R. Mittermeir (ed.)

Shifting Paradigms in Software Engineering

NJ SZT

Springer-Verlag Wien New York

R. Mittermeir (ed.)

Shifting Paradigms in Software Engineering

Proceedings of the 7th Joint Conference
of the Austrian Computer Society (OCG) and
the John von Neumann Society
for Computing Sciences (NJSZT)
in Klagenfurt, Austria, 1992

Springer-Verlag Wien New York

Prof. Dipl.-Ing. Mag. Dr. Roland Mittermeir
Institut für Informatik
Universität für Bildungswissenschaften Klagenfurt, Klagenfurt, Austria

Printed on acid-free paper

With 77 Figures

ISBN-13: 978-3-211-82408-5 e-ISBN-13: 978-3-7091-9258-0
DOI: 10.1007/978-3-7091-9258-0

PREFACE

SPSE '92 - **S**hifting **P**aradigms in **S**oftware **E**ngineering is held as part of the "Bildungswissenschaftliche Woche 92" at the University Klagenfurt, partially overlapping with the conference "Informatik in der Schule - Informatik für die Schule"[1]. SPSE '92 is the 7th joint conference of the OCG, the Austrian Computer Society, and NJSZT, the Hungarian John von Neumann Society for Computing Sciences. As such, it is a conference which has already substantial tradition. On the other hand, SPSE '92 is distinct from its predecessors, since - as a sign of maturity of computer science in the region - it limits its scope to a particular aspect of the computing sciences, to software engineering and notably to the shifting paradigms we currently witness in this discipline.

The shift of paradigms which currently takes place in software engineering has certainly many roots. Some of them can be found in object-orientation and the new opportunities and challenges offered by this approach for software construction. Building large software systems by combining interacting software objects might seem as frightening and revolutionary to a software engineer well trained in structured top-downism as the PC-revolution has been disturbing to the manager of a huge mainframe-based computing center. But the similarity might carry on: as PC's have not and will not replace mainframes, there will also be coexistence between classical and object-oriented approaches to software development.

The notion of coexistence is becoming ubiquitous though. The clear distinction into software systems on one hand, data base systems on the other hand, and artificial intelligence systems hopping on some shoulders can no longer be maintained. The boundaries become blurred and will eventually fade away. However, the textbook wisdom, especially as far as methodological aspects are concerned, is different in each of these three domains. Hence, the stability the discipline "Software Engineering" has acquired throughout the last 25 years is vanishing. Old teachings can no longer be fully backed, new ideas pop up - not all of them well tested, not all of them worth to be pursued, but several of them worth critical study and evaluation.

In the light of this situation of the discipline, where leading authorities in the field state that software engineering as needed in the 90s is both, beyond software and beyond engineering, the program committee has invited researchers and practitioners in the geographic domain of the two sponsoring societies and neighbouring countries to share with us what they consider as key factors with respect to application development, the underlying theory, and, last not least, the challenges for (continuing) education stemming from these shifting paradigms.

The response to this call for papers has been excellent. Hence, the program committee had an easy task to select out of the submissions those papers which warrant presentation and inclusion in the conference proceedings, those which have been accepted for presentation, but seemed to be yet too unrefined to warrant a full length publication, and finally, isolate those which just did not make it to this conference. I may state also with great pleasure, that in

[1] Mittermeir R.T., Kofler E., Steinberger H.: "Informatik in der Schule - Informatik für die Schule", Vol. 10 of "Bildungswissenschaftliche Fortbildungstagungen an der Universität Klagenfurt", Böhlau Verlag, Wien 1992.

spite of the regional focus of the conference which resulted from the partnership of the OCG and the NJSZT, we have been open to - and actively requested - papers from neighbouring countries. This openness was rewarded by submissions from the CSFR, Slovenia, Tunesia as well as speakers and (co-)authors from Canada, Russia and the US. So, we see this long established Austro-Hungarian partnership conference flourishing into a truly international venture.

The program consists of two keynote speeches, eight technical sessions, and one panel discussion. The latter should specifically address the chances and challenges facing (relatively) small countries in the light of concerted research efforts in the EEC, the US and Japan.

To conclude, I'd like to thank all authors for their effort and the members of the program committee as well as the referees for their kind support and cooperation. Special thanks go also to the two societies, notably their presidents and their secretariats, which deserve special mention.

Concerning the local organization, I have to say that this conference would not be, but for the dedication and support from Dr. Steinberger, Mag. Kofler, Mag. Janesch and Mr. Hüttel. My expression of gratitude goes to them as well as to all those who financially supported this conference.

Roland Mittermeir
Program Committee
Chairperson

CONTENTS

Shifting Paradigms in Software Engineering
Klagenfurt, 21. - 23. September 1992

PROGRAM COMMITTEE

This conference has been made possible by grants from the following organizations:

Bundesministerium für Wissenschaft und Forschung
Verein der Freunde des Instituts für Informatik
Digital Equipment Corporation
Siemens AG Österreich
BACHER Electronics

OPENING LECTURE

Chair: R. Mittermeir

Software Engineering: Beyond Software and Beyond Engineering

L. Belady
Mitsubishi Electric Research Laboratories
Cambridge, Mass. USA

Two distinct types of software are emerging. One type includes traditional program "components" which are relatively easy to specify and to sell in large numbers. The other is the software "glue" to integrate islands of computer applications into enterprise-wide systems. Building the second type demands more than software engineering. Expertise in computer hardware and in the application domain are indispensible.

PROJECT MANAGEMENT

Chair: E. Knuth

Computer Integrated Work Management (CIW)

Univ.-Prof. Dr. Gerhard Chroust
Systemtechnik und Automation
Kepler Universität Linz
Altenbergerstr. 69
A-4040 Linz, Austria

Abstract

Software project management is often hampered by a lack of complete and up-to-date information on planned and actual activities, on their actual status, etc. At the same time **software engineering environments** have gained widespread acceptance and use within the last decade, providing **guidance** for the **development process** and integrating **access to tools**. They can provide most of the information needed for project management. An attainable vision of the future is the **integration** of classical project management with process guidance in order to arrive at **Computer Integrated Work Management (CIW).**

Advantages are an effective communication between process guidance and project management and the ability to hide the added complexity from the user by adequate filtering on a need-to-know basis.

1 The Process Guidance/Project Management Gap

The need to control the development of systems (including software engineering projects) has long been understood. **Project Management** has a long standing tradition in engineering disciplines - building the pyramids was obviously an admirable achievement of project management. With respect to software projects it seems that we are not so successful [1]. Some of the reasons are eloquently discussed by F. Brooks in his famous paper 'No Silver Bullet" [2]. The reason for this state of affairs is partly due to the separation of Process Guidance and Project Management (see below).

1.1 Computer Aided Process Guidance

In order to bring a touch of industrialisation into software engineering we have seen the introduction of software engineering environments (also called Integrated Project Support Environments, etc. [7][10][11]) within the last decade.

The main purpose of these environments is threefold:

- providing an integrated, uniform access to a **tool set** [9][12],
- guiding the user through a pre-defined sequence of steps, defined in the **process model** [5][8].
- relieving the user from many administrative details like storing/retrieving results, finding standards and explanations, completing reports.

The availability of sufficient computing power fostered the idea to let the computer enforce observance of the intended (and pre-defined) process. The basic idea is rather simple, but nevertheless far-reaching. A **process model** defines like a template the way how development processes should be performed. This process model is in machine-readable form such that a **model interpreter** can present it step by step to the users (Fig. 1) via a so-called work bench. The model interpreter will help the users to follow the process (providing **process guidance**) and ensure observance of the intended process. At the same time the model interpreter takes care of the interface to the **tools**, relieving the developer of many boring and akward details. Additionally the model interpreter handles - in cooperation with an adequate repository - the retrieval and storage of the results. The result is a software engineering environments (Fig. 2) like ADPS [3].

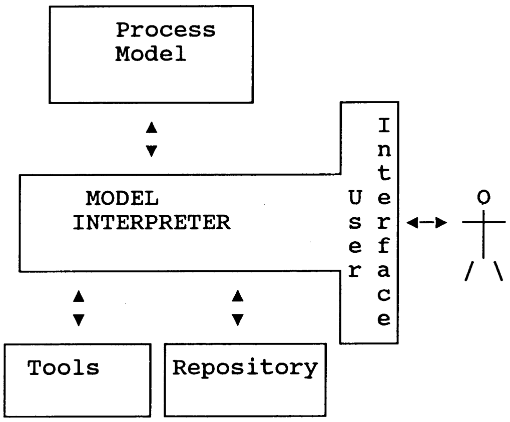


Figure 1: Process Model and Model Interpreter

1.2 Process Model

The process model plays a central role in guiding the user. It contains a detailed description of all activities to be performed in the course of the project. In its most basic form it consists of:

Result Classes: They describe all intermediate and final results of the development process.

Activity Classes: An activity class is the smallest unit of work identifiable at the chosen level of description. The activity class also defines the results to be used (the 'prerequisites') and the results to be produced (the 'deliverables'). Methods and/or tools are also identified.

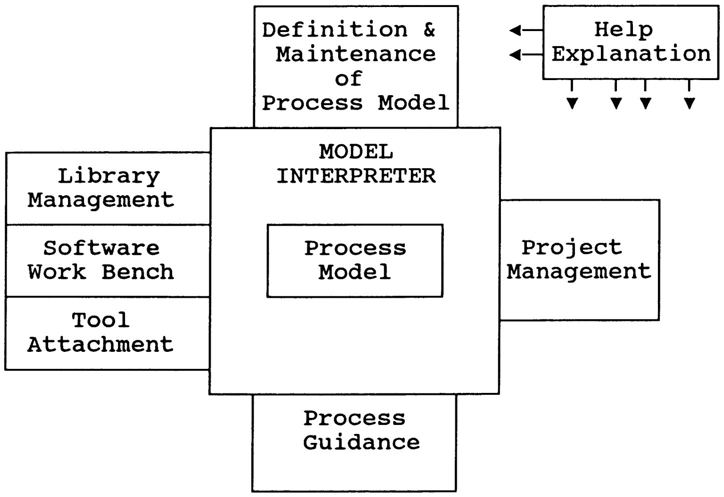


Figure 2: Components of the Software Engineering Environment

Result Class Structure: It describes the relationships between the various result classes (e.g. "object module *is compilation of* source module")

Activity Class Structure: It describes both the static relationships of activity classes (e.g. "coding *is part of* implementation phase") and their dynamic relationships (e.g. "coding *must occur after* design" [4]).

In most cases the process model is represented as a more or less strict network of activity classes and result classes, cf. Fig. 3 [3]). One has to keep in mind that the process model is a *template*. Each project will be an **instance** of the given process model, i.e. it will consist of activities, results, a result structure and an activity structure (cf. Fig. 6, left side), derived from the respective classes.

1.3 Classical Project Management

In the last few years we have seen a growth in project management tools [6] which provide all the functionality needed for successfully managing a project. Project management is mainly concerned with (cf. Fig. 6, right side):

Work Packages: These are the smallest units which are individually planned, they usually correspond to one or a small set of activities.

Resources: These comprise personell, money, software and hardware. In that respect we may also consider time as a resource, despite its slightly different nature.

Resource Constraints: Both the quantity, the timely availability and interdependencies between resources have to be taken into account.

Work Plan: The work plan tries to strike an acceptable compromise between the different requirements and constrains. It specifies a temporal ordering for the work packages based upon the logical dependecies (expressed in the Activity Structure) and the resource requirements.

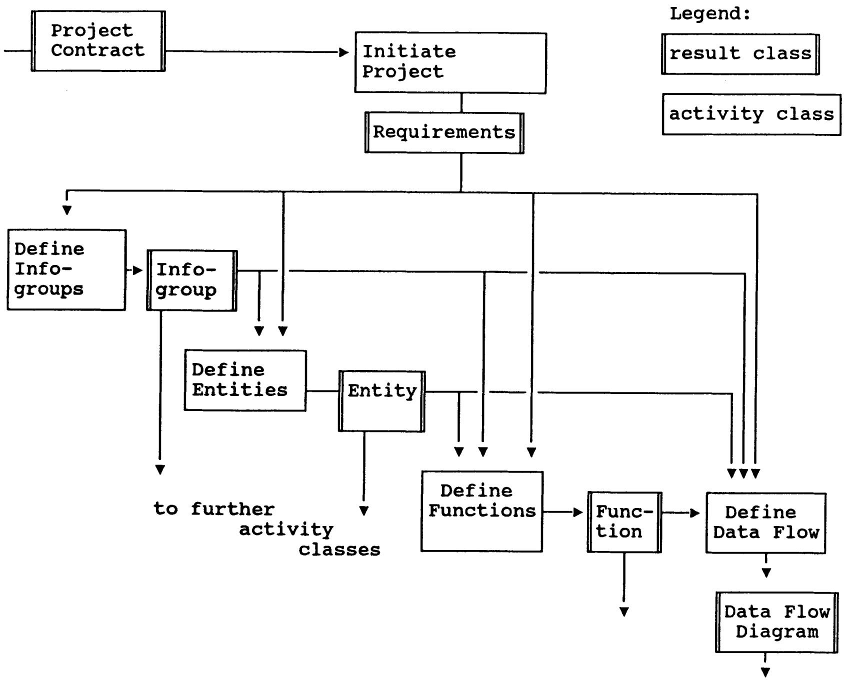


Figure 3: Section of the ADPS process model

2 Computer Integrated Work Management (CIW)

Project management, to do meaningful planning and control, needs accurate data about planned and actually performed activities and about the planned results and their status (Fig. 4). As long as the definition of the development process was largely intuitive or at best defined on paper [8], it was difficult to provide accurate data to project management (a developer is usually *'90% finished'*, no matter how much more has to be invested in his module). And many of the necessary activities where forgotten when establishing a project plan. Only the integration of process guidance and classical project management is able to provide the needed synergetic effect, both providing to the process guidance the necessary information about additional, resource-based constraints on sequencing and to project management the information about the planned and actual activities and results. Obviously most of these considerations must be based on the actual instances of the respective classes in the process model .

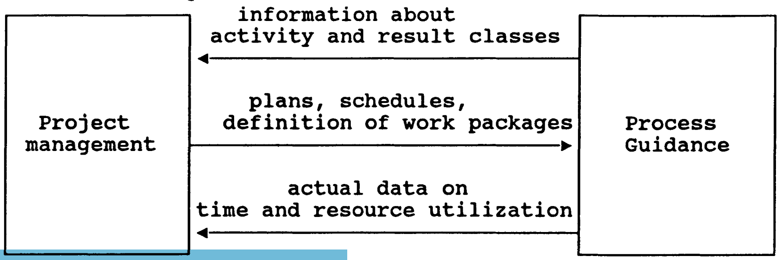

information about activity and result classes
plans, schedules, definition of work packages
actual data on time and resource utilization
Project management
Process Guidance

Figure 4: Cooperation of process guidance and project management

One can delineate the subareas of CIW as shown in Fig. 5. We may say that the areas of Configuration Management (including versioning), and Personell and Resource Management and Scheduling are well understood. The area of Activity Management just becoming state-of-the-art [5] [8].

What is new is the interface between the left and the right hand side of Fig. 5. This will be discussed in the sequel.

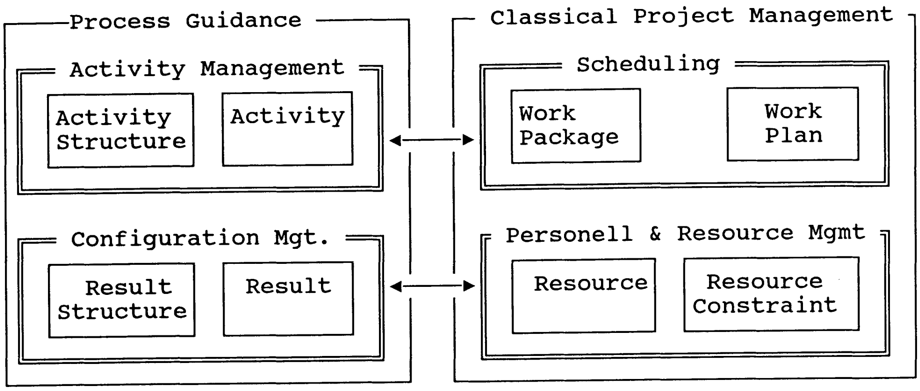


Figure 5: Areas of Computer Integrated Work Management

2.1 Components of CIW

CIW - as a synthesis of process guidance and resource management - will mainly involve the components shown in Fig. 6. In this figure the most important relationships between the individual components are shown, many others are implied. On the left hand side we recognize the domain of Configuration Management: results and their relationships. The relationships between the results imply the transformations of prerequite results into deliverables (the activities). Additionally the ('dynamic') order in which the activities should be performed must be defined (the Activity Structure).

On the right hand side the components of classical project management (which is primarily resource management) are shown. Planning and control is based upon work packages (each usually containing several activities) of the process model. Work packages may also contain further activities which are not in the process model like vacation, education etc. Each of them needs certain resources (based upon the resource need of the contained activities). Resource constraints put restrictions on admissible work plans.

This point of view separates the influence of the *logical* structure of the process (expressed in the process model) from resource-oriented concerns (as reflected in the work plan).

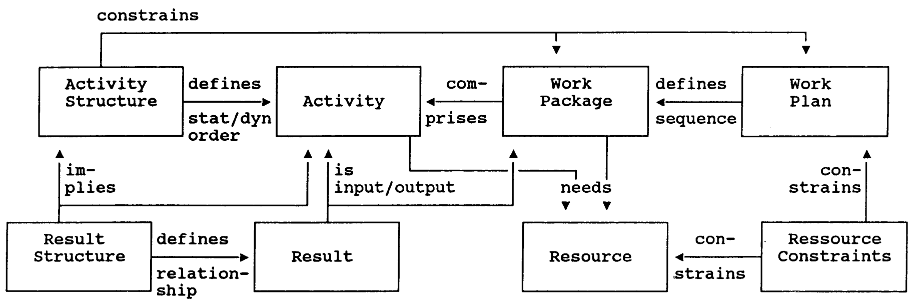


Figure 6: The basic components of Computer Integrated Work Management

2.2 Interfacing Process Guidance and Scheduling

Fig. 6 shows the necessary information exchange for integrating process guidance with project management. One can identify five key relationships:

Work Package comprises Activity: Usually several activities will be associated with one work package. Additionally not every activity needs planning. Typically a compilation, although usually an activity on its own, will not appear in the work plan. The actual granularity (and thus the number of activities collected in one work package) depends on numerous parameters like criticality of the project, experience of development team, size of project, enterprise culture etc. Despite the fact that two projects may use the same process model their work packages and the work plan may be drastically different.

Fig. 7 on its left hand side shows a rudimentary process model together with the produced results ('Spec0', 'Design0', ...). Several work packages (WrkP1 to WrkP5) have been defined. For the activity 'Code' two work packages have been defined, another work packages is concerned just with education.

Activity Structure constrains Work Package: The activity structure is mainly induced by the dependencies between the data produced and used. A work package may not contain an agglommeration of activities which violates the data dependencies.

Activity structure constrains Work Plan: Similarly the sequencing of the individual work packages must take into account the data dependencies between the respective work packages.

Work Package needs Resource: Based on the resource need of the activities contained in a work package, the resource demand of the work package can be derived.

Resource Constraint constrains Work Plan: Resource constraints (e.g. restricted availability of a specialist, of hardware, time constraints) impose further restrictions on the sequencing of otherwise independent work packages in the work plan.

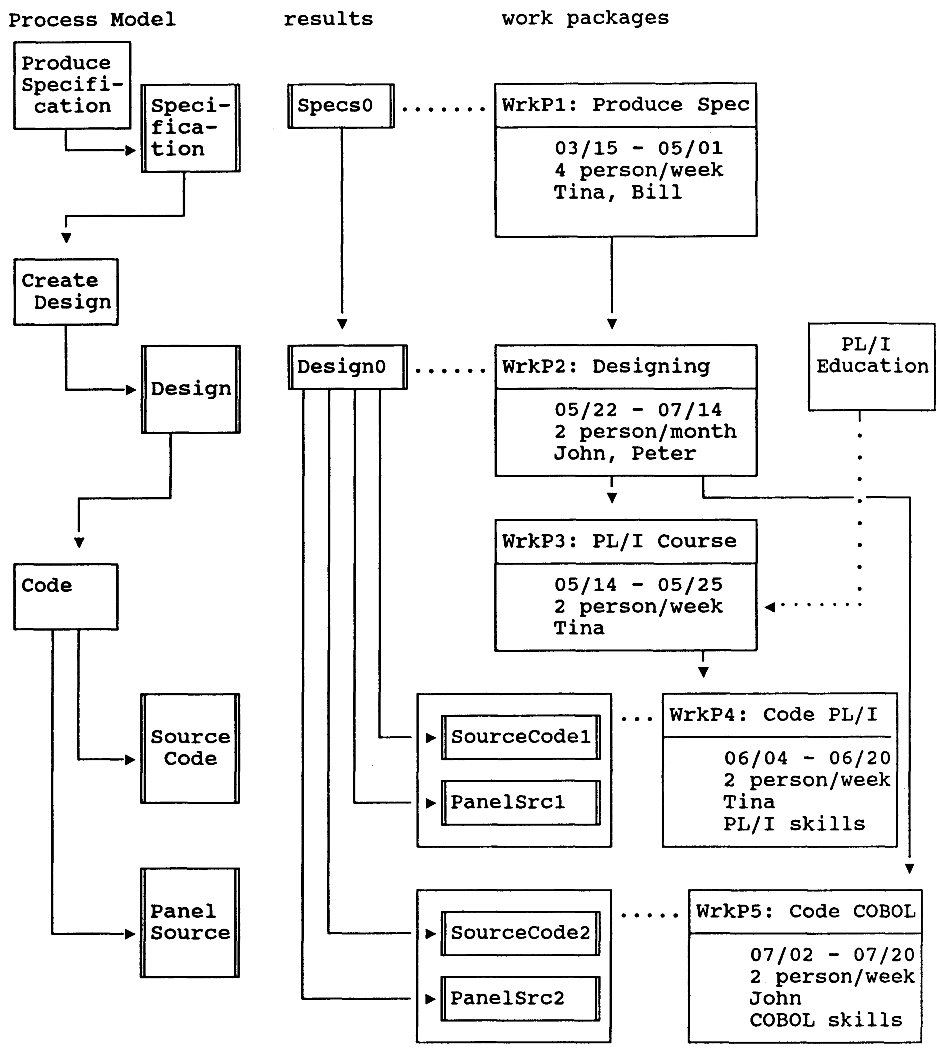


Figure 7: The relation of a process models to its instances and work packages

3 Personalizing the Work Management

Up to now the *whole* development process had been considered. The combination of process guidance and classical project management will obviously increase the complexity of the information to be administered and presented. In order not to overwhelm the individual user, it is necessary to *reduce* this complexity by providing individualized views. This can be achieved by providing a **resource-oriented view** of the process. For each individual resource - especially for a developer - one can isolate those activities which are his/her concern (Fig. 8). The project planner/leader still has access to the totality of information and can make the necessary adaptions. An individual user will generally only see those work packages which concern him (the 'need-to-know'). This can be achieved by a **To-do-List** (Fig. 10). In the example the To-do-List contains Tina's work packages from Fig. 7 plus a few others (these could even result from a different project assignment). The To-do-Lists are periodically updated by checking whether further work packages became ready, etc. (Fig. 9).

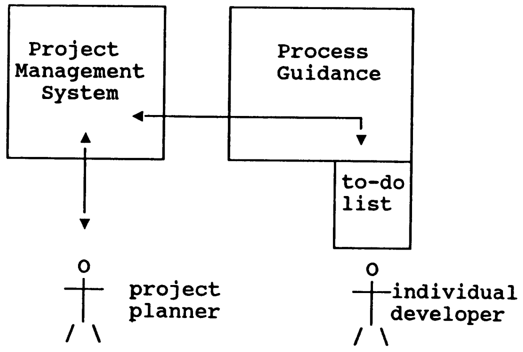


Figure 8: Attachment of Project Management

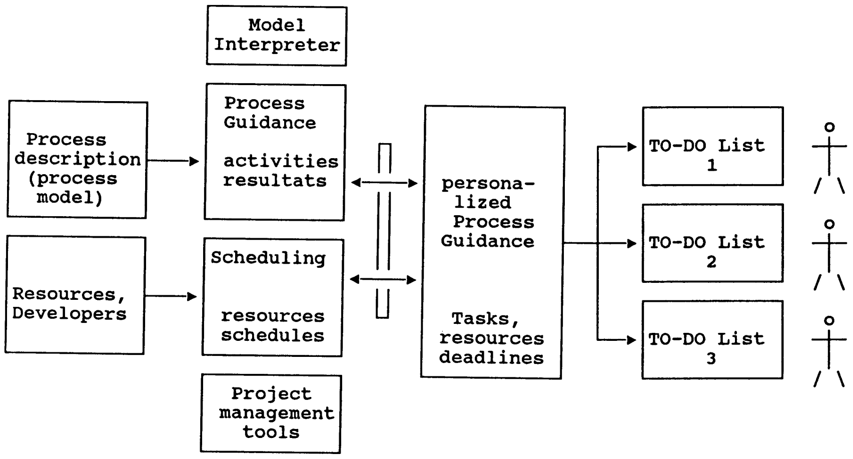
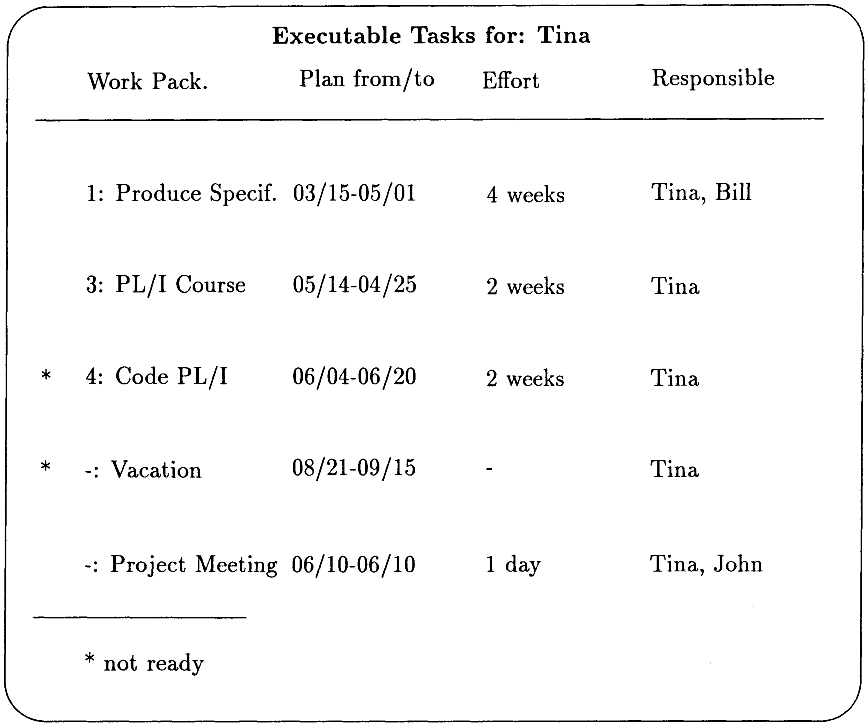


Figure 9: Personalized Task Management



Executable Tasks for: Tina

	Work Pack.	Plan from/to	Effort	Responsible
	1: Produce Specif.	03/15-05/01	4 weeks	Tina, Bill
	3: PL/I Course	05/14-04/25	2 weeks	Tina
*	4: Code PL/I	06/04-06/20	2 weeks	Tina
*	-: Vacation	08/21-09/15	-	Tina
	-: Project Meeting	06/10-06/10	1 day	Tina, John

* not ready

Figure 10: To-do-List

4 Summary

The proven usefulness of computer support for project management and the gradual acceptance of software engineering environments as the path to a more reliable, stable and productive system development make **Computer Integrated Work Management (CIW)** the next logical step. CIW carries with it the promise of integrating both project guidance and project management based on commonly available information. At the same time the complexity for the individual user can be reduced on a need-to-know basis.

References

[1] Brooks F.P.: The Mythical Man-Month.- Addison-Wesley 1975

[2] Brooks F.P.Jr.: No Silver Bullet - Essence and Accidents of Software Engineering.- Kugler H.J. (ed.): Information Processing 86, IFIP Congress 1986 pp.1069-1076

[3] Chroust G.: Application Development Project Support (ADPS) - An Environment for Industrial Application Development.- ACM Software Engineering Notes, vol. 14 (1989) no. 5, pp. 83-104

[4] Chroust G., Goldmann H., Gschwandtner O.: The Role of Work Management in Application Development.- IBM System Journal, vol. 29 (1990) no. 2, pp. 189-208

[5] Chroust G.: Modelle der Software-Entwicklung - Aufbau und Interpretation von Vorgehensmodellen.- Oldenbourg Verlag, 1992

[6] Elzer P.F. (ed.): Multidimensionales Software-Projektmanagement.- AIT Verlag Hallbergmoos 1991

[7] Huenke H. (ed.): Software Engineering Environments.- Proceedings, Lahnstein, BRD, 1980, North Holland 1981

[8] Humphrey W.S.: Managing the Software Process.- Addison-Wesley Mass 1989

[9] Martin J.: Information Engineering, Book I: Introduction.- Prentice Hall, Englewood Cliffs 1989

[10] Oesterle H.: Anleitung zu einer praxisorientierten Software-Entwicklungsumgebung, Band 1.- AIT Verlag München 1988.

[11] Oesterle H.: Computer Aided Software Engineering - Von Programmiersprachen zu Softwareentwicklungsumgebungen.- Kurbel K., Strunz H. (eds.): Handbuch der Wirtschaftsinformatik.- Pöschel Stuttgart 1990, pp. 345-361

[12] Stork B.: Toolintegration in Software-Entwicklungsumgebungen.- Angew. Informatik 1985, No. 2, pp. 49-57

Methods and Tools for Systems Engineering and Application Software Development

G. Klimkó, P. Krauth, B. Molnár

Information Technology Foundation of Hungarian Academy of Sciences
H-1525 Budapest 114. P.O.B. 49, Hungary,
Telephone: +36 1 169-9499, Fax: +36 1 155-3376
e-mail: h4445mol@ella.hu

Abstract. A general picture of the recent development in the field of systems engineering and application software development is presented. In the last years new aspects of systems development have been recognized. However, these are not technical ones, and they serve the users' interests rather than the developers'. Methodologies have been worked out for the new areas, and also supporting tools emerged. This process has an impact on the software providers, because the users expect them to be knowledgeable on the new areas, too. In short terms, the meaning of 'structured paradigm' has been widened, and we can talk a certain change of paradigms. The paper encounters some of the new areas and briefly describes a methodology of that area.

1. Introduction

The paradigm problem. When we are talking about 'paradigms', we might have to define what do we mean by this word. In the computing community people tend to use the word with a certain technical sense, like 'object-oriented paradigm', or 'knowledge-based paradigm'. This refers usually to the technical background, how the software engineer describes the system under investigation and how he builds the supporting software of the system. These questions mean problem for (and only for) the software engineer himself. Users of the system would perhaps be not too interested in such technical details.

Specifically, in the world of application software builders, the word 'structured paradigm' has a common use. This term usually refers to two separate meanings. Using a 'structured paradigm' indicates that during systems analysis and design a structured methodology is to be used, and in the implementation phase structured programming concepts will be followed. That way, the basic meaning of the word 'paradigm' was widened, because it pertains not only to the technical software design process, but to the phases of systems analysis and design, too.

The main structuring tools in the description of the application software building process are the *life cycle models* (waterfall [Layzell 1989], evolution [Booch 1991] etc.). Life cycle models for systems analysis and design were already formed in the 70s. Systems modelling techniques were invented (data flow diagrams, Petri nets, entity-relationship model, relational data analysis etc.), which tackle separate aspects of a system. These techniques serve two purposes. Firstly they give a systematic, conscious and usually semi-formal way of describing the given system. Secondly, they act as a communication tool with the user. The techniques were incorporated in a structured framework, that prescribes, how to use the techniques. The framework is based upon a life cycle model. Finally, methodologies were formed. Thus, a methodology is a combination of techniques and methods in a disciplined way. A methodology must have an underlying philosophy and life cycle model, Examples of such methodologies are JSP [Jackson 1992], Yourdon [Yourdon 1975], Merise [Matheron 1990], SSADM [NCC 1990], SDM [Turner 1990] etc. Different methodologies cover different extent of the life of systems building. For example the SDM (Systems Development Method) relates up to the implementation, however Merise does not cover implementation.

A standardization process have begun in the area of systems analysis, mainly in the governmental sector on a nation-wide scale. For example, in the UK there is SSADM, in France there is Merise, in the Netherlands SDM. In these countries, the standardization is in different stages (in the UK SSADM is just before becoming a British standard). As a consequence of the national governments forced standardization process, the *de facto standard* systems methodologies themselves has changed.

Having only national methodologies, however, is not enough. After 1992, on the new common European market there will be a demand for a commonly understandable systems analysis methodology. To achieve this, the Euromethod project was initiated. As a result of this project, in 1995 we shall have a *Generic Process Model* as life cycle model and a *Unified Terminology* [Euromethod 1991].

The life-cycle models focused again on the technical aspects of the development. They concern more about the systems analyst and the application developer rather than about the customers of that application. From the point of view of the customer there are other extremely important aspects in using information technology (IT). This aspect can be called as the business view of the usage of IT. Examples of the new areas (aspects) are IT strategy planning, IT project management, quality management, IT risk evaluation and security analysis, systems maintenance, software package evaluation and selection. The common attribute of these is that they serve more the interests of the user, rather than of the developer.

In summary, the way how the users of IT are thinking of the usage of IT has been changed. The applications software providers can not dictate anymore only with technical justifications. This phenomenon has a influence on the of the application software builder community. The meaning of 'structured paradigm' is widening now. In that sense we can speak about changing paradigms.

Slow industrial take-up. The surveys on the usage of methodologies or just the analysis techniques show a surprisingly low percentage of penetration [OECD 1991], [Rock-Evans 1989]. This is really astonishing if we think the governments (eg. UK, France) or big private companies (eg. Arthur Andersen, McDonnel-Douglas) how strongly favour the usage of systems development methodologies. There is a certain agreement on the fact, that the usage of methodologies result in better quality software, too. So the expectations for the new IT areas give a very sad predictions, if even such a well-known area like systems analysis is so badly handled.

The question is, how can we improve this situation ? The techniques and the methodologies must be obviously understandable and attractive. To demonstrate their power, we believe a good infrastructural background would be enough. A very good example of such a background is the support around SSADM. In the following paragraphs, a short description of this infrastructure is given.

SSADM is in the public domain, that is, the documentation of the method is publicly available. Other activities of systems analysis like estimating are covered in separate subject guides. Several textbooks on SSADM are available eg. [Ashworth 1989], [Eva 1992]. As SSADM is not committed to any company, there is no danger to stick to a specific vendor's method.

There is a central governmental organization, the Central Computer and Telecommunication Agency (CCTA) which is responsible for the maintenance of SSADM. The SSADM Users Group was formed to collect feedback on the method, and based upon this information the method is regularly updated.

SSADM is taught at the British universities. This approach assures that necessarily educated personnel is available. There are a large number of consulting firms that teach SSADM, too. The teaching materials are also evaluated by CCTA. This procedure assures the quality of the education.

There is an SSADM examination procedure controlled by the Information Systems Engineering Board of the British Computer Society. More than 2000 systems analysts has passed this examination up today, including non-UK experts, too. The existence of such an examination procedure makes it impossible to misuse the method. Because the usage of SSADM is forced at governmental sites, for applicants of IT jobs at these sites to have an SSADM certificate usually is a must.

The widespread usage of SSADM lead to the appearance of the SSADM computer support tools (currently there exist more than a dozen of this commercially available CASE tools). These tools are ranked by the CCTA on a 1 to 5 point scale. Every year there is review of the CASE tools at the regular meeting of the SSADM User's Group.

This infrastructure has been built up through almost a decade. It serves an excellent example how to support IT activities that could be followed on other areas. Therefore it is suggested, that the way how SSADM and its infrastructure was built, is to be used on the other areas, too. For this purposes the lessons learned on systems analysis can be applied in four stages. In the first stage techniques has to be developed and then into turn to be incorporated into structured frameworks. This forms a methodology. In the second stage a standardization approach would be reasonable on a nation-wide level. In the third stage a proper infrastructural background has to set up, that assures maintenance and feedback. In the fourth stage an European standardization process is to be initiated.

Design evaluation. Assessment of the goodness and validation of a particular design was always a crucial point in systems engineering. The problem is: how can we judge if a design decision were good, and how can we assure in advance that no 'bad' decision will be made. To answer such a question, first the real meaning of 'good' and 'bad' has to be defined in a objective way, that is, in quantifiable terms. This evaluation has to be incorporated into the systems analysis methodology. In order to be able to recognize bad points in the systems analysis documentation, the used methodology can use dichotomy. This means that an aspect of the system under investigation must be depicted from more than one viewpoint, and resulting documents (products) has to be cross-checked. The usage of graphical, formal or semi-formal description techniques are recommended in the documenting methods.

The detection of the bad decisions can be incorporated into the project management methodology. Software and design quality assurance is the way to handle this task. Organizing the quality assurance process is clearly different from the traditional technical activities of systems development. However, without quality assurance it is not possible to achieve good quality systems. In such parts of the world like chemistry or power plant control, quality assurance is an obvious must. The relevance of quality assurance in the IT industry, however, is not really recognized yet. Project management and quality assurance can be handled with help of methodologies the same manner, as we did earlier on the field of systems analysis.

In the paper we outline the current state-of-the-art on the areas of systems analysis, IT strategy planning, IT project management, IT risk analysis and management, looking at them from the user's perspective. These major areas all serve as a tool in order to achieve quality software products. For each area, an example of a corresponding methodology is given, and a supporting tool is mentioned. All the examples are excerpted from the UK practice. There are several reasons for choosing British examples. Most of the shortly described methodologies are in the public domain, so they are easily available. Very strong and well-sounded infrastructural support is available for IT services in the UK [CCTA 1990], [CCTA 1991a].

That is, there is a coordinating organization and therefore the ways of development and feedback are assured. According to the surveys, UK is the leader in the usage of analysis techniques (cca. 33%).

Among the IT leaders in Europe, the French practice could have been an other possible choice, but documents are mainly accessible only in the French language, which is not widely spoken in Hungary. In the UK, naturally all the documentation is in English, which is the *de facto* the working language of the computing society. In Germany, only at the military are standard methodologies, and it is very difficult even to have literature on them because of the nature of the applications.

Although in Hungary the different techniques and methods are taught, there is no widespread usage of methodologies and/or analysis techniques. There is no preferred or recommended systems analysis method even in the government sector. However, as Hungary likes to join the European Communities, it has to conform to the European expectations on the IT field, too. It has no sense to develop for Hungary own national methodologies for the separate IT fields. Rather, we have to choose from the elaborated ones. Taking into consideration the above mentioned facts, the British practice is a definite candidate for this purpose.

2. Systems analysis and design

In this section an overall view of the SSADM [NCC 1990] (Structured Systems Analysis and Design Method) will be presented, and supporting CASE tools will be mentioned. A short paragraph on the Euromethod standardization project closes the section.

Roots of SSADM. For UK governmental sites at IT development projects the use of SSADM is compulsory from the beginning of the 80s. The main reason of introducing such a recommendation was to get to such a situation, where different IT projects within the government can be compared against each other and thus be under control. SSADM was developed by LBMS (Learmonth and Burchett Management Systems, a private firm) then it was purchased by the British Government. The methodology is in the public domain, users need not have to pay for the usage of SSADM. The owner of the method is the CCTA.

SSADM framework. Being a structured methodology, SSADM breaks down the development process into modules. Modules are built from stages, stages are in turn defined by steps. At the end of any module the development can be cancelled by the user. This way the development process is more strictly controlled.

For all these building bricks it is clearly defined, what are its inputs and outputs, what are the preconditions to start, what techniques should be applied. The inputs and outputs are called products. Products are built up in a hierarchy structure. For all products, there is detailed description in the reference manual. The product descriptions include quality criteria and dependency descriptions, too. By these criteria the quality of the products can be checked and measured. Products are interdependent and are required to be updated at separate steps.

SSADM techniques. SSADM is a data- and user-driven methodology. There is a strong emphasis on the communication with the user. This is done mainly via graphical techniques rather than a verbal way. The base of most techniques is a Jackson-like notation. On the other side, a lot of investigation is done, what data is to be stored in the system and how will it change. SSADM applies the popular and well known-techniques of Data Flow Modelling, Entity-Relationship Modelling and Relational Data Analysis. The other, maybe not so well-known techniques include Event Modelling, Function Definition and Dialogue Design. Two special techniques, the Business System Option (BSO) and Technical System Option (TSO) make the user to be real control of the development. The selected BSO must define the scope of the IT system, the selected TSO must clearly define the hardware-software basis of the implementation of the selected BSO. The use of these two techniques help to lessen the usual debate on the delivered system.

Each technique is documented individually in the reference manual. Dependencies among the techniques are precisely described. For each step it is prescribed, which techniques should be used and what will be the result (product) of the techniques.

Computer support. Because SSADM uses lots of graphical techniques, and the results of the techniques are in often cross-checked, there was an obvious need for supporting CASE tools. However, only the technical need would have been enough to press CASE builders creating such tools. It was the widespread usage of SSADM that lead to the appearance of the SSADM computer support tools (currently there exist more than a dozen of this commercially available CASE tools). These tools are ranked by the CCTA on a 1 to 5 point scale. Every year there is review of the CASE tools at the regular meeting of the SSADM User's Group.

The price and quality of these CASE tools is disperse. Price categories start at 500 pound for single-user tools on an AT category machine (PC SELECT). In the midrange of the price categories, one of the leader products is SSADM Engineer from LBMS. The reason for pointing out this product is its very sound technical ground. This is a multi-user tool that supports most of the SSADM products. The designers chose the PC with DOS as the hardware/software basic platform, that can be usually easily provided. They also avoided the trap of developing their own database, network and user interface. For these purposes the tool uses off-the-shelf products, namely a commercially available relational database server for the data dictionary (with the SQL interface), the NETBIOS as network interface and Microsoft Windows as user interface. This foundation provides a technically superior solution, with excellent facilities. At the upper price bound (cca. 50.000 pound) you can find excellent product running on workstations only with superb supporting capabilities (SSADM-SF from Systematica).

European integration. There are nation-wide accepted systems analysis methodologies in other European countries. (in France Merise, in the Netherlands SDM, in Spain MEIN etc. [Euromethod 1991]) The European integration process obviously popped up the need for a common language. For this reason, the Commission of the European Communities initiated the so-called Euromethod project. This based upon six European and one American systems analysis and design methodologies. However, the wide-spread use of the national methodologies would make the introduction of a new super-method very difficult (if not even impossible). The purpose of Euromethod is therefore *...to help participants in IS planning and engineering activities choose the most cost effective approach to meeting their problems... It will be an umbrella methodology which harmonies the disparate methods currently use for Information Systems Engineering in Europe.'* [Euromethod 1991]. The scheduled finish of the Euromethod project is 1995.

3. IT strategy planning

IT strategic planning must not be confused with business planning, although the results of business planning can be utilized in the IT strategic planning process. In any organization that utilizes IT services, the costs and spendings must be justified for the management [CCTA 1991]. That supposes, that the organization does know its business aims and it is able to plan, where and how to use IT. Ideally, IT must serve the real business needs of the organizations. IT strategic planning concerns in the IT activities of an organization for a 3-5 year scope.

The aspect of making IT strategy planning is definitely not a technical one. The results of an IT strategy planning is more interesting for the business managers than to the IT people. One of the aims of IT strategy planning is, that business people have to understand and commit to the use of IT within their organizations.

There are structured methodologies in this area, too. Big consulting companies (eg. Logica) usually have an own IT strategy planning methodology, but these are not public. In the UK, on this area there is no recommended methodology for the governmental sector. However, the Information Systems Guides book A2 from CCTA does contain guidelines for IT strategy planning. The LBMS Strategic Planning Method (LSPM) , which will be shortly described here, is a commercially available and conform methodology to the IT Infrastructure Library [LBMS 1992]. LSPM education and reference manuals can be purchased and than internally used.

Roots of LSPM. The owner of the methodology is a consulting firm, their experience is summarized in LSPM. The economy related parts of LSPM are based upon the works of Porter and Parsons, the data analysis part bases on the entity-relationship model.

LSPM framework. In LSPM the strategic planning process is divided into steps, steps into tasks. There are role descriptions for the participants in the strategic analysis. Having studied the business environment, LSPM investigates the current use of IT first, then its future use. The investigations are done parallel from two viewpoint within LSPM. One is oriented toward a business description of the organization, the other is connected to traditional data analysis. At the end these two views are merged into a portfolio of IT project specifications. These projects are to be implemented in the organization in the next 3-5 years.

LSPM techniques. Simple identification and categorization techniques are used to describe the business/service areas, their importance and the used IT strategy. An organization can judge the necessity of IT on a specific area on this basis. The strategic analysis process involves a lot of interviews with upper management. To help this activity, certain techniques are presented for planning and making interviews.

The structure of the organization changes frequently, but the structure of data maintained at the organization does not vary so often. This is the reason, that simplified (not too detailed) data models are set up, that describes the ideal, the current and the transitional data structure for the organization. The well-known entity-relationship modelling techniques are used here, with a special attention not to run into the not necessary, deeper details. The transitional data models are connected to the project in portfolio.

Computer support. There is software support for LSPM. The tool supports the graphical data modeling techniques, and all the collected information is stored in one database. Various reports are available to present the results.

4. IT project management

Project management is classic bottleneck of IT projects. In the UK governmental sector the usage of the PRINCE (PRojects In Controlled Environment) methodology is compulsory on IT projects [CCTA 1991b]. PRINCE is definitely designed for (but not restricted to) IT projects. It interfaces to SSADM and CRAMM. SSADM itself does not contain project management guidelines (for example, as an alternate approach, the Merise systems analysis methodology does).

Roots of PRINCE. The ancestor of the method (called PROMPT) was developed by LBMS. The current form of PRINCE was worked out by the CCTA. CCTA is the owner of PRINCE, which is in the public domain.

PRINCE framework. PRINCE consists of five components. It defines the project organization, the necessary plans, gives controls, lays down the description rules of products and activities and has a configuration control component.

PRINCE has two underlying principles. In PRINCE it was recognized, that the three contributing but different views in IT development process, namely the business, the technical aspects and the user interests have to be separated. The different interests are represented by different persons in the separate project organizations. For each participant of an IT project, his/her role and responsibility is (and must be) precisely defined. The structure of the project is illustrated in Fig. 1. The usual practice of having just one powerful person (the project manager) is abandoned, a technique of controlling the project manager itself is included in the method.

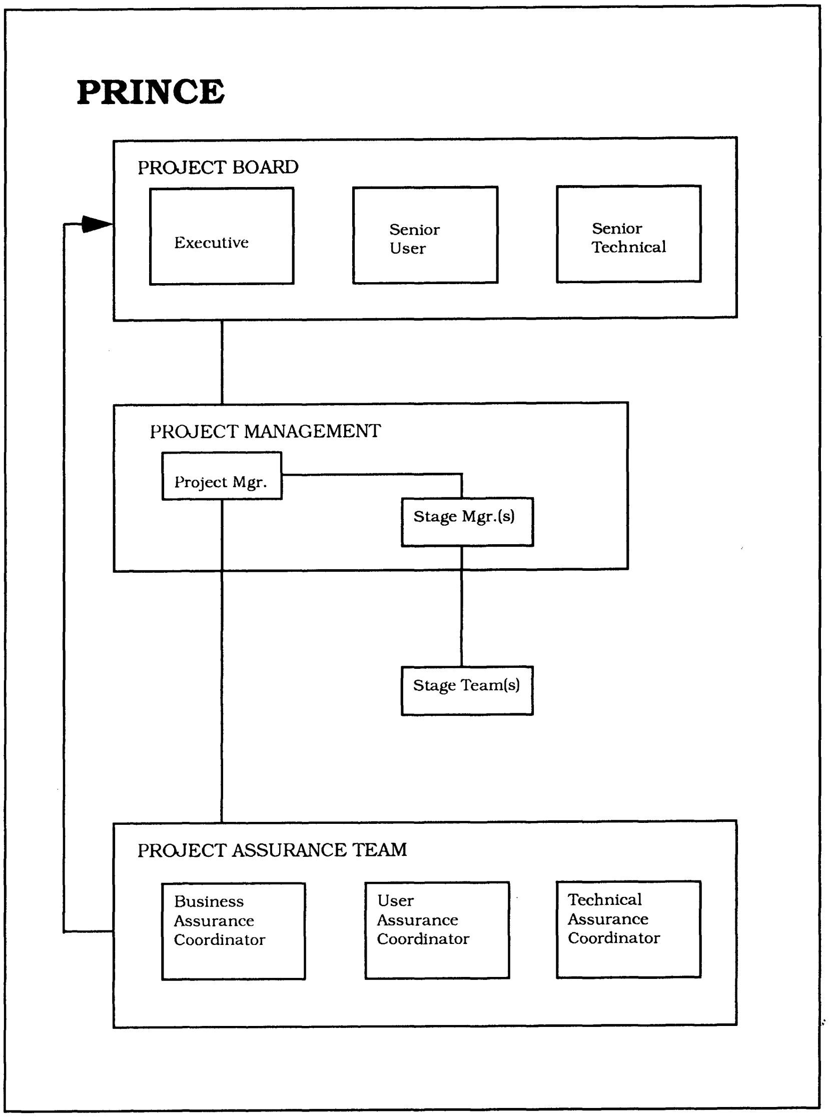

PRINCE
PROJECT BOARD
Executive
Senior User
Senior Technical
PROJECT MANAGEMENT
Project Mgr.
Stage Mgr.(s)
Stage Team(s)
PROJECT ASSURANCE TEAM
Business Assurance Coordinator
User Assurance Coordinator
Technical Assurance Coordinator

Fig. 1

On the other hand, PRINCE is product (and not activity) oriented. That is, before a development stage it has to be defined clearly, what are (and what are not) the deliverables of that development. A Product Breakdown Structure is to be created before any activity starts. The definition of the products must include quality criteria and must be agreed on between the user and developer. This approach helps to avoid the usual misunderstanding between the user and the developer, when the development is finished.

PRINCE techniques. For the organization component, PRINCE has a dictionary that describes precisely the roles in an IT project. This includes the responsibilities, the specific tasks and required knowledge and experience descriptions for each role, too.

In the techniques of the PRINCE planning component there is nothing IT specific. Types of plans include technical, resource and exception plans. Plans have to be produced on different levels (project, stage, detailed) for the technical process, for the usage of the resources, for the case of exception.

The controls component of PRINCE differentiate management and product controls. Management controls are built in at specific phases of the development, at the so-called control points. For each control point the objective, the attendees, their roles and an activity checklist is provided.

For product controls, the quality review and a technical exceptions handling technique is used. Procedures for both techniques are defined in a great detail. Technical exceptions are subdivided into project issue reports, off specification reports and requests for changes. Products are subdivided into technical, management and quality products. For each product there is a product description that contains the purpose, composition, format and quality criteria of that product.

Configuration management is built into the product controls. That is, there are elaborate configuration identification and status accounting schemas.

Computer support. Just currently are emerging the supporting tools for PRINCE. An example is Kernel-PMS/PRINCE from Transaction Point.

5. IT risk analysis

IT risk analysis has a growing importance. As more and more confidential data of vital importance are stored at the organizations, security aspects became essential. The increasing appearance of computer crimes stress the need to handle the risks and threats more formally. The main problem is, that no system could be made absolutely secure. What we can do is to recognize the possible threats and to decide if it is worth to have sufficient countermeasures or it is not.

In the UK CRAMM (CCTA's Risk Analysis and Management Method) is a preferred method for UK Government [CCTA 1991c]. It is widely used by Australian and New Zealand governments.

Roots of CRAMM. CRAMM was developed BIS Information Systems. CRAMM is owned now by CCTA.

CRAMM framework. CRAMM consists of three stages. Firstly, it identifies the assets, threats and vulnerabilities. Both the technical and the non-technical aspects of security are covered. Secondly, it recognizes the risks and offers countermeasures. Management can decide, if they see enough justification to implement specific countermeasures or not. At the end of each stage there is a management review meeting.

In Stage 1, the main task is to do a data, software and physical asset valuation. It is done by qualitative techniques on a scale 1 to 10. The techniques incorporate absolute figures in order to assure that the measurement is not subjective. Data and software assets are valued in different terms. Data gathering is done via interviews. In Stage 2, a threat and vulnerability assessment is to be done. Generic types of threats are defined as hardware and software malfunctions, accidental threats (eg. fire, disaster, staff shortage etc.) and deliberate threats (willful damage, infiltration etc.). The assessment is done by using questionnaires. These are available from the CRAMM supporting software. Having scaled the threat, vulnerability and asset values, measures of the risks to the system are calculated. In Stage 3, countermeasure selection is done. Based upon the identified risk values, specific countermeasures could be selected. These are presented together to the management with a prioritisation scheme and a 'what-if' scenario exploration. Management can then decide if they do want a specific countermeasure to be implemented.

CRAMM techniques. The most used technique is interviewing. For each threat there is a predefined questionnaire, and predefined roles for the interview.

Computer support. There is a software package called CRAMM itself, owned by CCTA. A lot of the technical details of CRAMM (like questionnaires, countermeasures etc.) are published via this software. That is, without the software the method can not be used.

6. Lessons learned

By the example of SSADM we have seen how a methodology can be a successful and accepted standard. If we want the IT community to accept the importance of the above described new areas, one can see the possibility to use the same approach as in the UK was done is the case of SSADM. So what do we need for the new areas ?

Solid foundation. The techniques of the methodology must be well understood and accepted. (Entity-Relationship Modelling and Relational Data Analysis are good examples in SSADM). The usage of such well-founded techniques encourages the users of the methodology. The structural framework and the products of the process must be also well defined.

Driving force. It is not easy to get people to use a methodology. There must be a true force behind it, but this force must be a competent one.

Strong infrastructure. Infrastructural background must support the maintenance and the education of the methodology. There must be an owner of the methodology which forces the evolution of the methodology. The feedback from the users of the methodology have to be taken into mind. The education of the methodology must centrally and periodically evaluated, in order to ensure the required quality of the education. If it is possible, software supporting tools must be developed for the methodology. These have to be also periodically evaluated.

7. Conclusions

Looking at IT from the user's point of view (that is, from the business view) has popped up new aspects. For these aspects, the technique of applying a structured methodology seems to be fruitful. This tendency is going on, the methods and methodologies are being defined (and refined). However, the industrial take-up of the different techniques and methodologies is still slow. In order to achieve better quality software products, policy makers must accelerate this process. The users of IT services must realize their need to force their suppliers to use such methodologies, and the suppliers must be supported with strong infrastructure of that methodologies.

There are other important aspects that surely will be involved in this process. These include package selection, systems maintenance and facility management. For these aspects methods and techniques are emerging, but they have not been stabilized yet.

CASE providers must have a close look at the new areas. These areas are candidates where to give support to the application software providers. For software systems houses it is vital to deal with these areas in order to keep their competitive position. As a consequence, the structured paradigm of systems building is widening and changing.

8. Bibliography and References

1. Asworth, C. Goodland, M. *SSADM: A Practical Approach,* McGraw-Hill Book Company, 1989
2. Booch, Grady, *Object-Oriented Design*, The Benjamin /Cummings Publishing Company, Inc. (1991)
3. Cameron, J.R., *JSP and JSD: The Jackson Approach to Software Development*, IEEE Comput. Soc., (1983)
4. CCTA, *IT Infrastructure Library,* HMSO 1990
5. CCTA, *The Information Systems Guides, Management and Planning Set,* John Wiley and Sons Ltd., (1991a)
6. CCTA, *PRINCE. Structured Project Management,* NCC Blackwell Ltd., (1991b).
7. CCTA, *CRAMM User's Guide Version 2.0, CRAMM Management Guide.* (1991c)
8. Euromethod Phase 1/3, *Introduction and Progress Report*
9. Eva, M., *SSADM Version 4: A User's Guide*, McGraw-Hill, (1992).
10. Hewett, J., Durham, T., *CASE: The Next Step*, Ovum Ltd., (1989)
11. Jackson, M.A., *System Development*, Englewood Cliffs, Prentice Hall, (1982).
12. LBMS, *LBMS Strategic Planning for Information Technology* 1992
13. Layzell, P., Loucopoulus, P., *Systems Analysis and Development,* (3rd edition), Chartwell-Bratt, (1989).
14. Matheron, J.P., *Comprendre Merise,* Outils Conceptuels et Organisationnels, Editions EYROLLES, (1990).
15. NCC (National Computing Centre), *SSADM Manual Version 4*, NCC Blackwell, (1990).
16. Noble, F.: *Seven Ways to Develop Office Systems: A Managerial Comparison of Office System Development Methodologies,* The Computer Journal, Vol. 34, No. 2. 1991 April
17. OECD: *Software Engineering: The Policy Challenge,* Computer Communications Policy, 1991.
18. Page-Jones, Meilir., *The Practical Guide to Structured Systems Design,* 2nd edition, Englewood Cliffs, N.J.: Prentice-Hall, (1988).
19. Rock-Evans, R., Engeline, B.: *Analysis Techniques for CASE: a Detailed Evaluation,* Ovum Ltd.(1989).
20. Turner, W. S., Langenhorst, R. P., Hice, G. F., Eilers, H. B., Uijttenbroek, A. A., *SDM system development methodology,* Elsevier Science Publishers B.V. (North-Holland)/Pandata, (1990).
21. Yourdon, E., Constantine, L.L., *Structured Design,* Yourdon Press, (1975).

Exploratory Software Development with Class Libraries

Johannes Sametinger, Alois Stritzinger

Christian Doppler Labor für Software Engineering
Institut für Wirtschaftsinformatik
Johannes Kepler Universität Linz
A-4040 Linz, Austria

Abstract. Software development based on the classical software life-cycle proves inadequate for many ambitious projects. Exploratory software development is an alternative way of building software systems by eliminating deficiencies of the conventional software life cycle. Instead of exactly defining the various phases of the life cycle, exploratory software development takes small development steps, whereby a single step results in an extension or an improvement of the existing system.

The object-oriented programming paradigm has resulted in increased reuse of existing software components. Therefore, class libraries will become very important in the near future. Exploratory software development is very well suited to this situation and thus provides a major step forward in economically developing software systems.

In this paper we depict deficiencies of the classical software life cycle, present the exploratory software development strategy, and especially illustrate exploratory software development in combination with the reuse of class libraries.

1 Classical Software Life Cycle

Software is usually developed according to the classical software life-cycle. Various models for this life cycle do exist, but basically they are very similar (see [Boehm79, Pomberger 86]). According to the software life cycle the software development process is divided in well-defined phases. In general, each phase has to be finished before the next one can be started (see Fig. 1). This enforces a linear process, which implies that executable programs are available very late. Therefore, any misunderstandings between customers and developers remain hidden for a long time. Besides, any technical problems (e.g., an inefficient file system) cannot be perceived before the test phase. Usually modifications becoming necessary are very costly because they are so late.

The classical software life cycle presupposes static requirements and does not deal with incomplete and inconsistent specifications. For given and static specifications, software developers have to deliver a tailor-made design and a corresponding implementation. The better the implemented program fulfills the given requirements, the better was the work of the

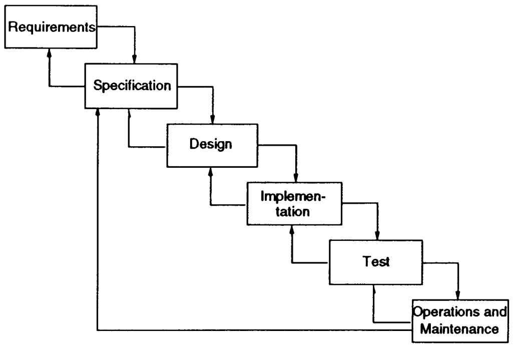


Fig. 1: Classical Software Life cycle

software developers. This approach is in contradiction to reality, because past experience has shown that programs need to be continuously modified and extended. This results in thousands of programmers being engaged with adapting existing software systems to new or changed requirements. Statistics even say that nowadays more time is spent on software maintenance than on software development (see e.g., [Gibson89]). This unsatisfactory situation is partly propagated by the classical software life cycle.

2 Exploratory Software Development

Recently the term *prototyping* has become a buzzword (see [Bischofberger91, Budde84]). The emphasis of prototyping is on the evaluation rather than on long-term use. Software prototypes very often implement the user interface of an application program in order to give potential users an early possibility to evaluate the usefulness and the proper design (of the user interface) of the product. This communication vehicle between developers and customers helps to avoid misunderstandings and usually improves the user interface considerably. However, software prototypes are not restricted to user interface aspects; they can be extended to the finished product step by step.

The term prototyping stems from industry, where prototypes are first models of a certain product. Such prototypes (e.g., cars) are used to investigate certain aspects of a product before it goes into production. As software is simply copied rather than produced in quantity, the term software prototype is somewhat misleading. Besides, this approach can be used not only at the beginning of software development but throughout the whole life cycle. For that reason we prefer the term *exploratory software development.* To begin with, exploratory software development means the production of software to meet the known requirements. Testing the product leads to more requirements and results in modifications and tests to fulfill them. This process is repeated until the developed software system performs satisfactory (see [Sandberg87]). Exploratory software development is a strategy that is best suited when an inherent goal of the project is to identify elusive requirements (specification), to

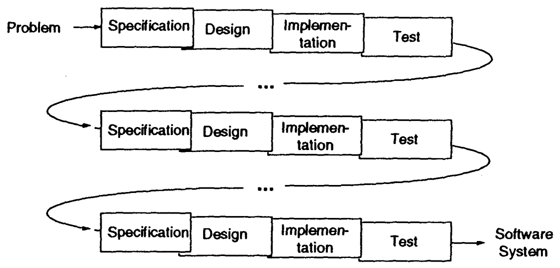


Fig. 2: Exploratory Software Development

establish a suitable system architecture (design), or to explore possible implementation techniques.

Exploratory software development involves repeatedly applying small steps. Each step results (ideally) in an improvement of the current program version until both the developer and the customer are satisfied with the result. Typically one step lasts several hours or even less (see Fig. 2).

When using exploratory software development, programmers have to work with utmost discipline. For example, extending the functionality of a system before its existing parts have reached a (preliminary) satisfactory condition is inexpedient. Additionally, programmers should be aware of writing all the code in a "quick and dirty" fashion, though sometimes it might be useful to temporarily use "quick and dirty" solutions.

The usefulness of exploratory software development emerges from the lack of alternatives in many situations. Both customers and developers not yet knowing exactly what they really want is a typical development situation. Programmers also might not know how to (best) solve certain (implementation) problems. In these cases it is appropriate to work with experimental versions of the software system. By experimenting both customers and developers can gain new insights into their problem domains and thus come closer to better solutions.

Another important justification for using exploratory software development is the increase in complexity of today's software systems. High complexity makes it impossible for human reasoning to deal with all the problems in a linear way, as the classical software life cycle proposes.

Software can best be developed in an exploratory way whenever one or more of the following conditions hold:

- The specification is very vague. Customers are unable to clearly specify their wishes and needs.
- Critical design decisions cannot be made based on theoretical considerations.

- Software developers do not have enough experience with the implementation of similar systems (and the system to be developed is sufficiently complex).
- Programmers do not have (enough) experience in using the programming language or library. (It is impossible to gain familiarity with a class library without experimenting.)
- The system to be developed is too complex and too ambitious to be built linearly.

In our opinion, about half of all projects satisfy one or more of the conditions mentioned above and thus are candidates for exploratory development. The main advantages of exploratory software development are:

- Experimental program versions are excellent vehicles for communication among developers and customers.
- The exploratory approach reduces risks because typically problems are perceived earlier than in the classical software life cycle.
- Stepwise developed programs are better structured and better suited for modifications and extensions because programmers are forced to permanently modify and extend the current version of the software system to be developed. This encourages and trains programmers to write better modifiable code.
- As modifying the system is part of the work being permanently done, it is easier to take new ideas into consideration. The statement: "The next time I would try a wholly different approach!" is more seldom among exploratory programmers.
- Programmers are strongly motivated by working on an executable program rather than writing specifications and design papers for a long time without having an executable program.

Unfortunately, there are also some disadvantages:

- Exploratory development in large teams is possible only when the software system can be clearly separated into various parts.
- It is difficult to estimate the duration and the costs of a certain project. New estimation methods have to be found for this purpose.
- Programmers have to be well trained and to work with discipline. This is extremely necessary in exploratory software development because otherwise the resulting programs are not easily modified ore extended.
- Documentation gets lost in the shuffle.
- Version control and backtracking need to be supported (by tools).

In commercial software projects these disadvantages may be too hard. In order to get estimates of the cost and the duration of a project, we suggest making a rudimentary specification and an initial design according to the classical software life cycle and applying the exploratory approach in the next steps only. This makes it possible to divide a project into small and easily surveyed parts that can be processed by small programming teams.

3 Reusable Class Libraries and Application Frameworks

Conventional libraries, toolboxes, drawing routines, etc. offer fixed functionality at a higher abstraction level than bare programming languages. In the design of the software system the designers have to consider the interfaces of the given components carefully and have to use the provided functions in an appropriate manner. Usually it is not a major problem to build a system upon such libraries when their functions and components are not strongly interrelated. This holds for simple user interface components, data containers, and mathematical and graphical operations.

When working with application frameworks, which define the core structure of the overall application, the designers cannot develop an architecture top-down. In this case the architecture is already predefined to a certain degree by the set of related framework classes which anticipate very early design decisions. The job of the designers is to append the application-specific functionality at appropriate places in the framework. The more powerful and extensive the framework is, the more design decisions are already anticipated in the provided classes.

Commercial applications usually do not use domain-specific interaction techniques or sophisticated algorithms. For such applications classical design methods become superfluous. Although complex software systems could never be designed by means of applying classical techniques and methods such as stepwise refinement or the Jackson System Development Method (see [Cameron89]) alone, application frameworks make these aids less useful. This does not imply that classical techniques will become obsolete as a consequence of frameworks, but their use will be restricted to certain domain-specific components.

Another drawback of classical design methods stems from the fact that applications made from frameworks are implemented in an object-oriented way. Object-oriented systems cannot be designed adequately by means of classical methods. A considerable number of software engineering scientists see the need for a new or modified design method to overcome the current dilemma. A rapidly increasing flood of articles and books about object-oriented design methods, e.g., [Booch86, Coad90, Rumbaugh91], mirrors the expectations of the unhappy software industry.

4 Exploratory Development Approach with Class Libraries

Powerful and well-structured class libraries are a crucial advantage for exploratory software development. The quality and extent of the library used are often more important than the power of the programming language or the development tools.

The exploratory approach has proven its excellence particularly in the development of highly interactive applications with graphical user interfaces. Below we will describe the various tasks that are typical in exploratory software development with class libraries. In general these tasks are seldom completed at once. Usually one does just a portion of a certain task; the next step is taken at the next iteration of the cycle. Furthermore, one should keep in mind that not everything can be done right the first time. But even when information is missing to make a sound design decision, one should not hesitate too much. Experimentation and exploration often lead to better solutions than intense analytical studies. The steps of the exploratory development approach are as follows (see also [Stritzinger92]):

Step 1:

Start with the design of the user interface in a prototyping-oriented way. Concentrate on the essentials first. Whenever some parts of the interface are unclear, try a rudimentary design.

Step 2:

Try to identify classes for the implementation of the user interface components. An extensive class library should offer a lot of support in this respect. Typical classes include: Window, Menu, View, TextView, ListView, and control elements like Button and Scrollbar. If you cannot find exactly what you are looking for, search for classes that already implement part of the desired functionality. Inheriting is most often cheaper than implementing.

Step 3:

Try to identify classes that describe important objects in your problem domain. These classes often correspond to object categories of the real world (employee, car, etc.). Although it is not as likely as with user interface classes, there is still a chance to find classes in the library from which you can inherit. If objects in your program have a close correspondence to real-world objects, slight changes in the real world will just cause slight changes in the program. All objects that describe application-specific data should be connected somehow. This complex object web is usually called the *model*. Relationships among model objects can either be established by application-specific compound objects (faculty, assemblyLine, etc.) or by general-purpose collection objects (queue, tree, etc.). The whole model should be accessible by a single (or a small number of) reference(s). If there are objects that share a lot of commonalties but differ in some respects, the commonalties should be described collectively (factored into a common superclass). In many cases abstract classes are rather useful. Abstract classes (e.g., GraphicShape) are classes that do not have instances; they just serve for factoring commonalties out of their subclasses. The more complex the problem is, the more imaginary classes have to be invented. Finding appropriate imaginary classes is a very difficult job that requires some experience. Fortunately, you can find such classes incrementally.

Step 4:

Identify relevant object states for all classes. Object attributes that carry state information are (usually) modeled as instance variables of the corresponding class. Redundancy among instance variables should be avoided.

Step 5:

Think about the messages (operations) your objects should respond to. Each instance variable has to be addressed; i.e., each variable must get a value and must be accessible somehow. The semantics of each message should be clearly describable. Messages should be as powerful as possible, but as flexible as necessary.

Step 6:

Implement a method for each message. Do not duplicate code from superclasses; send messages to invoke the overridden method instead. Extensive methods should be split into several, possibly private methods.

The above steps are often performed in a non-sequential way. For instance, it may happen that while implementing a method the need for an additional instance variable arises. Simultaneous development of various small life cycles is typical for the reuse of class libraries and is also called a *cluster model* (see [Meyer88], [Pree91]).

It is always advisable to define classes somewhat more generally than actually necessary. Modifying and extending existing code is typical in the exploratory approach. The more general classes are, the less widespread is the impact of changes and extensions.

5 Conclusion and Outlook

In summary, we claim that an exploratory, object-oriented development approach together with application frameworks is the most productive way to develop highly interactive applications with high quality standards. The problems in designing complex systems are rather a symptom of an insufficient strategy than a lack of methods. Innovative and sophisticated software systems can never be developed in a linear process of applying recipes. Similar to other high-tech products, knowledge, skills, experience and motivation play a crucial role in the successful realization of ideas.

In our opinion, one of the strongest drawbacks of object-oriented software development is the huge complexity of many widespread class libraries and application frameworks. This complexity, together with the manifold structuring options of object-oriented programming, make extremely high demands on programmers – even with an exploratory approach. Many programmers in the field are unable to take advantage of these powerful techniques. Therefore software engineering experts are called upon to develop tools that permit less experienced programmers to utilize the advantages of object-oriented programming with class libraries by helping to master the complexity and by supporting the comprehension process (see [Sametinger90] for an example).

A first step in the right direction is so-called interface builders. By means of interface builders construction of complex user interfaces can be done in a simple, interactive way by directly manipulating interface components. 4th generation systems form another possibility for a quick development of applications at a high level of abstraction. The drawback of 4th generation systems is often the connection between user interface and database, which usually have to be programmed with a rather conventional programming language. The developers are confronted with a huge gap in the abstraction level whenever the built-in functionality is not sufficient. Furthermore, only a minority of contemporary 4th generation systems are based on the object-oriented paradigm.

The goal of a thoroughly seamless development process at a very high abstraction level could be reached by a kind of tool (or tool set) which could be called *application builder* or

5th generation system. Such a system should support interactive, graphical construction of user interfaces and (external and internal) data models. In addition, a 5th generation system should offer the opportunity to combine predefined, reusable and user-defined building blocks in a comfortable, yet flexible and preferably visual way.

Unfortunately, such 5th generation systems are not available yet. But there is a good chance that mechanisms and tools will be developed which can fulfill the goal of a thoroughly seamless development process at a high abstraction level. Then object-oriented programming with extensive libraries will become a widespread technology available to almost everybody.

6 References

1. Bischofberger W., Kolb D., Pomberger G., Pree W., Schlemm H.: Prototyping-Oriented Software Development - Concepts and Tools, Structured Programming, Vol. 12, No. 1, New York, 1991
2. Boehm B., W.: Software Engineering, in Classics in Software Engineering, Yourdon N.E. Editor, pp. 325-361, Yourdon Press, 1979.
3. Booch G.: Object-Oriented Development, IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986.
4. Budde R., et al (Editors): Approaches to Prototyping, Springer-Verlag, 1984.
5. Cameron J.: JSP & JSD: The Jackson Approach to Software Development, IEEE Computer Society Press, 1989.
6. Coad P., Yourdon E.; Object-Oriented Analysis, Yourdon Press Computing Series, Prentice Hall, 1990.
7. Gibson V.R., Senn J.A.: System Structure and Software Maintenance Performance, Communications of the ACM, Vol. 32, No. 3, pp. 347-358, 1989.
8. Meyer B.: Object-Oriented Software Construction, Prentice Hall, 1988.
9. Pomberger G.: Software Engineering and Modula-2, Prentice Hall, 1986.
10. Pree W.: Object-Oriented Software Development Based on Clusters: Concepts, Consequences and Examples, TOOLs Pacific (Technology of Object-Oriented Languages and Systems), pp. 111-117, 1991.
11. Rumbaugh J., et al: Object-Oriented Modeling and Design, Prentice Hall, 1991.
12. Sametinger J.: A Tool for the Maintenance of C++ Programs, Proceedings of the Conference on Software Maintenance, San Diego, CA, pp. 54-59, 1990.
13. Sandberg D.W.: Smalltalk and Exploratory Programming, ACM Sigplan Notices, Vol. 22, No. 10, 1987.
14. Stritzinger A.: Reusable Software Components and Application Frameworks—Concepts, Design Principles and Implications, to be published in VWGÖ, Vienna, 1992.

SOFTWARE PROCESS IMPROVEMENT BY MEASUREMENT BOOTSTRAP/ESPRIT PROJECT 5441

Volkmar Haase
Graz University of Technology
Graz, Austria

Richard Messnarz
Graz University of Technology
Graz, Austria

Robert M. Cachia
Etnoteam SpA
Milan, Italy

Abstract. A new paradigm in software engineering claims that the quality of a product is highly impacted by the quality of the process which gives rise to it. To reduce the risk to product and project we therefore seek to quantify the quality of the development process. Software process measurement represents an evaluation of all the management activities, methods, and technologies that are employed to develop a software product. BOOTSTRAP developed a method to determine the profile of a Software Producing Unit (SPU) showing its strengths and weaknesses. This paper is intended to illustrate a methodology of software process measurement and will present some sample results.

1 Introduction

An SPU (Software Producing Unit) is a software producing company of small or medium size or a department in a large company in which projects are performed to develop software products. An SPU consists of projects that are software producing entities, and an organization and management built around these projects. A project is an entity within an SPU which has a well-defined goal and has to exploit the resources provided by the SPU to develop a certain software product according to a time schedule.
About 7 years ago the US DoD (Department of Defence) began to assess the development process of its contractors. Since then only contractors with a software process of high quality have been awarded further contracts. These SCEs (Software Capability Evaluations) have been performed by the SEI (Software Engineering Institute) [BOL91]. In addition teams of software development organizations wanting to develop software for the DoD have been taught by the SEI how to perform software process assessments. Software process assessments are based on a questionnaire which contains nearly the same questions as those used for SCEs. These assessments help to identify the key strengths and problems of an SPU, and to create action plans to improve the software process, so that the SPU has better chances of doing well at an SCE and getting a contract [BOL91, HUM91, HUM91a].
The SEI model differentiates between 5 different maturity levels of Software Producing Units [ESP91, HUM89, HUM91a, PAU91].

Level 1: Initial Process
Level 2: Repeatable Process
Level 3: Defined Process
Level 4: Managed Process
Level 5: Optimizing Process

A level between 2 and 5 is assigned to every question of the questionnaire. After the evaluation of the questionnaire it is possible to identify on which level of maturity the SPU is located.
BOOTSTRAP adopted and extended the SEI questionnaire and adapted it to the European software industry including non-defence. Further we developed an improved evaluation method to calculate the maturity level of an SPU [ESP91, HAS91].

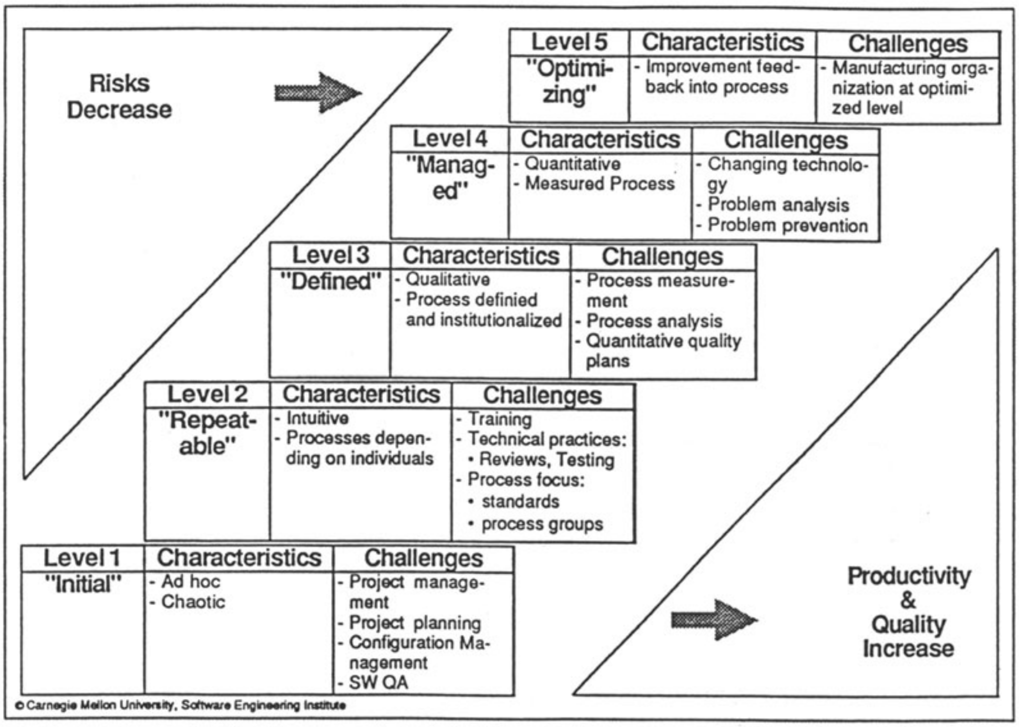

Risks Decrease
Level 5 "Optimi-zing"
Characteristics - Improvement feed-back into process
Challenges - Manufacturing orga-nization at optimi-zed level
Level 4 "Manag-ed"
Characteristics - Quantitative - Measured Process
Challenges - Changing technolo-gy - Problem analysis - Problem prevention
Level 3 "Defined"
Characteristics - Qualitative - Process definied and institutionalized
Challenges - Process measure-ment - Process analysis - Quantitative quality plans
Level 2 "Repeat-able"
Characteristics - Intuitive - Processes depen-ding on individuals
Challenges - Training - Technical practices: • Reviews, Testing - Process focus: • standards • process groups
Level 1 "Initial"
Characteristics - Ad hoc - Chaotic
Challenges - Project manage-ment - Project planning - Configuration Ma-nagement - SW QA
Productivity & Quality Increase
© Carnegie Mellon University, Software Engineering Institute

Fig. 1: Maturity levels according to the SEI

2 BOOTSTRAP's Approach

BOOTSTRAP attempts to identify all individual attributes of a software development organization or individual project and assigns all questions to process quality attributes as well as levels. It is not only possible to calculate the maturity level of an SPU or a project but also the attainment on a particular process quality attribute.
The SEI questionnaire initially only allowed a question to be answered by yes or no (black/white) [BOL91]. BOOTSTRAP, seeking to obtain more detailed and precise results, differentiates between 1 (0 percent / weak or absent), 2 (33 percent / basic or fair), 3 (66 percent / significant or strong), 4 (100 percent / extensive or complete). A maximum deviation of 0.5 on a discrete scale of 1 to 4 for the evaluation of a question corresponds to a deviation of approximately 17% on a percentage scale. This maximum deviation would be 50% in case of a yes/no scale. As in the real world, a process is seldom in a 0% or 100% state, a 4 point scale seems to be more precise and technically more sound.
BOOTSTRAP is not only based on the SEI model but also on the ISO standard 9000-3 [ISO87, ESP91] for quality assurance and quality management and on the ESA-PSS 005

standard [ESA91, ESP91] for the software life cycle. We also take into account some key aspects of the spiral model like risk management and prototyping. This, for example, has lead to a further process quality attribute entitled Risk Avoidance and Management.(see Fig. 2). Initially we used weighted questions, because some questions seemed more important than others. After evaluating about 30 SPUs and 60 projects we found that evaluations based on weighted scores did not differ significantly from those based on non-weighted scores. For 86 percent of the evaluations the difference between weighted and non-weighted satisfaction percentage was equal or lower than 5 percent for each level. The maximum difference observed was 8 percent. There is a high correlation between weighted and non weighted satisfaction percentages. The correlation coefficient, which we derived from the comparison between weighted and non-weighted satisfaction percentages, is 0.98 for level 2, 0.9943 for level 3, and 0.9842 for level 4. Moreover it is quite difficult to select a weight set sensible for all situations. Thus the notion of question weights does not seem very useful.

Risk Avoidance and Management

Question Nr.	Assigned Level	Text
2219	3	Existence of requirements to identify, assess and document risks to project and product associated with modifying Software Life Cycle (SLC) or Non-SLC activities
2220	2	Existence of a requirement for identifying the parts of a specification more likely to show instability
2221	2	Existence of guidelines for taking into account, at high level design phases, the possible instability in parts of the specification
Answers:	absent / basic / significant / extensive	

Fig. 2: Sample questions of BOOTSTRAP's questionnaire

We have a 4 point scale for the evaluation of a question and if we assume that a question can be evaluated with a maximum deviation of d = 0.5, the following standard deviation can be derived:

$$\text{Sigma} = d * \text{SQRT}(Nq) \quad (1)$$

Nq ... Number of questions

Sigma ... Standard deviation from the total score for Nq questions based on the assumption that scores might be given for a certain question with a deviation of 0.5.

From this standard deviation we can calculate a range for scores:

$$\text{Score\_Low} = (\text{score}[1]+\text{score}[2]+ \dots +\text{score}[Nq]) - \text{Sigma}$$
$$\text{Score\_High} = (\text{score}[1]+\text{score}[2]+ \dots +\text{score}[Nq]) + \text{Sigma}$$

If we calculate the maturity level of an SPU or a project based on the scores Score_Low and Score_High, the maximum difference between ML[Score_Low] and ML[Score_High] will be lower than 0.2 (Nq is over 100 for an SPU or a project).

ML[Score_Low] = Calculated maturity level based on Score_Low scores
ML[Score_High] = Calculated maturity level based on Score_High scores

Thus for the calculated maturity level of an SPU or a project we obtain a standard deviation of approximately 0.1, so that we have selected a scale going up in quarters, from 1.00, 1.25, .. up to 4.50, 4.75, 5.

BOOTSTRAP has separate questionnaires for the assessment of an SPU's quality system (Global Questionnaire) and the assessment of the projects within this SPU (Project Questionnaire) [ESP91]. In the Global Questionnaire we inquire into the recommendation of certain procedures, methods, standards or technologies, whereas in the Project Questionnaire we then ask about their adoption. This means that we check first if an SPU provides all necessary resources and secondly how effectively the projects are using these resources. Hence we can determine whether a project A uses some resources better than a project B and, if so, we can analyze this situation.
BOOTSTRAP emphasizes that organization is most important and that methodology is more important than technology [ESP91, HUM91, PAU91]. A project without organization is nearly certain to result in a disaster. And it does not help to buy a technology when the software engineers either cannot understand the method of the technology or do not accept the underlying methodology. Therefore the technology must be integrated in the existing environment. To be accepted the technology must adapt to the corporate culture, and management has to be sensitive to the need for training to enable the developers to use the methodology and technology effectively.

2.1 Individual Attributes of an SPU According to BOOTSTRAP

ORGANIZATION

- Quality Assurance
- Resource Management
  - Staffing
  - Training

METHODOLOGY

- Process Related Functions
  - Process Description
  - Process Measurement
  - Process Control
- Life Cycle Independent Functions
  - Risk Avoidance & Management
  - Project Management
  - Quality Management
  - Configuration & Change Management
- Life Cycle Functions
  - Development Model
  - Requirements
    - User Requirements
    - Software Requirements
  - Architectural Design
  - Detailed Design
  - Testing
    - Unit Testing
    - Integration Testing
    - Acceptance Testing & Transfer
  - Operation & Maintenance

3 Considerations on Questionnaire Evaluations

A key question when performing assessments is how an SPU-wide, project-wide or attribute-specific maturity level can be calculated from the set of answers obtained during an assessment. Bootstrap has tried to develop an algorithm that produces more reliable results than the SEI one as it is able to take into account the following facts:

The Algorithm Fits the Complexity of Software Engineering. Our metric, which is based on the calculation of steps and a variable scale, is dynamic. This means that any change in the questionnaire automatically leads to a modified scale, which provides the basis for the calculation of the steps. Additionally we get different scales depending on the characteristics of the SPU type, so that our metric always automatically adapts to the current SPU profile. Nevertheless the results remain comparable because they are mapped onto a maturity level scale. We can even calculate the SEI maturity level. Thus we can compare our results with the SEI results, but this is not true the other way round.(see 3.1)

The Algorithm Minimizes the Dependence on Individual Assessors. The SEI algorithm uses key questions which have to be satisfied to fulfill a certain level [BOL91]. BOOTSTRAP does not use single questions but key clusters of questions (key attributes). Quality management, for example, is a key attribute consisting of 7 questions which have to be satisfied with a threshold percentage to fulfill a certain level. Thus we do not only count yes/no answers for important questions, but we look at clusters of about between 4 to 7 questions. That way the BOOTSTRAP algorithm seeks to minimize the dependence on "assessor behaviour" in judging individual questions. Avoiding such "singularities" resulting from strange SPU or project behaviour has been an explicit design objective of this algorithm. The same considerations led to our early choice of a 4 point reply set rather than yes/no. (see rule 3 in 3.3)

The Algorithm Awards Planned Innovation. The SEI algorithm is strictly sequential. Only if level i is satisfied by a minimum of about 80 percent and if nearly all key questions on level i are answered by yes, does the SEI algorithm take into account the scores on level i+1 [BOL91]. This does not award SPUs and projects which plan and stagger innovation over a period of time. Thus for the calculation of the steps we also take into account scores which the SPU or project gained on the next higher level. (see rule 2 in 3.3)

The Algorithm Is Based on Steps. If the evaluation is only based on percentages, you will get equal distances between the levels of the maturity scale, although there are different numbers of questions for each level. In the SEI questionnaire the number of questions decreases as the level increases.

$t[2] > t[3] > t[4] > t[5]$, with $t[i]$ ... total number of questions on level i, for $i = 2..5$

This is due to the fact that only few SPUs on levels 4 and 5 are known and well characterized so far. The experiences we gain from SPUs which are on level 4 will help us to define the full set of questions for checking characteristics of SPUs on levels 4 or 5. For levels 2 and 3 nearly 100% of the questions have already been identified.

From formula (1) we can conclude that the proportion between the standard deviation Sigma and the total score for Nq questions decreases if Nq increases. This means that with an increasing number of questions we obtain more reliable and precise results. To fulfill, for example, 75% on level 2 would mean to answer a lot more questions by yes than on level 4.

Even if we had the same number of questions for each level we would have to take into account that depending on the SPU type (e.g. commercial systems, embedded systems) different numbers of questions might be applicable for each level.

$$d[i] <= t[i], \text{ for } i = 2..5$$

t[i] ... total number of questions on level i
d[i] ... number of applicable questions on level i

BOOTSTRAP has developed an algorithm which uses steps instead of percentages and a scale with variable distances between the levels.
Only if d[1] = d[2] = d[3] = d[4] = d[5], is the calculation of steps equal to the calculation of percentages.

The Algorithm Has Enhanced Evaluation Capabilities. As BOOTSTRAP has two questionnaires, one for the SPU and one for projects, it is able to compare the SPU profile with the profile of the projects [ESP91]. Additionally we have designed an algorithm which cannot only calculate the maturity level of an SPU or a project but also of each individual attribute. (see rule 4 in 3.3)

3.1 Dynamic Scale

The distances d[i] between the levels (see Fig. 3) are defined by the number of applicable questions. There are different distances between the levels because we have different numbers t[i] of questions for each level and due to the size and structure of an SPU type different numbers of questions d[i] <= t[i] might be applicable. So the distances d[2], d[3], .., d[5] are not constant but variable. For an SPU A, for example, 40 questions might be applicable on level 3, whereas for an SPU B 44 questions might be applicable on the same level. Thus we obtain a scale depending on the particular characteristics of the SPU type.

```
I----------I---------------I-----I----I
1    d2    2     d3        3 d4  4 d5 5
```

d[i] ... Number of applicable questions on
level i, for i = 2..5

Fig. 3: Scale According to the Maturity Levels

3.2 Motivation

We can compare the approach of the BOOTSTRAP level algorithm with a mountain, with a number of steps leading from the foot up to the peak. Each step represents one question in the questionnaire. Every SPU tries to master a number of steps to get as close as possible to the peak of the mountain. The foot of the mountain would be level 1 and the peak corresponds to level 5. We calculate the number of steps which the SPU has fulfilled in climbing up the mountain. (<--> Number of steps (questions) which the SPU has satisfied on the way from 1 to 5 on the scale above)

Thus we can identify on which level (or between which levels) the SPU is located. But this is only a first approximation of the appropriate maturity level and is a nominal value to be refined in subsequent steps (see 3.3).

3.3 Description of the Algorithm

A filled in BOOTSTRAP Questionnaire Q is a subset of NxLxS, with L=(2,3,4,5) representing the set of levels, S=(0,1,2,3,4) representing the set of possible scores and N representing the set of question numbers. Each evaluated question is an element of set Q. The maturity level ML is a function which maps Q, or a subset V of Q in case of an individual attribute, onto a value between 1 and 5 on the maturity level scale.

$$ML: V \dashrightarrow [1,5]$$

$$Q \ldots \text{ subset of } N \times L \times S$$

$$V \ldots \text{ subset of } Q$$

The algorithm works in two phases. First the number of steps is calculated regarding the restrictions which are described in the 4 rules listed below. Then the steps are put on the dynamic scale and the steps-value is transformed into a maturity level value.

$$ML(V) = G(F(V))$$

$$F: V \dashrightarrow [0,D],\ D = d[2]+d[3]+d[4]+d[5]$$

$$G: [0,D] \dashrightarrow [1,5]$$

F is a function of all scores given for questions which are elements of V and it calculates the number of the achieved steps:

$$F(V) = F(\text{score}[x1],\text{score}[x2],\ldots,\text{score}[xn]), \text{ with}$$

$$|V|=n, \text{ and } x1,x2,\ldots, xn \ldots \text{ elements of } V$$

$$\text{score}[xj] \ldots \text{ element of } S, \text{ given score for answer } j, \text{ for } j = 1..n$$

The following rules must be followed for the calculation of the number of steps:

1.
If all questions on level i are satisfied by a percentage[i] >= Defined Threshold, we define that level i is fully satisfied.

2.
If an SPU or project is between level i and i+1 after calculating the steps, the calculation has to be based only on the steps achieved on levels 2 to i+2.

3.
To reach the next higher level an SPU or project must satisfy all key attributes on the current level with a certain minimum.

4.
To calculate the maturity level of an SPU or project we need the restrictions of both 2. and 3. To calculate the maturity level of an individual process quality attribute we need the restriction in point 3, only if a defined key attribute is a subset of the process quality attribute.

Thus the BOOTSTRAP level algorithm allows the calculation of the maturity level of individual attributes, yielding synthetic indicators useful in identifying problem areas in the process. It is technically sound to have a global indicator (maturity level) and lower level synthetic indicators computed in the same way.

4 Sample BOOTSTRAP Results

We are reproducing histograms showing the maturity level of an SPU, of its projects, and of the individual attributes both for the SPU and its projects. (see Fig. 4 and 5)
Every Project Questionnaire contains nearly the same individual attributes as the Global Questionnaire. Using the BOOTSTRAP level algorithm we can calculate a characteristic profile for every project as we can calculate one for the SPU. We can then compare the profiles of different projects and the profile of a project with that of the SPU. This enables us to find weak points within an SPU quickly and easily.
The data analysis in Fig. 4 and 5 shows the structure of an SPU XX and one of its projects XX1 and is based on the calculation of appropriate levels for individual attributes according to the BOOTSTRAP level algorithm.

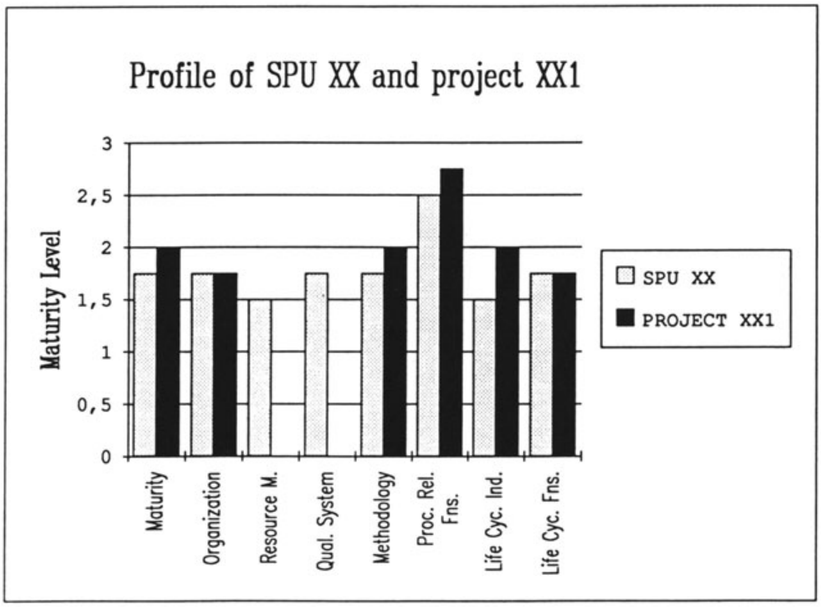


Fig. 4: Overview of the Maturity Levels of SPU XX and Project XX1

4.1 Comments on the Profiles of SPU XX and Project XX1 (see Fig. 4 and Fig. 5)

Resource management is low (1.5) and the quality system (1.75), which provides the basis for quality management, does not seem to work very well. The assessed SPU XX was found to be very weak in project management (1.25), although project management is basically performed at project level (2.00). This suggests that upper management does not recommend the use of project management methods, and nearly nothing has been done to select and refine methods and procedures. This caused the project managers to react by themselves and to develop their own individual methods. Such a situation, however, leads to the problem that all

the projects use different methods, so that project management reviews are very difficult because no standard method is followed.
Further, the means for configuration and change management (2.00) are basically provided by the SPU and they are effectively used in project XX1 (2.75). The SPU XX does not recommend any method for analysis and design of the software system which has to be developed. Thus also project XX1 lacks an effective methodology for requirements specification and architectural design. Concerning detailed design project XX1 does not use all of the available resources and we have to check if this is caused by the project typology or if these resources could be used more effectively. The SPU recommends the use of a development model which is followed at project level. Quality management, testing, and maintenance are equally weak in the SPU XX as well as in project XX1.

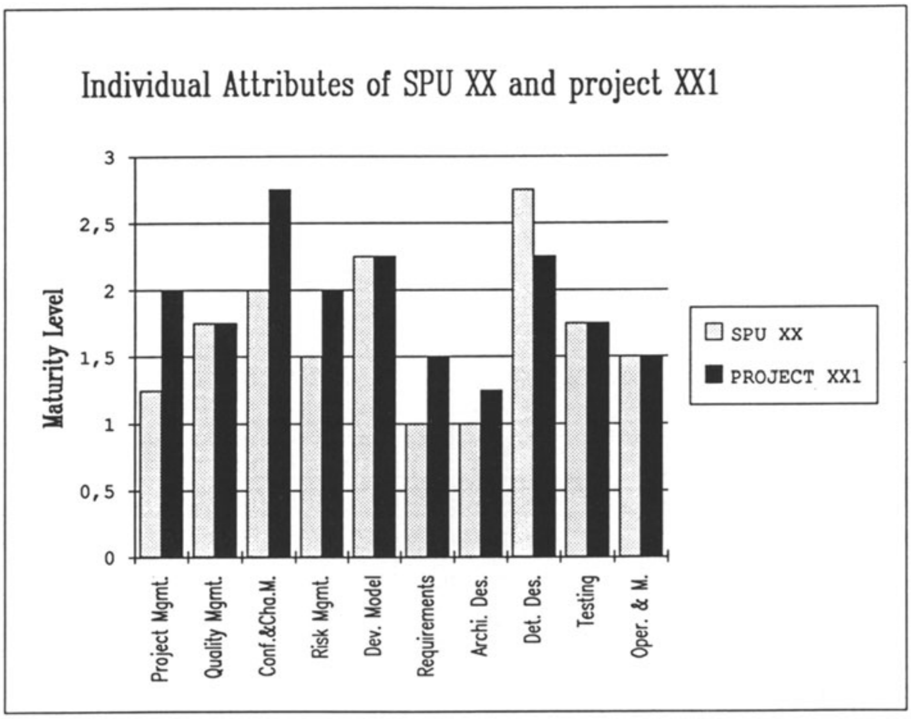


Fig. 5: Individual Attributes of Methodology of SPU XX and Project XX1

5 BOOTSTRAP's Future

BOOTSTRAP's long term task is to perform assessments all over Europe and determine a profile of the European software industry. We want to identify its key strengths and weaknesses. The profile of every assessed SPU can be compared with average values of appropriate subsets (e.g. similar size, same branch) of all European profiles. Thus we can determine the position of an SPU within the European market.
The SEI promotes the use of a SEPG (Software Engineering Process Group) [ESP91, HUM89, HUM91a, PAU91] which serves as the focal point of process improvement, performs assessments, and creates action plans to improve an existing environment. It also establishes standards and procedures [HAS91]. BOOTSTRAP's assessment activities can be seen as one possible instance of an international SEPG in the European context.

Acknowledgments

This work has been partly funded by the CEC commission under the ESPRIT project 5441, BOOTSTRAP. We would like to thank the commission and we are also grateful to the support of all the partners of the BOOTSTRAP consortium.

References

[BOL91] T.B. Bollinger , C. McGowan : A Critical Look at Software Capability Evaluations.IEEE Software, July 1991, pp. 25-41

[ESA91] ESA Board for Software Standardization and Control: ESA Software Engineering Standards, European Space Agency, Paris, France: 1991

[ESP91] ESPRIT Project 5441 BOOTSTRAP: Phase I Interim Report.Composite Deliverable 7, Commission for European Communities (CEC), July 1991

[HAS91] V.Haase, R. Messnarz: A Survey Study on Approaches in Technology Transfer, Software Management and Organization. Report 305, Institutes for Information Processing Graz, June 1991

[HUM91] Humphrey W.S., Bill Curtis : Comments on 'A Critical Look'. IEEE Software, July 1991, pp.42-46

[HUM89] Humphrey W.S.: Managing the Software Process, 494p., SEI Software Engineering Institute , New York , Amsterdam , Bonn , Madrid , Tokyo : Addison - Wesley Publishing Company 1989

[HUM91a]Humphrey W.S. , T.R. Snyder, R.R. Willis: Software Process Improvement at Hughes Aircraft. IEEE Software, July 1991, pp. 11-23

[ISO87] ISO 9000-3 : European Standard for Quality Management and Quality Assurance, European Committee for Standardization, Bruxelles: 1987

[PAU91] M.C. Paulk , B. Curtis , M.B. Chrissis : Capability Maturity Model for Software, Software Engineering Institute , Carnegie Mellon University, Pittsburgh, August 1991

ARTIFICIAL INTELLIGENCE - MODELLING ASPECTS

Chair: V. Haase

A framework for reconciliation of the meta-structure of repositories and structured methodologies

Beyond Software Engineering

Bálint Molnár

Information Technology Foundation of Hungarian Academy of Sciences
H-1525 Budapest 114. P.O.B. 49, Hungary,
Telephone: +36 1 169-9499, Fax: +36 1 155-3376
e-mail: h4445mol@ella.hu

Abstract. In this paper, an attempt is presented in order to map the widely-accepted meta-structure of information resource dictionary systems (repositories) and the meta-structure of structured methodologies for systems analysis and design on the field of very large information systems. An effort is made to map these two architectures on each other using the object-oriented principles creating a theoretical framework, furthermore the applicability of object-oriented approach is investigated.
The aim of this research is twofold (1) to help understand the design process of very large information systems (VLIS) and (2) to create a base to analyze the problem solving activities.

1. Introduction

There are some standardized repository or Information Resource Dictionary System (e.g. ANSI IRDS [IRDS 1988, IRDS 1988b], ISO IRDS [IRDS 1990, IRDS 1990b], IBM AD/Cycle [IBM 1989], DEC ATIS, etc), some of them are defined by international standardization bodies, some of them are defined by huge organizations as a *de facto* standard. There are some similarities among them due to the early standardization efforts in their structures and in the applied concepts. The theoretical skeleton of these definitions provides a good opportunity to use their vocabulary of notions in practice even in the case when tools in a certain environment do not cling to one of the standards entirely.
The definition of repositories does not imply any particular methodologies but the conçeptual structure can be used to arrange the entities and objects of a certain information systems design methodology in this framework.
There are some comprehensive meta-model of information system development methodologies [Hesse 1988] and there do exist meta-models for the single, concrete methodologies as well.
In this paper, the following theses might have to be proven:

- The structure of the repositories and meta-models of methodologies are orthogonal and this property can be exploited to reconcile these two viewpoints in a practically useful framework.
- The object-oriented approach is quiet useful but enforcing it on the meta-modelling of methodologies is not beneficial if it means to drop out the immanent dichotomy of the different system viewpoints in order to represent the universe of

discourse more smoothly and apparently with fewer conflicts.
The rapidly changing technology during long projects coerces that the employed tools for analysis, design and implementation have to be replaced with a more advanced one and even the methodology might have to be improved or enhanced either evolutionary or revolutionary way. However, if we have a sound base for representing and interpreting of the collected information in a meta-model, the right place for the pieces of information can be found more easily in a slightly or drastically changed environment so in spite of alteration in circumstances the project can be adapted to the new situation. Some project management methods can be considered as product-oriented (e.g. PRINCE [CCTA 1991]) so several entities of the meta-model as deliverables or products of a certain project have well-defined places in the meta-model and can be dealt with the project management in the same framework, this makes easier the adaptation process to changes caused by either the project or the methodology or the tool and environment.

The meta-modelling of the process of very large information systems may assist to build interfaces based on the theoretical framework to the various CASE tools utilizing the standard IRDS structure as a solid foundation. If we can identify the problem solving activities and associate them to generic task concept in this context [Chandrasekaran 1988] the software engineering or information engineering activities can be supported by artificial intelligence techniques and algorithms.

2. An overview about the Information Resource Dictionary Systems (IRDS)

There were some standardization efforts to define firstly an advanced data dictionary, later repository or recently Information Resource Dictionary System (IRDS). There is an ANSI standard [IRD 1988], there is an ISO standard [IRDS 1990] and there exists the IBM AD/Cycle Repository [IBM 1989]. But in fact, the tool vendors give lip service to the standards, most of them have a strategy to conform with one of them with added value. The users of the CASE (Computer Aided Software Engineering), I-CASE (Integrated CASE) or IPSE (Integrated Project Support Environment) [Gane 1990] face with a future with two or even three industry standards and a fair number of non-standard but technically advanced products which might apply object-oriented technology, object-oriented or entity-relationship database management systems, perhaps some expert database technology.
The premature endeavors to standardize the repositories have merit, the differences between the ANSI, ISO and IBM are matters of detail, not of basic notions.
We can conclude the flexibility and customizability can be considered as an important feature, the repository concept is proliferated in the industry among product vendors so it will become relatively easy to adjust a given project to the minor differences between two or even three realization of that concept.

2.1 The most important properties of repositories

In the following sections, the significant properties of the repositories are overviewed.
The ANSI and ISO standard distinguishes four levels:

- IRD Schema Description - IRD Definition Schema Level
- IRD Schema - IRD Definition Level
- IRD (the Dictionary) - IRD Level
- "Real World" Instances - Application Level

These levels can be formed into a table in the following way [Sibley 1986] :

Level	Data Category	Name in the IRDS
0	Data	--
1	Meta-Data	IRDS Database
2	Meta-Meta-Data	IRDS Schema
3	Meta-Meta-Meta-Data	IRDS Meta-Schema

The level 1, the meta-data consist of entities, relationships and attributes, these form the dictionary database.
The level 2, meta-meta-data are meta-entities, meta-relationships and meta-attributes thus every type of component (entity, relationship, attribute) of the level 1 appears as a meta-entity, i.e., entity-type, relationship-type, attribute-type. The meta-relationships and meta-attributes can be used to describe the structure of the repository and can be employed to define rules, syntactic and semantic checkings to be imposed on the level 1 components. Hence, the permissible repository structure can be determined by meta-relationships. The meta-entity 'generic_procedure' or 'heuristic_rules' can realize the various constraint on the level 1 components (e.g. referential integrity constraint in a relational database system [Date 1981].
The level 3, meta-meta-meta-data termed as the meta-schema level is not yet implemented in commercial repositories, as far as we know. Nevertheless, in principle the level 3 components would be the types of the existing level 2 components.
The meta-meta-entities at the level 3 comprise the meta-entities, meta-relationships and meta-attributes at level 2. For information resource and life cycle management, some meta-meta-entities might be defined as attribute-validation-procedure, attribute-validation-data, life-cycle-stage-name, life-cycle-status-name [Goldfine 1985].
The ISO and ANSI approaches are very close to each other but the ISO is more SQL-oriented, i.e, the ISO strategy is to specify an IRDS that could be accessed through SQL. With the ANSI IRDS, the SQL is a potential implementation tool.
The AD/Cycle Repository manager distinguishes two domains - (1) specification, (2) run-time services.
The specification domain is further divided into three views:

- the conceptual view
- the storage view
- the logical view

The conceptual view operates with similar concepts as the IRDS four level model, supports the following modelling components [IBM 1989, Maciaszek 1991].

- *entity*
- *conventional relationship* with some limitations
- *is-attribute-of* relationships
- *is-part-of* relationships
- *is-a* relationships
- *is-constraint-on* and *is-heuristic-of*

3. A four level meta-model for information system development

In order to understand the development process of business application systems, several models are created to illuminate the various sides of the information modelling process [Hesse 1988, Brodie 1982, Essink 1986]. The modelling of a concrete information system is carried out by various methodologies, the most developed ones try to comprehensively perceive the diverse aspects of business areas [Cameron 1983, Eva 1992, Jackson 1982, Longworth 1986, Matheron 1990, Turner 1990, Yourdon 1975].
Meta-models are used to depict the information systems development process in a comprehensive and concise way and to structure the products and deliverables of a project

and the related managing and controlling activities. A specific meta-model exists in the context of a project, an application domain and a methodology, furthermore it determines the basic concepts, vocabulary and terminology, relationships and organizing rules for the given situation.

3.1 A four-level meta-model

In this section, a brief overview will be given about a meta-model [Hesse 1988]. The original version of the meta-model uses the term 'level' so we do not want to deviate from it but we think of them as the various aspects or sides of the information system modelling.
The user level of the meta-model attempts to describe the universe of discourse, the application domain in terms of users. There is only one category and one relationship type in this model, namely, *user concept* and *refers to*.
The functional design level reflects the system analyst/designer viewpoint. The application of semi-formal description techniques (data flow diagrams, entity-relationship data modelling), the counterpoint of the active and passive elements of the application, i.e., the dichotomy of the data model and the function model [Brodie 1982] are the characteristic of this level.
The technical design level deals with the transformation of the functional design into the predefined technical system architecture, with the synthesis of the result of the analysis. This corresponds to the logical/physical design phase in the project life cycle. In this context, the terms 'building blocks' of the system can be used which consist of 'data types' and 'operations' or 'logical database transactions'. Object-orientation is suitable for that level, data are encapsulated in the building blocks and manipulated by operations which have exclusive access to the data. The interfaces to other building blocks are realized through those operations that can be accessed from outside. The Ada language [Pyle 1984] support such features but there are several other languages.
The implementation level takes care of programming, testing, integration and installation of the information system in a specific computer system environment.

3.2 A modified version of meta-model

The outlined meta-model is a good tool to arrange the components of the development process but it can be enhanced and adopted to an environment where for instance the newest version of SSADM is made use of [Eva 1992, NCC 1990] (Fig. 1).
One of the deficits of the meta-model is that it does not utilize the diverging viewpoints of the users and the business analysts. The user level is too simple although recently the computing-conscious users think in terms of dialogues, logical screens, menus and screens etc. This aspect should incorporate the functional requirements from the user side, the user roles and the related user activities. The differing perception of the system by the users and the analyst should be exploited through confronting the user and the functional aspects.
The functional level or aspect can be refined further. The classical models [Brodie 1982] fall into two categories dealing with the dynamic and static sides of an information system, i.e., the data model and the function/process model. The active components can be classified into process model, function model, event model, dialogue model according to the most modern analysis methods [Eva 1992, Matheron 1990, Turner 1990]. The function model means in this context the representation of the user requirements from the viewpoints of analysts so it is a variety of the functional requirements incorporated into the user level.
The technical design level should be adopted to the chosen technical environment whether 3 GL or 4 GL or object-oriented but because of the lack of space and complexity of the related questions we cannot go into detail. The meta-model at this level depends on the peculiarity of the concrete environment and in generality we cannot go beyond in [Hesse 1988] described structure.

4. The meta-model in the structure of IRDS

In this section, the relation of the user aspect and the functional design aspect to the structure of IRDS will be investigated, the connection of the technical and the implementation side is discussed briefly elsewhere [Goldfine 1985].

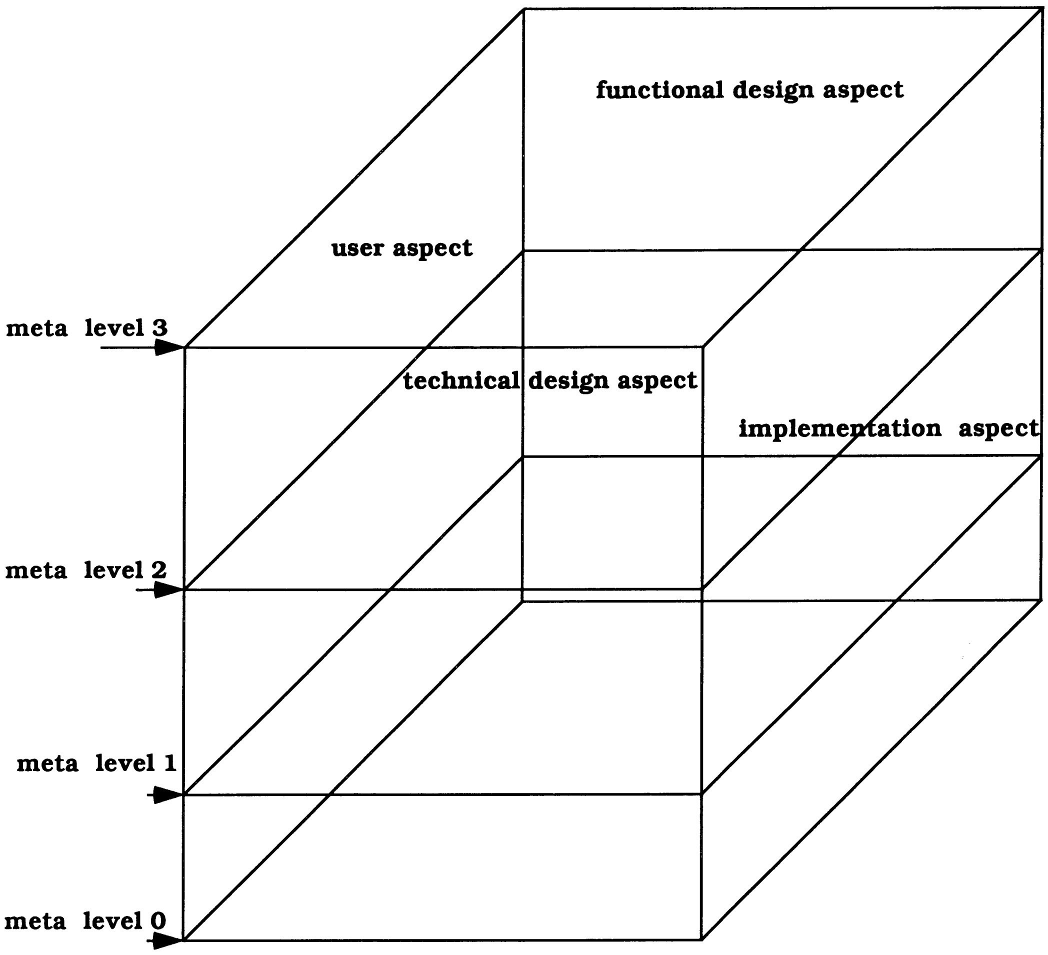

functional design aspect
user aspect
meta level 3
technical design aspect
implementation aspect
meta level 2
meta level 1
meta level 0

Figure 1.

4.1 The reconciliation of the orthogonal views

At the user level or aspect, the meta-meta-data is the *user concept*, the functional and user interface requirements appear at the meta-data level, the *is_a_part_of* relationships connect them to the outline or logical dialogue design, screen design, functional requirements for the system services, user roles and user activities. There exist relationship-types associating the previously mentioned entity-types to each other. For instance, a user-activity-instance *use* a logical-design-dialogue or a user-role-instance *execute* a user-activity., etc. The *use* relationship-type and the *user-activity-type* connected to each other through a meta-relationship. The functional requirements, the logical dialogues (screens, menus, etc.) and user roles can be connected similar way utilizing that the meta-relationships are able to describe more-than-binary relationships.
The functional design level or aspect of the meta-model provides the categories which comprise a sound base to build up a consistent and concise model of the concrete information or business application system. For this aim, an effective framework is needed in order to structure and exactly describe all relevant side of the application system and all significant pieces of information representing the properties and features of the system. This viewpoint showing the analysts' results is in a rival position to the user aspect.
The service requirements of the system should be collected into functions, the functions can be classified into a hierarchy of application functions, namely, main application functions, (normal) application functions and elementary application functions. The hierarchy can be defined by the recursive decomposed_into relationship.
The same procedure can be followed in the case of the process and event model. The DFD (data flow diagram, [Eva 1992, Longworth 1986] technique offers a similar hierarchy for processes, for the different levels and other components (dataflow, external entity, etc.) of diagrams (Fig. 3).
The events can be structured into main events and sub_events in an analogue way using decomposed_into relationship, but the events have relations to the effects and through the effects to the operations (logical database transactions). For example, an event-instance *cause* effect-instance on entity-type, an event-instance *trigger* a data-flow-instance in a DFD. The meta-entities are the *event-type*, the *entity-type*, *effect-type*, *operations-type*, *cause-relationship-type*, *trigger-relationship-type*, etc. The relationships are realized among them through meta-relationships (Fig. 2).
The effects on entities of events should be arranged into a Jackson-like structure diagrammatically in SSADM (ELH, entity life history). However, the Jackson structure is equivalent to a regular expression so the syntactic checking is well-defined and simply conceptualized so the generic procedure can be put into a meta-entity.
The semantic and syntactic checking , e.g. DeMarco level balancing [Longworth 1986], should be connected to certain components (meta-entities), that is, the information flows coming into or out of a higher level process (symbol) are equivalent to information flows appearing on an one-level-lower diagram crossing the boundaries and stepping in and out of this diagram which represents the higher level process in detail. These generic procedures can be placed into a meta-entity or meta-meta-entity and their effects can be imposed this way and then executed at the lower level.
The function and the processes should be correlated to each other, a function can *contain* several (elementary) processes. The principles of grouping the processes into functions may be based on the cohesion and coupling [Yourdon 1975], how much the processes close to each other in a certain metric. The classical properties are the data, control and common environment coupling, furthermore the coincidental, logical, temporal, procedural, communicational, sequential cohesion. These properties might be the attributes of the concrete relationships.
The structuring of the knowledge about the method in meta-entity or meta-meta-entity is important even in the case if there is not a computerized support in order that the right place may be seen where it belongs.

4.2 Object-orientation and the meta-model

Around the object-orientation, there is fairly great confusion, therefore our understanding should be clarified firstly.
Object-oriented languages and object-oriented database management systems offer the following features more or less.

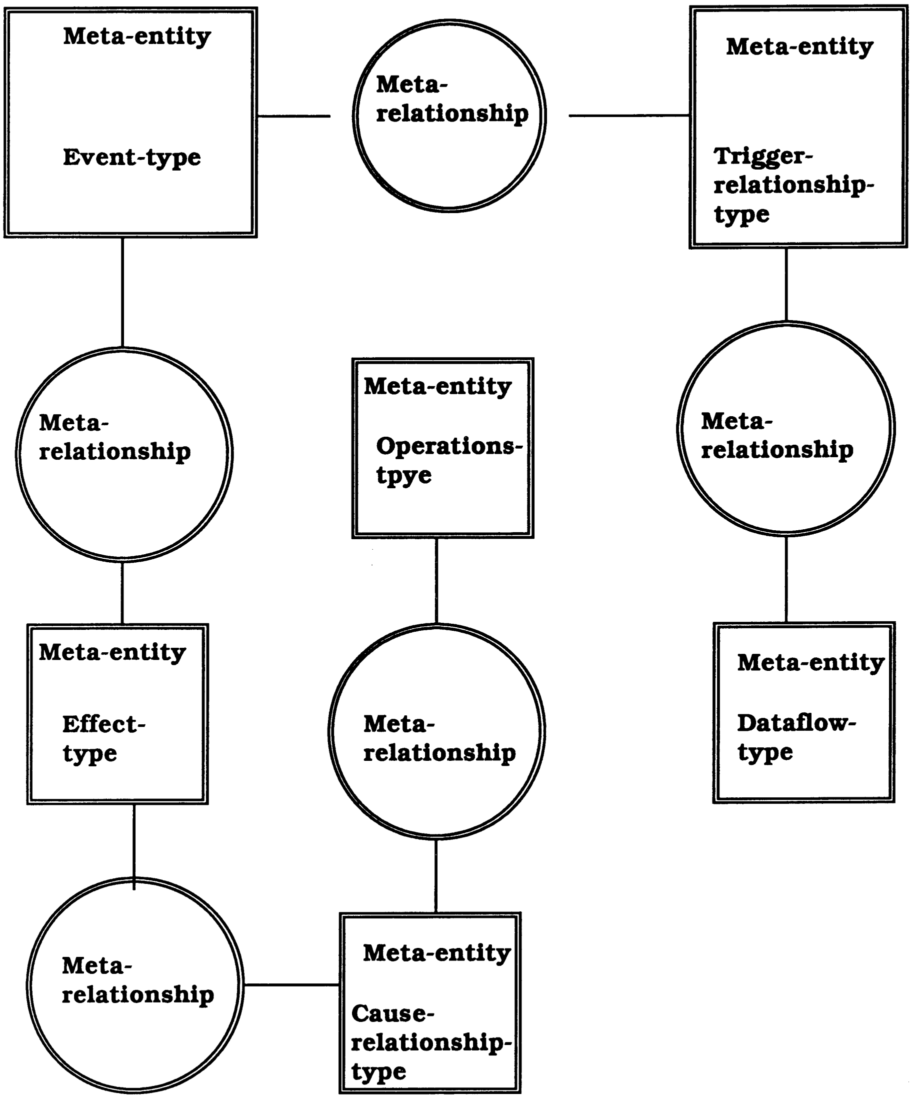

Meta-entity
Event-type
Meta-relationship
Meta-entity
Trigger-relationship-type
Meta-relationship
Meta-entity
Operations-tpye
Meta-relationship
Meta-entity
Effect-type
Meta-relationship
Meta-entity
Dataflow-type
Meta-relationship
Meta-entity
Cause-relationship-type

Figure 2.

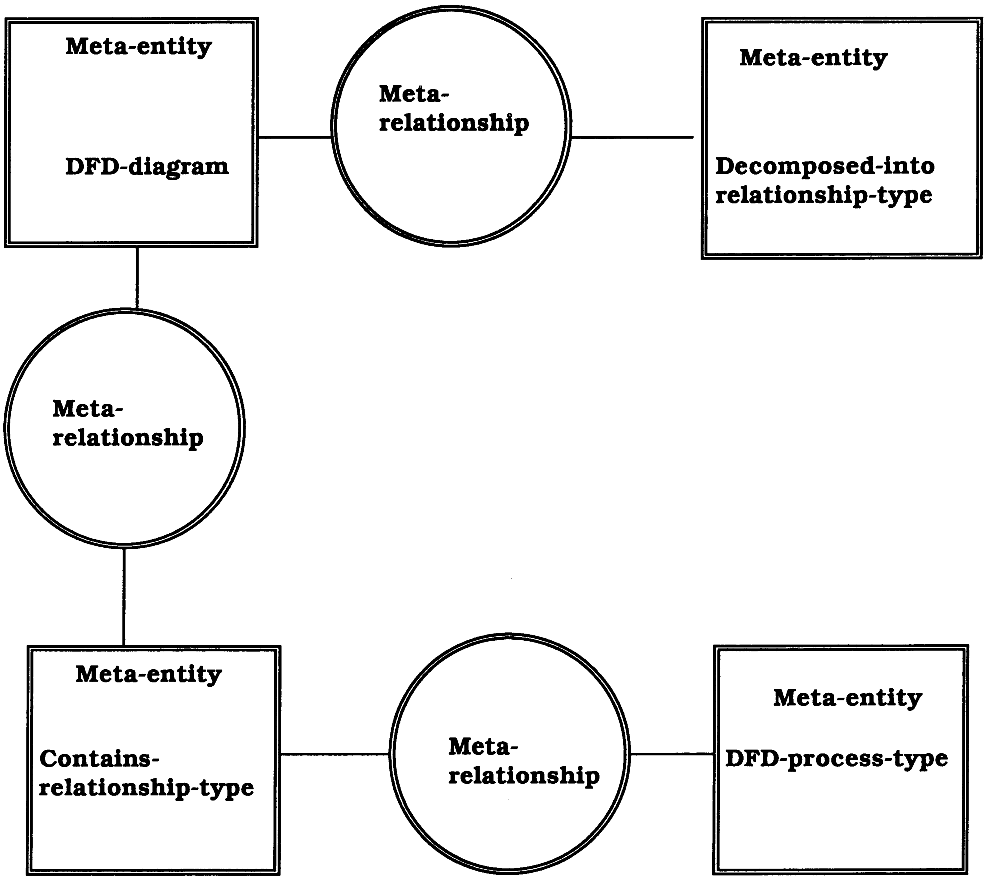

Meta-entity
DFD-diagram
Meta-relationship
Meta-entity
Decomposed-into relationship-type
Meta-relationship
Meta-entity
Contains-relationship-type
Meta-relationship
Meta-entity
DFD-process-type

Figure 3.

Every entity (here "Any concrete or abstract thing of interest including association among things", [van Griethyuysen 82]) in the universe of discourse is an *object* and all objects are classified into classes that themselves are objects [Stefik 1986]. The class hierarchy has a single root, class *object*; all entities are instances of that class. Class *class* is the subclass of class *object* whose instances are entities which represent classes; class *object* and all its subclasses are thus instances of class *class*. Class *relation* is the subclass of class object whose instances are entities representing relations; example instances of that class are relations *subclass* and *instance*. Class *individual* is the subclass of class *object* whose instances are entities which neither represent classes nor relations.
More technically, object-orientation means:

- data encapsulation
- inheritance
- polymorphism.

A technical and theoretical feature is:

- abstraction which has two roles:
  - implementation role
  - modelling role, i.e, the correct description of the requirements

The knowledge about the universe of discourse can be structured along various ontological planes or meta-levels; relation *instance* achieves the transition between planes.
The components of meta-levels of IRDS and the elements of the different aspects of the application system meta-model can be considered as objects. The hierarchy among the meta-levels in IRDS can be regarded as a class hierarchy. A level or aspect of the application system meta-model consists of entities, meta-entities, meta-meta-entities can be identified with the hierarchy of *objects* or *classes*, the relationship-type-entities and meta-relationships may be with the *relations*.

4.3 Discussion

The above outlined identification or mapping of hierarchy of objects and entities is fairly superficial.
In the context of the IRDS, the hierarchy of notions about the universe of discourse is identified in the form of entity instances, entity types, entity classes (meta-entities). The behavior of the entities can be specified by propositions (or rules). The constraints and the rules may be placed in meta-entities (see 2.1) as generic procedures, i.e., in terms of IRDS, at least one level higher than the entities themselves.
The events in the context of the model of a certain information system can be considered as something happened in either the universe of discourse or the environment or in the information system. The events can cause the modification of entity instances, at the level of the environment or at the level of the universe of discourse can alter some meta-entities or meta-meta-entities.
In the object-oriented approach, events are modelled as messages that are passed on to other objects in order to respond. Object has states and records states [Booch 1986] so a part of the universe of discourse is altered by transitions from one state to another and reacts to events by changing the state of the object or certain objects and by this way the state of the given system.
The rules or propositions can be stored in the slots of objects which are only accessible from within methods attached to the class of objects in which the slots are defined.
The IRDS separation of entities representing data structures from rules specifying control does not match the object-oriented concept because objects specify a composite of data and activity.
In the IRDS meta-model, entities are not considered to possess attributes, instead attributes are regarded as entities in their own right. This is contrary to object-oriented approaches in which attributes are components of objects.
The IRDS view of relationships does not fit the object-oriented conceptualization of relationships between objects being either caused by events or specified in terms of a classification hierarchy. In the object-oriented approach, the most straightforward way of supporting a relationship between objects is to define a slot on one that holds the identifier of another, the IRDS relationship concept is close to the Entity-Relationships notion but the solution to implement it similar. The problem is that the standard object-oriented model cannot represent the constraint that the two objects must point at each other.
Hence, the object-oriented approach has several common features with the concepts of IRDS

meta-model, but by no means all.
Booch's classification of objects by their behaviors is the following [Booch 1986]:

- actor objects; task-oriented objects may possess few data item and much complex algorithmic processing; perform actions which influence other objects in the system
- Server objects; data-oriented objects which undergo no operations other than simple updates to their attributes; the recipients of an actor activity and are related to the entity concept in the IRDS.
- Agent objects; mixture of the above outlined features

In order to resolve the differences in a theoretical framework, the entities at all level in the IRDS may be mapped onto server object type and the generic procedures onto actor object type. The behavior modelling remains within the meta-entities representing generic procedures and these are can be identified with the active objects, i.e. the actor object types. By this way, the elements of the meta-model of information systems development can be symbolized by objects.
There is a temptation to define 'larger' objects in which the differences may vanish between the meta-levels of IRDS and the aspects or levels of meta-model. The argument is that such a simple object-oriented structure can be more easily handled, provides more opportunity for reuse of components, the transition between the functional and technical level might be smoother, etc.
We do not agree with such an approach because the confronting views give chances to exploit the deviations in order to understand more profoundly the system. The object-oriented approach helps to structure the available knowledge in a comfortable way in the above mentioned style and if a tool is accessible which more or less object-oriented the outlined orthogonal views can be implemented.
If the collected information is represented in objects or frames, some reasoning mechanism might be used. For instance, the processes can be grouped into functions, the inconsistencies between the user aspect and functional aspect can be disclosed, etc. Some conflict resolution algorithm can be used to make the model consistent and to eliminate the deviations, namely the assumption based truth maintenance, the blackboard architecture and the reasoning with objects can be combined together [Bachimont 1991, Barbuceanu 1990, Molnár 1991].
Even if there is not available a tool the outlined structure aids to locate the places where a certain piece of knowledge about the methods and techniques should be used.

5. Summary and conclusions

The outlined framework is only a brief introduction how the meta-model of information system development process and the structure of IRDS can be used and these two orthogonal views how can be fitted together. In this framework, there is an intention to collide the diverse viewpoints and explicitly exploit it in order to build up a consistent model.
Such a framework gives a good guidance how the components, products, deliverables or documents relates to each other and therefore makes it easier to map a concrete project into a concrete tool or environment (e.g. CASE). This mapping process may need a more detailed meta-model of the concrete methodology but this framework can be refined further.
It seems worth investigating the object-oriented approach in this context too as the object-oriented models focus on the definition and inheritance of behavioral capabilities, in the form of operations or methods embedded within the objects, and also support simpler capabilities for structuring complex objects to view object-oriented models as having both structural and behavioral *encapsulation* facilities.
This framework facilitated creating the product descriptions for a large project with a new CASE tool and finding sub-optimal solutions for defining relationships and entities in the data dictionary of the CASE tool so it proved its usefulness.
If there were available a development dictionary which has services for customization and representing the knowledge attached to the methodology this framework and the collected and structured knowledge straightforwardly can be used.
Several systems and approaches are proposed and experimented [Demetrovics 1982, Molnár 1991, Konsynski 1984, Rouge 1990] whose architecture appearing in the practice would provide a good opportunity for incorporating the outlined framework of application system development.

6. Acknowledgements

The author thank the referees for their important and valuable comments and remarks which the author was not able totally incorporate in the text because of the short of space.

7. Bibliography and References

1. Bachimont, Bruno., 'DOTMS: A Dynamic Object-Based Truth Maintenance System to Mange Consistency in a Blackboard', in *Proc. 11th International Conference, Expert Systems and Their Applications, General Conference, Second Generation Expert System*, Avignon, France, EC2, pp 109-122 (1991).
2. Barbuceanu, M. , Trausan-Matu, S., Molnar, B. Concurrent Refinement: A Model and Shell for Hierarchic Problem Solving, *Proc. 10th International Workshop, Expert Systems and Their Applications, General Conference*, Avignon, France, EC2, 873-891 (1990).
3. Booch, G., 'Object Oriented Development' , *IEEE Transactions on Software Engineering*, Vol. 12 No. 2, pp 211-221, (1986).
4. Brodie, M. L., Silva, E., 'Active and passive component modelling: ACM/PCM' in Olle, T. W., Sol, H. G., Verrijn-Stuart, A. A. (eds.), *Information system design methodologies: A comparative view*, Elsevier Science Publishers B. V. (North-Holland), (1982).
5. Cameron, J.R., *JSP and JSD: The Jackson Approach to Software Development*, IEEE Computer. Soc., (1983).
6. CCTA (Central Computer and Telecommunication Agency), *PRINCE, Structured Project Management*, NCC Blackwell Ltd., (1991).
7. Chandrasekaran, B., 'Design: An Information Processing-Level Analysis', Technical Report, The Ohio State University, Department of Computer and Information Science, Laboratory for Artificial Intelligence Research, (January 1988).
8. Gane, C., *Computer Aided Software Engineering, the methodologies, the products and the future*, Prentice-Hall, (1990).
9. Date, C.J., *An Introduction to Database Systems*, Addison-Wesley, (1981).
10. Demetrovics, J., Knuth, E., Rado, P., 'Specification Metasystems', *Computer*, pp 20-35, (April 1982).
11. Essink, L. J. B., 'A modelling approach to information system development', in Olle, T. W., Sol, H. G., Verrijn-Stuart, A. A. (eds.), *Information system design methodologies: Improving the practice*, Elsevier Science Publishers B. V. (North-Holland), (1986).
12. Eva, M., *SSADM Version 4: A user's guide*, McGraw-Hill, (1992).
13. Goldfine, A., 'The Information Resource Dictionary System', in Chen, P.P. (ed.), *Entity-Relationship Approach, The Use of ER Concept in Knowledge Representation*, IEEE Computer Society Press/North-Holland, pp 114-122, (1985).
14. Hesse, W., Bosman, J. W., ten Damme, A. B. J., 'A four-level metamodel for application system development', in Bullinger, H.-J., et al. (eds.), *EURINFO '88, Information Technology for Organizational Systems*, Elsevier Science Publishers B. V. (North-Holland), pp 575-581, (1988).
15. Hewett, J., Durham, T., *CASE: The next step*, Ovum Ltd., (1989).
16. IBM, *Systems Application Architecture. AD/Cycle Concepts*, GC26-4531-0, (1989).
17. *IRDS: Information Resource Dictionary System*, American National Standard for Information Systems, X3.138-1988, (1988).
18. *IRDS: Information Resource Dictionary System Services Interface*, draft proposed American National Standard for Information Systems, (1988b).
19. ISO 10 0027: *Information Resource Dictionary System - Framework*, (1990).
20. ISO 10 0728: *Information Resource Dictionary System - Services Interface*, draft International Standard, (1991).
21. Jackson, M.A., *System Development*, Englewood Cliffs, Prentice Hall, (1982).
22. Konsynski, B.R., 'Databases for Information Systems Design' in *New Directions for Database Systems*, Ariav, Gad., Clifford, James. (eds.), Ablex Publishing Corp., pp 124-145, (1984).
23. Longworth, G., Nichols, D. *SSADM Manual Vol. 1-2*, NCC Blackwell, (1986).
24. Maciaszek, L. A., 'AD/Cycle Repository Manager from Object-Oriented Perspective', *ACM SIGSOFT Software Engineering Notes*, Vol. 16, No. 1, pp 50-53, (Jan 1991).

25. Matheron, J.P., *Comprendre Merise, Outils Conceptuels et Organisationnels*, Editions EYROLLES, (1990).
26. Molnár, B., Frigó, J., 'Application of AI in Software and Information Engineering', *Engineering Applications of Artificial Intelligence*, Vol. 4, No. 6., pp 439-443, (1991).
27. NCC (National Computing Centre), *SSADM Manual Version Four*, NCC Blackwell, (1990).
28. Pyle, I.C., *The Ada programming language*, Second Edition, Prentice-Hall, (1985).
29. Rouge, A., 'Techniques et Outils Intelligence Artificielle Comme Support Methodologique du Developpement & de la Maintenance des Bases de Donnees', *Proc. 10th International Workshop , Expert Systems and Their Applications, General Conference*, Avignon, France, EC2, pp 807-821, (1990).
30. Sibley, E. H., 'An Expert Database System Architecture Based on an Active and Extensible Dictionary System', in Kerschberg, L. (ed.), *Expert Database Systems*, The Benjamin/Cummings Publishing Company, Inc., pp 401-422, (1986).
31. Stefik, M., Bobrow, D., 'Object-oriented programming: themes and variations', *The AI magazine*, No. 6, pp 40-62, (1986).
32. Turner, W. S., Langenhorst, R. P., Hice, G. F., Eilers, H. B., Uijttenbroek, A. A., *SDM system development methodology*, Elsevier Science Publishers B.V. (North-Holland)/Pandata, (1990).
33. van Griethyuysen (ed), *'Concepts and terminology for the conceptual schema and the information base, computers and information processing'*, ISO/TC97/SC5/WG3 International Organization for Standardization, Geneva, Switzerland, (1982).
34. Winkler, J., 'The entity-relationship approach and the information resource dictionary standard', in Batini, C., (ed.), *Entity-Relationship Approach*, Elsevier Science Publisher B. V. (North-Holland), pp 3-19, (1989).
35. Yourdon, E., Constantine, L.L., *Structured Design*, Yourdon Press, (1975).

The Use of Deep Knowledge from the Perspectives of Cooperative Problem Solving, Systems Modeling, and Cognitive Psychology

Miklós Biró and István Maros*

Computer and Automation Institute
Hungarian Academy of Sciences
Budapest, Kende u. 13-17. H-1111, Hungary

Abstract. One of the points of the paper is that the exploitation of deep knowledge may well contribute to the appreciation of technology supporting small groups. The term deep knowledge is used in this paper for knowledge that is not only derived from rules acquired from experts, but is complemented with the application of usually numerical algorithms which incorporate a deep body of mathematical knowledge. A natural requirement in this context is the involvement of end-users in the mathematical modeling process. The selective usefulness of relational and functional modeling is analysed from this point of view. A graph theoretic algorithm is proposed for the support of relation building by end-users. The idea of spreadstructure is highlighted. Modeling is analysed from the perspective of cognitive psycholgy. Finally, a prototype modeling support system is presented which is built on the Microsoft Windows environment.

1. Introduction

The primary objective of this paper is the synthesis of ideas and techniques that could promote the use of group decision support technology for cooperative problem solving by even small groups. The issues are examined and the ideas are synthesized from a broad range of perspectives.

The concept of deep knowledge can be approached from different points of view. In artificial intelligence deep knowledge is usually considered as knowledge which can be accessed by going deeply down into the search tree. In the more specific field of expert systems, "the term

* Research supported by OTKA grants no. 2571, 2575 and 2587.

second- generation expert system is used to denote systems which employ both experiential, shallow knowledge and theoretical, deep knowledge" [18]. These approaches are not contradictory at all, even if there may be disagreement about definitions. The importance of the concept is nevertheless unquestionable.

The term deep knowledge is used in this paper for knowledge that is not logically derived from rules acquired from experts, but is generated by usually numerical algorithms based on underlying mathematics. The input of these algorithms is usually a mathematical model. In this paper we focus on linear programming models.

A central theme of the paper is that the most appropriate model representations and tools are different in each of the phases of the cooperative problem solving process. The difference is caused by both the nature of the task to be performed, and the professional background of the principal human actor of each phase.

An issue with an effect opposing to the consideration of the above one, is that the transition between the phases and their corresponding representations should be as smooth as possible for technical and psychological reasons.

Ideas contributing to the alleviation of the above conflict are discussed from the perspectives of the problem solving process, systems modeling, and psychology. The sections of the paper are organized around these perspectives. New ideas are discussed in the context of a prototype modeling support system which supports hierarchical concept generation and the building of relations from the concepts. These relations represent matrix entries in a linear programming model.

2. Inhibiting factors and their neutralization

Two extreme factors which inhibit the use of group decision support technology for cooperative problem solving are:

(1) The technology intervening between the participants in the cooperative problem solving process.

(2) The usual requirement for the knowledge and domain specific interpretation of a number of mathematical terms and approaches.

Let us discuss these factors in more detail.

2.1. Factor (1)

Nunamaker, Applegate and Konsysnski [25] give account of extensive experiences with advanced group decision support software (PLEXSYS) and hardware facilities. They report that small groups were frustrated despite of the high level of technology whose intervention between participants may be more inhibiting than stimulating. Their conclusion is that the inhibiting effects are only counterbalanced by information processing benefits if the group size is large enough. These experiences are drawn from unstructured problems, where ideas can only be generated by the participants themselves.

We propose that the unstructured, brainstorming style generation of ideas be coupled with a natural generation of deep knowledge producing models. Deep knowledge may be successfully exploited by even small groups since the generation of optimal alternative solutions through model experiments assumes the use of a computerized system anyway. Thus, the inhibiting effects of the intervening technology are counterbalanced by the computing power necessary in this case.

2.2. Factor (2)

The primary concern here is the complexity of formulating and manipulating a sophisticated model which presumes the use of the technology (see factor (1) above) in the first place.

Even though the complexity of deep knowledge generation is not relevant to cooperative problem solving only, model formulation issues are more acute in this context because of the need to support users with a large variety of possible backgrounds. In this paper we focus on techniques which cause the least mental strain on the end-user while switching between unstructured idea generation and deep knowledge generation environments.

Our approach is based on the opinion that if "the final user is also the model builder, the modeler understands and trusts the model, and is likely to implement the solution" [27]. Model building and model management techniques meant for modeling-experts are only considered in this paper from the point of view of their potential applicability by non-expert users. Not neglecting however the dangers of end-user modeling [13], we consider these tools as most useful in the model rectification phase performed by a modeling-expert after the completion of the brainstorming and initial relational model building phases performed by the end-users. These phases are discussed in the following section.

3. Perspective of the problem solving process

Since our focus is deep knowledge generation in a cooperative problem solving environment, we isolate the following phases of the cooperative problem solving process:

(1) Idea and concept generation through brainstorming. These include decision criteria as well as potential alternatives suggested by the decision makers.
(2) Initial relational model building by the end-users.
(3) Model rectification by a modeling-expert, solution of the model, and interpretation of the results (sensitivity, postoptimality analyses).
(4) Inclusion of the generated solution among the decision alternatives with any comments and assumptions related to the underlying model.
(5) Evaluation of the alternatives (sensitivity analysis, ranking).

Of course, these steps may be performed repeatedly according to the classical modeling cycle and can be complemented by problem partitioning techniques as analysed in [32].

Phase (1) is usually supported by group decision support systems. Model building systems also support phase (1), they do not allow however for a self-contained structuring of the concepts independently from the model under construction. This problem is discussed in more detail in the following section.

Phase (2) for deep knowledge generation is one of the central issues of this paper. It is not supported by either existing group decision support systems or model building systems. The target users of model building systems are usually supposed to be modeling-experts, not end-users. Spreadsheets and even relational data base management systems prove however the viability of the phase suggested above. Its necessity, as a means of building end-users trust in the model, has already been discussed in the previous sections. We will return to this issue from other perspectives.

Phase (3) encompasses the usual phases of modeling in existing model building systems. Since the details are extensively discussed in the literature, they are omitted in this paper.

Phase (4) is also pertinent to the central theme of the paper. In our approach, the solution generated by a mathematical model is only considered as an alternative to creative solutions suggested by the decision makers. In addition, the same model may yield several alternative solutions when model experiments are performed with varied parameters. Comments and opinions may be attached to these solutions by either the modeling expert or the decision makers.

Usual model representations and techniques appropriate in phase (5) include spreadsheets, and multi-attribute utility decomposition (MAUD). The methods enhancing the choice from the set of suggested or generated alternatives (AHP [29], ELECTRE, PROMETHEE [28], [8]) are not detailed in this paper.

The system described in [3] is based on MAUD and supports the hierarchical development and evaluation of decision criteria. Its advantage is that it can be readily coupled with the modeling support system described in this paper, since their concept structures can be identical. This is an example of the support of smooth transition between the brainstorming and modeling phases. The smooth transition between these and the final evaluation phases will be supported by the later defined spreadstructures.

4. Perspective of systems modeling

It has been discussed in the introduction that different model representations are more or less appropriate in different phases of the cooperative problem solving process. In our opinion the building of a concept hierarchy is appropriate in the brainstorming phase, relation matrices in the initial model building phase, semantic nets or relation matrices in the model rectification phase, and spreadsheets or later defined spreadstructures in the interpretation and evaluation phases. These representations are discussed below and their selective application is suggested according to the above opinion.

4.1. Hierarchies and semantic nets

The following are the fundamental reasons for the hierarchical structure of complex systems as discussed by Herbert A.Simon [30] [31]:

(1) Hierarchical systems are most apt for evolution among systems with given size and complexity, since the components of a hierarchy are themselves hierarchies which are stable structures.
(2) The information transfer requirement between the components of a hierarchical systems is less than in other systems.

(3) The local complexity of a hierarchical system is highly independent on its size.

Let us examine some essentially different applications of hierarchies for problem solving.

- Conceptual hierarchy.

In the brainstorming phase, a hierarchy is useful as a mental guidance for the consideration of all relevant concepts of the problem. For the above reasons, an appealing hierarchy building tool is particularly important in a system which is meant to motivate the user who enters the concepts himself.

The "modular structure" in the framework of structured modeling introduced by Geoffrion [14] [15] and the AHP by Saaty [29] are designed to accommodate hierarchical conceptual structures.

Psychological reasons for using a hierarchy and motivating the users to enter their own concepts are discussed later.

- Functional hierarchy and semantic net.

Gerlach and Kuo [16] apply a semantic network representation of model components, which is a hierarchy at the same time. This is a functional hierarchy which is meant to be built by an expert. This semantic network is similar to the functional hierarchies used by Müller-Merbach [24], the networks (element graph, genus graph) defined by Geoffrion [14] [15], the frame-based representation by Binbasioglu and Jarke [1], the graph based representation by Liang [19], and the LPN network introduced by Egli [11] and further developed by Hürlimann [17]. Its advantage is the fortunate combination of the classification and functional relationships in a single graph. The element and genus graphs of Geoffrion are, however, not restricted to hierarchies, but can be directed acyclic graphs. The inclusion of this latter feature into the combined semantic network would unfortunately make it much less manageable.

- Block decomposition hierarchy.

A third type of hierarchy relevant to our study is block decomposition appearing in the LPFORM system developed by Ma, Murphy, Stohr [20]. LPFORM provides a consistent graphical interface for building the matrix of a linear programming problem starting from blocks with interconnections. The detailed content of the blocks can be specified interactively or even retrieved from a database using a relational query language. The interconnections are specified using the activity modeling approach of [10], which is relatively natural for novice users as well. However, "the target user for LPFORM is primarily someone knowledgeable about LP". A semantic net (not a tree) representation is also provided in LPFORM for representing the relationships of models in the model base. Even though the semantic net defined in [16] is restricted to trees for usability reasons, it serves partially the same purpose.

4.2. Relational modeling

A serious drawback of the use of semantic nets in the model building phase of the cooperative problem solving process is that partial structures and definitions are to be fixed early, making subsequent changes more difficult. Vepsalainen [33] suggests a relational modeling approach based on diagonal semantic and activity matrices. This technique makes it easy to experiment with structures without committing to a specific decomposition.

The superiority of relational modeling to semantic networks in the model building phase can also be traced to reasons similar to those of the superiority of the relational data model to the network data model in data base management. "It provides a means of describing data with its natural structure only - that is, without superimposing any additional structure for machine representation purposes." [9] The underlying reason of the success of spreadsheets is also related to this fact.

After the above arguments for relational modeling, it must be remarked that nevertheless, network models have an undeniable expressive power. The controversy could be dissolved with techniques that would allow a smooth transition between the two representations. The spreadstructure idea described later in this paper is a contribution in this direction.

5. Perspective of psychology

There is a paradigm in the science of cognitive psychology which is built on the concept of cognitive patterns [22]. Cognitive patterns are models of the complex knowledge structures appearing and evolving in the human brain. It is an experimental fact that even the perception of our everyday environment is restricted to those phenomena for which we already have cognitive patterns. A model of the storage area of these cognitive patterns is called Long Term Memory (LTM). The buffer between LTM and the real world is called Short Term Memory (STM). It is a stable experimental fact as well, that the capacity of STM is 7 plus or minus 2 units of information. Nevertheless, a unit of information may mean a highly complex cognitive pattern transferred from LTM. New and improved patterns migrate to the LTM from the STM, but the details of this process are very little known.

Cognitive patterns can be categorized into everyday patterns and professional patterns which are connected. This connection is however much looser for an apprentice and it becomes mature at the master level [22].

How are the above concepts related to the issues of this paper?

Experts participating in a cooperative problem solving process may have different professional backgrounds which implies that their professional cognitive patterns are different. A tool supporting cooperative problem solving must provide support for each individual expert and for the group as a whole. Thus, the model representations offerred by the system must be appealing to all of the participants, which implies that they must be as close as possible to everyday cognitive patterns. Tabular representations in both ralational matrices and spreadsheets satisfy this requirement, since tables are incorporated among our cognitive patterns at the elementary school level. This is another fundamental reason of their general success.

A point of view opposing but in fact complementary to the above one, is that an individual expert will find the system appealing if he can find model representations close to his professional patterns. It is an experimental fact as well, that there may be an essential decrease in problem solving efficiency if the representation of the problem is not familiar, even if it is completely isomorphic to a familiar one [22]. By consequent, it is important to offer model representations most appropriate for each of the experts in the different phases of the cooperative problem solving process.

The following are further psychological factors which contribute to the popularity of the system:

(1) The user must not be a passive observer or plain server of the system.

This means among others that the concept hierarchy must be built by the users themselves since the model will only be familiar to them in this case. This gives a feeling of active participation at the same time. We discussed earlier that users should stay involved with the building of even a deep knowledge generating model, so that they do not lose contact. An expert can give technical advice and help however.

(2) The user must not be expected to perform complex operations or interpretations.

This issue is related to the inhibiting effects of excessive learning requirements imposed on the user. Gerlach and Kuo [16] highlight the importance of user interface design in this respect. In their approach however, it is an expert who predefines the model. In the modeling support system described below, user involvement is stressed in the model definition phase as well, not leaving the user interface design out of sight either. This will naturally reduce learning problems even if an expert performs rectifications on the model as discussed earlier.

(3) The advantages provided by the system must overweigh the burden imposed by the intervening technology.

The issue of the facility for deep knowledge generation for this purpose is a central theme of this paper. Most of the discussed ideas are focused on relieving the contradiction of this requirement with the previous one.

(4) The psychological fact that humans cannot take much more than seven concepts simultaneously into consideration has already been mentioned.

The use of hierarchies helps in this respect, since the number of direct descendents of an entry can be restricted to be no more than the magic number.

(5) Floyd, Turner, and Roscoe Davis [12] highlight the importance of "computer based gaming" as a means of unfreezing the users.

The "point and shoot" style generation of new relations between entities in the modeling support system below has resemblance with the style of computer games. This will increase the willingness of the participants to experiment with the system.

6. Modeling approach and new ideas to be implemented

The prototype MOdeling Support SYstem MOSSY takes advantage of the Microsoft Windows environment running on IBM/PC compatible computers. MOSSY supports the initial generation and hierarchical structuring of ideas in the form of objects that can be manipulated on the screen using a mouse. In addition to the hierarchical structuring of the objects, relations can be established between any pair of them. The entity-relationship model generated in this way is explicitely visualized and made accessible in a window. MOSSY incorporates a user interface management system (UIMS) which allows the coupling of any information to the objects in the most suitable form.

6.1. The prototype system

In the example presented in the appendix, relations represent matrix entries in a linear programming model. MOSSY is developed to the point of generating an MPS format input file and solving the model.

The target user of MOSSY is not necessarily knowledgeable about LP. However, keeping the requirements for widespread use and deep knowledge handling in focus, MOSSY allows any user to first build a hierarchy of his own concepts, then relate these concepts by simply clicking with the mouse on their window representations. This is a rather simple, even relaxing process, during which even data or any information characterizing the ralations can be entered. The block structure of the resulting complete relation matrix is instantly seen on a proportionally sizeable map of the relation window whose visible area may actually contain relatively few of the entities.

Even though it cannot be expected from end-users that they build a correct mathematical model, there are essential psychological benefits in motivating them to go as far as possible. These benefits have already been discussed. An LP expert can be called upon to rectify the model after the end-user has partially defined it. The model building systems mentioned earlier can provide the necessary support for the expert.

The presence of a modeling expert is also necessary for enforcing compliance with the verification and validation requirements of the model life-cycle as cautioned by Gass [13] and mentioned earlier. The approach proposed above guarantees however both the preservation of user interest and the compliance with professional standards.

An ultimate solution to the above problem would be the elaboration of a model building expert system which could partially relieve the requirement for direct expert involvement [6].

The matrix representation of the entities and their relationships in MOSSY bears the same advantages over functional network representation as those mentioned in the section on relational modeling. A further advantage of building a matrix as suggested in MOSSY is that only the relevant relations have to be dealt with preserving in this way the advantages of sparse matrix definition techniques (e.g. MPS format). This approach is on the other hand at a far higher level. The entities are immediately visible and accessible in matrix form and can even be transferred into a spreadsheet.

Direct simultaneous contact with spreadsheets (e.g. Microsoft Excel) is supported by MOSSY through the clipboard of Windows. There is a possibility for dynamic data exchange as well.

6.2. Support for building a relation matrix from differently structured or unstructured concepts.

When a network representation is used, the activity model is considered to be the definition of flows with various inputs and outputs. When a relational representation is used, relations are established between selected entities. Let us assume that relations are established by the user as suggested in MOSSY, and not by an expert. A problem occurs when the entities on both sides of the relation selected by the user have to be assigned to either rows or columns of the matrix. When an entity is selected for the first time in any relation, it is assigned to a row if it is the first operand of the relation and to a column if it is the second. It cannot be expected however, that the user will always select entities already assigned to rows as first operands and entities assigned to columns as second operands. The question is whether the selected operands of a new relation can be assigned to a row and a column in consistency with previous assignments of the entities in other relations.

MOSSY is designed to provide support for this assignment by applying algorithmic techniques. The problem can be formulated in graph theoretical terms, and turns out to be a special case of the Precoloring Extension problem introduced and extensively studied in [4], [5]. The problem in graph theoretical terms is deciding whether a given two-coloring of a bipartite graph can be extended when a new edge and a new node are introduced. Precoloring Extension can be efficiently solved in this special case, since the problem is simply deciding whether the new graph is still bipartite.

6.3. Spreadstructure

Some fundamental reasons of the success of spreadsheets have already been mentioned. In fact, spreadsheets were among the first software tools which have led to the widespread use of DSS within organizations [25]. In MOSSY we are planning to learn from the success of spreadsheets, and include immediate expression evaluation capabilities naturally attached to the conceptual hierarchy built by the user [26]. This facility is an important step toward widespread use, since one of the drawbacks of spreadsheets is that they do not visually support the manipulation of complex structures other than tables and matrices. This is, however, meaningful only if the conceptual hierarchy built for managing the complex structure reflects functional relationships at the same time.

A general system supporting immediate expression evaluation based on an arbitrarily structured construct could be called SPREADSTRUCTURE instead of spreadsheet. Such an object-oriented spreadstructure could even prompt for unspecified values or expressions, and take advantage of artificial intelligence techniques. A spreadstructure would provide an appropriate transitional representation between the brainstorming and evaluation phases of the problem solving process. The implementation of dynamic link between a functional spreadstructure and a corresponding relational spreadsheet would provide smooth transition to the model building phase as well. (A commercial realization close to the spreadstructure idea is Borland ObjectVision for Windows which was announced after the publication of an earlier report [7] already containing the idea.)

7. Conclusion

Deep knowledge generation has been shown to be a necessary facility of group decision support technology for cooperative problem solving intended for widespread use by small groups. On the other hand, widespread use presumes that the system satisfies a number of requirements which have been examined from the points of view of the problem solving process, systems modeling, and psychology.

A facility for building a concept hierarchy is shown to be useful in the brainstorming phase of the cooperative problem solving process.

In order to keep the interest of the users alive, it has been suggested that they get involved in the initial building of the model. Once the model is solved, its solution is considered as an alternative to creative solutions suggested by the decion makers and to other solutions obtained through model experiments.

Relational modeling has been found to be more suitable for end-user model building than the semantic net approach, similarly to the superiority of relational data base management to network data base management.

Smooth transition between the different representations should be supported since the cognitive patterns of the end-users and modeling experts may be different, and different representations may be more or less appropriate in the various phases of the cooperative problem solving process.

Support based on a combinatorial algorithm is provided for the establishment of relations by the end-user.

The idea of a general system supporting immediate expression evaluation, in the spreadsheet tradition, on an arbitrarily structured construct has been raised under the name SPREADSTRUCTURE.

Acknowledgment

We express our thanks to Prof. Tibor Vámos for his helpful criticism of a previous version of this paper.

References

[1] M. Binbasioglu and M. Jarke, Domain Specific DSS Tools for Knowledge-Based Model Building, Decision Support Systems, 2(1986)213-223.

[2] M. Biró, P. Turchányi and M. Vermes, CONDOR-GDSS CONsensus Development and Operations Research tools Group Decision Support System, MTA SzTAKI Report, Budapest, Hungary, 23/1989.

[3] M. Biró, P. Csáki and M. Vermes, WINGDSS Group Decision Support System under MS-Windows, Proceedings of the Second Conference on Artificial Intelligence, John von Neumann Society for Computer Sciences, (ed. by I.Fekete and P.Koch), Budapest, Hungary, (1991) pp.263-274.

[4] M. Biró, M. Hujter, On a graph coloring problem with applications in scheduling theory, in: H. Sachs, Ed., Proceedings of the International Conference "Discrete Mathematics" (Eisenach), Technische Hochschule Ilmenau, Germany, (1990).

[5] M. Biró, M. Hujter and Z. Tuza, Precoloring Extension I: Interval Graphs, Discrete Mathematics 100(1992)(to appear).

[6] M. Biró, J. Mayer, T. Rapcsák and M. Vermes, Building Mathematical Programming Expert Systems, Proceedings of the Second Conference on Artificial Intelligence, John von Neumann Society for Computer Sciences, (ed. by I.Fekete and P.Koch), Budapest, Hungary, (1991) pp.155-162.

[7] M. Biró, I. Maros, Deep knowledge for group decision support, MTA SzTAKI Report, Budapest, Hungary, 42/1991.

[8] J.P. Brans, Ph. Vincke and B. Marechal, How to select and how to rank projects: The PROMETHEE-method, European Journal of Operational Research, 24(1986)228- 238.

[9] E.F. Codd, A Relational Model of Data for Large Shared Data Banks, Communacations of the ACM, 13, no.6(1970)377-387.

[10] G.B. Dantzig, Linear Programming and Extensions (Princeton University Press, Princeton, New Jersey, 1963).

[11] G. Egli, Ein Multiperiodenmodell der linearen Optimierung für die schweizerische Ernährungsplannung in Krisenzeiten, Dissertation, University of Fribourg, Switzerland, (1980).

[12] S.A. Floyd, C.F. Turner and K. Roscoe Davis, Model-Based Decision Support Systems: An Effective Implementation Framework, Computers Opns. Res., 16, no.5(1989)481-491.

[13] S.I. Gass, Model World: Danger, Beware the User as Modeler, Interfaces, 20, no.3(1990)60-64.

[14] A.M. Geoffrion, An Introduction to Structured Modeling, Management Science, 33, no.5(1987)547-588.

[15] A.M. Geoffrion, The Formal Aspects of Structured Modeling, Operations Research, 37, no.1(1989)30-51.

[16] J. Gerlach and F. Kuo, An Approach to Dialog Management for Presentation and Manipulation of Composite Models in Decision Support Systems, Decision Support Systems, 6(1990)227-242.

[17] T. Hürlimann, LPL: A Structured Language for Modeling Linear Programs (Verlag Peter Lang, Bern, 1987).

[18] M. Klein and L.B. Methlie, Expert Systems A Decision Support Approach (Addison-Wesley, Wokingham, England, 1990).

[19] T. Liang, Development of a Knowledge-Based Model Management System, Operations Research, 36, no.6(1988)849-863.

[20] P. Ma, F.H. Murphy and E.A. Stohr, Representing Knowledge about Linear Programming Formulation, Annals of Operations Research, 21(1989)149-172.

[21] I. Maros, MILP linear programming optimizer for personal computers under DOS, Preprints in Optimization, Institute of Applied Mathematics, Braunschweig University of Technology, (1990).

[22] L. Mérö, Ways of Thinking. The Limits of Rational Thought and Artificial Intelligence (World Scientific Publ., London, 1991).

[23] F.H. Murphy and E.A. Stohr, An Intelligent System for Formulating Linear Programs, Decision Support Systems, 2(1986)39-47.

[24] H. Müller-Merbach, Model Design Based on the Systems Approach, J. Opl. Res Soc., 34, no.8(1983)739-751.

[25] J.F. Nunamaker, L.M. Applegate and B.R. Konsynski, Computer-Aided Deliberation: Model Management and Group Decision Support, Operations Research, 36, no.6(1988)826-848.

[26] W.E. Pracht, An Object Oriented Approach for Business Problem Modeling, in: M.G.Singh, K.S.Hindi and D.Salassa, Eds., Managerial Decision Support Systems (Elsevier Science Publishers B.V., North-Holland, 1988).

[27] A. Roy, L. Lasdon and D. Plane, End-user optimization with spreadsheet models, European Journal of Operational Research, 39(1989)131-137.

[28] B. Roy and Ph. Vincke, Multicriteria analysis: Survey and new tendencies, European Journal of Operational Research, 8(1981)207-218.

[29] T.L. Saaty, The Analytic Hierarchy Process (McGraw Hill, 1980).

[30] H.A. Simon, The Sciences of the Artificial (M.I.T. Press, Cambridge, Massachusetts, 1969).

[31] H.A. Simon, The New Science of Management Decision (Prentice Hall, Englewood Cliffs, New Jersey, 1977).

[32] R.G. Smith and R. Davis, Frameworks for Cooperation in Distributed Problem Solving, IEEE Transactions on Systems, Man, and Cybernetics, 11, no.1(1981)61-70.

[33] A.P.J. Vepsalainen, A Relational View of Activities for Systems Analysis and Design, Decision Support Systems, 4(1988)209-224.

LESSONS OF A FIRST-YEAR USE OF THE AUTOMATED REASONING TOOL

J. Váncza and A. Márkus

Computer and Automation Institute
Hungarian Academy of Sciences
H-1518 Budapest P.O.B. 63
e-mail: h140van@ella.hu

Abstract. The paper discusses problems we have encountered while using the advanced knowledge representation and reasoning system ART for developing an automated process planning system. First, key concepts and distinct *modus operandi* of ART are presented through showing how they match the requirements of the process planning task. Then we discuss the lessons that previous experience and skill in application of conventional programming methods is the main factor that makes programming in an integrated knowledge based environment more cumbersome than expected. Finally, a knowledge compilation strategy is outlined that would enable us to deliver results to more traditional and simple computing environments.

1 Introduction

Artificial intelligence applications progress along two paths: while one path leads through the selection or development of tools to be given to the domain experts, choosing the other way means that AI is used mostly for analyzing the domain and the perspectives of the solution processes. Having chosen this way, what the end user meets is rather a result of AI methods than actual AI tools and techniques themselves.

Working in a project in computer-aided generation of manufacturing process plans (CAPP), it has turned out soon that the complexity of our tasks and the fragmented nature of relevant domain knowledge are very much against the application of straightforward algorithmic methods and call for the application of AI tools. Moreover, aiming at an efficient use of the limited human and computing resources through the separation of research, development and application environments, we have adopted the second approach of AI applications. The aim of this paper is to present the lessons of our first-year use of the Automated Reasoning Tool of Inference Corp. (henceforth ART), as applied for building this process planning system.

ART is one of the most advanced integrated knowledge representation and reasoning systems that was conceived in the mid eighties as a complete tool-kit for building large-scale knowledge based applications. It supports object-oriented and rule based programming, hypothetical and temporal reasoning, and access to conventional languages. Major components of this integrated system are: (1) a language for knowledge representation and rule based programming, together with its inference engine, and (2) an environment for supporting program development. For a detailed description we refer to the tutorials and manuals (ART Reference Manual 1988, Clayton 1987).

The paper first discusses the problems encountered while developing this process planning system: there will be shown what kinds of engineering knowledge have been represented by what features of ART, what issues have turned out simple, and what are the hard ones. The second half of the paper tries to form generalizations of the lessons we have got and advocates for a new style of program development.

2 Process planning and ART - how they fit to each other

2.1 Our approach to the process planning problem

The primary objective of manufacturing process planning is to specify and arrange the order of manufacturing operations and to select resources (machine tools, tools, fixtures) that are needed for transforming the blank part to its final form. Moreover, process plans have to be executable in the sense that the selected machine tools be capable to produce the part and be available when actually needed. Economic considerations of improving cost effectiveness and productivity are also of primary importance.

In process planning, so-called manufacturing features of workpieces (slots, pockets, holes, faces etc.) are the key concepts that permit the localized representation and manipulation of planning knowledge. A manufacturing feature is a maximal technological entity for which all applicable processing methods have been collected. (N.B., maximal here means that, with respect to processing methods, no more complex entities can be constructed without facing the need of considering other features.) The set of the applicable manufacturing processes provides an implicit definition of the feature and, at the sane time, it establishes links to the representation of related concepts (such as machines, tools, sequencing and equivalence constraints between processes). In spite of the fact that our planning method is being built on the concept of features, i.e. on a concept with much local flavor, planning inevitably must incorporate the concept of global economical optimum as well.

Our process planning method works as follows: a global and robust optimization process runs in the middle of several, highly domain-specific, local reasoning and optimization steps that are handled by dedicated tools. These steps have been defined so that combinatorial complexity of global optimization could be focused into a single, well formalized step, even if this step grows unusually large. For driving the global optimization process, genetic algorithms have been applied (see Váncza and Márkus 1991).

Within the above framework, domain specific knowledge is represented and manipulated by ART; it is ART who builds up the search space for the genetics-driven optimization. Through representing domain knowledge in ART we could get rid of several simplifying assumptions that became unwarranted *de facto* standards of present days' process planning systems.

2.2 A correspondence schema

ART has provided appropriate tools and reasoning techniques for capturing and modeling basic concepts as well as thinking particular to process planning. Below there are given pairs of closely related concepts of process planning versus ART (mappings in square brackets have not yet been verified by implementation):

objects (as features, processes, machines)	--- schemata
taxonomies	--- inheritance networks of schemata
geometric, tolerance relations	--- customized relations
part model	--- network of schema instances
rules for selecting, reference features, processes, setups, machines, ordering constraints, etc.	--- forward chaining rules

rules analyzing the part model	--- backward chaining rules
local scope of rules	--- patterns
external procedures, global optimization	--- rule and LISP processing intermingled
satisfying constraints on reference features	--- hypothetical reasoning
maintaining alternative part interpretations	--- [hypothetical reasoning with worlds]
causal reasoning on the part model	--- [hypothetical and temporal reasoning]

2.3 Definition of conceptual models

The world of process planning consists of complex objects like the workpiece to be produced, manufacturing features that build up the workpiece, machine tools, fixtures, cutting tools and other equipments that may have a contribution to the production process. Moreover, we consider also the manufacturing processes as objects of this world. Typically, these objects (especially features and processes) have a large, heterogeneous set of characteristics, that, however, do not provide clear-cut conceptual boundaries. Hence, techniques for constructing open-ended flexible conceptual models are sought for.

ART supports linking the facts which are related to a particular object. Such objects are called schemata; once a schema is defined ART can reason about it in terms of its related facts. A schema may be defined by inheritance relationships to other schema(ta). Both kinds of standard inheritance relations, i.e. **is-a** and **instance-of,** are supported.

Using schemata and **is-a** relations, we have structured and defined many concepts of our domain. First of all, conceptual taxonomies for the types of features and subfeatures of prismatic parts, machining processes, machine tools as well as cutting tools have been created. As an example of the hierarchical taxonomy of features, see. Fig. 1.

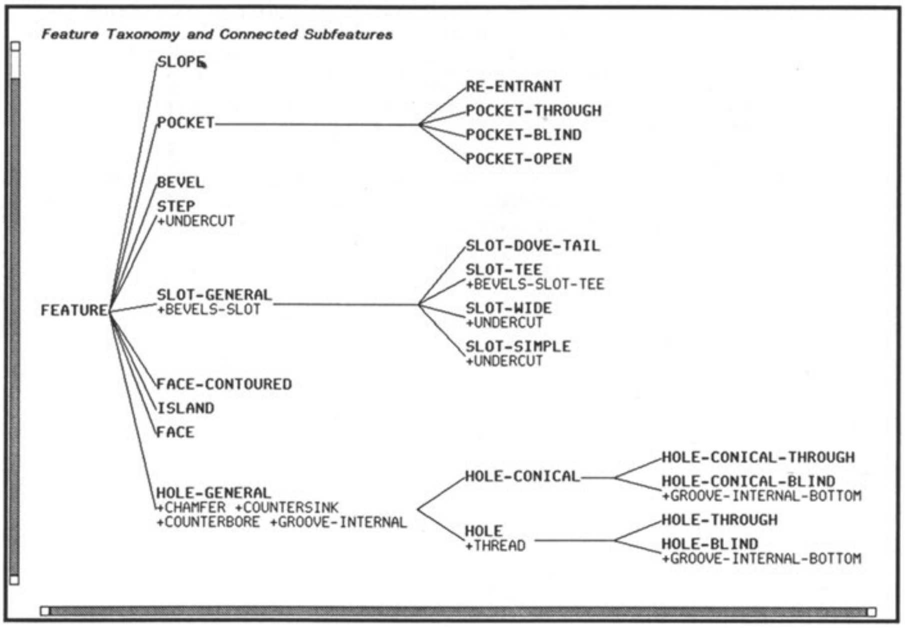


Fig. 1 The hierarchical taxonomy of features and subfeatures

With using the above concepts, actual planning tasks (i.e. descriptions of a particular part, available machines and applicable processes) are specified as instances of the general objects. As an example, see the description of a particular hole in Fig. 2.

Schema TH5

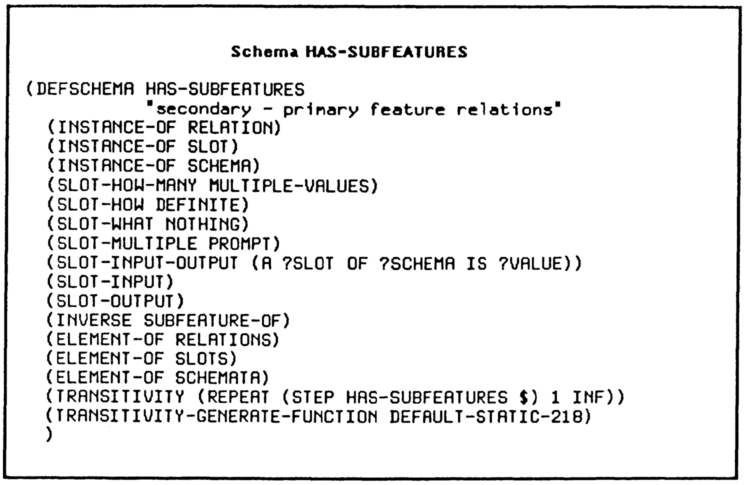
```
(DEFSCHEMA TH5
           "threaded hole in H7"
  (BELONGS-TO-PART PART#2-SLIDE)
  (CHILD-OF H7)
  (IS-A COMPOUND-FEATURE)
  (INSTANCE-OF HOLE-BLIND)
  (INSTANCE-OF HOLE)
  (INSTANCE-OF HOLE-GENERAL)
  (INSTANCE-OF FEATURE)
  (LOCATION (X F5 -12.5))
  (LOCATION (Z F4 18))
  (ORIENTATION (+Y))
  (DIAMETER 4)
  (DEPTH 35)
  (BOTTOM-TYPE FLAT)
  (HAS-SUBFEATURES TH5-THR1)
  (FEATURE-NAME HOLE-BLIND)
  (FEATURE-HU-NAME ZSAKFURAT)
  (SURFACE-FINISH)
  (MATERIAL-QUALITY)
  (LOCATION-TOL)
  (APPROACH-DIRECTIONS)
  (INITIAL-STATE RAW)
  (PLAN-TYPES)
  (STATE)
  (NETWORKS)
  (BASE-TYPE 0-POINT)
  (INDEX 17)
  (BASE (Z F4 SMOOTH))
  (BASE (X F5 SMOOTH))
  (BASE (Y F3 SMOOTH))
  (FIRST-STATE)
  (LAST-STATE)
  (THROUGH-TYPE BLIND)
  (DIAMETER-TOL)
  (CIRCULARITY)
  (STRAIGHTNESS)
  (COLLINEARITY)
  (PARALLELISM)
  (ORTHOGONALITY)
  (ANGULARITY)
  (DEPTH-TOL)
  (SIDE-SURFACE-FINISH)
  (BOTTOM-SURFACE-FINISH)
  )
```

Fig. 2 Schema of a particular instance of a feature

2.4 Definition of relations and semantic networks

Instances of features correspond to subproblems of the planning problem among which several relations and dependencies may exist. Geometric relations between features can be interactions (when two feature volumes physically meet each other through nesting or intersection), or non-contact type relations (when no physical interaction occurs but other geometric relations - parallelism, coaxiality or perpendicularity - exist between features). Since tolerance and other requirements have to be dealt with in the planning problem, a rich vocabulary of relations is needed for constructing a useful model of the part.

As a matter of fact, an ART schema is a semantic net that organizes knowledge by defining objects in terms of their mutual relations. The user has the means to define his custom relations, characterized by the arity, inheritance procedures, direction, transitivity, and input/output format (see Fig. 3). If needed, relations may also call new relations into existence.



Schema HAS-SUBFEATURES

```
(DEFSCHEMA HAS-SUBFEATURES
           "secondary - primary feature relations"
  (INSTANCE-OF RELATION)
  (INSTANCE-OF SLOT)
  (INSTANCE-OF SCHEMA)
  (SLOT-HOW-MANY MULTIPLE-VALUES)
  (SLOT-HOW DEFINITE)
  (SLOT-WHAT NOTHING)
  (SLOT-MULTIPLE PROMPT)
  (SLOT-INPUT-OUTPUT (A ?SLOT OF ?SCHEMA IS ?VALUE))
  (SLOT-INPUT)
  (SLOT-OUTPUT)
  (INVERSE SUBFEATURE-OF)
  (ELEMENT-OF RELATIONS)
  (ELEMENT-OF SLOTS)
  (ELEMENT-OF SCHEMATA)
  (TRANSITIVITY (REPEAT (STEP HAS-SUBFEATURES $) 1 INF))
  (TRANSITIVITY-GENERATE-FUNCTION DEFAULT-STATIC-218)
  )
```

Fig. 3 The definition of a relation

A part is modeled by a semantic network of instances of features that build up the part. Moreover, features themselves may be compound objects consisting of primary features and subfeatures, like a hole and a chamfer on its mouth. The consistence of the part model is checked up automatically while the model is constructed, i.e. when the declarations of the features are compiled: relations of particular objects that are in conflict with the standard and/or customized declarations about the characteristics of the relations are detected by a built-in mechanism of ART.

2.5 Pattern matching and forward reasoning

A typical process planning problem contains a huge body of facts form which solutions must be constructed. Departing from an analysis of the required properties of the part and the capabilities of available resources, the CAPP system has to suggest alternatives of machining processes, machines, tools, orientations and reference surfaces. In order to avoid negative interactions between manufacturing processes, precedence constraints should be set on the ordering of the actions. (E.g., whenever a cross hole intersects a deep hole, the deep hole must be drilled prior the cross hole in order to avoid the leaking of the coolant and the subsequent breaking of the drill).

The above activities can be supported by forward reasoning made by rules that detect either the existence or absence of certain facts and act whenever a specific situation is found. The left-hand side of a rule is a conjunction of positive and/or negated conditions expressed in terms of existentially quantified predicates, which themselves may contain negations (so a condition may say that in the database "there exists no slot with a surface finish that is not rough"). Pattern matching in ART performs much more than a simple test of Boolean conditions on a set of variables when matched with a given set of database elements: it makes a search to determine all combinations of variable bindings that simultaneously satisfy the conditions.

We have many groups of rules for accomplishing distinct planning subtasks. Most of these rules have a rather limited, local scope; they look for a specific feature plus some closely related, neighboring features. This fragmented, fine-grained representation of the domain knowledge has several benefits: (1) it fits to the cognitive structures of process engineers (for a detailed discussion, see (Váncza and Márkus 1992)), (2) the rule base can be upgraded relatively easily, and (3) it allows for an efficient execution of the program.

```
                    Rule BASE-TYPE-SELECT-2-POINTS-3

(DEFRULE BASE-TYPE-SELECT-2-POINTS-3
         "cylindrical features appropriate for 2-point bases"
  (DECLARE (SALIENCE *BASE-SELECTION-SALIENCE*))
  (INSTANCE-OF ?FEATURE FEATURE)
  (NOT (BASE-TYPE ?FEATURE ?))
  (INSTANCE-OF ?FEATURE HOLE)
  (DIAMETER ?FEATURE ?D)
  (LENGTH ?FEATURE ?L)
  (INSTANCE-OF ?PART PART)
  (WEIGHT ?PART ?W)
  (TEST (OR (AND (>= ?W 0.5) (>= ?D 20) (>= ?L 10))
            (AND (< ?W 0.5) (>= ?D 10))))
  =>
  (ASSERT (BASE-TYPE ?FEATURE 2-POINTS))
  )
```

Fig. 4 A forward chaining rule

2.6 Interrogating object descriptions by backward chaining

For keeping descriptions compact and concise, we do not require that all details (actually slots) of features making up a part be fully specified at the very beginning. However, there is often a need to derive missing information from available data. E.g., if the location tolerance of a particular subfeature is not given explicitly, then it should be derived from the tolerance of its primary feature, or, if even this data is missing, then from the tolerance of a feature-pattern, or, as the last resort, from the value of the general tolerance assigned to the whole part.

ART has a backward chaining mechanism for creating such facts that may be required by partially matched rules. Once a pattern on the left hand side of a forward rule cannot be matched due to lack of some data, such missing data can be regarded as a goal, and backward chaining rules can be activated to supply these facts, either by transforming information stored in another form or by interrogating the user. Accordingly, part models can be kept as small as possible.

We have applied backward reasoning for analyzing the description of the part, i.e. for filling in details that had not been given in the original description but are needed at the present stage of problem solving (see Fig. 5).

```
                       Rule SEARCH-CIRCULARITY

(DEFRULE SEARCH-CIRCULARITY
         "looks for the circularity of a rotational (sub)feature"
 (DECLARE (SALIENCE *SEARCH-SALIENCE*))
 (GOAL (CIRCULARITY ?X ?))
 (NOT (CIRCULARITY ?X ?))
 (OR (AND (INSTANCE-OF ?X HOLE-GENERAL)
          (BELONGS-TO-PART ?X ?PART))
     (AND (INSTANCE-OF ?X SUBFEATURE)
          (SUBFEATURE-OF ?X ?Y)
          (INSTANCE-OF ?Y HOLE-GENERAL)
          (BELONGS-TO-PART ?Y ?PART)))
 (GENERAL-TOL ?PART ?TOL)      ;default is the general tolerance
 =>
 (ASSERT (CIRCULARITY ?X ?TOL))
 )
```

Fig. 5 A backward chaining rule

However, when using backward rules, there is a danger of futile deduction: as a matter of fact, the superfluous generation of goals can be stopped by specific means of ART that discriminate explicit facts from those that could be implied from facts already in the database. (By the way, there is another use of backward chaining when intermediate results produced by forward reasoning are checked by backward rules.)

2.7 Hypothetical and temporal reasoning

There are situations when planning must be pursued in several parallel directions by maintaining alternative hypotheses until some of them become infeasible. This situation originates from the fact that the structure of process planning problems, as produced along features of the part, rarely suggest a unique decomposition of the problem: due to feature interactions there might emerge several competitive interpretations of the same part, each one as valid as the other, but with different major consequences in terms of cost factors of the plans.

As we have pointed out (Váncza and Márkus 1992), for the purposes of process planning a domain theory is needed that allows causal reasoning about changes caused by manufacturing processes themselves. (E.g., if the planner sees that a hole H within slot S is to be made before milling slot S, it should be able to infer that, if made in this sequence, hole H is deeper than it is in the case when H is made after slot S. Similarly, when planning a milling process for the slot the planner should know, actually infer, whether a specific tool trajectory will cause a clash between the tool and other regions of the part.) By eluding an explicit and exhaustive representation of preconditions and effects of manufacturing processes, causal reasoning gives a handy opportunity not to hide laws of the domain. From the assumptions that (1) nothing changes unless it is caused by some factor, and that (2) cause always precedes effect, it follows that nothing changes until it actually has to change. Given a causal domain theory, manufacturing processes would trigger only initial changes on the part model and the causal rules of the world would govern all subsequent changes.

ART has a so-called viewpoint mechanism that is appropriate for exploring hypothetical alternatives and/or modeling situations that change with time. Information whose validity depends on specific hypothetical assumptions can be stored in viewpoints, too. A tree of viewpoints can be developed whose nodes represent distinct assumptions. A viewpoint can be discarded if its facts or their logical consequences are unacceptable or contradictory to each other: the so-called poisoning of such viewpoint deletes all descendant viewpoints. As another extreme, viewpoints that are not contradictory to each other may be merged into a single one.

The concept in ART dedicated to handling temporal information is the so-called extent of facts: assigned to a fact extents delimit the set of situations in which that fact is true. Viewpoints that keep track hypothetical dependencies of facts on the one hand, and extents that constrain the temporal validity of facts can be combined to form multiple-level viewpoints. As a matter of fact, this platform provides efficient means for non-monotonic reasoning, so we hope that the viewpoint mechanism of ART will support the construction of a full-fledged causal domain theory for process planning.

We have made experiments with the viewpoint mechanism of ART in order to find good combinations of reference surfaces for all applicable machining processes of the plan. The above problem has quite a few solutions to be found in a huge search space (Váncza and Márkus 1991). First results suggest that the viewpoint mechanism is indeed appropriate for this purpose, provided that one (1) can define strong enough constraints for poisoning unfeasible hypotheses, and (2) has sophisticated strategies for controlling the order in which hypotheses are generated, merged and discarded.

2.8 Integration of external processes

Rule based reasoning is suggested for tasks for which neither a single, nor an optimal solution is sought (Cooper et al. 1988). In rough terms, rules should define only a set of constraints which the final solution must conform. However, this style of problem solving is certainly inappropriate for handling the global optimization objectives of process planning.

There are stages of the planning process when engineering analysis is to be performed (e.g., when chains of dimensioning and tolerances are to be checked up or transformed). For dealing with such cases, pieces of procedural code are handy for formulating numeric algorithms. Fortunately, ART is smoothly embedded into the underlying LISP environment since it supports calling LISP programs on both sides of the rules. External programs on the left-hand side help to express further constrains for pattern matching that are beyond the capabilities of the pattern language. On the right-hand side any LISP programs can be evaluated, e.g. for making computations that supply further data to be stored in working memory. (As a matter of fact, passing variable bindings from rules to external procedures is not always the very best way to pass data, especially when large amounts of facts are concerned. Thus we have written transformation rules that build bridges between ART and external optimization programs; they work by generating ART data structures from LISP structures and back.)

3 How to learn the art of using ART

3.1 The ART way of pattern matching

The efficient use of a rule based system largely depends on whether its built-in pattern matching mechanism can do the bulk of the work by itself. This general statement, far from being a novelty (Brownston et al. 1985), is particularly relevant when programming in ART: compared with other tools for building knowledge based systems (Mettrey 1991, Mettrey 1992) ART has an outstanding capability for matching conditions of rules to the actual contents of the database.

By the way, if one starts ART with some logic programming background, it is better to forget the Prolog meaning of pattern matching at all: considering the *facts* of ART, patterns are lists, matching supports the use of single- and multiple field variables and wild-cards, augmented with the use of built-in and external predicates for constraining the values of variables. Considering *schemata* of ART, all this gets even more difficult and the parallel with Prolog pattern matching nearly disappears.

3.2 The procedural semantics of the ART rules

Although the well known but rather superficial doctrine of rule based programming says that rules should be used to capture the declarative knowledge of the application domain piece by piece, our experience suggests that in all but the simplest cases an additional, procedural meaning is attached both to the rules as seen one by one and to the whole set of rules of an application. Users usually consider that (1) both the conditions on the left hand side and the actions on the right are visited in their textual order, and (2) rules will fire before, together with, or after some other rules. While the first kind of expectations causes not much trouble with ART, the rules' ordering in time is a far more intricate issue.

First of all, not the rules are the atomic entities that should be related to each other: since the same rule may be used at different stages of the problem solving process again and again, objects to be sequenced in time are not the rules themselves but the activations of the rules. As a matter of fact, this difference is especially important in cases when problem solving consists of goal-driven stages mixed with forward chaining ones: having ended a long sleeping period, a forward chaining rule may start a new phase of activities as soon as some goal driven rule provides the facts that have been missing up to this point.

While writing a set of related rules, let they either be forward or backward chaining ones, one has to be attentive of the relative timing of their activations. The exact order of the activations is, however, hard to predict since it is influenced by several factors. Although some handles are offered to the user just for expressing control aspects (e.g. assigning a constant priority to the rules by the so-called salience values of rules), there are further, sometimes rather intricate factors that are not documented as control features of the system (maybe worst of all these factors is the order in which the rules are (re)declared and (re)compiled).

3.3 Hidden factors of control

The most important factor within the gray area of control is, as a matter of fact, just the conflict resolution strategy of the inference engine: nowhere in the manuals is it specified which one of the pending rule activations will fire, so the user may not know more than a statement that the activated rule must be of the highest salience present on the agenda at that moment.

In comparison with logic programming, the situation with hidden control factors of ART is quite interesting: Why may a Prolog user have a full control of the execution of his/her program, in spite of the fact that the underlying logic mechanism has no concept of sequencing conditions and rules? Why can the execution of ART not be defined by a meta inference engine, just as Prolog can simply defined by a meta interpreter?

Although we do not know answers from the authors of ART, our suspicion is that one should share preference among factors such as disciplined use of the rule based programming paradigm, or efficiency issues, or a business-like interest in hiding valuable implementation details. Another, more highbrow reason may be that leaving these control issues open (or at least, undocumented) enforces a kind of discipline on the user who has to adhere to a style that is thought as best for rule based knowledge representation. If rules are indeed separate pieces of knowledge, then their run-time relation belongs to the authority of the inference engine and not of the user. Accordingly, when the user has some specific course of actions in mind, it is better for him/her not to use rules for executing these actions but to call for traditional algorithmic tools. Since ART supports both starting its engine from another program and calling up non-ART programs from both sides of ART rules, this standpoint is hard to be questioned. On the other hand, one can not access, even in read-only mode, ART working memory through any other means than using rules. So the above argumentation can hardly be accepted as an ultimate answer: a duplication of the data (in one representation for ART, in another for the procedures whenever they need it) can be defended neither on the theoretical nor on the practical level. Accordingly, the gray area of control should be considered as a matter of efficiency and of the implementors' development and business strategy. As for efficiency, aspects of human and machine efficiency are nicely coordinated in ART and we claim that this coordination is a key factor of the success of ART. Accordingly, even if the integration of these two faces of efficiency have lead to a considerable loosening of the user's control over the system, the result may be worth the price.

As for hiding design details, this is again a matter of style: down to a level, near to uniform in depth across the whole system, users may see anything by using services of a friendly set of tools. However, anything below this level is strictly hidden so that users can not drive ART crazy or inefficient.

In addition, procedural (or, better to say, control) aspects of problem solving with ART can not be described even in terms of rule activations: activations are made in an autonomous way by the inference engine who chooses them from an agenda. Accordingly, if the user wants to have a feeling how ART works on solving his/her problem, he/she has to conceive the changes of the agenda.

3.4 Conventional programming constructs versus rule interactions

In traditional programming languages well-proven control cliches provide the means for (1) coercing the sequences of computing steps (conditionals, cycles), (2) avoiding interaction between parts of the code that should remain unrelated and (3) writing similar code only once (procedures). All this together makes the problem-solving process more tractable and comprehensible to humans, and, at the same time, more efficient in machine terms.

However, in case of rule based programming the role of control cliches and interaction among the pieces of code is just reversed: we can't help but try to implement the above cliches by rule interactions. (An interaction between two activations happens when the order in which the rules fire results in a difference of the result of these actions. Activations interact either directly, if an activation asserts or retracts an element of the database that is a precondition of another matching, or indirectly, through modifying the sequence of the instantiated rules waiting on the agenda.) Actually, for this purpose there are no other means in our hands; e.g. if we want that certain rules fire in a predetermined sequence then we have to distribute this information of ordering among the rules concerned.

In other words, programming in ART largely disables the use of our conventions for expressing the control of programs. No wonder, questions emerge whether we really need these programming cliches and/or what can rule based programming offer instead of them.

3.5 Pattern matching and control

To begin with a simple example, let's consider iteration: it is needed whenever the extreme of some similar elements is looked for (e.g. one needs to find the deepest of the holes on a face of the part). Supposing that no results of a previous investigation have been stored, object(s) with the extreme value can be found only by visiting and comparing all candidates. Accordingly, if the inference engine does not provide a wired-in solution, then there is no other choice than the search cycle implemented manually.

Furthermore, what to do if there are more than one objects with the same extreme value; e.g. there can be found two holes of the same, maximal depth? Should the rule referring to these objects fire immediately after each other as many times as many instantiation it actually has? Indeed, such a regime could be regarded as the most natural extension of selecting from among activations; but what to do if the firing with the first of the equivalent extremes results in actions that destroy conditions of the next rule activations? Anyway, even this most simple thought experiment could suggest that extending the power of the rule syntax and providing more fixed constructs, e.g. for iteration, may easily lead to messy situations; accordingly, the use of hand-made cycles may be more safe, as far as the outcomes of using such constructs are easier to browse and debug.

Summing up, the powerful pattern matching causes no troubles as long as the user can imagine all the situations he/she can ever meet while running the program. Beyond this point, ART presupposes a working knowledge of classical data structures and computing algorithms, as well as skill in the use of traditional languages, especially LISP. No wonder that a widely used introduction to rule based programming (Brownston et al. 1985) regards mastering of basic computer-science concepts covered in (Wirth 1976) as a prerequisite of mastering rule based systems.

What can we do on a higher level of abstraction of representing declarative knowledge, i.e. when dealing with schemata, with multiple levels of viewpoints etc.? After this first year, we can't say more than it is better to shadow prior knowledge, to begin with a *tabula rasa*, as far as concepts and techniques of traditional programming are concerned. The double view may cause serious conflicts and is a source of perplexity.

3.6 Will rule based systems deliver new control structures?

Up to now, there is no widely accepted choice of control structures suited to complex rule based systems. A technical reason might be that each rule based system has its own version of pattern matching and a strategy, or even more ones, for choosing the next rule firing from the agenda. A control concept that is good for one system may be inefficient, unclear if used with another version of pattern matching and firing strategy.

In our opinion the basic contradiction lies deeper, between the global and hierarchical nature of the conventional control structures versus the fact that, as for rule based systems, control should be implemented in a distributed manner, in a medium that has no conceptual mechanism other than that of rules.

Although the way out from this situation may lead towards handling the agenda in novel ways, our immediate aim is to have a better understanding of control in rule based systems and to develop a transparent style of programming through working only with specific rule interactions. The application of rule interactions for implementing typical control structures should be elaborated case by case, within each environment.

4 Directions of further work: knowledge compilation or ART as a delivery system

Running on a Symbolics 3620 with Genera 7.2., ART is now being used as a tool for generating an automated process planning system. At a later stage of the project, for everyday practical use our results are to be delivered to more traditional and simple computing environment.

We deem the task of rewriting the prototype process planning system into a form that is executable on a simpler computing platform unfeasible. This skepticism is grounded by the following facts: (1) a good deal of expertise is captured by the patterns of rules that heavily exploit the powerful pattern matcher of ART, (2) control of the whole program is distributed among interacting rules, (3) most rules are senseless outside the context of some other rules, and, finally, (4) the genetic algorithm performing plan optimization requires large enough dynamic memory and high speed of computation. Due to the first three reasons (those that might be common to most ART applications), the re-implementation of even a less competent version of the prototype system would be extremely difficult.

Instead of rewriting the prototype system by hand-coding, now we are looking for automatic methods for picking up and putting together those fragments of domain knowledge that may bear relevance to the solution of a particular class of the planning problem. Fortunately enough, the problem domain encourages the use of a method known as knowledge compilation (Goel 1991). In our cast of the method, given the model of a manufacturing system together with the local manufacturing practice and the set of its products, the question is how the system's production can be improved by taking advantage of the similarity of the parts and technologies. A well-established approach leads through working out so-called group technologies: similar parts are collected into groups, each of which will have its so-called group technology. In case when a new part arrives, its process plan will be generated through the part's classification into one of the groups and by adapting the corresponding technology.

The main difficulty with generating group technologies is caused by the incomplete and conflicting nature of available domain knowledge, the intermingled relations bound both to the production environment and engineering practice, and to the particular solutions of earlier tasks. This problem can be approached as formation of concepts and theories by means of symbolic learning: departing from empirical facts and a domain theory, one should create a representation of the pieces of knowledge that is adequate with the domain and, at the same time, can be used efficiently.

Accordingly, our aim is a learning system that is able to create group technologies for the families of parts, based on individual part and technology descriptions, and the linked representations of parts, process plans, tools, machines, and manufacturing processes. While inductive, similarity based learning methods should be used to find shared features and technologies, analytic methods should refine the plans to the right level of specificity and abstraction. Final results are to be delivered for other, more conventional computing platforms where they should be able to work independently both from the original, general-purpose process planning and the learning components.

Acknowledgement

The authors wish to thank M. Horváth for many helpful discussions on process planning issues. This research was supported by the National Research Found of Hungary (OTKA), grant No. 412.

References

1. ART Reference Manual, Inference Corp. 1988.
2. L. Brownston, R. Farrel, E. Kant and N. Martin, Programming Expert Systems in OPS5, Addison-Wesley, 1985.
3. B. D. Clayton, ART Programming Tutorial, Vol. 1-4, Inference Corp. 1987.
4. T. A. Cooper and N. Wogrin, Rule-based Programming with OPS5, Morgan Kaufmann, 1988.
5. A. K. Goel, Knowledge Compilation, *IEEE Expert*, April 1991, 71-73.
6. W. Mettrey, A Comparative Evaluation of Expert System Tools, *Computer*, Vol. 4 No. 2, 19-31, 1991.
7. W. Mettrey, Expert Systems and Tools: Myths and Realities, *IEEE Expert*, February 1992, 4-12.
8. J. Váncza and A. Márkus, Genetic Algorithms in Process Planning, *Computers in Industry*, Vol. 17., 181-194, 1991.
9. J. Váncza and A. Márkus, Features and the Principle of Locality in Process Planning, to appear in *Int. Journal of Computer Integrated Manufacturing*, 1992.
10. N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

ARTIFICIAL INTELLIGENCE -
TOOL BUILDING ASPECTS

Chair: P. Zinterhof

Architectural Considerations for Extending a Relational DBMS with Deductive Capabilities[1]

Michael Dobrovnik, Roland T. Mittermeir

Institut für Informatik
Universität Klagenfurt
Universitätsstraße 65-67
A-9020 Klagenfurt, Austria
e-mail: {michi,mittermeir}@ifi.uni-klu.ac.at

Abstract

This paper describes the development rationale and the architecture of a prototypical expert-database system. Knowledge processing capabilities of SQL were enhanced by extending the language by recursive views. This work is based on an evolutionary approach; smooth integration with the base language was an important development aim.

After a discussion of the main design alternatives, the architecture of a prototype is presented. Finally the progress of the project is described and possibilities for further extension are indicated.

1 Recursive Views

1.1 Motivation

A host of modern applications demand knowledge processing capabilities in combination with the support of large scale volume data processing capabilities and multi-user support for concurrent access and flexible combination of persistent information as provided by todays data base systems. But classical expert system shells lack important features needed in conjunction with bulk transaction processing, support for persistence, and integrity preservation over long spans of time. Hence, systems supporting multi-paradigm applications become increasingly important.

[1]The work on this project was partly supported by the Austrian Fonds zur Förderung der wissenschaftlichen Forschung under Contract P6772P.

At the time this project started, various options to achieve the above aim have already been proposed in the literature (see e.g. [Gall81, Gall84, Brod86, Kers86, Wied 86]). They can be classified into three broad categories:

- extensions of logical programming languages or expert system shells by appropriate permanent storage management (back-end storage management);
- development of database systems with "logical" query languages;
- extensions of database systems by "reasoning facilities".

In the project on which we are reporting here, the latter approach had been adopted. However, we wanted to follow this approach in such a way that we could fully build SQL's high acceptance in the marketplace. To achieve this aim, a solid formal definition of certain SQL features became necessary before searching for an adequate linguistic and architectural design of such an extension. While the formal aspects have been reported already, this paper presents the architectural considerations which guided this project.

The choice for this approach has been founded on the consideration that relational database systems enjoy high penetration into a host of application areas. One reason for this success surely is the widespread use of standard database languages such as SQL [SQL86,Date87]. SQL can be characterized as an end user oriented, mainly declarative language which plays a central role in the database field, even in spite of its well known deficiencies [Date87].

One of the most important restrictions of SQL is its lack of computational completeness [Aho79]. So, an important class of systems such as knowledge based systems or decision support systems, but also technical systems demanding special search characteristics [Boud92], are not well supported. A particular reason for this deficiency is that recursive problems cannot be adequately attacked by means of standard SQL. But recursion plays an important role in deductive systems. Two of the most prominent textbook examples for this class of problems are path problems and bill of material calculations. Hence, the main idea of the XPL*SQL-project was to extend the capabilities of SQL in such a way that the extended language provides good support for a broad range of recursive problems.

The linguistic mechanism we needed for obtaining our aim was the well known view mechanism. It allows to create virtual relations by declaring a rule that describes how to compute them. The view construction mechanism has been extended to support **recursive views.**

1.2 General Transitive Closure

The transitive closure T of a relation R is defined as [Eder90a]:

$$LFP(T = union(R, COMP(R,T)))$$

COMP means composition and is an equijoin where the join-attributes are eliminated by projection. The least-fixpoint operator *LFP* evaluates T to the smallest set, for which the equation is valid.

To demonstrate this concept, let us consider a binary relation *flight(from,to)*, which associates cities that can be reached with one single flight. This relation clearly is transitive, so it makes sense to compute the transitive closure *connection* of *flight*, which contains all flight connections between two cities, formally:

$$LFP(connection = union\ (connection,\ flight \bowtie_{flight.to=connection.from} connection))$$

It has to be pointed out, though, that the concept of transitive closure of a relation may not contain any attributes pertaining to the specific association just established. E.g. in the example, it is not possible to total the distance or the duration of connections. Certainly, this is a main disadvantage of pure transitive closure and makes it unsuitable for a large class of applications. Therefore, the concept was generalized [Eder90a,Eder90b] in the following way:

$$LFP(GT = union(R,COMPEX(R,GT)))$$

There, *R* is a base relation as before, *GT* is the generalized transitive closure. The main difference between transitive closure and general transitive closure lies in *COMPEX*. *COMPEX* stands for composition-expression and is a selection on the carthesian product of *R* and *GT*, combined with a projection which may also contain arithmetic expressions. The introduction of this composition-expression allows the definition of attribute values as computable functions, whereas the generalization from the equijoin to a selection on the carthesian product allows to formulate non-trivial conditions for linking tuples. An example for general transitive closure will be given in a subsequent section.

1.3 Integration of Generalized Transitive Closure into SQL

General transitive closure is a special form of a linear recursive deduction rule. When one considers SQL, there is a mechanism which allows for the definition of derived relations, which are better known as **views**. A view is a virtual relation whose extension is computed according to a declarative specification, the view definition, which can be seen as a deduction rule. Whereas one could argue that from such a perspective, SQL is a language with deductive components, there is one main shortcoming of views in standard SQL. The language explicitly forbids to reference the view to be defined in the definition part itself, i.e. recursion is not permitted.

Considering the fact, that views can be interpreted as nonrecursive deduction rules, and that views are a well understood feature of SQL which is broadly used in practice, it seems to be promising to extend the view concept and to explicitely allow the definition of **recursive views**. This evolutionary approach not only integrates very well with the basic language, it has as main advantage, that it does not require any change in the application pattern. Neither a user querying a view, nor any special tool (application generator, report writer, ...) using those views, need to take special consideration as to whether a view is defined recursively or in the usual way.

However, there are some minor deficiencies one has to bear in mind using recursive views. In general, recursive views may not be updated, queries on them can take longer to complete than on conventional views, and the results of a query may be infinite. While the first and second points are inherently connected with recursive views, the possibility for infinite results requires special treatment (see [Eder90a]).

Nevertheless, besides increasing the expressive power of the language, this specific approach meets some important criteria for extending a language [Mitt88]. The principle of recursive views is easy and safe to use and it incorporates a minimal number of new constructs. The new feature is orthogonal to existing language elements, it can be formally described, and it can be optimized to some extent.

1.4 Syntax of Recursive Views

The syntactical extensions of the definition of SQL are mainly captured in one single place, namely the *recursive-view-definition-statement* which is presented (in a slightly simplified form) in Figure 1. Other aspects of the language, notably the select statement, remained unchanged.

A simple example of the application of the new construct can be found in the appendix. Now we briefly give an informal description of some of the nonterminals mentioned in Figure 1. For a more thorough treatment, we refer to [Eder90a, Eder90b, and Dobr91].

The *attributed-column-list* extends the standard *column-list* of SQL. With *INC* and *DEC* respectively, the specification of monotonous characteristics of certain attributes is allowed. This information is crucial in optimization and assuring the finiteness of certain queries. The *set-type* specifies, whether a certain view should be treated as a set-relation, having only distinct tuples and where duplicates have to be eliminated, or as a multiset-relation, where duplicate tuples must be taken into account.

It should be noted that recursive views can be used as targets of queries like any other table or conventional view (with some minimal restrictions, see [Eder90a]). As small as the syntactical extensions to standard SQL for the definition of recursive views may be, the possibility to use recursive views in virtually all contexts where ordinary views are permitted implies that fundamental changes in the SQL-interpreter must be made.

```
statement ::=  ... |
               create-view-statement |
               create-recursive-view-statement |
               ...

create-view-statement ::=
               CREATE VIEW viewname [ (column-list) ]
               AS SELECT [ set-type ] select-list
               FROM table-reference-list
               [ where-clause ]
               [ group-by-clause ] [ having-clause ];

create-recursive-view-statement ::=
               CREATE VIEW viewname (attributed-column-list)
               AS [ set-type ] FIXPOINT
               OF table-name [ (column-list) ]
               BY SELECT select-list
               FROM table-reference, view-reference
               where-clause;

attributed-column-list ::=
               column-name [ INC | DEC ] [, attributed-column-list ]

set-type ::=
               ALL | DISTINCT
```

Fig. 1: Syntax Extension

2 Considering Architectural Alternatives

The main design variants we investigated have been to build an entirely new system completely from scratch, to integrate the new functionality into an existing system, and to construct an add-on or a frontend to an operational system. We will weigh these alternatives against each other in the sequel.

In deciding on the architectural alternative to be pursued for the proposed extensions, we considered technical as well as economic aspects. The reasons for considering technical arguments need no further explanation. The economic aspects have been considered in spite of us being located at a university institute. Since our research is mainly sponsored by governement money, we considered it important that its results would be at least in principle exploitable by some local software producer or software house without placing undue risks on the developer of customer of such a system.

2.1 Build from Scratch

The design and implementation of a new DBMS, which supports the concept of recursive views would not only be challenging, but would also offer a wealth of advantages:

* No restrictions from existing systems would have to be taken into account.
* The whole system could be constructed with special considerations to the deductive component and its implications.
* The recursive views would be deeply integrated into the DBMS (Fig. 2).
* The highest degree of optimization and, hence, highest performance, would be possible.
* One single interface for tools and application programs could be defined and the tools provided could support the complete language.

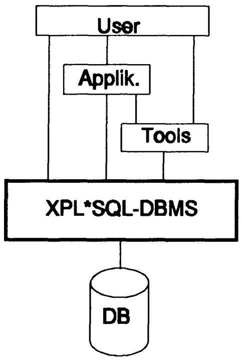


Fig. 2: Build Totally New System

The main drawbacks of this approach are the extremely high costs and the long development time that would be needed to build a DBMS totally from scratch. A great deal of the effort would be used for the design and implementation of functional aspects, which would have been only of subordinate interest in the given context. These aspects have been particularly important in our design considerations. Not only, that we didn't feel in a position to acquire the ressources for a full fledged development of an operational knowledge-base management system which would show all properties of a modern database system. We have even been sceptical about our own greediness, which might arise from good ideas in several directions off the mainstream line of thought, endangering the project to result in a never ending venture.

Besides these aspects, several aspects which might stem from the particular economic context (small country with moderate DP-industry only) in which our university is placed were considered. There is no large scale international vendor of data base systems around. Hence, the acceptance of a system based on a full integration of the database and knowledge-base aspects of the system with managers responsible for the applications to be supported by this system would have to be projected as being very low. The risk, that the developer of such a huge system might not survive would probably be too high for a responsible DP-manager.

Further, the evolutionary idea behind the construct and the language extension would be reduced to the appearance of such a system to the user (investment in training and education), since changing the vendor of one's DBMS would rather have the flair of a revolution than that of a smooth change in most of the cases.

2.2 Extending an Existing System

The internal extension of an existing system, which is well established in the market, has a much higher degree of potential for success. In contrast with the development of a totally new system, this approach poses major restrictions on design decisions, because of the high amount of investments in the basic SQL-DBMS, which must be protected. Yet it is possible to construct and present a uniform interface for users, application programs and tools. The integration of recursive views into the system and the supporting tools could be quite strong (Figure 3).

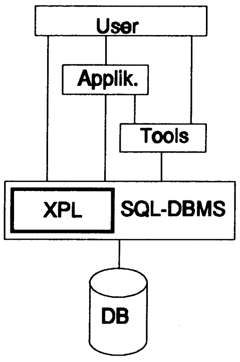


Fig. 3: Embedded Development

The extension based development would allow for moderate costs. It would also have a much higher acceptance in the market, because it would not look like a major change in the computing environment. The impact of such a system could be compared to that of a new release of a DBMS, just incorporating some (very nice) new features. However, one has to see very clearly, that such an argument would be deceiving, since the coupling between the extensions and the base-DBMS would have to be so tight, that with most modifications (new versions) of the base DBMS, a new version of the XPL-extension would also have to be supplied. This, however, would also require not only the adequate economic resources but also very intimate contact between the developer of the DBMS and the developer of the expert system extensions.

The main disadvantage of this kind of extension is that the developper of the extension must have full access to all internals (source and documentation) of an existing DBMS, and that one would have to constantly adapt the extension to the new releases of the database system itself, which usually would mean that if the developer of the extensions is not also the developer of the base system itself, he would be heavily dependend on him.

2.3 Add-on to some Existing System

This alternative form of enhancement of a DBMS is implemented in the same way as every other application program (Figure 4). Therefore, (virtually) no knowledge of the underlying DBMS internals is required.

This variant has a lot of disadvantages, if seen from a solely technical point of view. The uniform interface to other application programs and the possibility to make use of the language extension in the tools supplied with the DBMS must be given up. Further, the user has to make right from the beginning a choice, whether working with XPL or with pure SQL is needed. An awkward consequence of this choice would be that in cases, where recursive views and base views have to be used concurrently, the results of the recursive views would need to be materialized and explicitly transfered into the "ordinary" database management system, or the add-on has to be powerfull enough to process also data contained in the conventional data base of stored facts. This later option would require however full SQL capabilities and, hence, would lead us to the fourth option.

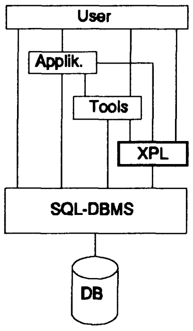


Fig. 4: Add-On to Existing System

2.4 Frontend to an Existing System

The merits of this option become directly visible, when considering the shortcommings of the adds-on alternative. Here, we do not consider the extension to be just an add-on where the clients (user, application programs and tools) have to switch between the base system and the enhancement. We rather assume it to be a real front end, allowing the clients to access the system in a completely transparent way (Figure 5).

The advantage of this solution would be - like with the previous case - that it could be implemented and maintained with comparatively moderate effort. Further, the interfaces to both, the data base management system it utilizes underneath, as well as to applications and tools would be clear cut. Therefore, no severe dependence between the developer of the DBMS and the developer of the XPL-extension would come up. Hence, even in the economic and institutional environment in which this development had to be undertaken (and for which it had been targeted), this approach seemed feasible.

Of course, there is also a price to be paid for such an architectural decision: Any SQL statement needs to be first analyzed by the XPL system and in case it is an "ordinary" SQL statement, the same analysis has to be repeated within the DBMS itself. Given the predominant structure of SQL-statements, this overhead would be marginal though. Hence, performance surely will be suboptimal due to the partly duplicated execution of operations and due to the coarse tuning of the frontend with respect to internals of the base system. Additionally, main components of the SQL-DBMS must be reimplemented (in a simplified form) in the frontend itself.

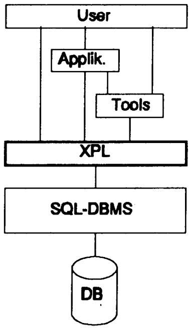


Fig. 5: Frontend to existing System

Despite the shallow integration of the frontend, it will not be completely independent from the SQL-DBMS and it will also not be portable per se, since the (highly implementation specific) catalog of the underlying system must be accessed.

From a broader perspective, however, this model doesn't look so bad as stated above, especially if one considers the possibility to market it as a special "preprocessor". This poses absolutely no hidden risk for potential customers. They can continue to use their existing DBMS, existing applications are totally unaware of the extended functionality whereas new applications can make instant use of the frontend. Since the development costs for the frontend itself can be held at a relatively low level, it would also be affordable.

This model also allows for real third-party development of the system in contrast with the internal extension of an existing system. Besides the fact that the specifications of a DBMS's external interfaces are publicly available, they also tend to be relatively stable, as compared to internal interfaces. Further, the evolutionary risk is reduced by the fact that new releases of systems are usually upwards compatible. Hence, even if the developer of the frontend cannot keep pace with the developer of the main system, the detrimental effects on the applications will be limited.

3 Architecture of the Prototype Actually Implemented

In this section, we sketch the architecture and the components of the implemented prototype, which is a frontend to an existing system (Figure 6). This decision is based on several reasons. First, we had no access to all internal information of an existing DBMS which would be necessary to extend it. Second, we had no intention to put much effort into components which are not in the center of our interest. Further, we didn't feel in the position to develop YADE (yet another database environment) and to become another DBMS vendor.

The aim of the prototype was to provide an extended SQL-based command interface, which allows one to define and query recursive views in addition to the functionality of standard SQL, and which can be used for further study.

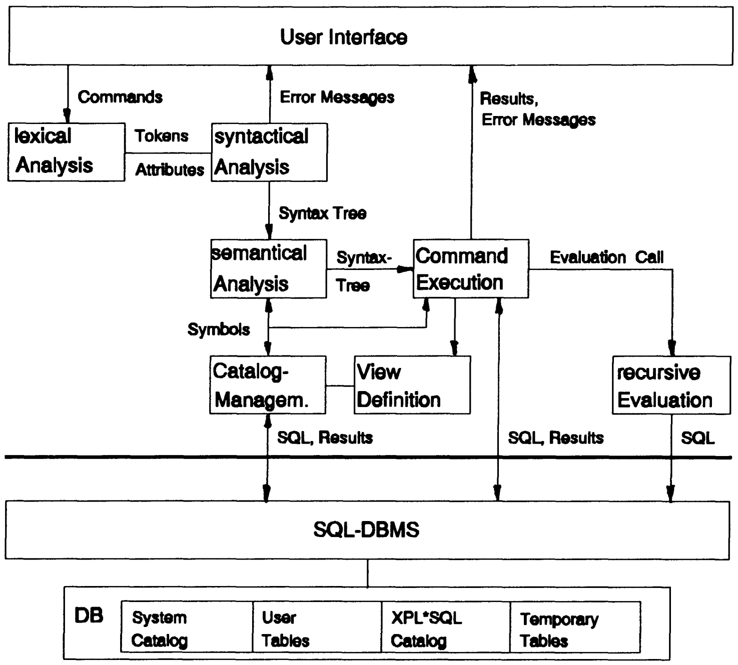


Fig. 6: Architectural Overview

The user interface component consists of a very simple line editor, which can be used to enter the extended data definition and data manipulation commands, and a rather rudimentary formatting capability for query results. Error messages are also displayed through these components. The user interface is solely character based.

All SQL commands coming from the user interface are fed into a lexical analyzer which transforms the commands from the textual form into an attributed stream of tokens.

This stream of attributes and tokens is the input for the parser. This component analyzes the stream for its syntactical correctness and constructs a syntax tree representing the structure of the command.

The semantic analysis component processes the syntax tree and extends it with new attributes. Here, not only name resolution of database objects (tables and attributes) by means of queries performed by a catalog component is carried out, but also the semantic correctness of the command is checked (at least to a certain degree).

The command executor decomposes the (possibly complex) command into smaller units, which can be executed in isolation from other units. Each unit is classified, whether it references a recursive view or just makes use of standard tables and views. If recursive views are referenced, termination and efficiency become key issues. To allow for the broadest set of safe applications [Eder90a], we check what expressions can be propagated into the computation. What is to be propagated is determined in a special part of the command executor. The thus rearranged statements are then ready for recursive evaluation. The results of this evaluation are stored in temporary tables, which are maintained by the base DBMS.

Knowing the temporary tables just computed, the command executor reconstructs an SQL-statement from the syntax tree. This SQL-command may not only be just a part of the initial SQL-command, it may also differ from it. This difference is due to the fact, that the names of the recursive views have to be substituted by the names of the temporary tables which contain the evaluation results of the recursive views referenced. The modified statement will be evaluated directly by the SQL-DBMS. Results and error messages are sent to the user interface component.

Note, that up to this point in the analysis process, all SQL-statements need to be analyzed, regardless of whether they do define or reference recursive views or not. The user does not need to switch between two different systems, and the extension is totally transparent to him.

The catalog management component updates the symbol table, based on the information contained in the system catalog of the underlying SQL-DBMS as well as in a special catalog which is used solely for the storage and retrieval of the definitions of recursive views and their corresponding attributes.

The view definition component computes the attribute dependency graph [Eder90a], which is used to classify the attributes of the view. This classification information together with the view definition is stored in the special catalog tables.

The recursive evaluator implements the algorithms to compute the results of recursive views [Eder90a, Eder90b]. It uses information from the special extendend catalog (XPL*SQL-catalog) and those constraints of the query at hand which can be propagated. The schema information concerning the relevant temporary tables is passed as a parameter to the recursive evaluator.

4 State of the Project

Currently, the implementation of a first version of the prototype is finished. It builds on the ALLBASE DBMS, running under HP-UX. It allows to interactively define recursive views and to query a database including tables, regular views and recursive views. Its actual design and implementation took about six person month.

As extensions, we forsee that the prototype could be extended to offer a programming interface allowing application programs to use the enhanced abilities of the system. A lot more of semantic checks could be added and performed in the frontend itself. This would allow for the detection of a large number of errors early in the interpretation process; errors could thus be catched before a lot of time is consumed by the evaluation of recursive views. This computation could be enhanced further by incorporating the propagation of additional kinds of restrictions into the evaluation process.

Further work will include adapting the frontend to other DBMSs and to integrate further extensions, namely extreme-value selections and aggregates. There are also plans to make use of the enhanced functionality in the context of a software engineering environment, which demands the ability to define and to use recursive views.

5 Assessment

The choosen architectural variant was adequate and allowed us to concentrate mostly on the new and specific aspects of the system without forcing us to deal with lots of internals of existing DBMS or tons of (unavailable) documentation. It was possible to demonstrate major aspects of the concepts reported in [Eder90a, Eder90b] and to substancially increase the expressive power of a relational DBMS with a rather limited effort.

We conclude that this architectural variant may be well suited when development takes place under the assumption of a third party producer with limited resources. It poses few risks, because it guarantees the highest possible independence from the vendor of the basic DBMS, and promises rather short development time with moderate cost.

Appendix:

Example of General Transitive Closure

Consider a relation *direct* with the following schema

direct(from, to, km, mins, hops)

where each of its tuples represent a direct flight which starts in city *from* and is destined to city *to*. The distance and duration of the flights are recorded in columns *km* and *mins*. The attribute *hops* contains the number of intermediate landings, which is zero in all tuples of relation *direct*, since we are considering direct flights only.

The following definition of a recursive view computes all possible flight connections between all pairs of cities, summing up distance, durations and number of hops:

```
CREATE VIEW connection (from, to, km INC, mins INC, hops INC)
AS FIXPOINT OF direct
BY SELECT d.from, c.to, d.km + c.km, d.mins + c.mins, c.hops + 1
FROM direct d, connection c
WHERE d.to = c.from;
```

This view can be used as a query target like every other table or conventional view (with some minimal restrictions, see [Eder90a]). A more complex example of an application of recursive views in the context of CPM-charts can be found in [Dobr91].

References

[Aho79] A. Aho, J. Ullmann: "Universality of Data Retrieval Languages", ACM Symp. on Principles of Programming Languages, 1979, pp. 110-120

[Boud92] N. Boudriga, A. Mili, R. Mittermeir: "Semantic Based Software Retrieval to Support Rapid Prototyping", Structured Programming, Vol. 13, No. 3, 1992

[Brod 86] Brodie M.L., Mylopoulos J.: "On Knowledge Base Management Systems", Springer Verlag, 1986

[Date87] C.J. Date: "A Guide to the SQL Standard", Addison-Wesley, Reading, 1987

[Dobr91] M. Dobrovnik: "IXPL*SQL. Erweiterung der Abfragesprache SQL um rekursive Views", Diplomarbeit, Institut für Informatik, Universität Klagenfurt, Klagenfurt, 1991

[Eder90a] J. Eder: "Extending SQL with General Transitive Closure and Extreme Value Selections", IEEE Transactions on Knowledge ans Data Engineering, Vol. 2, No. 4, Dec. 1990, pp. 381-390

[Eder90b] J. Eder: "General Transitive Closure of Relations containing Duplicates", Information Systems, Vol. 15, No.3, 1990, pp. 335-347

[Gall81] Gallaire H., Minker J., Nicolas J.-M.(eds): "Advances in Database Theory", Plenum Press, 1982

[Gall84] Gallaire H., Minker J., Nicolas J.-M.: "Logic and Databases: A Deductive Approach", ACM Computing Surveys, Vol. 16/2, June 1984, pp. 153 - 185.

[Kers86] Kerschberg L. (ed).: "Expert Database Systems", Benjamin/Cummings, 1986

[Mitt88] R.T. Mittermeir, J. Eder: "XPL*SQL. Research on new AI-Languages", Proc. 6th European Oracle User's group conference, Paris, April 1988

[SQL86] Database Language SQL, Document ANSI X3.135-1986

[Wied 86] Wiederhold G.: "Knowledge and Database Management", IEEE Software, Vol. 1/1, Jan. 1984, pp. 63 - 73

FuzzyExpert: A Case Study in PC-Based Expert System Development

Jan Žižka
Computer Center, Brno Technical University
Údolní 19, 602 00 Brno, Czechoslovakia

Abstract. Like many other complex software products, expert systems are leaving their original hardware platforms – mainframes and minis. In particular, the fuzzy set theory-based expert system *FuzzyExpert* was developed for the personal computer (PC) environment using various integrated paradigms. However, as the experience described in this paper indicates, the process of downsizing encounters many problematic issues. For the hardware base, *FuzzyExpert*'s developers chose the *IBM/PC* compatible, but this environment presents memory-related constraints. To circumvent these problems, *FuzzyExpert*'s developers employed a virtual memory mechanism. Software issues primarily concern performance, derived from the absence of multitasking in *MS-DOS*. As a solution to this problem, the system uses a preempting technique. This paper further presents principles of *FuzzyExpert*'s user interface, which is based on object-oriented programming.

1. Introduction

Artificial intelligence and, especially, the area of expert systems (ES) has progressed in a relatively short time from an academic discipline to a commercially viable technology. Expert systems offer the opportunity to organize human expertise and experience into a form that the computer can manipulate. However, much of human knowledge is incomplete, imprecise, approximate, or subjective. Consequently, conventional method-based computer modeling of many non-numeric problems does not provide satisfactory results. With improvements in problem solving tools, expert systems now represent an alternative programming model, yet the technology is complex and not easily mastered. Successful adoption of an expert system as a practical, useful tool depends on several important features, which constitute today's widely recognized expert system paradigms (Payne and McArthur 90; Giarratano and Riley 89):

- suitable knowledge representation;
- user confidence in the system's conclusion;
- high speed of execution;
- appropriate user interface.

To satisfy such needs, expert system developers must possess adequate hardware and software tools. The following sections describe one experience with developing a PC-based expert system, *FuzzyExpert*, which processes vague knowledge. *FuzzyExpert*'s development team strove to create an expert system efficiently running on standard *IBM/PC* compatibles under *MS-DOS*, equipped with a friendly user interface, and providing as simple knowledge and fact representation as possible. The system has been, above all, intended for users who need to experiment with fuzzy knowledge bases before they implement particular applications, such as fuzzy process control, decision-making systems, diagnostic systems, empirical research processing, etc. Aside from these application areas, *FuzzyExpert* can be used for knowledge base tuning (e.g. reducing sets of rules to a necessary minimum), for testing correctness and completeness of knowledge bases, or simply as a training tool. The developers started with a fuzzy set theory-based prototype originally developed on a mainframe. With the complete change of hardware and software environments, the team had to sort out many problems.

2. *FuzzyExpert*'s Fundamentals

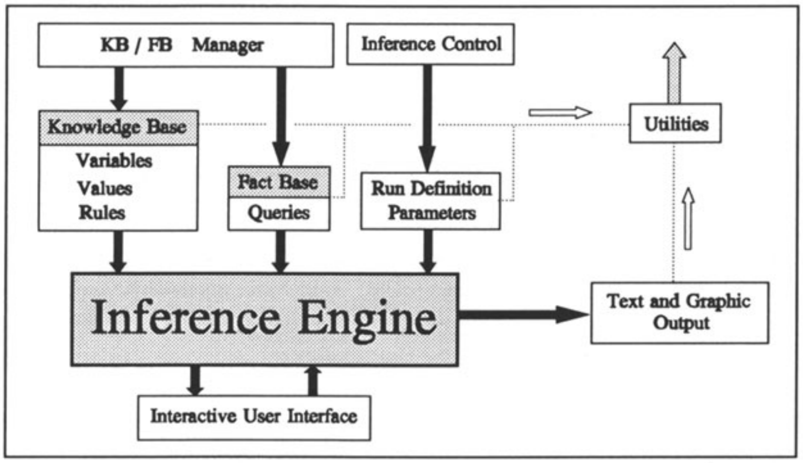


Fig. 1. *FuzzyExpert*'s general architecture

FuzzyExpert is a rule-based expert system supporting approximate reasoning based on fuzzy set theory (Zimmerman 85). Fig. 1 shows the basic components of the system. Besides the core (i.e. the inference engine, knowledge base [**KB**], and fact base [**FB**]), several additional constituents are integrated within the expert system. ***Knowledge Base/Fact Base Manager*** assists knowledge engineers and users in creating rules and queries. Moreover, it checks data integrity inside individual KBs and FBs as well as between a KB and its related FBs. ***Inference Control*** enables the inference engine to run with various parameters. ***Interactive User Interface*** supports communication between the user and the system during computation. ***Utilities*** provides, for example, file management, report generation, and data format conversion. The following sections describe these components in more detail.

2.1. Knowledge and Fact Base

The system's inference engine processes two input data sets:

– ***rules***, which are stored in a knowledge base;
– ***queries***, which represent a base of facts.

Setting up KBs is the task of knowledge engineers who must transform experts' knowledge into computer-acceptable data. These data represent a certain reality, described with various linguistic attributes (variables).

FuzzyExpert enables its users to define attribute values as fuzzy sets. [A fuzzy set is defined, in turn, using a membership function that assigns a value $\mu(\mathbf{x})$ to each coordinate $\mathbf{x}$ within a universe $\mathbf{U}$ ($0 \leq \mu(\mathbf{x}) \leq 1$).] As shown in Fig. 2, nine predefined shapes of the membership function serve to represent particular attribute values, allowing the system to model both crisp and vague **linguistic values**. In practice, these shapes prove to be sufficient. To define any value, the user must select one of the shapes that is suitable for a given case, then specify the fuzzy set's location on its universe with 1 to 4 breakpoints. When the user wants to express the value "I do not know" or "it does not matter" for one or more values, the rightmost shape in Fig. 2 accommodates this need; no breakpoints are necessary because the value is defined on the whole universe and it has no influence on the result.

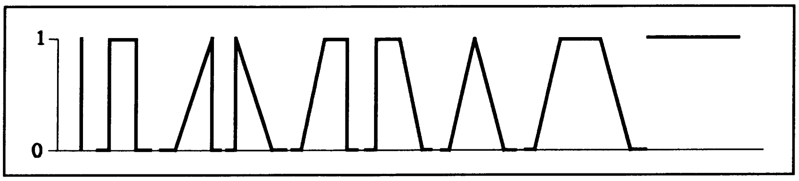


Fig. 2. Predefined shape of the membership function

Any effective combination of attribute values can make up a rule, which takes the form of an ***IF–THEN–ELSE*** clause. Attribute values in rules are usually assigned by a knowledge engineer as a result of the process called "knowledge acquisition". To create a new fuzzy KB, the knowledge engineer must complete several steps in the specified order:

1) defining all linguistic attributes that describe the problem modeled;
2) detailing the output attribute;
3) assigning linguistic values to each attribute;
4) making up rules as combinations of linguistic values.

A **rule** can be formally introduced in the following way:

Let $\mathbf{x}_j \in \mathbf{U}_j$ (j=1,2,...,n) denote an independent attribute taking its linguistic (fuzzy) values $\mathbf{A}_{ij}$ from a universe U_j. Let $\mathbf{y} \in \mathbf{U}_B$ stand for a dependent attribute defined on a universe $\mathbf{U}_B$; furthermore, let $\mathbf{B}_i$ mean a fuzzy value defined on the universe $\mathbf{U}_B$. Then, the following clause:

$$\mathbf{R}_i \equiv \textbf{\textit{if}}\ \mathrm{x}_1 = \mathrm{A}_{i1}\ \textbf{\textit{and}}\ \mathrm{x}_2 = \mathrm{A}_{i2}\ \textbf{\textit{and}}\ \dots\ \textbf{\textit{and}}\ \mathrm{x}_n = \mathrm{A}_{in}\ \textbf{\textit{then}}\ \mathrm{y} = \mathbf{B}_i\ \textbf{\textit{else}}\ \dots$$

represents the i-th rule (i=1,2,...,m) in the formal description of a problem.

All attributes can be defined on different universes with different units of measure, which makes *FuzzyExpert* work with cylindrical extensions of the attribute values to the Cartesian product of the universes. Consequently, the system can easily look for an answer in the multidimensional space.

A **query** (hypothesis) can be expressed in a similar way:

$$\mathrm{Q} \equiv \mathrm{x}_1 = \mathrm{A}_1\ \textbf{\textit{and}}\ \mathrm{x}_2 = \mathrm{A}_2\ \textbf{\textit{and}}\ \dots\ \textbf{\textit{and}}\ \mathrm{x}_n = \mathrm{A}_n$$

Here A_j (j=1,2,...,n) stands for a fuzzy set defined on its corresponding universe U_j. Queries are created by the user, who assigns values A_i to the set of attributes. The user can define values or, when convenient, take advantage of the values defined by the knowledge engineer in the KB.

2.2. Inference Process

Generally, attribute values in a query can differ to a greater or lesser degree from their counterparts in rules: $A_j \neq A_{ij}$. The inference engine's targets are to find which rules match a given query and what is the degree of match. For the expert system designer to decide which inference method would provide the best results is not an easy and straightforward task, especially when the system is intended for approximate reasoning with non-crisp values. However, ***generalized modus ponens*** (GMP) seems to be the contemporary paradigm for fuzzy set-based ESs.

The GMP's principle can be briefly explained as follows. Let $\mathcal{A}$ stand for an antecedent, let $\mathcal{B}$ stand for a consequent (i.e. the answer), and let $\mathcal{A} \Rightarrow \mathcal{B}$ denote the implication. Unlike traditional two- or multi-valued logic, GMP makes possible the conclusion $\mathcal{B}'$ when an antecedent $\mathcal{A}' \neq \mathcal{A}$ (provided that $\mathcal{A} \Rightarrow \mathcal{B}$ is valid). The inference engine computes the consequent $\mathcal{B}'$ as the composition of $\mathcal{A}'$ and $\mathcal{R}$:

$$\mathcal{B}' = \mathcal{A}' \circ \mathcal{R} = Q \circ \mathcal{R},$$

where $\mathcal{R}$ is a fuzzy relation made up by an aggregation of rules and $\circ$ means the operator of composition. Rules in a KB are aggregated by way of interpreting the ***else*** operator between each pair of rules with the operator of disjunction (the *disjunctive model*):

$$\mathcal{R} = R_1 \cup R_2 \cup \dots \cup R_m,$$

where $\mathcal{R} \subset U_R = U_1 \times U_2 \times \dots \times U_n \times U_B$ (Cartesian product). A query $Q \equiv \mathcal{A}'$ is a fuzzy relation, too, on the universe $U_A = U_1 \times U_2 \times \dots \times U_n$.

As its output, the inference engine provides values of the membership function of $\mathcal{B}'$. To obtain these values, the system interprets the operators ***and*** and ***then*** as ***min*** (minimum) and the operator $\cup$ as ***max*** (maximum). Then, it computes individual matches between the query Q and each rule R_i. Any match contributes to the result, so the answer $\mathcal{B}'$ consists of superimposed values of all relevant B_i, which are cutoff at the height corresponding to the degree of the match.

The general form of a rule can also be rewritten in this way:

$$R_i \equiv A_{i1} \cap A_{i2} \cap \dots \cap A_{in} \cap B_i,$$

where the operator $\cap$ means ***min***. This form has one interesting implication: because the ***min*** operation is commutative (i.e. $A \cap B = B \cap A$), the order of B_i and any A_{ij} can be changed. Consequently, the user is allowed to look for an attribute value A_{ij} provided he/she knows (or supposes) the value B_i.

If the user requires a single value instead of the resulting fuzzy set, two possible ways have been suggested (Graham and Jones 88):

- *Defuzzyfication* of $\mathcal{B}'$ into a single scalar. *FuzzyExpert* computes the gravity center. (The other possibility would be to compute the point of maximum.);
- *Linguistic approximation* of $\mathcal{B}'$ using a verbal description. Because of its ambiguity, this method is left to the user.

Remark: Generalized modus ponens and the disjunctive model are not, of course, the only candidates for the inference mechanism. It is possible to use other tautologies, such as modus tollens, syllogism, or contraposition; however, GMP is widely preferred. On the other hand, experimenting with the *conjunctive* model (rules are aggregated using the operator $\cap$) seems to be quite meaningful (Kopřiva 88). Unlike its disjunctive counterpart, the conjunctive model provides more determinate answers, which usually do not cover such a wide interval on the output universe. If a knowledge base, interpreted with the conjunctive model, contains at least one rule that disagrees with a query, the inference engine would not provide an answer. This approach can be called "pessimistic" in contrast to the "optimistic" disjunctive model, which gives a positive answer whenever it finds at least one rule matching a query. The structure of *FuzzyExpert*'s inference engine allows an exchange of both models.

3. Implementing *FuzzyExpert* in a PC Environment

The PC environment often seemingly lacks speed, a suitable platform for software development, and sufficient screen size and resolution. Most PCs depend on *Intel 80x86* technology, which restricts operating systems and applications working in the real mode to a 1MB address space (although, in practice, only 640KB are accessible). PC operating systems, such as *MS-DOS*, provide relatively simple capabilities and do not directly support true multitasking or more advanced techniques like virtual memory.

To complete the PC implementation in a short period of time, developers choose Borland's *Turbo Pascal* programming language (version 6.0) for two main reasons: 1) the mainframe prototype was written in Pascal and 2) *Turbo Pascal* is a commonly used programming language, providing a rich set of tools.

3.1. Memory Issues

During the inference process, *FuzzyExpert* looks for a match between a query and a set of rules. Because the search for a match occurs sequentially – the inference mechanism consecutively compares the query against each rule – the system should keep as much data in the computer's main memory as possible. This strategy, however, often meets with serious space problems because a knowledge base can contain hundreds or thousands of rules; each rule, in turn, can hold many values.

The last mainframe prototype version of *FuzzyExpert* ran on the *EC-1045* computer, a Soviet *IBM/370* clone with 4MB of RAM. The *EC-1045*'s operating system allows a program to access up to 16M of virtual memory transparently. For that reason, the prototype previously could handle huge amounts of data without any special programming considerations. Given an *IBM/PC* compatible environment, by contrast, the expert system must process data in the comparatively small heap. Unfortunately, memory restrictions do not stop here. Due to the *Intel 80x86* chip's architecture, an individual data item cannot exceed the address space of one segment (i.e. 64KB). This particular stumbling block arises when a program defines a large array of variables using long data structures such as records.

The simplest solution to the drawback of memory constraints might entail limiting the number of attributes and rules that a user can specify. However, attributes and rules maintain an inversely proportional relationship: the lower the number of attributes, the higher the number of rules and vice versa. The expert system designer cannot easily set the upper bounds of these two parameters because any reasonable combination is allowable; from the perspective of memory utilization, therefore, expert systems demand dynamic control.

A radical technique was employed to solve the problem – a virtual memory mechanism. Specifically, it uses *Object Professional*, TurboPower Software's development tool for object-oriented *Turbo Pascal* programming, which provides a nearly effortless means to circumvent *MS-DOS*'s inherent memory constraints through its virtual large arrays. In fact, this mechanism dynamically uses RAM, expanded memory, extended memory, or disk-based paging, allowing individual data items to exceed 64K bytes in size. The dimensions and data type of a large array may be specified at run time rather than during compilation. Objects for managing a large array are arranged according to the hierarchy shown in Fig. 3.

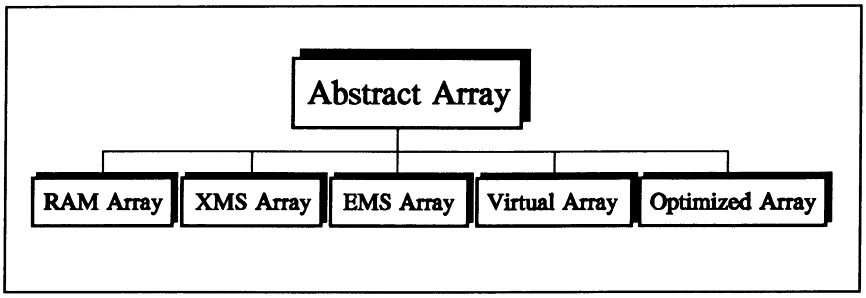


Fig. 3. Hierarchy of large array objects

AbstractArray defines the common methods (e.g. storing or retrieving an array element) used by all of the array types. *RAMArray*, *XMSArray*, *EMSArray*, and *VirtualArray* implement the storage mechanism for heap, *XMS*, *EMS*, and disk-based arrays, respectively. *OpArray* (Optimized Array) can store an array using any of the four types; the choice depends on the computer resources available at run time and on a user-defined priority. Any of the large array types can be stored on disk as a file and later reloaded by any other array type.

This flexibility exacts inevitable costs in overhead, leading to slower access of data types exceeding 64KB. Overhead results mainly from the fact that for each access to a dynamically allocated array element, the routines must calculate a page (segment) and an offset within the page to locate the data.

The *OpArray* method minimizes overhead and slowdown, namely in cases with a low number of attributes and rules, because it automatically uses free space in *RAM* or *XMS/EMS* (if available). This approach brings the mainframe's advantageous virtual memory techniques nearer to the PC world and thus enables downsizing of programs and systems originally developed in a quite different environment.

3.2. Performance Issues

Performance of an expert system (specifically, the inference engine's response time) is a very important criterion. Two fundamental factors affecting system performance are efficient hardware and effective software implementation.

When the system's inference engine was carefully profiled, it revealed disappointing response times in many cases, so the program developers looked for its bottlenecks. Because processing of real numbers engrosses the main CPU load, critical strictures appeared among functions that frequently work with reals. Specifically, the function that compares two real number arrays (often used by *FuzzyExpert*'s inference engine to find a degree of match between a query and a rule) presented the most serious problem. In spite of using various artificial intelligence methods to speed up extensive searches (e.g. alpha-beta pruning), the program developers could not overcome the ultimate flaw: *Turbo Pascal*, like almost all programming languages, surprisingly does not provide any high-level means to compare arrays of the same type directly. The only possibility involves comparing pairs of individual elements in a loop, which is a time-consuming process even with a math coprocessor. Replacing the Pascal code with assembler instructions resulted in a suitable solution for the following reasons: 1) *Turbo Pascal* 6.0 readily supports the use of inline assembler code through its built-in assembler and 2) the inline assembler code can directly refer to the Pascal code (e.g. labels and data items). Of course, such a solution reduces the ability to move the source code to another hardware platform. However, if a sequence of assembler instructions forms a closed unit, such as a function or a procedure, the program developer does not sacrifice too much portability (replacing a unit of code so the program can run on a computer from a different family is always easier than trying to isolate machine-specific code dispersed throughout the program). The result of replacing the *Turbo Pascal* loop with a sequence of assembler instructions was astounding, for the speed of the comparison function improved roughly 10 times. Interestingly, the mainframe predecessor of *FuzzyExpert* suffered from the same problem, and the solution was similar – an assembler routine.

Another substantial speed improvement was achieved through passing large data items (e.g. arrays and records) as variable rather than value parameters. With a variable parameter, the caller passes only a pointer to the parameter without copying the data itself into an auxiliary area in main memory, as value parameters typically do.

Perhaps the most burdensome obstacle to better utilization of a PC concerns *MS-DOS*'s lack of support for true **multitasking**. This deficiency is particularly detrimental in situations when the user views results displayed on the screen while the program idly waits. Instead, the inference engine could process another query in the background, thus reducing the inescapable time interval needed to obtain the next result. As the user examines the screen, the inference process simultaneously runs in the background. Whenever the display process needs the CPU, it preempts the background computation; after finishing its action, the display process returns control to the inference process. This procedure decreases CPU dead time and provides faster total system response. (The display process gives a user supplemental information, such as explanation of the result, the gravity center coordinate, individual components of the result, etc.) Implementation of the interrupt handling was not very difficult, but one serious problem emerged. *Turbo Pascal*'s input/output routines and memory management routines, which invoke *MS-DOS* non-reentrant system calls, cannot be used in an interrupt service routine (ISR). The principal solution to this problem (i.e. essentially rendering input/output operations possible in an ISR) is shown in Fig. 4. The ISR first clears the interrupt, restores the previous interrupt vector, disables subsequent interruptions, and then passes control to the Interrupt server. The server routine, which is a part of *FuzzyExpert's* Interactive User Interface, communicates with the user, displays what is asked for, and then returns control back to the ISR and, in turn, to the Inference process.

4. User Interface

FuzzyExpert's interface gives users the ability to effectively maintain various data files required by its inference engine, a characteristic fostered by the consistency inherent to object-oriented programming (OOP). Unfortunately, problems abounded on the path to this interface, related both to specific requirements of the system and to inherent drawbacks of OOP.

Almost every aspect of *FuzzyExpert*'s front end depends on OOP. Specifically, each user interface-related object (e.g. pick list, entry screen, or dialog box) originates with a window object, which consists of data and methods to handle features common to all objects, at the root of the object hierarchy. For example, a window can include a title header. It can support scroll bars for vertical or horizontal adjustments. If the user wants a hot spot that, when clicked, closes a window, the object complies. A window moves, too. The most powerful method of a window, however, processes keystrokes entered by the user when the window is active. Essentially a large **CASE** statement in a control loop, the Process method fields a key press that it anticipates, then returns to the top of the loop to await the next

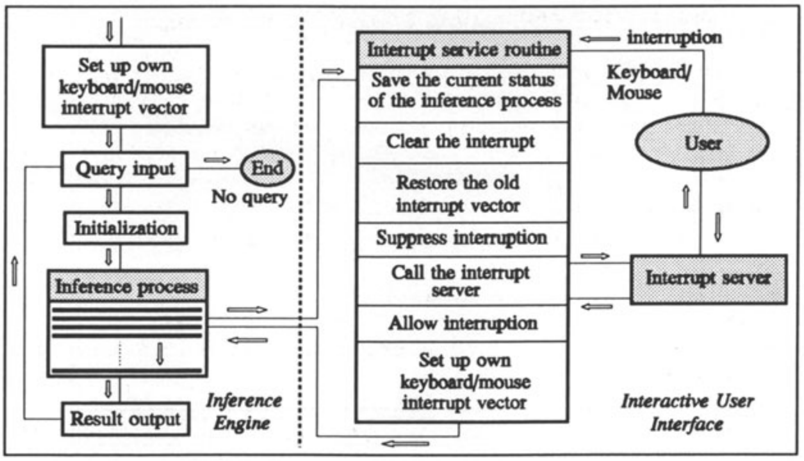

Set up own keyboard/mouse interrupt vector
Query input
End
No query
Initialization
Inference process
Result output
Inference Engine
Interrupt service routine
Save the current status of the inference process
Clear the interrupt
Restore the old interrupt vector
Suppress interruption
Call the interrupt server
Allow interruption
Set up own keyboard/mouse interrupt vector
interruption
Keyboard/ Mouse
User
Interrupt server
Interactive User Interface

Fig. 4. Principle mechanism for preemption of the background computation

keystroke. Some key presses cause the Process method to exit, allowing the user to provide unique handling. Obviously, any user interface object might have a need for these capabilities. Through object-oriented programming, a descendant object can very easily utilize a window feature simply by calling the appropriate method.

Development of *FuzzyExpert*'s user interface encountered one significant problem. Although the system runs in text mode, which entails less arduous programming than graphics mode, the full complement of 256 ASCII characters does not contain some odd symbols required to paint a fuzzy set shape on the screen. Consequently, *FuzzyExpert* programs the computer's EGA or VGA video display card to create 11 of these unusual characters.

4.1. *FuzzyExpert*'s Knowledge Base/Fact Base Manager

FuzzyExpert's user interface provides a highly structured means to create the input data files necessary to run the inference engine. Through the **Variables** option on the main menu, the user can build a linguistic attributes file. *FuzzyExpert* displays a dialog box containing an entry screen for 14 linguistic attribute records (actually, a 14-record view of a whole file). By clicking on a pushbutton, the user can define the values associated with the currently highlighted linguistic attribute. In response, *FuzzyExpert* draws a dialog box containing an entry screen for 14 linguistic attribute value records (again, a 14-record view of a whole file). The user can only edit a value's name in this dialog box (its value type and breakpoint coordinate fields are read-only), but by clicking on a pushbutton, the user can determine the fuzzy set shape (i.e. type) and breakpoint coordinates of the currently highlighted linguistic attribute value. *FuzzyExpert*, in turn, presents another dialog box that includes a pick list of predefined fuzzy set shapes and the appropriate number of entry fields for the selected shape's breakpoint coordinates. In Fig. 5, for example, a triangular-shaped fuzzy set is currently chosen in the pick list, and as a result, only three entry fields appear. Thus, with a few keystrokes, the user can construct the linguistic attributes and values files essential for the inference engine.

To create a file of rules or a file of queries for an inference engine run, *FuzzyExpert* offers two separate options on its main menu: **Rules** and **Queries**. However, with some minor exceptions, identical processing occurs for these input data file types. *FuzzyExpert* displays a read-only window that lists existing rules or queries. By clicking on a pushbutton, the user

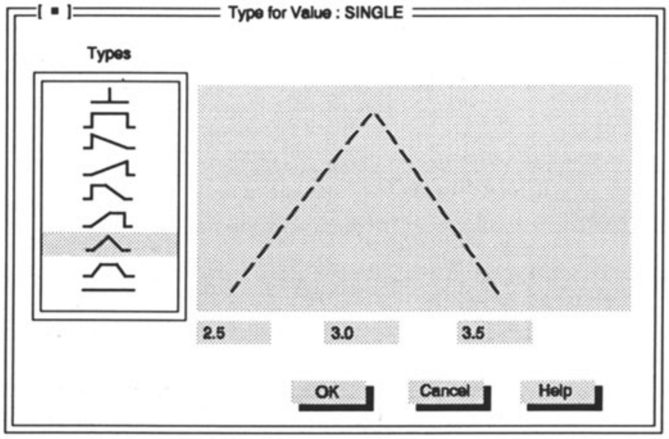


Fig. 5. Attribute value type dialog box with predefined fuzzy set shapes

can update the currently highlighted rule or query. Because a rules or a queries file must relate to a specific linguistic attributes file, *FuzzyExpert* draws a dialog box containing a scrolling entry screen with exactly one field for each linguistic attribute; the user must enter a previously defined value in every field. Thus, the user can generate two more elements – the rules file and the queries file – necessary for a run of *FuzzyExpert*'s inference engine.

Another important role of the Knowledge/Fact Base Manager is preventing the inference engine from crashing due to problems concerning data integrity. Rules and queries files derive from a linguistic attributes file; each rule or query must include one value for every defined attribute. Assuming the user creates a rules file then deletes a record from the linguistic attributes file, an incongruity exists that would force the inference engine to abort. To remedy this problem, *FuzzyExpert* tracks all modifications to a linguistic attributes file and its associated values file. When the user finishes editing these files, *FuzzyExpert* automatically reflects changes in the related rules and queries files.

4.2. *FuzzyExpert*'s Run Definition Facility

Integrating these distinct files and delineating parameters for execution of the inference engine, *FuzzyExpert* offers the simple mechanism of a run definition file. Through the main menu's **Inference** option, the user can design up to 100 different run definition files for a single linguistic attributes file. *FuzzyExpert* presents a rich dialog box containing, among other items, entry fields for a description of this particular run definition file, the specific rules file and queries file that the inference engine should read, as well as many parameters. Parameters in the file enable a number of special functions, for example:

- omitting some attributes from the run (so called non-live variables);
- selecting a constant or a variable scale of the output attribute universe;
- restricting the percentage of activated rules (when a query does not result in an answer after processing the requested percentage of rules, the inference engine ignores the rest of the rules to prevent long, unnecessary computations for ill-formulated queries);
- displaying only those components of an answer that have cutoff values greater than a user-demanded threshold;

- computing primary consistencies of KBs (the left side of each rule is treated consecutively as a query, and the inference engine looks for a match between the rule and the rest of the KB; this function assists users and knowledge engineers in searching for knowledge gaps in KBs);
- stretching original attribute values in a query when the inference engine cannot obtain an answer (fuzzy set breakpoints on the universe axis are stretched to the left and to the right so each value becomes "wider", which increases the possibility of getting a conclusion). This feature can help users to find out what additional knowledge is necessary to improve the system's inference results. Fig. 6 illustrates the effect of stretching.

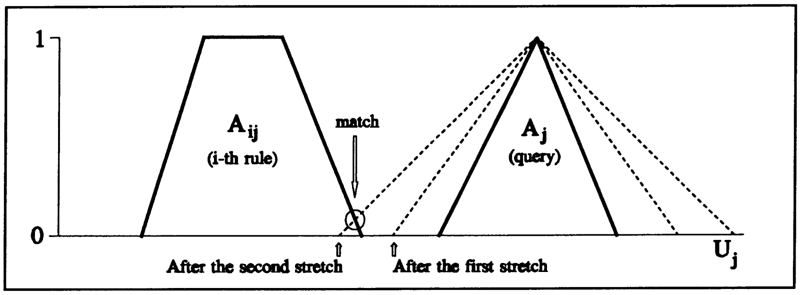


Fig. 6. The value A_j matches the value A_{ij} after the second stretch

After the user establishes a run definition file, he/she can start the inference process merely by selecting the run definition file.

4.3. Output Interface

FuzzyExpert displays its output in conformity with the input data: answers to the user's queries appear on the screen in graphics mode as compound fuzzy set shapes. The user can ask for supplemental information, including the gravity center, outlines of individual fuzzy set components, and an explanation window. *FuzzyExpert* saves each answer on disk so it can be easily redrawn later. Moreover, a detailed description of the inference process's conclusions is stored in a text file. *FuzzyExpert*'s supporting utilities enable printing of these graphic and text files as well as exporting of graphic screens to several common file formats (e.g. PCX, TIFF). Fig. 7 illustrates a graphic output screen of the inference engine.

5. Conclusion and Recommendations

This paper presents problem areas that developers of PC-based expert systems can encounter, stemming from hardware specificities and from software complexities. Effective memory management remains mere wishful thinking: a user's process must control utilization by itself. In spite of these imperfections, PCs now dominate the computing world, so expert systems must shift to this platform. On the software side, *FuzzyExpert* was implemented in Borland's *Turbo Pascal*, although profiling revealed several serious bottlenecks that only inline assembler instructions could bypass. This solution, while improving *FuzzyExpert*'s performance, decreased its portability. *Turbo Pascal*'s object-oriented extension supports such important and, at the same time, difficult tasks as developing *FuzzyExpert*'s user-friendly interface. To briefly summarize *FuzzyExpert*'s implementation experience, today's hardware and software provide a powerful base for complex software system development; however,

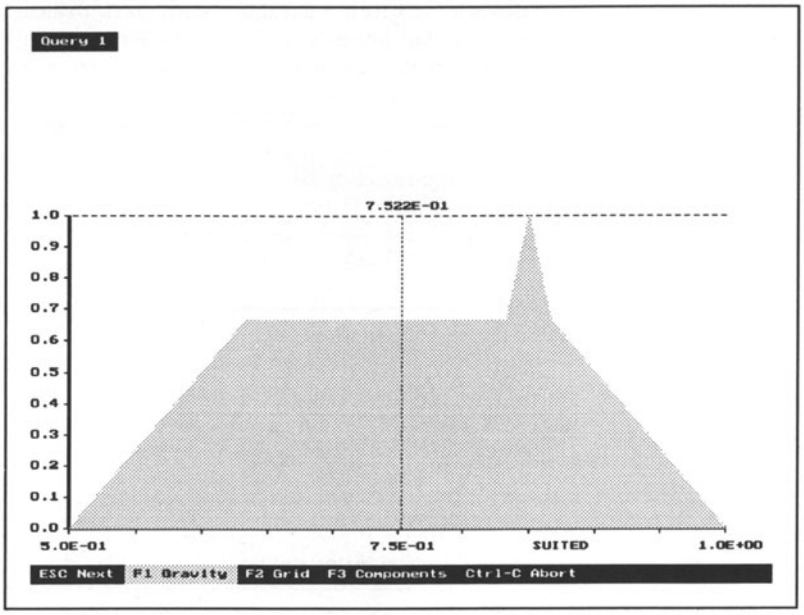


Fig. 7. An example of the inference engine graphic output

many issues still await their perfect solutions. A brief study showed that *UNIX*-based systems would provide a far more convenient environment for developing and running expert systems. The *UNIX* operating system naturally includes a virtual memory mechanism as well as multitasking. Despite the implementation difficulties, the result – *FuzzyExpert* for PCs – is a useful tool with many possible applications.

6. References

1. Giarratano, J. and Riley, G. *Expert Systems: Principles and Programming*. PWS-KENT Publishing Company, U.S.A., 1989.

2. Graham, I. and Jones, P.L. *Expert Systems: Knowledge, Uncertainty and Decision*. Chapman and Hall, London, 1988.

3. Kopřiva, J. *Fuzzy Deductive System, Its Implementation and Application*. Proceedings of "Modern programming 1988", pp. 119-130, Červený Kláštor, Czechoslovakia, May 29 – July 3, 1988. In Czech.

4. Payne, E.C. and McArthur, R.C. *Developing Expert Systems*. John Wiley & Sons, Inc., U.S.A., 1990.

5. Zimmerman, H.-J. *Fuzzy Set Theory and Its Applications*. Kluwer-Nijhoff Publishing, U.S.A., 1985.

A CLAUSE INDEXING METHOD

Kálmán Balogh

"IQSOFT" SzKI
INTELLIGENT SOFTWARE CO. LTD.
H-1251 Budapest,P.O.Box 73.Hungary
(H-1011 Budapest, Iskola u. 10.)
E-mail: BALOGH@IQSOFT.HU
H1395bal@ella.UUCP
Phone: (36-1)201 67 64
Telefax: (36-1) 201 71 25

Abstract

An indexing method for clauses of predicate logic is discussed. The method is based on the decision tree corresponding to the argument expressions of procedure heads. The method is efficiently applicable for procedures containing a lot of clauses, to direct both or- and and-parallelism. It is indicated, how to apply the method to a knowledge base of frames or objects. This indexing method suits well to common inheritance operations, and increases their efficiency.

Keywords: predicate logic, Prolog, knowledge base, data base, indexing, decision tree, frame, object, inheritance.

1 Introduction

Traditional execution method of Prolog evaluates a call of a predicate through linear search for alternative clauses in the corresponding procedure when backtracking. This search is inefficient, if the procedure contains a lot of clauses. Introduction of any indexing methods would increase efficiency. In fact, DEC-10 Prolog [1] and some other implementations of Prolog apply indexing of clause heads according to the main functors of their first arguments for a long time.

Indexing can be explored not only when searching for matching alternatives, that is evaluating "or" branches of the execution tree. Evaluation of conjunctive subgoals in a parallel, mutually dependent way can be supported by indexing their definitions (the corresponding procedures) in accordance with each other.

Indexing of procedures should not be treated independently of the evaluation mechanism. When determining the indexing method described here my aim was

- to prepare evaluation of procedures statically as far as possible;

- to store procedures in an irredundant way, where it does not conflict with functional requirements (e.g. order prescription for alternatives, generality of argument expressions) or with the former aim.

According to the different kinds of functional and evaluation requirements, a variety of index structures and searching algorithms is determined.

The paper is divided into two main parts. In section 2 the indexing method based on decision trees is introduced, while in section 3 it is shown, how the method can be explored when structuring a knowledge base.

2 The indexing method

This indexing indexing method is generalized from that of MProlog [2] for static clauses.

The method is introduced in three steps. In the first step a variable free world is supposed, which is extended in the further steps with handling variables in calls, then in heads of procedures, too. These steps result in three main variants of the indexing method, as described in section 2.1, 2.2 and 2.3.

Refined versions of the above variants can be derived, if other requirements, being orthogonal to the former one (dealing with indexing from the point of view of variables) are considered, too. Two further dimensions of such requirements are investigated here. The first one is, at what extent the original order of clauses constituting a procedure should be preserved. The other dimension is, whether during parameter passing the input/output role of some arguments of the defined predicate is restricted or not. Treatment of these additional requirements is discussed within the three-step description of the main variant.

Further refinement possibilities, being common for the previous variants are mentioned in section 2.4 .

2.1 The variable free case

The main purpose is to prepare matching of a given procedure to its possible calls as far as possible. This aim is reached, if clauses of the procedure are stored in a decision tree corresponding to the expressions occuring in the heads of clauses of that procedure.

First a special case of simple constant arguments is described, then handling of compound arguments is discussed.

2.1.1 Treatment of heads containing simple arguments

Let us see an example of a unary procedure:

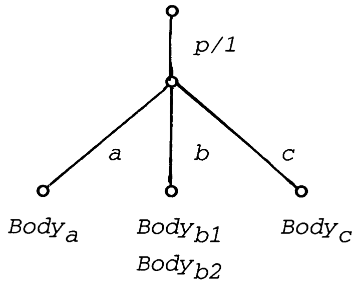
p(a):- Body$_a$.

p(b):- Body$_{b1}$.

p(c):- Body$_c$.

p(b):- Body$_{b2}$.

The decision tree corresponding to this procedure is



Let us call the list of clauses referred by the same leaf of the tree to be a *partition*.

The example shows, that the second aim of the introduction of indexing - irredundancy of both data structures and processing - is also reached, as common components of different clause heads are extracted. In this example it is trivial, but this property will hold for the more complicated versions of the method, too.

Another important property of the decision tree (which also will hold for further variants) is, that branches of the decision tree originated at the same node are exclusive alternatives. Original order of the clauses of the procedure is preserved (if it is preserved within partitions), when searching for matching alternatives to a call.

As the order preserving property of the method is independent of the order of the branches of a node, it is possible to use any kind of indexing methods (e.g. logarithmic search within lexicographically ordered symbols, B-tree handling,hashing) to improve efficiency of search within a great number of alternative branches.

If the defined predicate has arity more then one, then (in case of simple arguments) one can attach a layer of the tree to each argument position. Order of the layers shows the order of the decisions according to the arguments. This order is arbitrary; either it may be the original order of arguments, or it can be prescribed by the user through a so-called match-order declaration [2].

2.1.2 Treatment of heads containing compound arguments

If the arguments of the predicate definition may contain variable free expressions, then indexing the definition according to the designator of the head argument expressions can be made in a way analogous trivially to the former case. The *designator* of an expression is the term *Name/Arity*, where *Name* is the name and *Arity* is the arity of the main functor of the expression. The designator of a simple constant *C* can be regarded to be *C/0* .

Indexing according to the designators can be extended to the deeper level of argument expressions. The decision tree can be built according to the processing of the argument expressions, e.g. in a depth first left to right order.

The branches introduced so far are said to be of type *des-branch* (branches for given distinct designators).

2.2 Handling variables in the call

When calls with (unbound) variables are allowed, a further type of branch should be introduced. The new branch is called *var-branch*, which lists all clauses attached to the node.

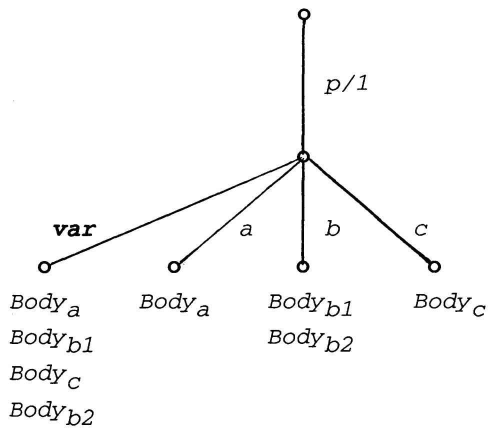
In case of the example of 2.1.1, the tree corresponding to the procedure for *p/1*, when calls with variable arguments are allowed is

$p(a) :- Body_a.$

$p(b) :- Body_{b1}.$

$p(c) :- Body_c.$

$p(b) :- Body_{b2}.$



Explicit representation of the var-branch in the tree

- causes redundancy in storage (multiple references to the same clauses)
- preserves original order of matching clauses.

Indexing is equally efficient in each variation of i/o role (during parameter passing) of the arguments. This property will hold also in further versions of the method, but speed of execution of the indexing is slower, where we choose an irredundant representation.

In the next refinement of the method var-branches are eliminated from the representation, so each clause is referred to only once from the tree. The price of it is, that the algorithm should enumerate all the branches corresponding to a call with a variable, and the method preserves original order of the clauses only within partitions.

A var-branch is not needed for an argument position, if the user states in a so-called mode declaration, that the arguments should be concrete in the procedure calls for that argument position. This possibility is given e.g. in MProlog [2] for main designators of arguments.

2.3 Handling variables in the heads of the procedure

Clauses of a procedure, heads of which contain variables in the position in question, can be successfully matched with an arbitrary call of the predicate.

The method does not take account of multiple occurences of the same variable in a head; the tree contains only context free information.

In order to handle the above mentioned clauses, a new type of branches,called *else-branch* is introduced. Such a clause should be inserted into all alternative branches of the else-branch, too. For this price original order of matching clauses is preserved.

Let us extend the example of 2.1.1, to see the effect on the representation. The new procedure for *p/1* is

p(a):- Body$_a$.

p(b):- Body$_{b1}$.

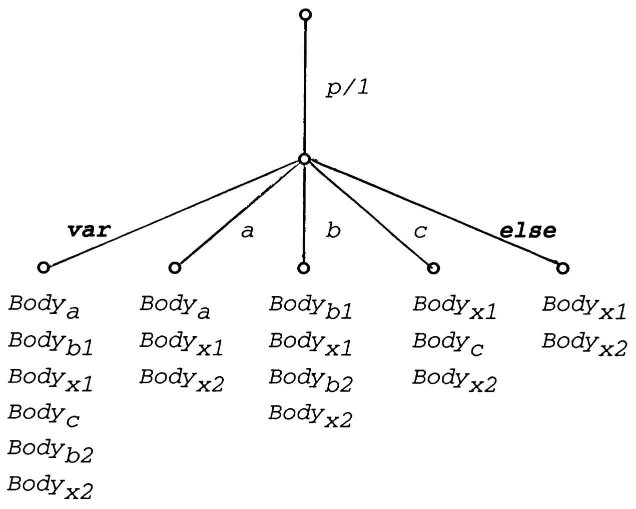
p(X):- Body$_{x1}$.

p(c):- Body$_c$.

p(b):- Body$_{b2}$.

p(X):- Body$_{x2}$.

The tree corresponding to the above procedure in the general case is


p/1
var
a
b
c
else
Body$_a$ Body$_{b1}$ Body$_{x1}$ Body$_c$ Body$_{b2}$ Body$_{x2}$
Body$_a$ Body$_{x1}$ Body$_{x2}$
Body$_{b1}$ Body$_{x1}$ Body$_{b2}$ Body$_{x2}$
Body$_{x1}$ Body$_c$ Body$_{x2}$
Body$_{x1}$ Body$_{x2}$

If the clauses, heads of which contain variables in the position in question are deleted from the var- and des-branches of the representation, then each new clause is referred once by the tree. In this case the algorithm should enumerate the clauses on the else-branch, having enumerated those referred by the matching des-branch(es). Original order of the matching concrete and general clauses is preserved only separately, within the two groups, but each of the concrete alternatives will precede any of the general ones.

Else-branch is needless, if the user states in a declaration, or it is verified by preprocessing the definition, that the corresponding arguments should be concrete in the heads of the procedure.

2.4 Common refinement possibilities

Let us suppose, that execution of the procedures is preceded by preprocessing of the bodies of the clauses in addition to that of their heads. This allows us to bind each call to those subtrees within their definition, which can be determined by the statically existing argument expressions. The binding can be refined during the execution, thus focussing on the smallest possible subtree.

Binding information is valuable for dynamic memory management. On one hand it helps increasing the effect of garbage collection. On the other hand it makes possible increasing efficiency of secondary storage management through a dedicated paging system built on a hardware virtual memory.

According to our purpose whether to apply the method for static or dynamic procedures, different refinements of the method are suitable to implement.

On the other hand, specification of the built-in dynamic procedure handling predicates should be synchronized with the method. Refering to clauses via an external (source level) sequence number allows only a low level, algorithmic interface for the user. This facility can be overridden by giving (also) more Prolog-like nondeterministic and backtrackable procedures, based on general searching possibilities, which can be efficiently implemented by the indexing method.

3 Application of the indexing method for frames

Each way of indexing helps in efficient implementation of structured knowledge bases, e.g. that of frames or objects. In the following it is shown in a frame terminology, that the above method of indexing helps both structuring of knowledge and inheritance in an extremely efficient way, compared to other indexing methods.

3.1 Representation of frames

Let us assume for simplicity, that frames have the following form

frame Frame_id with parameter P.

  slot S_name_1: S_value_1.

  . . . .

  slot S_name_m: S_value_m.

endframe.

Here *Frame_id* and *P* are arbitrary terms [3].

This form will be sufficient to show, how the previous indexing method can be applied.

A frame of the above form can be represented in Prolog as a set of clauses corresponding to the predicates frame/2 and slot/4

frame(Frame_id,P).

slot $(Frame\_id,P,S\_name_1,S\_value_1)$.

. . . .

slot $(Frame\_id,P,S\_name_m,S\_value_m)$.

In general, if frames have further kinds of components, a frame system can be represented in Prolog through partitioning the procedures according to the frame identifiers

predicate designator / partition corresponding to a frame	*frame/2*	*slot/4*	. . .
Frame_id$_1$			
Frame_id$_2$			
. . .			
Frame_id$_n$			

3.2 Description of frame structure and inheritance through inheritance rules

Inheritance can be described directly by clauses [4] of form

Frame_id$_1$ *inherits Component from Frame_id*$_2$
if Body.

Indirect description of inheritance by using binary relations between frame identifiers has the following form

Frame_id$_1$ *is related to Frame_id*$_2$ *by Rel.*

Frame_id inherits Component through Rel
if Body.

3.3 Reflecting frame structure through the construction of the frame identifiers

An example is shown, how to avoid redundant storage of overlapping frames by properly constructed frame identifiers.

If we have two overlapping frames, named f_1 and f_2, we do not want to store their common components twice. This is the case, if, for instance, one wants to store production rules of form

if f_1 *is_in_working_memory and*

f_2 *is_in_working_memory*

then Conclusion

knowing, that role of the two components of the condition part of the rules is symmetric.

In order to access these rules both from f_1 and from f_2, but store them irredundantly, an auxiliary frame named $f_1 \cap f_2$ is introduced, and the rules are placed into this frame. More generally, we make the representation disjoint by introducing the auxiliary frames named $f_1 \setminus f_2$, $f_1 \cap f_2$ and $f_2 \setminus f_1$

$$\begin{tikzcd}
f_1\setminus f_2 \arrow[dr, dash] & & f_1\cap f_2 \arrow[dl, dash] \arrow[dr, dash] & & f_2\setminus f_1 \arrow[dl, dash] \\
& f_1 & & f_2 .
\end{tikzcd}$$

Inheritance can be expressed stating the following rules

F inherits All from (F_).

F_1 *inherits All from* F_2

if F_1 *is_conjunctive_component_of* F_2.

(The last predicate can be defined in Prolog easily.)

3.4 Reducing inheritance back to subsumption

Special frame structures can be described using properly constructed frame identifiers through the subsumability of these identifiers. We say, that *term T1 subsumes term T2*, if *T1* and *T2* are unifiable without binding any variables in *T1*. E. g. the term *parallelogram(square(X))* subsumes the term *parallelogram(X)* .

By the subsumability of frame identifiers the most frequent common situations can be expressed, among others hierarchic structures.

Execution of inheritance operations in general needs inference, that is evaluating inheritance rules. Deriving inheritance back to subsumption checking puts execution to unification level, so it allows a far more efficient execution. Implementation of subsumption checking is even more efficient, if it is specialized according to the indexing method described in section 2 .

3.5 Inheritance strategies

Tools are needed for the user for conflict resolution among multiple sources of inheritance. Possible reasons of nondeterminism of inheritance are enumerated first, then ways for solutions are sketched.

The pure method of inheritance through subsumption has a serious drawback: the user cannot control, which component of the frame is to be inherited from where (if the frame has more than one parent according to the frame identifiers, or there are exceptional connections among frames, which are given by inheritance rules). This problem arises also, when the inheritance rules are conflicting.

It is also worthy to give possibilities to indicate types of inheritance (whether the inheritance should be e.g. deterministic, classic or default).

Means are needed to describe the strategy, which determines, whether to search for the source of inheritance within the ancestors of a parent (to search first in depth), or within the brothers of the parent (to search first in breadth), and in the latter case determines the source of inheritance within the parents.

The above problems can be solved efficiently by assigning suitable built_in predicates to specific strategies, argument of which is the reference to the frame component. A more general solution would be to give for the user a binary predicate, to allow encapsulating also the description of strategy beside the frame reference into an argument of this predicate.

It is worth assigning strategies not only to references (dynamically), but rather to slots of a frame definition by declaration. It is nice and clearly arranged, if this declaration is part of the creation of the frame.

4 Conclusion

Ways of indexing of procedures and kinds of evaluation mechanisms should be related suitably, when building and processing knowledge bases. Consequences of this observation corresponding to a rather abstract level of notions are described above. If one concretises stepwise the notions in question, further fruits of the method can gather.

Some of the benefits of the decision tree based indexing method are indicated here. Further investigation should be taken to elaborate and implement these possibilities and find other correlations between indexing and evaluation mechanisms, e.g. depending on whether

- the evaluation mechanism is based on structure sharing or copying
- the programming language is Prolog or LDL [5], [6]
- the method is applied to a static or a dynamic definition (supplied with a variety of basic operations).

The indexing method based on decision trees can be generalized to be based on decision graphs [7].

References

[1] *Warren, D., Pereira, F. and Pereira, L. M.: User's Guide to DECsystem-10 Prolog*, Occassional Paper 15, Dept. of Artificial Intelligence, University of Edinburgh, 1979.

[2] *MProlog Language Reference Manual. MProlog* is a registered trademark of IQSOFT SzKI, Intelligent Software Co. Ltd., H-1251 Budapest, P.O.Box 73., Hungary

[3] *C. Zaniolo: Object-oriented programming in Prolog*, IEEE, Proc. Symposium on Logic Programming, 1984, Atlantic City, pp. 265-270.

[4] *A. Domán: Object-Prolog:Dynamic object oriented representation of knowledge*, Proc. SCS Multi Conf. on AI & Simulation, 1988., San Diego.

[5] *S. Naqvi - S. Tsur: A Logical Language for Data and Knowledge Bases*, Computer Science Press, New York, 1989.

[6] *G. Gardarin - P. Valduriez: ESQL: An Extended SQL with object and deductive capabilities*, Rapports de Recherche No 1185, INRIA, Le Chesnay, France, 1990.

[7] *S. Kliger - E. Shapiro: From Decision Trees to Decision Graphs*, Prceedings of ICLP, 1990.

KEYNOTE LECTURE

Chair: G. Pomberger

Software Engineering for Real-Time Systems

H.Kopetz

Technical University of Vienna
Austria

Abstract. A hard real-time system has to produce the correct results at the intended points in time. In such a system a failure in the time domain can be as critical as a failure in the value domain. In this paper it is claimed that an engineering approach to the design of the application software for a hard real-time system is only possible if the run-time architecture is based on the time triggered paradigm.

1. Introduction

At present, real-time system development resembles sometimes a "black art". Modules of conventionally designed software are integrated by "real-time specialists" who tune the system parameters (e.g., task priorities, buffer sizes, etc.,) during an extensive trial and error testing period, consuming more than 50% of a projects resources. Why the system performs its functions at the end is sometimes a miracle, even to the "real-time specialists".

Temporal properties are system properties. They depend on the behavior of all levels of an architecture, e.g., the hardware, the operating system, and the application software. A systematic design of real-time software is only possible if the underlying hardware and operating system guarantee a predictable temporal behavior. In this paper we examine the architectural prerequisites for an engineering approach to the development of real-time systems, as proposed in [1].

This paper is organized as follows. After a classification of real-time systems we present a set of key design problems that have to be solved in any rational real-time software development process. We then examine proposed solutions and conclude that only time-triggered architectures support an engineering approach to hard real-time system design.

2. What is a real-time system ?

In many models of natural phenomena (e.g., Newtonian mechanics), time is considered as an independent variable which determines the sequence of states of the considered system. The basic constants of physics are defined in relation to a standard of time, the physical second. If we intend to control the behavior of a natural system, we have to act on the system at precise moments in time.

We define a *real-time system* as a system that changes its state as a function of (real) time. Our interest focuses on real-time systems that contain embedded computer systems. It is sensible to decompose such a real-time system into a set of *clusters*, e.g., the *controlled object*, the *computer system* and a *human operator* (Fig.1). We call the controlled object and the operator the *environment* of the computer system. The computer system must react to stimuli from the controlled object (or the operator) within time intervals dictated by its environment. Such a computer system is called a *real-time computer system.*

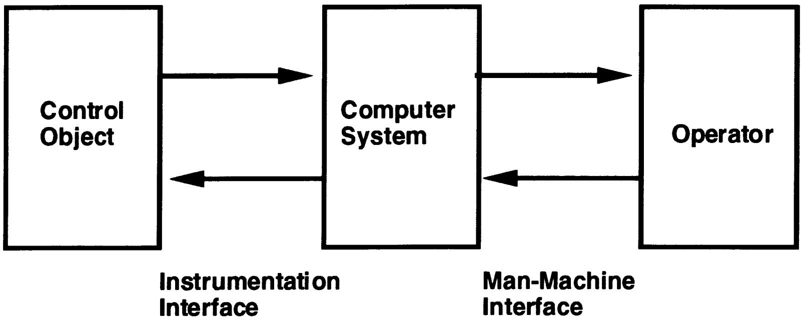


Fig.1: A real-time computer system

Since the real-time computer system is only a part of the total real time system, there must be interfaces between the real-time computer system and its environment. We call the interface between the real-time computer and the controlled object the *instrumentation* interface, consisting of sensors and actuators, and the interface between the real-time computer system and the operator the *man-machine* or operator interface.

Nowadays, most real-time computer systems are distributed. They consist of a set of nodes interconnected by a real-time communication system. Access to this real-time communication system must be controlled by a *real-time protocol,* i.e. a protocol that has a known small maximum execution time.

Based on the above definition of a real-time computer system it follows that the duration between a stimulus from the environment and the response to the environment must be time constrained. We call the sequence of all communication and processing steps between such a stimulus and response a *real-time (RT) transaction.* A RT-transaction must deliver the correct result at the intended point in time. Otherwise, the real-time computer system has failed.

Any real-time computer system has a finite processing capacity. If we intend to guarantee by design that the given temporal requirements of all critical real-time transactions can be satisfied then we have to postulate a set of assumptions about the behavior of the environment. The *load hypothesis* and the *fault hypothesis* are two of these important assumptions.

Load Hypothesis. The load hypothesis defines the *peak load* that is assumed to be generated by the environment. It can be expressed by specifying the minimum time interval between--or the maximum rate of--each real-time transaction. Peak load implies that all specified transactions will occur with their maximum specified rate. In many applications the utility of the real-time system is highest in a *rare event situation* that leads to a peak load scenario. Consider the case of a nuclear power station monitoring and shutdown system. It is probable that in case of the rare event of an reactor incident--e.g., the rupture of a pipe--many alarms will be activated simultaneously and will thus generate a correlated load. Statistical arguments about the low probability for the occurrence of peak load, based on the argument that the tail of a load distribution of independent events is very small are not valid in such a situation. If a real-time system is not designed to handle the peak load it can happen that the system will fail when it is needed most urgently.

Fault Hypothesis. The fault-hypothesis defines the types and frequency of faults that a fault-tolerant system must be capable of handling. If the identified fault scenario develops, the system must still provide the specified level of service. If the environment generates more faults than specified in the fault-hypothesis, then even a fault tolerant system may fail. The worst scenario that a fault-tolerant real-time system must be capable of handling exists if the peak-load and the maximum number of faults occur at the same time.

Even a perfect fault-tolerant real-time system will fail if the load-hypothesis or the fault hypothesis are unrealistic, i.e., they do not properly capture the behavior of the environment. The concept of *assumption coverage* defines the probability that the fault and load hypothesis--and all other assumptions made about the behavior of the environment--are in agreement with reality.

3 Classification of RT-Systems

We call a real-time system as *soft*, if the consequences of a timing failure are in the same order of magnitude as the utility of the operational system. Consider, e.g., a letter sorting machine. If a letter is placed in the wrong bin because of a timing failure, the consequences are not very serious--the letter will have to be sorted again.

If the consequences of a timing or value failure can be catastrophic, i.e., the cost of such a failure can be orders of magnitude higher that the normal utility of the system, then we call the system a *hard real-time system.* A railway signalling system is a good example of a hard real time system.

For some hard real-time systems one or more safe states can be identified that can be accessed in case of a system failure. Consider the example of the railway signalling system. In case a failure is detected it is possible to stop all trains and set all signals to red to avoid a catastrophe. If such a safe state can be identified, than we call the system a *fail-safe system.* Note, that fail-safeness is a characteristic of the control object, not the computer system. In fail safe applications the computer system must have a high error detection coverage, i.e., the probability that an error is detected, provided it has occurred, must be close to one.

There are, however, applications where such a safe state cannot be identified, e.g., a flight control system aboard an airplane. In such an application the computer system must provide a minimal level of service even in the case of failure in order to avoid a catastrophe. This is reason why these applications are called *fail operational.*

In the rest of this paper we will focus on hard real-time systems.

4. Key Design Problems

In this section we discuss some of the key problems in the design of fault-tolerant distributed hard real-time computer systems.

4.1 Flow Control

Flow control is concerned with the synchronization of the speed of the sender of information with the speed of the receiver, such that the receiver can follow the sender.

Since the controlled object in many real-time systems is not in the *sphere of control* of the computer system, there is no possibility to limit the occurrence of events in the controlled object in case the computer system cannot follow. Therefore provisions must be made that correlated event showers can be buffered at the interface between the controlled object and the computer system. Several engineering solutions are applied to restrict the flow of events at this interface. These include hardware implemented low pass filters, intermediate buffering of events in hardware and/or software, etc.. However it is still one of the difficult design problems to devise a flow control schema for a real time system that

* protects the computer system from overload situations caused by a faulty sensor or a correlated event showers and at the same time
* makes sure that no important events are suppressed by the flow control mechanism.

4.2 Scheduling

In general, the problem of deciding whether a set of real-time tasks whose execution is constrained by some dependency relation (e.g., mutual exclusion), is schedulable, belongs to the class of NP-complete problems[4]. Finding a *feasible schedule*, provided it exists, is another difficult problem. The known analytical solutions to the dynamic scheduling problem [7] assume stringent constraints on the interaction properties of task sets that are difficult to meet in distributed real-time systems. In practice most dynamic real-time systems resort to *static priority scheduling*. During the commissioning of the system the static priorities are tuned to handle the observed load patterns. No analytical guarantees about the peak load performance can be given.

4.3 Testing for Timeliness

In many real-time system project more than 50% of the resources are spent on testing. It is very difficult to design a constructive test suite to systematically test the temporal behavior of a complex real-time system if no temporal encapsulation is enforced by the system architecture.

4.4 Error Detection

In a real-time computer system we have to detect value errors and timing errors before an erroneous output is delivered to the control object. Error detection has to be performed at the receiver and at the sender of information. The provision of an error detection schema that will detect all errors specified in the fault hypothesis with a small latency is another difficult design problem.

4.5 Replica Determinisms

In many real-time applications the time needed to perform checkpointing and backward recovery after a fault has occurred is not available. Therefore fault-tolerance in distributed real-time systems has to be based on active redundancy. Active redundancy requires *replica determinism*, i.e., the active replicas must take the same decisions at about the same time in order to maintain *state synchronism.* If replica determinism is maintained, fault-tolerance can be implemented by duplex fail-silent selfchecking nodes (or by Triple Modular Redundancy with voting if the fail-silent assumption is not supported).

5. The solution space

Depending on the triggering mechanisms for the start of the communication and processing activities in each node of a computer system, two distinctly different approaches to the design of real-time computer applications can be distinguished. In the *event triggered* (ET) approach all communication and processing activities are initiated whenever a significant change of state, i.e., an event, is recognized. In the *time triggered (TT)* approach all communication and processing activities are initiated periodically at predetermined points in time. In the following sections we will analyze the problem solving potential these two competing design philosophies.

5.1 Event triggered systems

In a purely event triggered (ET) system all system activities are initiated by the occurrence of significant events in the control object or the computer system. In many implementations of ET-systems the signalling of significant events is realized by the well known interrupt mechanism, which brings the occurrence of a significant event to the attention of the CPU.

Flow Control. Within an ET-system, *explicit flow control* mechanisms with buffering have to be implemented between a sending and a receiving entity. The time span which an event message has to wait in a buffer before it can be processed reduces the temporal accuracy of the observation and must thus be limited. The provision of the proper buffer size is a delicate problem in the design of ET-systems.

Scheduling. Operating systems for ET-systems are demand driven and require a dynamic scheduling strategy. Since it is difficult to systematically tackle the complex scheduling problem in the available restricted time span, in practice most ET-systems resort to a simple static priority scheduling. During the commissioning of the system the static priorities are tuned to handle the observed load patterns. No analytical guarantees about the peak load performance can be given.

Testing for Timeliness. The confidence in the timeliness of an ET-system can only be established by extensive system tests on simulated loads. Testing on real loads is not sufficient, because the *rare events*, which the system has to handle (e.g., the occurrence of a serious fault in the controlled object), will not occur frequently enough in an operational environment to gain confidence in the peak load performance of the system. The predictable behavior of the system in rare-event situations is of paramount utility in many real-time applications

Since no detailed plans for the intended temporal behavior of the tasks of an ET-system exist, it is not possible to perform "constructive" performance testing at the task level. In a system where all scheduling decisions concerning the task execution and the access to the communication system are dynamic, no *temporal encapsulation of the tasks* exists, i.e., a variation in the timing of any task can have consequences on the timing of many other tasks in different nodes. The critical issue during the evaluation of an ET-system is thus reduced to the question, whether the simulated load patterns used in the system test are representative of the load patterns that will develop in the real application context. This question is very difficult to answer with confidence.

Replica Determinims. State synchronism is difficult to achieve in asynchronous ET-systems based on a dynamic preemptive scheduling strategy.

Consider the case a fault-tolerant distributed system. Two identical fail-silent nodes operate in parallel in order to tolerate a crash failure of one of the nodes. Since the two processors are driven by different quartz crystals, their processing speeds will not be identical. In case a significant external event requires an immediate task preemption, the two processors will most probably be interrupted at different points of their execution sequence. It may even happen that the faster processor has already finished its current task, while the slower one has to execute a few more instructions. Since in this case only the slower task will have to perform a context switch, the state synchronism between the two processors is lost.

Error detection. In an ET architecture the point in time, when a message will be sent, is only known to the sender. Therefore a message loss can only be detected by a bidirectional communication protocol, e.g., of the PAR type. Error detection at the receiver requires an additional mechanism, e.g., a watchdog timer.

5.2 Time triggered systems

In a time-triggered (TT) architecture all system activities are initiated by the progression of time. There is only one interrupt in the system, the periodic clock interrupt, which partitions the continuum of time into a sequence of equidistant granules. The state variables of the control object are observed (polled) at recurring predetermined points in time.

Flow Control. The flow control in a TT-system is *implicit*. During system design appropriate observation, message, and task activation rates are determined for the different state variables of the RT-object, based on their specified dynamics. It has to be assured at design that all receiver processes can handle these rates. If the state variables change faster than specified, then some short lived intermediate states will not be reflected in the observations and will be lost. Yet, even in a peak load situation, the number of messages per unit time, i.e., the message rate, remains constant.

This implicit flow control will only function properly if the instrumentation of a TT-system supports the *state view*. If necessary, a local microcontroller has to store the events that occurred in the last polling cycle and transform them into their state equivalent. Consider the example of a push button, which is a typical *event sensor*. The local logic in this sensor must assure that the state "push button pressed" is true for an interval that is longer than the granularity of the observation grid.

Scheduling. Operating systems for TT-systems are based on a set of static predetermined schedules, one for each operational mode of the system. These schedules gurantee the deadlines of all time-critical tasks, observe all necessary task dependencies and provide an implicit synchronization of the tasks at run time. At run time a simple table lookup is performed by the operating system to determine which task has to be executed at particular points in real time, the grid points of the *action grid* [5]. The difference between two adjacent grid points of the action grid determines the *basic cycle time* of the real-time executive. The basic cycle time is a lower bound for the responsiveness of a TT-system.

In TT-systems all input/output activities are preplanned and realized by polling the appropriate sensors and actuators at the specified times. Access to the LAN is also predetermined, e.g., by a synchronous time division multiple access protocol.

In the MARS system [1], which is a TT-architecture, the gridpoints of the observation grid, the action grid and the access to the LAN are all synchronized with the global time. Since the temporal uncertainty of the communication protocols is smaller than the basic cycle time, the whole system can be viewed as a distributed state machine [8].

Testing for Timeliness. In a TT-system, the results of the performance test of every system task can be compared with the established detailed plans. Since the time-base is discrete and determined by the granularity of the action grid, every input case can be reproduced in the domains of time and value. The temporal encapsulation of the nodes, achieved by the TDMA communication protocol, supports constructive testing.

Replica Determinism. In a TT-system all task switches, mode switches, and communication activities are synchronized globally by the action grid. Nondeterministic decisions can be avoided and replica determinism can be maintained without additional interreplica communication.

The basic cycle time of a TT-system introduces a discrete time base of specified granularity. Since a TT-system operates quasi-synchronously, TMR structures as well as selfchecking duplex nodes can be supported for the implementation of active redundancy without any difficulty.

Error Detection. In a TT-system the error detection is performed by the receiver of the information based on the global knowledge about the expected arrival time of each message. Fault-tolerance can be achieved by massive redundancy, i.e., sending a message k+1 times if k transient failures are to be tolerated.

The periodic transmission of message rounds makes it possible to implement efficient membership protocols in a TT architecture. Such a membership protocol informs sender and receiver about the proper operation of all nodes.

6. Consequences for the software engineer

Many real-time system designs are based on the principle of resource inadequacy[3]. It is assumed that the provision of sufficient resources to handle every possible situation is economically not viable and that an event triggered dynamic resource allocation strategy based on resource sharing and probabilistic arguments about the expected load and fault scenarios is acceptable. We call such systems *best effort systems*. These systems do not require a rigorous specification of the load and fault hypothesis. The design proceeds according to the principle "best effort taken" and the sufficiency of the design is established during the extensive test and integration phase.

At present, the majority of real-time systems is designed according to this best effort paradigm. It is expected that this will change radically in the future. The widespread use of computers in safety critical applications, e.g., in the field of automotive electronics, will raise the public awareness and concern about computer related accidents and force the designer to provide convincing arguments that the design will function properly under all stated conditions. On the other side, the decreasing cost of microelectronic components diminishes the economic necessity for resource sharing.

These developments offer excellent new possibilities for the software engineering community. If a software engineer can start from the specified fault and load hypothesis and can deliver a design that makes it possible to reason about the adequacy of the design without reference to probabilistic arguments, even in the case of the peak load and fault scenario, then we speak of a system with a *guaranteed response*. Guaranteed response systems are based on the principle of resource adequacy, i.e., there are enough computing resources available to handle the specified peak load and the fault scenario. The probability of failure of a perfect system with guaranteed response is reduced to the probability that the assumptions will hold in practice, i.e., the assumption coverage [6].

Considering the present state of understanding and the discussion in the previous sections, guaranteed response systems can only be designed in time triggered (TT) architectures. A consequent development of a software engineering methodology for the design of TT systems will thus have a marked impact on the computer industry.

References

1. Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P, and Schutz, W.The design of real-time systems: From specification to implementation and verification, Software Engineering Journal, May, 1991, p. 72 - 82

2. Kopetz, H., Kim, K., Temporal Uncertainties in Interactions among Real-Time Objects, Proc. of the 9th IEEE Symp. on Reliable Distributed Systems, Huntsville, Al, Oct. 1990

3. Lawson, H.W., Cy-Clone: An Approach to the Engineering of Resource Adequate Cyclic Real-Time Systems, Journal of Real-Time System, Vol.4, 1992, pp.55-83

4. Mok, A.K., Fundamental design problems of distributed systems for the hard real-time environment, Ph.D. dissertation, M.I.T., 1983

5. Specification and Design for Dependability, Esprit Project Nr. 3092 (PDCS: Predictably Dependable Computing Systems), 1st Year Report, LAAS, Toulouse, 1990

6. Powell, D., Fault Assumptions and Assumption Coverage, PDCS report RB4 (2nd year deliverable 1991) and Report LAAS, Toulouse Nr. 90.074, Dec. 1990

7. Sha, L., Rajkumar, R., Lehoczky, J.P., Priority Inheritence Protocols: An Approach to Real-Time Synchronization, IEEE Transactions on Computers, Vol. 39, No. 9, Sept. 1990, pp. 1175-1185

8. F.B. Schneider, Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, ACM Computing Surveys, Vol 22, Nr. 4, December 1990, pp. 299-320

FEATURES OF PROGRAMMING LANGUAGES

Chair: G. Pomberger

A Comparison of Modula-3 and Oberon-2

Laszlo Böszörmenyi

Institut für Informatik
Universität Klagenfurt
Universitätsstr. 65-67
A-9022 Klagenfurt / Austria

Keywords

Object-oriented programming languages, Modula-3, Oberon-2, Evaluation of software, Comparison of programming languages.

Abstract

Two modern programming languages - *Modula-3* and *Oberon-2* - are compared in respect to the way how they handle module interfaces, type equivalence, subtyping, concurrency and exception-handling. An assessment of the two languages is given discussing the value and cost of every feature.

Introduction

Two new programming languages are compared: Modula-3 [Nelson91] and Oberon-2 [Mössenböck91a, Wirth88]. Both languages are successors of Modula-2 [Wirth82]. Thus, they are quite similar, which makes it easier to compare them, but more difficult to evaluate the differences.

The comparison relys on the following principles: Features which can be implemented without compiler support (e.g., in a module or in a class) should not be incorporated into a language. Even those features should be omitted, which are expensive in the compiler, and could be implemented easily and with an almost full functionality without compiler support. On the other hand, features, which cause high costs in many user programs, should be incorporated into the language, even if it is expensive in the compiler. Features that enhance the safety of large programs should also be incorporated for (almost) any price.

The following comparison tries to concentrate on the *essential* features, which have a major influence on the global *structure* of programs, such as modules, classes, procedures, and processes. Moreover, the type systems of the two languages are compared. The presented features are investigated first of all from the point of their *cost/performance* ratio. *Safety* properties and*understandability* are also considered. An exhaustive comparison of the two languages is beyond the scope of this paper.

Both Modula-3 and Oberon-2 support strong type checking. They both support the notion of a module as the unit of compilation and static encapsulation. Both language support object-oriented programming by providing tools for subclassing with single inheritance. Both languages rely on garbage collection. The most essential differences are in their type systems, i.e., in the way they define type equivalence, abstract data types and inheritance.

Modula-3 is more powerful: it offers some features, which have no counterpart in Oberon-2. The following chapters will compare those features which are available in both languages and will make some cost/performance statements for those features which are only available in Modula-3.

1. Information hiding and module interfaces

Information hiding is one of the most important concepts of modern programming languages. Many object-oriented languages use the notion of a class both for specifying abstract data types and for information hiding [Meyer89]. In these languages the class is the basic unit for software construction, often it is the compilation unit as well.

Modula-3 and Oberon-2 share the notion of a module for information hiding and separate compilation. Modules are static units grouping together closely related data and code. They constitute a syntactical wall against other modules hiding their private data from illegal access. A module may explicitly export names which can then be imported by other modules (clients). The declarations of the exported names make up the interface of a module.

The module was already a central concept of Modula-2, where the interface of a module is given in a so-called definition module and the actual implementation in a corresponding implementation module. The designers of Oberon-2 and Modula-3 have agreed that the solution used in Modula-2 is insufficient. Consequently, the concept has been changed in both languages, and the differences between the two approaches are very instructive.

In Oberon-2 we have no explicit definition module. We write a module and simply mark those identifiers which should be exported, by an export mark (normally a "*"; the mark "-" can be used for read-only export). This solution is not only simple, but even selective. It makes it very easy to make only a part of a structure visible. In the following example

```
MODULE M;
        . . .
TYPE
  Rec* = RECORD
            f1*: INTEGER; f2-: REAL; f3: LONGREAL
          END
        . . .
END M.
```

the field *f1* is accesible in any other module that imports *M*, *f2* is accessible for reading and *f3* is unknown outside *M*.
The interface of a module can be extracted with the help of special tools (this extract may miss - depending on the tool - the original comments), resulting in a pseudo definition module.

In Modula-3, the concept of the explicit interface unit has not been omitted, quite the opposite, it has become more powerful than in Modula-2. In contrast to Modula-2, an interface can be implemented by more than one implementation modules, and an implementation module may export more than one interfaces. The first feature can be used to break a large implementation into several modules. The second feature can be used to distinguish between parts of an interface, e.g., to export one interface for everybody, and a second one only for trusted clients. For hidden and partially hidden types the notion of opaque types is introduced (Modula-2 has the same notion, but with poor semantics). An opaque type is a name that denotes an unknown subtype (see below and [Nelson91]) of some reference type. Different scopes can reveal different information about an opaque type, i.e., there may be several partial revelations and one complete revelation of an opaque type. The complete revelation must be branded (see below); this makes the type unique.

These differences are very typical for the different views (or "paradigms", to say it nicely) behind the two languages. If we look at the implementation cost, it is obvious that Oberon-2's solution is much cheaper for the compiler, because it does not have to check, whether the declarations given in an interface are equivalent with those in the corresponding implementation. In Modula-3, even the linker must be involved, because it has to check whether the impelmentation of an interface is complete and unambiguous. Even at the user's side, on first sight, it seems to be cheaper to just mark some names rather than to write out an explicit interface specification. Oberon-2's solution has a programmer in mind, who explores his data structures and algorithms, and eventually marks those names he believes to be useful for the outer world. This idea works fine in the case of smaller systems. However, in the case of large systems, it is the interfaces which play the central role in the design process, the actual implementations are almost secondary. Therefore, the additional burden, put by the necessity of defining explicit interfaces is an advantage rather than a disadvantage. Modula-3 suggests (almost forces) the programmer to design his/her interfaces separately from the actual implementation. Oberon-2 does not forbid that either, but it is not the implicit suggestion. Programming languages are not just tools for ideally trained programmers; they teach through their implicit suggestions as well.

As a consequence, we may say that the way to sepcify module interfaces in Oberon-2 is efficient and convenient. However, Modula-3 suggests a better style of programming, which - in the case of large systems - may be worth the additional effort, made by the compiler and the user.

2. Types

The most interesting part of the comparison of Modula-3 and Oberon-2 is their type systems.

2.1. Type equivalence

Modula-3 uses *structural equivalence*, Oberon-2 uses *name equivalence* (similar to Modula-2).

Name equivalence in Oberon-2 means that two types are the same, if they are denoted by the same identifier, or if they are declared explicitly to be the same (in the form of T1 = T2). Oberon-2 defines the notion of *equal* types as well in the sense that two types are equal if they are the same, or if they are open array types with equal element types, or procedure types with the same formal parameter list. The latter notion is obviously a kind of structural equivalence, so we can say, Oberon-2 uses mainly name equivalence, and makes an exception for open array and procedure types. (Open arrays were stepchildren already in Modula-2, and they still have an exceptional status in Oberon-2.)

Let us take the following example:

```
TYPE
 T1 = ARRAY 10 OF INTEGER;
 T2 = ARRAY 10 OF INTEGER;
 T3 = T2; T4 = T3;
 O1 = ARRAY OF INTEGER;
 O2 = ARRAY OF INTEGER;
```

In Oberon-2, T1 and T2 are distinct, T2 and T3 are the same (T4 and T2 are hoped to be the same, actually it does not follow necessarily from the definition), and O1 and O2 are equal. It is not so easy to understand, why O1 and O2 are "more equal" than T1 and T2.

Structural equivalence in Modula-3 means that two types are the same if their definition becomes the same when expanded; i.e., if all constant expressions are replaced by their values and all type names are replaced by their definitions. (In the case of recursive types, expansion is defined as the infinite limit of the partial expansions, which is probably not an easily understandable concept.)

To give some examples [Nelson91]:

```
TYPE
  R1 = RECORD a: INTEGER END;
  R2 = RECORD b: INTEGER END;
  List1 = REF RECORD x: INTEGER; link: List1 END;
  List2 = REF RECORD x: INTEGER;
          link: REF RECORD x: INTEGER; link: List2 END;
     END;
```

Types R1 and R2 are different (the type constructor is the same, but the arguments of the type constructor - the name of the record fields - are different). List1 and List2 are the same, because they both lead to the same infinite expansion (and therefore, they both can be reduced to the same canonical form: List1).

The main problem with structural equivalence is that equivalence of the structure of two types may be accidental. For example, if we write

```
TYPE
  Apples  = REF RECORD count: INTEGER END;
  Oranges = REF RECORD count: INTEGER END;
  Fruits  = REF RECORD count: INTEGER END;

PROCEDURE Q(fruit: Fruits) = . . .
PROCEDURE P(apple: Apples) = . . .
```

structural equivalence allows us to call Q with arguments of type Fruits, Apples and Oranges (which is probably desirable), but it also allows us to call P with an argument of type Oranges (which is probably undesirable). The latter problem can be solved in Modula-3 by using branded types. A branded type is unique, regardless of its structural identity with other types. The brand may be a user-defined string, or an implicit unique brand, assigned by the system.

So, we can say that Modula-3 uses structural equivalence which has to be restricted in certain cases and Oberon-2 uses name equivalence which has to be extended in some other cases.

The important question is: which type system is better? In [Nelson91] we find the honest statement that it is lastly a matter of taste. Maybe that's the true answer, but a bit more exact evaluation might be more helpful. Implementing name equivalence in the compiler is obviously much cheaper than implementing structural equivalence (it is trivial to compare two names, but it is non-trivial to compare structures that may be even recursive). However, there are cases, where user programs have to pay a high price if structural equivalence is not available. Many operations are only related to the structure of some data, and in those cases name equivalence is a severe restriction. Another benefit of structural equivalence is that it makes it easier for two programs to exchange data structures (via a file or a network) that are structurally equivalent but not necessarily declared with the same type identifier. The Modula-3 environment allows the programmer to store a data structure of type T1, together with its type information. When the data structure is read into some variable which has the type name T2, it is automatically checked whether the two types are equivalent. The Oberon-2 environment also allows a programmer to store a data structure together with its type name. However, other programs can read this data only into variables with the same type name.

From a didactical point of view, structural equivalence seems to be more natural. On the other hand, structural equivalence combined with opaque and branded types might be more difficult to understand than name equivalence with its few exceptions. As a consequence, we may say that for an undergraduate course, Oberon-2's type equivalence notion can be preferred. However, if we want to consider persistent or remote objects (either in education or practice), then Modula-3's concept of type equivalence seems to be a good value for its price.

2.2 Classes and inheritence

For object-oriented programming, probably the most important question is: how does a language express classes (abstract data types) and class hierarchies (inheritance)?

Oberon-2 provides *extensible record types* and *type-bound procedures* to express classes with methods. It is interesting to mention that Oberon - the direct predecessor of Oberon-2 - had no type-bound procedures, actually the only language feature for expressing classes was the notion of extensible records. It should be mentioned as well that this very fundamental concept was already sufficient to implement a substantial object-oriented operating system [Wirth89a, Reiser91].

Modula-3 provides the *object* type to express classes with methods. An object is always a reference (if not NIL) to a data record paired with a method suite. If *o* is an object, *f* a data field, and *m* a method of the object, then *o.f* is a reference to the field *f*, and *o.m(...)* is a call on the method *m*. Object types cannot be dereferenced, i.e., the entire data record cannot be referenced, only the individual fields.

Modula-3 defines a general subtyping rule, which can be applied to several kinds of types. The subtyping relation is denoted by "<:", and the general rule says: If T <: U, then every value of type T is also a value of type U. This rule can be applied to objects, procedures, arrays, references, subranges and packed types.

The declaration and usage of classes in Modula-3 and Oberon-2 is compared in Table 1. (Note that in Modula-3, objects are always references, in Oberon-2, they may be both pointers and records.)

	Oberon-2	Modula-3
	TYPE	TYPE
1	SuperR = RECORD f1: INTEGER END;	---
2	SubR = RECORD(SuperR) f2: REAL END;	---
3	Super = POINTER TO SuperR;	Super = OBJECT f1: INTEGER END;
4	Sub = POINTER TO SubR;	Sub = Super OBJECT f2: REAL END;
	VAR	VAR
5	superR: SuperR; subR: SubR;	---
6	super: Super; sub: Sub;	super: Super; sub: Sub;
	...	...
7	super:= sub;	super:= sub;
8	sub:= super(Super);	sub:= super;
9	superR:= subR;	---
10	subR:= superR(Sub);	---
11	IF super IS Sub THEN ...	IF ISTYPE(super, Sub) THEN ...
12	super(Sub).f2:= 1.1;	NARROW(super, Sub).f2:= 1.1;

Table 1.

In Oberon-2, SubR is an *extention* of SuperR (SuperR is the *base* type of SubR) and therefore *inherits* the field f1. SubR adds a new field f2. Pointers take over the extention relation of records, so Sub is also an extention of Super. Extended types can be regarded as subtypes of their base types (which correspond to supertypes in this case).

In Modula-3, Sub is a *subtype* of Super (Super is a *supertype* of Sub). Sub inherits the field f1 and adds the field f2.

A subtype object can be assigned to a supertype variable (lines 7 and 9). In the record assignment (available only in Oberon-2) only the field f1 is assigned (corresponds to a projection of the variable's value onto the subspace spanned by the base type [Wirth89b]). A supertype object can only be assigned to a subtype variable if its run-time type (its dynamic type) is this subtype. This requires a run-time type check which has to be written as an explicit *type guard* in Oberon-2 and is done implicitly in Modula-3 (lines 8 and 10). If this type check fails, a run-time error occurs. Line 11 shows a *type test*. It checks whether the dynamic type of super is Sub. Line 12 shows, how to use fields which are not part of the static type but which belong to the dynamic type - via a type guard in Oberon-2, and via the narrow statement in Modula-3.

Both languages allow to associate operations with objects, i.e., to specify methods (in Oberon-2 they are called type-bound procedures, in Modula-3 they are called methods). The methods associated with a supertype are inherited by the subtype and can be overridden there. Both languages allow to call an overridden method of a supertype (to make a super call). In Modula-3, method names can be even redeclared in a subtype, in which case the original names are masked by the new ones. The old names can, however, be accessed by using *narrow* (which should be better called *broaden* in this case). Redeclaring a method can be used among others to change a method's parameters in the subtype (the parameters of the overriding and the overridden methods must be of course the same). The possibility to both redeclare and override a method is powerful but maybe confusing.

Let us now try to compare the two approaches.

The concept of extended records in Oberon-2 is simple and together with the appropriate rules for assignments, it can be used to express a class hierarchy. The interesting point is that in Oberon-2 subclassing is expressed in terms of "conventional" concepts, i.e., Oberon-2 introduces object-oriented programming in terms of non-object-oriented concepts. However, the lack of an explicit subtyping rule is confusing - to my opinion. The concept of type extension alone is not sufficient to express subclassing. This must be explicitly expressed with the help of the rules of assignments. These are even different for pointer and record variables, because the latter normally do not change their dynamic types, except when they are passed via a *var* parameter - which might be not so easy to understand for undergraduate students.

Modula-3 has a general subtyping rule, and the object type-hierarchy is a natural application of this rule on object types. Therefore, in Modula-3, the assignment rules of objects *follow* from the subtyping rule, while in Oberon-2, subtyping is partly *defined by* the assigment rules. Thus, this issue is better defined in Modula-3. In Modula-3, records, referenced by an object type, do not fall under the subtyping relation, therefore, the rules defining the cases when a variable changes its dynamic type are simpler. The price for the this is that objects are always references. Apart from this difference, the two approaches have about the same power and their implementations should cost about the same.

2.3. Procedures

Both languages support the notion of a procedure. Procedures in Oberon-2 are very similar to procedures in Modula-2 (apart from type-bound procedures). In Modula-3, there are quite a few additional properties of procedures. Besides value and variable parameters, read-only parameters are available as well. Functions may return values of any type but an open array. (Oberon-2 restricts function return types to basic types and pointers.) Formal parameters may have default values which are taken if the corresponding actual parameter is missing. Binding of parameters may be by position or by keyword.

These features can all be implemented efficiently. Read-only parameters can be used to pass larger types efficiently (via a reference). Default parameters can be used to simulate procedures with a variable number of parameters. Restricted function return types are a matter of discussion. In principle, basic types and pointers are sufficient, since a complex type can always be substituted by a pointer. However, this restriction is not only inconvenient for the programmer but also expensive because working with data on the heap is usually more expensive than working with data on the stack.

3. Exceptions

Modula-3 provides exception handling, Oberon-2 does not. In Modula-3, exceptions can be declared (with an optional parameter), they can be raised and can be caught by exception handlers. Raising an exception exits active scopes repeatedly until a scope is found for which an exception handler is declared. If there is no such scope, the computation terminates in some system-dependent way (e.g. by calling the debugger).

An exception can be caught by the TRY statement:

```
TRY
Body
EXCEPT
id1 (v1) => Handler1
| . . .
| idn (vn) => Handlern
ELSE Handler0
END
```

id1 to idn stand for exception names, v1 to vn for parameters of the exceptions.

Exceptions are in dispute, especially because they can be easily misused for masking some errors [Meyer89]. With that point of view, the *else* clause is especially dangerous, because it catches and handles all non-expected errors. This could be extremely bad if an implementation module hides an ill-designed exception handler which simply "swallows" some errors without notifing its clients. Another difficulty with exceptions is that they are often used in a bad style. For example, in the module *Scan* in the SRC library [Harbison92], which exports procedures for reading data in an expected format, e.g. integers, reals etc., the exception *BadFormat* is raised, if the input does not conform to the expected format. This usage of exceptions regards a mistyped user input as an exceptional case - which should be considered rather normal.

Another form of exception handling is used for finalization. In this case, the TRY statement has the form:

```
TRY S1 FINALLY S2 END
```

This statement excutes S1 and after that S2 even if an exception was raised in S1. After executing S2 the exception is propagated to the enclosing scopes to be caught by an exception handler there. This kind of try statement construct can be used for finalization (e.g. closing files) in the case of errors. It can enhance the safety of programs considerably.

The question is again, is it worthwile to burden the compiler with exception handling? An ill-designed exception handling system could confuse everything by ignoring errors that shouldn't be ignored or by handling them at the wrong place. On the other hand, the lack of exception handling can easily lead to systems which react to exceptional cases in an extremely rigid way. Compiler support for exception handling is worth its price, if a fast reaction on errors is required, or if the loss of data (e.g. loss of files that could not be closed by a crashed program) is critical.

4. Concurrency

Oberon-2 does not provide any special support for concurrency, Modula-3 provides the data type *mutex* and the *lock* statement to support concurrency. Moreover, a standard library module is available that provides threads.

It is interesting to look at the way how concurrency is supported in the Modula family of languages. The original Modula language [Wirth77] still had the concept of concurrent processes, of mutual exclusion and a slightly modified version of Hoare's monitor concept. These concepts were replaced in Modula-2 by the more fundamental concept of coroutines. In Modula-2, a coroutine is a procedure that can be started as a quasi-parallel process with explicit points of control transfer. Thus, Modula-2 threw some concepts out of the language and expressed them in terms of others (coroutines are expressed in terms of procedures). Beside the theoretical beauty, the solution of Modula-2 gives entire freedom in writing schedulers, without forcing any given concept (e.g. that of the monitor) on the user. The price for this freedom is the loss of language support for expressing parallel concepts, which is quite a high price.

Regarding this history it is not too surprising that in Oberon-2 all support for concurrency has been moved from the language to a module providing coroutines. This is especially understandable if we consider the Oberon operating system, which uses a very special approach to support multi-tasking without multi-processing [Wirth89a].

Now, let us compare the costs. The Oberon implementation has obviously no costs at all. In Modula-3, the actual costs in the compiler are quite low, since the language supports only the mutex type and the lock statement. Mutex is an opaque subtype of *root* - the root of all objects. As a consequence, we can declare additional object types which are subtypes of mutex. This way to define objects for which mutual exclusion is necessary, is not only a convenient but also an efficient way.

The semantics of the lock statement is defined as follows. If *S* is a statment, we may write:

```
VAR m: MUTEX
 ...
LOCK m DO S END.
```

The lock statement is equivalent to:

```
Thread.Acquire(m);
TRY S FINALLY Thread.Release(m) END
```

Thread.Acquire(m) and Thread.Release(m) are procedures exported by the Thread interface [Nelson91] and do what their names suggest; Aquire locks m (waits if the lock is already held) and Release unlocks it. The essential part of the story is the way how Release is used. The *finally* part of the try statement is executed even if S fails. Thus, even erronous programs can use the locking feature safely. This kind of safety can hardly be achieved without language support.

Let us consider another example:

```
LOOP
 ...
 LOCK m DO
  ...
  IF b THEN EXIT END
   (*EXIT raises the exit-exception and jumps to the statement after the END of LOOP*)
  ...
 END (*LOCK*)
 ...
END (*LOOP*)
```

A loop statement (LOOP S END) executes *S* repeatedly until the exit-exception is raised. As a consequence, in the above example, the exit-exception forces the call of Thread.Release, and *m* will be unlocked. This is another example of how to get safer and simpler user programs for a moderate price in the compiler.

Coming back to the comparative question, we may state that the lack of any support for concurrency in Oberon-2 is only acceptable if we really do not need concurrency. The solution of Modula-3 is efficient and moderate and, therefore, is worth its price. However, Modula-3 supports concurrency adapting an implicit model of communication - via a common store. It is an open question at the moment, which language could provide a better support for a distributed memory model.

5. Additional features of Modula-3

5.1. Modified features from Modula-2

Some features that are unsatisfactory in Modula-2, are omitted in Oberon-2 and redefined in a clean way in Modula-3.

The type *cardinal*. Cardinals were introduced in Modula-2 with poor semantics (e.g., they are assignment-compatible but not expression-compatible with integers). In Modula-3, cardinal is defined as a subrange of integer (which is a clean notion).

Subranges. Modula-2 subranges are not quite clean either (they follow a special type equivalence rule). In the elegant solution of Modula-3, the subtyping rule is applied to them.

Enumerations. Identifiers of a Modula-2 enumeration list may cause name clashes if the enumeration type is imported. In Modula-3, the identifiers of an enumeration list must be qualified by the name of the enumeration type.

5.2. New features in Modula-3

The following list contains a number of Modula-3 features, which are neither available either in Modula-2 nor in Oberon-2.

Initialization of variables. Variables can be initialized at their declaration. This feature is especially useful in the case of arrays and records.

Generics. In a generic interface or module, some of the imported interface names are treated as formal parameters, to be bound to actual interfaces when the generic module is instantiated.

Isolation of unsafe code. In unsafe modules low-level programming features are available, as explicit storage disposal or unchecked type transfer.

6. Implementation

It is a difficult question, to which extent actual implementations should be considered, when comparing languages. Implementability surely has to be considered, but probably not actual implementations. However, it must be stated that at the time being, there is a specific difference between the implementations of Oberon-2 and Modula-3.

Oberon-2 has an extremely fast compiler, integrated into a convenient programming environment. Modula-3 has a slow compiler with some modest support, embedded in a not very friendly environment.

This difference could be regarded as a temporary prove that the design of Oberon-2 is superior. It is noteworthy to mention that the design process of the Oberon-2 language started with the absolute minimum considered [Wirth88], and later, on the basis of experiences, some further features (e.g. type-bound procedures) were added. The opposite approach - first provide more features than necessary, and select the necessary ones later - has no chance to succeed. If a feature is introduced into a language, one can be sure that some people will use it and find it indispensable.

If a better Modula-3 implementation will be available soon, which allows for an efficient use of the more powerful features of this language, we may hope that users will be able to choose between the two languages on the basis of their needs and not on the basis of the availability of appropriate implementations.

Conclusion

There is a continuously growing need for evaluating programming languages (and other software designs as well). However, there are no exact methods to do that, programmers prefer to speak about their "favorite" languages, which is a sign for the "subject-oriented" approach, used in selecting programming languages. In this paper, an attempt was made to compare and evaluate two programming langugaes in a fairly objective manner, with moderate efforts. Two modern languages, Modula-3 and Oberon-2, were compared. Both languages were found to be clean and consistent. Oberon-2 generally takes the simpler way, Modula-3 is more powerful and more expensive. As a consequence, Oberon-2 fits better for small programs and undergraduate courses, Modula-3 fits better for large programs (possibly in a distributed environment) and for teaching more advanced features.

Acknowledgements

My thanks go to H. Mössenböck, J. Templ, H. Eberle, G. Nelson, M. Jordan, B. Kalsow and R. Mittermeir for many inspiring discussions.

References

[Harbison92] S. P. Harbison: *Modula-3*; Prentice-Hall, Englewood Cliffs, NJ, 1992

[Nelson91] Greg Nelson et al.: *Systems Programming with Modula-3;* Prentice Hall, Englewood Cliffs, NJ. 1991

[Meyer89] From Structured Programming to Object-Oriented Design: *The Road to Eifel;* Structured Programming Vol.10, No.1 1989

[Mössenböck91a] H. Mössenböck, N. Wirth: *The Programming Language Oberon-2;* Structured Programming Vol.12, No.4 1991

[Mössenböck91b] H. Mössenböck: *Object-Oriented Programming in Oberon-2*; 2nd International Modula-2 Conference; Loughborough, September, 1991

[Reiser91] M. Reiser: *The Oberon System;* User Manual and Programmer's Guide Addison-Wesley, 1991

[Wirth77] N. Wirth: *Modula - A language for Modular Multiprogramming;* Software Practice and Experience, Vol.7, No.1, 1977

[Wirth82] N. Wirth: *Programming in Modula-2;* Springer Verlag, 1982

[Wirth88] N. Wirth: *The Programming Language Oberon*; Software Practice and Experience, Vol.18, No.7, 1988

[Wirth89a] N. Wirth J. Gutknecht: *The Oberon System;* Software Practice and Experience, Vol.19, No.9, 1989

[Wirth89b] N. Wirth: *Modula-2 and Object-Oriented Programming;* Proc. of the First International Modula-2 Conference; Bled, Yugoslavia, 1989.

Appendix

As a matter of interest, two simplified versions of a generic binary tree are given, implemented in Oberon-2 and in Modula-3. The tree is generic; it does not make any assumption about the type of the search keys. The Oberon-2 version was designed by H.P. Mössenböck, the Modula-3 version by the author.

The Oberon-2 version consists of a single module, the exported identifiers are marked (by * or -):

```
MODULE BinTree;  (* HM 11.6.91 *)

 TYPE
   Node* = POINTER TO NodeDesc;
   NodeDesc* = RECORD
     left, right: Node
  END;

   Tree* = RECORD root: Node END;

 PROCEDURE (x: Node) less* (y: Node): BOOLEAN;  (*abstract method*)
 END less;
 PROCEDURE (x: Node) equal* (y: Node): BOOLEAN;  (*abstract method*)
 END equal;

 PROCEDURE (VAR t: Tree) Insert* (n: Node);
   VAR p, father: Node;
 BEGIN p := t.root; father := NIL;
  WHILE p # NIL DO
    IF p.equal(n) THEN RETURN END;
    father := p;
    IF n.less(p) THEN p := p.left ELSE p := p.right END
  END;
   n.left := NIL; n.right := NIL;
   IF father = NIL THEN t.root := n
   ELSIF n.less(father) THEN father.left := n
   ELSE father.right := n
  END
 END Insert;

 PROCEDURE (VAR t: Tree) Init*;
 BEGIN t.root := NIL
 END Init;

END BinTree.
```

The Modula-3 version consists of an interface and an implementation. The types *PublicNode* and *PublicTree* are entirely revealed in the interface. The methods in *PublicNode* are deferred [Meyer]: they must be overriden by the user, otherwise an exception is raised. *Tree* and *Node* are revealed in the impelementation module.

```
INTERFACE BinTree;  (*LB 30.01.92*)

 TYPE
   Node  <: PublicNode;
   Tree  <: PublicTree;

   PublicNode = OBJECT
          METHODS
            less (y: Node): BOOLEAN; (*abstract method*)
            equal (y: Node): BOOLEAN; (*abstract method*)
          END;

   PublicTree =  OBJECT
          METHODS
            init ();
            insert (n: Node);
          END;

END BinTree.


MODULE BinTree;  (*LB 30.01.92*)

 TYPE
  REVEAL Node = PublicNode BRANDED OBJECT
            left, right: Node;
         END;

  REVEAL Tree = PublicTree BRANDED OBJECT
            root: Node;
         OVERRIDES
            init:= InitTree;
            search:= Search;
            insert:= Insert;
         END;

  PROCEDURE Insert(t: Tree; n: Node) =
  VAR father: Node := NIL; p: Node := t.root;
  BEGIN
   WHILE p # NIL DO
    IF p.equal(n) THEN RETURN END;
    father:= p;
    IF n.less(p) THEN p:= p.left ELSE p:= p.right END;
   END; (*WHILE*)
   n.left:= NIL; n.right:= NIL;
   IF father = NIL THEN t.root:= n
   ELSIF n.less(father) THEN father.left:= n
   ELSE father.right:= n
   END
  END Insert;

  PROCEDURE InitTree(t: Tree) =
  BEGIN
   t.root:= NIL
  END InitTree;
BEGIN
END BinTree.
```

Discrete event simulation in object oriented languages

Gy. Gyepesi, T. Szép, F. Jamrik, G. Janek, E. Knuth

Computer and Automation Institute
Hungarian Academy of Sciences
H-1519 POB. 63, Budapest

Abstract. Those who remember SIMULA 67, the grandmother of object oriented languages, know that it contained powerful and elegant mechanisms for the control of quasi-parallel processes and a high level technique for discrete event management based on the concept of an abstract time axis. Surprisingly, none of the modern object oriented languages implemented these particularly useful concepts. This paper presents an approach how two of the leading object oriented languages C++ and (a dialect of) Smalltalk have been extended to incorporate such mechanisms.

1. Introduction

As it is known, the language SIMULA [9] (formerly called SIMULA 67) played a pioneering role in the advent of object oriented technologies. Though it lacked some of the important modern concepts like polimorphism and encapsulation, however, it contained a particularly effective concept the discrete time oriented quasi-parallel behaviour, never again implemented by other object oriented languages.

The fundamental notion in SIMULA is the abstract time axis which transparently and dynamically controls the scheduling of all process objects of the system (amongst them the user program as a whole too). To implement this behaviour, SIMULA used a special version of basic quasi-parallel control primitives. Of course, the higher level behaviour and the language formalisms associated can also be built over any other known parallel or parallel engines (like the ones given in papers [Ghezzi 85, Muhlbeim 88, PARLE 87, Ruppelt 89, Thomas 87]).

The following paragraphs summarize the concepts used by the SIMULA language.

1.1. Quasi-parallel sequencing

This concept offers a low-level control mechanism enabling us to suspend the execution sequence of statements at certain points in class bodies in such a way that a) the whole environment is preserved and; b) the control can at any time be resumed again. At a given moment any number of suspended execution sequences (in object instances) can coexist. Suspended objects can later be resumed by sending them a "resume" message from any other object (that is they are awakened explicitly, in contrast to the way explained in 1.2).

Statements implementing this behaviour in SIMULA are denoted as *detach*, *resume(object)*, and *call(object)*. They are accessible for the users, though normally they are not used explicitly. The main purpose of these procedures is providing a basis for the discrete time oriented behaviour of processes as described in the next point.

1.2. The time axis

Historically, the simulation of discrete event based parallel processes was the basic paradigm SIMULA addressed (and solved, in fact, in a far more elegant way than its competitors like GPSS [3], and SIMSCRIPT [Johnson 72]). It invented a more general concept, nowadays called "object orientation" by chance.

The main concept of SIMULA for discrete time oriented behaviour is the class *process*. Instances of this class (i.e. process objects) can be scheduled dynamically in a simulated time axis (called the *sequencing set*). The real fun starts when the main program (by convention, a process object too) suspends itself. At this point the time axis gains control and governs all further behaviour. Processes schedule continuations of themselves (or of other processes) for given time points dynamically at the time axis. These continues until no further events are scheduled.

More exactly, the behaviour based on the time axis is implemented by the following main commands:

hold(interval)
The process issuing this command is suspended. When the time interval specified is elapsed (in simulated time), the process is resumed automatically.

passivate
The process issuing this command is suspended, but not scheduled for reactivation. (Passive processes can only be activated by other processes in an explicit way.)

activate process *at/delay* time
By this command processes can schedule (or reschedule) the activation time of other processes. The clause "at" refers to (simulated) absolute time points, while "delay" refers to relative ones.

cancel process
Cancel the scheduling of a given process (if exists).

There are several other useful commands available, but not detailed here.

1.3. An example

The above concepts can well be illustrated by the following beautiful example (published by the University of Oslo many years ago, nevertheless it does nothing with simulation indeed). It is perhaps the most elegant prime number generator available in the literature. The whole program looks like as follows:

```
 1:     begin

 2:     process class prime(p); integer p;
 3:     begin print(p);
 4:       while true do begin
 5:         if nextev.evtime-time>2
 6:         then activate new prim(time+2) delay 2;
 7:         hold(2*p);
 8:     end end;

 9:     activate new prime(3) at 3;
10:     hold(limit);
11:     end;
```

The algorithm prints (for simplicity, only the odd) prime numbers until the number "limit", and works as follows. Line 9 creates the first prime (prime 3) and schedules its activation at simulated time 3. Line 10 suspends the main program (until the limit is reached in simulated time) and passes control to the time axis. Since there is only one event scheduled at this moment (prime 3 at time 3), the time advances to 3 and prime 3 gains control (gets resumed, activated).

Prime 3 prints the number 3 according to line 3. An "infinite" loop begins then. First it is checked if the next scheduled event is farther than two units. (In fact, this is the essence of the algorithm. If not, the number time+2 is not a prime, as it will be obvious later.) Now the only further event is the main program scheduled at "limit" which we suppose is far enough. Therefore a new prime namely prime 5 is generated and scheduled at 3+2. Prime 3 is suspended then for a period of 2*3, that is it will continue - its own filtering work - at time 9.

After time 3, the next event prime 5 gains control at time 5. Since the next event (prime 3 at time 9) is farther than 2 units, it will generate a new prime, the prime 7, and the process continues. All the generated new prime processes advance filtering then in parallel.

The essence of the algorithm is a careful and elegant balance of control. Though the parallel processes proliferate, however, only those events are scheduled which are really needed for the temporal decision whether the next odd number is prime or not. This results in a particular efficiency in addition to beauty of the program.

2. The "Yarn" model

Below we introduce a model which can generally be used as a basis to implement quasi parallel behaviour in a variety of modern languages. We will use common terminologies of object oriented languages (like class, method, message, descendant, object, receiver, etc.) without explaining them.

Yarn

This phrase will refer to parallel branches in our model. Yarns are created by a special message which duplicates the creating branch (its local environment). Methods needed to implement the quasi parallel behaviour will be defined in a generic class named Yarn. The user code of a particular quasi parallel process is to be given by the method named *body* in a descendant class of Yarn. This way, any number of coexisting user processes can be created.

stitch

On creating a new branch the old one still keeps the control. The control can actually be passed by the special method "stitch(branch)".

back

A branch always remember the one activated it. The control can be given back by the method "back". (The method "stitch" can also be used for the same purpose.) On terminating a branch, the "back" method is automatically invoked.

Parallel branches can also communicate by the control passing methods. For this purpose parametric versions of them named *istitch* and *iback* are provided too. The parameter used can be of any object (except a Yarn one).

For more details of the exact behaviour and for additional methods introduced we refer to the technical definition of Yarn [Szep 92].

3. Extending the Actor system

3.1. The Actor language

The Actor language and environment [Franz 90] (trade mark of The WhiteWater Group, Evanston, Illinois, USA) is a true SMALLTALK [10] dialect. It differs only in certain notations and implementation techniques both for efficiency reasons. In fact, presently, Actor is the only professional SMALLTALK-type development environment for MS WINDOWS [5]. For this reason, we chose Actor as the basis for our extensions.

Like SMALLTALK, Actor is based on the message sending paradigm. Actor's general notation is the following:

```
message(receiver, arguments)
```

3.2. Implementation of Yarn in Actor

A single class named Yarn implements all the required behaviour. The Actor version of Yarn is based on a stack-saving technique. For efficiency reasons, only the part the stack which is used in the parallel work is duplicated. (Special tools are available to set or adjust its level.) The basic stack-saving methods are implemented on a binary level, and are not available for the users. On loading the Yarn extension of Actor, all the necessary binary adjustments are done automatically.

3.3. Methods implemented

The following methods are provided to realize the functionality defined in the Yarn model:

Def back(YarnClass)
Def iback(YarnClass,arg)
Return control to the calling branch.

Private body(Yarn,arg)
Dummy at the generic level. Must not be called explicitly.

Def close(YarnClass)
Terminate and delete all branches except the main one.

Def close(Yarn)
Terminate a particular branch.

Def fibre(YarnClass)
Returns the currently active branch.

Def from(Yarn)
Returns the branch which activated the current one by "stitch" or "istitch" (but not by "back" or "iback").

Def stitch(Yarn)
Def istitch(Yarn,arg)
Transfer control to the body of the receiver branch.

Def main(YarnClass)
Returns the main branch. This contains the original Actor environment and can not be terminated.

Def new(YarnClass)
Create a new parallel branch.

Def state(Yarn)
Return the current state of a branch. Possible values are: #active, #inactive, #terminated.

3.4. Example

All collection objects in Actor posses a *do* method having a block argument. On sending a "do" to any collection, the argument is executed for each of its elements. (This technique is elegant and particularly useful when members of the collection are not addressable directly like in cases of sets and trees.)

Unfortunately, the "do" can be sent to a single collection only. In many cases, it would be useful to traverse structures in parallel (like comparing or copying them). Using the Yarn technique, parallel versions of "do" methods can easily be defined however. An example is provided with Yarn which looks like:

parDo(Yarn,anArray,aBlock)

Parallel *do*. The array argument can be any array of collection objects. For instance, we can send the method in the following way:

```
parDo(Yarn, tuple(aTree1, aTree2), aBlock);
```

Now, if our purpose is to find the number of differences between the trees given, the argument block can be defined as:

```
aBlock :=
{ using(pair)
  if pair[0] <> pair[1]
  then differences := differences + 1
  endIf
}
```

(We note that this version of parDo terminates if any of the collections traversed are exhausted. For different behaviour, the user can define a private parDo in any other way.)

4. Simulation technique in Actor

Based on the methods of Yarn a discrete event oriented layer is also built. The corresponding simulation methods are given in three classes:

4.1. The Process class

This class is defined as a descendant of Yarn. Therefore, its time-controlled behaviour should be described in its "body" method. Special methods available are the following:

Def activateAt(Process,time)
Def activateAtPrior(Process,time)
Def activateDelay(Process,interval)
Def activateDelayPrior(Process,interval)

Schedule the activation/reactivation time of the receiver process at the given time or after the given interval. Prior schedules it as the first event at the given time point.

Def activate(Process)
Def activateBefore(Process,aProcess)
Def activateAfter(Process,aProcess)

Schedule with respect to another process on its activation time. The direct form "activate" means: after "current".

Def passivate(Process)

Stop the execution of the process (self) without terminating. Transfer control to the hidden time-control mechanism. (Passivated processes can then be activated by other ones.)

Def hold(Process,interval)
Suspends self for the specified period. (Schedules self at the time point current time + interval; and passivates self then).

Def wait(Process,EventQueue)
A useful utility which passivates the process and also adds it to an ordered collection (a queue - a typical one in simulation applications).

Def time(ProcessClass)
Returns the current value of the simulated time (as real).

Def current(ProcessClass)
Returns the process object currently possessing the control.

Def activity(Process)
This is the name of the method to be used to describe the body of the user process. Dummy at the generic level.

Def status(Process)
Returns the status of the process. Possible values are: #scheduled, #passive, #terminated.

Def cancel(Process)
Removes the scheduling notice of the receiver process.

Def evtime(Process)
Returns the time point (as real) at which the receiver is scheduled.

4.2. MainProcess

As a descendant of Process with a different body method, it serves to store the main simulation program. The only additional method provided is:

Def simulation(MainProcess)
Start the simulation.

4.3. The time axis

The time axis is simulated by the class named *SequencingSet* which is defined as a descendant of the Actor class OrderedCollection. It has no public methods. Once the "simulation" method is sent to the MainProcess, the SequencingSet governs all the control needed for the model.

4.4. Example

The following example is a simplified outline for a traffic simulation where cars arrive at a traffic light and wait in a queue until it is green. (For more exact description of the example we refer to [4].)

```
inherit(MainProcess,#TrafficSimulation,#(#queue,#lamp));
Def activity(self)
{ queue:=new(EventQueue,1);
  activate(lamp:=new(TrafficLamp));
  activate(new(CarGenerator));
  hold(self,limit);
}

inherit(Process,#CarGenerator,#(#no))
Def activity(self)
{ no:=0;
  loop while true begin
    no:=no+1;
    activate(new(Car),no);
    hold(self,random);
  endloop;
}

inherit(Process,#Car,#(no))
Def activity(self,n)
{ no:=n;
  hold(self,random);
  if size(queue)>0 or not(green(lamp))
  then
    wait(self,queue);
    continue(self)
  endif;
}

inherit(Process,#TrafficLamp,#(#green))
Def activity(self)
{ loop while true
  begin
    hold(self,random);
    green:=not(green);
    if green
    then
      activate(first(queue));
    endif;
  end;
}
```

5. Implementation in C++

The C++ implementation of both Yarn and Simulation is fundamentally the same as above, here, however, these are adapted to the different nature and style of the whole environment. Main differences are as follows:

a) The C++ version of Yarn is implemented by the stack-changing technique. It means that for each parallel branch a new stack is allocated from the Windows global heap. This leads to a different stack initialization technique (required to be tailored by the user, - not detailed here).
b) Since the C++ development environment is not interactive (in the way as Smalltalk), a couple of methods are not needed, however, a special technique is necessary to handle program termination.

Some of the most important methods implemented are the following:

static LPvoid back();
static LPvoid iback(LPvoid par);
Transfers control back to the one called the current.

virtual LPvoid body(LPvoid par) = 0;
Abstract method for the user body. All descendants must define it concretely.

static void exit(int status);
Equivalent to the standard "exit", but attempts to return to the main branch.

static LPvoid stitch(Yarn& dest);
static LPvoid istitch(Yarn& dest,LPvoid par);
Quasi parallel version of transferring the control. Returns when the control is returned from the called branch.

Further available methods are similar to those given for the Actor version. The simulation layer is also elaborated for the C++ case, this, however, is not detailed here.

6. Conclusions

A new technique with corresponding tools has been presented for MS Windows application programming consisting of two self-contained layers, one for pure quasi parallel programming, the other for a simulated time-controlled behaviour of processes built of discrete events. Quasi parallel programming is a reasonable alternative of the real parallel one for problems containing parallel components in nature. The simulation layer provides the forgotten special power of the SIMULA language in modelling the interaction of discrete processes.

The tools experimentally developed are now available for MS Windows 3.0 with Actor version 3.0 or 3.1, with Borland C++ 2.0 or 3.0 (moreover for Turbo Pascal for Windows too, - not detailed in the paper).

References

1. Franz, M. Object-Oriented Programming Featuring Actor. Scott, Foresman IBM Computer Books, USA, 1990.
2. Ghezzi, C. Concurrency in Programming Languages: A Survey. Parallel Computing 2(3), pp229-241, 1985.
3. GPSS, General Purpose Simulation System V, User Manual, IBM Corporation, 1991.
4. Johnson, G.D. SIMSCRIPT II.5, User's Manual, Release 6, C.A.C.I. 1972.
5. Microsoft Windows, version 3.0, Microsoft Corporation, 1991.
6. Muhlbeim, H. et al. MUPPET: A programming environment for message-based multiprocessors. Parallel Computing 8, pp201-221, 1988.
7. PARLE. Proc. Parallel Architectures and Languages Europe. Eidhoven, The Netherlands, Lecture Notes in Comp. Sci. Springer, 1987.
8. Ruppelt, Th., Wirtz, G. Automatic transformation of high-level object oriented specification into parallel programs. Parallel Computing 10, pp15-28, 1989.
9. SIMULA Standard. Simula Standards Group, Oslo, Norway. 1989..
10. Smalltalk-80. Byte Magazine, August, 1981.
11. Szep, T. Technical reference for Yarn (in Hungarian). Hungarian Academy of Sciences, Budapest, 1992.
12. Thomas, I. Object oriented programming on transputers. Proc. BCS Workshop on Parallel and Distributed Object Oriented Programming, 1987.

OBJECT-ORIENTED SOFTWARE DEVELOPMENT

Chair: P. Hanak

An Approach to the Classification of Object-Based Parallel Programming Paradigms

Georg Pigel
Institut für Statistik und Informatik
Abteilung für angewandte Informatik
Universität Wien
Lenaug. 2/8
1080 Wien, Austria

Abstract

This paper tries to present a classification scheme for object-based concurrent paradigms. Based on the discussion how concurrency is introduced into a system a classification scheme will be presented and applied on examples. Then the classification will be refined and corresponding features of our example systems discussed. In the end a summary will be presented and an outlook on further research will be given.

Keywords: Object-based, concurrency, classification.

1 Introduction

Creating software systems is a task proposing high demands on the developer's intellectual skills and creativity. Programming was thought to be an art for a long time, and it took until the mid-seventies to develop widely accepted software engineering techniques. But these techniques were not able to cope with the ever increasing demands on today's software. Especially high maintenance costs, missing concepts for reusability, and enormous difficulties in creating portable software together with the rise of new user interaction techniques (GUIs) and an increasing demand for distributed and parallel programming lead to the wish for new software development paradigms. So the new software development paradigm of "object-orientedness"* was born. The arguments for the use of object-oriented programming concepts stated by different authors are manyfold:

- Object-orientedness catches the whole world, consisting of data and functional aspects. It was claimed that functional decomposition (e. g. structured analysis [Gane 79]) only can catch the half of the real world, that consists of functional aspects. Data modelling (e. g. entity-relationship diagrams [Bach 73]) can describe the "data half" of the real world very well, but lacks of descriptive power of the functional aspects. Doing data modelling and functional decomposition in parallel leads to inconsistencies between the different documents produced as results, due

*As the definitions of object-oriented, class-based, and object-based are given some sections below, I will use object-oriented where I mean object-based, as it is the more known word, whereas the term "object-based" could spread confusion about its meaning.

to the different views of the world, thus opening a gap which sometimes becomes nearly unbridgeable and containing severe impacts on software quality [Coad 90].

- In [Cox 87] it is stated that bulk is bad. Long programs are harder to write, to debug, to maintain and to understand and reuse. With object-oriented techniques one can produce shorter programs, that are therefore easier to debug, maintain and reuse.

- Similarly it is claimed in [Cox 87] that "surface area" is bad. With "surface area" it is meant, what the programmer needs to know about a piece of code if he wants to use it. The concepts of information hiding, abstraction and the message passing mechanism contained in object-oriented systems make it not necessary to know anything about the implementation of an object, i. e. objects have well defined, clean interfaces. The programmer only needs to know a small "surface area" when using existing code.

Creating systems containing parallelism is even more complex than creating sequential systems. First of all not only shorter or longer pieces of sequential code (dependent on the grain of parallelism exploited in the system) have to be correct, but also the additional complexity of entities communicating in a practically unpredictable sequence has to be taken into account. There is a couple of reasons, why concepts related to object-oriented systems could help coping with this additional complexity:

- Thinking in terms of objects helps the developer to understand the problem space better, and therefore makes it easier to exploit the parallelism inherent to a certain application.

- Objects lend themselves to define the grain of parallelism to be exploited in the system. Of course it often is usefull to exploit parallelism inside an object to improve performance, but this does not add to the overall complexity, if this concurrency is strictly hidden from the outside, with objects seen as selfcontaining entities.

- As communication patterns were found to be the most important feature to classify parallel algorithms [Levi 87] [Nels 87] [Babb 87], the fact, that communication via message passing is an essential part of the object-oriented paradigm, and has not to be added in a more, but often less natural way makes this paradigm even more suited.

Parallel programming is most often motivated as being the most natural way of improving performance of a problem solution. But performance is one of the not-so-good points of current state of the art object-oriented environments.[†]

2 Examples for Object-Based Parallel Programming Paradigms

In the last years numerous programming languages and paradigms for object-oriented concurrent programming were introduced. The need to find out common trends or differences between those ideas, but also to gain a sound basis for comparing the different ideas, leads to the neccessity of a classification of the introduced models.

Due to lack of space, we will restrict ourselves to five widely known models for object-oriented concurrent programming. We will introduce them shortly and then we will use these models in the remaining part of this paper to demonstrate how to apply our scheme of classification.

[†]Interpretation instead of compiling, automatic garbage collection and late binding are the most power consuming features found in most object-oriented systems.

2.1 Actors

As there is a wide range of actor-based languages [Lieb 87] [Atta 87] [DiSa 91] [Loya 91] we have chosen to discuss the implementation independent basic concepts behind Actors as described in [Agha 89] [Agha 86].

Actors are computational agents, distributed in time and space. Each actor has a *mail address* and a *behaviour*. An actor can influence the actions of another actor by sending it (or itself) a message. To send a message, the mail address of the receipent must be known to the sender. In the actor model buffering of messages is provided, leading to asynchronous communication. Actors can be created dynamically. The state of an actor is defined by its *behaviour*. An actor is able to compute a *replacement behaviour*. Actors never change their behaviour (this is similar to the "singles assignment rule" known from dataflow languages), but create new actors with these newly computed replacement behaviours.

2.2 Smalltalk-80

As Smalltalk-80 is probably the most known object-oriented laguage, we will reference to [Gold 83] for a detailed language description and only explain how Smalltalk-80 handles parallelism. Without going into any details, it can be said, that concurrency is obtained in Smalltalk-80 by sending a *fork* message to a block context. In the following example, taken from [Yoko 87], after sending a *fork* message part (i) and part (ii) are executed concurrently .

```
/t1/
.....
[...(i)...]fork.
....(ii)....
.....
```

In the example $t1$ is a common variable. Mutual exclusion must be done explicitly by the methods which want to use the object using semphores. Semaphores are provided by a class *Semaphore*. This leads to a distributed form of control of synchronization which is hard to develop and debug, and a bit contrary to seeing objects as self contained entities. Messages have the semantic of function calls, i. e. the sender sends a message and is blocked until it receives a return value. Smalltalk-80 has been critisized because its extension to concurrency reminds more on conventional languages as *Parallel Fortran*, not fitting well into the object-oriented world. For details see [Gold 83].

2.3 ConcurrentSmalltalk

ConcurrentSmalltalk [Yoko 87] was developed on the basis of Smalltalk-80, but although one of the primary goals of its implementation was to keep binary code compatability to Smalltalk-80, there are certain differences. ConcurrentSmalltalk has objects as grain of parallelism. There are two kinds of message-passing mechanisms: *Synchronous method calls* which are compatible to Smalltalk-80 message passing. *Asynchronous method calls*, which have no equivalent in Smalltalk-80, allow the sender to continue working without waiting for the receiver to reply. Asynchronous method calls return a *CBox* to the sender. The return value of the object which received an asynchronous method call is buffered in the CBox until the caller retrieves the value. If the calling object tries to retrieve a return value not delivered yet, it is blocked. Therefore CBoxes also can be seen as a synchronization mechanism. For compatibility reasons to Smalltalk-80 shared variables are supported. As the mechanism of shared variables is contrary to the basic idea that an object is a selfcontained entity their use should be avoided. In ConcurrentSmalltalk there is no concurrency inside an object.

2.4 DistributedConcurrentSmalltalk

DistributedConcurrentSmalltalk is the extension of DistributedSmalltalk to a distributed interpersonal environment [Naka 89]. In DistributedConcurrentSmalltalk there can be multiple threads inside an object, differently to ConcurrentSmalltalk. These threads have to synchronize internally inside the objects by using guarded commands.

2.5 HOOD Nets

HOOD nets, as described in [Giov 90], are no programming language but a design paradigm. "HOOD (Hierarchical Object Oriented Design) is the standard ESA (European Space Agency) method for the architectural design phase of the Software Life Cycle" [Giov 90]. Details on HOOD can be found in [HOOD 89a] [HOOD 89b]. HOOD nets are based on the Petri net formalism, exactly said on high-level Petri nets [Genr 91] [Jens 91]. HOOD nets are normally used for developing systems implemented with ADA. Nevertheless they are programming language independent. HOOD objects consist of a public interface and an internal implementation (as is normal in object oriented concepts). A HOOD design document is a tree. A complex object can be splitted into several child objects of less complexity. Objects are re-entrantable, that means that synchronization mechanisms for conflicting methods inside an object must be provided. The control flow of the system is modelled by a OPeration Control Structure (OPCS). An OPCS net can be seen as a sequence of net blocks. Net blocks can be invoked iteratively, alternatively or in parallel.

3 Basic Features of Object-Based Concurrent Systems

Having introduced our example paradigms, it is time to define the basic terms, following the definitions in [Wegn 90] as far as possible. The definitions 1,2,3 and 4 are directly adapted from [Wegn 90].

Definition 1 (Object-Based Systems) *Object-based systems are systems whose basic entities are build on the concepts of consisting of data plus methods communicating via a message passing mechanism.*

Definition 2 (Class-Based Systems) *Class-based systems are object-based systems where each object belongs to a class.*

Definition 3 (Object-Oriented Systems) *Object-oriented systems are class-based systems where hierarchies of classes are build by inheritance relations.*

Let's apply these definitions on our examples. HOOD nets have objects as basic entities but no classes. Therefore they are object-based. Actors have objects, classes (because an equivalence relationship between actors with the same behaviour can be defined), but no inheritence relationship between classes. Consequently the Actors paradigm is class-based. Smalltalk-80, ConcurrentSmalltalk and DistributedConcurrentSmalltalk have objects, classes, and inheritence. Therefore they are truly object-oriented systems by the definition in [Wegn 90].

In the remainder of the paper we will not distinguish between object-based, class-based and object-oriented models, but lead our discussion in the widest possible range, i. e. object-based systems, for the following arguments:

- Classes are a very powerfull means for structuring the system statically, but per se have no impact on concurreny in a system, which is a dynamic feature.

- Inheritance has an impact on the run time behaviour of a system, but does not change the pattern of communication between objects. Besides that, we do not want to go into details of the semantics of inheritance and therefore will not take it into considerations in this discussion.

- As stated before, performance considerations are often the driving force behind the creation of concurrent systems. But object-oriented paradigms are normally inherently coupled with very dynamic allocation and freeing of system resources, most often done in a way transparent to the user. Hardware specific details are hidden from the user very strictly. But as parallel programming for reasons of performance often forces to make use of special features of the hardware, we do not want to exclude such systems from the discussion here and therefore take the most general approach.

To avoid possibly arising confusion in the following discussion, we want to cite another definition from Wegner [Wegn 90]:

Definition 4 (Active Objects) *Active objects are objects, which may already be active, when receiving a message, so that incoming messages must synchronize with ongoing activities of the object.*

We do not consider this definition to be usefull, because in any system containing any kind of concurrency, it always may happen, that a message is sent to an object, which is currently working. The only way to prevent this, would be to provide a systemwide global clock, which would supply points at the time axis, at which messages could be sent. But we can not imagine, why this should be usefull. Therefore, we always have active objects in a concurrent object-based system in the sense of Wegners's definition.

Having spoken so much about parallelism and objects-based models of concurrency, it is time for the basic question: How is concurrency introduced into an object-based system? One possibility consists of more than one object knowing what to do without receiving a message first. Then there is more than one flow of control in parallel in the system right from the beginning, although all method calls may follow strictly the function call semantics as is the case in the Remote Procedure Call (RPC) model. The combination of several starting objects in combination with RPC leads to a static number of concurrent tasks in a system. This can simplify the administration of resources in the system and therefore the prediction of system performance. How can objects know what they have to do without receiving a message first? It might be, that their "job" is kind of "hard coded" into them, e. g. such objects always have to control a system resource, or the objects are producing periodic signals. Otherwise the object could be an interface to the environment of the system, for example a window retrieving input from the user. It is quite natural, that there is more than one such object, just think of a database system, to which more than one terminal is connected. Naturally the terminals are internally modelled by objects.

The second possibility is, that there exists some means of splitting the control flow similiar to the UNIX *fork()* system call. That means introducing some kind of asynchronism into the system. As the only means of communication between objects (and all entities in an object-based system are objects) is message passing there must be an asynchronous message passing mechanism.

These two possibilities can be considered as being the generic constructs to introduce concurrency into an object-based system. These constructs may either be found alone or both in combination in a system, but one of these generic constructs has to be included in a concurrent object-based system, otherwise the system neccessarily is strictly sequential. Let's define:

Definition 5 (Vivid Objects) *Vivid objects are objects which can send messages without receiving a message first.*

An example for a system consisting of vivid objects would be a distributed process controlling system where each sensor sends its values either periodically or if some limit value is reached, e. g. a certain temperature has been exceeded. ADA tasks are vivid objects for instance.

Definition 6 (Passive Objects) *Passive objects can send messages to other objects or to themselves only in response to a message received first.*

To continue the example above, if the sensors were being polled by some master station periodically, they would be passive objects. The master station would be vivid of course. In general, objects as known from popular object-based languages as C++, Objective C , or Simula) are passive objects. Of course, even in such systems, there must be one initial object, which ist the starting point of program execution and therefore must be vivid.

Definition 7 (Vivid System) *A vivid system is an object-based system where more than one object is vivid (i. e. can send messages, which are not sent in response to a message received first) at a time.*

The definition of a **passive system** is analogous.

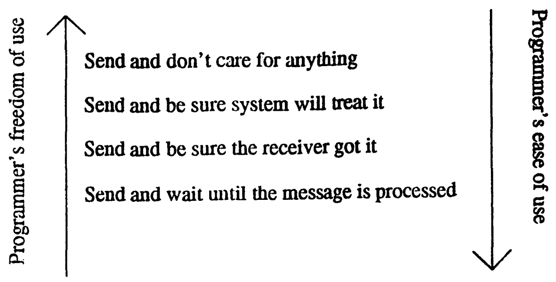
Now it is time to pay attention to the second generic construct, the asynchronous message passing. To clarify our point of view, we first of all propose the following definitions:

Definition 8 (Synchronous Message Passing) *Synchronous message passing means that the sender is blocked until it receives a return value, i.e. message passing has function call semantics.*

This mechanism is very convenient to the developer and reduces synchronization problems significantly. Nevertheless it can lead to unsatisfying solutions: Imagine a vector object, having several point objects as attributes. If another vector has to be added, with synchronous message passing, one coordinate of the vector has to be added sequentially after the other. The more natural solution, adding coordinates in parallel would be impossible, if only this mechanism is supplied.

Definition 9 (Asynchronous Message Passing) *The sender may continue to work without retrieving a return value. The receiver can process the message independently of the sender.*

Now the vector object could send *add*-messages without waiting for a return value before sending the next, thus splitting up the flow of control and exploiting additional parallelism.



Programmer's freedom of use
Send and don't care for anything
Send and be sure system will treat it
Send and be sure the receiver got it
Send and wait until the message is processed
Programmer's ease of use

Figurel: Four communication paradigms

These last two definitions must be handled carefully. First of all consider that, according to this definition, the communication mechanism used in OCCAM would be asynchronous ! Only a full rendezvous as for example provided by the RPC mechanism of ADA would be considered synchronous. Let's clarify our view by dividing communication mechanisms into four classes:

a) The sender sends and does not care at all what happens to the message. This may look very awkward at a first glance, but can be very efficient in certain situations. This point of view is treated exhaustively in [Salt 84].

b) The sender gives his message to the underlying communication mechanism and can be sure, that this underlying mechanism has really received this message for further treatment. It is now in the responsibility of the communication system to buffer the message if necessary and deliver it at the right address reliably. This mechanism is very convenient for the programmer but adds additional complexity to the system and makes it harder to predict the performance of the system on one hand. It also adds the full burden of supplying a buffering mechanism and some kind of flow control to the system. There also exists the danger of deadlocks and system breakdowns due to a runout of internal resources.

c) The sender is blocked until the receiver is ready to retrieve the message. This does not mean that the object which has got the message will process it successfully. Therefore special considerations concerning error handling in a system based on this message passing mechanism have to be kept in mid. A similar mechanism of message passing (where message passing of course has different semantics) is used in OCCAM.

d) The sender is blocked until it receives a return value signalling the successful processing of the message. This means sending a message has function semantics. This last model is known as "rendezvous" in the literature and was first implemented in the RPC mechanism of ADA. This is very convenient for the programmer, as he can use message passing the same way as he is used from function calls in sequential programming. But we are convinced, that most often the restrictions imposed on the flow of control and the lack of being able to distribute a computation over several entities limits the applicability of this mechanism if system performance is the primary goal.

The reader should be aware that only the last model does not allow a split of control flow and therefore will be called synchronous according to our definition.

	passive objects	vivid objects
synchronous message passing	class 1	class 2
asynchronous message passing	class 3	class 4

Tablel: Basic classification

4 Classification

Object-based concurrent systems now can be classified by applying the definitions of vivid systems and synchronous and asynchronous message passing given above. This leads to four possible classes (see table 1).

We try to give a short charcterization of the most basic features of these four classes: Class one is a purely sequential system as implemented in C++ for example and therefore not of interest to us. Class two systems contain a static amount of parallelism, which cannot be extended at run time. In systems of class three a number of object working in parallel is dynamically changing during run time, but all messages being sent in such a system have a common root, as change of flow of control with ongoing time can be described by a tree. The systems in class four combine the features of class two and class three systems and allow least prediction of their runtime behaviour.

Let's apply this classification on our examples now:

Smalltalk-80 Smalltalk-80 does not allow objects to send messages without an impetus from the outside. Therefore it has passive objects. But it has a kind of asynchronous message call, even only in a very restricted form: The class *block* has a method called *fork* which is asynchronous in our definition and allows split of control flow. So Smalltalk-80 turns out to be a class three concurrent object-oriented system.

ConcurrentSmalltalk ConcurrentSmalltalk also falls into class three. Nevertheless its asynchronous method call fit better into the object-oriented context: Concurrency is not limited as being a method of a special class, but possible in connection with every object. Secondly, although for reasons of compatability the questionable concept of Smalltalk-80 is still supported, it is adwised to see objects in ConcurrentSmalltalk as self-contained entities, which is better conforming to the concept of objects.

DistributedConcurrentSmalltalk As DistributedConcurrentSmalltalk is an extension to ConcurrentSmalltalk, it also provides the asynchronous message passing mechanism described above. But as DistributedConcurrentSmalltalk is an interpersonal system, it is a vivid system, as there may be more than one user in the system at a time. Therefore DistributedConcurrentSmalltalk is a class four system.

Actors As stated in [Agha 86] "all computation in an actor system is the result of processing communications". Therefore it is a passive system[‡] Communication is buffered in an actor system, as explained before under communication mechanism b). This means that message passing is asynchronous according to our definition. Consequently actors is a class three system.

[‡]There are models of actor systems seen as vivid systems (e. g. [Kafu 91]), but we will keep to the basic description of actors, where the possibility of more than one "initial actor" is not stated explicitely.

HOOD nets Due to their tree structure with a root node as starting point, HOOD nets are a passive system. Child nodes, which are subnets modelling an object, can be called in parallel. Consequently there exists some kind of asynchronous message passing. Due to this feature HOOD nets are a class three system by our classification.

So far there is still missing the discussion of an important feature of concurrent object-based systems: Can an object handle more than one message in parallel? Can there be more than one method in process inside an object? Although the implementation of concurrency inside an object should be invisible from the outside, as such parallelism can influence the system performance considerably, it has to be seen as a crucial feature, which must be taken into consideration. To exploit this charcteristic for our classification, we give these definitions:

Definition 10 (Multi-Threaded Object) *An object is multi-threaded if there can be more than one method of an object in process at a time.*

Definition 11 (Multi-Threaded System) *A system is a multi-threaded system if it contains at least one multi-threaded object.*

Single-Threaded Systems are defined analogously.

Multi-threaded objects can be a very natural source of parallelism: Let's think of an object, which behaves like an undivideable logical entity to the outside. As an example, there can be an object "employee" in a payroll program with a number of attributs like working hours, salary, number of children and first name of her husband, all of which themselves are objects. It consequently would be a severe violation against object-oriented concepts to change any of these attributes from the outside, but there must be methods of the "employee"-object which will consequently also result in messages addressed to the attribute objects. Now everyone can imagine methods, which are sent to the employee, but only concern one of the attributes. Can there be anything more natural than allowing such methods, only concerning different attributes, to be processed in parallel inside the object? Only allowing single-threaded objects could prevent improving system performance, given the existence of such compound objects in a system.

Nevertheless it is clear that a multi-threaded system must provide some synchronization concept to ensure mutual exclusion between methods which concern the same attributes as semaphores [Dijk 65], guarded commands [Hans 78], monitors [Tane 87] or even some special communication mechanism as in CSP [Hoar 78]. We do not want to discuss the question, which of these mechanisms is most suited for the object-based paradigm, but we do have the feeling, that semaphores are not fitting well into object-based concepts, because they lead to a distribution of control in a non-modular way, contrary to concepts of abstraction and information-hiding.

By taking into account that a system may be multi-threaded or not, we have eight possible classes (see table 2). Nevertheless it should be clear that a multi-threaded class one (i. e. a sequential) system has no sensible interpretation.

Let's classify our examples once again:

Smalltalk-80 In Smalltalk-80 an object cannot protect itself against violation of the consistency of its internal attributes because of receipt of multiple messages at a time. The objects sending messages have to ensure mutual exclusion by using global semaphores. Nevertheless such semaphores also could be used for mutual exclusion of methods inside an object. As a consequence Smalltalk-80 is multi-threaded, though the implementation must be seen as a violation of basic concepts of objects.

	passive objects		vivid objects	
	single threaded	multi threaded	single threaded	multi threaded
synchronous message passing	class 1 S	class 1 M	class 2 S	class 2 M
asynchronous message passing	class 3 S	class 3 M	class 4 S	class 4 M

Table2: Extended classification

ConcurrentSmalltalk For reasons of compatibility to Smalltalk-80 there are so called non-atomic objects in ConcurrentSmalltalk, which are multi-threaded with the problems described above. There are also atomic objects, which only allow one method being executed at a time. Consequently, according to our definition ConcurrentSmalltalk must be considered multi-threaded, although only due to its compatability to Smalltalk-80.

DistributedConcurrentSmalltalk DistributedConcurrentSmalltalk has single activity objects and multiple activity objects, which are called multi-threaded according to our definition. DistributedConcurrentSmalltalk supports an exclusive and a conditional synchronization mechanism. Exclusive synchronization is done by an object by defining exclusive relations between two methods. This relation leads to serialization between several activities inside an object. Conditional synchronization is done by each method having a guard, similiar to ADA. By including all these concepts DistributedConcurrentSmalltalk must be called a multi-threaded system, with an implementation perfectly fitting into the object-based paradigm.

HOOD nets As objects in HOOD nets are re-entrantable they are multi-threaded. HOOD nets have their own definition of active and passive objects. Passive objects, as defined in HOOD nets, have no control over the execution of methods on their data. Therefore they only can be used if all methods are executable in parallel on them without leading to inconsistencies. This means, that they must be "functional objects" with no state associated. Otherwise, if the object must contain a state, an active object has to be used, which can delay the execution of methods, this way enforcing mutual exclusion. The mechanism used for internal synchronization is not specified.

Actors Actors only fetch another message from their message queue if they have finished processing the previous communication, which does not mean that the task has been processed. The message could have been forwarded to another actor, which can still be processing it. Therefore an actor based system follows a strictly single-threaded concurrent object-based paradigm.

Table 3 sums up the application of our classification scheme for a set of object-based concurrent pragraming paradigms, presented at the ECOOP-OOPSLA workshop on object-based concurrent programming during the last years.

5 Summary

After motivating the use of object-based techniques in the development of systems containing concurrency five related, but rather different object-based example systems were introduced. Then the basic definitions of *object-based*, *class-based*, and *object-oriented* systems were given. The ways of introducing concurrency into object-based systems were

Name of the System	*Classification*	
ABCL/1	class 3	S
Actors	class 3	M
ConcurrentSmalltalk	class 3	M
CORAL	class 3	M
DistributedConcurrentSmalltalk	class 4	M
HERAKLIT	class 2	M
HOOD nets	class 3	M
Matroshka	class 3	M
Orient84/K	class 3	S
POOL	class 2	S
Smalltalk-80	class 3	M

Table3: Application of the extended classification

discussed, and a classification scheme, based on the discussion, was introduced and applied on example systems. Lastly the question of concurrency inside objects was raised and added to the classification scheme as being discovered a basic feature of object-based concurrent systems. Our examples were classified again and their mechanisms to control concurrency inside objects (if any) were examined. At the end a table containing the application of our proposal for classification on a larger set of systems, which were not discussed here due to lack of space, was given.

6 Further Research

First of all the performance as being one of the most important reasons for developing concurrent systems of the different classes has to be examined. Are systems falling into one class are performing significantly better than the others? Is the additional complexity needed by multi-threaded systems paid back in an adequate gain of performance? Another point of interest are error handling mechanisms applied in those classes of our classification containing asynchronous message passing. Of course, one also has to ask, if the classification presented above is valid for all possible systems, can it be extended, refined? And, to come to an end, the basic question, if there is one class superior to all others, has to be investigated.

To answer these questions must be the goal of further work.

References

[Agha 86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Agha 89] G. Agha. "Foundational Issues in Concurrent Programming". *SIGPLAN NOTICES*, Vol. 24, No. 4, pp. 66 – 65, April 1989.

[Atta 87] G. Attardi. "Concurrent Strategy Execution in Omega". In: *Object-Oriented Concurrent Programming*, MIT Press, 1987.

[Babb 87] R. G. Babb and D. C. DiNucci. "Design and Implementation of Parallel Programs with Large-Grain Dataflow". In: L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds., *The Characteristics of Parallel Algorithms*, pp. 335 – 349, MIT Press, 1987.

[Bach 73] C. W. Bachmann. "The Programmer as Navigator". *ACM*, Vol. 16, No. 11, 1973.

[Coad 90] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, 1990.

[Cox 87] B. J. Cox. *Object Oriented Programming.* Addison-Wesley, April 1987.

[Dijk 65] E. W. Dijkstra. "Co-operating Sequential Processes". In: F. Gennys, Ed., *Programming Languages*, London Academic Press, 1965.

[DiSa 91] M. DiSanto and G. Iannello. "Implementing Actor-Based Primitives on Distributed Memory". *OOPS Messenger*, Vol. 2, No. 2, pp. 45 – 49, April 1991.

[Gane 79] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques.* Prentice-Hall, 1979.

[Genr 91] H. J. Genrich. "Predicate-Transition Nets". In: K. Jensen and G. Rozenberg, Eds., *High-level Petri Nets*, Chap. Section A, pp. 3 – 43, Springer-Verlag, 1991.

[Giov 90] R. D. Giovanni. "Petri Nets and Software Engineering: HOOD Nets". In: *11th International Conference on Application and Theory of Petri Nets*, pp. 123 – 138, June 1990.

[Gold 83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Hans 78] P. B. Hansen. "Distributed Processes, A Concurrent Programming Concept". *CACM*, Vol. 11, No. 21, pp. 934 – 941, 1978.

[Hoar 78] C. A. R. Hoare. "Communicating Sequential Processes". *CACM*, August 1978.

[HOOD 89a] *HOOD Reference Manual.* HOOD Working Group, European Space Agency, wme/89-173/jb, issue 3.0 Ed., September 1989.

[HOOD 89b] *HOOD User Manual.* HOOD Working Group, European Space Agency, wme/89-353/jb, issue 3.0 Ed., December 1989.

[Jens 91] K. Jensen. "Coloured Petri Nets: A High Level Language for System Design". In: K. Jensen and G. Rozenberg, Eds., *High-level Petri Nets*, Chap. Section A, pp. 44 – 117, Springer-Verlag, 1991.

[Kafu 91] D. Kafura, D. Washabaugh, and J. Nelson. "Progress in the Garbage Collection of Active Objects". *OOPS Messenger*, Vol. 2, No. 2, pp. 59 – 63, April 1991.

[Levi 87] S. P. Levitan. "Measuring Communications Structures in Parallel Architectures and Algorithms". In: L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds., *The Characteristics of Parallel Algorithms*, pp. 101 – 138, MIT Press, 1987.

[Lieb 87] H. Lieberman. "Concurrent Object-Oriented Programming in Act 1". In: *Object-Oriented Concurrent Programming*, MIT Press, 1987.

[Loya 91] J. P. Loyall, S. M. Kaplan, and S. K. Goering. "Specification and Implementation of Actors with Graph Rewriting". *OOPS Messenger*, Vol. 2, No. 2, pp. 73 – 77, April 1991.

[Naka 89] T. Nakajima, Y. Yokote, M. Tokoro, S. Ochiai, and T. Nagamatsu. "Distributed Concurrent Smalltalk, A Language and System for the Interpersonal Environment". *SIGPLAN NOTICES*, Vol. 24, No. 4, pp. 66 – 65, April 1989.

[Nels 87] P. A. Nelson and L. Snyder. "Programming Paradigms for Nonshared Memory Parallel Computers". In: L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds., *The Characteristics of Parallel Algorithms*, pp. 3 – 20, MIT Press, 1987.

[Salt 84] Saltzer. "End to End Arguments in System Design". *ACM Transactions on Comp. Systems*, Vol. 2, No. 4, pp. 277 – 288, November 1984.

[Tane 87] A. S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, 1987.

[Wegn 90] P. Wegner. "Concepts and Paradigms of Object-Oriented Programming". *OOPS Messenger*, Vol. 1, No. 1, pp. 7 –87, August 1990.

[Yoko 87] Y. Yokote and M. Tokoro. "Concurrent Programming in Concurrent Smalltalk". In: *Object-Oriented Concurrent Programming*, MIT Press, 1987.

Finite State Machines and Object Orientation

R. Lewandowski
M. Mulazzani
Alcatel Austria–ELIN Research Centre
Ruthnergasse 1–7
A–1210 Vienna, Austria

Abstract

Finite State Machines (FSM) are an established approach for modeling the behavior in reactive systems. At the same time object oriented techniques are spreading on the market. This report investigates Finite State Machines and their similarities to and extensions with object oriented concepts.

First, basic similarities of the traditional Finite State Machines with respect to object orientation are explored, covering encapsulation, typing, system structuring and instantiation. Then, some object oriented extensions of FSMs (inheritance, virtual transitions, ...) are shown with the example of OSDL (currently under standardization by CCITT, an OO extension of SDL from CCITT). Finally, state charts from Harel are investigated. They provide extensions to FSMs which are not object oriented. But there exists an interesting mapping of their extensions to classes, inheritance and composition, providing a new view on FSMs, states and transitions.

1. Introduction

For several years now Finite State Machines (FSMs) and Extended Finite State Machines (EFSMs) are used in the area of real time systems as a standard technique. They provide the means to effectively describe system behavior and they are well suited to model the change of behavior in systems. One big application area are telecommunication systems. CCITT (International Telegraph and Telephone Consultative Committee) recommends the use of SDL [CCIT89], [Saca89] (based on the FSM concept) for the software development of telecommunication services.

On the other hand, object oriented technology has strongly emerged on the market. The concepts of encapsulation, information hiding, abstract data types and inheritance provide new means for system development. Availability of object oriented languages and programming environments, as well as the emergence of object oriented methods allow for the adoption of the object oriented technology into an industrial context.

So coming from the application area of telecommunication systems, the question arises, how object oriented concepts will fit or will be integrated into the development process. It is the goal of this report to discuss the concept of finite state machines and their links, similarities and extensions with object oriented concepts.

Section 2 starts with the basics about FSMs. It gives a short introduction to FSMs and their representation forms, which is then evolved into a discussion on structuring aspects with FSMs, showing a first set of similarities to object orientation. A different approach is presented in section 3. OSDL is an object oriented extension of SDL under standardization from CCITT, the section discusses how object oriented principles are integrated into the FSM approach. As the third main approach, the state charts from Harel [Hare87] are presented in section 4. They are a powerful, not object–oriented extension of FSMs. But it turns out that there exists an interesting object oriented analogy of the extensions which is presented in section 5. Finally, the summary collects the results and gives an outlook for further topics and open questions.

2. Finite State Machine (FSM)

FSM Definitions

Sequential Machines (Finite State Machines): A FSM is a machine with memory containing the state. Operations are determined by input events and the current state.

Mathematically a FSM is a 5–tuple, (I, S, O, NSF, OF), where

- I is a finite set of input symbols.
- S is a set of mutually exclusive states (static waiting).
- O is a finite set of output symbols.
- NSF is a mapping of I and S onto S called the next state function (this mapping is often called transition).
- OF is a mapping of I and S onto O called the output function.

FSMs are characterized by discrete–valued inputs, outputs and internal elements [Hatl87], [Hopc79].

With such a FSM it is possible to express behavioral aspects of a system. The states are used to define conditions in which the system reacts to specific events. Reaction here means the transition to another specific condition. Only very simple systems can be sufficiently described by the usage of FSMs. This is because the number of different conditions in which a system can be, is usually too large. The number of states in a software system equals the number of all possible combinations of values of all data. This phenomena is called ”the explosion of states”.

An example for the application of an FSM is a traffic light. There exist four different states: green, yellow, red and red–yellow . Only one input signal named *change* is defined for this FSM. Depending on the current state and the input signal the next state is determined. If the actual state is green, the input signal *change* will cause a transition to yellow a.s.o..

Extended FSM: In EFSMs not all conditions of a system are modelled with states. States are only used to model the essential conditions. States are abstractions representing *groups* of conditions of a system. For these states also transitions can be defined as for normal FSMs, but in an extended form.

In an EFSM the output signal and the next state is determined not only from the previous state and the input symbol (here called signal) but also from other data. Data of an EFSM can be classified into four categories:

- the state variable which holds the actual state

- the local variables which hold additional information to the state
- the temporary variables which are used temporally during state transition e.g. a counter variable which needs no remembrance
- the input signal variables and the output signal variables.

In EFSMs the behavior is defined with states as abstractions of conditions. Transitions depend on actual state, input signal and values of additional data.

Representation of FSMs

Different representations are used in order to define FSMs. Common notations for FSMs are *state transition diagram, state transition matrix and SDL diagrams.*

State Transition Diagrams: State transition diagrams are directed graphs. The different states are represented as nodes. The transitions (caused by incoming signals) are represented with directed edges between the states. The incoming and outgoing signals are shown as annotation of the edges. The state transition diagram shows the sequence of signals and conditions within a system (see Figure 1 where A, B, C are states, r, s, t are incoming signals and u, v, w, x, y, z are outgoing signals).

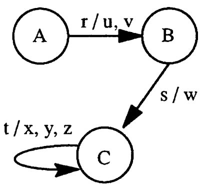


Figure 1: Example of a state transition diagram with incoming and outgoing signals

State Transition Matrix: In a state transition matrix the states are represented by rows. The incoming signals are shown as columns. Whenever an incoming signal is accepted in a state the according transition and output signals are written into the specified field of the matrix. This matrix tends to have a lot of empty fields due to the number of not allowed signals in a state. Figure 2 gives an example.

	onhook	offhook	ring
onhook state		offhook state/ dial_tone_on	ringing state/ ring_line
offhook state	onhook state/ dial_tone_off		
conversation state	onhook_state/ disconnect_line		
ringing state	onhook state/ disconnect_line	conversation state/ connect_line	

Figure 2: Example of a state transition matrix

Software Description Language (SDL): While the previous notations are only able to define FSMs, the *software description language* [Saca89] is able to define EFSMs. SDL has a graphical and a textual representation. Graphical SDL is a kind of *flow chart* extended by special symbols like *state symbol*, *incoming signal* and *outgoing signal*. This allows to express both, the FSM aspect and the control flow of transitions.

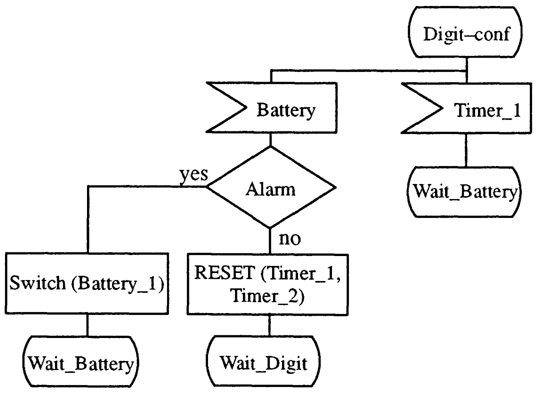


Figure 3: SDL example

An example of this notation is shown in Figure 3. Two transitions are defined for the state *Digit-conf*. Receiving the signal *Timer_1* will change the state to *Wait_Battery*. In case of receiving signal *Battery* the condition of *Alarm* will be tested. If the condition is true the procedure *Switch(Battery_1)* is called. The transition ends by changing the state to *Wait_Battery*. If the condition *Alarm* is false the procedure *RESET(Timer_1, Timer_2)* is called and the state is changed to *Wait_Digit*. SDL offers additional constructs which allow to model typical situations. A special symbol can be used for "all other signals", i.e. defining a transition for the unexpected signals in a state. It is possible to store signals for later use. A transition can be associated to an incoming signal which is valid in any state, and many other possibilities.

Structuring of FSMs

A large and complex FSM is hard to understand, even for the designer himself. A state transition graph showing all states and transitions of a FSM possibly does not even fit on a single page. But it is possible to show views of the FSM thus helping a reader to understand it. Different groups of signals are shown on different state transition diagrams. Figure 4 gives an example which shows two separate views of a FSM for two different logical parts of behavior. The addition of both diagrams results in a more complex diagram which would be more difficult to read and understand.

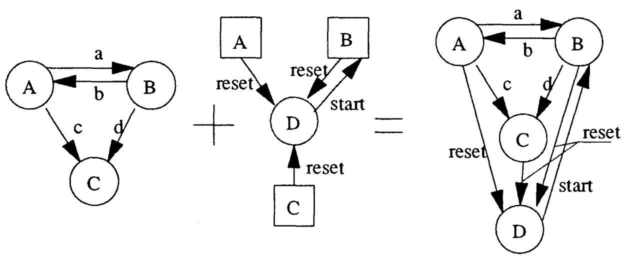


Figure 4: Managing complexity with different views

To indicate copies of states in the different diagrams they are shown by a rectangle instead of a circle.

Structuring of Systems with several FSMs

Systems and especially large systems have to be structured in order to master their complexity. FSMs, when combined with the process model (a FSM instance is a thread of control) are well suited to express the behavior of a system. The system is divided in several parts (processes) each of them being modelled with a FSM. So the system is seen as being built out of several FSMs, each of them having data (current state) and input signals.

With such a view of cooperating FSMs, several issues become important which are discussed hereafter: How to express interaction between FSMs ? How are FSMs used (instances of FSMs) ?

Message Sequence Charts (Scenarios): Message sequence charts show the interaction between different FSMs, they show the signal flow and its timing. Each such scenario shows one example of an interaction, i.e. one specific situation of interaction. In the notation the FSMs are drawn as vertical bars, the vertical dimension represents the passing of time. The signals are shown as directed lines between the FSMs. In this notation it is easy to express the duration of signal exchange, sequence of signals and concurrency.

Figure 5 shows an example of a message sequence chart. One specific flow of signals is shown for four FSMs. The directed arrows do *not* show the transitions but the *flow* of signals from one FSM to another.

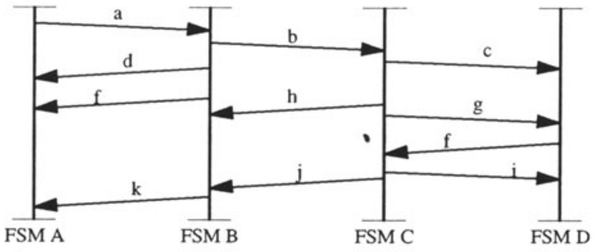


Figure 5: Example of a scenario

Instances of FSMs: Each FSM has to store the actual state and its local variables. An additional concept is the instantiation mechanism for FSMs (e.g. process instance in CHILL [CCIT86]). Figure 6 shows the relation between several instances of a FSM and the FSM itself. The FSM shows the common behavior of all instances. The FSM is used as a *type*. The instances of the FSM hold their *own* local variables and the *actual state*. In languages without special language constructs for FSM the storage allocation for each instance has to be implemented explicitly (or has to be generated automatically).

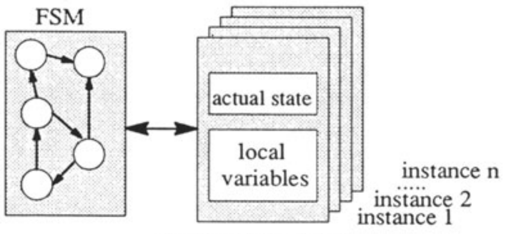


Figure 6: FSM and Instances

Similarities to Object Orientation

Although the discussion up to now focused on properties and usage of FSMs, some similarities of FSMs with object oriented concepts can already be seen:

The notion of FSMs having data (including state) and input signals is quite analogous to objects, FSMs encapsulate data and allow a client only to operate on these data by means of signals.

The system being seen as consisting of interacting FSMs is another similarity to OO. Communication between several FSMs is done by means of sending and receiving signals. It is worth noting, that the scenarios used for FSMs are quite equivalent to the object interaction diagrams as recently introduced into the Booch method [Booc91b].

Another similarity exists between FSMs and OO. A FSM can be seen as type. Instances of a FSM exist, each having its own data. This allows to create several copies of the FSM in a system. Each of these instances has the same properties as defined for the FSM. FSMs map to classes, and the instantiation of FSMs directly corresponds to object oriented approaches.

Finally, the question of granularity and complexity (combination and splitting of FSM) is valid also in object oriented systems and vice versa.

However, it should be noted here, that it is not our argumentation that FSM and object orientation is the same (there are certainly differences). We just want to point out that there are certain similarities.

The following sections now explore in more detail the links of FSMs with the object oriented approach, covering object oriented extensions of FSMs as well as object oriented views of FSMs.

3. Concepts of OSDL

OSDL is an extension of SDL with concepts of object oriented techniques [Moll87]. It was intended to keep the changes within the semantics of SDL as small as possible. SDL supports encapsulation by means of the process concept. A process encapsulates data and the associated operations.

OSDL distinguishes between types and instances. Process instances are derived from process types. Inheritance is used to support specialization of process types.

Single inheritance of a process type (FSM) allows to add new transitions with new input signals to the inherited ones. A state transition matrix is well suited to visualize this. Figure 7 shows a state transition matrix of an SDL specification. This process type has two states: *Even* and *Odd*. The following input signals are used in transitions: *Probe, Result, Endgame and Bump*. Figure 8 now shows *Special Game*, a specialization of *Game*. *Special Game* inherits from *Game* which could be seen in the new state transition matrix: several transitions (with new input signals) and states are added (here *Evil* and *WereEvil* are added signals and *Chance* is an additional State). Defined transitions from the super–type cannot be redefined (overwritten) within the definition of the sub–type.

	Probe	Result	Endgame	Bump
Even				
Odd				

Figure 7: State transition matrix of *Game*

	Probe	Result	Endgame	Bump	Evil	WereEvil
Even	These transitions are inherited from *Game*					
Odd						
Chance						

Figure 8: State transition matrix of *Special Game*

Another object oriented concept introduced for process types in OSDL is the *virtual procedure*. Virtual procedures of a super–type can be defined concretely in sub–types (derived process). This allows to define a transition in an abstract process type only partly and leave some parts open (virtual procedures) to be defined within specializations of the process type. Figure 9 shows for the already mentioned example *Game* the process definition of *General Game*. Two procedures *ProbeWhenEven* and *ProbeWhenOdd* are defined virtually (grey boxes). Figure 10 shows the definition of process *Game*. The process *Game* inherits all transitions defined by *General Game*. The virtual procedures *PropeWhenEven* and *ProbeWhenOdd* are defined in detail with the SDL notation.

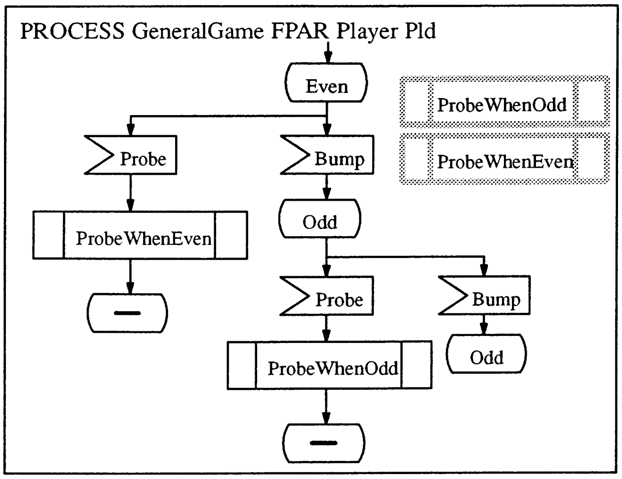


Figure 9: Virtual procedures

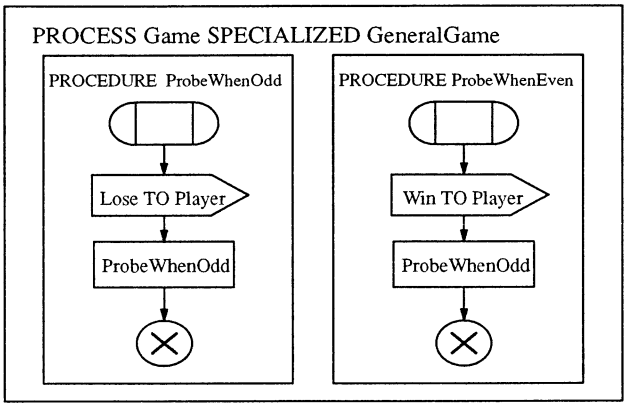


Figure 10: Definition of virtual procedures

While *virtual procedures* are used to leave specific parts of transitions undefined, *virtual transitions* allow to define the complete transition in detail in the derived process. For a *virtual transition* only the *state* and the *input signal* are defined. The super–type allows to define a default transition which can be overwritten within derived types. This concept stresses a subtype either to use the default transition or to redefine a transition to a more specialized one. Figure 11 shows two process types inheriting from a super–type. The super–type defines that there has to exist a transition for *State A* and signal *S*. The super–type also defines a default transition. The derived process drawn on the left hand side defines the *virtual input* (= virtual transition) to a concrete transition. The other derived process defines the virtual input to a *save* (storing of the signal for later usage).

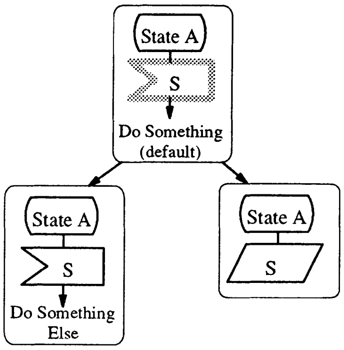


Figure 11: *Virtual Input* specialized to an *Input* or *Save*

The following table 1 gives an overview how the concepts introduced by OSDL can be mapped to object oriented concepts.

Systems modelled with Finite State Machines	Systems modelled with Objects
Finite State Machine	Class
State	Data (members)
Process	Instance
Input signal	Operation
Transition	Operation implementation
Output signal	Called operations in operation implementation
Virtual procedure	Virtual operation
Virtual transition	Virtual operation

Table 1: Finite State Machine and Object Oriented Concepts

4. State Charts, Non Object Oriented Extension of FSM

While *OSDL* allows for abstraction of transitions, *State Charts* [Hare87] provides a different idea of abstraction of states. State charts are a visual formalism for describing states and transitions in a modular fashion, enabling clustering, orthogonality (i.e. concurrency) and refinement.

State charts have a similar semantic like state transition diagrams but with some extensions. Simple state transition diagrams are expressed with the same notation as used for state transition diagrams. The small difference is that states are drawn with rounded boxes instead of circles.

Refinement of States: One of the extensions to state transition diagrams is the refinement of states. A superstate can be refined into substates with the semantics of XOR. The superstate can only be exactly one of its substates at a time. As usual for FSMs, transitions are attached to these substates. Transitions can also be attached to superstates with the following semantic: A transition defined for a superstate means that this transition is defined for all of its substates.

An example is shown in Figure 12 where picture I. shows a superstate *D* which is refined into substate *A* and *C*. If the system is in the superstate *D* it is either in state *A* or *C* (XOR semantics). Signal *b* is valid for both of the substates *A* and *C*. Therefore the transition is attached to the superstate D. Signals *a* and *c* can be received in state *B* and cause transitions to *A* respective *C*. Picture II. gives an equivalent FSM without superstates.

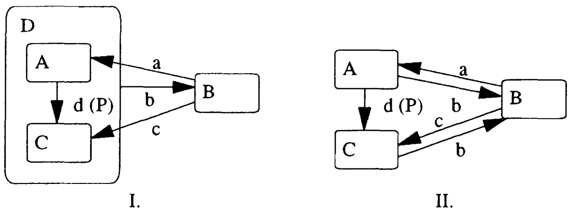


Figure 12: Abstraction in State Charts

Figure 13/I. shows the previous state chart at a higher level of abstraction (no inner details). Picture II. then shows the inner details of state *D*, the substates and their transitions. At the same time it shows another extension, the annotation of signals by conditions e.g. *d(P)* were *d* is the input signal and *P* is a condition. The transition is only made if the condition is true.

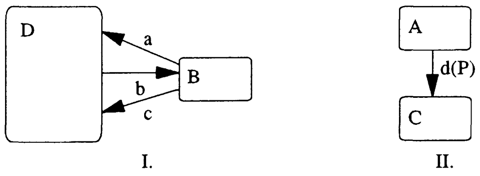


Figure 13: Abstraction in State Charts

The superstate/substate concept offers two ways of usage: refinement (as it was introduced in this section, i.e. top–down approach) or clustering and abstraction (bottom–up approach). What to choose depends on the situation.

Orthogonality (Concurrency) of States: State charts also allow AND decomposition, capturing the property that, being in a state, the system must be in *all* of its AND components. The orthogonal product of the components is called the *AND state*. For the orthogonality it is required that the transitions of one state machine are independent of the actual state of the other state machine and vice versa.

Figure 14 shows an example of an AND state. The dashed line in picture I. between state *A* and *D* shows the AND composition of *A* and *D*. *Y* is called the orthogonal product of *A* and *D* and is itself a state. In picture I. the arrow with the black spot on its shaft defines the initial state. *A* and *D* are not completely orthogonal, in A there exists a transition from *C* to *B* which is annotated by *b (in G)* indicating that the transition takes only place if the current *substate* of *D* is *G*. Picture II. shows an AND–free equivalent to picture I.

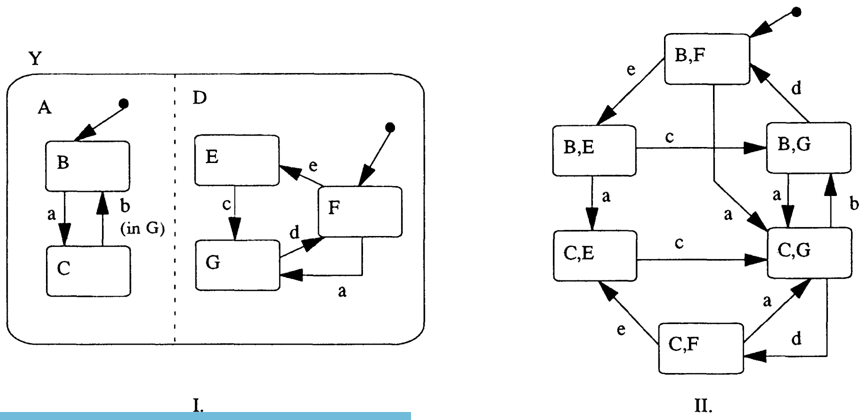


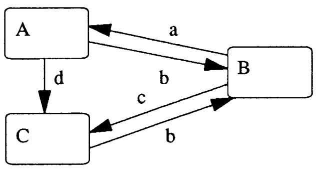
Figure 14: Orthogonality in State Charts

There are several further extensions in State Charts, including *history, condition and selection entrances, delays and timeouts*. However, they are beyond the scope of this paper, for details see [Hare87].

State refinement (XOR) and state orthogonality (AND) are quite abstract means. They provide new concepts for structuring FSMs and are intended to increase the power of FSMs for behavior modelling. The next section investigates the similarities of these extensions with object orientation.

5. Object Oriented Analogy to State Charts

The object oriented view of the state chart extensions follows the basic idea to map states to classes [Hüne91], [Vans91]. Input signals accepted in the states are mapped to the operations of the class. Figure 15 shows an example of a FSM and the corresponding classes. The names of the classes are taken from the states, the signals from the FSM are mapped to operations. The transitions (the arrows from one state to another) are indicated as comments (they would correspond to the implementation of the operation).



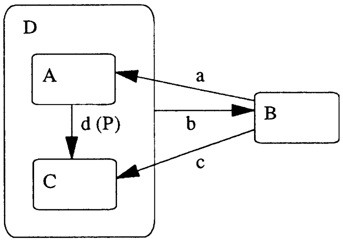
```
Class State_A
defined operations:
    d   // performs desired action
        // and transition to State C
    b   // performs desired action
        // and transition to State B
```

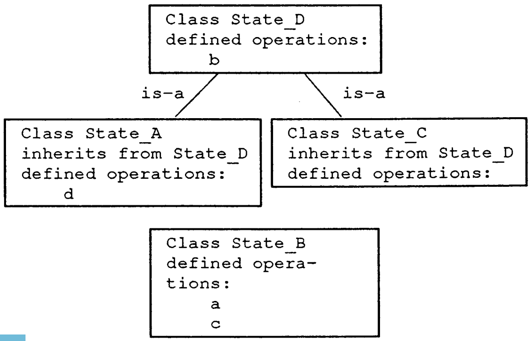
```
Class State_B
defined operations:
    a   // performs desired action
        // and transition to State A
    c   // performs desired action
        // and transition to State C
```

```
Class State_C
defined operations:
    b   // performs desired action
        // and transition to State B
```

Figure 15: FSM mapped to Classes

With this object oriented view, the analogy of state refinement (superstate, XOR) is easy to express. State charts use the superstate to show common properties of states. In the object oriented paradigm it is possible to show common properties of classes with inheritance. If we look at class *State_A* and class *State_C* we will find an operation *b* which is defined similar for both classes. In the object oriented paradigm this could be modelled with inheritance. A new class *State_D* is introduced with operation *b*. Both classes *State_A* and *State_B* inherit from class *State_D*. Classes *State_A* and *State_D* which are derived from *State_D*, have to define only the remaining operations.





```
Class State_D
defined operations:
    b
```

```
Class State_A
inherits from State_D
defined operations:
    d
```

```
Class State_C
inherits from State_D
defined operations:
```

```
Class State_B
defined opera-
tions:
    a
    c
```

Figure 16: Abstract States mapped to Classes

Figure 16 shows Harels notation on the left hand side and the object oriented equivalent on the other side. The object oriented analogy of superstates is the inheritance between classes.

In contrast to that, the AND decomposition in state charts captures the property that, being in an AND state, the system must be in *all* of its AND components. The AND state can be seen as a composition of all its states. In the object oriented paradigm there exists a composition hierarchy (whole–part), each class can be seen as the composition of other classes. These classes are often modelled as data (members) of the *composite* class.

Figure 17 shows an AND state *Y* defined by Harels formalism. This AND state has two components, state *A* and state *D*. The AND state consists always of both components *A* and *D*. There is no point in time where state *Y* is either only in state *A* or state *D*. This semantics is similar to the semantics of whole–part relation in the object oriented paradigm shown on the right hand side. Composite class *state_Y* is built out of class *state_A* and class *state_D*, composition–relation is modelled with data members of a class *state_Y*.

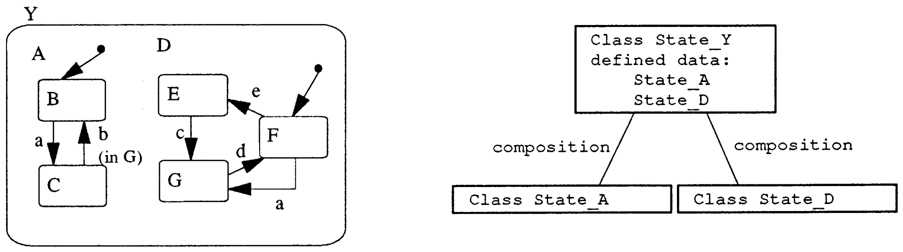


Figure 17: AND States mapped to Classes

The object oriented analogy to Harels AND states is the whole–part relation.

It is also interesting to see, that for the combination of AND and XOR states for state charts maps to the combination of inheritance and whole–part: Class *State_A* (in Harels state chart an XOR of B and C) and *State_D* (XOR of E,F, and G) from figure 19 can be refined in the object oriented view by inheritance (is–a relation) analogous to Figure 16.

Harel extended the FSM and gave a complete new view of states and FSMs. The similarities between Harels states and the object oriented paradigm allows now to map ideas from one model to the other. This allows to find new aspects in state charts by looking at the object oriented paradigm and vice versa. How do virtual operations map to states? What does multiple inheritance or access control for members of classes mean for states. There are a lot of unanswered questions which are under further investigation.

The following table 2 summarizes the mapping of Harels formalism to object oriented concepts.

Systems modelled with State Charts	Systems modelled with Objects
State	Class
XOR state	Base–class (inheritance)
AND state	Composite class (data members)
Signal	Operation
Transition	Operation implementation

Table 2: Harels Formalism and Object Oriented Concepts

6. Summary

The question "How will Finite State Machines integrate object oriented principles?" does not have a single answer. This report addresses three approaches: the link of traditional FSMs with object orientation; specific object oriented extensions to FSMs in OSDL; and an object oriented view of the state structuring in Harel's state charts.

Traditional FSMs and EFSMs show several characteristics, which are similar to object oriented concepts. The property of FSMs to allow access to its data only by means of signals corresponds to encapsulation. With FSMs, systems are modeled as sets of interacting and cooperating FSMs, which fits to object orientation. Finally, FSMs are types (behavior templates) which are instantiated at run–time. However, this should not be over–interpreted, these are just similarities in characteristics. It is not our argumentation that FSMs are object–oriented.

One actual enhancement of FSMs with object oriented constructs is being done for CCITT with the definition of OSDL (extension of SDL from CCITT). This mainly includes inheritance, but also virtual procedures, virtual transitions and other aspects. OSDL shows how object oriented constructs can be used within the formalism of FSMs.

Harel's state charts extend the FSM approach with state structuring, including state refinement (XOR state) and orthogonality of states (AND state). While these extensions are not directly connected to object orientation, it turns out that from an object oriented point of view, these extensions nicely map to inheritance and composition structure. This interpretation of state charts raises interesting questions concerning the mapping of ideas from the FSM model to the object oriented model and vice versa.

These three approaches only show some aspects of the issues involved when considering the link between FSMs and object orientation. Areas for further investigations include:

- Conflicting and contradicting issues between FSMs and OO
- OO extensions of FSMs (e.g. OSDL) and their link to OO languages
- link of FSMs with object oriented analysis and design methods [Booc91a], [Coad91], [Rumb91] as well as the extensions of real time methods (based on FSMs) with object oriented concepts.

7. References

[Booc91a] G. Booch, "*Object Oriented Design with Applications*", Benjamin/Cummings Publishing Company, 1991.

[Booc91b] G. Booch, M. Goldberg, "*Object Oriented Design*", Rational Course Handout, 30. Oct. 1991.

[CCIT86] —, "*The CCITT High Level Language CHILL User's Manual*", International Telegraph and Telephone Consultative Committee (CCITT), Geneva, 1986.

[CCIT89] —, "*CCITT Blue Book, Recommendations Z.100: Functional Specification and Description Language (SDL)*", International Telegraph and Telephone Consultative Committee (CCITT), Geneva, 1989.

[Coad91] P. Coad, E. Yourdon, "*Object Oriented Analysis*", Yourdon Press Computing Series, 1991.

[Hatl87] D.H. Hatley, I.A. Pirbhai, "*Strategies for Real–Time System Specification*", Dorset House Publishing, 1987.

[Hare87] D. Harel, "*Statecharts: A visual formalism for complex systems*", Science of Computer Programming, Vol. 8, pp. 231–247.

[Hopc79] J.E. Hopcroft, J.D. Ullman, "*Introduction to Automata Theory, Languages and Computation*", Addison–Welsey Publishing Company, 1979.

[Hüne91] I. Hüneke, "*Finite State Machines: a model of behavior for C++*", The C++ Report, Vol. 3/1, Jan. 1991

[Meye88] B. Meyer, "*Object Oriented Software Construction*", Prentice Hall, 1988.

[Moll87] B. Moller–Pedersen, D. Belsnes, "*Rationale and Tutorial on OSDL: An Object–Oriented Extension of SDL*", Computer Networks and ISDN Systems, Vol. 13/2, pp. 97–117, 1987.

[Rumb91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "*Object–Oriented Modeling and Design*", Prentice Hall, 1991.

[Saca89] R. Sacaro, J.R.W. Smith, *Telecommunications System Engineering using SDL*", Elsevier Science Publishers B.V., 1989.

[Stro88] B. Stroustrup, "*What is Object–Oriented Programming?*", IEEE Software, Vol. 5/3, pp. 10–20, May 1988.

[Vans91] J. Vanslembrouck, "*Relating Extended Finite State Machines with Object–Orientation*", Software Engineering Report nr. 914009, Alcatel Bell Telephone, Jan 1991.

Enhancing Reusability and Simplifying the OO Development with the Use of Events and Object Environment

Krista Rizman, Ivan Rozman

University of Maribor, Faculty of Technical Sciences
P. O. Box 224, 62000 Maribor, Slovenia, Europe
E-mail:rizman@uni-mb.ac.mail.yu

Abstract. Object-oriented(OO) software development enhances reusability. But reuse and object composition are not straightforward. Only compatible components, which conform to the same client-server protocol, can be composed. In this paper we propose an event-driven approach to OO software development which enhances reusability by increasing the openness of objects and provides a simple composition principle. It bases on concepts of events and object environment. Object environment serves as a mediator among independent objects. It consists of agents that monitor and respond to object notifications of events that occur through the life-cycle of each object. Events enhance object openness and so reusability and allow uncoupled programming style. Uncoupled programming style together with a simple composition principle provided by the use of the object environment allow easy production of powerful building blocks and simple construction of the complex software from powerful building blocks.

1 Introduction

The last ten years, there is an explosion of complexity in construction of software development. First, there is the complexity of behavior of real world system, part of which has to be modeled and verified in the early stages of development, i. e. in the requirement specification step.

The complexity of results of analysis and also of design is the second form of complexity. Mountains of documents and diagrams contained by traditional specifications are very hard to be exactly verified by end users. But only end users can exactly verify specifications, because only they exactly know what for a system they want and how it must work. Operational specifications called prototypes reduce the complexity of specifications and increase understandability of them. Executable specifications make the verification process to the end user easier. Problems caused by complexity of system structure and behavior can be avoided with iterative - spiral or fountain development life-cycle [1]. We found prototyping to be the best technique for performing the iterative development [2]. The prototype grows with each iteration, and refined each time, eventually becomes the end product.

2 Object technology

Prototyping approach is particularly appropriate when object technology is used which enhances reusability.

Libraries that have been around for a long time, are more difficult for using than libraries of classes. At using procedures an assumption must be made about the context in which they are to be invoked. For instance when using a GKS procedure for drawing a circle we must know in which viewport the circle will be and viewport has to be opened before drawing so as the workstation and GKS.

Since objects "are" self-contained behavioral units, it is easier to create units (objects) that can be taken out of the context and reused in another context. Generalization of features is possible via inheritance. We have written above "are", because *objects communicating by methods are interconnected too much, in our opinion. In order to make objects self-contained, it is required that interaction abilities of an object are described independently of formalisms to ensure these interactions. In other words, compositions have to be separated from components to increase reusability of components and to enhance understandability of applications.*

During the development of an object-oriented system, developers are faced with problems appearing at reuse of components, that are *interconnected to much* and *with the lack of a simple composition principle.*

There is a move in object-oriented software development methodologies from the data-driven object-oriented design to the responsibility-driven design [3, 4] and interaction-oriented development [5] with the goal to increase encapsulation. But, both new approaches to software design, the responsibility-driven and interaction-oriented, base on the description of object responsibilities and thus client-server relationships although the client-server protocol *limits the reuse* of object-oriented software. Only components which are compatible - which obey the same client-server protocol may be composed and may collaborate and perform some system functionality.

Software reusability is of great importance for the efficient development of large systems. Object-oriented approach to software development enhances software reusability because it provides a simple mechanisms for incremental modifications and compositions of software. These mechanisms are inheritance, dynamic binding, and message passing [5]. The composition of reusable components is made difficult by incompatibility of two or more existing components which perform the required functionalities but they do not satisfy the same client-server protocol.

As an example, consider the designing of an information system about people, where the required statistical data about people are graphically presented by dial or histogram. Consider that we have classes People, Dial and Histogram from other applications in a class library. They can be reused for this system. They are written in Smalltalk in Listing 1.

These three classes do not satisfy the same client-server protocol and cannot be reused without modifications for our application. Instead of the message show: aValue, the paint: aValue message should be sent to objects of class Dial and draw: aValue to objects of class Histogram. These are interface incompatibilities. Then message getValue is unnecessary. This is an incompatibility, where an object does not perform required actions in response to a received message. It is called causal incompatibility. In strongly typed languages (such as C++, Eiffel) also type incompatibilities can occur.

The design of reusable classes is also made difficult because in message passing systems a process cannot disseminate new results without knowing precisely where to send them. Objects have to know about its surrounding objects.

This and all forms of incompatibilities can be avoided by introduction of an object environment and by designing objects that instead of invoking the behavior of other objects with sending messages to them, inform the environment about interesting changes of values of its state variables [6]. Changes of object state variables are called events. Environment monitors and responds to notifications of events by initiating actions.

The introduction of an object environment for description of compositions of objects allows the separation of components from compositions which has already been mentioned as a request for increasing object reusability and understandability of applications.

```
class              People
superclass         Set
instance variables   views
instance methods
init ...
setValue: aValue
  views do:[:view|view show:aValue]
attachView: aView
 ...views  add:...
detachView: aView
 ...
older: years
    |anumber|
     anumber := 0.
    self do:[aPerson|
      (aPerson age > years)
      ifTrue:[anumber := anumber + 1]]
    self setValue:anumber.
younger: years
    ...
    self setValue:anumber.
old: years
    ...
    self setValue:anumber.
```

```
class               Person
superclass          Object
instance variables surname age...
instance method     ...
age
  ^age      ...
```

```
class                 Dial
superclass            Object
instance variables    boundingBox subject
instance methods ...
boundingBox: aValue   ...boundingBox := aValue.
paint                 ... subject getValue.
setSubject:anObject   ...subject:=anObject.
```

```
class                  Histogram
superclass             Object
instance variables     boundingBox
instance methods
boundingBox: aValue
  boundingBox := aValue.
draw : aValue   ...
```

Listing 1: Classes People, Person, Dial and Histogram

3 An Object Environment

Besides classes representing all real world entities involved in an application, an object environment is a constituent part of each object-oriented application.

Each object can inform the object environment about a number of named events that occur in an object through its life-time. Each event has a name and each may have no, one or more attributes. When an event occurs in an object, the object sends a message to the environment of the form:

E event-name [: event-attribute { keyword: attribute}]

where [] means option and {} means iteration (0,1 or more times) of parts of an event message.

After getting information about the occurrence of an event in an object, the environment activates appropriate actions performed by one or more objects of the application.

Environment is the set of application event agents which bind together objects of an application. Event agents are event-action rules. One or more rules, called a subsystem manager, define the run-time interactions and enforce the required behavioral relationships or constraints between two or among more objects forming a subsystem. A subsystem consists of a set of objects dedicated to a special goal (functionality) according to a set of local event-action rules performed by a subsystem manager.

A number of run-time interactions between objects correspond to enforce relationships or constraints between objects. Such interactions can be simple defined by interobject rules (i.e. event-action rules named event agents) and they need not necessarily be explicitly described by means of procedural transactions. Besides agents which monitor and respond to events, the environment contains also transactions. Transactions instantiate the system, initiate objects of the system, prompt the user to initiate events, or remind the user that some action is to be performed. All other required application processes are expressed by behavior relationships among objects defined in an object environment by subsystem managers.

The suggested event-driven design provides a way of declarative description of application functionality by means of event-action rules defined in an object environment.

Listing 2 shows the realization of our application - information system about people by the use of an environment E.

Because an object can belong to more subsystems, more subsystem managers can require object to perform the different or the same actions (defined by methods of a class of the object).

```
class                 Dial
superclass            Object
instance variables    boundingBox
instance methods
boundingBox: aValue
  boundingBox := aValue.
draw : aValue
```

```
class                 Histogram
superclass            Object
instance variables    boundingBox
instance methods
boundingBox: aValue
  boundingBox := aValue.
paint: aValue
```

Listing 2: Information system about people designed by means of an object environment

```
class                Person
superclass           Object
instance variables surname age...
instance method      ...
age
  ^age       ...
```

```
class               People
superclass          Set
instance variables
instance methods
init ...
setValue: aValue
  E valueChanged:aValue.
 ...
older: years
     |anumber|
      anumber := 0.
     self do:[aPerson|
       (aPerson age > years)
       ifTrue:[anumber := anumber + 1]]
     self setValue:anumber.
younger: years
     ...
     self setValue:anumber.
old: years
     ...
     self setValue:anumber.
```

```
class                    E
superclass               Object
instance variables        views  people
class methods
"transaction: initialization"
init
  |a|     views:=Set new.   a:=Dial new.
  a boundingBox:50@50 extent:10@10.                  TRANSACTION
  views add: a.   a := Histogram new.
  a boundingBox: 80@80 extent: 10@10.
  views add: a.
  subject:=Subject new. people:=People init.
  E initialized
initialized  "transaction: user queries"
    people older:50. people younger:10.

"Subsystem manager              Reflection"
"of classes Subject, Dial and Histogram"
"All views reflect the subject value and
graphically represent them."                                           SUB-
valueChanged: anObject with: aValue              EVENT                SYSTEM
  views do:[aView                                                    MANAGER
  (aView isKindOf Dial)                                      EVENT-
    ifTrue:[aView paint:aValue]                  ACTIONS    ACTION
  (aView isKindOf Histogram)                                  RULE
     ifTrue:[aView draw:aValue]
```

Listing 2 (continued)

4 Benefits of Introduction of Events and Object Environment

The introduction of events and object environment as an event management system solves many problems in the object-oriented software development.

▪ Simple composition principle.

Incompatible objects can be composed without any modifications, because the object environment plays the role of adapters between otherwise incompatible objects. It receives messages about events that occur in an object (client of the client-server communication protocol) and translates them to calls that the other object (server) can understand.

▪ Increasing the reusability.

The use of an object environment enhances the class reusability by delaying the client-server binding. The environment permits the description of behavior specific to a particular application after the application components have been fully specified. This can be done without modifying the implementation of any component.

▪ Uncoupled programming style, which facilities establishing a

powerful libraries of easy reusable classes.
The fact that the client objects in the suggested events-environment model need not know anything about the servers and vice-versa, is central to the programming style. The sender notifies the environment about an event and sends event attributes. An environment then calls required actions and sends them all necessary received event attributes. Such communication enhances reusability and promotes uncoupled programming style. This facilitates establishing a large collection of independent reusable classes for software communities [7].

▪ Easy understandable and reusable programs.

It cannot happen that pieces of code failed to be object-oriented and at the same time difficult to be reused, because objects are not responsible for performing the ordered sequences of actions required to provide all application functionalities. This task is done by the object environment where the control flow of program is implemented.

5 How the Contents of an Object Environment can be Reused

Object environment defines all relationships among objects of an application. But same relationships can appear among many different objects in different applications. The question is how to reuse definitions of relationships. Complex structure and behavior of many applications from different domains can be the same. The same relationship as it is between People and Histogram in upper application exists for example between ball and obstacles in a Brickles game, where each movement of ball should be followed with position changes of obstacles. Obstacles in one way reflect the position of ball.

The common complex structure and behavior should be defined very abstractly to facilitate reuse. The complex structure and behavior of both applications consists of views (histogram and ball obstacles respectively) which always reflect the value of a subject (required statistical data about people and ball position respectively). Abstract description of this common complex structure and behavior is given in Figure 1 by means of E-R diagram and in Listing 3 with a class SubjectView which manages more subjects, each of which is presented with more views.

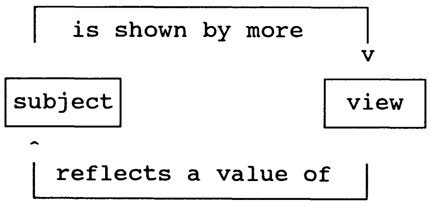


Figure 1: E-R diagram modeling relationships between parts and whole.

```
class SubjectView
superclass  Object
instance variables dic
instance methods  ...
init
  dic:=Net new.dic init.
attachView:aView to:aSubject
  dic connect:aSubject with:aView
allViewsOf: aSubject
^dic connectedWith:aSubject
"After initiation of subject,
 views have to be initiated."
initedS:aSubject
  self allViews:aSubject
   do:[:aView|E initW:aView]
    E do:aSubject
" All views allways reflect
subject value."
changedS:aSubject on:aValue
 self allViews:aSubject
  do:[:aView|
   E update:View for:aValue
endS
   self allViews:aSubject do:[
   aView| E end:aView .
```

```
class Net
superclass   Object
instance variables
instance methods
init
 connections:=Dictionary new
connect:aNode with:aNode
    ...
connectedWith:aNode
    ^...
```

Listing 3: Class SubjectView

6 The Event-Driven Object-Oriented Development

The use of events and object environment for the late binding of objects of an application enhances reusability by increasing the openness of objects. This nice property of suggested object design is not very efficient for software development without an appropriate development methodology. Following steps are suggested in the suggested event-driven object-oriented software development:

▪ Identify objects (and classes) in the application domain.

▪ Identify relationships between objects. Represent entities and relationships in E-R diagram.

▪ Identify and define structure and abstract behavior of each object. Develop or find and extent specification (class) for each object if necessary. Some objects are designed to be active. Active objects inform the object environment about changes of its states by event messages.

▪ Identify and define complex relationships. Classes which define parts of the E-R diagram of application are founded and reused or new classes have to be described.

▪ Put all classes together by means of an object environment and by abstract classes in strongly typed languages. First define transactions and then describe each relationship between two or among more objects by an event-action rule of the object environment. Event-action rules can define the required relationships also by reuse of existing classes modeling required relationships.

7 Conclusions

The use of events as a support for describing the system behavior is an old one. Events are used together with the triggering mechanism in many extensions of the data-flow analysis method for description of the system behavior.

The lack of efficiency is a major problem of many systems using some form of triggering concept. We have avoided this problem by the exact description of the order of actions executed at each event. In this way, there is no necessary search among objects of a system interested in particular event. Events are handled immediately after they occur.

The use of an object environment and events allows the extension of the traditional client-server communication paradigm. Events and object environment provide a simple mechanism for modeling system complexity and behavior. They increase object openness and reusability and provide a simple mechanism for describing the behavior compositions, constraints and dependencies among objects of an application.

Events generated by objects of a class are a part of a class and should be appropriately specified. Most events should be placed by the class designer, because they are a part of a class description. Environments containing subsystem managers are designed by the application developer.

Problems concerning the generation of events are not simple ones. There are many questions about generation of events: Which are active objects or which objects should be designed as active? What events should be generated? What event arguments are necessary? These questions have a great effect on the reusability and suitability of the event-driven design for the development of large library of reusable classes for software communities.

The suggested development methodology and design enable assembling applications rather than programming, what we still do today. We think, that the construction of complex software systems from powerful building blocks can greatly increase productivity.

At the moment, we can not give any experimental results about the efficiency of the suggested design for real applications in terms of software productivity and quality.

The suggested event-driven design enhances reusability. And considering the fact that when the effort required to produce a new code is larger than the effort of reusing the existed code the reusability is in proportion to productivity, we can conclude that the suggested approach improves productivity.

Software complexity theory suggests that a program with a larger variable span and live variable, decision count and readability will be more sensitive for future modifications. Thus, the software quality of a program with a large decision count, readability, variable span and live variable is considered to be poor. With observing of Listings 1 and 2 and the design of our information system about people done by means of an object environment which reuses SubjectView class for modeling complex behavior and structure between people and views can be concluded that variable span and number of live variables is decreased.

References

1. B. Henderson-Sellers, J. M. Edwards, "The object oriented system life-cycle", Communication on the ACM, vol 33, no. 9, Sept. 1990.

2. K. Rizman, I. Rozman, A computer aided prototyping methodology, ACM SIGSOFT SOFTWARE ENGENERING NOTES, Vol 14., No. 6, 68-72 (1989).

3. R.Wirfs-Broock, Object-Oriented Design: A Responsibility-Driven Approach", OOPSLA'89 Proceedings, 71-75 (1989).

4. R. Wirfs-Brock, B. Wilkerson, L. Wiener, "Designing Object Oriented Software", Prentice Hall, 1990.

5. R. Helm, I.M. Holland, D.Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", ECOOP/OPSLA'90 Proceedings, October, 1990, pp. 169-180.

6. B. Meyer, Reusability: The Case for Object-Oriented Design, IEEE Software, March 1987, 50-64 (1987).

7. S. Gibs, D. Tsichritzis, E. Casias, O. Niersatz, X. Bintando: Class Management for Software Communities, Communications of the ACM, vol. 33, no. 9, September 1990, 90-103 (1990).

8. G.Booch, Object-Oriented Design, The Benjamin Cummings Publishing Company Inc., (1991).

9. P. Coad, E. Yourdon: Object-Oriented Analysis, Yourdon Press, Prentice Hall, 1990.

10. A. Goldberg, D.Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

11. W. LaLonde , J. Pugh: Designing is Hard: Object-Oriented Software Is Different!, Journal on Object Oriented Programming, March/April 1989, 46-55 (1989).

12. D. Teanzer, M. Ganti, S. Padar, "Object-oriented Software Reuse: The Yoyo Problem", Journal on Object Oriented Programming, Sept./Oct. 1989, 30-35 (1989).

THE CHALLENGE OF COPING WITH COMPLEXITY

Chair: B. Dömölki

Usability is a Good Investment

Dipl.O.W.Sc.Inst. Tamas Marx
IQSOFT SzKI Intelligent Software Co.Ltd.

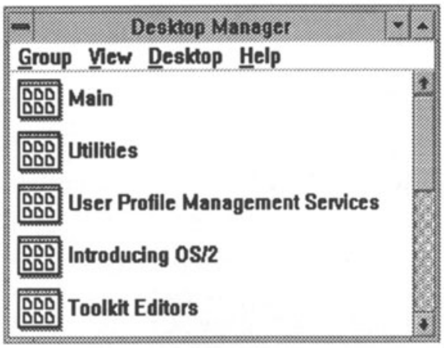


Figure 1

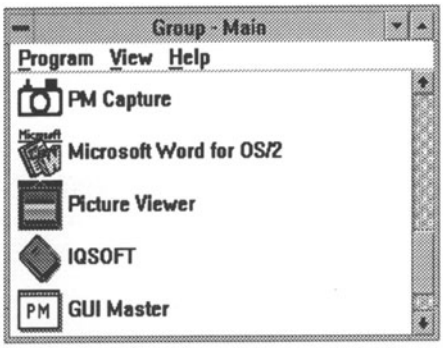


Figure 2

Abstract: Visual identification is certainly quicker than reading. However the above three examples show us that the use of grafical user interface is not as simple as puting little pictures everywhere. They are useful in Figure 2 but they are useless in Figure 1.

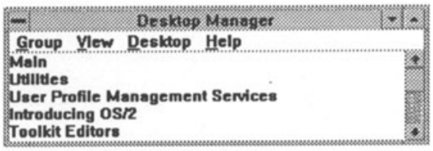


Figure 3

Technology has changed. New interfaces gave programmers possibilities they have never dreamt of. How and when to exploit these possibilities are the question!

- User interfaces are the decisive factors of the success on the market of a new software or your application in your company.

- There are terminals in front of more and more people, this should be the quickest growing branch of data processing. It is not.

- Usability testing is the part of the computer industry where we have do deal with the human factors.
  Software engineers alone can not handle this problem.

- How to mesure usability and how to estimate the profitability on any investment in usability are the question we will have to answer.

The speach will try to address the above problems with real examples on the screen and numbers from real projects will try to show you how to make your applications more usable and how to estimate its profits.

A Metaphor-Based Design Approach of a Graphical User Interface for Database Systems

G. Haring and M. Tscheligi

Department of Applied Computer Science
Institute of Statistics and Computer Science
University of Vienna
Lenaugasse 2/8
A-1080 Wien
Email: A4424DAF@AWIUNI11.BITNET

Abstract. The appropriate design of user interfaces has a fundamental influence on the acceptance of software systems. Today´s technology supports the realization of attractive user interfaces, which represent the functionality of the application to the user, based on the mental model. The paper describes the prototype design of a direct manipulative, graphical user interface for the core functionality of database systems. A new two-phase interleaving prototype development cycle is proposed for the design process. The general design philosophy and some basic interacting user interface objects, based on the real life look metaphor, are described in detail. Finally the embedding software architecture is outlined.

1 Introduction

Beside the processing of text the management of data is the main centre of interest of today´s office activities. The support of data management activities is presented to the human by data base systems. These systems offer a wide and complex range of functionalities including schema design, querying, browsing and manipulation for different classes of users (database designer, application developer, end user). A lot of database knowledge is necessary to use the data base system with the aimed success.

The amount of theoretical database knowledge should be minimized by an intuitive and easy understandable user interface. The importance of the user interface nowadays is widely accepted by the computer community. Due to this importance the solution of the user interface problem has to be considered as a leading activity in every software development process . Therefore steps of user interface development must be integrated in software engineering methodologies to achieve a sophisticated form of human computer communication.

* this work was supported by the Austrian National Bank under grant 3692

Graphical user interfaces are becoming mandatory for every interactive software system. This modern type of user interfaces takes advantage of the visual channel of humans. The states of the application are transferred to the user by graphical presentations and the user is able to manipulate these presentations to transmit the intentions to the computer system. The manipulation of the graphical objects results in an altered internal state. Direct manipulation [7] is an important term often mentioned in this context. Graphical interaction techniques and direct manipulation have to be used to hide the complexity of data base systems to the user. Too frequently the user interface is oriented to the underlying data model. In this paper we describe the result of going in the opposite direction: hiding basic and theoretical data base concepts to the user. The design of the user interface is primarily influenced by the users mental imagination of data management tasks and is based on a collection of real life looking interacting objects.

Prototyping is accepted as the leading approach for the development of user interfaces due to the early evaluation possibility. Unfortunately existing implementation environments for graphical user interfaces are not the ideal platform for an efficient and rapid evolutionary prototyping process. This statement is extremely valid when alternative user interface techniques have to be tested, without any alignment to existing user interface standards. Therefore a two step prototyping process was introduced during the development of our user interface prototype.

The aim of this paper is to show the general concepts of an alternative user interface for common database functionalities, where look and feel is based on the usage of user oriented metaphors. Before the specific appearance and behavior of the user interface is demonstrated we outline the above mentioned prototyping process together with a clarifying discussion of nomenclature necessary to describe activities of the user interface development process. Afterwards some remarks are made concerning the specific software environment and software architecture where the development of the user interface prototype took place. In the last section some conclusions are drawn from this design project.

2 A Prototyping Oriented Development Approach

So far a well defined characterization of necessary activities for the construction of a graphical user interface is missing. To define a structured methodology for this task we use terms already established in the software engineering community to circumscribe the process of development: the analysis of existing designer models and mental models of potential users results in a user interface requirements definition, the user interface design activity yields in a user interface specification (in our case a written specification is almost completely replaced by a first prototype), the user interface implementation design results in a general software architecture for this type of user interface and the user interface implementation activity leads to the second prototype.

This activities are shown in Fig. 1 in a top down form unless some of the activities can be overlapping and iteratively repeated. In particular this is necessary during the design, where several versions of prototypes have to be produced before a sophisticated level was reached. Regarding this overlapping the user interface implementation design started before the first prototyping process was finished. The general user interface philosophy created during the design activity is suffice for the conceptual work on the architecture.

2.1 Designer Models and User Models

The designer model is the mental imagination of the basic data base functionality on the side of the user interface designer transferred to users by existing data base systems. With already existing and running software and the accompanying documentation a specific mental model from the functionality of underlying data management tasks is built on the user´s side. The evaluation of some existing data base systems aimed at the identification of existing designer models and to get a feeling of implemented data base system functionality. In the evaluation mainly PC-based products were used including Apple´s Hypercard approach to get an input also from non traditional data base oriented solutions.

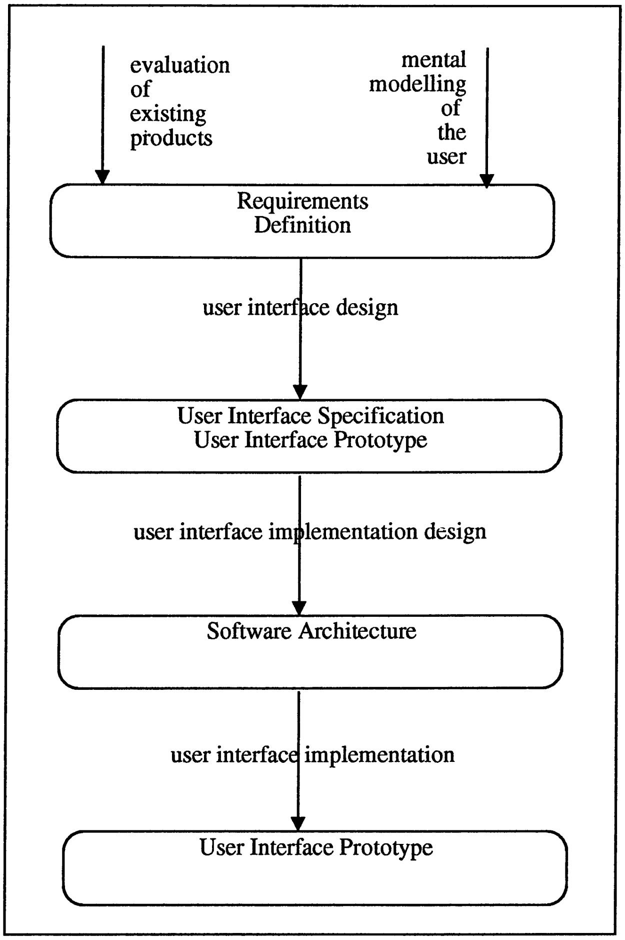

evaluation of existing products
mental modelling of the user
Requirements Definition
user interface design
User Interface Specification
User Interface Prototype
user interface implementation design
Software Architecture
user interface implementation
User Interface Prototype

Figure 1

For the second activity leading to the requirements definition we tried to identify the ideas of potential data base system users without any reference to existing software solutions of data base management functionality. The users had to show the different ideas by sketching it on paper. In Table 1 the results of evaluation is shown for the principle data base concepts "database" and "file".

group	database	file (table)
1	filing cabinet	folder
2	wall cabinet	folder
3	archive	cardfile
4	wall cabinet	table book
5	collection of books	book
6	cabinet with books	folder
7	cabinets	folder, books

Table 1

This survey shows a clear preference for objects already used within non computer supported data management in everyday office life. The results of this evaluation encouraged our attempt to maximize the integration of a real life oriented metaphor.

2.2 User Interface Design - The First Prototype

As the first step within this activity the general principles of the intended user interface were specified. The whole user interface is totally composed of objects, each with a specific semantic regarding data base tasks or general object manipulation tasks. A set of general manipulation primitives [6, 10] were defined to reach a consistent and object independent manipulation style. In the following individual graphical objects are invented for specific data base functionalities to get a starting point for the following prototyping cycle.

There was already mentioned that available user interface implementation platforms are not the ideal environment for really rapid prototyping. User interface implementation within a window environment requires a lot of programming to yield a working piece of software. User interface toolkits at a higher level of abstraction are oriented towards a predefined and specific user interface style with minimal freedom for the goal of alternative user interface testing.

Therefore we decided to use another type of product for the user interface design activity to experiment with different design alternatives not restricted by common user interface implementation conditions. In particular MacroMind Director [5] was used which is a tool for the assembly of multimedia presentations. In this case multimedia presentations denote the production of high quality animated color graphics for different purposes. With this orientation to animation the intended user interface can be demonstrated not only statically but also dynamically. Several steps of evaluation and redesign were necessary to yield a sophisticated prototype which could be used as basis for further development and transformation to the target implementation environment.

2.3 User Interface Implementation Design

As every software system user interface code demands for a careful design of the internal software structure. The term implementation design was chosen to create a distinction to the external (user oriented) activity of user interface look and feel design.

Within this activity a general object oriented software architecture was introduced to realize this type of user interfaces with reusable components. The activity of user interface implementation design started before the previous activity was finished with the general interaction possibilities defined at the beginning of the design activity as general guideline for the software structure.

2.4 User Interface Implementation - The Second Prototype

Based on the software architecture defined in the above mentioned activity the user interface objects specified within the first prototyping cycle were transformed to the target environment successively. Though we denote this activity with the term implementation it is a matter of fact that in this activity also prototyping took place. This is due in part to different software and hardware platforms. The first prototyping process was done with a Macintosh and a one button mouse and the second prototype was produced on a SUN workstation with a three button mouse. In addition the permanent attempt to improve the user interface is another reason.

The prototype was oriented to the NeWS windowing environment [3] and implemented in the Postscript extension of NeWS for the device dependent part together with some amount of C++-Code for a device independent part. Due to availability of a working prototype many problems of communicating the ideas of the user interface designer to the implementation group was weakened.

3 The General User Interface Philosophy

The whole user interface is composed out of different objects. The available objects conceptually put the user into an office environment. All database functionality is embedded in manipulation relationships between these available objects. With this approach menus common in usual systems are obsolete. The mouse cursor acts as special object with which the semantic of the interaction relationships between different objects are triggered for the most part. Some manipulations of the objects are directly carried out by the mouse. In a metaphorical sense the mouse cursor acts as the lengthening of the user hand and touches the objects in question. This corresponds to the general directness demand of direct manipulative systems [4]. The keyboard is only used for data entry (characters, numbers).

Three possible operations with the mouse are mapped to higher level manipulation primitives. The general idea of defining manipulation primitives results in a clear and consistent manipulation philosophy which is intuitively understandable and rememberable for the user of the system. If the primitive is applicable the general semantic of the manipulation primitive is the same regardless of the object type involved.

The mouse operations used for this purpose are a single click (pressing and releasing a mouse button without any movement in between), a double click (pressing and releasing a mouse button without any movement twice in a very short period) and dragging (pressing the button, dragging the mouse to another position on the screen and releasing the button).

The mapping of mouse buttons to higher level manipulation primitives is defined in Table 2. With the *activation* of an object the user defines special interest on the object mainly to prepare the object for further operations or select associated object attributes as current adjustments. The activation is accompanied with object specific feedback to show the user possible usage possibilities. With the *open/close* primitive the user usually toggles between an external and an internal presentation for objects which are used as data containers. The *positioning* manipulation primitive allows the user to alter the current object position. If there exists a special interaction relationship between the manipulated object and another touched object the specific functionality is initiated if both objects are overlapping. Objects can be rotated by a *rotate* primitive and resized by a *resize* primitive as known from some window managers.

interaction primitive	mouse operation
activate	single click with left mouse button within object region
open/close	single click with right mouse button within object region
position	dragging with the left mouse button within object region
rotate	dragging of a object corner with the left mouse button
resize	dragging of an edge or corner with the right mouse button

Table 2

The problem of getting objects for the first time is solved by the usage of a special catalogue object. The catalogue contains all the objects available in the system and the user gets them by simply opening and dragging the objects out. The catalog is self reproducing to enable more than one instance of a special object. In addition to the general manipulation primitives some other interaction relationships are collectively in existence: copying with a copying machine, printing with a printer object, coloring of objects with a color bucket and a subsequent selection from a color palette which represents available color possibilities, deleting objects with a trash can and scrolling with an fully independent scrollbar with an alternative look and feel.

4 The Look and Feel of User Interface Objects

In the following subsections objects from the prototyped data base user interface are described according to the object oriented organization of the user interface. These objects and their relationships together result in the user interface for this type of functionalities. If other objects are referenced in the object specific description these are written in italic form.

4.1 Cardfile

For this object a rectangular colored cardfile representation is used also showing cards within the cardfile (Fig. 2). The amount of visible cards shows the current utilization of the cardfile. On the front side a label object can be placed. With this label the user is able to enter the name of the card file. With the open/close manipulation primitive the cardfile can be opened and then cards are presented in a stack form. Cards only exist behind a special object mask.

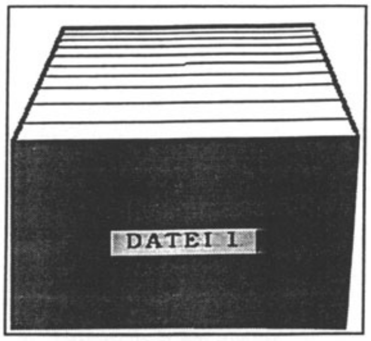


Figure 2

The presentation of the stack is visually strengthened by a background color in which the stack is contained. This background color can also be used to close the cardfile. By positioning the cardfile (in an opened or closed state) to another cardfile the contents of the cardfile can be transferred to the target object. At the end of each stack an empty card is available to input new data. For changing the visible card the interaction relationship with the *scrollbar* can be used.

4.2 Label

By activating the label a cursor appears at the beginning of the label. Now the user is allowed to input or edit a name. The label adapts to the length of text. One or more labels can be positioned to a cardfile or another object. The label also interacts with the scrollbar to see hidden (e. g. the label was reduced in size before) contents of the label.

4.3 Masks

Masks are a special form of card dedicated to the definition of views on existing cardfiles. Without a mask nothing can be seen from the data contained on a specific card. The mask is also an object which comes in an open and closed presentation. The closed form can be seen in Fig. 3. Available masks (open or closed) can be attached (in conceptual terms the mask is inserted before the data cards) by activating or by positioning. At any time only one mask can be active.

The open mask is presented in an A4 paper sheet form and can contain fields. This fields are the placeholders for the data. On every place in the card additional text can be inserted without fields. The fields can be selected from an object called attribute list, from the list of computed fields and/or from the list of joined fields. The contents of the fields can be protected from editing by placing a grid over some part of the mask.

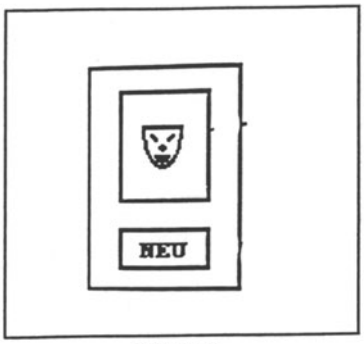


Figure 3

4.4 Fields

Fields are represented by a rectangular white bar. Fields can be obtained from the attribute list, the list of computed fields or the list of joined fields. By activating the field the cursor is positioned at the beginning of the field and the user is able to enter something. The field is aimed to different forms of data (text, images, sound) and automatically alters the size if necessary by the contents. If a field does not show all the data at once the srollbar is usable again.

4.5 Query Card

The catalog contains also a special form of card for the execution and definition of queries (query card tool, Fig. 4). By activating a special query card the query associated with the manipulated query card is applied. Several queries can be active at the same time. A subsequent query is applied to the state of the cardfile after the already activated query.

At the same time the query card is integrated into the appropriate place at a special presentation called the top view (Fig. 5). The top view shows the partition of the card file caused by applied queries. An activated query card partitions the card space according to the query condition.

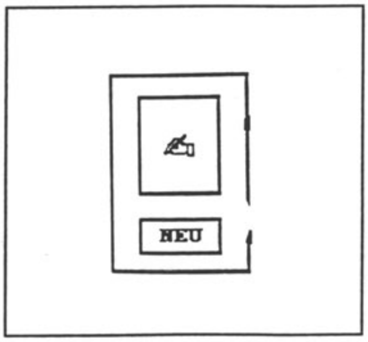


Figure 4

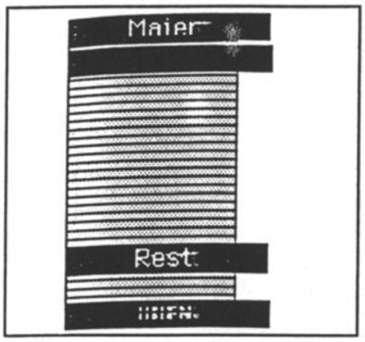


Figure 5

The query card is shown between the cards which fulfill the query condition and the cards where this is not the case. In the top view existing query cards can be activated and removed directly. An empty query card is always shown at the back of the top view, which is replaced automatically by a new one if it is used. To deactivate any queries the correponding cards can be placed behind the empty card for further usage.

The query card can be opened. In the opened presentation one or more conditions can be attached to fields selected from the attribute list. With a special object billiard ball a sorting direction can be specified.

4.6 Scrollbar

The scrollbar can be attached to any object with scrollable contents in order to see other parts of the content. The scrollbar is shown in Fig. 6. Visually the scrollbar comes with two triangles within the scrollbar frame. The triangles serve as slider which can be moved. By activating one of the triangles the scrolling is executed by one line, the activation out of the triangles but within the scrollbar borders causes a page oriented movement.

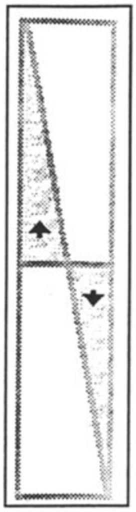


Figure 6

As special feature of this type of scrollbar is the possibility of scrolling in alternative directions. The same scrollbar can be used to scroll horizontal, vertical or diagonal. This is simply adjusted by rotating the scrollbar object with the rotating primitive. So scrolling towards the third dimension necessary within the used cardfile representation can be done.

4.7 Browsebox

The browsebox is an object to give the user another possibility to view the data inside a cardfil. By opening the browsebox a table is presented with all available data within a specific cardfile. The first line shows the names of the fields which values are shown on the subsequent lines in traditional form. Usually the available space is restricted and therefore the scrollbar can be used again. By activating the desired portion of the table some data can be edited or added. As soon as the data is altered within the table it is also taken over to the card representation.

4.8 Tab

This object is intended for marking special data cards, mask cards or query cards (Fig.7). It is for example useful if the user wants to find a particular card out of the lot of card usually found in a cardfile. The tab can be designated by the object label, as any other object. To attach a tab to a card the tab has to be positioned on top of the target card. The association holds as long as the tab is not removed from the surface of this object. By activating the tab the associated card is activated.

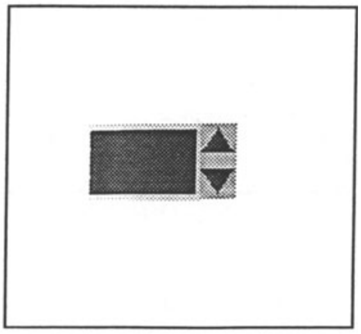


Figure 7

4.9 Attribute List

In the opened presentation the attribute list contains all available fields of a cardfile (Fig. 8). This object is always presented on top of all other objects and gets a transparent appearance. New fields can be added and existing fields can be edited. By opening the attribute list (the closed is shown Fig. 9) the fields are shown from all existing cardfiles with the current color of the cardfile. If the cardfile is positioned over a cardfile only the fields of the touched cardfile are visible. The activation of special field entry causes the field to appear on a mask object.

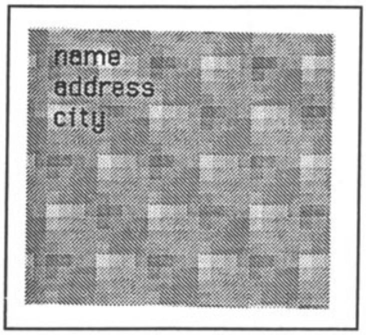


Figure 8

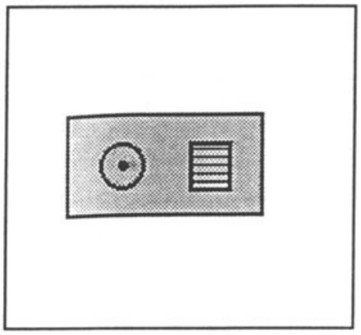


Figure 9

4.10 List of Computed Fields

All computed objects of a cardfile are represented with this type of list. Again this list always lays on top of other objects and the visual presentation is transparent. Computed fields are defined by an arithmetic expression using fields of the attribute list. The opened presentation allows editing of existing and creation of new computed fields. By simply opening the list of computed fields the fields from all cardfiles are presented. After the positioning over a specific cardfile only a specific set of computed fields is presented. The activation of a special field entry results in an integration into a mask card.

4.11 Billiard Ball

The billiard ball object acts as a possibility to define fields used for sorting the cardfile. The user has to activate the billiard ball for that field which should be used for the sorting. If within a sorting definition another sorting is requested a further billiard ball has to be used. This ball is used with the subsequent number to indicate the next sorting level. The defined sorting order also can be stored with a query card.

4.12 Data Base Manager

Joining data base parts is very complex task very hard to understand especially for the naive user. In particular this sort of task in this form is not existent in reality without a computer supported solution. Nevertheless an object had to be defined with some connection to real life appearance supporting the formation of the mental model based on analogy.

A database manager consists out of different regions: a head, a body and some hands (Fig. 10). The hat of the data base manager shows the operators forming the condition under the card files are joined. The connection is established by positioning one hand of the data base manager to the first cardfile and the second hand of the data base manager to the second cardfile. Automatically the data base manager gets a new hand which also can be used to integrate another cardfile. The specific fields are selected by the finger of the corresponding hand using the attribute list.

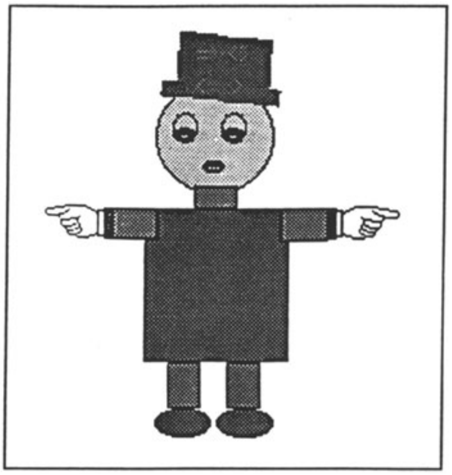


Figure 10

The activation of the head stands for the start of the execution of the defined functionality. As special progress feedback the eyes of the data base manager are rolled. The result of the join is represented within the body as a different object virtual cardfile.

4.13 Virtual Cardfile

The virtual cardfile is represented within the body of the data base manager and represents the result of a join. By activating the head of the data base manager the user is able to learn more about the join conditions used. Automatically the connections are shown with the hands of the data base manager. In a special list object called list of joined fields the involved fields and operators are visualized. The structure of the virtual cardfile can be changed by using other fields or another operator. By opening the virtual cardfile virtual cards are represented.

4.14 Virtual Cards

On one card the stacks of all involved cardfiles are visualized which cards satisfy the join conditions. To indicate the source cardfile the same underlying color is used. The activation within a virtual card leads to the possibility of editing the contents. An empty card is available at the end of each stack. By srolling to the card of one stack the corresponding card of the other stack is also visualized. As with the regular form of a cardfile objects like the browsebox, query card, mask card or billiard balls are also applicable.

4.15 List of Joined Fields

Like the other lists in this user interface approach this object shows the fields involved in the join on a separate list. This list is also applicable for mask cards by the activation of a special field. The list of joined fields cannot be edited.

4.16 Freezer

The freezer object is used to transform a virtual cardfile to a "normal" cardfile. The fields are combined and presented on one card. Joined fields exist only once. By default the fields are layouted in a column format but the user can redefine it or create other masks.

4.17 Grid

As already mentioned the grid object is used to restrict the access to some objects. If an object is protected by a grid only a well defined group of persons is able to manipulate the object. In the current version the grid is only used for protecting fields.

5 Software Architecture

In this section some remarks are made concerning the software architecture of the second prototype. A detailed description of the implementation can be found in [8, 9]. The architecture is based on the principles of application frameworks or user interface frameworks [2]. Application frameworks offer some amount of user interface code in form of reusable and extendable standard objects. Application frameworks utilize the techniques of object oriented programming for the implementation of graphical user interfaces.

Usual frameworks only support primitive mouse events. All higher level manipulation primitives have to be implemented for every new problem situation. Therefore direct manipulative user interfaces as introduced in this paper are not supported in a sufficient way.

So the support of interaction primitives defined above was the first important goal for our specific software architecture. The user interface programmer has the possibility to control the interaction primitives (enable, restrict or forbid) and select suitable forms of notification after an manipulation.

The second goal was to combine the higher level of abstraction with a sophisticated form of portability and reusability for different windowing platforms. Therefore the whole framework was splitted up into two functional parts.

The first application framework implements dialog control functionality [1] without any consideration to the presentation peculiarity regarding the concrete presentation of objects. The presentation part is the second application framework which is aimed at the presentation of the different graphical objects and the preparation of user interaction within a concrete windowing environment and without the consideration of their effects to and the triggers within the dialog control part.

The dialogue control part contains the whole user oriented semantic of the user interface. Only in this part objects are created and destroyed. As already mentioned the dialogue control was implemented in C++. The presentation part was mainly realized in Display Postscript with a C interface to the control part. Display Postscript is an extended version of Postscript used in the NeWS windowing environment for the application programmer interface.

6 Conclusions

This paper presents results from ongoing research regarding alternative methods of human computer communication. The user interface style introduced here is characterized by an exceptional object orientation from the user side of the system. The explicit usage of metaphors is another goal for the system. Several functions of database functionalities are considered and transformed into the selected interaction philosophy but by no means our system covers all available functionality of today´s data base systems. Due to the object based style missing objects can be easily integrated into the overall interaction style.

Existing user interface objects have to be refined in particular regarding the formulation of queries or other types definition tasks using a more visual oriented definition language. Additional objects are also necessary for the organization of the office due to the huge amount of existing objects. Further releases of the design not reported here include some objects for this task.

The two phase prototyping cycle used for the development of the user interface was very helpful to achieve an early discussion base for the evaluation of ideas concerning the design and the transformation of user interface concepts to more implementation oriented development activities. Future work regarding the development methodology is necessary for a better support of the requirements definition support and a better orientation of existing prototyping tools to the needs of alternative user interface designs to weaken the need of using different development environments.

References

1 L. Bass L., J. Coutaz J.: Developing Software for the User Interface. Reading, Mass.: Addison-Wesley 1991
2 M. Dodani, C. Hughes, M. Moshell M.: Seperation of Powers. Byte, March 1991
3 J. Gosling, D. S. H. Rosenthal, M. J. Arden: The NeWS Book. An Introduction to the Network/extensible Window System,.SUN Technical Reference Library. New York: Springer 1989
4 E. L. Hutchins, J. D. Hollan, D. A. Norman: Direct Manipulation Interfaces. In: D. A. Norman, S. W Draper (eds.): User Centered System Design: New Perspectives in Human Computer Interaction. Hillsdale: Lawrence Erlbaum 1986, pp. 87-124
5 MacroMind Inc., 410 Townsed St., Suite 408, San Francisco, CA 94107.
6 F. Penz, M. Manhartsberger, M. Tscheligi: The World of Objects - A Visual Object Based Interaction Language. In: Proceedings of the 10th Interdisciplinary Workshop on Informatics and Psychology, Schärding, Austria, May 21-23 (1991)
7 B. Shneiderman: Direct Manipulation: A Step Beyond Programming Languages. IEEE Computer 16, 8, 57-69 (1983)
8 B. Strassl B., F. Penz: CommonInteract - ein objektorientiertes System zur Entwicklung direkt manipulativer Benutzerschnittstellen. In: Proceedings UNIX Forum IV, Vienna, Austria, Oktober 1991 (in german)
9 B. Strassl B., F. Penz: CommonInteract - an Object Oriented Architecture for Portable Direct Manipulative User Interfaces, appears in Journal of Object Oriented Computing
10 M. Tscheligi, F. Penz, M. Manhartsberger: N/JOY-The World of Objects. In: IEEE Workshop on Visual Languages, Kobe, Japan, October 8-11 (1991)

Links in Hypermedia Systems

Frank Kappe, Hermann Maurer

Institute for Foundations of Information Processing
and Computer Supported New Media (IICM),
Graz University of Technology, Graz, Austria

Ivan Tomek

Jodrey School of Computer Science, Acadia University,
Wolfville, Nova Scotia, Canada

Abstract

In hypermedia systems, pieces of information (so-called nodes) are tied together by so-called links. This paradigm is often considered the as the single most important feature of hypertext/hypermedia systems. However, in actual implementations of such systems there are a number of questions (open or partially resolved), design issues, and tradeoffs related to features and attributes of links.

In this talk, we discuss these questions as well as possible answers, including:

- Should links be single-ended or multi-ended?
- Should links be unidirectional or bidirectional?
- What types of entities should links be attached to?
- What kind of media should links be attached to?
- What should be the granularity of link attachment points?
- What information should be displayed before activating a link?
- Should links be typed?
- Should links have attributes?
- Should links be used for specification of node attributes?
- Should links be cold, warm or hot?
- How should links be displayed?
- How should links be created?
- How should consistency of links be maintained?

In addition, some notes concerning the actual implementation of links in a general-purpose, large-scale, multi-user hypermedia system will be made.

A NEW APPROACH TO DEFINING SOFTWARE DESIGN COMPLEXITY

László Varga
Department of General Computer Science
L. Eötvös University,
H-1117. Budapest, Bogdánfy u. 10/b.

Abstract: A general method is given for defining architectural design complexity measures. Desired properties of a measure are described by functional equations. Two cases of descriptions are considered. Complexity measures are given as the solutions of functional equations. Other complexity measures can be regarded as special cases of the solutions. A new measure is also presented.

Introduction

Software design is the most critical part of the software development process. In this period of software life cycle the structure of the pending software system is defined and the system is fully specified. The quality of a software design plays an important role in reducing software cost. This is because, researchers have attempted to find quality measures for characterizing software design. Among the quality measures probably the most important is the complexity measure.

In spite of the importance of software complexity it is insufficiently known and defined. It is necessary to distinquish between computational and psychological complexity of software.

Computational complexity is a qualitative characterization of algorithmic solution of a problem. It is measured by the amount of resources used by the solution.

Psychological complexity is a qualitative characterization of the misunderstandebility of a software. It can be measured as the difficulty of performing programming tasks as coding, testing and modifying software. Objective of this paper is psychological complexity.

The most often cited software complexity measures [Halstead 1977, McCabe 1976, Prather 1984] treat a program as a symple body of code. More recently, complexity investigations have attempted to characterize complexity of the relationships among the modules of a system [Card 1988, McCabe 1989].

In the paper [McCabe 1989] the cyclomatic complexity is applied to architectural hierarchical design of a system.

Common feature of almost all measures mentioned above is that measures are based on intuitions. For example, idea of cyclomatic complexity measure is that the difficulty in understanding a program can be approximated by the maximum number of linearly independent paths through a program.

In the paper [Daróczy 1988] the authors proposed a new approach to defining complexity measures. New feature of this approach is the following: Intuition is used for describing the desired properties of a measure and functional equations are used for describing properties.

The objective of this article is to extend the functional equational method into architectural hierarchical design of a programming system.

Architectural design

During the design phase of program life cycle, the system which satisfies the requirements, must be decomposed into seperate components. In this paper, the components are modules.

There have been many design methodologies developed in different applications. Among them the top-down functional decomposition has been widely used. At this method the programming system as a hierarchy of parts can be described by structure chart. An example of a structure chart is shown in Figure 1.

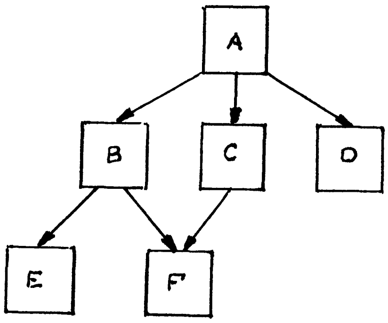


Figure 1. Sample structure chart.

Generaly, a sturcture chart defines how modules work together, but it does not define how each module works.

Software design is described by a suitable language. Design description languages generally use the control structures of high level programming languages with natural language description of operations. Variants of high-level programming languages such as ADA could be used as design description languages. For example, a high-level description of the well known spelling checher could be:

```
procedure SPELLCHECK is
begin
    produce list of words in document in short order
    loop
        get word from word list
        if word not in dictionary then
            handle unknown word
        end if
        exit when all word processed
    end loop
    create new dictionary
end SPELLCHEK
```

Compound operations such as

"produce list of words in document in short order"

"handle unknown word" etc.

can be realized by procedure call.

Module specification generally gives precondations to procedure calls. For example the operation

"handle unknown word"

has the precondation

"not all word processed and the given word is not in dictionary".

According to this comments the stucture design of a programming system is defined as follows:

Definition 1. A structure chart $SC = (N, E, m, T)$ is a directed graph with a finite, nonempty set of nodes N, a finite, nonempty set of edges E, a master node $m \in N$, and a finite, nonempty set of terminal nodes $T \subset N$.

- Each node $n \in N$ lies on some path from m to $t \in T$.
- $m \in M$ is a unique node which has no predecessor.
- Terminal nodes ($\forall t \in T$) are characterized by the property that they have no successor.
- Each path in SC has no cirle.
- Nodes are used for representing modules.
- An edge is an ordered pair of nodes $n_1 \rightarrow n_2$ and it means, that module n_1 calles module n_2 or module n_2 is called by module n_1.

Definition 2. The module is a "case" structure plus its module function.

$$\text{case } (\alpha_1 : \text{ call } m_1;\ \alpha_2 : \text{ call } m_2;\ \ldots\ \alpha_n : \text{call } m_n);\ S$$

where $\alpha_i (i = 1, 2, \dots, n)$ is logical expression, $m_i (i = 1, 2, \dots, n)$ is module name and S is the module function. Terminal modules in SC have no case structure.

Definition of design complexity measures

A recursive definition could be given for design complexity measures using the hierarchical structure of SC.

Let the module

$$m = \text{case } (\alpha_1 : \text{call } m_1;\ \dots;\ \alpha_n : \text{call } m_n); S$$

be given with the complexity measures

$$a(\alpha_1);\ a(\alpha_2);\ \dots;\ a(\alpha_n);\ b(S);$$

$$c(\text{call } m_1);\ c(\text{call } m_2);\ \dots;\ c(\text{call } m_n).$$

The design complexity measure of module m is

$$dc(m) = f(a(\alpha_1), c(\text{call } m_1); \dots;\ a(\alpha_n), c(\text{call } m_n);\ b(s))$$

where f is a function to be determined.

The question is what kind of function f characterize the design complexity of a program sufficiently? To find an appropriate measure its properties should be formulated.

Let

$$f(x_1, y_1; \dots; x_n, y_n; z)$$

be the function in demand. Obvious to investigate the result of changes in its argument.

<u>First appromation</u>

The function

$$f(x_1 + x'_1, y_1 + y'_1; \dots; x_n + x'_n, y_n + y'_n; z + z')$$

is characterized by the property:

$$f(x_1 + x'_1, y_1 + y'_1; \dots; x_n + x'_n, y_n + y'_n; z + z') \geq$$

$$f(x_1, y_1; \dots; x_n, y_n; z) + f(x'_1, y'_1; \dots; x'_n, y'_n; z')$$

Our requirement is not very profound. It means that the complexity of a system is greater than or equal to the complexity of its original complexity plus the complexity of increments. A solutions for f could be get using the equality relation.

Theorem 1.

Let us now suppose that

$$f(x_1 + x'_1; y_1 + y'_1; \ldots; z + z') = f(x_1, y_1; \ldots; z) + f(x'_1, y'_1; \ldots; z'),$$

then

$$f(x_1, y_1; \ldots; x_n, y_n; z) = \sum_{i=1}^{n} (d_i x_i + e_i y_i) + gz$$

with convenient constants $d_i, e_i, i = 1, 2, \ldots, n; g$.

The formula could be proved by using the solution of the well known Cauchy equation.

How can we get a better approximation? It is obvious to suppose that the difficulty of understanding the relationships among modules depends on the complexity of decisions. In this case we get the following:

<u>Second approximation</u>

$$f(x_1 + x'_1, y_1 + y'_1; \ldots; x_n + x'_n, y_n + y'_n; z + z' =$$

$$f(x_1, y_1; \ldots; x_n y_n; z) + f(x'_1, y'_1; \ldots; x'_n, y'_n, z') +$$

$$f(x_1, y'_1; \ldots; x_n, y'_n; 0) + f(x'_1, y_1; \ldots; x'_n, y_n; 0)$$

Theorem 2.

$$f(x_1, y_1; \ldots; x_n, y_n; z) = \sum_{i=1}^{n} h_i x_i + gz$$

with convenient constants $h_i, i = 1, 2, \ldots, n; g$.

Proof of the theorem also could be derived from the solution of Chauchy equation.

If we suppose, that

$$c(\text{call } m) = i(m) + dc(m),$$

where $i(m)$ is the complexity of modula interface, then both approximations provide a recursive definition for the design complexity measure.

Really, the second approximation yields an extension of the measure given by [Prather 1984] to arhitechtural design of a system.

An example

The question is what kind of weights could be choosen in our formulae? Let all be equal to unit.

Let us see the structure chart in Figure 1., with its associated functions:

$$\begin{aligned}
&A = \underline{\text{case}}(\alpha_{AB}: \text{ call } B;\ \alpha_{AC}: \text{call } C;\ \alpha_{AD}: \text{call } D);\ S_A;\\
&B = \underline{\text{case}}(\alpha_{BE}: \text{ call } E;\ \alpha_{BF}: \text{call } F);\ S_B;\\
&C = \underline{\text{if}}\ \alpha_{CF}\ \underline{\text{then}} \text{ call } F;\ S_C;\\
&D = S_D;\ E = S_E;\ F = S_F.
\end{aligned}$$

Using the short forms:

$$a(\alpha_{XY}) = a_{XY};\ i(X) = i_X;\ b(S_X) = b_X,$$

second approximation gives the following design complexity measures:

$$\begin{aligned}
dc(D) &= b_D;\ dc(E) = b_E;\ dc(F) = b_F;\\
dc(C) &= a_{CF}(i_F + b_F) + b_C\\
dc(B) &= a_{BE}(i_E + b_E) + a_{BF}(i_F + b_F) + b_B\\
dc(A) &= a_{AB}a_{BE}(i_E + b_E) +\\
&\quad (a_{AB}a_{BF} + a_{AC}a_{CF})(i_F + b_F) +\\
&\quad a_{AB}(i_B + b_B) + a_{AC}(i_C + b_C) +\\
&\quad a_{AD}(i_D + b_D) + b_A
\end{aligned}$$

Let $a_{XY} = 1$ and $i_X = 0$ for all X, Y in the system. If the structure chart is a tree then the formula is reduced to McCabe design complexity measure.

REFERENCES

1. Card, N.D. and Agresti, W.W., Measuring software design complexity. The J. of Syst. and Softw. 8.(1988) 185-197
2. Daróczy, Z. and Varga, L., A new approach to defining software complexity measures. Acta Cybernetica 8(1988) 287-291
3. Halstead, M.H., Elements of software science. Elsevier, New York, (1977)
4. McCabe, T.J., A complexity measure. IEEE Trans. Software Eng. 2.(1976) 308-320
5. McCabe, T.J. and Butter, C.W., Design complexity measurement and testing. Communications of the ACM, 32(1989) 1415-1425
6. Prather, R.E., An axiomatic theory of software complexity measure, The Comp. J. 4(1984) 340-347

METHODOLOGY AND EXPERIENCE

Chair: G. Klimko

SOFTWARE DEVELOPMENT ON THE BASIS OF FRAME-CHANNEL MODEL

H. Maurer, N. Scherbakov
IIG, Technical University
Graz, Austria.

Abstract. In this article, a new paradigm in software engineering is discussed. In accordance with this paradigm a software system can be seen as a number of so-called frames connected by a number of channels.

Hence, we call this model the frame-channel model. Frames can encapsulate concrete actions such as execution of procedures, interpretation of database queries, infer procedures, and so on. The concept of channels allows to combine a number of frames into single software system in an elegant fashion.

The model can also be used in coauthoring numerous, large, software-related documents throughout the software life cycle.

1. INTRODUCTION

In order to successfully develop large software systems more or less formal models must be used. Such formal models are particularly important in the context of computer aided design of software systems [3]. In this case, the users i.e. the software developers, prepare and assess concrete decisions about a certain software project by means of computer systems [1]. In this paper we introduce a novel computer-based model for the described purpose.

Basically our idea is to describe an object-oriented software design approach in which we deal with the problem of addressability in a new way. Usually, messages in an object-oriented system are either sent to explicitly named objects (creating the well-known problems of naming conflicts, etc.) or else they are sent to all objects and only those with specific properties will act on them. The latter approach leads to serious problems in systematic program debugging and specification. We are choosing as alternative a hypermedia-kind of network consisting of links (which we call channels) allowing to pass messages from an object to a selected number of others.

We hope that such a "hypermedia" structure of a large software system leads to new possibilities in software development, verification and specification including reusability of source codes and software documents. The main motivation for reusing existing source codes and software documents in general is to improve qualities and productivities within a well-coordinated organization and increase usabilities of resources.

2. FRAME-CHANNEL MODEL

The frame-channel model is a paradigm which allows to formally define the structure of a large software product and, thus, manage the process of its development [5].

Within this model, the internal structure of a software system is perceived as a frame structure which includes:

- a number of so-called frames;
- and a number of channels ,which are functional relationships ("links") between frames.

A certain frame can be defined in the form of either a basic procedure or a frame structure. Note the recursive definition which allows to apply the same model on different levels of abstraction.

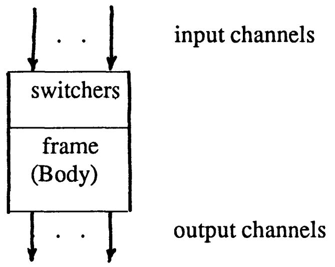
In analogy, channels can be seen as an unified approach to the interface between functional parts of a software product.

A frame includes a number of *switchers* which are special logical conditions, and a body which is either a basic procedure or a frame structure.

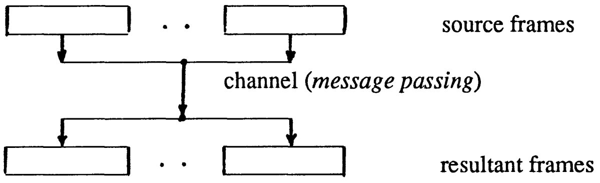
The main action which can be applied to a frame is to activate it. When a certain frame is activated, body is evaluated i.e. analysed or interpreted. If the body is a basic procedure, then this procedure is executed. If the body is another frame structure, then the activation is recursively applied to this structure.

One channel can connect an arbitrary number of frames. More precisely, each frame can be connected to a number of so-called input channels, and to a number of output channels. In analogy, some frames can be defined as sources of a certain channel. and some - as results.

For instance,



and from the "channel´s" point of view:



A channel can be also activated. A certain channel is activated as a result of the activation of one of its source frames.

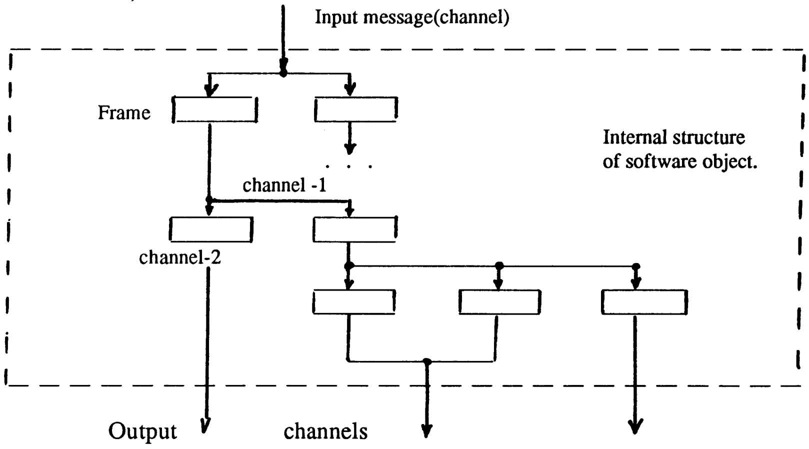
At this point, we can describe the frame activation process in more detail.

Once a certain channel is activated, it contains a number of so-called messages which are available to resultant frames. The term "message" is to be understood as a number of words. For instance, (READ FILE), (25 16) (1) and so on. Such messages are called mails during our discussion. Thus, an activated channel includes 0, 1 or more mails which are available to resultant frames. Resultant frames have a fixed order within a certain channel. Hence, we may use terms "first", "last" and "next" resultant frame. In order to activate a concrete resultant frame, we scan through the sequential list of resultant frames, and check the value of the corresponding switchers i.e. their logical conditions. If a "current" switcher has the logical value "TRUE", then a frame which includes this switcher is activated. Once a certain frame is activated, the input channel immediately ceases to be activated. Thus, only one resultant frame can be activated.

3. SOFTWARE OBJECTS

In accordance with the frame-channel paradigm, a software system can be seen as a kind of software object. A certain software object implements a concrete algorithm by judging a certain collection of input messages and by generating output messages. In turn, this judging of input messages can be seen as a multi-steps process of activation of "internal" frames and channels which define different reactions (or responses) to certain messages (i.e. algorithms). The term "internal frames and channels" is perceived here as purpose-oriented structure of a particular software object. The judging procedure results in the activation of exactly one output channel which corresponds to the result of the algorithm.

For instance,



In addition to these two active data structure types - frames and channels, the internal representation (or topology) of software objects may also include mail boxes.

Each mail-box has unique name, and contains either a number of concrete messages (mails) or the special code NULL-value. In the latter case the mail box is empty.

Note that a mail box is a passive data structure. That is, messages within a certain mail box can be used or modified at any time by means of the unique name of this mail box, but the mail box cannot modify other messages, activate frames and channels.

Messages within a concrete channel or a concrete mail box have a fixed order. Thus, we can refer to a "first" message, to a "second" message and so on. When a certain action refers to a message by means of a channel name or a mail box name, the "first" message is refered to. If the same action puts a new message into a certain channel or a mail box, the new message becomes the "last" one. Thus, mail boxes act as queues.

4. MESSAGE PROCESSING AND SWITCHERS

In most respects, the activation procedure of a certain frame can be seen as a number of actions which deal with messages. To address a certain message, the message operations use a *reference* to the message.

The form of a reference is:

$$\langle message\rangle = \left\{ \begin{array}{l} \underline{USER} \\ \langle name\rangle\ \underline{CHANNEL} \\ \langle name\rangle\ \underline{BOX} \\ "\langle message\rangle" \end{array} \right\}$$

Thus, a certain reference can address or point to:
- a message on the user´s screen ("USER" option);
- a first message within a certain channel ("CHANNEL" option)
- a first message within a certain mail-box ("BOX" option);
- a concrete message which is defined in the form: "message"

Generally, a switcher is a special *logical function* which takes two messages as parameters, and produces the logical value "TRUE" or "FALSE". In order to distinguish between parameters of a switcher, we call the first message an input message, and the second - a pattern.

$$\begin{tikzcd}
\text{(input message)} \arrow[dr] & & \text{(pattern)} \arrow[dl] \\
& \boxed{\text{switcher}} \arrow[d, Rightarrow, "\text{"TRUE" or "FALSE"}"] & \\
& {} &
\end{tikzcd}$$

Let us also introduce some operations which deal with messages.

The operation

GET(<reference>)

gets the message from the user's screen, from a certain channel or from a certain mail-box. Note that the message taken from either a channel or a mail-box is deleted either from that channel or mail-box. The accepted message is available for further processing using the GET operation under discussion. If the operation is applied to an "empty" mail box or to an "empty" channel, then the result of the comparison

GET(. . .) = NULL_VALUE yields "TRUE".

Thus, the mail box can be seen as a special type of external variable which is assigned either a patricular value or a special code "value is unknown". Such a variable can be used in order to define rather sophisticated algorithms of logical inference.

The operation

$$\underline{\text{SEND}}(\text{<reference>})[\text{TO } (\begin{Bmatrix} \text{<name> } \underline{\text{CHANNEL}} \\ \text{<name> } \underline{\text{BOX}} \end{Bmatrix})]$$

sends the message
- to the user's screen (default option)
- into a certain channel ("CHANNEL" option);
- into a certain mail-box ("BOX" option).

There is the concept of frame type and of an instance of a certain type. Thus, users can build new types on the base of previously defined ones and then apply instances of certain types in concrete software projects. This corresponds to the concept of modularity and reusability.

In our context, the possibility to parametrize (and hence generalize) the structure of a concrete software object is of particular importance.

To accomplish this, an arbitrary number of so-called unresolved references can be used.

More precisely, the definition of a frame type includes the number of parameters, i.e. the number of unresolved references to abstract messages. An unresolved reference is coded in the form: & <name_of_parameter>

Thus, a concrete instance of a software object can defined as an instance of one previously defined frame type or as a certain combination of such instances by means of assigning concrete sources of messages (channels or mail boxes) or particular messages to unresolved references.

The term "combination" implies a number of connections between input and/or output channels of different instances.

It should be especially noted that the discussed methods allow to build new frame types in analogy to building concrete instances of software objects. In other words, the designers are allowed to build new frame types on the basis of a current set of existing types.

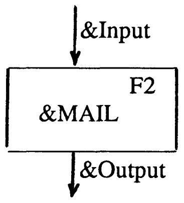
5. THREE LEVELS OF ABSTRACTION

We now come to the essence of this method, i.e. its actual application.

Three levels of detail exist during the definition of a certain software object.

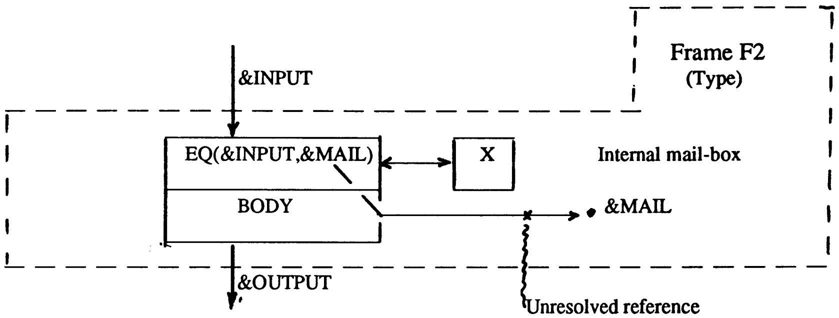
On the first level, the prototype developer (can be the author or a specially appointed person) deals with a number of basic functions, and with the rules of frame type definition. These rules are fairly trivial ones and include special statements to define a switcher, and a special statement to define a body as sequential set of basic functions. It should be noted that all frame types defined on this level contain exactly one input channel i.e. unresolved reference &INPUT, and exactly one output channel i.e. unresolved reference &OUTPUT.

For instance, the frame type F2:



Can be defined by:

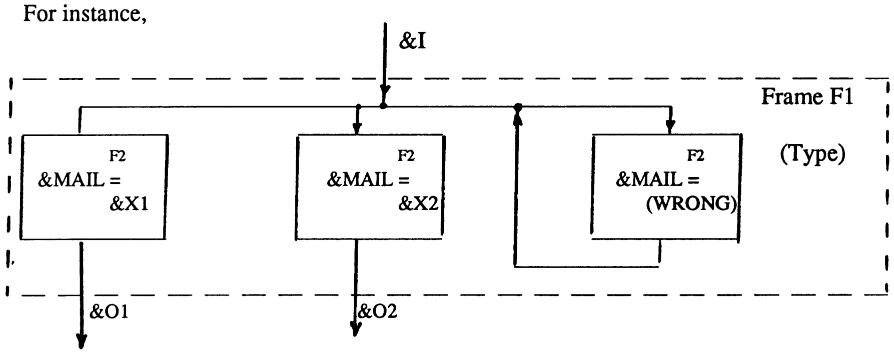
```
DECLARE FRAME F2
    SWITCHER: EQ(&INPUT, &MAIL)
    BODY: SEND(GET(&INPUT)) TO (X)
             GET(&MAIL)
              SEND(X) TO (&MAIL)
              SEND(&INPUT,&OUTPUT)
END FRAME F2;
```



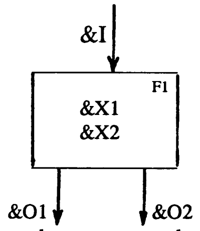
Of course, within the body of a certain frame some additional functions can be used. These additional functions are perceived as a possibility to invoke external procedures or a whole software system during the interpretation of this frame. The results of the execution of such external procedures can be handled by this frame for further processing by means of the concept of messages. Thus, we can say that the concept of frames allows us to encapsulate external procedures or operations within a certain frame arbitrarily.

On the second level, the prototype developers deal only with previously defined types of frames i.e. with the current library of types. They can apply simple rules in order to build new frame types on the basis of the current library of types. The building rules are fully defined by the described concept of internal presentation of software objects. That is, the developer can connect frame types using the channel metaphor, and can assign concrete references to unresolved ones.

For instance,



Now this structure can be seen as a new type of frame having two unresolved references: &X1 and &X2.



On the third level, the user has got a number of frame types which can be interpreted as completed software objects. The subset of such software objects is a user's own software design philosophy. This philosophy can be easily applied by means of setting a prescribed collection of parameters i.e. unresolved references.

It is very important to realize that the number of currently available frame types (the current library of frame type) can be dynamically changed at any time as a result of experience gained.

In other words, when the user starts to apply the software design system, the system can be seen as a prototype of the real system that is needed. This prototype can be successfully applied because it contains a collection of typical software objects, but what is more important, the prototyped number of software objects can be dynamically extended by means of the previously described possibilities to define new frame types.

6. CONCLUSION

There are some properties of our model which are of potential benefit from the point of view of the management of software projects[1,2]:

- the model includes a clear and convenient graphic notation;
- the model can be easily metaphorised for a concrete application[4];
- the model allows the formal verification of a project or of its part[3];
- the model can be applied on the different levels of software specification and implementation;
- the model supports rapid *prototyping*, including the possibility to apply a previous version of the software system within the latest version[2].

The model is mainly oriented towards the specification of so-called database systems, it pays a lot of attention to compatibility between different data models, between different data sub-languages and/or conventional programming languages.

The model was successfully applied in the number of rather big software projects. For instance, a working prototype of database system applied to control manufacturing activity of one of the biggest factories of St.Petersburg (around 15000 employees), was developed and installed during 3 months. Then, the system was applied as developing prototype for the period of 5 years. Now the system includes about 100 working versions. The most attractive feature of this approach is that the end-users were actually involved in the process of system development. They feel themselves as authors of this system and the painful transformation of end-users' needs to software was considerably simplified.

The same approach was applied during the development of a hyper-media system at Technical University of Graz. This approach permitted the usage of different types of end-user interfaces and dynamical assessment of their comparative efficiency from the user's point of view.

REFERENCES:

1. P. Bruce, S. M. Pederson: The software development project, Wiley-Interscience, NY, USA (1982), pp. 210
2. C. Choppy, S. Kaplan: Mixing abstract and concrete modules: Specification, Development and Prototyping In :12th International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA (1990), pp. 173-185
3. P. Freeman: Strategic directions in Software Engineering: Past, Present and Future, in: Ritter G.X. (Ed.), International processing 89. Proceedings of the IFIP 11th World Computer Congress, pp. 205-210, North-Holland publishing Company 1989.
4. R.H. Thayer: Software Engineering project management: A Top-Down View, in E. Nahouraii et. al. (Eds.): Software Engineering Management, pp. 230-235, IEEE Computer Society Press, Washington, DC, USA (1988).
5. H. Maurer, N. Scherbakov: The HM-Data Model; IIG Report Graz (1992).

Design Environment and Technologies Applied within the AXE 10 Software Design Process

Sead Kotlo

Ericsson Technika Kft., Hungary
1108 Budapest, Venyige u. 3

Abstract. Software design process for the digital switching system AXE 10, the main Ericsson's product in the field of digital switching as regards the public telecommunication, is divided in a number of subprocesses/phases. Further on, each of them includes a number of activities logically related to each other.
In order to make possible and successful cooperation of a number of dislocated design centres being involved in huge or medium size development projects going on within the company, to provide for all of them the same design conditions, to ensure the quality required by international standards as well as to increase the productivity, a special design environment, methods and tools have been developed by Ericsson supporting different phases within the AXE 10 software design process.
On the example of the Software Design Centre built up at Ericsson Technika Kft. in Budapest/Hungary, enabling actually performance of a remote software development, the following aspects regarding design of AXE 10 software products will be discussed :

- connectivity environment which makes possible a software development using resources on geographically distant locations;
- well defined development methodology, supported both by standards and design and implementation tools, providing conditions for several groups of mutually unknown people with different backgrounds and experiences to work and act as a team;
- comprehensive testing methods/tools which permit the extensive verification of the software remotely from, or even without, the hardware to be controlled by it.

Integration of Object-Oriented Software Development and Prototyping: Approaches and Consequences

Wolfgang Pree
Department of Computer Science, Washington University
One Brookings Drive, St. Louis, Missouri 63130, U.S.A.
wolfgang@amadeus.wustl.edu

C. Doppler Laboratory for Software Engineering
Johannes Kepler University of Linz, Austria

Abstract. Although object-oriented application frameworks like MacApp [13], AppKit [8] and ET++ [12] substantially ease the building of graphic, direct-manipulation user interfaces, the level of abstraction is considered to be too low to support prototyping such interfaces in a comfortable way. Thus we implemented a user interface prototyping tool based on an object-oriented application framework.

The most important part of a software prototype is its dynamic behavior. On the basis of the tool mentioned above we discuss several ways in which means of adding dynamic behavior to a user interface prototype can be smoothly combined in one tool, in particular combining conventional and object-oriented software. Finally, we categorize user interface prototyping tools available today according to the concepts they offer for dynamic behavior specification.

Keywords: graphic direct-manipulation user interfaces, prototyping, object-oriented programming, application frameworks, multi-paradigm systems, C++

INTRODUCTION

We presuppose that the reader is familiar with object-oriented concepts (independent of a specific language): encapsulation, data abstraction, inheritance, polymorphism and dynamic binding, as well as with principles of graphic user interface application frameworks like MacApp, AppKit and ET++.

Such user interface frameworks offer several advantages: User interface look–and–feel standards are "wired" into the framework components. Furthermore, experience has proven that writing a complex application based on an application framework can result in a reduction in source code size of 80% and more compared to software written with the support of conventionally implemented libraries.

Apart from this enormous code reduction, application frameworks have other important benefits: the abstraction level is raised, and a standardization is achieved in terms of both the user interface and the code structure. However, the abstraction level of an application framework is considered to be too low to support prototyping in a comfortable way. Implementing applications with a framework absolutely requires specialized programming ability (especially in object-oriented programming). Furthermore, the programmer must become familiar with the particular application framework—a time investment that cannot be neglected.

This fact is contrary to the philosophy of prototyping. Therefore we implemented DICE[1] [9, 10] (**D**ynamic **I**nterface **C**reation **E**nvironment) for/with the application framework ET++ in order to extend this tool in the direction of prototyping. The subsequent section describes several ways to specify dynamic behavior as offered by DICE. What sets DICE apart from other available prototyping tools is that it elegantly combines commonly used concepts to add dynamic behavior to a prototype. Furthermore, due to its object-oriented implementation DICE's specification component is extensible in a straightforward fashion.

We implemented DICE with the application framework ET++ for the following reasons: Compared to other available application frameworks, ET++ was the cleanest object-oriented implementation, based on a small set of

[1] This project was supported by Siemens AG Munich

basic mechanisms. ET++ provides a homogenous object-oriented class library that integrates user interface building blocks, basic data structures, and high level application components. ET++ was implemented in C++ and runs under UNIX and either SunWindows, NeWS, or the X11 window system. The design and implementation of ET++ is described in detail in [4, 11, 12].

ADDING FUNCTIONALITY TO A DICE PROTOTYPE

Prototyping is a paradigm that is well established in research and practice for enhancing the Software Life Cycle and improving software quality. There are various publications discussing definitions of prototyping in depth (e.g., [2, 3, 9]). User Interface Prototyping in particular is important for the development of applications that have graphic direct-manipulation user interfaces by providing better requirement definitions. Prototyping this kind of user interfaces with proper tools can significantly reduce the implementation effort (especially if the prototype can be enhanced to the final product).

It is not enough to just describe screen layouts, since the most important aspect of a user interface prototype is its dynamic behavior. In order to support evolutionary prototyping it should be possible to portray the dynamic behavior of a system and at the same time to enhance the prototype to an accomplished application. For this purpose most tools available today provide interfaces to procedural languages or some kind of an integrated procedural language.

DICE supports the graphic specification of the (static) user interface layout similar to other available tools: User interface elements offered in a palette (e.g., action button, labeled radio/toggle button, editable text field, non-editable text field, menu, text subwindow—a subwindow containing a full-fledged text-editor, list subwindow—a subwindow containing a list of selectable text items) are placed into windows simply by dragging them from a palette to the appropriate window. Attributes of interface elements (like the text displayed inside an action button) are defined in dialog boxes. For example, Figure 1 shows the attribute specification of an action button labeled "Stop".

In order to enhance a prototype's functionality DICE offers three possibilities:

- Without programming: Interface elements communicate with one another by sending *predefined messages*.
- With conventional or object-oriented programming: A protocol was developed that allows the prototype to be connected with other *UNIX processes* using one of UNIX's Interprocess Communication mechanisms.
- With object-oriented programming: *Subclasses of ET++ classes* can be generated. Application-specific behavior is added in subclasses of the generated classes.

DICE either operates in a specification mode or a test mode. DICE lets the user transform the specification of a prototype (its static and dynamic behavior) into an operational one within a neglectable amount of time (a fraction of a second on a SUN Sparc Station 1+).

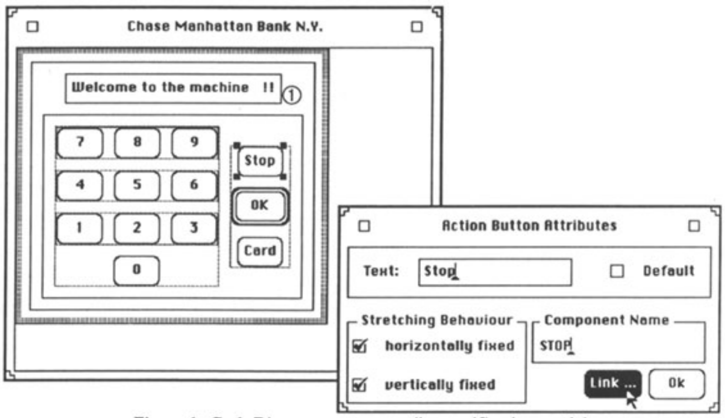


Figure 1: Cash Dispenser prototype (in specification mode)

Predefined Messages

Each user interface element has certain messages assigned that it "understands": For instance, the messages "Open" and "Close" are assigned to a window. All other interface elements understand at least "Enable" and "Disable". In addition, text subwindows, non-editable text fields and editable text fields change their text if they receive a "SetText(...)" message. A list subwindow switches its list if it receives a "SetList(...)" message. Labeled radio and toggle buttons alter their state depending on the parameter value of a "SetState(...)" message.

DICE realizes *state transitions* (in finite automata terminology) in the following way: From each element that can be activated (buttons and menu items), any number of messages to other elements can be specified by means of DICE's Message Editor (see below). If the prototype is tested (i.e., the prototype specification is transformed into an operational prototype) and an interface element is activated in the test mode, the messages specified for that element are sent to their receivers. They effect the corresponding change(s) (=state transition(s)) in the user interface. Thus rudimentary dynamics are realized without programming effort.

Let us take a simple cash dispenser prototype (see Figure 1) as an example. We want the display (① in Figure 1) to show the text "Oops—Stop Button Pressed" when the button labeled "Stop" is pressed. To specify this functionality, one presses the "Link..." button in the attribute sheet (= the dialog box where attributes of the selected user interface element can be edited) of the "Stop" button (see Figure 1). (We assume that the component name of the display field is "Display" and that the button labeled "Stop" has the component name "STOP".) By means of DICE's Message Editor (see Figure 2), the desired dynamic behavior can then be defined for the "Stop" button (i.e., that the message "SetText(...)" is to be sent to the non-editable text field "Display" when the "STOP" button is pressed—the button with the component name "STOP" as its sender (see ① in Figure 2)). After the button "Set Up Link" of the Message Editor (see Figure 2) is pressed the appropriate text string has to be provided as parameter of the message "SetText(...)" by means of a text editor.

The left list ("Target Objects") in the Message Editor displays component names of already existing user interface elements. After a component name is selected in the left list, all messages that are understood by the selected user interface element are displayed in the list "Possible Messages". The right list of already defined messages shows message names together with the component names of their receivers (in our example the message "Disable", which is to be sent to the button with the component name "OkButton", is already defined, the button with the component name "STOP" being the sender). After the "Set Up Link" button is pressed as demonstrated in Figure 2 and the appropriate text string is specified, the message "SetText(...)" (to be sent to the component named "Display") will be added to the list of already defined messages.

Connection of a Prototype with Other UNIX Processes

Algorithmic components of a DICE prototype can be implemented in any formalism and communicate with the user interface prototype specified with DICE by means of a simple protocol that is described below. The integration requires *no code generation* for the user interface part and thus no compile/link/go cycles. An arbitrary number of components implemented in different formalisms can be connected with a user interface prototype that is specified and tested within DICE.

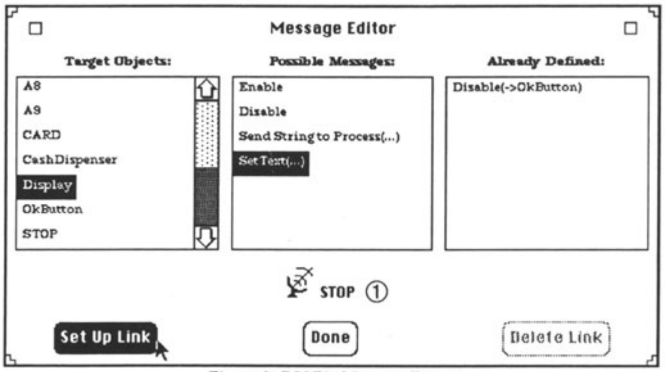


Figure 2: DICE's Message Editor

Communication Concept: Since DICE is implemented on UNIX systems, the UNIX Interprocess Communication mechanisms (e.g., sockets, shared memory) are used for interprocess communication of independent processes (see Figure 3). The interface specified with DICE and the process(es) interacting with the interface form a UIMS (User Interface Management System) with mixed control [1, 5]. This means that an application's "work" is accomplished by various loosely coupled parts of a software system. In case of DICE a DICE user interface prototype forms all visible parts of the user interface and maybe some basic functionality specified by means of predefined messages. Other functionality may be spread over several system parts that are coupled with the user interface by a simple protocol as described below.

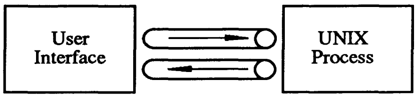


Figure 3: Connection between a user interface prototype and an arbitrary process

Communication Protocol

We illustrate this protocol as far as it is necessary to understand DICE's interprocess communication concept.

User Interface Prototype -> Connected Process: If a user interface element of a protoype is activated in DICE's test mode (activatable user interface elements are all kinds of buttons, text items in a list subwindow, and menu items), an element identifier and its value are sent to the connected process(es) in the following format: identifier=value The identifier is usually the component name of the activated element. If a menu item is selected, the identifier is the component name of the user interface element the menu is part of (e.g., a list subwindow) concatenated with a dot (".") and the text of the selected menu item. If a text item in a list subwindow is selected, the identifier consists of the component name of the list subwindow concatenated with a dot (".") and the text of of the selected text item.

Activated action buttons, menu items, and text items in list subwindows always send TRUE as their value. Labeled radio and toggle buttons send either TRUE or FALSE as value (depending on their state).

Connected Process -> User Interface Prototype: A connected process can ask for the value of an interface element by sending identifier ? to the user interface prototype. If a user interface element exists that matches identifier, it "answers" as if it had been activated using the format described above. Values of user interface elements can be changed from the connected process by sending identifier=value to it. This allows some special changes in the user interface, too: windows, for example, can be opened or closed using the value OPEN or CLOSE. A list subwindow accepts EMPTY as value (to empty the list). A text string sent to a list subwindow as value means that this text is to be appended as a list item in the correspondent list subwindow.

The communication protocol is the precondition that a user interface developed with DICE can be connected with any conventional or object-oriented software system. E.g., the functionality of the cash dispenser specified in Figure 1 was implemented in C. (It could also be implemented in Cobol or Fortran or what else is available.) Necessary modifications or enhancements of the functionality are implemented in a C program. Immediately after compiling and starting this program, the modified functionality can be tested together with the user interface prototype (in test mode) without restarting DICE, even without switching from the test mode to the specification mode and back to the test mode.

The development of software systems that are to be connected with the interface prototype can be supported by available methods and tools. Pomberger [9], for instance, describes a tool that allows prototyping-oriented incremental software development. Due to DICE's Communication Protocol it was easy to combine this tool with DICE.

On the other hand, it is, of course, possible to connect a user interface prototype specified and tested in DICE with object-oriented systems developed by means of any domain-specific class libraries that might be available.

Generating Application Framework Subclasses

DICE simulates the static and dynamic behavior of a specified prototype when that prototype is tested. Thus no code generation and no compile/link/go cycles are necessary for testing. In order to enhance the prototype by means of the application framework ET++, DICE allows the creation of subclasses of ET++ classes. The compilation of the generated classes results in an application which works exactly like the specified prototype.

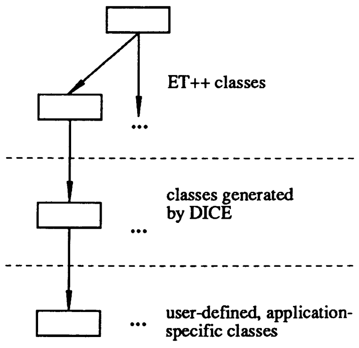

ET++ classes
classes generated by DICE
user-defined, application-specific classes

Figure 4: Code generation concept

The generated classes need not (and should not) be changed when further functionality is added in the sense of evolutionary prototyping. Additional functionality can be implemented in subclasses of the generated classes by overriding or extending the corresponding dynamically bound methods (see Figure 4).

Let us look at the cash dispenser interface (Figure 1) again: When the "Ok" button in the window titled "Chase Manhattan Bank N.Y." is pressed, the correctness of the displayed amount should be checked. This functionality could not be provided by DICE's prototyping facilities. Therefore we would like to add special code in order to implement this behavior.

DICE uses the component names of user interface elements in the generated code. Component names can be defined for each user interface element in the corresponding attribute sheet (see, for example, Figure 1: the component name of the button labeled "Stop" is "STOP"). We assume that the button labeled "Ok" has the component name "OkButton" and that the window titled "Chase Manhattan Bank N.Y." has the component name "CashDispenser".

So DICE generates a class CashDispenser. DICE reuses behavior implemented in the ET++ class Document by generating CashDispenser as subclass of it. Document, for example, manages a window in which the appropriate contents is displayed. Furthermore, the ET++ class Document has a dynamically bound method Control which is called each time a user interface element is activated inside a window associated with a Document object. Thus the method Control is used in the generated code to implement the behavior of user interface elements specified by means of predefined messages. Since no behavior was specified by means of predefined messages for the button with the component name "OkButton" the code generated by DICE is the following:

```
class CashDispenser: public Document {
    ...
    void Control(int id) {
        ...
        case OkButton:
            break;  // no action
        ...
    }
};
```

In order to check the correctness of the amount, we implement a class ExtCashDispenser (stands for "Extended Cash Dispenser"). The presented code fragment is simplified in order to stress the essential idea of adding functionality in subclasses of generated classes.

```
class ExtCashDispenser: public CashDispenser {
    ...
    void Control(int id) {
        ...
```

```
                        case OkButton:
                                int disp=Display->Val();
                                if (AmountOk(disp))
                                        ...
                                break;
                        ...
                        CashDispenser::Control(id);
                }
        };
```

To sum up, this kind of code generation separates changes of the user interface from hand-coded functionality as far as possible. For instance, if the user interface layout is changed, code (i.e., ET++ subclasses) must be generated again. The user-defined classes that have been derived from the originally generated classes are not concerned. Changes of these classes only become necessary if interface elements are removed (which would result in extrenous code) or switched between windows of the prototype.

CATEGORIZATION OF USER INTERFACE PROTOTYPING TOOLS

Prototypes built with DICE (i.e., prototypes that are executable within DICE in the test mode as well as ET++ applications generated from the prototype specification) are *finite* automata consisting of a finite number of states (the static layout of user interfaces) and state transitions (the dynamic behavior). We call this basic structure of a prototype its *application model.*

Applications built with state-of-the-art application frameworks are typically *infinite* automata: states and state transitions are described in classes from which an arbitrary (and theoretically unlimited) number of instances can be created. So the number of states and state transitions is not limited. For instance, a text editor application may have an arbitrary number of documents (= windows) in which text can be edited. Though the windows of one such text editor can be specified with DICE (e.g., by means of the text subwindow), the prototype as well as the eventually generated application have only the specified windows—the text editor application is not instantiable.

Thus the underlying application model of DICE prototypes and the application model of typical applications that are built on top of state-of-the-art application frameworks differ considerably. Since DICE's application model is a subset of the application model of a modern user interface framework, it is easy to generate subclasses of such a framework (ET++ in case of DICE), so that the transformation of the generated classes into an executable program results in an application which works exactly like the prototype specified with DICE. In order to project DICE's application model to an application framework, the generated classes have to eliminate many mechanisms provided by the framework classes: in ET++, for example, the complete document management done in class Application becomes superfluous.

Abstraction Level of Dynamic Behavior Specification

In general, the abstraction level of the specification of dynamic behavior determines whether the application model of the specified prototype can correspond to the application model of typical framework applications. User interface prototyping tools known today that allow the specification of dynamic behavior on an abstraction level higher than that of a programming language rely on the concept that applications with graphic, direct-manipulation user interfaces are finite automata—an application model that does not match that of modern application frameworks. The main reason for this fact is that the application model represented by finite automata can be specified with graphic editors in an easy and intuitive way.

The more sophisticated application models of modern application frameworks would require other graphic-oriented specification techniques. Such *visual programming* editors have not reached the maturity to allow use in this context [7]. NeXT Interface Builder [8] supports the building of applications that adhere to the application model of a modern application framework (AppKit) at the cost of specifying dynamic behavior on the programming language level (At first glance, the possibility offered by NeXT Interface Builder seems to be identical with predefined messages in DICE, but there is one crucial difference: In NeXT Interface Builder message connections between objects (called *sender* and *target* in this context) are method calls of the target object issued by an activated sender object. The messages that objects "understand" must be implemented in classes.)

Supported Application Area

Another important issue of user interface prototyping has to be taken into consideration, too: Many commercial data processing applications heavily rely on database management systems. Evolutionary prototyping of applications belonging to this category could benefit a lot if the user interface prototyping tool or the generated executable prototypes could be integrated with a (relational or object-oriented) database management system. Tools that allow user interface prototyping and the development of a database management system are often called fourth generation systems [6]. Though the term fourth generation system has not been standardized yet, we give a possible definition of such a system: fourth generation systems are built around a database management system and enable the developer to specifiy/implement not only the user interface layout but also data models, reports and consistency rules on a high abstraction level. They typically provide standard search and sort facilities and procedural languages for implementing dynamic behavior.

If a user interface prototyping tool is used within a fourth generation system the kind of code generation (based on a conventionally implemented toolkit or an application framework) is almost irrelevant because the user interface of commercial data processing applications (often called *information systems*) can be completely specified with available user interface prototyping tools in most cases: text fields, buttons, lists and text editors are sufficient for this application category. The system developer usually does not need (user interface) application framework classes in order to enhance a prototype. Moreover, the finite application model of almost all user interface prototyping tools available today meets the requirements of information systems: it is, for example, not desirable to instantiate an arbitrary number of input masks that are used to enter data into a database.

SUMMARIZING REMARKS

Depending on the level of abstraction of the specification of dynamic behavior we can divide high-level user interface prototyping tools into two categories: tools which support prototyping of information systems and tools that help to reduce the implementation effort if an application framework is used. All tools which are based on the finite automata application model are especially suited for prototyping information systems and thus belong to the first category. Their application model is only a subset of the infinite automata application model of user interface application frameworks. Thus the development of software systems with the infinite application model of user interface application frameworks is not supported.

An example of a tool that belongs to the second category is NeXT Interface Builder. Research (especially in visual programming) is necessary in order to allow the specification of dynamic behavior on an abstraction level higher than that of a programming language and to retain the application model of a state-of-the-art user interface application framework.

REFERENCES

1. Betts B., et al.: Goals and Objectives for User Interface Software; in: Computer Graphics, Vol. 21, No. 2, April 1987.
2. Budde R. et al.: Approaches to Prototyping; in Proceedings of the Working Conference on Prototyping, Namur, October '83, Springer 1984.
3. Floyd, C.: A Systematic Look at Prototyping; in: Approaches to Prototyping, Springer, 1984.
4. Gamma E., Weinand A., Marty R.: Integration of a Programming Environment into ET++: A Case Study; Proceedings of the 1989 ECOOP, July 1989.
5. Hayes P.J., Szekely P.A., Lerner R.A.: Design Alternatives for User Interface Management Systems Based on Experience with COUSIN; in: Human Factors in Computing Systems: CHI'85 Conference Proceedings, Boston, Mass., April 1985.
6. Holloway S.: Background to Forth Generation; in Fourth Generation Languages and Application Generators, The Technical Press, 1986.
7. Myers B.: User-Interface Tools: Introduction and Survey; IEEE Software, 6(1), January 1989.
8. NeXT, Inc.: 1.0 Technical Documentation: Concepts; NeXT, Inc., Redwood City, CA, 1990.
9. Pomberger G., Bischofberger W., Kolb D., Pree W., Schlemm H.: Prototyping-Oriented Software Development, Concepts and Tools; in Structured Programming Vol.12, No.1, Springer 1991.
10. Pree W.: Object-Oriented Versus Conventional Construction of User Interface Prototyping Tools; PhD thesis, Johannes Kepler University of Linz, 1991.
11. Weinand A., Gamma E., Marty R.: ET++ - An Object-Oriented Application Framework in C++; OOPSLA'88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, 1988.
12. Weinand A., Gamma E., Marty R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework; in Structured Programming Vol.10, No.2, Springer 1989.
13. Wilson D.A., Rosenstein L.S., Shafer D.: Programming with MacApp; Addison-Wesley, 1990.

Trademarks:

MacApp is a trademark of Apple Computer Inc.

App Kit is a trademark of NeXT Inc.

SunWindows and NeWS are trademarks of Sun Microsystems.

UNIX and C++ are trademarks of AT&T.

Object-Oriented Analysis and Design — A Case Study

Wolfgang Eder[1] Gerti Kappel[2] Jan Overbeck[1] Michael Schrefl[3]

[1] Inst. f. Informationssysteme Technische Universität Wien
[2] Inst. f. Statistik u.Informatik Universität Wien
[3] Inst. f. Wirtschaftsinformatik Universität Linz

Abstract

Several methods for object-oriented system development have been published by the scientific community. Recently, industrial software developers are also attracted by the object-oriented paradigm and consider switching from structured techniques to an object-oriented approach to system development. A question commonly asked by industry is, how both approaches compare on industrial applications. To investigate on this issue, a case study has been undertaken.

A configuration management system, which had originally been developed following the structured analysis and design approach, was modelled using an object-oriented modeling technique[1]. The main lessons learned are the following:

1. The effort put into the analysis was considered higher in contrast to our experience in non object-oriented projects. During design and implementation, however, the analysis effort proved useful as there was a smooth transition from analysis to design and from design to implementation.

2. The object classes of the design could be easily mapped into object classes of the implementation using the MacApp application framework. The application framework proved highly reusable and easily customizable for implementing the case study.

3. The object-oriented model proved stable against major changes in the system's requirements. It is believed that the object-oriented approach is suitable for applications with evolving reqirements.

4. Some "objects" of the object-oriented solution were already present in the original (non object-oriented) solution in terms of a set of procedures manipulating the same data structure. As object-orientation was not known to the original project team, some of the semantics of these "objects" had to be handcoded while others were not present at all. Thus the benefits of object-oriented development could be fully exploited in the object-oriented solution.

The presentation gives an overview of the case study introducing the object-oriented analysis model, the object-oriented design model, and selected parts of the object-oriented implementation. The above mentioned experiences will be discussed on behalf of these models.

[1] J.Rumbaugh, etal. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1990.

SOFTWARE ENGINEERING EDUCATION

Chair: D. Sima

Small Is Beautiful, Isn't It?

Contradictions in Software Engineering Education

Péter Hanák, Zoltán László
Hanak@inf.bme.hu, h4245las@ella.hu
Faculty of Electrical Engineering and Informatics,
Technical University Budapest
H–1521 Budapest, Hungary

Abstract

Staff and students have recently faced the latest reforms in informatics education at the Section of Technical Informatics, TU Budapest. In this paper, partly *post factum,* we pose ourselves such questions as

- which topics of informatics, and in particular of software engineering, ought to be taught,
- how these topics should and could be taught,
- who and how will present these topics,
- how to convince students of their importance and usefulness (i.e. how to motivate students),
- what jobs will be open for software engineers in Hungary or in other countries,
- what are the skills a university student should acquire,
- what is the optimal ratio of theory and practice, etc.

We do not promise the answers to these and many similar questions. However, we do try to reveal contradictions in software engineering education in a small country, partly by comparing our problems to those discussed in the literature, and partly by presenting our experiences and approaches at TU Budapest — in the hope that it will trigger vivid discussions at the conference on Shifting Paradigms in Software Engineering in Klagenfurt.

1 Introduction

Informatics (as it is called in Continental Europe) or computer/computing science (as the American/British traditionally call it) is related to mathematics, engineering and management. Depending on the School where it is taught one of its aspects is emphasized. Nonetheless, in the curriculum a proper balance is desirable. In Section 6 the newest curriculum of *technical informatics* at TU Budapest, centred around software engineering (SE), is presented, and the sequence of courses that determine its SE content is discussed, with an eye kept on this balance.

Engineering is defined in [10] as 'creating cost-effective solutions to practical problems by applying scientific knowledge to building things in the service of mankind'. SE is claimed in the same paper to be more 'a statement of aspiration' than a 'description of accomplishment' because of the 'lack of widespread routine application of scientific knowledge to a wide range of practical design tasks'.

It's not at all easy to determine the necessary content of a degree programme in informatics and in particular in SE. Beside personal ambitions, local expertise and available infrastructure the *immaturity of the subject* causes most difficulties: the 40-year history of computing education is the history of *permanent shifting,* triggered by technological changes, from technical peculiarities to higher-level concepts and solutions.

By now, programming-in-the-small is more or less well understood, and the educational community has the necessary skill in teaching the widely accepted, sound principles of algorithmic and data abstraction. On the other hand, programming-in-the-large is far from being established: it is a field of discussions and beliefs. After all, small is beautiful, isn't it?

While we still strive for theories, methodologies and tools necessary to create huge but correct and secure software systems one wonders if the well-known educational difficulties are only due to this lack of knowledge, or if they are of intrinsic character. That is, we have to face the problem of teaching complex things while being restricted in resources, time and prerequisite knowledge. Although this situation is not unusual in the engineering education we again have to 'reinvent the wheel' in the SE field — as it occurred e.g. with structured programming.

Then, we should ask ourselves whether the idealized practice of system design can be abstracted and taught at all, or only some well-sounding principles and slogans can be collected and presented in the classroom. The authors, graduated in electrical engineering and gained practice in the design of medium-size hardware/software systems, have the impression that systematically only small-scale design of digital systems is and can be taught. It would be interesting to study other, more mature and less dramatically changing fields of engineering (e.g. civil or mechanical engineering) in order to reveal the similarities and the differences, and to see whether there are any general methods applicable also to SE.

Education is usually told to be *the art of concealment,* but it could well be called *the art of selection,* i.e. of choosing things worth to know. Very frequently, we teach nonessential and unimportant topics just because they are well known and easy to teach.

The dilemma was also admitted during a workshop at Brown University in 1990 [3]: many invited speakers questioned even the widely accepted contents of introductory programming courses describing them as nonrelevant. Further, 'it was a humbling experience to see that after twenty-five years of teaching computing in major universities, we still don't know how to do it' [4].

In the first paragraph we used the term 'technical informatics'. In Hungary, it has been used since 1991 meaning informatics education at universities and colleges of *technology,* in contrast to 'theoretical informatics' being or to be taught at universities of natural sciences, 'econometrics' at universities of economy, and 'library informatics' at universities of liberal arts, etc. In the courses of technical informatics 'much more attention is given to the hardware aspects of information systems than anywhere else' [8], and to its engineering character.

2 Contradictions and other problems

Here is a list of the most striking contradictions that deeply affect engineering education in general, and SE education in particular. We suppose they are understandable without further explanation.

- Things easy to teach and grade vs. things essential to know;
- things motivated by fashion ('attractive knowledge') vs. things worth to know ('painful knowledge');
- theoretical background vs. practical skills;
- concrete and specific vs. abstract and universal knowledge;
- student's self-motivation vs. teacher's pressure;
- real things (languages, tools, etc.) with practical value vs. educational versions with didactic value;
- exact facts and algorithms vs. philosophy and descriptive methodologies;
- stand-alone knowledge vs. system-wide or embedded knowledge;
- bottom-up approach in education vs. top-down approach in application.

Below is another list of important problems related to informatics and SE education; more problems and detailed explanations can be found, for example, in [3] and [11].

- Small classroom problems does not strive for methodologies and development tools; what's more, methodologies and tools cause additional problems;
- larger classroom problems are usually too specific and need detailed background knowledge of some other field;
- larger problems, to be solved as projects, imply management problems and increased trainer's effort;
- most phases of a development project (writing project plans, specifications, detailed designs, documentation, test plans, manuals, etc.) are much more boring than simply coding and debugging programs;
- there are plenty of bad patterns to follow while good patterns are rare in practice;
- most students lack the experience of unsuccessful projects, therefore they don't feel the need for formals methods and systematic approaches;
- engineering students, accustomed to facts and algorithms, are reluctant to descriptive methodologies and philosophy;
- students rarely study programs written by others (be good or poor ones);
- the effort needed to read, check and comment student projects is tremendous;
- because the profession of SE lacks self-confidence, we do not dare to demand thorough and precisely documented designs from the students — in this respect, we should learn from mechanical design and architecture;
- an interesting phenomenon is the following: as soon as we fully understand a problem of programming methodology (structured language constructs, abstract data types, etc.), we usually consider it as an unworthy topic for a university-level course on computing (that's why the content of introductory programming courses changes so frequently);
- development tools and environments are rapidly becoming obsolete (often triggered by the business interests of huge, multinational corporations), requiring additional training efforts from staff members. (E.g. between 1987 and 1991 the following Turbo Pascal versions were used with first-year students in informatics at TU Budapest: 3.0, 4.0, 5.0, 5.5 and 6.0, i.e. a new version each year! Add to them various versions of assemblers, editors, "C" compilers, CASE-tools under MS-DOS, plus other machines, other operating systems, other languages...)

Obviously, most of these problems can only be overcome if we carefully select the topics to be taught. Experience (e.g. [3]) shows that it is far from being simple.

3 Paradigm shifting

The word 'paradigm' is relatively new to informatics, and made as fast a carrier as the terms 'modular', 'structured', 'software engineering', 'object oriented', etc. did earlier. (Eventually, fashion can be traced on titles of professional books that often appear with the same traditional content but under attractive titles.) One feels temptation to define what 'paradigm' might mean but a definition would hardly contribute to its better understanding (like with the other terms mentioned above). We need time to get familiar with this new term and its possible meanings.

Paradigms are changing because so far no paradigm has solved the problem of the mass-production of software — and because *only new promises bring money.*

If we forget this last remark for a while, and try to reveal other driving forces of paradigm shifting, the title of this conference also assumes, then we have to admit that, above all, it is *the technological development* that made these changes possible and inevitable. Indeed, the development of technology has made those tools available and those methodologies implementable *that had already been known earlier.*

Table 1 below enumerates some concepts, the approximate dates when they first appeared, and when they became or will presumably become generally accepted, implemented and productive. (Of course, dates are rough estimates.)

Table 1: Some concepts

Concept	Appearance	Acceptance
Structured Programming (SP)	1965	1985
Structured Analysis (SA)	1970	1990
Object-Oriented Programming (OOP)	1967	1990
Functional Programming (FP)	1950	1990
Logic Programming (LP)	1960	1988
Artificial Intelligence (AI)	1960	1990
Program Proving (PP)	1960	????
Formal Specification (FS)	1960	1995
Software Metrics (SM)	1965	????
Software Reuse (SR)	1980	1995

We should admit that general acceptance is by no means influenced by scientific value or theoretical superiority: implementation and technology are that really count. For example, take the case BASIC we know too well: computing scientists, methodologists and didacticians fought bitterly over two decades against BASIC whose popularity had even been fortified by first generation PC's — with almost no result. Then, faster than it arrived, it has been blown away by the IBM PC/AT boom of recent years. Those who like similar debates should draw the moral of the story themselves.

[12] describes another convincing example: although the term *algebra* had been known and *paper-mills* had been used in Europe since around 1200,

> 'it was not until the 16th century that algebra was really accepted as a formal method. In that century, there was the struggle between the Abacists and the Algorithmists, between the concrete calculation by means of *calculi,* little stones or coins, and the abstract calculation on paper and by more and more formal rules. The Algorithmists won, because *suddenly paper could be produced at a much lower price* — which shows once more how much we depend on technology, even in such mental aspects.'

In the following section we try to argue for a necessary change in SE education.

4 Approaches in Software Engineering Education

Software engineering, like many technical terms, has various interpretations. It's no surprise that there are different approaches to SE education.

In the narrow sense, it means a single course or a few related courses (for undergraduates) trying to cover the whole process of software development, usually based on the waterfall-model, including principles, methodologies, tools and techniques, see e.g. [11].

In a broader sense, however, we may extend the concept of SE to include other courses related to software development: courses on introduction to programming, algorithms and data structures, program design and programming technology, object-oriented programming, mathematical logic and logic programming, functional programming, formal specification and program proving, compiler construction, operating systems, database technology, software quality assurance, etc.

Since program development is a constructive discipline it is hard to imagine a series of pure lectures without related project activities. [11] classifies various models of SE courses and project styles many of them we have also applied. Later, while surveying past and present of informatics education at TU Budapest, we shall summarize our experiences with these models.

Like other disciplines, SE has also been passing through various phases of self-development: empirical, descriptive and formalized.

It is well known that the theory of software development is lagging behind practice; nonetheless, it is more than regrettable that at most universities of sciences and of technology SE courses are only taught as empirical and descriptive subjects.

Formal description and specification

Formalization is going into two directions: graphical and algebraic. In some cases graphical formalization is based on strict mathematical models, and can be manipulated mathematically (e.g. Petri-nets). In most cases, however, this correspondence is missing.

Recent graphical formalization methods, supported by CASE-tools, are mainly used in architectural design, and does not help much in transforming the design into an executable program. The gap is large, and research results do not promise fast solution. Repository-based object-oriented techniques are now claimed to diminish this gap [9]. Graphical formalization methods are gaining popularity since (1) computer graphics is fashionable; (2) they are attractive at first sight and claimed to be 'easy-to-use' (which is rarely true), (3) have a long tradition (e.g. blueprints, flow-charts), (4) have some software support (e.g. Teamwork, SSADM Engineer), and therefore (5) software firms are interested in their dissemination.

On the other hand, algebraic methods, based on sound mathematical theory, pencil and paper — or a text editor, proved their usefulness in the formal specification of small pieces of programs (its supporters like to remark that even huge programs are small depending on the level of abstraction). The authors would also like to believe in the superiority of these techniques that promise — at least in the *far* future — the possibility of automatized transformation from specification to executable code. As for the present, these methods are distinguished by their unambiguity, precision and rigour; virtues that we eagerly need in SE education. Unfortunately, current formal specification methods are limited; their acceptance by students is at least controversial, by colleagues is even worse.

One leading expert, who advocates algebraic methods, writes [5]:

> 'No professional architect, bridge builder or car designer would work with specifications of the same shoddy nature that one finds in software engineering. ...One hears

that software projects are larger and more complex than other classical engineering projects, but that is even more — and not less — reason to be more professional.

...while texts on discrete mathematics for computer science students have a chapter on logic, the material is rarely used in the rest of the text. Hence, the student and the instructor come away with the feeling that the mathematical tools are of academic interest only. They have seen some of the techniques, but lack skill in their use and question their applicability. ...The retort "We know what we want to do, and it's too big a task to formalize" is heard far too often. ...Have you ever heard a physicist say that their problems are too big and complex to be handled by mathematical techniques?

...I am not advocating the formal proof of correctness of all programs. ...In developing a calculational skill, one learns that formalization can lead to crisper and more precise descriptions. One learns that the form of the formalization can itself lend insight into developing a solution. One acquires the urge to clarify and simplify, to seek the right notation in which to express a problem. One acquires a frame of mind that encourages precision and rigor.'

In the ideal case, the rigour of the algebraic formalization and the transparency of the graphical approach should be combined.

Variety or diversity

Unfortunately, there is little consensus about how a standard undergraduate informatics curriculum should be improved. [7] enumerates four contradicting opinions:

1. recent informatics education should be replaced with a traditional program based on standard engineering and the usual topics in mathematics;
2. informatics should be a new form of mathematics that deals with the verification of symbol manipulators;
3. (bookstore shelves give the impression that) informatics is primarily a matter of learning BASIC — or Pascal, if you like;
4. programming, as an engineering discipline, should be based on a minimal number of concepts: recursion, prefix notation and the list structure.

Then, continuing, [7] completely denies the formal specification approach:

'...fundamentalist views ...leave little or no room for incremental system building. In addition, the proof techniques apply to the elementary constructs of a programming language and use a tedious notation that has the flavor of an assembly code. This approach seems to encourage the view that each program must be proven correct as if it were the only one of its kind in the world.'

Reasonings, like this one, show that most of us hear only what we want to hear. Those, who remember the bitter fight between followers and opponents of structured programming, would recognize the only unrefutable argument in a somewhat modified form [5]: 'Calculational techniques deserve to be given a fair chance, especially since nothing else has appeared on the horizon to solve the ills of the profession.'

In summary, not much help can be found in the literature when someone tries to determine the fundamental content of SE education. Usually [3], there is an introductory course on computing (based on Pascal or similar language), another one on algorithms and data structures, and then an undergraduate course on SE based on one out of four popular textbooks [11]. Other courses, if any, rely too heavily on local people, their research interest and faith. As [6] writes, career success in United States universities 'depends very much on ... obtaining outside funding through grants, publishing in journals and teaching — in precisely that order'. He and others blame this practice for ignoring engineering and didactic issues in informatics education: 'Many of those who are most interested in educational issues are at institutions that do not afford faculty much time for scholarly activities. Unfortunately, the reverse is often true as well: too many faculty at research-oriented institutions are not as concerned as they should be with issues in undergraduate education' [1].

5 Local history and experiences

The degree programme in *informatics* at TU Budapest, like at many other universities of technology, grew out of Electrical Engineering rather late, in 1986, as a new curriculum offered for a number of students by teams composed of lecturers of existing departments. The electrical engineering topics still dominated this curriculum, only ca 1/3 of the courses were specific to informatics.

Based on former experiences and in answer to the changing demands of a changing society, a significantly modified (hopefully improved) degree programme in *technical informatics* has replaced the previous one since 1991 (see Section 6 and [8]). It is distinguished by a ca 2:1 ratio in favour of subjects related to informatics. The tendencies are also reflected in the new name of the Faculty: since May 1992 it is called *Faculty of Electrical Engineering and Informatics.*

So far, the history of programming and software engineering education at TU Budapest can be divided into three phases. Below, we try to characterize the SE content of these phases, and draw some lessons if possible. Of course, this summary necessarily contains simplifications — and it reflects how the authors see the happenings now. (The notation *Sx* below means Semester *x* in which the course is taught.)

5.1 Phase I. Electrical engineering (ca 1970–1986)

Computer hardware and programming education started around 1970 at TU Budapest.

Lectures: Introduction to Programming: assembly + ALGOL or FORTRAN, later assembly + Pascal (S1), Digital design, incl. microprocessors (S3–S4), Computer systems, incl. programming issues (S5), Peripherals & interface design (S6–S7), Computer networks (S8).

Labs: Digital measurements (S4–S5).

Projects: Peripheral interface design for microprocessors; individual student projects and thesis works (more and more shifted from hardware to software development).

This somewhat 'overmature' pioneering phase extended over almost two decades. Software development was only marginally treated in the courses while more and more student projects and thesis works dealt with program development, in response to external needs and student motivation. The inflexible curriculum structure, strengthened by the selfish interests of most engineering departments, hindered the inclusion of new subjects for a long time.

Projects were carried out by usually one, sometimes two students under the guidance of a staff member — cooperation in larger teams was not required. The size of the projects was rather small: typically a peripheral interface board and a device driver, or some stand-alone program had to be developed. Measurement labs covered SSI and MSI, later LSI circuits, incl. microprocessors, with some assembly-level programming. The ultimate goal of the education was to produce electrical engineers who were *able to develop* new equipment, devices and systems.

As design methodology, the well-known Karnough-tables, state-transition diagrams and tables, etc. were introduced for small hardware systems. For medium and large systems, because no systematic methods were known, standard MSI and LSI elements, principles and thumb rules, combined with block-diagram techniques, were taught. In case of software nothing better than flow-charts, NS-diagrams and pseudo-languages were used, based on principles of T-D and partly B-U methodology.

5.2 Phase II. Informatics (1986–1991)

Last graduates of this programme will finish their studies in 1995.

Lectures: Programming and problem solving: assembly + Pascal (S1–S2), Principles of program design (S3–S4), Systems programming (S5–S6), Program design (S7), Informatics systems (S7).

Labs: Computing in assembly, Pascal and C (S1–S2–S3).

Projects: Programming projects in small teams (2–4), individual development projects, causally participation in faculty teams, thesis works.

Developers of Curriculum 1986 preferred a small number of comprehensive courses. As a consequence, the course Principles of program design comprised two, more or less independent parts: Theory of algorithms and Formal languages; the course Systems programming consisted of three parts: Operating systems, Computer networks and Databases.

The content of the course Program design, sometimes also called Software engineering — the authors' main field of interest — has slightly changed over the years. It has covered the phases of software life cycle, but not necessarily in the same order and detail: requirements analysis has only been mentioned; Jackson, Jackson–Warnier and DeMarco notations have been introduced as formal techniques of program design, finite automata modelling and SSADM methodology have also been discussed. Testing, reliability, software metrics, quality assurance, maintenance have been covered to some extent. The course has been concluded with a summary of formal specification methods and an introduction to program proving, based on predicate calculus and the guarded command notation.

The content of a related course, Informatics systems, was only vaguely described in Curriculum 1986, and since then it has changed a lot. Based on invited lecturers from external firms, topics of artificial intelligence, expert systems, relational knowledge bases, man-machine communication, and later Prolog and logic programming have been presented. However, as it has never turned into a well-structured course, this title has been left out from Curriculum 1991.

It should not be left unmentioned that the intensive involvement of external lecturers resulted in a number of problems: it proved (once again) that no single course can be given by many people (more than two), and that education must not be based on non-staff members deeply involved in business or management.

The five-year existence of informatics education at TU Budapest has also yielded other useful experiences; let's see a short account.

Programming projects. First-year informatics students had intensive laboratory exercises in semesters 1, 2 and 3 on IBM PCs. In the first half of each semester they acquired some skills in a programming language and environment (assembly, Pascal and C, respectively), and then in the second half they were given larger tasks — toy projects (simulate a programable calculator, design and implement a graphical editor, simulate a three-cabin elevator, etc.) — to be solved in teams of three or four, supervised by staff members. The students were asked to refine requirements, divide the task into subtasks and assign them to team members, design the whole program and split it into modules, design, implement and integrate these modules, and finally complete the user and developer documentations.

Almost everybody worked with enthusiasm and spent uncountably many hours with the project (lecturers of other courses were less enthusiastic...). In some teams, instinctive team leaders emerged capable to coordinate activities of other team members. In most students' life, it was the first time they had to cooperate in teams. Nonetheless, as it turned out later when they took the course on Program design, they had not known much about systematic specification and design methods; therefore they suggested to shift the Program Design course into the early semesters.

Teams consisted of 2 to 4 members; we have never tried to start big (say, 20-member) projects with students as we feared of preparation and management problems. (As a counter-example see [11] describing a successful, real-life campus-project). It should also be mentioned that some supervisors (lab coaches) preferred to give the students one-person tasks. (Small is beautiful, isn't it?) It would be interesting to ask those students if they felt later that they had missed something; or to study their attitude towards programming projects, and look for possible differences.

Lectures. Course titles often did not correspond to their content. Two-semester courses frequently consisted of completely distinct topics. The content of some courses was poorly defined, and in a number of cases there were also 'implementation problems'. Textbooks were almost completely missing. Further, the number of electrical engineering courses was still too high while important topics (e.g. logic, functional programming, compiler construction) were missing from Curriculum 1986.

Labs and projects. The curriculum imposed a high amount of lab and project activities: programming exercises in semesters 1 to 3, individual projects in semesters 4 to 7, pre-thesis and thesis projects in semesters 8 to 10. These activities required very intensive participation, both of students and of staff. In the beginning, insufficient hardware resources caused problems, later the lack of manpower — a new phenomenon at TU Budapest – caused most difficulties. Therefore, we have involved senior students as supervisors. They have the advantage that they know the programming environments much better than we do but they need supervision and guidance themselves — an additional burden on staff.

Within the course on Program design, we applied various project models. At first, student teams were asked to systematically redesign a project they carried out earlier. As it turned out, nobody applied the methods presented in the classes but used the old *ad-hoc* ones. They complained because of the number of pages they had to write. Nevertheless, many colleagues liked the idea since documentations that otherwise would not be completed were finished. One year later no project accompanied the course — we regretted it afterwards. In the next year all student teams were given the same design-without-implementation task (Tangram). In the beginning, most teams worked with enthusiasm but only a few produced a systematic, well-considered design; probably, its main reason was that they had not confronted with implementation problems that would forced them to reconsider and improve the original design.

5.3 Phase III. Technical informatics (1991–)

Many of the above experiences and other considerations were taken into account in the (often controversial) design process of the new curriculum (see Section 6). Very importantly, university authorities have been challenged to react to social and political changes in the country, namely

- the weakening of COCOM-restrictions and the appearance of multinational computer companies in Hungary have slowly been resulting in a better infrastructure;
- the anticipated collapse of Hungarian industry, including electronics and computer manufacturers, and at the same time a vivid interest in information and computing

services result more students at sections of informatics and less in other branches of engineering;
- in answer to these changes and utilizing new opportunities, more universities and colleges of technology, including newly established, private ones, offer degree courses in informatics.

For example, for the year 1992/93 the Faculty of Electrical Engineering and Informatics could enrol only 382 (1991/92: 460) freshmen in electrical engineering while the acceptance level at the entrance examinations has been lowered to 80 scores (1991/92: 101!) out of 120. At least for a while, technical informatics has not been losing its attractiveness: for 1992/93 the faculty has enroled 146 (1991/92: 75) freshmen in informatics while the acceptance level was set at 100 scores (1991/92: 111).

University authorities and staff should overcome many other problems if TU Budapest wants to remain attractive and keep its leading role in informatics education in Hungary, e.g.

- Since universities in Western-Europe and the United States, further private firms, home and abroad, offer more attractive and in many cases easier professional careers and first of all much higher wages many younger colleagues have left TU Budapest.
- For the same reason, it is not easy to recruit new staff members with proper educational background or experience. (On the other hand, there is a surplus of 'real' electrical, mechanical, etc. engineers.)
- Departments devoted to one or another field of informatics are still missing at TU Budapest. Partly due to this fact, no significant working groups and personalities have emerged. While the establishment of one or even more informatics departments can no longer be delayed without long-lasting consequences, the conditions, both personnel and financial, are much worse than it was a decade ago.
- While many of our graduates go abroad to work, higher education is still free of charge in Hungary — this is, however, another story.

Fortunately, conference papers on informatics education, like e.g. [3], mitigate, to some extent, this rather dim picture: at least we know we are not completely alone! For example, [2] complains as in Germany 'there are more vacancies for faculty staff than good candidates, because industry offers so many interesting job opportunities.' And what [6] says we also know too well: 'From the beginning, computing scientists have had to convince colleagues from other, more mature disciplines that computing is a discipline.'

6 Revised curriculum

The Curriculum 1991 of the degree programme in technical informatics at TU Budapest reflects a two-level structure: an undergraduate level (3 years) and a graduate level (2 years), with no formal boundary between these two levels. The undergraduate programme is divided into three main blocks. One of them consists of courses related to mathematics, information and coding theory. Another one contains subjects specific to electrical engineering. The third block is devoted to informatics: courses related to programming, computing science and software engineering. The content of the graduate level varies since it consists of modular and elective courses. They give the students the opportunity to acquire special knowledge according their interests and abilities.

In the first six semesters, the exercises in the computing laboratory help the students gain the necessary skills in programming and computer applications. The project laboratories in semester 8 and 9 give them the opportunity to work in bigger teams; traditionally, they join a research or development group of the faculty.

Below, an excerpt from the schedule of courses is reproduced. (See [8] for a fuller account. In case of possible deviations the current report is valid since it reflects newer developments.) Only the undergraduate courses closely related to SE are shown in Table 2. For comparison: the total number of hours is 24 a week, ca 20% less than in Curriculum 1986.

Table 2: Undergraduate courses related to SE
(e – examination, p – practical exercise, s – signature, i.e. no grade)

Course name	Hours/week with requirement						
	total	in semester					
		1	2	3	4	5	6
Programming	4	2e	2p				
Programming Technology	4			4e			
Theory of Algorithms	4			4e			
Formal Languages	4				4e		
Mathematical Logic	4				4e		
Programming Paradigms	4					4e	
Operating Systems	4					4e	
Databases	4						4e
Computing Laboratory	12	2s	2s	2p	2p	2p	2p

Graduate courses are much more flexible than in earlier curricula. Only the framework is set up, the content depends on future needs and possibilities. Their structure is depicted in Table 3. Course descriptions may be obtained from the authors.

Table 3: Framework for graduate courses

Course name	Hours/week with requirement				
	total	in semester			
		7	8	9	10
Module 1	12	4e	4e	4e	
	4	4e			
	6	2p	2p	2p	
Module 2	12	4e	4e	4e	
	4	4e			
	6	2p	2p	2p	
Elective courses	12	4e	4e	4e	
Project laboratory	12		6p	6p	
Thesis work	24				24

Further, the development of Curriculum 1991 aimed at

- decreasing the cost of education (by reducing the extent of labs and projects);
- decreasing the number of weekly hours in classes;
- increasing the flexibility of studies and meeting individual needs of students (modules, electives);
- adding courses to the core curriculum that strengthen its computing science and software engineering character (Math logic, Paradigms, Programming technology);
- creating space for formal specification and design techniques, functional and logic programming, etc.

Many important topics like compiler construction, artificial intelligence and expert systems, network and information systems management, neural networks, robotics, etc. will be covered by modular and elective courses.

7 Conclusion

It is not enough that we, computing professionals at academia, are convinced about the necessity of informatics education: it is our duty, too, that government and university authorities, professionals in business and industry, colleagues in other engineering disciplines accept *informatics as a discipline*. In the long run, 'information industry' could fill the space caused by the collapse of other branches (electronics, metallurgy, mining, etc.) in Hungary. However, to promote its proper development, *positive discrimination is necessary* at TU Budapest. Conferences, like this, contribute to make a clear picture: where we are, what we do, where we go. Even more if nobody knows *the* answer.

Acknowledgements

The authors express their thanks to their colleagues at TU Budapest who, like them, took part in the elaboration of the computing science and SE courses for the section of technical informatics. Nonetheless, their views, manifested in this article, are not necessarily shared by the others.

References

[1] Bruce, K.B.: *Creating a new model curriculum: a rationale for* Computing Curricula 1990. In [3], pp. 23–35.

[2] Brauer, W.: *Informatics education at West German universities.* In [3], pp. 125–131.

[3] *Education & Computing.* Issue on 'Informatics Curricula for the 1990s'. Vol. 7, Nos. 1–2, ISSN 017–9287, Elsevier.

[4] Gries, D. – Levrat, B. – Wegner, P.: *Foreword. Informatics Curricula for the 1990s.* In [3], pp. 3–8.

[5] Gries, D.: *Improving the curriculum through the teaching of calculation and discrimination.* In [3], pp. 61–72.

[6] Gibbs, N.E.: *Software engineering and computer science: the impending split?* In [3], pp. 111–117.

[7] Habermann, A.N.: *Introductory education in computer science.* In [3], pp. 73–86.

[8] Györfi, L. – Hanák, P. – Selényi, E.: *The Degree Programme in Technical Informatics at the Technical University Budapest.* Budapest, 1992.

[9] Reé, B.: *Feasibility of Repository-Based CASE Environments.* Master thesis. TU Budapest, 1992.

[10] Shaw, M.: *Prospects for an Engineering Discipline of Software.* IEEE Software 7, 6 (November 1990), pp. 15–24.

[11] Shaw, M. – Tomayko, J.E.: *Models for Undergraduate Project Courses in Software Engineering.* CMU–CS–91–17. September, 1991.

[12] Zemanek, H.: *Formalization. History, Present, and Future.* Programming Methodology, 4th Informatik Symposium, Germany, Wildbad, Sept. 25–27, 1974, in Lecture Notes in Computer Science, Vol. 23, pp. 477–501.

Teaching Programming Via Specification, Execution and Modification of Reusable Components: An Integrated Approach

Ahmed Ferchichi
Université de Tunis III
Institut Supérieur de Gestion de Tunis
Département Informatique
41 Rue de la Liberté 2000 le Bardo

Abstract. In computer science curriculum, it is a challenge to define and teach a first programming course including software engineering concepts and integrating programming paradigms. The aim of this paper is to present a teaching approach to address this problem. Experimented in the university of Tunis III, the approach is caracterised by the use of software library units representing actually Prolog and Ada programs respectively as specifications and implementations. Specifically, at the level of its goal, the approach is oriented by external and internal software qualities; at the level of its strategy, the approach is based on program execution and program modification; and at the level of its conceptual formalisation, the approach uses a relational view.

Keywords. Teaching Programming, Software Engineering, Software Reuse, Software Modification.

1. Introduction

The debate of how to teach a first year programming course is as old as programming itself and, in all likelihood, will remain in the center stage of computer science education. In recent years, several important languages specifically designed for software engineering [1] have emerged, most notably Ada. These languages may be used at both the software design and the implementation stages of the development process [8].

Good Programming involves the systematic mastery of complexity. It is not an easy subject to teach. The principal tenet is that abstraction and specification are necessary for any effective approach to programming ([5],[6]).

In this paper, we present a teaching approach related to a first programming course including software engineering concepts and integrating programming paradigms.

We consider that the first contact of students with programming is of prime importance and ought to be controlled carefully [3]. Based on the use of a software library components where some units are considered as specifications and others as implementations, our teaching approach uses successively external then internal program use. So, it introduces external program qualities by external program execution and internal qualities by internal program modification, hence tools and concepts achieving this goal as defined recently in software engineering [8] are progressively introduced.

The next section presents the programming course concerned by the approach. Section 3 deals with the presentation of our teaching approach based on two kinds of activities: program execution and program modification. In section 4, we present an illustrative example. In section 5, we present some remarks showing the interest of the selected approach. And section 6 outlines future work and perspectives.

2 Course Description and Organisation

2.1 Course Objectives

We are concerned by a first programming course caracterized by the following premises:
- The integration of functional, logic, and object oriented programming paradigms as programming tools.
- The introduction of a simplified software life cycle for program development as programming model [7].
- The study of software product and process qualities as programming goal.

2.2 Software Libraries and Case Studies

2.2.1 Software Libraries

All the teaching activity is based on the use of the following software libraries:
- Environment library,
- Programming language library,
- Data structure library,
- Domain oriented library.

These libraries represent the teaching library and are respectively related to:
- The used programming environment as the operating system, the editor, and the compiler.
- The programming langage represented by the primitive data types.
- The classical data types, as sets, lists, stacks, and queues.
- Programs which are specific to particular domain-specific applications.

2.2.2 Case Studies Applications

Case studies applications are directly linked to the teaching library. Along the specification activity, the teacher asks students to access the teaching library, execute some selected software units in order to report their description in separate specification files. Along the implementation activity, students access initial implementation texts related to the specified selected software units and modify them progressively and iteratively in order to achieve particular software qualities.

We justify the previous kinds of libraries by the following premises:

- The environment library is naturally the first one used by students; also, we want to help the student apprehend it and understand it using the same approach.

- The language library makes practical the idea that a programming language can be viewed as a set of programs: for this purpose, the predefined data types in this library are organised as accessible and independent units.

- The data structure library shows that there is no conceptual difference between data types and data structures and enables us to enrich and make more abstract the programming langage used.

- Finally, the domain oriented library constitutes a set of prototypes helping students understand more complex specifications.

2.3 Course Structure

The programming course is organised into three kinds of sessions:
- theoretical course session (36 hours),
- application course session (72 hours), and
- practical course session (108 hours).

These sessions are organised during 24 weeks with respectively the following credits: (1 x 1h30), (2 x 1h30), and (3 x 1h30). The course begins with practical sessions during 10 days (3 hours per day) in order to introduce students rapidly to the environment library. During these sessions, students organise their libraries to make them ready to accept specification and implementation files. We recall that all kinds of sessions are organised around the teaching library.

3 Teaching Strategies

In practice, we want students to build a software library similar to the teaching library with software units having particular qualities. Using Ada as a programming langage, each software unit is organised into separate specification and implementation files. Because it is difficult to understand specifications, we have decided to make them executable as Prolog programs. So, the work is organised following the two next sequential programming activities:

- specification activity, and
- implementation activity.

To learn specification and implementation, we use the following sequential teaching activities:

- program execution, and
- program modification.

The aim of the next sections is to explain in more detail the programming and teaching activities and their relationship.

3.1 Learning Specification by Program Execution

Given software units, the aim of this activity is to lead students understand external program qualities. To this effect, we follow two steps:

- defining program abstractions, and
- matching program abstractions against program specifications.

3.1.1 Defining Program Abstractions

The main question that we adress in this step is:

Given a program P. What is the meaning of P?

Using the teaching library and external execution of its software units, the aim of this step is to make students learn how to approach program abstractions. In this step, we characterise each program by its input space, output space, domain, codomain, and relation between input and output spaces. We consider in this step two kinds of programs: non deterministic programs and deterministic programs representing respectively specifications and implementations. We recall that actually specifications are Prolog programs [4] and implementations are Ada programs. Given a selected program from the teaching library, this program is first executed, and then its abstraction is written in a separate Ada specification file.

3.1.2 Comparing Program Abstractions

The main question that we tackle in this step is:

Given a program P defined by its specification T and its implementation I. When can we say that I is correct w.r.t. T?

Given the software library, we treat essentially the correctness quality. Considering that a program is defined by its specification and implementation, the correctness criteria is adressed by comparing abstractions respectively associated to the specification and implementation of each selected program.

Illustrative examples processes defineetness, partial correctness, total correctness, empty and full specifications, implementations having larger domains than specifications, and specifications having several implementations.

3.2 Learning Implementation by Program Execution and Modification

To guide students in implementing specifications, we consider two steps: solving specifications by external program use and solving specifications by internal program use.

3.2.1 Solving Specifications by Program Execution

The main question addressed in this step is:

> Given a software library L and a specification T. Is there a program P in L having an implementation I that is correct w.r.t. T? If not, is it possible to find a set of implementations doing the same thing by external user synchronisation and communication?

Hence, in this step, we introduce students to use the library to solve specifications. When a student is looking for an implementation, he has to compare his specification to actual specifications in the library to be able to determine whether he can associate to it a given implementation. When there is no implementation satisfying a given specification, students look for a set of implementations and try to find an external sequence, choice, or iteration use.

In this activity, the student has to synchronize the communication between different units: each time he uses a unit, he compares his intended specification with the unit's specification. In this processes, he learns some elements about specification decomposition and program composition.

3.2.2 Satisfying Specifications by Program Modification

The main question addressed in this step is:

> Given a software library L and a specification T. Is there in L a program P having an implementation I that is correct w.r.t. T? If not, is it possible to find a set of implementations doing the same thing by internal synchronisation and communication or modification?

This question is made harder by some constraints such as software internal qualities. We answer it by considering a set of parameters that we change alternatively during the course, applying them to data types and data structures.

3.2.2.1 Modification Parameters

The set of parameters we consider is defined by the following elements related to a considered data type or data structure:

- specification,
- implementation style,
- representation type, and
- test procedure.

The specification can be defined:

- without exeptions, or
- with exceptions.

Operations are defined as
- procedures,
- functions, or
- pocedures and functions.

These operations can be:
- binary having two arguments, or
- n-ary having more than two arguments.

They can also be:
- non generic, or
- generic.

Implementations have styles in
- the functional style using recursion, or in
- the imperative style using iteration and assignment.

A representation type can be:
- a predefined data type, or
- a new data type.

And it can be:
- not private, or
- private.

A test procedure uses a data type or data structure package to be tested.

All the activity is conducted by combining these parameters going from one version to another. We are guided by the idea of program modification and the interest of program qualities.

A particular attention is given to a data structure list. This data structure is lisp oriented and it introduces the following operations: init, cons, first, and empty. Also when this structure is implemented, we use it to implement the other recursive data structure implementations.

3.2.2.2 Programming Steps

The following activities are followed by students and teachers:
- Understanding specification by execution.
- Designing an implementation version.
- Maintaining this version by modification.

4 Illustrative Example

The aim of this example is to illustrate the student work when he is defining the function of a particular software unit and comparing abstractions.

For that, let me consider that each software unit P is caracterised by specification T and implementation I. Because specifications and implementations are executable, we caracterise them respectively by (Xt,Yt,[T]), and (Xi,Yi,[I]) where Xt, Yt, [T] denote respectively the input space, the output space, and the function of the specification; and Xi, Yi, [I] denote respectively the input space, the output space, and the function of the implementation.

Consider now that the student is asked to execute the algorithmic specification add1 and the implementation add2 defined on the same and equal input and output spaces. His report should be as follows:

```
        For specification add1

        S = integer x integer x integer
        add1 = (S, S, [add1])

        [add1] = {((0,0,-),(-,-,0)),
                   (0,1,-),(-,-,1)),
                   (1,0,-),(-,-,1))}.

      For implementation add2

        S = integer x integer x integer
        add2 = (S, S, [add2])

        [add2] = {((0,0,-),(0,0,0)),
                   (0,1,-),(0,1,1)),
                   (1,0,-),(1,0,1))}
                   (1,1,-),(1,1,2))}.
```

So, we introduce progressively a mathematical logic notation to describe the link between the input and output data. The descriptions given above can also be denoted in closed form by the following formulas:

```
        S = {(x,y,z)/ (x,y,z) in (integer x integer x integer)}

        [add1] = {((x,y,z),(x',y',z'))/ x in {0,1} and y in {0,1}
                                   and z'=x+y and x'=x and y'=y}

        [add2] = {((x,y,z),(x',y',z'))/ x in {0,1} and y in {0,1}
                                   and z'=x+y}.
```

This enables us to verify that students are able to evaluate predicates and understand what they are representing.

In the begining of program execution activity, we choose software units having small spaces and domains. In the other cases we determine only partially the program function.

When we are concerned by the data type specification, each element of the input space is a sequence made up of some operations of the data type. For example, the following elements belong to the abstraction [stack]:

```
        (init.push(a).push(b).push(c).top,c)
        (init.top,error).
```

Considering the preceding abstractions related to add1 and add2. To verify that add2 is correct w.r.t. add1, we execute add1 and add2 for each add1 input data and compare their associated set of outputs. We obtain the following results:

   Let me define the domain of add1 by:

d = {(0,0,-),(0,1,-),(1,0,-)}.

For the input data (0,0,-)
add2 computes the output data set O1= {(0,0,0)}
add1 computes the output data set O2= {(-,-,0)}.
We note that O1 is included in O2.

For the input data (0,1,-)
add2 computes the output data set O1={(0,1,1)}
add1 computes the output data set O2={(-,-,1)}.
We note that O1 is included in O2.

For the input data (1,0,-)
add2 computes the output data set O1={(1,0,1)}
add1 computes the output data set O2={(-,-,1)}.
We note that O1 is included in O2.

So the student concludes that add2 is correct (totally) w.r.t. add1. The reader can note that the verification algorithm interpretes the following correctness formula:

> Implementation I is correct w.r.t. specification T if and only if for each input data s of T, I defines an output data s' defined by T.

Given a software unit P, when its associated specification T is executed and its abstraction determined, the student creates for it an Ada separate specification file, then compiles it and catalogues it.

For example, the following specification file is created for the software unit add:

```
with types; use types;
procedure add(s1: in S; s2: out S);
-- begin specification
     -- prec: x in {0,1} and y in {0,1}
     -- posc: z' = x + y
-- end specification;
```

The reader can note that the preconditions and postconditions formulas are directly extracted from the relation representing the abstraction of add1. The unit types is here a package containing the declaration of the type S.

Also, because during program execution activity, students manipulate data spaces, we have found that they are more prepared to make the transition from graphical and mathematical description of a program unit to its Ada representation, accept the langage syntax, and understand more easily the semantic. For space S, e.g., the following description is used in package types:

```
type S = record
           x: integer;
           y: integer;
           z: integer;
         end record;
```

5 Interests

The execution phase is made necessary by the fact that with the recent advent of microcomputers, access to computing facilities is growing more wide-spread. This is causing a myriad of problems ranging from heterogeneity of students'backgrounds to ill-conseived first contacts with programming [2].

It becomes possible to teach all the data types operations of the language by executing these data types as separate programs. When the data types of the language are executed externally, we describe this execution and assimilate it to a specification. Because programs are considered as black boxes, we expect that students will refer to them in their problem solving activity on the basis of their abstraction.

Also, students learn data abstraction because they are asked to find a unit declaration: they have to find the parameters, their names, and types; all the declaration is application oriented and is not influenced by the names of the primitive data types.

Students also learn some elements of specification decomposition and program composition because they are asked to solve specifications by external communication between program units.

Solving specifications by execution evoques also the hierarchy between abstractions and prepares the later explanation that implementations are more specific than specifications which can be interpreted that they contain more details. Hence the student library is reorganised as a set of units, each unit being characterised by one specification (program) and one or more than one implementation (program). So, the basic programming culture of the student is updated at the same time. This means that we expect students to use specifications in program design activity.

Hence, it becomes possible to enrich the student culture by virtual specifications. The student is invited to imagine specifications, write their corresponding abstractions, and catalogue the description in a separate specification file. When later he recognises the specification, he can use it.

By giving the syntactic declaration of each operation in one data type, and using the specification coming from the specification step using external execution, it becomes easy to reimplement the langage operations using various parameters as operation name, and operation form. This work introduces naturally the import export notion, the subroutine call, and the sequence control structure. Also, it completes the first programming culture related to the programming language.

External use of program units will surely raise exception messages, then the previous implementations are naturally updated: the exception mechanism is introduced along with the conditional control structure.

Hiding information and using program abstractions during implementation is learned using different versions of the test procedure of the data structure. When a data structure is not private, students observe that they can directly access the representation structure and must change their programs when the implementation uses another representation structure. When a data structure is private, they observe that the compiler prevents them from accessing the representation structure and observe that they do not have to change the procedure test even though the representation structure changes.

The inheritance concept is introduced first by using any defined data structure as a new representation structure and next by adding new operations to the data structure. The following operations are used to enrich the data structure:

- make full
- make empty
- insert by position or value after or before an element determined by position or value.
- delete by position or value after or before an element determined by position or value.
- modify by position or value after or before an element determined by position or value.
- list all elements.

The genericity concept is introduced after developing specialised particular versions.

Functional programming style is learned using the following recursive unit model:

```
with operations; use operations;
function iter_recursively(x: in T1) return T2 is
  begin -- iter_recursively
    if p(x)
      then return it(x)
      else return
             cons(tr(first(x)),iter_recursively(rest(x)));
      end if;
  end iter_recursively;
```

where c, it, cons, tr, first, and rest are functions depending on the considered specification at hand, and separately developped and tested.

A particular attention is given to the list data structure. This data structure is lisp oriented and introduces the following operations: init, cons, first, and empty. Also when this structure is implemented, we use it to implement the other recursive data structure implementations.

The logic programming style and specification writing is introduced first by authorizing the reading access of Prolog file texts and next by asking students to formalise specifications they have written during execution activity. But this step is not yet experimented. Following the same idea, we can also organise the teaching of specification validation as we have organised the teaching of verification.

The iteration technique is introduced later in the course using the following unit model:

```
with operations; use operations;
procedure iter(x: in T1; y: out T2) is
  x1: T1 := x;
  y1: T2;
  stop: boolean;
  begin -- iter
    it(y1)
    iv(x1)
    ir(stop);
    while (not stop)
      loop
        tr(x1,y1)
        av(x1);
        ar(x1,y1,stop);
      end loop;
      y := y1;
  end iter;
```

where operations depend on the specification.

When we are faced with an iterative specification, we make its implementation equivalent to the problem of finding the following three pairs of subroutines (tr,it), (av,iv), (ar,ia) respectively related to the definition of the output y, the process of the input x, and the definition of the iteration test. Our conviction is that students can learn a great deal about how to do implementations by using abstractions and how to import and export program units before to teach them iteration, as it is done in most introductory programming courses.

6. Conclusion

Including multi-programming paradigms in first programming courses becomes a necessity on one hand and constitutes a special challenge for teachers on the other hand. In this paper we have proposed an approach to handle this problem.

Overall, we feel fairly satisfied with the course as it is now, though we are seeking to improve it through interactions with other teachers of similar courses.

Acknowledgement

The author gratefully acknowledge Pr. R. Mittermeir (Klagenfurt University) and Pr. A. Mili (Tunis University) for previous discussions on the subject. This work is supported by a grant from FRST (The National Foundation for Scientific and Technical Research).

References

1. Boehm, B. Software Engineering Economics. Prentice-Hall, 1981.

2. Dijkstra, E.W. Selected Writing on Computing: A Personal

Perspective. Springer Verlag, 1982.

3. Ferchichi, A., and A. Jaoua. Teaching First Year Programming: A Proposal. SIGCSE Bulletin, September, 1987.

4. Ferchichi, A., et A. Mili. Spécification Relationnelle Assistée par Prolog. Département d'Informatique, Université d'Ottawa, Mars 1992.

5. Liskov, B., and J. Guttag. Abstraction and Specification in Program Development. Mass., MIT Press, 1986.

6. Meyer, B. Object-oriented Software Construction. Prentice Hall International, New York, 1988.

7. Mili, A., N. Boudriga, and F. Mili. Towards Structured Specifying: Theory, Practice and Applications. Chichester, Ellis Horwood Ltd., 1989.

8. Wiener, R., and R. Sincovec. Software Engineering With Modula-2 and Ada. New York, John Wiley and Sons Inc., 1984.

TEACHING AND TRAINING IN THE CASE TOOL ENVIRONMENT

Tatjana Welzer, Jozsef Györkös
University of Maribor
Faculty of Technical Sciences
SLO-62000 MARIBOR, Slovenia
Smetanova 17
Tel. + 038 62 25 461, Fax + 030 62 212 013
e-mail: welzer@uni-mb.ac.mail.yu
gyorkos@uni-mb.ac.mail.yu

Our contribution focuses on the experiences gained while training different groups of users in the CASE tool environment. Most of these tools are based on the classical design methodos like the Chen E-R diagram, or the DeMarco-Yourdan and Gane&Sarson techniques [2], which are reasonably well known to users.

An effective use of any CASE tool should be supported by a basic knowledge of the chosen structured methods [1,3]. This fact leads to a division of users into different groups: users experienced in software development but with little knowledge of structured methodos, users experienced both in software development and in structured methods and inexperienced users who are usually experts in structured methods.

The intention of our presentation is not to draw a general conclusion about which group of users is more successfuler in teaching and training, but only to highlight a problem of basic knowledge as well as of experience in the CASE tool environment.

The first group that is involved in our experiment is assembled from young engineers inexperienced in software development, who gained quite a lot of knowledge about structutred methods at the university , while in the second and the third groups, users experienced in the development of different projects are involved. They have a lot of experience and expert knowledge about software development and in addition they are not (the third group) or just partly (the second group) familiar with the basic knowledge of the techniques mentioned above. Because of this they are sometimes intolerant towards the new way of work supported by the CASE tool. Therefore we try to combine training in the CASE tool environment with teaching of some of the skills needed in the relevant structured methods [4].

References

1. J.M. Clifton: An Industry approach to the Software Engineering Course, ACM SIGCSE BULLETIN Vol. 23, No. 1, March 1991, pp.296-299.

2. C. Finkelstein: An Introduction to Information Engineering, Addison Wesley, 1989.

3. J.E. Tomayako: Teaching Software Development in a Studio Environment, ACM SIGCSE BULLETIN Vol. 23, No1., March 1991 pp.300-303.

4. T.Welzer,J.Györkös:Teaching structured techniques supported by CASE tool,CASE 2,Rijeka,1990, pp.2/1-5.

SCIENCE POLICY

Chair: G. Haring

Research Policy in Information Technologies for Small European Countries

The dynamism in research and in industrial development in Information Technology as well as concentrated efforts to boost R&D in this discipline on national and supranational levels poses the special question of how to meet this challenge from the perspective of a small country.

In Europe, the various programs to foster information technology have already a solid tradition. Special schemes have been developed by the EEC-Commission for establishing programs, soliciting and evaluating proposals as well as monitoring the progress of sponsored projects. The high volume of research money thus available and the possibility for setting trends poses special challenges for such Central-European countries as Austria - just seeking EEC-membership -, Hungary - being in the process of change in its economic, and hence also research policy system -, and Slovenia - a country which obtained independence just recently.

The following distinguished IT-policy leaders are invited to address under the chairmenship of G. Haring, President of the Austrian Computer Society, the consequences to be drawn from the technical, economical, and political circumstances we currently witness in central Europe:

P. Lepape (CEC, Directorate-General XIII, Brussels);
C. Bašković (Ministrstvo za Znanost in Tehnologijo, Ljubljana);
L. Nyíri (National Committee for Technological Development, Budapest);
N. Roszenich (Bundesministerium f. Wissenschaft und Forschung, Wien)