Software engineering for telecommunications : bringing research results into industrial practice

Jean-Pierre HUBAUX*
article invité

Abstract

As networks and services are becoming more and more sophisticated, telecommunications software is growing in complexity. But, despite growing industrial needs, progress towards more software productivity has been extremely slow. This paper considers the software engineering problems from an industrial point of view. The industrial environment is therefore presented in some detail, stressing the influence of major events (e.g. company mergers) on fundamental software development issues (e.g. configuration management). The evolution of this environment, caused by recent trends such as deregulation, is then described, forecasting the consequences for software development. Then some emerging software techniques are presented as applied (or applicable) to telecommunications software. The paper concludes with a discussion of several research issues that could be of interest for industry within a reasonable time-frame..

Key words : Software engineering, Telecommunication industry, Telecommunication switching, Evolution.

GÉNIE LOGICIEL POUR LES TÉLÉCOMMUNICATIONS : METTRE EN PRATIQUE INDUSTRIELLE LES RÉSULTATS DE LA RECHERCHE

Résumé

Avec le développement rapide des réseaux et des services, le logiciel de télécommunication devient de plus en plus complexe. Mais, en dépit de besoins industriels croissants, la productivité du logiciel n'a progressé que lentement jusqu'à présent. Cet article considère la problématique du génie logiciel d'un point de vue industriel. La réalité industrielle est présentée en mettant en lumière l'influence d'événements importants (par exemple la fusion d'entreprises) sur certains aspects fondamentaux du développement de logiciel (par exemple la gestion de configuration). L'évolution de cette réalité, causée par des tendances récentes telles que la déréglementation, est ensuite décrite, en détaillant les conséquences prévisibles sur le développement de logiciel. L'article se termine par une discussion de plusieurs axes de recherche susceptibles d'intéresser l'industrie dans un avenir proche.

Mots clés : Génie logiciel, Industrie télécommunication, Commutation télécommunication, Evolution.

Contents

I. INTRODUCTION

In the last couple of decades, transmission and switching techniques have made impressive progress, covering several orders of magnitude of throughput. This allows us to envisage the deployment of ubiquitous broad-

* Laboratoire de Télécommunications, Ecole polytechnique fédérale de Lausanne, Ecublens, CH-1015 Lausanne.

band networks supporting an almost unlimited range of services. Obviously, more and more software is required to monitor the equipment that constitute these networks. Now, it is well known that during the same time-frame, software engineering progress has been conversely very slow. Software engineering issues thus loom large as major hurdles to telecommunications systems development.

Manufacturers are dedicating more and more resources to software development (*). At the same time, many software issues are tackled by the research community both inside and outside the telecommunications framework. Despite these efforts, know-how transfers to industry are difficult to perform

Indeed, it seems that severe *communication* problems are preventing a fruitful dialogue from taking place among the concerned people. These problems arise not only within the research industry dialogue, but also between **network specialist** (concentrating on protocols, services, traffic engineering and the related standards), **system specialists** (focussing on performance and reliability aspects of the systems, hardware design and software architecture) and **software specialists** (mostly motivated by languages, compilers, debuggers and configuration management).

This article aims at encouraging this dialogue, notably by describing the external factors affecting software engineering practice in the telecommunications industry (Fig. 1). For this purpose, we will sometimes contrast the research community to the industrial community, although the boundary between the two is obviously not that sharp.

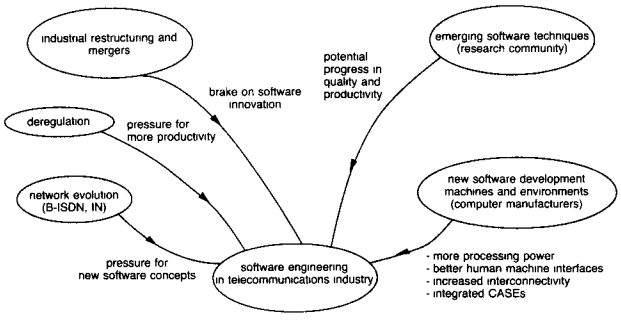

FIG. 1. — Influences on software engineering in telecommunications industry.

Les influences exercées sur le génie logiciel dans l'industrie des télécommunications.

Section 2 describes the influence of the capitalistic game on the software engineering activity and presents some fundamental software issues. Section 3 explains how recent trends like deregulation and the appearance of new network concepts (e.g. B-ISDN and intelligent network) are encouraging both software productivity and new software concepts. Emerging software techniques that could bring progress in software quality and productivity are described in section 4. Finally, section 5 discusses several research issues that could be of interest for industry within a reasonable time-frame.

Note : this paper focuses primarily on public switching networks and public switching systems (or exchanges). Nevertheless, the conclusions drawn are also applicable to packet switches and PABXs whose problems can be considered as a subset of the public switching ones.

II. INDUSTRIAL SITUATION IN THE PERSPECTIVE OF SOFTWARE ENGINEERING

II.1. Markets and companies.

For several decades, the development of public switching systems by the equipment manufacturers has mostly been carried out within the national borders of each (developed) country. This phenomenon, still perceptible today, can be explained by the dominant role played by each national operator (PTT) : not only have the national PTTs been the major customer, but, due to the high profitability of their telecommunications activities, they traditionally have contributed to the research leading to the development of new switching system architectures. Moreover, national PTTs generally have been in charge of producing the standards applicable to the equipment they purchased as well as of testing the conformity of this equipment.

This financial and technical dependence on the public operator, associated with a strong protectionism, resulted in a tradition of relatively low competition between the public switching manufacturers.

II.2. Mergers.

During the eighties, things began to change dramatically. The tremendous growth of research and development costs led companies to seek both international outlets for their products and to reach a critical size. The means used to reach these objectives have been company mergers (*).

Mergers are a way to survive when competition becomes international. But they have a severe impact on the ongoing projects of the affected companies, especially on software development. The primary reason for

(*) According to [Belanger 90a], over 60 percent of AT&T R&D employees were involved in software activity in 1989.

(*) In the following, we use the term *merger* in a very broad sense, covering also company acquisitions and joint ventures; in this context, merging means to share technical resources and is not necessarily related to the financial control.

this is that mergers and projects are not synchronous : merging decisions take place according to the company strategy and the international opportunities, while the unfolding of a project is influenced by the market, the technical constraints and the in-house resources availability. Nevertheless, projects are obviously expected to survive a merger (*).

A possible consequence of a merger is to end in an incoherent product line. In software terms, this means amongst others incompatible development methods and tools as well as different programming languages. This may lead to poor internal synergy and to excessive political power of some projects as compared with that of the company general departments (e.g. department of software tools development).

Moreover, mergers may require team restructuring, possibly entailing loss of motivation and high turnover. This may severely weaken the continuity of development methods, leading to a situation where the source code remains the only reliable information on the software. Reverse engineering is then required to restore the software documentation and eventually retrieve control of the project.

II.3. Black, white and grey software components.

Another fundamental factor that may influence a project unfolding is that nowadays the teams involved in the development are located in different sites, possibly in different countries.

This may obviously be explained by mergers, but also by licensing and by multiple applications projects. Licensing consists in selling a product as well as the related know-how and production facilities to another company (generally abroad, typically in a large developing country); the licensed company would then make further developments on this basis in order to satisfy its own market requirements; in most cases it would continue to take into account the developments realized by the licensing company.

Multiple applications projects are a way of distributing the heavy research and development costs of a given product amongst several applications. For example, a local exchange, a transit exchange, a packet switching system and an operation and maintenance Center can be (and indeed are commonly) developed from a single basic product.

For the sake of simplicity, we shall consider below the case of a site developing a product core (hardware core and system software) and periodically delivering new releases to several other sites, each of them being in charge of one application (Fig. 2).

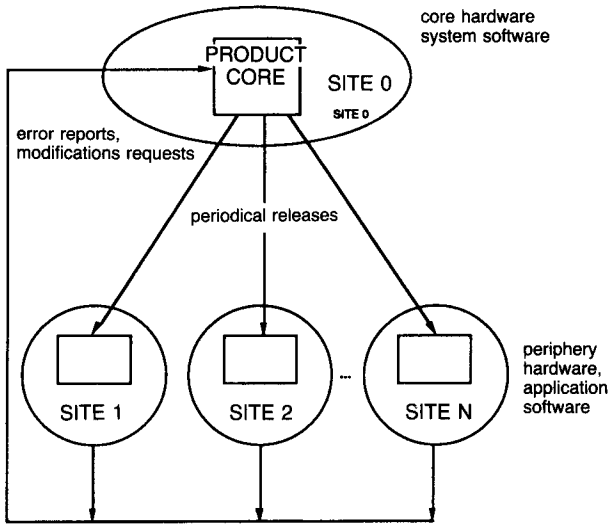

FIG. 2. — Multisite project organization.

Organisation d'un projet multisite.

As far as software development is concerned, multisite projects may entail tremendous complexity in configuration management. A few definitions are helpful to describe this problem :

— the software of the product is partitioned into several *components*; each component is typically the result of a link editing; in most cases, components communicate with each other by means of messages, under control of the real time kernel;

— the components are classified in three different categories, according to the site in charge of them : a *black component* (BC) may be modified only by site 0, a *white component* (WC) may be modified by sites 1 to N (but not by site 0) and a *grey component* (GC) may be modified by any site.

In order to illustrate the unfolding of a multi-site project, we shall focus on the relationships between site 0 and one of the application sites, say site 1. As it is very difficult to define a precise boundary between the common interest software developed by site 0 and the application software developed by site 1, people may be tempted to make use of grey modules.

But grey modules have severe drawbacks. First, they generate dangerous interference with the other applications development. Second, they dilute responsibilities, especially for quality control. Third, they entail a huge reworking overhead (Fig. 3).

The only clean way to handle grey components is to avoid them. In a real size project this is a very ambitious target, but it is worth doing. Avoiding grey components requires that the software architecture is suitable for the multisite aspect of the project. Should new (and possibly unexpected) applications appear during the project, then the architecture should be redesigned if necessary.

Making the software architecture independent of the derived applications may entail a degradation of the software performance (decrease of execution speed, increase

(*) Obviously, some ongoing projects must be abandoned after a merger.

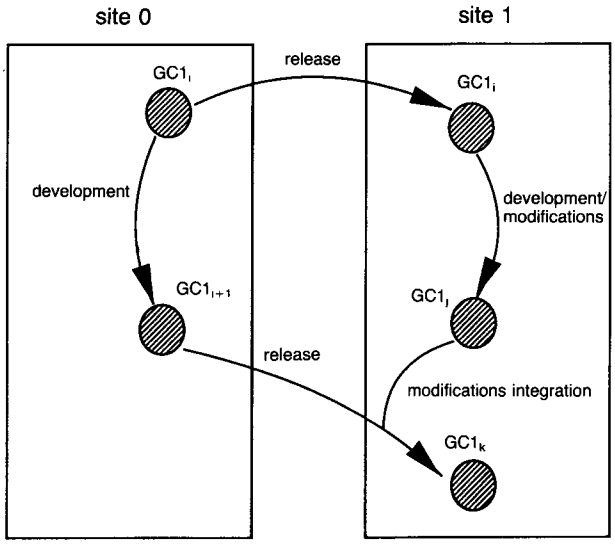

FIG. 3. — Grey modules processing.

The grey component number 1 is released at version $i(GC1_i)$ to application site 1. The development of the component proceeds then in parallel on both sites (since these developments are different, the outcoming component in site 1 is called $GC1_j$ and not $GC1_{i+1}$). When site 1 receives component $GC1_{i+1}$, the differences between $GC1_{i+1}$ and $GC1_i$ on the one hand and between $GC1_j$ and $GC1_i$ on the other hand must be manually integrated into the component $GC1_k$. This task is obviously tedious and error prone.

Traitement de modules gris.

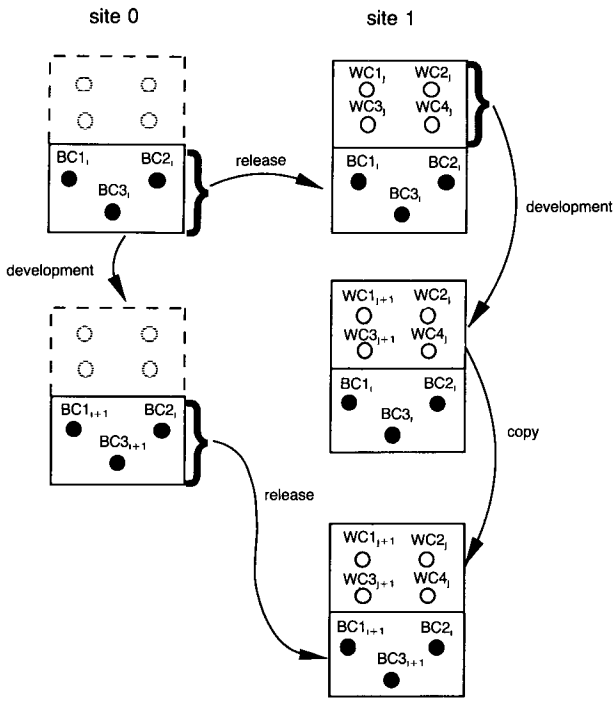

FIG. 4. — Project unfolding in absence of grey modules.

A product is assumed to have 3 black components, initially at release i ($BC1_i$, $BC2_i$ and $BC3_i$) and 4 white components, initially at release j ($WC1_j$, $WC2_j$, $WC3_j$ and $WC4_j$). During the development, some of these components are modified, and the a release numbers are increased. The figure shows the smooth integration of these developments. (White components are drawn with dotted lines in site 0 because they are used there in a very simplified version limited to testing purposes.)

Déplacement de projet en l'absence de modules gris.

of memory consumption). But these consequences must be accepted (provided they remain within reasonable limits), in the same way as nowadays the software community accepts the overhead due to a layered model, as for example the OSI Reference Model.

Once the grey modules have been avoided, the handling of the various releases can be straightforward, as shown in Figure 4. The only critical point is then the interface between black and white components. Evolutions of this interface must be admitted, motivated either by site 0 or by site 1. The latter may require new functions of the system software (e.g. new data access primitives), while the former may impose interface modifications requested by other applications.

We may conclude this point by observing that software architecture, applications and team organization are strongly interdependent. Configuration management is extremely complex for multisite projects. Still today, the market does not provide products to handle this problem. Each manufacturer has therefore developed its own tools and methods. The consequences of the proliferation of home-made tools will be discussed in the next subsection.

II.4. Typical software engineering problems.

In the previous subsections, the impacts of industrial restructuring on software development have been described. To complete the framework, we shall now consider some typical software engineering problems arising in the telecommunications industry.

II.4.1. Variety of functions expected from a system.

The tangible functions of a switching system are those related to communications processing. Extensive use of software is made to support them. Nevertheless, it is well known that protocols represent only a part of the total software; other complex functions, such as data management and defense/maintenance actually require a huge amount of software, although their processing power consumption is significantly lower.

A thorough data management is necessary because the amount of data managed by the system is very large. Moreover, the relationships between the various data can be of high complexity and the reliability constraints as well as the access methods are diverse : for example, semi-permanent routing tables require high protection against loss or wrong overwriting, while transient call related data require high access speed in both reading and writing.

Defense and maintenance are closely related to the system architecture, and are therefore defined in very vague terms in the (international) recommendations. Defense is related to the capacity of the system to react to hardware or software failures : detect the failure, save the data that may be saved, activate the back-up resource, isolate the faulty subsystem, output a message to the operator, activate the maintenance.

Maintenance is in charge of the diagnosis. In the case of a hardware failure, maintenance would identify the faulty board (if a unique identification is not possible, maintenance would indicate a set of boards); in the case of a software failure, maintenance would provide in-depth information on the system state in order to facilitate eventual debugging activities.

Hence data management and defense/maintenance represent a substantial part of the software; as their characteristics are significantly different from those of protocol software, specific languages, methods and tools may be necessary to support their development.

II.4.2. Lack of measurement techniques.

Measuring software cost and complexity is a very difficult task [Boehm 81]. The difficulty is even greater in the case of telecommunications software, because of the aforementioned multi-applications project organization and software diversity.

As long as software measuring will remain so difficult, software development planning will be a major challenge. The market pressure may convert evaluation uncertainties into a dangerous optimism of the project managers; time pressure would subsequently encourage the development teams to take methodological short cuts, eventually entailing integration problems and delivery delays.

II.4.3. Immature tool market.

In the same way as extensive use of computer aided design is used extensively for hardware development, a coherent set of tools is necessary to assist the different steps of software development :

— specification tools for both functional specifications and system (general and detailed) design,

— editors, compilers and linkers for programming and machine code generation,

— debuggers,

— and last but not least, configuration management tools.

These needs are common to the whole telecommunications software community, but until recently the tools provided by the market were not suitable for industrial development (with the notable exception of compilers for widely-used languages such as C). This situation has arisen mainly because software standards have not yet stabilized. It has led to a proliferation of home-made tools.

Due to competition fear and to marketing difficulties, home-made tools generally are not sold to other manufacturers, thus wasting research and development resources.

Home-made tools have another negative effect : they require in-house tool development professionals; later on, these professionals will be extremely reluctant to introduce tools coming from outside of the company.

II.5. Software practice.

To conclude this section, we shall briefly consider some aspects of industrial software practice.

II.5.1. Specification and design.

For specification and design, the necessity to make use of dedicated languages, backed by adequate methods, has been recognized long ago. Important efforts have been made among others by the standardization bodies who have defined formal description techniques (FDTS) like SDL [CCITT 87] [Belina 91] [Saracco 89], Estelle [ISO 89a] [Ejik 89] and Lotos [ISO 89b] [Diaz 89].

The problem is that until now the acceptance of these techniques has been very low in the industrial community. A first reason for this is that for a long time telecommunications applications specialists have been motivated mostly by applications while languages specialists have been motivated mostly by languages. A second reason is that telecommunications specialists are often conservative, maybe because of the severe reliability requirements put on the products they develop. A striking example can be found in ISDN protocol recommendation [CCITT 88b], where only a poor subset of SDL is used, moreover in an erroneous way.

The consequence is that still today specification and design are very often expressed in natural language, leading to dangerous misunderstandings. FDTS, being part of the emerging techniques, will be discussed in section 4.

II.5.2. Programming.

In the seventies, the CCITT tried to choose, among the existing programming languages, the language that could have been suitable for stored-program-control switching systems. As no agreement was reached, a special purpose working group was set up in order to devise a new language. The outcome was a sophisticated real-time strongly typed language called Chill (CCITT high level language) (*). Chill was first recommended by the CCITT in 1980, and then refined and enhanced in 1984 and 1988 [CCITT 88a]. Now, the CCITT has a lot of experience in the standardization of protocols. But standardizing a language is not the same as standardizing a protocol : a protocol conformance can be checked by the external behaviour of a product, while this is not true for a language.

Chill is a marvelous illustration of the misunderstandings that may arise between language specialists, system developers, marketing departments and licensees.

As already mentioned, the language specialists defined the Chill language, claiming that it would (or should) be **the** telecommunications high level language. Many licensees believed that Chill could be the solution to the

(*) For a brief history of Chill, see e.g. [Sammer 82]. Interesting research results on the language can be found in [Chill 90].

severe know-how transfer problems they were beginning to face in the software domain. From their point of view, a unique language could have meant homogeneous computer facilities, a reduced set of necessary tools and therefore a simplification of training programs, even in the case of purchasing from different manufacturers.

Therefore, the use of Chill became a mandatory requirement in the international tenders for switching systems. Hence, the marketing department of several large companies imposed the use of this language on the system developers who in turn required a compiler and the associated tools from their language specialists. Many Chill compilers have thus been developed and used in the last decade.

But this logic was wrong. Actually, Chill, in spite of its name, is not a language but only a recommendation. This recommendation describes a very large amount of language constructs, from which only a subset is implemented in each compiler. These subsets significantly differ from one compiler to another, which means that all Chill compilers are incompatible. The lack of success of Chill, except in the public switching community, is a consequence of this major flaw.

Many people did not understand on time that the utmost qualities of a programming language are its availability (including portability) and its industrial support, not its syntactic novelties. If a language is too specific, its spreading might well be too limited, entailing unbearable economic difficulties in developing and maintaining it satisfactorily.

Another language with a comparable level of sophistication is ADA. The great advantage of ADA, compared with Chill, is that the standardization body, the US Department of Defense, has an enormous incentive power on the research it funds and on the products it purchases. Moreover, conformance tests have been defined, allowing to certify whether a new compiler conforms to the standard or not. This rigorous strategy has opened the way to a real ADA compilers (and related tools) market, with a beneficial competition between several software houses.

Nevertheless, ADA has not yet become popular in the telecommunications industry, mostly because of competition from Chill.

Actually, the emerging language for switching systems is C. C has tremendous advantages : it is closely related to Unix (see section 4 for a discussion of operating systems), it is reasonably easy to learn, it is widespread in the computer science community and — last but not least — it is permissive, thus covering the needs of system programming. However, this permissiveness may be misused and lead to poor quality code.

II.5.3. Testing.

The aims of testing are to verify that a system (or a subsystem) is conformant to its expected behaviour. Testing raises several major problems.

First of all, it is extremely difficult to have a rigorous and methodical approach for testing if the reference (namely the specifications) is not rigorously defined. As a consequence, test cases are generated purely manually, basing solely on the developers' experience.

A second problem, closely related to the first one, is how to measure the test coverage of a given set of tests. It is well known that an exhaustive test is not a realistic target and that bugs are easy to find and to fix only when they are numerous. Hence, the testing engineers must find a compromise between the testing costs and the costs entailed by remaining bugs (e.g. penalties).

A third difficulty is to perform the tests themselves. Switching systems are primarily designed to manage communications, not to run test cases. Thus, performing tests on the target machine, taking into account the related peripherals, is generally a very tedious and time consuming task. For this reason, many tests are run on a host machine, with a reasonably user-friendly environment. The problem is then the significance of the tests themselves.

Finally, the high complexity of the relations between the software components severely impacts the testing strategy. It is extremely difficult to forecast what the consequences of the new release of a given piece of software will be in terms of correctness.

These problems have encouraged the manufacturers to adopt a very pragmatic approach for testing. The usual strategy is to test small subsets of the software on a host separately (unit tests); once a satisfactory level of confidence is reached, the subsets are progressively put together and tested under conditions that look more and more like the operational ones (integration tests).

III. THE RULES OF THE GAME ARE CHANGING

In the previous section we described the industrial factors influencing the software engineering practices. In particular, we stressed the importance of the industrial context evolution.

Many things will continue to change in the telecommunications world in the coming years. This section is an attempt to forecast the impact of these changes on the software activity.

III.1. Progress in telecommunications technique.

The telephone network is monitored by a hierarchy of switching systems, connected with each other by transmission links (Fig. 5). This hierarchy is somewhat meshed, in order to provide alternate routings in case of congestion or failure. Communications are generally circuit oriented. The allocation of the network resources to the various communications is strictly controlled by the network itself.

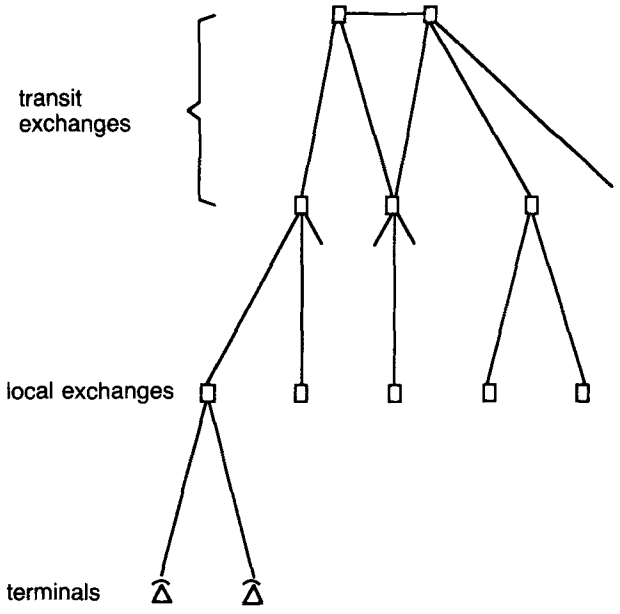

FIG. 5. — Simplified telephonic network organization.
Organisation simplifiée du réseau téléphonique.

The narrowband ISDN (N-ISDN) has been defined and is being deployed respecting this organization. The broadband ISDN (B-ISDN), based on the asynchronous transfer mode (ATM) [CCITT 90], is also strongly influenced by this philosophy.

In the meantime, computer interconnection has made tremendous progress, especially in the domain of local area networks (LANS). In a LAN, the *switching system* cannot be identified as precisely as in a telephone network. Actually, the packet routing function is the responsibility of all the devices connected to the network. The network health relies on the discipline of its users. Metropolitan area networks (MANS) are an extension of these concepts for the interconnection of computers — or rather of LANS — in a given urban region. Obviously, some modifications are necessary to allow the fair use of the same transmission medium by computers of different organizations and to guarantee confidentiality.

Metropolitan area networks will be interconnected through ATM (*). This means that two very different network architectures will closely interwork. Hence, the nature of switching will fundamentally change, strongly modifying the principles of the underlying software.

First, the influence of the computer science community on telecommunications will significantly increase.

Second, the gateways will require the development of sophisticated and performant software, in order to monitor the interworking between the different networks.

Third, the terminal equipment will play a major role in the communications control; the more teleservices and the related protocols will have to be supported, the more software will be present in the terminals [Lubich 90].

(*) DQDB, the protocol defined for MANS by the IEEE, manipulates packets compatible with the ATM cell format [IEEE 89].

III.2. Deregulation.

Several years ago, it was recognized that PTTS can no longer keep their monopolistic position. The regulating and the network operating activities should have been clearly separated, while the service provision should have been opened to competition. In Europe, these trends as well as the abolition of protectionism among the EC states are strongly supported by the European Commission [EC 87].

This evolution will obviously fuel competition, thus encouraging the PTTS to speed up their network enhancements, especially as far as new services provision is concerned. It has motivated the definition of the intelligent network concept.

III.3. The intelligent network.

The intelligent network aims at facilitating flexibility in service configuration and timeliness in service development [ETSI 90] [Robrock 91]. For this purpose, the intelligent network provides a standardized software interface, called the application programming interface (API), on which new services can be installed. The API is defined in a conceptual framework, called the intelligent network conceptual model (INCM) [ETSI 90].

The deployment of the intelligent network has two consequences. First, new players will enter the telecommunications arena in order to provide the expected services (**). Second, the manufacturers will lose their monopolistic position for service software provisioning to the public operators.

These consequences will open competition between manufacturers and software houses and hence encourage more software development efficiency. New problems like feature interaction will appear [Peter 90], requiring the identification of suitable methods to solve them.

The intelligent network trend is schematically positioned with respect to the computer interconnection trend in Figure 6.

III.4. Standardization and conceptual model.

The major changes that the telecommunications community is facing can also be perceived in the standardization process. In the realm of signalling systems, a lot of energy has been dedicated by standardization bodies to the definition of the content of the messages carried by the transmission medium. Now more and more attention is devoted to the standardization of interfaces that are internal to the systems, like the aforementioned API. This system-internal standardization process has been practi-

(**) The level of freedom that will be given to these newcomers by the network operators is still uncertain, especially for those services that may affect critical functions like billing or network security.

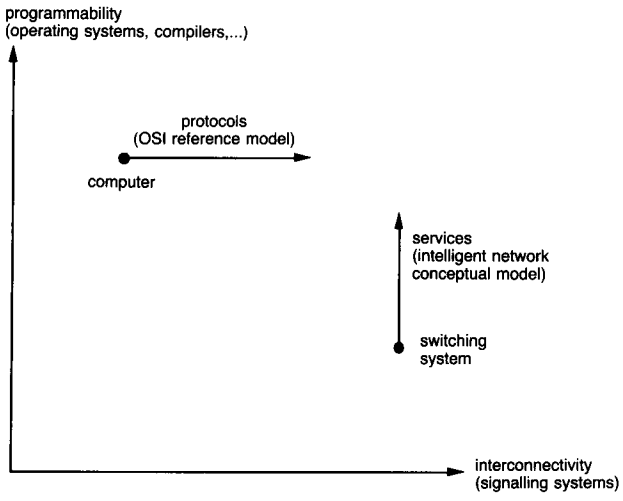

FIG. 6. — Evolution of the computer towards more interconnectivity and of the switching system towards more programmability.

Evolution des ordinateurs vers plus d'interconnectivité et des systèmes de commutation vers plus de programmabilité.

ced for a long time in the computer science community, notably for operating systems and database management systems.

In standardization bodies like CCITT, ISO and ETSI, conceptual models are devised in order to specify the framework of new network architectures, typically like the intelligent network and the telecommunications management network. Hence, conceptual models are an abstract view of functions; these functions are then implemented (at least partly) in software.

Conceptual models may reach a significant level of complexity. Therefore, they require languages to express them as well as methods to assist the process of converting them into software. An example of such a language and method, applied to the issues of services definition, can be found in [Key 90].

IV. EMERGING SOFTWARE TECHNIQUES

Having briefly described the evolution of the telecommunications world from a software engineering point of view, we shall now consider and shortly discuss software techniques provided by the research community, that are (or at least could be) helpful for industry.

IV.1. Formal description techniques.

Natural language is still today the most common way to express specifications. Nevertheless, natural language has at least two severe drawbacks : it allows ambiguities (including overspecification and contradictions) and it is not understandable by a computer.

As already mentioned in section 2, several formal description techniques (FDTS) have been defined, allowing a rigorous expression of specifications, sophisticated coherence checks, semi-automatic code generation, test case generation and conformance testing. The FDTS considered here or suitable for real time software like defense mechanisms and protocols.

It is not our intention to describe these techniques but rather to see how and when they can be used by industry. Hence, we shall now briefly discuss two of them.

IV.1.1. SDL.

SDL stands for specification and description language. It is relatively popular in the telecommunications community because its basic concepts are easy to learn and use; moreover, its graphical form (SDL/GR) is generally appreciated.

Some commercial environments [Faergemand 89] are now available on most conventional workstations. They provide interesting features like assisted editing, animation and semi-automatic code generation. Other advanced functions like dynamic verification (e.g. deadlock identification) and automatic test case generation should appear on the market in the coming years. Another interesting aspect of SDL is its object oriented extensions; these extensions are expected to be integrated to the 1992 recommendation of SDL [Moller-Pedersen 87] (*). Finally, significant progress has been made in the formal foundation of SDL during the last years [Broy 91]. Nevertheless, SDL has also its drawbacks. We mention here two of them.

The first is that SDL is based on an asynchronous communication model : processes and blocks communicate with each other by means of signals, stored in (theoretically) infinite queues before being processed. Hence, the real behaviour (and more specifically the response time) of a system modelled in this way is extremely difficult to forecast.

The second is that the data model, based on abstract data types, is not well understood by the language users, maybe because of its complexity. As a result, data are rarely correctly specified in SDL, which severely reduces the interest of the specification. These difficulties are a bit surprising, since the predefined sorts in SDL make it possible to treat data like in Pascal. Is Pascal-type data too abstract for most SDL users?

IV.1.2. Lotos.

Unlike SDL, Lotos is based on a synchronous communications model. Its sound mathematical foundation is expected to allow sophisticated controls and eventually correctness proofs.

Unfortunately, in spite of its potentialities, Lotos has not yet been completely convincing. The language seems to be hard to learn and to use, especially as far as data

(*) How manageable the underlying object model will be and what usage will effectively be done of it are still open questions.

modelling is concerned (the data model is the same as in SDL). The related tools are difficult to implement; until today, Lotos interpreters and compilers have not reached an industrial maturity. Moreover, exhaustive proof of correctness is currently out of question for real-life complex systems.

Hence, it is clear that the *return on investment* of learning and using Lotos — or an equivalent language — is not attractive for an engineer already involved in project developments. Therefore, the solution lies in enhancing engineering education and training; but this might well take a whole (human) generation.

IV.2. Protocol engineering.

With the development of networks and distributed systems, the need for communication software has dramatically increased during the two last decades, resulting in the discipline commonly named protocol enginnering [Nussbaumer 86, 91] [Tanenbaum 89]. Perhaps because of its fascinating combinatorial complexity, protocol engineering has drawn substantial attention from the research community, which has significantly contributed to the progress in this area (see e.g. [Holzmann 91] [Rudin 88]).

At the present stage, sound foundations like finite state machines and petri nets have been devised and widely used in this realm; the aforementioned FDTS are based on these foundations. Notable results have been reached in protocol specification, validation, implementation and testing. Nevertheless, the transfer of these results to industry has been until now limited, mostly because of the lack of reliable tools and appropriate education.

Protocol engineering has been strongly influenced by the OSI reference model and many efforts have been made to study this layered architecture [Bochmann 90]. Now, it is well known that a straightforward implementation of the OSI protocols may entail severe performance problems. This concern has encouraged the research of new protocol architectures, especially in the realm of broadband networks [Haas 91] [Conti 91], as well as the expression of performance constraints within FDTS [Hogrefe 91].

IV.3. Object oriented approach.

The object-oriented paradigm is a new approach for both design and implementation [Meyer 88]. It is considered to represent a significant progress with respect to the now traditional structured approach. Thanks to features like encapsulation and inheritance, developers hope to improve the situation in critical software engineering areas like software products extensions (typically due to changes of functional specifications) and software reuse.

Unfortunately, the terms *object* and *object oriented* have many different definitions that are confusing for potential users. For the sake of understandability, we shall distinguish two families of object models :

1) In the realm of knowledge engineering, objects — often called *frames* in this case — contain attributes representing the state and the behaviour of the object. Frames are generally structured in classes; these classes are generally frames themselves. Relationships between the frames can be described, entailing the concept of semantic networks. Examples of commercial products are Kee [Kee 90] and Spoke [Spoke 90]. Until recently these languages were generally Lisp-based.

2) In the realm of software engineering, objects are software components characterized by a state and a set of operations (generally called methods). Also in this case, objects are generally structured in classes. Languages of this family can be either native like Smalltalk [Goldberg 83] or obtained as an extension of an existing language like C++ [Stroustrup 86].

During the last few years, the object oriented approach has gained notable success in the telecommunications community. A review of several projects and environments, completed with a comparison of the object models used, can be found in [Lai 91].

Network management is a domain where the object oriented approach has begun to be widely used. For example, in Recommendation [ISO 90], the common management information service handles the physical and logical components of the managed network as objects. Semantic networks are used to model the links between the (telecommunications) network components. Composite objects are used to organize the managed objects in a *Management Information tree*, according to a *belongs-to* hierarchy.

In this approach, the various objects are defined by means of stepwise refinement, thanks to the inheritance mechanism. Inheritance, when properly used, allows to maintain the compatibility between software releases of two open systems evolving in a non simultaneous way; it also supports the smooth introduction of new hardware products in the network, the new products being defined as subclasses of a given standardized class (*).

Clearly, for this kind of need, object models belonging to the previously mentioned family 1 are the best suited.

As far as software development is concerned, some investigations have been made to express the design of complex telecommunications applications by means of knowledge based systems (family 1). This allows to take advantage of both the expression power of the associated languages and the user-friendliness of the underlying environment. Once the design has been performed and checked (e.g. by animation), it is then converted into a real-time implementation in order to be run on the target machine. This implementation can be supported either by a Lisp-oriented operating system [Arnold 90] or by an object-oriented programming language belonging to family 2 [Hubaux 90].

(*) In ISO, the concept of *allomorphism* is defined for this purpose.

Finally, an object oriented approach may be an efficient strategy to solve the problems raised by grey modules in multi-site projects (see section 2).

IV.4. Expert systems.

Expert systems represent the most developed branch of artificial intelligence (AI). To build up an expert system, the knowledge of a human expert is *captured* by a knowledge engineer and translated into the expert system language; generally, this language is rule-based. When the expertise is required by the user, the expert system is run : an inference engine *fires* the rules. During the last years, extensive use of expert systems has been made in a multitude of areas ranging from medical diagnosis to geological exploration.

In the telecommunications field, expert systems can be used for various purposes. The most promising field seems to be troubleshooting, for both switching system maintenance and network management. The reasons for this are related to 1) the complexity of the domain, 2) the need to make the knowledge of a small number of expert operators available to the whole operators community and 3) the necessity of cooperative problem-solving between the system (or the network) and the operator.

Many attempts have been made, mostly by operating companies and switching systems manufacturers, to implement troubleshooting functions by means of expert systems technology [Liebowitz 88]. The description of these attempts is well beyond the scope of this paper. Nevertheless, it is important to notice that — at least until now — expert systems did not permeate the telecommunications field as successfully as one could initially hope. This is mostly due to two reasons. The first is that AI languages and environments require a huge amount of computing power, entailing the need of expensive machines as well as severe performance problems for online applications. The second is the lack of methods : a set of production rules is very difficult to structure and the evolution of an expert system composed of even only 150 to 200 rules may be difficult to keep under control.

To overcome the second flaw, a promising way is to combine the production rule paradigm with the object (or frame) paradigm. Family 1 environments mentioned in the previous subsection are a first step in this direction, but a lot of theoretical and experimental work is still necessary before expert systems will be successfully and economically applied to broad industrial-size telecommunications problems.

IV.5. Relational database management systems.

Relational database management systems (RDBMS) have become very popular in many computer science applications. When switching systems developers discovered that the amount of data managed by the system was dramatically increasing with the number of connected lines and the sophistication of offered features, they began to implement RDBMS in the core of the software.

Later on, it appeared that database systems could also be used in a network context, leading to the concept of distributed database systems [Ozsu 91]. The distributed database systems technique is particularly suited for implementing the data model of complex network functions such as network management and the intelligent network [Aiken 91].

Several benefits can be obtained by using RDBMS :

— RDBMS are an efficient way to organize the complex data structures handled by telecommunications systems, like for example the translation tables of an exchange;

— by means of relational algebra, RDBMS allow advanced table processing functions, as for example the creation of different views of the data, according to the needs of the applications;

— reliable data protection mechanisms are also supported, including the coherence of data between the central memory and the disk;

— by means of access primitives, RDBMS provide an interesting degree of independence between the data organization and the application programs using those data; this independence is of invaluable help when the software evolves;

— finally, the use of RDBMS may allow the integration of market-provided database products, or at least the usage of standards like SQL.

Unfortunately, RDBMS have a major drawback : their access and control mechanisms entail severe processing overhead. This overhead can be tolerated for data management and maintenance functions but is not acceptable for stringent real-time functions like call processing (see subsection II.4. for a discussion of the variety of functions expected from a system).

The solution generally applied in the switching systems software consists in discriminating the database accesses according to the specific needs. It takes advantage of the fact that the real time constrained accesses are generally only read accesses, while the write accesses are typically provoked by an operator request and may therefore have longer response times. Figure 7 shows the principle of the solution.

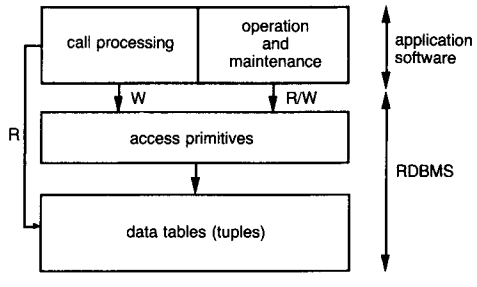

FIG. 7. — Read (R) and Write (W) accesses to cope with real time constraints.

Accès lecture (R) et écriture (W) pour faire face aux contraintes en temps réel.

Obviously this solution is more a compromise than a panacea. Making data organization directly visible to a part of the application software notably reduces the advantages of RDBMS, especially as far as independence between data and programs is concerned. Nevertheless, RDBMS are already successfully used in telecommunications systems and they will most probably play a major role in future developments.

IV.6. Operating systems.

In order to cope with the specific hardware architecture of their switching systems, the manufacturers have developed specific operating systems (sometimes called executives) on top of which they have developed specific telecommunications applications. An obvious drawback to this strategy is that it severely reduces the possibility to incorporate applications purchased on the market into the system. Another major drawback is that the portability of applications between host and target systems requires the emulation of the operating system on the host. Now, this portability is very useful, since generally unit tests are performed on the host system, while integration tests are performed on the target.

A promising way to solve this problem is to make use of a standard at the interface between the operating system and the applications. A description of such an approach, based on UNIX, can be found in [Zimmermann 91].

V. DISCUSSION

In the previous sections, we described the software engineering issues in the realm of the telecommunications industry; then we showed how these issues will evolve because of the major changes occurring in the telecommunications world; finally we presented some software techniques, provided by the research community, and used (or usable) in the telecommunications industry.

We will now discuss the behaviour of people both in research and industry, before coming back to technical issues.

Generally researchers are more encouraged to publish theoretical results than to really care about industrial needs. Especially in the domain of software engineering, researchers tend to work for researchers : they provide sophisticated solutions suited to very well educated people, free from industrial constraints such as product line coherence.

In order to make their research results easily applicable, they tend to make the following (generally implicit) assumptions about the industrial context to which these results should be transferred :

— organization stability,
— development *from scratch*,
— mono-application, mono-site project.

As described in section 2, these assumptions are rarely met in real life. Industrial engineers are generally strongly tied to the existing bulk of code and to compatibility constraints. Because of this, they normally lack *experimental courage*; so, for example, the few quantitative experiments on software productivity come mostly from universities [Kelly 90], not from industry.

Actually, the industrial software activity, especially in the telecommunications domain, is characterized by a tremendous inertia. Due to the size of existing code (which represents an enormous investment), manufacturers only rarely have the opportunity to introduce new languages, tools and methods in their developments (*). As the products are getting more and more complex, the inertia is ever increasing. Hence, the manufacturers will put more and more emphasis on pragmatism and continuity in the coming years.

The research community must take this reality into account. Researchers should not limit their efforts to the privileged moments in which manufacturers are allowed to make significant evolutions but should rather also propose (and demonstrate the interest of) innovative solutions respecting the continuity constraints. We mention here some issues that could be tackled in this framework.

As previously mentioned, the source code is often the most reliable information on a given software product. **Reverse engineering** aims at restoring usable information from the code itself; to out knowledge, until now very few research efforts have been carried out on the topic of reverse engineering applied to telecommunications software, in spite of important (and still growing) needs. An interesting approach is descrived in [Belanger 90b].

Closely related to the previous one is the **test cases generation** issue. Generally, researchers strive to automatically generate test cases from a specification written by means of an FDT, typically Lotos. This is an interesting but long term problem. A manufacturer would be much happier to know how to devise a testing strategy, how to share the testing effort between unit and integration tests, how to assist test cases generation from an already existing source code and how to measure the coverage of a given test suite.

A third example of an issue that can be developed while respecting continuity is **telecommunications software measurements**. Effective measurement techniques would provide invaluable information usable either for development cost estimations or for complexity evaluation [Yu 90]; in this way they would allow to confidently forecast the testing effort and to estimate the number of residual bugs. Unfortunately, the conventional software engineering measurement techniques, based on the

(*) The existing software is considered to have such a value that manufacturers tend to realize the hardware evolution of a given product while keeping the same code.

number of lines of code, are not suitable for telecommunications software; this is because real time software is only poorly characterized by its size (the number of exchanged messages would be a better measure) and because the number of source lines modifications notably exceeds the number of line creations. Now, Quality assurance will be meaningful only when such measurements become reliable.

Independently from the continuity constraints, two other issues worth making an increased research effort have been introduced in the previous sections.

The first is **configuration management**. As already mentioned, multi-application and multisite projects are becoming current practice, thus increasing the configuration management complexity. Obviously, this problem is not restricted to telecommunications software and some interesting solutions can be expected from the software engineering community. Nevertheless, the needs of the manufacturers are far from being satisfied today. In particular, the link between te configuration management system and each of the installed target systems is still a problematic topic.

The second is **conceptual model** support. We have already described the importance of these models in the standardization process. Object oriented techniques and similar approaches must be thoroughly investigated in order to support the related concepts and assist their conversion into operational software.

Several examples of research issues have been descrived in this section, for which the ongoing research effort seems to be limited when compared with the potential benefits that industry could expect. They share three a priori researcher-repellent characteristics : they are not very fashionable, they have scarce theoretical foundations and they require a close collaboration with industrial partners. Nevertheless, they constitute an indispensable first step before effectively tackling even more ambitious issues such as reuse and human factors in telecommunications software engineering.

ACKNOWLEDGMENT

The author would like to thank Ferenc Belina, Christian Destor, Pierre-Alain Etique, Claude Petitpierre and Eric Smith for their invaluable comments on earlier versions of this paper. Trademarks : ADA (DoD), Kee (Intellicorp), Spoke (Alcatel Alsthom Recherche), Smalltalk (Xerox), Unix (Bell Laboratories).

Manuscrit reçu le 4 octobre 1991,
article invité.

REFERENCES

[Aiken 91] AIKEN (J.), *et al.* Achieving interoperability with distributed relational databases. *IEEE Network Magazine* (Jan. 1991).

[Arnold 90] ARNOLD (E. C.), BROWN (D. W.). Object oriented software technologies applied to switching system architectures and software development process. *International Switching Symposium* (1990),Stockholm.

[Belanger 90a] BELANGER (D.), *et al.* Evolution of software development environments. *AT&T technical Journal* (March/Apr. 1990).

[Belanger 90b] BELANGER (D.), *et al.* Toward a software information system. *AT&T technical Journal* (March/Apr. 1990).

[Belina 91] BELINA (F.), *et al.* SDL with applications from protocol specification. *Prentice-Hall* (1991).

[Bochmann 90] BOCHMANN (G. V.). Protocol specification for OSI. *Computer networks and ISDN systems* (1989/90), **18**.

[Boehm 81] BOEHM (B.). Software engineering economics. *Prentice-Hall* (1981), Englewood Cliffs, NJ.

[Broy 91] BROY (M.). Toward a formal foundation of the specification and description language SDL. *Formal aspects of computing* (Jan.-March 1991).

[CCITT 87] Specification and description language SDL. *CCITT Recommendation Z.100 Blue Book* (1987), volume X.1.

[CCITT 88a] CCITT high level language (Chill). *CCITT Recommendation Z.200 Blue Book* (1988), volume X.6.

[CCITT 88b] ISDN user-network interface layer 3 specification. *CCITT Recommendation* (1988), Q.931.

[CCITT 90] Draft recommendation I.121 : Broadband aspects of ISDN, CCITT SG XVIII. *Report R.34* (June 1990).

[Chill 90] Proceedings of the 5th Chill Conference, Rio de Janeiro (March 1990).

[Conti 91] CONTI (G.). IAPA : a new protocol architecture based on interacting activities. *IEEE, Proceedings of the 16th annual conference on local computer networks* (Oct. 1991) Minneapolis.

[Diaz 89] DIAZ (M.), *et al.* (eds). The formal description technique Estelle. *North-Holland* (1989).

[EC 87] European community commission. Green Paper (1987).

[Eijk 89] EIJK (P. V.), *et al.* (eds). The formal description technique Lotos. *North-Holland* (1989).

[ETSI 90] Intelligent network : framework. *ETSI draft technical report DTR/NA-6001* (Sep. 1990).

[Faergemand 89] FAERGEMAND (O.), MARQUES (M. M.) (eds). SDL 89. The language at work *North-Holland* (1989).

[Goldberg 83] GOLDBERG (A.), ROBSON (D.). Smalltalk-80 : the language and its implementation. *Addison-Wesley* (1983).

[Haas 91] HAAS (Z.). A protocol structure for high-speed communication over B-ISDN. *IEEE Network Magazine* (Jan. 1991).

[Hogrefe 91] HOGREFE (D.), *et al.* Hierarchical performance evaluation based on formally specified communication protocols. *IEEE Trans. Computers* (Apr. 1991).

[Holzmann 91] HOLZMANN (G.). Design and validation of computer protocols. *Prentice-Hall* (1991).

[Hubaux 90] HUBAUX (J. P.), *et al.* Telecommunications object oriented prototyping and implementation environment. *Dixièmes Journées Internationales sur l'Intelligence Artificielle* (1990), Avignon.

[IEEE 89] IEEE 802.6, Proposed standard-distributed queue dual bus. *Document 802.6*, 89/45 (Aug. 89).

[ISO 89a] Information processing systems. Open systems interconnection Estelle. A formal description technique based on the temporal ordering of observational behaviour. *ISO Recommendation 9074* (1989).

[ISO 89b] Information processing systems. Open systems interconnection Lotos. A formal description technique based on the temporal ordering of observational behaviour. *ISO Recommendation 8807* (1989).

[ISO 90] ISO 9595 information processing systems. Open systems interconnection. Common management information service definition (1990).

[KEE 90] Knowledge engineering environment-reference manual, Intellicorp (1990).

[Kelly 90] KELLY (J.), MURPHY (S.). Achieving dependability throughout the development process : A distributed software experiment. *IEEE Trans. Software Engineering* (Feb. 1990).

[Key 90] KEY (M.), *et al.* Rosa : an object-oriented architecture for open services. *British Telecom Journal* (Oct. 1990).

[Lai 91] LAI (M.), *et al.* An analysis of object characterization for a video-on-demand application. *Telecommunications information networking architecture Workshop* (1991), Chantilly.

[Liebowitz 88] LIEBOWITZ (J.). Expert system applications to telecommunications. *Wiley Interscience* (1988).

[Lubich 90] LUBICH (H.). Multim ETH, a collaborative editing and conferencing project. *Computer networks and ISDN systems* (1990), **19,** n° 3-5.

[Meyer 88] MEYER (B.). Object-oriented software construction. *Prentice-Hall* (1988).

[Moller-Pedersen] MOLLER-PEDERSEN (B.), *et al.* Rationale and tutorial on OSDL : an object oriented extension of SDL. *Computer networks and ISDN Systems* (1987), **13,** n° 2.

[Nussbaumer 86] NUSSBAUMER (H.). Téléinformatique I et II. *PPUR* (1986).

[Nussbaumer 91] NUSSBAUMER (H.). Téléinformatique III et IV. *PPUR* (1991).

[Ozsu 91] OZSU (T.), VALDURIEZ (P.). Principles of distributed database systems. *Prentice-Hall* (1991).

[Peter 90] PETER (P. T.). Supporting service development for intelligent networks. *IEEE Journal on Selected Areas in Communications* (Feb. 1990).

[Robrock 91] ROBROCK (R. B.). The intelligent network. Changing the face of telecommunications. *Proceedings of the IEEE* (Jan. 1991).

[Rudin 88] RUDIN (H.). Protocol engineering : a critical assessment. Protocol specification, testing and verification VIII. Aggarwal (S.), Sabhani (K.) (eds). *IFIP, Elsevier Science Publishers* (1988), B.V.

[Sammer 82] SAMMER (W.), SCHWARTZEL (H.). Chill, Eine moderne Programmiersprache für die Systemtechnik. *Springer Verlag* (1982).

[Saracco 89] SARACCO (R.), *et al.* Telecommunications systems engineering using SDL. *North-Holland* (1989).

[Spoke 90] Spoke 3.0 Reference Manual. *Alcatel ISR* (1990).

[Stroustrup 86] STROUSTRUP (B.). The C++ programming language. *Addison-Wesley* (1986).

[Tanenbaum 89] TANENBAUM (A. S.). Computer networks (Second edition). *Prentice-Hall* (1989).

[Yu 90] YU (W. D.). A modelling approach to software cost estimation. *IEEE Journal on Selected Areas in Communications* (Feb. 1990).

[Zimmermann 91] ZIMMERMANN (H.), GUILLEMONT (M.) Chorus : the micro-Kernel based realtime distributed UNIX. *Telecommunications information networking architecture Workshop* (1991), Chantilly.