UNIVERSITA' DEGLI STUDI
DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE

BULL ITALIA
CENTRO STUDI
INFORMATICA
E AUTOMAZIONE

Note di Software

Anno 1991
51
Marzo

Note di software è una pubblicazione trimestrale, edita a cura del Dipartimento di Scienze dell' Informazione dell' Università di Milano e dal Centro Studi Informatica e Automazione della Bull Italia.

Note di software intende costituire uno strumento per la circolazione di informazioni, idee, metodologie ed esperienze nell'area delle tecnologie e della produzione del software.

La collaborazione a **Note di software** è libera. Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Direttore responsabile o con il Comitato di Redazione. In particolare si sollecitano contributi nella forma di monografie, da inviare in triplice copia, e che non dovrebbero superare le cinquemila parole.
I lavori ricevuti per la pubblicazione verranno revisionati dai membri del Comitato di Revisione Scientifica, che possono avvalersi della collaborazione di specialisti del settore. I lavori pubblicati riflettono comunque le opinioni dei loro autori.

Note di software viene distribuito ad Aziende, Enti e specialisti del settore che ne facciano richiesta alla Redazione.

Autorizzazione del Tribunale di
Milano n. 87 del 23 febbraio 1977

Supporto tecnico - gestionale e distribuzione: Domenico Asinelli
Bull Italia, via Tazzoli 6
20154 Milano, Tel. 02 - 67792960

UNIVERSITA' DEGLI STUDI
DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE

Bull Italia
CENTRO STUDI
INFORMATICA
E AUTOMAZIONE

Note di Software

Anno 1991
51
Marzo

QUESTO NUMERO

Il primo articolo è un "position paper" sugli sviluppi software di grandi dimensioni. E' questo il campo di esperienza di Antonio Pizzarello, che per molti anni ha guidato progetti del genere e sviluppato metodiche originali.

Nel lavoro che segue viene descritto HyperX, una macchina astratta ipermediale. Un obbiettivo degli autori - A. Limongiello, F. Soresi, A. Spinelli, W. Vannini - è di contribuire alla definizione di un modello dei dati standard.

E. Trentin e P. Miotti discutono nel terzo articolo alcuni importanti aspetti attinenti le esigenze e le problematiche insite nella realizzazione di una macchina astratta per gli ipertesti, con riferimento a un modello denominato HE .

Una interfaccia uomo-macchina basata sulla conoscenza è l'oggetto del lavoro di R. Barbò, C. Ferri, P. Salvaneschi. L'interfaccia è diretta a facilitare l'uso di un pacchetto software per l'analisi dei segnali .

A. Beltramini e F. Casè descrivono un programma per la configurazione automatica dello hardware di un sistema per il controllo di processi industriali. Il programma è stato realizzato con l'impiego combinato di tecniche di programmazione dichiarative e procedurali.

In chiusura, ripubblichiamo (con molte scuse all'autrice) un articolo di Rita Pizzi apparso sul penultimo numero della rivista con sviste tipografiche che ne alteravano la comprensione.

LA REDAZIONE

SOFTWARE TECHNOLOGY

A. Pizzarello, Bull HN
Phoenix Product Division, Phoenix, Arizona

RIASSUNTO

Questo articolo fornisce una panoramica del processo di sviluppo usato per la creazione di software di considerevole dimensione. Vengono illustrati i punti di forza e gli inconvenienti del processo guidato dell'implementazione oggi in uso e verrà fatto un confronto con i metodi formali. Infatti i risultati di ricerca nell'ambito della formalizzazione della fase di disegno dei programmi hanno oggi particolare rilevanza, anche se la loro applicabilità pratica viene messa in discussione da parte di alcuni ricercatori.
Il tema principale di questo articolo tratta del fatto che solo alcuni tipi di metodi formali possano supportare una disciplina mentale e di organizzazione necessaria per introdurre una reale fase di disegno nel processo di sviluppo del software.
Il trasferimento della tecnologia dei metodi formali nella pratica comune dello sviluppo nel software non è certamente un compito semplice: alcune delle soluzioni possibili a questo problema vengono discusse alla luce dei risultati di una significativa esperienza industriale.

ABSTRACT

This paper surveys the development process used for the creation of software of substantial size. The strengths and the drawbacks of the currently used implementation driven process are highlighted, and are compared with those of formal methods. Research results on the formalization of program design are found to be interesting, even though their practical applicability is questioned by some researchers. The theme of this paper is that only some kind of formal method can supply the necessary mental and organizational discipline for introducing a true design phase in the software development process. The transfering of the formal method technology into the common practice of software development is certainly not a simple task. Some of the possible solutions to this problem are discussed in light of the results of a significant industrial experience.

1. INTRODUCTION

The word technology (from the Greek "technologia" meaning the systematic treatment of an art) is usually defined as "a technical method for achieving a practical purpose."

In the software industry the practical purpose is the creation of code capable of coherently driving a system in a way considered satisfactory by a community of users.

The technical method currently used is representing design ideas in an imperative (coding) language and then, through a process of tests and error correction, modify the code until an acceptable level of error rate is reached. Since the coding language is the only formalized object involved, this process may be defined as implementation driven.

To a superficial observer this approach may seem not only practical but even desirable. In fact, what can be better than direct experimentation for the validation of

a product? The only problem with this approach is that it simply does not work for large systems. The evidence can be found in the cost of system software development which is universally deemed to be intolerably high. Also schedule uncertainties plague the industry, and the final product's reliability is not acceptable for many interesting critical applications.

Whenever an unsatisfactory situation exists in the industry, it is reasonable to expect the research community to be working at some of the fundamentals to discover a better approach (actually only the principles for a better approach). This has been precisely the situation for the last three decades after the alarming definition of a software crisis was publicized [1]. Since then, many miracle cures have been discovered but the state of crisis remains as critical as before (if not more so) in spite of all efforts.

Notable among all investigations of the last three decades is the idea of treating the program (or the software system if a more fashionable term is preferred) as a pure intellectual object and as a part of some mathematical formalism. In this way, it is hoped the properties of the program can be studied before any code is written, thus making it possible to infer what the correct functions and their interrelationship should be without any experimentation. Since the cost, schedule slippage, and unreliability of software systems stem from the long duration of the test phase, the unpredictability of its duration and the uncertainty of its results, the idea of discovering errors at earlier stages has considerable practical appeal.

Surprisingly however, practical software development has not been affected by the research of formal design as expected. The reasons for this lag in the application of such research are many. One is the relatively slow progress of the research itself, others may be related to social and educational problems.

Recently, however, some success stories have emerged. This note analyzes the shortcomings of the current "implementation driven" approach and suggests an organizational framework for the introduction of formal design methods in a practical software development process.

2.THE IMPLEMENTATION DRIVEN PROCESS

The main characteristics of the implementation driven process are two:

a) the use of an imperative (machine executable) language as the sole formalism for supporting design ideas.

b) the use of tests as the main vehicle of error detection.

The conceptual problem with the first point is that code does not describe the desired properties of the system as a whole but is only a sequence of commands supposedly realizing them. Consequently, the global system properties, that is, what the system is supposed to do and not to do, can be only vaguely expressed in a human language, usually in an operational fashion which reflects the view of the system during its operation. Some low level of confidence about the system correctness may be gained via discussions among the designers.

Errors are usually discovered during tests. This is the observation of undesired behavior of the system. The undesirability is judged by the only possible way, that is, with respect to the external requirements.
There is no clear way to correlate this misbehavior with a violation of some necessary system property (expressed in terms of internal states), simply because these properties are not defined (or even conceived) by the designers. The correction of the misbehavior may not remove the cause but only mask the observed effect which often is only one of the many possible effects of the hidden fault.

Another problem is that because of its discrete nature, it is impossible to infer correctness of software from the successful completion of a few test cases. As a consequence, obtaining an acceptable error rate through the use of a set of test cases does not permit any inference about the correctness of the system, because there is no known method of extending the results of a set of test cases to a broader domain.

To make things even more interesting, the latency of software faults is enormous. This fact translates into the need for very long periods of testing before an acceptable

error rate can be obtained. Worse yet if the system is installed in many sites after the system tests have shown acceptable error rates, the exposure to multiple sites increases the error rate as seen by the software producer. Even if the error rate per site is acceptable to the individual user, the overall cost of the repairs over a relatively large community of users may become an intolerable burden.

3. VARIETY OF SOFTWARE SYSTEMS

Software systems are not all the same. E.W. Dijkstra once said that 80% of programming problems are trivial, 20% impossible, but there is a fluctuating fringe of interesting problems between the two extremes which causes all the troubles [2]. One way to trivialize these trouble-making problems is to write their solution (possibly in a correct manner) and then reuse this solution (either in code or in a design form) many times. Typical examples of systems that are developed by using this approach are mathematical function libraries, application packages. etc.

A way to classify software (suggested by D. Chan [3]) is to plot the productivity figures as a function of the specialization of its application domain. Single purpose software (such as the use of a spread sheet on a PC, or the use of 4th generation languages) allows the rapid encoding of complex problems, therefore may be considered as conducive to highest productivity. On the other extreme, software with a broad application domain such as operating systems, compilers, etc., require considerable work for their production and therefore can be considered as low productivity products. The point here is that in order to increase productivity, we must create effective support software for specialized domains even though this software requires the usage of other software much more difficult to produce.

Unfortunately, narrow application domain software is not always satisfactory because one of the most attractive (and expensive) features of software is the possibility of creating a tailor-made system for a specific application. Reusable software appears to be the tool for obtaining adequate productivity for the creation of tailor-made systems.

4. SYSTEM INTEGRATION

What appears to be the commonly accepted approach to the preparation of tailor-made systems is to put together pre-existing software (and hardware) with some modifications and additions to the software in order to satisfy the particular site requirements. This type of operations has caused the introduction of a new buzz-word in the computer industry: system integration.

Today's system integration is a big business. According to a recent report (Info Corp - Gartner Group), system integration business in the U.S. for the 1989 produced vendor revenues of $5.9 billion [4]. The prediction for 1994 is of a market of $14.6 billion. This is a considerable market, and therefore, is interesting to be analyzed in the light of the software productivity problem.

In many cases of system integration, the assumption that pre-existing software can be readily used is not valid. It is not uncommon to observe that up to 80% of the existing software may need to be rewritten. Even the best system integrators consider this a very risky business. For example, the system conceived for the New Jersey Department of Motor Vehicles was a complete failure. Another system for United Airlines was cancelled after a huge investment [4]. (The expenditure was estimated at around $300 million.)

The main cause of all these difficulties is what people with experience of large system software development would consider as very familiar: the necessity of a very long period of system testing during which both the meaning of the re-used software and its role in shaping up the desired system properties are discovered. If we reconsider the main characteristics of the implementation driven development process defined above, we can shed some light on the reasons for system integration difficulties. Since the only precise description of the software to be integrated or modified is the code itself, the only way to understand its meaning is to read it or to execute it on a machine. This is clearly an uncertain process which, when coupled with the equally uncertain process of determining the system's global properties, accounts for most of the failures.

A factor that increases system integration difflculties even further is that these large systems are intrinsically

non-deterministic, mostly due to some degree of parallel processing. The design of parallelism is a difficult art which is only partly mastered by those very few developers who were involved for a long time in operating systems or communication software design. The historical examination of the creation of some of these systems (such as OS 360, GCOS, ARPANET, etc.) shows that coping with parallelism forces the introduction of some strongly limiting design constraints. In other words, potential parallelism is only partially exploited because of the intrinsic difficulties of the design of correct parallel systems. Furthermore, none of the currently existing parallel systems are flawless. All operating system investigators are familiar with mysterious system crashes or other undesired phenomena that very often cannot be repeated for analysis.

The last source of difficulties is also a consequence of the limitation of the implementation driven process. Large systems, regardless of whether they are integrations of pre-existing software or not, need several persons for their development. If the properties of the overall system are not precisely known in advance, the boundaries of each person's work module are known only vaguely. This fact often causes the erroneous preparation or modification of modules, but more damagingly, it allows erroneous modifications of the already erroneous modules during the long system test period. Hence, cost overruns and, in many cases, total failure to bring the system error rate below an acceptable level, is a too frequent outcome.

5. RESEARCH ACTIVITIES

The software crisis has stimulated a great deal of research all attempting to find the "final solution" to the software development problem.

The research activities followed (and still follows) many different directions. One that has been very much publicized in the recent past is the improvement of the tools used for implementation in the hope that these added features would percolate up to the level of system design. Notable examples of this approach are programming languages such as ADA which have introduced packages as a structuring entity, and the so-called CASE tools which are supposed to control the complexity of the code integration task, etc.

The interesting point about these improvements in the implemention techniques is that they have a rather limited effect whenever the size of the system becomes substantial. B. Bohem (TRW) has collected a large amount of data about the effect of various "remedies" on software productivity ~5~. Although not completely ineffective, all the effect of improved tools and techniques is orders of magnitude less than the effect of system size and of the quality of the people involved in the development. This fact points out that we need to look elsewhere for a research result that can substantially affect productivity for large system developments.

Another popular field of research, aimed to improve the implementation driven process, is testing. This fleld is the source of a number of negative results all showing the inanity of attempting to determine a set of test cases sufficient to prove correctness even for the simplest programs. A limitation of testing is suggestive of a current practical misinterpretation of the testing role. Most errors are the omission of logic needed for some purpose. If the only source of precise documentation is the program text itself, omission is much harder to discover than inconsistency. This suggests that some other precise description beside the program text is necessary in order to establish correctness via testing. This limitation adds up to the fact that we know not how to test a program in such a way that through the use of a reasonably small number of test cases we can completely characterize its behavior.

The research on testing however has not been just a land of negative answers. Statistical testing has been shown to be an effective quality assurance tool. However, it is not an exaggeration to say that most of the research attempting to improve the current implementation driven approach has not been a resounding success.

6. FORMAL DESIGN METHODS

Since the pioneering work of R. Floyd [6] and C.A.R. Hoare [7] in the late sixties, a notable portion of the research community has attempted to introduce means for verifying the correctness of a program without executing it(mentally or on a real machine). The basic advantage of a non-execution verification approach would be that a true design phase, analogous to the common practice of modern engineering, could be introduced in the programming activity. The key feature of modern engineering is that paper prototypes are conceived and their validity verifled before the final product is built. In this way, the final product behavior is predicted with great accuracy.

The original research work on formal design of programs only attempted to show the consistency between two formal description of the program, namely the text and its specifications expressed in a mathematical formalism. This approach was called proof of correctness (clearly a misnomer since the only proof is that of consistency between two descriptions). A remarkable contribution to the formal design research was the work of E.W. Dijkstra [8] whoenvisioned a way to use the proof itself as a guidance for the thinking process used by the designer. Dijkstra has perfected this research work to a high degree, and he now presents an extremely powerful and simple technique for proofs and program design through symbol manipulation [12] [13].

The use of formal methods has had many detractors particularly in the academic world [9]. The practicality and the limits of these methods were (and still are) being questioned by many on the grounds of complexity of the proofs, lack of interest on the part of the programming community, and impossibility to scale-up the methods to large system designs, etc [9]. All these objections have been brought by notable researchers and professionals and seem to contain considerable truth. The main characteristic of these objections is that they fail to see the importance of introducing a true design phase in what is currently a cut and try process.

The most attractive feature of formal methods is their promise of introducing an"applied mathematics for software engineering" [14]. In all engineering activities. experience has shown that the introduction of a mathematical discipline into the designing process improves the confldence in the correctness of design by a considerable amount. The principal problem in the development of software of any substantial size is the exceedingly long and expensive period of system tests during which some design errors may be discovered. It seems therefore obvious that an attempt to introduce a mathematical discipline into the software design process is worth trying.

On the other hand the criticisms about the inadequacy of the currently known formal methods need to be addressed and if possible answered. Since most of these criticisms boil down to "real programmers won't do proofs", the argumentation in abstract seems to be a sterile one. Unfortunately. the growing number of published success stories [15] is not considered convincing either, since as recently as the Spring of 1990 a rather animated public debate was held during the ACM Computer Science Conference on this subject.

Of considerable practical interest is the work done recently at the University of Texas Austin. This work has been used in some practical applications. at least one of which of very substantial size and presenting all the characteristics of the system integration process sketched above.

7. A DESIGN DISCIPLINE

The University of Texas Austin work can be analyzed in two phases. First the fundamental part (due mostly to E.W. Dijkstra [12] [13]). which deals with the problem of "streamlining the mathematical argument" in such a way that rigorous proofs become concise and readable by human beings. as well as potentially automatically verifiable. In order to accomplish this goal, Predicate Calculus has been redeflned (see [13] chapters 1 through 5 for a complete description). giving evidence to the algebraic properties of the logical operators. A proof is thus reduced to a sequence of transformations of formulas into other formulas. Properties expressed by an equivalence can be proven by a series of manipulations of one of the members (usually the one looking as the most intricate) with the purpose oftransforming it in the other one. These manipulations are mostly mechanical applications of the algebraic properties of the logical operators. An elegant proof format has been defined. This format makes both the writing and reading of proofs a relatively easy task.

A second fundamental accomplishment is the definition of the semantics of a simple sequential programming language in such a way that programs can be developed by almost automatic transformation of specifications written as predicate calculus expressions.

All of the above provides a basic discipline that can be taught to individual programmers. In practice, once the programmer has mastered this discipline full armamentarium, she acquires a much more effective way of thinking. This way of thinking manifests itself by the acquisition of heuristics that guide the task of designing programs. In other words, trained programmers become more effective even though rarely needs to conduct formal proofs.

On the basis of the experience made in a large industrial organization, (Bull HN Phoenix Products Division), the learning of this programming discipline improves the individual's skills considerably. Also, the discipline is learnt by experienced programmers with reasonable ease, provided that the necessary educational period is followed by management encouragement and support. The net result is to obtain a much higher quality personnel, one of the two factors that, in Bohem measurements [5], ranks as having the greatest effectiveness.

However, the presence of better programmers may improve only the quality of the piece parts. The whole system is a different matter which is only indirectly affected by better individual modules, particularly in consideration of the fact that systems are often the integration of pre-existing modules. If the overall system design is flawed, we must expect the long and expensive system tests period to be almost unaffected by a better quality of those newly written individual modules. Clearly the acquisition of this fundamental discipline on the part of the system designers is of great advantage for their work because it encourages a more rigorous and economical way of thinking about software design. However, for the support of the global system design activity, more is needed.

This problem is attacked by the work done by Chandy and Misra, called UNITY [10].

There are many interesting features in Chandy and Misra work but for its practical application the following appears to be of fundamental importance. The design is considered a sequence of refinements of specifications (expressed in a non-imperative specification language), rather than layers of code. However, whenever a proper level of detail is reached, an algorithm can be defined for guiding the implementation.

The form of these algorithms is rather peculiar. In UNITY, an algorithm may be expressed in a programming language where a set of single simultaneous assignments can potentially be executed. No explicit sequencing operator, such as the semicolon, is available to the UNITY programmer. Of course conditions can be added to these assignments to control their execution as required by the specifications. The interesting point is that this execution model faithfully mimics the practical integration into a system of separately written modules. A powerful proof theory (using Dijkstra proof format and presentation of the predicate calculus) allows rigorous reasoning about the correctness of the system. Of particular interest is the use of the "union theorem" which states what properties must be expected from the union of two UNITY algorithms once the properties of the two parts are known. In practice the union theorem models the necessary reasoning to design the integration of new modules with an existing system.

UNITY has been applied in practice to a large scale software development and a description of this application can be found in [16].

8. TECHNOLOGY TRANSFER

One of the most difficult problems in the modern industrial world is to transform research results into better products. A reason for the steep difficulty of this process (called technology transfer, another buzz-word) is the complete separation of intents of the US research community and the industry.

The main purpose of research work is to be published for the appreciation of other researchers. The driving force for the industrial developer is cost and timeliness of product introduction. These different motivating factors make it difficult for the two worlds to communicate. Research results as published, need to be made usable by groups with very different cultural backgrounds and motivations.

This unavoidable difference of emphasis and motivation makes particularly difficult the introduction of the research on formal methods in the industry. In fact, even the argumentation against the practicality of formal methods (such as those in [9] or more recently those voiced during the 1990 ACM Computer Conference debate), are of little importance from the industry point of view. The cost and risks of system integration accomplished by an implementation driven proces, is such to stimulate considerable investments for the accommodations of formal methods. The problem is finding better programmers and system designers and this requires a disciplined way of thinking about software design.

The practical difficulties are not the absence of a community that can appreciate proofs [9] (they are seldom necessary), or the fact that proven correct programs once in execution may be unreliable because of imperfect hardware, (it is an age old problem, and in practice, almost irrelevant), but the much more mundane need to know how to design a new part that must be added to an existing system, how to describe with some precision for the posterity the meaning of an interface, etc.

In other words, the system design phase must become capable of producing a rigorous (formal) system description of the interplay between the parts, the rigorous specifications of the various part all supported by justifications of the design rationale. Occasionally these justifications need to be presented in the form of a mathematical proof. However, in most cases, this will not be necessary, because proofs would be utterly trivial. The most important advantage is the economy of thought on the part of system designers who can now rely on the guidance of a theory for the precise expression of their ideas as well as a proof method whenever necessary.

The technology transfer approach for formal methods should be essentially, but not exclusively, an educational one. The reason is that the new culture of a design driven development process would not be possible unless a receptive and sufficiently large group of developers exist. A second important point is the creation of a management structure to support the new approach.

The third point is the use of the method suitable for the problems the development organization is supposed to solve. This last point is often overlooked and it should be analyzed in light of the fact that the technology to be transferred is a way of thinking. If the formal method, for example. is adequate only to deal with some particular architecture, it should be expected that it will not be effective and therefore rejected by the designers community.

Formal methods are best suited for applied projects, possibly complex and risky rather than small and artificially created problems. Naturally, this may be considered reckless by management and therefore, never attempted. To minimize the perceived risk, however, it is always possible to precede the real project with several experiments. In this way also on the job training of the personnel can be obtained.

9. A RELEVANT PRACTICAL CASE

A project developed along the above guidelines is described in [16]. This project is a very crucial piece of software in a mainframe class operating system nucleus. The methodology used is that of UNITY, which proved to be suitable for this kind of program development. UNITY is general (absolutely architecture independent) and, as stated earlier, permits the rigorous reasoning about the integration of modules via the union theorem.

This project however was not the first UNITY application attempted in the organization. A previous example described in [11] was the formalization using UNITY of an already implemented piece of software (also in the nucleus of a mainframe class operating system). This experiment was particular instructive because it permitted the comparison of the new design with the historical record of the already implemented software. The formalization was instrumental in discovering several flaws which, in the implemented version, were discovered only after the system was undergoing system tests or was already in use at customer sites. This was sufficient argument to convince the organization of the value of applying the methodology to an ambitious project.

The ambitious project is the redesign of that portion of the GCOS8 operating system dealing with the management of the physical I/O operations and of the hardware configuration of the I/O subsystem. This portion of the operating system is the most frequently executed. The

original design is almost three decades old and was submitted to a continuous series of changes. The most recent data (last 2 years) indicated that approximately 350 change transmittals per year were applied to that code. Because of the performance requirements, the intimate interaction with hardware constructs (such as interrupt cells, mailboxes, status words, etc.) and the antiquity of the software, the code is written in assembly language.

The I/O subsystem requires considerable parallel processing not only because GCOS8 is a multiprocessor system but also because channels can change state of the main memory independently from the CPU's. In the original code and to a greater extent in the new design, the hardware configuration can be changed (by changing state of peripherals, adding or removing new equipment) without suspending the operation of the I/O subsystem. The system also allows the on-line test and diagnostic of any hardware box in the I/O hardware.

The introduction of changes in the current I/O subsystem is very difficult, Actually it is considered the most expensive system modification task for the development organization. This high level of expense is due to the very long period needed to identify the particular lines of code to be changed and the even longer period of time necessary to be confident of a reasonable level of stability once the changes are integrated in the system.

The process of formal design took a long period of time (approximately 12 months) because all the possible interactions of the rest of the system with the I/O subsystem needed to be discovered and then formalized. Conversations with developers, supported by code reading, were used as input and once formal specifications were written they were checked with the developers again. A first set of formal specifications were written in about a month on the basis of an informal design document prepared by the author and other developers. These formal specifications underwent continous changes in order to accommodate the reality of the current system. Since UNITY specifications are layered and the consistency between layers is subject of formal proving, the proof process was instrumental in clarifying many obscure points (obscure to the developers) as well as in a few instances to discover inconsistencies in the existing code. The resulting specifications were used to derive a UNITY program which was then used to guide the implementation.

The code (about 5000 lines of both new code and changes), was then written in less than a week and it ran without any error for about two months of testing.

The reports describing the details of the whole design, including specifications, proofs and the UNITY program represent a total of 76 pages. All the formal work was done by Mark Staskauskas, U.T. Austin [16], and the implementation was done by Jeff Behm(Bull HN Phoenix Product Division), who completed the implementation only four months after he was first introduced to the UNITY methodology.

10. CONCLUDING REMARKS

The use of formal methods appears as the only hope for introducing a design driven process in the software development community. This objective is much more important when the growing impact of system integration is considered. However, introducing this new technology in practice will require considerable effort in people re-education as well as in reorganizing the current implementation driven management structure.

There is a growing number of practical projects which have shown that formal methods can be advantageously applied. In the case of one development organization the introduction of Dijkstra's discipline and of the UNITY technology have been extended to the design of a highly critical and difflcult portion of an operating system. Even though a complete evaluation of this experience is not yet available, the preliminary results and the evaluation of the approach used thus far should be of value for the software community.

11. REFERENCES

[1] P. Naur, B. Randell, Eds, SOFTWARE ENGINEERING, Report on a Conference Sponsored by the NATO Science Committee, Garmish, W. Germany, 1968.

[2] E. W. Dijkstra, VERBAL PRESENTATION, University of Texas, Year of Programming, Austin, Texas, 1989

[3] D. Chan, VERBAL PRESENTATION AT BULL HN, Phoenix, Arizona, 1989

[4] B. Digrius, SYSTEM INTEGRATION, Gartner Group, IBM Large System Conference, Atlanta, Georgia, 1989

[5] B. Bohem, IMPROVING SOFTWARE PRODUCTIVITY, Keynote Address at 8th IPCCC, Phoenix, Arizona, 1989

[6] R.W. Floyd, ASSIGNING, MEANING TO PROGRAMS, Proc. of a Symposium in Applied Mathematics, Providence, Rhode Island, 1967

[7] C.A.R. Hoare, AN AXIOMATIC APPROACH TO COMPUTER PROGRAMMING, CACM 12, 10, 1969

[8] E.W. Dijkstra, A DISCIPLINE OF PROGRAMMING, Prentice Hall, 1976

[9] R.A. DeMillo. R.J. Lipton. A.J. Perlis, SOCIAL PROCESSES AND PROOFS OF THEOREMS AND PROGRAMS, CACM 22, 5, 1979

[10] K. Mani Chandy, J. Misra, PARALLEL PROGRAM DESIGN: A FOUNDATION, Addison-Wesley, 1988

[11] A. Pizzarello, PARALLEL PROGRAMMING IN PRACTICE, Workshop on Parallel Programming October 1989, organized by AICA at Casalecchio del Reno, Italy.

[12] E. Cohen, PROGRAMMING IN THE 1990'S, Springer-Verlag 1990

[13] E.W. Dijkstra, C . S . Scholten, PREDICATE CALCULUS AND PROGRAM SEMANTICS, Springer-Verlag, 1990

[14] MCC, FM TODAY Newsletter of the MCC Formal Methods Transition Study Vol, 1 No. 1, MCC Corp, Austin, Texas, 1990

[15] IEEE Software, Sept 1990 and IEEE Computer, Sept 1990 - are special issues dedicated to the subject of formal methods.

[16] M. Staskauskas, AN EXCERCISE IN VERIFYING CONCURRENT PROGRAMS IN INDUSTRY: THE I/O SUBSYSTEM, to appear in the Proceedings of the IEEE Tenth Annual International Phoenix Computer and Communication Conference, 1991

NOTE DI SOFTWARE 51

HYPERX: AN ABSTRACT HYPERMACHINE

A. Limongiello, F. Soresi, A. Spinelli, W. Vannini,
Advanced Systems Studies Department, Bull HN Information Systems Italia
Milano

RIASSUNTO

In questo articolo viene descritto HyperX, una macchina astratta ipermediale e il suo modello dei dati. Particolare attenzione è data alla descrizione del processo che ha portato alla definizione della macchina astratta. Il modello dei dati è un'evoluzione dei modelli Dexter e Lange. Il modello stratificato di riferimento include la conversione a e da altri sistemi e l'integrazione con altri modelli. HyperX intende contribuire alla definizione di un modello dei dati standard e alla definizione del formato di comunicazione.

ABSTRACT

This paper defines HyperX, a hypermedia abstract machine and its data model. A special attention is given to the description of the process which brought to the abstract machine specification. The data model is an evolution of the Dexter and the Lange models. The layered reference model for hypermedia systems includes conversion to and from other systems and integration with other models. HyperX intends to contribute to the definition of a standard hypermedia data model and interchange format.

1. INTRODUCTION

This note deals with the basic layers of an abstract hypermedia machine (HyperX), and with the objects in that machine. Our purpose is to outline requirements for general purpose hypermedia systems, in order to contribute to the definition of a standard hypermedia data model and interchange format.

The machine is open: it can integrate external models and operations, and encompasses

both the logical and presentation structures [31] of hypermedia, that is, both the structured information content and the way this information is presented to the user.

The name HyperX has several meanings. The Hyper prefix wants to clarify the technology referred to (Hypermedia). X is a variable to which the "hyper" paradigm is applied; it reminds the Unix Operating System, the environment where the HyperX prototype will run; last but not least, it suggests the openness of this abstract machine, which eXchanges information among Hypermedia Systems.
In order to define a hypermedia abstract machine, we introduce the "hypermedia metaphor", which will be the focus of an abstraction and specialization process.

The starting point of the process is the structure of an Operating System; each functional layer of the OS machine will be reinterpreted according to the hypermedia metaphor. Subsequently, the data structures will be revisited, becoming our hypermedia data model.
We focus our attention on the lower layers of the machine, and namely on the data manipulation and language layers.

Our abstract machine hierarchy is also an implementation hierarchy, which ultimately relies on the Operating System machine.
In this paper, we use margin notes as substitutions for subparagraphs when the numeration would be too deep; margin notes constitute a sort of keyword-like reference to paragraphs. Whenever a word comes from the lexicon which is common in the literature, we use the italic face. Generally we quote the source explicitely only at the first occurence; further references are implicit.

Section 2 outlines the state of the art in the field.

Section 3 defines the hypermedia metaphor, and applies abstraction and specialization to the Operating
System machine, in order to express the functional layers of the hypermedia machine.

Section 4 defines the data model of the hypermachine while the section 5 defines the functionalities.

Appendix A formalises the concepts introduced in the paper.

2. STATE OF THE ART

Our goal is to capture the abstract (implementation-independent) structure of all hypertext systems. As far as regards the methodology of definition, we took a declarative attitude [19, 20, 13], which is most suitable for mathematical elaborations. We selected the abstract machine methodology ([3], [22], [11] and [37]), which permits to build implementation-independent devices. For the hypertext-specific features we used mainly [30].
Sections 2.1 and 2.2 deal with methodological issues about abstract machine specification, while section 2.3. introduces existing reference models for hypermedia.

2.1 Declarative systems

Declarative systems combine a declarative formalism with a control strategy [13] ; a specification (or program) in a declarative formalism has two possible interpretations (or semantics): one is operational, the other is mathematical. As [13] points out:

"To generalize Kowalsky's dictum "Algorithm = Logic + Control":

Declarative programming language = declarative formalism + control strategy
It is the control strategy that enables a program to be executed on a computer. The nature of the control strategy determines the operational semantics of a program, while the declarative semantics is obtained by reading the program as a logical or functional formula."

The operational interpretation explains the meaning of a specification in terms of a sequence of states of a machine (real or abstract), and usually is more intuitive, especially for programmers [8]. The mathematical (see e.g. [12]) interpretation maps a specification onto a mathematical model (a theory in an appropriate logic [26]), and is more tractable with mathematical tools.

We generalise further that equation, substituting "declarative programming language" with "declarative system". Our system has indeed a language, but it is not necessarily a programming language.
In this paper we choose as our declarative formalism the functional notation, as far as regards functionalities, and set-theoretical equations for the data structures.
The instantiation of these ideas is carried on in § 3.1.

2.2. Abstract Machine Methodology

The HyperX machine has a layered structure. Each level is characterised by a set of objects (resources) and by a language to manipolate those resources. The language at each level could be described as in § 2.1, but this description is outside the scope of this paper, so we only define the operational semantics of the lowest level as an abstract machine.

Abstract Machine Definition

"An *abstract machine* is a set of data structures, and a set of algorithms which can access and manipulate these data structures. Any abstract machine has its own machine language which is used to write *abstract programs*. [...] Abstract programs are executed by a special algorithm of the abstract machine, the *interpreter*. [...] Data structures of an Abstract Machine are encapsulated, that is, they are accessible and modifiable only through appropriate primitive operations, provided by the machine: they are defined by an abstraction process." [22]
An abstract machine is usually subdivided into abstract modules, also known as *abstract data types*.
This methodology encompasses both functional and data abstractions: functional abstraction allows to build complex functionalities from simpler ones, while data abstraction provides for representation-independent handling of data. Thus, an abstract machine is a software system which is independent of the implementation. In fact, the

result of our work will be the definition of an abstract hypermedia machine, completely independent of the implementation.

Our methodology for the design of functionalities refers to [3], and has two main characteristics: Design methodology

- it is **structured**: first, we identify the characteristics which are common to all abstraction levels; subsequently, we deal with the specific features of each level, and with the relationships among levels.
- it is specified "**language first**": the features of each level are inferred from the upper level features. Functionalities at each level are expressed with a specific language, whose semantic is most suited to handle the objects which are visible at that level.

As to data and information organisation, we were inspired by the ISO three-level approach (see, e.g., [36]), which defines three levels for data: Data organisation

1. The *information* or conceptual level.
2. The *data* or external level.
3. The *internal* or technical level.

Level 1 models the "Universe Of Discourse", from a completely abstract standpoint; level 2 focuses on how the system will manifest itself to the users; level 3 is devoted to all the implementation-dependent aspects.

2.3. Existing Reference Models

The [30] workshop on hypertext standardisation, held in Gaithersburg during January 1990, has to be considered as a milestone towards the definition of a hypermedia reference model. Of particular interest to our approach are three models: [9], [21] and [33]. For our analysis of the state of the art, only two of them were selected: [9] and [21].
We consider [9] as a reference model for current commercial products; [21], on the other hand, is a soundly formalised model using a well-known specification language. [33] was not taken into consideration for the moment, because it accounts more for methodological considerations than for the actual production of a reference model.

2.3.1. The Dexter Model

The Dexter Model [9] addresses the issue of giving a formalised synthesis of the current generation of hypermedia systems, in order to contribute to the development of interchange and interoperability standards.

The model defines three layers: *storage* (defining a network of nodes and links), *runtime* (supporting user's interaction), and *within-component* (defining the structure and the content of nodes). The model focuses on the storage layer, and on its interfaces to the other layers. Model Layers

Components, i.e. atomic pieces of multimedia information, are the basic addressable units in the storage layer, and are either atoms, links or a composite entity made up from other components. Components

Links realise relations among *components*. A link is a sequence of two or more 'endpoint specifications' or *specifiers*, each of which can be mapped by appropriate functions to (a part of) a *component*. Specifiers have direction and presentation attributes. This structure provides for directed links of arbitrary arity. Links

Components are retrieved by means of the *accessor* and *resolver* functions. The *accessor* function returns a *component* given its identifier. The *resolver* function takes a component specification, that is a description which may vary with time ("the latest paper by John Smith"), and 'resolves' it to a unique identifier (UID). Accessor and Resolver Functions

In addition to its *content*, a component also contains information on: *anchors* within that component, presentation specifications (used by the runtime layer), and a set of arbitrary attribute/value pairs. Components' structure

Addressing within a component is guaranteed by the mechanism of *anchors*: an application uniquely identifies, within a component (anchor ID), structural elements, like a region, an item, a substructure. A *link* addresses a *component's* anchor ID, and the mapping of that ID onto the proper part of the component (anchor value) is left to the specific application. Anchors
[9] treats within-component structure and runtime issues to be outside the hypertext model *per se*, and purposely does not elaborate these layers. Specific content/structure and presentation reference models will be used in conjunction with the Dexter model to take these issues into account.

The within-component layer defines the contents and structures within components and also maintains anchor-ID/anchor-value consistency through any number of editions. Anchors provide an interface between the storage and the within-component layers. Within-component layer

The runtime layer deals with presentation of information to the user. Interface between storage and runtime layers is guaranteed by presentation specification. Runtime Layer

2.3.2. The Lange Model

The Lange model [21] is an object-oriented model of hypertext, specified using the Vienna Development Method [35], which has been originated from prototyping activities in the field of electronic instrumentation.
[21] defines three main abstract data types of hypertext: nodes, networks (sets of links) and structures. These abstract data types are merged with object-oriented DB's, formally defining access operations. Presentation and browsing semantic are not included in the model, for better separation of the data and computational layers.
[21] compares hypertext to a database of documents interconnected by cross-references. The user may traverse references in any order, thus performing non-linear reading [29].

The basic information fragments are called *nodes*, whose intended content is a single topic, i.e. an atomic piece of multimedia information. Each of them has a user-defined name and a set of attributes identifying its type of content.

Node

Nodes have an internal substructure, called *schema*; actually, the schema is organised as a set of uniquely named slots, comparable to a record data type. Consecutive portions of slot contents are called *handles*, and are the smallest addressable unit in the model.

Node Substructure
Is a Schema

Links are connections between an arbitrary number of *anchors* and an arbitrary number of *destinations*. Anchors and destinations have the same domain: each of them may be an entire node, a slot inside a node, a handle inside a slot of a certain node, another link.
Thus the model encompasses both *n*-ary and 2nd-order links (see after). Links are always *invertible*, i.e. given a destination, it is possible to obtain all the links (and all the anchors) pointing there. The *buttons*, i.e. handles to which a link is attached, permit a finer linking grain than card-based systems. Links are typed (via attributes), thus making possible some special browsing functionalities.

Link

[21] introduces another interesting type of anchor, containing a function denotation; this models active links, which are interpreted at browsing time, evaluating the associated function when the link is traversed.

Networks are sets of links, somewhat similar to "webs" of [16].

Networks

Structures are introduced as alternative views on the hyperbase, thus proposing a solution to two classical problems: "getting lost in hypertext" and "hypertext linearisation". Structures impose a hierarchical structure over the non-linear hypertext. They may also be seen as an alternative to networks, defining implicit hierarchical links.

Structures

Lange's model is very open as far as regards the attributes: each basic object of the model has a set of attributes. Issues such as attribute types and domains are not specified.

Attributes

The model relies on a object-oriented database layer, which is invoked for creating new instances of objects and for such niceties as access control and version management.

Object - orientation

3. THE HYPERMACHINE

The first view of the HyperX system depicts its "conceptual level" [36].

We try to explain a hypermedia machine exploiting the well-known Operating system layered structure. This consideration originated from our current work on the experimental development of an Hypermedia Operating system: we feel that this approach has not been given sufficient attention in the literature.
We want to cast a new metaphor -the hypermedia metaphor, expressed by a "network" of nodes and links- into traditional computing systems' design; all functional system

components will be reinterpreted through this metaphor, in order to reach a definition of an hypertext machine.

On the other hand, we would like to provide a reference architecture, but no single view on the hypertext systems. We define a general framework for a whole class of hypermedia systems. In fact, we define generally what an hypermedia system is, by describing its layered architecture and its common data structures (we call them HYTE). But from our experience of hypertext researchers, we synthetised a common view on the basic functionalities of the lower layer of our architecture, what we call the netBase. Of this layer we give a declarative description of the functionalities and data structures in a set-theoretic functional language, as well as an operational informal interpretation.

First, we lay down the schema of an Operating System, as separate functional layers which interchange data in the form of BYTE's; the BYTE is an abstract data structure which becomes, at different levels, an 8-bit byte, a file, a source program, an object program, and so on.
Then, while keeping the functional layers, we introduce a new abstract data structure: the HYTE, which has the structure of a network of nodes and links.
A "node", as well as a "link", will be instantiated to different sorts of resources, according to the layer. As above said, we intend to interpret systems' design under the hypertextual metaphor.
At this point, we substitute the BYTE with the HYTE, and map accordingly the Operating System functionalities to new HyperX functionalities.
At the end of the process, we outline a layered HyperX structure, including both the newly found functionalities and the HYTE.

step 1:
    Identify usual Operating System functions and BYTE;
step 2:
    Identify the HYTE;
step 3:
    Keep the O.S. structure and redefine machine functionalities in terms of HyperX
    components handling HYTE's;
step 4:
    HYTE in the layers.

Fig. 1 - From BYTE to HYTE

The outcome of this process is a reference architecture for the HyperX system, clarifying functionalities as well as abstract data structures.

3.1. Operating System Framework (step 1)

As is outlined in [25], at the bottom of a usual OS functional hierarchy we find the Information Management (File System); this level keeps track of the resources (information), its location, use, status, etc.

User Interface
Applications
Tools
Language
File Management

Fig. 2 - The OS layers

Above this level, we have several languages and their compilers. Then, Applications and the User Interface.
In this view, a BYTE is a sort of variable which may in turn have as value an actual byte, a file, an object program, a symbolic program, an application's data structure, a succession of events, depending on context. There is just one structural commonality within all the objects in these layers: the concept of sequence, which the BYTE essentially captures.

Layer	BYTE	Interpreter
User I/F	events	humans
Applications	applicat. data structures	applications/utilities
Languages	symbolic program	editor, macro process.
Compilers	object program	compilers/assemblers
Info Management	byte/file	File System

Fig. 3 - BYTE in the layers

3.2. Identifying the HYTE (step 2)

As a second step, we replace the abstract variable BYTE with a different one, named HYTE (for HyperX BYTE), while keeping the functional decomposition of the machine. All we know about a HYTE is that it is composed of "nodes" and "links", i.e. its structure is an hypergraph instead of a sequence.

$$BYTE = thing^*$$
$$HYTE = Nodes \times Links$$
$$Nodes = thing^*$$
$$Links = Nodes \cup Nodes^2 \cup Nodes^3 \cup .... \cup Nodes^n$$

Fig. 4 - BYTEs and HYTEs

Of course a "node" (for example), will have a different identity, according to the layer; the same will happen to a "link".

3.3. Redefining machine functionalities (step 3)

As a third step, machine functionalities are modified accordingly to the new underlying structure, so as to yield a new machine, the Hypermachine.

Intelligent User Interface
Applications
HyperEditor
HyperLanguage
NetBase

Fig. 5 - The Hypermachine: the HyperX layers

In this framework, the Information Management layer becomes the NetBase, which is a node and link storage and retrieval facility. The system supports a HyperLanguage specifically devoted to node and link handling.

A HyperEditor, which combines structured document editing with network handling capabilities, has to be provided by the system, relying on the HyperLanguage facilities; applications are written in the HyperLanguage and make use of an Intelligent User Interface.

3.4. What A HYTE Is In The Different Hypermachine Layers (step 4)

The last step yields instances of a HYTE through the above layers.

Layer	Node	Link	Interpreter
User I/F	Icon	Event	Intell. UIMS
Applications	Program	Message	Tools
HyperEditor	HyperX Node	HyperX Link	HyperEditor
HyperLanguage	Object	Message	HyperLang. int.
NetBase	persist. Node	persist. Link,	NetBase machine

Fig. 6 - Identity of a HYTE through the layers

At this step, we have an abstraction variable (HYTE), which has a "node part" and a "link part". These parts are mapped on different entities at each level of the machine. Each layer has an interpreter, which dispatches the control to the different

modules of that layer. Figure 6 is explained in detail later on.
On the file system layer, the node part of the HYTE is interpreted as a document, in the general framework of ODA/SGML hierarchical structures; links are arbitrary relations among (subsets of) documents. The control program of this layer is the NetBase handler, which is essentially an object oriented documentary database.
On the language layer, nodes are seen as objects, and links as messages, in the sense of Object-Oriented programming. The interpreter program is the interpreter of the HyperLanguage. The HyperLanguage is an extension of a high level language with specific hypermedia capabilities.
On the "tools" layer, the interpretation function maps the abstract HYTE on the "real" HyperX nodes and links, via the usage of the HyperEditor, which is a program allowing to author, browse and navigate the hyperNet.

At the application level, abstract nodes become programs, and abstract links become messages (which establish relationships among nodes). There is no predefined interpreter at this level, and the control is always held by the application.
At the user interface level, nodes are graphical entities such as icons and windows, and links are events affecting those entities. The interpreter is an Intelligent User Interface Management System.

4. THE DATA MODEL

Our Abstract Variable was defined as composed by abstract parts: nodes and links, that are a generalization of a Network. Hereafter our attention will be dedicated to the NetBase layer, which is analogous to file system layer.
The basic objects identified in that layer are tree-structured multimedia documents (henceforth referred to as "Structures"), sequences of addresses into these Structures ("Anchors") and relationships among Anchors ("Links").

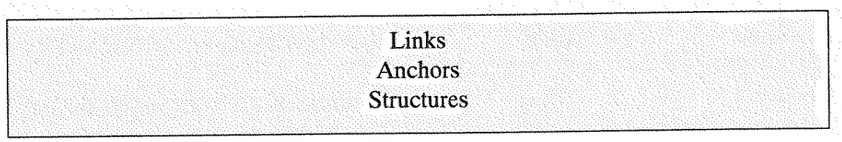

Fig. 7 - The Basic Layers

Object-orientation

Each of these basic object categories is organised as a hierarchy in the object-oriented sense. For instance, one may have classes of links, where each link inherits from the class some general attributes, but has a set of private local attributes.

Each of these objects includes, in addition to its main logical or structural meaning, additional information on how this meaning should be presented to the user, thus including an external layer in ISO-model sense [36] (see section 2.1).
For example, a textual document includes its text, its chapter-section-paragraph structure and specification of its appearance (fonts to be used, alignment, etc.).
The following three paragraphs deal with the abstract definition of the three basic object classes of HyperX.

4.1. Structures

Structures are a means to cast a framework over an otherwise unorganised stream of bytes containing multimedia information. Once a framework has been imposed, any portion of that multimedia information can be addressed.
The simplest form of Structure is the sequence, and that is the form in which information is usually stored on magnetic media. Humans, on the other hand, have a very ancient and sophisticated tradition in structuring information[1] for interchange purposes: printed (or written) documents. The structural and expressive richness has greatly evolved over time, and is currently established [23, 31], with a few exceptions [29], in a form referred to as "hierarchical documents". This form was chosen for our *Structures*: a tree-like hierarchy whose nodes and leaves contain multimedia information.

Indeed, *Structures* take more into account than the simple organisation of content; inside an instance of a Structure we distinguish: the content, the logical structure, the layout structure and a set of attribute-value pairs.

Structures = *Content* x *Logical_Structure* x *Layout_Structure* x *Attribute**

Content

Using the terminology of [31], the content is a set of *content* portions, each of which is a sequence of bytes, representing a homogeneous (monomedia) piece of multimedia information such as a text passage, a geometric-graphic image, a raster-graphic image, or even a sound-track.

Logical Structure

The *logical structure* provides organisation of the *content* in a hierarchy (a tree, as explained before). Traditional documents, for example, have a logical structure defined by Chapters, Sections, Paragraphs and so forth.

Layout Structure

The *layout structure* associates both content and logical structure to *presentation elements*, such as fonts, pages and the like.

Attributes

Each Structure has characteristics which are not part of the content, and of the logical or presentation structures (e.g. owner, date of creation, keywords). This information is represented outside the logical and the presentation structures, as a set of *attributes*: an attribute is a pair whose first element is the attribute name and whose second element is the value (an arbitrary string of bytes). Attributes are freely definable by the user.

For example, in this paper the *content* is the set of all the text fragments and of all the pictures; the logical structure is the tree of chapters/paragraphs/... . Possible attributes are the pairs:
<date, "22APR90" >,
< keywords, [hypermedia, model, abstract machine] >
<whenOpened,
send([getID(me), getUserID(me), longDate()], logfile) >.

[1] The authors recognise that distinction should be made between information and the representation of information. Documents indeed contain a rapresentation of information.

The layout structure of this document associates to the class "Paragraph" the font "Times Roman", the style "plain" the dimension "12pt", the alignment "justified". The document's logical structure places this Paragraph instance in page . The attributes of *this* Paragraph instance redefine the style of the 39th word to be "Italic".
An important consequence of the adoption of Structures is the possibility of recognising *metatext*, i.e. text speaking of other text [23]. Typical instances of metatext are tables of contents, foot- and margin-notes, quotations, bibliographies, glossaries, etc. For example, a Structure specified with SGML may contain appropriate markers for all the preceding literary devices.
With the above resorts, *Structures* exert the purpose to extract a meaning from an otherwise hermetic sequence of bytes, in order to communicate some information.

4.2. Anchors

During informal (and sometimes also formal) discussions a hypermedia is usually referred to as "a set of nodes connected by links". The concept of node may appear under different names such as *card* in NoteCards, *statement* [2], *frame* [18], but always means an "atomic piece of information", something close to a concept.

This point of view permeates the current generation of hypermedia systems, born with the purpose to provide a new means for electronic representation of documentary information.
Our model attempts to merge hypermedia technology with literary tradition, in fact, we provide complete, full-featured Structures, which model closely the traditional structured documents. This approach obviates problems found in earlier systems based on small, screen-sized "cards" or "frames". On the other hand, we still need a device for addressing points or portions of Structures; we call *Anchor* such a device.
An Anchor is therefore defined as one or more paths into Structures, a presentation specification and some attributes.

$$Anchors = Path^{+} \times Presentation\_spec \times Attribute^{*}$$

Since Structures are hierarchical, an address is a path into a tree, that is, a sequence of consecutive arcs. A set of *paths* can address an entire Structure, a substructure, a content portion, a span inside a content portions, and even a single point.
A Unix-flavoured example of path can be:
/usr/jim/report/1/2
representing section 2 of chapter 1 of the document "report" of some "jim".

Different types of content (we are dealing with multimedia information) may require different addressing mechanisms: Anchors and Paths

- a structure, a content portion, or a point inside it, is addressed with one path;
- a span of text inside a content portion is addressed with two paths (start and end point);
- a polygonal area in a bitmap image is defined as the ordered sequence of the paths of all its vertices.

In order to guarantee enough generality to encompass all these (and possibly more) addressing mechanisms, an Anchor may contain a variable number of paths (but at least one).

An Anchor also includes presentation information, specifying how a given anchor should be shown to the user.

Indeed, an Anchor can either be specified in a *declarative* or in a *procedural* fashion. In the latter case, the actions required to specify the Anchor are encoded in some of the Anchor's own attributes. The interpreter able to perform those actions is the HyperLanguage interpreter. Declarative and Procedural Anchors

As an example, if an Anchor points to a spreadsheet file, this Anchor will contain information to launch the spreadsheet, load the proper file and select the desired area. This piece of information is encoded in an attribute, e.g. "macro_commands".

The presentation specification for an anchor could include, for instance, that the area specified by the anchor should be displayed in reverse video, or flashing. Anchors' Presentation Specifications

Attributes may specify, for instance, whether an Anchor points into a Structure or into the data of a foreign application. Anchor Attributes

Anchors are objects organised in a type hierarchy in the object-oriented sense.

The purpose of Anchors is to highlight parts of Structures; typically, Anchors will identify those parts which are (or must be) related to others. Anchor Hierarchy

4.3. Links

A *Link* is a connection among anchors and/or lower-order links. Links are not restricted as concerns arity, type or direction. Each link has a type, which carries the meaning of the connection; types are organised in a user-extendable type hierarchy. Formally, the same situation can be described by considering types as *n*-ary relations, and links as elements of those relations.
In our view, link types should represent relations of some significance to the user. The complexity of connections which can be established should not be bounded, since tracing links is a stepwise process which can address very complex purposes in just a few stages.

Links pointing to other links are also known as higher-order links [21]. These are relations which range over other relations. As shown below, relations of such a kind are present in everyday language; this observation is a strong motivation to support higher-order links, since the ultimate goal of a hypermedia system is to enrich the expressiveness of natural language. Higher-order links

Links, like the other objects in our model, have a set of attributes and a set of presentation specifications.

$$Link = (Anchor\_id \cup Link\_id)^{+} \times Presentation\_spec \times Attribute^{+}$$

For example, consider some criminal records concerning thefts. Each theft will have a victim, one executor, an object which is stolen, and a modality of execution. This kind of connection can be represented by links of type "theft". That is, each link is an instance of the link class "theft", which inherits all the properties of the class "quaternary-link" which, in turn, is a "Link". Each "arc" of the link will be labelled according to the information it points to.

But if one wants to group thefts according to their type (hold-ups, ...) one needs a

second-order link which groups thefts of the same kind. This link will have, as its ancestor in the hierarchy, the "variable-arity-2nd-order-link".
Links are intended to model those pieces of knowledge which are better expressed as relationships among concepts than as isolated statements.

5. FUNCTIONALITIES

The basic operations on the data model defined in the previous section are divided into two groups: the CADM group (Create, Access, Delete and Modify), and the TI group (translation and integration).

5.1 CADM functions

The data manipulation functions need the concept of a *permanent Store*, which is capable of mapping identifiers onto values. In fact, all CADM functions accept as first argument a Store, and those functions which involve changes to the NetBase return a modified copy of the Store. We do not specify the structure of the store in full detail, since this specification is the subject matter of particular hypermedia implementations. All store-handling functions preserve the *consistency property*: a given store is consistent iff, for all anchors, if the anchor makes reference to a structure identifier, that structure identifier has a value in the store; similarly, for all links, if the link makes reference to an anchor or link identifier, that identifier has a value in the store. This property prevents the "DangLink" (dangling link) problem.

5.1.1. Creation functions

Creating a structure

The **create_structure** function takes as input the current Store, a name for the structure, a specification of the content and logical/layout structure in a suitable formalism (e.g. ODA, SGML), and a set of attribute-value pairs.

If the name of the structure is already bound to some value in the Store, the value of the function is the unchanged Store; otherwise, a new Store is returned, which is a conservative extension of the current Store, associating the Structure name to a tuple containing the content and the attribute/value pairs.

Creating anchors

create_anchor takes as input the current store, a name for the anchor, a sequence of paths into structures, a presentation specification and a set of attributes. The semantic is similar to that of **create_structure**.

Creating a link

create_link takes as input: the current store, a name for the link, a sequence of anchors names and/or link names (for higher order links), a presentation specification and a set of attributes, and, as above, persistently associates that name to that *Link*.

5.1.2. Access to objects

Given the name (or a description) of an object, it is always possible to have access to the information carried by that object.

Accessing objects

The **access_structure, access_anchor** and **access_link** functions use the current Store (passed as argument) to map names of objects onto objects' contents.

Creation is a means to add information to the system. But, if some information is already present in the system (via previous creations), there is another group of

functionalities: given an elementary object X, find all upper-level objects which make reference to X. So, each object may have answer to the question: who points to me?

Retrieval of already-existing objects

For instance, given a book, retrieve all passages (anchors) quoted by other books; given one of these passages, find the original citation (through a link).

In our model, we have two functions: **retrieve_anchors**, given a Store and a structure name, returns all the anchors which point to that structure; **retrieve_links**, given a Store and an anchor (or link) name, returns all the links which point to that anchor.

5.1.3 Deletion of objects

The functions **delete_structure, delete_anchor** and **delete_link** take care of the destruction of existing objects; as usual, they take as arguments the current Store and the name of the object to delete and return a modified Store.

5.1.4 Modifying objects

The functions **modify_structure, modify_anchor** and **modify_link** take care of the modification of existing objects; as usual, they take as arguments the current Store and the name of the object to modify, a description of the new state of the object, and return a modified Store.

5.2. TI functions"

While the CADM functions handle the Store, the Translation and Integration functions handle the Environment. We characterise an Environment by the set of all the "Abstract Data Types" (ADT) which are known in that Environment. So, HyperX is characterised by the ADT's *structure, anchor* and *link*. We define *external Environments*, with respect to HyperX, as the environments which include at least one of HyperX's ADTs.

5.2.1 Translation

By translation we mean the transformation of objects in one model into other objects in a different model, with a minimal loss of information. This implies that all the three basic ADTs of HyperX are identifiable in the external environment.
It is then reasonable to suppose that in the external model we can find essentially the same functions from a semantical standpoint, but with a different syntax. Then, translation means the transformation of our syntax into another syntax, by means of an **export** function. Of course, the passage in the opposite direction leads to the definition of the **import** function.

How are those functions defined?

In order to **import** external objects into HyperX, given the data storage of the external system, it is necessary to identify the link/anchor/structure layers; then, suitable maps must be devised, which list all external objects, namely **list_structures, list_anchors, list_link**s, which, given an external Store and the name of the external Environment, enumerate all the structures/anchors/link in that external Store. . They will establish

one-to-one maps in the best case (most probably one to many), but always respecting HyperX layers' order. In this case, translation is a one-stage process.
If suitable reference standards are available for each layer, maps between layers of the different models are easier to define. This is in favour of the adoption of official or de *facto* standards (ODA or SGML for structures, Xanadu for links,...).

5.2.2 Integration

Our model (as a matter of fact, *any* model) does not encompass all the possible domains; it may guarantee partial compatibility with other approaches only by analogy, encapsulating them in its own view. In fact, at least some of the HyperX ADTs must have an analogous in the external model, and namely, either the Link ADT or both the Link and the Anchor ADTs.

No Structures

If the external model does not have an immediate analogous of Structures, but has an analogous of Anchors and Links, HyperX will map external data objects to empty Structures referring to those external data objects. HyperX will invoke external tools to deal with external data objects, while retaining control on the anchor/link mechanism.

No Structures and no Anchors

If the external model does not provide support for anchoring, HyperX will consider the links as pointing to the whole data objects they are pointing to, i.e. there will be implicit anchors to the whole structure.

Typically, foreign applications will directly handle their data structure level, leaving to the hypermedia system the control of anchors and links. Since the application's data structure is not known, the meaning of anchors is not completely clear to the system. For instance, an anchor may contain information for performing some commands via a macro recorder.

"No Structures"

In the case of "friendly" applications, i.e. applications known to the system, some analogical interpretations of the anchors are possible. For instance, it may be possible to devise a translation from a path-like address to a range of cells in a spreadsheet.

"No Structures and No Anchor"

In the case of "alien" applications where only Links are identifiable, the system cannot have any clues on what an Anchor is specifying. A uniform addressing mechanism anyway guarantees that the alien application can still be exploited within HyperX.

An example

For example, let's suppose that HyperX is implemented on top of MS-Windows in MS-DOS environment and the user wants to create a binary link between a HyperX document (say XDoc) and WORD-for-Windows document (say WordDoc). Two anchors will be created: one will address the HyperX document in the standard way; the other will contain a sequence of commands, to launch Word, open the document and go to the desired position. Then a link is created, pointing to these two anchors.
In terms of our model, the process will be

```
create_anchor(
        anchor1,
        < [],  x doc/1/1, reverse >)
```



```
create_anchor(
        anchor2,
        <
                [type=macro],
                ["wordWordDoc","4*PgDn","6*ArrowDn],
                none >)

create_link(
        my_link,
        < [type=integration], [anchor1, anchor2], none >)
```

These actions will create two anchors, the IDs of which will be **anchor1** and **anchor2**, and a link, whose type will be "integration" and whose id will be **my_link**.

6. CONCLUSION

A prototype of the HyperX system is being implemented using an object oriented development toolkit system and a relational DBMS at our site, using Bull machinery. It will demonstrate some of the aforementioned concepts and serve as a testbed for possible commercial exploitations.
In this paper, the three-layer structure of the Dexter model has been refined towards a completely formal abstract machine specification.

The approaches from Lange, Dexter and [27] models have been synthetised in a unique data model, formalising the concept of external application.
As a result, a comprehensive set of user requirements for the standard to be defined has been presented; moreover, a set of concrete proposals has been introduced.

7. ACKNOWLEDGEMENTS

The authors wish to thank Mr. Antonio Cicu, Director of Advanced Systems Studies Department of Bull HN Italia, where this work has been accomplished, for discussions, constructive criticism, revision and support.
This study, funded by Bull Italia with the purpose of clarifying requirements for exploitation of hypermedia technology in Office and CASE applications, did contribute to some ideas which were applied by the authors in the KWICK ESPRIT II project (project nr. 2466) [38], notably the openness characteristics of the system and some aspects of the Notetaker application [39].

Appendix A. Basic Layer Specification

The basic layers are here outlined as a set of set-theoretic statements. Please note that we have indicated the exclusive union of sets A and B by the symbol $\oplus$, with the following meaning:
$A \oplus B = ((\{'A'\} \times A) \cup (\{'B'\} \times B))$,
i.e. all elements of A and B are "labeled" in order to distinguish from which set they come.

Elements of the Domain

Preliminaries

Content = Ascii_seq ⊕ Geom_graph ⊕ Raster_graph ⊕ Sound
Ascii_seq = Geom_graph = Raster_graph = Sound = Byte*
Path = Symbol⁺
Symbol = Integer ⊕ Name
Integer = [0-9]⁺
Name = [a-zA-Z]⁺
Element_id = Anchor_id = Link_id = Path
Attrval = Attr_name x Byte*
Attr_name = [a-zA-z]⁺
Present_spec = Byte*

Structures

Struct = min S such that:
if c ∈ Content <
<'leaf', Attrval*, c> ∈ S,
if $s_1,...,s_n$ ∈ S <
<'tree', Attrval*, <$s_1,...,s_n$>> ∈ S

Structures = Struct x Present_spec

Anchors

Anchors = Attrval* x Element_id⁺ x Present_spec

Links

Links = Attrval* x (Anchor_id ⊕ Link_id)⁺ x Present_spec

Store

Store = Path → (Structures ⊕ Anchor ⊕ Links)

Environment

Environments = { MSword, HyperCard, Guide, HyperX, ... }

ExternalStores = Byte*

Basic functions

Following is a specification for the basic functions in the system

Creation functions

create_link: Store x Link_id x Link → Store
create_anchor: Store x Anchor_id x Anchors → Store
create_struct: Store x Struct_id x Structures → Store

Accessor functions

access_link: Store x Link_id → Link
access_anchor: Store x Anchor_id → Anchors
access_structure: Store x Struct_id ⊕ Element_id+ → Structures
retrieve_anchor: Store x Struct_id → Anchor_id⁺
retrieve_link: Store x Anchor_id → Link_id⁺

Delete Functions

delete_link: Store x Link_id → Store
delete_anchor: Store x Anchor_id → Store
delete_structure: Store x Structure_id → Store

Modify Functions

modify_link: Store x Link_id x Links → Store
modify_anchor: Store x Anchor_id x Anchors → Store

Translation functions

modify_structure: Store x Structure_id x Structures → Store
import: External_Store x Environments → Store
export: Store x Environments → External_Store

8. BIBLIOGRAPHY

[1] Gardner, M., THE AMBIDEXTROUS UNIVERSE, 1979, Charles Scribner's Sons.

[2] Englebart, D.C., AUTHORSHIP PROVISIONS IN AUGMENT, Proceedings of the IEEE COMPCON, Spring, 1984.

[3] Baiardi F., Tomasi A., Vanneschi M., ARCHITETTURA DEI SISTEMI DI ELABORAZIONE Franco Angeli Libri Milano, Italy, 1988.

[4] Brodie M.L., Mylopoulos J., Schmidt J.W., ON CONCEPTUAL MODELLING, Springer Verlag, 1984.

[5] Carlson, D.,A., Ram, S., HYPERINTELLIGENCE: THE NEXT FRONTIER, CACM, 33(3), 1990.

[6] Collins A., Smith E.E., READINGS IN COGNITIVE SCIENCE, Morgan Kaufmann, 1988.

[7] Conklin, J., HYPERTEXT: AN INTRODUCTION AND SURVEY, Computer Magazine, September 1987.

[8] De Francesco, N., De Nicola, R., SEMANTICA OPERAZIONALE E DENOTAZIONALE DI UN SEMPLICE LINGUAGGIO FUNZIONALE, Servizio editoriale universitario, Pisa, 1987.

[9] Halasz, F., Schwartz, M., THE DEXTER HYPERTEXT REFERENCE MODEL in [30].

[10] Eberts, R.E., Eberts, C.G, Four Approaches to Human Computer Interaction, in INTELLIGENT INTERFACES, THEORY, RESEARCH AND DESIGN, P.A. Hancock & M. H. Chignell eds., North-Holland 1989.

[11] Ghezzi, C., Jazayeri, M., PROGRAMMING LANGUAGE CONCEPTS, Jon Wiley and Son, New York, 1987.

[12] Gordon, M., J., C., The denotational description of programming languages, Springer-Verlag, New York, 1979.

[13] Gregory, S., Parallel logic programming in PARLOG, Addison Wesley, Wokingham, 1987.

[14] HYPERTEXT '87 PROCEEDINGS, November 13-15, Chapel Hill, NC, ACM Press, 1987.

[15] HYPERTEXT '89 PROCEEDINGS, November 5-8, Pittsburgh Pennsylvania, ACM Press, 1989.

[16] Meyrowitz, N., Intermedia: the Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework, OOPSLA '86 PROCEEDINGS, 21(11), ACM SIGPLAN Notices, 1986.

[17] Hickman, F.R., The Pragmatic Application of the KADS Methodology, in: FIFTH INTERNATIONAL EXPERT SYSTEMS CONFERENCE, London, 68 June 1989, Learned Information, 1989.

[18] Akscyn, R., McCracken, D.L., Yoder, E.A., KMS, A Distributed Hypertext for Managing Knowledge in Organizations, COMMUNICATIONS OF THE ACM, 31(7), July 1988.

[19] Kowalsky, R., A., "Algorithm = logic + control", COMMUNICATION OF THE ACM, 22, July 1979.

[20], Kowalsky, R.,A., LOGIC FOR PROBLEM SOLVING, North Holland, New York, 1979.

[21] Lange, D., A FORMAL MODEL OF HYPERTEXT in [30].

[22] Barbuti, R., Bellia, M., Degano, P., Levi, G., PRINCIPI E TECNICHE DI PROGETTAZIONE DEL SOFTWARE, Franco Angeli, Milano, 1987.

[23] Limongiello, A., Soresi, F., Spinelli, A., Vannini, W., FROM TEXT TO HYPERTEXT THROUGH METATEXT in [30].

[24] Limongiello, A., Soresi, F., Spinelli, A., Vannini, W., AZIMUTH: A COGNITIVE HYPERMEDIA MODEL, submitted to European Conference on Hypertext, 1990.

[25] Madnick, Donovan, OPERATING SYSTEMS, Prentice-Hall, 1977.

[26] Manna, Z., Waldinger, R., THE LOGICAL BASIS FOR COMPUTER PROGRAMMING, Addison-Wesley, 1985.

[27] Marshall, C, A MULTI-TIERED APPROACH TO HYPERTEXT INTEGRATION, NEGOTIATING STANDARDS FOR A HETEROGENEOUS APPLICATION ENVIRONMENT, in [30].

[28] Morrison, A., W., Hypertext and Expert SystemsExperience and Prospects, in: FIFTH INTERNATIONAL EXPERT SYSTEMS CONFERENCE, London, 68 June 1989, Learned Information, 1989.

[29] Nelson, T.H., LITERARY MACHINES, Strathclyde 1988.

[30] PROCEEDINGS OF THE FIRST HYPERTEXT STANDARDISATION WORKSHOP, Gaithersburg MD, USA, January 16-18, 1990.

[31] ISO 8613-1, OFFICE DOCUMENT ARCHITECTURE (ODA) STANDARD, available from International Standards Organization.

[32] ISO 8879 STANDARD GENERALIZED MARKUP LANGUAGE (SGML), 1986, available from ISO.

[33] Furuta, R., Stotts, P.D., THE TRELLIS HYPERTEXT REFERENCE MODEL, in [30].

[34] Trigg, R.,H., Moran, T., P., Halasz, F., G., Adaptability and Tailorability in NoteCards, HUMAN COMPUTER INTERACTION, INTERACT '87, Bullinger, H., Shackel, B. (Eds.), Elsevier S.P., 1987.

[35] Bjorner, D., Jones, C.D., FORMAL SPECIFICATION AND SOFTWARE DEVELOPMENT, Prentice-Hall International, 1982.

[36] Vonk R., PROTOTYPING: THE EFFECTIVE USE OF CASE TECHNOLOGY, Prentice Hall 1989.

[37] Warren, D.H.D., AN ABSTRACT PROLOG INTRUCTION SET, SRI technical note 309, Menlo Park, CA, SRI International, 1983.

[38] K.W.I.C.K. Consortium KWICK TECHNICAL ANNEX 1989.

[39] Limongiello, A., Soresi, F., Spinelli, A., NOTETAKER REQUIREMENTS, D.S.S.A. Internal Note, Milan, Italy, October 1990.

HE: UNA MACCHINA ASTRATTA PER LA GESTIONE DI SISTEMI IPERMEDIALI

Edmondo Trentin [1], Paolo Miotti [2]
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano (*)

RIASSUNTO

Questa nota discute alcuni importanti aspetti attinenti le esigenze e le problematiche insite nella realizzazione di una macchina astratta per gli ipertesti, facendo riferimento ad un modello concreto, denominato HE, del quale sono illustrate le caratteristiche e le peculiarità rispetto ad altri prodotti simili.

ABSTRACT

This note discusses some important aspects and problems that arise when designing a hypertext abstract machine, by referring to a concrete model, called HE, showing its features and its peculiarities in comparison to other similar tools.

1. INTRODUZIONE

La struttura di una tipica applicazione ipertestuale [6] può generalmente essere descritta come uno strato software che da una parte interagisce con l'utente finale attraverso una opportuna interfaccia, dall'altra si occupa di organizzare le informazioni in modo coerente e funzionale su memoria di massa, facendo leva direttamente sul sistema operativo, su un DBMS [9], o su un server di rete, a seconda dei casi.

(*) Attuale Indirizzo:
[1] S.G.S. Informatica, Cavalese (TN)
[2] Etnoteam S.p.A., Milano

Mentre sono disponibili praticamente in qualsiasi ambiente pacchetti di grafica-windowing (tipicamente sotto forma di una libreria di funzioni) per la costruzione agevolata di sofisticate interfacce utente, altrettanto non si può dire di strumenti che siano di esplicito aiuto nella costruzione e gestione delle basi di dati necessarie all'hypermedia.

Sebbene siano stati realizzati dei prodotti orientati in questo senso [4], il loro utilizzo è ancora fortemente circoscritto e scarsamente esplorato. Questo nonostante le problematiche che sorgono quando il management delle basi di dati è interamente a carico degli applicativi: necessità di individuare una forma adeguata per rappresentare la struttura a grafo dei dati, algoritmi efficienti per l'accesso ai dati debugging del codice dedicato al trattamento di una siffatta base di dati. A ciò si aggiunga che applicazioni diverse, facendo affidamento su differenti modelli per l'organizzazione delle informazioni, si trovano impossibilitate ad accedere le une ai dati delle altre, dati potenzialmente di grande interesse. Questo è sicuramente uno dei maggiori ostacoli sulla via della diffusione degli ipertesti nel mondo reale.

Da quanto detto appare evidente la necessità di stru-

menti software che si propongano come modelli completi di gestione di database di tipo ipertestuale, che offrano al progettista di applicazioni una visione astratta degli oggetti di cui un ipermedia si compone, lasciandolo libero di operare su questi oggetti tramite un insieme di funzionalità ben determinate. È naturale che uno strumento del tipo appena esposto (parleremo nel seguito di "macchina astratta"), per raggiungere l'obiettivo prefissosi, dovrà possedere una sufficiente generalità, che lo renda utilizzabile da parte di applicazioni strutturate in maniera estremamente differenziata, con esigenze dissimili per quel che riguarda la natura dei dati trattati (pur ricadendo questi ultimi all'interno della definizione di hypermedia).
Inoltre, in aggiunta alla generalità, una macchina astratta dovrà permettere anche modalità di caratterizzazione degli oggetti che tratta, dando ad ognuno la possibilità di imporre una certa forma, una certa tipologia, ai propri dati.
Di queste necessità, e di altre forse meno evidenti ma di indubbia importanza, si è tenuto conto durante la progettazione e lo sviluppo di HE.

2. HE: HYPERTEXT ENGINE

HE (Hypertext Engine) è una macchina astratta per la creazione e la gestione di database di natura ipertestuale.

Nella realtà implementativa è una libreria di funzioni C che solleva l'applicazione dalla necessità di occuparsi dell'organizzazione dei dati su memoria di massa, mettendo a disposizione del progettista un insieme di entità proprie di un sistema ipertestuale e di funzionalità costruite attorno a queste entità, permettendo un trattamento affidabile, omogeneo e potente dell'ipertesto stesso. Le entità in questione sono esprimibili attraverso le categorie di Ipertesto, Nodo, Link e Contesto.
Secondo il modello proposto da HE, un ipertesto è un grafo i cui archi (links) designano relazioni di tipo generico -tipo che si può di volta in volta specificare - tra coppie di nodi; informazioni possono essere associate tanto ad un nodo quanto ad un link, e non c'è (potenzialmente) limitazione al numero di Nodi e di Links, e nemmeno alla topologia del grafo (albero, foresta, grafo diretto aciclico, grafo generico...).
La natura delle informazioni è multimediale, cioé HE permette all'applicazione di associare a nodi e links informazioni di vario tipo (files di testo, immagini, codice eseguibile, suoni,...) ed in numero arbitrario. È dunque corretto parlare di hypermedia. Risulta evidente come, in questa maniera, venga favorita la facoltà di integrazione all'interno dell'ipertesto di dati ottenuti da fonti diverse e strutturati in maniera differenziata. Ciò è estremamente importante - come è sottolineato anche in [12] e [13] - qualora si tenga presente che uno degli obiettivi per il successo di un sistema ipertestuale è proprio la possibilità di sfruttare adeguatamente le informazioni già esistenti, indipendentemente dal loro formato, senza costringere chi redige i nuovi documenti a rifare tutto da capo. Riscontri applicativi di un tale aspetto possono trovarsi nei lavori presentati in [11], [16] e [17].

L'approccio usato da HE è quello di mantenere in un database centralizzato tutti i dati relativi alla struttura grafica dell'ipertesto - uno scheletro di per sé privo di contenuti informativi - e di associare ai nodi e ai links, tramite un opportuno meccanismo, le informazioni vere e proprie. Queste possono fisicamente trovarsi in qualsiasi altra locazione accessibile del sistema nel quale l'applicazione viene eseguita, ad esempio in una certa directory di un dato device. In questo modo è realizzata una utile forma di device independence: HE tratta tutte le informazioni alla stessa maniera, indipendentemente dalla loro natura e dalla loro collocazione, senza interagire direttamente con esse, ma limitandosi a registrarne l'indirizzo completo (device, pathname, ecc.) ed eventualmente la modalità di accesso.

L'uso di questi dati ai fini del reperimento fisico dell'informazione concreta è a carico dell'applicazione.
Ad ogni nodo o link si può associare un numero arbitrario di coppie (Attributo, Valore), sostanzialmente stringhe che permettono di connotare in un certo modo un oggetto all'interno di un ipertesto. È questo uno degli aspetti fondamentali di HE poiché permette all'applicazione di caratterizzare le varie entità di cui l'ipermedia si compone, in modo da adattarle ai propri fini: per esempio ad ogni link si può associare un attributo "relazione" il cui valore può essere "include", "crossreference", "supports", a seconda dei casi; si ottiene in questo modo una tipizzazione dei links che è analoga a quella attuata da numerosi sistemi ipertestuali concreti (si può farsene un'idea consultando [2], [6] e [21]). HE offre naturalmente opportuni meccanismi per accedere

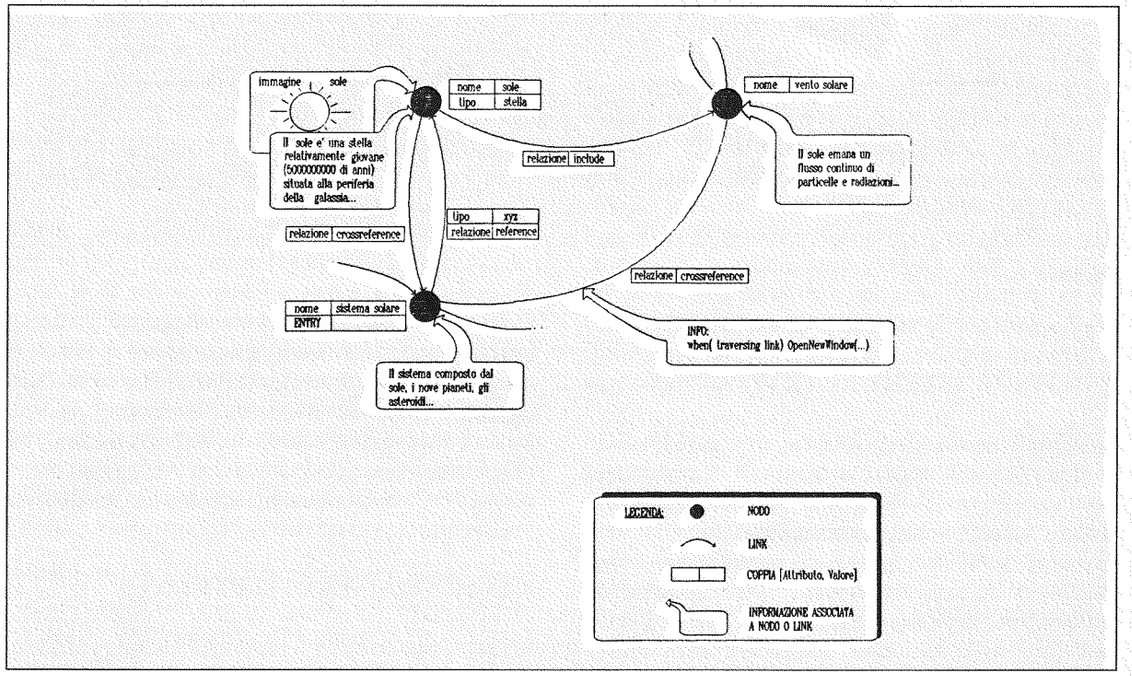

Figura 1

alle coppie (Attributo, Valore) e sfruttarle adeguatamente.

La figura 1 mostra schematicamente una porzione di un ipertesto secondo il modello proposto da HE: sono facilmente individuabili le tipologie di oggetti descritte fino a questo momento.

L'ultima entità fondamentale di cui HE si occupa è il contesto. Un contesto è sostanzialmente un sottoinsieme dei nodi (o dei links) che costituiscono l'intero ipertesto. Ogn contesto è individuato da HE in risposta ad una query che l'applicazione richiede di soddisfare: in pratica HE costruisce un contesto come l'insieme di tutti gli oggetti di un certo tipo che soddisfano la query stessa. Ogni query è un'espressione che riguarda le coppie (Attributo, Valore), le informazioni, ed altri parametri che l'oggetto che entrerà a far parte del contesto si richiede possieda. È evidente come questo discorso costituisca la risposta fornita da HE al fondamentale problema dell'Information Retrieval (I.R.) in un ipertesto [8]; il modello di I.R. basato su keywords (in [21] ne è presentata un'applicazione), ad esempio, è realizzabile nel seguente modo: si associano ai nodi degli attributi "keyword" i cui valori sono proprio le parole chiave relative ai nodi stessi; il reperimento nell'ipertesto delle informazioni attinenti ad una certa keyword W è ora ottenuto facendo leva su HE, affinché determini l'insieme dei nodi che possiedono attributo "keyword" con valore W, ossia (utilizzando la terminologia introdotta poc'anzi) il contesto dei nodi che soddisfano una query del tipo "ha attributo 'keyword' con valore W".

Da un contesto se ne può staccare un altro, più piccolo, sfruttando una ulteriore selezione in base ad una query più fine e così via.
Il meccanismo dei contesti come attuazione del principio di ricerca attraverso filtri successivi a partire dal contesto di tutti i contesti (tutto l'ipertesto) è ulteriormente potenziato dalla possibilità offerta da HE di effettuare operazioni di tipo insiemistico sui contesti stessi: unione, differenza simmetrica, intersezione, ecc.
Fare l'intersezione tra il contesto ottenuto in risposta alla query Q1 e quello ottenuto dalla query Q2 equivale, in questa ottica, a determinare il contesto (insieme)

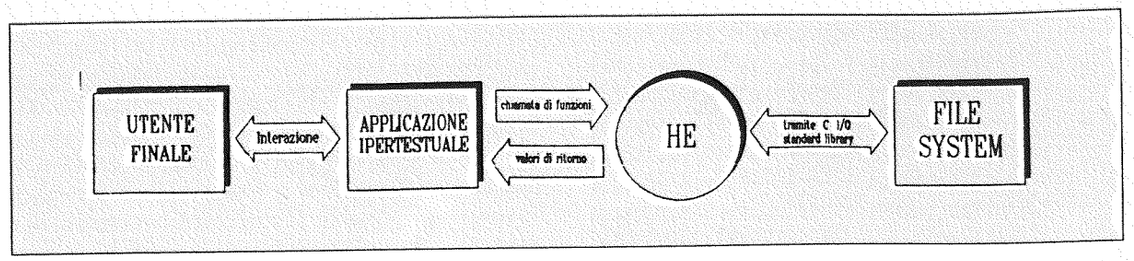

Figura 2

degli oggetti che soddisfano la query Q1 AND Q2.

3. GLI OGGETTI CHE HE TRATTA

La figura 2 illustra i rapporti intercorrenti tra le varie parti in gioco nell'uso di una applicazione ipertestuale basata su HE, con l'utente finale ed il file system del sistema operativo agli estremi opposti.
Come già accennato, HE è stato implementato come una libreria di funzioni (ottenuta dalla compilazione di 12000 Clinee di codice C e che occupa 72K di memoria): l'uso della macchina nell'ambito del sorgente avviene tramite chiamate a tali funzioni, passando opportuni parametri, come avviene nell'uso della standard library del C.

Il principio fondamentale è il seguente: ad ogni categoria di oggetto (Nodo, Link, Contesto,...) resa disponibile da HE corrisponde un nuovo tipo di dato (tipo NODE, LINK, CONTEXT,... rispettivamente). Per potere usare gli oggetti di un ipertesto si dichiarano altrettante variabili del tipo desiderato. Esse ricevono come valore uno handle (identificatore) dell'oggetto corrispondente, attraverso il valore di ritorno di determinate funzioni: ci sono funzioni che creano un nuovo oggetto e ne restituiscono lo handle, altre che forniscono l'identificatore di un nodo o di un link che soddisfa certi requisiti sugli attributi, e così via. Una volta in possesso dello handle di un oggetto, lo si può usare come parametro di altre funzioni per operare in svariati modi sull'oggetto stesso.

Da un certo punto di vista gli oggetti trattati da HE sono simili a quelli tipici della programmazione object-oriented [7]: ogni tipo (Nodo, Ipertesto,...) può essere pensato come una classe, analogamente a quanto avviene ad esempio in C++ [22]; dichiarare una variabile del tipo T ed assegnarle il valore ricavato da una funzione che genera un particolare oggetto di quel tipo, restituendone l'identificatore, equivale ad istanziare la classe T per ottenere un oggetto ben determinato. Ogni oggetto, inoltre, viene trattato da HE attraverso un certo numero di funzioni (metodi) messe a disposizione del programmatore e che agiscono in modo trasparente su dati "privati", di cui chiunque, eccetto HE, può ignorare l'esistenza.

4. FUNZIONALITÀ OFFERTE DA HE

HE fornisce innanzitutto funzionalità relative alla creazione di nuovi oggetti. Vi è libertà sul numero di oggetti che si possono creare in un ipertesto, come pure sul ruolo che in esso giocheranno: ad esempio si può generare un link come entità a sé, senza imporre ad esso un particolare nodo sorgente ed un particolare nodo destinazione, e specificare solo in un secondo tempo quali nodi il link connette. Questo lascia una maggiore libertà a chi impiega la macchina e fa giocare potenzialmente ai links un ruolo individuale più importante di quanto non avvenga nelle comuni applicazioni ipertestuali.

Come si è visto precedentemente, HE dispone poi di funzioni per associare informazioni di natura diversa a nodi e links, e per dare a questi degli attributi con certi valori; attributi e valori possono anche venire modificati in tempi successivi, per aggiornare la base di dati ad una nuova situazione, o cancellati.
Oltre alle coppie (Attributo, Valore) è possibile rimuovere naturalmente anche le informazioni associate ad un oggetto, o infine eliminare da un ipertesto nodi o links, o rimuovere l'intero ipertesto dalla propria collocazione fisica nel sistema in uso.

Apposite funzioni permettono di ottenere tutte le caratteristiche e le informazioni relative ad un certo oggetto,

ed eventualmente di alterarle.
Sono inoltre forniti meccanismi di ricerca di oggetti i cui attributi o il cui stato soddisfino certe condizioni, con la possibilità di costruire insiemi - i già citati "contesti" - di oggetti accomunati dal fatto di rispondere proprio a tali requisiti.

Altre funzioni ancora permettono di ottenere statistiche su di un ipertesto (numero di nodi o links, quantità di memoria di massa occupata espressa in numero di bytes, ecc...), di creare un duplicato dell'ipertesto stesso in una arbitraria locazione fisica del sistema, di ottenere la data e ora di ultimo accesso, di gestire un valore di "protezione" (modo d'accesso), e così via.
Interessante, da ultima, la possiblità di linkare tra loro ipertesti distinti, ricorrendo al cosiddetto "extern link" e alle funzioni ad esso collegate, rendendo così possibile la correlazione tra basi di dati diverse, create potenzialmente da utenti separati tramite applicazioni distinte. A questo proposito, è interessante osservare come - perlomeno da un punto di vista teorico - l'impiego di una macchina astratta come HE nella costruzione di applicazioni ipertestuali tende a definire uno standard, relativo alla struttura esterna dei dati. Questo coincide con la rimozione di uno dei grandi ostacoli che si trovano sulla via della diffusione degli ipertesti nel mondo reale, e va nella stessa direzione indicata in [16] per la definizione di un protocollo standard per la gestione dei links, e in [10] per quanto riguarda la creazione di una simbologia standard da adottare nelle interfacce.

La tavola I presenta un elenco delle principali funzioni di libreria, scelte tra le novantasette di cui si compone l'attuale implementazione di HE, con una breve descrizione delle finalità di ciascuna di esse.

Tavola I: alcune delle principali funzioni di HE

APERTURA/CHIUSURA DI UN IPERTESTO	
OpenHypertext()	Apre un ipertesto, creandolo se non esiste già, e ne ritorna lo handle.
CloseHypertext()	Chiude un ipertesto del quale non si ha più bisogno.
CREAZIONE DI OGGETTI	
CreateNode()	Crea un nuovo nodo e ne ritorna lo handle, senza associare ad esso attributi, informazioni o links.
CreateLink()	Crea un nuovo link e ne ritorna lo handle; permette di specificare un nodo sorgente e un nodo destinazione per il nuovo link.
CreateExternLink()	Crea un nuovo link tra nodi di ipertesti separati, ritornandone lo handle.
GetNodeContext()	Crea un insieme (contesto) dei nodi che soddisfano la query passata come parametro, e ne ritorna lo handle.
GetLinkContext()	Crea un insieme (contesto) dei links che soddisfano la query passata come parametro, e ne ritorna lo handle.
CANCELLAZIONE DI OGGETTI	
DeleteHypertext()	Rimuove la directory che rappresenta un ipertesto dal file system.
RemoveNode()	Rimuove un nodo da un ipertesto, cancellando anche i links che escono da esso e gli attributi ad esso associati. Lo handle del nodo non è più valido.
RemoveLink()	Rimuove un link da un ipertesto, cancellando anche gli attributi ad esso associati. Lo handle del link non è più valido.
RemoveContext()	Rimuove un contesto da un ipertesto. Lo handle del c non è più valido.

continua Tavola I:

LINKING DI NODI	
SetSource()	Pone un determinato nodo come sorgente di un link.
GetSource()	Ritorna lo handle del nodo sorgente del link passato come parametro.
SetDestination()	Pone un determinato nodo come destinazione di un link.
GetDestination()	Ritorna lo handle del nodo destinazione del link passato come parametro.
CARATTERIZZAZIONE DI OGGETTI	
AddNodeAttribute() AddLinkAttribute()	Associano la coppia [Attributo, Valore], passata come parametro, all'oggetto specificato.
GetNodeAttribute() GetLinkAttribute()	Queste funzioni permettono di ricavare gli attributi di un certo oggetto, i corrispondenti valori, o entrambi.
ChangeNodeAttribute() ChangeLinkAttribute() ChangeLinkAttribute()	Modificano il valore di un certo attributo di uno specifico oggetto ponendolo pari al parametro passato.
RemoveNodeAttribute() RemoveLinkAttribute()	Rimuovono una determinata coppia [Attributo, Valore] dall'oggetto cui era associata.
GESTIONE DELLE INFORMAZIONI ASSOCIATE	
AddNodeInfo() AddLinkInfo()	Permettono di specificare device, tipo e indirizzo completo della nuova informazione che si vuole associare a un certo oggetto.
GetNodeInfo() GetLinkInfo()	Ricavano l'indirizzo completo, device e tipo di un'informazione associata all'oggetto passato come parametro.
RemoveNodedeInfo() RemoveLinkInfo()	Rimuovono l'associazione tra un oggetto specificato e una certa informazione. Lo stato fisico di questa rimane inalterato.
RICERCA	
GetMatchNode() GetMatchLink()	Ritornano gli handle degli oggetti i cui attributi hanno i valori specificati come parametri.
GetNodeContext() GetLinkContext()	Queste funzioni, già descritte, sono spesso usate per la ricerca.
GENERALITA' SU UN IPERTESTO	
SetPassword()	Pone la password di accesso all'ipertesto pari alla stringa passata come parametro.
SetProtection()	Permette di specificare un livello di protezione (modo d'accesso) per l'intero ipertesto.

5. ALCUNI ASPETTI ARCHITETTURALI

Vale qui la pena di accennare ad alcuni fondamentali aspetti delle strutture dati gestite da HE. Una rappresentazione grafica dei concetti principali si può osservare in figura 3: qui hNode è una variabile che il progettista dello strato applicativo ha dichiarato essere di tipo NODE; essa ha ricevuto come valore - in risposta alla chiamata di una funzione di HE - lo handle di uno specifico nodo. Tale handle non è altro che l'indice, nella tavola dei nodi, del record contenente i dati che servono a caratterizzare il nodo stesso, tra cui alcuni puntatori ad altre tavole contenenti informazioni ad esso correlate (links di cui il nodo in questione è sorgente, le coppie (Attributo, Valore) associate al nodo stesso, ecc.).

Il discorso può essere esteso al caso generale: le informazioni necessarie alla macchina per rappresentare i dati relativi all'ipertesto vengono memorizzate in files strutturati come tavole, sequenze di records di lunghezza predefinita. Ad ogni tipo di entità trattato da HE corrisponde in pratica una tavola, l'accesso alla quale avviene direttamente tramite le funzioni di libreria di I/O standard del C. C'è dunque una tavola dei nodi, una dei links, una delle coppie (Attributo, Valore) relative ai nodi, una delle informazioni associate ai links e via di seguito.

Ad uno specifico oggetto fornito da HE, quale un determinato nodo, corrisponde un ben preciso record nella tavola relativa agli oggetti di quel tipo: l'indice di questo record nel file viene usato come handle dell'oggetto in questione. I campi dei singoli records contengono tutte le informazioni necessarie a caratterizzare un oggetto della classe corrispondente, principalmente puntatori ai records delle altre tavole correlate all'elemento in esame.

6. UN RAFFRONTO CON HAM

HAM (Hypertext Abstract Machine [4]) è una macchina astratta per la gestione di sistemi ipertestuali, esattamente come HE.
Poiché si tratta del primo e più noto prodotto del genere di cui ci sia dato di avere notizia, è interessante cercare di capire quali differenze esistono tra i due modelli.
Entrambe le macchine forniscono un modello astratto di ipertesto, si occupano di gestire la base di dati e richiedono la presenza di uno strato software che si sovrapponga ad esse e garantisca l'interazione con l'utente.

Entrambe forniscono nodi, links, contesti ed ipertesti, e la caratterizzazione degli oggetti avviene attraverso coppie (Attributo, Valore). Tuttavia, mentre le infor-

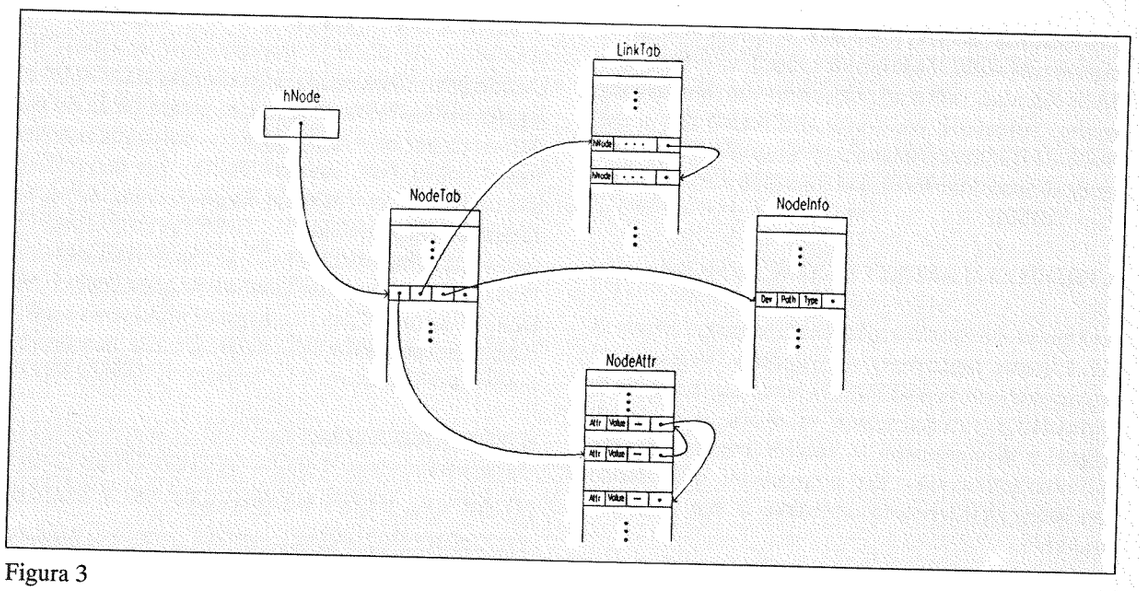

Figura 3

mazioni che HE permette di associare ai nodi e ai links sono multimediali nel vero senso della parola e senza imposizioni sul formato, HAM assume che esse siano sotto forma di files di testo o di blocchi binari di lunghezza predefinita (questa è indubbiamente una grossa limitazione, soprattutto in luce del discorso sull'integrazione di dati esterni che si era fatto (precedentemente).

Un altro aspetto di rilievo è il concetto di link: in HAM esso è, secondo la concezione tradizionale, una semplice relazione tra coppie di nodi; in HE, come si è già spiegato, si tratta invece di un oggetto autonomo, dotato di vita propria. La cosa più importante, a tale proposito, è che con HE è possibile associare informazioni anche ai links, con HAM no: questo renderebbe difficile e macchinoso, per esempio, modellare i bottoni di HyperCard, che possono essere pensati come links ai quali è associato uno script in HyperTalk; d'altra parte HAM è stato realizzato prima dello stesso HyperCard.

Anche i concetti di contesto si differenziano nelle due macchine: per HE il contesto è un insieme (di nodi oppure di links, mai di oggetti promiscui) costruito in risposta ad una determinata query, e sul quale sono possibili operazioni tipicamente insiemistiche; per HAM un contesto è invece un sottografo (nodi e links) dell'intero ipertesto, ricadendo quindi nella categoria generale di ipertesto secondo il modello proposto da HE.
Procedendo sulla falsariga di quanto si è fatto con HAM, un altro raffronto possibile - che non apporta peraltro sostanziali modifiche al discorso fino a qui svolto - è quello con il lavoro presentato in [18], dove è trattata la costruzione di una macchina astratta basata su un DBMS.

7. PROBLEMI APERTI E CONCLUSIONI

Una questione che resta aperta nella versione attuale di HE è quella riguardante il cosiddetto "versioning", sostanzialmente la possibilità di associare data e ora di particolari eventi (creazione, modifica,...) ai singoli oggetti e di conservare le vecchie copie di oggetti successivamente cancellati o modificati, mantenendo una storia dell'ipertesto attraverso il suo cammino evolutivo.

Attualmente HE non si occupa esplicitamente del versioning, che si può comunque simulare grazie alle coppie (Attributo, Valore).
Un altro aspetto importante è costituito dalla possibilità di munire HE di un interprete di un apposito linguaggio, del quale è stata analizzata la possibilità di implementazione; di esso è disponibile allo stato attuale solo un subset, costituito dal linguaggio per le query che permettono di determinare i contesti. Un interprete completo potrebbe essere invocato dall'utente per occuparsi di operazioni più generali sull'ipertesto, rimpiazzando magari l'uso delle chiamate alle funzioni di cui HE si compone.
Scripts potrebbero poi essere associati ai links (alla maniera di HyperCard) o ai nodi, per essere eseguiti al momento dell'accesso da parte della macchina a tali oggetti.

Secondo uno schema adottato a suo tempo con successo in [4], la bontà di HE è stata valutata anche cercando di simulare, mediante gli strumenti forniti dalla macchina, numerosi ipertesti concreti, quali ad esempio Guide, HyperTIES e HyperCard (come pure altri sistemi descritti in [2] e [3]). Il fatto che si sia riusciti a trovare, per ciascuna delle situazioni prese in esame, un modello rappresentativo basato su HE, è un'indicazione del fatto che il prodotto realizzato è - quanto meno - al passo con le esigenze attuali e dunque soddisfa le aspettative.

8. RINGRAZIAMENTI

Desideriamo innanzitutto ringraziare la ETNOTEAM S.p.A. che ha messo a nostra disposizione le risorse necessarie per lo sviluppo di HE; i nostri ringraziamenti vanno poi al Prof. Roberto Polillo, per le proficue discussioni; al Prof. Gianni Degli Antoni, che ci ha fornito alcuni stimolanti
motivi di riflessione; al Dott. Gilberto Romanato, al Dott. Le Van Huu, alla Dott.ssa Roberta Macchi, al Dott. Gianluigi Castelli e a quanti altri hanno contribuito, direttamente o indirettamente, alla riuscita del nostro lavoro.

9. BIBLIOGRAFIA

[1] AA.VV., HUG - HYPERTEXT USER GROUP SECONDO INCONTRO, Atti dell'incontro, Milano, 2-3 Dicembre 1988

[2] AA.VV., HYPERTEXT 87 PAPERS, The University of North Carolina, 13-15 Novembre 1987

[3] AA.VV., HYPERTEXT 89 PROCEEDINGS, ACM, 5-8 Novembre 1989

[4] Campbell B., Goodman J., HAM: A GENERAL PURPOSE HYPERTEXT ABSTRACT MACHINE, Hypertext 87 Papers, 1987

[5] Codd E.F., EXTENDING THE DATABASE RELATIONAL MODEL TO CAPTURE MORE MEANING, ACM Transactions on Database Systems, Dicembre 1979

[6] Conklin J., HYPERTEXT: AN INTRODUCTION AND SURVEY, IEEE Computer, Settembre 1987

[7] Cox B., OBJECT ORIENTED PROGRAMMING: AN EVOLUTIONARY APPROACH, Addison Wesley, 1986

[8] Croft W.B., Turtle H., A RETRIEVAL MODEL FOR INCORPORATING HYPERTEXT LINKS, Hypertext 89 Proceedings, ACM, 1989

[9] Date C.J., AN INTRODUCTION TO DATABASE SYSTEMS, Addison-Wesley, 1977

[10] Evenson S., Rheinfrank J., Wulff W., TOWARD A DESIGN LANGUAGE FOR REPRESENTING HYPERMEDIA CUES, Hypertext 89 Proceedings, ACM, 1989

[11] Fischer G., McCall R., Morch A., JANUS: INTEGRATING HYPERTEXT WITH A KNOWLEDGE-BASED DESIGN ENVIRONMENT, Hypertext 89 Proceedings, ACM, 1989

[12] Garg P.K., ABSTRACTION MECHANISMS IN HYPERTEXT, Hypertext 87 Papers, 1987

[13] Glushko R.J., DESIGN ISSUES FOR MULTI-DOCUMENT HYPERTEXTS, Hypertext 89 Proceedings, 1989

[14] Halasz F.G., REFLECTIONS ON NOTECARDS: SEVEN ISSUES FOR THE NEXT GENERATION OF HYPERMEDIA SYSTEMS, Hypertext 87 Papers, 1987

[15] Meyrowitz N., HYPERTEXT AND HYPERMEDIA SYSTEM DESIGN, ACM Seminar, Milano, 10-11 Maggio 1990

[16] Pearl A., SUN'S LINK SERVICE: A PROTOCOL FOR OPEN LINKING, Hypertext 89 Proceedings, 1989

[17] Remde J.R., Gomez L.M., Landauer T.K., SUPERBOOK: AN AUTOMATIC TOOL FOR INFORMATION EXPLORATION - HYPERTEXT?, Hypertext 87 Papers, 1987

[18] Romanato G., HMS HYPERMEDIA MANAGEMENT SYSTEM, Tesi di Laurea in Scienze dell'Informazione, Università di Milano, 1989

[19] Shneiderman B., Kearsley G., HYPERTEXT HANDS ON!, Addison-Wesley, 1989

[20] Streitz N., COGNITIVE THEORIES OF WRITING AND HYPERTEXT ENVIRONMENTS, Course Notes in Hypertext 89 Proceedings, ACM, 1989

[21] Walker J.H., DOCUMENT EXAMINER: DELIVERY INTERFACE FOR HYPERTEXT DOCUMENTS, Hypertext 87 Papers, 1987

[22] Wiener R.S., Pinson L.J., AN INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING AND C++, Addison-Wesley, 1988

UN'INTERFACCIA BASATA SULLA CONOSCENZA PER L'ANALISI DEI SEGNALI

Roberto Barbò (*), Claudio Ferri (**), Paolo Salvaneschi (**)
(*) Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
(**) ISMES S.p.A. - Bergamo

RIASSUNTO

Questo articolo descrive un'interfaccia uomo/macchina knowledge-based capace di assistere gli ingegneri nell'uso di un pacchetto software per l'analisi dei segnali. Viene quindi esaminato il modello concettuale che abbiamo sviluppato, la cui implementazione ha permesso la costruzione di una shell specializzata per la generazione di assistenti knowledge-based in questo specifico settore.

ABSTRACT

This paper describes a knowledge-based user interface able to assist engineers with the use of a signal analysis software package. A conceptual model, consisting of different types of knowledge and reasoning, has been developed and implemented, creating a specialized shell for the generation of knowledge-based aids in this specific field.

1. INTRODUZIONE

Per un effettivo uso dei pacchetti sw. di analisi dei segnali, gli ingegneri non richiedono solo basi di dati, un ricco insieme di utensili computazionali ed una efficace interfaccia uomo/macchina ma, anche, del *supporto knowledge-based* sull'utilizzo degli utensili computazionali. Contrariamente, si rischierebbe di non sfruttare appieno le potenzialità del pacchetto software.

In questo settore, esiste già della ricerca che mostra un crescente interesse ed utilizzo dei metodi e delle tecnologie dell'*Intelligenza Artificiale* [1,2]. Generalmente è stata data importanza alle nuove tecnologie di programmazione. Non è, invece, stata prestata una simile attenzione all'identificazione ed alla rappresentazione dei vari tipi di conoscenza richiesti e ad una chiara separazione tra la fase di modellazione concettuale e la fase di implementazione.

In questo contesto si inserisce il nostro lavoro [3,4] che tende a raggiungere un duplice obiettivo. In primo luogo, *modellare il processo di soluzione dei problemi e la conoscenza* utilizzata da un ingegnere durante l'attività di analisi dei segnali svolta per mezzo di software complesso. In secondo luogo, sviluppare una *shell specializzata* per la generazione di assistenti knowledge-based per attività di analisi dei segnali.

In particolare, il fine ultimo del nostro lavoro è "accrescere" un pacchetto sw di analisi dei segnali ricorrendo all'impiego di una interfaccia uomo/macchina knowledge-based, che sia in grado di dotarlo di una rappresentazione esplicita della competenza sullo specifico dominio per adempiere al ruolo di supporto.

2. LO SPECIFICO PROBLEMA

Il presente lavoro è nato dalla necessità di supportare gli ingegneri nell'utilizzo di ISA (Ismes Signal Analysis) [5]. Questo è un pacchetto sw. sviluppato dall'ISMES ed ora ampiamente usato per elaborare grandi quantità di segnali, per lo studio del comportamento dinamico di strutture civili e meccaniche, e dei fenomeni vibratori.

Le principali componenti di ISA sono una base di dati, un insieme di utensili computazionali, grafici e di gestione dati ed una interfaccia uomo/macchina. Dal punto di vista dell'utente, ISA appare come un insieme di utensili, che possono essere invocati direttamente, ed un gruppo di operatori, applicabili ad ogni utensile che sia stato precedentemente invocato.

I problemi che abbiamo dovuto affrontare non sono legati alla "maneggevolezza" di ISA, ma al livello di conoscenza relativo al dominio dell'analisi dei segnali: allo scopo di risolvere uno specifico problema, è necessario un *aiuto su quali utensili computazionali utilizzare e su come utilizzarli.* In particolare, occorre fornire un aiuto nella fase di selezione dell'appropriata attività di elaborazione da una classe di utensili candidati e nella fase di definizione dei parametri di elaborazione dell'attivita selezionata. Simili problemi possono essere risolti se si dispone di conoscenza teorica e di un buon bagaglio di euristiche sullo specifico dominio.

3. OBIETTIVI E METODI

Partendo dall'obiettivo generale di assistere gli ingegneri nell'uso del sistema ISA, abbiamo deciso di fornire del supporto per una scelta ottimale dell'attivita di elaborazione e per un suo corretto utilizzo. L'approccio seguito è stato quello di "accrescere" ISA progettando e realizzando un prototipo di interfaccia uomo/macchina knowledge-based. In primo luogo, deve essere modellato il processo di soluzione dei problemi e la conoscenza utilizzata da un ingegnere, durante il processo di problem-solving con ISA. In particolare bisogna individuare e rappresentare adeguatamente i vari tipi di conoscenza che l'ingegnere adopera durante tale processo, e separare chiaramente la fase di modellazione del problema dalla sua implementazione in uno specifico ambiente. In secondo luogo, considerata l'utilità di fornire del supporto esperto per l'esecuzione di diverse attività di analisi dei segnali, deve essere sviluppata una shell specializzata per la definizione ed esecuzione dei corrispondenti assistenti knowledge-based.

4. IL MODELLO DEL SISTEMA

Il sistema di supporto all'utilizzo di ISA è costituito da tre componenti principali: (1) un *modello della conoscenza sul dominio,* (2) dei *meccanismi di ragionamento,* (3) una *interfaccia uomo/macchina.*

Data un'attività di analisi dei segnali, è possibile, all'interno del nostro sistema, costruire un suo specifico modello di conoscenza. I meccanismi di ragionamento sono quindi in grado di considerare tale modello di conoscenza, permettendo al sistema, attraverso l'interfaccia uomo/macchina, di supportare l'utente nell'elaborazione dell'attivita corrispondente.

4.1. Il modello della conoscenza sul dominio

Tale componente consiste di un insieme di modelli, ognuno dei quali rappresenta una specifica attività di analisi dei segnali. Abbiamo identificato tre tipi di conoscenza che vengono usate nell'ambito dell'elaborazione di una qualsiasi attività di analisi dei segnali e che descrivono ogni specifico modello: *conoscenza procedurale, conoscenza euristica e conoscenza di base.*

4.1.1. Conoscenza procedurale

Il risultato dell'attivita di soluzione di problemi di un ingegnere, che utilizza un pacchetto sw. per l'analisi dei segnali, è l'identificazione di una procedura da applicare ad uno o più segnali, oppure ad una o più serie di segnali. Per una data classe di problemi noti l'ingegnere utilizza uno *schema di possibili procedure* (schema procedurale).
Un tale schema è formato da parti ben strutturate di conoscenza procedurale collegate da punti di decisione, che devono essere risolti per ottenere un'effettiva procedura.

Il formalismo utilizzato per la modellazione di uno schema procedurale è quello delle reti di Petri del tipo Posti e Transizioni (reti P/T) [6]. Tale formalismo è

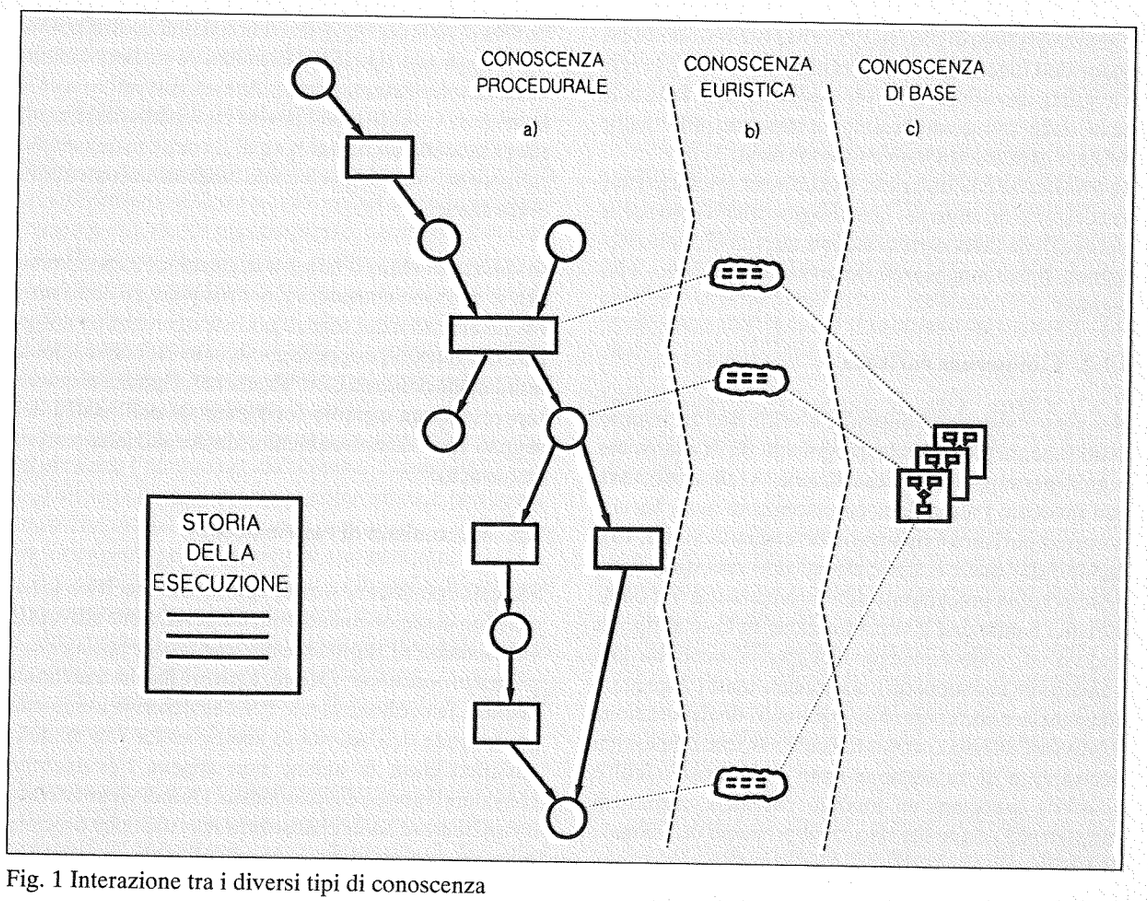

Fig. 1 Interazione tra i diversi tipi di conoscenza

stato scelto, principalmente, perchè ci permette di rappresentare esplicitamente i punti di decisione e di modellare uno schema procedurale a diversi livelli di astrazione, ma, anche, per la sua interessante rappresentazione grafica.
In figura 1a è mostrata la rete P/T che rappresenta lo schema procedurale di un'ipotetica attività di analisi dei segnali.

Le reti P/T che abbiamo usato seguono una *particolare interpretazione,* che deriva dall'aver attribuito alle loro componenti (posti, transizioni e marche) un particolare significato in relazione alla nostra applicazione. I *posti* modellano un contenitore di segnali. Particolari tipi di posti sono i punti di decisione e i punti di valutazione. I *punti di decisione* (detti punti di conflitto nel linguaggio di Petri) rappresentano situazioni in cui l'ingegnere deve decidere il cammino da seguire all'interno di uno schema procedurale. I *punti di valutazione* rappresentano degli stati in cui l'utente, non soddisfatto dai risultati ottenuti fino a quel momento, può comandare una ri-esecuzione dello schema procedurale. In questo caso, utilizzando la storia dell'esecuzione (struttura dati in cui vengono registrate le informazioni relative all'esecuzione dello schema procedurale considerato), il sistema fornisce dei consigli tenendo presente la nonbontà delle scelte fatte precedentemente. Le *transizioni* modellano un contenitore di un'attività di analisi dei segnali. Anche in questo caso esistono transizioni particolari come le transizioni composte e le transizioni con supporto. All'interno del nostro sistema è possibile associare ad una transizione, appartenente ad una certa rete P/T, un'ulteriore rete P/T. Transizioni che hanno una tale caratteristica sono dette *transizioni composte.*

L'uso combinato di più reti P/T ci ha fatto introdurre un opportuno meccanismo di controllo gerarchico: le reti a livelli di astrazione inferiori ereditano delle informazioni dalle reti gerarchicamente superiori alle quali, infine, restituiscono nuove informazioni.
Le *transizioni con supporto,* invece, sono quelle per cui è necessario un aiuto per la definizione dei parametri di elaborazione della corrispondente attività di analisi dei segnali. Infine, una *marca* rappresenta il modello di un segnale.

4.1.2. Conoscenza euristica

Nel nostro sistema, l'euristica è utilizzata all'interno dello schema procedurale di ogni attività di analisi dei segnali per supportarne l'esecuzione. A tale scopo, essa può invocare l'aiuto della conoscenza di base. La conoscenza euristica è ripartita in diversi moduli (fig. 1b), ognuno dei quali è associato ad uno specifico nodo dello schema procedurale dell'attivita scelta. I moduli euristici hanno una diversa funzione in base al tipo di nodo a cui sono connessi. Nel caso il nodo sia una transizione (con supporto), tale modulo aiuta l'ingegnere ad istanziare correttamente i parametri di elaborazione relativi all'attività computazionale associata. I punti di decisione e di valutazione sono i soli posti a cui è possibile associare un modulo euristico, capace di supportare l'utente nel rispettivo processo decisionale (4.1.b.).

Affinchè un insieme di euristiche sia effettivo [7] è necessario specificare per ogni euristica una situazione (contesto) in cui le sue azioni sono particolarmente appropriate o non-appropriate. Il formalismo di rappresentazione scelto è un linguaggio a regole, il cui modello sintattico è del tipo: IF<antecedente> THEN<conseguente> DO<azioni>.

4.1.3. Conoscenza di base

Durante un simile processo di soluzione dei problemi, l'ingegnere non utilizza solo euristiche ma anche conoscenza teorica sul dominio. Questa (fig. 1c) è costituita dalle relazioni matematiche fondamentali che esistono tra i parametri coinvolti nell'elaborazione di attività di analisi dei segnali.

Considerati i vantaggi (p.e. correttezza, modularità, facilita di spiegazione) derivanti da una codificazione esplicita di tale tipo di conoscenza, le relazioni matematiche sono state rappresentate mediante reti concetti-attori (reti C/A) [8]. Nel nostro caso particolare, ogni attore modella un contenitore di formule matematiche equivalenti, mentre un concetto modella un contenitore di parametro.

In situazioni reali, il numero di relazioni matematiche che si debbono considerare per risolvere un certo problema di analisi dei segnali ci costringerebbe ad utilizzare una rete C/A troppo ingombrante. Abbiamo quindi individuato delle reti C/A "atomiche". Ognuna di queste rappresenta un capitolo fondamentale dell'analisi dei segnali (p.e. campionamento, trasformazione tempo/frequenza).

4.2. Meccanismi di ragionamento

Su ogni tipo di conoscenza agisce uno specifico meccanismo di ragionamento. Inizialmente viene attivato il *meccanismo di ragionamento che agisce sulla conoscenza procedurale.* Questo, a partire da una marcatura iniziale, fa evolvere la rete P/T che descrive lo schema procedurale dell'attivita di analisi scelta. Un modulo euristico entra in azione non appena l'evoluzione considera il nodo a cui è connesso. Quindi viene attivato il *ragionatore sulla conoscenza euristica* che procede in modo backward con una modalità che permette di inoltrare in modo forward tutte le azioni di quelle regole i cui antecedenti sono verificati dai dati della memoria di lavoro. Il *meccanismo di ragionamento sulla conoscenza di base* viene attivato all'interno di un modulo euristico e percorre, in modo goal-directed, le reti C/A allo scopo di calcolare il valore di un certo parametro.

4.3. Interfaccia uomo/macchina

L'idea di base, su cui è incentrata l'interfaccia grafica, è l'osservazione che gran parte della conoscenza usata dal sistema è modellata con formalismi dotati di rappresentazione grafica. Inoltre i meccanismi di ragionamento relativi alla conoscenza procedurale e di base agiscono percorrendo grafi, quindi si prestano ad essere animati sullo schermo. Sostanzialmente il sistema rende comprensibile all'utente il suo funzionamento rendendo trasparente la conoscenza in esso contenuta ed il suo utilizzo. Delle funzionalità ipertestuali permettono all'utente di interagire direttamente con tutti gli oggetti grafici visualizzati sullo schermo.

5. INTEGRAZIONE CON ISA

Esistono diversi approcci che si possono seguire nella costruzione di un sistema che fornisce aiuto knowledge-based per l'uso di un complesso pacchetto software. Questi vanno dai semplici sistemi di consultazione "off-line" ai complessi "sistemi di suggerimento sempre presenti". L'approccio che abbiamo scelto per l'integrazione di questo sistema di supporto con ISA è una via di mezzo tra i due considerati sopra, e può essere caratterizzato dai seguenti requisiti:
- all'utente deve essere lasciata la possibilità di usare ISA senza l'intervento del sistema di supporto;
- il sistema di supporto deve poter lanciare una procedura ISA, al fine di generare e raccogliere le necessarie informazioni per adempiere al suo compito;
- il sistema di supporto deve essere in grado di prelevare informazioni dalla base di dati di ISA;
- tra i "side-effects" di una sessione con il sistema di supporto ci deve essere la generazione, in generale parziale, della procedura ISA che esegue il processo di elaborazione identificato.

La comunicazione avviene attraverso tre moduli la cui funzione è rispettivamente la generazione di procedure parziali di ISA, l'esecuzione dei tools di ISA ed il prelievo dei dati da ISA.

6. IMPLEMENTAZIONE DEL SISTEMA

Il nostro sistema è stato realizzato su una workstation SUN/UNIX utilizzando Nexpert Object, un ambiente ibrido per la generazione di sistemi esperti, ed il linguaggio di programmazione C. Nexpert Object è stato utilizzato come linguaggio object-oriented per l'implementazione delle reti di P/T e C/A, e nella sua forma più completa per la codifica dei moduli euristici. I meccanismi di ragionamento che agiscono su entrambi i tipi di reti sono stati realizzati, per la loro natura proceurale-ricorsiva, utilizzando il linguaggio C.

Il sistema che abbiamo sviluppato consiste in una *shell specializzata,* per la definizione e l'esecuzione di assistenti knowledge-based dell'analisi dei segnali, una *interfaccia uomo/macchina,* e di *due specifici assistenti:* uno per il calcolo dell'autodensità spettrale di potenza secondo il metodo di Bartlett (vedi 8.) e l'altro per la scelta del filtro digitale ottimale tra quelli disponibili.
Il sistema deve essere ulteriormente sviluppato in due direzioni. Da una parte, bisogna dotare la sua interfaccia uomo/macchina di una veste grafica animata ed interattività (vedi 4.3.): attualmente lo stile di interazione con l'utente è del tipo Domanda/Risposta. Dall'altra, si deve integrare il sistema con ISA (5.), considerato che nello stato attuale esso fornisce solo del supporto "off-line".

7. IL SISTEMA COME SHELL PER LA COSTRUZIONE DI ASSISTENTI KNOWLEDGE-BASED

Uno dei principali risultati raggiunti è l'aver realizzato una shell specializzata per la generazione di assistenti knowledge-based. Il sistema ISA è costituito da diverse attività di analisi dei segnali, per molte delle quali risulta utile poter costruire uno specifico assistente. La successione di passi che si devono seguire per produrre un assistente knowledge-based per un'attività di analisi dei segnali, utilizzando la shell specializzata, si può cos∞ schematizzare:

- Definizione del modello concettuale dell'attivita.
Lo scopo è la definizione concettuale del modello di conoscenza associato all'attività di analisi dei segnali in questione. Il nostro contributo, è l'aver proposto una struttura formale secondo la quale rappresentare tale modello.

- Implementazione del modello concettuale dell'attivita.
Lo scopo è la codifica del modello, identificato al passo precedente, in modo che possa essere utilizzato dal nostro sistema. Durante l'implementazione sono adoperati quegli oggetti della shell specializzata utili per costruire le reti della conoscenza procedurale e di base. Inoltre, vengono completamente riutilizzate le procedure che implementano i meccanismi di ragionamento.

8. UNO SPECIFICO ASSISTENTE KNOWLEDGE - BASED

In questa sezione riportiamo alcune note relative al prototipo dell'assistente knowledge-based per il calco-

lo dell'autodensità spettrale di potenza secondo il metodo di Bartlett (ADS).

Considerato un segnale, x(t), random, stazionario e nel dominio del tempo, lo si suddivide in campioni ("fette") di uguale lunghezza T, di ognuno dei quali viene calcolata la trasformata di Fourier $X_k(f,T)$, dove k è l'ordine relativo al campione. Quindi, il calcolo dell'autodensità spettrale di potenza si effettua computando la seguente funzione:

$$G_{xx} = \lim_{T \to \infty} 2/T * E [| X_k(f,T) |^2]$$

dove E[] indica l'operatore statistico di calcolo del valore medio.
Le situazioni che rendono problematico tale calcolo e che quindi richiedono del supporto esperto sono l'esecuzione dell'operazione di suddivisione del segnale di input (mediazione tra errore statistico e sistematico) e la scelta della finestra ottimale durante il calcolo della trasformata di Fourier.

Il modello di conoscenza relativo a queta attività è rappresentato in Figura 2. In particolare, esistono quattro moduli euristici. Il modulo N1 contiene le euristiche per la scelta di uno dei due modi disponibili per effettuare la suddivisione del segnale di input. Chiamiamo Suddivisione-A, la suddivisione che minimizza l'errore sistematico. La Suddivisione-B minimizza, invece, l'errore statistico. Il problema è bilanciare correttamente questi due tipi di errori. I moduli N2 e N3 supportano l'utente rispettivamente nella definizione dei parametri di elaborazione delle attività corrispondenti alle transizioni a cui sono connessi. Il modulo N4 aiuta l'utente nella scelta della finestra ottimale. Infine, si noti che i moduli N1 e N2 possono invocare la conoscenza di base per portare a termine il loro compito di supporto.
Tale conoscenza di base è composta da diverse reti C/A "atomiche", ognuna delle quali rappresenta uno

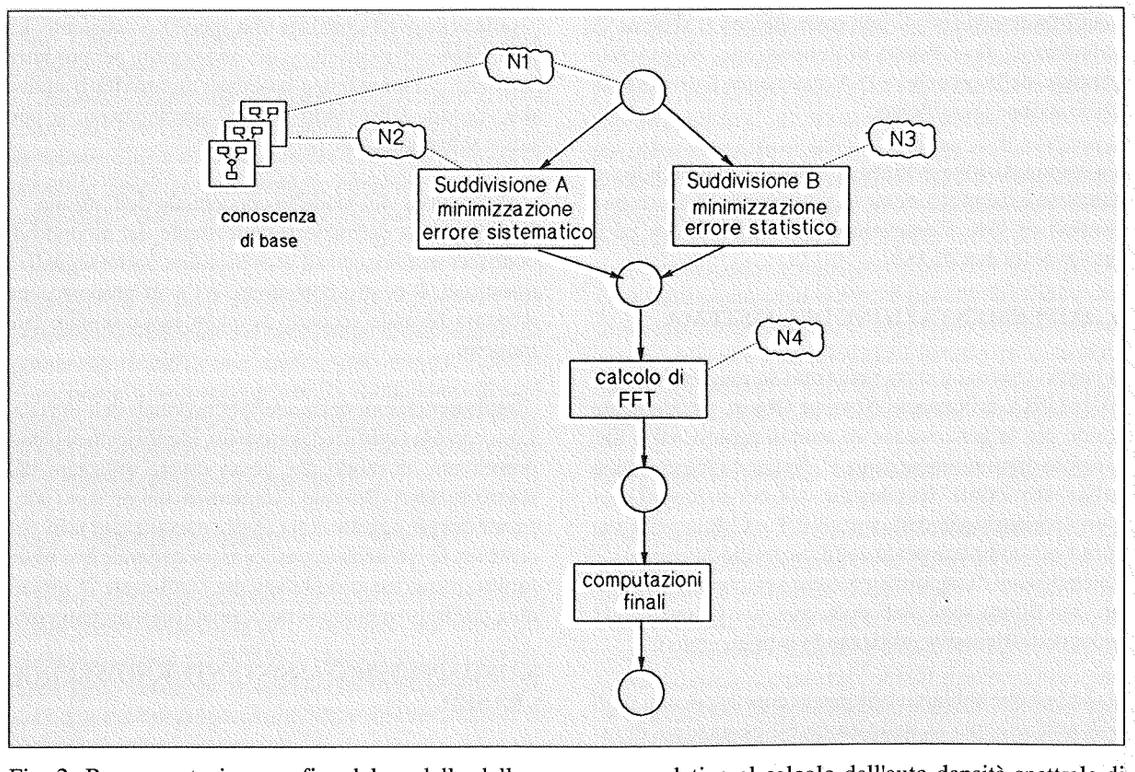

Fig. 2 Rappresentazione grafica del modello della conoscenza relativo al calcolo dell'auto densità spettrale di potenza (metodo di Bartlett)

specifico capitolo dell'analisi dei segnali ed un preciso significato nel calcolo dell'ADS. A scopo esemplificativo, in Figura 3 viene mostrata la rete C/A che rappresenta le relazioni matematiche che costituiscono il capitolo "Trasformazione Tempo-Frequenza". Il significato di questa rete è chiarito dalle seguenti formule:

(A1) $SAA = 1/LTD$ e $LTD = 1/SAA$

dove SAA è la spaziatura dell'asse delle ascisse dell'ADS e LTD è la lunghezza temporale del campione;

(A2) $VFA = 1/(2*SAI)$

dove VFA è il valore finale dell'asse delle ascisse dell'ADS e SAI è la spaziatura dell'asse delle ascisse del segnale di cui si calcola l'ADS;

(A3) $NVA = VFA/SAA + 1$ e $SAA = VFA/(NVA - 1)$

dove NVA è il numero dei valori dell'ADS;

(A4) $NVA = LDC/2 + 1$ e $LDC = 2*NVA - 2$

dove LDC è il numero dei valori del campione.

9. CONCLUSIONI

Due sono i risultati principali del nostro lavoro. Da un lato, l'aver definito un modello concettuale adatto alla rappresentazione di assistenti knowledge-based per la risoluzione di problemi di scelta ed utilizzo di attività computazionali. Tale modello integra e rappresenta separatamente diversi tipi di conoscenza. Dall'altro, l'aver utilizzato questa idea per l'implementazione di una shell specializzata per la generazione di assistenti knowledge-based per attività dell'analisi dei segnali. Con essa abbiamo sviluppato due specifici assistenti che costituiscono una prima validazione a quanto proposto.

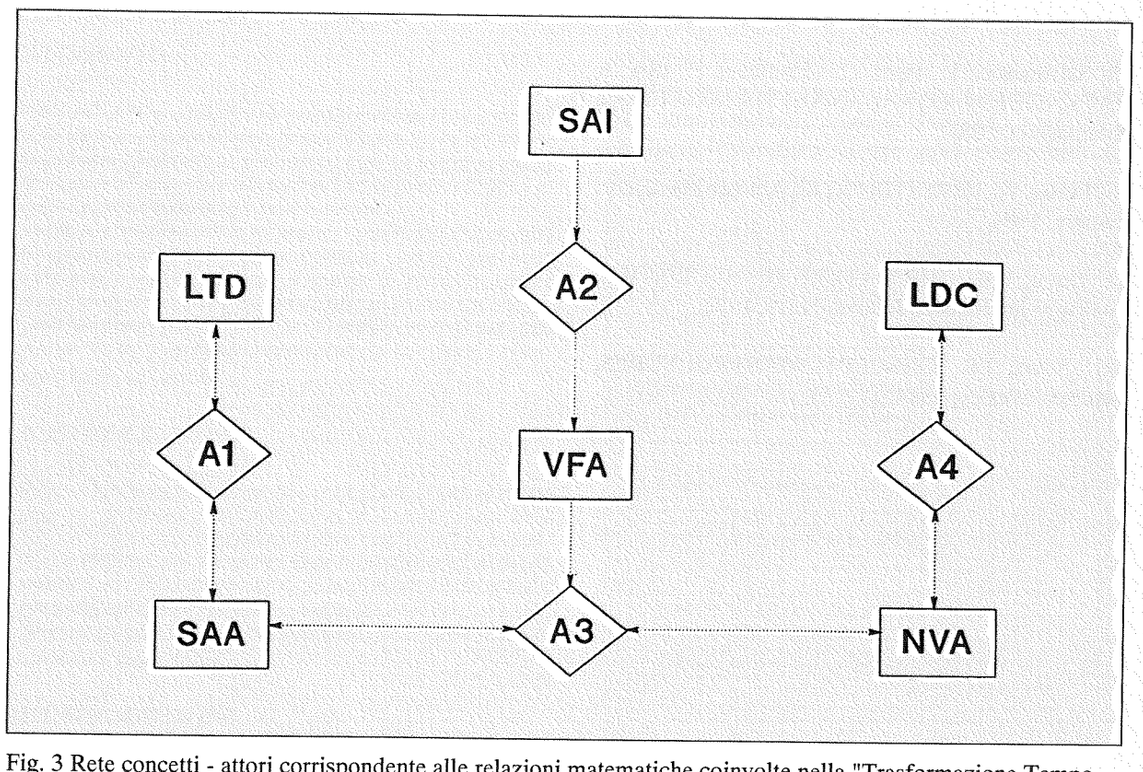

Fig. 3 Rete concetti - attori corrispondente alle relazioni matematiche coinvolte nella "Trasformazione Tempo - Frequenza".

10. BIBLIOGRAFIA

[1] Li, X., Morizet-Mahoudeaux, P., Trigano, P., Gaillard, P., A SPECTRAL ANALYSIS EXPERT SYSTEM, Dept. Gènie Informatique, Compiègne Cedex (FR), 1985.

[2] Gui Lin Nie, Morizet-Mahoudeaux, P., Gaillard, P., A KNOWLEDGE-BASED SYSTEM FOR DIGITAL FILTERS SYNTHESIS, Signal Processing IV - EURASIP, 1988.

[3] Barbò, R., UN SISTEMA INTELLIGENTE DI SUPPORTO ALL'ANALISI DEI SEGNALI, Tesi di Laurea in Scienze dell'Informazione, Università degli Studi di Milano, Febbraio 1990.

[4] Barbò, R., Ferri, C., Salvaneschi, P., A KNOWLEDGE-BASED INTERFACE TO ASSIST IN SIGNAL ANALYSIS, Signal Processing V - EURASIP, 1990.

[5] Salvaneschi, P., Ferri, C., Gambirasi, P., ISA: A PACKAGE FOR SIGNAL ANALYSIS, Decus Europe Symposium, 1985.

[6] Reisig, W., LE RETI DI PETRI, Arnoldo Mondadori Editore, 1984.

[7] Lenat, D.B., THE NATURE OF HEURISTICS, Artificial Intelligence Journal, N. 19, 1982.

[8] Sowa, J.F., CONCEPTUAL STRUCTURES, Addison-Wesley, 1984.

PROPAC: UN PROGRAMMA PER LA CONFIGURAZIONE AUTOMATICA DELLO HARDWARE DEL SISTEMA D.D.C. ME.S.A.

Angelo Beltramini (*), Fabio Casè
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
(*) ME.S.A. - Montedison Sistemi d'Automazione S.p.A. - Milano

RIASSUNTO

PROPAC (PROgramma Prolog per l'Automatica Configurazione dello hardware) è stato sviluppato presso ME.S.A. S.p.A. nell'ambito di un lavoro di tesi finalizzato alla creazione di un programma per la configurazione automatica dello hardware del sistema D.D.C. ME.S.A. S.p.A. per il controllo dei processi industriali.
PROPAC è stato realizzato con l'impiego "combinato" di tecniche di programmazione dichiarative e procedurali: esso non può essere incluso nel novero dei programmi di Intelligenza Artificiale, tuttavia alcuni interessanti concetti elaborati in tale campo sono stati vantaggiosamente impiegati per conseguire l'obiettivo primario della produzione di un programma caratterizzato da un codice sorgente aggiornabile con facilità.

ABSTRACT

PROPAC (PROlog Program for Automatic Configuration tasks) is a program developed at ME.S.A. S.p.A. to meet the needs for a ME.S.A. D.D.C. System automatic hardware configuration program. Wether PROPAC is not an Artificial Intelligence program, it is worth nothing it was realized by merging declarative and procedural programming techniques: it is believed such approach realizes an interesting compromise between computational effectiveness and easy source code understanding.

1. INTRODUZIONE

Il sistema D.D.C. (Direct Digital Control) prodotto da ME.S.A. S.p.A. trova la sua più naturale applicazione all'interno di una fabbrica (Figura 1) la quale si compone di un insieme di processi; ciascun processo si suddivide ulteriormente in processi elementari, costituiti da una unità controllata ed una unità controllante che

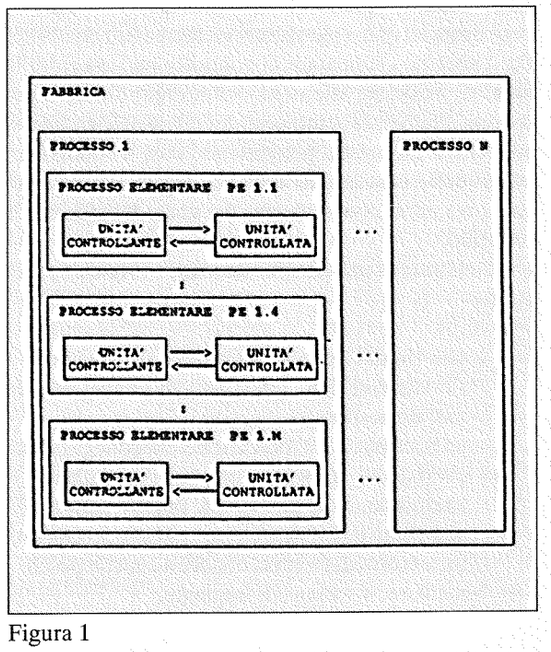

Figura 1

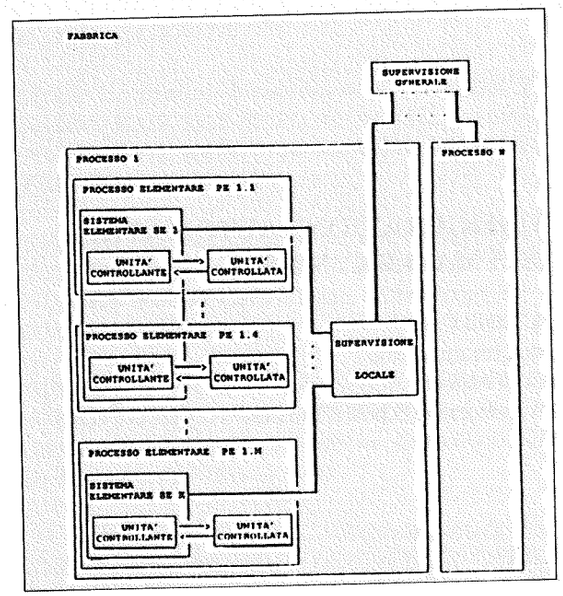

Figura 2

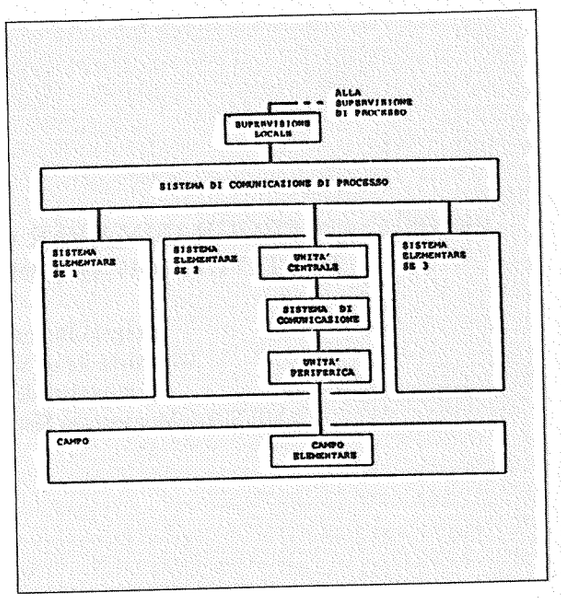

Figura 3

governa il funzionamento dell'unità controllata ad essa associata.Il sistema D.D.C. ME.S.A. S.p.A. (Figura 2) coordina le attività di ciascun processo tramite una supervisione generale che si connette ad una supervisione locale ad ogni processo; ogni supervisione locale è collegata ad uno o più sistemi elementari di controllo.
Ogni sistema elementare di controllo può eseguire le funzioni associate alle unità controllanti di uno o più processi elementari (tutti però compresi nell'ambito di uno stesso processo), permettendo così il soddisfacimento delle contrastanti esigenze legate alla velocità operativa ed al contenimento dei costi del sistema di controllo.
L'architettura di un sistema elementare (Figura 3) prevede:

- una Unità Centrale, contenente il calcolatore e l'estensione della interfaccia di I/O del calcolatore medesimo;
- un Sistema di Comunicazione, che collega l'unità centrale all'unità periferica, con modalità di trasmissione dei dati seriale oppure parallela;
- una Unità Periferica, che si connette ad una rete di sensori ed attuatori dislocati sul campo elementare, o sezione di impianto.

Il nucleo del sistema D.D.C. ME.S.A. S.p.A. è dunque rappresentato dal sistema elementare, il cui hardware è costituito da un insieme di schede aventi caratteristiche predefinite.

Il processo di configurazione dello hardware di un sistema elementare si suddivide nelle seguenti due attività fondamentali:

- determinazione del numero e della tipologia di schede necessarie per conferire al sistema elementare le caratteristiche desiderate (es.: quantità di segnali da e per il campo da gestire, velocità di acquisizione dei segnali e/o attuazione dei comandi, modalità di trasmissione dei dati attuata dal sistema di comunicazione);
- determinazione dei collegamenti da realizzare tra le schede così prescelte (includendo i collegamenti tra le schede dell'unità periferica e la rete di sensori ed attuatori dislocati sul campo elementare).

Si osservi, fatto di fondamentale importanza, che le attività appena descritte debbono svolgersi nel rispetto di precise regole imposte dalle caratteristiche dello hardware utilizzato (regole di configurazione).
ME.S.A. S.p.A. ha realizzato un ambiente per la configurazione automatica dello hardware del proprio sistema D.D.C. (Figura 4); tale ambiente è costituito da:

- l'Interfaccia Utente, che grazie ad un opportuno insieme di maschere video consente all'utente di interagire "amichevolmente" con la restante parte dell'ambiente complessivo,
- il Questionario, un programma per la costituzione guidata di una Base Dati contenente le informazioni riguardanti ciascun processo;
- il Configuratore Hardware, un programma per la configurazione automatica dello hardware di un singolo sistema elementare;
- la Base Dati, costituita con l'impiego del Questionario.

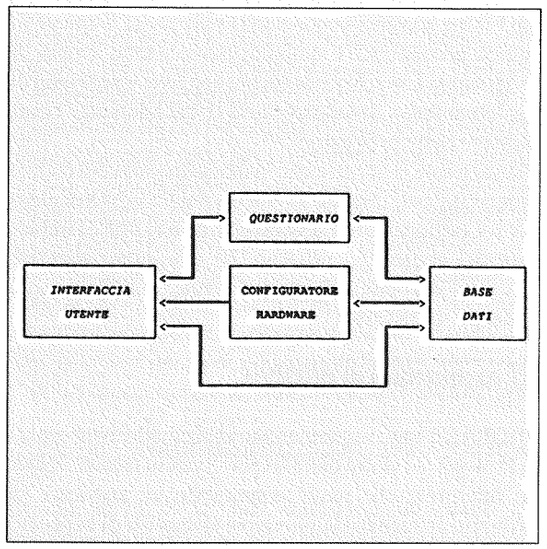

Figura 4

L'intero ambiente è implementato in Linguaggio C e risulta altamente portabile (può essere installato su PC con almeno 640 KByte di memoria RAM) nonchè caratterizzato da buone prestazioni complessive.
Sottoposto ad un approfondito esame, il Configuratore Hardware, pur dimostrandosi in grado di eseguire il processo di configurazione con efficienza estrema, ha evidenziato tuttavia difficoltà di aggiornamento del proprio codice sorgente, essenzialmente dipendenti dalla natura procedurale del linguaggio utilizzato.
L'esigenza di aggiornamento del codice, propria di qualunque prodotto software, è in questo caso determinata dalla eventuale variazione delle caratteristiche delle schede utilizzate per costituire lo hardware di un sistema elementare. Non di rado tali variazioni implicano corrispondenti variazioni dell'insieme di regole di configurazione, alle quali il Configuratore Hardware deve necessariamente adeguarsi.
Purtroppo, la natura procedurale del linguaggio utilizzato per implementare il Configuratore Hardware costringe ad esprimere le regole di configurazione in modo implicito: implementando cioè opportuni algoritmi in grado di eseguire il processo di configurazione nel rispetto delle regole corrispondenti.
Conseguentemente, l'esigenza di aggiornamento del codice si traduce spesso nella necessità di modificare, o nel caso peggiore ristrutturare totalmente, uno o più algoritmi.
In seguito, le nuove procedure implementate dovranno essere rese pienamente compatibili rispetto a quelle già esistenti, grazie ad un'opera di affinamento e messa a punto del codice tediosa e prolungata nel tempo.

2. PROPAC

La soluzione individuata per il problema in esame è stata quella di produrre un nuovo programma di configurazione, denominato PROPAC (PROgramma Prolog per la Automatica Configurazione dello hardware): esso è caratterizzato da un codice sorgente più chiaro e comprensibile, aggiornabile quindi con maggior facilità.
Per conseguire tali obiettivi si sono utilizzate tecniche di programmazione dichiarative, valutando molteplici ipotesi di soluzione: dalla implementazione di un sistema esperto [1][2] al semplice impiego di un linguaggio di programmazione non procedurale.
Gli elementi che hanno determinato la scelta finale sono legati alle esigenze, fondamentali, di compatibilità, portabilità ed efficienza: al nuovo programma era richiesto di inserirsi con piena compatibilità all'interno dell'ambiente illustrato in Figura 4 mantenendone inalterate le caratteristiche di portabilità; inoltre l'inevitabile peggioramento delle prestazioni, connesso all'impiego di tecniche di programmazione dichiarative, avrebbe dovuto risultare contenuto entro limiti accettabili.
L'ipotesi concernente l'impiego di un sistema esperto è stata scartata perchè incapace di rispettare i vincoli di portabilità, dato che un sistema esperto necessita di una quantità di memoria RAM dell'ordine dei Megabyte.

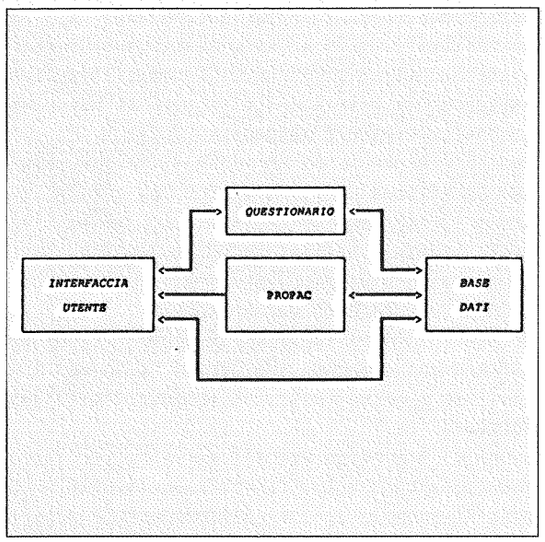

Figura 5

Inoltre ben difficilmente un sistema esperto avrebbe consentito l'elaborazione, in un tempo accettabile, delle informazioni relative ad un sistema elementare "realistico", il cui hardware è costituito da qualche centinaio di schede dedicate alla gestione di molte centinaia (talvolta migliaia) di segnali da e per il campo elementare.
L'attenzione si è pertanto orientata verso l'impiego del linguaggio Prolog [3] e vi sono molteplici ragioni che giustificano tale scelta: dalla chiarezza e sinteticità sintattica possedute dal linguaggio prescelto, alla sua particolare attitudine a rappresentare oggetti e relazioni tra oggetti [4].
A queste doti particolarmente interessanti si contrappongono una efficienza computazionale poco soddisfacente e la necessità, posta dallo specifico problema affrontato, di interagire con una base dati appositamente concepita per operare in un ambiente implementato in Linguaggio C (Figura 5).
PROPAC non è implementato esclusivamente in linguaggio prolog poichè, se così fosse, non potrebbe inserirsi nell'ambiente per la configurazione automatica; esso può definirsi piuttosto un programma "ibrido", realizzato con l'ausilio di tecniche di programmazione dichiarative e procedurali.
Il parziale impiego di codice procedurale si è reso necessario, considerata la necessità di realizzare una interfaccia software in grado di consentire al programma Prolog la consultazione della base dati; l'impiego di codice procedurale è stato poi esteso a tutte le parti di programma non particolarmente soggette a problemi di aggiornamento, limitando così il decadimento complessivo delle prestazioni.
È importante osservare che la maggior parte del codice procedurale utilizzato è identico a quello implementato per il Configuratore Hardware: la possibilità di impiegare del codice già esistente ha consentito sensibili riduzioni dei tempi di progettazione, realizzazione e collaudo di PROPAC.

2.1 Architettura di PROPAC

PROPAC è stato implementato utilizzando un interprete Prolog2, prodotto dalla Expert System International: un interprete che rende disponibile un ambiente di sviluppo arricchito da numerosi strumenti di supporto (editor, debugger, error handler,..., help on-line) e che consente l'interfacciamento tra moduli scritti in codice Prolog e moduli scritti in codice procedurale, detti moduli di codice esterno.
Il compito assegnato ai moduli di codice esterno è quello di sostituire temporaneamente l'Algoritmo di Unificazione [5] (ovvero il motore inferenziale dell'interprete Prolog) eseguendo funzioni per le quali esso non è stato progettato: nel caso di PROPAC, la cui architettura è illustrata in Figura 6, i moduli di codice

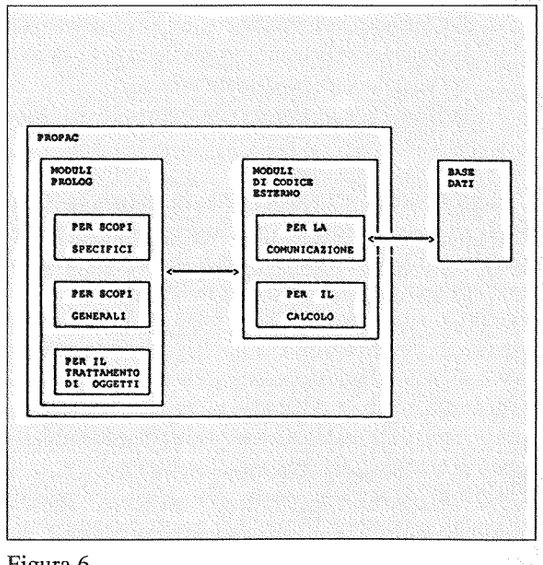

Figura 6

esterno eseguono sia funzioni di calcolo (moduli per il calcolo) che di interfacciamento con la Base Dati (moduli per la comunicazione).
Ai moduli scritti in linguaggio Prolog spetta il compito di coordinare il processo di configurazione e di eseguirne una cospicua parte: speciali accorgimenti sono stati adottati nell'intento di realizzare moduli caratterizzati da un codice sorgente facilmente aggiornabile.
In particolare, si è fatto tesoro di concetti mutuati dal campo dell'Intelligenza Artificiale quali: settorizzazione della conoscenza, meccanismi di incapsulamento dei dati (detti anche di information hiding) e meccanismi per la propagazione automatica delle informazioni.

2.2 Rappresentazione della conoscenza in PROPAC

Anche se PROPAC non costituisce un esempio di programma di Intelligenza Artificiale, la sua progettazione è stata impostata come un tipico problema di rappresentazione della conoscenza: in questo caso la conoscenza è, ovviamente, quella relativa al processo di configurazione.
Una suddivisione netta separa (settorizza) la conoscenza relativa agli oggetti coinvolti nel processo di configurazione dalla conoscenza relativa alle attività costituenti il processo medesimo: la prima è espressa all'interno dei moduli contenenti predicati per il trattamento di oggetti, la seconda all'interno dei moduli contenenti predicati per scopi generali e predicati per scopi specifici.
I moduli per il trattamento di oggetti consentono la creazione e la manipolazione controllata di oggetti (descrittori di segnale, di scheda e di sottotelaio): le variazioni eventualmente apportate alla struttura dati associata a ciascun oggetto interessano solo ed esclusivamente tali moduli, mentre i moduli rimanenti (la maggior parte) non necessitano di alcuna modifica.
Il medesimo insieme di predicati provvede, se possibile, alla propagazione automatica delle informazioni, garantendo in tale modo l'assoluta coerenza dei dati di volta in volta elaborati.
La conoscenza riguardante le attività costituenti il processo di configurazione risulta ulteriormente suddivisa in attività di alto livello ed attività di basso livello. ogni attività di alto livello è "scomponibile" in un opportuno sottoinsieme di attività di basso livello. In origine, vi era l'intenzione di associare ad ogni attività (a qualunque livello) un opportuno predicato che esprimesse le particolari proprietà logiche dell'attività medesima; ciò tuttavia avrebbe provocato una indesiderata proliferazione di predicati di basso livello, generando confusione e difficoltà di comprensione.
Successivamente, i risultati emersi conducendo una breve analisi preliminare hanno permesso di individuare molte attività di basso livello esprimibili nei termini di un comune insieme di proprietà logiche: conseguentemente è stato implementato un ridotto numero di predicati per scopi generali, successivamente adottati per la realizzazione di predicati per scopi specifici (associati cioè alle attività di alto livello).
I predicati per scopi generali contribuiscono a rendere più comprensibile ed affidabile il codice sorgente di PROPAC poichè:

- esprimono la natura fondamentale delle principali attività di basso livello costituenti il processo di configurazione: la comprensione del significato di tali attività (e del processo di configurazione in generale) risulta quindi più immediato;
- il loro ridotto numero contribuisce alla compattezza complessiva del codice sorgente;
- sono ampiamente sfruttabili per implementare predicati per scopi specifici garantendo una elevata affidabilità: ciò si traduce in sicurezza e rapidità di aggiornamento del codice sorgente.

2.3 Prestazioni di PROPAC

Sottoposto a ripetute prove di collaudo, PROPAC ha evidenziato un comportamento tipicamente lineare: il tempo richiesto per configurare un sistema elementare cresce cioè linearmente con la quantità di dati da elaborare.
Per un programma Prolog si tratta di un ottimo risultato in senso assoluto, tuttavia il Configuratore Hardware è in grado di eseguire il medesimo compito in un tempo inferiore del trenta per cento circa.
Considerando però quale parametro per una valutazione complessiva delle prestazioni anche la facilità di aggiornamento del codice sorgente, la bilancia torna a pendere a favore di PROPAC, che sotto quest'ultimo aspetto risulta assai migliore del Configuratore Hardware.
Non si deve infine dimenticare che l'architettura degli attuali calcolatori, sostanzialmente del tipo Macchina di Von Neumann, li rende poco adatti alla esecuzione di

computazioni non numeriche: macchine appositamente progettate per eseguire efficientemente l'Algoritmo di Unificazione (che è sostanzialmente un processo di pattern matching iterato) garantirebbero indubbiamente prestazioni migliori.

3. CONCLUSIONI

L'attuale versione di PROPAC, come avviene per ogni prototipo, è suscettibile di miglioramenti, sia di dettaglio che sostanziali.
Traa questi ultimi spicca la possibilità di distribuire in modo differente rispetto all'attuale le attività da svolgere tra i moduli scritti in linguaggio Prolog e i moduli scritti in linguaggio C.
A questi ultimi spetterebbe il solo compito di interfacciare l'ambiente Prolog con la Base Dati: scomparirebbero pertanto i moduli preposti al calcolo, che nella versione attuale di PROPAC sono stati inclusi più per ridurre i tempi di realizzazione che per reale necessità. PROPAC perderebbe in tal modo quasi completamente la sua peculiare connotazione di programma ibrido ma, nondimeno, potrebbero essere vantaggiosamente rivedute talune sue caratteristiche attuali, forzatamente imposte dalla necessità di interfacciare tra loro due ambienti, quello procedurale e quello dichiarativo, per natura completamente diversi tra loro.
Attualmente quello di uniformare l'ambiente di programmazione è un obiettivo ambizioso, se parallelamente non si è disposti ad accettare un sensibile decremento delle prestazioni; d'altra parte costringere il programmatore ad alternare continuamente la sua "interpretazione del mondo" da procedurale a dichiarativa è oltremodo innaturale.
La realizzazione di macchine orientate all'esecuzione di computazioni non numeriche potrebbe contribuire validamente alla soluzione di questa situazione controversa.

4. BIBLIOGRAFIA

[1] P. Jackson, INTRODUCTION TO EXPERT SYSTEMS, Addison Wesley, 1986.

[2] J. Quinlan, APPLICATION OF EXPERT SYSTEMS, Addison Wesley, 1987.

[3] W.F. Clocksin, C.S. Mellish, PROGRAMMING IN PROLOG, Springer-Verlag, 1981.

[4] H. Coelo, J.C. Lotta, L. Moniz Pereira, HOW TO SOLVE IT WITH PROLOG 3rd Edition, Laboratorio Nacional de Engenharia Civil, Lisboa 1982.

[5] J.W. Lloyd, FOUNDATIONS OF LOGIC PROGRAMMING, Springer-Verlag, 1987.

SELF-OBSERVATION IN ARTIFICIAL REALITY

Rita Pizzi
Dipartimento di Scienze della Informazione
Università degli Studi
Milano

RIASSUNTO

Poichè la Realtà Artificiale, in quanto rappresentata sullo schermo di un calcolatore, è a sua volta parte della realtà, ogniqualvolta tale schermo è compreso nello spazio di osservazione del soggetto si ottiene una riflessione infinita di livelli cognitivi.
L'articolo formalizza tale situazione in un'equazione della Realtà Artificiale", che viene dimostrato essere un'equazione al punto fisso.
Questo risultato èconfortato empiricamente da esperimenti sul videofeedback, dai quali emerge che la riflessione infinita dà luogo ad un'immagine di equilibrio che corrisponde ad un punto fisso - o ad altri attrattori - nello spazio delle immagini.
Vengono cioè generate strutture dissipative che implicano un aumento globale del contenuto informativo del sistema e sottolineano la natura temporale - oltre che spaziale - dell'autoosservazione.
L'autoosservazione quindi non è circolare perchè ogni riflessione si riferisce ad un tempo differente, che puo' osservare una realtà modificata.
Dunque un sistema in grado di autoosservarsi può trarre nuova informazione da questo procedimento, evitando situazioni di loop e potenziando le proprie capacità di problem solving.

ABSTRACT

As Artificial Reality, represented on the computer screen, is in its turn a part of reality, we obtain an infinite reflexion of cognitive levels whenever the screen is included in the subject's observation space.
The paper formalizes such a situation with an "equation of Artificial Reality", which can be shown to be a fixed-point equation.
This result is empirically supported by experiments on video-feedback, which show that infinite reflexion gives rise to an equilibrium image that corresponds to a fixed point - or other attractors - in the image space.
Therefore dissipative structures are generated, which imply an increase of the system's information content and stress the temporal nature of self-observation, apart from the spatial one.
Thus self-observation is not circular because every reflexion refers to a different time, therefore it can observe a different reality. A system able to observe itself can draw new information from this procedure, in such a way as to avoid loop situations and potentiate its problem solving capabilities.

The theoretical aspects of Artificial Reality make it possible to reconsider the deepest aspects of knowledge, of conceptual entities which constitute it, and their reciprocal connections.
In fact Artificial Reality reproduces on the computer screen in a meaningful way not only the reality image, but also its structural and cognitive connections [1].
Actually we note that Artificial Reality, represented on the screen, is in its turn a part of reality. Knowledge becomes this way a part of reality and as a part of reality it can be dealt with and elaborated.
Besides, we note that whenever the screen is included in the observation space, an infinite reflection takes place between cognitive levels.
The investigation of the concept of reflexive architecture can help to develop a realistic artificial model of

intelligence. In fact intelligence can be defined as a system able to manipulate knowledge and to manage the reflexive interaction between different representation levels.
Many authors believe that interaction between representation levels and ability to manage this interaction are the key for the development of authentically intelligent systems [2].
In fact it is well-known that computational problems can arise when computers have to deal with complex recursive situations, like self-observation problems.
It is possible to show that such limits can be seen as computational implications of Goedel's Theorem.
The philosopher J.R. Lucas maintains that non-paradoxal self-observation takes place in human brain thanks to its ability to observe reality and its modifications, and to the following comparison with its internal computational version [3].
This way the brain can attribute truth values otherwise indefinite .
As a matter of fact, a conscious being can look into himself and examine his own actions without being different from the person who acts. On the other hand, a machine has to add new instructions to its program in order to examine it, and this way it becomes a different machine, which has to be examined in its turn, and so on.
The conscious mind can interrupt this circularity because meta-knowledge and knowledge are mixed into a single observer, and they are not an infinite tower of self-observers. It is a "fold-back", i.e. a feedback between different knowledge levels [4].
It is to be noted that an advanced computer program could perform this procedure, too: as the human brain can find out a propositional truth by comparing it with reality it perceives, as a computer can be endowed with a supervisor which compares the results of an internal subroutine with the external reality.
Therefore computational circularity can be avoided by endowing the system with the possibility of observing the modifications which occur in time in the reality it is part of.
We can analyze now these considerations according to the Artificial Reality approach.
Once we define an observation space on the screen - i.e. the portion of reality with whose artificial model the subject can interact -, we can put inside of the observation space a portion R of reality:

$$r(R)$$

A screen visualizes such a reality:

$$s(R)$$

Let us consider the situation where the screen s, which is depicting the reality R contained in the observation space, becomes itself inserted in the observation space. This modified reality may in its turn be visualized on the screen, and so on, originating the following infinite sequence (fig.1)

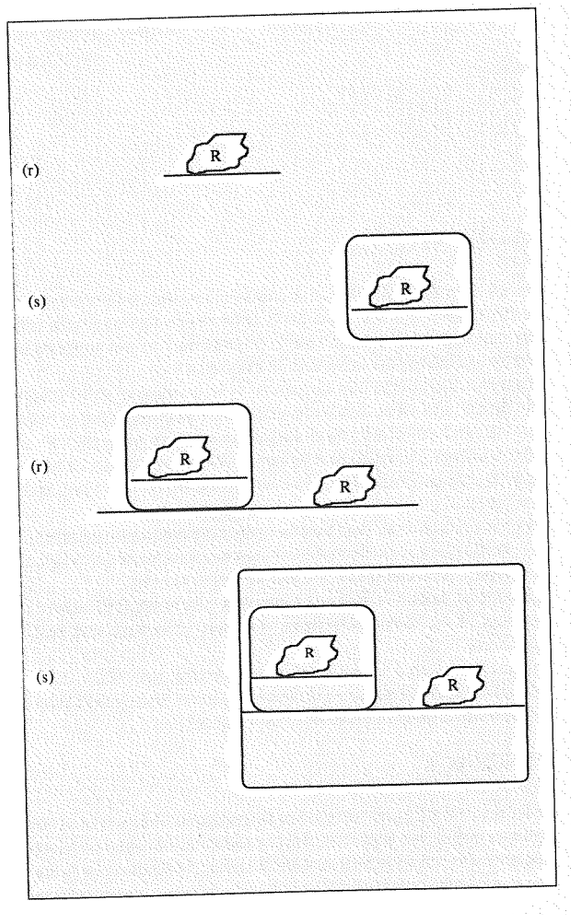

Figura 1 The sequence of the equations of A. R.

$$\begin{array}{ccc} (r) & & (s) \\ r(R) & \longrightarrow & s(R) \\ r\,(s(R)+R) & \longrightarrow & s(s(R)+R) \\ r(s(R)+R)+R) & \ldots\ldots & \end{array}$$

(where we use the operation "+" between elements of R to represent space juxtaposition), so obtaining the recurrence equation

$$\mathbb{R}_{k+1} = s(\mathbb{R}_k + R)$$
$$\mathbb{R}_0 = R$$

If the resulting sequence admits limit, we denote it as

$$\mathbb{R}^{\wedge} = \lim \mathbb{R}_k$$

and we rewrite the equation of Artificial Reality [5]

$$\mathbb{R}^{\wedge} = s(\mathbb{R}^{\wedge}) + R$$

It is to be noted that the equation of AR is a fixed-point equation.

In fact, it is possible to consider the realities $\{\mathbb{R}_k\}$ as a set ordered by the relation of approximation $\subseteq$,where information increases monotonically, i.e. the information content of $\mathbb{R}_k$ is included in that of $\mathbb{R}_{k+1}$:

$$\mathbb{R}_0 \subseteq \mathbb{R}_1 \subseteq \ldots\ldots \subseteq \mathbb{R}_{k+1} \subseteq \ldots\ldots$$

because at each step a new screen is inserted in the observation space of reality:

step k ——> $r(s(R) + \ldots\ldots + R)$
step k+1 ——> $r(s(R+\varepsilon) + \ldots\ldots + R)$ where $\varepsilon = s(R)$.

The order relation generated this way satisfies the necessary conditions:

1) reflexivity:

$$\mathbb{R}_k \subseteq \mathbb{R}_k$$

2) antisymmetry:

$$\mathbb{R}_k \subseteq \mathbb{R}_h \subseteq \mathbb{R}_k \;\Longrightarrow\; \mathbb{R}_k = \mathbb{R}_h$$

3) transitivity:

$$\mathbb{R}_k \subseteq \mathbb{R}_{k+1} \subseteq \mathbb{R}_{k+2} \;\Longrightarrow\; \mathbb{R}_k \subseteq \mathbb{R}_{k+2}$$

Now it can be easily shown that $\{\mathbb{R}_k\}$ is a complete lattice (6).
In fact,

- it is a chain, because $\forall k \quad \mathbb{R}_k \subseteq \mathbb{R}_{k+1}$.

Anyway , it is easy to show that each pair $\mathbb{R}_k$, $\mathbb{R}_h$ of the set $\{\mathbb{R}_k\}$ has both a meet and a join , consequently $\{\mathbb{R}_k\}$ is a lattice.

- it is a complete lattice because it can be easily seen that every subset has a join.

c) if we put now

$$s'(\mathbb{R}_k) = s(\mathbb{R}_k) + R$$

we see that s' has the necessary features:

- monotonicity, because due to the information content we have

$$\forall\, \mathbb{R}_k, \mathbb{R}_h, \mathbb{R}_k \subseteq \mathbb{R}_h \;\Longrightarrow\; s'(\mathbb{R}_k) \subseteq s'(\mathbb{R}_h)$$

- continuity, because it maintains the limits. In fact, due to the lattice completeness, it is possible to write

$$s'(\overset{\infty}{\underset{}{U}} \mathbb{R}_k) = \overset{\infty}{\underset{}{U}} s'(\mathbb{R}_k)$$

where $\overset{\infty}{\underset{}{U}} \mathbb{R}_k$ is the limit of the sequence $\mathbb{R}_k$.
It is in fact

$$s'(\mathbb{R}_k) = s(\mathbb{R}_k) + R = \mathbb{R}_{k+1}$$

and it follows that:

$$s'(\overset{\infty}{\underset{}{U}} \mathbb{R}_k) = \overset{\infty}{\underset{}{U}} s'(\mathbb{R}_h) = \overset{\infty}{\underset{}{U}} s'(\mathbb{R}_k)$$

i.e. the information content of $s'(\mathbb{R}_k)$ increases

with continuity with respect to the information content of $\mathbb{R}_k$.

The following fixed-point theorems are now easy to prove:

I) Due to the monotonicity of s', a unique minimal element $\mathbb{R}^\wedge$ exists such that

$$s'(\mathbb{R}^\wedge) = \mathbb{R}^\wedge$$

i.e.

$$s(\mathbb{R}^\wedge + R) = \mathbb{R}^\wedge$$

II) Due to the continuity of s', a unique minimal element $\mathbb{R}^\wedge$ exists such that

$$\mathbb{R}^\wedge = U\, s'^{\,n}(\perp)$$

where

$$m'^{\,n}(\perp) = m'(m'(.....m'(\perp)....)) = m(m(.....m(\perp)+R).....+R)+R)$$

and $\perp$ is the least element of R:

$$\perp \subseteq \mathbb{R}_k \qquad \forall k.$$

Here it is

$$\perp = R.$$

We can now observe that this result has been empirically supported by interesting studies of J.P. Crutchfield (7).
Crutchfield developed a system where a camera placed in front of a monitor converts the optical image on the monitor into an electronic signal that is then converted by the monitor into an image on its screen. This image is then electronically converted and in its turn displayed on the monitor, and so on.
The author proposes a continuum model of this video-feedback similar to the reaction-diffusion equations of chemical dynamics and biological morphogenesis. Video-feedback proves to be a dissipative dynamical system.
In fact, if we consider a single state of the video-feedback (an entire image), the state is specified by the intensity function I(x), and it follows that

$$I_{n+1} = T(I_n)$$

where I_{n+1} consists of two parts: the old image and the incoming image from the monitor screen. If we compute even the spatial diffusion which contributes to the intensity at a point, due to a spatial coupling to neighboring pixels, we have

$$(.)\quad I_{n+1}(\underline{x}) = L\ I_n(\underline{x}) + s\, f\ I_n(b\ R\ \underline{x})$$

where L is the decay factor of the old image, R is the possible image rotation, b the zoom factor, $f \subseteq [0,1]$ the parameter f/stop, $s \subseteq [+1,-1]$ the possible luminance inversion.

Crutchfield observes that Turing's equation for the evolution of the field $\underline{I}=(I_1,....,I_n)$ of concentration variables for biological morphogenesis

$$(..)\quad d\underline{I}/dt = \underline{F}(\underline{I}) + D\,\nabla^2\,\underline{I}$$

(where $\underline{F}$ is the "reaction" dynamics of the concentration variables and D the matrix of spatial coupling and diffusion rate of the concentration variables) is analogous to the continuous form of (.)

$$d\underline{I}/dt = L\,\underline{I}(\underline{x}) + s\,f\,\underline{I}(b\,R\,\underline{x}) + s\ \nabla^2\,\underline{I}(\underline{x})$$

where s is the matrix of the spatial diffusion rate.
Now, Turing showed that the equation (..) gives rise to spatial patterns that can oscillate.
Actually, the activated video-feedback gives rise to an equilibrium image which corresponds to a fixed point in the image space.
This image can be shown to be an attractor if we note that whenever we perturb the system slightly (for example by waving a finger between monitor and camera), the system gives rise to transients which are shortly replaced by the original fixed point.
Depending on the parameter settings (rotation, contrast, magnification, and so on) we always obtain attractors, in particular limit cycles or chaotic attractors.

In conclusion, the video-feedback behaviour emphasizes the temporal nature of self-observation, apart from the spatial one.
It can be asserted that reality in video-feedback is not an endless iteration of identical images, but it generates dissipative structures, which imply an entropy decrease, therefore an increase of the information content.
It is obvious that the information increase becomes really meaningful to the observer if the reality R visualized by the screen is not static but dynamic.

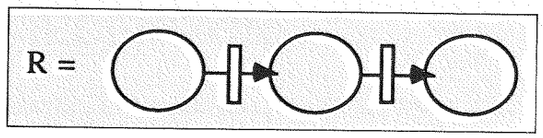

Figura 2 C/E Petri Net representing reality

Actually, if we consider reality R as a C/E Petri net, we can represent it as (fig.2)

and its modifications at each step of time will be (fig. 3)

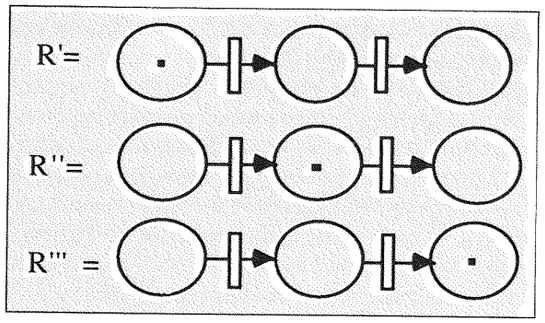

Figura 3 The C/E/ Petri Net at different times

Thus the iteration included in the equation of Artificial Reality is not inconsistent because at each step it refers to modified realities. If we represent the first steps, we obtain therefore (fig. 4)

Self-observation is not circular because every reflextion includes at each step a different time, thus it observes a different reality (fig. 5).
Such a conclusion agrees with the theories above mentioned, which maintain that computational circularity can be avoided if the system is able to observe its own modifications.

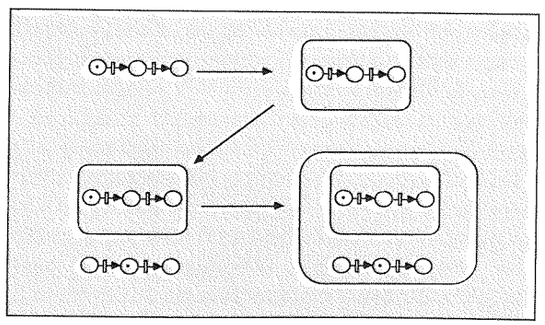

Figura 4 First steps of reality reflection

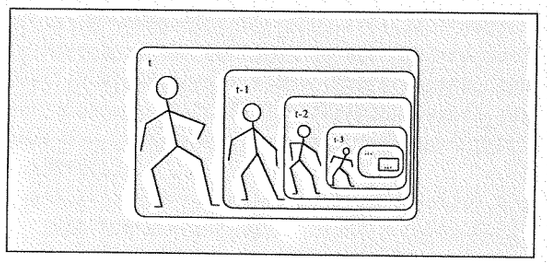

Figura 5 Self-observation at different times

A system able to observe itself can continuously compare its present state with the past states and draw new information from the comparison.
Such new information can be enough both to avoid loop situations, and to potentiate the problem solving capabilities of the system.

ACKNOWLEDGEMENTS

I would like to thank Prof. G. Degli Antoni for his fundamental contribution to this paper, Prof. G. Mauri for his valuable advice, and Dr. F. Ponti, who pointed out Crutchfield's work to me.

REFERENCES

[1] Degli Antoni G., ARTIFICIAL WORLDS, Proc. II Conference of Hypertext User Group, Milan 1988.

[2] Maes P., Nardi D., ed., METALEVEL ARCHITECTURES AND REFLECTION, North Holland 1988.

[3] Lucas J.R., MINDS, MACHINES AND GOEDEL, Philosophy 36, 1961.

[4] Hofstadter D.,Göedel, ESCHER, BACH: AN ETERNAL GOLDEN BRAID, The Harvester Press, 1979.

[5] Pizzi R., Degli Antoni G., AN INTRODUCTORY MODEL OF ARTIFICIAL REALITY: RELATIONSHIP WITH SPENCER-BROWN'S CALCULUS OF INDICATIONS, Proc. First Conference on Cyberspace, Austin 1990, in press.

[6] Scott D., DATA TYPES AS LATTICES, MS Lecture, Amsterdam, June 1972.

[7] Crutchfield J.P., SPACE-TIME DYNAMICS IN VIDEO-FEEDBACK, Physica 10 D, 229, 1984.

RECENSIONI

David A. Evans and Vimla L. Patel (eds.), COGNITIVE SCIENCE IN MEDICINE, The MIT Press, Cambridge (Mass.), + 421 pp XXI .

Il libro collettivo a cura di David A. Evans e Vimla L. Patel affronta in modo ricco e completo molte delle rilevanti tematiche che emergono dagli studi cognitivi nel campo della medicina. Il contenuto del libro rimanda alla scienza cognitiva come ad un campo di ricerche multidisciplinare che riguarda la costruzione di modelli del conoscere medico. In tale area vengono compiute molti studi dal punto di vista dei sistemi esperti e degli *intelligent tutoring systems* ma anche dal punto di vista delle ricerche psicologiche che tentano di utilizzare alcuni concetti provenienti dall'ambito informatico per considerare la loro validità nell'ambito sperimentale. Il libro è dedicato ad illustrare molte ricerche e alcuni risultati appartenenti soprattutto a quest'ultimo settore.
Se consideriamo la pratica medica essenzialmente come una *cognitive problem-solving activity* noi possiamo avere a disposizione molti dati di base osservabili e misurabili: rendiconti verbali, spiegazioni, interazioni discorsive nel quadro della pratica medica, procedimenti decisionali. I saggi contenuti nel volume illustrano come le scienze sperimentali abbiano prodotto parecchie analisi (1) della relazione fra il dominio di conoscenza e il comportamento effettivo nella risoluzione di problemi; (2) sulle fonti degli errori e dei pregiudizi nella rappresentazione dei problemi e nel livello decisionale; (3) sul ruolo del discorso nella strutturazione e nella acquisizione dei dati, nell'insegnamento e nello studio dell'attività di *problem-solving*; (4) sull'identificazione di metodi formali in grado di dar luogo ad una corretta valutazione delle varie *performances*.
Queste ricerche, suggeriscono i curatori del volume, costituiscono una prospettiva *meta-scientifica* per la scienza medica, cioè un insieme di *modelli* di analisi. Due sono le argomentazioni fondamentali che sono date a sostegno dell'utilità di tali ricerche: gli errori della pratica medica comportano aumenti di costi e devono perciò essere evitati; l'insegnamento della medicina è per più versi inefficiente. La convinzione fondamentale che sottostà a tali ricerche cognitive è che il *problem-solving* medico è sottodeterminato rispetto alla teoria medica; ciò fa della pratica della medicina clinica un esercizio squisitamente umano, *ill-structured*: inoltre le teorie che sono implicate sono *incomplete* e il dominio di conoscenza è *ontologicamente complesso ed eterogeneo*. Per tali motivi il *problem-solving* medico si mostra refrattario ad una conoscenza completa e soddisfacente.
Per quanto riguarda lo statuto epistemologico delle scienze biomediche e cliniche Patel, Evans e Guy J. Groen (Cap. 3, *Biomedical Knowledge and Clinical Reasoning*) aderiscono al punto di vista già dato dal filosofo della medicina Kenneth F. Schaffner ["Exemplar Reasoning about Biological Models and Diseases: A Relation between the Philosophy of Medicine and Philosophy of Science", *Journal of Medicine & Philosophy 11* (1986), pp. 63-80] secondo cui le scienze biomediche possono essere considerate come insiemi di *partially overlapping models* (spesso costruiti all'incrocio di discipline differenti) di fenomeni semi-indipendenti, collegati per analogia, e aventi a che fare con *prototypical cases*. Nelle scienze biomediche il ruolo delle generalizzazioni consiste nell'usare esplicitamente degli *esemplari*<[1] e nel cogliere le relazioni causali fra di essi, mentre nelle scienze fisiche rimanda alla costituzione di leggi astratte che collegano un'ampia varietà di esemplari. Inoltre le scienze "cliniche" differiscono dalle scienze biomediche di base per il fatto che gli esemplari nella medicina clinica riguardano *anormalità* di individui: "un'importante e forse implicita componente della teoria medica comporta modelli del comportamento biomedico normativo. Dal momento che questo, inoltre, può essere basato su insiemi di esemplari, si vede come sia possibile considerare che la medicina clinica, come teoria scientifica, è una teoria basata su modelli di modelli - e chiaramente non un prodotto diretto degli assiomi della biologia" (Cap. 3, p. 56).
Ricerche empiriche come quelle illustrate nel volume (condotte per lo più con il metodo delle analisi delle proposizioni) si pongono numerosi interrogativi: ammesso che gli esperti certamente hanno una conoscenza superiore a quella dei non esperti, tuttavia, a costituire la differenza è la quantità di conoscenza o la sua organizzazione? Qual'è il ruolo delle scienze di base

[1] Gli esemplari sono considerati da Kuhn come dei problemi accettati e prototipali che si incontrano sia nell'apprendere una disciplina sia nelle discussioni che appartengono alla ricerca effettiva e contemporanea - : in questo senso il metodo scientifico non ha uno statuto indipendente ma deriva dal contesto in cui una teoria è applicata.

nello sviluppo dell'*expertise medica*? Quale tipo specifico di conoscenza interviene nelle decisioni che riguardano la terapia? Come, nell'apprendimento, la nuova conoscenza interagisce con quella vecchia? Qual'è la differenza fra la *problem-solving performance* del medico teorico e del medico pratico? Ecco alcune risposte: l'expertise non deriva da una accresciuta abilità a ragionare a partire dai primi principi utilizzando modelli fisiopatologici derivati dalla scienza di base; nell'insegnamento le spiegazioni in termini fisiopatologici si mostrano efficaci soltanto quando sono presentate all'interno del contesto clinico e pertanto pare plausibile affermare che esse vanno "sempre" integrate in tale contesto (ciò comporta quindi un rovesciamento del *curriculum*) (capp. 1, *Issues of Cognitive Science in Medicine* [Evans], 3, *Biomedical Knowledge and Clinical Reasoning* [Patel, Evans e Guy Groen] e 5, *Application of Cognitive Theory to the Student-Teacher Dialogue* [Kenneth R. Hammond, Elizabeth Frederick, Nichole Robillard e Doreen Victor]); le spiegazioni in termini fisiopatologici non favoriscono la soluzione dei problemi ma casomai la loro spiegazione e la comunicazione fra i clinici nei casi di particolare incertezza, inoltre si constata che esse sono usate più dai principianti che non dagli esperti ma in modo molto meno selettivo; talvolta si può mostrare che i modelli dei processi fisiologici così come sono dati nei manuali sono scientificamente infondati e ciò non di meno nell'insegnamento e nella ricerca continuano ad essere utilizzati - nelle lezioni, nelle spiegazioni e anche nello sviluppo di nuove ipotesi e teorie - (Cap. 4, *The Nature of Conceptual Understanding in Biomedicine: The Deep Structure of Complex Ideas and the Development of Misconceptions* [Paul J. Feltovich, Rand J. Spiro e Richard L. Coulson]).

E ancora: gli esperti fanno diagnosi accurate anche quando falliscono nello spiegarle correttamente (il che suggerisce che le caratteristiche di una buona spiegazione non sempre sono correlate alle caratteristiche cognitive del *problem solving*); allorché si affronta il problema della valutazione delle competenze si constata per esempio che l'esperto organizza i dati in gruppi più significativi rispetto al principiante (la *performance* di quest'ultimo è quindi caratterizzata non da una conoscenza insufficiente ma dall' incapacità di organizzare la conoscenza stessa); l'esattezza della diagnosi è collegata con il suo carattere induttivo - la rottura di questo tipo di inferenza e la sua complicazione con una parte deduttiva di controllo caratterizza la diagnosi problematica e inesatta; nella medicina di base si assiste a molti momenti di negoziazione fra medico e paziente che possono essere analizzati dal punto di vista della particolare e complessa interazione "discorsiva" che è in atto (capp. 6, *Managing Coherence and Context in Medical Problem-Solving Discourse* [Evans e Cindy Gadd] e 7, *A Cognitive Framework for Doctor-Patient Interaction* [Patel, Evans e David Kaufman]).

La coerenza domina il contenuto; come osserva Evans: "una componente importante della conoscenza effettiva è il *comportamento* [*behaviour*] - dunque l'acquisizione dei concetti può essere separata dal comportamento -" (Cap. 1, p. 15). Conseguentemente è possibile affermare che la medicina è interpretativa: l'*expertise* è ottenuta attraverso la pratica piuttosto che attraverso la conoscenza teorica. Tuttavia conoscenza e pratica sono strettamente interrelate attraverso una rete intricata di scambi di conoscenza. Un'ultima importante questione, un po' trascurata nel volume, è quella epistemologica dell'ampliamento della conoscenza medica: tale questione è connessa con la realizzazione di un modello epistemologico adeguato del mutamento scientifico nonché con il problema fondamentale della scoperta scientifica e della costruzione di teorie. E' di grande interesse affrontare l'analisi della crescita della conoscenza medica (in IA o in altri campi della scienza biomedica e bioingegneristica): ciò comporta la modellizzazione di aspetti differenti delle funzioni e delle strutture biomediche. Per esempio, nel campo ristretto della medicina, come possiamo, al livello dell'invenzione, identificare i legami esistenti tra i concetti e le metafore utilizzati dai medici pratici e i formalismi utilizzati nel modello teorico? Quale tipo di paradigma epistemologico può sottostare a questi differenti ambiti?

Altre interessanti trattazioni sono quelle date nel capitolo 2, *Estrogen Replacement Decision of Third-Year Residents: Clinical Intuition and Decision Analysis* [Arthur Elstein, James Dod e Gerald B. Holzman], nel cap. 8, *GUIDON-MANAGE: Teaching the Process of Medical Diagnosis* [Naomi S. Rodolots e William J. Clancey] e 10, *Context-Specific Requirements for Models of Expertise* [Alan Lesgold]. Il capitolo 9, redatto dal filosofo della scienza Clark Glimour (*When Less is More*) è un interessante intervento intorno ad alcuni problemi generali riguardanti i fondamenti

empirici e razionali delle scienze cognitive.

Lorenzo Magnani

R. Pfeifer et al eds. CONNECTIONISM IN PERSPECTIVE North-Holland, 1989, pp. 517.

Il Connessionismo, paradigma nato negli anni '40, è stato uno dei due filoni dell'Intelligenza Artificiale fino alla fine degli anni '60, ed ha poi conosciuto un lungo periodo di stasi fino all'inizio degli anni '80. Da allora ha ripreso a diffondersi, favorendo il conseguimento di notevoli risultati di ricerca, tanto da essere ora considerato un paradigma alternativo all'Intelligenza Artificiale "classica" (la GOFAI, acronimo di Good Old Fashioned Artificial Intelligence). Nel loro saggio introduttivo, i curatori del libro prendono in considerazione la possibilità che lo sviluppo di questa disciplina porti ad una vera e propria rivoluzione scientifica secondo l'accezione di Kuhn. Sono infatti verificate alcune condizioni necessarie perché ciò possa avvenire. Innanzitutto le regole della ricerca "ordinaria" sono ormai diventate piuttosto vaghe: c'è infatti una gran varietà di modelli nel campo della GOFAI, come i sistemi di produzione, le reti semantiche, i box models; in secondo luogo le anomalie, cioè le difficoltà croniche nell'affrontare certi tipi di problemi, sono sempre più difficili da controllare: nell'ambito della GOFAI il problema che sembra oggi molto difficile da risolvere è l'apprendimento attraverso l'esperienza, per realizzare il quale sono state messe a punto tecniche molto complesse e tuttavia non sempre soddisfacenti. In terza istanza, il connessionismo è il nuovo paradigma che si sta imponendo con sempre maggior successo, e all'intemo del quale i problemi che per il corrente paradigma sembrano insormontabili, forse possono venir risolti con discreta semplicità: la nuova prospettiva permette infatti di realizzare in un modo semplice e naturale sistemi che apprendono. La quarta condizione verificata nella situazione attuale è la presenza di un'ampia intersezione tra il vecchio ed il nuovo paradigma, intersezione che non sarà mai una completa sovrapposizione. Secondo Kuhn per avere una rivoluzione scientifica è altresì necessario che i due paradigmi siano tra loro incommensurabili, si pongano cioè domande diverse, guardino alla realtà da punti di vista del tutto estranei. Ciò non vale per la GOFAI ed il connessionismo: anche dalla lettura di questo volume appare evidente che i due paradigmi sono complementari: dove l'uno è più debole l'altro è molto potente, permettendo la realizzazione di sistemi ibridi.

Il libro consiste di una raccolta di articoli presentati al convegno su "Connectionism in Perspective", svoltosi nel 1989 in Svizzera. I vari contributi offrono nel loro insieme una buona prospettiva sul Connessionismo, esaminato nelle potenzialità e nei limiti.

I sistemi connessionisti sono intrinsecamente paralleli, e la rappresentazione della conoscenza è distribuita, a differenza dei sistemi classici di Intelligenza Artificiale, in cui la computazione è sequenziale e la rappresentazione della conoscenza è locale (simbolica). Nella prima sezione del libro, dedicata appunto alla *rappresentazione della conoscenza* e alla *memoria,* viene messo in luce il fatto che il parallelismo permette di rispettare il limite dei cento passi di computazione, già posto, ben lungi tuttavia dall'essere rispettato, in ambito di Intelligenza Artificiale classica. Il modello connessionista si presta inoltre bene alla realizzazione di *memorie associative* e *content addressable.* Secondo alcuni autori però, molte caratteristiche del modo umano di rappresentare la conoscenza non sono ben descrivibili da un modello connessionista, che rimane comunque molto valido per l'implementazione di modelli simbolici.

L'apprendimento (seconda sezione del libro) è un aspetto essenziale dei sistemi connessionisti: le reti, per svolgere il compito a cui sono preposte, devono imparare a trattare il problema, e sarebbe impossibile costruire una rete in grado di operare correttamente definendone a priori le connessioni. Ciò non avviene nei sistemi di Intelligenza Artificiale, che devono essere corredati di una grande quantità di conoscenza iniziale, e di un gran numero di regole per poterla elaborare. Il problema che sorge per i sistemi connessionisti (è trattato in particolare nell'ultima sezione del libro, dedicata a *problem solving and reasoning)* è analogo al "collo di bottiglia di Feigenbaum", tipico dei sistemi di Intelligenza Artificiale classica. Ciò che è rappresentato dalla difficoltà che l'ingegnere della conoscenza ha nel definire le strutture dati e le regole, nei sistemi connessionisti è rappresentato dal tempo di apprendimento, giacché occorre effettuare centinaia di sessioni di trainig sugli insiemi di dati prima di conseguire i primi risultati.

Nella terza sezione del libro viene affrontato il tema della *percezione.*

Il modello connessionista sembra garantire ottimi risultati, soprattutto nella percezione di basso livello (sensoriale). Sono stati progettati con successo ad esempio sistemi di visione e sistemi di movimento basati su sensori. Tale successo è dovuto alla capacità dei sistemi connessionisti di dare buone prestazioni anche in caso di input confusi o con errori, ed alla loro capacità di generalizzare. Gli psicologi obbiettano al modello connessionista una duplice perplessità: in primo luogo, suggeriscono il fatto che i processi cognitivi umani di alto livello sono simbolici, mentre il connessionismo non opera sui simboli, ma su rappresentazioni distribuite; in secondo luogo, che le reti reali di neuroni contengono solo connessioni in avanti, mentre la più importante regola di aggiornamento dei pesi delle connessoni utilizzata per l'apprendimento nei sistemi connessionisti (la back propagation) è all'indietro.

Annamaria Zanaboni

H.Unbehauen, G.P.Rao, IDENTIFICATION OF CONTINOUS SYTEMS, North-Holland Systems and Control Series Vol.10, 1987, pp. 378.

La descrizione dei processi che avvengono in sistemi complessi (economici, ecologici, biologici, ecc.) in termini di modelli dinamici giuoca spesso un ruolo rilevante nell'analisi di tali sistemi: questo è evidente, per esempio, nel caso di processi che avvengono con vincoli temporali. Modelli dinamici possono essere inoltre utilizzati per simulare su elaboratori veloci il comportamento di un sistema sotto varie condizioni, realizzando cioè "esperimenti" simulati.
L'identificazione di un sistema dinamico, consiste proprio nella determinazione di un modello matematico appropriato alla sua descrizione, modello che può essere fenomenologico, empirico, o matematico. Una seconda classificazione di modelli dinamici può essere data in base alla loro trattazione del tempo: e cioè modelli a tempo continuo o tempo discreto; la distinzione è fondamentale, in quanto ogni approccio richiede strumenti matematici adeguati: modelli a tempo continuo richiederanno essenzialmente una descrizione analitica, con l'uso di derivate, integrali e operatori definiti in un dominio opportuno, mentre modelli a tempo discreto un trattamento di tipo algebrico. Si deve osservare che, mentre la letteratura sull'identificazione di modelli discreti si è sviluppata enormemente negli ultimi vent'anni, recentemente sono state proposte alcune tecniche promettenti anche per l'identificazione di modelli continui.
Il volume è dedicato completamente ad una rassegna dei più recenti approcci all'identificazione di modelli continui (CMI), raccogliendo per la prima volta in modo comprensivo tecniche diverse e trattando sia modelli parametrici che non parametrici. Le tecniche descritte vanno da semplici metodi grafici o numerici, fino all'uso di algoritmi ricorsivi utilizzabili per implementazioni "real time". Sono inclusi metodi recenti basati sull'uso di funzioni ortogonali.
Diamo rapidamente una descrizione del contenuto dei vari capitoli: il primo capitolo contiene una chiara e comprensiva descrizione della problematica della identificazione di modelli continui, mentre nel secondo si presentano modelli rilevanti, sia parametrici che non parametrici. Il terzo capitolo tratta segnali e procedure per il loro trattamento.
Dopo questi capitoli più introduttivi, nel quarto capitolo si tratta dell'identificazione di modelli non parametrici sia per il caso deterministico che stocastico, mentre nel quinto capitolo si discute diffusamente il problema della stima dei parametri in modelli continui. Nel sesto capitolo vengono presentati modelli adattivi; infine, nel settimo capitolo, si presentano alcune estensioni delle tecniche trattate nel quinto capitolo per trattare situazioni particolari come modelli con elementi non lineari, parametri variabili, vincoli temporali non noti a priori, sistemi MIMO, ecc.
Il volume è di grande interesse per ricercatori nel settore, e, anche se non pensato espressamente come supporto didattico, può essere utilizzato in un corso ingegneristico di Sistemi. Anche se non di facile lettura, dato che richiede un certo background matematico, è abbastanza autoconsistente, ricco di esempi, e dotato una vasta bibliografia divisa per capitoli.

N.Sabadini

J. M. Hughes, PROGRAMMING IN ZORTECH C++ WITH VERSION 2, Sigma Press, Wilmslow, 1990 - Distribuito da Wiley. pp. 317. Sterline 16.95.

"Oops", secondo il dizionario Webster, è un'interiezione che indica scusa o sorpresa, come sanno tutti i lettori

di fumetti. Tutti gli informatici invece sanno che *OOP* non è il singolare di oops, ma l'acronimo di *Object Oriented Programming*. Circa poi il sapere che cosa sia OOP, questo è un altro discorso, tant'è vero che il creatore di C++, Bjarne Stroustrup, cercò di precisarlo in un noto articolo intitolato "What is OOP?" che si può leggere nel volume 276 delle Lecture Notes in Computer Science.

L'incertezza sul modo di intendere diversi aspetti di OOP non ha impedito (anzi, probabilmente ha favorito) il diffondersi tra gli addetti ai lavori di un gergo, abbastanza sconcertante per i profani, fatto di classi virtuali e non, incapsulamenti, ereditarietà singola e multipla, polimorfismo, sovraccarichi... Ciò nonostante, l'OOP sembra effettivamente costituire una soluzione almeno provvisoria ai problemi connessi con la continua crescita della complessità dei programmi, e allora non resta che cercare di assimilarne il linguaggio, i concetti e la filosofia, e com'è noto non c'è nulla di meglio per capire che cercare di fare con le proprie mani e la propria testa.

Tra i diversi ambienti di programmazione orientata agli oggetti quello basato sul linguaggio C++ è uno dei più diffusi e anche quello per cui esistono il maggior numero di testi e manuali. Il volume che recensiamo va collocato nella categoria dei manuali, dal momento che si riferisce a uno specifico linguaggio e per di più al "dialetto" relativo a una data versione realizzata dalla Zortech Ltd. Questa versione di C++ è compilata direttamente, senza una fase intermedia di passaggio al linguaggio C, e gira su PC compatibili con disco rigido, floppy e un DOS almeno 2.11.

Le informazioni sopra citate compaiono nelle prime due pagine, ma immediatamente dopo c'è una frase che non ci si aspetterebbe di trovare su un manuale:"E' meno facile descrivere l'equipaggiamento mentale e l'esperienza di cui avete bisogno per programmare in C++". Di fatto, il volume si discosta spesso dall'esposizione tipica di un manuale e, pur contenendo una presentazione che sembra piuttosto completa e sistematica, riesce ad apparire più vario e interessante della maggior parte dei manuali.

In primo luogo non presuppone una conoscenza del C, ma ne dà una paziente e dettagliata esposizione, ricca di esempi e di osservazioni utili. Per esempio, a proposito delle dichiarazioni di variabili di tipo *array* richiama esplicitamente l'attenzione sul fatto che gli indici partono dal valore 0, e spiega le differenze da Pascal e Basic. Le istruzioni di I/O sono spiegate molto chiaramente all'inizio, e riprese poi in un capitolo finale sui file. L'introduzione dei puntatori è particolarmente graduale e chiara, e un intero capitolo è dedicato alla relazione tra puntatori e funzioni e all'allocazione dinamica. Nell'esporre le chiamate ricorsive viene usata come esempio la consueta funzione per i numeri di Fibonacci, evitando peraltro la versione più ovvia e inefficiente, e riprendendo più volte l'esempio per illustrare le ulteriori possibilità di C++.

Gli esempi sono molto numerosi: naturalmente sono in genere piuttosto brevi, sia per semplicità che per ragioni di spazio; l'esempio più ampio si trova nel capitolo che illustra la derivazione di nuovi tipi di dati da tipi preesistenti, dove è presentato un programma che esemplifica la gestione di una biblioteca, con periodici da consultare e libri e compact-disc per il prestito.

Ogni capitolo termina con un breve riassunto e quattro o cinque semplici esercizi, di cui il primo è sempre un invito a eseguire i programmi presentati nel testo: purtroppo non mi è stato fornito il compilatore insieme con il volume da recensire, ma si suppone che il lettore abbia accesso a una macchina dotata di C++, e usi gli esempi anche per sperimentare praticamente il linguaggio. Dagli accenni nel testo sembra che il compilatore sia alquanto più amichevole di un normale compilatore C, fino a segnalare per esempio l'uso di "=" in luogo di "==" in un confronto.

Il volume è dotato di un indice degli operatori e di un indice analitico, che tuttavia non è molto accurato in quanto non riporta tutte le occorrenze di un termine: per esempio, l'istruzione *return* non vi compare, pur essendo spiegata nel testo a pagina 105. In compenso, per una volta posso segnalare di non aver trovato un solo errore di stampa!

Mi sembra di poter concludere che l'autore è riuscito quanto meno a evitare di usare troppo disinvoltamente

il gergo di OOP, e a ridurre le difficoltà, direi quasi a sdrammatizzarle, col tono amichevole di chi dà buoni consigli e dimostra comprensione per le perplessità di chi legge: una dote purtroppo nient'affatto frequente.

M. Torelli

F. Filippazzi e G. Occhini, IL COMPUTER, Il Sole 24 Ore Libri, 1990, pp. 477

E' certamente da consigliare a tutti questo nuovo libro di Franco Filippazzi e Giulio Occhini. Lo potranno trovare interessante gli specialisti, che spesso trascurano le novità e gli sviluppi più promettenti dell'informatica, ma soprattutto le persone non esperte che desiderano conoscere a fondo i concetti che stanno alla base della scienza informatica, la sua dimensione applicativa e le prospettive di sviluppo. Il testo, che propone un linguaggio costantemente semplice e chiaro, é diviso in tre parti: nella prima, gli autori spiegano con dovizia di dettagli sia l'architettura fisica che la struttura logica degli elaboratori elettronici. Le diverse tipologie delle applicazioni dell'informatica, con un'analisi della loro dimensione economica e sociale, vengono prese in esame nella seconda parte. Nell'ultima parte, infine, gli autori delineano il futuro del computer, evidenziandone l'attuale mutamento di ruolo: da macchina per eseguire calcoli a strumento attivo nelle diverse forme della comunicazione umana.

Roberta Macchi

ERRATA CORRIGE

Interacting with artificial realities

Pag. 18, quarta riga: invece di: (LNGL67) si deve leggere: [14]
Pag. 21, terza riga: invece di [3]. si deve leggere: [4]
Pag. 28, penultima riga: invece di: Kruegerg, si deve leggere: Krueger

A Model for Artificial Reality

Page 29: Riassunto, fourth line: read ''simulazioni'' instead of ''simulatzioni";
Page 29: Abstract, second column, eigtll line: read ''they can be divided into hard and soft agents ' instead of "they can divided in hard and soft agents":
Page 29: Abstract, second column, eleventh line read ''themselves'' instead ot` ``themeselves';
Page 30: third paragraph, first line read ''Computers permit software simulation" instead of ''computers make possible...";
Page 32: fifteenth line, first column read ''instantaneously" instead of "instantenously";
Page 32: thirteenth line, second column read "the second order" instead of "the first order'';
Page 34: first column, sixth line read "chosen" instead of "choosed'';
Page 35: read "**P5. Principle of continuity or spatial positioning**'' instead of "Priciple....':
Page 36: read ''**7. ACKNOWLEDGEMENTS**" instead of ''Aknowledgements".

A Project on Sounds in Artificial Realities

Page 177: Introduction, first line read [3] instead of (GnA 89);
Page 178: first line read "makes it possible.'' instead of "makes possible...'';
Page 178: A Simple Sound Propagation Model, second paragraph, fourth line read [6];
Page 179: ninth line, read [7] instead of [8];
Page 180: end of second paragraph read [41 instead of (Gardin 90):
Page 180: end of third paragraph delete three points;
Page 181: References [6], [7] and [8] change to [5], [6] and [7].

EDITO DA CALEIDOGRAF S.R.L.
VIA L. DA VINCI N.4/6 TEL. 039 - 587541
22058 OSNAGO (CO)

Spedizione in abbonamento postale, Gruppo IV, 1° semestre.
Pubblicità Inferiore al 70%
N. 51, Marzo 1991

TAXE PERCUE
TASSA RISCOSSA
P.P. P.T. Piacenza F.