Assessing the software crisis

Veldwijk, R.J.; Boogaard, M.; Spoor, E.

1992

document version
Publisher's PDF, also known as Version of record

Faculteit der Economische Wetenschappen en Econometrie

Serie Research Memoranda

Assessing the Software Crisis:
Why Information Systems Are Beyond Control

R.J. Veldwijk
M. Boogaard
E.R.K. Spoor

Research Memorandum 1992-6
maart 1992

vrije Universiteit *amsterdam*

Assessing the Software Crisis:

Why Information Systems Are Beyond Control

R.J. Veldwijk
M. Boogaard
E.R.K. Spoor

Vrije Universiteit

FEWEC/BIK

De Boelelaan 1105

1081 HV Amsterdam

The Netherlands

Key Words: Software Crisis, Software Maintenance, Flexibility, Productivity, Data Dictionaries.

Abstract
The purpose of this paper is to introduce a conceptual framework that explains why progress in the information systems field does not improve the maintainability of application software. Based on the premise that this software crisis is a flexibility crisis, rather then a productivity crisis, the causes of information systems rigidity are examined. Subsequently, arguments are supplied for the position that the directions in current information systems research offer no prospect for solving today's maintenance problems. Finally, two approaches for solving the software crisis, suggested by the conceptual framework that explains the crisis' persistence, are examined and found to be useful to some extent.

1. Introduction

Almost 25 years ago the term 'software crisis' was first used for the phenomenon that a large part of EDP-budgets is spent on maintaining existing business information systems (ISs), at the expense of fulfilling the demand for new ones covering new application areas. During this quarter century much has changed for the better, but the software crisis is still here. Recent estimations by Hager (1991) and by Sharpe et al (1991) indicate that 50 to 70 percent of EDP-budgets is spent on maintenance. Martin and McClure (1983) claim that 80 percent of this maintenance expense is normally associated with

changing requirements, that the application backlog is typically 3 to 4 years, that schedule and cost overrun for software projects is often as high as 200 percent and 300 percent respectively and that the total yearly cost of maintaining information systems exceeds $30 billion worldwide (1983 figure). It is very hard to find activities outside the EDP domain in which management can produce the kind of cost overruns described and still remain in charge.

Of course there is a fine line between maintaining existing ISs and creating new ones. For instance, the conversion of a batch-oriented IS to an on-line equivalent will be regarded as (adaptive) maintenance, while rebuilding a modernized version of the IS with some new functionality added will probably not be reported as such. For another thing, it may be the case that maintenance, backlog and overrun figures are biased by devious practice of IS users and/or IS professionals. The authors know of organizations in which IS users record requests for new functionality under the heading of maintenance and of organizations in which the need for new applications is deliberately exaggerated and development costs are deliberately underestimated. Because maintenance is often assigned a generally higher priority and EDP-budgets are limited this type of behavior often pays.

Although the figures may be overestimated, the general feeling of IS users and professionals alike is that the maintenance percentage is indeed far too high and quite immune to the technological changes that have occurred over the last two decades. Moreover, if the figures above were biased considerably, the conclusion would still be that EDP-budgets are very hard to manage.

Although there seems to be agreement on the existence of the software crisis phenomenon, IS researchers do not seem to regard it as an interesting subject for research. Considering that IS maintenance is an economically very significant activity this is hard to explain. Of course the software crisis is frequently mentioned in scientific literature, but generally indirectly. Usually, research interest focuses on the beneficial effects on the software crisis of proposed IS development methodologies, data models, programming concepts and the like. Moreover, claimed beneficial effects are often credited to either increased labor productivity of IS professionals or to increased control over the *process* of IS development and maintenance.

The authors take the position that the software crisis is basically a *flexibility crisis* rather than a management crisis or a productivity crisis. The present paper intends show that this hypothesis is a useful one in the sense that (1) it provides an explanation for the persistent software crisis phenomenon and that (2) it makes it possible to assess the effects of the outcomes of IS research and development on the software crisis.

As we shall see, the flexibility approach of the software crisis leads to a pessimistic view on its future, because the software architecture underlying ISs is inherently *in*flexible. To prove our point, it is necessary to examine the nature of flexibility and compare information systems with systems that display higher levels of flexibility, like human beings and organizations. We shall find that

even if it is as yet impossible to achieve the kind of flexibility displayed by humans or organizations, it is still possible to compare what makes these systems flexible to what makes our information systems inflexible and to guide our research and IS implementations by the outcomes of this comparison.

The paper is organized as follows. Section 2 takes a fundamental view of ISs by considering them together with human beings and organizations as *representation* systems, i.e., systems organized in such a way to reflect the state and development of some outside world. The objective of the discussion is to obtain an understanding of the nature of flexibility and to create a basis for comparing human systems with information systems. Section 3 discusses the inflexible nature of software architectures in relation to the flexibility concept. Sections 4 and 5 focus on the question whether the software crisis can be successfully resolved by the traditional approaches of IS research. In section 5 the idea that increases in labor productivity can solve the software crisis is refuted. Section 6 does the same for the idea that IS maintenance overruns can be eliminated by documentation and CASE strategies. Sections 6 and 7 discuss two approaches directed at increasing IS flexibility that may offer a partial solution to the software crisis. Section 6 discusses the idea of enhancing flexibility by introducing multiple abstraction levels in programming languages. Section 7 discusses a different approach directed at increasing the level of self-knowledge of ISs. Finally, section 8 contains a number of concluding remarks about our position presented in this paper.

2. Information Systems, Representation Systems and Flexibility

We ourselves are the most impressive examples of flexible information processing systems. Therefore, if flexibility is the starting point for assessing the software crisis, the sophisticated and flexible mental models of human beings provide an interesting field of study for IS researchers. Unfortunately, the human mind seems to be the monopoly of artificial intelligence (AI) researchers, who are usually more interested in emulating the process of reasoning than at finding the roots of human flexibility (see, e.g., Barr and Feigenbaum 1981). Another problem lies in the fact that the human mind is so much more flexible than our present ISs that it quickly becomes too abstract a metaphor for ISs, rather than something IS research can learn from.

In order to focus on useful similarities between human beings and ISs we need a point of view that applies to humans and ISs alike. With respect to flexibility the concept of *'representation system'* provides such a unifying view. Hofstadter and Dennett (1981, p.192) define a representation system as "*an active, self-updating collection of structures, organized to 'mirror' the world as it evolves*". These authors apply the concept to the human mind, but it can also be applied to many human artifacts (like clocks and ISs) and to systems of which humans are a part (like organizations). Representation systems are models that mirror a part of their environment. They fail in whatever purposes they serve or goals they seek if the extent to

which their model represents reality falls below a certain level.

The representation system that is the human mind has the ability to collect information within an established framework (like: "John is an airplane pilot") as well as altering the framework itself (like: "for every airplane pilot the number of flight hours must be known"). Of course we can distinguish the two pieces of information as changes in the representation system's *contents* and in its *structure* respectively, but from a representation system's point of view this division is irrelevant. Both pieces of information are just changes to the representation system needed to retain its correspondence with reality. The representation system concept thus directs us to a fundamental question: if a representation system must reflect some part of a changing reality over time, what then constitutes the difference between *updating* the information content of a representation system and *changing* its structure. If we look at the mental models in our minds, we fail to find a sharp distinction or even a distinction at all.

If we turn our attention to ISs we find that these representation systems fail to offer this kind of flexibility. If the two pieces of information about "John being a pilot" and "the number of flight hours being relevant for the class of pilots" are submitted to an IS, the former normally amounts to a simple update, while the latter requires a change to a database schema and a number of application programs.

Hofstadter and Dennett ascribe the flexibility of the human representation system mainly to the fact that this system not only represents the outside world, but also itself and its relationships with the outside world. Moreover, they stress the importance of multiple abstraction levels that can be mapped in some way to each other. They argue that flexibility is basically derived from self-knowledge on a high level of abstraction (Hofstadter and Dennett, 1981, p.200). Self-knowledge is essential if a representation system must be able to change itself. Multiple levels of abstraction are necessary because rigidity cannot be wholly avoided. In order to be as flexible as possible, those parts of the representation system that cannot be changed by the system should at least be generalized as much as possible.

The important question is whether this abstract view of representation systems can be meaningfully applied to ISs, given the limitations of our knowledge about human representation systems and the enormous difference in sophistication between human and human-made representation systems. In the following sections we shall demonstrate that this is to some extent possible.

3. Information Systems Flexibility and Software Architecture

The discussion of the previous section suggests that ISs have some property that explains their limited flexibility. The observation in the introduction that the software crisis is immune to increases in productivity suggests that this property is part of an architecture underlying all IS software. This suggestion is supported by the fact that the computer was originally believed to be a very flexible

apparatus. The root of this flexibility was supposed to be the stored program principle stipulating that there is no fundamental distinction between program and data (see Wilkes 1991). Data can be treated as program and vice versa. The utilization of this property was considered to be very promising. To quote Alan Turing (1950): "*... a machine can undoubtedly be its own subject matter. It may be used to help in making up its own programs, or to predict the effect of alterations in its own structure. By observing the outcome of its own behaviour it can modify its own programs so as to achieve some purpose more effectively. These are possibilities of the near future, rather than Utopian dreams.*".

This quotation contains the key elements of flexibility ascribed by Hofstadter and Dennett to the human representation system. If Turing is right a comparison between the human mind and a computerized IS is meaningful. However, we must find out first why Turing's ideas have not been realized during the past four decades.

The question why Turing's prediction in 1950 has not come true can of course be answered by adopting the weak AI view that the human mind (including its flexibility) cannot be emulated by Turing machines. However, an alternative answer to this question would be that IS research and development have not taken Turing's direction. Instead the approach has been to segregate everything that can be distinguished, starting with programs and data. Programs and data structures are presupposed to be stable. The distinguishing aspect of all representation systems, i.e. mirroring change, has been confined to the domain of instance data. Only this part of ISs is presupposed to be subject to change within the fixed framework of programs and data structures (see figure 1). Whenever this assumption does not hold true, human interference is unavoidable. Flexibility would require program maintenance requires to be executed automatically. Since this requires programs to be treated as data, the idea of segregation would be violated. Because changes occur constantly in these 'stable' parts of ISs this architecture can be called '*inherently inflexible*'.

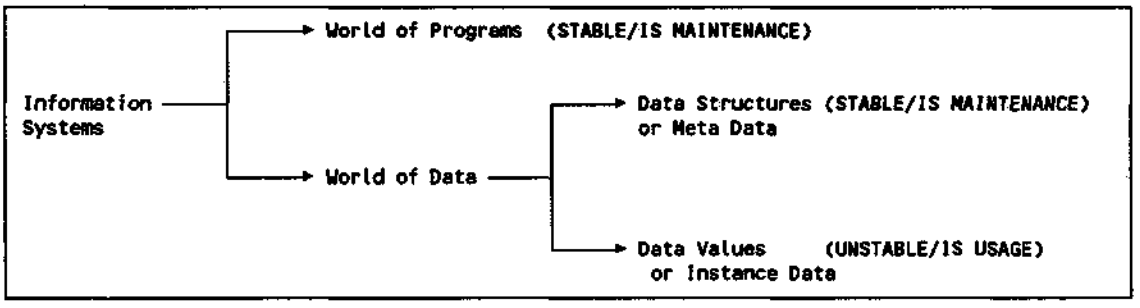

Figure 1: Information Systems Software Architecture

Obviously there is a very good reason to create this dichotomy between fixed and flexible parts of ISs: it is very hard to imagine an alternative software architecture that conforms to Turing's ideas on the one hand and is manageable by normally talented people on the other. This latter condition must be satisfied as long as changes can

occur in the 'stable' parts of the IS[1].

If the preceding analysis holds true, the IS community is found to be in a very contradictory situation: the inherently inflexible software architecture leads to ISs that are relatively easy to comprehend, but are very vulnerable to maintenance. It remains possible to view ISs as representation systems, but only if we include a human component, the maintenance programmer, as part of the system. Thus, the strive for control over IS development and maintenance leads to a loss of control, ultimately leading to a negation of the goal of automation: replacing human labor by computer labor.

At this point we have developed a framework with which we can make predictions about the future of the software crisis and on the merits of IS research and development on the software crisis. We shall do both in the following two sections.

4. The Software Crisis and Increased Maintenance Productivity

As we have noted in the introduction, optimism about solving the software crisis is often based on the fact that developments in IS research lead to a continuous increases in labor productivity. Although productivity has gone up considerably since the days of assembly languages and flat files, and will probably increase further, the software architecture introduced in the preceding section has never been abandoned. What has occurred is the replacement of the 'stable' parts of ISs (programs and data structures) by higher level equivalents, but within the framework of figure 1. Machine languages have been replaced by procedural languages and these by non-procedural languages. Flat file data structures have accordingly been replaced by file management systems and relational table structures.

Even if the productivity gains have not resulted in diminished maintenance figures, it is still possible to remain optimistic in the long run because the cost of creating and maintaining ISs may be reduced to insignificance. Specifically, if IS maintenance productivity were to equal the productivity of IS usage, the previous section's discussion of the distinction between IS maintenance and usage would become of philosophical interest only.

Considering the persistence of the software crisis, one could respond to this long term optimism by adopting Keynes' motto that "in the long run, we shall all be dead". However, we feel that even in the very long run, increases in productivity will not solve the maintenance problem. There are at least three reasons for this pessimism.

Most important, IS maintenance always requires insight in the structure and the workings of the IS as a whole. Errors are always possible if people have less than perfect knowledge about the IS's structure, or are less than perfect in executing the changes to the IS. Whatever its productivity, IS maintenance will remain a tedious

---

[1] If such changes leading to maintenance would never occur the ideal of strong AI would be realized.

process of analyzing the IS's current structure, executing the changes needed, and testing and debugging the altered IS.

Another effect that explains the continuing importance of maintenance is that increased productivity generally leads to larger or more tightly integrated ISs, as has been noted by Nolan (1979) and has been affirmed by the recent interest in Electronic Data Interchange. Such ISs are more difficult to comprehend and are thus harder to maintain than ISs that are a product of 'isolated automation'. Since the demand for further integration of ISs will persist indefinitely, productivity increases will be compensated by increased IS complexity.

As third and final effect that counters the beneficial effects of productivity increases is that these lead to an enlargement of the area's in which ISs are applied. Examples are executive ISs and decision support ISs. Because these new areas are commonly characterized by their relative instability, new maintenance effort will remain the same of may even increase.

The key problem behind our claim that productivity increases within the software architecture of figure 1 do not alleviate the software crisis is that programs and data structures are by their nature opaque to the IS itself. Changes in these black boxes requires the outside intervention of human representation systems. Since these cannot have a perfect representation of the IS's structure and are not perfect as agents of change, IS maintenance remains problematic.

5. The Software Crisis and Increased Control over Information Systems

Apart from increased labor productivity, optimism about resolving the software crisis is often based on perceived progress in managing the process of IS development and maintenance. The idea is that IS development and maintenance can be managed by methodical work methods resulting in ISs that are thoroughly standardized and documented and thus maintainable. Even if maintaining ISs remains time and money consuming, much would be gained if the process would be under human control. The project overrun figures in the introduction clearly show that this is not the case as yet.

Once again the representation systems view suggests that this optimism is not warranted. The idea behind the documenting approach is that a representation system can be managed by storing its description in another representation system that reflects the IS structure at a higher level of abstraction. If this description is much more easily accessible to maintenance programmers or IS management, it becomes possible to avoid unpleasant surprises in the process of IS development and maintenance.

According to the representation systems view the standardization and documentation approach can only offer a partial solution to managing ISs. This is because the representation system called 'documentation' must continuously match the IS it represents or else become useless. If updating the documentation is a manual process, so is certifying the correspondence between the IS and its high level representation, and maintaining the adherence to standards.

However, there is a difference between maintaining the correspondence between the IS's instance data and the outside world on the one hand and the IS's structure and its documentation on the other. The distinction is that the maintenance programmer always has something more reliable (if less accessible) than IS documentation at hand, namely the IS's source program listings. The maintenance programmer has to consult these program sources anyway, not only to execute the desired changes to the IS, but also to inquire about low level details not included in the IS's documentation. As a consequence, high level documentation is often of little value to the maintenance programmer, especially if he or she has detailed knowledge about the structure of the IS. All this results in an strong incentive to update the documentation in a sloppy manner or not at all. Of course, documentation that is not totally reliable is not likely to be used in subsequent IS maintenance activities. Thus, a vicious circle develops, leading to less and less useful documentation. Finally, because the cost of re-documenting an IS manually is high and the beneficial effects of documentation are hard to grasp for the IS's users, investments in re-documenting ISs are rare. Like politics, IS maintenance is often an art of 'muddling through'.

The aim of getting better control over IS development and maintenance has been revived by the advent of Computer Aided Software Engineering (CASE) technology, specifically by forward engineering, reverse-engineering and re-engineering software tools. Partly, these developments result in increased labor productivity, the effects of which were discussed in the previous section. But for another part, CASE technology aims at enhancing management control over IS development and maintenance activities.

Again the representation systems view does not support the more ambitious claims of the effects of CASE technology on the software crisis. Reverse engineering and forward engineering only cover a part of the IS development and maintenance process. If CASE technology were to cover the entire process, CASE would just be the next step in the quest to develop ever more high level software languages. It seems safe to conclude that if third generation and fourth generation languages were unable to solve the software crisis, so will an approach that is at best equivalent to an incomplete fifth generation language. The same line of reasoning can be applied to less ambitious approaches that limit themselves to checking the correspondence between IS structure and IS documentation.

To summarize: the key problem behind our claim that approaches aimed at improving the manageability of IS maintenance within the architecture of figure 1 cannot alleviate the software crisis is that these approaches offer nothing but a higher level description of the same inflexible ISs. Maintaining a higher level IS description instead of a lower level description enhances productivity, but it offers no more flexibility. The documentation approach is no different from the CASE approach, but has the extra disadvantage that manually generated IS documentation is inherently unstable.

6. Flexibility and Multiple Levels of Abstraction

The representation systems approach we have adopted in section 2 has provided a framework with which to assess the software crisis phenomenon. Applying this framework to historical IS research and practice has lead to a pessimistic long term view of IS maintainability. Although we believe this view to be generally correct, it should still be possible to apply the representation systems view of ISs in a positive manner. We shall do so in this section and the following.

As has been noted in section 2, flexibility requires multiple levels of abstraction that are in some way formally mapped to one another. Rigidity should only be allowed at the highest level and the specification of lower levels should be drawn from higher level specifications.

Whatever criticism one may have with respect to historical trends in the IS domain, it cannot be denied that the level of abstraction of programming tools has gone up. In the previous section we have been rather negative about the impact of this evolution on IS flexibility, but this position must be somewhat modified. If a higher level programming language does nothing more than obscure lower level details, we adhere to our position of the previous section that such a language may improve labor productivity, but not IS flexibility. Examples are procedural languages or most program generators. If, on the other hand, a higher level language obscures low level details without eliminating freedom in specifying these details, flexibility is indeed enhanced. Examples of this approach exist in the form of the physical data independence (PDI) feature of relational DBMSs and the information hiding feature advocated by the object-oriented approach.

In order to appreciate the potential value of the approach we shall take a closer look at the PDI feature of RDBMS products. These products only allow database access through the SQL language, which is the highest level interface available to the database. SQL lacks a lot of 'technical' information about file structures, access paths, user authorization, etcetera. Yet, sophisticated RDBMS products allow these low level details to be specified, by means of Data Definition Language (DDL) or Data Control Language (DCL) statements that are part of the SQL language. If at any time the low level specifications need to be altered, programs do not have to be changed. IS maintenance has been changed into IS update.

It is interesting to note that using the PDI feature of RDBMS products is generally considered by IS professionals to be IS maintenance (or IS *tuning* at best). This can be largely explained by the observation in section 3 that IS field has a strong propensity to segregate everything that can be conceptually distinguished. In the case of PDI DDL or DCL statements are simply updates operations of a special relational database, the catalog, disguised as programs. Indeed, DDL and DCL are completely unnecessary (see Buitendijk and Lek, 1991).

As it is, the level of flexibility of RDBMS products is only partially utilized, because updating the catalog database with DDL or DCL the IS professional's job. In fact, if either the instance database or the catalog database were to contain information about

application program priorities and IS performance, the task of IS professionals to specify database implementation details like indexes and file structures would have become obsolete. The fact that it has not is evidence that even simple forms of flexibility are not high on the agenda.

To recapitulate: flexibility can be enhanced by higher level languages, provided the details that are eliminated at the highest level can still be designated on a lower level by updating a meta database or catalog. The potential of flexibility is truly realized in the spirit of Hofstadter and Dennett's idea of representation systems if these updates are executed on the basis of higher level information that makes sense to the users of the IS.

Applying the idea of multiple abstraction levels to ISs depends heavily on fundamental research and its application in programming tools. With respect to relational theory and its implementation, flexibility is limited to technical aspects, despite claims that the relational model also offers flexibility at an even higher abstraction level. Although these claims are doubtful (Graaff, 1992), the preceding discussion advocates the idea of directing IS research towards developing data models on top of other data models.

7. Flexibility and Self-Reference

Next to multiple abstraction levels, the other important feature of advanced representation systems was considered to be self-reference. Self-reference was described in section 2 to be a systems representation of its own structure in its own model. Applied to ISs within the software architecture of section 3, this idea amounts to transferring the IS's specifications from its programs and data structures to the contents of its database.

Up to now, the decision about what kind of changes can be brought about by IS usage and what kind of changes require IS maintenance has been largely based on convention. The transfer of IS specifications to the instance data accessible by its programs has been limited to trivial cases. An example of such a trivial change can be found in the way in which for instance tax percentages are treated. In the past, it was not uncustomary to hard-code such a percentage in the application programs. Nowadays, the tax percentage would be derived from a table in the database and a program would be supplied to update this percentage if needed.

It is quite possible to pursue this strategy much further. In the case of the tax-percentage it could also occur that its database and screen formats need to be changed due to a change in its value from 30 percent to 33.33 percent. If the principle that was applied to data values is also applied to data formats and the like, we are really creating an IS with a great deal of information about its own structure. Because this information can be updated a part of IS maintenance becomes IS usage, thereby enhancing labor productivity through increased flexibility and reducing not only the amount of IS maintenance but also its character. Another beneficial spin off is that programs become much more generalized and thus reusable.

If this strategy is pursued consistently, we find ourselves creating and maintaining ISs that are structured around a more or less fixed framework of highly reusable software. In popular terms: ISs should consist of user-written software tightly integrated with a stable 'tool' with which to maintain the ISs self description. Integration is achieved by defining the database containing information about the IS's environment with the 'tool' database containing information about the IS itself. An example of an implemented IS that displays this self-referential architecture can be found in Boogaard, Veldwijk and Spoor (1992).

Of course the self-referential strategy takes us back to the subject of case tools. However the self-referential approach differs from the traditional CASE-approach in the sense that all application programs, whether provided by a tool vendor or by the application developers, operate on one integrated database. This requires that what can be called the the CASE database is completely transparent to the IS developers and that these write their application programs in order to prevent that information about the IS is redundantly coded in their application programs.

The approach described in this section has two distinct disadvantages. Conceptually least important is that blurring the distinction between package software and user-written software leads to a serious marketing and legal problems. Of more importance is the fact that highly generalized programs are conceptually very hard to comprehend whenever IS maintenance becomes necessary. Thus, although we have remained within the software architecture described in section 3, we can apply it in such a way that we are confronted with the same problems of comprehension that led us to introduce the architecture in the first place. The conclusion must be that introducing self-referential aspects in ISs by storing information about the IS into its database can provide a partial solution to IS maintenance problems. Like the approach described in the previous section, it promises the abolition of certain kinds of IS maintenance, thereby enhancing labor productivity and IS manageability through increased IS flexibility.

8. Concluding Remarks

In this paper we have attempted to develop a point of view that sheds light on the persistence of the software crisis in the midst of what seems to be rapid technical progress. Although our argument has been that traditional approaches directed at enhancing labor productivity or IS manageability cannot provide a solution, this should not be interpreted in the sense that these enhancements are worthless. Indeed, if we would still be building and maintaining the simple, isolated and batch oriented systems of the sixties, the problem of organizations applying information technology would not be IS maintenance. The software crisis persists because ISs remain as complex and rigid as before to the people who have to maintain them.

Although a fundamental solution to the flexibility problem that causes the software crisis is not in sight, there exist at least the

two partial solutions we have described in the previous two sections. What is more, IS developing organizations are neither totally dependent on the outcome of further research, nor on implementations of this research by DBMS and CASE tool vendors. To the extent that rigidity in ISs is caused by rigid design approaches there exists a cure for today's maintenance problems.

References

Boogaard, M., Veldwijk, R.J., and Spoor, E.R.K., "An Alternative Approach to Generalization in the Relational Model." To appear in the proceedings of *IFIP WG 8.1 Working Conference*, Alexandria, Egypt, April 1992.

Barr, A., and Feigenbaum, E.A., *The Handbook of Artificial Intelligence, Volume I, II, III*, Pitman Books, London, 1981.

Buitendijk, R.B., and Lek, H. van der, "Direct Manipulation of a Data Dictionary with SQL.", *Information Systems*, Vol. 16, No. 3, 1991, pp. 323-333.

Graaff, J.M. de, Veldwijk, R.J., and Boogaard, M., "Why Views Do Not Provide Logical data Independence.", *Research Memorandum 1992-4 Vrije Universiteit, Amsterdam*, 1991

Hager, J.A., "Software Cost Reduction Methods in Practice: A Post-Mortem Analysis.", *Journal of Systems Software*, 1991, pp. 67-77

Hofstadter, D.R., and Dennett, D.C., *The Mind's I: Fantasies and Reflections of Self and Soul*, Penguin Books, London, 1981.

Martin, J., and McClure, C., *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, Englewood Cliffs, 1983.

Nolan R.L., "Managing the Crises in Data Processing.", *Harvard Business Review*, March-April 1979, pp. 115-125.

Sharpe, S., Haworth, D.A., and Hale, D., "Characteristics of Empirical Software Maintenance Studies: 1980-1989.", *Software Maintenance: Research and Practice*, Vol. 3, 1991, pp. 1-15.

Turing, A.M., "Computing Machinery and Intelligence.", *Mind*, Vol. LIX, No. 236, 1950.

Wilkes, M.V., "Babbage's Expectations for his Engines.", *Annals of the History of Computing*, Vol. 13, No. 2, 1991, pp. 141-145.

The MESDAG Research Group

INTRODUCTION

The MESDAG project is a joint project endorsed by three organizations in the Netherlands: the N.V. Nederlandse Spoorwegen (The Netherlands Railways Company), RAET N.V. and the Vrije Universiteit of Amsterdam. The MESDAG project originated at RAET N.V. during the second half of 1989 as an outgrowth of research done in the field of active data dictionary models. This research and a prototype of an active data dictionary form the basis for the mission of the MESDAG project that officially started its activities in September 1990.

MESDAG is an abbreviation of:

MEta **S**ystems **D**esign **A**nd **G**eneration

MISSION AND OBJECTIVES

The mission of the MESDAG project is to prove the feasibility of developing inherently flexible information systems by introducing higher levels of logical data independence.

Derived from this mission following are the two main objectives:

1. Examine the feasibility and initiate the development of an active, self-referential data dictionary model in which both a description of the database data and a description of all specifiable application design data can be stored. This data dictionary model should contain sufficient semantic aspects (like domains, constraints and time aspects) to assure the integrity, consistency and validity of the stored (meta) data, to avoid maintenance and to support query-formulation independent of current database structure.

2. Examine the feasibility and initiate the development of the possibilities of data dictionaries in general and the described data dictionary in specific. This analysis of possibilities is directed at the embedding in and developing methods, techniques, methodologic guidelines and automated tools for the design, implementation and maintenance of flexible information systems.

MEMBERS OF THE MESDAG RESEARCH GROUP

1. Dr. E.R.K. Spoor

Dr. E.R.K. Spoor is associate professor at the Vrije Universiteit Amsterdam. He teaches and consults in the area of database systems and database development with a focus on the use of these technologies in organizations. His eighteen years of experience with computer technology includes eight years with NCR and six years with the Vrije Universiteit, first as a systems engineer and later as a computer scientist. He is one of the founders and board members of two automation oriented organizations: PSB (Amsterdam) and VDA (Hilversum).

2. Drs. R.J. Veldwijk

Drs. R.J. Veldwijk graduated from the Vrije Universiteit Amsterdam in 1986. In his quality as consultant at RAET N.V. Utrecht, he is among others responsible for the design and implementation of advanced database architectures. His main interest lies in developing and implementing self-knowledgeable database models, aimed at reducing maintenance costs and at improving the accessibility of databases by end-users. Furthermore he teaches courses in data modelling.

3. Drs. M. Boogaard

Drs. M. Boogaard is assistant researcher at the Vrije Universiteit Amsterdam. Furthermore, he is part-time involved in projects by the Netherlands Railways Company. He graduated from the Vrije Universiteit Amsterdam, in August 1990. The objective of his research is to develop an approach to achieve higher levels of logical data independence for both end-users and application programs and to analyze the consequences of the level of logical data independence accomplished on the system development life cycle in general and on software maintenance and database inquiry in particular.

ACCOMMODATION ADDRESS

Vrije Universiteit
Faculteit Economie & Econometrie
Vakgroep BIK
De Boelelaan 1105
1081 HV Amsterdam Phone: +31-20-548708
The Netherlands Fax : +31-20-6462645

1991-1	N.M. van Dijk	On the Effect of Small Loss Probabilities in Input/Output Transmission Delay Systems
1991-2	N.M. van Dijk	Letters to the Editor: On a Simple Proof of Uniformization for Continious and Discrete-State Continious-Time Markov Chains
1991-3	N.M. van Dijk P.G. Taylor	An Error Bound for Approximating Discrete Time Servicing by a Processor Sharing Modification
1991-4	W. Henderson C.E.M. Pearce P.G. Taylor N.M. van Dijk	Insensitivity in Discrete Time Generalized Semi-Markov Processes
1991-5	N.M. van Dijk	On Error Bound Analysis for Transient Continuous-Time Markov Reward Structures
1991-6	N.M. van Dijk	On Uniformization for Nonhomogeneous Markov Chains
1991-7	N.M. van Dijk	Product Forms for Metropolitan Area Networks
1991-8	N.M. van Dijk	A Product Form Extension for Discrete-Time Communication Protocols
1991-9	N.M. van Dijk	A Note on Monotonicity in Multicasting
1991-10	N.M. van Dijk	An Exact Solution for a Finite Slotted Server Model
1991-11	N.M. van Dijk	On Product Form Approximations for Communication Networks with Losses: Error Bounds
1991-12	N.M. van Dijk	Simple Performability Bounds for Communication Networks
1991-13	N.M. van Dijk	Product Forms for Queueing Networks with Limited Clusters
1991-14	F.A.G. den Butter	Technische Ontwikkeling, Groei en Arbeidsproduktiviteit
1991-15	J.C.J.M. van den Bergh, P. Nijkamp	Operationalizing Sustainable Development: Dynamic Economic-Ecological Models
1991-16	J.C.J.M. van den Bergh	Sustainable Economic Development: An Overview
1991-17	J. Barendregt	Het mededingingsbeleid in Nederland: Konjunktuurgevoeligheid en effektiviteit
1991-18	B. Hanzon	On the Closure of Several Sets of ARMA and Linear State Space Models with a given Structure
1991-19	S. Eijffinger A. van Rixtel	The Japanese Financial System and Monetary Policy: a Descriptive Review
1991-20	L.J.G. van Wissen F. Bonnerman	A Dynamic Model of Simultaneous Migration and Labour Market Behaviour