METHODS AND TOOLS FOR INFORMATION SYSTEMS DESIGN

S. KRAKOWIAK
IMAG, Université de Grenoble
B.P. 53 38041 GRENOBLE-Cedex

ABSTRACT

The purpose of this survey is to present, in a comprehensive manner, some important concepts that influenced thè design of information systems in the last few years. Emphasis is placed on recent progress in design methods, and on the development of tools that may be used to apply these methods.

Some aspects of recent computer-implemented systems for assistance to requirements analysis and system design are examined. The paper then reviews some advances in the design of data and control structures. The impact of the abstract data type concept and its use in system design is analyzed. Recent progress in the control of parallel process cooperation is finally presented, with reference to distributed systems.

1. - ARCHITECTURAL PRINCIPLES

A *system* may be defined, in general terms, as a set of interacting components. In a man-made (as opposed to natural) system, these components are designed to operate together towards some defined objective or purpose. A component of a system may be an elementary object, or a system in itself (in which case it is called a sub-system).

Information processing is a global term for the set of operations (input, output, transmission, storage, retrieval, transformation,..) that may be applied to data*. The purpose of an *information system* is to provide a support for a variety of information processing tasks (technical, clerical or managerial) that are required by an organization. Such a system is not closed, i.e. it interacts with an environment which is made up of physical objects and human users. This environment is responsible for information exchange with the system, but also for various kinds of unwanted interference.

* In the sense of the IFIP guide to concepts and terms in Data Processing (Gould 71) : "a representation of facts or ideas in a formalized manner capable of being communicated or manipulated by some process".

In the above general characterization of a system, they keywords are "interacting" and "purpose". "Interaction" means that a major part of the designer's activity must be concerned with the proper definition and management of the relations between the parts of a system. Decomposition, modularization and interface definition are one side of this activity; synchronization between parallel processes is another aspect.

"Purpose" means that a system, or a part of it, has a specific function that must be clearly defined and stated. Proper specification is a necessity if the design and development process is to be kept under control.

Computer programming is often referred to as an art rather than a scientific activity (Knuth 74a). Such reference applies more generally to information systems design. A number of factors account for this situation :

- User needs and requirements are ill-defined; and when defined, they are often mutually conflicting.
- Information systems have long lives and interact with a changing and complex environment; therefore, they are subject to constant modification.
- Large information systems are very complex creations, which cannot in general be completely mastered by a single person's mind.

Therefore, it is quite characteristic that information systems design is often compared to such activities as architecture or city planning, which are design activities with a long history. Alexander's book, "Notes on the synthesis of form" (Alexander 64) (which is mainly concerned with city-planning, although it defines a very general approach to the design of complex systems)is often quoted in relation to program and information systems design. Alexander analyses the transition from "unselfconscious" to"selfconscious" design. In the first attitude, design principles are unstated and transmitted by tradition; in the latter one, the design process relies on a wealth of explicit methods. Another fruitful source of inspiration is the methodological approach followed by Polyà in his book "How to solve it" (Polyà 71) : e.g. the imbedding of a problem in a (well chosen) more general solvable problem, and the identification and reuse of already available results. As humorously pointed by Hamming in his 1968 Turing Lecture (Hamming 69), we are "standing on each other's feet" rather than on other people's shoulders.

In the rest of this survey, we shall try to give a review of some methods and tools that are currently being used to help the information system designer in his task.

The intellectual aids of the system designer are now well identified and we shall only recall them briefly :

1) *Decomposition* of a complex object into more manageable parts is an old methodological principle. However, sheer decomposition is of no avail if the relations between the parts are too complex or ill-defined. Therefore, decomposition must be conducted in a systematic fashion and a number of guidelines have been proposed and illustrated : information hiding (Parnas 71), conceptual abstraction (Dijkstra 72), ease of modification and extension, measures of intermodule coupling (Myers 75) protection of sensitive information, decentralization of resource allocation decisions.

2) *Abstraction* is the intellectual operation whereby a representation, or abstract model, of the behaviour of a complex object is constructed, which only retains some relevant properties and omits irrelevant ones. An abstract model is nothing but the well-known mathematical concept of an equivalence class. The construction of an abstract model results from an explicit choice of the equivalence relation (the selection of the "relevant" properties). An *abstract machine* (Dijkstra 72) is one which exhibits a defined pattern of behaviour regarded as appropriate to the solution of a specific problem. A *abstract data-type* (Liskov 74) is a mechanism which allows the designer to construct information sets which may only be manipulated through a specified set of access functions, and whose behaviour is defined independently of their implementation. This point will be developped in a later section of this paper.

3) *Refinement* is the process by which abstract objects are eventually implemented. The elementary refinement step is to construct an object in terms of more primitive objects by the application of a set of composition rules. A "good" set of composition rules is therefore an essential tool.

Criteria of "goodness" are conceptual simplicity, ease of use and understanding, provability (in a more or less formal sense), efficient implementation. Some agreement has been reached on such elementary sets of composition rules : record structuring for data (Hoare 72), elementary conditional and iterative constructs for sequential programs, monitor structures for concurrent processes. In spite of the availability of such tools, the refinement process does not follow an automatic procedure and relies on the designer's insight and the application of a systematic method. The use of so-called "structured programming" primitives is by no means an insurance against the production of incorrect programs, as illustrated for instance, in (Henderson 72, Gerhart 76). The second reference contains an analysis of a number of errors found in "example" programs published in papers or texts about structured programming. However, the use of well-designed constructions has a positive influence on the process of refinement because it forces the designer to state his assumptions more explicitly. This in turn should eventually make the programs more amenable to an informal "proof".

4) Since design is not a purely deductive activity, the design process usually involves *iteration*. It is well-known that a good way to improve the quality of a design is, at a certain point, to start everything again from scratch, with the augmented knowledge and insight gained from the first attempt. Some caution should however be exercised against the overconfidence and tendency to oversophistication known as the "second system effect" (Brooks 75).

The application of systematic methods to all phases of the life cycle of an information system (from initial requirements to maintenance and modification) is greatly enhanced by the use of appropriate tools. The most widely known are programming languages. However, other kinds of tools have been developped in the recent years and it now appears that programming languages (or more precisely, their compilers) are only parts of more general systems for assistance to system development. It is now widely realized that source program texts, and more generally all sorts of texts such as specifications may be considered as data on which a number of processing operations may be made. The "standard" processing on a source program text is its translation into executable code; but other operations may be considered such as source program transformation, documentation retrieval, analysis of requirements.

In the following section of this survey, we shall review the evolution of the design process and of the tools which may assist the designer in his task. Then we shall give an account of the current trends and perspectives in the design of data and control structures.

2. - FROM SPECIFICATION TO IMPLEMENTATION

During the process of design and development, an information system takes a number of different forms : initial proposal (at a very high degree of generality), overall requirements, functional specifications, component specifications are examples of such forms. The ultimate form is a set of hardware, software and operating rules which together constitute the operational system.

A number of these forms are essentially descriptions. Several terms are currently used in relation to these descriptions.

1) *Requirements* usually refer to an overall description, expressed in the terms of the user, of what the system is intended to do, and of various external constraints.

2) *Specifications*, while having the same general meaning usually have a more precise and even formal connotation.

3) *Documentation* is a general term that applies to all the written material that is used in conjunction with a project description. A more specific meaning is frequently associated with a detailed description of the final form of a system. This description is often (if at all) produced a posteriori.

The designer's dream would be a formal (automated) procedure to obtain the system from its requirements. Although such a goal seems out of reach, the strive for the application of rigorous methods to the design process has led to a number of very significant efforts towards a more systematic treatment of the specifications and documentation.

The main trends of this evolution may be summed up as follows :

1) The specification and documentation process is carried out in a continuous fashion throughout the design. The main result is that the documentation applies not only to the final product, but to all the intermediate stages of its evolution, i.e. to the design process itself. Thus, the main design decisions are made explicit.

2) There is an attempt to introduce more formality in the specifications. The main investigation lines are the definition of specification languages and the use of set-theoretical and algebraic techniques.

3) A consequence of the formalization of the design and specification process is that the use of computerized aids becomes possible. Thus, a number of systems for computer-aided development of software are currently being experimented with.

These ideas have been actually with us for a long time. For instance, an overall scheme for system design by continuous refinement and partial simulation was proposed in (Zurcher 68); a systematic approach to program specification and construction was also investigated by the same time. However, it is not until the recent years that these ideas were applied to the actual design of sizable programs.

We shall review the recent evolution along three main directions : systematic program construction, computer aids to system design and development, and formal approaches to specifications.

2.1. Methods for systematic program construction

Since the pioneering work of (Dijkstra 72), a large amount of literature has been published on the subject of systematic program construction. We shall not attempt to review this work, but we shall make the following remark : systematic programming (the original expression "structured programming" has somehow degenerated into a buzzword) refers to a methodological approach, to a new attitude towards the act of program design, rather than to the strict application of some recipe. As a consequence, it may be very difficult (as experience has shown), to promote the use of systematic methods if adequate tools are not available.
This is especially true in a production (as opposed to academic) environment, where external constraints may impose the use of ill-suited languages. With regard to this remark, we shall restrict this review to a very limited aspect : the use of some sort of formalized methodology to assist in the development of programs. The methods that we shall examine are designed to be used manually (without computer assistance) and they often rely on a graphical language. All of these methods are based on some form of decomposition and stepwise refinement. As a consequence, various forms of tree-structured diagrams are basic ingredients of the methods.

SADT (Ross 77), developed by Softech, HIPO and Composite Design (Myers 75) developed by IBM, involve decomposition of a system into units (parts, modules,.). The relations and interfaces between these parts are formally described. Design criteria such as minimal coupling may be applied. The diagrams are used for documentation, for review of the design before implementation, and as a guide to implementation.

A more formal approach is proposed in (Warnier 72) and (Jackson 75). Both methods are mainly designed for the construction of data processing applications (as opposed to operating systems or real-time software). The main idea of (Jackson 75) is to set up a mapping between the structure of a file, or set of files, and the structure of the program that operates on these data. File structures are constructed from elementary components by the operations of concatenation, selection and iteration; this structure is reflected in the programs. Refinement may be applied, if necessary, to both program and data structures. A simple graphical language is used to document the design process.

Finally, a still more formal method has been developed in (Abrial 74, 77). The initial requirements are expressed in a specification language based on sets and relations (a similar approach is followed in SETL (Schwartz 72)). Specifications written in this language are then transformed by hand, using a set of semantics-preserving transformations, into programs written in another language, which assumes more specific implementation choices. This process is iterated until a working program is obtained. The validity of the design process relies on the correctness of the program transformation mechanisms. This method has been successfully experimented in an industrial environment and appears as very promising.

2.2. Computer aids to systems design and production

Most computer aids to system design and production may be roughly classified under two headings. In the first class, emphasis is on the early steps of design, specification and evaluation. In the second class, actual programs are manipulated and executable code is produced. Both types of systems have evaluation, testing and documentation editing facilities. Current research is under way to construct systems that would encompass all phases of the design and production process.

A general model for a computer-based system for assistance to system design is given on Figure 1.

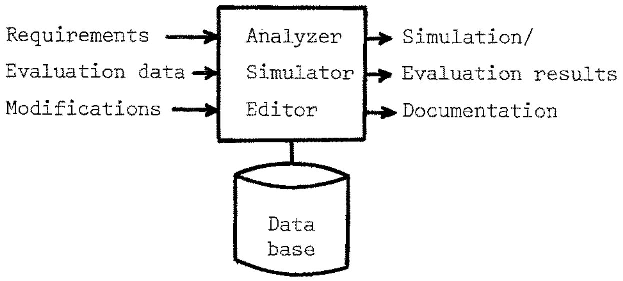

Figure 1.

All information relevant to the design is progressively entered into a data base which records every step of the development process. The data base is used for the production of documentation on the project and for evaluation of the design.

This type of system is exemplified by the PSL/PSA system (Teichroew 77) developed as a part of the ISDOS project at the University of Michigan. PSL (the "Problem Statement Language") allows the designer to describe a system design as a set of "objects" connected through "relationships". Specific types of objects and relationships are available for the description of a variety of aspects of information systems (input-output, hierarchical grouping, data structures, performance,...).

During the project, the result of every step in specification and development is described as a PSL program. Descriptions written in PSL are processed by a Problem Statement Analyzer (PSA) which analyzes the information provided and enters it into the data base. PSA also contains some evaluation features which may help the analyst to evaluate the impact of potential improvements. The system is reportedly used in a variety of industrial environments.

Similar systems are CADES (CADES 73), DACC (Boehm 75), and TOPD (Henderson 73). Although such systems offer no substitute to the design itself, they help the designer by forcing him to formally express the requirements, by providing checklists for relevant questions, by producing readable documentation in a standard form and by evaluating the effect of design decisions.

Another class of computer aids may be represented by the general scheme of Figure 2.

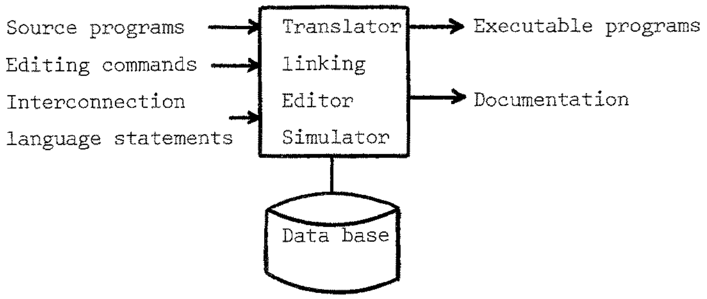

Figure 2.

These systems may be described as "software factories". The main capabilities that they provide are as follows :

- creation, editing and modification of source programs
- program library management
- compilation and linking of programs
- debugging, testing and simulation
- documentation production

It should be noted that the production machine, on which these development tools are implemented, may well be distinct from the target machine for which software is produced.

Two important components of a software development system are the librarian in charge of the program data base, and the language processor(s). Such an environment is well suited to the development of modular programs. The overall architecture of a system in terms of elementary components may be expressed in a module interconnection language, while individual modules are developed using a language processor that supports the concept of a module.

A number of production systems designed along these lines are currently being used or experimented with. The "workbench machines" described in (Ivie 77) are used as a network, connected to target machines by high-speed lines, and include development facilities for several target machines. Several recently developed systems are based on PASCAL or extended versions of this language (Donzeau-Gouge 75, Geschke 77, Krakowiak 76).

A natural extension would be a general system design and production facility that would integrate the capabilities described in figures 1 and 2, i.e. would encompass all stages of development from specifications to code production. This is actually a stated objective of some of the above mentioned systems (Boehm 75, Lucena 76, Teichroew 77). The essential steps of specifications writing and of building programs from specifications remains however the designer's task, but useful assistance may be provided (e.g. in the way of automatic consistency checks). Progress in this domain depends on advances in problem specification, a subject that will be reviewed in the next section.

2.3. Advances in specification techniques

The most widely used specification methods presently rely on natural language. Concern for software reliability has recently fostered the development of more formal methods. The purpose of such work is to allow the writing of specifications that could be amenable to formal verification and from which an implementation could be easily derived either by hand or by a formal procedure.

The first step is to define a unit for specification. Methods for decomposition and hierarchical structuring described for program design clearly extend to specifications. Therefore, most of the work on specifications has concentrated on specification techniques that apply to the basic building blocks that support simple abstractions (see section 3 of this paper), i.e. essentially multiple entry modules (Guttag 76, Liskov 75).

Specification methods fall into two classes : operational and definitional. In the operational approach, the specified operation is described in terms of some already defined "machine" (or set of operations). In the definitional approach, an operation is described by its effect, as a set of pre-and post-conditions for the state of the object upon which the operation is applied.

Current specification methods usually combine both types of definitions. An operational specification is often used as a guide for implementation, whereas a definitional specification is more readily usable as a guide for testing and verification.

Both methods are presently being investigated. The already mentioned work of (Abrial 74, 77) uses an operational specification, and the emphasis is on the stepwise transformation of this programmed specification into an implementation. On the other hand (Guttag 76) uses algebraic techniques for the specification of abstract data types and the emphasis is on completeness and consistency verification.

It seems that we are still a long way from using formal specification techniques in large scale projects. Meanwhile, the introduction of even primitive techniques to make specifications more formal will certainly provide an incentive towards a more systematic practice. Formal specifications per se cannot regarded as a panacea; after all, formality of the mathematical notation does not prevent mathematicians to occasionnaly write erroneous proofs! Instructive discussions of the relations between mathematical thinking and programming methodology may be found in (Dijkstra 76, Gerhart 76, Mills 75, Schwartz 72).

Finally, we should not leave the subject of specification without a world about the specification of the user interface, i.e. the language by which an information system and its human users achieve communication. This includes the design of the command language (including control and data description) by which the system is operated, as well as the design of the output language in which results are given. Little formal consideration seems to be given to these subjects, with the result that the above mentioned "languages" often hardly deserve this name at all. Notorious examples are the command languages used to instruct operating systems : the obscurity and lack of logical structure of most of these "languages" are well-known facts.
The interested reader should refer to part 4 of (Naur 74) which is devoted to a thorough survey of the design principles that apply to data interchange between man and computer. This is a difficult field of study where contributions are needed from ergonomists and psychologists. Advances in the technology of graphical information displays should open new directions for progress in this field.

3. - THE DESIGN OF CONTROL AND DATA STRUCTURES

An elementary information processing task may be described as the operation of a *procedure* on some *data*. The execution of such a task is called a *process*. Concurrently executing (or logically independent) tasks are described by a system of concurrent processes, which may interact through shared resources or informations.

Procedures, data and processes are thus the main components of any information system. Much effort has been devoted to devise abstract (i.e. implementation independent) models for these three classes of objects, and to develop structuring tools based on such models. In the following sections, we shall try to give an account of the present state and future trends of this evolution.

3.1. Data structures

The use of abstraction for the design of data structures is more recent than for programs. The underlying idea is that a data structure is more adequately defined (for a user of this structure) in terms of the allowed access operations than in terms of its implementation. The data abstraction operation consists in the definition of a model of behaviour (an *abstract data type*) according to which a class of objects may be generated. The properties of such an object (an *instance* of the type) are defined by the specifications of the abstract type. These properties do not depend on the implementation of the object. Some important properties follow :

1) The user of an object needs only to know the specifications of its abstract type and should make no assumption on its internal structure.

2) An object may be implemented in a number of ways. From a user's standpoint all these implementations are equivalent (except perhaps as regards efficiency) as long as they conform to the object's data type specifications.

3) The user of an object may not access, retrieve or modify any part of this object except through the specified access procedures.

A number of schemes have been proposed to implement the idea of data abstractions. This variety is reflected in the number of different terms that were recently introduced : *abstract type* (Liskov 74), *abstract machine* (Dijkstra 72), *capsule* (Horning 76) denote general abstraction mechanisms, while *class* (Dahl 72), *cluster* (Liskov 74), *form* (Wulf 76) refer to specific implementations of such mechanisms, and *module* is used in both (and other) contexts (e.g. Parnas 72, Wirth 77).

The general pattern that appears to be common to these proposals is that an object generated by an abstract data type may be described as follows :

- the object is represented by a set of information ("state variables"), together with a set of access procedures; the user interface is defined by these access procedures,

- the state of the object is defined at any time as the value of the state variables,
- the state receives an initial value when the object is created;
- the access procedures are the only way of access to the data part of the object; in most proposals, this restriction is enforced at compile time;
- the effect of the access procedures may be specified in terms of an initial and final state. As a consequence, the state of the object, at any time, only depends on the sequence of operations that were executed since its creation.

On the other hand, some other issues are still controversial such as :

- separate compilation of abstract types;
- mechanisms for the construction of parameterized or generic abstract types;
- efficient implementation;
- mechanisms for parameter passing.

A number of experiments with the implementation and use of abstract types are currently under way. Experience with actual use of languages including this concept is still limited (Geschke 77) and seems to be restricted to the production of systems programs (see however (Hammer 76) for a discussion of the use of abstract types in data base design). Some tentative conclusions may be drawn from the first results :

1) The use of new data structuring mechanisms does not automatically result in better (more reliable, understandable, efficient) programs. A good tool supplements the designer's skill but offers no substitute for it.

2) A strict compile-time type checking system must tolerate some exceptions (for logical or efficiency reasons). Such exceptions should be made as explicit as possible to make the user aware of the potential dangers.

As noted in (Geschke 77), early experience with this new data structuring concepts can be compared to experience with the use of "structured" control constructs. Such constructs help their user to acquire a good style of program design which may afterwards be put into practice with languages that do not support them. As a consequence, we would recommend early acquaintance with these mechanisms in computer science education.

3.2. Control structures for sequential programs

One of the main results of the recent advances in systematic programming (Dijkstra 72), (Knuth 74 b), (Mills 75), (Wirth 76) is that an adequate tool for the construction of sequential programs is the set of three elementary constructs: sequence, selection (*if-then-else*), and iteration *(while-do)*, possibly supplemented by *case* and *repeat-until*, together with the basic abstraction device provided by procedures. Even for widely used languages that do not include these constructs (such as FORTRAN or COBOL), adherence to a programming discipline may be enforced by the use of a preprocessor or by a set of standard rules of transcription.

The benefits of the systematic use of a small number of simple and well defined constructs are presently recognized and largely illustrated by a number of published examples (e.g. in the references quoted at the beginning of this section). However, an important feature that appears in the programs of many large scale information systems is not easily captured by these constructs. The operation of such systems may be described as a "normal case" algorithm together with a number of "exceptions". Exceptions may include hardware failure, erroneous data, or any condition specified by the designer. The exception-handling mechanisms often account for a large fraction of the total size, cost and complexity of the system.

The problem of exception handling has been the subject of intense research since its practical importance was realized. A number of methods have been proposed, but it does not seem that a single solution to the problem has achieved pre-eminence. A complete review of recent work, together with some new proposals, may be found in (Levin 77).

Exceptions may be regarded as "special cases" and handled in the same way as "normal cases" e.g. by means of return values that indicate abnormal return from a procedure call. This way of doing, however, is detrimental to a good understandability of the programs. An acceptable exception handling mechanism should be adapted to any abstraction-defining constructs used in the program : if an object is defined by an abstraction mechanism that encapsulates its internal structure, any exceptional conditions arising when the object is used should be expressed in terms of the abstraction by which it is defined. In other words, for example, an exceptional condition detected when a programmer-defined data structure is misused should not be expressed in terms of memory addresses, as is too often experienced! An exceptional condition should be propagated through the abstraction levels until enough information is available to allow its processing.

An adequate expression language for the definition of exception detection and handling should allow to clearly separate what is considered a normal case and what is considered an exception; it should also provide means for binding the detection of an exception to its processing.

While the main issues in the design of exception handling mechanisms are now being understood (at least for sequential programs), especially in the context of abstraction- defining constructs, we are clearly lacking experience with the actual use of such mechanisms. Some of the recent proposals are currently being implemented under experimental conditions and user experience is eagerly awaited.

3.3. Parallel processes

Parallel processes provide a means for structuring systems in which a number of loosely coupled activities cooperate towards a common task. A great variety of methods have been devised to achieve interprocess cooperation. Semaphores provide a general tool which has been widely used in the design of operating systems, and which has been included as an elementary synchronizing operation in the hardware of a number of computers. However, some considerations have recently led to the development of more elaborate tools :

1) The trend towards the use of high-level languages for the design and implementation of systems programs : high-level synchronizing constructs were needed especially for inclusion in the data abstraction mechanism provided by these languages.

2) The growing concern for mechanisms amenable to precise specification and correctness proofs.

3) The advent of distributed systems, in which processes do not share a common store.

3.3.1. High-level synchronizing tools

Monitors (Hoare 74) were introduced to implement data structures which may be shared by several processes, and used through a set of access procedures. The synchronizing mechanism built into the monitor ensures mutual exclusion for the execution of access procedures, and allows to enforce a scheduling discipline among processes by means of a set of queues associated with activation conditions.

Monitors have been included in several programming languages (e.g. Concurrent Pascal (Brinch Hansen 77), Modula (Wirth 77)). Efficient implementations of monitors have been devised and some experience has been collected, which seems to demonstrate the usefulness of this construct. However, when programming with monitors, one must explicitly describe the scheduling operations in terms of waiting and activation primitives. In many cases, one would wish a more global and implicit expression of synchronizing conditions in terms of procedure executions, considered as elementary units of process activity. This has led to the development of more formal constructs.
Path expressions (Habermann 75) and various forms of event counters (e.g. Robert 77) were introduced in an attempt to express synchronizing conditions in a module in terms of procedure executions. These synchronizing conditions are described by regular expressions (path expressions) or by algebraic relations between the values of event counters. The formality of these expressions makes these mechanisms amenable to proofs. Experience with their actual use is still very limited. The main difficulty with their use seems to arise when synchronizing conditions

in a module are execution-dependent, i.e. if they are expressed in terms of the value of internal variables of the module or of procedure parameters.

3.3.2. Process cooperation in distributed systems

A great deal of interest has arisen for distributed computing in the recent years. Three main reasons account for this interest :

1) The availability of low-cost computing power allows one to devise highly parallel computing systems constructed form a large number of interconnected processors.

2) The development of computer networks makes resource sharing possible netween geographically distant centers.

3) Increasing concern for reliability leads to the distribution of work between interconnected computers e.g. in industrial process management.

In spite of an intense activity, it does not seem that the ambitious goals set up several years ago have really been attained. A number of fundamental problems in distributed computing are still awaiting a solution. We shall try to analyse what appear to be the main issues in this fields.

We shall first set up a model of a distributed system as a set of entities connected by communication lines. We shall consider each of these entities as a self-contained module. Each of these modules is associated with a set of cooperating processes which share this module; communication between processes on different modules is achieved by asynchronous messages (this is the only possibility in the absence of a common store).

Besides the absence of a common store, a distributed system is characterized by the absence of a common clock. More precisely, the time scale which applies to message transmission is not negligible with respect to the local time scale in an individual process. Moreover, the transmission lines may usually not be regarded as reliable and message loss is not an exceptional event.

Some of the main problems in such a structure may be summed up as follows :

- how to achieve state consistency between data in different modules (this amounts to solve the mutual exclusion problem between two distant processes);
- how to ensure a sufficient overall reliability to the system in spite of the unreliable communication mechanism;
- how to express a computation distributed among several distinct modules (this may not be done by intermodule procedure calls because of the message mechanism, and new linguistic constructs are needed);

- how to survive a failure in one of the communicating modules.

Only partial solutions have been proposed so far to all of these problems. The model of a set of modules connected by asynchronous message lines seems to be the paradigm for a variety of situations : cooperating processes in the kernel of an operating system, multiprocessor systems, actor models in Artificial Intelligence, distributed data bases, loosely connected processors in industrial control applications. We think that a systematic investigation of this model (as initiated e.g. in (Feldman 77)) should contribute to give a sound framework to the design of distributed applications.

4. CONCLUSION

In this survey, we have tried to discuss a number of views pertaining to information systems design. Our conclusion will be very brief : design essentially remains a human activity, and no magic sophisticated device will ever replace thorough analysis, careful expression of requirements, clear separation of correctness and efficiency concerns, and strive for conceptual simplicity. The main achievement of the recent years' effort is that we are in the process of founding the designer's skill on an explicitly transferrable body of knowledge. In addition, we are learning to make a good use of computers to assist the designer as well as the implementor of information systems. In this respect, the importance of well-designed tools should not be underestimated, because the use of well chosen tools forces us to ask the "right questions", and because the applicability of a design method is greatly enhanced if the method is supported by a set of appropriate tools.

REFERENCES

ABRIAL J.R. : Data semantics, *Proc. IFIP Working Conf. on Data Base Management Systems* (Klimbie and Koffeman, eds.), North-Holland (1974).

ABRIAL J.R. : Méthode et langage de spécification. (Unpublished notes, 1977).

ALEXANDER C. : *Notes on the synthesis of form,* Harvard University Press, 1964.

BOEHM B.W., McCLEAN R.K., URFRIG D.B. : Some experience with automated aids to the design of large-scale software, Proc. Intern. Conf. on Reliable Software, SIGPLAN Notices 10,6 (juin 1975).

BRINCH HANSEN P. : *The architecture of concurrent programs,* Prentice Hall (1977).

BROOKS F.P. : *The mythical man-month,* Addison-Wesley, 1975.

CADES : Computer-Aided Design and Evaluation System (a series of articles in *Computer Weekly,* (July 1973).

DAHL O.J. : Hierarchical program structures, in *Structured Programming* (Dahl, Dijkstra, Hoare), APIC Studies in Automatic Programming n°8, Academic Press (1972).

DIJKSTRA E.W. : Notes on Structured Programming, in *Structured Programming* (Dahl, Dijkstra, Hoare), APIC Studies in Automatic Programming, n°8, Academic Press (1972).

DIJKSTRA E.W. : *A discipline of programming,* Prentice Hall (1976).

DONZEAU-GOUGE V., HUET G., LANG B., LEVY J.J. : A structure-oriented program editor : a first step towards computer-assisted programming, *Proc. ICS Conf.,* Antibes (May 1975).

FELDMAN J.A. : A programming methodology for distributed computing (among other things), *TR-9, Dept. of Computer Science, Univ. of Rochester* 1977.

GERHART S.L. and YELOWITZ L. : Observations of fallibility in applications of modern programming methodologies, *IEEE Trans. Software Engineering, SE-2, 3* (Sept. 1976).

GESCHKE C.M., MORRIS J.H., SATTERTHWAITE E.H. : Early experience with Mesa, *Comm. ACM,* 20, 8 (Aug. 1977).

GOULD I.H. : (Ed.) *IFIP Guide to concepts and terms in data processing,* North-Holland, 1971.

GUTTAG J. : Abstract data types and the development of data structures, Proc. SIGPLAN/SIGMOD Conf. on Data, *SIGPLAN Notices* 8,2 (march 1976). (To appear in Comm. ACM).

HABERMANN A.N. : Path expressions *Dept. of Computer Science, Carnegie Mellon University* (1975).

HAMMER M. : Data abstractions for data bases, Proc. SIGPLAN/SIGMOD Conf. on Data *SIGPLAN Notices,* 8,2 (March 1976).

HAMMING R.W. : One man's view of computer science, *Journal A.C.M.,* 16,1 (Jan.1969).

HENDERSON P., SNOWDON R. : An experiment in structured programming, *BIT* 12,1 (1972).

HENDERSON P., SNOWDON R. : A tool for structured program development; *Proc IFIP Congress 1974,* vol 2, North-Holland (1974).

HORNING J.J. : Some desirable properties of data abstraction facilities, *Proc. SIGPLAN/SIGMOD Conf. on Data, SIGPLAN Notices* 8,2 (march 1976).

HOARE C.A.R. : Notes on data structuring, in *Structured Programming* (Dahl, Dijkstra, Hoare), APIC Studies in Data Processing n°8 Academic Press (1972).

HOARE C.A.R. : Monitors : an operating systems structuring concept, *Comm. ACM,* 17, 10 (1974).

IVIE E.L. : The programmer's workbench - a machine for software development, *Comm. ACM* 20,10 (oct. 1977).

JACKSON M.A. : *Principles of program design,* APIC Studies in Data Processing n°12, Academic Press (1975).

KNUTH D.E. : Computer Programming as an art, *Comm. ACM* 17,12 (Dec. 1974 a).

KNUTH D.E. : Structured programming with goto statements, *Comp. Surveys,* 6,4 (Dec. 1974 b).

KRAKOWIAK S., LUCAS M., MONTUELLE J., MOSSIERE J. : A modular approach to the structured design of operating systems, *Proc. MRI Symp. on Computer Software Engineering,* Polytechnic Institute of New-York (1976).

LISKOV B.H., ZILLES S.N. : Programming with abstract data types, *Proc. SIGPLAN Symp. on Very High Level Languages,* SIGPLAN Notices, 9,5 (1974).

LISKOV B.H., ZILLES S.N. : Specification techniques for data abstractions, *IEEE Trans. Software Engineering,* SE-1 (March 1975).

LUCENA C.J., COWAIN D.D. : Toward a system's environment for computer assisted programming, *Inf. Proc. Letters,* 5,2 (June 1976).

MILLS H.D. : How to write correct programs and know it, Proc. Int. Conf. on reliable software, *SIGPLAN Notices* 10,6 (June 1975).

MYERS G.J. : *Reliable software through composite design,* Petrocelli/Charter (1975).

NAUR P. : *Concise survey of computer methods,* Studentlitteratur, Lund (1974).

PARNAS D.L. : Information distribution aspects of design methodology. *Proc. IFIP Congress* (1971).

PARNAS D.L. : On the criteria to be used in decomposing a system into modules, *Comm. ACM,* 15,12 (Dec. 1972).

POLYÀ G. : *How to solve it,* Princeton University Press (1971).

ROBERT P., VERJUS J.P. : Towards autonomous descriptions of synchronization modules, *Proc. IFIP Congress,* (1977).

ROSS D.T., SCHOMAN K.E. Jr : Structured analysis for requirements definition, *IEEE Trans. Software Engineering,* SE-3, 1 (Jan. 1977).

SCHNEIDER B.R. Jr : *Travels in computerland, or incompatibilities and interfaces,* Addison-Wesley (1974).

TEICHROEW D., HERSHEY E.A., III, PSL/PSA : A computer-aided technique for structured documentation and analysis of information processing systems, *IEEE Trans. Software Engineering,* SE-3,1 (Jan. 1977).

WIRTH N. : *Algorithms + data structures = Programs,* Prentice Hall (1976).

WIRTH N. : Modula, a language for modular multiprogramming, *Software Practice and experience* 7,1 (1977).

ZURCHER F.W., RANDELL B. : Iterative multi-level modelling : a methodology for computer system design, Proc. IFIP Congress (1968).