meyer, B. *Object-Oriented Software Construction.* Englewood Cliffs, N.J., Prentice Hall, 1988.

10 Coad, P, Yourdon, E. *Object-oriented analysis.* Englewood Cliffs, N.J., Prentice-Hall, 1991. ISBN 0-13-629981-4.

11 Baber, R L. *Error-free software,* New York, Wiley, 1991. ISBN 0471-93016-4.

12 Baber, R L. *The spine of software: designing provable correct software – theory and practice.* Chichester, Wiley, 1987.

13 Ince, D. Prototyping. In: *Software Engineer's Reference Book.* J A McDermid (ed). Butterworth, 3-12, 1991. ISBN 0-7506-1040-9.

14 Gilb, T. *Principles of software engineering management.* Reading, Mass., Addison-Wesley, 1988. ISBN 0-201-19246-2.

15 Boehm, B W. Software life cycle factors. In: *Handbook of Software Engineering.* Ramamorthy (ed). Van Nostrand Reinhold, 494-518, 1984. ISBN 0-442-26251-5.

16 Lewi, J et al. *Software Development by LL(1) Syntax description.* Chichester, Wiley, 1992. ISBN 0-471-93148-9.

17 Zawe, P. The operational versus the conventional approach to software development. *Communication of the ACM 2,* 104-118, 1984.

18 Boehm, B W. A spiral model of software development and enhancement. *IEEE Computer,* 21(5), 61-72, 1988.

19 Basili, V R, Reiter, W R Jr. A Controlled Experiment Quantitatively Comparing Software Development Approaches, *IEEE Transaction on Software Engineering,* 3, 299-320, 1981.

20 Keyes, J. Gather a baseline to asses case impact. *IEEE Software Magazine,* 30, 30-43, 1990.

21 Brooks, W D. Software Technology Payoff: Some Statistical Evidence, *Journal of systems and software,* 2, 3-9, 1981.

A data flow approach to interoperability

BY ARVE MEISINGSET

681.3.01

Abstract

This paper provides basic language notions and motivation for understanding the 'Draft Interoperability Reference Model' (1, 2, 3). This reference model identifies candidate interfaces for interoperability extensively between databases, work stations, dictionaries, etc. This Abstract provides a technical summary, which may be found difficult to read and therefore can be skipped in a first reading.

The paper discusses language notions for data definition, logical operations and control. Different groupings of these notions are shown to lead to different computer software architectures. A data flow architecture for the communication between processes is proposed. The processes are controlled by schemata which contain data definitions and logical operations. The contents of the schemata provide the application specific behaviour of the system.

In order to avoid introducing implementation details, the notion of control flow is discarded altogether. Two-way mappings are used to state the combined precedence-succedence relations between schemata. This allows data flow to be stated as mappings between schemata without bothering about the existence and functioning of processes. A mapping states permissible flow of data. The two-way mapping allows data flow in both directions between schemata, meaning that no distinction is made between input and output to a system. Each collection of data can appear as both input and output.

All permissible forms of data in a system are specified in schemata. This introduces a layered architecture. For processes to be able to communicate, they have to share a common language. Each schema constitutes such a candidate set of definitions that processes can share in order to be able to communicate. The different kinds of schemata are:

- *Layout schema*
- *Contents schema*
- *Terminology schema*
- *Concept schema*
- *Internal Terminology schema*
- *Internal Distribution schema*
- *Internal Physical schema.*

This layering results in a basic interoperability reference model; the schemata constitute reference points which define candidate interfaces for communication. The schema data are themselves data. This implies that there are just as many candidate forms of schema data as for any other data. Hence, a nesting of the reference model is introduced. Therefore, we will have to talk about the Layout form of the Layout schema, etc.

This proposed general reference model for information systems is compared with the Telecommunications Management Network (TMN) functional architecture. The TMN functional architecture defines interfaces between the TMN and the outside world. This may be appropriate for stating organisational boundaries. However, when compared with the more general reference model presented in this paper, it becomes evident that the TMN model is not clear on what reference points are intended, and that a better methodological approach is needed. A comparison with the Intelligent Network (IN) architecture is made, as well. This architecture is better on the distribution and implementation aspects. However, the IN architecture is not capable of sorting the external interface to the manager from the services to the customer in an appropriate way. This is due to the lack of nesting of the architecture.

This paper is based on experience obtained during the development of the DATRAN and DIMAN tools and previous contributions to CCITT SG X 'Languages and Methods'.

Data flow

This section introduces some basic notions needed for analysing interoperation between software systems, or function blocks inside software systems.

A computer software system can receive input data and issue output data according to rules stated in a set of program statements. This program code will comprise data definitions, logic and control statements. See Figure 1a.

The control statements put constraints on the sequence in which the logic is to be performed. Most current computers (the so-called 'von Neuman architecture') require that the statements are performed in a strict sequence (allowing branching and feedback) and allow no parallelism. This control flow together with the logical (including arithmetical) operations are depicted in Figure 1b.

The data flow, in Figure 1c, depicts data to the logical operations and their outcome. We observe that the data flow states what operations are to be performed on which data. The data flow permits parallelism and is not as restrictive as the control flow.

Data definitions are data which declare the permissible structure of data instances. Logical operations are data which state constraints and derivations on these data. Control flow is a flow of data which is added to restrict the sequencing of the logical operations on the data.

However, more ways exist to achieve the desired computational result than what is stated in a chosen data flow, e.g. observe that the parenthesis in the shown formula in Figure 1c can be changed without altering the result.
(X - 3) * 2 + (Y+1)=2 * X + Y - 5, etc.

A specification of what data are needed for producing which data, is called a precedence graph (4). This is shown in figure 2a. To avoid arbitrary introduction of unnecessary sequences, the precedence graphs are only detailed to a certain level. Decomposition is illustrated in Figure 2b. The functions needed are associated with the leaf nodes of the graphs. However, when carrying out the decomposition, no association to functions is needed.

Precedence relations between data are identified by asking what data are needed for producing which data, starting from the output border, ending up on the input border of the analysed system. The converse succedence relations are found by starting from the inputs asking what out-

puts can be produced. When the graph this way is made stable, the combined precedence and succedence analysis is replaced by decomposition, called component analysis, of the data and nodes, and the analysis is repeated on a more detailed level.

We observe that precedence graphs are less concerned with implementation than data and control flow graphs. However, in practice, analysts are regularly confusing these issues. In complex systems they are unconsciously introducing both data flow and control flow in their analysis, which ideally should be concerned with precedence relations only.

In order to separate data and control, we will introduce direct precedence relations between data. This is shown in Figure 2c.

Rather than associating the logical operations with control, we will associate them with the data. The result is shown in Figure 2c. Here the logical operations are depicted as being subordinate to the data which initiate the processing. Other approaches are conceivable. However, the details of language design are outside the scope of this paper.

The processes associated with control are considered to be generic and to perform the following function:

For each data item appearing on the input

- *Check if its class exists*
- *Validate its stated references*
- *Enforce the stated logical constraints and derive the prescribed data*
- *Issue the derived output.*

In order to access input to the validation functions, precedence relations are needed. In order to assign the result to output, succedence relations are needed. Therefore, two-way mappings are introduced. The resulting architecture is called a data flow architecture, because processing is initiated based on the appearance of input data.

Precedence relations are similar to functional dependencies used when normalising data according to the Relational model. However, functional dependencies are stated between individual data items, while precedence relations are stated between data sets.

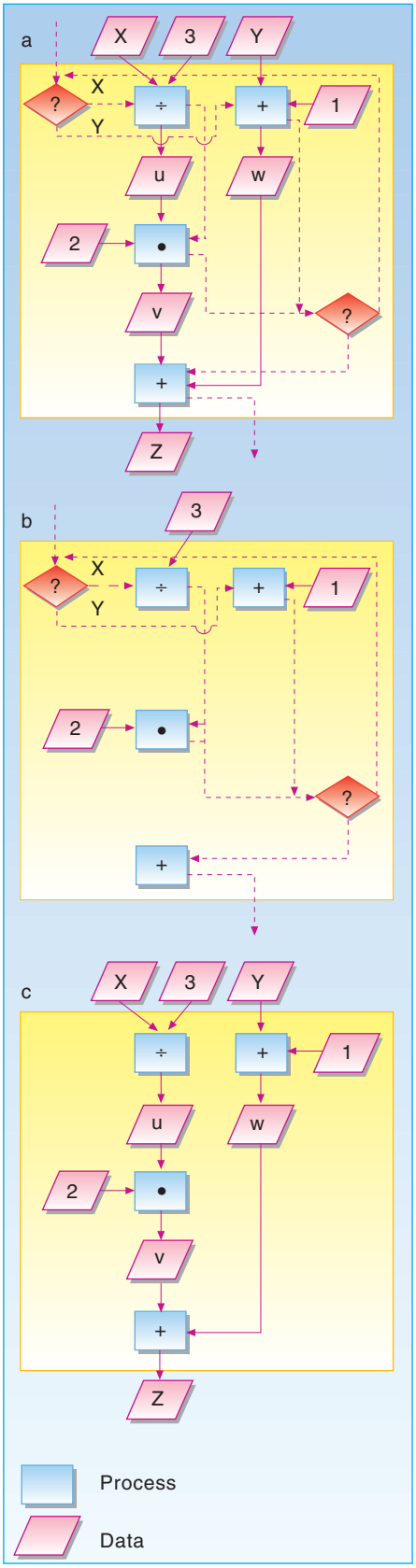


Figure 1 Depiction of an example total program (a), control flow (b) and data flow (c). The function performed is:
*Z := (X - 3) * 2 + (Y + 1)*

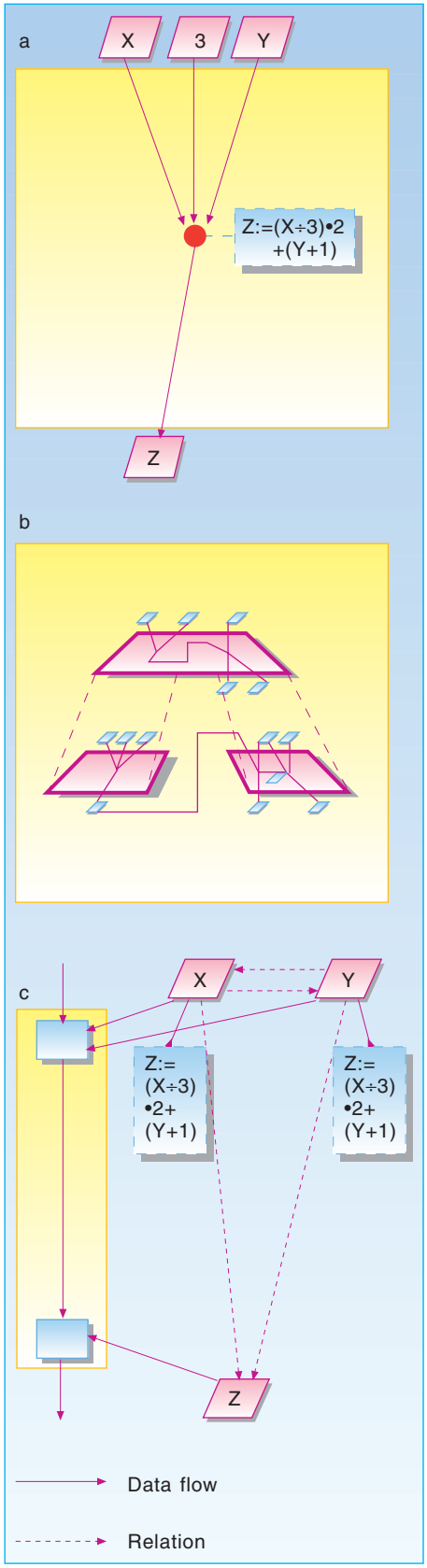


Figure 2 Simple example precedence graph (a) and how it is identified by decomposition (b). In (c) data are associated with relations and logic

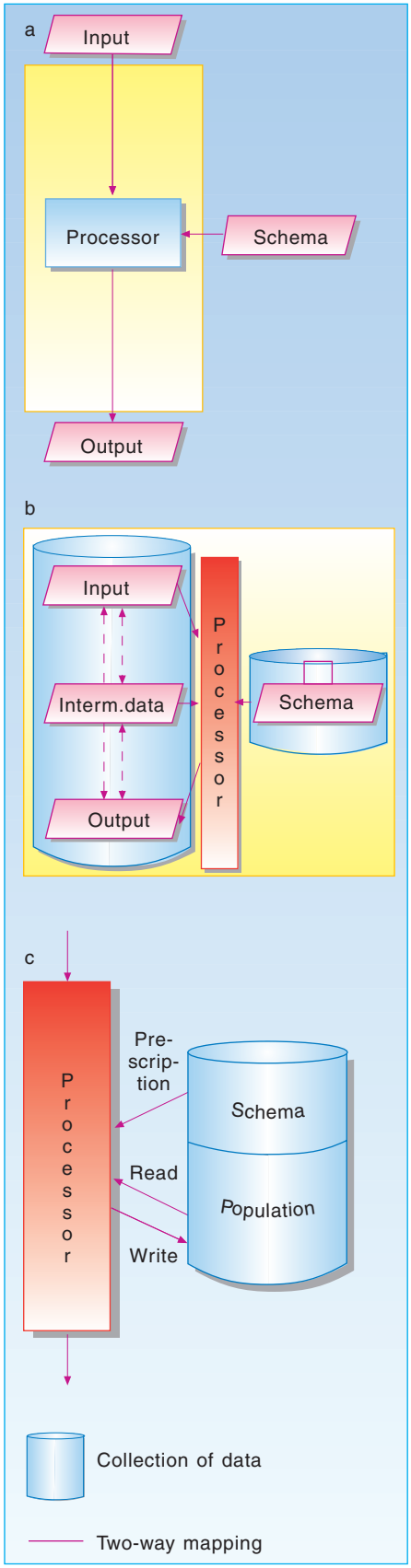


Figure 3 Separation of Schema (a). Identification of Population (b). Unification of Schema and Population data (c)

Software in the Data flow architecture is partitioned into:

- Schema, consisting of Data declarations and Logical statements to be enforced on these data
- Processor, the mechanism which implements the control functions that enforce the rules stated in the Schema.

This generic separation is depicted in Figure 3.

The collection of data instances which are enforced according to the Schema of the software system are collectively called the Population relative to the Schema. The Population data comprises input, output and intermediate data, e.g. u, v, w in Figure 1.

We recognise that, so far, all graphs illustrate data classes and not data instances, except in Figure 2c. Here the direct data flow arrows between the processes depict flow of data instances that are enforced according to the rules stated among the classes found in the Schema. There may be no absolute distinction between Schema and Population data. To be schema data and population data are just roles played by the data relative to the processor and to each other. However, this issue is outside the scope of this paper.

We will end this section with a last remark about notation. The mapping between schemata states that there exist (two-way) references between data inside the schemata. It is the references between the data items that state the exact data flow. The details of this is also outside the scope of this paper.

The notions introduced in this section allow the separation of data definitions, including logical operations, from control flow. This again allows the system analyst to concentrate on defining the schema part without bothering about implementation. We will use these notions to identify candidate interfaces for interoperation between software function blocks.

Layering

For two processors to be able to communicate, they have to share a common 'language', i.e. they must have the same data definitions for the communicated data. Therefore, in order to identify interfaces inside a software system, we have to identify the data that can be communicated between software blocks and constraints and derivation rules for these data. These rules make up the schemata. Hence, we will first identify the candidate schemata.

Data on different media can be defined by separate schemata for each medium:

- External schemata define the external presentation and manipulation of data
- Internal schemata define the internal organisation and behaviour of data.

See Figure 4. If we want to allow communication from all to all media, we have to define mappings between every two schemata. Rather than mapping each schema to every other schema, we introduce a common centralised schema, labelled the Application schema.

This Application schema

- defines the constraints and derivations which have to be enforced for all data, independently of which medium they are presented on.

See Figure 5. Additional notions are defined as follows:

- System schema contains, except from the External, Application, and Internal schemata:
  - System security data, including data for access control
  - System directory data, including data for configuration control
- System processor, includes the External, Application and Internal processors, administrates their interoperation and undertakes directory and security functions
- System population contains, except from the External, Application, and Internal populations, data instances of the System directory and System security
- The notion of a Layer comprises a grouping of processors which enforce a set of schemata on corresponding populations, including these schemata and populations; no population can have schemata outside the layer; the processors, schemata and populations of one layer have similar functionalities relative to their environment.

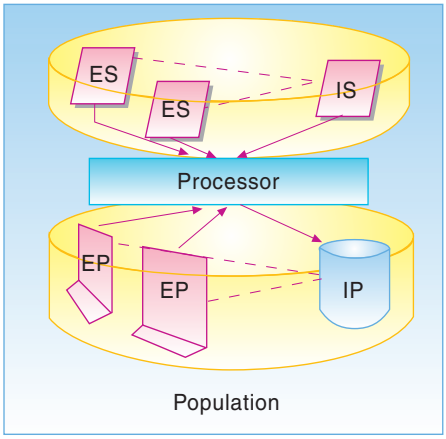


Figure 4 Depiction of presentation schemata for each medium. Mappings between data are indicated by dashed lines. The sylinders indicate collections of data

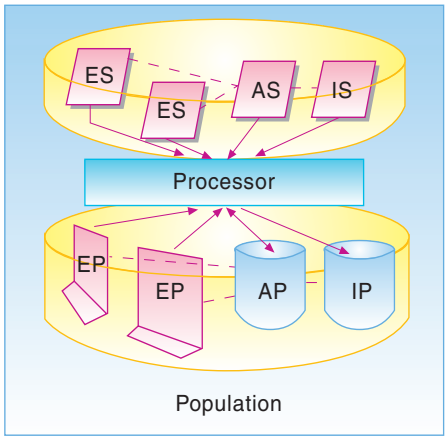


Figure 5 Introduction of one centralised Application schema for each system. S = schema, P = population, E = external, A = application, I = internal

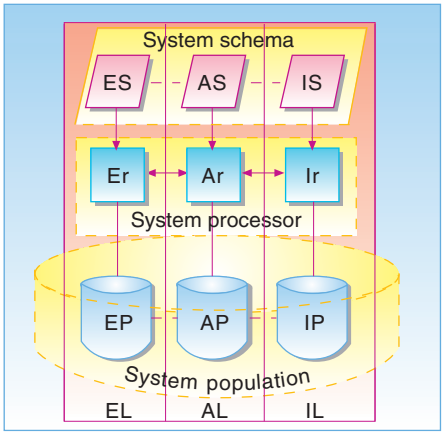


Figure 6 The 3-layered software architecture, consisting of the External (EL), Application (AL) and Internal layers (IL), r = processor

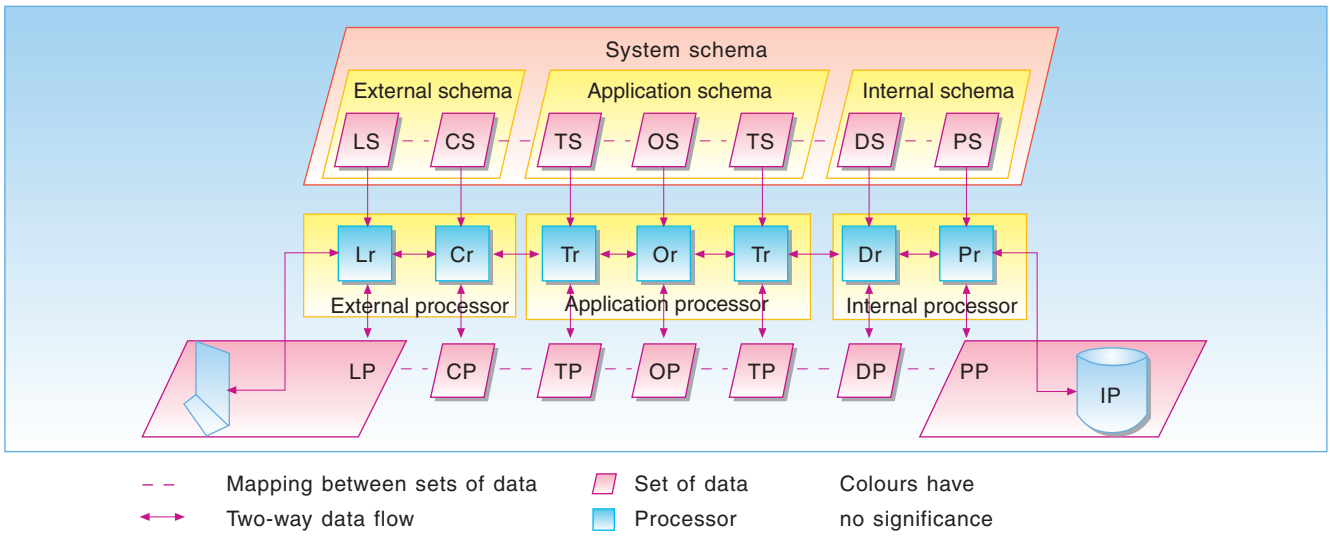


Figure 7 Data transformation architecture. Each layer contains schemata, corresponding populations and a processor. L = layout, C = contents, T = terminology
The layered architecture can undertake transformation of data between any two media, here examplified by a screen and a database. O = concept, D = distribution, P = physical

See Figure 6. Each layer can be decomposed into a sublayer, containing corresponding component schemata:

- External schema (ES), is composed of
  - Layout schemata (LS), which defines the way data are presented to the user
  - Contents schema (CS), which defines the contents and structure of the selected data and permissible operations on these data in a specific context
- Application schema (AS), is composed of

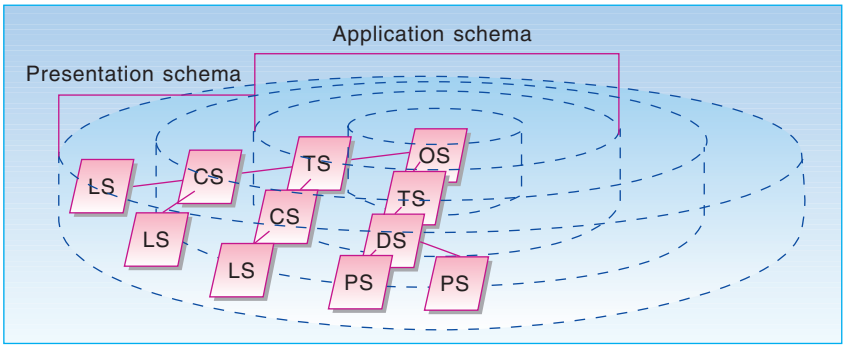


Figure 8 Data flow between layers. The external and internal layers are unified into Presentation (sub)layers around a common Application layer.
Mappings between collections of data state the permissible data flow (both ways), without any concern of processors undertaking the communication.
The use of Concept schema can be avoided by using one of the Terminology schemata as a substitute

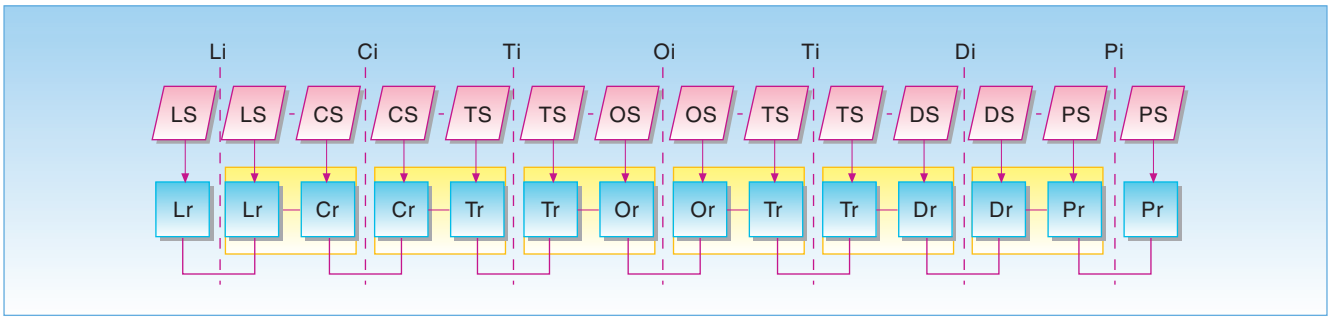


Figure 9 Candidate interfaces (i) for interoperability between function blocks

Yellow boxes are used to depict the processes (i.e. function boxes).
Note that communication can take place between identical schemata only

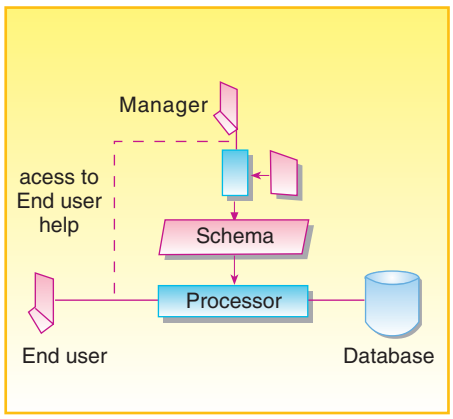


Figure 10 Nesting of the Reference model

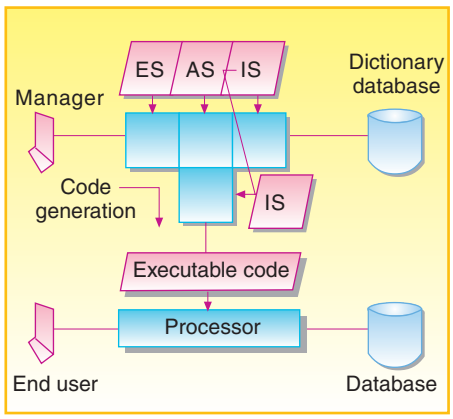


Figure 11 Code generation

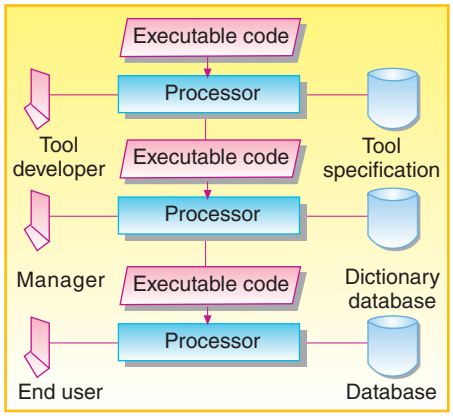


Figure 12 Bootstrapping of the system

  - Terminology schema (TS), which defines the common terminology and grammar for a set of external schemata
  - Concept schema (OS), which defines the common structure, constraints and derivation of data, common for all terminologies that are used in a system

- Internal schema (IS), is composed of
  - Distribution schema (DS), which defines the subsetting of the Application schema for one medium and the permissible operations on this medium
  - Physical schema (PS), which defines the internal storage, accessing, implementation and communication of data and their behaviour.

See Figure 7.

It is possible to have several alternative presentations, defined in the Layout schemata, from one Contents schema. It is possible to have several alternative selections of data, defined in the Contents schemata, from one Terminology schema. It is possible to have several alternative terminologies, defined in the Terminology schemata, of the same notions, defined in the single Concept schema of one software system. See Figure 8. This figure illustrates that mappings asserting permissible data flow can be stated between schemata without any mentioning of processes.

Figure 9 illustrates how interfaces can be identified and processes can be added when the schemata are given. Two processes have to share a common language to be able to communicate. The schemata of the reference model make up the candidate 'languages' for communication. Therefore, they are called Reference points for communication. The schemata serve as communication protocols between communicating processes. Data are enforced according to the (schema) protocol on both sides of the communication link. Note that all transformation and derivation of data take place inside the processes.

Nesting

In some cases the initial common language of two processes can be very limited. This language can be extended by data definitions using basic constructs of the limited language only. The data definitions must then be communicated using the limited language, before communicating the data instances according to these definitions. The limited language is a schema of the data definitions. The data definitions make up a schema to the data instances. The limited language is a meta-schema relative to the data instances. This way, by recursive usage of the reference model, we can get general and powerful means of communication and computation. The details of this approach will not be dealt with in this paper.

Let us study a second way of nesting the reference model. The current layered data flow architecture, as depicted in Figures 7 and 8, is appropriate for communicating data instances between two media. The data instances are basically the same on these media. However, their physical appearances and organisations

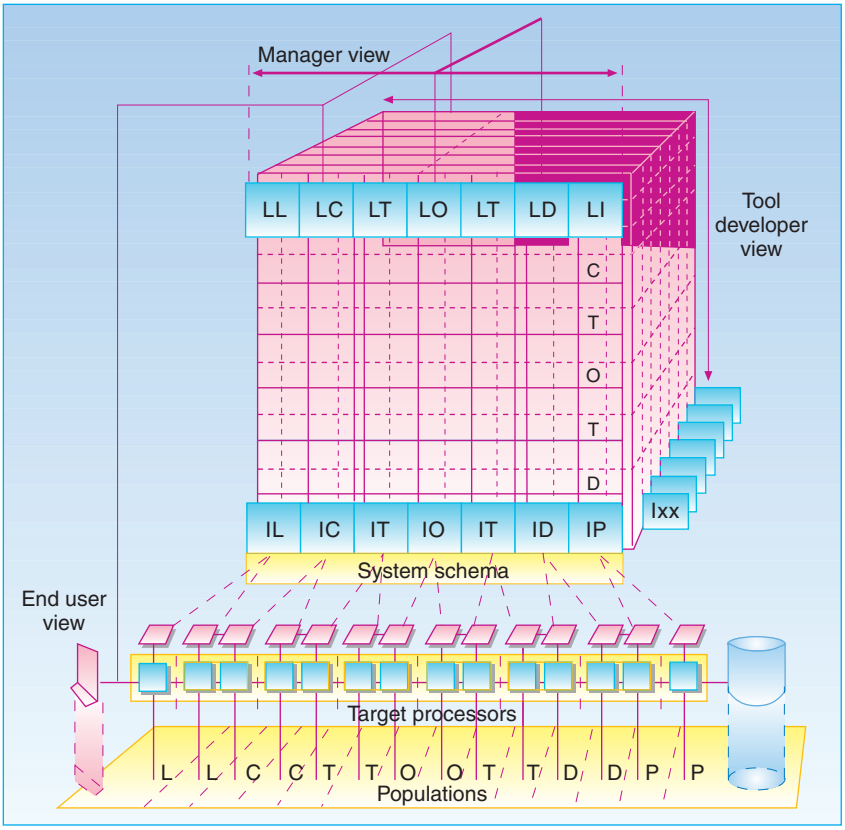


Figure 13 The schema cube. The horizontal dimension illustrates usage, i.e. data flow between the HMI and the internal database. The vertical dimension illustrates management, i.e. data flow from the manager to generated code.

The dimension going into the plane illustrates tool development. This defines the manager's environment

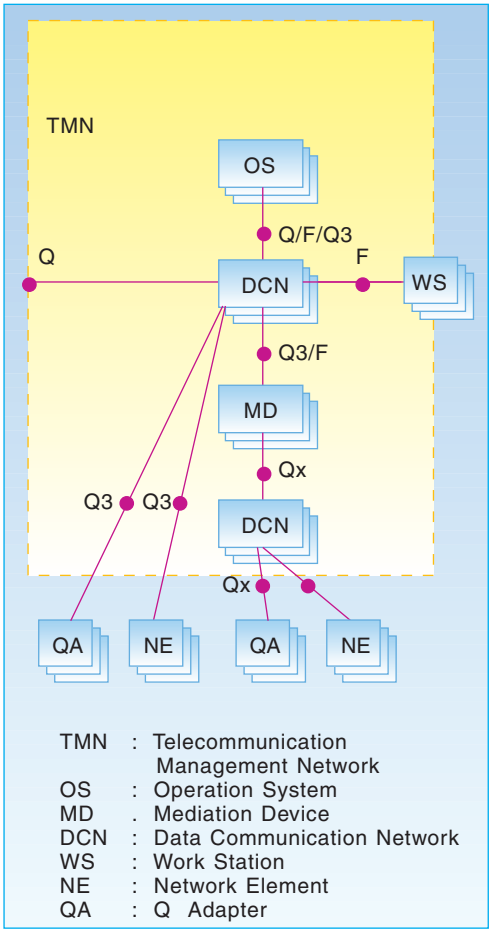


Figure 14 The TMN physical architecture

can differ. Also, their classes, defined in the schemata, can be different. This allows for substituting e.g. French headings with Norwegian ones. If, however, we want to replace one set of instances in one terminology with another set in another terminology, the existence of both and the 'synonymity' references between them have to be persistently recorded in a database. For example, John may correspond to 12345, Bill to 54321, etc. This way, the references between data in different Terminology schemata may not only state flow of transient data, but can state references between persistent data. Since all persistent data are stored in a database, the reference model itself is needed to manage these references of the reference model. This way, we get a nesting of the reference model.

A third way of nesting the reference model is shown in Figures 10 through 13. Since all schema data are ordinary data, these data can be managed using the reference model itself. This is illustrated both for the management of schemata and meta-schemata. The result is a very complex topic, labelled 'The schema cube'. The main message in these figures is that it is not sufficient to settle what schema (or interface) to specify. You also have to choose what form (and what interfaces) to apply to this specification. For example, you may choose a Layout form for the specification of the Layout schema, or you may choose the internal Terminology form, for programming the Layout schema.

Comparisons

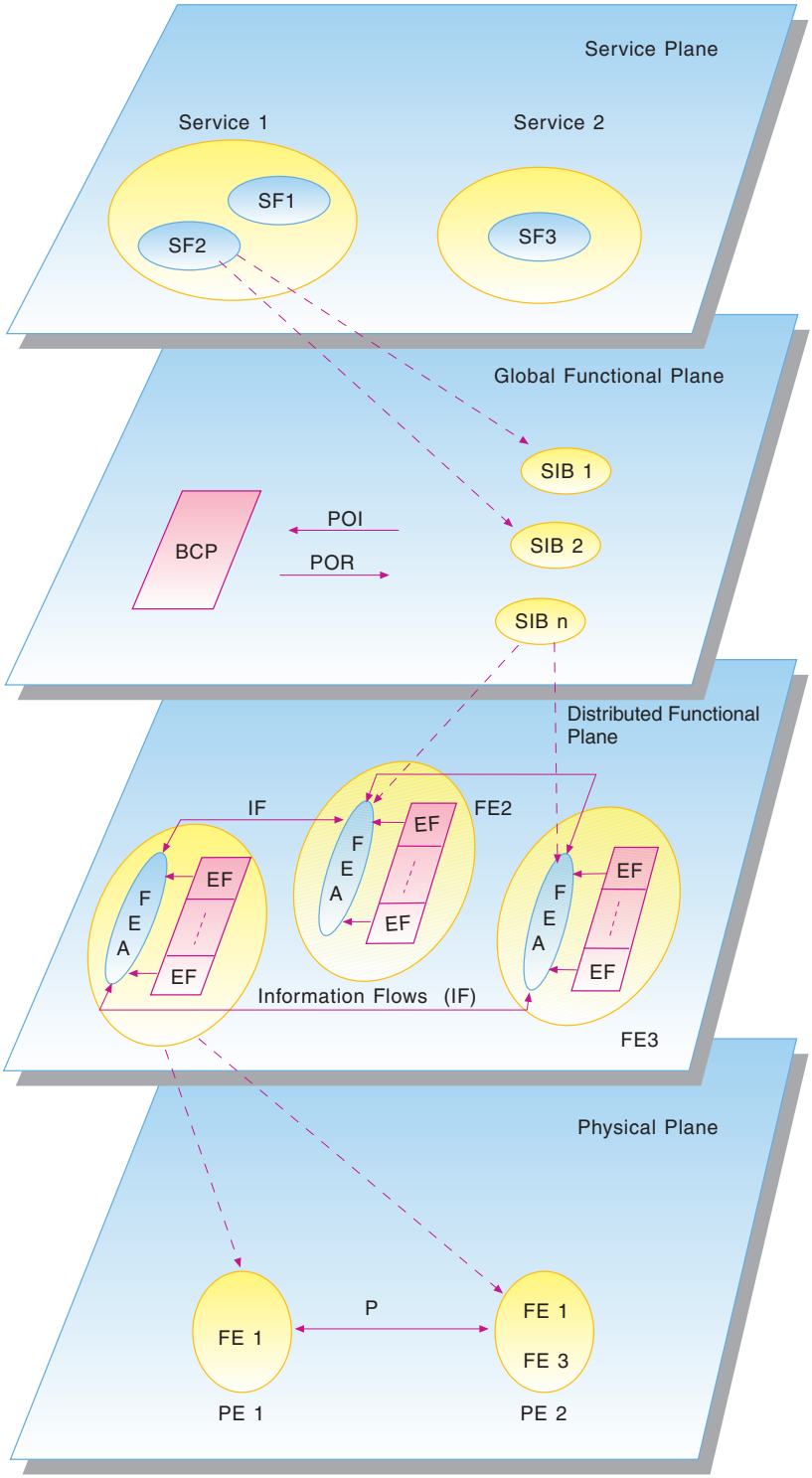
Figure 14 depicts the overall Telecommunications Management Network reference model (5). This reference model delimits the TMN from its outside world, and work is undertaken to define the interfaces to this outside world. However, this way, the TMN model is an organisational model, which defines what is inside and outside TMN, and is not a reference model in the technical sense we have been discussing reference models in this paper. Exceptions can be made for the f and g interfaces, which are technical. However, no progress is made on specifying these. Figure 9 depicts the available reference points from a technical point of view. It is not clear which of these reference points are intended for any of the TMN interfaces. Also, even if TMN languages are defined for some of the TMN interfaces, it is not clear what schema they correspond to, and the relevant requirements on these languages are not clearly identified.

The Intelligent Network architecture is shown in Figure 15 (6,7). This architecture has got a much more technical character than that of the TMN model. The bottom 'planes' correspond quite well to the internal layer of Figure 7. But a minor warning must be made. In Figure 9, the users, terminals, subsystems, logical files, and physical resources are defined in the directory part of the System schema and not inside the Distribution schema. The Distribution schema states just what contents is mapped to a physical resource. This allows communication and 'distribution' to take place on any of the interfaces. In the IN

architecture this seems to take place in the Distribution plane only.

When we come to the top planes of the IN architecture, it diverges even more from the Interoperability reference model presented in this paper. The Function plane may correspond to the Application schema. But the Service plane does not fit in. The reason is that what is specified inside the Service plane corresponds to the External schema of the telecommunication service user. Therefore, this should be at the same level as the Function plane. For accessing both the Function plane and Service plane we need the Manager's external schemata, i.e. of the TMN operator. This requires a nesting of the reference model, as shown in Figures 10 through 13, which is not provided in the IN architecture.

This comparison of reference models is a first exposition of what kind of difficulties will be encountered when attempting to analyse and integrate different systems developed according to different reference models. (8) provides a comparison of the TMN and IN reference models. The result is summarised in Figure 16. However, the paper lacks a technical and methodological analysis, as suggested in this article. The methodological problem with the two reference models and the comparison is that they start with identifying function blocks, which necessarily must be implementation oriented. As explained in this paper, the analysis and design of reference models should start with identifying the data and their relationships.



SIB : Service Independent Building Block
FE : Functional Entity
SF : Service Feature
IF : Information Flow
POI : Point Of Initiation
FEA : Functional Entity Action
PE : Physical Entity
EF : Elementary Function
P : Protocol
POR : Point Of Return
BCP : Basic Call Process SIB
– → : Pointer

Figure 15 IN conceptual model

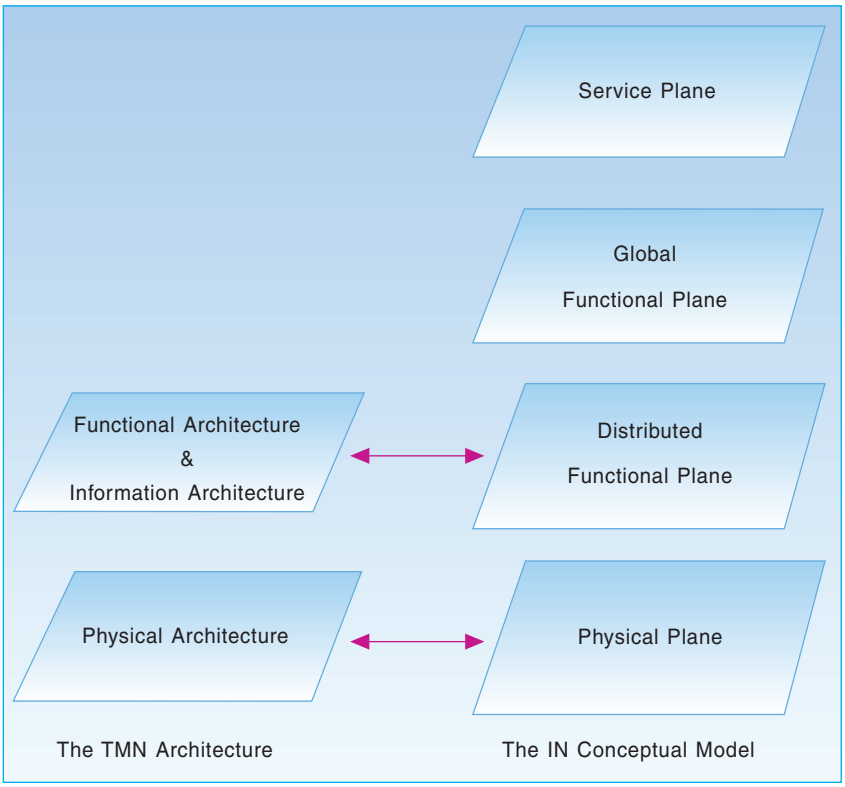


Figure 16 Correspondence of the TMN architectures and the planes of the INCM

References

1 Meisingset, A. *Draft interoperability reference model.* Norwegian Telecom Research, Systemutvikling 2000, D-3 (2). Also contributed as Working document to ETIS, Berlin 1993.

2 Meisingset, A. *Identification of the f interface.* Norwegian Telecom Research, Systemutvikling 2000, D-5 (2). Also contributed as Working document to ITU-TS SG 4, Bath 1993.

3 ITU. *Draft Recommendations Z.35x and Appendixes to draft Recommendations,* 1992. (COM X-R 12-E.)

4 Langefors, B. *Theoretical Analysis of Information Systems.* Studentlitteratur Lund, 1966. Philadelphia, Auerbach Publishers Inc., fourth edition, 1973.

5 ITU. *Draft Recommendations M.3010,* 1991. (COM IV-R 28-E.)

6 ITU. *Draft Recommendations Q.1201,* 1992. (COM XI-R 108-E.)

7 Skolt, E. *Standardisation activities in the intelligent network area. Telektronikk* 88(2), 1992.

8 ETSI. *Baseline document on the integration of IN and TMN.* European Telecommunications Standards Institute, 1990.

The draft CCITT formalism for specifying Human-Machine Interfaces

BY ARVE MEISINGSET

681.327.2

Scope and its impact

This paper introduces the formalism part of the draft CCITT Data Oriented Human Machine Interface (HMI) Specification Technique (1). This formalism can in technical terms be characterised as a context-sensitive language in an attachment grammar for specifying the terminology and grammar of HMI data. The formalism allows the developers and expert users to use the end users' own terminology only when writing and reading HMI specifications.

In order to enforce harmonised presentations across many screens, there is a need for a common and centralised definition of the

- terminology
- grammar

for large application areas. This is called the Terminology schema of the HMI. In order to avoid divergent and hence, not easily recognisable presentations, we have to define all terms as they are actually presented. Likewise, in order to make the statements recognisable to the end users, we have to define the common presentation rules/word order for all statements in this application area.

This objective, to define and harmonise the terminology and grammar, differs from the scope of most other ongoing work. Most work on human-machine interfaces (HMIs) have focused on style guidelines for surface structure aspects, like windowing, use of buttons, menus, colours, etc. Most work on specification techniques have focused on defining concepts or facts independently of presentation form and word order. However, when wanting to make data recognisable to the end users, harmonisation of the deep structure form (terminology) and word order is needed. Hence, to define concepts and facts only is not satisfactory.

For these reasons, the draft CCITT HMI Specification Technique is addressing other aspects than that of other and related work, e.g. OSI management specifications (2) for the management of telecommunications networks. The different objectives will lead to different solutions and to some co-ordination problems. Issues not considered in previous specifications will have to be added and lead to additions to and modifications of existing specifications. The HMI is such a central issue that the concerns made in this area will have to be taken as a starting point for the design of any information system.

The formalism and its motivation

Local names

In HMIs for large application areas we want to have capabilities to reuse the same class labels for different purposes in different contexts. Therefore, the HMI formalism has to handle local class labels within the context of a superior class label. This is illustrated in Figure 1.

Data tree

The upper part of Figure 1 depicts a data tree. This notion of all data making up a tree is implied by the requirement of handling local names. A data item in the data tree can have one superior data item only. Each data item can have several subordinate data items. Each label is only unique within the scope of the superior

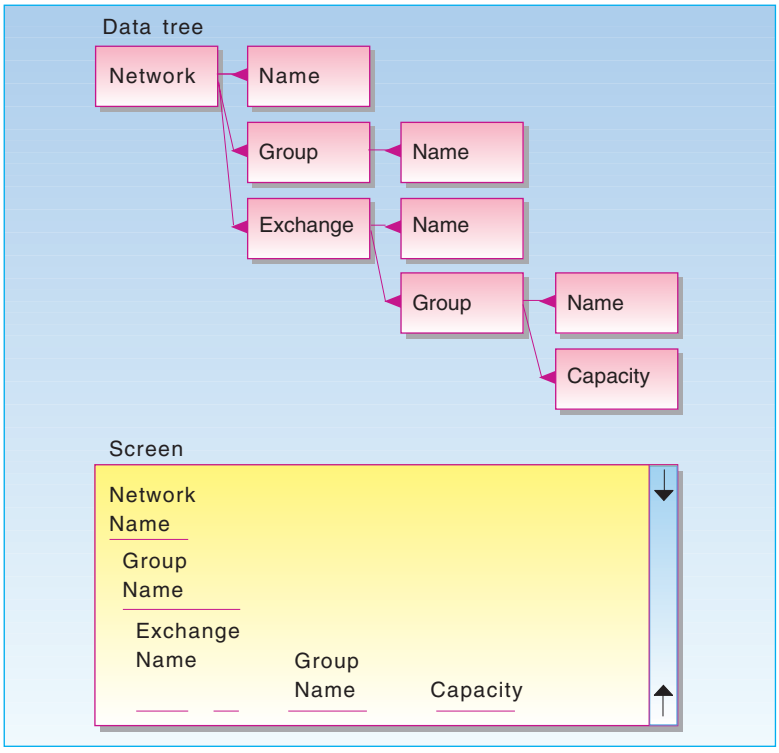


Figure 1 Local names
A Group of the Network is a multiplex group. A Group of an Exchange is a circuit group. However, they are both just labelled Group. The two groups have different attributes. In their different contexts, Network and Exchange respectively, the two identical labels both appear as headings on the screens. Also, in this example there are three different Name attributes in different contexts having different value sets. See the example screen picture

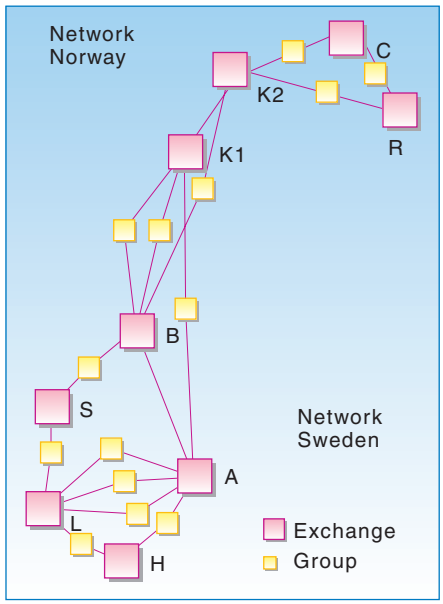


Figure 2 Example graphical presentation
The presented Network object NORWAY is only indicated alphanumerically. The containment arrows are likewise suppressed

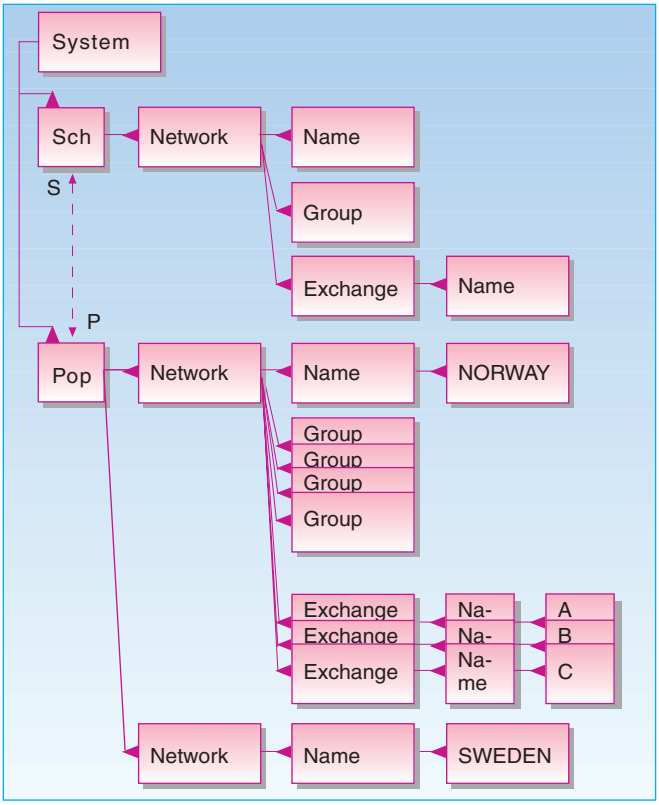


Figure 3 Correspondence between classes and instances
All classes are specified in a schema. This is indicated by the S at the end of the dashed two-way arrow. The data instances are found in the population, indicated by the P. Note that the data items in the populations are exact copies of data items in the schema branch. In this example, classes for the individual values are not illustrated. Also the references between Groups and Exchanges, shown in the map, are not illustrated

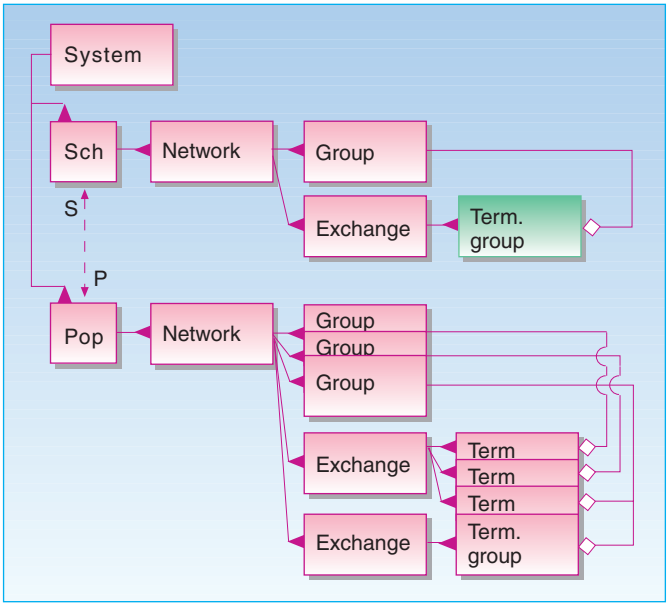


Figure 4 References
According to the schema, an Exchange can have subordinate Terminated groups, which refer (by an existential condition) to Group. Terminated group is a Role of a Group as observed from an Exchange. In the population one Exchange instance has three subordinate Terminated groups, each referring to different Groups. However, the last Exchange has one Terminated group only, referring to the same Group as that of the last Terminated group of the previous Exchange. This way, it is stated that this Group connects two Exchanges

data item and hence cannot be addressed without traversing the data tree from the root via all intermediate data items to the searched data item. Hence, in the basic data tree there is no relation notion, only subordinate and superior data items. This containment is indicated by a bird-foot arrow.

Instance labels

Figure 2 depicts instances of Exchanges in a map. The Exchanges are connected by Groups. The same icon is used for and is repeated for each Exchange and Group. Hence, the Exchange label is not only referring to the class, but the Exchange label, here symbolised by the icon, is repeated for every Exchange instance. Hence, the class labels do not only appear as intermediate 'non-terminal' nodes in the specification 'grammar', but the class labels are copied into every instance. This is illustrated in Figure 3.

In most alternative formalisms, only the values – the leaf nodes of the syntax tree – are left in the finally produced population. In an HMI formalism we see that the superior labels are necessary to indicate the classes and the contexts of the data instances. This is the way context-sensitivity is provided in the HMI formalism.

Instantiation

In Figure 3 we observe that superior nodes always must appear, while subordinate nodes may be omitted – if not otherwise required. Therefore, we call this formalism an attachment grammar, not a rewriting grammar. Note that due to the use of local names, the terms cannot be listed independently of the grammar. The data tree is the HMI grammar and lists the terms.

Since exactly the same notions appear in the schema and the corresponding population, we can use the same formalism for both. This is also different from most current approaches, which use separate class templates in the schema and have different or no notation for the instances. A benefit of using the same notions is that the user can easily foresee the permissible instances when reading the schema, or, when knowing some instances, he can easily find his way in the corresponding specifications (in the schema). Copying is the only transformation from classes to instances.

Significant duplicates

In Figures 2 and 3 we observe that many Groups appear without a name assigned to them. In fact, in a graphical dialogue to a network planning system, the entire network can be created without assigning names. Therefore, the HMI must be capable of handling significant duplicates. The fact that the EDP system may assign internal surrogates which uniquely identify each data item is outside the scope of our discussion on HMIs. The impact of this is that the HMI formalism itself must be capable of handling significant duplicates – in (nested) ordered lists.

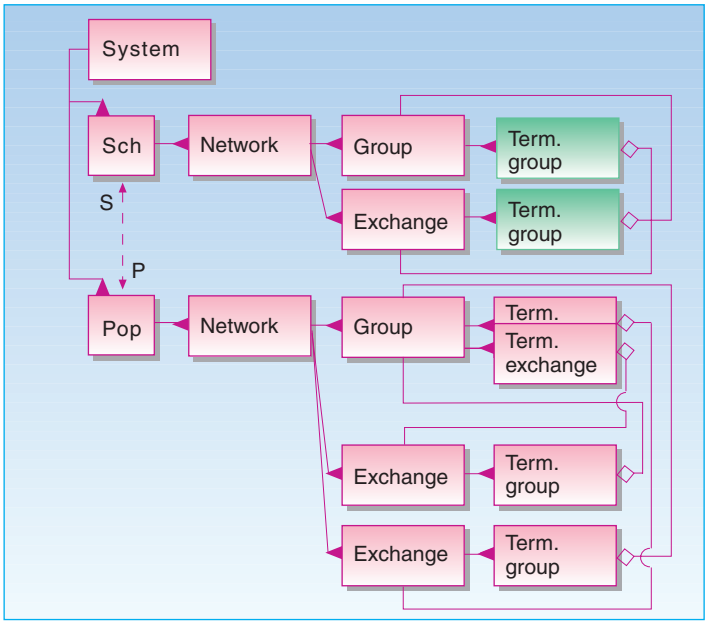


Figure 5 Mutually dependent references
A Group has two Terminating exchanges. Each of the two referenced Exchanges has a Terminating group referring to the Group. The intention is that there should be a mutual dependency between two opposite roles, Terminating exchange and Terminating group. We have so far not introduced means to express this dependency

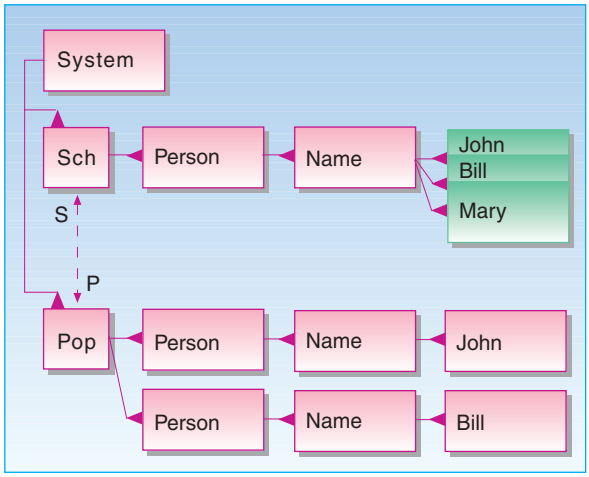


Figure 6 Values
Name in the schema has three subordinate data items: John, Bill, Mary. These can be instantiated in the same way as any other data item, like John and Bill as shown in the population. In this case we have instantiated one data item only subordinate to each Name. Also, the two shown subordinate items, John and Bill, of different Names of different Persons are different. However, there is nothing prohibiting multivalues and significant duplicates if these are not explicitly prohibited in the schema. Means for this are not yet fully introduced

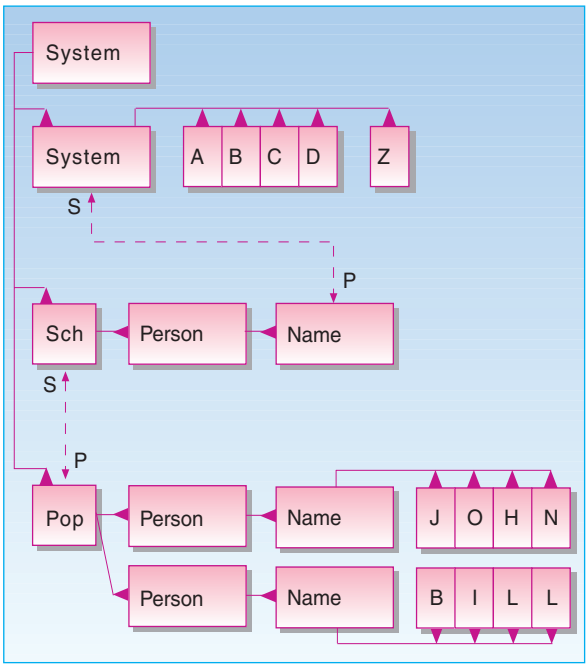


Figure 7 Recursive instantiation
Name of Person 'inherits' permissible subordinate values from its S(chema), which is Alpha. The subordinate data items of Alpha make up the alphabet extensively. There is no restriction on how many values can be contained in one Name. Hence, a Name can be any sequence of letters subordinate to Alpha. Pop 'inherits' its classes from Sch, and we see the chosen instances John and Bill

References

References between different nodes in the data tree can be stated by introducing Existential conditions. See figure 4. Such a condition states that the conditioned data item can only exist if the referenced data item exists. The conditioned data item is called a Role of the Referenced data item.

Also, independent or mutually dependent references in opposite directions between two data items can be introduced. This is illustrated in Figure 5.

Note that there can be several references between two data items, and the references may not necessarily be interdependent.

We see that the graphs are starting to become cluttered by all the details introduced. Later we shall introduce a simpler shorthand notation. However, for the time being we will stay with the current notation to show clearly some of the details involved.

Values

Values have been introduced informally in Figures 2 and 3. Figure 6 illustrates how we can specify a finite permissible value set.

We observe that everything permissible in the population first has to be stated in the corresponding schema. The schema contains the originals which the instances are copies of.

Recursion

We are now ready to introduce recursion. The means for this is the schema-population reference. We have already stated and shown that for HMIs we will use the same formalism for instances and classes. Therefore, the schema branch of the data tree, or parts of it, can be considered instances of still other classes. This is illustrated in Figure 7.

When the schema-population references are applied recursively, we can consider the schema data to be contained in a database in the same way as the population data. The form of the schema data is controlled by a meta-schema. The effect of this is for the end user that he can access population data and schema data in exactly the same way, i.e. by the same means. Also, he can navigate between these data without having to change presentation style or dialogue. In fact, this has been a major objective of the CCITT Data Oriented HMI Specification Technique, to make all HMI specifications accessible and readable by end

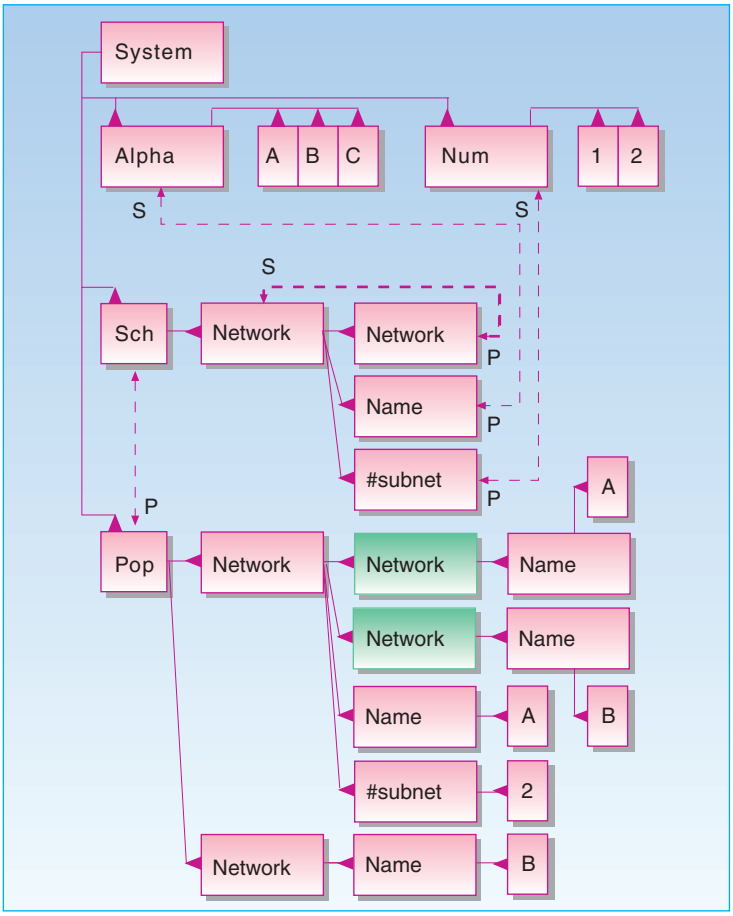


Figure 8 Real recursion
Sch has a subordinate Network, which has another subordinate Network in addition to Name and #subnet (number of subnets). Name and #subnet inherit their permissible values from Alpha and Num, in the ordinary way. The important construction here is the subordinate Network, which inherits all (subordinate) properties from its superior Network. This way, even the inheritance mechanism is inherited. Therefore, this allows any Network in Pop to contain Networks recursively in an arbitrarily large tree. Note that the subordinate Networks are identified locally to their superior Networks only

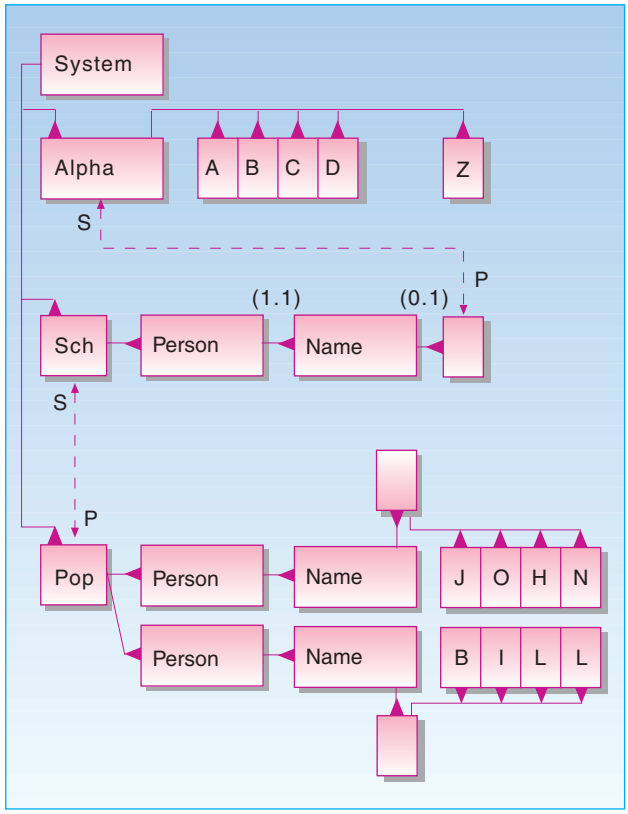


Figure 9 Cardinality
The schema contains no constraint on the minimum or maximum number of Persons in the Pop. The depicted Pop contains two Persons. However, for each Person there has to be exactly one Name. For each Name there can be minimum zero and maximum one subordinate value (here indicated by a blank box). Note the distinction between the cardinality constraints on the Name itself relative to its superior Person and the cardinality constraints on the value subordinate to the Name. The value can contain an arbitrarily long sequence of letters – inherited from Alpha. Also, the value can contain significant duplicates, like the double l in Bill. Note that in this example the value itself (subordinate to Name) has no label

users. The purpose of this paper is to introduce the formal apparatus for making this goal achievable.

The scope of the end users' HMIs is defined to comprise:

- HMI population data
- HMI Layout Schemata
- HMI Contents Schemata
- HMI Terminology Schema.

For explanations, see (1, 3, 4, 5). The contents of the Internal Schemata is normally considered to be outside the scope of the HMI.

Since the same formalism is used for both instances and classes, we do not need to distinguish between classes (e.g. Name) and types (Alpha). We can use the same formalism for instances, classes and meta-classes. 'Inheritance' of 'code' from one of them to the other is stated by using the schema-population references. Note that there is no strict hierarchy between instances and classes. One data item can contain schema-references to several other data items, and classes can even 'inherit' from their instances. Note also that there is no distinction between inheritance and instantiation. This is due to the required free formation of class and instance labels, which makes ordinary inheritance obsolete.

In the above example we have used the schema-population mechanism repeatedly. However, this is not really recursion. Real recursion we get when the same classes are used to define the contents of themselves. This is illustrated in Figure 8.

Figure 8 should clearly exemplify that a data item is not defined extensionally by its component data items. Rather, several data items can be attached to an existing data item.

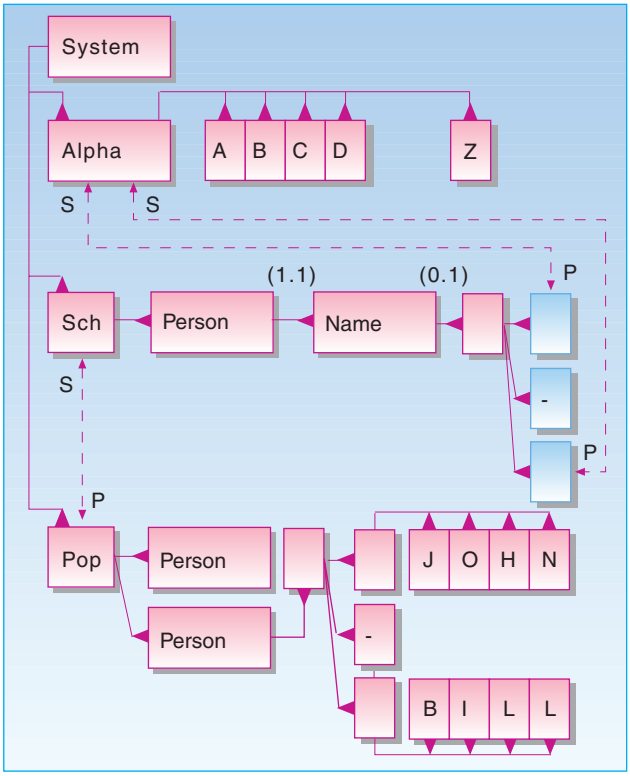


Figure 10 Specification of syntax of values
A Name is defined to consist of three parts: a first name, a blank (-) and a family name. The depicted Person's Name is John Grey

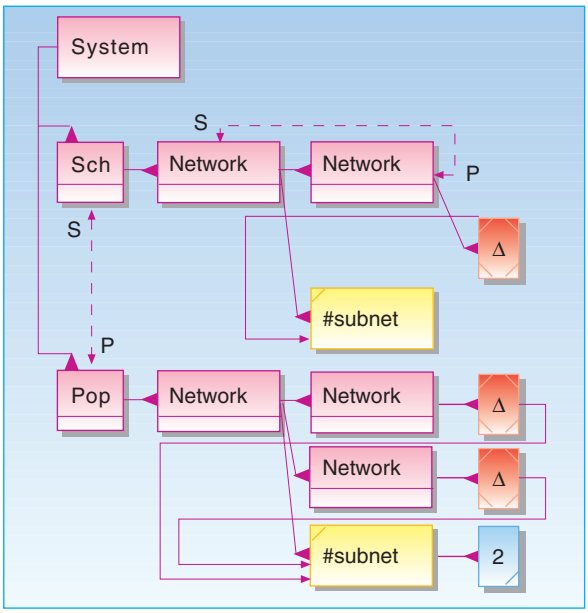


Figure 12 Functions
Δ is an incremental derivation Function. When a subordinate Network is inserted, the #subnet of its superior Network is incremented by 1. The details of the navigation to the right #subnet is not covered by this paper. If a subordinate Network is deleted, the #subnet of its superior Network is decremented by 1. The shown Δ Function has no Input, only an Output. Inputs and Outputs use hooked arrows. Note that functions receiving data subordinate to #subnet are not illustrated

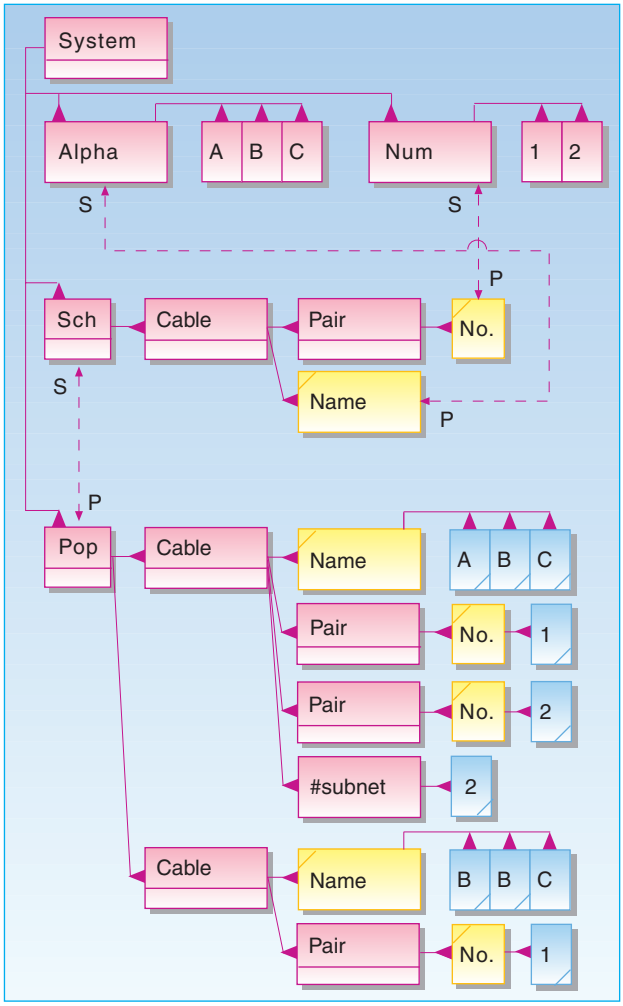


Figure 11 Local codes
Pair being defined subordinate to Cable implies that uniqueness of the Pair identifying attribute No. is only controlled within the scope of the superior Cable

Cardinality constraints

So far, we have introduced three notions of constraints: all data make up a data tree, every data item is an instance of one class only, and we have existential conditions. Now, we introduce the cardinality constraint on the minimum and maximum number of instances of a data item relative to its superior data item. See examples in Figure 9.

The formalism allows the HMI data designer to choose between multi-valued attributes (i.e. several values in one Name) and repeating attributes (i.e. several Names of one Person) or the combination or prohibition of these options.

A unified language

The same formalism as already introduced can be used to state the detailed syntax of the values. See Figure 10.

Observe in Figure 11 that the detailed structure of the value remains available in the instantiated data. This structure can, of course, be suppressed in the mapping to the final presentation on the end user's screen. However, it can be important to make this structure available to the user. This is analogous to the presentation of control characters in document handling systems.

The strength of the HMI formalism is, as shown, that we can use exactly the same basic notions to specify the overall structure of the data as well as the detailed syntactical structure of the values. Also, the formalism can express the required complex context-sensitive

syntactical structures of HMI data, which is normally not achievable in most alternative formalisms.

Local identification of instances

The purpose of using local object class labels is to state that the instances of the subordinate object class are identified locally to an instance of the superior class. See Figure 11, and about objects in Figures 14 and 16. In general, instances are identified locally to each other when the corresponding classes are identified locally to each other.

Functions

Cardinality constraints and existential conditions are just special cases of a general Function notion. Functions are used to state arbitrary constraints on data and derivations of data. Functions are considered to produce data in an algorithmic way.

A Function can be defined subordinate to any data item and is instantiated in the same way as any other data item. Also, a Function can be defined to contain any other data item, including subordinate Functions.

Inputs and Outputs are special cases of subordinate data items to Functions. Inputs and Outputs are stating references to / roles of other data items in the data tree. A Function is illustrated in Figure 12.

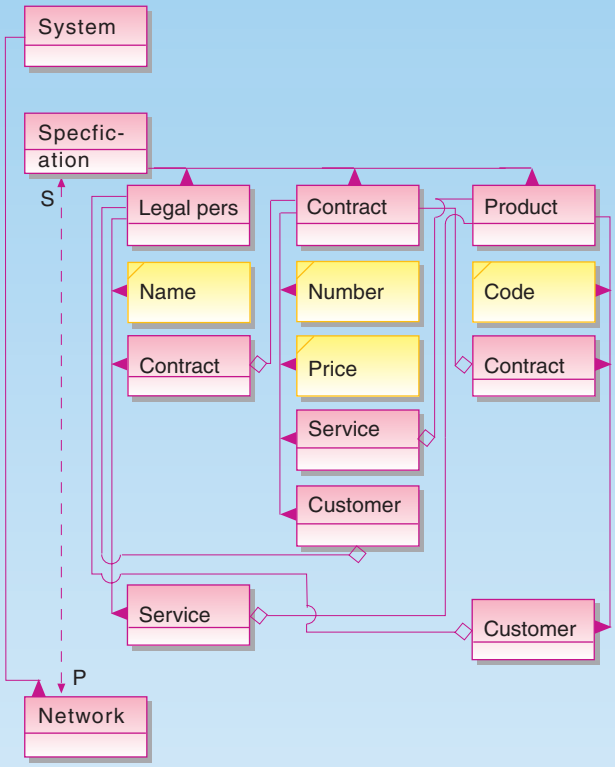
The Function notion allows constraint and derivation routines to be attached to any data item in the data tree, much the same way as in attribute grammars. This way, the HMI grammar is not only an attachment grammar in a data tree, but it also allows constraints, derivations and references to other branches of the tree.

Figure 14 Simplified alphanumeric notation for both classes and instances

The Specification has three subordinate object classes (underlined) with subordinate attributes and object classes. The latter are stating references to other object classes, but this is not indicated. Indentations indicate subordinate data items. Natural language definitions and explanations can be associated with each indented data item. References can be indicated by statements like 'Service refers to Product', etc. Cardinality constraints, not shown in Figure 14, are added. Corresponding instances are shown subordinate to Network. S(chema) and P(opulation) references are indicated

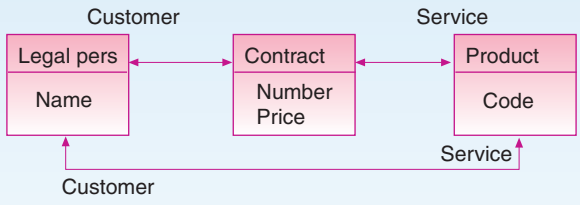
System
  Specification
    P
    Legal person
      Name (1,1)
      Contract
      Service
    Contract
      Number (1,1)
      Price (0,1)
      Service
      Customer (1,1)
    Product
      Code (1,1)
      Contract (0,1)
      Customer (0,1)
  Network
    S
    Legal person
      Name
      Contract
    Legal person
      Name
      Contract
      Service
      Service
    Contract
      Number
      Price
      Service
      Customer
    Contract
      Number
      Price
      Service
      Service
      Customer
    Product
      Code
      Contract
    Product
      Code
      Contract
      Customer

a. Detailed notation



The Specification has three subordinate object classes (underlined): Legal pers, Contract, Product. These have the attributes Name, Number and Price, and Code respectively. The Contract is connected to the two others by references in both directions, introducing the roles Contract and Customer, and Service and Contract respectively. Legal pers and Product are interconnected by the two opposite roles, Service and Customer.

b. Shorthand notation



The System, Specification, and Network object classes and the schema-population reference are suppressed. Attributes are indented and depicted inside their superior object classes. Two-way arrows indicate mutually dependent references. One-way arrows indicate single references (not shown). Role labels are written next to the object class they refer to. Role labels can be suppressed if they are identical to the class label of the referenced object class. Cardinality constraints can be placed next to the Role labels (not shown).

Figure 13 Detailed and shorthand notations

Summary and comparison

In the previous section we have introduced some basic notions of the HMI formalism. However, we have not introduced the very primitive notions and not a complete set of notions. However, the set introduced should provide a good understanding of how the formalism can be used and rationales for introducing these notions in an HMI formalism.

The examples shown, so far, contain a lot of details. In order to make the figures more compact and more readable, we introduce a shorthand notation in Figure 13. This shows that the graphical notation has similarities to that of the Entity-relationship formalism, but the details and the application area are distinctively different.

Figure 14 provides an alphanumerical notation for defining subordinate data items only. Both classes and instances are shown. Typically, definitions and explanations are associated with each class entry and are indented to the same level.

At this point a short comparison with the OSI Management formalism (2) is of interest. The OSI Management formalism requires that all object class labels are globally unique and does not allow use of local class labels. Hence, the formalism is not capable of defining the terms as they appear on the HMI. Also, the restriction to the use of global class labels only, will make the data tree of the instances different from the data tree of their classes, i.e. they will not be homomorphic. This will make it difficult to the user to foresee the structure of the instances when the classes are known and vice versa. In fact, the OSI Management formalism applies to classes only and not to instances. This shortcoming on how to define class labels is even more evident for attributes. The formalism allows attribute groups to be defined when attributes are given. However, it does not allow attributes to be defined locally to other attributes and does not allow definition of repeating attributes, but allows for multi-values. The attributes are defined in separete packages, which make the specifications more difficult to overview. The formalism is not applicable to defining values, and use the ASN.1 notation for this purpose. Significant duplicates are neither permitted for classes nor for instances. The concrete syntaxes of the OSI Management formalism and ASN.1 are overloaded with technical and reserved words, which make them inapplicable for HMI usage. This syntax is such that the formalism cannot be used to define the structure of the dictionary containing the specifications, i.e. the formalism is not applicable to defining meta-schemata.

It is evident that the OSI Management formalism is not appropriate as a user oriented specification language, but must be considered a candidate high level programming language, or an implementation specification language, maybe for the internal Terminology schema (3). However, other object oriented languages or other high level languages are more general and powerful for this usage. Alternative formalisms exist for the specification of HMI data, like the Entity-Relationship formalism and the Relational (Model) formalism. These have similar and even more severe shortcomings than the OSI Management formalism.

Abbreviations

CCITT	Consultative Committee for International Telegraph and Telephone
EDP	Electronic Data Processing
HMI	Human-Machine Interfaces

References

1 ITU. *Draft Recommendations Z.35x and Appendices to Draft Recommendations.* (CCITT COM X-R 12.)

2 Kåråsen, A G. The OSI management formalism. *Telektronikk,* 89(2/3), 90-96, 1993 (this issue).

3 Meisingset, A. A data flow approach to interoperability. *Telektronikk,* 89(2/3), 52-59, 1993 (this issue).

4 Meisingset, A. *Perspectives on the CCITT Data Oriented Human-Machine Interface Specification Technique.* SDL Forum, Glasgow, 1991. Kjeller, Norwegian Telecom Research, 1991. (TF lecture F10/91.)

5 Meisingset, A. *The CCITT data oriented Human-Machine Interface specification technique.* SETSS, Florence, 1992. Kjeller, Norwegian Telecom Research, 1992. (TF lecture F24/92.)

The CCITT Specification and Description Language – SDL

BY ASTRID NYENG

681.3.04

Abstract

This paper gives an introduction to the CCITT Specification and Description Language SDL. The basic structuring notions in the language are process and block. A process is an extended finite state machine having a finite set of states, a valid input signal set and transitions (with possibly outputs of signals) between the states. The block construct groups together processes into a system with communicating, concurrently executing state machines. The communication is asynchronous and based on sending signals to and reading signals from infinite input buffers.

1 Introduction

SDL (1) is the Specification and Description Language recommended by CCITT (International Telegraph and Telephone Consultative Committee) for unambiguous specification of the behaviour of telecommunications systems. SDL is a fairly old language. The first version was issued in 1976, followed by new versions in 1980, 1984, and 1988. SDL has become a big language. In this introduction we will try to introduce the basic constructs without going into all possibilities and details. For those interested in a more thorough description of the language, we refer to textbooks, e.g (2, 3). The object oriented extensions are introduced elsewhere in this issue (4).

SDL has two alternative syntaxes, one graphical and one textual. In this presentation we will concentrate on the graphical syntax, since this form is regarded as being the main reason why SDL has gained in popularity compared to other specification languages missing a graphical syntax. The textual syntax is mainly motivated by the need of having an interchange format between tools.

Traditionally, SDL has been used to specify behaviour of real time systems, like telephone exchanges. We now see efforts to use it in new areas, like the specification of Intelligent Networks services (5).

2 The basic ideas

The most basic construct in SDL is the *process.* The process is an *Extended Finite State Machine.* A Finite State Machine consists of a set of states, a starting state, an input alphabet and a state transfer function. When receiving an input, the finite state machine goes through a transition to a new state as defined by the state transfer function. Thus, the essential elements of the SDL process are states, inputs (signals), and nextstates. In addition, a transition is composed of a possibly empty set of additional elements that specify certain actions (described later). A transition from a specific state is initiated by an input and terminates in a nextstate. An input consumes the corresponding signal from the input queue of the process.

An *extended* finite state machine is extended to include local variables. This allows transitions from one state to depend on the current value of local variables. Figure 1 gives an example of a simple process specification. The figure also shows the basic symbols of *states, inputs, outputs, tasks, decisions,* and the declaration of local *variables.*

One process gives only a sequential behaviour through a finite set of states. *Concurrent* behaviour is introduced in SDL by the block construct, which groups together a set of communicating processes. The processes are connected by *signal routes* which communicate the signals between the processes. One process can *output* signals which are delivered to other processes by the use of a signal route. The signals are put in the receiving process' input port. The input port is a FIFO-queue (First In First Out), so that the signals are treated in the same order as they arrive. The communication is asynchronous, i.e. there is no hands-hake-mechanism where the process which sends a signal has to wait for a reply. Figure 2 gives an example use of the block construct.

We have now introduced the most basic constructs in SDL, processes which are machines that are either in a state or is performing a transition between two states, blocks that group together communicating processes, and signal routes that convey the signals between the processes. In the next section we will treat the different constructs in greater detail.

3 Basic SDL

Basic SDL refers to a central portion of the language which is sufficient for many SDL specifications. This part of the language was the first to be defined, and the additional constructs are defined by giving rules for how they can be expressed using basic constructs only.

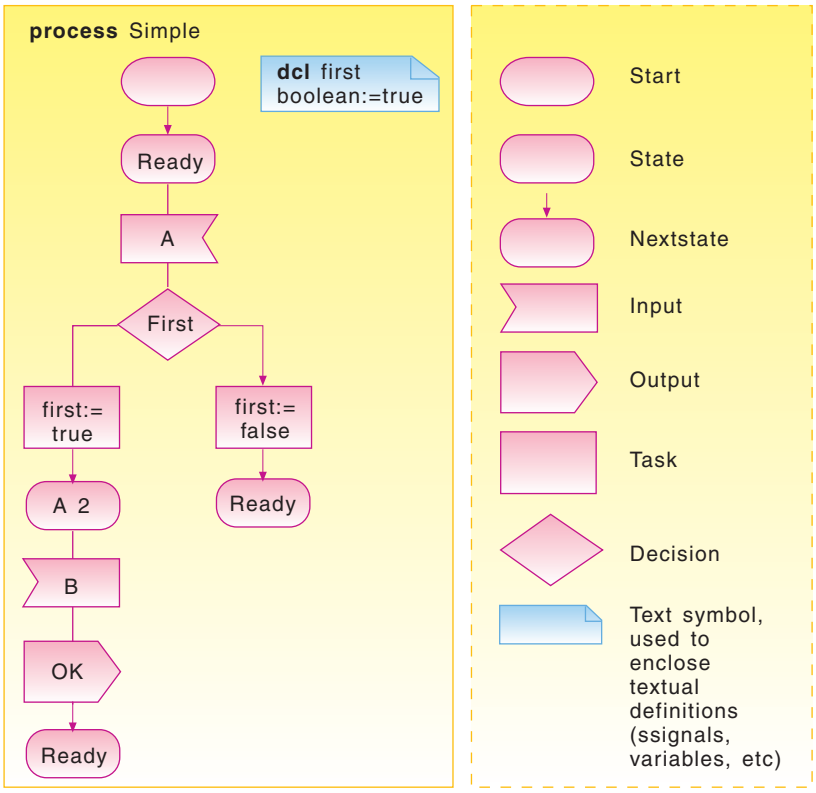


Figure 1 A simple SDL process
The process accepts strings containing repeated sequences of AAB. The process starts in the Ready state. In this state it only accepts the input A. After receiving A, the process checks whether it is the first or second A received. If it is the first, the process goes back to the same state, Ready, and waits for another A. If it has received two, it goes to the state A1 and waits for B

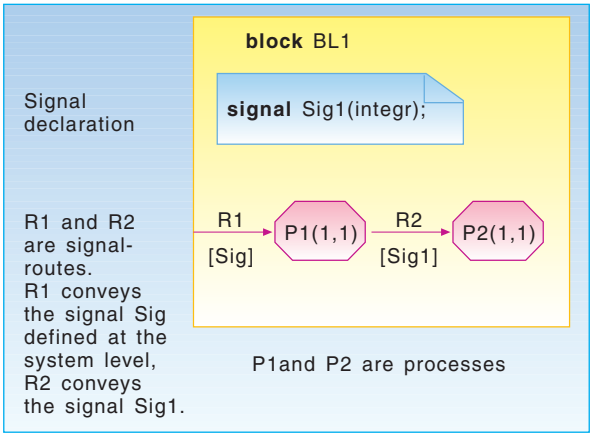


Figure 2 Example use of the block and signal route constructs
The block BLI consists of the two processes P1 and P2. The two processes are concurrently executing machines communicating via the signal routes

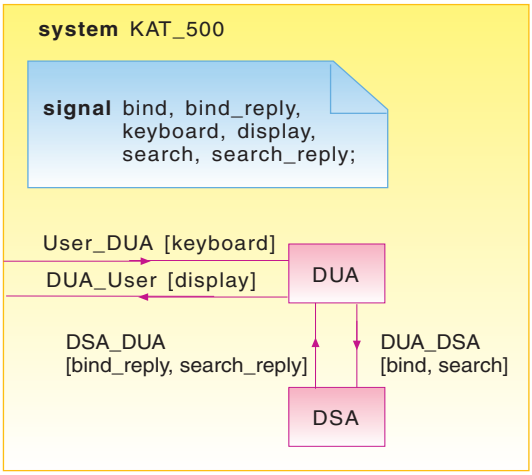


Figure 3 Example use of the block and channel constructs to specify system structure
The system KAT-500 consists of the Directory User Agent block (DUA) and the Directory System Agent block (DSA). The DUA can ask for a connection to the directory service by sending the signal bind via the channel DUA_DSA, and can ask for the search operation by sending the signal search. Response is received via the channel DSA_DUA. The DUA communicates with the user of the directory service by sending signals to the environment using the channel DUA_User, and receiving signals over the channel User_DUA

3.1 Specification of structure

SDL has several structuring concepts. The aim of this structuring is to manage complexity by hiding details in one part of the specification from another part and also to offer specifications of different details.

The most important structuring concept is the already mentioned block concept which groups processes. An SDL *system* can be partitioned into several blocks, connected by *channels.* The channels are either bidirectional or unidirectional communicating paths which convey the signals from processes within one block to processes within another block.

Figure 3 shows how the directory system KAT-500 (6) is specified as an SDL-system consisting of the two blocks DUA (Directory User Agent) and DSA (Directory System Agent) connected by the two channels DUA_DSA and DSA_DUA.

An SDL specification contains a hierarchy of components arranged in a tree-like structure. The system components are blocks, channels, data types, and signal definitions. The specification of a block can either be done locally, i.e. within the system, or be removed from the context and be referred to within the system. For real specifications of some size, the latter alternative is most frequently used. For the example in Figure 3, this means that the blocks DUA and DSA are only referenced within the system KAT-500, their specification is elsewhere. All specifications of components (signals, data types, variables) are visible from the point where they are specified and down to the leaves of the hierarchy. At the block level, processes, signals, data types, and signalroutes are defined, the processes can in addition contain *variables, timers,* and *procedures.* The procedure construct is similar to procedures found in programming languages. The procedure construct can be used to embody parts of the process graph, such that it can be used in different places in the process and also hide details in the process graph. A procedure is a state machine that is created when the procedure is called and stops when the call is returned. Figure 4 shows the main hierarchical levels in SDL.

3.2 Specification of behaviour

The process is probably the best known and most used construct in SDL. The processes deal with the dynamic behaviour of the system. A process definition can have one or more instances during its lifetime. The number of instances is declared in the process heading and gives the initial number of instances, i.e. the number of instances created at system initialisation, and a maximum number of instances which can exist at the same time. New instances can be created by other processes.

A process definition has a *declaration part* and a *process graph* (see Figure 1). In the declaration part, local signals, data types and variables are specified. The process graph is the specification of the state machine, which defines the process behaviour. The process graph contains a start transition which ends in some state. The start transition is the first transition to be executed after the process is initialised. The process graph has zero or more states in which the process awaits arrival of signals. In each state the process accepts one or more signals which trigger a possibly empty sequence of actions followed by a nextstate. Figure 5 shows the graphical symbols of the various actions allowed in the transitions.

3.3 Specification of communication

In the previous section we introduced the different constructs used to specify one process. We did not go into detail on how the communication between the processes takes place. Every process instance has an associated input queue where signals belonging to its so-called *complete valid input signal set* is put. The complete valid input signal set defines those signals that the process can accept. The allowed signals can be carried on one or more signal routes leading to the process. A signal can arrive at any time. Therefore every state should define what actions to perform for all signals. For some combinations of states and signals the transition can be empty and the state unchanged. This is not specified by the SDL user, since all signals not mentioned in a state are consumed and followed by an implicit transition back to the same state.

Outputs are used to send signals to other processes. The signals may contain values as parameters. There must always be a communication path between the sender and the receiver. If the receiving process does not exist when the signal is sent, the signal is discarded. SDL provides two possible ways to address the signal receiver:

- *Explicitly,* by specifying a process identifier (PId). In SDL, every process has a unique address, of the predefined sort PId. The SDL system manages all PIds and ensures that they are unique. There are four predefined expressions giving values of PId for each process. These are explained in Figure 6.

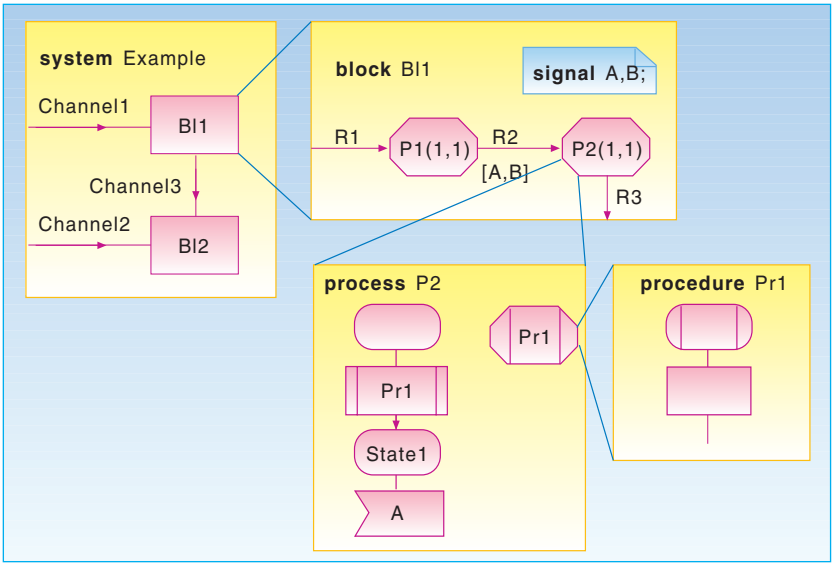


Figure 4 The four main hierarchical levels in SDL
SDL has four main hierarchical levels, the system, the block, the process, and the procedure. In addition, SDL defines block substructures (block in blocks) and services (another way of structuring processes).

System Example consists of two blocks, the block Bl1 consists of two processes, the process P2 defines and calls the procedure Pr1

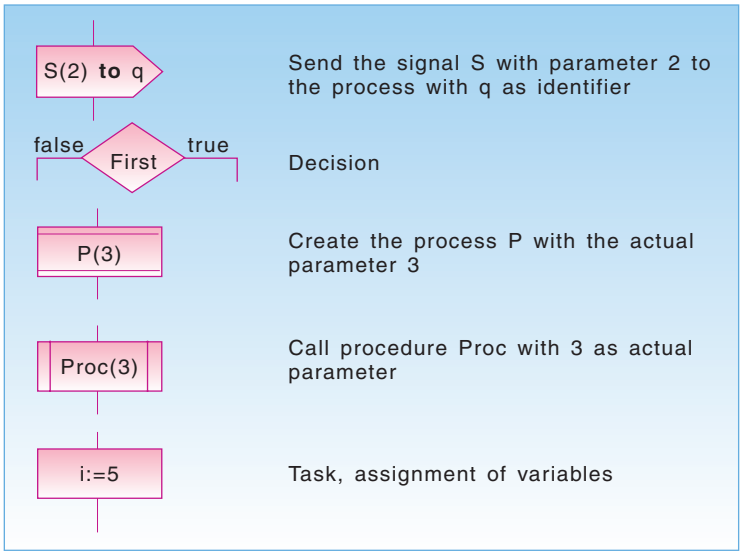


Figure 5 Actions in transitions

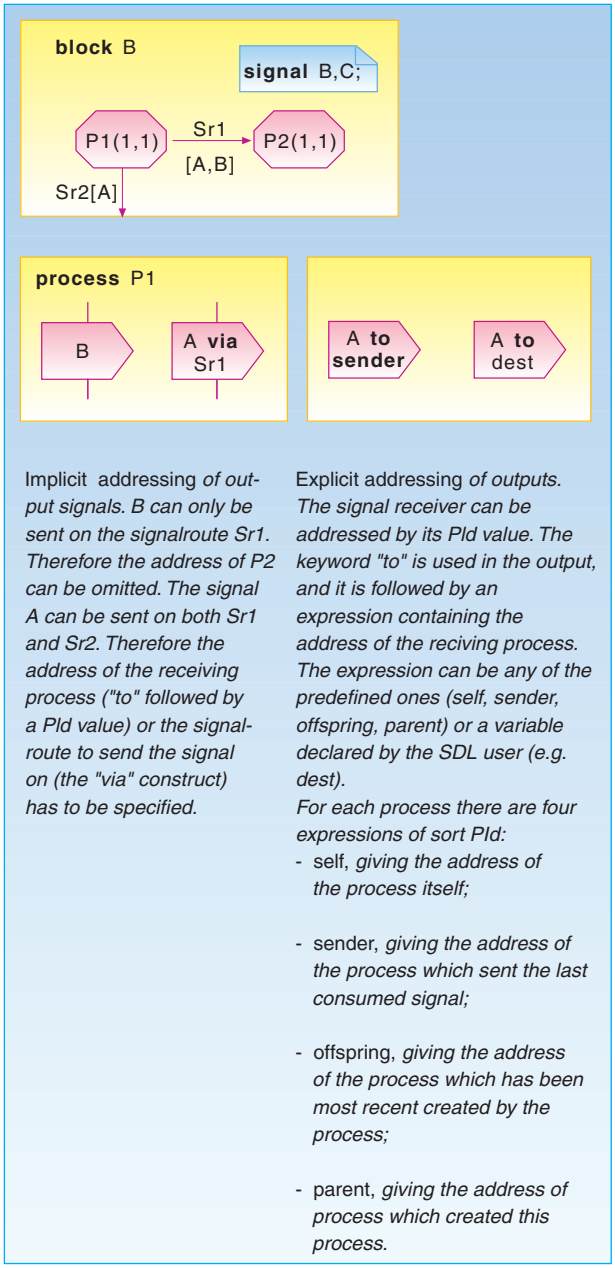


Figure 6 Specification of communication and addressing of signals

- *Implicitly,* by omitting the address. Then the system structure and its communication paths are used to decide which process will receive the signal. It is possible to state explicitly on which signal routes to send the signal.

Signal routes may be uni- or bidirectional, just like the channels. Each direction has an attached signallist, which is a list of the names of the signals that can be carried on the signal route in that direction.

Figure 6 shows an example of how two processes can communicate sending signals on signalroutes by using different addressing mechanisms.

4 Additional concepts

For most specifications, the concepts introduced above are sufficient. The system consists of blocks connected by channels, the blocks consist of processes connected with signal routes, and processes are specified by a process graph.

SDL has a number of predefined data types (in SDL called *sorts)* and constructs like *struct* and *array* which for most users are sufficient to declare the needed variables.

Here we will briefly mention additional constructs which for the advanced SDL user may add to the language power for specifying large and complex systems. So far we have only seen three levels of specifications, the system level, the block level and the process level. In addition, it

```
newtype Number
    literals 0,1;
    operators
        "+": Number,Number -> Number;
    axioms
        for all a,b,c in Number (
            a+0 == a;
            a+b == b+a;
            (a+b)+c == a+(b+c);)
endnewtype;
```

Figure 7 Example data type specification
The declared newtype is Number. It is defined to be composed of two literals, 0 and 1. The operator '+' allows a Number to be generated from other numbers by repeated applications of the operator. The axioms specify which terms represent the same value. For instance, 1 + 0 == 1, i.e. 1 + 0 and 1 represent the same value

is possible to define a *block substructure,* i.e. blocks within blocks in an unrestricted number of levels. Processes can be partitioned into *services,* each service being a state machine itself and representing a partial behaviour of the process.

SDL has a large number of so-called *shorthands* allowing users to express a specific behaviour in a compact way. All these constructs can be expressed using more primitive constructs. An example of such a construct is *export,* which allows specification of exported variables, i.e. variables owned by one process can be made visible to another process. The export construct can be expressed using signal exchange where the exporter sends the value of the variable as a signal parameter to the importing process. Another shorthand is the *asterisk* concept, e.g. asterisk input. An asterisk input is an input symbol with an asterisk as signal name. This represents a rest list of signals, for which no transition is specified elsewhere for the state in question. It is then possible to state that the signals not explicitly mentioned will be treated the same way, e.g. for specifying error handling. For the rest of the shorthands, we refer to textbooks (2, 3).

5 Data definition

So far we have only seen declaration and use of data. Variables in processes and procedures are in SDL declared to be of specific types. These variables are used in expressions in the tasks. We will now briefly present how to define data in SDL.

Data in SDL is based on the notion of *abstract data types* (ADTs). This is probably the most powerful part of the language. Despite this fact, it is not much used. This is because ADTs are regarded as difficult both to read and write by non experts. When we say that ADTs are not much used, we refer to user defined types. The language has a number of useful predefined data types (integer, real, boolean, etc.) and constructs like structs and arrays which are sufficient for many users.

A data type is called a sort in SDL. The specification of a sort has the following components:

- a *name* of the sort
- a set of *literals* (value names)
- a set of *operators* on the values
- a set of *axioms* (or equations) defining the operators.

Figure 7 gives an example of a data type.

The literals, which can be considered as operators without parameters, define a subset of the values of a sort. In general, a sort may have many more values. All the values of a sort can be generated by the repeated application of the operators to the values of the sort. A combination of applications of operators is called a *term.* The axioms are used to define which terms represent the same value.

6 The future of SDL

We have now given an overview of SDL, its structuring mechanisms, the behaviour part, the communication aspects and the abstract data types. Hopefully, we have communicated the basic ideas and what they can express.

In addition to the constructs mentioned here, there is an overwhelming number of other concepts, some adding to the language's power, others to its flexibility and complexity.

SDL is still a language in development. In the years 1988 to 1992 many new features have been added. These include among others object oriented concepts, remote procedures, algorithmic data definition and a library concept. All these new features have even further added to the complexity of the language.

In the future, simplification of SDL will be essential. Work is initiated in CCITT SG 10 to harmonise the concepts and hopefully reduce the size of the language.

Another important work item is how to combine the use of SDL and ASN.1 (7). ASN.1 is much used to specify and standardise communication protocols. Those who are implementing such protocols do not want to specify already defined ASN.1 types using SDL.

References

1 CCITT. *CCITT Specification and Description Language.* Geneva, 1988. (Blue Book, Fascicle X.1-5, Recommendation Z.100.)

2 Belina, F et al. *SDL with Applications from protocol specification.* Englewood Cliffs, N.J., Prentice Hall, 1991. ISBN 0-13-785890-6.

3 Saracco, R et al. *Telecommunications Systems Engineering using SDL.* Amsterdam, North-Holland, 1989. ISBN 0-444-88084-4.

4 Møller-Pedersen, B. SDL-92 as an object oriented notation, *Telektronikk,* 89(2/3), 71-83, 1993 (this issue).

5 Møller-Pedersen, B et al. *SDL-92 in the description of Intelligent Network services.* Kjeller, Norwegian Telecom Research, 1992. (TF report R44/92.)

6 Bechmann, T et al. *Katalogsystemet KAT-500: Spesifikasjon av katalogklienten DUA(PC) ved bruk av spesifikasjonsspraket SDL.* Kjeller, Norwegian Telecom Research, 1991 (TF report R37/91.)

7 CCITT. *Recommendation X.208, Information technology – Open Systems Interconnection – Abstract Syntax Notation One (ASN.1),* March 1988. (Blue Book, Fascicle VIII.4.)

SDL-92 as an object oriented notation

BY BIRGER MØLLER-PEDERSEN

1 Introduction

The last years have seen a lot of object oriented analysis and specification notations emerge. They have emerged from converted Structured Analysis advocates through many years, popped up as extensions of ER notations, come about as projections of object oriented programming practice into analysis, or simply sold as object oriented notations because the keywords **class** and **object** are used.

Many of these notations are simple-minded, with simple-minded objects that are nothing but a collection of data attributes with associated operations, with methods and message passing that is nothing more than a remote procedure call, and few of them are real languages with real semantics, with the implication that deriving implementations from them is more or less to do it all over again.

Behind the big scenes, in real-life projects (and big projects), the community of SDL users tend to see their own application of SDL as nothing to talk about. They are just making very complex systems consisting of real-time, interacting, concurrent processes, that really send messages.

In March 1992 the CCITT plenary issued the 1992 version of the CCITT recommended language SDL for specification and description of systems, and now SDL92 is (yet another) member of the set of object oriented notations. The difference from most other notations is that it is not just fancy graphics (in fact some of the most fancy are left to tools to provide), but there is a real language behind, with a formal definition. As a standardised language, it also has its weak points, but most of what will be expected by an object oriented notation (and some more) is supported.

This contribution aims at introducing SDL92 as an object oriented notation, by using the same kind of gentle, informal introduction to the concepts that is often seen in papers on object oriented notations, by providing a simple example and by giving some elements of a method as the introduction goes along.

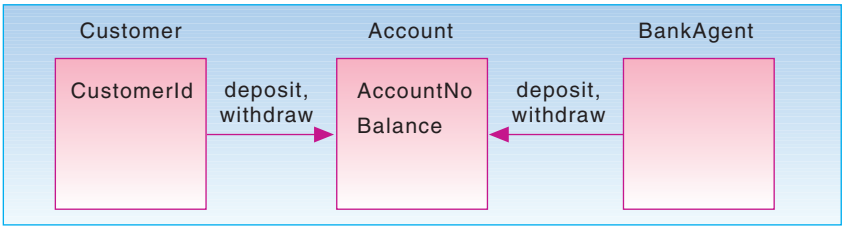


Figure 1 Elements of an example Bank System
The Bank System consists of the following components: Account, Customers, and BankAgents. The BankAgent requesting operations on an Account is supposed to model any automatic transaction on the Account, e.g. deposit of salary and withdrawals in order to pay interest and repayment on loans in the bank. This is an informal sketch of some of the attributes and of interactions between these objects; this would be the kind of specification supported by many object oriented analysis notations

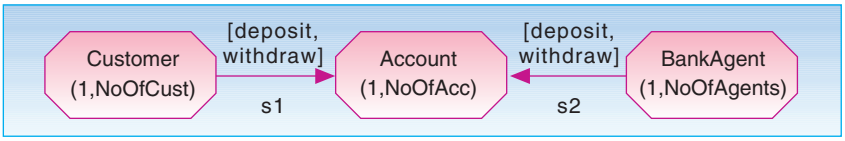


Figure 2 Process Sets representing the elements of the bank system
The parentheses are stating the initial and the maximum number of process instances in the sets. If these numbers are not known (or not interesting at this point), then they are not specified. The implication will be that there will be initially 1 element, and that there is no limit on the number of instances. The Customer and BankAgent processes may concurrently perform deposits and withdrawals on the same account. The signals on the signal routes (s1 and s2) indicate possible interaction by signal exchange. The deposit and withdraw signals will carry the amount to deposit and withdraw, respectively

2 SDL systems and SDL systems specifications

SDL is a language intended for use in analysis and specification of systems. This is done by making *SDL systems* that are models of the real world systems. The real world systems can either be existing or be planned to come into existence. SDL systems are specified in *SDL system specifications.*

The identified components of the real world systems are represented by *instances* as part of the SDL systems. The classification of components into categories and subcategories are represented by *types* and *subtypes* of instances.

As part of the analysis and specification, sets of application specific concepts will often be identified. These are in SDL represented by *packages of type definitions.*

SDL is a language which may be used in different ways – there is no specific method of identifying components and their properties, or for identifying categories of components. In this respect, SDL is different from most analysis notations that often come with a method. In the following overview no attempt is made to justify the choice of components and their properties in the example system – the intent is to introduce the main elements of the language, and not a method. Figure 1 gives an informal sketch of the example system being used here.

3 Processes, the main objects of SDL Systems

In different notations there are different approaches to what the main elements are. In extended Entity-Relationship formalisms they are sets of entities, in scenario based notations (or in role models) they are roles, and in object oriented notations they are either objects, classes of objects or mixtures of these. Often there is no hint on how the behaviour of these elements are organised, if they have behaviour at all.

An SDL system consists of a set of *instances.* Instances may be of different *kinds,* and their properties may either be directly defined or they may be defined by means of a type. If it is important to express that a system has only one instance with a given set of properties, then the instance is specified directly, without introducing any type in addition. If the system has several instances with

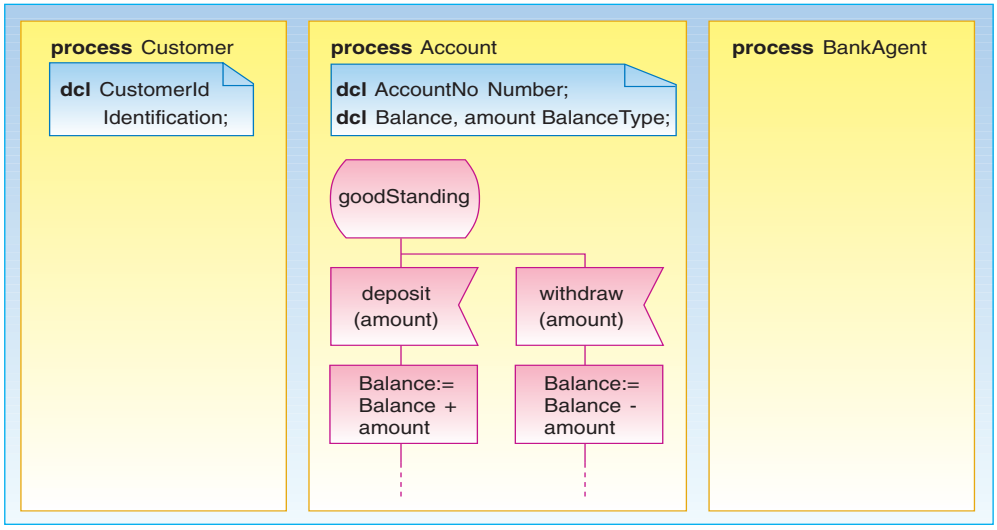


Figure 3 Process Diagrams specifying the properties of processes
Only the Account is (partially) specified: two variables (of types Number and BalanceType) and a fragment of the process behaviour is specified: in the state goodStanding it will accept both deposit and withdraw signals, and the Balance will be updated accordingly

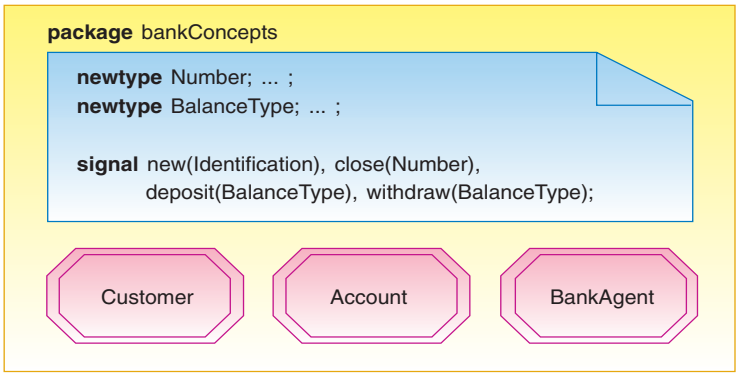


Figure 4 Package of Type Definitions
The Process types Customer, Account, and BankAgent are specified as part of a package, together with the types of attributes and signals used in bank systems. The process type symbols in the package diagram indicate only that three process types are defined as part of the bankConcepts package. The symbols are merely diagram refrences to process type diagrams defining the properties of the process types

the same set of properties, then a type is defined, and instances are created according to this.

The main kind of instances in SDL systems are processes. A *process* is characterised by *attributes* in terms of *variables* and *procedures,* and by a certain *behaviour.* Processes behave concurrently and interact by means of asynchronous signal exchange. The behaviour of a process is represented by an Extended Finite State Machine. Attributes of processes can be *variables* and *procedures.* Procedure attributes are exported so that other processes can request them to be performed. Variables are of data types defining values with associated operators or of Process Identifier types defining references to processes.

In a model of a bank system, where it is important to model that customers and bank agents may update an account concurrently, processes are obvious candidates for the representation of these components of the model.

Processes are elements of *process sets.* When the components of the system have been identified, and they should be represented by processes in the model SDL system, then they are contained in process sets. The notation for process sets is illustrated in Figure 2. The specification of a process set includes the specification of the *initial* number and the *maximum* number of instances in the set, i.e. cardinality constraints.

The simplest form of interaction between processes is exchange of *signals.* In order to specify that processes may exchange signals, the process set symbols are connected by *signal routes.* The signal routes between the process sets and the signals associated with the signal routes specify possible signal exchanges. Other notations like Message Sequence Charts are needed to illustrate the sequences of signals sent between two or more processes.

Signal routes indicate a combination of control and data flow. Processes execute independently of each other, but the flow of control of a process may in a given state be influenced by the reception of signals sent from other processes. Signals may carry data values which the receiving process may use in its further execution.

The notation for specifying process sets is used to indicate that the system will have sets of process instances; the properties of the processes are described in separate *process diagrams,* see Figure 3 for examples.

4 Classification of processes: Process types

Often there will be different sets of objects with the same properties in a system (e.g. a set of good and a set of bad customers), or the same category of objects should be part of different systems. In object oriented notations this is reflected by defining classes of objects. In SDL it is reflected by defining *process types.* These may either be specified as part of the specification of a single SDL system, or as part of the specification of a *package.*

Most object oriented methods will have an activity that from a dictionary of terms within an application domain, (or by some kind of analysis, or by experience from other similar systems), identifies application specific concepts. In SDL, these concepts are represented by types of various kinds of instances, and for a given application domain these may be collected into packages. Figure 4 gives an example on how a package of bank system specific types is specified. The advantage of the package mechanism is that it allows the definition of collections of related types without introducing arti-

ficial types just for this purpose. Types in SDL are intended for the representation of common properties of *instances* of systems, and collections of types are not instances.

The properties of process types are specified by means of *process type diagrams.* Process type diagrams are similar to process diagrams: they specify the attributes and behaviour of types of processes. The only differences are the extra keyword **type** and the *gates* at the border of the diagram frame, see Figure 5 for examples. Different process sets of the same process type may be connected by signal routes to different other sets. For this purpose, the process type defines gates as connection points for signal routes. The constraints on the gates (in terms of ingoing and outgoing signals) make it possible to specify the behaviour of process types without knowing in which process set the instances of the type will be and how the process sets are connected to other sets. Gates can only be connected by signal routes which carry the signals of the constraint.

Process sets may be specified according to process types, as illustrated in Figure 6. The process set specification will specify the name of the set and the process type.

Note the distinction between process *types,* process *sets* and process *instances.* Process types only define the common properties of instances (of the type), while process sets have cardinality. Variables of Process Identifier type identify process instances, and not sets. Signal routes connect process sets.

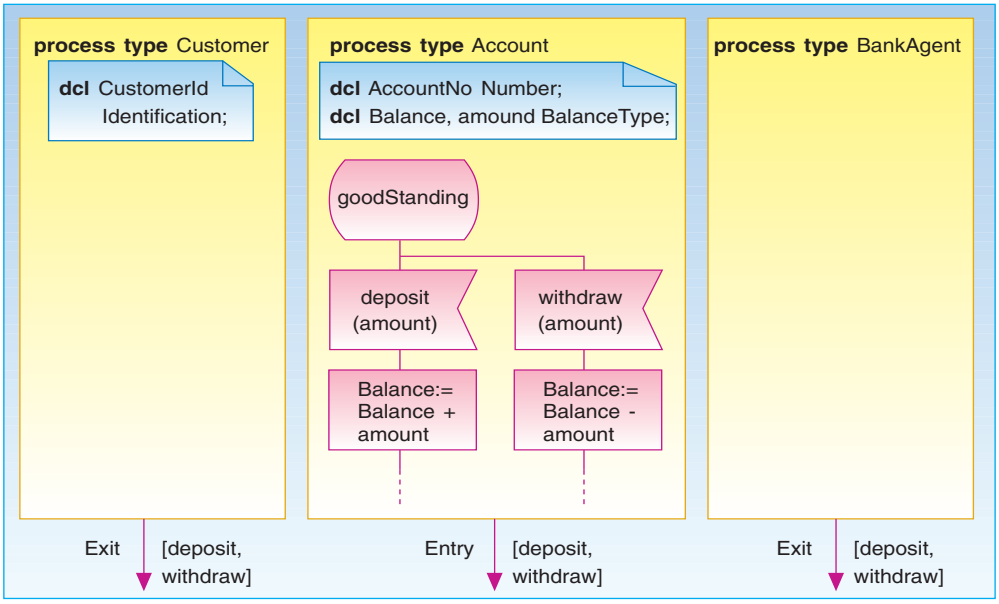


Figure 5 Process Type Diagram
The keyword **type** *indicates that these are type diagrams. Entry and Exit at the border of the diagram are gates. Gates are connection points for signal routes. In Figure 6 these gates are connected by signal routes*

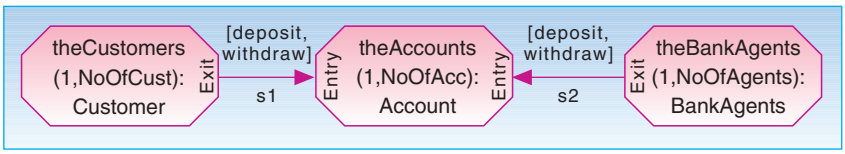


Figure 6 Process sets according to Process Types
In the specification of a process set according to a type there is both a process set name (e.g. theBankAgents) and a type name (BankAgent). The gates are the ones defined in corresponding process type diagrams

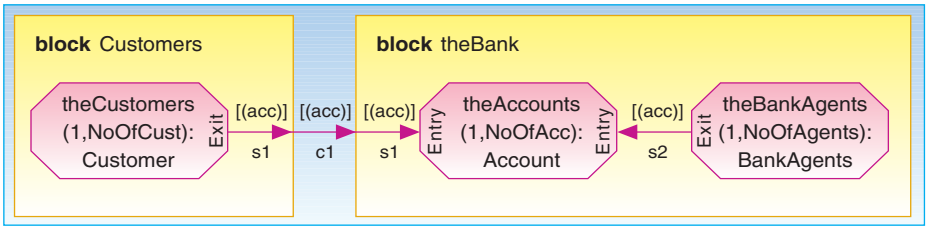


Figure 7 Blocks of Processes
By using the block concept, it is indicated that Account and BankAgent processes are part of the Bank, while Customer processes are not

5 Grouping of objects: Blocks of processes

Unlike most notations for object oriented analysis and specification, SDL does not only support the specification of classes and objects (by means of process types and processes) with their interaction, but also the grouping of objects into larger units. This may either be used in order to decompose a large system into large functional units (if functional decomposition is the preferred technique) or it may be used to structure the SDL system into parts that reflect parts of the real world system.

Blocks in SDL are containers for either sets of processes connected by signal routes, or for a substructure of blocks connected by *channels.* These blocks may in turn consist of either process sets or a substructure of blocks. Figures 7 and 8 give examples of both alternatives. There is no specific behaviour associated with a block, and blocks cannot have attributes in terms of variables or exported procedures. Therefore, the behaviour of a block is only the combined behaviour of its processes.

In addition to containing processes or blocks, a block may have data type definitions and signal definitions. Signals being used in the interaction between processes in a block may therefore be defined locally to this block (providing a local name space).

Signals sent on channels between blocks will be split among the processes (or blocks) inside the outer blocks. The connected channels and signal routes give the allowed signal paths. Blocks also provide encapsulation, not only for signal definitions as mentioned above, but also for process instance creation. Apart from the initially created processes, all other processes have to be created by some pro-

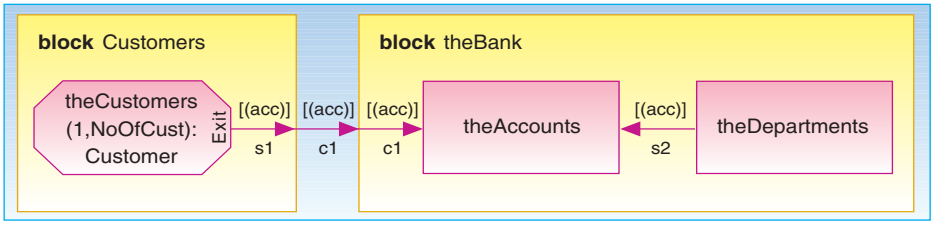


Figure 8 Block with Subblocks
theBank block is further decomposed into two blocks. theDepartments may e.g. contain BankAgents, while the theAccounts may contain the Account processes (not shown here)

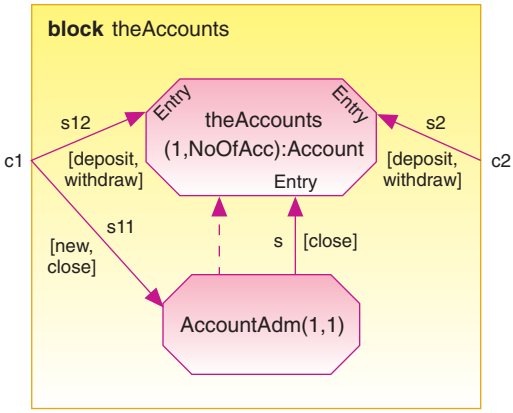


Figure 9 The Interior of theAccounts block
Accounts will not (and cannot) be created directly by Customers (outside the block), but by a special process, AccountAdm, which will assign a unique AccountNumber to the Account. This process is part of the block containing the Accounts. Note that the incoming signals are divided into (new,close) that are directed to AccountAdm, and (deposit,withdraw) which are directed to theAccounts. The dashed line from AccountAdm to theAccount indicates that AccountAdm creates processes in theAccount process set

cess (by the execution of a create request), and this may only take place within a block. The implication of this is that creation of processes in a block from processes outside this block must be modelled by sending special signals for this purpose. See Figure 9 for an illustration of both the splitting of channels and remote process creation.

6 Specification of systems: Set of blocks connected by channels

Having identified the right set of blocks and processes and their connections, the whole truth about a system is expressed in a *system specification.* The system specification delimits the system from the environment of the system. An SDL system consists of a set of blocks connected by channels. Blocks may also be connected with the environment by means of channels. Figure 10 gives an example on a system diagram.

If the system interacts with its environment, then the signals used for that purpose is defined as part of the system. The SDL system assumes that the environment has SDL processes which may receive signals from the system and send signals to the system. However, these processes are not specified.

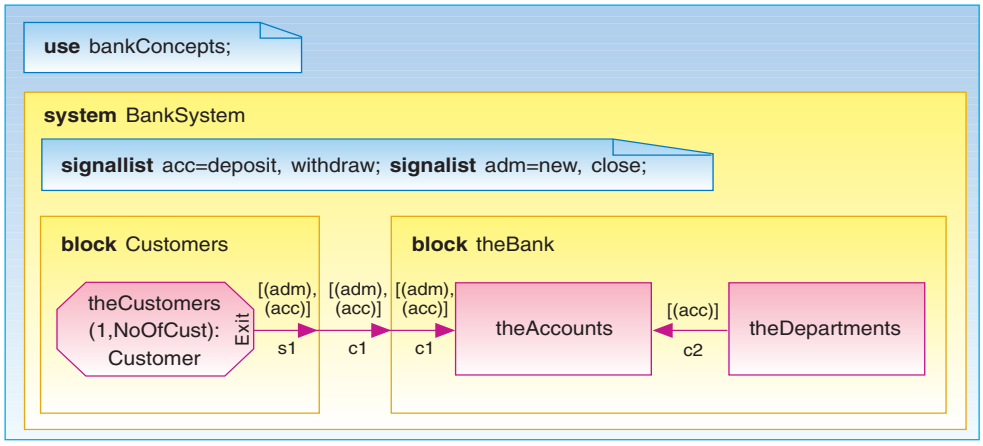


Figure 10 System Diagram
A complete specification of a Bank system. The specification uses the types defined in the package bankConcepts. By incorporating the customers in the system, there is no interaction with the environment. If not incorporated, the c1 channel would just connect theBank block with the frame of the diagram, indicating interaction with the customer processes that are then supposed to be in the environment

7 Specification of properties of attributes: Data types

Data types of variables define values in terms of literals, operators with a signature, and the behaviour of the operators.

In the example used here, only simple structured data types are used:

```
newtype BalanceType
   struct
      dollars
Integer;
      cents
Integer;
endnewtype BalanceType;
newtype Number
   struct
      countrycode
Integer;
      bankcode
Integer;
      customercode
Integer;
endnewtype Number
```

The operators may either be specified by means of axioms or by means of operator diagrams that resemble procedure diagrams.

8 Specification of behaviour: States and transitions

With respect to behaviour, a process is an Extended Finite State Machine: When started, a process executes its *start transition* and enters the first *state.* The reception of a *signal* triggers a *transition* from one *state* to a *next state*. In transitions, a process may execute *actions.* Actions can assign values (of expressions) to variables of the processes, branch on values of expressions, call procedures, create new processes, and output signals to other processes. Figure 11 is a behaviour specification of a process type.

Each *process* receives (and keeps) signals in a *queue.* When being in a state, the process takes from the queue the first signal that is of one of the types indicated in the input symbols. Other signals in the queue will be discarded, unless explicitly *saved* in that state. Depending on which signal has arrived first, the corresponding transition is executed and the next state is entered.

9 Attributes of potential behaviour: (Remote) procedures

The behaviour of a process or process type may be structured by means of *partial state/transition diagrams* represented by *procedures.* A procedure defines behaviour by means of the same elements as a process, i.e. states and transitions, and a process follows the pattern of behaviour by calling the procedure.

Processes may export a procedure so that other processes can request it (by remote procedure calls) to be executed by the process which exports it, see Figure 12 for an example.

In simple cases, all exported procedures are accepted in all states, but in general it is possible to specify that a procedure will not be accepted in certain states, just as for signals.

While process communication by means of signals is asynchronous, the process calling a remote procedure is synchronised with the called party until the procedure has been executed. When the called party executes the transition associated with the procedure input, the caller may proceed on its own.

10 Classification of blocks: Block types and specialisation of these by adding attributes

Types are used in order to model concepts from the application domain and to classify similar instances. The use of types when defining new instances or types has been illustrated above. The notions of generalised and specialised concepts are in SDL represented by (super)types and subtypes.

Blocks may be used either to group processes in different *functional units* or they may be used to model directly "physical" entities in the system being described. In the BankSystem the blocks are used to model that the customers are not part of the bank: there is a block containing customers and a block that represents the bank.

Suppose that the bank block should model the whole of a given bank, and that it is important to model that it consists of headquarters and a number of branches, see Figure 13. Each branch will have departments and accounts, the same will the headquarters. A branch will *in addition* have a bank manager, while the headquarters *in addition* will have a board.

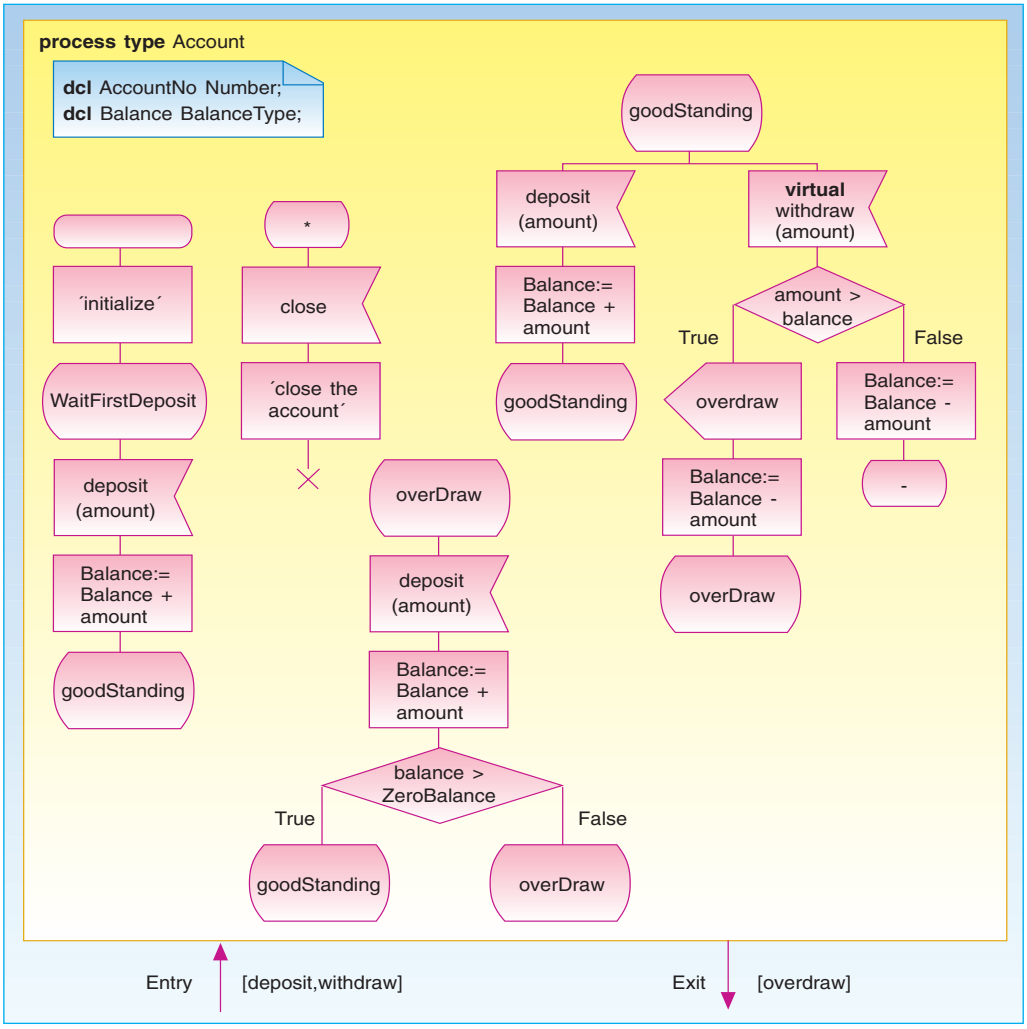


Figure 11 Specification of Behaviour (of Account processes) by States and Transitions
Compared to former sketches of the theAccount process type, this one also outputs the signal overdraw. In all states it will accept the signal close (means all states). In state goodStanding it will accept both deposit and withdraw, while in state overDrawn it will only accept deposit. As Account is specified here, withdraws in state overDrawn will be discarded (signals of type withdraw is not input in state overDrawn); if they were saved, then they could have been handled in other states. The input transition withdraw in state goodStanding is specified to be a virtual transition. This implies that the transition may be redefined in subtypes of Account. Non-virtual transition cannot be* redefined *in subtypes*

This classification is in SDL supported by *block types* and by *specialisation of block types.* Figure 14 indicates the subtype relation, while Figure 15 gives the details of the block subtype diagrams.

In general a (sub)type of any kind of instance (and not only blocks) can be defined as a *specialisation* of another (super)type. A subtype *inherits* all the properties defined in the supertype definition. In addition it may *add properties* and it may *redefine virtual types and virtual transitions.* Added properties must not define entities with the same name as defined in the supertype.

11 Specialisation of behaviour I: Adding attributes, states and transitions

When classifying processes into process types, the need for organising the identified process types in a subtype hierarchy will arise. Figure 16 illustrates a typical process type hierarchy in the bank account context. The behaviour of the

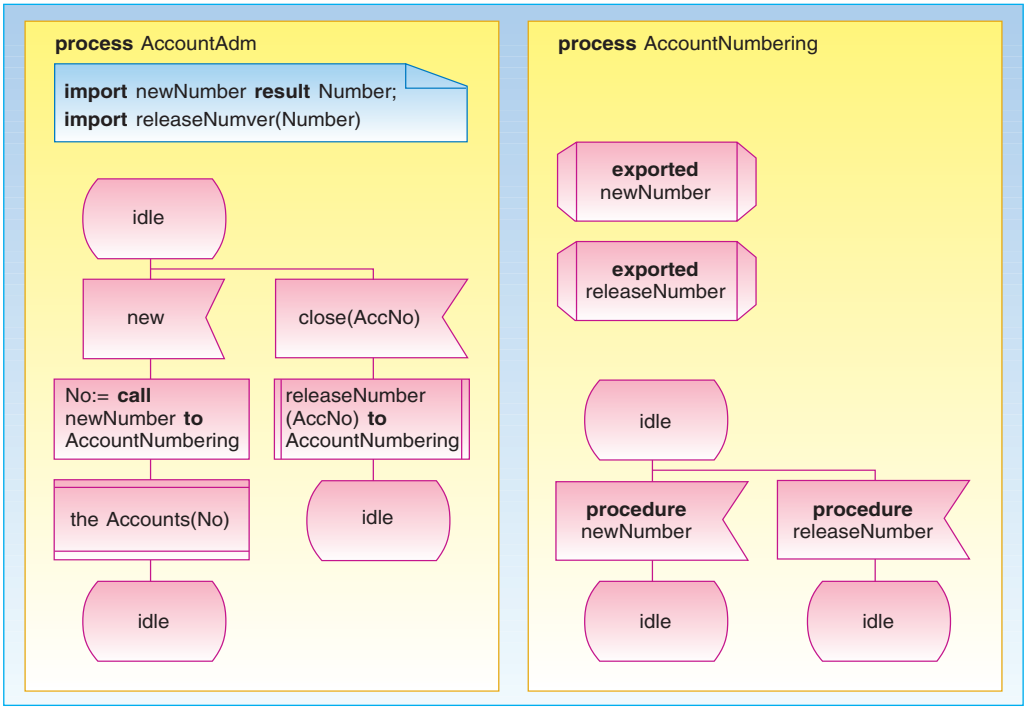


Figure 12 Remote Procedures used to provide a new, unique account number, and to release the use of a number
The procedure newNumber is a value returning procedure. *This fact is used in the call of the procedure. The number obtained from the procedure (No) is given to the new process created in the process set theAccounts. The releaseNumber procedure is just called, with the number to be released as a parameter*

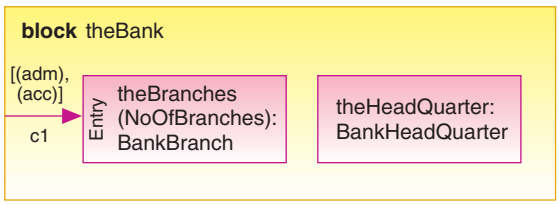


Figure 13 Blocks according to Block Types
A bank with headquarters and a set of branches

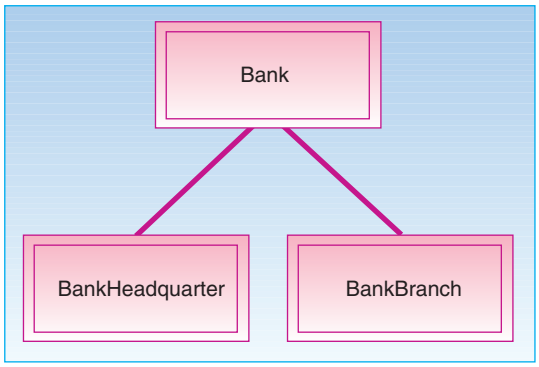


Figure 14 Illustration of the Block Subtype Hierarchy
Block subtype relation, with the Bank representing the common properties of BankHeadquarters and BankBranch. This is not formal SDL, but just an illustration

subtypes are supposed to inherit the behaviour of the process supertype.

As for a block subtype, a process subtype *inherits* all the properties defined in the process supertype, and it may *add properties.* Not only attributes, like variables and procedures, are inherited, but the behaviour specification of the supertype is inherited as well.

Transitions specified in a subtype are simply added to the transitions specified in the supertype: the resulting (sub)type will thereby inherit all the transitions of the supertype, see Figure 17 for an example.

12 Specialisation of behaviour II: Redefining virtual transitions

A general type intended to act as a supertype will often have some properties that should be defined differently in different subtypes, while other properties should remain the same for all subtypes. SDL supports the possibility of redefining local types and transitions. Types and transitions which can be redefined in subtypes are called *virtual types* and *virtual transitions.* The behaviour of an instance of a subtype will follow the redefinitions of virtual types and transitions, also for the part of the behaviour specified in the supertype definition. As an example, calls of a virtual procedure will be calls of the redefined procedure, also for calls being specified in the supertype.

The notion of virtual procedures that may be redefined in subclasses is a well-known object oriented language mechanism for specialising behaviour. A more direct way of specialising behaviour is provided in SDL by means of virtual transitions. Figure 18 illustrates the redefinition of a virtual transition.

In addition to virtual input transitions, it is also possible to specify the start transition to be virtual; and as for a virtual input transition it may be redefined in subtypes. A virtual save can be redefined to an input transition.

13 Specialisation of behaviour III: Redefining virtual procedures and types

If it is not intended that a whole transition shall be redefinable, then *virtual procedures* is used. Figures 19 and 20 give an example of the definition and redefinition of virtual procedures.

In order for the supertype with the virtual procedures to ensure that redefinitions are not redefinitions to just any procedures with the same name, virtual procedures may be *constrained* by a procedure. Redefinitions then have to be specialisations of this constraint procedure. The specialisation of procedures follows the same pattern as for process types: adding variables, states and transitions, and redefining virtual transitions and virtual procedures. The constraint procedure may be so simple that it only specifies the required parameters, but it may also specify behaviour. This behaviour is then assured to be part of the redefinitions.

Types in general (and not only procedures) may be defined as virtual types. In addition a virtual type may be given a *constraint,* in terms of a type. This implies that the redefinition cannot be just any type, but it has to be a subtype of the constraint.

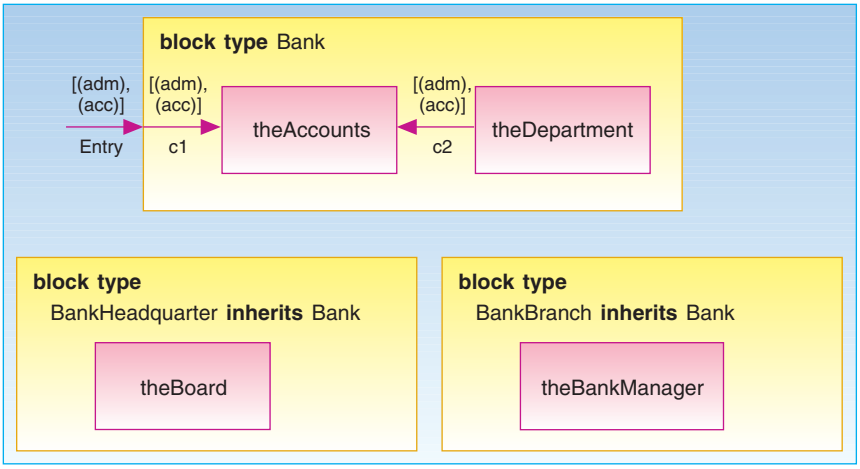


Figure 15 Block Types inheriting from a common Block Type
The common properties of a bank are modelled by a block type Bank. The block type Headquarters is a subtype *of this, inheriting the properties of Bank and* adding *a Board, while BankBranch is a* subtype *of Bank,* adding *the BankManager*

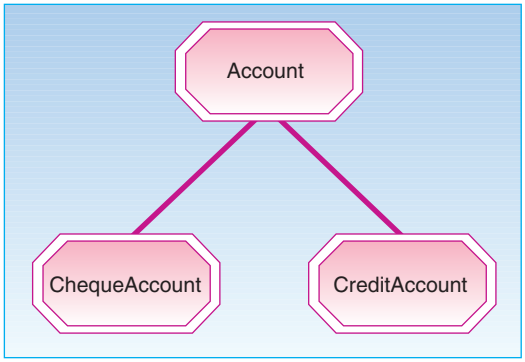


Figure 16 Illustration of Subtype Hierarchy of Process Types

The analysis of accounts in the bank system may have concluded that there are two special types of accounts: cheque accounts, which will allow withdrawals based on cheques, and credit accounts, which have a limit on overdraw. Specifying the corresponding process types as subtypes of Account ensures that they will have all the properties of Account. The figure is not formal SDL, but just an illustration

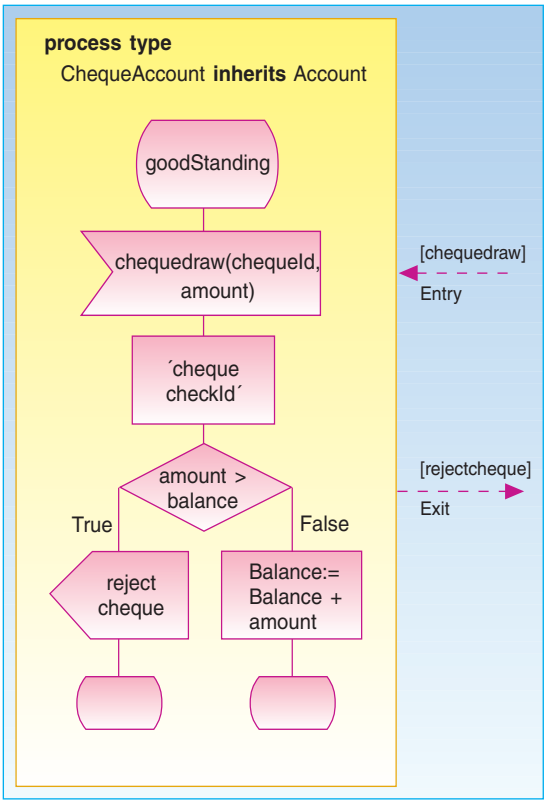


Figure 17 Adding Transitions in a Subtype
The process type ChequeAccount inherits the properties of process type Account, and adds the input of a new type of signal (chequedraw) in state goodStanding and a corresponding transition. The fact that the gates Entry and Exit are inherited (and not added) is indicated by the gate symbols being dashed

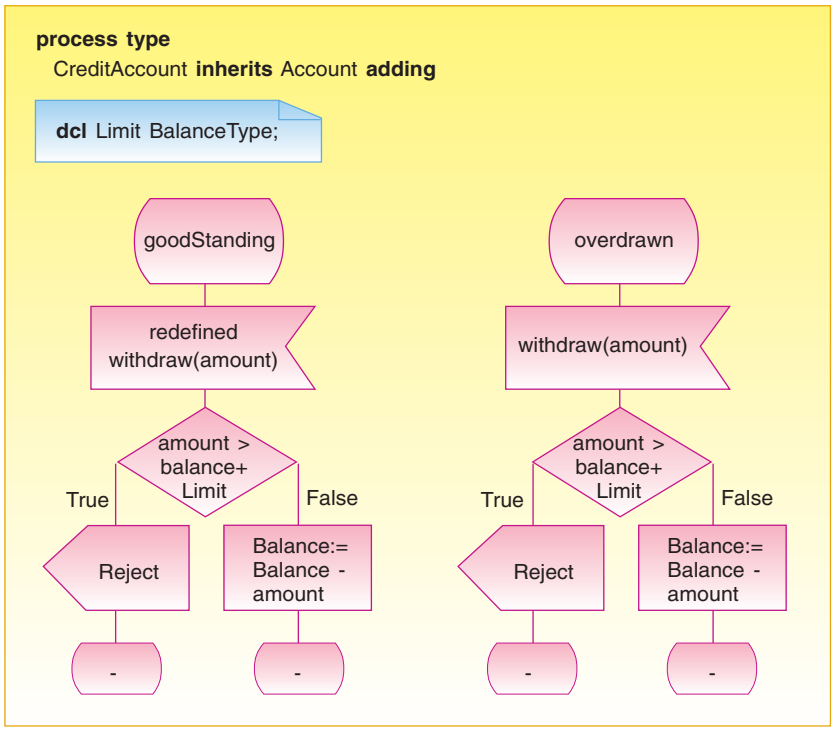


Figure 18 Redefining a virtual Transition in a Subtype
The virtual transition withdraw in state goodStanding is redefined in the subtype CreditAccount, in order to express that for a credit account a withdraw is allowed until a certain Limit amount in addition to the balance. If not redefined, then withdraw in state goodStanding would have the effect as specified in the supertype Account, i.e. just testing if amount is greater than balance. By adding a withdraw transition in state overdrawn it is also specified that for a credit account it is also possible to withdraw from an overdrawn account. (Remember that Account simply specifies that withdraw in state overDrawn will be discarded)

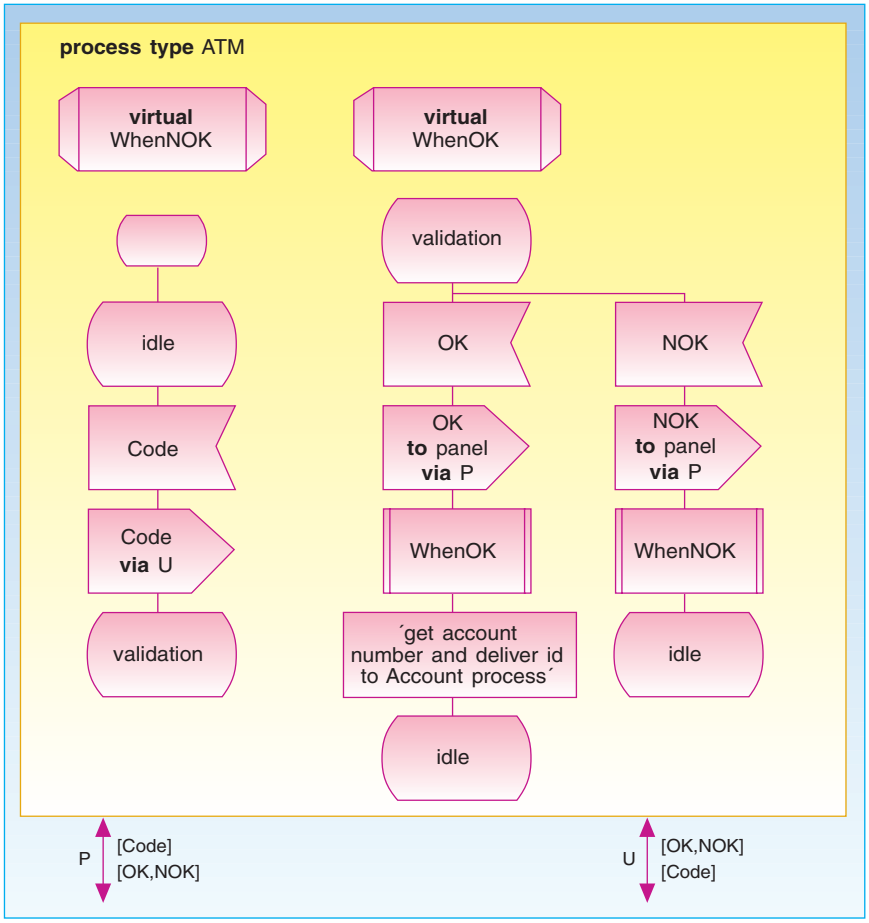


Figure 19 Process type with virtual Procedures
A partial ATM (Automatic Teller Machine) type is specified. It is supposed to model an automatic teller machine being used in order to access accounts for withdrawal of cash. In the complete bank system it will be in between customers and the accounts. Based upon a code and the account number, the machine will grant access to the account – in SDL terms this access is provided by sending to the Customer process a Process Identifier denoting the actual Account process. The parts of the ATM that control the user interface of the machine, in case the code is either OK or not OK, are represented by two virtual procedures. These may then be redefined in different subtypes of ATM representing different ways of presenting this information to the customer. The behaviour of ATM that has to be the same for all subtypes is specified as non-virtuals

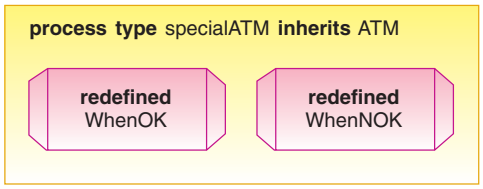


Figure 20 Polite ATM with redefined Procedures
A subtype of ATM is specified. The redefined procedures will have separate procedure diagrams giving the properties of the redefinitions

14 From partial functionality to complete processes: Composition of services

Instead of specifying the complete behaviour of a process type, it is possible to define partial behaviour by means of *service types*. A process type can then be defined to be a composition of service instances according to these service types. In addition to services, the combined process can have variables. Services do not execute concurrently with other services, but alternate between them – uniquely determined by the incoming signal or by signals sent from one service to another. Services share the variables of the enclosing process and may directly access these.

In some approaches it is recommended to identify different scenarios or different roles of a phenomenon in different contexts, and then later combine these. By considering a little more comprehensive account concept than shown above, two different scenarios may be identified: one covering the interaction between an account and a customer (with the signals deposit and withdraw) and one covering the interaction with an administrator (e.g. with the signals status and close).

In order to completely specify the behaviour of the different roles by service types, the attributes of the combined role (represented by a process) have to be manipulated by the services. In this case both roles have to manipulate e.g. the Balance attribute. From a methodological point of view, however, it should be possible to define the service types independently of where they will be used to define the combined process type. It should even be possible to use the same service types in the composition of different process types.

SDL provides the notion of *parameterised type* for this purpose. A parameterised type is (partially) independent of where it is defined, and this is expressed by *context parameters*. When the parameterised type is used in different contexts, *actual context parameters* are provided in terms of identifiers to definitions in the *actual enclosing context*.

In this case, the independency of the Balance is expressed by a variable context parameter, see Figures 21 and 22, and in the combined process type, the actual context parameter is the variable Balance in the enclosing process type, see Figure 23.

The only requirement for the aggregation of services into processes is that the input signal sets of the services are disjoint.

As for block and process types, service types may also be organised in subtype hierarchies.

15 Keeping diagrams together: Diagram references and scope/visibility rules

An SDL system specification consists of a system specification with enclosed specifications of block types and blocks, these will again consist of specifications of process types and process sets, etc. This *nesting* of specifications is the basis for normal *scope-rules* and *visibility rules* known from block structured languages.

In principle, the corresponding diagrams could be nested in order to provide this hierarchy of specifications within specifications. For practical purposes (as e.g. paper size) it is, however, possible to represent nested diagrams by so-called *references* (in the enclosing diagram) to *referenced diagrams.* As an example a block type reference in a system diagram reference a block type diagram that is nested in the system diagram. Signals defined in the system diagram are thereby visible in the block type diagram. Figure 24 illustrates the principle.

A diagram may be split into a number of *pages.* In that case each page is numbered in the rightmost upper corner of the frame symbol. The page numbering consists of the page number followed by the total number of pages enclosed by (), e.g. 1 (4), 2 (4), 3 (4), 4 (4).

16 SDL-92 in perspective

A recent survey of notations for object oriented analysis, specification, and design concludes that most of them lacks mechanisms to support the following cases:

- The number of objects and classes are large, and they have a natural grouping in some kind of units. These units are not objects as supported by the notations, and if they should be objects, they would be objects of a special kind that may enclose other objects in a different way than part-objects.

  Most notations are very good at describing the attributes of objects or classes in isolation, and the relations between a limited set of these.

  The notion of blocks in SDL provides a consistent grouping of processes.

- When most of the objects in a given system relies on the interaction with a common *resource* object, then graphical notations for expressing this will have problems.

  SDL provides identifiers that may identify common processes by means of names. Nesting of definitions, and scope-rules provides convenient identification of common definitions.

In general terms, these problems stem from the fact that there is no language behind most of these notations. SDL is a complete (even formal) language, where the graphical syntax is just one concrete

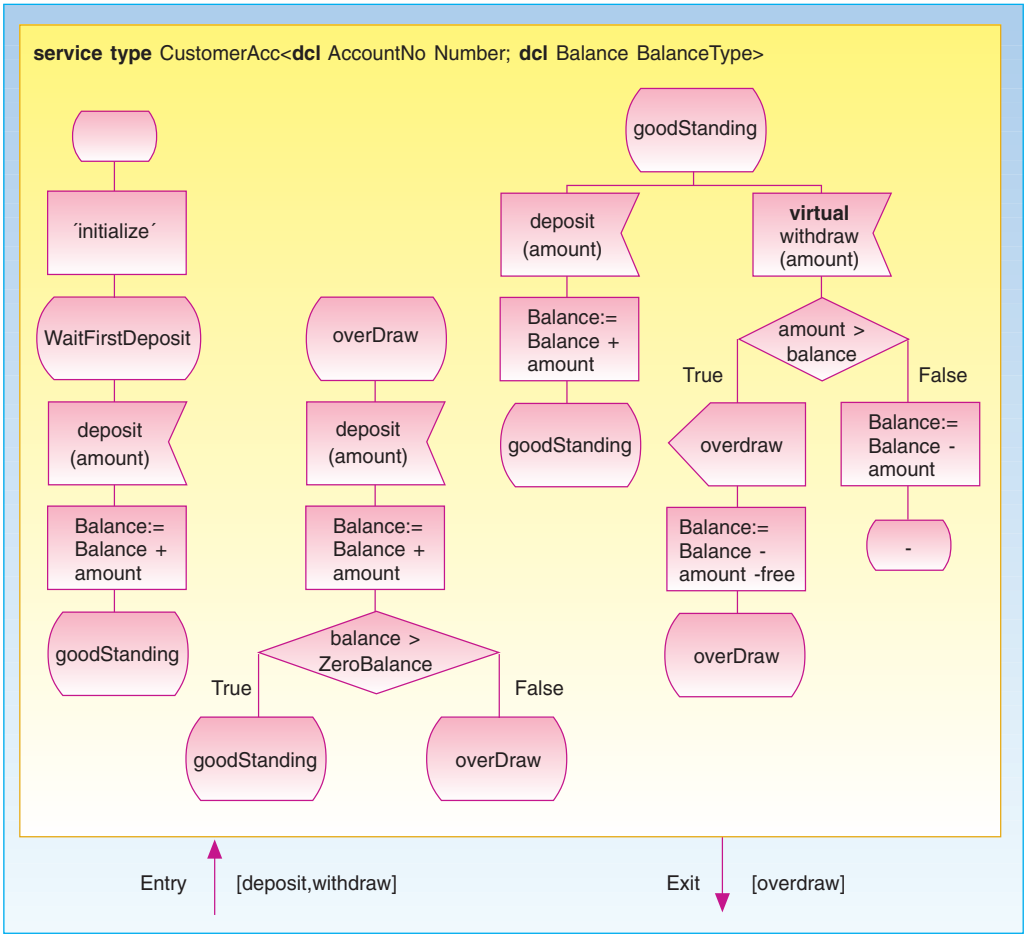


Figure 21 Service Type representing the CustomerAcc role of an Account
Note that the variables AccountNo and Balance are context parameters (enclosed in < >). The reason for this is that the final Account will be a composition of a service instance of this type with a service of a service type representing another role. The combined process will have the variables representing the account number and the balance. The service type manipulates the context parameters as if they were variables

way of representing an SDL specification.

The following issues are often discussed in the evaluation of object oriented notations.

Multiple Inheritance is (almost) regarded as a must for notations that aims at reflecting concept classification hierarchies, even though existing solutions on the problems with multiple inheritance are rather technical and ad hoc. Some of the approaches give, however, detailed advice on how to avoid multiple inheritance if possible. The reason is that the technical solutions make it complicated to use the mechanism properly.

SDL only supports single inheritance. The reason is that multiple inheritance used for concept specialisation has not reached a state-of-the-art understanding yet. As an example, no solutions provide an answer to the problem involved if the superclasses of a class are not the final classes, but only superclasses for a set of hierarchies.

Virtual procedures are in most notations not directly supported, in the sense that there is no distinction between virtual and non-virtual procedures (called methods, operations, ...). In general it is assumed that *all* procedures may be redefined in subclasses, and for some of the notations (or rather the accompanying method) it is advocated that redefinitions should be *extensions* of the original procedure in the superclass, without providing language mechanisms for it.

SDL makes a distinction between virtual and non-virtual procedures (and types in

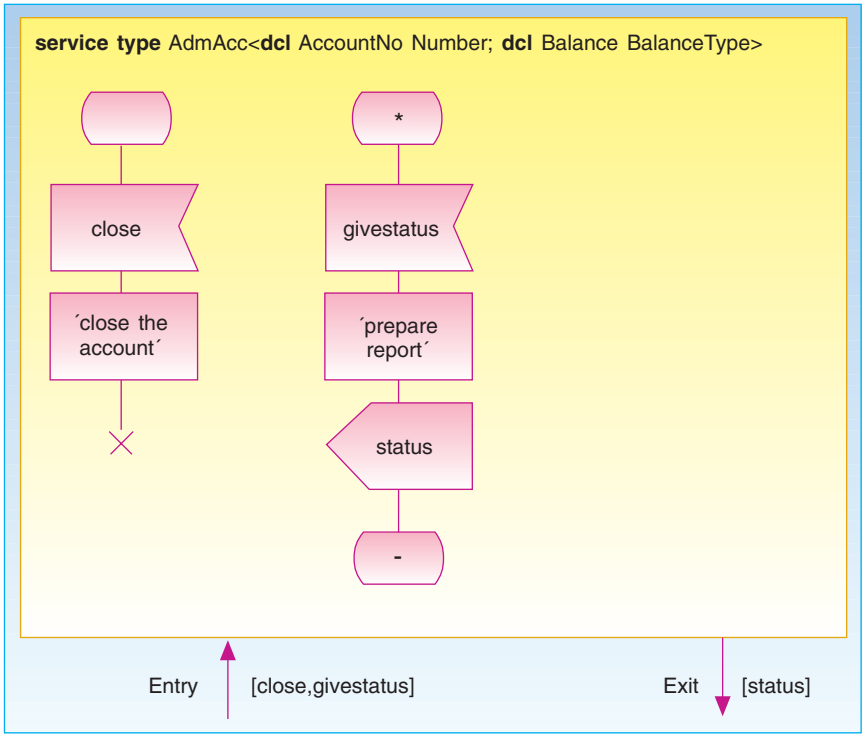


Figure 22 Service Type representing the AdmAcc role
Also here the variables AccountNo and Balance are context parameters (enclosed in < >)

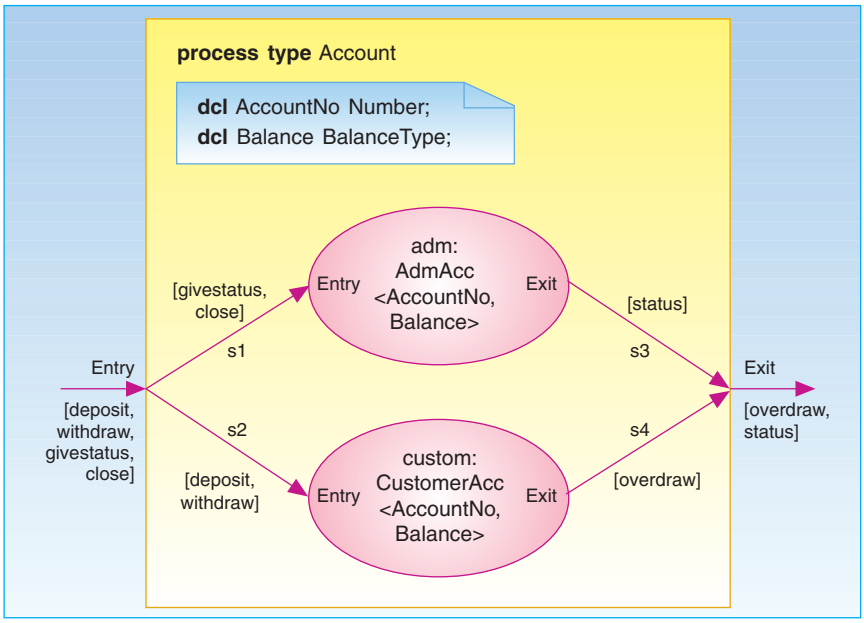


Figure 23 Composition of Services
Two service instances of type AdmAcc and CustomerAcc are composed into a process type representing a complete Account. The actual context parameters are identifiers to the variable definitions in the enclosing process type. Manipulations of the context parameters in the service types will thereby become manipulations of the variables in the process. In this example the Entry gate of Account has been kept, with the implication that the incoming signals are split and directed to each of the services. Correspondingly for the Exit gate. Alternatively, the aggregated Account could have had one set of gates for the administrative interaction and one set of gates for the interaction with the customer

general), and it enforces (syntactically) that redefinitions are extensions of the virtuality constraint. The advantage of this distinction is that it is possible to express, for a general type, that some (crucial) procedures should not be redefined.

Polymorphism is regarded as one of the big advantages of object orientation. There are many definitions on polymorphism, so the following just gives the facts for SDL:

- The same type of signal, e.g. S, may be sent to processes of different types, and as variables of type Process Identifier are not typed, then the effect of an

  ```
  output S to aP
  ```

  where aP is a Process Type variable, depends on which process is identified by aP.
- Remote procedures may be virtual, so a remote call to such a virtual procedure vP of a supertype

  ```
  call vP to aP
  ```

  will depend on the subtype of the process identified by aP, and the redefined procedure for the actual subtype will be performed.

17 Conclusion

The major language mechanisms of SDL supporting object orientation has been gradually and informally introduced by means of one example system. Focus has been on the structural aspects of SDL. The behaviour of processes, services and procedures can be specified in more detail than covered here. By using the mechanisms presented here, it is possible to sketch the structure of SDL systems consisting of blocks connected by channels, blocks consisting of blocks or processes, processes possibly consisting of services, and procedures being used for representing patterns of behaviour. By using the full language, a complete, formal specification is obtained. This formal specification can then be used for different kinds of analysis.

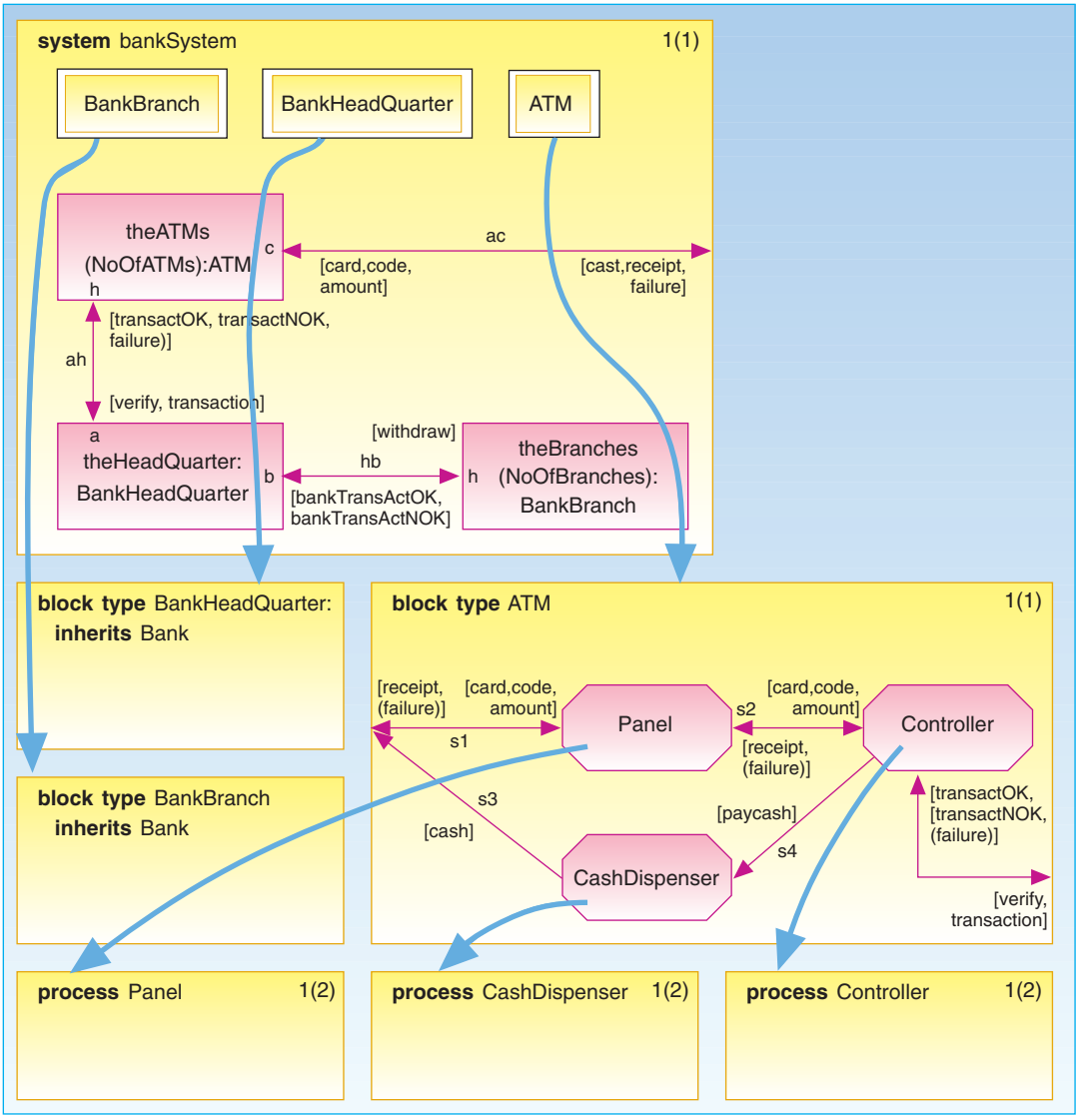


Figure 24 Referenced Diagram
The three block type symbols in the system diagram indicate that three block types are defined locally to the system, and they reference three block type diagrams. The block symbols with the names theATMs, theHeadQuarters and theBranches are not references to block diagrams, but simply specifications of block sets according to type. The three process symbols are also specifications of processes to be parts of blocks of type ATM, but they are also references to diagrams defining the properties of these process instances. Reference symbols are used in the case where an enclosing diagram refers to diagrams that logically are defined in the enclosing diagram. In order to identify diagrams that are defined in some enclosing diagram, identifiers are used; the examples here are the block supertype identifier Bank in the two block type diagrams BankHeadQuarters and BankBranch. Bank identifies a block type diagram defining the block type Bank

References

1 Belsnes, D, Dahle, H P, Møller-Pedersen, B. *Definition of OSDL, an object oriented extension of SDL.* Mjølner Report Series N-EB-7, January 1987.

2 Belsnes, D, Dahle, H P, Møller-Pedersen, B. *Rationale and Tutorial on OSDL: an object oriented extension of SDL.* Mjølner Report Series N-EB-6, April 1987.

3 Belsnes, D, Dahle, H P, Møller-Pedersen, B. Rationale and Tutorial on OSDL: an object oriented extension of SDL. In: *SDL 87: State of the Art and Future Trends, Proceedings of the Third SDL Forum 1987.* Amsterdam, North-Holland, 1987.

4 Belsnes, D, Dahle, H P, Møller-Pedersen, B. Rationale and Tutorial on OSDL: an object oriented extension of SDL. *Computer Networks and ISDN Systems,* 13(2), 1987.

5 Belsnes, D, Dahle, H P, Møller-Pedersen, B. *Transformation of OSDL to SDL.* Contract 700276.

6 Belsnes, D, Dahle, H P, Møller-Pedersen, B. *Revised Rationale and Tutorial on OSDL, an object oriented extension of SDL.* Contract 700276.

7 Bræk, R, Haugen, Ø. *Engineering Real Time Systems: An Object-oriented Methodology using SDL.* Englewood Cliffs, N.J., Prentice Hall, 1993. (BCS Practitioner Series.)

8 Haugen, Ø, Møller-Pedersen, B. *Tutorial on Object-Oriented SDL.* SPECS-SISU Report, Report No 91002. SISU c/o Norwegian Computing Centre, P.O. Box 114 Blindern, N-0314 Oslo.

9 Hauge, T, Haugen, Ø. OST - An Object-oriented SDL Tool. *Fourth SDL Forum, Lisbon, Portugal 9-13 October 1989.*

10 Haugen, Ø. Applying Object-Oriented SDL. *Fifth SDL Forum, Glasgow, UK, 29 September - 4 October 1991.*

11 Møller-Pedersen, B, Haugen, Ø, Belina, F. Object-Oriented SDL. *Tele,* 1, 1991.

12 Olsen, A et al. *Systems Engineering using SDL-92.* Amsterdam, North-Holland, 1993.

The story behind object orientation in SDL-92

This version of the story is the version as seen from the author, and it may therefore be biased towards Norwegian contributions. If the complete story should be written it would require some more work together with the many people involved; time has not allowed that.

It all started in the sixties with the definition of SIMULA.

Developed in a setting where the problems were to make operational models of complex systems, SIMULA became not only a programming language, but also a modelling language. Inspired by this use, the pure specification language DELTA was developed, also at the Norwegian Computing Centre. In the eighties, the combined experience with SIMULA and DELTA led to the definition of an object oriented programming language, BETA (Norwegian Computing Centre, University of Aarhus, University of Aalborg). The approach in this series of developments has been called the Scandinavian approach to object orientation.

For SDL it started in 1985 when EB Technology (now ABB Corporate Research) started work on the design of a graphical, object oriented specification language. With background in the SIMULA tradition, the group at EB Technology had no doubts that this was possible to do in Norway. The Norwegian Computing Centre was engaged to help in defining it. The first intentions were to design a language from scratch.

This project triggered in 1986 the Møjlner Project, a 3 year Nordic research project with participants from Denmark, Norway, and Sweden, funded by the Nordic Industry Fund. The theme of the project was object orientation, and it should produce languages and tools for industrial use. At that time, object orientation was still for the few and still regarded as something useful for making toy systems in isolated laboratories. Discussions early in the project revealed that users of the project results would rather see an object oriented extension of the CCITT recommended language SDL, than a new language. The reason was that SDL was widely used in some of the organisations. So, while the Danish part of the project (Sysware Aps and University of Aarhus, Computer Science Dept.) made the first implementation of BETA and elements of a programming environment, and the Swedish part (TeleLOGIC and University of Lund) made an advanced programming environment for SIMULA, then the Norwegian part of the project (EB Technology and Norwegian Computing Center) aimed at defining an object oriented extension of SDL and implementing a tool supporting this. The working title of the extended language became OSDL.

The first proposal was made as part of the Mjølner Project (1). It is dated January 12, 1987. The authors had no previous experience with SDL, which also meant that they had no predefined attitudes towards what was possible or not.

The main ideas behind the proposal were based upon the Scandinavian approach to object orientation. Two examples may illustrate this: According to this approach, objects shall have their own action sequence, and they shall be able to act concurrently – this led to the choice of SDL processes to be the primary objects. It shall also be possible to specialise action sequences – this led to the notion of virtual transitions. Virtual procedures is a well-known and accepted mechanism in almost all object oriented languages, so that was introduced as well.

Around this first proposal the Norwegian part of the project had very valuable discussions with the Swedish partners in the project, including Televerket. The first proposal was a rewrite of the whole recommendation.

In light of object orientation, some of the (strange) differences between e.g. blocks and systems, were removed, and processes and blocks were allowed to be at the same level.

A separate proposal on remote procedures in SDL came at the same time from Sweden. This notion was also part of OSDL. At later stages these proposals were merged.

A tutorial on the extensions was ready April 30, 1987 (2). This formed the basis for the paper at the SDL Forum in the Hague, April 1987, where OSDL was presented for the first time (3). The authors were rather new in this field, so they were not aware that among the audience were most of the people that were going to join the work on object oriented SDL the next 5 years. These were especially people being active in the Study Group X of CCITT, maintaining SDL.

A more comprehensive version of the conference paper appeared later in Computer Networks (4).

A contract with Norwegian Telecom Research brought about the important work on transforming OSDL to SDL (5). In addition, specific language mechanisms as e.g. virtual process types were asked for, and included (6). This work turned out to be important when it was presented as a proposal for CCITT.

The RACE I project SPECS started in 1988. The aim was to reach a common semantic platform for SDL and LOTOS, and to define corresponding methods for SDL and LOTOS. EB Technology became partner of the project, with Norwegian Computing Center, Kvatro, and DELAB as subcontractors.

The project had to adhere to standard languages, so SDL-88 was the language. On the other hand, for the methodology part the project studied mechanisms for component composition and reuse. It was therefore natural to continue the work on object oriented SDl in the project.

Even though SPECS was an important vehicle for the development of SDL-92, the final SPECS Book is based upon SDL-88. The reason is that it was not possible to coordinate the various methods and tools developed in the project on SDL-92.

The SPECS project was financed by NTNF and NTR

The SISU Project started at almost the same time, with participation from a series of Norwegian companies involved in real-time systems engineering. The project decided to base its methodology work on OSDL.

A requirement for using OSDL was that the extensions became part of the CCITT Standard. The SISU project therefore contributed to the work in CCITT.

The CCITT work on extending SDL started at the last meeting of the 1984-1988 study period. The first document describing the extensions as needed by CCITT is dated December 23, 1988. The real work began at the CCITT meeting in May, 1989.

The first question was whether the extensions should be contained in a separate recommendation or incorporated in Z.100. For the first 2 years the work was directed towards a separate recommendation. When the work became more detailed, it turned out that there would be many cross-references between the two, so it was decided to merge the extensions into Z.100 and at the same time organise the document so that it would be easy to spot the extensions.

During 1989 and first part of 1990, two proposals were regarded as kind of competing: The OSDL proposal and a Danish proposal on parameterised definitions, expected to cover both context parameters, gates, and virtuals. At the last meeting in 1989 it was decided to merge the two proposals, and at the June CCITT meeting in Espoo, Finland, the two proposals were merged into one written proposal. The Danish proposal corresponds more or less to the notion of context parameters, so they came in addition to virtuals from the OSDL proposal. At one point in time, it was investigated if the context parameters could also cover gates, but it turned out that gates were of a different nature: context parameters are parameters of definitions (disappearing when given actual parameters), while instances of types with gates will have the gates as connection points.

Through a number of experts meetings and more formal CCITT meetings during 1990 and 1991, the proposal was re-worked several times, implications for the rest of SDL was considered and a formal definition was given by transformation to Basic SDL.

Even though it started in Norway, the work with the proposal could not have been undertaken without the efforts of the members of the Study Group X.

- The Chairman of the group and rapporteur for SDL maintenance kept all the loose ends, ran very efficient meetings, and as a last contribution in the study period, he edited the final version of the SDL-92 standard.
- The formal definition rapporteur pin-pointed many inconsistencies and points for discussions and clarification.
- The rapporteur for the Methodology Guidelines (and therefore engaged in what the users might think about the new SDL) gave valuable contributions.
- Telia Research made a semantic checker for the new SDL, thereby posing nasty questions, but also contributing with possible answers.
- The UK delegation was always ready with a good formulation and good choice of words – in addition to representing all the good traditions of SDL.

The CCITT plenary meeting in Helsinki in March, 1993 recommended a new version of SDL with the extensions as proposed by Study Group X.

In addition to the CCITT work, several of the people involved have contributed to the spreading of the ideas to established and potential users of SDL. Highlights in this activities are (according to the knowledge of the author):

- OSDL courses held at EB as part of the Møjlner project and OSDL as part of university courses, 1989-1992
- Numerous SISU courses, based on methodology work in the SISU project, and leading to a book (7)
- Numerous talks by the Chairman of the study group
- Talk on tools for OSDL at the SDL Forum 1989, by EB Technology (9)
- A combined SPECS-SISU tutorial (8)
- A talk on applying Object-oriented SDL (10)
- Tutorials at the FORTE 90 conference and at the SDL Forum 91, based on the SPECS-SISU tutorial, by Norwegian Computing Centre
- A joint talk at Telebras in Brazil, in conjunction with the CCITT meeting in Recife, by Norwegian Computing Centre, TFL and Telia Research
- Various SDL-92 seminars at: TeleLOGIC, SPECS, ARISE, GPT, BNR, ATT, by Norwegian Computing Centre
- Forthcoming book on SDL-92 (12).

An introduction to TMN

BY STÅLE WOLLAND

621.39.05:65

Motivation, scope and objectives

Management of telecommunications resources have up to now been a mixture of manual, automated (or computer-supported) and automatic functions. These have usually been performed by stand-alone, ad-hoc procedures or systems. The systems have been equipment specific and vendor supplied. Or they have been developed by the telecoms operators as stand-alone systems for a specific purpose.

The last few years have seen a growing effort to specify the management aspects of the various telecoms resources (network elements, networks and services) so as to present themselves to the management functions in a uniform manner. This work has focused on harmonised management interfaces and generic management functions. By standardising the interfaces there is no longer a need to duplicate large parts of the specification and development work for each operator and supplier.

The potential for re-using data definitions, registrations and repositories, management procedures and software or indeed management hardware resources like processors, storages or communication channels have until recently only been exploited to a small extent. The savings in having a common set of management resources (people, data and procedures) all interacting using a common management network (physical or logical) promise to be very great indeed.

Some of the benefits have been touched upon already:

- Vendor independence
- Equipment, network and service independence
- Reusability of management data and functions
- Reusability of management personnel and organisation (the choice of centralised or distributed management becomes more flexible)
- Internationally standardised management interfaces
- Less costly specification and procurement of management applications
- Increased interoperability of national and international management resources.

All these factors should lead to decreased specification, procurement and operational costs for management. Other benefits are increased functionality and performance including quality of service.

The possibility of allowing customers or other operators access to a restricted set of management functions is also being explored. The obvious threat is security and integrity of potentially valuable and sensitive management data and functions. On the other hand, there are considerable savings to be had for both providers and users with such facilities.

TMN –Telecommunications Management Network – is the term adopted by ETSI and CCITT for describing the means to transport and process information related to the management of telecoms resources. The term denotes an integrated and interoperable approach to distributed management of these resources. That is to say, the monitoring, co-ordination and control that is required for smooth operation of all the telecoms means. There is a very similar notion within ISO/IEC for the management of OSI (Open System Interconnection) resources. There it is called OSI Management. Work is going on to bring these notions in line. This harmonisation work is done by CCITT SG VII and ISO/IEC JTC1/
SC21/WG4. The two notions are now to a large extent identical.

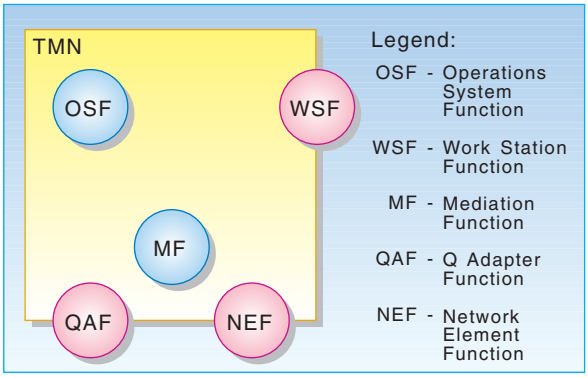


Figure 1 TMN Basic concepts (functional view)

Basic concepts of TMN

The basic ideas of TMN are interoperability, re-use and standardisation of management capabilities together with an evolutionary approach to the introduction of TMN into the present network.

The basic concepts used for TMN are illustrated in Figures 1 and 6. A further detailing of the concepts can be found in CCITT Recommendation M.3010,(1), ISO/IEC IS 7498-4 (2) and ISO/IEC IS 10040 (3).

The functional view of TMN is described in terms of function blocks and reference points between these. The purpose of the function blocks is to group related functionality together so that specification and implementation may be simpler. The interaction between the function blocks defines service boundaries and they are described by reference points which are collections of data and operations. The reference points are candidates for implementation as interfaces. There are five types of function blocks. Operations Systems Function (OSF), Mediation Function (MF), Work Station Function (WSF), Network Element Function (NEF) and Q Adapter Function (QAF). The OSF processes information for the purpose of management of telecoms resources or other management resources. The NEF communicates with the TMN for the purpose of being managed. The WSF provides the means to interpret and present the TMN information to the management end user. The MF acts on information passing between an OSF and an NEF or a QAF by storing, adapting, filtering, thresholding and condensing this information to facilitate the communication. The QAF connects non-TMN entities to the TMN by translating between a TMN reference point and a non-TMN (proprietary) reference point.

The function blocks are interconnected by reference points (RP). The reference points are conceptual points describing the functional interaction or information passing between two function blocks. They define the service boundaries between the blocks.

The first type of RP is between the OSFs (where some of the management capability is implemented) and the telecoms resources (Network Elements, Networks, Services). This is denoted q (q3 and qx; see Figure 6).

The second type of RP is between the management functions and the presentation tool (in most cases a work station). This RP is called f.

A third type of RP is between two OSFs in two different TMN domains, e.g. two telecoms operators. This RP is foreseen

Table 1 Relationship of functional blocks to functional components

Functional block	Functional components	Associated message communications functions
OSF	MIB, OSF-MAF (A/M), HMA	MCF_x, MCF_{q3}, MCF_f
OSF subordinate(1)	MIB, OSF-MAF (A/M), ICF, HMA	MCF_x, MCF_{q3}, MCF_f
WSF	PF	MCF_f
NEF_{q3}(2)	MIB, NEF-MAF (A)	MCF_{q3}
NEF_{qx}(2)	MIB, NEF-MAF (A)	MCF_{qx}
MF	MIB, MF-MAF (A/M), ICF, HMA	MCF_{q3}, MCF_{qx}, MCF_f
QAF_{q3}(3)(4)	MIB, QAF-MAF (A/M), ICF	MCF_{q3}, MCF_m
QAF_{qx}(4)	MIB, QAF-MAF (A/M), ICF	MCF_{qx}, MCF_m

Legend: PF = Presentation Function
MCF = Message Communications Function
MIB = Management Information Base
MAF = Management Application Function
ICF = Information Conversion Function
A/M = Agent/Manager
HMA = Human Machine Adapter

Note: MAF (A/M) means management application function in Agent or manager role.

(1) This is an OSF in the subordinate layer of the logical layered architecture.

(2) The NEFs also include Telecommunications and Support resources that are outside of the TMN.

(3) When QAF_{q3} is used in a manager role the q3 reference point lies between the QAF and an OSF.

(4) The use of QAF in the manager role is for further study.

In this table the subscripts indicate at which reference point the functional component applies. Individual functional components may not appear or may appear multiple times in a given instance of a functional block. An example of multiple occurrences is of several different management application functions (MAFs) in the same instance of a functional block.

to have security attributes to enable the negotiation of strict security functions in addition to a (more restricted) set of management functions. This RP is called x.

A reference point can be implemented by adopting a particular communication protocol and a particular communication service (message set). They are then called interfaces and capital letters are used to denote them. In TMN the RPs are candidate interfaces. They become interfaces in case the function blocks are implemented in different physical locations.

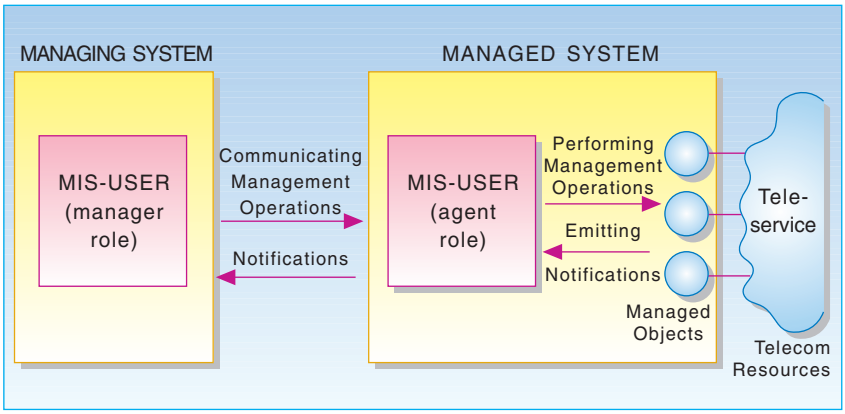


Figure 2 Basic TMN communication and processing

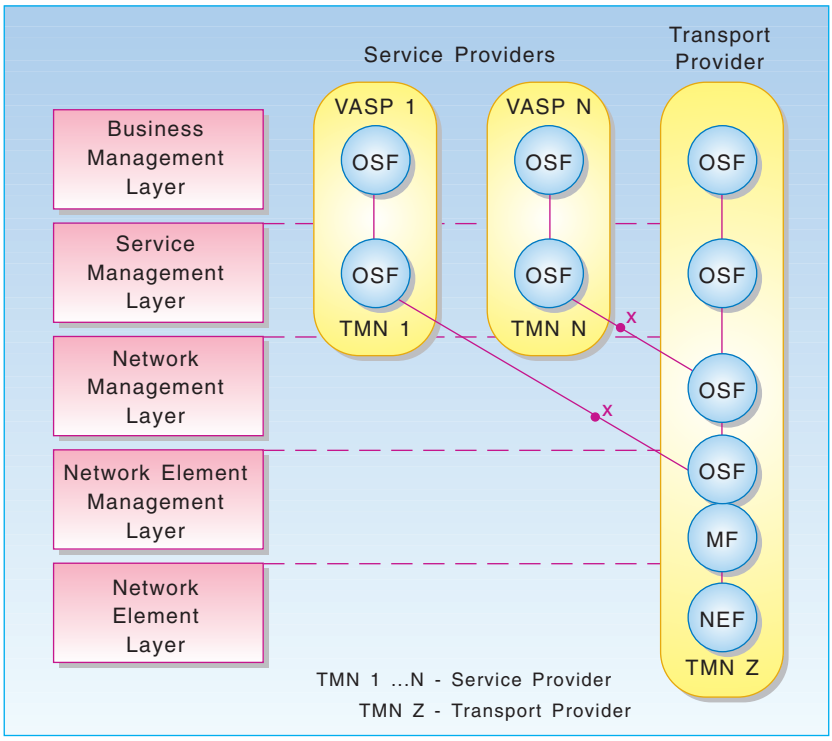


Figure 3 Non-peer-to-peer interaction between management operators and layers (example)

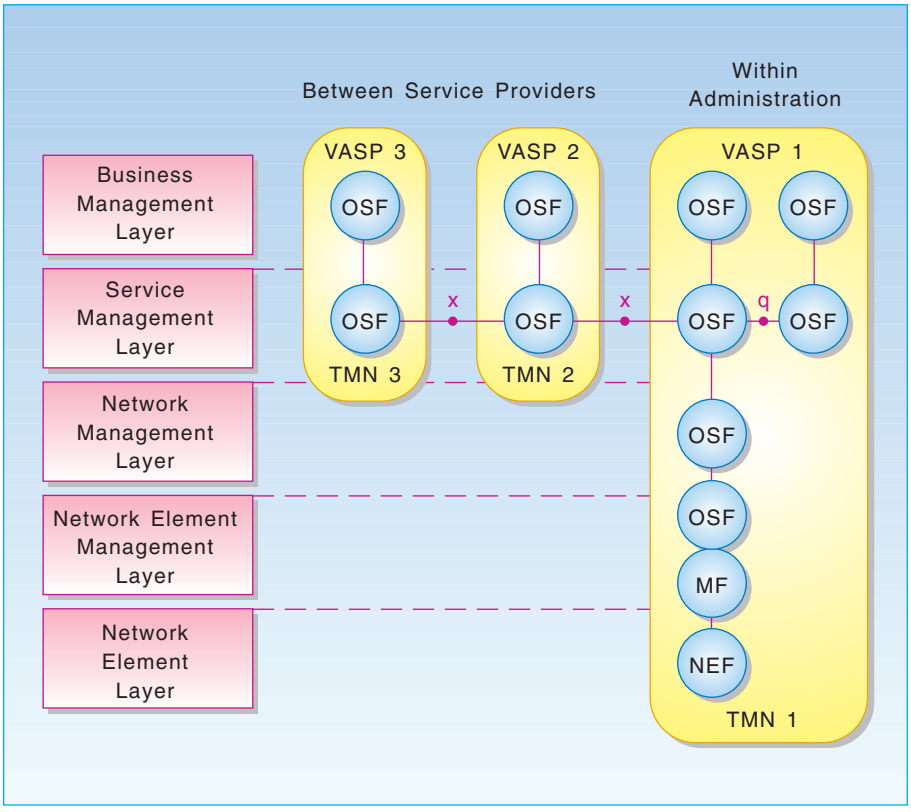


Figure 4 Peer-to-peer interaction between operators and layers (example)

This is summed up in Figure 6, which is a functional representation of the management capabilities.

It is important to note that in TMN it is the interfaces that are being standardised internationally and not the function blocks. The status in 1993 is that the Q3 interface is fairly well advanced on the way to being standardised. The X interface is at an early stage and the F interface is beginning to receive the first attention.

The function blocks are composed of elementary function blocks called functional components. There are six types of functional components: Management Application Function (MAF), Management Information Base (MIB), Information Conversion Function (ICF), Presentation Function (PF), Human Machine Adaptation (HMA) and Message Communication Function (MCF). MAFs perform the actual management functions by processing Managed Objects (MO)(see below and also separate article in this volume). MIBs are conceptual repositories for the MOs. The ICFs convert between the different information models of two interfaces. The HMAs perform the conversion of the information model of the MAF to the information model of the PFs. The PFs perform the translation between the TMN information model to the end user presentation model. The MCFs are performing the physical exchange of management messages with peer entities. The definitions of these can be found in CCITT Rec. M.3010 (1). Table 1 gives the relationship between the function blocks and the functional components.

Basic management communication and processing

The basic TMN management communication interactions and operations are shown in Figure 2.

The Management Information System User (MIS-U) can either be a management application (OSF) in the Manager role (which is a role that the management process can play during one particular management interaction in the communication process) or in the Agent role. The definition of the manager is that it can direct management operations to the remote agent and receive management notifications. The definition of the agent (which again is a role the process can

have during one particular interaction) can perform management operations and receive management notifications from the Managed Objects (MO) and receive directives for operations from and send notifications to the manager.

The MOs are elementary management capabilities seen from the MIS and they represent an abstraction in the form of a management view of the telecoms resources. An MO is defined in terms of attributes it possesses, operations that may be performed on it, notifications it may issue and its relationships with other MOs.

There are two things to note about the MOs. The first is that MOs are being standardised internationally. The second is that MOs are specified in an object oriented fashion. The template for the specification is the ISO/IEC OSI Guidelines for the Description of Managed Objects (GDMO) (4). For a further definition of MOs, see separate article in this volume (6).

The managing and managed systems together are viewed as a distributed management application. They interact via standardised interfaces with protocols that reference the MOs. As already mentioned, these interfaces are essential for the TMN concept.

The idea of a mediation function was already introduced above. From Figure 6 and Figure 7, a further elaboration can be made. As the name indicates, this function takes care of a set of pre-processing tasks before the management data and operations are processed proper. Mediation is foreseen for communication with data concatenation (to reduce the volume of data flows) and connection concentration (to reduce the number of connections between a potentially large number of network elements and a smaller number of operations systems (OS)). Other examples of management mediation is decision making (for example thresholding alarms), routing of management messages and conversion between different management interfaces. The Mediation Device (MD) shall on the one hand interconnect telecoms resources to OSs via the mediation function and on the other hand interconnect telecoms resources to OSs again via the mediation function. The telecoms resources will in some cases support a more simple management interface than the OS. Hence, a mediation device will have two TMN interfaces – one for the OS side and one for the NE side.

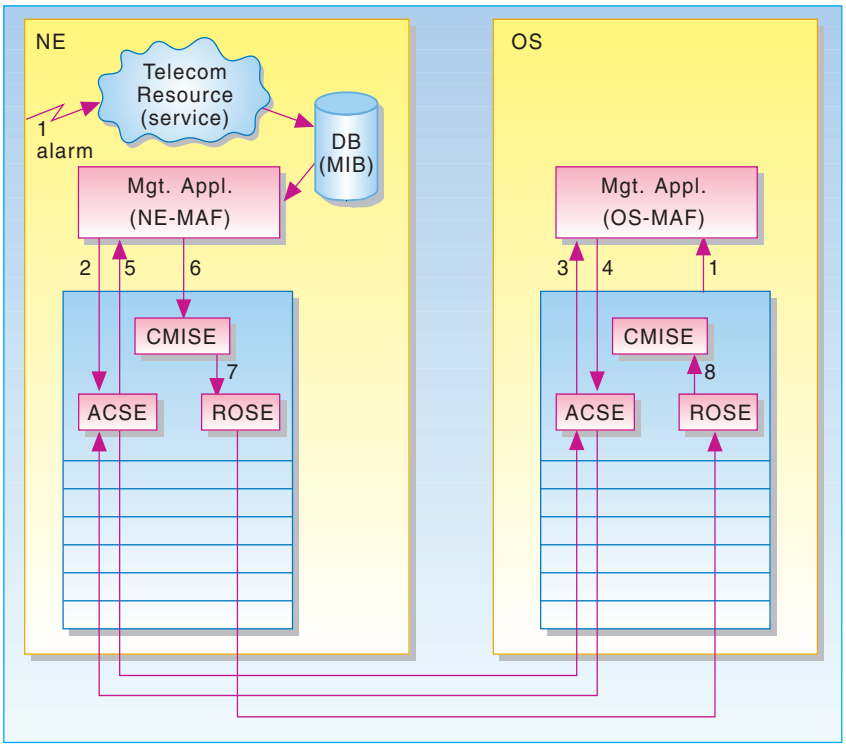


Figure 5 Example of management communication and processing (physical)

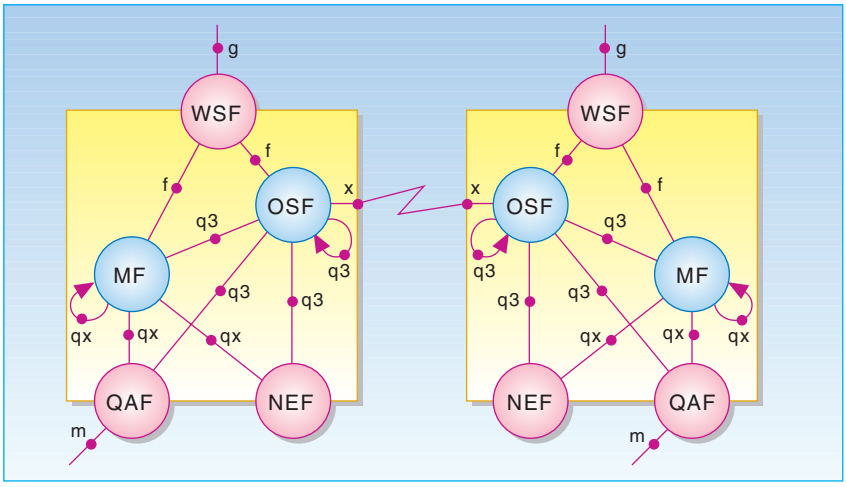


Figure 6 TMN interaction (functional view)

Adaptation is a management function required in case the interface to a telecoms resource is not TMN standard. The adaptation function will either be performed in the network element or in a separate Adapter. The purpose is to convert a proprietary interface of the NE to a TMN interface. This will allow interconnecting for example existing non-TMN NEs with a TMN. This will also permit internal, proprietary interfaces for internal communication inside the telecoms resources, while providing a TMN interface for external management communication. Hence, an adapter will only have one TMN interface.

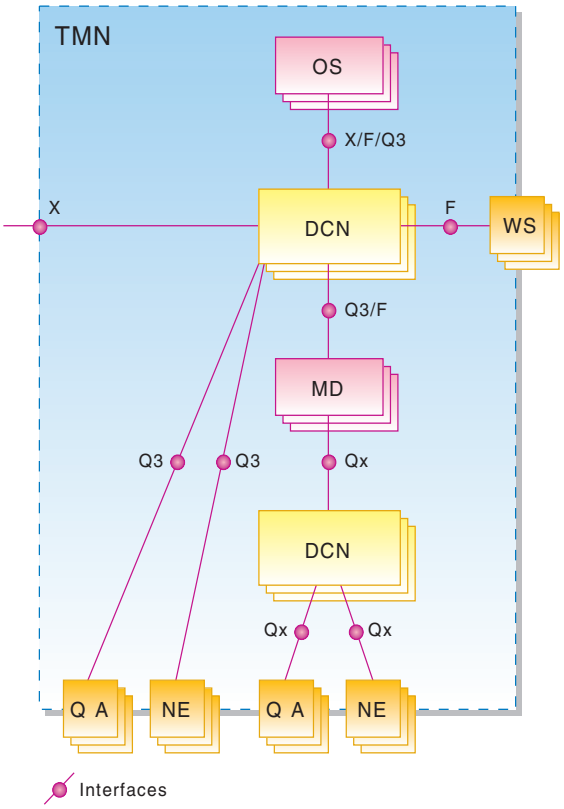


Figure 7 Simplified physical architecture (example)

Note 1: For this simplified example the building blocks are considered to contain only their mandatory functions (see Table 1).

Note 2: The interfaces shown on either side of the DCN are actually a single interface between end systems for Layers 4 and above. For Layers 1 to 3 they represent the physical, link, and network interface between an end system and the DCN

Logical layered architecture

ISO categorises management into five functional areas: Fault, performance, configuration, accounting, and security. CCITT and ETSI use a different categorisation with more functional areas when seen from the management end operator. This division is not important. The final management capability to be provided at the operator interface is not for standardisation, but will be decided by the requirements of the individual telecoms operators. Their needs will depend on the business idea for their offering of the telecoms service, their investment profile and their management organisational requirements. This offering of management capability will be an area open for competition between the telecoms operators in the future. This does not, however, preclude some aspects of this interface to be standardised.

Seen from the manager/agent view, the management capability can be divided into four areas with respect to what aspects of the telecoms resources they address: Network Element, Network, Service, and Business. This grouping of management functionalities is called the Logical Layered Architecture (LLA) and is recommended for TMN. If these layers are implemented in different physical resources or locations, there will be an interface between the layers and the layers will be implemented in different OSs.

How is management performed?

Figure 5 shows an example of how management interaction and processing take place in the case of an alarm for a telecoms resource (NE).

The alarm causing event happens for some reason inside the telecoms resource. The event is registered there and is communicated through a non-TMN interface to an MO. There is an ensuing state change of this object (attribute value change). For certain events there is defined a certain behaviour for the MO. In this case there is a notification emitted by the MO. The NE-MAF as an agent receives the alarm notification from the MO and decides to report this to the manager. The agent first establishes the connection over the TMN to the manager by invoking the communication service primitives of the ISO OSI ACSE (Association Control Service Element) which is always used for TMN communication. As a result of this invocation, an association (connection) is set up between the agent and the manager management processes. This is done prior to the management data exchange proper. Management data exchange is performed by invoking the ISO/IEC OSI and CCITT CMISE (Common Management Service Element) (5) service primitives for communicating management data. The CMISE services basically reference the MOs for the purpose of processing these. The CMISE makes use of the common ISO OSI ROSE (Remote Operations Service Element) to communicate its management operations and data directives. By using these communication services and protocols, the management functions can exchange their operations on managed objects and receive notifications. By processing the operations on the MOs, the MAFs can effect management of the telecoms resources in a generic and standardised way.

Centralised or distributed management?

TMN is a distributed concept for centralised management!

TMN will allow centralised management through the use of harmonised management interfaces to telecoms resources, an interoperable management network, mediation devices and adapters.

TMN will allow an evolutionary approach (gradual introduction) by using adapters that will allow co-existence of present non-TMN management and telecoms systems and new TMN systems. Hence, the investment in management resources and equipment today should be well protected. A revolutionary approach would have been meaningless in terms of economics.

Achieving centralised and standardised management for the different network and service resources can potentially be very cost-saving. A cost-reduction can be obtained from a decrease in the number of required management personnel, a possible simplification in the training required due to the introduction of automation (although potentially training can become more complex if more sophisticated management is undertaken), the re-use of management systems that require to be specified, developed and procured, the re-use of the management data that require to be defined, registered, processed and stored, the re-use of the processing platforms that the management systems run on, etc.

But TMN is a concept for distributed management as well!

TMN encompasses concepts for peer-to-peer management communication between managing systems and managed systems. In a particular management data exchange, the managed system acts as an agent and the managing system acts as the manager. The agent is capable of receiving management directives from the manager and executing the directives by accessing the managed objects, processing these and returning answers if required. Hence, even if TMN is taking place in an integrated fashion logically,

the management processing can be realised as a distributed process.

This characteristic of logical integration, but optional physical centralisation or distribution, allows a great degree of flexibility in the implementation of the management functions.

TMN functional reference model

Figures 1 and 6 summarise the TMN functional reference model.

Here, the types of functional blocks and reference points that interconnect these are identified for TMN.

Table 1 gives the relationships between the TMN function blocks and the TMN functional components that can exist in a TMN.

From Figure 6 we notice that an operator is capable of interacting directly over the network both with the operations system and the mediation device through the F interface. This is not the case for the telecoms resource (here represented by the NE) and the adapter.

From Figures 3 and 4 we also notice that two telecoms operators (either service operators or network operators) can interact directly with each other through the X interface. We see that normally two service providers (in this case they are value-added-service-providers, VASPs) are expected to interact via the x reference point at the service layer in a peer-to-peer fashion. However, in special cases the interaction may take place via an X interface other than the service layer. This is shown in Figure 3, where the interaction is not required to be peer-to-peer.

Conclusions

TMN is a wide and complex area, and international agreement has taken a long time to emerge. Due to the wide scope of TMN, it will take some time yet before the final shape can be seen. However, TMN is a concept for gradual introduction and enough of the specifications are in place to allow early realisations at this stage. Since the basic idea is to perform generic, standardised management over a large and diverse range of elements, networks and services, it is not surprising that the realisation will take its time.

However, the status in 1993 for TMN is that the first implementable and indeed implemented functions are beginning to emerge. For management communication, the ISO/IEC CMIS/CMIP protocol has been standardised and implementations can be obtained commercially.

The proposed standards for the managed objects are in many cases close to finalisation. In this area the work is going on with full strength. The MOs for network element management are fairly well specified. The MOs for network level management and service management are yet at an early stage. Naturally, the generic aspects of management are specified first. MOs describing a wide spectre of common management capabilities like fault handling and performance handling, etc., are addressed in a series of emerging standards and recommendations. In addition, managed objects for specific areas such as management of SDH (Synchronous Digital Hierarchy) and ATM (Asynchronous Transfer Mode) are receiving a lot of attention. The first proposals have been available for some time.

References

1 CCITT. *Principles for a Telecommunications Management Network. (*Recommendation M.3010.)

2 ISO. *OSI Basic Reference Model Part 4: Management Framework. (*ISO/IEC IS 7498-4.)

3 ISO. *OSI Systems Management Overview. (*ISO/IEC IS 10040.)

4 ISO. *OSI Structure of Management Information – Part 4: Guidelines for the Definition of Managed Objects. (*ISO/IEC IS 10165-4.)

5 ISO. *OSI Common Management Information Service Definition/Common Management Protocol Specification.* (ISO/IEC IS 9595/9596.)

6 Kåråsen, A G. The structure of OSI management information. *Telektronikk,* 89(2/3), 90-96, 1993 (this issue).

The structure of OSI management information

BY ANNE-GRETHE KÅRÅSEN

681.327.8:006

1 Introduction

This article presents the structure of OSI management information. Details concerning the notation used to express this information are not included. However, the notation of the OSI management formalism is visualised by presenting some examples. First, the context in which OSI management information is applied, namely OSI management, will be briefly described.

OSI is an abbreviation for Open Systems Interconnection, which defines a framework for interconnections between systems that are "open" to each other. When providing interconnection services between open systems, facilities are needed to control, co-ordinate and monitor the communications resources involved. Within an OSI environment, this is the concern of OSI management.

OSI management is required for several purposes, and covers the following functional areas:

- fault management
- accounting management
- configuration management
- performance management
- security management.

OSI management is accomplished through systems management, (N)-layer management and (N)-layer operations. Systems management provides the normal management mechanisms within an OSI environment.

The systems management model, described in (1), identifies the various aspects of systems management. These aspects are:

- management information
- management functions
- management communications
- organisational aspects.

(2) introduces the OSI management framework, and (1) gives a general description of systems management. The logical structure used to describe systems management information, i.e. the information aspect of the systems management model, is the topic of this article. The main source of information on the topic is (3).

2 General

The management information model provides methods to model the management aspects of communications resources, and to structure the exchange of management information between systems. The information model is object oriented, i.e. it is characterised by the definition of objects with specified properties. In general, these objects are abstractions of physical or logical entities. Objects that represent communications resources being subject to management are termed managed objects, and are OSI management's view of the resources in question.

Management of a communications environment is an information processing application. Management applications perform the management activities by establishing associations between systems management application entities. These application entities communicate using systems management services and protocols. Management applications are termed MIS-users, and each interaction takes place between two MIS-users, acting as manager and agent, respectively. Figure 1 illustrates systems management interactions and related terms.

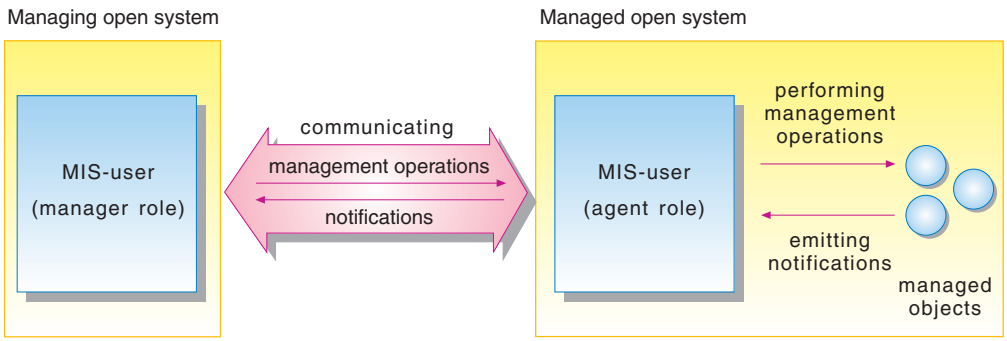


Figure 1 Systems management interactions

Management has access to managed object properties that are visible at the managed object boundary, i.e. properties exposed through the object's attributes, operations and notifications (see chapter 3). The internal functioning of the managed object and the resource being represented are otherwise not visible to management. This principle of encapsulation serves to protect the integrity of the managed object and the resource it represents.

The set of managed objects within a system, together with their attributes, constitute the system's Management Information Base (MIB). The MIB comprises all the information that may be transferred or affected through the use of management protocols within the system.

3 Managed object classes

Managed objects having the same defined properties are instances of the same managed object class. Specification of a managed object class and its associated properties is documented by using a set of templates (see chapter 9).

The managed object class definition, as specified by templates, consists of:

- the position in the inheritance hierarchy (see chapter 4)
- a collection of mandatory packages (see section 3.1)
- a collection of conditional packages, and the condition under which each package is included
- an object identifier (see chapter 6).

A package structure consists of the following:

- attributes
- operations
- notifications
- behaviour.

Generic and specialised managed object classes are defined in several CCITT Recommendations, and are also defined by other international organisations, such as ISO/IEC and ETSI. These organisations collaborate in the field of systems management. Within OSI management, (4) defines the generic managed object classes from which other classes are derived. To illustrate some of the terms introduced in this article, the definition of one of these generic managed object classes is shown in Figure 2. The logRecord managed object

class definition is chosen as an example because it is comparatively short and simple.

3.1 Packages

The properties of managed objects, described through their attributes, operations, notifications and behaviour, are collected in packages within the managed objects. This is primarily done to ease the specification task. If a package comprises some particular set of properties that are common to several managed object classes, previously defined packages may be re-used in new class definitions.

Packages are either mandatory or conditional when included in a specific managed object class definition. Attributes, operations, behaviour, and notifications defined in mandatory packages will be common to all instances of a given class. Attributes, operations, behaviour, and notifications defined in a conditional package will be common to those managed object instances that satisfy the conditions associated with that particular package.

Only one instance of a given package may be included in a managed object. For this reason, name bindings are not assigned to packages. Name bindings are defined separately, and are not part of the managed object class definition (see section 5.2).

To ensure the integrity of managed objects, operations are always performed on the managed object encapsulating the package, not on the package itself. As packages are an integral part of the managed object, they cannot be instantiated by themselves. A package and the managed object to which it belongs must be instantiated and deleted simultaneously.

Managed objects include a separate Packages attribute that identifies all packages being present in that particular managed object instance.

The logRecord managed object class includes the mandatory package logRecordPackage, and the package definition is directly included in the managed object class definition. If the package in question had been defined separately, using a complete package template as described in (5), the construct "CHARACTERIZED BY logRecordPackage" could have been used in the managed object class definition, referencing the package.

```
logRecord    MANAGED OBJECT CLASS
DERIVED FROM                    top;
CHARACTERIZED BY
logRecordPackage                PACKAGE
        BEHAVIOUR
        logRecordBehaviour      BEHAVIOUR
        DEFINED AS "This managed object represents the information
        stored in logs";;
        ATTRIBUTES
        logRecordId             GET,
        loggingTime             GET;;

REGISTERED AS  {smi2MObjectClass 7};
```

Figure 2 The logRecord managed object class definition (from (4))

3.2 Attributes

Attributes are properties of managed objects. Attributes are formally defined by using an attribute template, described in (5).

An attribute has an associated value, and this value may have a simple or complex structure. Attribute values may be visible at the managed object boundary, and may determine or reflect the behaviour of the managed object. Attribute values may be observed and manipulated by attribute oriented operations, described in section 8.3.

The syntax of an attribute is an ASN.1 type (see chapter 11) that abstractly describes how instances of the attribute value are carried in communication protocols. The concrete transfer syntax is not specified by the ASN.1 type.

A number of attribute properties are not directly included in the attribute definition. These properties are listed when referencing the attribute in the managed object class definition. Attribute value restrictions, default or initial attribute values associated with the attribute, and operations that may be performed on the attribute, are included in the property list.

Restrictions on attribute values may be defined through the use of permitted and/or required value sets for the attribute. The permitted value set specifies all the values that the attribute is permitted to take. The permitted value set will be a subset of the attribute syntax values. The required value set specifies all the values that the attribute must be capable of taking. The required value set will be a subset of the permitted value set. Restrictions on the attribute value set should only be made if they are based on specific features inherent in the semantics of the attribute.

An attribute may be set-valued, i.e. its value is an unordered set of members of a given type. The size of the set is variable, and the set may be empty. However, the cardinality of the set is subject to restrictions. Members of set-valued attributes may be added or removed by use of defined operations.

Several attributes may be addressed collectively by defining an attribute group. An attribute group may be fixed or extensible. The attributes comprising the attribute group are specified individually, and operations addressed to the attribute group will in fact be performed on each attribute in the group. Attribute groups do not have values of their own.

The attributes included in the logRecord managed object class are both single valued. No attribute value restrictions are defined, and the attributes have no defined initial or default values. The property lists consist of "GET", and this indicates that both attributes are read-only. The logRecord managed object class does not include any attribute groups.

The logRecordId attribute definition and ASN.1 syntax is shown in Figure 3.

3.3 Operations

Two types of operations are defined, namely operations that affect the attributes of a managed object and more complex operations that affect the managed object as a whole. Create, Delete, and Action are operations of the latter type. Operations are further described in chapter 8.

3.4 Notifications

Managed objects may emit notifications that contain information relating to internal or external events that have

```
logRecordId    ATTRIBUTE
        WITH ATTRIBUTE SYNTAX  Attribute-ASN1Module.LogRecordId;
        MATCHES FOR EQUALITY, ORDERING;

REGISTERED AS    {smi2AttributeID 3};

LogRecordId ::= SimpleNameType (WITH COMPONENTS {number PRESENT,
                                                 string ABSENT})

SimpleNameType ::= CHOICE {number  INTEGER,
                           string GraphicString}
```

Figure 3 The logRecordId attribute definition and ASN.1 syntax (from (4))

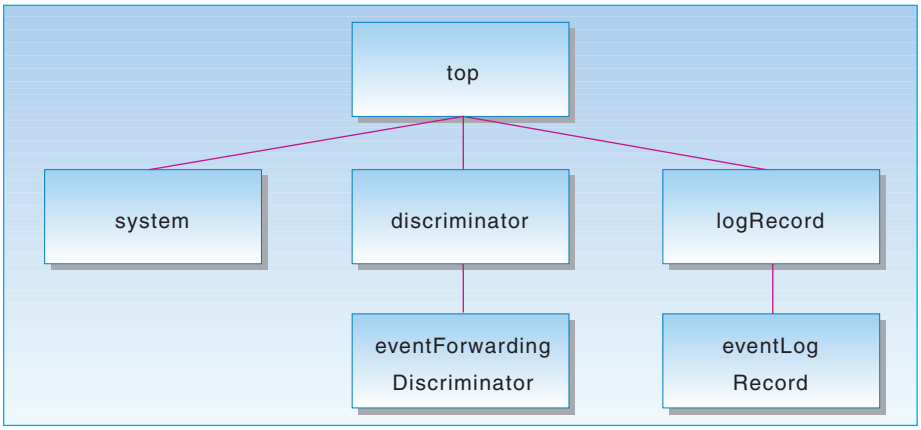


Figure 4 Illustration of an inheritance hierarchy (extract from (4))

occurred. Notifications, and the information they contain, are part of the managed object class definition. The notifications themselves are formally defined by use of a notification template, described in (5).

3.5 Behaviour

Behaviour is a description of the way in which managed objects or associated attributes, name bindings, notifications and actions interact with the actual resources they model and with each other. In particular, behaviour definitions are necessary to describe semantics, e.g. constraints, that are of dynamical nature.

As of today, behaviour definitions are documented by use of natural language text. This causes several problems. A given behaviour description may often be interpreted in different ways, and it cannot be automatically converted into executable code.

The use of formal description techniques for the specification of behaviour, e.g. SDL and Z, is under study.

4 Specialisation and inheritance

All managed object classes defined for the purpose of OSI management are positioned in a common inheritance hierarchy. The ultimate superclass in this class hierarchy is the managed object class “top”.

A managed object class is specialised from another managed object class by defining it as an extension of the other managed object class. Such extensions are made by defining packages that include some new properties in addition to those already present in the original managed object class. The new managed object class is a subclass of the original managed object class, which in turn is the superclass of the new class. The position in the inheritance hierarchy is included in every managed object class definition. The construct “DERIVED FROM top;” in Figure 2 indicates that the logRecord managed object class is specialised directly from “top”.

Some superclasses are defined solely for the purpose of providing a common base from which to specialise subclasses. These superclasses are never instantiated. Top is an uninstantiable superclass. (4) defines top and some other generic managed object classes from which subclasses are specialised. Figure 4 shows an example of an inheritance hierarchy.

A subclass inherits all the properties of the superclass, i.e. all operations, attributes, notifications, and behaviour. Within OSI management, only strict inheritance is permitted. Every instance of a subclass shall be compatible with its superclass, and specialisation by deleting any of the superclass properties is not allowed.

A subclass may be specialised from more than one superclass. This is termed multiple inheritance. Care must be taken to avoid contradictions in the subclass definition when it is specialised by multiple inheritance. If the same properties are present in multiple superclasses, these properties will only be inherited once.

5 Containment and naming

5.1 Containment

A managed object of one class may contain other managed objects of the same or different classes. This containment relationship is between managed object instances, not classes. A managed object may only be contained in one other managed object. A containing managed object may itself be contained in another managed object.

The containment relationship may be used to model physical or organisational hierarchies. Figure 5 gives an example of the latter.

A contained managed object may be subject to static or dynamic constraints because of its position in the containment hierarchy. If so, this must either be specified in the containment relation definition or in the class definition of the contained or containing managed object.

5.2 The naming tree and name bindings

The containment relationship is used for naming managed objects. A managed object is named in terms of its containing managed object. Contained and containing managed objects are referred to as subordinate and superior objects, respectively.

The combination of the superior object name and a Relative Distinguished Name (RDN) constitutes an unambiguous name of the subordinate object. The RDN uniquely identifies the subordinate object in relation to its superior object. This naming relationship is recursive, and all managed objects will have distinguished names that are formed by the sequence of RDNs of the object itself and each of its superior objects.

The complete naming structure within OSI management is a hierarchy with a single root. The hierarchy is referred to as the naming tree, and the top level is termed "root". Root is an object that has no associated properties, i.e. a null object, and it always exists.

When a managed object class is defined, rules have to be made for naming instances of that class. A name binding relationship identifies a superior object class from which instances of the managed object class may be named, and one or more attributes that may be used for RDN. Name bindings are not a property of the superior or subordinate managed object class. Multiple name binding relationships may be defined for a given class, and the managed object instances may use different name bindings. To ensure unambiguous naming, each managed object instance is permitted to have only one name binding.

Certain rules in connection with object creation and deletion are also included in the name binding definition.

Name binding relationships are formally specified by using a name binding template, described in (5). Figure 6 shows a name binding defined for the logRecord managed object class (and subclasses). The superior object class is the "log" managed object class (and subclasses). The attribute logRecordId is to be used for RDN. The name binding also includes a rule that prohibits the deletion of a logRecord managed object if it contains other managed objects.

5.3 Name structure

The name binding specifies which attribute to use for RDN. Certain requirements have to be satisfied in order to use an attribute for this purpose. The attribute must be included in a mandatory package, and it must be able to retain a fixed value through the lifetime of the managed object that uses it for naming. It must also be testable for equality.

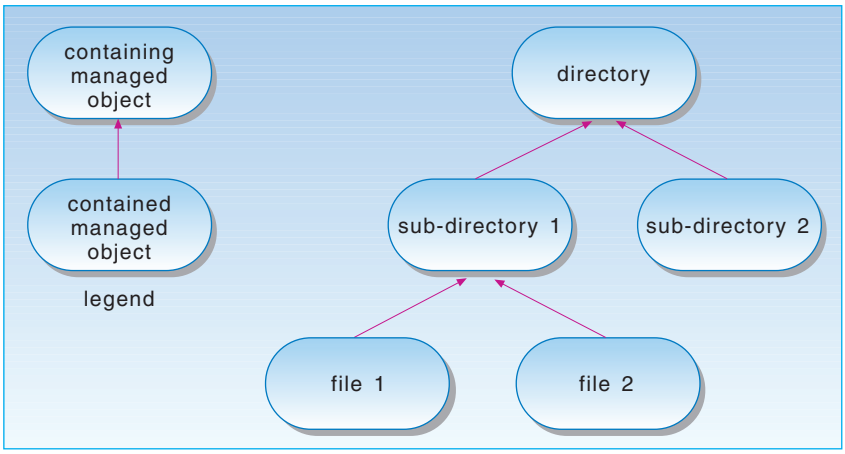


Figure 5 Illustration of containment relationship

```
logRecord-log   NAME BINDING
      SUBORDINATE OBJECT CLASS logRecord AND SUBCLASSES;
      NAMED BY
      SUPERIOR OBJECT CLASS log AND SUBCLASSES;
      WITH ATTRIBUTE   logRecordId;
      DELETE
      ONLY-IF-NO-CONTAINED-OBJECTS;
REGISTERED AS   {smi2NameBinding 3};
```

Figure 6 A name binding definition for the logRecord managed object class (from (4))

(4) defines the managed object class "system". A system managed object represents the managed system, and it has two attributes that may be used for naming, systemId and systemTitle.

Within OSI systems management, either global or local name forms may be used. The global name form specifies the name with respect to the global root. The local name form specifies the name with respect to the system managed object, which will be the top level in the local naming tree. The local name form does not provide identification that is globally unique, but may be used within OSI systems management.

6 Object identifiers

The definition of managed object classes, name bindings, actions, and notifications are identified by use of object identifiers. Package and attribute definitions need object identifier values if referenced in a managed object class definition. Once an object identifier value has been assigned, a definition must not be changed in any way that alters the semantics of the item defined.

Object identifiers are globally unique. The object identifier type is a simple ASN.1 type, defined in (6). An object identifier will be a sequence of integers, and object identifiers constitute a hierarchy termed the object identifier tree. Although this hierarchical structure is not related to either the naming tree or containment tree, the principle of allocating global object identifiers is analogous to the global naming of managed objects. All object identifier values registered in systems management Recommendations and Standards are allocated under the arc (joint-iso-ccitt ms(9)) in the object identifier tree. (5) defines the allocation of arcs below this level.

The construct "REGISTERED AS {smi2MObjectClass 7}" in Figure 2 assigns an object identifier to the logRecord managed object class definition. The object identifier smi2MObjectClass is specified by the assignment: smi2MObjectClass OBJECT IDENTIFIER ::= {joint-iso-ccitt ms(9) smi(3) part2(2) managed-ObjectClass(3)}.

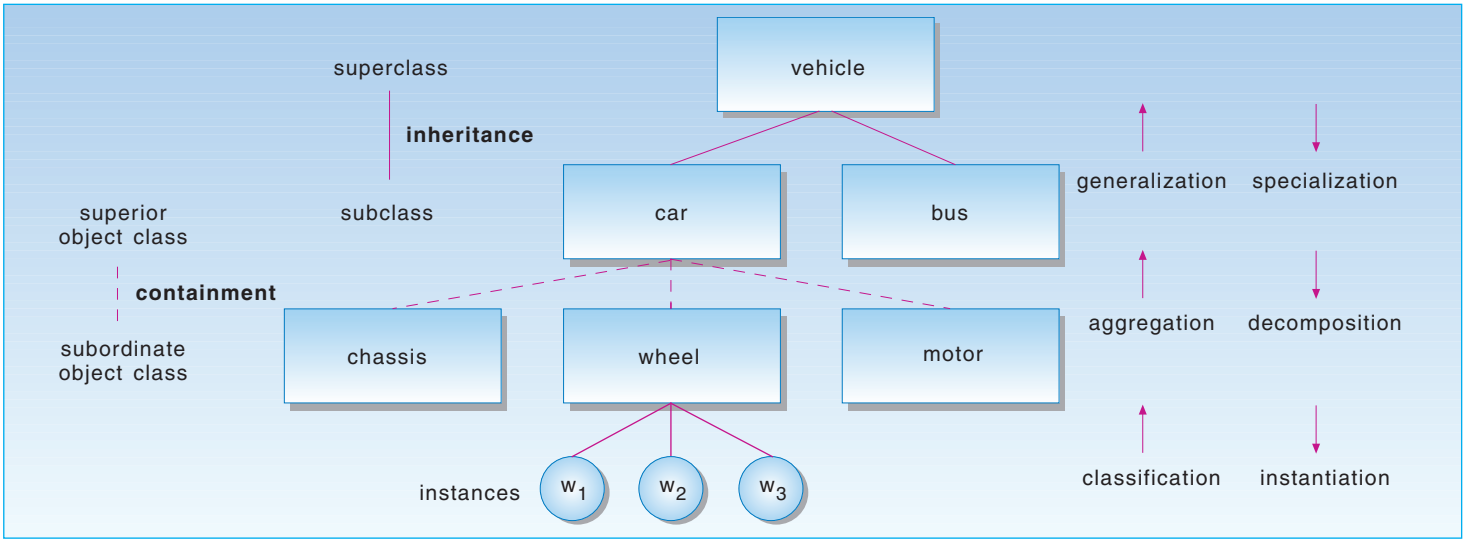


Figure 7 Illustration of terms for inheritance and containment (naming)

7 Compatibility and interoperability

7.1 Interoperability

When performing systems management, the managing and managed systems must have a certain common knowledge of relevant management information. This principle of shared management knowledge between managing and managed systems is described in (1).

Interoperability is required between managing and managed systems. For the management of a given managed object, this implies that a system must be able to manage another system if the managed system has equal or less knowledge of the managed object class definition of the object in question. To the extent feasible, a system must be able to manage another system even if the managed system has greater knowledge of the managed object class definition of the object in question. This interoperability must be maintained if the managed system is enhanced, or if managed object definitions are extended. Shared management knowledge may not be sufficient to ensure interoperability between systems.

7.2 Compatibility rules

Rules are defined for compatibility between managed object classes, and this contributes to attaining interoperability between the managing and managed systems. These rules ensure that a managed object being an instance of one managed object class (the extended managed object), is compatible with the definition of a second managed object class (the compatible managed object class). While this principle is used for defining strict inheritance, it may also apply to managed object classes that are not related by inheritance.

In connection with extending managed objects, rules for compatibility are defined for the following:

- adding properties in general
- package conditions
- constraints on attribute values and attribute groups
- constraints on action and notification parameters
- extensions to behaviour definitions.

The compatibility rules are described in (3).

7.3 Allomorphism

Allomorphism is a method the managed system may use to ensure interoperability. Allomorphism is a property of managed objects. It entails that a managed object may be managed as if it were an instance of one or more managed object classes other than its actual class. This is possible if the managed object supports allomorphism and is compatible with these other managed object classes.

An allomorphic class may be one of the superclasses of the managed object's actual class, but this is not required. On the other hand, compatibility is an absolute requirement for allomorphism.

If a managed object supports allomorphism, this must be considered when performing management operations and emitting notifications. Allomorphic behaviour, if any, is included in operation and notification definitions.

If the managed system does not provide the capability of allomorphism, a managed object will always respond according to its actual class definition. In this case, the managing system has to handle any unexpected or unintelligible information it receives from the managed system.

Some level of interoperability between the managed and managing systems may be achieved even if the compatibility rules are not satisfied. However, the degree of interoperability between the systems is closely related to the degree of compatibility.

8 Systems management operations

Management operations may be performed on managed objects if the operations are part of the managed object class definition. The effect of such operations on the managed object and its attributes, is also part of the definition. If an operation affects related managed objects, this will be specified as well.

There are two types of management operations; those that apply to the managed object attributes, and those that apply to the managed object as a whole. In order for an operation to succeed, the managing system invoking the operation must have the necessary access rights, and no consistency constraints must be violated. Such consistency constraints are part of the attribute or managed object class definitions.

Some operations are always confirmed, i.e. the operation invoker requires feedback on the result of the operation. Other operations may be confirmed or unconfirmed as selected by the operation invoker. The management operations referred to in sections 8.3 and 8.4 are primitive operations visible at the managed object boundary. For such operations, management has access to some information, even if the operation mode is unconfirmed.

Figure 1 illustrates the interactions taking place between manager and agent when performing management operations.

8.1 Scoping and filtering

Selecting managed objects for management operations involves two phases: scoping and filtering.

Scoping entails an identification of the managed objects that may be eligible for an operation. Scoping is based on the management information tree concept, which denotes the hierarchical arrangement of names of managed object instances in a MIB. The management information tree of a given system is identical to the local naming tree of that system. A base managed object is given for the scoping, and this object is defined to be the root of the (management information) subtree where the search is to start. Four different scoping levels are defined, the base object alone and the whole subtree being the two extreme levels.

Filtering entails applying a set of tests (filters) to the scoped managed objects. Filters are expressed in terms of assertions about the presence or values of certain attributes in the managed object. Attribute value assertions may be grouped together in a filter using logical operators. (3) describes the concept of filtering in more detail.

The subset of scoped managed objects that satisfy the filter is selected for the management operation in question.

Scoping and filtering is actually part of the management communications aspect of systems management. Facilities required to perform scoping and filtering are provided by CMIS (see chapter 10).

8.2 Synchronisation

When identical management operations are performed on several managed objects, synchronisation of these operations may be necessary. Two types of synchronisation are defined: atomic and best effort.

Atomic synchronisation implies that the operations shall succeed on all the managed objects selected for the operations. If that is not possible, the operations shall not be performed at all. Each operation must have a definition of success. Atomic synchronisation does not apply to the Create operation.

Best effort synchronisation implies, as the name indicates, that the operations shall be performed on the selected managed objects to the extent possible.

Operations synchronisation is actually part of the management communications aspect of systems management. Facilities required to perform operations synchronisation are provided by CMIS (see chapter 10).

8.3 Attribute oriented operations

Attribute oriented operations are defined to be performed on the managed object, not directly on the attributes. This enables the managed object to control any manipulation of the attribute values, and thereby ensure internal consistency.

The managed object receiving the operation request must determine if and how the operation is to be performed. This is usually done by filtering, based on information given in the operation request.

Attribute oriented operations operate on a list of attributes. This implies that all the attributes included in the attribute list of an operation request will be part of a single operation. The operation succeeds if it is performed successfully on all attributes in the list. After the operation, relevant information on attribute identifiers and values, and any error indications, will be available to management at the managed object boundary.

Because of existing relationships in the underlying resources, operations performed on attributes in a managed object may have indirect effects. Attributes within the same managed object may be modified, and attributes in related managed objects may be affected as well. The behaviour of the managed object may change as its attributes are modified, and so may the behaviour of related managed objects.

The following attribute oriented operations are defined:

- Get attribute value
- Replace attribute value
- Replace-with-default value
- Add member
- Remove member.

Descriptions of these operations are given in (3).

8.4 Object oriented operations

As opposed to attribute oriented operations, some management operations apply to the managed object as a whole. The effect of these operations is generally not confined to modification of attribute values. The following primitive operations are defined:

- Create
- Delete
- Action.

The Create operation requests the creation and initialisation of a managed object instance. The Delete operation requests the managed object to delete itself, and may also entail the deletion of contained objects. The Action operation requests the managed object to perform some specified action and to indicate the result.

The managed object class definition shall include the semantics of these operations. Interactions with related managed objects shall be specified, as well as the effects the operations may have on the resource being represented.

Detailed descriptions of these operations are given in (3).

9 Notational tools

The definition of managed object classes, packages, attributes, etc., have to be formally documented. (5) defines a set of templates for this purpose. The following templates are defined:

- managed object class template
- package template
- parameter template
- name binding template
- attribute template
- attribute group template
- behaviour template
- action template
- notification template.

The templates define the constructs that are to be included, mandatory or conditional, and the order in which the constructs shall appear. The definitions shown in Figures 2, 3, and 6 are made according to the templates defined for managed object classes, attributes and name bindings, respectively.

If attributes, operations, notifications, etc., are defined by the use of their own templates, and are assigned separate object identifiers, they may be globally referenced in other managed object class definitions. In order to increase commonality of definitions, possible re-use of already defined managed object properties should always be considered when defining new managed object classes.

10 CMIS and CMIP

The Common Management Information Service (CMIS) is an application service used by application processes to exchange systems management information and commands within a management environment. Systems management communication takes place at the application layer, ref. (7).

The CMIS definition includes the set of service primitives constituting each service element (CMISE), the parameters that are passed in each service primitive, and the necessary semantic description of the service primitives.

As indicated in Figure 1, there are two types of information transfer, namely operations and notifications. CMIS provides services accordingly; a management operation service and a management notification service. In addition to this, CMIS provides facilities to enable linking of multiple responses to a given operation. It also enables scoping, filtering, and operation synchronisation as described in sections 8.1 and 8.2. It may be noted that CMIS does not provide operations synchronisation across multiple attributes within a managed object.

The CMISE services are listed below:

- M-EVENT-REPORT
- M-GET
- M-SET
- M-ACTION
- M-CREATE
- M-DELETE.

Establishment and release of application associations are controlled by using ACSE, defined in (8).

(9) defines CMIS, CMISE and the associated service primitives and parameters.

The Common Management Information Protocol (CMIP) is a communication protocol used for exchanging management information between application layer entities. CMIP supports CMIS. Information on CMIP and associated subjects is found in (10).

11 ASN.1

Instances of attribute values are carried in management protocols. In order to interpret these values correctly, their syntax has to be formally defined. Abstract Syntax Notation One (ASN.1) specifies a collection of data types for this purpose. In addition to defining the type of the attribute value, the ASN.1 data type will define the type of the attribute itself, e.g. if the attribute is single-valued or set-valued. Once an ASN.1 type has been assigned to an attribute, instances of this attribute will be assigned values of that given type.

ASN.1 data types may be specified in connection with parameters, notifications and actions as well. The ASN.1 type of a parameter may either be specified directly or by referring to an attribute whose syntax is used. The syntax of information and replies associated with actions and notifications may be specified directly by including information syntax and/or reply syntax in the definition of a given action or notification.

The topic of ASN.1 is rather extensive, and deserves a paper of its own. Reference is given to (6) for details.

12 Abbreviations

ACSE	Association Control Service Element
ASN.1	Abstract Syntax Notation One
CCITT	The International Telegraph and Telephone Consultative Committee
CMIP	Common Management Information Protocol
CMIS	Common Management Information Services
CMISE	Common Management Information Service Element
ETSI	European Telecommunications Standards Institute
IEC	International Electrotechnical Commission
ISO	International Organisation for Standardisation
MIB	Management Information Base
MIS	Management Information Services
OSI	Open Systems Interconnection
RDN	Relative Distinguished Name
SDL	Specification Description Language.

References

1 CCITT. *Recommendation X.701,* 1992.

2 CCITT. *Recommendation X.700,* 1992.

3 CCITT. *Recommendation X.720,* 1992.

4 CCITT. *Recommendation X.721,* 1992.

5 CCITT. *Recommendation X.722,* 1992.

6 CCITT. *Draft Recommendation X.208-1,* 1992.

7 CCITT. *Recommendation X.200,* 1988.

8 CCITT. *Recommendation X.217,* 1988.

9 CCITT. *Recommendation X.710,* 1991.

10 CCITT. *Recommendation X.711,* 1991.

Network management systems in Norwegian Telecom

BY KNUT JOHANNESSEN

621.39.05:65

Abstract

This article provides an overview of software systems for network management in Norwegian Telecom. A few words are needed to explain what is meant by network management.

The digitalisation of the telecommunication network has introduced a large number of processors into the network. In many cases the processors constitute full-blown complex computers. The processors detect, process, communicate, and implement status, alarm and control information to and from the network. This information is managed in a separate set of computers and software applications outside the network itself. These supporting systems we call Operation, Management and Administration Systems (OM&A systems), or for short network management systems.

In this article we will concentrate on applications which support the management of the network. Below we will explain how we delimit the scope of this presentation. Norwegian Telecom currently has a long range of software systems for the management of the network. Most of these are closely related to the network elements, e.g. System 12 exchanges from Alcatel and AXE exchanges from Ericsson. Many of these systems have functions for the handling and configuration of equipment, while the applications developed by NT itself are more concerned with the integrative management of components and services. This comprises charging, performance monitoring, alarm management, and administrative databases about resources and couplings in the network.

Introduction

Norwegian Telecom has currently several network management systems.

In this article we will delimit the presentation to technical support systems outside the network elements. Also, basic systems to undertake and support communication to, from and between network management systems are left out. Systems on Personal Computers, e.g. network planning systems, are left out as well. Finally, reporting and statistics production systems are excluded. The list is delimited to systems considered to be commonly used or recommended by the Networks Division of Norwegian Telecom. This means that systems used by other divisions or business units are not covered.

The network management systems are grouped into the categories common systems, access and transport network, telephony and ISDN, service network, and a category of other networks. This categorisation is not officially recognised, but corresponds loosely to the internal organisation of the Networks Division and helps to indicate what technology is managed. The systems are listed alphabetically in each category, independently of size or importance.

From a functional point of view, network management is often divided into four levels: network element, network, service, and business. We will only consider the first three of these. The first level is typically concerned with alarms and aspects of one network component. The second level is concerned with resource utilisation, resource management and the topology of the network as a whole. The third level single out these aspects for a single service or customer.

Table 1 provides an overview of network management systems. The overview uses the three levels presented above, and uses a sub-categorisation of five application areas which are commonly known as functional areas of OSI management.

Common systems

Common systems are systems which support several of the categories outlined in the introduction.

AK is an alarm and command system which collects operation and environment alarms and communicates information for remote control of telecommunication equipment. The AK systems also conveys alarms to alarm

Table 1 Classification of systems

	Fault	Configuration	Accounting	Performance	Security
Service	INFOSYS	INFOSYS INSA TELSIS-LK NUMSYS		MEAS	INFOSYS
Network	DGROK INSA TMS	DGROK INKA INSA NMS/DXX INTRA NETCON-D TELSIS-LK		AUTRAX MEAS TVP TVPM	
Network element	AK ALARM-S12 BSCOPT/SSC CU/PDA CMAS DKO FBS/DIGIMUX INFOSYS MCC (DEC) NETCON-D NMAS NMS (Sprint) OPENVIEW REVAL SLMS STRAX SSC TDT2 TMS TPRF3 TREF	AK CMAS FBS/DIGIMUX INFOSYS INRA CMBAS MCC (DEC) NETCON-D NMAS NMS/DXX NMS (Sprint) NUMSYS OPENVIEW RLPLAN SALSA SMAS SYMKOM STAREK STRAX	FILTRANS NESSY NMAS NMS (Sprint) OOBS/DEBO SMART SPESREGN TCA TELL VISDA	AOE ATME AUTRAX CMAS DKO LV/Measurement MCC (DEC) NESSY NMAS OPENVIEW SKUR	NMAS SMAS

centres and security corporations (the Al-Tel service). The AK system is implemented on Norsk Data Computers, ND-100.

INSA is a database application for the administration of the long lines and trunk network. A relatively new usage of INSA is the integrated analysis of alarms from various network elements and graphical presentation of possible causes, resulting consequences and rerouting alternatives. INSA runs on IBM mainframes and graphics is provided on both PCs and Unix workstations.

INTRA is a database application for the administration of equipment and cross couplings inside transmission and switching centres. INTRA runs on IBM mainframes.

TMOS is a family of Telecommunication management operation support systems from Ericsson. Norwegian Telecom has installed three sub-systems: NMAS, SMAS, and CMAS. These are shortly described in subsequent sections. TMOS runs under SunOS on Sun computers.

TMS, Telecommunication Management System, is undertaking integration of alarms from various sources (System 12, MTX and AK). The most important function is alarm analysis provided together with INSA.

Access and transport network

The access and transport network comprises the common infrastructure of the network. Most of the support systems in this category are test and supervision systems. These are based on measurement equipment close to the network elements and central systems to manage the measurements. Measurement equipment and central equipment are often connected by the circuit switched DATEX network. The measurement equipment is not included in the list provided below.

ATME is Automatic transmission measuring equipment for measurement of the quality of the long lines network. The system runs on ND-100 computers in trunk exchanges all over the country.

BSCOPT provides graphical and alphanumerical presentations on PCs from a support system for radio links.

CU/PDA is a PC-based system for the identification of failures of 64 kbit/s DIGITAL circuits. The system is used to create test loops on all parts of a circuit.

DGROK is a system which can switch high level multiplex systems to a reserve. This is a centralised system running on a VAX computer.

DKO provides supervision of 140 Mbit/s multiplex systems on radio links and optical fibres.

FBS/Digimux provides alarm information, reconfiguration and tests of subscriber modems in the Digital multiplexing network. The system runs under SunOS.

INKA is a database application providing topographic maps about the cable network. INKA runs on AIX.

INRA is a database application providing data about radio link and broadcasting stations and is communicating Autocad drawings about these stations. The system runs on IBM mainframes.

LV/Measurement equipment is used in some regions for remote control of subscriber line measurements and tests of subscriber modems. These measurements do not include lines covered by SLMS or measurement systems inside the network elements.

NETCON-D/DACS provides alarm and quality supervision and configuration control of the DACS network. DACS is a first-generation digital access and cross-connect equipment. The system runs on PC user terminals connected to an OS/2 server.

NMS/DXX provides alarm supervision and configuration control of the DXX network. DXX is a second-generation digital access and cross-connect equipment. The system uses both OS/2 terminals and an OS/2 server.

RLPLAN is a database application about the positioning of and equipment in radio stations and provides information about channels and frequencies. The system runs on IBM mainframes. Graphical presentations of height curves is provided by the Diman tool.

SSC, System and station control equipment, collects and presents alarms from country wide radio links and broadcasting stations. The system runs on NEC computers.

STAREK is a database application containing data about broadcasting stations. The system runs on IBM mainframes.

TELSIS-LK is a database application about the subscriber network. TELSIS-LK is a subsystem of the TELSIS system and runs on IBM mainframes.

Telephony and ISDN

Telephony and ISDN belongs to the category service network, but is treated separately, due to the size of the area.

Alarm-S12 is a system for presenting alarm information for all Alcatel S12 exchanges controlled by a Network Switching Centre. The system runs on ND-500.

AOE is used to transfer data from AUTRAX to Norsk Data computers. The system runs on ND-100.

AUTRAX, Automatic traffic recording terminal, provides traffic measurements on analogue exchanges, both for operation and maintenance and production of statistics. The data are transferred to MEAS on Norsk Data computers.

FILTRANS is a set of small software systems that are used for communication and conversion of data files from Ericsson AXE phase 0 exchanges. FILTRANS utilises basic functions in NMAS release 1.

MEAS provides measurements and statistics for operation and planning. Currently the system runs on Norsk Data computers.

NESSY administrates file transfer between S12 and its support systems. The system runs on ND-500.

NMAS, Network management system, undertakes alarm, command and file management of AXE and S12 exchanges. Norwegian Telecom has developed some minor systems to interconnect NMAS with existing administrative systems, such as RITEL for billing.

NUMSYS is a database application which manages subscriber data to S12 and AXE phase 0 exchanges. The system runs on ND-500 computers.

OOBS/DEBO, Originating observation system/Detailed billing observation, is used to handle bill complaints related to S12 exchanges. FILTRANS is used for the same function related to AXE phase 0 exchanges. SENTAX will be used for ISDN.

REVAL, Report evaluation, is used to ease the understanding of reports from

S12. The system runs on ND-500 computers.

SALSA is a system for inputting subscriber data to S12 and AXE exchanges and for the planning of cabling. The system runs on SunOS.

SKUR provides control of registers in analogue exchanges. The data are analysed on ND-100 computers.

SLMS is a Subscriber line measurement system for exchanges which do not have built-in functions for this.

SMART, Subscriber monitoring and registration terminal, is used to control billing information from analogue exchanges. The system runs on ND-100.

SPESREGN, Specified billing, converts billing information from S12 format to RITEL-FAKTURA, the common billing system for Norwegian Telecom. SENTAX will replace SPESREGN for S12 package 4. FILTRANS and a conversion program converts AXE data to S12 format. SPESREGN runs on ND-500.

SYMKOM is a new system which converts data from SKUTE (a new system for the management of customer data) to Man-Machine Language commands towards S12 and AXE. SYMCOM runs under UNIX.

TCA, Telephone call analyser, registers data from analogue exchanges for the handling of billing complaints. The presentation part of the system runs on ND-100.

TELL is used to supervise abnormal changes in billing data. Currently this system is to a large extent replaced by TURBO-TELL and SUPER-TELL. Later these will be replaced by SENTAX. The system runs on ND-500.

TREF is a system for registration and handling of failures on subscriber lines connected to both analogue and digital exchanges. TREF has interfaces to a long range of other computer systems. The system runs on ND-500.

VISDA is used to collect large amounts of data from S12 and package 1-3 and AXE phase 0. VISDA will be replaced by SYMKOM for S12 package 4. The system runs on ND-500.

Service networks

Service networks is a category of networks and equipment which provide either a single or a family of similar services (e.g. X.25 data communication).

CMAS, Cellular management system, is a subsystem of TMOS from Ericsson and is used for network management of the GSM mobile network.

INFOSYS provides customer access to information about the Datapak and the Datex networks. The system is implemented under IBM AIX.

MBAS is a database application about base stations in the mobile network. The system runs on IBM mainframes.

NMS, Sprint network management system, provides network management of network elements which provide interconnection between the ISDN network and Datapak.

STRAX provides alarm information mainly from the Datex, Telex and Mobile network exchanges in the Oslo region. The system runs under IBM AIX.

TDT2, Telenet diagnostic tool, is used to monitor and control status, configuration and performance of Datapak exchanges. The system is implemented on Prime computers.

TPRF3, Telenet processor reporting facility, distributes alarms from the network elements of the Datapak network via a national centre to regional centres.

TVP is used to supervise the traffic congestion and quality of service in the Datapak packet switched network. The system consists of several remote units connected to a central. Statistics is generated on an ND-100 computer.

TVPM is a mobile network version of TVP, mentioned above.

Other networks

This category includes networks and systems not covered by previous categories.

HP Open view, Network node manager, is used for the management of internal ARPA-IP networks.

MCC from DEC uses the Simple Network Management Protocol to manage broadband Supernet interconnecting universities in Norway. The system runs under Unix.

SMAS, Service management network, is a subsystem of TMOS from Ericsson and is used for network management of Intelligent Network services. The system also allows assembling of service components to create more complex services. The system runs on SunOS.

Centralised network management

BY EINAR LUDVIGSEN

621.39.05:65

Abstract

This article gives a presentation of key tools for efficient centralised network management in Norwegian Telecom. The following measures are identified as prerequisites for obtaining intelligent network management:

- *a realistic data model of the network*
- *a detailed database of the network*
- *analysis and derivation routines on the network data*
- *automatic/semi automatic graphics from the network database.*

Norwegian Telecom has implemented intelligent network management through the following software tools:

- *the INSA database application for administering long lines and trunk networks, including alarm handling and the provisioning of graphics*
- *the INTRA database application for transmission equipment and cross couplings inside transmission centres, including related applications*
- *the DATRAN tool for efficient development and usage of database applications*
- *the DIMAN tool for presenting, manipulating and communicating complex graphics to remote workstations via narrowband wide area networks.*

Situation assessment

The Telecommunications Management Network interface recommendations are regulatory rules for the outline of the Network Element boundaries to the Operation, Administration and Management systems. Also recommendations of interfaces for the interchange of data between applications are being developed. However, the interface recommendations do not contain a design of the OA&M systems themselves and do not currently recommend their functioning – neither their detailed software architecture, nor their concrete human-machine interfaces. So far, the OA&M systems have to be designed and developed by the telecommunication operators or vendors themselves. The need to adjust the OA&M systems to national terminologies, existing administrative systems, organisation dependent regulations, competitive functions, etc., can for a long time imply a need to adjust the OA&M systems to the particular operator's organisation.

The TMN interface standardisation will restrict the Network Element vendors to compete on price, quality and service, while the technical specifications for the NE's are rather fixed. However, the same means will provide a preparation and opening up of the market for generalised or tailored OA&M systems.

INSA	
End users:	1620
Number of databases:	1
Database size (bto):	1.5 Gb
Data model - objects:	87, implemented through 207 record types
- relations:	312
- attributes:	2000
Screen pictures:	ca. 100

INTRA	
End users:	825
Number of databases:	7
Total database size (bto):	2.7 Gb
Data model - objects:	73, implemented through 96 record types
- relations:	214
- attributes:	570
Screen pictures:	ca. 50

Figure 1 Key figures for INSA and INTRA
DATRAN allows for repeating attribute group(s). This is the main reason why 207 record types are needed to implement 87 object classes in INSA. DATRAN allows screen pictures which are larger than the screen in both the horizontal and the vertical dimensions. Some empty screen pictures in INSA can take up 6 full screens each. Also, different kinds of editing is permitted in the same screen picture. This is why 2000 attributes can be presented in different ways in several screens using 100 screen pictures only. Generic databases for dictionary, access control, electronic main and graph management are not included in the above figures. The figures provide some insight into the complexity of the INSA and INTRA applications

Intelligent Network management

The most important resource for intelligent network management is a networked database of all resources and connections in the telecommunications network. Such a database will provide means to

- reason about the causes and consequences of alarms and failures in the network and provide rerouting alternatives
- automatically present different geographic or schematic graphs of the network, showing its current constituent structure, plans, routing alternatives, sub-networks, etc.

A prerequisite to achieving intelligent network management, is a database that is a realistic model of the network itself, so precise that it can replace the network for all reasoning purposes. Some computer vendors have these kinds of tools

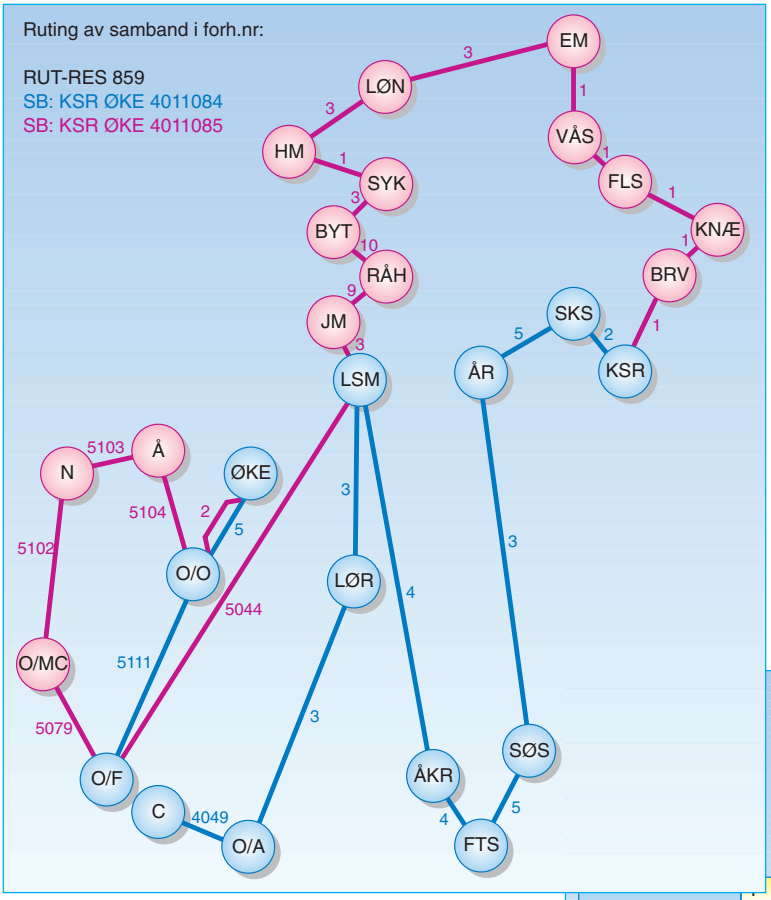


Figure 2 Routing of two circuits, required to have different routing, between two end stations
The graph is provided automatically from the INSA database by the use of DIMAN. The end user can move icons and otherwise edit and transform the graph without destroying the connections and references to the source data in the INSA database. Manual production of these graphs by using a commercial drawing package would have been unfeasible

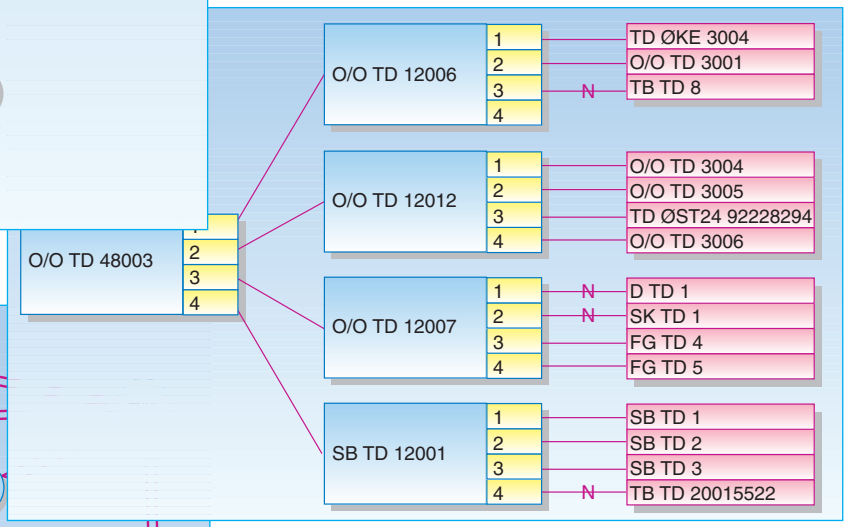


Figure 3 Part of a multiplex hierarchy
The graph is provided automatically from the INSA database using DIMAN. The end user can edit the graph and get the old editions included in new versions, which are produced automatically and contain new data. This way, DATRAN and DIMAN integrate manual editing and automatic production of graphs

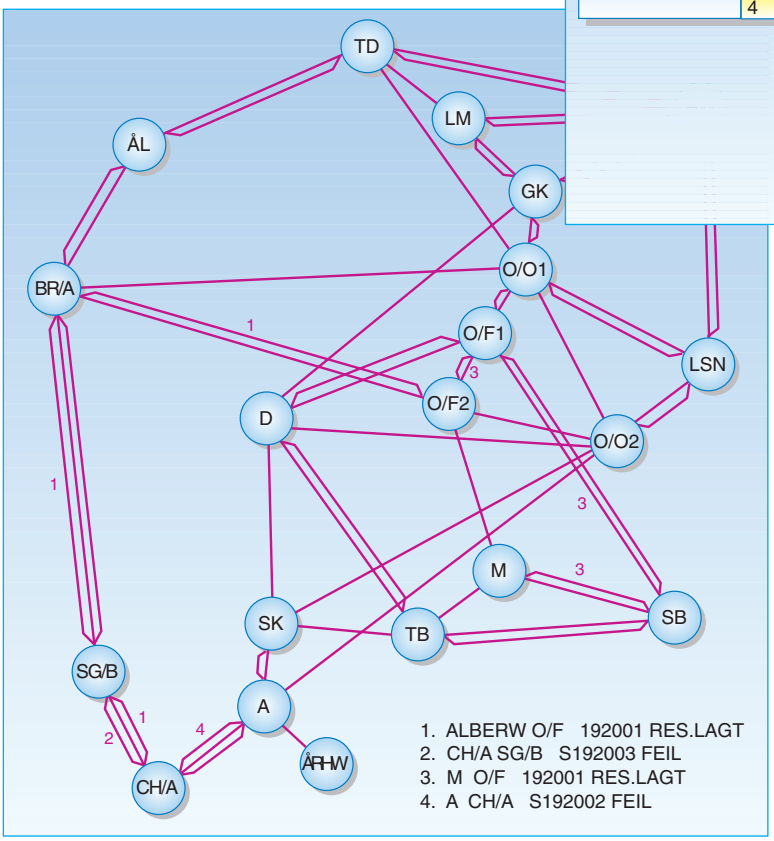


Figure 4 The DGROK network
The DGROK network allows automatic re-routing of digital multiplex groups. The graph shows free, occupied, fault, planned and reserved resources and is automatically updated in real time when the state of a resource is changed. The graph is provided automatically from the INSA database using DIMAN

for computer networks. However, our experience is that public telecommunications networks, have a much more complex structure, due to the use of very high level multiplexing, diverse technologies and complex administrative procedures and relations to other systems. For example, existing files for the cable network and the radio link network often tend to serve as references to detailed paper based graphs and plans rather than being full blown models of the network. Therefore, in the INSA project, Norwegian Telecom developed a precise data model of the long lines and trunk network in Norway for all technologies, including satellites, radio links, cables, multiplexers, switches, circuit groups, circuit routing, data networks, mobile communication networks, rerouting plans, etc.

The INSA data model forms the basis for the INSA database for the long lines and trunk network for all of Norway. While the INSA database is used to administer the resources between the transmission centres, the INTRA database – implemented by the same tools – has been added to administer equipment and cross couplings inside each transmission and switching centre.

INSA also includes an order management system, allowing the operators to issue the appropriate routing and cross coupling orders to the local technicians and to get reports of execution back. The order management system covers both the INSA and INTRA system. In addition to an electronic document interchange-part (EDI), the order management system contains a historical database. In this database information about how resources were disposed or how circuits were routed before a particular routing and cross coupling order was executed, can be obtained.

The reporting back routines in the order management system automatically updates the INSA and INTRA databases. By means of this, the databases show at any time a real picture of the long lines and trunk network, which is utilised in Network Management.

Currently commands to the network itself cannot be issued automatically from the INSA system; whenever appropriate, such commands have to be keyed into a dedicated system for this purpose.

The INSA and INTRA databases have become key tools for administering the long lines and trunk network in Norway in an efficient way. However, more benefits are provided by creating add-ons to these database applications.

Automatic graphics

Many administrations, including Norwegian Telecom, have been developing schematic graphs of the network – often by using commercial PC tools. However, one recurrent problem is to keep the graphs updated as the resources of and the routing in the network are changing. This is solved in INSA by providing a graphical interface to the network database itself, where the graphs are automatically updated at the users premises. The user can even modify the graphs manually and keep these modifications maintained when the database is updated. We would like to point out that in most graphs of the network, we do not want a pure geographic presentation, rather a more schematic and readable presentation of the structure of the network is wanted, and still we want the graphs to be produced automatically. INSA does this.

INSA automatically produces a large variety of schematic graphs of the multiplex hierarchy, personal communication network, leased lines networks for large customers, routing of circuits, circuit groups, etc. for the whole or a specified part of the network.

Real time analysis

The above measures, in the following descending priority,

- a realistic data model
- a detailed database
- analysis and derivation routines
- automatic graphics

are needed to provide intelligent management of the network. For real time analysis of the network, in addition, a data communication facility between the Network Elements and the network database is needed. This data communication facility may be a recommended TMN interface or various proprietary solutions, for which adaptions have to be made.

Various alarms from Network Elements – from switches and pieces of transmission equipment – are automatically fed into the INSA database application according to the users' selections and an automatic analysis procedure is started. Then the users initiate a presentation of the selected alarms in a map of the involved resources. A next graph shows common resources for the alarms, which are candidate sources for the primary alarms and failures. The last step in the analysis shows rerouting alternatives to the involved resources. The entire analysis and graphical presentations can take place in a few minutes. And the initiation of the analysis and graphical presentation of the results can take place at remote terminals all over the country. The users can also initiate an analysis procedure to detect the consequences of the alarms. Instead of a graphic presentation, the user can initiate a paper list presentation. This presentation can be focused either on leased lines customers, or on the various service networks. In addition to feeding alarms into the INSA system, the consequences of planned outage of a transmission system can be detected by the same analysis. This gives us the possibility to point out a suitable period to do necessary maintenance on Network Elements.

Past and present

Without these tools, the OA&M operators previously spent hours and days investigating the sources of the alarms, and taking the necessary actions. The knowledgeable OA&M reader is aware of the complexity of the tasks involved. If a large transmission centre has a blackout, tens of thousands of circuits can be involved, making it just about impossible to sort out all parties involved and to create satisfactory alternative solutions in a short period of time. If a radio link to the western part of the country is involved, this can cause fallouts also in the northern and southern part of the country, due to complex multiplexing and routing. Often the reserves for the sub-groups can be unknowingly routed on the same fallen out link. And often we are not able to spot manually all rerouting alternatives existing as many step paths of free resources in the network. Therefore intelligent network management facilities are needed.

Norwegian Telecom has made attempts to install commercial tools for automatic presentation of alarms in pre-made graphs. The problems with these tools have been the manual updating of the graphs and the lack of intelligence in the analysis, due to the lack of access to the network database itself. Furthermore, neither initiation nor utilisation of the analysis were easily distributed all over the country. The INSA system provides all this.

Future

A further development of the alarm handling, in the INSA system, will be the development of a method to link detected alarms to registered transmission systems in the INSA database. This gives us a possibility to produce statistics, telling us something about the reliability and availability of the different transmission systems.

In an earlier paragraph we described the analysis which led to the detection of consequences of alarms from network elements or planned outage of a transmission system. This consequence analysis makes it possible for us to inform important customers automatically when their connections are affected by network failure or planned outage. In principle, all customers having leased lines or data connections, can be informed. Norwegian Telecom has, however, decided that only strategic customers, that is, customers with a particular service contract, will receive this kind of information.

The analysis starts with the alarms fed into the INSA database. The results of the analysis are distributed to the customers via the TELEMAX-400 network to a telefax machine, or to a PC. A test project, with very few (2 -3) customers involved, was started in December 1992.

In the digital multiplex network, Norwegian Telecom has introduced a Digital Rerouting System (DGROK). Today, when reserves for sub-groups are activated, the INSA database has to be updated by an operator to obtain a correct result of the consequence analysis. In the future, an automatic updating of the INSA database from the DGROK system will be developed.

The DATRAN and DIMAN tools

BY CATO NORDLUND

681.3.01

Background

DATRAN is a general application which easily can be tailored to reading and writing in different databases. DATRAN has been used to replace traditional development for several large and complex applications. Each time costs and development time have been considerably reduced.

DIMAN is a dialogue manager for alphanumeric and graphic dialogues. This provides an advanced graphical front-end to DATRAN applications.

This article gives an overview of the mentioned products as well as their usages.

The work with DATRAN started back in the late seventies and was from the beginning connected to the INSA project within Norwegian Telecom. The task was to develop a system for the administration of the long lines and trunk network, a very complex application, with all kinds of telecommunication technologies involved. The system was to be on-line all the time, and its data should be updated from terminals all over the country. Experiences from several projects developing similar systems, led to the conclusion that enough EDP personnel was not available to solve the task in a traditional way. Hence an activity to develop the necessary tools to reduce significantly the programming and testing efforts was established. This work was conducted by Norwegian Telecom Research.

Even at that time software products for applications like reporting, salaries, and accounting were available. Also more general programs for word processing and text editing existed, but tools for application development were scarce and poor.

However, almost all programs in an on-line database application do very much the same: read some data from the database, present them to the user, receive input, validate it, update the database and display some sort of confirmation. Complexity is added due to access control, where the program must consider user categories versus function and data ownership, and also when data is gathered through several forms to assure completeness before updating. Much of the effort is spent to write and debug code eventually not to be used, write and debug similar code for slightly different cases, propagate corrections and improvements into earlier written code, make user interface convenient and consistent, and so on.

If one could make one product doing all these operations, controlled by parameters or a specification of the application, most of the repetitive and time consuming work could be removed from the system developing process. Also, since the same code can be used by all applications, the quality will increase for all users. In addition, the same common user interface will be provided for all applications, and thereby greatly reduce the need for user education concerning the EDP system. This is of course the same advantages as acquired for all generalised software, but which is unfortunately not so well focused in on-line database applications.

And DATRAN was developed according to these guidelines. The name "DATRAN" is an abbreviation for "DAta TRANsformation", which accentuates the main idea of the product. Data are read from one medium and written on another, and the whole process is controlled by rules specific for that particular transformation. The rules are stated in application specific schemata and are removed from the generic transformation program. The contents of the schemata are treated as ordinary data and are themselves managed in an ordinary DATRAN database.

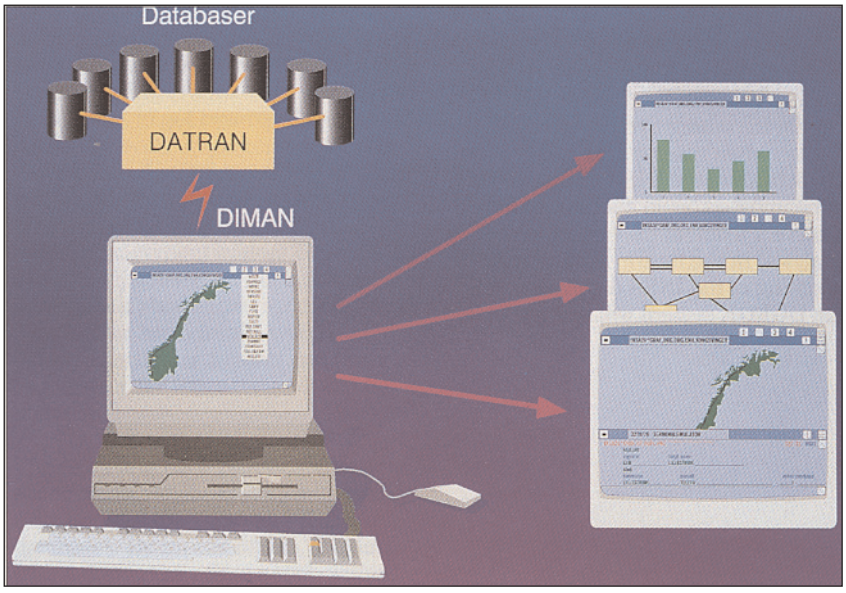


Figure 1 DATRAN and DIMAN
DATRAN manages data, enforces data structure and application logic on central IBM mainframes. Presentations can be made on dumb terminals, PCs or workstations. DIMAN provides a seamless dialogue of alphanumerical and graphical presentations

DATRAN

DATRAN is based on a layered architecture, commonly referred to as a 3 schema architecture. Most important is the application layer. Here the application is defined, with objects, related objects, attributes and their valid values. The names used in the application layer are from the user terminology, not the programming or database terms. Often the application layer is called a data model for the application, but what it really is, is the complete set of names and restrictions for the application system. As these definitions are readily available for users as well as system developers, they are often accompanied by textual explanations not used by DATRAN itself.

The internal layer of the architecture contains the database entities. That is areas, records, sets, items, table spaces, tables, columns, keys, programs and so on, depending on the chosen database system for storing the application data. Much of the content in the internal layer can be generated automatically from the application layer, but there will always remain some choices and implementations to be made by the database administrator. The application layer is

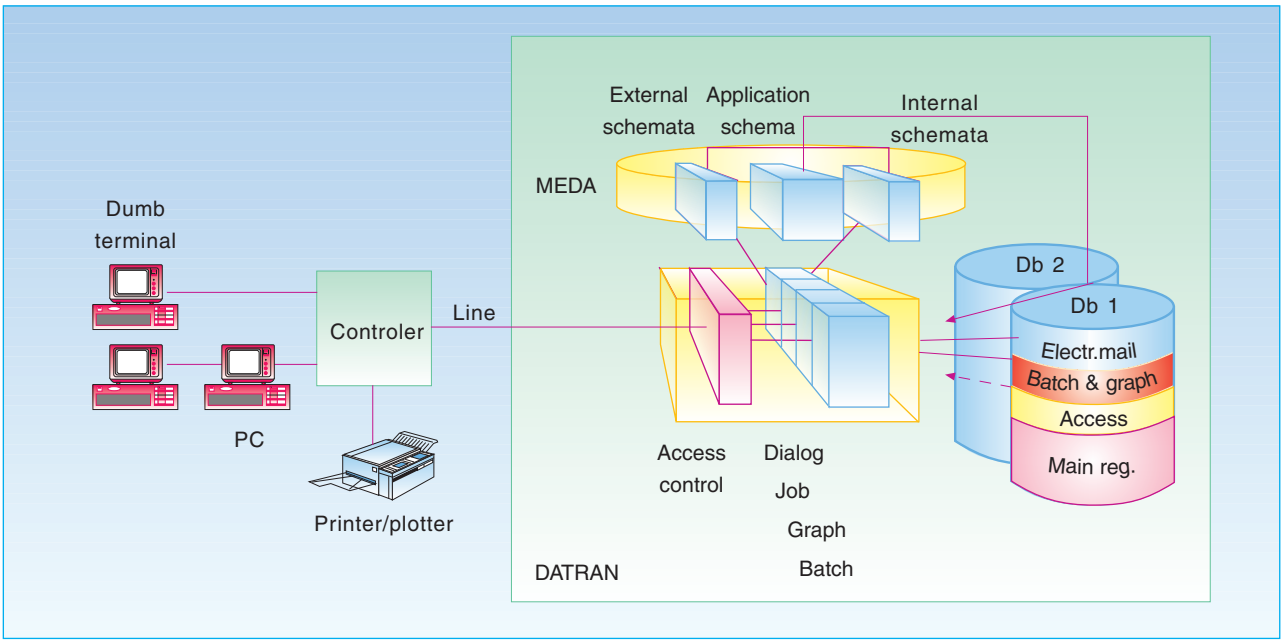


Figure 2 The DATRAN environment
DATRAN is controlled by specifications of the application, contained in the 3-layered MEDA dictionary. The layers contain External, Application and Internal schemata. All implementations (of control structures) are hidden from the developer inside DATRAN itself. The Dictionary is an ordinary DATRAN application. DATRAN contains an advanced access control system. Also, DATRAN contains pre-made subsystems for dialogue handling, job control, graph management and batch processing. Pre-made databases are provided for the dictionary, electronic mail, graph management and access control

mapped to the internal layer, so that every entity in the application layer has a representation in the internal layer. The internal layer specifies how the application data are stored.

The screen definitions of the system constitutes the external, or better – presentation layer of the application. Each of the presentation forms consists merely of a list of objects, related objects and attributes from the application layer. However, the entities cannot be chosen arbitrarily, they must form a subset of what is previously defined in the application layer. If a data item is allowed to be updated through that particular form, it is provided with an operation code for insertion, modification or deletion. The developer needs not bother with how the operations are implemented. This way of realising a screen picture, approaches a theoretical minimum of effort involved.

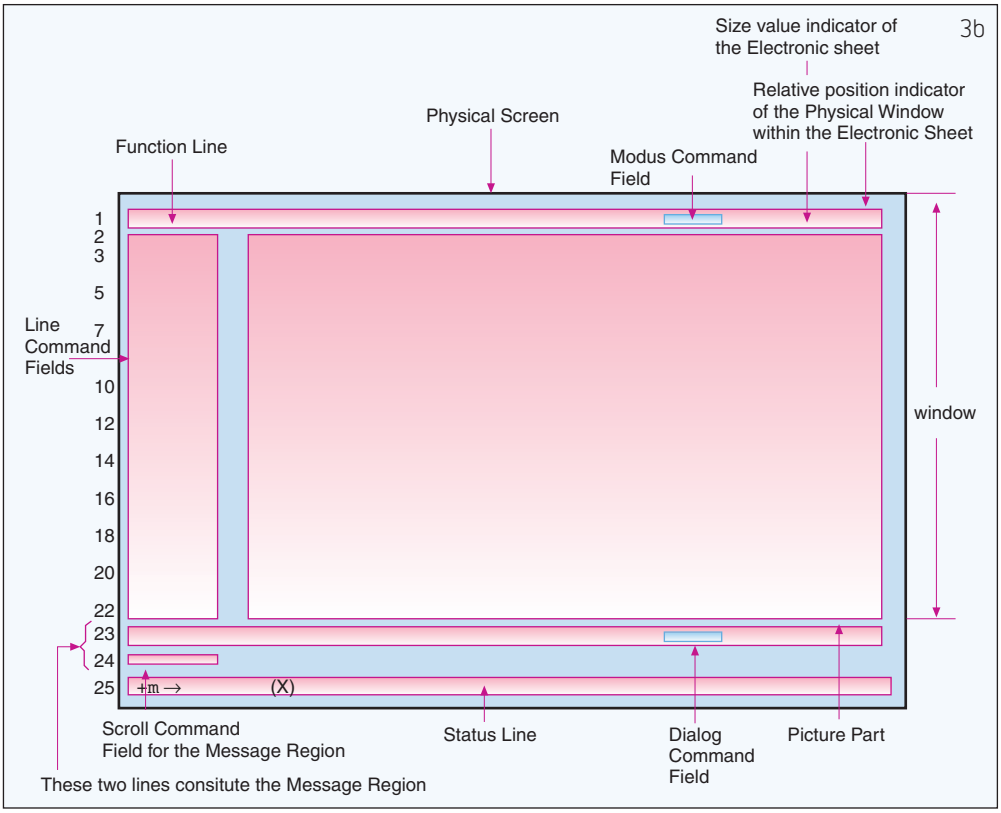
The presentation of data in a form follows a predefined grammar. It means that the context of the data item is significant. Related objects are placed close to their origin object, and attributes are placed so that it is clear to which object they belong. Any user aware of this fact will understand a lot about the application merely by having a look at the form. The layout of data items in a form is automatically generated by the system, but may to some extent be controlled by the developer, and even by the end user himself. No traditional " screen painting" is needed. The form may contain more data than what can be seen within a single screen simultaneously.

When DATRAN retrieves data from the database, the items are arranged according to the actual form. Headings are provided only when necessary. Thus the form is expanded when multiple occurrences of the same data type are to be presented. Not all the data in the database are necessarily retrieved in one step, only as much as what fills up a predefined or user defined work area is retrieved. Browsing of data outside the actual screen is possible, (up, down, left, right), and more data may be read from the database in a next retrieving step. The various retrieving operations are initiated by commands and retrieving criteria given by the user. The commands are independent of the application, they belong to DATRAN. Transitions are possible to all data in the form, if there is defined a suitable form for the item pointed at. There is no application code involved to provide these operations.

When data are inserted or modified, the given input is controlled according to what is defined as valid in the application schema, and default values are provided. Also the updating operations are initiated by standard DATRAN commands and involve no application code. However, more complex operations involving calculations, derivations and wanted redundancy in the data system must be provided as code in some form. In DATRAN these tasks are performed by COBOL programs, automatically invoked by DATRAN when the specified operations are executed. These application dependent programs are written in accordance with the rules in the application layer of the system, but are addressing the entities in the internal layer. The programs do NOT contain any presentation specific code, so typically in a DATRAN system there are far less programs than objects. The centralised application logic makes all objects behave in a consistent way independent of which context they are used in.

```
TDMO*READ, COMPANY, COMPANY, TELE                          21/72  0001/1
          company
          SSL
          address                                  telephone      status
          SYSTEMSERVICES LMTD., OSLO NORWAY        410120         A
          remark
          -CONSULTANTS IN ELECTRONIC DATA PROCESSING
          -DISTRIBUTORS OF DATRAN
          -OFFERS CUSTOMIZED COURSES IN DATRAN AND SYS.DEV.METHODS
          sales agent                                              district
          code       name                                          code
          RT         RICARDO TOWERS
          SG         SERGE GAMBI                                   STB
          JEJ        JOHN ERIC JACOBSEN                            JAP
                                                                   STB
          OL         ODD LARSON                                    LIL
          LTH        LIV TORUNN HOPE                               BAR
          ED         EDWARD BERNARDO                               PHI
          warehouse
          code       name/address                                  telephone
          SSL        WAREHOUSE 1                                   06 202020
          OCW        OSLO CENTRAL WAREHOUSE                        02 039303
IDMS7*IDMO, MESSAGE,, ROGERS, IDF0210, 92.0.131*14.08              01  1   0025
          01                    MORE NEXT PAGE                     02  00  01401
```

3a



3b

The reason for coding in COBOL is obvious. The programming language must have a runtime system working with the commonly used transaction monitors as well as in batch and against the chosen database system. Also the language must be complete and preferably in common use.

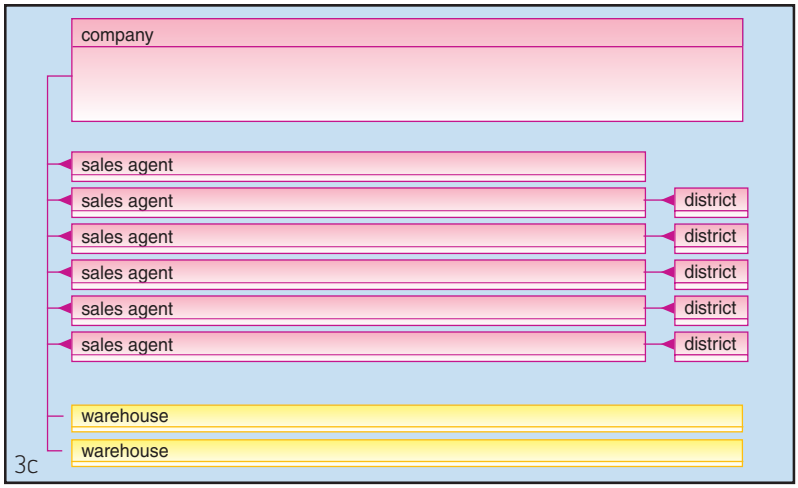
The dialogue itself is modeless. Every step in the end user's dialogue with DATRAN is a finished piece of work. All the work may not be done, but what is done has updated the database and is available for that user later on and also for other users. The idea is that the user shall have full control over the data processing system, not the other way around.

There are no traditional functions in a DATRAN application, only data. Thereby, not much work is done to analyse and program organisation of work, tasks and their sequences, routines, and so on. As a consequence of this, it is not necessary to make changes in the data system if and when the mentioned conditions are changed. The functions in DATRAN consist of screen forms and serve the purpose of access control. All the forms with the same set of access rights are gathered in the same function. When a user chooses a particular form, DATRAN checks in an access control database whether the user has the right to use the form. If not, access is denied. It is even possible to specify for which set of data occurrences access right is granted. Thus, there is no need for application code concerning access control.

In addition to these main functions of any on-line database system, DATRAN provides various means to facilitate the work for the user. Many of the facilities would have been far too expensive to implement in most conventionally developed systems. For instance, the user may on-line customise a screen form to serve his particular needs, remove data items that are not needed, reduce or remove headings, retrieve related data satisfying given criteria, and cut and paste data from several forms. The screen may be divided into several windows showing different or the same form with application data. It is also possible to copy data within or between windows. The form may be expanded at any point for insertion of new data occurrences.

End users write DATRAN commands either in particular command fields on the screen or in a general command area to the left of where the application data is presented. The most common commands have an equivalent PF-key, and pointing may be done by cursor positioning. All the items in the DATRAN screen, including the commands, are documented in their own database, available through the DATRAN Help command or the PF-key for Help.

Maybe the most powerful dialogue facility in DATRAN is how it handles dialogues. On user request DATRAN will save all the data related to the actual screen picture, briefly named a "dialogue". This makes it possible to re-invoke that picture or dialogue on request at a later stage, and to continue the work



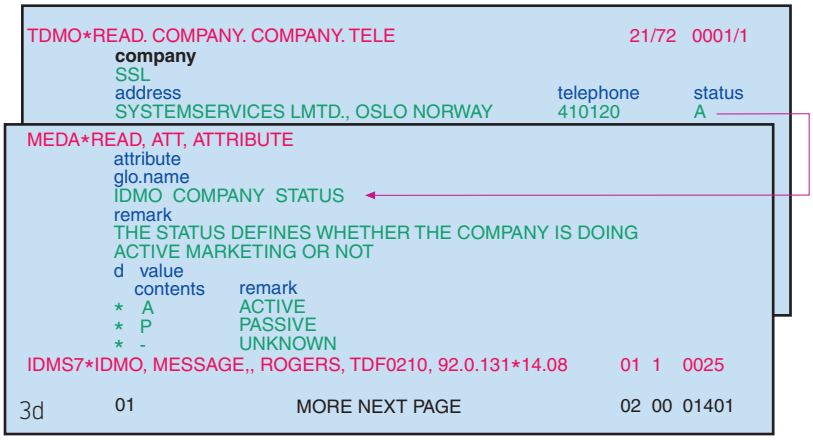


Figure 3 Figure (a) shows an ordinary alphanumeric picture from DATRAN. Figure (b) depicts the standard layout of a DATRAN picture. Figure (c) shows how data relate to each other. Figure (d) shows a Help screen for a specific field, provided by direct access to MEDA. The many pre-made dialogue features of DATRAN are not illustrated

with that origin. A common transition is to access the documentation database for viewing valid attribute values, and then to return to the application. Also the user may have a look at another object in the same database, or leap to a different (DATRAN) application system, possibly for cutting and pasting data.

DATRAN provides some mechanisms to automatically assign values to attributes. By choosing a particular form, insertion of an object will assure that the included attributes for that object will get their default values. If the form already presents a suitable occurrence of an object, it is straight forward for the user just to modify relevant attributes and in this way insert a similar object. Eventually, values can be inserted by an application program which may be invoked when the database operations are executed.

Internally DATRAN is implemented with the same structure as that of any other transaction program. On the top, a control module is handling the terminals and the DATRAN dialogues, as well as invoking the other modules in the runtime system:

- a module for transformation of input data to DATRAN internal format
- a module for analysis of commands and requests concerning the form in a window
- a module for analysis of commands acting on application data items and preparation for updating of the database
- a module for access control
- a module for updating the database
- a module for retrieving data
- a module for processing selection criteria for the retrieved data, and finally
- a module for transforming data from the DATRAN internal format to the screen format.

Even if this is not a client-server architecture, the separation of the various tasks in the processing of a transaction will facilitate a client-server like processing. See below about DIMAN.

The DATRAN system family

The DATRAN environment contains several subsystems and utilities to be mentioned:

MEDA, the MEta DAta system for DATRAN, is the system developer's tool. Specification data for the various applications are stored in the meta database, where they are updated and retrieved by MEDA, which is itself a DATRAN application. Via MEDA the application system is defined, with its three layers, as mentioned before. These meta data form an on-line documentation database, which is integrated with the application system. The meta data are used both by the end user and the system developer, as well as by DATRAN itself. However, when used by DATRAN, they are transformed from the database to a load module form, due to performance reasons. It is important to point out that the screen layout, as well as the commands applied in MEDA, are just the same in MEDA as in any other DATRAN application.

ADAD, the ADmission ADministration system for DATRAN, is used for updating the access control database that DATRAN uses to grant access. Even ADAD is a DATRAN application with all the common facilities. During access control database data are used in their original form, without any transformation. The ADAD database contains entities for users and user groups, terminals and terminal groups, systems, (DATRAN) functions and ownership of sets of data occurrences, called DATRAN data areas, as well as permissible combinations of these entities. ADAD is very close to a draft CCITT recommendation for access control administration, described elsewhere in this book.

JOBB is a third subsystem of DATRAN, and another DATRAN application. The JOBB database has data about (batch) jobs, parameters, JCL, users, and everything that is necessary to start a predefined batch job, but it does not contain any mechanism for supervision of the execution or eventually the printout. This is to some extent undertaken by GRAF.

GRAF contains structures for storing and maintaining graphical data and files. In this case the database serves more as a distribution means than an ordinary end user data system. DATRAN provides the required functions for transferring data to and from the database, and the recovery mechanisms in the database management system assures consistent updating of the data, as well as access by multiple users. The format of the graphical or file data stored in the GRAF database is unknown and of no interest to DATRAN, which in this case operates only as a server for this database.

In addition to these DATRAN database applications, there are a few tools and utilities:

SADA is a batch version of DATRAN, but only for data retrieval and printout. On the basis of an ordinary DATRAN presentation form, SADA selects one particular or any given number of object occurrences from the application database, retrieves the data specified, and presents them to a file or for printing.

For debugging of application programs as well as of the DATRAN modules, system developers are provided with the DATRAN DEBUG system, which makes it possible to stop at any label in a COBOL program and display the various data involved.

At last there is CDML, a Conversational Data Manipulation Language for quick and easy reading and writing in CODASYL databases. CDML has a shorthand notation for the data manipulation statements and operates on the unformatted database record. It needs no compilation or pre-processing and can access data through any existing sub-schema, if permission is granted by the database management system.

DATRAN is mainly written in COBOL.II, but there are a few minor assembler modules. The system runs on IBM mainframes in a pure IDMS environment or in a mixed CICS transaction and IDMS database set-up. Recently both the on-line and batch runtime system have been expanded with full access facility for DB2, and the main subsystems, MEDA and ADAD, have been given a DB2 implementation.

DATRAN applications

INSA was the first DATRAN application. This is an information system for the trunk and long lines network, which also was the justification for DATRAN itself. Since late 1983 INSA has grown in every way, data types, data occurrences, number of users and functionality. It is the most complex and one of the most important databases in Norwegian Telecom.

In 1987 Norsk Hydro was licensed to use DATRAN, and their integrated pension system, IPS, was operative less than a year later, realised through DATRAN.

In 1989 a new major DATRAN application was put into operation, INTRA, a system for managing the equipment and cross couplings in the transmission centre in the telecommunication network. This system has a close relationship to INSA, but has a much larger amount of data.

The same year SIRIUS/R, the main system for general accounting, budgeting and reporting in Norwegian Telecom was implemented using DATRAN. A year later, SIRIUS/R was followed by SIRIUS/P, a system for project planning and supervising.

In addition to these, there are other DATRAN applications:

- REGINA, instrument administration
- DATAKATT, data term catalogue
- STAREK, radio broadcasting station administration
- MARIT, marketing support system
- TIMEREG, internal activity reporting
- KATB, commercial telephone catalogue announcements
- INFUTS, portable and spare tele equipment administration.

Several experimental DATRAN systems have been developed. Among these are GEOINSA, a system for geographical data and map making related to INSA, TELEKAT, X.500 electronic directory, SDL, a CCITT specification database, JUDO, a journal and document handling system. As already mentioned, most subsystems of the DATRAN family are themselves implemented by DATRAN.

The reason for choosing DATRAN for several of the above systems has been to reduce development costs and time. No other tool could deliver the application in due time with the resources available. Application developers have been able to use DATRAN with very high productivity after just a few days of training.

Currently about 5,000 end users utilise DATRAN applications.

DIMAN

A separate product, but with a relationship to DATRAN, is DIMAN. This is a dialogue manager for alphanumeric as well as colour graphic dialogues. DIMAN was originally developed as a DOS PC-application, but is now converted to Windows and UNIX environments.

Most graphic applications require large amounts of data to be transferred from the host computer to the terminal where the data are presented. For this use narrowband lines are usually too time consuming or in practice impossible. A basic idea behind DIMAN is to reduce the data transport to an extent that ordinary terminal connections are usable for graphics. The terminal used is an ordinary Personal Computer. The reduction in data transport is achieved through the intensive use of inheritance. Only the significant, different data are transferred for each occurrence of the graphical objects.

Another feature that distinguishes DIMAN from most graphical systems is that the drawings are composed of interrelated, graphical objects (icons). It is not just the graphical primitives that are communicated. In this way, objects can be moved around without loosing their connections to other objects on the screen. DIMAN also allows icons to represent objects from an application database, reference these objects and make operations on them. This is taken advantage of when DIMAN operates in conjugation with DATRAN. Data are retrieved from the application database, provided with the necessary graphical information, and sent to DIMAN as a meta file containing data types as well as data occurrences. DIMAN then makes the presentation on the screen.

When more processing is necessary to produce a drawing, the task is done as a batch job, and the result is stored in a suitable database, namely the DATRAN

subsystem GRAF. DATRAN and DIMAN have the required protocol to exchange these drawings, as well as ordinary files stored there.

Among the more usual features of DIMAN is windowing, dialogue handling of the same type as in DATRAN, mouse operations, zooming, panning, local editing of the drawings, printing and plotting the whole or part of the graphs. Close integration of graphics and alphanumeric forms is a strength of DIMAN. Edited drawings can be stored on the PC or sent back for storing in the GRAF database. Old editing and new graphs may be integrated and automatically kept consistent.

DIMAN has been used for presentation of alarm analysis for the telecommunication network, as well as for presenting alternatives for circuit routing, specific sub-networks and so on. Another application is the presentation of height diagrams between existing and planned radio link stations. This is of great value as a planning tool in mountainous Norway. In addition business graphics as well as data models are provided.

Currently some tens of different graph types exist. For these several thousands graph instances are produced automatically and distributed to some hundred users by DATRAN and DIMAN.

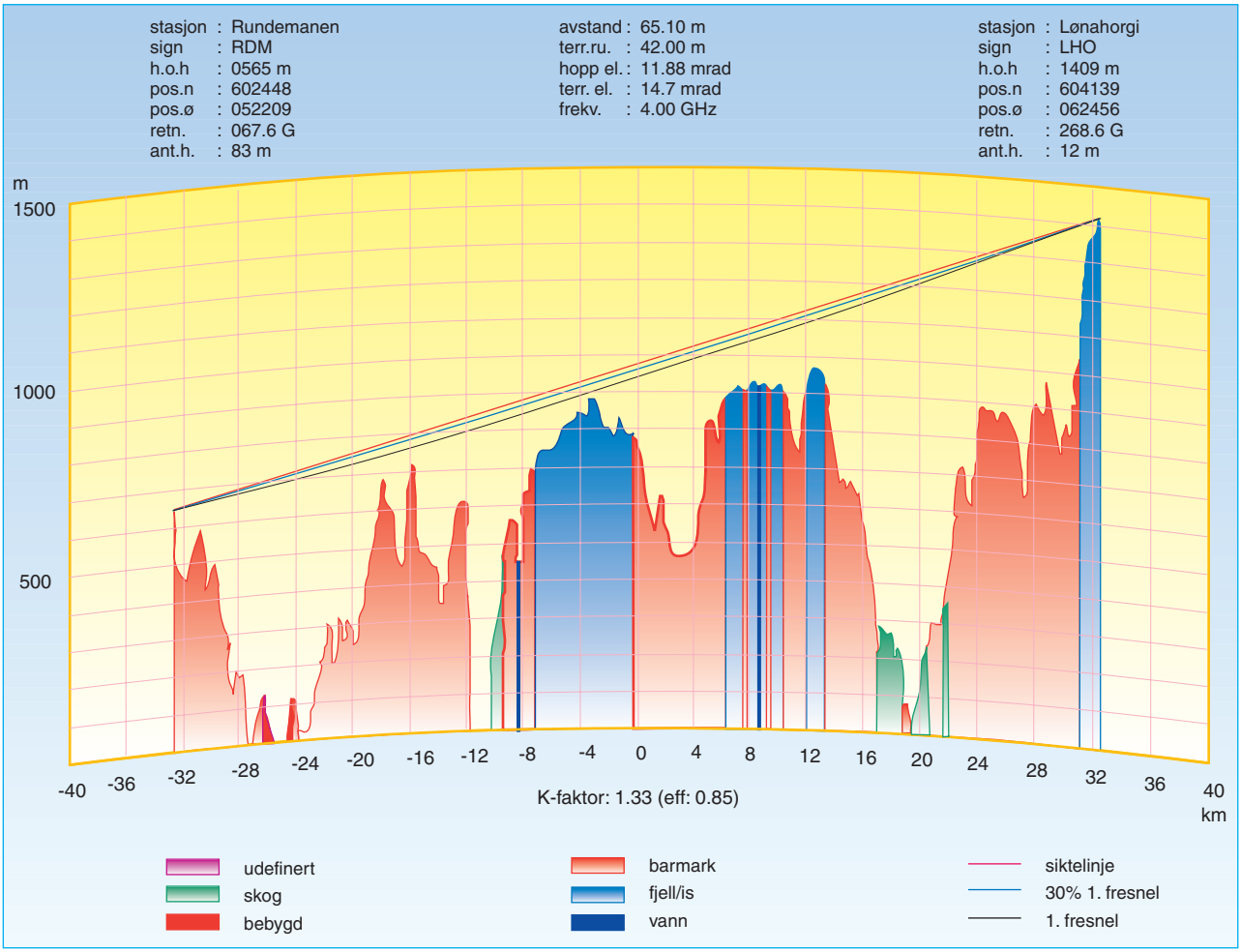


Figure 4 Graphical presentation by DIMAN
The example shows a height curve between two radio stations. When geographical co-ordinates of the two stations are provided, the curve is produced automatically from a height database of all of Norway

DIBAS – A management system for distributed databases

BY EIRIK DAHLE AND HELGE BERG

681.3.07

Abstract

The dramatic improvements in hardware, cost and performance for data processing and communication permit rethinking of where data, systems and computer hardware belong in the organisation. So far, most database applications have been implemented as rather isolated systems *having little or no exchange of data. However, users require data from several systems. This implies that* integration of data *and systems has to be addressed, and at the same time providing the local control and autonomy required by the users. Both intra-application integration (integration of data stored on different geographical sites for the same application) and inter-application integration (integration of data from different applications possibly stored on different sites) are required. Organisational changes may also affect the way data, systems and hardware are organised, because of new requirements resulting from changes in responsibility, management control, user control, etc. The solutions sought after must provide a trade-off between the integration and the local control and flexibility for organisational changes. This article will discuss how distributed databases can contribute to this.*

Distributed databases *can provide feasible technical options/solutions to the user requirements. They can increase data sharing, local autonomy, availability and performance. Distribution transparency of data to applications is often wanted. This means that the aspects related to distribution of data are invisible to the application programs. Theory of distributed databases was developed in the late 70s and the 80s, but still there are few commercial products that have implemented the wanted features suggested in theory. A distributed database system easily becomes very complex. Therefore, more modest approaches are developed, which provide a selection of important features.*

Norwegian Telecom Research has several years of experience with developing distributed database management systems. The Telstøtt project developed a prototype DDBMS, called TelSQL, which provided full integration of databases (1). The Dibas *distributed database management system takes a practical approach to distributed databases. Dibas is based on the idea that the responsibility for generation and maintenance of data are allocated to a number of organisational units located at different geographical sites, e.g. regions. Reflecting this responsibility structure, the tables of the database can be partitioned into ownership fragments, where updates of the data are allowed only at the owner database site. The ownership fragments can be fully or partially distributed to (i.e. replicated at) other sites, so that the other organisational units can read relevant data. The replicas are distributed asynchronously. This means that consistency of data is controlled, but is not enforced in real time.*

In this article, Dibas is presented in the perspective of distributed database theory. Two database applications, MEAS and TRANE, illustrate the feasibility of Dibas. MEAS will establish seven isolated regional databases and wants interconnection, while TRANE has a centralised database but considers distributing the data.

1 Distributed databases

1.1 What is a distributed database?

Definition (2): "A distributed database is a collection of data which are distributed over different computers of a computer network. Each site of the network has autonomous processing capability, and can perform local applications. Each site also participates in the execution of at least one global application, which requires accessing data at several sites using a communication subsystem." Figure 1 illustrates this definition.

Each computer with its local database constitutes one site which is an autonomous database system. During normal operation the applications which are requested from the terminals at one site may only access the database at that site. These applications are called local applications. The definition emphasises the existence of some applications which access data at more than one site. These applications are called global applications or distributed applications.

In this article only *relational* distributed database systems are considered.

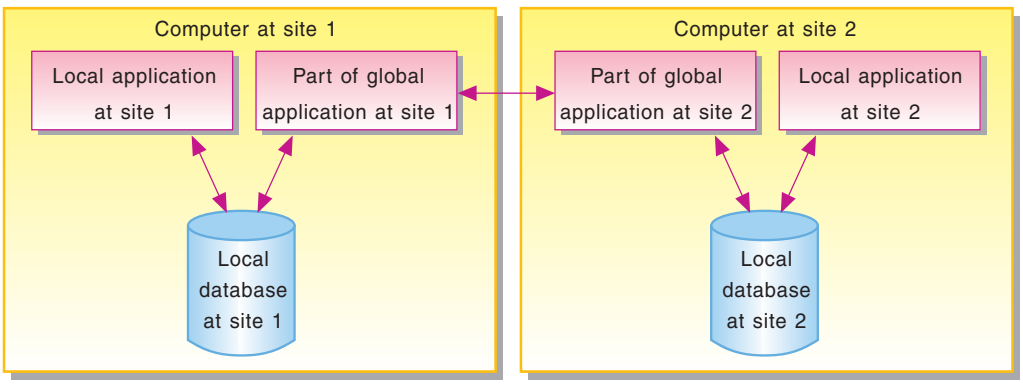


Figure 1 A distributed database. A global application accesses data at several sites

1.2 Fragmentation and replication

Performance and availability enhancements in a distributed database is usually achieved by storing copies of data on sites where the data are frequently used. Each table in a relational database can be split into several non-overlapping portions which are called *fragments*. The process of defining the fragments is called fragmentation. There are two types of fragmentation, horizontal fragmentation and vertical fragmentation. Fragments can be located on one or several sites of the network.

Horizontal fragmentation means partitioning the tuples of a table into subsets, each subset being a fragment. This way of partitioning is useful if the users/applications at one site use one subset of the tuples of a table more frequently than other tuples.

Vertical fragmentation is the subdivisioning of attributes of a table into groups, each group is called a vertical fragment. Vertical fragments are defined when users/applications at different sites access different groups of attributes of a table.

It is also possible to apply a combination of both vertical and horisontal partitioning. This is called mixed fragmentation.

Replication consists of locating identical fragments at several sites of the network in order to increase performance and availability when the same data are frequently used at several sites. Replication introduces redundancy to the database. Note that in traditional databases redundancy is reduced as far as possible to avoid inconsistencies and save storage space.

1.3 Distributed database management system

Definition (2): A distributed database management system (DDBMS) supports the creation and maintenance of distributed databases.

What is the difference between a DDBMS and an ordinary Data Base Management System (DBMS)? An ordinary DBMS can offer remote access to data in other DBMSs. But a DBMS usually does not provide fragmentation and replication and supports only a limited degree of distribution transparency, while these capabilities usually are supported properly by a DDBMS. A DDBMS typically uses DBMSs for storage of data on the local sites.

An important distinction between different types of DDBMSs is whether they are homogeneous or heterogeneous. The term homogeneous refers to a DDBMS with the same DBMS product on each site. Heterogeneous DDBMSs use at least two different local DBMSs, and this adds to the complexity. Interfacing of different DBMSs will be simplified by using the SQL standard (3).

1.4 Distribution transparency

A central notion in database theory is the notion of data independence, which means that the physical organisation of data is hidden to the application programs. Data independence is supported by the three schema architecture of ANSI/SPARC, since the internal schema is hidden for the application programs. In distributed databases we are concerned with distribution transparency, which means independence of application programs from the distribution of data. *The degree of transparency can vary in a range from complete transparency, providing a single database image, to no transparency at all and thus providing a multi database image.*

Analogous with distribution transparency, (4) states a fundamental principle of distributed databases; "To the user, a distributed system should look exactly like a non-distributed system". From this principle, twelve objectives (analogous to transparency) for distributed databases have been derived:

1 Local autonomy. A database site should not depend on other sites in order to function properly.

2 No reliance on central site. There must be no reliance of a master site because of problems with bottlenecks and central site down time.

3 Continuous operation. Shutdown of the system should not be necessary when table definitions are changed or new sites are added.

4 Location independence, also called *location transparency*. This means that the location of data at various sites should be hidden to the application program. This transparency is supported by the allocation schema of the proposed reference architecture in section 1.5.

5 Fragmentation independence, also called *fragmentation transparency*. This means that application programs should only relate to full relations/tables and that the actual fragmentation should be hidden. This transparency is supported by the fragmentation schema in section 1.5.

6 Replication independence, also called *replication transparency*. This means that the replication of fragments should be invisible to application programs. Replication transparency is part of the location transparency notion, and is supported by the allocation schema in section 1.5. The basic problem with data replication is that an update on an object must be propagated to all stored copies (replicas) of that object. A strategy of propagating updates synchronously within the same transaction to all copies may be unacceptable, since it requires two-phase-commit transactions (see section 1.6) between nodes. This reduces performance and autonomy. This problem can be removed by relaxing the up-to-date property of stored copies, see section 1.7.

7 Distributed query processing. If the application program makes queries about data which are stored on another site, the system must transparently process the query at the involved nodes and compose the result as if it was a local query. This involves:

- Translation of the "global" query into local queries on the fragments stored locally.
- Query optimisation. Queries accessing several sites will result in a message overhead compared to a centralised database. Effective optimisation strategies are crucial to avoid large amounts of data being shuffled between sites. A "send the query to the location of the data" strategy will reduce this problem compared to a big-inhale strategy. Replication of relevant data so that global applications are not necessary, can reduce this problem.

8 Distributed transaction management. A distributed database transaction can involve execution of statements (e.g. updates) at several sites. A transaction is a unit of work which either should be completely executed or rolled back to the initial state. Concurrency and recovery issues must also be handled, see section 1.6.

9 Hardware independence.

10 Operating system independence. This is relevant in case of use of various operating systems.

11 Network independence. This is relevant in case of use of several networks.

12 DBMS independence, also called *local mapping transparency.* This means that the mapping to the local DBMS is hidden to the application program. This transparency is supported by the local mapping schema in section 1.5.

1.5 A reference architecture for distributed databases

The reference architecture depicted in Figure 2 only shows a possible *organisation of schemata* in a distributed database. A schema contains definitions of the data to be processed and rules for the data. Communication aspects and implementation aspects are not shown in this architecture. The architecture is based on the *three schema architecture of ANSI/SPARC,* developed for non-distributed databases, where the three following kinds of schemata are covered:

- The *external schema,* which defines a presentation form for a subset of the information defined in the conceptual schema. There will be several external schemata related to the conceptual schema.
- The *conceptual schema,* which defines the complete information content of the database. It includes integrity rules and business rules for the data.
- The internal schema, which defines the internal storage of data. There is one internal schema related to the conceptual schema.

In *distributed databases* the conceptual schema is also called the global schema. It defines all the data which can be contained in the distributed database as if the database was not distributed at all. We will focus on distribution and storage of the data. Hence, the ANSI/SPARC architecture has been extended with an additional schema called the distribution schema, which defines fragmentation and allocation of data to database sites. There will be several internal schemata (one per site). Because DBMSs are used on each site for storage, the internal schemata are covered by the database schemata of the DBMSs on each site. There are mappings between all the schemata (external, conceptual, distribution and internal) of the architecture. An important mapping for distributed databases is the mapping between the distribution schema and the database schemata of each site. The mapping is defined in the local mapping schema. The resulting extensions and modifications are:

- The *distribution schema,* which defines the distribution rules for the data in the conceptual schema and has been split into two schemata; fragmentation schema and allocation schema. The *fragmentation schema* defines the splitting of a table into several non-overlapping fragments. The *allocation schema* defines on which site(s) a fragment is located. The architecture supports separation of data fragmentation from data allocation and explicit control of redundancy. Note that the conceptual schema and the distribution schema are DBMS independent schemata and also independent of local naming and mappings
- The *local mapping schema,* which maps the allocated fragments to the data definitions of the local DBMSs

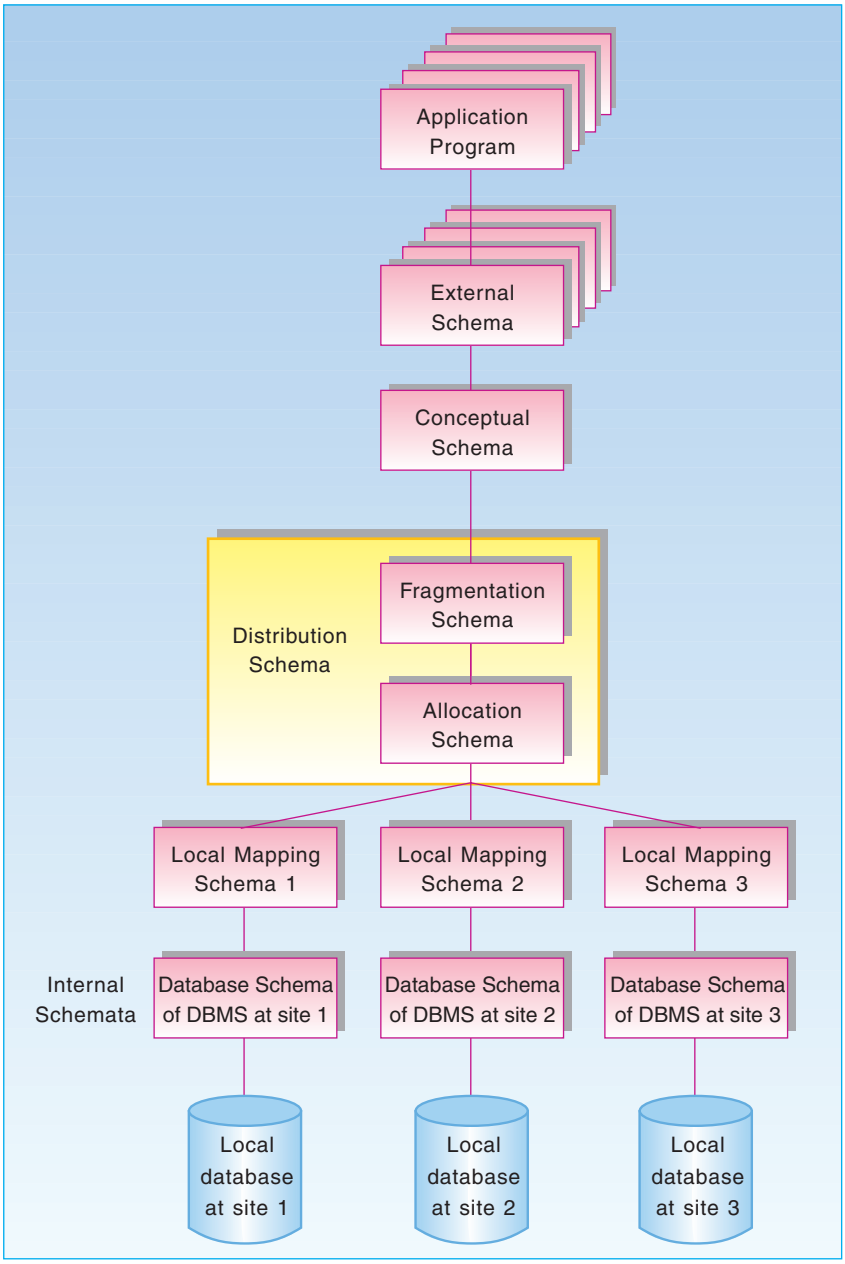


Figure 2 A reference architecture for organisation of schemata in a distributed database. All the schema types will typically be present on all sites. The three branches of the lower part of the figure indicate that these schemata are site dependent. The conceptual schema and the distribution schema are relevant on all sites

- The *database schema,* which contains the data definitions of the local DBMS.

When an application program wants to access data in the databases, the program inputs e.g. a query which is processed through all the schema-levels of the architecture. At the conceptual schema level, the global query will be evaluated, while at the distribution schema level, the query has been split into queries on fragments, which are evaluated against the fragmentation schema. The sites that are going to process the queries can be decided from information in the allocation schema. At the involved sites, the fragment queries are mapped onto the local database queries and evaluated against the local database schemata. This finally results in database operation(s)

against the database(s). On the way back from the databases to the application program, the results of the local database queries will be combined into one result and given as output to the application program.

1.6 Distributed transaction management

1.6.1 Transactions

Users/application programs use transactions in order to access the database. A transaction is a logical unit of work and can consist of one or several database operations in a sequence. A transaction has the following properties, called *ACID-properties:*

- Atomicity: All operations are executed (commitment) or none (rollback/abortion). This is true even if the transaction is interrupted by a system crash or an abortion. Transaction abortion typically stems from system overload and deadlocks.
- Consistency: If a transaction is executed, it will be correct with respect to integrity constraints of the application.
- Isolation: Processing units outside the transaction will either see none or all of the transaction changes. If several transactions are executed concurrently, they must not interfere with each other, and the result must be the same as if they were executed serially in some order. This is called *serialisability* of transactions, and serialisability is a criterion for correctness. *Concurrency* control maintains serialisability. In order to avoid that other transactions read partial results of a transaction, the data that have been read or updated by the transaction, are *locked.* Other transactions have to wait until the locks are released before they can access these data. The locks are released once the execution of a transaction has finished. In certain cases the locking can lead to *deadlock* situations, where transactions wait for each other. Deadlock detection and resolution are usually handled by the DBMS.
- Durability: Once a transaction has been committed, the changes are permanent and survive all kinds of failures. The activity of ensuring durability is called *recovery.*

So far, the discussion of this section has been valid for both non-distributed and distributed databases. In distributed databases there will be distributed transactions as well as local transactions. A distributed transaction contains database operations that are executed at several different sites, and still the ordinary transaction properties have to be maintained.

1.6.2 Concurrency

The most common techniques for concurrency control and recovery in a distributed database are the *two-phase-locking (2PL)* protocol and the *two-phase-commitment (2PC)* protocol.

The 2PL protocol have these two phases:

- In the first phase locks on the data to be accessed are acquired before the data are accessed.
- In the second phase all the locks are released. New locks must not be acquired.

The 2PL protocol guarantees serialisability of transactions if the transactions hold all their locks until commitment. For distributed transactions, serialisability is often called *global serialisability.* The locking creates overhead compared to a centralised database, because each lock needs to be communicated by a message. For replicated data, all the copies have to be locked. An alternative locking method for replicated data is primary copy locking, where one of the copies are privileged, see section 1.7. Deadlocks are often called *global deadlocks* for distributed transactions. Deadlock detection and resolution is more difficult in a distributed system because it involves several sites.

1.6.3 Recovery

A distributed transaction can be divided into a number of subtransactions, each subtransaction being the part of the transaction that is executed on only one site. The atomicity property requires that each subtransaction is atomic and that all subtransactions go the same way (commit or rollback) on each site also in case of system crashes or transaction abortion. The 2 PC-protocol ensures atomicity. It has a co-ordinator which sends messages to the participants and takes the decision to commit or rollback. Roughly, the two phases are:

- In the first phase all the participating sites get ready to go either way. Data are stored on each site before execution of the transaction and after the transaction has been executed. The coordinator asks prepare-to-commit, and the participants reply ready, if they are ready to commit, or rollback, if they cannot support both the initial and the final state of the data. When all the participants have answered, the co-ordinator can take the decision either to commit or to rollback.
- In the second phase the commit or rollback decision is communicated to all the participants, which bring the data to the right state (final or initial, respectively).

The 2PC-protocol means overhead and loss of autonomy, see section 1.7 for alternatives.

1.7 Increasing performance, availability and autonomy

Maintenance of global consistency when replicated data are updated, may become costly. It is suggested in several papers (e.g. (1), (5) and (6)) that availability, autonomy and performance can be increased in a distributed database if it is acceptable to relax consistency and distribution transparency. Often the up-to-date requirement can be relaxed and asynchronous propagation of replicas from a master fragment will be sufficient. Asynchronous means that updates to the master fragment will not be distributed to the replicas within the same transaction, but later.

The approach proposed in (5) is to partition the database into fragments and assign to each fragment a controlling entity called an agent. Data can only be updated by the owner agent, which is a database site or a user. When a fragment has been updated, the updates are propagated asynchronously to other sites, transaction by transaction. Global serialisability of transactions is not guaranteed. A less strict correctness criterion, fragmentwise serialisability, is introduced:

- Transactions which exclusively update a particular fragment are globally serialisable.
- Transactions which read the contents of a fragment at a site never see a partial result of a transaction.

- There is no guarantee that transactions which references data in several fragments are globally serialisable.

2 The Dibas DDBMS

2.1 Introduction

The vendors of relational DBMSs also have products for distributed databases. They typically offer remote access to data, support for execution of distributed transactions and some other features. Fragmentation and replication are usually not supported. Distribution transparency is only partly supported. Support for creation and maintenance of the distributed database are typically not good.

Dibas has been designed from the experience of the Telstøtt project (1), from the ideas of section 1.7 and from the application requirements of TRANE (see chapter 4).

2.2 Basic idea

Dibas is a homogeneous DDBMS, which is rather general in the sense that it can be used for various applications. A distributed database managed by Dibas consists of a number of Sybase (7) databases containing distributed tables with common table definitions in all the databases. Dibas provides the following features for distributed tables (see (8) for a more comprehensive description):

1 A table can be partitioned into fragments. Each fragment is owned exclusively by one database site, which means that updates to this fragment are only allowed by users at this site. A distributed table will have ownership fragments at all sites. An ownership fragment can be empty, include some tuples or contain all the tuples of the table.

2 Read access to fragments owned at other sites is provided by replication. Fragments can be fully or partially replicated to other sites. Each site can receive different subsets of the ownership fragment of a site. There is only read access to replicas.

3 Updates are propagated asynchronously. A separate distribution program performs asynchronous propagation of replicated data. Only changes to the data after the last distribution of a copy, are distributed. The distribution is initiated periodically and/or on request from the users. Hence Dibas offers relaxed, but controlled consistency.

4 A distributed data definition language called DSQL has been defined to express the ownership and distribution rules of a table, see section 2.3 and 2.4. The full set of rules for each table are equal at all sites.

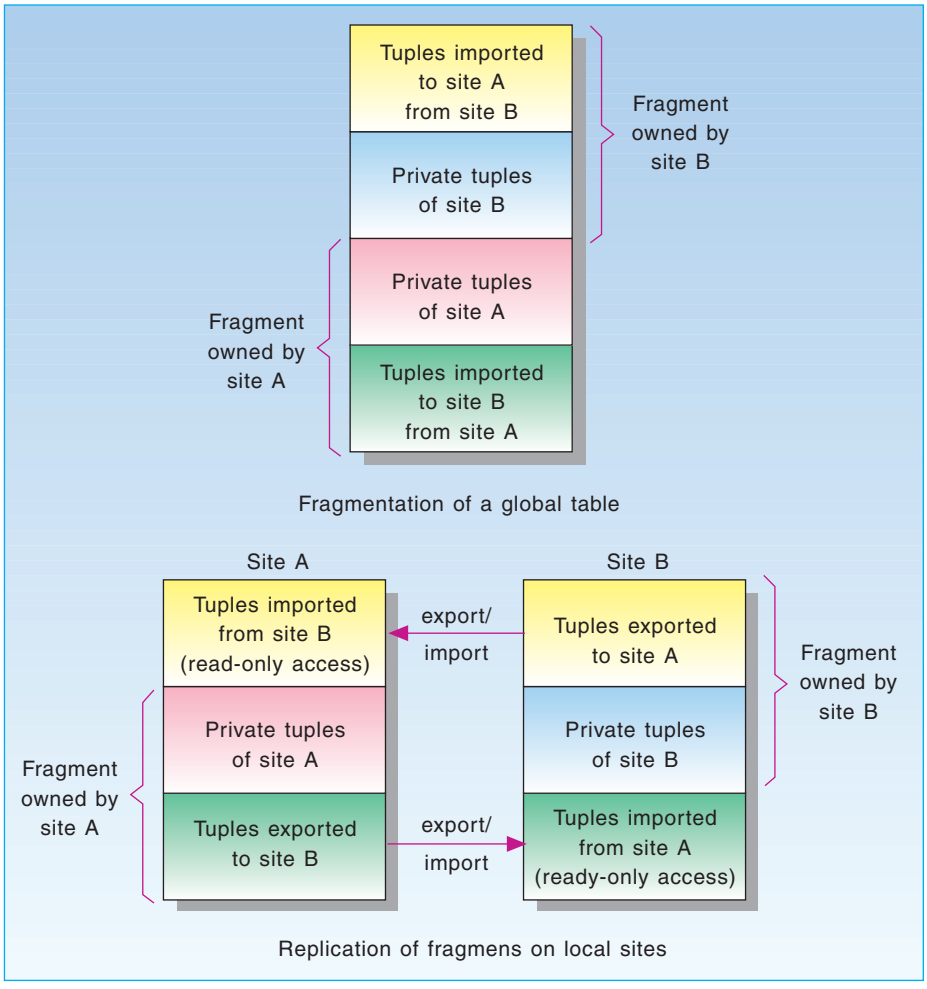


Figure 3 A distributed table in Dibas

Figure 3 illustrates the fragmentation and replication of a distributed table in Dibas. It is also allowed to define local tables on each site which can be read and updated by users at the local site exclusively.

The result of 2 and 3 above is that the application programs on the local sites will only execute as local applications. They do not require distributed query processing and distributed transactions. There are no reliance on a central site. Therefore Dibas provides a high degree of autonomy. Sybase mechanisms for concurrency control and recovery can be utilised. This is a major simplification. Also, installations of the Dibas system are equal at all sites.

Dibas has good support for distribution transparency. This means that the applications can be developed as if they were running on a centralised database, because they see the same tables and expect to find the needed data. Each site only holds relevant data for its own users. Read transactions on these relevant data are offered complete distribution transparency except for reduced timeliness of replicated data. Update transactions can only access owned data. If this restriction corresponds to defined responsibility for data and needed access control, transparency will be offered for update transactions. Dibas can only run on Sun work stations under the UNIX operating system at the moment.

2.3 Definition of ownership fragments

In Dibas, there is one rule specifying *ownership* for each distributed table. The ownership rule together with data in tables determine the partitioning into non-overlapping fragments. Because different tables will require different fragmentation, several *types of ownership* for a table have been defined. These types are presented below with the attached DSQL syntax:

1 *Entire table.* The table is owned entirely by a given site. This ownership type is particularly useful for standards/domain/type information which should be updated by one organisational unit at one site and be replicated to several/all other sites. The owner site can be defined at compilation time (the expression `BY <site>`) or defined run-time in a dictionary table:

```
[DEFINE] OWNERSHIP FOR
<tables> [AS] OWNED ENTI-
RELY [BY <site>]
```

2 *Column value.* Ownership is determined by the value of a given column which directly tells the ownership site of each tuple:

```
[DEFINE] OWNERSHIP FOR
<tables> [AS] OWNED
DIRECTLY GIVEN BY <attr>
```

3 *Indirect through foreign key.* Ownership for each tuple of the table is determined indirectly through a foreign key to another table. For applications it is common that ownership of tuples of a table can be derived from another table in this way, see chapter 4.

```
[DEFINE] OWNERSHIP FOR
<tables> [AS] OWNED
INDIRECTLY GIVEN BY
<attrlist> VIA <table>
```

Usually the ownership of a tuple is the same as the tuple it references through a foreign key. However, it is possible to define a separate ownership rule for the tables that references another table (column value ownership):

```
[DEFINE] INDIRECT OWNERS-
HIP VIA <table> [GIVEN BY
<attr>]
```

2.4 Definition of replication

Distribution rules specify the replication of data. Distribution should also as far as possible be determined without modifications to the application's database schema. The distribution fragments will always be subset of the ownership fragment. The *distribution types* are:

1 *Entire fragment.* The entire ownership fragment is distributed to all other sites (see the first DSQL expression) or to specified sites (see the second DSQL expression). In case of the latter, a dictionary table will contain data about the recipient sites. This makes it possible to decide and modify the list of recipient sites at run time.

```
[DEFINE] DISTRIBUTION FOR
<tables> [AS] DISTRIBUTE
TO ALL
```

```
[DEFINE] DISTRIBUTION FOR
<tables> [AS] DISTRIBUTE
TO SOME
```

2 *Indirect through foreign key.* The distribution fragment and the recipient sites are determined through a foreign key to another table. This referenced table is usually a dictionary table which contains tuple identifiers and the recipient sites for these particular tuples. See chapter 4 for examples.

```
[DEFINE] DISTRIBUTION FOR
<tables> [AS] DISTRIBUTE
INDIRECTLY GIVEN BY
<attrlist> VIA <table>
```

3 *Arbitrary restrictions.* Sometimes it is necessary to define arbitrary conditions for the distribution. Arbitrary restrictions are expressed in SQL and can be added to all the distribution types:

```
[WITH] [<tables>]
RESTRICTION <cond>]
```

4 *Combination of distribution types.* There can be several distribution rules for each table.

2.5 Creation of the distributed database

The ordinary application database schema (definitions of tables, indexes, triggers, etc.) and the additional ownership and distribution definitions for each of the tables to be distributed are taken as input to a definition tool, which parses the information and creates the definitions to be installed in the database, see Figure 4.

For each table, the definition tool creates a trigger (a trigger is a database object which is activated by operations on a table) and a log table. If there already exists an application trigger, this trigger will be extended. The purpose of the trigger is to check correct ownership based on the defined ownership rule and to log changes to data including a timestamp for the time of the change in the log table. This means that the ownership rule in DSQL has been translated into an ownership SQL-expression.

For each table, the definition tool also creates an SQL-expression which extracts the tuples to be distributed to the recipient sites from an owner site. In normal operation, the SQL-expression only extracts changes to the data, i.e. tuples from the log tables. Initially, full replicas are extracted from the tables themselves. The SQL-expressions are created from the distribution rules defined for the tables.

Administration utilities install the database definitions generated by the Dibas definition tool, install owned data and install the Dibas distribution program.

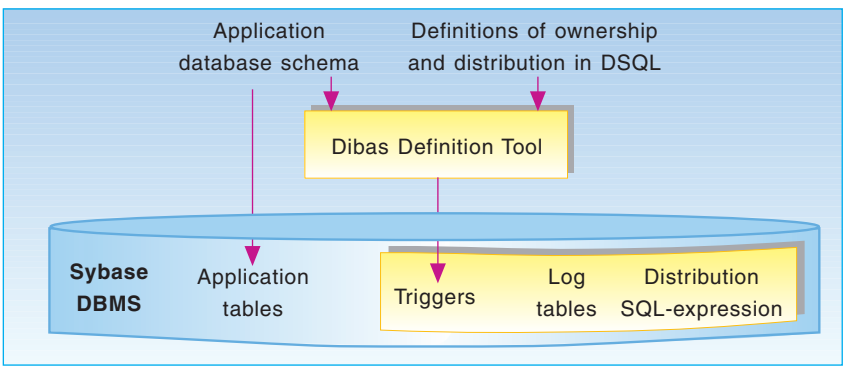


Figure 4 Creation of the distributed database

2.6 Run time system components of Dibas

The *application programs* run as local applications on each site and access the Sybase DBMS directly. Sybase triggers are activated immediately by insert, delete and update operations on the database to verify ownership and to log changes.

A separate *distribution program* propagates the updates which are to be replicated. The distribution fragments to be distributed are selected by the distribution SQL-expressions which are stored in the Dibas dictionary.

Also a tool for *run time administration* of Dibas offers user initiation of the distribution program, information about timeliness of replicated data, parameters for control of distribution, etc.

2.7 Dibas and the proposed reference architecture for distributed databases

Dibas mainly follows the previously described reference architecture. The *conceptual schema* will be the ordinary database schema of the application.

The *distribution schema* of Dibas consists of ownership fragment definitions and distribution definitions expressed in DSQL for each distributed table. The ownership fragments defines the fragmentation and corresponds to the fragmentation schema, but it also defines the allocation of the master copy of the fragment. Hence, it also has allocation schema information. The distribution definitions in DSQL defines replication, which belongs in the allocation schema.

Dibas is a homogeneous DDBMS with Sybase as the local DBMS. There is no local naming or other mappings between the tables of the conceptual schema and the local database tables. Hence, the *local mapping schema* disappears from the architecture. Because Dibas and the database applications are identical on all sites, the *database schemata* will be identical as well.

2.8 Management of distributed transactions in Dibas

The only global application in Dibas is the distribution program, which transfers data from one site to another. The Sybase 2PC-protocol is used by this program. In case of crashes and other failures, ordinary recovery of the local Sybase databases will be sufficient. Global deadlocks will not appear, because the program itself will do a time-out if it has to wait for locks.

3 Advantages and disadvantages of Dibas

The list of advantages and disadvantages has been assembled from (2), (3), (4), (5), (6), (9), (10), and our own experience.

3.1 Advantages of Dibas

3.1.1 Local autonomy and economic reasons

Many organisations are decentralised and a distributed database fits more naturally to the structure of the organisation. Distribution allows individual organisational units to exercise *local control* over their own data and the computers involved in the operation.

Due to the closeness of data, it is often easier to motivate users to feel responsible for the timeliness and integrity of the data. Thus, *higher quality* of data may be achieved.

An organisational unit with responsibility for its own economic results prefers to control its investments and costs. It can control the cost of operating a local database, while expenses to be paid to a central computer division are harder to control. In the past, hardware costs were smaller for large mainframes than for smaller hardware. Hardware was the most significant cost driver when a system was bought. Today the economy of scale motivation for having large, centralised computer centres has gone. It may be cheaper to assemble a network of smaller machines with a distributed database.

3.1.2 Interconnection of existing databases: Data sharing

If several databases already exist in an organisation and the necessity of performing global applications arises, then Dibas can provide integration of data in the databases. This can be a smooth transition to data sharing. The distributed database will be created bottom-up from the existing local databases.

3.1.3 Performance

Data in a Dibas distributed database are stored in the databases where they are used. If only a small amount of the data owned by the other sites are relevant for a local site, this means that the data volume in the local database can be reduced considerably compared to a centralised database containing the sum of data for all organisational units. *Reduced data volume* can improve the response time.

The existence of several database sites doing autonomous processing increases the performance through *parallelism* (the transaction load is divided on a number of database sites).

When running local applications, the communication is reduced compared to a centralised database and thus lead to reduced response time.

3.1.4 Local flexibility

A centralised database with access from a lot of users and with a heavy transaction load will require strict control of the applications which access the database, development of additional applications ,etc., and this prevents flexibility. For Dibas where each database site is autonomous and has fewer users and smaller load, less control is required. This eases the usage of new tools, creation of additional tables, development of additional applications, and experimenting in general.

3.2 Disadvantages of Dibas

3.2.1 Increased complexity and economic reasons

Compared to a centralised database or to several isolated databases, the distributed database adds to complexity and can add extra costs. The Dibas system components have to be installed, operated and maintained. Also, the distribution of data needs to be managed.

However, since Dibas is a relatively simple system, the complexity overhead and the economic overhead is kept low.

3.2.2 Timeliness and correctness

Disadvantages with Dibas are relaxed timeliness and correctness. The correctness of Dibas is less strict than the global serialisability (see section 1.6) and fragmentwise serialisability (see section 1.7). Dibas offers local consistency and serialisability at each site (handled by Sybase), which ensures that locally owned data will be correct. Data are propagated to other sites table by table according to timestamps. Transactions may see inconsistent data because data which have been updated on other sites may not have been propagated yet.

3.2.3 Ownership restriction

Data can only be updated on the owner site. This restriction cannot be accepted for some applications.

The potential security problems of distributed databases is avoided in Dibas. Users can only update data on their own site, and only relevant data are available from other databases.

4 Application examples

In this chapter we present two database applications, MEAS and TRANE, which are candidates for a distributed database solution. The cases have different initial states and therefore there are different motivations for a distributed solution. Examples of the suitability of the Dibas mechanisms are given for the MEAS case. Before presenting the applications, trends that will impact the applications and increase the need for data transfer mechanisms are outlined.

4.1 Trends in traffic measurements of the telecom network

The text of this section is based on (11). New opportunities for traffic measurements emerge with the introduction of digital switches. This has revolutionised the possibility of conducting measurements of the traffic load on different network elements. These measurements will give better data on which to base decisions concerning utilisation, reconfiguration, extension and administration of the telecom network. Measurements showing free capacity make possible a more dynamic utilisation of the telecom network.

The increasing amount of measurements data are stored in files and then transformed from files to databases. All the new measurements data give rise to an increasing need for storage capacity.

New user groups take advantage of the measurement data. Planners and administrators of the telecom network have been the traditional users of the traffic measurement data. Today there is a trend that economists need this kind of data, e.g. to find the price elasticity of different services. Also, marketing people and managers in general demand this kind of data.

Because of the increased use and importance of measurement data, high quality and availability is of vital importance. The data quality should be improved.

The impact of the above trends is:

- The increasing volume of data generated locally implies a need for increasing local storage capacity.
- Measurement data are actually generated and stored locally. There is a need for some mechanisms to transfer data.
- High quality data are demanded in both regions and central offices to provide a countrywide view.

4.2 MEAS

MEAS (MEasurements And Statistics) is an application for processing and storing incoming traffic measurements data from different types of switches. The primary goal of the MEAS-application is to produce data on which to base decisions concerning the addition of new equipment and optimal usage of existing equipment in the telecom network.

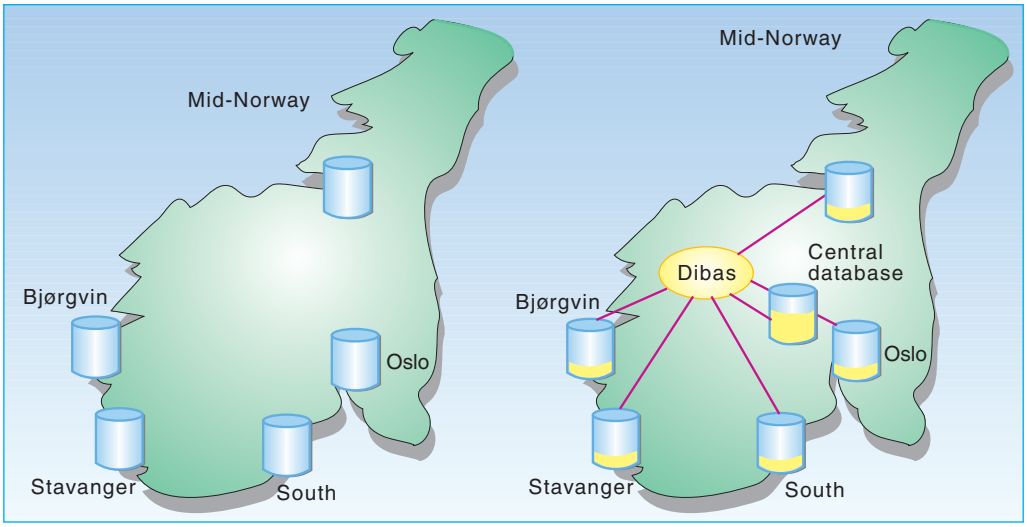


Figure 5 The MEAS system

a) The initial state consisting of "isolated" regional databases

b) Distributed database for MEAS with an additional central database

Cross border data can be exchanged between regions. Data can be distributed from the regional databases to the central database in order to provide a countrywide view of relevant data about the network and measurement statistics. Selected measurements can also be provided to the central database. The full set of measurement data is only stored locally. Replicated data have been indicated in yellow in the figure

Examples of different types of measurements:

- Traffic on trunks between switches
- Traffic on trunks to Remote Subscriber Units (RSUs)
- Traffic on trunks to Private Automatic Branch Exchange (PABXs)
- Utilisation of enhanced services
- Accounting data.

Measurements data are generated from the switches by the TMOS-system and other systems, see (12).

MEAS consists of seven isolated, regional databases. The purpose of the distributed database solution will be to provide *exchange of relevant data* between existing regional databases and also to support the establishment of a central database with a countrywide view, see Figure 5.

4.2.1 The regional structure of the organisation and its influence on the data

The network equipment is located in regions. Regional administrators are responsible for maintaining the equipment and the data about it. There are two classes of data: Network configuration data, which describe available equipment and the current configuration, and measurement data, which describe current load and performance of the equipment.

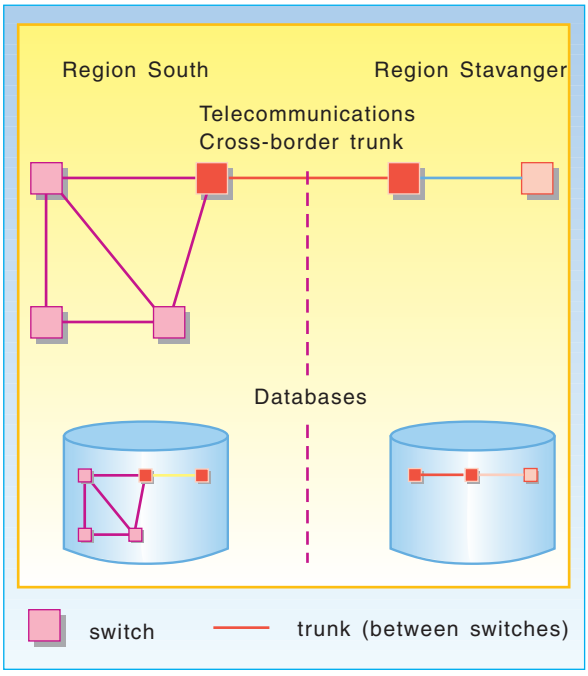


Figure 6 The telecommunications network and its database representation for two regions. Cross-border-trunks and connected switches are replicated. Local switches and trunks are stored only in the local database

In Dibas tables can be fragmented into ownership fragments to support the regional control. For example the table SWITCH can be fragmented into regional ownership fragments by using the existing relation to the table ORGUNIT (organisational units). Ownership type 3 of section 2.3, “Indirect through foreign key”, can be used to define ownership for SWITCH.

A lot of equipment data and other data, like routing data, subscriber data, and measurements are related to the switch. Therefore, in most cases, these “related” data should be owned by the owner of the switch. There is already defined a foreign key to the table SWITCH from these related data in the data model. In Dibas the ownership can be defined as “Indirect through foreign key” from the related tables to the SWITCH-table. Thus the ownership is transitively defined via SWITCH to ORGUNIT.

Note that no extension of the MEAS datamodel is necessary.

4.2.2 From regional databases to a central database and vice versa

The central administration needs copies of the regional network configuration data and regional measurement data (e.g. busiest hour per day) for the entire country in order to:

- Produce reports covering the complete network
- Check the quality of data
- Assure appropriate dimensioning of the telecom network.

Today, in order to produce the necessary reports, a lot of manual procedures must be performed. Files must be transferred from regions, pre-processed and loaded into an IBM-system called System410 (see (13)) for further processing.

The disadvantages of the current solution are (11):

- The need of manual routines and people in order to transfer and prepare MEAS data for System410. It takes a long time to get data from the regions, to produce the reports and report back to the users in the regions
- 12 % of measurements are lost.

It would be preferable if the relevant measurement data and network configuration data automatically could be transferred from the regions to the central administration without a long time delay. The statistical data could then be transferred back immediately after being produced. In Dibas the distribution of all regional data for specified tables to the central administration can be defined by using distribution type 1 of section 2.4, “Entire fragment”, and specify the central site as the recipient. This will result in central tables consisting of replicas of regional fragments.

The co-ordinator wants to order measurements on specified trunks and to get the results as soon as possible. The traffic load measurements can be returned to the co-ordinator without a long time delay. If there is an extraordinary situation, like the olympic games, national co-ordination of network traffic can be done. This can be achieved by the Dibas system in a similar manner as mentioned previously.

Domain tables are tables that specify legal values. The introduction of a central database makes it possible to have centralised control over these domain tables. If the central administration is the only one which can update these domain tables and distribute them to the regions, this ensures consistency between the domain data of all regional databases. In Dibas this is achieved for all domain tables by specifying ownership type 1, “Entire table” (owned by central administration) and distribution type 1, “Entire fragment” to all.

4.2.3 Exchange of data between regions

Some resources cross regional borders, like trunks connecting switches located in different regions, see Figure 6. In order to improve the quality of data, it is preferable to measure the load on a trunk from both switches, located in different regions. Sometimes measurements from the switches are incorrect or do not exist at all. Therefore, each region wants a copy of measurements conducted in other regions to correlate these with their own measurements on border crossing trunks.

The challenge is to distribute only relevant data between neighbouring regions. The relevant table fragment is usually a very small subset of the ownership fragment, so we do not want to use distribution type 1, “Entire fragment”. This neighbour distribution is illustrated for the TRUNK-table. An additional table, TRUNK_DIST, which contains data about distribution of TRUNKs must be created, and makes it possible to specify distribution of individual trunks to selected recipients. The individual distribution is achieved for the TRUNK table by distribution type 2, “Indirect through foreign key” from TRUNK to TRUNK_DIST. The TRUNK_DIST table is initialised with distribution data about the border crossing trunks.

Other “related” tables, describing measurement on TRUNKs, etc., should also be distributed to the neighbouring region. This can also be achieved by distribution type “Indirect through foreign key” from the measurement table to TRUNK_DIST.

4.3 TRANE

The objective of the TRANE (traffic planning) application is to support traffic planning for the telecommunication network in regions, as well as countrywide. The system contains historical, present, and prognostic data. TRANE is initially a centralised database. The purpose of distributing the centralised database is to achieve local autonomy, better performance and availability, see Figure 7.

Concerning the distribution, there are very few differences between MEAS and TRANE but TRANE needs distribution of planning data instead of measurements data. The central administration wants read access to regional data describing the network and regional plans. There is a need for exchanging border crossing data. Most of the necessary ownership and distribution information is easily derivable from existing data. Quite similar ownership and distribution rules as in the MEAS case can be used to distribute TRANE. Domain data should be distributed from central to regional databases in the same way as shown in the MEAS case.

5 Conclusion

The Dibas DDBMS will be useful to database applications with some of the following characteristics:

- The organisation wants to have local control.
- Responsibility for maintaining data is clearly defined and distributed to organisational units at different sites. It will then be possible to define ownership fragments for tables.
- Large volumes of input data are generated locally.
- The transaction load can be divided on several sites (parallelism).
- There is a need for partial replication of the data. The data volume of a local database will be significantly less than the volume of a centralised database having all data, and therefore performance can be improved.
- Users have local area network connection to the local database while having wide area network connection to a centralised database.
- The existing databases are distributed but they lack suitable interconnection. This means that the databases have been installed and that administration of the databases has already been established. Cost and training overhead will then be reduced.

In chapter 4 the distribution requirements of two applications at Norwegian Telecom have been described. Dibas seems to fulfil the requirements for a distributed database for the MEAS and TRANE applications. Interconnection of databases (MEAS), increased performance (TRANE), local autonomy (MEAS and TRANE) and improved data quality (MEAS; correlation of measurements, and maintenance of domain data on one site) can be achieved. The following Dibas mechanisms are useful:

- *Horizontal fragmentation,* because some tables, e.g. switch and trunk tables, will need locally stored partitions in each region
- *Local control of updates* on horizontal fragments, because each region has responsibility for and will update their own data exclusively

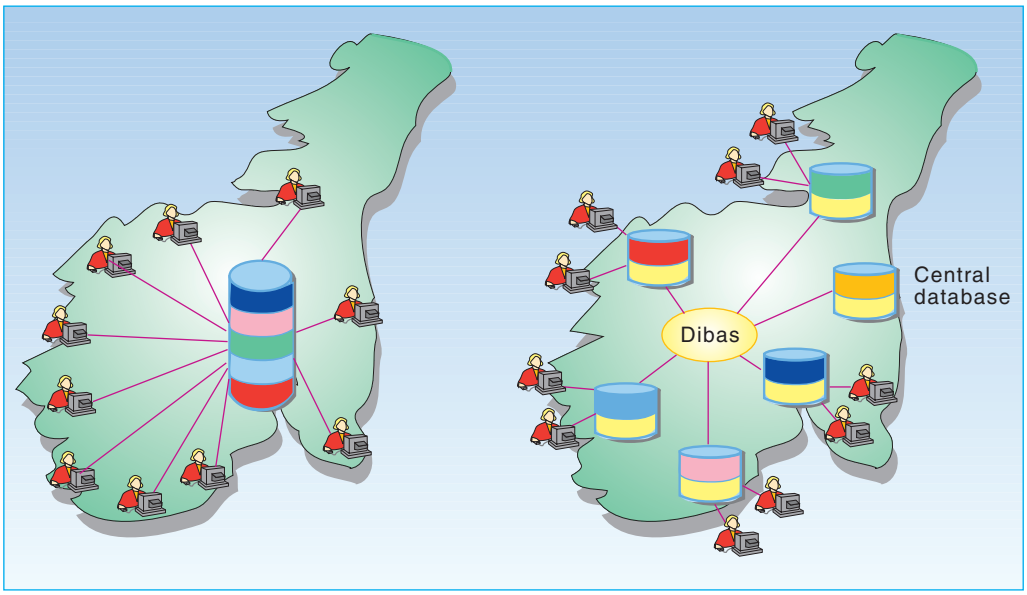


Figure 7 The TRANE system

a) The initial state

b) Distributed database for TRANE

Performance will be increased because the total transaction load will be divided on all the regional databases. An additional improvement results from significantly less amount of data in each regional database than in the initial centralised database. The total set of data will be partitioned into regional parts and allocated to the regions where they belong. The amount of replicated data is small compared to the owned data in each region

- *Replication of fragments or parts of fragments which are owned by other database sites,* because the users/-applications want
  - . Data from the regional databases available in the central database
  - . Data from the central database available in the regional databases
  - . Exchange of cross border data between regional databases.

Modifications to the application database to implement ownership and distribution are not needed. The definition language DSQL is well suited.

References

1 Risnes, O (ed). *TelSQL, A DBMS platform for TMN applications.* Telstøtt project document, March 14, 1991.

2 Ceri, S, Pelagatti, G. *Distributed databases, principles and systems.* New York, McGraw-Hill, 1986.

3 Stonebreaker, M. *Future trends in database systems.*

4 Date, C J. *An introduction to database systems,* vol 1. Fifth edition. Reading, Mass., Addison-Wesley, 1990.

5 Garcia-Molina, H, Kogan, B. Achieving high availability in distributed databases. *IEEE Transactions on Software Engineering,* 14, 886-896, 1988.

6 Pu, C, Leff, A. Replica control in distributed systems: An asynchronous approach. *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data,* 377-423, October 1992.

7 *Sybase Commands Reference,* release 4.2, Emeryville, Sybase Inc., May 1990.

8 Didriksen, T. *Dibas System Documentation: Part 3. Implementation of Run Time System.* Kjeller, Norwegian Telecom Research, 1993. (TF paper N15/93.)

9 McKeney, J, McFarlan, W. The information archipelago – gaps and bridges. *A paper of Software State of the Art: Selected Papers, by T DeMarco and T Lister.*

10 Korth, Silberschatz. *Database system concepts.* New York, McGraw-Hill, 1986.

11 Nilsen, K. *Trafikkmålinger og trafikkstatistikk i Televerket,* 15 Jan 1993. (Arbeidsdokument i Televerket.)

12 Johannesen, K. Network Management Systems in Norwegian Telecom, *Telektronikk,* 89(2/3), 97-99, 1993 (this issue).

13 Østlie, A, Edvardsen, J I. Trafikkmålinger i telefonnettet. *Vektern,* 1, 1990.

Data design for access control administration

BY ARVE MEISINGSET

681.327.2

Abstract

The purpose of this paper is to

- *illustrate use of the draft CCITT Human-Machine Interface formalism (1)*
- *illustrate some of the considerations undertaken when designing HMI data*
- *present a specification of an advanced Access Control Administration System (1)*
- *compare this with alternative models for Access Control.*

The messages conveyed in this paper can be summarised as follows:

- *The HMI formalism provides more syntactical and expressive power and precision than many alternative formalisms.*
- *The HMI formalism is used to define data as they appear on the HMI, and this should not be confused with similar, e.g. internal or conceptual, specifications of databases.*
- *Data design is a complex task to undertake; alternative designs have great consequences to the end users; to foresee these consequences requires much experience, and you have no certain knowledge on how much the design can be improved.*
- *Current models for access control have definite shortcomings, and improvements are proposed.*

The proposed specification is an extract of (1) and is an extension of the Access Control Administration system ADAD, which has been used during more than ten years within Norwegian Telecom.

Data design

(1) contains an Annex providing a method and practical guidelines for data design for an application area. These issues will not be presented here. Rather, the development of an Application schema for Access Control Administration will be used to exemplify data design.

The specification of the Access Control Administration application area is based on the notion of an access control matrix. See Figure 1.

The matrix can be constructed by using Users corresponding to lines, Resources corresponding to rows and Access rights as the elements of the matrix. The elements indicate relationships between particular Users and Resources. This is shown in Figure 2. See notation in (1, 2). An example access matrix is depicted in the Population. The object classes for the access matrix are shown in the Schema. We observe that the relational object class Access right serves as the class for all the matrix elements. Note that there is no need for Access rights for non-existing elements, as would have been the case in a real matrix. If only one Access right between one User and one Resource is wanted, this uniqueness requirement will have to be explicitly stated. Note that cardinality constraints would not do for this. However, if one Access right can refer to only one User and one Resource, this can be stated by using the cardinality constraints (1,1) associated to the User and the Resource references, respectively.

We also observe that in this simple access matrix all Users, all Resources and all Access rights have to be globally and uniquely identified independently of each other. In particular this is unfortunate for the Access rights, as this will impose a vast amount of administrative work to assign identifiers, insert the Access rights and to state references to and from Users and Resources. Therefore, the Access rights could have been replaced by two-way references between Users and Resources. However, this will not allow attributes to be associated with the Access rights.

Since an Access right can only exist if a corresponding User exists, we can avoid the cumbersome administration of globally unique Access right identities by defining Access rights locally to the Users. This is shown in Figure 3. We see that the Access right identities can be reused within different Users. Also, since the number of Access rights per User is expected to be small, usually ten or less, the Access right identities can be dropped altogether. This requires that the administrator must be able to point at the appropriate Access rights at his HMI. This way, we have got a polarised (non-symmetrical) access matrix, which allows one Access right to refer to several Resources. This makes the matrix very compact, which means that the amount of data to read and to write will be small and efficient to the access control administrator.

We believe that many Users in an organisation will often have the same Access rights. Then it will be very inefficient to insert similar Access rights to each of these Users. Therefore, we can create groups of Users having the same Access rights. This is achieved by a two-way reference from Users to Users. The Member users inherit all rights from their Set of users.

Similarly, the Resource side of the matrix can be compressed by introducing groups of Resources. The compression is introduced by recursive references in Figure 4. The figure is extended from the previous ones to illustrate the compression.

A two-dimensional matrix is a too simplistic model for a realistic Access Control Administration system. Already we have allowed a fan out of references to several Resources from one Access right. This will be used to state that access is granted to the disjunction of the listed Resources, i.e. access is granted to each of the listed Resources from this Access right.

However, we also need mechanisms to state that access is granted to a conjunction of Resources. In particular this is needed in order to state that a User can

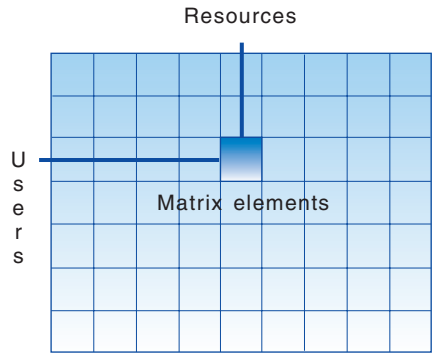


Figure 1 Access matrix
Each filled in matrix element states that a User is granted access to the appointed Resource

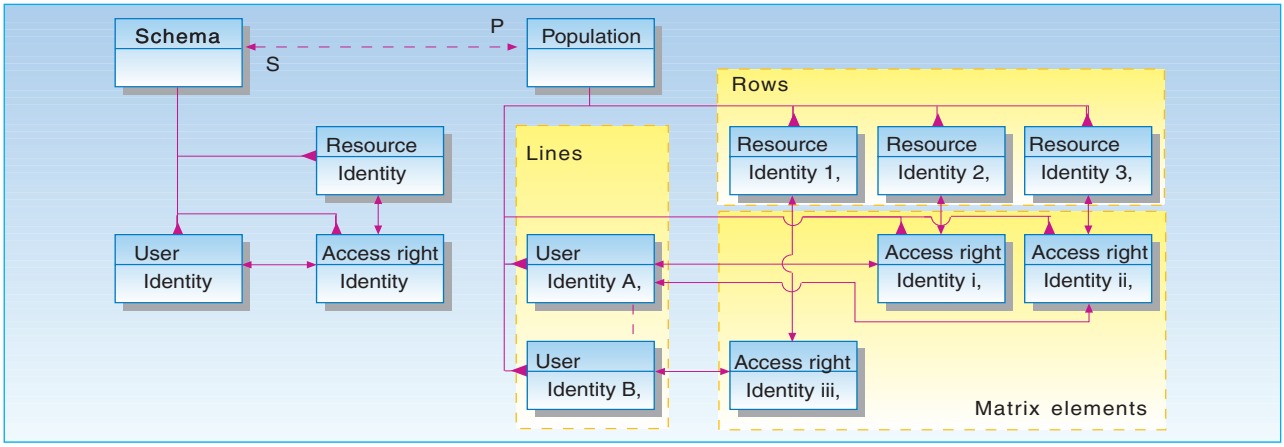


Figure 2 Formal specification of an Access matrix
The Schema states the permissible structure of all 'access matrix' Populations. Access right corresponds to the matrix element. However, several Access rights are allowed between a User and a Resource

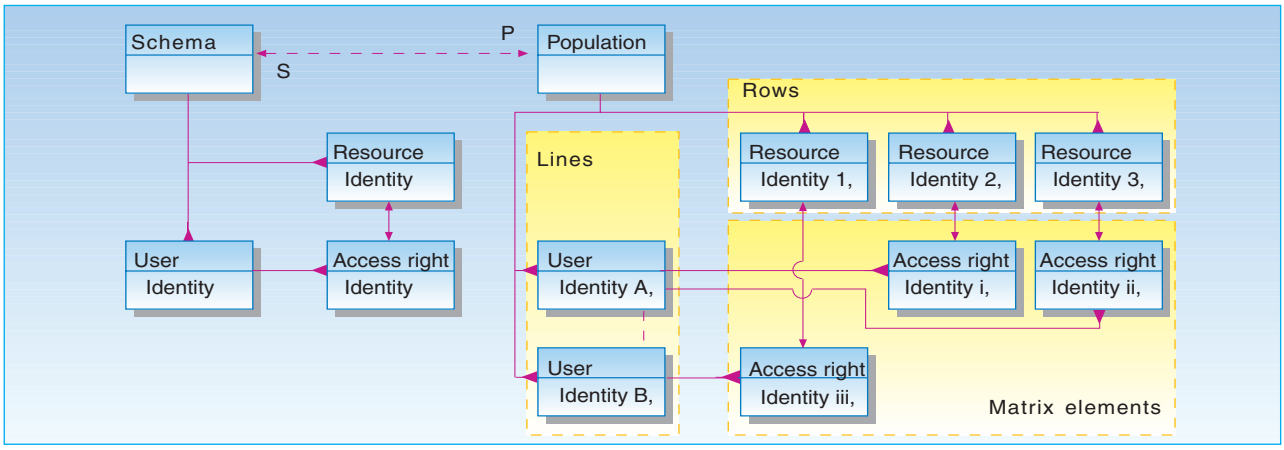


Figure 3 Access rights contained locally in Users
Access right has no more a globally unique identifier locally to Schema. Except from the change of name scopes, this specification allows the same statements as in Figure 2

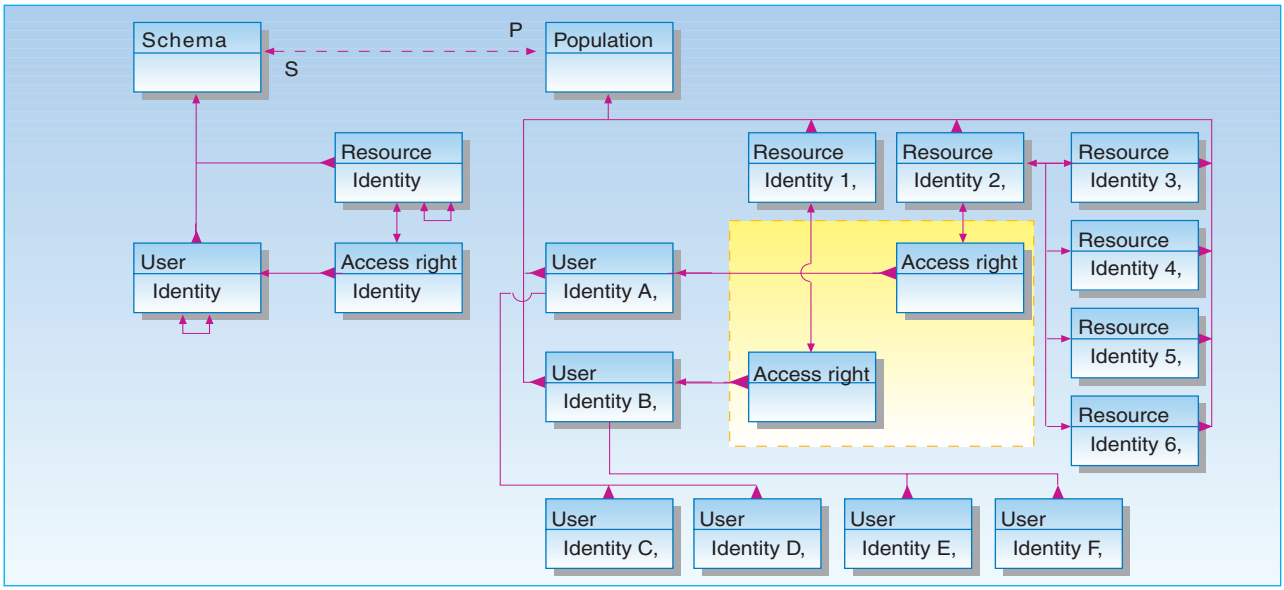


Figure 4 Compressed access matrix
Users and Resources are assembled into superior Users and superior Resources. Access rights stated for the superior items are inherited to the subordinates. This provides a compact matrix and efficient administration

only apply a certain external Function, e.g. screens, or set of Functions to a certain set of data. This set of data we call a Data context. Note that a Data context is identifying a set of Population data. This set can be whatever collection of data for which users need a specific access right. The set can be all data of a certain class, some instances of a class, some attributes of some object classes, etc. A three-dimensional polarised fanned out access matrix allowing to state access rights to both disjunctions of similar resources and conjunctions of dissimilar resources is shown in Figure 5. A formal specification is shown in Figure 6.

Lastly, we will allow Users to be defined locally to Users in a hierarchy. This is provided by the two-way schema-population reference (S-P) between subordinate User and superior User. A similar hierarchy is provided for Functions. There are also many other aspects of the final specification which we have no room to discuss in this paper. However, the resulting specification of Example Recommendation YYY on Access Control Administration follows below. This specification is organised into the following sections:

- Application area
- Application schema graph
- Textual documentation
- Remarks.

Roles and references are explicitly stated in the Application schema graph. The Textual documentation includes the data tree only, without references, as they are only expressed in the natural language text, which is left out here, due to lack of space. For the same reason, the Remarks are left out. Note that Time interval appears in several of the object classes. The specification could have been made more compact by the use of inheritance, i.e. a schema-reference, from a common source. However, this inheritance may not be of relevance to the end user or expert user reading the specification.

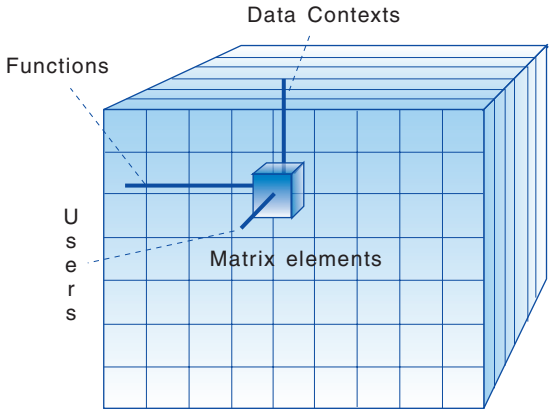


Figure 5 Three-dimensional Access matrix
The resources are split into external Functions and Data contexts. This allows for stating that a certain User can only access a Data context by using certain Functions. Arbitrary access is disallowed

Example Recommendation YYY

Access control administration

Application area

Access Control is the prevention of unauthorised use of a resource, including the prevention of use of a resource in an unauthorised manner.

Access Control Administration, as defined here, is delimited to the administration of the access rights of the users.

This recommendation is concerned with the access to those resources which are defined in the External layer, Application layer and the Internal layer of the Human-Machine Interface Reference Model (1, 3).

This recommendation covers Access Control Administration for all kinds of users, including human beings and computer systems.

This recommendation views the resources from the perspective of an access control administrator. The resources are identified within the scope of one Access Control Administration system only. The resources can be related to one application, a part of an application or several applications. This recommendation does not cover the co-ordination of several Access Control Administration systems. However, how this can be achieved is indicated in some of the Remarks – which are left out in this extract.

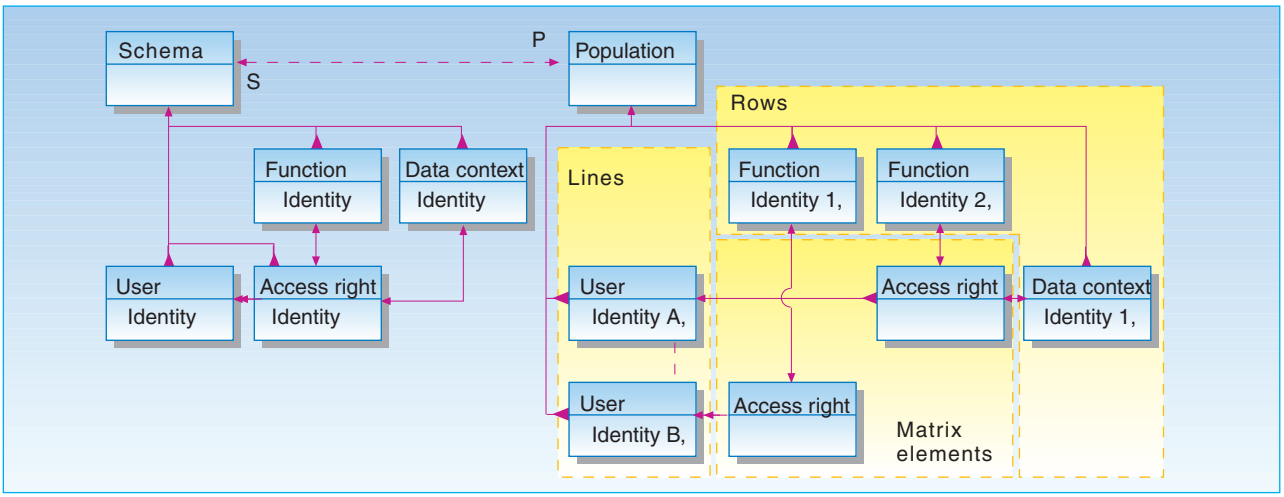


Figure 6 Specification of access rights to conjunctions of Functions and Data contexts
This is a formal specification of the three-dimensional Access matrix in Figure 5. Several Access rights connecting a User and a particular resource are allowed

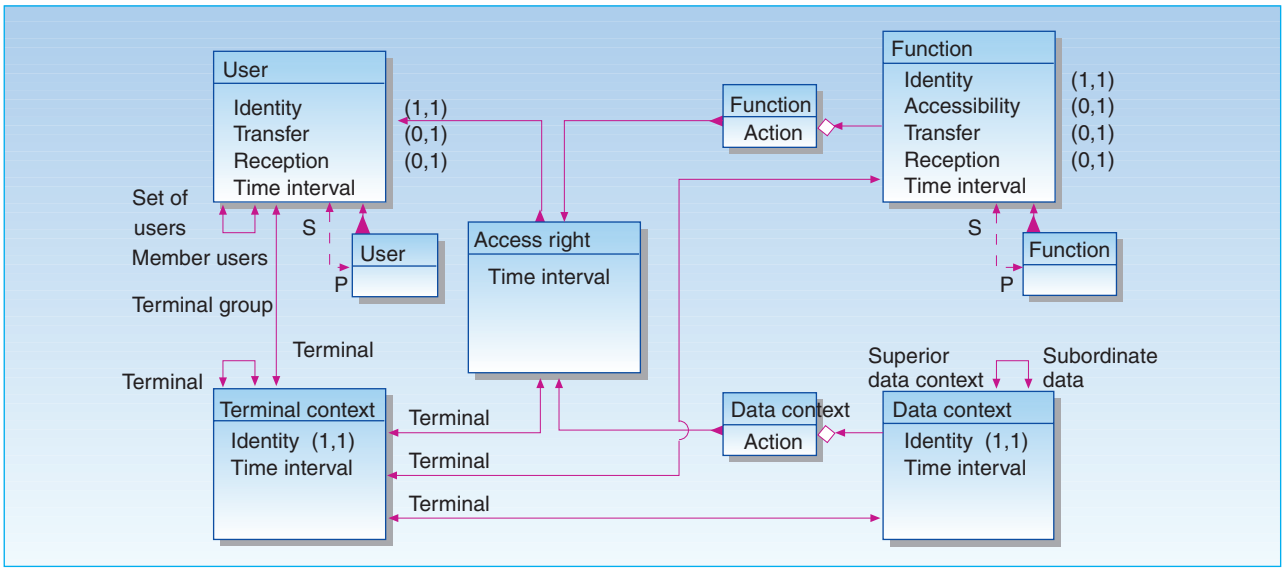


Figure 7 Application schema graph for access control administration
Subordinate User inherits (S-P) attributes from superior User. This allows for instantiation of a directory tree of Users. Similarly for Functions. One User and one Data context may be grouped into several Set of users and Superior data contexts, respectively. These groupings allow for inheritance of Access rights, but are independent of name scopes. Note that inheritance of Access rights is outside the scope of the Access Control Administration system. It concerns the Access Control system only and is therefore not stated here. The administrator has to know the implications for the Access Control system, but the inheritance statement is outside his scope. This may or may not be the case for inheritance of a common definition of Time interval, as well. Note that the class label Function is re-used for several purposes. Also, that permissible Actions can be associated with the Function and Data context roles. For further explanation of the figure, see (1). Note all the details provided in this very compact specification

Application schema graph

The application schema graph is shown in Figure 7.

Textual documentation:

User
. Identity (1,1)
. Transfer (0,1)
. . Yes, (0,1)
. . No, (0,1)
. . Possible, (0,1)
. Reception (0,1)
. . Yes, (0,1)
. . , (0,1)
. Time interval
. . Start (0,1)
. . . Year (1, 1)
. . . Month (1, 1)
. . . Day (1, 1)
. . . Hour (1, 1)
. . . Minute (1, 1)
. . . Second (1, 1)
. . Stop (0,1)
. . . Year (1, 1)
. . . Month (1, 1)
. . . Day (1, 1)
. . . Hour (1, 1)
. . . Minute (1, 1)
. . . Second (1, 1)
. User
. . User (0,1)
. Member user
. Set of users
. Terminal
. Access right
. . Time interval
. . . Start (0,1)
. . . . Year (1, 1)
. . . . Month (1, 1)
. . . . Day (1, 1)
. . . . Hour (1, 1)
. . . . Minute (1, 1)
. . . . Second (1, 1)
. . . Stop (0,1)
. . . . Year (1, 1)
. . . . Month (1, 1)
. . . . Day (1, 1)
. . . . Hour (1, 1)
. . . . Minute (1, 1)
. . . . Second (1, 1)
. . User (0,1)
. . Function
. . . Action
. . . . Read, (0, 1)
. . . . Insert, (0, 1)
. . . . Modify, (0, 1)
. . . . Delete, (0, 1)
. . . Access right
. . Data context
. . . Action
. . . . Read, (0, 1)
. . . . Insert, (0, 1)
. . . . Modify, (0, 1)
. . . . Delete, (0, 1)
. . . Access right
. . Terminal
Function
. Identity (1,1)
. Accessibility (0,1)
. . Public, (0,1)
. . , (0,1)
. Transfer (0,1)
. . Yes, (0,1)
. . No, (0,1)
. . Possible, (0,1)
. Reception (0,1)
. . Yes, (0,1)
. . , (0,1)
. Time interval
. . Start (0,1)
. . . Year (1, 1)
. . . Month (1, 1)
. . . Day (1, 1)
. . . Hour (1, 1)
. . . Minute (1, 1)
. . . Second (1, 1)
. . Stop (0,1)
. . . Year (1, 1)
. . . Month (1, 1)
. . . Day (1, 1)
. . . Hour (1, 1)
. . . Minute (1, 1)

. . . Second (1, 1)
. Function
. . Function (0,1)
. Terminal
. Function
Data context
. Identity
. Time interval
. . Start (0,1)
. . . Year (1, 1)
. . . Month (1, 1)
. . . Day (1, 1)
. . . Hour (1, 1)
. . . Minute (1, 1)
. . . Second (1, 1)
. . Stop (0,1)
. . . Year (1, 1)
. . . Month (1, 1)
. . . Day (1, 1)
. . . Hour (1, 1)
. . . Minute (1, 1)
. . . Second (1, 1)
. Subordinate data
. Superior data context
. Terminal
. Data context
Terminal context
. Identity (1,1)
. Time interval
. . Start (0,1)
. . . Year (1, 1)
. . . Month (1, 1)
. . . Day (1, 1)
. . . Hour (1, 1)
. . . Minute (1, 1)
. . . Second (1, 1)
. . Stop (0,1)
. . . Year (1, 1)
. . . Month (1, 1)
. . . Day (1, 1)
. . . Hour (1, 1)
. . . Minute (1, 1)
. . . Second (1, 1)
. Terminal
. Terminal group
. Access right
. User
. Function
. Data context

Bell-LaPadula

The Bell-LaPadula model (4) for access control introduces the notions of Clearance and Classification: Users are assigned a Clearance level. Resources are assigned a Classification level. If the Clearance is greater or equal to the Classification, the user is granted access to the resource.

This model can be interpreted using the Application Schema in Figure 7. Some

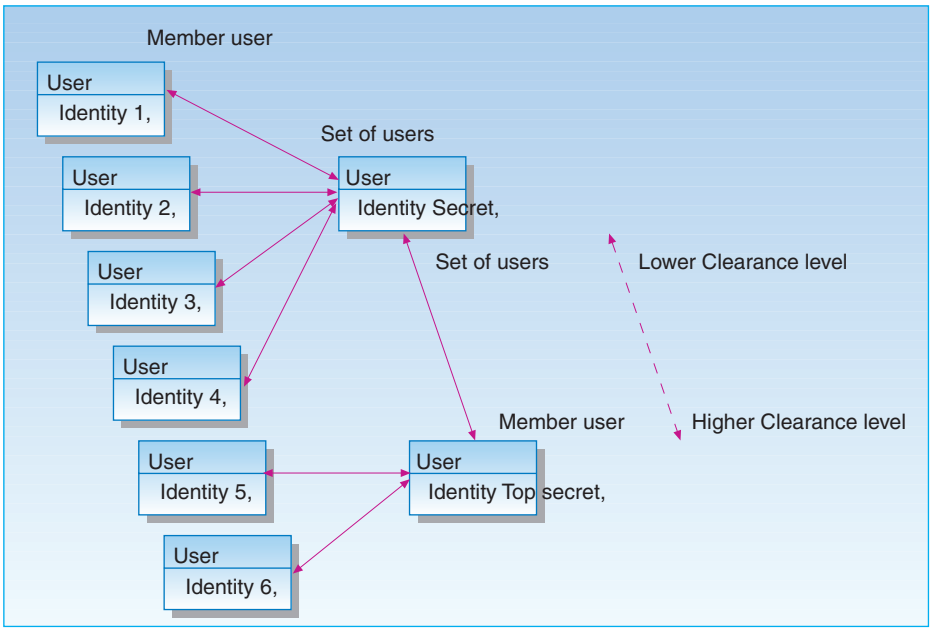


Figure 8 Ordering of Clearance levels
The two-way reference Set of users/Member user replaces the Clearance levels. In fact, the User notion is a more general and flexible way to label user groups. The use of subordinate User is not illustrated here

instances of the user side are illustrated in Figure 8 . A 'Member user' inherits all the access rights of its 'Set of users' in addition to its own access rights.

Note that a hypothetical introduction of an extra attribute 'Clearance level' of 'User' would add no extra information, since the ordering of the values still had to be defined to the Access Control system and to the Access Control Administration authority. 'Clearance levels' correspond to the Identities of the Users. Introduction of an extra attribute for this would imply introduction of an alternative and redundant identifier, and nothing else.

From the above, we realise that exactly the same kind of reasoning applies for the assignment of sensitivity Classification levels to the resources, e.g. to Data context and Function. Hence, we do not repeat this argumentation, but conclude that sensitivity Classification level is a subset of the resource Identity values.

How Clearance levels relate to Classification levels is depicted in Figure 9.

We realise that the Clearance-Classification notion is just a trivial special case of the Member user and Subordinate data context notions. The User and Data context notions are much more powerful, as they allow inheritance from several superiors and contain several subordinate items. Also the User notion can be used to denote real organisation units, rather then the artificial rewriting into "Clearance levels". Similarly for the resources – Data context and Functions. Also the Access right notion can state more complex references between the Users and the Resources.

In particular the Clearance-Classification notion lacks the needed expressive power to arrange the Actions Read, Insert, Modify, Delete, etc. into appropriate sensitivity Classes. For instance, a User may have access rights to Insert data into a Data context, but not to Delete the same data. Another User may have rights to Delete the same data, but not to Insert new data. Hence, the different Actions cannot be arranged in a Classification hierarchy. Neither do we need the notion of sensitivity "Classification levels", since we can express the access rights directly from Users to the Resources, including statements about which Actions can be applied.

Finally, the Clearance-Classification notions introduces a strict and static numbering of levels which can be cumbersome to change when needed.

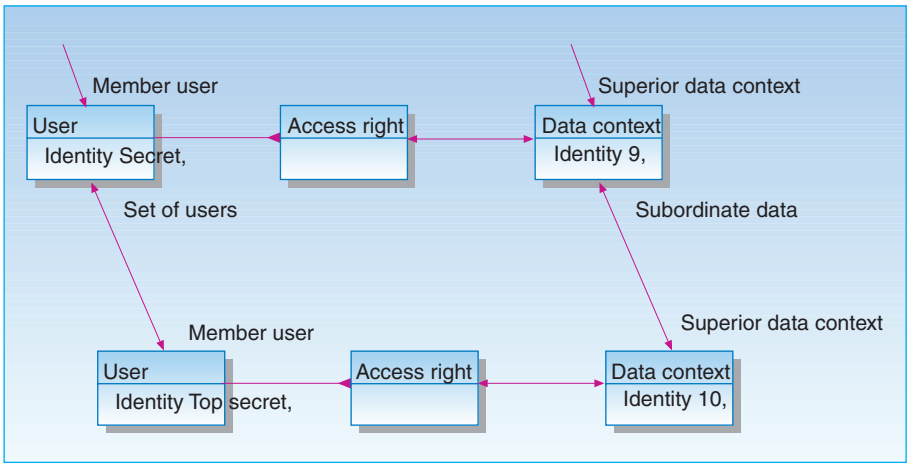


Figure 9 Access rights
This figure exemplifies how Access right statements replace the comparison of Clearance and Classification levels. Again, the mechanism is more general and flexible than the Bell-LaPadula model

Objects and attributes for access control

An interpretation of (5) is depicted in Figure 10. This is a simplification of (5), as a lot of inheritance is used to accomplish the shown structure. Attributes of subclasses of initiators are included in initiators.

Also, the specification in (5) is difficult to read, as it is spread over many sections (for the same item) within the document with references between them and out of the document. The notation is overloaded with technicalities which do not contribute significantly to the specification. Figure 10 uses the HMI graphical notation, as the OSI management formalism provides an alphanumeric notation only.

The specification (5) allows only the binary path via itemRule (similar to Access Right) between initiators (i.e. set

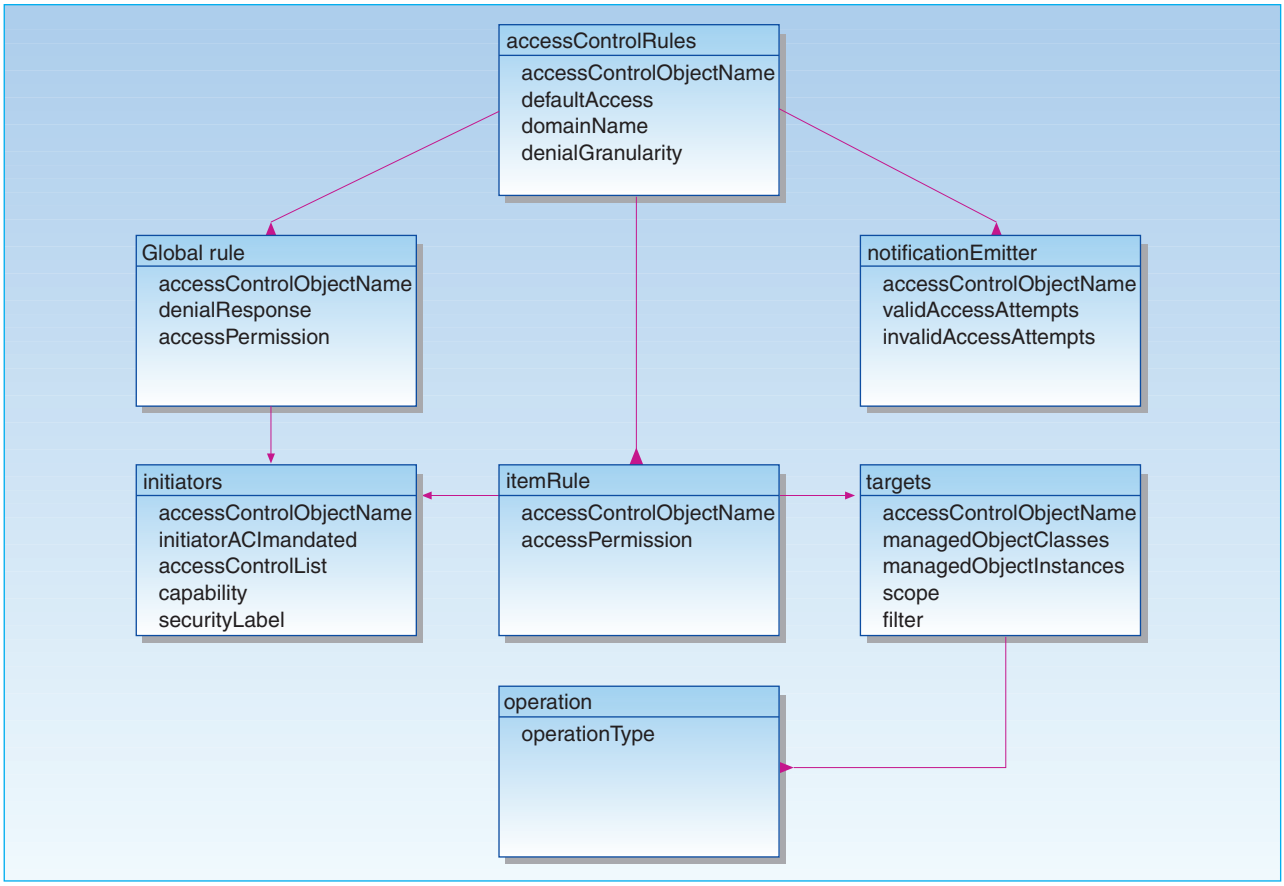


Figure 10 Objects and Attributes for Access Control
Here, only a two-way reference, i.e. a binary relationship, is provided between initiators and targets. Figure 7 allows for much more, provides more carefully designed name scopes and efficient administration

of users) and targets (i.e. set of resources). However, there is a lot of intricacy about this 'relation'. The references to initiators and targets are accomplished by the multi-valued attributes initiatorList and targetsList (not shown), respectively. Hence, each itemRule can connect several initiators and targets items. Each itemRule maps to the accessPermission attribute values deny or allow. This makes it difficult to overview the implications of the total set of maybe conflicting access right statements. Each itemRule has to be assigned a globally unique name.

In addition, globalRule has an accessPermission attribute which allows access from the referenced initiators to all targets to which access are not explicitly denied.

There is no explicit reference from initiators and targets to itemRule. The permissions of one initiators item can only be found by searching for it in the initiatorList of all itemRules. However, inclusions in the initiatorList and targetsList require that the referenced initiators and target items already exist.

Individual users are listed in accessControlLists of initiators. There is no means to list all initiators in which a user is listed, except via a global search of all initiators for the given user name in the accessControlList.

Also, the resources are listed within lists in targets. It is not clear how managedObjectClasses and managedObjectInstances relate to each other, i.e. how you specify certain instances of certain classes only. This may be done (redundantly?) via the Filter.

The Filter attribute references a filter name. The contents of the filter can be a complex ASN.1 expression.

There are no means to state access rights for individual users or resources without creating an initiators and/or targets item. Also, the individual users and resources are assigned globally unique names, and there are no means to define directory trees, allow delegation of rights or to inherit rights from other users or resources.

For each target item a finite set of operations can be specified. This means that if different users will have permission to perform different operations on the same resource, then different targets have to be defined for this resource.

No means is provided to state contextual access rights, e.g. that a particular user is granted access only if he is using a certain external function (e.g. program), a certain terminal and if the access is undertaken within a certain time interval.

Figure 10 can be interpreted to define data as they are communicated from an Access Control Administration system to an Access Control system. Ref. the notion of an Access Control Certificate (5). There seems to be no simple means to select the data which are needed for validating the access rights for a particular request. Also, the data definitions seem to be neither appropriate nor efficient for prescribing the data as seen by an access control administrator at his HMI.

A model of authorisation for next-generation database systems

(6) presents a specification which is definitely more developed than the previous two. However, this specification has also definite shortcomings.

The first problem is that (6) is based on Set theory, while the proposed specification in the current paper is based on lists within lists. Set theory disregards the details of data design and disallows many of the designs used in this paper. This paper is concerned with the exact design of data, their labels and identifiers, to make them manageable to the end users – in this case the access control administrator.

A second problem is that (6) is based on the algebraic theory of lattices. As explained in the section on the Bell-LaPadula model, this kind of ordering is too restrictive and artificial in an advanced access control administration system.

Concluding observations

This paper indicates how a rather advanced specification of Access Control Administration can be developed, taking the primitive notion of an access control matrix as the starting point. We observe that seemingly minor changes to the specification can have comprehensive consequences for the end users. Therefore, the designer should carefully figure out the implications of his design choices on the final presentations and instantiations to the end users. This study can conveniently be carried out by creating example instances in population graphs.

Experienced designers are likely to see more problems and more opportunities than novice designers. Users will have difficulties foreseeing the impact of the specifications and hence to validate them, even if the users seem to understand when the specifications are explained. The problem is to see the limitations and alternatives, which are not pointed out in the specification. Therefore, test usage of the resulting information system, allowing real users to provide feedback, is essential to identify problems and validate designs.

The specification technique allows and requires generality, precision and details which users are not likely to match in their requirement texts. Therefore, most of the data design is creative design work and is not like a compilation of end user requirements. The designers' responsibility is to create good and efficient data designs for their users. The task is not to organise existing terms only.

Also, the way the work is carried out is no guarantee for success, even if the development procedure used – the method – can provide some confidence about the result. But, even if you apply 'the same procedure as Michelangelo', it is not likely that you can produce something which matches his quality. So also for data designs. The problem is that there is no objective measure on how optimal is the current design and how much can be achieved by improvements. You only know this gain relative to the alternatives available.

It is likely that many choices will be made on subjective grounds and that they will cause disagreement. However, data designers should estimate costs of converting data to the proposed new design and estimate how much can be

gained, e.g. reduction of work, by using the new design. All concerns not quantified should also clearly be spelled out before decisions are taken for each and every data item.

The HMI formalism provides a very compact specification and provides good overview. See Figure 7. The boxes and arrows can easily be used by end users as icons to access more detailed explanations and specifications. The graph will become their 'brain map'.

Also, the HMI formalism allows usage and specification of syntaxes that are not matched by most other formalisms, e.g. the Entity-Relationship (ER) formalism and the OSI Management formalism. We have seen a tendency to use ER-diagrams as illustrations (bubble charts) rather than formal specifications. It is essential that knowledge is developed for making professional data designs. To design data for HMIs is more demanding than designing databases, from which data can be variously extracted and presented. The HMI data design prescribes the final presentation.

The comparison between the data design for Access Control Administration of this paper and other specifications from the literature indicates that much and challenging work remains to be done.

References

1 CCITT. *Draft recommendations Z.35x and Appendices to draft recommendations,* 1992. (COM X-R 12-E).

2 Meisingset, A. The draft CCITT formalism for specifying Human-Machine Interfaces. *Telektronikk,* 89(2/3), 60-66, 1993 (this issue).

3 Meisingset, A. A data flow approach to interoperability. *Telektronikk,* 89(2/3), 52-59, 1993 (this issue).

4 Hoff, P, Mogstad, S-A. *Tilgangskontroll-policyer basert på Bell-LaPaluda-modellen.* Kjeller, Norwegian Telecom Research, 1991. (TF paper N21/91.)

5 ISO, ITU-TS. *Information Technology – Open Systems Interconnection – Systems Management – Part 9: Objects and Attributes for Access Control,* 1993. (ISO/IEC JTC 1/SC 21 N7661. X.741.)

6 Rabitti, F et al. A model of authorisation for next-generation database systems. *ACM transactions on database systems,* 16(1), 1991.