Volume 14 Number 1 January 1990
YU ISSN 0350–5596

Informatica

A Journal of Computing and Informatics

The Slovene Society INFORMATIKA
Ljubljana

Informatica

A Journal of Computing and Informatics

Subscription Information

Informatica (YU ISSN 0350 – 5596) is published four times a year in Winter, Spring, Summer and Autumn (4 issues).

The subscription price for 1990 (Volume 14) is US$ 30 for companies and US$ 15 for individuals.

Claims for missing issues will be honoured free of charge within six months after the publication date of the issue.

Printed by Tiskarna Kresija

Informacija za naročnike

Informatica (YU ISSN 0350 – 5596) izide štirikrat na leto, in sicer v začetku januarja, aprila, julija in oktobra.

Letna naročnina v letu 1990 (letnik 14) se oblikuje z upoštevanjem tečaja domače valute in znaša okvirno za podjetja DEM 16, za zasebnike DEM 8, za študente DEM 4, za posamezno številko pa DM 5.

Številka žiro računa: 50101 – 678 – 51841.

Zahteva za izgubljeno številko časopisa se upošteva v roku šestih mesecev od izida in je brezplačna.

Tisk: Tiskarna Kresija, Ljubljana.

Na podlagi mnenja Republiškega komiteja za informiranje št. 23 – 85, z dne 29. 1. 1986, je časopis **Informatica** oproščen temeljnega davka od prometa proizvodov.

Pri financiranju časopisa **Informatica** *sodeluje Raziskovalna skupnost Slovenije.*

Volume 14 Number 1 January 1990
YU ISSN 0350-5596

Informatica

A Journal of Computing and Informatics

EDITOR-IN-CHIEF
Anton P. Železnikar
Iskra Delta Computers, Ljubljana

ASSOCIATE EDITOR
Rudolf Murn
Jožef Stefan Institute, Ljubljana

The Slovene Society INFORMATIKA
Ljubljana

Letnik 14 Številka 1 Januar 1990 YU ISSN 0350-5596

Informatica

Časopis za računalništvo in informatiko

VSEBINA

GIVING PRIORITY TO INFORMATIONAL TECHNOLOGY?

Keywords: computer technology, informational technology, informational language

Anton P. Železnikar
Iskra Delta

Summary. *In this article the necessity and the reasonableness of possible informational technology versus traditional computational technology are discussed. This orientation also calls for a new, more flexible, i.e. informational language.*

In his preface article, published in the *New Generation Computing* **6** (1989) 359-360, *Toshio Yokoi*, from the Japan Electronic Dictionary Research Institute (the group from ICOT), is putting the question whether to give priority to information-oriented technology over computer-oriented technology. He argues that priorities in information-processing technology are changing rapidly from computer to information orientation. Although he is still predominantly "thinking" in terms of traditional (e.g., Shannonian, metric) encapsulation of the term information, he can, in fact, not avoid (or reveal) the awareness of the difference existing between real (living, autopoietic, anthropological) information (as coming into existence, for instance, at the use of a dictionary) and the so-called computer-produced or data-structured "information". This awareness images evidently the difference he observes between the two technologies, where the informational reminds us of mind (or information-processing) and computational of brain (or machine, substance) structured and organized concept.

The increasing interest in artificial intelligence (and not only in knowledge-information processing) and man-machine interface is pointing to the direction, which can be understood as Informational. The consequence of disposed awareness is that the emphasis is already shifting from system (machine) software to application (practical information) software, where users are beginning to play their own productive role in creating their computer systems.

Computing is becoming communicational and modern communication has to become more and more computational. "Communication and information systems" is becoming a common and indivisible term. By virtue of such development, computing with communication is becoming Informational, not only in the sense of traditional (hard, mathematical) sciences, but more and more in the sense of that what living beings understand (or think) as information, as informationally arising phenomenology of information.

Yokoi argues that the point in creating of information-oriented technology implies the arising of information-oriented theory and basic technology. And he says that there have been various discussions on what this common base should be. It should be a language which encompasses both natural language and artificial language, where the last includes logical formulas, algebraic formulas, programming languages, graphic forms, etc.

In his articles concerning informational logic, algebra, and discourse, the author has shown one of the possibilities how the way to the so-called informational philosophy and theory could be traced and how this philosophy and theory could impact the fields of natural language theory, artificial language theory, the language theory integrating the two, and the language-processing technology (as stressed by *Yokoi*).

How could the informational logic and the informational algebra be used by an Electronic Dictionary Project?

References

Yokoi, T., Giving Proirity to "Information-Oriented Technology" over "Computer-Oriented Technology", *New Generation Computing* **6** (1989) 358-360.

Železnikar, A. P., Informational Logic I, II, III, IV, *Informatica* **12** (1988) *3*, 26-38; *4*, 3-20; *Informatica* **13** (1989) *1*, 25-42; *2*, 6-23.

Železnikar, A. P., An Informational Theory of Discourse I, *Informatica* **13** (1989) *4*, 16-37.

Železnikar, A. P., An Introduction to Informational Algebra, *Informatica* **14** (1990) *1*, 7-28.

ZEN AND THE ART OF MODULAR ENGINEERING

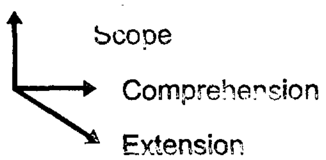
Keywords: modular engineering, modularization, software

Brian R Kirk MSc MBSC,
Robinson Associates, United Kingdom

Presented at the 1^{sr} Int'l Modula-2 Conference, October 12-13, 1989 Bled-Yugoslavia

INTRODUCTION

As time passes computing components became an integral part of more and larger systems. The diagram shows our dilemma



The **scope** of what is requested seems only to be limited by what can be imagined. Somehow as designers we must gain **comprehension** of all the implications of the whole system in all its states. And most taxing of all we must provide accurate solutions that support **extensions** to match the requirements as they evolve.

This paper offers an approach to coping with the dilemma, the title encapsulates the concepts.

Zen	looking inside in a search for understanding
Art	a fine skill
Modular	separate parts designed to be cohesive
Engineering	designing and building practical machines

A real product is used as an example of how modules and machines made from modules can provide reuse and extension of existing software, even when there are difficult constraints on the implementation.

The .objective of the paper is to pass on the experience learned whilst engineering a large real time software system. In particular the approaches used to divide and conquer the complexity and inherent concurrency may be of interest to implementors of high integrity systems. In all cases the pragmatic approach to the finding of practical solutions is described. The language Modula-2 has been used as the programming notation. It is now hard to conceive or believe that such a large system could have been created so effectively with any other available language. Particularly in a form that can be understood and extended with ease.

THE PRODUCT

Often it is necessary to update an existing product and give it a new image. In our case the requirement was to take a paper-tape based multi-axis machine tool and to match it to the current marketplace. The extensions included CAD, graphics, a file system, a printer, remote controlled operation - and all this with either English, French, German, Italian and Russian interaction with the user - see Figure 1.

The **form** of any design is a product of its designers interpretation of its requirements and constraints. In this case the constraints were formidable ...

1 the need to support all existing functionality

2 the impossibility of all but minor modification to existing software (some sources were lost!)

3 the need for a real-time response on the display, CNC, remote link and language translation

4 the need to interact in ad-hoc ways with 3 existing computers

5 only having a RAM memory of one third the size of the whole program

6 the need to make all the new software resilient to power failure for continuous operation

7 the Client's prior choice of DOS and GEM for filing, graphics and multi-tasking

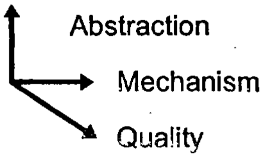
The completed software is large, it contains:

3 programs with 15 overlays
150 modules
2000 messages each in 5 languages
2 Mbytes of executable code
30 Mbytes of source code

It was developed by a team of 6 people over a period of 2 years. Had we realised initially the full scope of the requirements and the implications of the constraints we might never have started. Only the rigorous use of modular engineering concepts and carefully coordinated implementation in Modula-2 by a team of professional software engineers made the whole project feasible.

MODULAR ENGINEERING

Engineers analyse problems using concepts and then synthesize their solution by organising some physical form, in this case the software part of the system. The diagram shows the main criteria

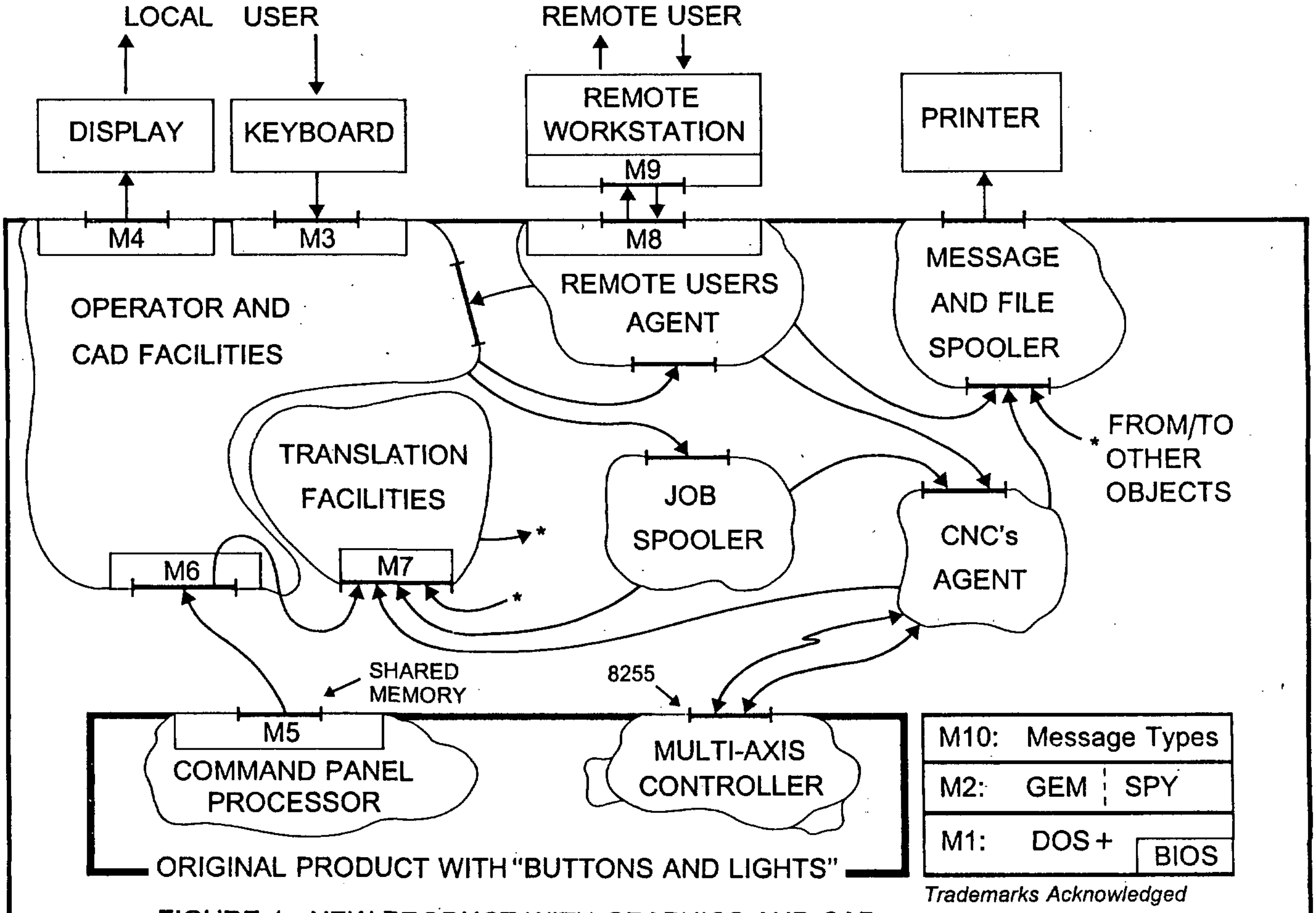


The **abstractions** we use to analyse and model the problem have evolved over the past 40 years of computing, these include

Names	for instructions, data and locations
Macros	to encapsulate and reuse the text of sequences of instructions or data
Procedures	to encapsulate and reuse sequences of instructions at runtime
Control Structures	to encapsulate the flow of control
Classes	to encapsulate evolutionary definitions in a reusable and extensible way
Modules	to encapsulate whole components, hiding information and/ or ownership
Extensible Modules	to encapsulate objects which have statically related definitions
Delegating Objects	to encapsulate objects which are dynamically related and extensible. An object which cannot provide a requested method delegates it to another object which can.

Languages provide a means to express solutions to problems in terms of these abstractions, for example, Assembler, Algol, Simula, Modula-2, Oberon and Delegate. The trend in abstraction is towards an object oriented approach because this minimises the distance between the problem and its programmed solution: "the solution is a simulation of the problem". In practice we have found Modula-2 an adequate language for expressing both modules and delegating objects, which are message driven tasks consisting of modules.

The **mechanisms** are simply ways of achieving something. For example in Figure 1, modules M5 and M6 provide an interface between various tasks in the two processors. In our first implementation M5 replaced the old graphics card driver and sent equivalent messages to M6. This made it possible to reuse the vast majority of the original software



Trademarks Acknowledged

FIGURE 1: NEW PRODUCT WITH GRAPHICS AND CAD

with minimal changes. Of course M6 completely hid the protocol and a rather nasty dual port RAM interface from all the new software. This technique was much too slow in practice and was later replaced by a set of records and update flags at agreed fixed positions in the shared memory. By using modules on each side to encapsulate the mechanisms it became possible to change the mechanism separately from the rest of the system. We found that a good test for the quality of a module's interface was to consider how much it would need to change if the mechanism it encapsulated but not necessarily the functionality, has to change.

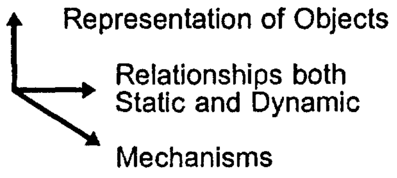
The **quality** of the implementation is the third main factor. Engineers differ from computer scientists in that they are faced with many practical constraints and exceptions yet their solution must be effective in actual use. For example the machine tool can cut diamonds and diamonds are valuable. The clients are not impressed by large diamonds that unfortunately have the wrong shape due to software errors. About 15% of the modules we wrote were **test harness** modules which either exercised the modules under test or acted as dummy modules for uncompleted parts of the system. Sometimes we wrote modules to provide rough prototypes of parts of the system that were poorly specified or particularly difficult to achieve. By isolating these areas adequate solutions were found quickly and the risk to the whole system minimised.

Sometimes the structure or quality of existing software was too risky to incorporate into the product. In these cases we 'reverse engineered' the software. This involved analysing the code to discover what the intended requirements were, we then made the requirements self-consistent. The software was then redesigned in line with the system model, mechanisms and modules. This concept provides clean maintainable software rather than horribly bodged incongruous coding - it also takes less effort.

The system was constructed as a 'pile of machines' implemented with programs, processes and modules. Always striving to verify that the partially complete system had 100% correct functionality within itself. This policy of **stepwise construction** of the system provided visibility of progress, a practical means to assess quality and confidence for our Clients.

CRITERIA FOR MODULARISATION

The main reason that we partition systems into subsystems and modules is to encapsulate our comprehension and thus extend our capabilities. This is achieved by using abstraction to separate out distinct parts of the problem. These abstractions are then implemented by building logical machines on top of physical ones to mechanise the abstraction in a form, and at a cost, which is appropriate to the user. In the past the criteria for modularisation were influenced by the 'everything is a hierarchy' view of programming, latterly the use of 'information hiding' as a criterion has been much more useful. What is really needed is a set of criteria that maximise the separation of ...


Representation of Objects
Relationships both Static and Dynamic
Mechanisms

At the same time we wish to optimise

- ease of comprehension
- ease of development by teams
- flexibility for extension possibilities

Perhaps the fundamental criterion is that each separate part, be it active, object or component module, should be testable. If it is not certain that something can be tested before it is built then there is little point in building it because there is no possibility for quality assessment or control.

Looking back on our projects we can now see the actual criteria that have been most effective, they include encapsulation of reuse, adaption, concurrency, consistency and mechanisms.

REUSE

It is a fact of life that reuse of what already exists is often essential. Usually the reason is short-term economic optimisation (this is rarely justified in practice!) but sometimes it is just not possible to reimplement old parts of a new system because there is not enough time or the knowledge is no longer available. In any case if a product is still 'alive' it certainly will need to be extended to match its behaviour and performance to the evolving needs of its users. This needs to be done with minimal modification of existing parts but the aim is to inherit the functionality and system model from the existing system. Unfortunately the mistakes and constraints are also inherited, the main disadvantages of standardization.

There are some classic examples of reuse in Figure 1. The whole of the old machine software is reused except for two modules that provide a new interface to the software extensions, eg module M5. More typical ones are M1 and M2 which provide 'cleaned up' interfaces to DOS and GEM. Indeed GEM provides both graphics and multiprogramming scheduling machines built on-top-of DOS. The GEM constraints of supporting only 4 programs (not tasks) and of round-robin scheduling were inherited by the system and distorted its form, reducing productivity.

ADAPTION

When creating large systems it always pays to make the software part as portable as possible. Conventionally this is done by providing 'device driver' modules which abstract away particular physical characteristics at the lowest level and offer a clean logical software interface instead. The client modules then use the clean interface so making it portable and also improving the flexibility for hardware machine choice. Typically these modules are hidden in 'the BIOS' but any new devices can have their drivers written in Modula-2.

Modules M3 and M4 are good examples in practice. M3 extends the normal DOS keyboard driver to support the storing of a keyboard history and also the alternative keyboard layout and coding needed to support Russian. Module 4 had to be rewritten to match a non-standard graphics display controller.

CONCURRENCY

The sequential instruction by instruction execution of programs by CPUs has unfortunately led generations of programmers to presume that concurrency does not exist. They inherently try to coerce the concurrency of the problem into a single stream of CPU instructions. The liberation from this mental straight-jacket is inherent in data flow diagrams which show the flow of information between processing activities. Their use has broken the curse of the flowchart which deems its uses to think only in terms of sequential control flow. Figure 1 takes the concept of a DFD further, it shows the flow of information between naturally concurrent objects in the system, be they logical or physical objects. By initially analysing the problem in terms of the concurrent objects it contains we get some very clear benefits ...

1 the implementation can be a simulation of the problem

2 the objects can be allocated to or shared between the processes/processors depending on their individual performance needs of throughput and response time

3 once the shared modules and interfaces between objects are defined and designed the implementation of the objects can be developed separately by members of a team

4 synchronisation and communication between objects can be optimised separately to suit the needs of the objects, see modules 5 and 6, 8 and 9 later

5 the system can be constructed incrementally by providing dummy objects as 'stubs'

Creating a concurrent-object-information-flow-diagram as the 'top' level of the analysis and design process gives a clear overview of the whole system. It is the equivalent of the hardware engineers 'system block diagram' and the architects initial building design sketches, nothing really new.

Figure 1 provides many examples. The soft shapes enclose the logical objects in the system and the arrows the information flows. The objects are in fact allocated processor time in a variety of ways

- in the original product each has its own processor
- in the extended product they share 3 tasks on another processor. The objects share the tasks but were originally written and tested separately, during optimisation they were coalesced to save memory overheads
- time critical objects such as timers are activated physically by CPU interrupt events

To provide the quasi-concurrency that the objects require the modules M1, M2 and M10 are used. M1 provides a clean interface to DOS and M2 multiplexes DOS to create 3 separate program environments. M10 provides datatype definitions for messages sent between the programs. Note that this module requires special version control treatment because it is shared by separate **programs**, if it is changed then all programs that use it need to be remade. It is noteworthy that no 'real time' operating system was used, objects being either co-operatively scheduled or event driven by real time events depending on their needs.

CONSISTENCY

The possibility for automating consistency checking is perhaps the greatest benefit of using non-permissive languages, 'C' and Assembler are permissive! Pascal introduced strong data type checking, Modula-2 has introduced the possibility of explicit control of visibility of module contents combined with an environment which automates inter-module consistency checking at both compile time and run-time. Even greater support is needed when modules are shared between separately compiled programs because a change made to satisfy one program may have nasty knock on effects which are inconsistent for the other programs. We tackled this problem by extending the PVCS Version Control system using batch command files to automate the consistent updating of modules shared between programs.

There are some examples of modules which enhance consistency in Figure 1. Module M10 contains a set of datatypes which define the format of messages passed between objects in the system. Variant records are used to overlay data of differing types over the same memory area. The generic data format is consistent with existing GEM messages so that new message formats become extensions to the existing ones. Module M6 contains a set of datatypes which define records in a shared dual port memory. It also hides access procedures which provide a synchronised read/write protocol with the Command Panel Processor, guaranteeing consistent atomic access to each whole record of fields. **Incidentally static compile time checking was helpful but we also found that dynamic runtime type checking of both subrange values and enumeration values was essential.** The reason for this is that the other processor and its interface module M5 were programmed in assembler code, of course without compile or runtime type checking. By having the benefit of runtime checking in the extended software it was possible to quickly detect and locate errors in the existing system.

By using these techniques it is possible to guarantee consistency at a system level even between modules shared by programs on the same processor or with 'foreign' modules on other processors. Much of the existing software was written in 'C'; by putting a veneer of Modula-2 over it it became possible to maintain strong type checking.

OWNERSHIP AND MECHANISMS

Perhaps the most powerful criteria is that of ownership. Here the concept is that the module owns a mechanism such as how a filing system is structured or how a protocol works. The interface, or external visibility, of the mechanism is minimised. Its clients are only informed of its information and services on a 'need to know' basis. When trying to partition a system, an object or program for modules, the main criteria are not only to encapsulate the ownership of mechanisms and/or information but also to guarantee that its behaviour and performance can be verified. If modules are well designed in this way then the mechanisms can evolve to meet the requirements with minimal changes to the client modules. The underlying architecture of objects, programs and module relationships should be resilient to particular choices of mechanisms chosen for an implementation.

Nearly every module shown in Figure 1 owns a separated mechanism. Module M7 owns the mechanism for converting to message values into a text string in either English, French, German, Italian or Russian. Initially the string was searched for in a linear way from floppy disk, just to get a prototype working quickly. Of course the mechanism was not fast enough for an interactive graphics based product. The implementation module was redesigned and rewritten 3 times, each time with a faster mechanism until the product performance goals were achieved. The definition module and client modules did not change at all.

Another pair of modules, M8 and M9, own the interprocessor protocol used to reliably convey commands and information between the machine and a remote workstation. The modules were written and exhaustively tested in isolation from the system. The protocol was specified as a finite state machine and implemented using a CASE statement and a variable of an enumerated type to represent the state. The list of meaningful names in the enumeration type made possible the creation of a readable program which could benefit from strong type checking. To improve the performance of the remote channel the mechanism for activating the protocol was changed from 'application polling' to 'timer event' activation. Once again this was achieved without altering the client modules.

BENEFITS OF THE EXPERIENCE

1 Partitioning by **concurrency of objects** leads to a very clean system design 'the solution is a simulation of the problem'.

2 The most useful criteria for modularisation are reuse, adaption, concurrency, consistency and ownership of mechanisms.

3 It is important to separate out parts that are likely to change often, such as the user dialog text.

4 Stepwise refinement is relevant to improving mechanisms without changing their functionality. Also product specifications evolve dynamically and it is necessary to consciously adapt and refine the program architecture to suit the new facilities required.

5 Modula-2 is an excellent language for engineering large systems in a contolled way. It is flexible enough to express both object oriented and structured programming concepts. It provides for both compile and runtime checking.

6 Specialised languages like SmallTalk would have been inappropriate for the solution; languages like 'C' are just too inherently lax and unreliable for such large projects.

7 The problem of controlling the integrity separate programs which share modules can easily be solved in the development environment.

8 Designing the system as a " pile of interacting machines" provides for great productivity, flexibility and portability.

ACKNOWLEDGEMENTS

Prof N Wirth - for separating out the concepts
B B McGibbon, S Doyle, J Teague - for help with system design
D Gifford, T Harris, D Fox - for reliable programming
C Diegez, P Zanolari, M Wakely - for modifying the existing parts
Dr B Schumacher, K Pferdekaemper, P Locati, M Bertoli - for project direction
J Woodhouse, S Doyle, G Luker - for typesetting.

REFERENCES

1 Dijkstra E W, May 1968'
"The structure of the 'THE multiprogramming system"
Comm of the ACM, N5, Vol 11.

2 Wirth N, April 1971
"Program Development by Stepwise Refinement"
Comm of the ACM

3 Parnas D L, Dec 1972
"On the criteria to be used in Decomposing Systems into Modules"
Comm of the ACM

4 Maruichi, Uchiki and Tomoro, 1987
"Behavioural Simulation based on Knowledge Objects"
Dept Electrical Engineering, Keio University, 3-14-1 Hiyoshi, Yokohama 223, Japan

5 Stein L A, Liebermann H, Ungar D, 1989
"A shared view of sharing: The Treaty of Orlando"
Object Oriented Concepts ... ISBN 0-201-14410-7
Addison Wesley

AN INTRODUCTION TO INFORMATIONAL ALGEBRA

Keywords: algebra, information; general, implicative, equivalence, modal, and self-informational algebra

A. P. Železnikar*

This article deals with algebraic concepts of information and brings five basic algebraic systems, called self-informational, general informational, implicatively informational, equivalence informational, and modal informational algebraic system, which are listed at the end of the article. Informational algebra considers the informational nature of its entities - operands and operators, and in this relation, it introduces traditional logical operators (implication, equivalence, disjunction, conjunction, etc.) as particularities, which project a self-informational or general informational algebra into a particular domain (for instance, implicative, equivalence, modal, etc.). The way to an informational logic is paved with basic reflection and determinations (definitions), which root in informational logic [1, 2, 3, 4]. This article shows, how a new paradigm in formalizing and automatizing of informational concepts can become possible. In this way, it is also a proposal for a sufficiently diverse but constructive mathematical and technological treatment of the arising informational phenomenology - also of the needs arising in the domain of the so-called information-oriented technology [12].

UVOD V INFORMACIJSKO ALGEBRO. Članek se ukvarja z algebraičnimi koncepti informacije in prikaže pet osnovnih algebraičnih sistemov, in sicer samo-informacijskega, splošno, implikativno, ekvivalenčno in modalno informacijskega, ki so zapisani na koncu članka. Informacijska algebra upošteva informacijsko naravo svojih entitet - operandov in operatorjev in tako uvaja tradicionalne logične operatorje (npr. implikacijo, ekvivalenco, disjunkcijo, konjunkcijo itd.) le kot posebnosti, ki projicirajo samo-informacijsko ali splošno informacijsko algebro v posebno področje (npr. implikativno, ekvivalenčno, modalno itd.). Pot do informacijske logike je podložena z osnovno refleksijo in opredelitvami (definicijami), ki temeljijo v informacijski logiki [1, 2, 3, 4]. Članek pokaže, kako lahko nova paradigma formalizacije in avtomatizacije informacijskih konceptov postane mogoča. V tem smislu je članek tudi predlog za dovolj diverzno vendar konstruktivno matematično in tehnološko obravnavo nastajajoče informacijske fenomenologije - tudi potreb v območju t.i. informacijsko usmerjene tehnologije [12].

> ... A priori and irrespective of any hypothesis concerning the essence of matter it is evident that the matter-of-factness of a body does not end there where we touch it. The body is present everywhere where its impacting can be sensed. Its force of attraction - if we speak only of it - acts upon sun, planets, maybe also upon the entire universe.
>
> Henri Bergson [10] 159-160

0. INTRODUCTION

> A priori and irrespective of any hypothesis concerning the essence of information it is evident that the informing of information does not end there where it is coming into existence. Information is present everywhere where its informing can be sensed. Its impacting - if we speak only of it - can inform living beings as well as the entire universe.
>
> Paraphrasing Henri Bergson informationally

Informational algebra or algebra of information is a set of definitions concerning informational axioms and informational rules

* Iskra Delta Computers, Development and Production Center, Stegne 15C, 61000 Ljubljana, Yugoslavia, Europe (or privately: Volaričeva 8, 61000 Ljubljana, Yugoslavia, Europe).

for formatting of formulas by which the construction or deduction of formulas or their transformation becomes constructively possible. Informational formulas are compositions of informational entities marking various informational processes and consisting of the so-called informational operands and informational operators. By this approach, informational algebra becomes an informational calculus not only for informational or informationally mechanical generation of formulas within a given algebraic system, but also for informational decomposition and through it for informational enriching, development, interpretation, and modeling of living and artificial informational systems. In this sense, systems represented by formulas are open, i.e., constructively growing, steady, and/or reducing formal systems. In general, an informational system is an informationally arising (changing) system, in which each informational entity possesses the possibility of informing, i.e. of informational arising.

Every algebraic approach concerns logical means, shaping the nature or the background of the algebraic approach. In this respect, informational algebra is logically grounded in informational logic [1, 2, 3, 4] and various concepts belonging to it [5, 6, 7, 8]. Informational algebra concerns informational entities which are informational operands and operators, aggregated to formulas. An informational formula marks descriptively a specific operand and so, can be informationally operated again. Within an informational algebra several categories of operands and operators can be distinguished, e.g., implicit and explicit ones, particularized and universalized ones, etc. Further, such algebra considers that informational operands can be decomposed into formulas which bring to the surface new operands and operators. In a similar way, informational operators can be decomposed, showing operational components of an operator decomposition. Thus, algebraic composition (building of operands, operators, and formulas) and decomposition (determining of operands' and operators' details) of informational entities, of operands as well as operators, are the most natural means of an informational algebra.

Within the study of informational algebra also the axiomatic nature of information can be considered and recognized. For instance, how does information perform as informational phenomenon of its own informing, how the marking or symbolism of informational phenomenology can be introduced, and last but not least, how informational arising, which is the phenomenon of informing of information, can be semantically captured, pragmatically composed and decomposed, and operationally marked and symbolized. It becomes evident that a symbolism possessing informational meaning, generality, and particularity is needed and has to be introduced in such a manner that it will embrace already existing mathematical and new informational conceptualism. For this purpose the consequent informational style of thinking and understanding becomes necessary, living existing formal and particularly mathematical doctrinairism behind it and surpassing the doctrinaire blocking by informational constructiveness and meaning. This does not mean at all that informational algebra cannot be concise, compact, and self-constructive discipline. However, it might be said that informational algebra will be conceptually broader from the standpoint of existing and abstractly comprehended algebraic disciplines, integrating them into a new, informational realm.

Introduction to informational algebra in this essay is the only beginning of such algebra, which as a new discipline is looming on the horizon of informational logic. The goal of such algebra is to enable formal analysis and modeling of various living and artificial informational system, for instance, compose them globally and decompose them into detail, particularize them on a given point of view and later on universalize them and enable their further decomposition, etc. At this time, introduction means also that some distinguished domains of informational algebra are yet not elaborated into the necessary detail. This essay is on the way to reveal significant and controversial details, particularities, and formalism of the future informational algebra.

1. CLASSICAL LOGICAL AND ALGEBRAIC APPROACH

At the beginning it is to stress that in the conceptualization of informational logic and informational algebra it would not be recommendable to proceed from the usual predicate calculi being determined within various mathematical theories. All these calculi are based on the category of truth and falsity which represents a very particular informational entity of belief or mathematical disciplinarity. Such determinism of research would fatally narrow the naturally open realm of informational investigation and exclude the main informational phenomenology from the formally structured and organized discourse. However, this does not mean that predicate calculi of mathematics would be excluded from the formal informational discourse; on the contrary, they can be integrated into the realm of informational investigation and present usable particularities of an informational calculus. Further, set-theoretical symbolism in its various form can be applied too, etc.

Let us show a set of rules of deduction as it appears within the classical logic. Let us introduce two separation symbols, '[' and ']', for expression of formal units. Let us introduce informational entities α, β, and γ, representing rather informational and not only propositional operands, and five "propositional" connectives: '$\neg$' for negation, '$\vee$' for disjunction, '$\wedge$' for conjunction, '$\Rightarrow$' for implication, and '$\equiv$' for equivalence. Under these conditions it is possible to accept or postulate some rules of deduction, belonging to a particular (informational, or in our case also propositional) language:

[1.1]:

$$[\alpha \Rightarrow [\beta \Rightarrow \alpha]]$$
$$[[\alpha \Rightarrow [\alpha \Rightarrow \beta]] \Rightarrow [\alpha \Rightarrow \beta]]$$
$$[[\alpha \Rightarrow \beta] \Rightarrow [[\beta \Rightarrow \gamma] \Rightarrow [\alpha \Rightarrow \gamma]]]$$
$$[[\alpha \wedge \beta] \Rightarrow \alpha]$$
$$[[\alpha \wedge \beta] \Rightarrow \beta]$$
$$[[\alpha \Rightarrow \beta] \Rightarrow [[\alpha \Rightarrow \gamma] \Rightarrow [\alpha \Rightarrow [\beta \wedge \gamma]]]]$$
$$[\alpha \Rightarrow [\alpha \vee \beta]]$$
$$[\beta \Rightarrow [\alpha \vee \beta]]$$
$$[[\alpha \Rightarrow \gamma] \Rightarrow [[\beta \Rightarrow \gamma] \Rightarrow [[\alpha \vee \beta] \Rightarrow \gamma]]]$$

$$
\begin{gathered}
[[\alpha \equiv \beta] \Rightarrow [\alpha \Rightarrow \beta]] \\
[[\alpha \equiv \beta] \Rightarrow [\beta \Rightarrow \alpha]] \\
[[\alpha \Rightarrow \beta] \Rightarrow [[\beta \Rightarrow \alpha] \Rightarrow [\alpha \equiv \beta]]] \\
[[\alpha \Rightarrow \beta] \Rightarrow [[\neg \beta] \Rightarrow [\neg \alpha]]] \\
[\alpha \Rightarrow [\neg [\neg \alpha]]] \\
[[\neg [\neg \alpha]] \Rightarrow \alpha]
\end{gathered}
$$

etc. Such kind of rules can be replaced by more general as well as more precise ones. For instance, instead of

[1.2]: $$[\alpha \Rightarrow [\beta \Rightarrow \alpha]]$$

there will be the first or universalized step

[1.3]: $$[\alpha \models [\beta \models \alpha]]$$

This will be followed by the second and more precise step

[1.4]: $$[[[\alpha \models] \models [[\beta \models \alpha] \models]]] \models]$$

In the third step the last formula can be particularized, e.g., into

[1.5]: $$[[[\alpha \models_T] \models_{\Rightarrow} [[\beta \models \alpha] \models_T]]] \models_T]$$

The meaning of operators $\models$ and $\models_T$ and expressions of the form $[\alpha \models]$ will be explained later.

Let us introduce also the universal and the existential quantifier, i.e., $\forall$ and $\exists$. In the framework of informational logic and informational algebra we will use also particularized quantifiers, i.e., $\forall_\pi$ and $\exists_\pi$, denoting the possibility π of $\forall$ and $\exists$, and reading them as "it is possible that for all" and "it is possible that there exist(s)", respectively. In the framework of informational logic, the rule [1.2] can be postulated as

[1.6]: $$[[\exists_\pi \beta] \centerdot [\alpha \Rightarrow [\beta \models \alpha]]]$$

This formula is read as "it is possible that there exist an informational operand β such that (operator $\centerdot$) if α is an informational operand, then (operator $\Rightarrow$) operand β informs (operator $\models$) operand α.

For the second rule in [1.1] there would be, for instance, in the framework of informational logic

[1.7]: $$[[[\forall \alpha] \exists \beta] \centerdot [[\alpha \models [\alpha \models \beta]] \Rightarrow [\alpha \models \beta]]]$$

The curiosity of this formula is, for instance, that existential quantifier $\exists$ performs as an explicit binary operator between operands $[\forall \alpha]$ and β and that $[[\forall \alpha] \exists \beta]$ is the left operand of operator $\centerdot$. Thus, this formula is read as "for all α there exists β such that if α informs $[\alpha \models \beta]$, then $[\alpha \models [\alpha \models \beta]]$ can be replaced by $[\alpha \models \beta]$. Thus, rule [1.7] can become a practical rule of formula reduction within informational algebra. Certainly, it is not necessarily true that

[1.8]: $$[[\alpha \models [\alpha \models \beta]] \Rightarrow [\alpha \models \beta]]$$

is an informationally valid formula, since in the left part $[\alpha \models [\alpha \models \beta]]$ of implication the process α informs the process $\alpha \models \beta$, and this informing might not be the same as $\alpha \models \beta$ on the right side of implication. In many cases it can be understood that $[\alpha \models \beta]$ is an indivisible process and in this manner α can impact this process merely as its entire structure. Thus, if $[[\alpha \Rightarrow [\alpha \Rightarrow \beta]] \Rightarrow [\alpha \Rightarrow \beta]]$ is proposition-logically acceptable, its informational counterpart $[[\alpha \models [\alpha \models \beta]] \Rightarrow [\alpha \models \beta]]$ might be not. This example shows the problem of informational universalization of proposition-logical formulas as truth and falsity are characteristically narrowed informational categories.

The difference which exists between the classical algebraic and logical approach on one side and informational approach on the other side lies in the fact that the first approach deals with rather static entities whereas the second one deals with processes and not only propositions in mind. In a similar manner, informational formulas have to be understood as processes by themselves and in this sense they underlie the principle of informational arising, i.e. development of given formulas as formulas through composition, decomposition, universalization, particularization, or simply by changing of formulas' instantaneous structure.

2. INFORMATIONAL OPERATORS

2.0. Introduction

The informational operand is determined to be informational entity marking an informational process which is comprehended as informational unity. Irrespective of the complexity of an informational operand which can be composed of various explicit and implicit informational operands and informational operators, this operand performs informationally, i.e., informs, counter-informs and embeds the self-produced and the arrived information.

An informational operand informs as informational unity and in this respect it informs self-informationally. In this paragraph we have to answer basic algebraic questions concerning the self-informational nature of informational operands. As it is already understood, an informational operand is in no way an informationally non-operative entity. It functions as an operand or as an operated entity merely in relation to operators being superior to it, which have the power to operate it informationally. However, within itself, an informational operand operates and is operated according to its informational constituents, which are informational operands as well as informational operators. And several components of an informational operand can operate within other informational operands. In this way it is to understand that an informational operand is a specific part of an informational system, of an informationally marked and operatively connected net of informational entities, which are informationally perplexed, interwoven, interactive, distributive, and distributed.

In this respect, the basic question which arises is how to ensure the expression of the described complexity and of the arising nature of information in a formal or symbolic manner. We shall recognize how the introduction of the metaoperator of informing $\models$ will explicitly keep the arising nature of informational entities occurring in a formula alive. Therefore, this implicit arising power has to be given to any operator of the type ⊨, regardless in which way it is or can be particularized or universalized. Usually we suppose that operator ⊨ performs as an expert operator (system) in the domain (field, discipline, realm, etc.) of its possible particularization or universalization.

2.1. On the Nature of Operator ⊨

In our further discussion we shall consequently use the symbol ⊨ as a general operator variable, which can be substituted by any other operator variable possessing a more precise or more determined operational meaning. Further, irrespective to the degree of its determination, any informational operator can be universalized with the intention to analyze or investigate the informational consequences of a particularized operator. For instance, logical or arithmetic operators will be replaced by more general operators with the intention to study more general properties of an informational operand (formula) which does not concern merely the informationally narrowed (particularized) aspects of logical truth or falsity and numerical value, respectively. On the other side, it will be possible to keep mathematically defined entities algorithmically stable in cases of necessity of artificial, technological, and symbolic systems (for instance, for the needs of today's artificial intelligence and classical mathematical systems).

In fact, the introduction of the notion of the informational metaoperator ⊨ [9] and its still general (universal) operational derivatives (for instance, ⫤, ⊭, ⫤̸, ⊫, ⫥, ⊯, ⫥̸, ⊢, ⊣, ⊬, ⊣̸, ⊩, ⫣, ⊮, ⫣̸, etc.) enables the development of the concept of informational arising. The nature of this operator is informing of operands to which this operator belongs and informing is by definition nothing else than informational arising in one or another way. Thus, the formula $\alpha \models$ expresses the property of informational operand α that it is in the process of informing, of sending or transmitting of information through its own informing. If we would write $\alpha \models_\pi$, it would mean that entity α can inform; but α as information can inform in each case. For instance, in the case of modal logic we could introduce the following definition:

[2.1]:

$$(\alpha \models) =_{\mathrm{Df}} (((\forall\, \mathfrak{M}) \wedge (\forall\, (\beta \in \mathfrak{M}))) \cdot (\alpha \models_{\mathfrak{M},\beta}))$$

Here, 𝔐 is the so-called model of possible worlds and β is a possible world. The operator $\models_{\mathfrak{M},\beta}$ is already the particularized form of operator ⊨. Simply, it is possible to say that operator ⊨ is determined within the informational domain (𝔐, β), which can be understood to be sufficiently general, adapted to instantaneous need and application. In this case, each informational entity α has the possibility to send information, to inform. This case represents the active role of information and also of data. $\alpha \models$ means that α informs in all models of possible worlds and in each possible world of the model.

The form $\models \alpha$ expresses the property of informational entity α to be informed, to be sensible to some extent for the reception of information by informing in itself as well as by informing of other informational entities. If we would write $\models_\pi \alpha$, we would say that information α could be informed. By definition, data as a particular, informationally restricted entity, cannot accept (receive) information. Thus, for instance, $\nvDash \alpha$ is valid. In the framework of modal logic it could be possible to set the following informational definition:

[2.2]:

$$(\models \alpha) =_{\mathrm{Df}} (((\exists\, \mathfrak{M}) \wedge (\exists\, (\beta \in \mathfrak{M}))) \cdot (\models_{\mathfrak{M},\beta} \alpha))$$

In this case, informational entity has the possibility to receive information or to be informed by itself or by other informational entities. This role of information lies in the activity of its receiving of information. Thus, $\models \alpha$ means that there exist suchlike models of possible worlds and a possible world β within them that α can be informed.

Operator ⊨ (and in this respect any informational operator) can perform as unary, binary, or multiplex operator. In the case $\alpha \models$, ⊨ is a unary, postfix operator, whereas in the case $\models \alpha$, ⊨ is a unary prefix operator. A binary form is, for instance, $\alpha \models \beta$, and a multiplex one, for instance, $\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta$. Various informational operators have been discussed in [2]. However, some definitions of informational operators may be helpful for more exhaustive understanding of their nature.

2.2. Implications and Definitions Concerning Unary Informational Operators

In general, every informational operator can appear as unary operator being connected with one or more operands. Let us discuss merely cases of the most general unary informational operators.

2.2.1. The Case 'α informs'

The form $\alpha \models$ or $\Dashv \alpha$ says that α informs. This formulas are implicatively open in the following sense:

[2.3]:

$$(\alpha \models) \Rightarrow$$
$$((\exists_\pi\, \xi, \eta, \ldots, \zeta) \cdot (\alpha \models \xi, \eta, \ldots, \zeta));$$
$$\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$$

$$((\exists_\pi\, \xi, \eta, \ldots, \zeta) \cdot (\xi, \eta, \ldots, \zeta \Dashv \alpha))$$
$$\Leftarrow (\Dashv \alpha);$$
$$\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$$

If α informs, then there may exist some informational entities ξ, η, ... , ζ, which are informed by α. The consequence of this implication might be the implication

[2.4]:

$$(\alpha \models) \Rightarrow (\exists_\pi\, (\alpha \models \alpha));$$
$$\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$$

$(\exists_\pi\ (\alpha \Dashv \alpha)) \Leftarrow (\Dashv \alpha);$
$"\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$

If α informs, then it could inform itself. Logically, the following inverse implication can be adopted:

[2.5]: $(\alpha \models \xi, \eta, \ldots, \zeta) \Rightarrow (\alpha \models);$
$"\models" \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$

$(\Dashv \alpha) \Leftarrow (\xi, \eta, \ldots, \zeta \Dashv \alpha);$
$"\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$

If α informs informational entities ξ, η, ... , ζ, then it informs in general too. The particular case of this implication is

[2.6]: $(\alpha \models \alpha) \Rightarrow (\alpha \models);$
$"\models" \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$

$(\Dashv \alpha) \Leftarrow (\alpha \Dashv \alpha);$
$"\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$

If α informs in itself, then it informs. Now, we can adopt the following complete list of implications proceeding from the previous discussion, which concern general informational operators (the so-called "informing" operators and the so-called "non-informing" ones), which perform (inform) from the left to the right and vice versa, i.e., operators $\models$, $\VDash$, $\vdash$, $\Vdash$, $\nvDash$, $\nVDash$, $\nvdash$, $\nVdash$, $\Dashv$, $\DashV$, $\dashv$, $\dashV$, $\not\Dashv$, $\not\DashV$, $\not\dashv$, and $\not\dashV$:

[2.7]:

$(\alpha \models) \Rightarrow$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\alpha \models \xi, \eta, \ldots, \zeta));$
$(\alpha \VDash) \Rightarrow$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\alpha \VDash \xi, \eta, \ldots, \zeta));$
$(\alpha \vdash) \Rightarrow$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\alpha \vdash \xi, \eta, \ldots, \zeta));$
$(\alpha \Vdash) \Rightarrow$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\alpha \Vdash \xi, \eta, \ldots, \zeta));$

$(\alpha \nvDash) \Rightarrow$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\alpha \nvDash \xi, \eta, \ldots, \zeta));$
$(\alpha \nVDash) \Rightarrow$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\alpha \nVDash \xi, \eta, \ldots, \zeta));$
$(\alpha \nvdash) \Rightarrow$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\alpha \nvdash \xi, \eta, \ldots, \zeta));$
$(\alpha \nVdash) \Rightarrow$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\alpha \nVdash \xi, \eta, \ldots, \zeta));$

$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\xi, \eta, \ldots, \zeta \Dashv \alpha))$
$\Leftarrow (\Dashv \alpha);$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\xi, \eta, \ldots, \zeta \DashV \alpha))$
$\Leftarrow (\DashV \alpha);$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\xi, \eta, \ldots, \zeta \dashv \alpha))$
$\Leftarrow (\dashv \alpha);$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\xi, \eta, \ldots, \zeta \dashV \alpha))$
$\Leftarrow (\dashV \alpha);$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\xi, \eta, \ldots, \zeta \not\Dashv \alpha))$
$\Leftarrow (\not\Dashv \alpha);$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\xi, \eta, \ldots, \zeta \not\DashV \alpha))$
$\Leftarrow (\not\DashV \alpha);$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\xi, \eta, \ldots, \zeta \not\dashv \alpha))$
$\Leftarrow (\not\dashv \alpha);$
$((\exists_\pi\ \xi, \eta, \ldots, \zeta)\bullet(\xi, \eta, \ldots, \zeta \not\dashV \alpha))$
$\Leftarrow (\not\dashV \alpha)$

At this occasion it becomes evident that it is possible to particularize the general informational operator $\models$ by the so-called general operators (metaoperators) $\models$ and $\nvDash$, general parallel operators $\VDash$ and $\nVDash$, general cyclic operators $\vdash$ and $\nvdash$, and general parallel-cyclic operators $\Vdash$ and $\nVdash$. Thus,

[2.8]: $(\alpha \models) =_{\mathrm{Df}} ((\alpha \VDash) \vee (\alpha \vdash) \vee (\alpha \Vdash));$
$(\alpha \nvDash) =_{\mathrm{Df}} ((\alpha \nVDash) \vee (\alpha \nvdash) \vee (\alpha \nVdash));$
$(\alpha \models) \Rightarrow ((\alpha \models) \vee (\alpha \nvDash))$

This definition says simply that α informs or does not inform in a parallel, cyclic, and/or parallel-cyclic manner. The parallel-cyclic case is to be understood as a parallel and cyclically perplexing complex mode of informing of an informational entity.

In the similar way the performance of operator $\Dashv$ can be defined. This operator demon-strates the diversity and alternativeness against the general operator $\models$. It can be showed how in cases of anthropological discourse this explicit informational operator becomes the necessity, delivering the unrevealed informing which lurks or waits in the background of each living informing and non-informing (for instance, as skepticism, unbelief, or simply counter-informing). Thus, adequately to [2.8] there is

[2.9]: $(\Dashv \alpha) =_{\mathrm{Df}} ((\DashV \alpha) \vee (\dashv \alpha) \vee (\dashV \alpha));$
$(\not\Dashv \alpha) =_{\mathrm{Df}} ((\not\DashV \alpha) \vee (\not\dashv \alpha) \vee (\not\dashV \alpha));$
$(\Dashv \alpha) \Rightarrow ((\Dashv \alpha) \vee (\not\Dashv \alpha));$

It is to understand that if operators $\models$, $\VDash$, $\vdash$, $\Vdash$, $\nvDash$, $\nVDash$, $\nvdash$, and $\nVdash$ inform in one way, then their counterparts $\Dashv$, $\DashV$, $\dashv$, $\dashV$, $\not\Dashv$, $\not\DashV$, $\not\dashv$, and $\not\dashV$ inform in another way. This verbal difference between the first and the second case (class) of various informational operators ensures that the alternative horizon of informing comes explicitly (formally) into existence too. Further, it is to understand that $\alpha \models$ can have the meaning (or metameaning) of $\Dashv \alpha$ too.

The first two formulas in expressions [2.8] and [2.9] state that in the domain of informational connectedness, which can be at most a cyclic, parallel, parallel-cyclic, parallel-serial, or parallel-sequential structure, general informing or non-informing is nothing else than a type of these kinds of informing. The last formula in [2.8] and [2.9] implicates merely the metarole (metameaning) of operators $\models$ and $\Dashv$, respectively. At last, operator $\models$ can take over the role to be the only informational metaoperator. Thus, for instance, $\alpha \models$ can have the meaning of $\alpha \models$ as well as of $\Dashv \alpha$, etc.

Up to now we have examined cases in which it was not said anything about the complexness or composedness of operands. In our case, operands are informational formulas marking informational composites of informational processes. Certainly, if α marks an operand, then $\alpha \vDash$ marks a formula of a single operand and operator, and this formula can be taken as operand too. Thus, it is possible to continue the discourse of unary informational operators concerning formula $\alpha \vDash$ as an operand.

By definition, if α marks an informational entity, then α informs. Inductively, on the basis of this fact, it is possible to construct an indefinite number of implications, namely,

[2.10]: $\alpha \Rightarrow (\alpha \vDash)$;
$(\alpha \vDash) \Rightarrow ((\alpha \vDash) \vDash)$;
$((\alpha \vDash) \vDash) \Rightarrow (((\alpha \vDash) \vDash) \vDash)$;
. . . .
"$\vDash$" $\in \{\vDash, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\}$

Thus, by the property of transitivity, there is

[2.11]: $\alpha \Rightarrow (\ \ldots\ ((\alpha \vDash) \vDash)\ \ldots \vDash)$;
"$\vDash$" $\in \{\vDash, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\}$

This formula says that informational entity can inform in all possible ways concerning informational operators $\vDash$, $\VDash$, $\vdash$, $\Vdash$, $\nvDash$, $\nVDash$, $\nvdash$, $\nVdash$. It says in a formally explicit way that α can be perplexedly complex in respect to parallel, serial, and parallel-serial informing. Thus, informational entity α marks a system or network of informational processes which constitute it informationally. By this, an informational entity becomes a systemic notion or notion of an informational network.

Formula [2.11] is an explicit expression (through the use of explicit informational operators of the type $\vDash$) of the arising nature of informational entity α. Besides of this explicitness of informational arising, there exists, by definition, also the operational implicitness (an implicit form of informational arising) of an informational entity α. This operational implicitness is coming to the surface when, for instance, an informational entity marked by α, is decomposed, and thus explicating its informational components (a composition of informational operators and operands). The origin of this discussion can be the following:

[2.12]: $(\alpha \vDash) \Rightarrow \mathfrak{I}_{\rightarrow}(\alpha)$ or simply
$(\alpha \vDash) \Rightarrow \mathfrak{I}(\alpha)$

where $\mathfrak{I}$ (or $\mathfrak{I}_{\rightarrow}$) is the implicit operator of informing or non-informing (or informing or non-informing from the left to the right). $\mathfrak{I}(\alpha)$ is a sort of functional expression which points out the operational component $\mathfrak{I}$ of the entity α. Obviously, inductively, the last expression can be expanded (decomposed), for instance, into

[2.13]:
$(\alpha \vDash) \Rightarrow \mathfrak{I}(\alpha)$;
$\mathfrak{I}(\alpha) \Rightarrow \mathfrak{I}(\mathfrak{I}\ \ldots\ (\mathfrak{I}(\alpha))\ \ldots\ )$;
$\mathfrak{I} \in \{\mathfrak{I}_{\vDash}, \mathfrak{I}_{\VDash}, \mathfrak{I}_{\vdash}, \mathfrak{I}_{\Vdash}, \mathfrak{I}_{\nvDash}, \mathfrak{I}_{\nVDash}, \mathfrak{I}_{\nvdash}, \mathfrak{I}_{\nVdash}\}$

where $\mathfrak{I}_{\vDash}$, $\mathfrak{I}_{\VDash}$, $\mathfrak{I}_{\vdash}$, $\mathfrak{I}_{\Vdash}$, $\mathfrak{I}_{\nvDash}$, $\mathfrak{I}_{\nVDash}$, $\mathfrak{I}_{\nvdash}$, and $\mathfrak{I}_{\nVdash}$ mark the so called general ($\vDash$), parallel ($\VDash$), cyclic ($\vdash$), and parallel-cyclic case ($\Vdash$) of informing and general ($\nvDash$), parallel ($\nVDash$), cyclic ($\nvdash$), and parallel-cyclic case ($\nVdash$) of non-informing, respectively.

Formulas [2.10] - [2.13] can be repeated for the so-called alternative case of informing concerning operators $\Dashv$, $\DashV$, $\dashv$, $\dashV$, $\not\Dashv$, $\not\DashV$, $\not\dashv$, and $\not\dashV$. These cases can be expressed in a strict symmetric (right-left) form, for instance:

[2.14]: $(\Dashv \alpha) \Leftarrow \alpha$;
$(\Dashv (\Dashv \alpha)) \Leftarrow (\Dashv \alpha)$;
$(\Dashv (\Dashv (\Dashv \alpha))) \Leftarrow (\Dashv (\Dashv \alpha))$;
. . . .
"$\Dashv$" $\in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$

This, by the property of transitivity, yields

[2.15]: $(\Dashv\ \ldots\ (\Dashv (\Dashv \alpha))\ \ldots\ ) \Leftarrow \alpha$;
"$\Dashv$" $\in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$

Analogously to [2.12] the following alternative formula is obtained:

[2.16]: $\mathfrak{I}_{\leftarrow}(\alpha) \Leftarrow (\Dashv \alpha)$ or simply
$\mathfrak{I}'(\alpha) \Leftarrow (\Dashv \alpha)$

where $\mathfrak{I}'$ (or $\mathfrak{I}_{\leftarrow}$) is the implicit alternative operator of informing or non-informing (from the right to the left). $\mathfrak{I}'(\alpha)$ is the alternative functional expression which points out the alternative operational component $\mathfrak{I}'$ of the entity α. Obviously, inductively, the last expression can be expanded (decomposed), for instance, into

[2.17]:
$\mathfrak{I}'(\alpha) \Leftarrow (\Dashv \alpha)$;
$['(\mathfrak{I}'\ \ldots\ (\mathfrak{I}'(\alpha))\ \ldots\ ) \Leftarrow \mathfrak{I}'(\alpha)$;
$\mathfrak{I}' \in \{\mathfrak{I}'_{\Dashv}, \mathfrak{I}'_{\DashV}, \mathfrak{I}'_{\dashv}, \mathfrak{I}'_{\dashV}, \mathfrak{I}'_{\not\Dashv}, \mathfrak{I}'_{\not\DashV}, \mathfrak{I}'_{\not\dashv}, \mathfrak{I}'_{\not\dashV}\}$

where $\mathfrak{I}'_{\Dashv}$, $\mathfrak{I}'_{\DashV}$, $\mathfrak{I}'_{\dashv}$, $\mathfrak{I}'_{\dashV}$, $\mathfrak{I}'_{\not\Dashv}$, $\mathfrak{I}'_{\not\DashV}$, $\mathfrak{I}'_{\not\dashv}$, and $\mathfrak{I}'_{\not\dashV}$ mark the so called alternative general ($\Dashv$), parallel ($\DashV$), cyclic ($\dashv$), and parallel-cyclic case ($\dashV$) of informing and alternative general ($\not\Dashv$), parallel ($\not\DashV$), cyclic ($\not\dashv$), and parallel-cyclic case ($\not\dashV$) of non-informing, respectively. In this manner the alternativeness of informing and non-informing in the case "to inform" is preserved also in an informationally implicit way.

The last question which we have to deal with more thoroughly within this section concerns the operational family of non-informing. The "non" in non-informing appears as a symbol of negation ($\neg$) and this operation is a regular unary connective of formal (mathematical) logic. Now, it is possible to show how the "logically" pure meaning of negation can become informationally contestable, questionable, and insufficient.

Let be

[2.18]: $\neg(\alpha \vDash) \Rightarrow (\alpha \nvDash)$;
"$\vDash$" $\in \{\vDash, \VDash, \vdash, \Vdash\}$;
"$\nvDash$" $\in \{\nvDash, \nVDash, \nvdash, \nVdash\}$;

$$\begin{aligned}
&(\not\Dashv\ \alpha) \Leftarrow (\neg(\Dashv\ \alpha);\\
&\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV\};\\
&\text{"}\not\Dashv\text{"} \in \{\not\Dashv,. \not\DashV, \not\dashv, \not\dashV\}
\end{aligned}$$

Concerning the last formula in which $\not\models$ is the operator of non-informing, it is possible to develop the following questions:

(1) If α marks an informational entity which informs ($\models$ or $\Dashv$), how does this entity not inform ($\not\models$ or $\not\Dashv$)? Evidently, $\neg$ as an informational operator of negation does not possess a totally negational meaning (operational power).

(2) How does α not inform ($\not\models$ or $\not\Dashv$) and what does this non-informing mean?

(3) If it is said that α does not inform in a certain way, then α either inhibits or is not capable to inform in a certain way. Thus, it is possible to say, for instance, that α informs in an inhibitory manner. This fact yields

[2.19]:
$$\begin{aligned}
&(\alpha \not\models) \Rightarrow (\alpha \models_i);\\
&\text{"}\not\models\text{"} \in \{\not\models, \nVDash, \nvdash, \nVdash\};\\
&\text{"}\models_i\text{"} \in \{\models_i, \VDash_i, \vdash_i, \Vdash_i\};\\
&\\
&(\Dashv_i\ \alpha) \Leftarrow (\not\Dashv\ \alpha);\\
&\text{"}\not\Dashv_i\text{"} \in \{\not\Dashv_i, \not\DashV_i, \not\dashv_i, \not\dashV_n\};\\
&\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV\}
\end{aligned}$$

The so-called non-informing ($\not\models$ or $\not\Dashv$) is nothing else than inhibitive informing.

(4) The reverse can also be certain. If α informs, then α does not inform in its all embracing informational variety or entirety. It informs only in a certain way and not in all possible (universal) ways. Thus,

[2.20]:
$$\begin{aligned}
&(\alpha \models) \Rightarrow (\alpha \not\models_n);\\
&\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash\};\\
&\text{"}\not\models_n\text{"} \in \{\not\models_n, \nVDash_n, \nvdash_n, \nVdash_u\};\\
&\\
&(\not\Dashv_n\ \alpha) \Leftarrow (\Dashv\ \alpha);\\
&\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV\};\\
&\text{"}\not\Dashv_n\text{"} \in \{\not\Dashv_n, \not\DashV_n, \not\dashv_n, \not\dashV_n\}
\end{aligned}$$

where $\not\models_n$ and $\not\Dashv_n$ mark the non-universal informing. This fact can be explained by the intentional nature of informational arising regardless of a certain informational entity. Any informational entity, as a process of its informational existing and arising of information, possesses a certain orientation or intentionality of its informing and only in this manner can inform or can be informed.

(5) If α informs a certain informational entity β, where $\alpha \models \beta$ or $\beta \Dashv \alpha$, and entity β is not "sufficiently" sensitive to the informing of α, then β is not "adequately" informed by α, i.e., $\alpha \not\models \beta$ or $\beta \not\Dashv \alpha$. Because of "specific" β's sensitivity in regard to α's informing, symbol $\models$ or $\Dashv$ or any other informational operator has to be understood as an operator composition of particular operators, i.e. in our case of $\models_\alpha$ and $\models_\beta$ or $\Dashv_\alpha$ and $\Dashv_\beta$, that is as that what α intends to inform and what β intends to perceive or, simply, what β can perceive as α's informing. That is why the operational composition $\models_\alpha \circ \models_\beta$ or $\Dashv_\alpha \circ \Dashv_\beta$ which constitutes operator $\models$ or $\Dashv$ in the relation $\alpha \models \beta$ or $\beta \Dashv \alpha$ can be comprehended also as

[2.21]:
$$\begin{aligned}
&(\alpha \models \beta) \Rightarrow ((\alpha \models_\alpha \circ \models_\beta\ \beta) \vee\\
&\qquad (\alpha \not\models_\alpha \circ \models_\beta\ \beta) \vee\\
&\qquad (\alpha \models_\alpha \circ \not\models_\beta\ \beta) \vee\\
&\qquad (\alpha \not\models_\alpha \circ \not\models_\beta\ \beta));\\
&\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash\};\\
&\text{"}\not\models\text{"} \in \{\not\models, \nVDash, \nvdash, \nVdash\};\\
&\\
&((\beta\ \not\Dashv_\beta \circ \not\Dashv_\alpha\ \alpha) \vee\\
&\ (d\ \not\Dashv_\beta \circ \Dashv_\alpha\ \alpha) \vee\\
&\ (\beta\ \Dashv_\beta \circ \not\Dashv_\alpha\ \alpha) \vee\\
&\ (\beta\ \not\Dashv_\beta \circ \Dashv_\alpha\ \alpha)) \Leftarrow (\beta \Dashv \alpha);\\
&\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV\};\\
&\text{"}\not\Dashv\text{"} \in \{\not\Dashv, \not\DashV, \not\dashv, \not\dashV\}
\end{aligned}$$

The case $\alpha \models_\alpha \circ \models_\beta \beta$ or $\beta \Dashv_\beta \circ \Dashv_\alpha \alpha$ can be taken as the most appropriate form of informing from entity α to entity β, where both α and β are acting (informing) as useful informational partners. The informational transmitter α transmits ($\models_\alpha$ or $\Dashv_\alpha$) information which can be conditionally ($\models_\beta$ or $\Dashv_\beta$) accepted by the informational receiver β.

In the case $\alpha \not\models_\alpha \circ \models_\beta \beta$ or $\beta \Dashv_\beta \circ \not\Dashv_\alpha \alpha$, α is not entirely in the position to inform in such a way to β that β could accept information (intentionally or essentially) mediated from α. It could be said that α does not entirely fulfill the informational expectance or capability of β.

In the case $\alpha \models_\alpha \circ \not\models_\beta \beta$ or $\beta \not\Dashv_\beta \circ \Dashv_\alpha \alpha$, β is not entirely in the position to be informed in the way in which α informs or mediates information. It is possible to say that β does not entirely fulfill the informational expectance or ability of α.

In the last case $\alpha \not\models_\alpha \circ \not\models_\beta \beta$ or $\beta \not\Dashv_\beta \circ \not\Dashv_\alpha \alpha$, the informing (processing) $\alpha \models \beta$ or $\beta \Dashv \alpha$ does practically (more exactly, particularly) not exist, for α cannot mediate information which β could be capable to accept. However, it does not mean that another particular process of the form $\alpha \models \beta$ or $\beta \Dashv \alpha$ is not taking place.

This discussion shows how operators $\models$ and $\not\models$ could be understood to be symmetrical to each other. Thus,

[2.22]:
$$\begin{aligned}
&(\alpha \not\models \beta) \Rightarrow ((\alpha \not\models_\alpha \circ \not\models_\beta\ \beta) \vee\\
&\qquad (\alpha \models_\alpha \circ \not\models_\beta\ \beta) \vee\\
&\qquad (\alpha \not\models_\alpha \circ \models_\beta\ \beta) \vee\\
&\qquad (\alpha \models_\alpha \circ \models_\beta\ \beta));\\
&\text{"}\not\models\text{"} \in \{\not\models, \nVDash, \nvdash, \nVdash\};\\
&\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash\};\\
&\\
&((\beta\ \Dashv_\beta \circ \Dashv_\alpha\ \alpha) \vee\\
&\ (d\ \Dashv_\beta \circ \not\Dashv_\alpha\ \alpha) \vee\\
&\ (\beta\ \not\Dashv_\beta \circ \Dashv_\alpha\ \alpha) \vee\\
&\ (\beta\ \not\Dashv_\beta \circ \not\Dashv_\alpha\ \alpha)) \Leftarrow (\beta \not\Dashv \alpha);\\
&\text{"}\not\Dashv\text{"} \in \{\not\Dashv, \not\DashV, \not\dashv, \not\dashV\};\\
&\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV\}
\end{aligned}$$

The conclusion of this discussion is that general informational operators belonging to the classes $\models$, $\nvDash$, $\Dashv$, and $\not\Dashv$ are relative to each other and that it is possible to use them according to the occurring circumstances, appropriateness, and needs.

2.2.2. The Case 'α is informed'

Implicitly, in the case $\alpha \models \beta$, we have learned a bit on the nature of the case 'to be informed'. Let us examine this case analogously to the case 'to inform' into more detail.

The form $\models \alpha$ or $\alpha \Dashv$ says that α is informed. This formula is implicatively open in the following sense:

[2.23]:

$$(\models \alpha) \Rightarrow$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\xi, \eta, \ldots, \zeta \models \alpha));$$
$$\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$$

$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\alpha \Dashv \xi, \eta, \ldots, \zeta))$$
$$\Leftarrow (\alpha \Dashv);$$
$$\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$$

If α is informed, then there may exist some informational entities ξ, η, ... , ζ, which inform α. The consequence of this implication might be the implication

[2.24]:

$$(\models \alpha) \Rightarrow (\exists_\pi (\alpha \models \alpha));$$
$$\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$$

$$(\exists_\pi (\alpha \Dashv \alpha)) \Leftarrow (\alpha \Dashv);$$
$$\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$$

If α is informed, then it could be informed by itself. Logically, the following inverse implication can be adopted:

[2.25]:

$$(\xi, \eta, \ldots, \zeta \models \alpha) \Rightarrow (\models \alpha);$$
$$\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$$

$$(\alpha \Dashv) \Leftarrow (\alpha \Dashv \xi, \eta, \ldots, \zeta);$$
$$\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$$

If α is informed by informational entities ξ, η, ... , ζ, then it is informed in general too. The particular case of this implication is

[2.26]:

$$(\alpha \models \alpha) \Rightarrow (\models \alpha);$$
$$\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$$

$$(\alpha \Dashv) \Leftarrow (\alpha \Dashv \alpha);$$
$$\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$$

If α is informed in itself, then it is informed. Now, we can adopt the following complete list of implications proceeding from the previous discussion, which concern general informational operators (the so-called "informing" operators and the so-called "non-informing" ones), which perform (inform) from the left to the right and vice versa, i.e., operators $\models$, $\VDash$, $\vdash$, $\Vdash$, $\nvDash$, $\nVDash$, $\nvdash$, $\nVdash$, $\Dashv$, $\DashV$, $\dashv$, $\dashV$, $\not\Dashv$, $\not\DashV$, $\not\dashv$, and $\not\dashV$:

[2.27]:

$$(\models \alpha) \Rightarrow$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\xi, \eta, \ldots, \zeta \models \alpha));$$
$$(\VDash \alpha) \Rightarrow$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\xi, \eta, \ldots, \zeta \VDash \alpha));$$
$$(\vdash \alpha) \Rightarrow$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\xi, \eta, \ldots, \zeta \vdash \alpha));$$
$$(\Vdash \alpha) \Rightarrow$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\xi, \eta, \ldots, \zeta \Vdash \alpha));$$

$$(\nvDash \alpha) \Rightarrow$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\xi, \eta, \ldots, \zeta \nvDash \alpha));$$
$$(\nVDash \alpha) \Rightarrow$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\xi, \eta, \ldots, \zeta \nVDash \alpha));$$
$$(\nvdash \alpha) \Rightarrow$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\xi,. \eta, \ldots, \zeta \nvdash \alpha));$$
$$(\nVdash \alpha) \Rightarrow$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\xi, \eta, \ldots, \zeta \nVdash \alpha));$$

$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\alpha \Dashv \xi, \eta, \ldots, \zeta))$$
$$\Leftarrow (\alpha \Dashv);$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\alpha \DashV \xi, \eta, \ldots, \zeta))$$
$$\Leftarrow (\alpha \DashV);$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\alpha \dashv \xi, \eta, \ldots, \zeta))$$
$$\Leftarrow (\alpha \dashv);$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\alpha \dashV \xi, \eta, \ldots, \zeta))$$
$$\Leftarrow (\alpha \dashV);$$

$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\alpha \not\Dashv \xi, \eta, \ldots, \zeta))$$
$$\Leftarrow (\alpha \not\Dashv);$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\alpha \not\DashV \xi, \eta, \ldots, \zeta))$$
$$\Leftarrow (\alpha \not\DashV);$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\alpha \not\dashv \xi, \eta, \ldots, \zeta))$$
$$\Leftarrow (\alpha \not\dashv);$$
$$((\exists_\pi \xi, \eta, \ldots, \zeta).(\alpha \not\dashV \xi, \eta, \ldots, \zeta))$$
$$\Leftarrow (\alpha \not\dashV)$$

Again, it becomes evident how it is possible to particularize the general informational operator $\models$ by the so-called general operators (metaoperators) $\models$ and $\nvDash$, general parallel operators $\VDash$ and $\nVDash$, general cyclic operators $\vdash$ and $\nvdash$, and general parallel-cyclic operators $\Vdash$ and $\nVdash$. Thus,

[2.28]:

$$(\models \alpha) =_{\mathrm{Df}} ((\VDash \alpha) \vee (\vdash \alpha) \vee (\Vdash \alpha));$$
$$(\nvDash \alpha) =_{\mathrm{Df}} ((\nVDash \alpha) \vee (\nvdash \alpha) \vee (\nVdash \alpha));$$
$$(\models \alpha) \Rightarrow ((\models \alpha) \vee (\nvDash \alpha))$$

This definition says simply that α is informed or not informed in a parallel, cyclic, and/or parallel-cyclic manner. Again, the parallel-cyclic case is to be understood as a parallel and cyclically perplexing complex mode of informing of an informational entity.

As previously, the performance of operator $\Dashv$ in the case "to be informed" can be defined. Again, this operator demon-strates the diversity and alternativeness against the general operator $\models$. It can be showed how in cases of anthropological (linguistic) discourse this explicit informational operator becomes the necessity, delivering the unrevealed informing which lurks or waits in the background of each living informing and non-informing (for instance, as skepticism, unbelief, the Other, or simply counter-informing). Thus, adequately to [2.28.] there is

[2.29]: $(\alpha \Dashv) =_{\mathrm{Df}} ((\alpha \DashV) \vee (\alpha \dashv) \vee (\alpha \dashV));$
$(\alpha \not\Dashv) =_{\mathrm{Df}} ((\alpha \not\DashV) \vee (\alpha \not\dashv) \vee (\alpha \not\dashV));$
$(\alpha \Dashv) \Rightarrow ((\alpha \Dashv) \vee (\alpha \not\Dashv));$

Again, it is to understand that if operators $\vDash$, $\VDash$, $\vdash$, $\Vdash$, $\nvDash$, $\nVDash$, $\nvdash$, and $\nVdash$ inform in one way, then their counterparts $\Dashv$, $\DashV$, $\dashv$, $\dashV$, $\not\Dashv$, $\not\DashV$, $\not\dashv$, and $\not\dashV$ inform in another way. This verbal difference between the first and the second case (class) of various informational operators ensures that the alternative horizon of informing in the case "to be informed" comes explicitly (formally) into existence too. Further, it is to understand that $\vDash \alpha$ can have the meaning (or metameaning) of $\alpha \Dashv$ too.

The first two formulas in expressions [2.28] and [2.29] in the case "to be informed" state that in the domain of informational connectedness, which can be at most a cyclic, parallel, parallel-cyclic, parallel-serial, or parallel-sequential structure, general informing or non-informing is nothing else than a type of these kinds of informing. The last formula in [2.28] and [2.29] implicates merely the metarole (metameaning) of operators $\vDash$ and $\Dashv$, respectively. At last, operator $\vDash$ can take over the role to be the only informational metaoperator. Thus, for instance, $\vDash \alpha$ can have the meaning of $\vDash \alpha$ as well as of $\alpha \Dashv$, etc.

Now, it is possible to continue the discourse of unary informational operators concerning formula $\vDash \alpha$ as an operand.

By definition, if α marks an informational entity, then α is informed. Inductively, on the basis of this fact, it is possible to construct an indefinite number of implications, namely,

[2.30]: $\alpha \Rightarrow (\vDash \alpha);$
$(\vDash \alpha) \Rightarrow (\vDash (\vDash \alpha));$
$(\vDash (\vDash \alpha)) \Rightarrow (\vDash (\vDash (\vDash \alpha)));$
$. \; . \; . \; . \; . \; . \; . \; . \; . \; . \; . \; . \; . \; . \; .$
$\text{"}\vDash\text{"} \in \{\vDash, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\}$

Thus, by the property of transitivity, there is

[2.31]: $\alpha \Rightarrow (\vDash \ldots (\vDash (\vDash \alpha)) \ldots);$
$\text{"}\vDash\text{"} \in \{\vDash, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\}$

This formula says that informational entity can be informed in all possible ways concerning informational operators $\vDash$, $\VDash$, $\vdash$, $\Vdash$, $\nvDash$, $\nVDash$, $\nvdash$, $\nVdash$. It says again in a formally explicit way that α can be perplexedly complex in respect to parallel, serial, and parallel-serial informing in the sense "to be informed". Thus, informational entity α marks a system or network of informational processes which constitute it informationally. By this, an informational entity becomes a systemic notion or notion of an informational network.

Formula [2.31] is an explicit expression (through the use of explicit informational operators of the type $\vDash$) of the arising nature of informational entity α in the sense "to be informed". Besides of this explicitness of informational arising, there exists, by definition, also the operational implicitness (an implicit form of informational arising) of an informational entity α. This operational implicitness is coming to the surface when, for instance, an informational entity marked by α, is decomposed, and thus explicating its informational components (a composition of informational operators and operands). The origin of this discussion can be the following:

[2.32]: $(\vDash \alpha) \Rightarrow \mathfrak{I}_{\rightarrow}(\alpha)$ or simply
$(\vDash \alpha) \Rightarrow \mathfrak{I}(\alpha)$

where $\mathfrak{I}$ (or $\mathfrak{I}_{\rightarrow}$) is the implicit operator of informing or non-informing (or informing or non-informing from the left to the right). $\mathfrak{I}(\alpha)$ is a sort of functional expression which points out the operational component $\mathfrak{I}$ of the entity α. Obviously, inductively, the last expression can be expanded (decomposed), for instance, into

[2.33]:
$(\vDash \alpha) \Rightarrow \mathfrak{I}(\alpha);$
$\mathfrak{I}(\alpha) \Rightarrow \mathfrak{I}(\mathfrak{I} \ldots (\mathfrak{I}(\alpha)) \ldots);$
$\mathfrak{I} \in \{\mathfrak{I}_{\vDash}, \mathfrak{I}_{\VDash}, \mathfrak{I}_{\vdash}, \mathfrak{I}_{\Vdash}, \mathfrak{I}_{\nvDash}, \mathfrak{I}_{\nVDash}, \mathfrak{I}_{\nvdash}, \mathfrak{I}_{\nVdash}\}$

where $\mathfrak{I}_{\vDash}$, $\mathfrak{I}_{\VDash}$, $\mathfrak{I}_{\vdash}$, $\mathfrak{I}_{\Vdash}$, $\mathfrak{I}_{\nvDash}$, $\mathfrak{I}_{\nVDash}$, $\mathfrak{I}_{\nvdash}$, and $\mathfrak{I}_{\nVdash}$ mark the so called general ($\vDash$), parallel ($\VDash$), cyclic ($\vdash$), and parallel-cyclic case ($\Vdash$) of informing and general ($\nvDash$), parallel ($\nVDash$), cyclic ($\nvdash$), and parallel-cyclic case ($\nVdash$) of non-informing, respectively.

Formulas [2.30] - [2.33] can be repeated for the so-called alternative case of "to be informed" concerning operators $\Dashv$, $\DashV$, $\dashv$, $\dashV$, $\not\Dashv$, $\not\DashV$, $\not\dashv$, and $\not\dashV$. Again, these cases can be expressed in a strict symmetric (right-left) form, for instance:

[2.34]: $(\alpha \Dashv) \Leftarrow \alpha;$
$((\alpha \Dashv) \Dashv) \Leftarrow (\alpha \Dashv);$
$(((\alpha \Dashv) \Dashv) \Dashv) \Leftarrow ((\alpha \Dashv) \Dashv);$
$. \; . \; . \; . \; . \; . \; . \; . \; . \; . \; . \; . \; . \; . \; .$
$\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$

This, by the property of transitivity, yields

[2.35]: $(\ldots ((\alpha \Dashv) \Dashv) \ldots \Dashv) \Leftarrow \alpha;$
$\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$

Analogously to [2.32] the following alternative formula is obtained:

[2.36]: $\mathfrak{I}_{\leftarrow}(\alpha) \Leftarrow (\alpha \Dashv)$ or simply
$\mathfrak{I}'(\alpha) \Leftarrow (\alpha \Dashv)$

where $\mathfrak{I}'$ (or $\mathfrak{I}_{\leftarrow}$) is the implicit alternative operator of informing or non-informing (from the right to the left). $\mathfrak{I}'(\alpha)$ is the alternative functional expression which points out the alternative operational component $\mathfrak{I}'$ of the entity α in the case "to be informed". Obviously, inductively, the last expression can be expanded (decomposed), for instance, into

[2.37]:
$\mathfrak{I}'(\alpha) \Leftarrow (\alpha \Dashv);$

$['(\mathfrak{I}' \ \ldots \ (\mathfrak{I}'(\alpha)) \ \ldots \ ) \Leftarrow \mathfrak{I}'(\alpha);$
$\mathfrak{I}' \in \{\mathfrak{I}'_{\Dashv}, \mathfrak{I}'_{\DashV}, \mathfrak{I}'_{\dashv}, \mathfrak{I}'_{\dashV}, \mathfrak{I}'_{\not\Dashv}, \mathfrak{I}'_{\not\DashV}, \mathfrak{I}'_{\not\dashv}, \mathfrak{I}'_{\not\dashV}\}$

where $\mathfrak{I}'_{\Dashv}$, $\mathfrak{I}'_{\DashV}$, $\mathfrak{I}'_{\dashv}$, $\mathfrak{I}'_{\dashV}$, $\mathfrak{I}'_{\not\Dashv}$, $\mathfrak{I}'_{\not\DashV}$, $\mathfrak{I}'_{\not\dashv}$, and $\mathfrak{I}'_{\not\dashV}$ mark the so called alternative general ($\Dashv$), parallel ($\DashV$), cyclic ($\dashv$), and parallel-cyclic case ($\dashV$) of informing and alternative general ($\not\Dashv$), parallel ($\not\DashV$), cyclic ($\not\dashv$), and parallel-cyclic case ($\not\dashV$) of non-informing, respectively. In this manner the alternativeness of informing and non-informing in the case "to be informed" is preserved also in an informationally implicit way.

Let us see how the "logically" pure meaning of negation can become informationally contestable, questionable, and insufficient in the case of "to be non-informed". Let be

[2.38]: $\neg(\models \alpha) \Rightarrow (\nvDash \alpha);$
$"\models" \in \{\models, \VDash, \vdash, \Vdash\};$
$"\nvDash" \in \{\nvDash, \nVDash, \nvdash; \nVdash\};$

$(\alpha \not\Dashv) \Leftarrow (\neg(\alpha \Dashv);$
$"\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV\};$
$"\not\Dashv" \in \{\not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$

Concerning the last formula in which $\nvDash$ is the operator of non-informing, it is possible to develop the following questions:

(1) If α marks an informational entity which is informed ($\models$ or $\Dashv$), how is this entity not informed ($\nvDash$ or $\not\Dashv$)? Again, evidently, $\neg$ as an informational operator of negation does not possess a totally negational meaning (operational power).

(2) How is α not informed ($\nvDash$ or $\not\Dashv$) and what does this non-informing mean?

(3) If it is said that α is not informed in a certain way, then α is either inhibited or is not capable to be informed in a certain way. Thus, it is possible to say, for instance, that α is informed in an inhibitory manner. This fact yields

[2.39]: $(\nvDash \alpha) \Rightarrow (\models_i \alpha);$
$"\nvDash" \in \{\nvDash, \nVDash, \nvdash, \nVdash\};$
$"\models_i" \in \{\models_i, \VDash_i, \vdash_i, \Vdash_i\};$

$(\alpha \Dashv_i) \Leftarrow (\alpha \not\Dashv);$
$"\not\Dashv_i" \in \{\not\Dashv_i, \not\DashV_i, \not\dashv_i, \not\dashV_n\};$
$"\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV\}$

The so-called non-informing ($\nvDash$ or $\not\Dashv$) can be understood as inhibitive informing.

(4) The reverse can also be certain. If α is informed, then α is not informed in its all embracing informational variety or entirety. It is informed only in a certain way and not in all possible (universal) ways. Thus,

[2.40]: $(\models \alpha) \Rightarrow (\nvDash_n \alpha);$
$"\models" \in \{\models, \VDash, \vdash, \Vdash\};$
$"\nvDash_n" \in \{\nvDash_n, \nVDash_n, \nvdash_n, \nVdash_u\};$

$(\alpha \not\Dashv_n) \Leftarrow (\alpha \Dashv);$
$"\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV\};$
$"\not\Dashv_n" \in \{\not\Dashv_n, \not\DashV_n, \not\dashv_n, \not\dashV_n\}$

where $\nvDash_n$ and $\not\Dashv_n$ mark the non-universal informing. This fact can be explained by the intentional nature of informational arising regardless of a certain informational entity. Any informational entity, as a process of its informational existing and arising of information, possesses a certain orientation or intentionality to be informed and only in this manner can inform or can be informed.

(5) If β is informed by a certain informational entity α, where $\alpha \models \beta$ or $\beta \Dashv \alpha$, and entity β is not "sufficiently" sensitive to the informing of α, then β is not "adequately" informed by α, i.e., $\alpha \nvDash \beta$ or $\beta \not\Dashv \alpha$. We have already examined this case by the expressions [2.21] and [2.22].

Again, the conclusion of this discussion is that general informational operators belonging to the classes $\models$, $\nvDash$, $\Dashv$, and $\not\Dashv$ are relative to each other and that it is possible to use them according to the occurring circumstances, appropriateness, and needs.

2.3. Implications and Definitions Concerning Binary Informational Operators

Because of informing of an informational entity α, there may exist an informational entity β, which can be reached by the informing of α. In this case we say that α informs β or β is informed by α. It seems that every unary informational operator appears to be at least the binary one and, in general, the multiplex informational operator. In fact, the emphasizing of the unary nature of an informational operator is nothing else than concealing of informational source or sink in regard to the operand, being informationally connected with the unary operator.

The form $\alpha \models \beta$ or $\beta \Dashv \alpha$ says that α informs β or that β is informed by α. But, these formulas are implicatively open in the following sense:

[2.41]:
$(\alpha \models \beta) \Rightarrow ((\alpha \models \beta) \models);$
$((\alpha \models \beta) \models) \ \mathrm{I}$
$((\exists_\pi \ \xi, \eta, \ \ldots \ , \zeta).((\alpha \models \beta) \models \xi, \eta, \ \ldots \ , \zeta));$
$(\alpha \models \beta) \Rightarrow (\models (\alpha \models \beta));$
$(\models (\alpha \models \beta)) \Rightarrow$
$((\exists_\pi \ \xi, \eta, \ \ldots \ , \zeta).(\xi, \eta, \ \ldots \ , \zeta \models (\alpha \models \beta)));$
$"\models" \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$

$(\Dashv (\beta \Dashv \alpha)) \Leftarrow (\beta \Dashv \alpha);$
$((\exists_\pi \ \xi, \eta, \ \ldots \ , \zeta).(\xi, \eta, \ \ldots \ , \zeta \Dashv (\beta \Dashv \alpha))$
$\quad \Leftarrow (\Dashv (\beta \Dashv \alpha));$
$(\beta \Dashv \alpha) \Dashv) \Leftarrow (\beta \Dashv \alpha);$
$((\exists_\pi \ \xi, \eta, \ \ldots \ , \zeta).((\beta \Dashv \alpha) \Dashv \xi, \eta, \ \ldots \ , \zeta))$
$\quad \Leftarrow ((\beta \Dashv \alpha) \Dashv);$
$"\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}$

If α informs β, i.e., $\alpha \models \beta$ or $\beta \Dashv \alpha$ then there may exist some informational entities ξ, η, $\ldots$, ζ, which are informed by the process $\alpha \models \beta$ or $\beta \Dashv \alpha$. The consequence of these implications might be

[2.42]:
$$\begin{array}{l}
(\alpha \models \beta) \Rightarrow (\exists_{\pi}\ ((\alpha \models \beta) \models (\alpha \models \beta)));\\
\text{"}\models\text{"} \in \{\models, \Vert\!\models, \vdash, \Vdash, \not\models, \Vert\!\not\models, \nvdash, \nVdash\};\\
\\
(\exists_{\pi}\ ((\beta \Dashv \alpha) \Dashv (\beta \Dashv \alpha))) \Leftarrow (\beta \Dashv \alpha);\\
\text{"}\Dashv\text{"} \in \{\Dashv, \Dashv\!\Vert, \dashv, \dashv\!\Vert, \not\Dashv, \not\Dashv\!\Vert, \not\dashv, \not\dashv\!\Vert\}
\end{array}$$

If α informs β∴ $\notin\delta\varepsilon\delta$∴ $\alpha \models \beta$ or $\beta \Dashv \alpha$, then these processes could inform itself.

It becomes evident (similar to the case of unary informational operators) that it is possible to particularize the general informational operator $\models$ by the so-called general operators (metaoperators) $\models$ and $\not\models$, general parallel operators $\Vert\!\models$ and $\Vert\!\not\models$, general cyclic operators $\vdash$ and $\nvdash$, and general parallel-cyclic operators $\Vdash$ and $\nVdash$. Thus,

[2.43]:
$$\begin{array}{l}
(\alpha \models \beta) =_{\mathrm{Df}} ((\alpha \Vert\!\models \beta) \vee (\alpha \vdash \beta) \vee (\alpha \Vdash \beta));\\
(\alpha \not\models \beta) =_{\mathrm{Df}} ((\alpha \Vert\!\not\models \beta) \vee (\alpha \nvdash \beta) \vee (\alpha \nVdash \beta));\\
(\alpha \models \beta) \Rightarrow ((\alpha \models \beta) \vee (\alpha \not\models \beta))
\end{array}$$

This definition says that α informs or does not inform β in a parallel, cyclic, and/or parallel-cyclic manner. The parallel-cyclic case is to be understood as a parallel and cyclically perplexing complex mode of informing of an informational entity.

In the similar way the performance of operator $\Dashv$ can be defined. This operator demon-strates the diversity and alternativeness against the general operator $\models$. Thus, adequately to [2.43] there is

[2.44]:
$$\begin{array}{l}
(\beta \Dashv \alpha) =_{\mathrm{Df}} ((\beta \Dashv\!\Vert \alpha) \vee (\beta \dashv \alpha) \vee (\beta \dashv\!\Vert \alpha));\\
(\beta \not\Dashv \alpha) =_{\mathrm{Df}} ((\beta \not\Dashv\!\Vert \alpha) \vee (\beta \not\dashv \alpha) \vee (\beta \not\dashv\!\Vert \alpha));\\
(\beta \Dashv \alpha) \Rightarrow ((\beta \Dashv \alpha) \vee (\beta \not\Dashv \alpha));
\end{array}$$

The first two formulas in expressions [2.43] and [2.44] state that in the domain of informational connectedness, which can be at most a cyclic, parallel, parallel-cyclic, parallel-serial, or parallel-sequential structure, general informing or non-informing is nothing else than a type of these kinds of informing. The last formula in [2.43] and [2.44] implicates merely the metarole (metameaning) of operators $\models$ and $\Dashv$, respectively. At last, operator $\models$ can take over the role to be the only informational metaoperator. Thus, for instance, $\alpha \models$ can have the meaning of $\alpha \models$ as well as of $\Dashv \alpha$, etc.

By definition, if $\alpha \models \beta$ or $\beta \Dashv \alpha$ marks an informational process, then α informs β in one or another way. Inductively, on the basis of this fact, it is possible to construct an indefinite number of implications, namely,

[2.45]:
$$\begin{array}{l}
(\alpha \models \beta) \Rightarrow ((\alpha \models \beta) \models);\\
((\alpha \models \beta) \models) \Rightarrow (((\alpha \models \beta) \models) \models);\\
(((\alpha \models \beta) \models) \models) \Rightarrow ((((\alpha \models \beta) \models) \models) \models);\\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\
(\alpha \models \beta) \Rightarrow (\models (\alpha \models \beta));\\
(\models (\alpha \models \beta)) \Rightarrow (\models (\models (\alpha \models \beta)));\\
(\models (\models (\alpha \models \beta))) \Rightarrow (\models (\models (\models (\alpha \models \beta))));\\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\
\text{"}\models\text{"} \in \{\models, \Vert\!\models, \vdash, \Vdash, \not\models, \Vert\!\not\models, \nvdash, \nVdash\};\\
\\
(\Dashv (\beta \Dashv \alpha)) \Leftarrow (\beta \Dashv \alpha);\\
(\Dashv (\Dashv (\beta \Dashv \alpha))) \Leftarrow (\Dashv (\beta \Dashv \alpha));\\
(\Dashv (\Dashv (\Dashv (\beta \Dashv \alpha)))) \Leftarrow (\Dashv (\Dashv (\beta \Dashv \alpha)));\\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\
((\beta \Dashv \alpha) \Dashv) \Leftarrow (\beta \Dashv \alpha);\\
(((\beta \Dashv \alpha) \Dashv) \Dashv) \Leftarrow ((\beta \Dashv \alpha) \Dashv);\\
((((\beta \Dashv \alpha) \Dashv) \Dashv) \Dashv) \Leftarrow (((\beta \Dashv \alpha) \Dashv) \Dashv);\\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\
\text{"}\Dashv\text{"} \in \{\Dashv, \Dashv\!\Vert, \dashv, \dashv\!\Vert, \not\Dashv, \not\Dashv\!\Vert, \not\dashv, \not\dashv\!\Vert\}
\end{array}$$

Thus, by the property of transitivity, there is

[2.46]:
$$\begin{array}{l}
(\alpha \models \beta) \Rightarrow (\ \ldots\ (((\alpha \models \beta) \models) \models)\ \ldots \models);\\
(\alpha \models \beta) \Rightarrow (\models\ \ldots\ (\models (\models (\alpha \models \beta)))\ \ldots\ );\\
\text{"}\models\text{"} \in \{\models, \Vert\!\models, \vdash, \Vdash, \not\models, \Vert\!\not\models, \nvdash, \nVdash\};\\
\\
(\Dashv\ \ldots\ (\Dashv (\Dashv (\beta \Dashv \alpha)))\ \ldots\ ) \Leftarrow (\beta \Dashv \alpha);\\
(\ \ldots\ (((\beta \Dashv \alpha) \Dashv) \Dashv)\ \ldots \Dashv) \Leftarrow (\beta \Dashv \alpha);\\
\text{"}\Dashv\text{"} \in \{\Dashv, \Dashv\!\Vert, \dashv, \dashv\!\Vert, \not\Dashv, \not\Dashv\!\Vert, \not\dashv, \not\dashv\!\Vert\}
\end{array}$$

Besides of this explicitness of informational arising, there exists, by definition, also the operational implicitness (an implicit form of informational arising) of an informational process $\alpha \models \beta$ or $\beta \Dashv \alpha$. This operational implicitness is coming to the surface when, for instance, an informational process marked by $\alpha \models \beta$ or $\beta \Dashv \alpha$, is decomposed, and thus explicating its informational components (a composition of informational operators and operands). Again, the origin of this discussion can be the following:

[2.47]:
$$\begin{array}{l}
((\alpha \models \beta) \models) \Rightarrow \mathfrak{I}_{\rightarrow}(\alpha \models \beta) \text{ or simply}\\
((\alpha \models \beta) \models) \Rightarrow \mathfrak{I}(\alpha \models \beta);\\
(\models (\alpha \models \beta)) \Rightarrow \mathfrak{I}_{\rightarrow}(\alpha \models \beta) \text{ or simply}\\
(\models \pi\alpha \models \beta)) \Rightarrow \mathfrak{I}(\alpha \models \beta);\\
\\
\mathfrak{I}_{\leftarrow}(\beta \Dashv \alpha) \Leftarrow (\Dashv \pi\beta \Dashv \alpha)) \text{ or simply}\\
\mathfrak{I}(\beta \Dashv \alpha) \Leftarrow (\Dashv \pi\beta \Dashv \alpha));\\
\mathfrak{I}_{\leftarrow}(\beta \Dashv \alpha) \Leftarrow ((\beta \Dashv \alpha) \Dashv) \text{ or simply}\\
\mathfrak{I}(\beta \Dashv \alpha) \Leftarrow ((\beta \Dashv \alpha) \Dashv);\\
\\
\text{"}\models\text{"} \in \{\models, \Vert\!\models, \vdash, \Vdash, \not\models, \Vert\!\not\models, \nvdash, \nVdash\};\\
\text{"}\Dashv\text{"} \in \{\Dashv, \Dashv\!\Vert, \dashv, \dashv\!\Vert, \not\Dashv, \not\Dashv\!\Vert, \not\dashv, \not\dashv\!\Vert\}
\end{array}$$

where $\mathfrak{I}$ (or $\mathfrak{I}_{\rightarrow}$ or $\mathfrak{I}_{\leftarrow}$) is the implicit operator of informing or non-informing (or informing or non-informing from the left to the right or from the right to left). $\mathfrak{I}(\alpha \models \beta)$ or $\mathfrak{I}(\beta \Dashv \alpha)$ is a functional expression which points out the operational component $\mathfrak{I}$ of the process $\alpha \models \beta$ or $\beta \Dashv \alpha$. Obviously, inductively, the last expressions can be expanded (decomposed), for instance, into

[2.48]:
$$\begin{array}{l}
((\alpha \models \beta) \models) \Rightarrow \mathfrak{I}(\alpha \models \beta);\\
(\models (\alpha \models \beta)) \Rightarrow \mathfrak{I}(\alpha \models \beta);\\
\mathfrak{I}(\alpha \models \beta) \Rightarrow \mathfrak{I}(\mathfrak{I}\ \ldots\ (\mathfrak{I}(\alpha \models \beta))\ \ldots\ );\\
\mathfrak{I} \in \{\mathfrak{I}_{\models}, \mathfrak{I}_{\Vert\models}, \mathfrak{I}_{\vdash}, \mathfrak{I}_{\Vdash}, \mathfrak{I}_{\not\models}, \mathfrak{I}_{\Vert\not\models}, \mathfrak{I}_{\nvdash}, \mathfrak{I}_{\nVdash}\};\\
\\
\mathfrak{I}(\beta \Dashv \alpha) \Leftarrow (\Dashv (\beta \Dashv \alpha));\\
\mathfrak{I}(\beta \Dashv \alpha) \Leftarrow ((\beta \Dashv \alpha) \Dashv);
\end{array}$$

$$\mathfrak{I}(\mathfrak{I}\ \ldots\ (\mathfrak{I}(\beta \Dashv \alpha))\ \ldots\ ) \Leftarrow \mathfrak{I}(\beta \Dashv \alpha);$$
$$\mathfrak{I} \in \{\mathfrak{I}_{\Dashv}, \mathfrak{I}_{\DashV}, \mathfrak{I}_{\dashv}, \mathfrak{I}_{\dashV}, \mathfrak{I}_{\nDashv}, \mathfrak{I}_{\nDashV}, \mathfrak{I}_{\ndashv}, \mathfrak{I}_{\ndashV}\}$$

where $\mathfrak{I}_{\models}$, $\mathfrak{I}_{\VDash}$, $\mathfrak{I}_{\vdash}$, $\mathfrak{I}_{\Vdash}$, $\mathfrak{I}_{\nvDash}$, $\mathfrak{I}_{\nVDash}$, $\mathfrak{I}_{\nvdash}$, $\mathfrak{I}_{\nVdash}$ and $\mathfrak{I}_{\Dashv}$, $\mathfrak{I}_{\DashV}$, $\mathfrak{I}_{\dashv}$, $\mathfrak{I}_{\dashV}$, $\mathfrak{I}_{\nDashv}$, $\mathfrak{I}_{\nDashV}$, $\mathfrak{I}_{\ndashv}$, $\mathfrak{I}_{\ndashV}$ mark the so called general ($\models$ and $\Dashv$), parallel ($\VDash$ and $\DashV$), cyclic ($\vdash$ and $\dashv$), and parallel-cyclic case ($\Vdash$ and $\dashV$) of informing and general ($\nvDash$ and $\nDashv$), parallel ($\nVDash$ and $\nDashV$), cyclic ($\nvdash$ and $\ndashv$), and parallel-cyclic case ($\nVdash$ and $\ndashV$) of non-informing, respectively.

2.4. Implications and Definitions Concerning Multiplex Informational Operators

Inductively, from binary informational operators it is possible to proceed to the case dealing with multiplex informational operators. In principle, by definition, each informational operator can perform as a unary, binary, or multiplex operator. The nature of the multiplex operators has to be explained. In the case of a unary operator, its operativeness remains open in the sense that there exist possibilities of its connection to other, explicitly hidden operands. In the case of a binary operator, the operator's connectivity concerns the left and the right operand, however, the openness in the sense of a unary operator to other, yet unrevealed operands still exists. Only in the case of a multiplex operator, the dilemma of openness can vanish, for multiplex operator, in any case, concerns a multiple of informational operands, however, remains still open to the unrevealed operands.

Because of informing of informational entities (operands), marked by $\alpha, \beta, \ldots, \gamma$, there may exist informational entities (operands), marked by $\xi, \eta, \ldots, \zeta$, which can be informationally reached by the informing of entities $\alpha, \beta, \ldots, \gamma$. In this case we say that $\alpha, \beta, \ldots, \gamma$ inform $\xi, \eta, \ldots, \zeta$ or that $\xi, \eta, \ldots, \zeta$ are informed by $\alpha, \beta, \ldots, \gamma$. In this respect it seems that every unary or binary informational operator appears to be also a (hidden, in its entirety unrevealed) multiplex informational operator.

The form $\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta$ or the form $\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma$ says that entities $\alpha, \beta, \ldots, \gamma$ perplexedly inform entities $\xi, \eta, \ldots, \zeta$ or that entities $\xi, \eta, \ldots, \zeta$ are perplexdly informed by entities $\alpha, \beta, \ldots, \gamma$. But, these formulas are inductively implicatively open in the following sense:

[2.49]:

$$(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \Rightarrow$$
$$((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models);$$

$$((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models) \Rightarrow$$
$$((\exists_\pi\ \varphi, \psi, \ldots, \tau).$$
$$((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models$$
$$\varphi, \psi, \ldots, \tau));$$

$$(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \Rightarrow$$
$$(\models (\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta));$$

$$(\models (\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta)) \Rightarrow$$
$$((\exists_\pi\ \varphi, \psi, \ldots, \tau).$$
$$(\varphi, \psi, \ldots, \tau \models$$
$$(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta)));$$

$$\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$$

$$(\Dashv (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma))$$
$$\Leftarrow (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma);$$

$$((\exists_\pi\ \varphi, \psi, \ldots, \tau).$$
$$(\varphi, \psi, \ldots, \tau$$
$$\Dashv (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma))$$
$$\Leftarrow (\Dashv (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma));$$

$$(\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \Dashv)$$
$$\Leftarrow (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma);$$

$$((\exists_\pi\ \varphi, \psi, \ldots, \tau).$$
$$((\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma)$$
$$\Dashv \varphi, \psi, \ldots, \tau))$$
$$\Leftarrow ((\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \Dashv);$$

$$\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \nDashv, \nDashV, \ndashv, \ndashV\}$$

If $\alpha, \beta, \ldots, \gamma$ inform $\xi, \eta, \ldots, \zeta$, i.e., $\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta$ or $\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma$ then there may exist some informational entities $\varphi, \psi, \ldots, \tau$, which are informed by the process $\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta$ or $\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma$. The consequence of these implications might be

[2.50]: $(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \Rightarrow$

$$(\exists_\pi\ ((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models$$
$$(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta)));$$
$$\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$$

$$(\exists_\pi\ ((\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma)$$
$$\Dashv (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma)))$$
$$\Leftarrow (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma);$$
$$\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \nDashv, \nDashV, \ndashv, \ndashV\}$$

If $\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta$ inform $\xi, \eta, \ldots, \zeta$, i.e., $\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta$ or $\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma$, then these processes could inform itself.

Again, it becomes evident (similar to the case of unary informational operators) that it is possible to particularize the multiplex general informational operator $\models$ by the so-called multiplex general operators (metaoperators) $\models$ and $\nvDash$, multiplex general parallel operators $\VDash$ and $\nVDash$, multiplex general cyclic operators $\vdash$ and $\nvdash$, and multiplex general parallel-cyclic operators $\Vdash$ and $\nVdash$. Thus,

[2.51]:

$$(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) =_{\mathrm{Df}}$$
$$((\alpha, \beta, \ldots, \gamma \VDash \xi, \eta, \ldots, \zeta) \vee$$
$$(\alpha, \beta, \ldots, \gamma \vdash \xi, \eta, \ldots, \zeta) \vee$$
$$(\alpha, \beta, \ldots, \gamma \Vdash \xi, \eta, \ldots, \zeta));$$

$$\begin{array}{l}
(\alpha, \beta, \ldots, \gamma \nvDash \xi, \eta, \ldots, \zeta) =_{\mathrm{Df}} \\
((\alpha, \beta, \ldots, \gamma \nVDash \xi, \eta, \ldots, \zeta) \vee \\
\;(\alpha, \beta, \ldots, \gamma \nvdash \xi, \eta, \ldots, \zeta) \vee \\
\;(\alpha, \beta, \ldots, \gamma \nVdash \xi, \eta, \ldots, \zeta));
\end{array}$$

$$\begin{array}{l}
(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \Rightarrow \\
((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \vee \\
\;(\alpha, \beta, \ldots, \gamma \nvDash \xi, \eta, \ldots, \zeta))
\end{array}$$

This definition says that α, β, ... , γ inform or do not inform ξ, η, ... , ζ in a parallel, cyclic, and/or parallel-cyclic manner. The parallel-cyclic case is to be understood as a parallel and cyclically perplexing complex mode of informing of an informational entity.

In the similar way the multiplex performance of operator $\Dashv$ can be defined. This operator demon-strates the diversity and alternativeness against the general multiplex operator $\models$. Thus, adequately to [2.51], there is

[2.52]:

$$\begin{array}{l}
(\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) =_{\mathrm{Df}} \\
((\xi, \eta, \ldots, \zeta \DashV \alpha, \beta, \ldots, \gamma) \vee \\
\;(\xi, \eta, \ldots, \zeta \dashv \alpha, \beta, \ldots, \gamma) \vee \\
\;(\xi, \eta, \ldots, \zeta \dashV \alpha, \beta, \ldots, \gamma));
\end{array}$$

$$\begin{array}{l}
(\xi, \eta, \ldots, \zeta \not\Dashv \alpha, \beta, \ldots, \gamma) =_{\mathrm{Df}} \\
((\xi, \eta, \ldots, \zeta \not\DashV \alpha, \beta, \ldots, \gamma) \vee \\
\;(\xi, \eta, \ldots, \zeta \not\dashv \alpha, \beta, \ldots, \gamma) \vee \\
\;(\xi, \eta, \ldots, \zeta \not\dashV \alpha, \beta, \ldots, \gamma));
\end{array}$$

$$\begin{array}{l}
(\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \Rightarrow \\
((\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \vee \\
\;(\xi, \eta, \ldots, \zeta \not\Dashv \alpha, \beta, \ldots, \gamma));
\end{array}$$

These cases of formulas concern multiplex informational operators in the similar way as were the cases of unary and binary informational operators.

By definition, if $\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta$ or $\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma$ marks an informational process, then α, β, ... , γ inform ξ, η, ... , ζ in one or another way. Inductively, on the basis of this fact, it is possible to construct an indefinite number of implications, namely,

[2.53]:

$$\begin{array}{l}
(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \Rightarrow \\
((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models); \\
((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models) \Rightarrow \\
(((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models) \models); \\
(((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models) \models) \Rightarrow \\
((((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models) \models) \models); \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \Rightarrow \\
(\models (\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta)); \\
(\models (\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta)) \Rightarrow \\
(\models (\models (\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta))); \\
(\models (\models (\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta))) \Rightarrow \\
(\models (\models (\models (\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta)))); \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};
\end{array}$$

$$\begin{array}{l}
(\Dashv (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma)) \\
\Leftarrow (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma); \\
(\Dashv (\Dashv (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma))) \\
\Leftarrow (\Dashv (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma)); \\
(\Dashv (\Dashv (\Dashv (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma)))) \\
\Leftarrow (\Dashv (\Dashv (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma))); \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
((\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \Dashv) \\
\Leftarrow (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma); \\
(((\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \Dashv) \Dashv) \\
\Leftarrow ((\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \Dashv); \\
((((\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \Dashv) \Dashv) \Dashv) \\
\Leftarrow (((\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \Dashv) \Dashv); \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}
\end{array}$$

Thus, by the property of transitivity, there is

[2.54]:

$$\begin{array}{l}
(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \Rightarrow \\
(\ldots (((\alpha, \beta, \ldots, \gamma \models \\
\qquad \xi, \eta, \ldots, \zeta) \models) \models) \ldots \models); \\
(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \Rightarrow \\
(\models \ldots (\models (\models (\alpha, \beta, \ldots, \gamma \models \\
\qquad \xi, \eta, \ldots, \zeta))) \ldots); \\
\text{"}\models\text{"} \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};
\end{array}$$

$$\begin{array}{l}
(\Dashv \ldots (\Dashv (\Dashv (\xi, \eta, \ldots, \zeta \Dashv \\
\qquad \alpha, \beta, \ldots, \gamma))) \ldots) \\
\Leftarrow (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma); \\
(\ldots (((\xi, \eta, \ldots, \zeta \Dashv \\
\qquad \alpha, \beta, \ldots, \gamma) \Dashv) \Dashv) \ldots \Dashv) \\
\Leftarrow (\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma); \\
\text{"}\Dashv\text{"} \in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\}
\end{array}$$

Besides of this explicitness of informational arising, there exists, by definition, also the operational implicitness (an implicit form of informational arising) of an informational process $\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta$ or $\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma$. This operational implicitness is coming to the surface when, for instance, an informational process marked by $\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta$ or $\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma$, is decomposed, and thus explicating its informational components (a composition of informational operators and operands). Again, the origin of this discussion can be the following:

[2.55]:

$$\begin{array}{l}
((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models) \Rightarrow \\
\mathfrak{I}_{\rightarrow}(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \text{ or simply} \\
((\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \models) \Rightarrow \\
\mathfrak{I}(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta); \\
(\models (\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta)) \Rightarrow \\
\mathfrak{I}_{\rightarrow}(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta) \text{ or simply} \\
(\models (\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta)) \Rightarrow \\
\mathfrak{I}(\alpha, \beta, \ldots, \gamma \models \xi, \eta, \ldots, \zeta);
\end{array}$$

$$\begin{array}{l}
\mathfrak{I}_{\leftarrow}(\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \\
\Leftarrow (\Dashv \pi\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma)) \text{ or simply} \\
\mathfrak{I}(\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma) \\
\Leftarrow (\Dashv \pi\xi, \eta, \ldots, \zeta \Dashv \alpha, \beta, \ldots, \gamma));
\end{array}$$

$$\begin{aligned}&\mathfrak{I}_{\leftarrow}(\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma)\\ \Leftarrow\ &((\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma) ⫤) \text{ or simply}\\ &\mathfrak{I}(\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma)\\ \Leftarrow\ &((\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma) ⫤);\end{aligned}$$

$$\text{"}⊨\text{"} \in \{⊨, ⊫, ⊢, ⊩, ⊭, ⊯, ⊬, ⊮\};$$
$$\text{"}⫤\text{"} \in \{⫤, ⫥, ⊣, ⫣, ⫤̸, ⫥̸, ⊣̸, ⫣̸\}$$

where $\mathfrak{I}$ (or $\mathfrak{I}_{\rightarrow}$ or $\mathfrak{I}_{\leftarrow}$) is the implicit operator of informing or non-informing (or informing or non-informing from the left to the right or from the right to left). $\mathfrak{I}(\alpha, \beta, \ldots , \gamma ⊨ \xi, \eta, \ldots , \zeta)$ or $\mathfrak{I}(\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma)$ is a functional expression which points out the operational component $\mathfrak{I}$ of the process $\alpha, \beta, \ldots , \gamma ⊨ \xi, \eta, \ldots , \zeta$ or $\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma$. Obviously, inductively, the last expressions can be expanded (decomposed), for instance, into

[2.56]:

$$\begin{aligned}&((\alpha, \beta, \ldots , \gamma ⊨ \xi, \eta, \ldots , \zeta) ⊨) \Rightarrow\\ &\mathfrak{I}(\alpha, \beta, \ldots , \gamma ⊨ \xi, \eta, \ldots , \zeta);\\ &(⊨ (\alpha, \beta, \ldots , \gamma ⊨ \xi, \eta, \ldots , \zeta)) \Rightarrow\\ &\mathfrak{I}(\alpha, \beta, \ldots , \gamma ⊨ \xi, \eta, \ldots , \zeta);\\ &\mathfrak{I}(\alpha, \beta, \ldots , \gamma ⊨ \xi, \eta, \ldots , \zeta) \Rightarrow\\ &\mathfrak{I}(\mathfrak{I} \ldots (\mathfrak{I}(\alpha, \beta, \ldots , \gamma ⊨\\ &\qquad\qquad \xi, \eta, \ldots , \zeta)) \ldots);\\ &\mathfrak{I} \in \{\mathfrak{I}_{⊨}, \mathfrak{I}_{⊫}, \mathfrak{I}_{⊢}, \mathfrak{I}_{⊩}, \mathfrak{I}_{⊭}, \mathfrak{I}_{⊯}, \mathfrak{I}_{⊬}, \mathfrak{I}_{⊮}\};\end{aligned}$$

$$\begin{aligned}&\mathfrak{I}(\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma)\\ \Leftarrow\ &(⫤ (\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma));\\ &\mathfrak{I}(\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma)\\ \Leftarrow\ &((\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma) ⫤);\\ &\mathfrak{I}(\mathfrak{I} \ldots (\mathfrak{I}(\xi, \eta, \ldots , \zeta ⫤\\ &\qquad\qquad \alpha, \beta, \ldots , \gamma)) \ldots)\\ \Leftarrow\ &\mathfrak{I}(\xi, \eta, \ldots , \zeta ⫤ \alpha, \beta, \ldots , \gamma);\\ &\mathfrak{I} \in \{\mathfrak{I}_{⫤}, \mathfrak{I}_{⫥}, \mathfrak{I}_{⊣}, \mathfrak{I}_{⫣}, \mathfrak{I}_{⫤̸}, \mathfrak{I}_{⫥̸}, \mathfrak{I}_{⊣̸}, \mathfrak{I}_{⫣̸}\}\end{aligned}$$

where $\mathfrak{I}_{⊨}$, $\mathfrak{I}_{⊫}$, $\mathfrak{I}_{⊢}$, $\mathfrak{I}_{⊩}$, $\mathfrak{I}_{⊭}$, $\mathfrak{I}_{⊯}$, $\mathfrak{I}_{⊬}$, $\mathfrak{I}_{⊮}$ and $\mathfrak{I}_{⫤}$, $\mathfrak{I}_{⫥}$, $\mathfrak{I}_{⊣}$, $\mathfrak{I}_{⫣}$, $\mathfrak{I}_{⫤̸}$, $\mathfrak{I}_{⫥̸}$, $\mathfrak{I}_{⊣̸}$, $\mathfrak{I}_{⫣̸}$ mark the so called general ($⊨$ and $⫤$), parallel ($⊫$ and $⫥$), cyclic ($⊢$ and $⊣$), and parallel-cyclic case ($⊩$ and $⫣$) of informing and general ($⊭$ and $⫤̸$), parallel ($⊯$ and $⫥̸$), cyclic ($⊬$ and $⊣̸$), and parallel-cyclic case ($⊮$ and $⫣̸$) of non-informing, respectively.

3. INFORMATIONAL OPERANDS

3.0. Introduction

Informational algebra organizes the transformation of informational formulas which are formal compositions of informational operands and operators. In formulas as in any formal system, operands are markers of distinct informational processes and as such can be decomposed into formulas in which to the original operand markers new informational compositions, i.e. formulas come into existence. The decomposition (in some cases called also the particularization) principle of operands enables the so-called informational analysis being the basic counter-informational property of the concept of information (in this case on the level of a formula). Thus, informational operands are markers (the simplest form of formulas or elementary formulas) which can be decomposed into informationally compound formulas. Thus, the notion of a concrete informational operand is always a relative one. By decomposition of an operand its operational components and, through these, its implicit operational nature are coming into explicitness (existence).

An informational operand - simple or composed - models an informational process. Formulas as operands are models of processes on the level of informational logic or informational algebra. Thus, logic or algebra becomes a tool for modeling of various informational processes or of processes which can be understood as informational or transformed, for instance, from physical, chemical, physiological, neuropsychological, cosmic processes, etc. into informational ones. On the level of anthropological awareness, informational processes concern the anthropological autopoiesis and, thus, experiencing and understanding of the world is nothing else than informational modeling within the possibilities of the autopoietic shell of a being. Then, informational or informationally modeling limits can concern only informational capabilities of the autopoietic shell. In this way, informing is always modeling and reality, and the awareness about reality of the world can be only informational, i.e., informationally modeled. An informational operand as informational formula is a formal mood of model which can be operated as any other informational entity.

3.1. Elementary informational operands

An elementary informational entity or operand α is elementary only to the extent that it informs and is informed. We shall see later how this property leads to the algebraic non-elementariness of an elementary operand. The consequence of this principle is that each elementary operand can always be algebraically transformed into a non-elementary expression, i.e., into composed (non-elementary) informational formula.

For, in general, informational operand informs and is informed, it is informationally open. The openness of an informational operand means that there may exist other informational operands which are informed by this operand and which can inform it. This fact can be expressed generally by unary, informational operators concerning the operand.

Further, an informational operand can be decomposed not only formally in the sense of unary operators, but also by informational analysis of its structure or organization. This process leads to the transformation of an elementary formula (a single operand) into informationally composed expression of operands and operators (formula).

3.2. Composed Informational Operands: the Real Informational Formulas

A single informational operand represents a trivial informational formula, for on the formalistic level it operates as a static or as a pure representational or marking entity. It merely designates an informational process which can possibly be decomposed into a more detailed operator and operand expression. In this way, in principle, the process of possible further decomposition never ends. However, by way of decomposition also processes of substitution, change, and vanishing of informational entities (operands and operators) can occur. We say that the decomposition process de-trivializes the marking nature of an operand.

To some extent, by rules of informational algebra, decomposition processes can be automatized, i.e., formal informational transformations of single operands and composite formulas can be performed by means of the algebraic apparatus. In this way also the reverse processes or rules to decomposition (we can call them universalization, recomposition, reduction, etc.) can be imagined, i.e. expressed in the informationally formal algebraic form. We have already learned some of these rules having universal and particular informational operators, for instance, the informational metaoperator $\models$ and its general, parallel, serial, parallel-serial, and to them alternative forms and their various particularizations (implication, equivalence, and other logical operators).

3.3. Ways of Decomposition and Recomposition (Marking, Symbolizing) of Operands

The rules of informational algebra concern decompositional as well as recompositional processes applied to various informational formulas. The question to state is what is the logic of informational decomposition and recomposition. It is evident that on the level of this decision various, linguistically or discursively structured logical approaches can be applied. The variety of extracting information (in the form of informational formulas) is, for instance, given by the so-called modi informationis [4]. Various operational concepts can be borrowed from the modal logic and from other theories of logic.

In informational algebra there do not exist rules prohibiting in advance the spontaneous processes of operand and operator particularization and universalization within decomposed and recomposed informational formulas.

By decomposition of a variable (informational operand), implicit variables are coming into existence. For example, if informational entity marked by α is decomposed, hidden variables arise or come into existence. By decomposition, the implicitness is made explicit, however, this is only another term for informational arising of the given informational entity α.

Instead of hidden variables it is possible to speak of variables, which do not inform and are non-informed yet. As we understand, the concept of decomposition puts an informational formula to the level of an arising informational entity. Thus, an informational formula itself performs as arising information.

3.4. Substitution within Decomposed Operands

As far as informational operands are merely marked and their decompositions do not exist, the initial process of their decomposing can begin. The process of decomposition is spontaneous to some extent, according to the semantic context of the formula in which operands appear.

Substitution of an operand by a formula is nothing else than a form of radical change or decomposition of the operand in question. It means that this operand transits from its marking-symbolic state into the formal composition which is structured on the level of operands connected via operators.

4. SOME FORMS OF INFORMATIONAL ALGEBRAS

> The principle of all principles comprises the thesis on the priority of the method. This principle decides about that which thing alone can satisfy the method.
>
> Martin Heidegger [11] 70

4.1. Introduction

In which way an informational algebra could be self-sufficient? How it could satisfy the metaprinciple which governs all the informational principles?

An informational algebra can concern several algebraic modes, from the most general to the most convenient, e.g., traditionally logical ones. The algebraic modes, we will deal with, are self-informational, general informational, informational implicative, informational equivalent, and informational modal. The last class of algebras will concern the most complex algebras imaginable today.

4.2. The Self-informational Algebra

The self-informational algebra is the basis of any informational algebra. Written algebraic expressions or formulas can be understood as being composed only of operands and operators. By another informational operation, called informational substitution and marked by the operator $\models_{\rightarrow}$, it is possible to construct recursively the entire system of algebraic rules of self-informational algebra. In the most general form this system becomes

$$\begin{aligned}[4.1]:\quad & \alpha \models_{\rightarrow} (\alpha;\ \models \alpha \models;\ \alpha \models;\ \models \alpha;\ \alpha \models \alpha);\\ & \text{"}\models\text{"} \in \{\models, ⊫, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash,\\ & \qquad ⫤, ⫥, \dashv, ⫣, \not⫤, \not⫥, \not\dashv, \not⫣\}\end{aligned}$$

etc. where occurring metaoperands α and occurring metaoperators ⊨ can be particularized from case to case, according .to the rєquirements (needs, circumstances). However, in this context one exception has to be mentioned: in ⊨ α ⊨, the case (α) is meant, where $\models_{(}$ α $\models_{)}$ is parenthetically particularized form of ⊨ α ⊨. Similarly, the semicolon in [4.1] can be understood as particularization of the form $\models_{;}$, etc. (For instance, also quotation-marks, ∈, {, }, and comma in [4.1] are nothing else than particular operators). A sequence of operands can be obtained by recursive use of α ⊨ α, where ⊨ can represent any separator, for instance, comma, semicolon, etc.

Formula [4.1] is read in the following way: an informational operand α can be substituted by (operator $\models_{\rightarrow}$) entities listed within the parentheses on the right side of $\models_{\rightarrow}$. Further, the metaoperator ⊨ can be particularized by any other general operator (on the right of ∈) and certainly (this is not marked explicitly by [4.1]) by any imaginable informational operator.

Let us look at a system of algebraic rules of self-informational algebra proceeding from the formula [4.1]. Let be, for instance:

[4.2]:

[1] "⊨" ∈ {⊨, ⊫, ⊢, ⊩, ⊭, ⊯, ⊬, ⊮};

[2] "⫤" ∈ {⫤, ⫥, ⊣, ⫣, ⫤̸, ⫥̸, ⊣̸, ⫣̸};

[3] α; (α); [single operand expressions];

[4] α ⊨ α; α ⫤ α;

[5] α ⊨ (α); (α) ⊨ α; (α) ⊨ (α);
(α) ⫤ α; α ⫤ (α); (α) ⫤ (α);

[6] α ⊨; ⫤ α; (α) ⊨; ⫤ (α);

[7] ⊨ α; α ⫤; ⊨ (α); (α) ⫤;

[8] α ⊨ (α ⊨); (⫤ α) ⫤ α;

[9] α ⊨ (⊨ α); (α ⫤) ⫤ α;

[10] α ⫤ (α ⊨); (α ⊨) ⫤ α;

[11] α ⫤ (⊨ α); (⊨ α) ⫤ α;

[12] (α ⊨) ⊨ α; α ⫤ (⫤ α);

[13] (⊨ α) ⊨ α; α ⫤ (α ⫤);

[14] (⫤ α) ⊨ α; α ⫤ (α ⊨);

[15] (α ⫤) ⊨ α; α ⫤ (⊨ α);

[16] α ⊨ (α ⊨ α); (α ⫤ α) ⫤ α;

[17] α ⫤ (α ⊨ α); (α ⊨ α) ⫤ α;

[18] (α ⊨ α) ⊨ α; α ⫤ (α ⫤ α);

[19] (α ⫤ α) ⊨ α; α ⫤ (α ⊨ α);

[20] α ⊨ (α ⊨ (α ⊨)); ((⫤ α) ⫤ α) ⫤ α;

[21] α ⊨ (α ⊨ (⊨ α)); ((α ⫤) ⫤ α) ⫤ α;

[22] α ⊨ ((α ⊨) ⊨ α); (α ⫤ (⫤ α)) ⫤ α;

[23] α ⊨ ((⊨ α) ⊨ α); (α ⫤ (α ⫤)) ⫤ α;

[24] ((α ⊨) ⊨ α) ⊨ α; α ⫤ (α ⫤ (⫤ α));

[25] ((⊨ α) ⊨ α) ⊨ α; α ⫤ (α ⫤ (α ⫤));

[26] α ⫤ (α ⊨ (α ⊨)); ((α ⊨) ⫤ α) ⊨ α;

[27] α ⫤ (α ⊨ (⊨ α)); ((⊨ α) ⫤ α) ⊨ α;

[28] α ⫤ ((α ⊨) ⊨ α); (α ⫤ (⊨ α)) ⊨ α;

[29] α ⊨ (α ⫤ (α ⊨)); ((⫤ α) ⊨ α) ⫤ α;

[30] α ⊨ (α ⊨ (⫤ α)); ((⫤ α) ⫤ α) ⊨ α;

. . . .

etc., ad infinitum. It is evident that arbitrarily complex formula consisting of operands α, operators ⊨ and ⫤, and parentheses '(' and ')' can be generated (automatically).

System [4.1] suggests also generation of systems of informational formulas. Such systems, if marked (similarly as the system [4.1]) can be parenthesized. From [4.1], for instance, system formulas

[4.3]:

[1] (α);

[2] (α; α; ... ; α);

[3] (α; (α)); ((α); α);

[4] (α; (α ⊨)); ((⫤ α); α);

[5] (α; (⊨ α)); ((α ⫤); α);

[6] (α; ((α ⊨); (⊨ α)); ((α ⫤); (⫤ α); α);

[7] (α; (α ⊨ α)); ((α ⫤ α); α);

[8] ((α ⊨); (α ⊨ α)); ((α ⫤ α); (⫤ α));

[9] ((⊨ α); (α ⊨ α)); ((α ⫤ α); (α ⫤));

. . . .

"⊨" ∈ {⊨, ⊫, ⊢, ⊩, ⊭, ⊯, ⊬, ⊮};

"⫤" ∈ {⫤, ⫥, ⊣, ⫣, ⫤̸, ⫥̸, ⊣̸, ⫣̸}

can be generated ad infinitum.

Further, it is worth to stress how the unary metaoperators ⊨ and ⫤ appear dually on the formal level in regard to each other. While, for instance, operator ⊨ is in the function 'to inform', operator ⫤ is in the function 'to be informed'. This duality extends also vice versa: while operator ⫤ is in the function 'to inform', operator ⊨ is in the function 'to be informed'. This is true in the case α ⊨ and α ⫤ and in the case ⊨ α and ⫤ α, where always the active and passive function of informing (to inform and to be informed, respectively) stay dually against each other. This fact might entitle the initial introduction of the dual operator ⫤ to the operator ⊨, pointing to deeper consequences which might follow from such initial (intuitive) choice.

4.3. The General Informational Algebra

The step from the self-informational algebra to the general informational algebra roots primarily in the substitution of informational operands α by differently marked informational operands, say α, β, γ, ... in algebraic systems [4.2] and [4.3]. Under these circumstances, markers of informational operands ξ belong to all possible operand markers, for instance to α, β, γ, ... In this case, system [4.2] simply passes over to the system

[4.4]:

[0] "ξ" ∈ {α, β, γ, ... , ξ, ...};

[1] "⊨" ∈ {⊨, ⊫, ⊢, ⊩, ⊭, ⊯, ⊬, ⊮};

[2] "⫤" ∈ {⫤, ⫥, ⊣, ⫣, ⫤̸, ⫥̸, ⊣̸, ⫣̸};

[3] ξ; (ξ); [single operand expressions];

[4] ξ ⊨ ξ; ξ ⫤ ξ;

[5] ξ ⊨ (ξ); (ξ) ⊨ ξ; (ξ) ⊨ (ξ);
(ξ) ⫤ ξ; ξ ⫤ (ξ); (ξ) ⫤ (ξ);

[6] ξ ⊨; ⫤ ξ; (ξ) ⊨; ⫤ (ξ);

[7] ⊨ ξ; ξ ⫤; ⊨ (ξ); (ξ) ⫤;

[8] ξ ⊨ (ξ ⊨); (⫤ ξ) ⫤ ξ;

[9] ξ ⊨ (⊨ ξ); (ξ ⫤) ⫤ ξ;

[10] ξ ⫤ (ξ ⊨); (ξ ⊨) ⫤ ξ;

[11] ξ ⫤ (⊨ ξ); (⊨ ξ) ⫤ ξ;
[12] (ξ ⊨) ⊨ ξ; ξ ⫤ (⫤ ξ);
[13] (⊨ ξ) ⊨ ξ; ξ ⫤ (ξ ⫤);
[14] (⫤ ξ) ⊨ ξ; ξ ⫤ (ξ ⊨);
[15] (ξ ⫤) ⊨ ξ; ξ ⫤ (⊨ ξ);
[16] ξ ⊨ (ξ ⊨ ξ); (ξ ⫤ ξ) ⫤ ξ;
[17] ξ ⫤ (ξ ⊨ ξ); (ξ ⊨ ξ) ⫤ ξ;
[18] (ξ ⊨ ξ) ⊨ ξ; ξ ⫤ (ξ ⫤ ξ);
[19] (ξ ⫤ ξ) ⊨ ξ; ξ ⫤ (ξ ⊨ ξ);
[20] ξ ⊨ (ξ ⊨ (ξ ⊨)); ((⫤ ξ) ⫤ ξ) ⫤ ξ;
[21] ξ ⊨ (ξ ⊨ (⊨ ξ)); ((ξ ⫤) ⫤ ξ) ⫤ ξ;
[22] ξ ⊨ ((ξ ⊨) ⊨ ξ); (ξ ⫤ (⫤ ξ)) ⫤ ξ;
[23] ξ ⊨ ((⊨ ξ) ⊨ ξ); (ξ ⫤ (ξ ⫤)) ⫤ ξ;
[24] ((ξ ⊨) ⊨ ξ) ⊨ ξ; ξ ⫤ (ξ ⫤ (⫤ ξ));
[25] ((⊨ ξ) ⊨ ξ) ⊨ ξ; ξ ⫤ (ξ ⫤ (ξ ⫤));
[26] ξ ⫤ (ξ ⊨ (ξ ⊨)); ((ξ ⊨) ⫤ ξ) ⊨ ξ;
[27] ξ ⫤ (ξ ⊨ (⊨ ξ)); ((⊨ ξ) ⫤ ξ) ⊨ ξ;
[28] ξ ⫤ ((ξ ⊨) ⊨ ξ); (ξ ⫤ (⊨ ξ)) ⊨ ξ;
[29] ξ ⊨ (ξ ⫤ (ξ ⊨)); ((⫤ ξ) ⊨ ξ) ⫤ ξ;
[30] ξ ⊨ (ξ ⊨ (⫤ ξ)); ((⫤ ξ) ⫤ ξ) ⊨ ξ;
. . . .

etc., ad infinitum. It is evident that arbitrarily complex formula consisting of operands α, β, γ, ... , ξ, ..., operators ⊨ and ⫤, and parentheses '(' and ')' can be generated (automatically). For instance, in case [30], there is ω ⊨ (τ ⊨ (⫤ ψ)); ((⫤ φ) ⫤ σ) ⊨ ρ.

Again, system [4.1] suggests also the generation of systems of informational formulas with differently marked operands. Such systems, if marked (similarly as the system [4.1]) can be parenthesized. From [4.3], for instance, system formulas

[4.5]:
[1] (ξ);
[2] (ξ; ξ; ... ; ξ);
[3] (ξ; (ξ)); ((ξ); ξ);
[4] (ξ⊥ (ξ ⊨)); ((⫤ ξ); ξ);
[5] (ξ; (⊨ ξ)); ((ξ ⫤); ξ);
[6] (ξ; ((ξ ⊨); (⊨ ξ)); ((ξ ⫤); (⫤ ξ); ξ);
[7] (ξ; (ξ ⊨ ξ)); ((ξ ⫤ ξ); ξ);
[8] ((ξ ⊨); (ξ ⊨ ξ)); ((ξ ⫤ ξ); (⫤ ξ));
[9] ((⊨ ξ); (ξ ⊨ ξ)); ((ξ ⫤ ξ); (ξ ⫤));
. . . .
"ξ" ∈ {α, β, γ, ... , ξ, ...};
"⊨" ∈ {⊨, ⊫, ⊢, ⊩, ⊭, ⊯, ⊬, ⊮};
"⫤" ∈ {⫤, ⫥, ⊣, ⫣, ⫤̸, ⫥̸, ⊣̸, ⫣̸}

can be generated ad infinitum. Thus, for instance, in case [9], there is ((⊨ α); (β ⊨ γ)); ((δ ⫤ ε); (ζ ⫤)).

4.4. The Implicative Informational Algebra

The implicative informational algebra concentrates on some "logical facts" which may be useful in the context of informational conceptualism, by which informational entities are (partly, not necessarily equivalently) informationally determined. It is evident that informational implications can proceed from the self-informational and general informational algebra, replacing informational metaoperators by implicative ones. However, these replacements cannot be arbitrary to keep the logical conceptualism valid. Partly, by introduction of logical (informational) implication into informational formulas, these formulas enter the realm of convenient logical truth and falsity.

For an informational entity α (an informational operand or a marked informational formula) it was said that it informs and is informed. This determination has also its implicative consequences, namely, α ⇒ (α ⊨) and α ⇒ (⊨ α), etc. Let us gather the most important implications concerning informational principles, although these implications cannot represent the entire possible realm of further, useful implications.

By implicative informational algebra we will enter into the domain of basic understanding of informational phenomenology. This understanding is natural since it proceeds from traditionally logical consequences resting in expressions of natural languages. In this way, the implicative algebra will become a careful searching of the roots belonging to the term 'information'. Thus, it is possible to propose the following initial set of implicative algebraic rules:

[4.6]:
[1] "ξ" ∈ {α, β, γ, ... , ξ, ...};
[2] "⊨" ∈ {⊨, ⊫, ⊢, ⊩, ⊭, ⊯, ⊬, ⊮};
[3] "⫤" ∈ {⫤, ⫥, ⊣, ⫣, ⫤̸, ⫥̸, ⊣̸, ⫣̸}
[4] ξ ⇒ (ξ ⊨); (⫤ ξ) ⇐ ξ;
ξ ⇒ (⊨ ξ); (ξ ⫤) ⇐ ξ;
ξ ⇒ (⫤ ξ); (ξ ⊨) ⇐ ξ;
ξ ⇒ (ξ ⫤); (⊨ ξ) ⇐ ξ;
[5] ξ ⇒ ((ξ ⊨); (⊨ ξ)); ((ξ ⫤); (⫤ ξ)) ⇐ ξ;
ξ ⇒ ((ξ ⊨); (ξ ⫤)); ((⊨ ξ); (⫤ ξ)) ⇐ ξ;
ξ ⇒ ((⫤ ξ); (⊨ ξ)); ((ξ ⫤); (ξ ⊨)) ⇐ ξ;
ξ ⇒ ((ξ ⫤); (⫤ ξ)); ((ξ ⊨); (⊨ ξ)) ⇐ ξ;
[6] ξ ⇒ ((ξ ⊨) ∨ (⊨ ξ)); ((ξ ⫤) ∨ (⫤ ξ)) ⇐ ξ;
ξ ⇒ ((ξ ⊨) ∨ (ξ ⫤)); ((⊨ ξ) ∨ (⫤ ξ)) ⇐ ξ;
ξ ⇒ ((⫤ ξ) ∨ (⊨ ξ)); ((ξ ⫤) ∨ (ξ ⊨)) ⇐ ξ;
ξ ⇒ ((ξ ⫤) ∨ (⫤ ξ)); ((ξ ⊨) ∨ (⊨ ξ)) ⇐ ξ;
[7] ξ ⇒ ((ξ ⊨); (⊨ ξ); (⫤ ξ));
((ξ ⫤); (⫤ ξ); (ξ ⊨)) ⇐ ξ;
ξ ⇒ ((ξ ⊨); (ξ ⫤); (⫤ ξ));
((⊨ ξ); (⫤ ξ); (ξ ⊨)) ⇐ ξ;
. . . .
ξ ⇒ ((ξ ⫤); (⫤ ξ); (⊨ ξ));
((ξ ⊨); (⊨ ξ); (ξ ⫤)) ⇐ ξ;
[8] ξ ⇒ ((ξ ⊨) ∨ (⊨ ξ) ∨ (⫤ ξ));
((ξ ⫤) ∨ (⫤ ξ) ∨ (ξ ⊨)) ⇐ ξ;
. . . .
[9] ξ ⇒ ((ξ ⊨); (⊨ ξ); (⫤ ξ); (ξ ⫤));
((ξ ⫤); (⫤ ξ); (ξ ⊨); (⊨ ξ)) ⇐ ξ;
[10] ξ ⇒ ((ξ ⊨) ∨ (⊨ ξ) ∨ (⫤ ξ) ∨ (ξ ⫤));
((ξ ⫤) ∨ (⫤ ξ) ∨ (ξ ⊨) ∨ (⊨ ξ)) ⇐ ξ;

[11] (ξ ⊨) ⇒ ((ξ ⊨) ⊨); ((⫤ (⫤ ξ) ⇐ (⫤ ξ);
(ξ ⊨) ⇒ (⫤ (ξ ⊨)); ((⫤ ξ) ⊨) ⇐ (⫤ ξ);
. . . .
(⊨ ξ) ⇒ ((⊨ ξ) ⊨); (⫤ (ξ ⫤)) ⇐ (ξ ⫤);
(⊨ ξ) ⇒ (⊨ (⊨ ξ)); ((ξ ⫤) ⫤) ⇐ (ξ ⫤);
[12] (ξ ⇒ (ξ ⊨)) ⇒ ((ξ ⊨) ⊨);
(⫤ (⫤ ξ)) ⇐ ((⫤ ξ) ⇐ ξ);
(ξ ⇒ (⊨ ξ)) ⇒ (⊨ (⊨ ξ));

$((\xi \Dashv) \Dashv) \Leftarrow ((\xi \Dashv) \Leftarrow \xi);$

. . . .

$((\models \xi) \Dashv) \Leftarrow ((\models \xi) \Leftarrow \xi);$

[13] $((\xi \models) \models) \Rightarrow (((\xi \models) \models) \models);$

$(\Dashv (\Dashv (\Dashv \xi))) \Leftarrow (\Dashv (\Dashv \xi));$

. . . .

[14] $\xi \Rightarrow (\xi \models \xi);\ (\xi \Dashv \xi) \Leftarrow \xi;$

$(\xi \models \xi) \Rightarrow \xi;\ \xi \Leftarrow (\xi \Dashv \xi);$

$\xi \Rightarrow (\xi \Dashv \xi);\ (\xi \models \xi) \Leftarrow \xi;$

$(\xi \Dashv \xi) \Rightarrow \xi;\ \xi \Leftarrow (\xi \models \xi);$

[15] $(\xi \models \xi) \Rightarrow (\models \xi);\ (\Dashv \xi) \Leftarrow (\xi \Dashv \xi);$

$(\xi \models \xi) \Rightarrow (\models \xi);\ (\xi \Dashv) \Leftarrow (\xi \Dashv \xi);$

$(\xi \models \xi) \Rightarrow (\xi \Dashv);\ (\xi \models) \Leftarrow (\xi \Dashv \xi);$

$(\xi \models \xi) \Rightarrow (\xi \Dashv);\ (\models \xi) \Leftarrow (\xi \Dashv \xi);$

[16] $\xi \Rightarrow ((\models \xi) \models);\ (\Dashv (\xi \Dashv)) \Leftarrow \xi;$

$\xi \Rightarrow ((\xi \Dashv) \models);\ (\Dashv (\models \xi)) \Leftarrow \xi;$

$\xi \Rightarrow (\Dashv (\models \xi));\ ((\xi \Dashv) \models) \Leftarrow \xi;$

$\xi \Rightarrow (\Dashv (\xi \Dashv));\ ((\models \xi) \models) \Leftarrow \xi;$

[17] $\xi \Rightarrow (\models (\xi \models));\ ((\Dashv \xi) \Dashv) \Leftarrow \xi;$

$\xi \Rightarrow (\models (\Dashv \xi));\ ((\models \xi) \Dashv) \Leftarrow \xi;$

$\xi \Rightarrow ((\xi \models) \Dashv);\ (\models (\Dashv \xi)) \Leftarrow \xi;$

$\xi \Rightarrow ((\Dashv \xi) \Dashv);\ (\models (\xi \models)) \Leftarrow \xi;$

. . . .

It can be imagined how further informational implications can be generated ad infinitum. In these implications, general informational operators can be particularized according to particular needs and requirements. In this sense, implicative informational algebra becomes applicatively flexible and not bounded as a traditional logic could be.

4.5. The Equivalence Informational Algebra

Similarly as the implicative informational algebra, the so-called equivalence informational algebra deals with particular "logical facts" which may be useful in the context of informational determinism, by which informational entities are (equivalently) informationally determined. It is evident that informational equivalences can proceed from the self-informational,general informational, and implicative informational algebra, replacing informational metaoperators and implicative operators by equivalence ones. However, these replacements cannot be arbitrary to keep the logical determinism valid. Partly, by introduction of logical (informational) equivalence into informational formulas, these formulas (partly) enter the realm of convenient logical truth and falsity.

For an arbitrary informational entity α (an informational operand or a marked informational formula) it was said that it informs and is informed. This determination has also its equivalence consequence, namely, $\alpha \Leftrightarrow ((\alpha \models);$ $(\models \alpha))$ or $\alpha \Leftrightarrow ((\alpha \models) \vee (\models \alpha))$ etc. We choose the symbol $\Leftrightarrow$ or $\Rightarrow\!\!\Leftarrow$ for the equivalence instead of symbol $\equiv$ to stress the deductive origin which concerns the implicative algebra. So, let us gather the most important equivalences concerning informational principles, although these equivalences cannot represent the entire possible realm of further, useful equivalences.

By equivalence informational algebra we will enter into the domain of basic understanding of informational notions. This understanding is natural since it proceeds from intuitive logical consequences resting in speech and writing of natural languages. In this way, the equivalence algebra will become a notional searching of the roots belonging to the determination of information. In this way it is possible to propose the following initial set of equivalence algebraic rules:

[4.6]:

[1] $"\xi" \in \{\alpha, \beta, \gamma, \ldots , \xi, \ldots\};$

[2] $"\models" \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$

[3] $"\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV, \nDashv, \nDashV, \ndashv, \ndashV\};$

[4] $\xi \Leftrightarrow ((\xi \models);\ (\models \xi));$

$((\xi \Dashv);\ (\Dashv \xi)) \Rightarrow\!\!\Leftarrow \xi;$

[5] $\xi \Leftrightarrow ((\xi \models) \vee (\models \xi));$

$((\xi \Dashv) \vee (\Dashv \xi)) \Rightarrow\!\!\Leftarrow \xi;$

[6] $\xi \Leftrightarrow ((\xi \models);\ (\models \xi);\ (\xi \Dashv);\ (\Dashv \xi));$

$((\xi \models);\ (\models \xi);\ (\xi \Dashv);\ (\Dashv \xi)) \Rightarrow\!\!\Leftarrow \xi;$

[7] $\xi \Leftrightarrow ((\xi \models) \vee (\models \xi) \vee (\xi \Dashv) \vee (\Dashv \xi));$

$((\xi \models) \vee (\models \xi) \vee (\xi \Dashv) \vee (\Dashv \xi)) \Rightarrow\!\!\Leftarrow \xi;$

[8] $\xi \Leftrightarrow ((\xi \models \xi);\ (\xi \Dashv \xi));$

$((\xi \models \xi);\ (\xi \Dashv \xi)) \Rightarrow\!\!\Leftarrow \xi;$

[9] $\xi \Leftrightarrow ((\xi \models \xi) \vee (\xi \Dashv \xi));$

$((\xi \models \xi) \vee (\xi \Dashv \xi)) \Rightarrow\!\!\Leftarrow \xi;$

[10] 'information_as_informational_system' $\Leftrightarrow$

$((\exists\ \xi).(\xi \in \{\alpha, \beta, \gamma, \ldots , \xi, \ldots\}) \wedge$

$(\exists \models).(\models \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\}) \wedge$

$(\exists \Dashv).(\Dashv \in \{\Dashv, \DashV, \dashv, \dashV, \nDashv, \nDashV, \ndashv, \ndashV\})).$

$((\xi \models);\ (\models \xi);\ (\xi \Dashv);\ (\Dashv \xi);\ (\xi \models \xi);$

$(\xi \Dashv \xi));$

'information as informational system' $\Leftrightarrow$

$((\exists\ \xi).(\xi \in \{\alpha, \beta, \gamma, \ldots , \xi, \ldots\}) \wedge$

$(\exists \models).(\models \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\}) \wedge$

$(\exists \Dashv).(\Dashv \in \{\Dashv, \DashV, \dashv, \dashV, \nDashv, \nDashV, \ndashv, \ndashV\})).$

$((\xi \models) \vee (\models \xi) \vee (\xi \Dashv) \vee (\Dashv \xi) \vee$

$(\xi \models \xi) \vee (\xi \Dashv \xi))$

etc. Formula [10] constitutes information as the most complex informational system in which informational entities (processes) are mutually perplexed in any imaginable form.

4.6. Some Transformation Rules

The class of equivalence rules can be supplemented by some formula transformation rules, which seems to be the subject of their definition. In these cases the equivalence signs $\equiv$ or $\not\equiv$ will be used. Such rules can be as follows:

[4.7]:

[1] $"\models" \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$

[2] $"\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV, \nDashv, \nDashV, \ndashv, \ndashV\};$

[3] $\alpha \equiv (\alpha \models \alpha);\ (\alpha \Dashv \alpha) \equiv \alpha;$

[4] $(\alpha \models \beta) \equiv (\alpha \models \beta;\ \alpha \models \beta;\ \beta \models \beta);$

[5] $(\beta \Dashv \beta;\ \beta \Dashv \alpha;\ \alpha \Dashv \alpha) \equiv (\beta \Dashv \alpha);$

[6] $(\alpha, \beta \models \gamma, \delta) \equiv$

$(\alpha \models \gamma;\ \alpha \models \delta;\ \beta \models \gamma;\ \beta \models \delta);$

[7] $(\delta \Dashv \beta;\ \gamma \Dashv \beta;\ \delta \Dashv \alpha;\ \gamma \Dashv \alpha) \equiv$
$(\delta,\ \gamma \Dashv \beta,\ \alpha);$

. . . .

etc. However, some implicative consequences cannot necessarily lead to an equivalence. Thus, for instance,

[4.8]: $((\alpha \Rightarrow \beta) \wedge (\alpha \Leftarrow \beta)) \Rightarrow (\alpha \not\equiv_{\nu} \beta)$

In this formula, $\not\equiv_{\nu}$ marks the informational operator possessing the meaning "is not necessarily equivalent". The particular symbol ν is used for marking the necessity. Instead of symbol $\not\equiv_{\nu}$, symbol $\equiv_{\pi}$ could be used, where π stands for possibility.

4.7. Possibilities of Modal Informational Algebras

The modal informational algebras will concern the realm of the so-called modus informationis [4]. Traditional logical concepts can be seen as particular (modal-trivial) cases of modal logic.

The origin of modal informational logic roots in the Latin "modus" as information, i.e. in measure, extent, rhythm, way, manner, method of informing, etc. Also, modus informationis is meant to be a way, manner, method, etc. of acquiring, gaining, extracting, or, in the most general way, of arising or coming of information into existence within and through information in question. In this respect, modal informational logic differs from the convenient modal logic, which is limited to the logic of necessity and possibility, of 'must be' and 'may be'.

In [4], several types of modus informationis have been discussed. What can be the algebraic consequence of these modi? It is evident that the way of informational acquiring concerning particular modi cannot be determined in general or in advance, for this acquiring depends, for instance, on the semantic analysis and intuition (meaning) of a particular informational case. Although in [4] we have discussed the most obvious, i.e. historically, culturally, logically accepted informational forms of reasoning, common sense, inference as informational modi (modus ponens, tollens, rectus, obliquus, procedendi, operandi, vivendi, possibilitatis, necessitatis), only the most convenient manners of informational reasoning have been touched. It is to understand that modal informational algebra has in general to do with discovering of certain information within information in question.

Let us list in short only the most important algebraic rules concerning modi informationis.

4.7.1. Informational Modus Ponens

Informational modus ponens originates in some general informational schemes, for instance,

[4.9]:

[1] "ξ", "η" $\in \{\alpha, \beta, \gamma, \ldots, \xi, \eta, \ldots\};$

[2] "$\models$" $\in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$

[3] "$\Dashv$" $\in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\};$

[4] $(\xi \models (\xi \models \eta)) \models \eta;$

[5] $\eta \Dashv ((\eta \Dashv \xi) \Dashv \xi);$

[6] $(\xi \models (\eta \Dashv \xi)) \models \eta;$

[7] $\eta \Dashv ((\xi \models \eta) \Dashv \xi);$

[8] $((\xi \models (\xi \models \eta));\ (\xi \models (\eta \Dashv \xi))) \models \eta;$

[9] $\eta \Dashv (((\xi \models \eta) \Dashv \xi);\ ((\eta \Dashv \xi) \Dashv \xi));$

[10] $((\xi \models (\xi \models \eta)) \models (\xi \models (\eta \Dashv \xi))) \models \eta;$

[11] $\eta \Dashv (((\xi \models \eta) \Dashv \xi) \Dashv ((\eta \Dashv \xi) \Dashv \xi));$

etc. Informational modus ponens can use several conjunctively and disjunctively implicative algebraic rules, for instance,

[4.10]:

[1] "ξ", "η" $\in \{\alpha, \beta, \gamma, \ldots, \xi, \eta, \ldots\};$

[2] "$\models$" $\in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$

[3] "$\Dashv$" $\in \{\Dashv, \DashV, \dashv, \dashV, \not\Dashv, \not\DashV, \not\dashv, \not\dashV\};$

[4] $(\xi \wedge (\xi \models \eta)) \Rightarrow \eta;$

[5] $\eta \Leftarrow (\xi \wedge (\eta \Dashv \xi));$

[6] $(\xi \wedge (\eta \Dashv \xi)) \Rightarrow \eta;$

[7] $\eta \Leftarrow (\xi \wedge (\xi \models \eta));$

[8] $(\xi \wedge ((\xi \models \eta);\ (\eta \Dashv \xi)) \Rightarrow \eta;$

[9] $\eta \Leftarrow (\xi \wedge ((\eta \Dashv \xi);\ (\xi \models \eta));$

[10] $(\xi \wedge ((\xi \models \eta) \vee (\eta \Dashv \xi)) \Rightarrow \eta;$

[11] $\eta \Leftarrow (\xi \wedge ((\eta \Dashv \xi) \vee (\xi \models \eta))$

etc. Let us read some of the rules belonging to the system [4.7]. Rule [4] $(\xi \wedge (\xi \models \eta)) \Rightarrow \eta$ is read in the following way: "if ξ is information and if ξ informs η in one way, then η is information in one way". Rule [5] $\eta \Leftarrow (\xi \wedge (\eta \Dashv \xi))$ can be read in several ways: "if ξ is information and if ξ informs η in another way, then η is information in another way"; or "η is information in another way, if ξ is information and if η is informed by ξ in another way".

4.7.2. Informational Modus Tollens

Similarly as in the case of modus ponens, it is possible to list some general informational schemes for informational modus tollens, for instance,

[4.11]:

[1] "ξ", "η" $\in \{\alpha, \beta, \gamma, \ldots, \xi, \eta, \ldots\};$

[2] "$\models$" $\in \{\models, \VDash, \vdash, \Vdash\};$ "$\nvDash$" $\in \{\nvDash, \nVDash, \nvdash, \nVdash\};$

[3] "$\Dashv$" $\in \{\Dashv, \DashV, \dashv, \dashV\};$ "$\not\Dashv$" $\in \{\not\Dashv, \not\DashV, \not\dashv, \not\dashV\};$

[4] $((\xi \models \eta) \models (\nvDash \eta)) \models (\xi \nvDash);$

[5] $(\not\Dashv \xi) \Dashv ((\eta \not\Dashv) \Dashv (\eta \Dashv \xi));$

[6] $((\eta \Dashv \xi) \models (\nvDash \eta)) \models (\xi \nvDash);$

[7] $(\not\Dashv \xi) \Dashv ((\eta \not\Dashv) \Dashv (\xi \models \eta));$

[8] $((\xi \models \eta) \models (\eta \not\Dashv)) \models (\xi \nvDash);$

[9] $(\not\Dashv \xi) \Dashv ((\nvDash \eta) \Dashv (\eta \Dashv \xi));$

[10] $((\xi \models \eta) \models (\nvDash \eta)) \models (\not\Dashv \xi);$

[11] $(\xi \nvDash) \Dashv ((\eta \not\Dashv) \Dashv (\eta \Dashv \xi));$

[12] $((\nvDash \eta) \Dashv (\xi \models \eta)) \models (\xi \nvDash);$

[13] $(\not\Dashv \xi) \Dashv ((\eta \Dashv \xi) \models (\eta \not\Dashv));$

[14] $(((\xi \models \eta) \models (\xi \nvDash));$
$((\eta \Dashv \xi) \models (\xi \nvDash))) \models \eta;$

[15] $\eta \Dashv (((\not\Dashv \xi) \Dashv (\xi \models \eta));$
$((\not\Dashv \xi) \Dashv (\eta \Dashv \xi)));$

[16] $(((\xi \vDash \eta) \vDash (\xi \nvDash)) \vDash$
$((\eta \Dashv \xi) \vDash (\xi \nvDash))) \vDash \eta;$
[17] $\eta \Dashv (((\not\Dashv \xi) \Dashv (\xi \vDash \eta)) \Dashv$
$((\not\Dashv \xi) \Dashv (\eta \Dashv \xi)));$

etc. Informational modus tollens can use several conjunctively and disjunctively implicative algebraic rules, for instance,

[4.12]:
[1] $"\xi", "\eta" \in \{\alpha, \beta, \gamma, \ldots , \xi, \eta, \ldots\};$
[2] $"\vDash" \in \{\vDash, \VDash, \vdash, \Vdash\}; "\nvDash" \in \{\nvDash, \nVDash, \nvdash, \nVdash\};$
[3] $"\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV\}; "\not\Dashv" \in \{\not\Dashv, \not\DashV, \not\dashv, \not\dashV\};$
[4] $((\xi \vDash \eta) \wedge (\nvDash \eta)) \Rightarrow (\xi \nvDash);$
[5] $(\not\Dashv \xi) \Leftarrow ((\eta \not\Dashv) \wedge (\eta \Dashv \xi));$
[6] $((\eta \Dashv \xi) \wedge (\nvDash \eta)) \Rightarrow (\xi \nvDash);$
[7] $(\not\Dashv \xi) \Leftarrow ((\eta \not\Dashv) \wedge (\xi \vDash \eta));$
[8] $((\xi \vDash \eta) \wedge (\eta \not\Dashv)) \Rightarrow (\xi \nvDash);$
[9] $(\not\Dashv \xi) \Leftarrow ((\nvDash \eta) \wedge (\eta \Dashv \xi));$
[10] $((\xi \vDash \eta) \wedge (\nvDash \eta)) \Rightarrow (\not\Dashv \xi);$
[11] $(\xi \nvDash) \Leftarrow ((\eta \not\Dashv) \wedge (\eta \Dashv \xi));$
[12] $((\nvDash \eta) \wedge (\xi \vDash \eta)) \Rightarrow (\xi \nvDash);$
[13] $(\not\Dashv \xi) \Leftarrow ((\eta \Dashv \xi) \wedge (\eta \not\Dashv));$
[14] $(((\xi \vDash \eta) \wedge (\xi \nvDash));$
$((\eta \Dashv \xi) \wedge (\xi \nvDash))) \Rightarrow \eta;$
[15] $\eta \Leftarrow (((\not\Dashv \xi) \wedge (\xi \vDash \eta));$
$((\not\Dashv \xi) \wedge (\eta \Dashv \xi)));$
[16] $(((\xi \vDash \eta) \wedge (\xi \nvDash)) \vee$
$((\eta \Dashv \xi) \wedge (\xi \nvDash))) \Rightarrow \eta;$
[17] $\eta \Leftarrow (((\not\Dashv \xi) \wedge (\xi \vDash \eta)) \vee$
$((\not\Dashv \xi) \wedge (\eta \Dashv \xi)));$

Let us read some of the rules belonging to the system [4.10]. Rule, marked by [4], $((\xi \vDash \eta) \wedge (\nvDash \eta)) \Rightarrow (\xi \nvDash)$, is read in the following way: "if ξ informs η and if η is not informed in one way, then ξ does not inform in one way". Rule [5], $(\not\Dashv \xi) \Leftarrow ((\eta \not\Dashv) \wedge (\eta \Dashv \xi))$, can be read in several ways: "if ξ informs η in another way and if η is not informed in another way, then ξ does not inform in another way"; or "ξ does not inform in another way, if η is informed by ξ in another way and if η is not informed". Etc.

4.7.3. Informational Modus Rectus

One of the aims of modus rectus is to detach or to extract intentional information, which informs intentionally within (as a part of) information in question. Thus, the intention of information images in informing of information, in that how information in question informs itself and other information and how it is informed by itself and by other information. If $\mathfrak{I}$ or $\mathfrak{I}(\beta)$ marks the implicit intentional informing of information β, then the following implicative informational rules, determining the intentional information α can be senseful:

[4.13]:
[1] $"\vDash_{\mathfrak{I}}" \in \{\vDash_{\mathfrak{I}}, \VDash_{\mathfrak{I}}, \vdash_{\mathfrak{I}}, \Vdash_{\mathfrak{I}}, \nvDash_{\mathfrak{I}}, \nVDash_{\mathfrak{I}}, \nvdash_{\mathfrak{I}}, \nVdash_{\mathfrak{I}}\};$
[2] $"\Dashv_{\mathfrak{I}}" \in \{\Dashv_{\mathfrak{I}}, \DashV_{\mathfrak{I}}, \dashv_{\mathfrak{I}}, \dashV_{\mathfrak{I}}, \not\Dashv_{\mathfrak{I}}, \not\DashV_{\mathfrak{I}}, \not\dashv_{\mathfrak{I}}, \not\dashV_{\mathfrak{I}}\};$
[3] $\alpha \Rightarrow ((\alpha \vDash_{\mathfrak{I}}); (\vDash_{\mathfrak{I}} \alpha));$
[4] $((\alpha \Dashv_{\mathfrak{I}}); (\Dashv_{\mathfrak{I}} \alpha)) \Leftarrow \alpha;$
[5] $\alpha \Rightarrow ((\alpha \vDash_{\mathfrak{I}}) \vee (\vDash_{\mathfrak{I}} \alpha));$
[6] $((\alpha \Dashv_{\mathfrak{I}}) \vee (\Dashv_{\mathfrak{I}} \alpha)) \Leftarrow \alpha;$
[7] $\alpha \Rightarrow ((\Dashv_{\mathfrak{I}} \alpha); (\vDash_{\mathfrak{I}} \alpha));$
[8] $((\alpha \Dashv_{\mathfrak{I}}); (\alpha \vDash_{\mathfrak{I}})) \Leftarrow \alpha;$
[9] $\alpha \Rightarrow ((\alpha \vDash_{\mathfrak{I}}); (\alpha \Dashv_{\mathfrak{I}}));$
[10] $((\vDash_{\mathfrak{I}} \alpha); (\Dashv_{\mathfrak{I}} \alpha)) \Leftarrow \alpha;$
[11] $\alpha \Rightarrow ((\alpha \vDash_{\mathfrak{I}}); (\vDash_{\mathfrak{I}} \alpha); (\alpha \Dashv_{\mathfrak{I}}); (\Dashv_{\mathfrak{I}} \alpha));$
[12] $((\alpha \vDash_{\mathfrak{I}}); (\vDash_{\mathfrak{I}} \alpha); (\alpha \Dashv_{\mathfrak{I}}); (\Dashv_{\mathfrak{I}} \alpha)) \Leftarrow \alpha;$
. . . .

In the last formulas, α marks the so-called intentional information of information β and operators $\vDash_{\mathfrak{I}}$ and $\Dashv_{\mathfrak{I}}$ are intentional informing or intentional non-informing of information β. Now, the informational modus rectus can be expressed in the following way: let α or $\alpha(\beta)$ be the intentional information of information β in question and let the implicit intentional informing of β be marked by $\mathfrak{I}$ or $\mathfrak{I}(\beta)$. Further, let β inform information γ. Let the task of informational modus rectus be to detach or extract the implicit intentional informing $\mathfrak{I}$ or $\mathfrak{I}(\beta)$ out of the processes of informing $\beta \vDash \gamma$, $\gamma \Dashv \beta$, etc. Thus, the following cases of informational modus rectus can be constructed:

[4.14]:
[1] $"\vDash_{\mathfrak{I}}" \in \{\vDash_{\mathfrak{I}}, \VDash_{\mathfrak{I}}, \vdash_{\mathfrak{I}}, \Vdash_{\mathfrak{I}}, \nvDash_{\mathfrak{I}}, \nVDash_{\mathfrak{I}}, \nvdash_{\mathfrak{I}}, \nVdash_{\mathfrak{I}}\};$
[2] $"\Dashv_{\mathfrak{I}}" \in \{\Dashv_{\mathfrak{I}}, \DashV_{\mathfrak{I}}, \dashv_{\mathfrak{I}}, \dashV_{\mathfrak{I}}, \not\Dashv_{\mathfrak{I}}, \not\DashV_{\mathfrak{I}}, \not\dashv_{\mathfrak{I}}, \not\dashV_{\mathfrak{I}}\};$
[3] $((\alpha \Rightarrow ((\alpha \vDash_{\mathfrak{I}(\beta)}); (\vDash_{\mathfrak{I}(\beta)} \alpha)));$
$(\beta \vDash_{\mathfrak{I}(\beta)} \gamma)) \Rightarrow \mathfrak{I}(\beta);$
[4] $\mathfrak{I}(\beta) \Leftarrow ((\gamma \Dashv_{\mathfrak{I}(\beta)} \beta);$
$(((\alpha \Dashv_{\mathfrak{I}(\beta)}); (\Dashv_{\mathfrak{I}(\beta)} \alpha)) \Leftarrow \alpha));$
[5] $((\alpha \Rightarrow ((\alpha \vDash_{\mathfrak{I}(\beta)}); (\vDash_{\mathfrak{I}(\beta)} \alpha))) \wedge$
$(\beta \vDash_{\mathfrak{I}(\beta)} \gamma)) \Rightarrow \mathfrak{I}(\beta);$
[6] $\mathfrak{I}(\beta) \Leftarrow ((\gamma \Dashv_{\mathfrak{I}(\beta)} \beta) \wedge$
$(((\alpha \Dashv_{\mathfrak{I}(\beta)}); (\Dashv_{\mathfrak{I}(\beta)} \alpha)) \Leftarrow \alpha));$
[7] $(\beta; (\beta \vDash \gamma)) \Rightarrow \mathfrak{I}(\beta);$
[8] $\mathfrak{I}(\beta) \Leftarrow ((\gamma \Dashv \beta); \beta);$
[9] $(\beta \wedge (\beta \vDash \gamma)) \Rightarrow \mathfrak{I}(\beta);$
[10] $\mathfrak{I}(\beta) \Leftarrow ((\gamma \Dashv \beta) \wedge \beta);$
. . . .

etc. Formulas [3] - [6] can be read in the following way: if α is intentional information of information β and if β intentionally informs information γ, then $\mathfrak{I}(\beta)$ is intentional informing, which can be detached out of the process $\beta \vDash_{\mathfrak{I}(\beta)} \gamma$, etc. by the informational modus rectus. Formulas [7] - [10] are generalizations which can be read: if β is information (and for any information some intentionality is supposed) and if information β informs information γ, then the intentionality of β can be detached out of this informing by the informational modus rectus.

4.7.4. The Informational Modus Obliquus

What could be the aim of informational modus obliquus? What kind of information could it extract from the information in question? Does it concern information of unawareness, doubt, indirectness, and absurdity? Informational modus obliquus deviates from intentional line of discourse, not going straight to the point, so it may enter into the domain of contradiction, by which the intention is not openly shown.

Let β be information in question, being investigated by informational modus obliquus against information of absurdity, α, with an implicit absurd informing $\mathfrak{A}$. If possible, informational modus obliquus may deliver α through the absurd informing of information β, when β informs γ. First, for information of absurdity α there is

[4.15]:

[1] $``\models_{\mathfrak{A}}" \in \{\models_{\mathfrak{A}}, \VDash_{\mathfrak{A}}, \vdash_{\mathfrak{A}}, \Vdash_{\mathfrak{A}}, \nvDash_{\mathfrak{A}}, \nVDash_{\mathfrak{A}}, \nvdash_{\mathfrak{A}}, \nVdash_{\mathfrak{A}}\};$
[2] $``\Dashv_{\mathfrak{A}}" \in \{\Dashv_{\mathfrak{A}}, \DashV_{\mathfrak{A}}, \dashv_{\mathfrak{A}}, \dashV_{\mathfrak{A}}, \nDashv_{\mathfrak{A}}, \nDashV_{\mathfrak{A}}, \ndashv_{\mathfrak{A}}, \ndashV_{\mathfrak{A}}\};$
[3] $\alpha \Rightarrow ((\alpha \models_{\mathfrak{A}}); (\models_{\mathfrak{A}} \alpha));$
[4] $((\alpha \Dashv_{\mathfrak{A}}); (\Dashv_{\mathfrak{A}} \alpha)) \Leftarrow \alpha;$
[5] $\alpha \Rightarrow ((\alpha \models_{\mathfrak{A}}) \vee (\models_{\mathfrak{A}} \alpha));$
[6] $((\alpha \Dashv_{\mathfrak{A}}) \vee (\Dashv_{\mathfrak{A}} \alpha)) \Leftarrow \alpha;$
[7] $\alpha \Rightarrow ((\Dashv_{\mathfrak{A}} \alpha); (\models_{\mathfrak{A}} \alpha));$
[8] $((\alpha \Dashv_{\mathfrak{A}}); (\alpha \models_{\mathfrak{A}})) \Leftarrow \alpha;$
[9] $\alpha \Rightarrow ((\alpha \models_{\mathfrak{A}}); (\alpha \Dashv_{\mathfrak{A}}));$
[10] $((\models_{\mathfrak{A}} \alpha); (\Dashv_{\mathfrak{A}} \alpha)) \Leftarrow \alpha;$
[11] $\alpha \Rightarrow ((\alpha \models_{\mathfrak{A}}); (\models_{\mathfrak{A}} \alpha); (\alpha \Dashv_{\mathfrak{A}}); (\Dashv_{\mathfrak{A}} \alpha));$
[12] $((\alpha \models_{\mathfrak{A}}); (\models_{\mathfrak{A}} \alpha); (\alpha \Dashv_{\mathfrak{A}}); (\Dashv_{\mathfrak{A}} \alpha)) \Leftarrow \alpha;$
. . . .

Let us show how information of absurdity α and its implicit informing $\mathfrak{A}$ can be detached out of information in question β, when $\beta \models \gamma$, $\gamma \Dashv \beta$, etc. There can exist the following informational modi obliquus:

[4.16]:

[1] $``\models_{\mathfrak{A}}" \in \{\models_{\mathfrak{A}}, \VDash_{\mathfrak{A}}, \vdash_{\mathfrak{A}}, \Vdash_{\mathfrak{A}}, \nvDash_{\mathfrak{A}}, \nVDash_{\mathfrak{A}}, \nvdash_{\mathfrak{A}}, \nVdash_{\mathfrak{A}}\};$
[2] $``\Dashv_{\mathfrak{A}}" \in \{\Dashv_{\mathfrak{A}}, \DashV_{\mathfrak{A}}, \dashv_{\mathfrak{A}}, \dashV_{\mathfrak{A}}, \nDashv_{\mathfrak{A}}, \nDashV_{\mathfrak{A}}, \ndashv_{\mathfrak{A}}, \ndashV_{\mathfrak{A}}\};$
[3] $``\models" \in \{\models, \VDash, \vdash, \Vdash, \nvDash, \nVDash, \nvdash, \nVdash\};$
[4] $``\Dashv" \in \{\Dashv, \DashV, \dashv, \dashV, \nDashv, \nDashV, \ndashv, \ndashV\};$
[5] $((\beta \models_{\mathfrak{A}} \alpha); (\beta \models \gamma)) \Rightarrow (\alpha \models_{\mathfrak{A}} \gamma);$
[6] $(\gamma \Dashv_{\mathfrak{A}} \alpha) \Leftarrow ((\gamma \Dashv \beta); (\alpha \Dashv_{\mathfrak{A}} \beta));$
[7] $(((\alpha \subset_{\mathfrak{A}} \beta); (\mathfrak{I}_{\mathfrak{A}} \subset \mathfrak{I}(\beta))); (\beta \models \gamma)) \Rightarrow$
$(\alpha, \mathfrak{I}_{\mathfrak{A}}(\alpha) \models_{\mathfrak{A}} \gamma);$
[8] $(\gamma \Dashv_{\mathfrak{A}} \mathfrak{I}_{\mathfrak{A}}(\alpha), \alpha)$
$\Leftarrow ((\gamma \Dashv \beta); ((\alpha \subset_{\mathfrak{A}} \beta); (\mathfrak{I}_{\mathfrak{A}} \subset \mathfrak{I}(\beta))));$
[9] $(((\alpha \subset_{\mathfrak{A}} \beta); (\alpha \models_{\mathfrak{A}} \mathfrak{I}(\beta))); (\beta \models \gamma)) \Rightarrow$
$([_{\mathfrak{A}}(\beta) \llcorner_{\mathfrak{A}} \mathfrak{I}_{\mathfrak{A}}(\gamma));$
[10] $(\mathfrak{I}_{\mathfrak{A}}(\gamma) \dashv_{\mathfrak{A}} \mathfrak{I}_{\mathfrak{A}}(\beta))$
$\Leftarrow ((\gamma \Dashv \beta); ((\mathfrak{I}(\beta) \Dashv_{\mathfrak{A}} \alpha); (\alpha \subset_{\mathfrak{A}} \beta)));$
. . . .

To these formulas some exlpanations are necessary. We introduced additional operators: $\subset$ marks informational inclusion and $\subset_{\mathfrak{A}}$ the so-called absurd informational inclusion; $\mathfrak{I}_{\mathfrak{A}}$ is an informationally non-identified absurd informing and $\mathfrak{I}(\xi)$ and $\mathfrak{I}_{\mathfrak{A}}(\xi)$ the so-called ξ-identified implicit informing and implicit absurd informing. Within this context, formulas of [4.16] can be read as follows:

[5]: if β absurdly (obliquely) informs α in one way and if β informs γ in one way, then α absurdly informs γ in one way.

[6]: if β absurdly (obliquely) informs α in another way and if β informs γ in another way, then α absurdly informs γ in another way.

[7]: if α absurdly exists (informs) within β in one way, if absurd (oblique) informing $\mathfrak{I}_{\mathfrak{A}}$ exists (informs) within the informing of β in one way, i.e. $\mathfrak{I}(\beta)$, and if β informs γ in one way, then absurd (oblique) information α and its informing absurdly inform γ in one way.

[10]: if α absurdly exists (informs) within β in another way, if α absurdly informs the informing $\mathfrak{I}(\beta)$ in another way, and if β informs γ in another way, then the absurd informing $\mathfrak{I}_{\mathfrak{A}}(\beta)$ causes the appearance of the absurd informing $\mathfrak{I}_{\mathfrak{A}}(\gamma)$ in another way. In common sense it would mean that γ begins to observe the absurd informing of β, which was caused by α.

In this sense, informational modus obliquus is by itself information, which is capable to reveal oblique informing of informational entities. Informational midi are nothing else than informational means by which informational entities observe and conclude in particular, modi-characteristic ways on information in question.

4.7.5. The Informational Modus Procedendi, Modus Operandi, Modus Vivendi, Modus Possibilitatis, and Modus Necessitatis

Modi informationis in the above title can be discussed in a similar way as have been modus ponens, modus tollens, modus rectus, and modus obliquus, considering their beginning (or original) presuppositions as determined in [4]. Hopefully, the reader will be capable to construct adequate formulas according to cases listed in [4] and certainly to his/her own imagination. Since, modi informationis belong to the most provoking informational constructs through the history of human informational evolution. As said, modi informationis belong to certain types of informational arising and concern primarily the observational and concluding (inferential) nature of information.

5. CONCLUSION

To offer a satisfactory answer to the question of possible informational algebras, a sufficiently exhaustive overview of regular and diversified logical and algebraic approaches of sciences and philosophies would be necessary. Such overview could be the basis for a more systematic treatment of a general informational algebra and its particularizations. Nowadays, it seems quite possible that any particular algebra, irrespective of its scientific or philosophical nature, can be universalized onto the level of the proposed self-informational or general informational algebra. On this basis, an adequate informational language could be proposed, covering also the more and more actualizing field of information-oriented technology.

On the other side, the represented instrumentalism of formal approach (say, informational algebra) could become a way to the analysis, synthesis, intuition, and formalization in the field of sciences as well as philosophies, for they, in the last consequence, can be understood to be nothing less than sufficiently complex disciplines of their own information.

Informational algebra seems to be extremely flexible and remains open to any further informational imagination. Its language can be easily conformed to fit the needs, applications, and concepts which may arise additionally during the process of formalization. It means that the impact on already achieved formalism or determined formal system remains open for further development. In this respect, informational algebra performs as regular, live information. In this direction it is possible to search for a new language, accentuated in [12].

REFERENCES

[1] Železnikar, A.P., Informational Logic I, Informatica 12 (1988) 3, 26-38.

[2] Železnikar, A.P., Informational Logic II, Informatica 12 (1988) 4, 3-20.

[3] Železnikar, A.P., Informational Logic III, Informatica 13 (1989) 1, 25-42.

[4] Železnikar, A.P., Informational Logic IV, Informatica 13 (1989) 2, 6-23.

[5] Železnikar, A.P., Information Determinations I, Cybernetica 31 (1988) 3, 181-213.

[6] Železnikar, A.P., Information Determinations II, Cybernetica 32 (1989) 1, 5-44.

[7] Železnikar, A.P., Informational Principles and Formalization (in Slovene), Informatica 13 (1989) 3, 21-40.

[8] Železnikar, A.P., An Informational Theory of Discourse I, Informatica 13 (1989) 4, 16-37.

[9] Železnikar, A.P., Informational Principles and Formalization (in Slovene), Informatica 13 (1989) 3, 21-40.

[10] Bergson, H., The Creative Evolution (in Slovene), Cankarjeva založba, Ljubljana (1983).

[11] Heidegger, M., Das Ende der Philosophie und die Aufgabe des Denkens; in Zur Sache des Denkens, Niemeyer, Tuebingen (1969).

[12] Yokoi, T., Giving Priority to "Information-Oriented Technology" over "Computer-Oriented Technology", New Generation Computing 6 (1989) 359-360.

MODULA-2 AND SOFTWARE ENGINEERING

Gustav Pomberger
INstitut für Wirtschaftsinformatik
Johannes Kepler Universitl of Linz
A-4040 Linz, Austria, e-mail: K2G0190 a AEARN

Keywords: modula-2, software engineering, models, principles

Presented at the 1^{sr} Int'l Modula-2 Conference, October 12-13, 1989 Bled-Yugoslavia

OVERVIEW

The title of this paper draws together a programming language, Modula-2, and a discipline in the area of computer science, software engineering. This raises several questions:

What is meant by software engineering? This is by no means intended to be a rhetorical question, for the perceptions of software engineering's tasks varies pronouncedly from the viewpoints of theoreticians and practitioners.

Even greater divergence can be found on the question of what a programming language has to do with software engineering. Some consider the choice of a programming language of utmost importance to the success of a software project and the quality of the resulting product, while others view the language as the least important tool of the development process.

Even the question of whether programming is more a science or more an art (or perhaps even a craft) evokes avid disagreement. I do not want to renew this old feud; I simply want to establish that elements of all of them are inherent in software development at this time, and this is likely to remain the case in the future.

When I use the term software engineering, I mean the application of scientific knowledge for the efficient production and application of reliable and efficient software (see [16]).

The successful development of large software systems is usually a multistage process. It usually begins with the determination and documentation of the functions and individual actions that are expected of the software system. This leads in the specification phase to a contract between the client and the software developer (requirements definition) that precisely delineates what the software system must be capable of.

The specification phase is followed by the design phase, which determines what kind of system architecture can meet the given requirements. The implementation phase attends to the realization of the complete design concept in a programming language.

The implementation of every single system component must be systematically tested. Subsequently the whole system must be tested with the goal of finding as many errors as possible and assuring that the implementation meets the requirements definition.

Upon completion of the test phase the software is installed and handed over to the client. The task of software maintenance is both to correct errors that arise during operation and to make system modifications and extensions. This task again includes all activities mentioned above—from the revision of the requirements analysis through renewed testing.

An engineering discipline is characterized by the construction of tools that help to systemize and rationalize the product development process, to improve the quality of products, and to guarantee efficient maintenance. A particularly important step in this direction was the development of programming languages as tools intended to help to achieve these goals. Unlike many others, I agree with B.W. Boehm that "choosing a programming language is like choosing a wife. It is hard to undo after getting involved and not to be taken lightly."

I likewise agree with R. Wiener and R. Sincovec [22] that "the choice of a programming language for implementing a large-scale software system is critical because the features of a programming language are strongly related to the software engineering process. Languages differ in the degree to which they support: readability, modular software construction, the control of side effects, information hiding, data abstraction, structured flow control, separate compilation with consistency checking, type checking among various components, dynamic memory management, and run-time checking. Languages that offer strong support in the above-listed areas provide the basis for constructing reliable and maintainable software."

I will discuss to what extent the principles of software engineering known today are supported (or not supported) by Modula-2. Detail is restricted by the size and extent of this paper. For this reason only the principles that I consider most important will be discussed. I will briefly discuss the characteristics of software development models in order to be able to explore:

- specification principles
  - requirement exploration by prototyping
- design principles
  - module-oriented architecture design
  - abstract data structures
  - abstract data types
  - functional abstraction
  - object-oriented design principles
- other evaluation criteria
  - division of labor in software development
  - structuring in the small (structured programming)
  - guaranteeing reliability and maintainability
  - exception handling
  - reusability of library modules
  - portability

SOFTWARE DEVELOPMENT MODELS

Wherever people are confronted with complex design tasks to be solved, they attempt to systematically organize the problem solving process, that is, to define an approach model. Such a model determines which criteria are to govern the problem solving process. It decomposes the problem-solving process into managable steps and determines what results must be produced after execution of a given step. This enables a stepwise planning, decision and implementation process.

These steps collectively and the chronological order of their execution is known as the software life cycle, an already classical term in computer science. The software life cycle has been described in numerous variations and forms (see [8], [20], [16], and [18]).

Studies have shown that the life cycle-oriented development method is the most commonly used approach in current software development, and that it has in general paid off. Application in the field, however, has also shown the limits and the weaknesses of this approach:

The model is based upon the (incorrect) assumption that the development process tends to be linear and that iterations between phases occur only as exceptions to the rule. Strict application of this development method requires that one phase can only be begun after the preceding phase is completed, that is, when the respective intermediate products are available. In reality, however, a complete specification or a suitable system architecture can seldom be produced straightoff. Usually the later phases have a strong impact on the earlier phases.

The strict discrimination of the individual phases is an unacceptable idealization. In reality the activities of the phases overlap and interaction between phases is much more complex than that exhibited in the sequential input/output model.

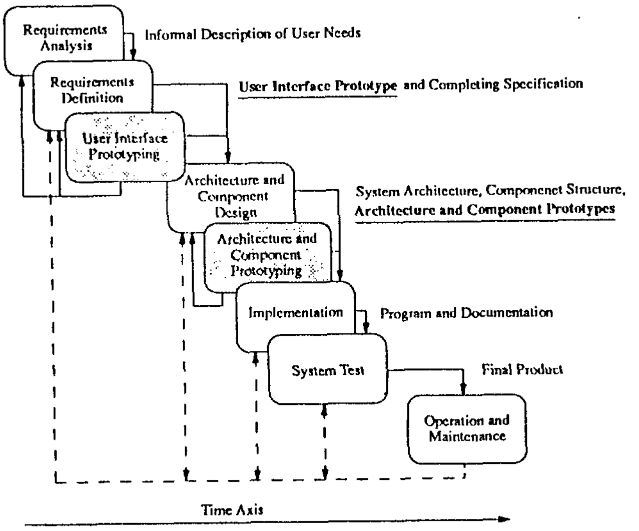

Requirements Analysis
Informal Description of User Needs
Requirements Definition
User Interface Prototype and Completing Specification
User Interface Prototyping
Architecture and Component Design
System Architecture, Componenct Structure, Architecture and Component Prototypes
Architecture and Component Prototyping
Implementation
Program and Documentation
System Test
Final Product
Operation and Maintenance
Time Axis

Figure 1 Prototyping-Orineted Software Life Cycle

The strictly sequential approach leads to tangible products or components being available only at a relatively late stage. Yet experience shows that the validation process cannot get by without experiments close to reality. Furthermore, modifications requested by the client can only be expressed relatively late, and integrating them at that stage can lead to substantial overhead.

It is often assumed—and current reports from research and industry confirm this assumption—that a prototyping-oriented development methodology can resolve some of the weaknesses of the life cycle-oriented development approach. A prototyping-oriented development is not radically different from a purely phase-oriented development strategy. Furthermore, the two are to be viewed more as complementary than as alternative. They differ most in the procedures and the results produced in the individual phases. Although the distinction of phases is maintained, problem analysis and specification overlap chronologically a great deal, and design, implementation and testing very much blend into one another (see Figure 1).

SPECIFICATION PRINCIPLES—EXPLORATION OF USER REQUIREMENTS

As our development model shows, one element of knowledge inherent in our definition of software engineering is that the specification and design processes should be carried out in a prototyping-supported manner.

The development of the user interface, for example, proves an exceptionally difficult task because the evaluation thereof is guided by highly subjective criteria and the user is hardly able to define in advance what he/she considers to be convenient interaction. Prototyping is an important—and, from my point of view, in most cases absolutely necessary—vehicle for the exploration of user requirements and thus for the specification of user interfaces.

We normally distinguish two approaches to prototyping: ***reusable code*** and ***executable specifications***. Modern programming languages like Modula-2 are significantly better suited for producing reusable components than was the case in older programming languages. Modula-2 is particularly handy for the building of module libraries. From the viewpoint of prototyping, however, a number of problems remain unsolved if one uses conventional programming languages such as Modula-2 for prototyping activities:

- How can the functionality of a library modules be provided generally enough that they can be integrated into a given prototype?
- The degree of abstraction of Modula-2 modules is too low; a prototype designer must revise code for every modification, no matter how small, and make his/her changes directly in the code; details of the prototype cannot be discussed with the user.
- Turnaround times for iterative refinements in a prototype are simply too high.

Although the availability of module libraries is steadily improving and the taxonomy of software components is beginning to emerge (already it is possible to distinguish components such as mathematical routine packages, message channels, input/output packages, parsers, scanners and filters to name a few), Modula-2 libraries are only to a limited extent (if at all) capable of meeting the demands of reusability of code as required for prototyping.

The other approach to rapid prototyping, executable specifications (an object of intense research efforts) is likewise not supported by Modula-2.

Since on the one hand we use Modula-2 as our implementation language in most cases in our research group (and the choice of Modula-2 is to be credited with considerable increases in efficiency and quality), and on the other hand we have recognized the value of prototyping-oriented software development and evaluated this in several research projects, it became necessary to develop special tools for prototyping.

For the prototyping process during the analysis and specification phases, we developed a declarative language for the description of executable specifications—our User Interface Specification Language (UISL, see [12] and [17]). Searching for methods for integration of high level prototypes and application parts written in Modula-2 as well as for validation of a system architecture before completely implementing it, we developed SCT, a tool for hybrid execution of hybrid software systems (see [1] and [2]). It allows for hybrid execution of Modula-2 software systems at any time during their development. Designed but not implemented modules are simulated, partially coded modules are interpreted, and modules which are coded and tested are directly executed. Furthermore, SCT allows for execution of hybrid software systems (systems written in different languages). This is achieved by providing the possibility of adding new execution tools to SCT's hybrid execution system (e.g., an interpreter for a user interface description language).

Applying SCT high level prototypes can be easily enhanced with Modula-2 code, allowing the development of better exploratory and evolutionary prototypes. Furthermore, SCT supports the validation of system architectures represented by Modula-2 definition modules by simulating data and control flows. Finally, SCT provides a comfortable interpretative programming environment allowing for fast implementation and experimentation with different realizations of the functionality provided by a module.

SOFTWARE DESIGN PRINCIPLES

The task of the design phase is the determination of the architecture of a software system—that is, to decide how to build the proposed system—with the goal of achieving an implementation that is as efficient as possible and meets all quality requirements. Because of the practically unlimited number of possibilities of determining the design of a planned system, the decisions made and the methods used in this phase pronouncedly influence the quality of the product and thereby its maintenance costs and degree of reliability.

The production of complex program systems necessitates a division of labor; that is, multiple persons are involved in the software development. It is clear that software development is a creative process, that the experience, creativity and innovation of the designer significantly affects the quality of the product. But as a rule the complexity of design decisions is so high that a systematic approach—a method and associated design principles—must be adhered to in order to guarantee a resulting product that is reliable and easy to maintain.

All software design involves a process of abstraction. Objects and operations identified in the real world domain must be modelled and expressed as corresponding operations and objects of the problem-solving domain.

Module-oriented and *object-oriented software design* are fundamental design principles resulting from computer science research in the 60s and 70s. Wiener and Sincovec write [22]:

"No longer is it necessary for the system designer to map the problem domain to the predefined data and control structure present in the implementation language. Instead, the designer may create his or her own abstract data types and functional abstractions and map the real world domain to these programmer-created abstractions. The mapping, incidentally, may be much more natural because of the virtually unlimited range of abstract types that can be invented by the software designer. ... the payoff for modular software design and implementation occurs when repairs or additions must be made to a software system."

Module-Oriented Design Principles

The goal of modular system design is the decomposition of a program system into a hierarchy of abstractions about which Wirth writes [24]: "The principle motivation behind the partitioning of a program into modules is—beside the use of modules provided by other programmers—the establishment of a hierarchy of abstraction."

The pillars of modular system architecture are ***module independence*** and ***data abstraction***. Module independence (freedom from interference) means that any module can be replaced by another module that adheres to the module interface without necessitating further changes in the system. That is, it must be possible to change details of the implementation of a particular module without influencing the remaining system components.

The basic building blocks of modularly constructed software systems are:

- abstract data structures
- abstract data types
- functional abstraction
- abstract, explicitly defined module interfaces

Although software engineering courses often teach that design should occur completely independently of the implementation language. I believe that this is only useful if the implementation language does not meet the requirements of software engineering. We are aware that a language reflects the habits and thought patterns of its designer. There is even a relationship between a natural language and the way a person who speaks the language thinks. The same is true for programming languages. The knowledge that lent it its structure and the concepts that form its basis influenced the way a programmer thinks, his/her design style, and the structure of the system he/she designs. The choice of a programming language often even determines how the task is solved because the language supports or excludes certain approaches to a solution. For example, a recursive tree traversal would never come to a Fortan programmer's mind.

Furthermore, we expect a good implementation language to be able to reflect the decomposition structures, abstraction levels, data structures and module interfaces that are identified in the design stage and that these can be tested at the interface level before all the implementation details are known.

The degree to which these requirements can be met is dependent upon the choice of a programming language. The question is to what extent Modula-2 supports the above-named criteria for modular system architectures.

The Modula-2 Module Concept

The realization of the module definition as given above is supported in an elegant manner by the module concept of Modula-2. The modular structure of Modula-2 can be viewed as a fence that encloses objects (data structures and procedures) and encapsules them apart from their environment. This fence can be opened for the purpose of communicating with the environment. However, the programmer must explicitly establish which objects are to be made known (that is, exported) to the outside and which objects the module will need (that is, import) from its environment. This meets the requirements of explicitly defined module interfaces.

From the viewpoint of the abstraction principle, the export interface can be seen as its specification. It contains all information regarding what the module is expected to do (that is, what objects and functions it makes available) and hides all details of the implementation thereof. It is thus also useful to separate the texts of the module specification and its implementation description. Modula-2 meets this requirement by separating the definition and implementation parts of a module.

One of the most important aspects of modularly constructed software systems is thus an explicit description of mutual effects (that is, interdependencies) among modules. The importance of such explicitness follows from the observation that all the effects of a local change on the global system must be completely determined by the dependency relations. In Modula-2 interfaces of modules as seen by the programmers are called definitions. Such module definitions may be regarded as public projections, and there is exactly one public projection of each module.

But this situation is less than satisfactory. In practice we often encounter situtions in which multiple views of a module can be seen as befitting the problem. Consider, for example, a module X for managing assembly lists in a production planning and control system. It is clear that a module A from the area of design requires different access functions than a module B from the area of work scheduling or a module C from the area of material disposition. A, B and C all work with the encapsuled data structures in X, although in different ways and with different requirements for access to the data structures of the assembly list encapsulated in X.

This is just one of many examples in which multiple interfaces to a single module are necessary, each with different levels of abstraction, in order to guarantee adequate application of the module with respect to the problem at hand.

Due to the one-to-one correspondence of modules and interface descriptions in Modula-2, multiple interfaces cannot be satisfactorily realized. Either all the different views are packed into a single interface—which increases the complexity of the interface, reduces the safety of the module, and destroys part of the abstraction—or the implementation is duplicated—that is, reusability is lost and maintainability is reduced.

Multiple interfaces of modules are thus an important concept in software engineering that is not supported by Modula-2. Ideas on the implementation of multiple interfaces can be found in [10].

Abstract Data Structures (Information Hiding)

The basic concept of Modula-2 is the establishment of a hierarchy of abstractions. Naturally, this includes the implementation of *abstract data structures.* The problem of specially identifying access operations to a(n abstract) data structure is solved in Modula-2 by dividing a capsule into two parts: one part visible to the user (the specifications or interface part) and containing the declaration of all access operations and any exported data types; the other part invisible to the user (the implementation part) and containing declarations of encapsulated data and algorithms in the capsule.

The module concept of Modula-2 includes the export of not just procedures, data types and constants; variables can likewise be exported (for example, to make access to a single data element more efficient). If a variable is exported, its value can be changed by the importing module. However, this violates the principle of information hiding and it must be clear to the importer that he/she is working with global data, and the efficiency thereby attained is countered by the disadvantages of exchanging data via global variables.

Modula-2 thus permits the implementation of data capsules, although the principle of information hiding is incompletely realized due to the possibility of exporting inner data structures together with their structure. In this sense it would be desirable to have exported variables that can be read but not written to by the client.

Abstract Data Types

Abstract data types are necessary when multiple examples of an abstract data structure are to be defined. Abstract data types can be implemented in Modula-2 by means of the module concept combined with the concept of opaque data types.

An abstract data type is defined as an opaque type in the definition module; that is, its realization remains hidden from the user and is determined in the implementation module—in contrast to Ada—which is a considerable advantage from the viewpoint of software engineering.

Unfortunately there is a catch to using opaque data types in Modula-2. Since the storage requirements of abstract data types must be known when the definition module is compiled, Modula-2 requires that the concrete type assigned to an abstract type must be of fixed length—that is, it must be a pointer type. Other types, in particular ARRAYs and RECORDs, are not permitted as abstract data types. They can, however, be realized as dynamically created objects and their pointer can be viewed as an abstract data type. This means a slight loss of efficiency, however. I personally consider the advantage of abstract data types to be greater than the disadvantage of the loss of efficiency.

It is much worse to have to dynamically allocate variables of abstract data types and to have to explicitly free their storage. Furthermore, the statement x:=y does not store a copy of y in x. This is a dangerous pitfall that can cause less experienced Modula-2 programmers to avoid the use of abstract data types.

In the process of designing a software system, we usually encounter modules or procedures that have a similar purpose but operate on data objects of different types, for example, modules for stacks, queues, trees, etc. What we want to have is a construct that permits the definition of templates for program units that need to be written only once and then tailored to the particular needs at translation time. This would be possible with *generic units*, but generic units are not available in Modula-2.

The data type WORD or ARRAY OF WORD serves as a lifebuoy in such cases. This allows, for example, the creation of a very general stack suitable for accepting simple objects (for example, CARDINAL) as well as structured objects (such as ARRAYs and RECORDs). I consider the omission of generic units (which are most uncomfortable from the viewpoint of the compiler designer) to be a clever decision which, because of the self-help available in the type WORD, is also acceptable at the practitioner level.

Functional Abstraction

Many software developmenters construct the software system architecture as a hierarchy of functional components, i.e., they employ the method of task-oriented stepwise refinement. Functional aspects are the focus of the method. Starting with the functional requirements, the task is decomposed into subtasks; each subtask (functional component) is then handled separately and again decomposed into subtasks until the resulting subtasks become so simple that they can be described with algorithms. That is, top down design proceeds from the general to the specific, from an identification of major system components to subcomponents and sub-subcomponents and so forth.

In the implementation we want to realize the hierarchical levels of the system architecture by mapping the functional components onto a set of (possibly nested) procedures that are used to implement the functional abstractions. The only language features we need to support top down design by stepwise refinement are procedures and the ability to group functional components into functional subsystems.

Through its procedure and module concepts, Modula-2 completely supports this method and permits the interfaces of the functional components and functional subsystems to be described precisely, yet, as the design process requires, abstractly enough.

Object-Oriented Design Principles

A design principle which has aroused a great deal of interest recently in computer science is *object-oriented system design*. Reduced to its fundamentals, object-oriented programming generates software by reproducing object descriptions. An object description contains definitions of data along with the specifications of actions that can be applied to these data.

In contrast to modular programming, object descriptions are only a kind of type description and do not form actually existing constructs as does a module in the sense of Modula-2. Only when an object description is instantiated is an object created.

However, object-oriented programming is more than just using abstract data types. It also involves inheritance and dynamic binding.

An important property of object-oriented system design is that the object descriptions do not contain complete definitions of the object's behavior and attributes, that is, all its data and actions. The object descriptions are ordered in a hierarchy in such a way that at any given hierarchy level only such data and actions are specified as were not already defined in superordinate object descriptions. Modifications of data and actions are thereby made without altering the superordinate object descriptions. This distinguishes object-oriented software development from module-oriented programming, in which the reuse of a module is only possible without changes in its implementation if the module's function completely fits into the new context without change.

The strength of object-oriented system design lies in the possibility of incrementally enhancing and adapting object descriptions without touching their code in the process. Instead of the libraries used in modular programming—with their reusable function modules whose components can be used in the construction of software—object-oriented programming uses libraries of object description hierarchies that form application frameworks. Examples include Smalltalk [9], MacApp [19] and ET++ [21].

As a rule, object-oriented programming builds on applications or parts of applications that are adapted to specific requirements, yet without changing these parts themselves. Thus later modifications can be made on the prefabricated application parts that remain completely transparent and spread to all derived applications without any further overhead.

The requirements placed on programming languages which support object-oriented system construction match those for languages which support modular system construction in many respects. In addition, they must support the following concepts (see [3]):

- *Data abstraction:* The description of abstract data types in the sense that they can occur directly in the declaration of other data structures must be possible.
- *Inheritance:* It must be possible to derive new data types by extending or modifying attributes and operations of existing types without needing to modify the description of the base types. Instances of a class C built on the basis of a class B are said to inherit the properties of B.
- *Polymorphism:* The compatibility of derived data types and their base types must be guaranteed. Object variables must be able to assume values of different (but related) data types at run-time.
- *Dynamic binding:* In the course of operations with objects, there must be the possibility of determining at run-time the concrete actions to be executed (dependent on the current dynamic data type of the objects).

In the object-oriented nomenclature, an abstract data type is known as a class. Every class defines which attributes its instances (the so-called objects) have and which operations are possible with them. Activating an operation with an object is often termed sending a message to the object. The object reacts by executing a method. A method describes which actions are to serve as the realization of an operation. This assignment of methods to messages is determined for each class by the respective class definition. The effect of sending a message differs from procedure invocations in conventional programming languages in that the determination of which method is to be executed occurs at run time.

The question is whether Modula-2 can be used to realize object-oriented system architectures and, if so, how it can be done. Object-oriented programming does not necessarily require an object-oriented programming language. Suggestions on how to implement objects in Modula-2 can be found in [4]. Every individual class can be defined in a separate definition module. Objects can be defined as pointers to records with two components: a pointer to a data structure describing its class and a pointer to the object's data (that is, instance variables). A class can be defined by a record containing a pointer to its superclass, the name of the class, and a collection of procedure variables which represent the messages understood by objects of this class.

Of course, some deficiencies must also be mentioned (see [4]). The programmer must be aware of the fact that objects are implemented as pointers. Thus, each object must explicitly be created. Also, the statement x:=y does not create a copy of the object y. Instead, a message send must be used. Changing a superclass' definition module requires changes to all of its subclasses. The requirement that every class must be defined in a separate definition module can lead to a large collection of modules that is difficult to understand and maintain.

Thus I cannot recommend Modula-2 to construct object-oriented system architectures as it was not designed as an object-oriented language. But many of the deficiencies mentioned above can be removed by attaching minor extensions to Modula-2.

There are, of course, some object-oriented extensions of Modula-2, among them Modula-3 [5] and pl Modula [11]. Niklaus Wirth himself also designed a new language named Oberon [23] that is based on Modula-2. Oberon was not designed as an object-oriented language either, but readily lends itself to the concept using type extensions and procedure variables. And an experimental extension of Oberon, called Object Oberon, has been developed that incorporates the concepts of class, method and message [14]. Including these concepts in Oberon improves its capabilities for object-oriented programming.

OTHER EVALUATION CRITERIA

I have discussed to what extent Modula-2 supports the most important software engineering principles for the exploration of user requirements (prototyping), for mastering complexity (structuring in the large), for engineering interfaces (information hiding, data abstraction), and for the design of the architecture of software systems (modular system construction, object-oriented system construction).

Beyond these aspects, we are interested from a software engineering viewpoint in several other criteria and how these are supported by the choice of Modula-2 as implementation language, for example:

- division of labor in software development
- structuring in the small (structured programming)
- guaranteeing reliability and maintainability
- exception handling
- reusability of library modules

Division of Labor in Software Development

The process of dividing the work load in software development is significantly supported if:

- separate interface description and implementation description of the system components is possible;
- separate compilation of units with strict cross checking is possible;
- type consistency checking between various components is provided; and
- the execution of a program unit is automatically prevented if the interface of a user component was modified and no consistency check followed the modification.

All of these properties are supported by Modula-2. This reduces the chances of the hard-to-localize kind of errors that arise from incompatible interfaces in divided-labor software development.

The concept of separate compilation coupled with the concept of strict type binding contributes to drastically increased productivity in a divided labor setting in a revolutionary way that is unfathomable to programmers in conventional languages such as Fortran or Cobol, while simultaneously (and at almost no additional cost) heightening reliability and maintainability.

Structuring in the Small

The goal of structuring the control flow of algorithms is to establish a correspondence between the static formulation of an algorithm and its dynamic behavior, to thereby reduce its susceptability to errors, and to enable the verification of the algorithm. The most important measure in this direction is the avoidance of unlimited flow structures which result from undisciplined use of unconditional transfers of control (goto statements). Thus many consider the absolute avoidance of such statements to be a fundamental requirement of structured programming, and they insist that control flow is to be structured by including only constructs that have a single entry and a single exit.

Modula-2 does not completely meet all these requirements of fundamentalist structured programming, for Modula-2 provides the RETURN and EXIT statements. But these disguised gotos do not compromise the software engineering principle in an essential manner, and they sometimes increase the efficiency and readability of programs if a loop/exit is used instead of some boolean variables and a conditional transfer test to circumvent the need for a loop exit statement. This latter technique often detracts from program clarity.

Although it is, of course, clear that good programming style is not characterized by the absence of goto statements alone, the lack of a goto statement in Modula-2 forces programming with well-defined transfers of control. This is an important property of Modula-2 from the viewpoint of software engineering, and I agree with B. Meyer's observation [13]:

> "It is hard to understand that, twenty years after 1968, a single letter about the goto construction should trigger endless letters to the Communications of the ACM, many of them advocating the use of gotos. Why not Roman numerals?"

Guaranteeing Reliability and Maintainability

Prerequisites to a *reliable, maintainable* software product include clear, consistent specifications, followed by the clean design of a modular architecture, followed by a readable description of the implementation, and culminated by a rigorous, systematic testing procedure aimed at both the individual components and their interaction.

Module independence is certainly one of the most important factors in the design of reliable and maintainable software systems. Guaranteeing the reliability and maintainability of a program system is less expensive as the components of a program system are easier to tune, to correct or to adapt to new requirements without affecting other parts of the system. The ability of the software designers to create module independence is very much related to the choice of the programming language to be used in implementing the system. The prominent concepts of Modula-2, such as information hiding, data abstraction, splitting definition and implementation of modules, and side effect avoidance through the use of proper variable scoping greatly affect the type of design and implementation and enhance reliability and maintainability considerably.

The *readability* of an implementation description is likewise an important criterion for maintainability. It depencdends on the structuredness of the system, on programming style, and on the expressive power of the implementation language used. Significant improvements in program readability result from:

- the use of descriptive names of arbitrary length
- the ability to define type names
- the ability to use abstract data types
- the compulsion to use formal object declarations (this serves as an identifier glossary)
- the possibility of reusing identifier names in the same program at different levels of locality

The clear lexical and syntactic construction of Modula-2 and the possibility of meeting the criteria named above assure a high documentation value. So long as an appropriate programming style is maintained, Modula-2 programs are more readable than PL/I, Cobol, Fortran or (in particular) C programs.

An additional important criterion for reliability and maintainability of a software system is its *testability*, which means its suitability to checking its correctness and localizing errors. The most important criteria for testability, most of which are fulfilled by Modula-2 are:

+ *Modularity of the system:* The system architecture is formed by a hierarchy of abstractions (modules). The interaction of modules is explicitly defined (import/export interfaces). Constructs are provided for structuring modules (functional components). Each functional component of a module has its own scope (nested locality).

+ *The ability to avoid side effects:* Communication among program units can only occur via explicitly described interfaces. Each program component has its own scope. Combination of data objects is only possible if their data types are compatible; implicit conversions are precluded.

+ *The ability to guarantee information hiding and data abstraction:* The data contents of a module and their representations are not visible to the outside and are thus protected from procedures that access them. Only procedures declared locally to the data can access the data structures, that is, know their concrete representations. The use of external data structures is precluded (module decoupling).

+ *Structuring of the control flow:* The control flow of an algorithm reflects its static structure. That is, the exclusive use of flow structure constructs with a single entry and a single exit is encouraged.

+ *The ability to check the consistency of module and procedure interfaces:* Interface descriptions (import/export procedure interfaces) are of a nature that a compile time check can be made to determine whether the client and the server (module/procedure) match one another.

+ *The readability of the implementation* (see above).

+ *The availability of run time checking facilities* such as range check, index check, etc.

– *The ability to specify semantic aspects (assertion mechanism):* Procedures and lower level units (i.e., loops) can be provided with assertions that describe semantic aspects of the program segment and can be evaluated at run time. The underlying idea (see [13])is programming by contract: "Every structure is charged with a precise task, defined by a specification that states precisely the obligations on the client, limiting the routine's responsibility (the preconditions) and the obligations on the routine, guaranteeing the client a certain result (the postconditions)."

Modula-2 not only permits but considerably supports these criteria for increased testability and thereby for heightening the reliability and maintainability with a single exception. An *assertion mechanism* as it is found, e.g., in Eiffel [13] is lacking in Modula-2. This drawback is, however, easier to accept in modular programming than in object-oriented programming because dynamic binding can obscure what actually happens in an object-oriented program.

Modula-2 supports measures to guarantee reliability and maintainability to an incomparabily greater extent than Fortran, Cobol and C, the most-used programming languages today. Studies in our research area (development of software engineering tools) have shown that the overhead for testing and maintaining of projects with Modula-2 as implementation language were less than 50% of the overhead in similar projects in which PL/I and C were used.

Exception Handling

In the execution of a program, events or conditions can occur (e.g., protocol errors in the transmission of data) that require special treatment. Language constructs for describing and handling such events (exceptions) contribute to the reliability and clarity of program systems. Thus a number of programming languages (e.g., Ada, Clu, Eiffel) incorporate language constructs for exception handling.

Such constructs do not exist in Modula-2, an absence which has often been identified as a drawback of the language. I cannot agree with this verdict. In all our projects I never encountered a case where programming out exception handling posed difficulties or detracted from the clarity of the program. Furthermore, it is easily possible to implement a mechanism for exception handling in Modula-2 with the help of coroutines and/or library modules.

Reusability of Library Modules

Modula-2 has provided us in particular with the separation of an interface description from the implementation of a module and the possibility of modifying the implementation without needing to change anything else in the rest of the system in which the module is imbedded (not even recompilation). Since the introduction of the module construct in programming languages, programmers expect significant improvement in the reusability of prefabricated software units as well as the creation and distribution of powerful module libraries.

Typical library modules contain a collection of procedures that implement often needed functions and belong together in some manner, e.g., a trigonometry module; or they implement an abstract data type that provides the client with a new data type and the operations defined on it, e.g., stack, queue, tree, sparse matrices; or they model physical systems to operate between hardware components and the rest of the software system, e.g., device drivers, communications modules. In addition to physical systems, logical/conceptional systems are naturally likewise modelled, i.e., made useful for other software components at a higher abstraction level, e.g., graphic modules, database modules.

In software engineering practice the situation often arises that a library module almost but not quite meets the requirements of a new application. Modifications become necessary. If changes only affect the implementation part, there is less problem. However, the definition part is often affected as well (perhaps a new albeit trivial operation is needed). A change in the definition part carries with it the ramifications that all client systems have to be compiled anew. In order to avoid this, there is no alternative but to copy the original module and to make the changes in the copy. With time this can lead to a whole family of different yet closely related modules. If a fundamental aspect of this module family needs to be modified, an aspect which is common to all the members, then each member of the module family has to be modified.

Another drawback that restricts reusability is that modules in the sense of Modula-2 define a static object and do not permit the definition of an object type. A module can thus not be defined once and be repeatedly instantiated. This proves to be a particular impediment in modelling abstract data types. I have discussed how abstract data types are reproduced with the module concept: A data object must be explicitly allocated with the invocation of a procedure; the data type itself is referenced with an opaque pointer. Thus reproduced abstract data types cannot occur directly in other data structures, or be transferred to other processes, or be output directly to files; one has only the opaque pointer as reference to the abstract data type. Special procedures have to be defined for such operations for each data object and have to be invoked by the client at the right time. A disagreeable side effect is that the client has to treat abstract data types differently from real data types.

In order to guarantee a sufficient measure of reusabililty, it must be possible to apply abstract data types for defining arbitrary data structures. Furthermore, it must be possible to enhance abstract data types in a simple and flexible manner without violating the principle of information hiding. This is not possible in Modula-2—or at least only in a very troublesome manner. (I alluded to this in the section on object-oriented system construction.) The reusability of library modules in Modula-2 thus does not completely meet the requirements of modern software engineering. Thus in our software development environment module libraries were used only for elementary tasks.

SUMMARY

My goal was to subject Modula-2 to critical analysis. I did not do this on the level of D. Moffat, who wrote [15]: "Modula-2 is not a general purpose language. Every general purpose language must also include some way to deal with large fixed-precision numbers for monetary quantities." I also did not seek to discuss what N. Wirth's Modula-2 Report did not precisely define, as, e.g., in B.J. Cornelius [6], [7]. Instead I sought to give an overview of the extent to which the language meets the requirements of software engineering at the end of the 80s.

Needless to say, at the start of this decade Modula-2 was a jewel—indeed, a diamond—that enriched the programming landscape. The ability to combine multiple procedures into a module, information hiding, separate compilation with full interface consistency checking, the ability to formulate parallel processes by means of the elementary concept of coroutines with various synchronization mechanisms, the support of most of the concepts of software engineering familiar at that time, the high documentation value of Modula-2, the compactness of the language, and the elegant syntax compared to other programming languages made Modula-2 a powerful tool for software engineers. All this makes it most incomprehensible that only a small segment of software engineering, mainly the academic sector, made use of this milestone language.

Today, at the end of the 80s, the world looks a little different. Software engineering has continued to develop—inspired by the fruitful works of N. Wirth and others. New programming paradigms have their consolidation phases behind them and new requirements for programming languages have evolved as a consequence. From my point of view the most important are: the availability of constructs for realizing object-oriented software architectures, the ability to create multiple interfaces to modules with respect to objects, and assertion mechanisms provided by a language to increase the reliability of programs. It is clear that the programming languages of the 70s to which Modula-2 belongs cannot completely meet these requirements. But from my point of view, these enhancements can be attached to Modula-2 with minor extensions, the subject of work currently in progress.

One step in this direction was, as mentioned above, the development of Oberon and the enhancements that led to Object Oberon. Modula-2, in terms of the fundamental concepts of the language and its cleanness and simplicity, forms a significantly better basis for further development in the directions mentioned than other programming languages, in particular C, which is so questionable from a software engineering viewpoint.

Our research group is among those that are working on further developments in the area of programming languages—naturally based on the solid foundation that Modula-2 provides.

REFERENCES

1. Bischofberger W., Keller R., 1989, Enhancing the Software Life Cycle by Prototyping, Structured Programming, Vol. 10, No. 1, Springer.

2. Bischofberger W., Pomberger G., 1989, SCT—A Tool for Hybrid Execution of Hybrid Software Systems, Proceedings of the First Annual Modula-2 Conference, Bled, Yugoslavia.

3. Blaschek G., Pomberger G., Stritzinger A., 1989, A Comparison of Object-Oriented Programming Languages, Structured Programming, Vol. 10, No. 4, Springer.

4. Blaschek G., 1989, Implementation of Objects in Modula-2, Structured Programming, Vol. 10, No. 3, Springer.

5. Cardelli L. et al., 1988, Modula-3 Report, Olivetti Research Center.

6. Cornelius B.J. (ed), 1986, Problems with the Report on Modula-2, Version 8, IST/5/13 Working Group paper N103, British Standards Institute.

7. Cornelius B.J., 1988, Problems with the Language Modula-2, Software—Practice and Experience, Vol 18, No. 6.

8. Fairley R., 1985, Software Engineering Concepts, McGraw Hill.

9. Goldberg A., Robson D., 1983, Smalltalk-80, The Language and Its Implementation, Addison-Wesley.

10. Gutknecht J., 1989, Variations on the Role of Module Interfaces, Structured Programming, Vol. 10, No. 1, Springer.

11. Henne E., et al., 1988, Modula-2 User Manual, pl Gesellschaft für Informatik (German).

12. Keller R., 1989, Prototypingorientierte Systemspezifikation (Prototyping-Oriented System Specification), Verlag Dr. Kovac, Hamburg, (German).

13. Meyer B., 1989, From Structured Programming to Object-Oriented Design: the Road to Eiffel, Structured Programming, Vol. 10, No. 1.

14. Mössenböck H., Templ J., 1989, Object Oberon—A Modest Object-Oriented Programming Language, Structured Programming, Vol. 10, No. 4.

15. Moffat D.V., 1984, Some Concerns About Modula-2, Sigplan Notices, Vol. 19, No.12.

16. Pomberger G., 1986, Software Engineering and Modula-2, Prentice Hall.

17. Pomberger G., Bischofberger W., Keller R., Schmidt D., 1988, Topos - A Toolset for Prototyping-Oriented Software Development, Proceedings of the CGL4, Paris.

18. Pressman R.S., 1987, Software Engineering: A Practitioner's Approach, 2nd edition, McGraw-Hill.

19. Schmucker K., 1985, Object-Oriented Programming for the Macintosh, Hayden.

20. Sommerville I., 1985, Software Engineering, 2nd edition, Addison-Wesley.

21. Weinand A., et al., 1989, Design and Implementation of ET++, a Seamless Object-Oriented Application Framework, Structured Programming, Vol. 10, No. 2, Springer.

22. Wiener R., Sincovec R., 1984, Software Engineering with Modula-2 and Ada, John Wiley & Sons.

23. Wirth N., 1987, From Modula-2 to Oberon and the Programming Language Oberon, ETH Report, Zurich.

24. Wirth N., 1988, Programming in Modula-2, 4th edition, Springer.

SUITABILITY OF »CASE« METHODS AND TOOLS FOR COMPUTER CONTROL SYSTEM

INFORMATICA 1/90

Keywords: case methods, computer control, tools

Janko Černetič
Institut »Jožef Stefan«, Ljubljana

Some questions are considered concerning the introduction of CASE (Computer-Aided Sofware Engineering) methods and tools into the development of computer-based control systems, with particular emphasis to process control and the domestic working environment. In the first part of the paper, the advantages of using CASE are discussed: the general ones, corresponding to any computer-based system, and the specific ones, corresponding to computer control systems. In the second part of the paper, a short review of real-time CASE methodology is given and the main benefits and problems occurring during the practical use of CASE are mentioned.

PRIMJERNOST 'CASE' METODA I ORUĐA ZA SISTEME RAČUNARSKE AUTOMATIZACIJE. U radu su razmatrana pitanja u vezi sa uvođenjem CASE (Computer - Aided Software Engineering) metoda i oruđa pri razvoju računarskih sistema vođenja, s posebnim osvrtom na sisteme procesne automatizacije i na domaće uvjete rada. U prvom dijelu rada spominju se prednosti CASE; najprije one općije, koje važe za bilo koji računarski projekt, a zatim i specifične prednosti za projekte računarske automatizacije. U drugom dijelu rada spomenute su glavne koristi i problemi pri uvođenju tih metoda i oruđa.

INTRODUCTION

In the last few years, the acronym CASE, denoting Computer- Aided Software Engineering, has become a significant keyword for the modern software engineering community. According to E.J. Chikofsky (1988), "CASE is primarily a production - oriented integration technology to meaningfully improve software and systems development". In fact, CASE integrates methods, computer-aided tools and an appropriate working environment. It is effectivity- and production-oriented, therefore it can be seen as a sort of engineering.

In this paper we wish to consider briefly the above-mentioned trends, particularly in the context of using CASE within the development of computer-based control systems. The interested reader should also refer to our associated paper (Rihar and Černetič, 1989) which will give more details on the practical use of CASE tools.

Before we begin, let us explain why CASE is interesting in the area of modern control systems. As a rule, such systems are all computer-based and are being classified into the broad category of real-time systems (Hindin and Rauch- Hindin, 1983). If it is said that "the tar pit of software engineering will be sticky for some time to come" (adapted from Weiss, 1985, citing Brooks, 1982), this is the more true in the case of real-time and control systems, respectively. In this situation, CASE is giving some hope to the worried project managers.

Because a national development project has been launched also in Yugoslavia (IJS, 1988), with the aim to improve the current engineering practices in the development of process control systems for the chemical and other processing industries, our opinion is that we can not afford blindly to ignore CASE.

2. SOME FEATURES OF 'CASE'

In brief, the main features of CASE are the following. First, they are historically based on the well known methods of structured systems analysis and systems (and software) design (De Marco, 1978; Page-Jones, 1980). Second, they *are characterized by: a systematic definition* and analysis of the problem, lucid graphic representations of the concerned system, black-box simplification and multilevel hierarchical structuring of system functions and strict criteria to assess the quality of resulting design solutions.

Third, CASE methods are - at least principally - independent of computer type, programming language and sort of application. The corresponding strategies are recognizing the need for iterative system development, whereby they are separating the design phase from the analysis and specification of requirements. In addition, they introduce procedures for the verification of resulting documents with regard to completeness, correctness and consistency.

All the above-mentioned features of CASE methods and strategies can be properly extended to CASE (software) tools. They support the methods by means of efficient graphics, friendly human interface, their extensive data processing and storage capabilities, as well as options for easier documentation.

From the features mentioned above, one can quickly derive the basic benefits of CASE in the development of general computer-based systems or corresponding software. Right at the beginning of the project, the designers are supported in the development of concise functional specifications, which can be verified almost automatically. Then, at the design phase, they can derive a sound system design from the specifications, following formalized rules and guidelines. Any changes in basic system specifications or design improvements can be introduced in a controlled and ordely manner, whereby all corresponding diagrams and documents are much easily modified than with the "paper-and-pencil" method.

In summary, then, it can be concluded that the use of CASE results in a system and software which is of a good quality, has an updated and clear documentation, and is easy (i.e. cheap!) to maintain.

3. 'CASE' ALSO FOR CONTROL SYSTEMS?

The control systems, being a specific subset of real-time systems, must incorporate all the quality attributes associated with general computer-based systems, but, however, still some more. Similarly as with real-time systems, there are a few specific functional and performance requirements (Pressman, 1987) *which are not easy to satisfy, namely:*

- response time constraints,
- data transfer rate and throughput,
- interrupts and context switching,
- resource allocation and priorities,
- error handling and fault recovery,
- task sychronisation and
- inter-task communication.

As Pressman puts it for the case of real-time systems, "each of these performance attributes can be specified, but it is extremely difficult to verify if systems elements will achieve desired responses, if system resources will be sufficient to satisfy computational requirements, or if processing algorithms will execute with sufficent speed". All in all, "the design of real-time computing systems is the most challenging and complex task that can be undertaken by a software engineer. By its very nature, software for real-time systems makes demands on analysis, design, and testing techniques that are unknown in other application areas" (Pressman, 1987).

For the case of process control systems, we may admit that their performance requirements usually are not so severe, regarding only the system response time and data throughput rates (Hindin and Rauch- Hindin, 1983). Unlike most other real-time systems, they are complex in terms of the extent of communication with their environment (process operators, sensors and actuators) and because of some sophisticated (advanced) control algorithms.

Nowadays, the control algorithms for the processing industries are being designed and partly verified by simulation, using separate CACSD (Computer- Aided Control System Design) packages. In a sense, there is a specific approach in deriving the functional requirements of modern process control systems which include advanced control techniques, such as e.g. optimizing control. At the time being, there are no indications that CASE and CACSD can somehow be "married", but, in our opinion, this process must take place in the near future.

Nevertheless, side by side with CACSD, CASE seems to be the right "tool-box" also for process control systems, particularly because there are some methodological approaches addressing the above mentioned peculiarities of real-time systems. We will mention them shortly in the next paragraph.

4. A SHORT REVIEW OF 'CASE' METHODOLOGY

In a short article about the benefits of CASE for software engineering managers, Collard (1988) states that in the past few years, CASE has evolved from a concept to an industry. It is hard to believe and imagine such an explosive progress without knowing that the fundamental methodology behind CASE already has quite a history. With the inclusion of some important keywords and bibliographical references, a short past and future evolution of general-purpose CASE can be given in Fig. 1.

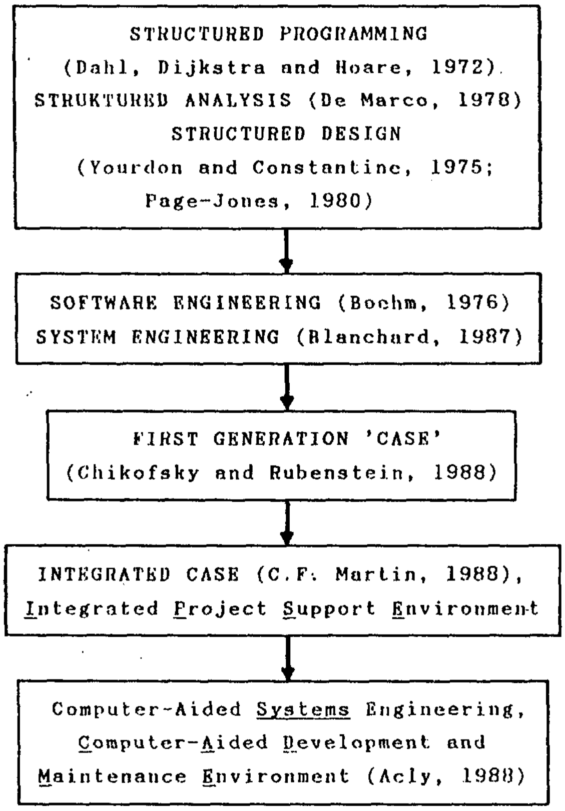

STRUCTURED PROGRAMMING
(Dahl, Dijkstra and Hoare, 1972)
STRUKTURED ANALYSIS (De Marco, 1978)
STRUCTURED DESIGN
(Yourdon and Constantine, 1975;
Page-Jones, 1980)
SOFTWARE ENGINEERING (Boehm, 1976)
SYSTEM ENGINEERING (Blanchard, 1987)
FIRST GENERATION 'CASE'
(Chikofsky and Rubenstein, 1988)
INTEGRATED CASE (C.F. Martin, 1988),
Integrated Project Support Environment
Computer-Aided Systems Engineering,
Computer-Aided Development and
Maintenance Environment (Acly, 1988)

Fig. 1. The evolution of CASE methodology

As the interested reader can obtain some good surveys of general CASE methods in this journal and other readily accessible literature (e.g. Gyorkos and coworkers, 1988; or Pressman, 1987), we can limit ourselves to mention real-time CASE methods and tools.

In general, there are two approach directions to real-time CASE, the obvious and the original. The obvious and the more frequent one is extending the general CASE methodology by existing elements, to cover the specific real-time system requirements, mentioned in chapter 3 of this paper. In contrast, the "original" direction is developing an entire new framework to deal with real-time problems.

In the following, we will briefly present three typical real-time CASE approaches (or strategies), covering the spectrum from original to the extended. Their authors are: a) Hassan Gomaa, b) Ward and Mellor, and c) Hatley and Pirbhai, respectively. All three have based their functional system representations upon the well-known "data-flow" diagrams and structured systems analysis/design (De Marco, 1978).

a) The Design Approach for Real- Time Systems (DARTS) . of Gomaa (1984, 1986) seems to be the most original one, as it specifically addresses the most important problems of real-time systems, i.e.:

- concurrency and task design,
- inter-task communication and synchronization, and
- state dependency and transaction processing.

In the DARTS, an entire life-cycle project phase is devoted to the design of tasks, i.e. structuring of software modules into concurrent processing units. Inter-task communication and synchronization is defined by means of special task interface modules, and a corresponding graphical notation is introduced (Fig. 2).

Because many real-time systems incorporate some degree of transaction processing, Gomaa has introduced an original solution to the problem of implementing a transaction which is dependent not only on the incoming data but also on the current state of the system (the so-called State Transition Manager module).

Another such useful representation, called the Event Sequence Diagram, shows the sequence of actions that are expected to take place when an external event occurs. There are still some interesting guidelines in DARTS for project organization, planning and management, such as the system architect and incremental development concepts, the former being taken from Brooks (1975).

b) In the Ward/Mellor approach, (Ward and Mellor, 1985; Ward, 1986), the authors propose the following, in addition to the previously known data-flow system modelling elements:

- extended notation to include control processes and flows,

- formation rules for a transformation schema to restrict ambiguous descriptions,
- execution rules for a transformation schema which are loosely based on the modified execution of a Petri net and visualized in terms of token placement,
- execution plans to manage the execution rules,
- extensions of the data-flow system model representing methods for dealing with certain problems of control transformations,
- separation of essential (i.e. functional) and implementation system models,
- hierarchies of transformation schemas to simplify the representation of complex systems.

Typical for this approach is that the control flows and processes appear together with data flows and processes on the same diagrams (Fig. 3). Each control process (transformation) must be associated with a state machine which, in turn, is being represented by a state-transition diagram or a corresponding state-transition matrix. In addition to ordinary data stores, used in the schema to indicate storage delays among transformations, buffers are introduced for exclusively storing information about discrete signals, implying destructive reading of its contents.

c) The Hatley/Pirbhai strategy (Hatley and Pirbhai, 1987) has some features in common with the Ward/Mellor approach, although - up to now - it has been elaborated more in detail primarily for system specification. The main features in common are:

- representation of control processes,
- representation of state transitions,
- part of formation rules,
- separation of requirements from implementation and
- hierarchical representation of complex systems.

In contrast to Ward and Mellor, Hatley and Pirbhai represent data flow (DFD) and control flow (CFD) separately. Moreover, they have devised detailed guidelines for how to separate the so-called "requirements model" from implementation, that is the "architecture model". The latter is being represented by a modified requirements model, augmented with implementation-technology dependent system features, such as: user interface processing, input/ output processing and maintenance, selftest and redundancy management processing (Fig. 4).

As depicted in Fig. 4, the requirements model consists of the process model and the control model. In addition, both are supported by a requirements dictionary. The process model is developed in a top-down fashion, beginning from the "context data-flow diagram", which is progressively broken down into a multi-leveled hierachy of more specific data-flow diagrams, with increasingly greater extent of details. The bottom level of the process model is simply described by short narrative, tabular or diagrammatic "process specifications" (PSPECS).

The control model looks similar to the process model, with the exception that it is completed by timing specifications and control specifications. The timing specification give the system timing contraints relative to its environment, whereas the control specifications define "trigger" signals to activate or deactivate particular processes in the process model. On the other hand, the primitive process specifications optionally define the so-called "data conditions", essentially control signals linking a data-flow diagram with a corresponding control-flow diagram. The structure of the requirements model is given in Fig. 5.

5. BENEFITS, PROBLEMS AND LIMITATIONS

The most complete qualitative representation of CASE benefits can be obtained from the features given in chapter 3. In addition to these, we could still mention better user involvement. As a representative of a major CASE tools developer and vendor (Arthur Andersen and Co.) says, "Automated design and prototyping tools play a powerful role in encouraging users to participate actively in ... (systems) design (R. O'Mahony, 1987).

A quantitative impression of CASE benefits can be derived from Fig. 6. Here data are depicted from a not so recent survey (1986) where the users by themselves have estimated the productivity improvement resulting from the use of Excelerator, the CASE tool of Index Technology Corporation (Chikofsky and Rubenstein, 1988). It is evident that, at that time, CASE had appeared to be the most valuable in the initial project phases. Other authors, e.g. Voelcker (1988), quote similar figures for productivity improvements.

But, unfortunately and in spite of proven benefits, it is reported that there are many problems associated with the introduction of CASE (see e.g. again Voelcker, 1988; or Shear, 1988). The main source of these problems is probably the novelty of CASE itself: "... there is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success, than to take the lead in the introduction of a new order of things ... " (a borrowed quotation from Pressman, 1987). No doubt that CASE is introducing "a new order of things" into the systems development process and, consequently, nobody likes to abandon his firmly established working habits, especially when this is associated with new learning effort and unclear future benefits. In the case of computer programming, the "moment of inertia" is still worse, as this profession is considered to be an intellectual art, impossible to fit in any ordered methodological framework.

Most often, this is the main problem behind the most common objections against CASE, just like the following few:

- if we do not begin with coding immediately, we will be late in delivery;
- the system user will not accept this way of doing;
- let the university people play with the "methods and approaches";
- we are working effectively without such guidelines;
- we will write system documentation later.

On the other side, there are some serious objections worth attention, because they are pointing to some general or specific limitation of (current-generation) CASE. Typical for this class are:

- productivity in system development is due mainly to good management.
- CASE is profitable only in "great" projects;
- how can I find a method suitable for my problem?

The first of these statements is absolutely true: CASE is no substitute for the skilled management of people, similarly as it cannot replace sound reasoning, although it supports both very well.

The second statement becomes true if the attribute "great" is defined in terms of system complexity. Indeed, CASE seems to be good for breaking down the requirements and the design of complex systems, such as typically are many modern computer-based process control systems.

The third of these objections, in the form of a question, can be resolved only by a good knowledge of available methods and tools.

In the context of our interest in CASE, we are perceiving two additional problems which, most probably, are still open. The first one has been already mentioned: there is a need to make a proper connection between the CACSD and CASE tools. The other is more specific: how to use or adapt CASE a) in our domestic social and working environment; and b) in typical research- intensive projects.

6. CONCLUSION

In our paper we tried to focus our reflections on the possible use of CASE in the development of process control systems. The statements given here represent a collection from the recently available foreign knowledge, mixed with some of our own opinions that were formed during a detailed study and numerous discussions of this topic.

This study has been started, and will be continued, in the framework of our efforts to find better ways of doing process control projects which will, hopefully, result in the advancement of our (control-science based) engineering profession and, second, in the conviction among our colleagues in industry, that it is worth-while to invest in domestic knowledge, instead of buying - and staying dependent of - foreign patents and licenses.

The preliminary results of our study indicate that CASE certainly is suitable for the development of computer-based process control systems, but, still some general and some specific problems concerning its introduction must be solved, before it can be used most effectively.

It would be to our sincere satisfaction if this knowledge can be of some value to any other professionals, dealing with the development in the demanding area of computer systems engineering.

<u>Acknowledgment</u>: The author is grateful to Dušan Osojnik, Rajko Kolar and Kordija Stiglic for valuable initial information; and further also to Marjan Rihar, Matjaž Šubelj and other colleagues for many stimulating discussions about CASE and software/systems engineering.

REFERENCES

Acly E., Looking beyond CASE, IEEE Software, March 1988, 39-43.

Blanchard B.S., Development of systems and equipment: Systems Engineering, Systems & Control Encyclopedia (Editor M.G. Singh), Pergamon Press, Oxford, 2, 995-1003.

Boehm B.W., Software Engineering, IEEE Trans. on Computers, C-25, 12(Dec. 1976), 1226-1241.

Brooks F.P., Jr., The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, Reading, MA, 1975, 1982.

Chikofsky E.J., Software technology people, IEEE Software, March 1988, 8-10.

Chikofsky E.J. and B.L. Rubenstein, CASE: Reliability engineering for information systems, IEEE Software, March 1988, 11-16.

Collard D., Selling CASE to your staff, Software Management, November 1988, 10.

Dahl O.-J., E.W. Dijkstra and C.A.R. Hoare, Structured Programming, Academic Press, London, 1972.

De Marco T., Structured Analysis and System Specification, Yourdon Press / Prentice-Hall, Englewood Cliffs, 1978.

Gomaa H., A software design method for real-time systems, Communications of the ACM, 27, 9 (Sept. 1984), 938-949.

Gomaa H., Software development of real-time systems, Communications of the ACM, 29, 7 (July 1986), 657-668.

Györkös J., I. Rozman and T. Welzer, A survey of most important and outstanding methods for software engineering, Informatica, 12, 2 (1988), 24-30.

Hatley D.J. and I.A. Pirbhai, Strategies for Real-Time System Specification, Dorset House Publishing, New York, 1987.

Hindin H.J., and W.B. Rauch-Hindin, Real-time systems, Electronic Design, January 6, 1983, 288-318.

Humphrey W.S., Characterizing the software process: a maturity framework, IEEE Software, March 1988, 73-79.

IJS (Institut "Jožef Stefan", Ljubljana), Prijedlog za poticanje projekta "Računalska automatizacija procesa u kemijskoj i procesnoj industriji, Ljubljana, april 1988; accepted for funding by the Yugoslav federal committee for science and technology, under the code PR-24.

Martin C.F., Second-generation CASE tools: a challenge to vendors, IEEE Software, March 1988, 46-49.

O'Mahony R., Successful systems: new ways to involve users, The Consultant Forum, 4, 2 (1987, Digital Equipment Corporation), 11-13.

Page-Jones M., The Practical Guide to Structured Systems Design, Yourdon Press, New York, 1980.

Pressman R.S., Software Engineering - A Practitioner's Approach, (2-nd Ed.) McGraw-Hill, New York, 1982, 1987.

Rihar M. and J. Černetič, Some practical aspects of using CASE tools, Informatica, 1989.

Shear D., CASE shows promise but confusion still exists, Electronic Design News, December 1988, 164-172.

Voelcker J., Automating SW: proceed with caution, IEEE Spectrum, July 1988, 25- 27.

Ward P.T. and S.J. Mellor, Structured Development for Real-Time Systems, Yourdon Press, New York, 1985.

Ward P.T., The transformation schema: An extension of the data flow diagram to represent control and timing, IEEE Trans. on Software Engineering, SE-12, 2, (Feb. 1986), 198-210.

Weiss E.A., The permanent software crisis - recommended classics for those in the ever- sticky software engineering tar pit, ABACUS, 3, 1, Fall 1985.

Yourdon E. and L.L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, Englewood Cliffs, 1975.

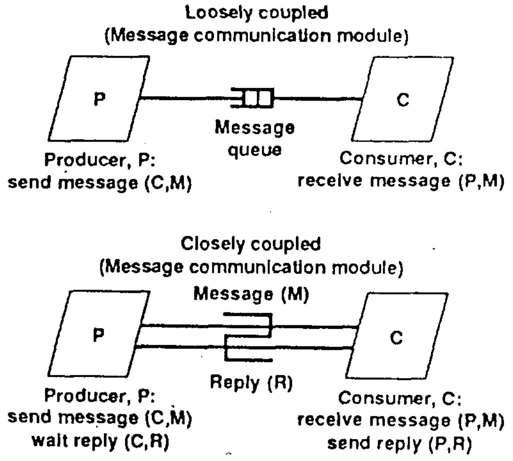


Fig. 2. a) DARTS notation for inter-task communication

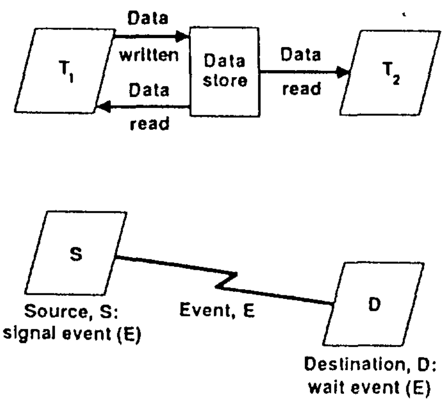


Fig.. 2. b) DARTS notation for information hiding and task synchronization.

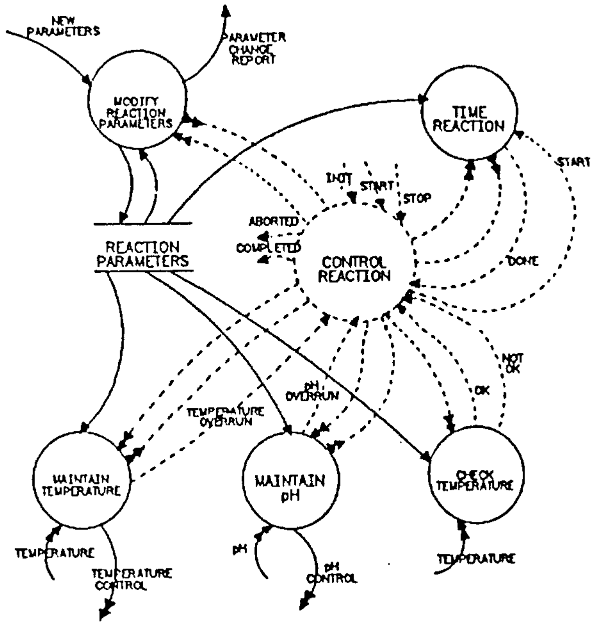


Fig. 3. Transformation schema, drawn in the Ward/Mellor notation, for a simple process control system.

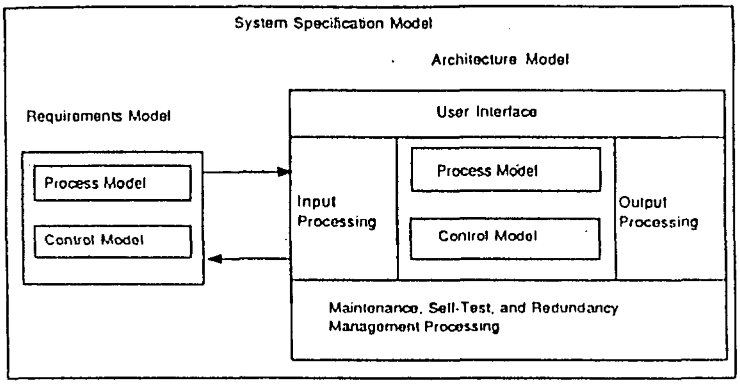


Fig. 4. Overall structure of the system specification model by Hatley and Pirbhai.

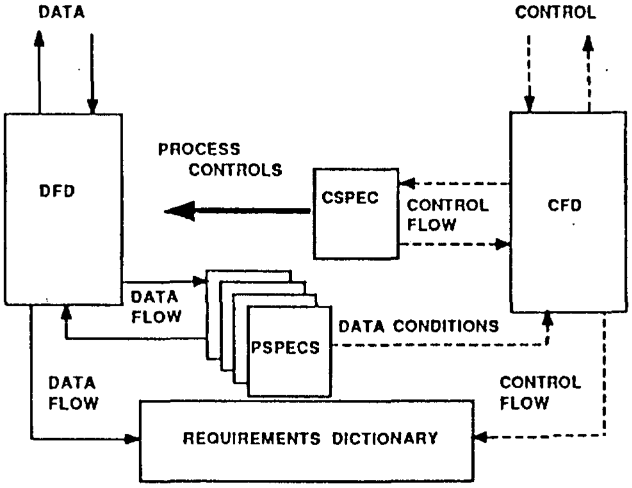


Fig. 5. The structure of the requirements model by Hatley and Pirbhai.

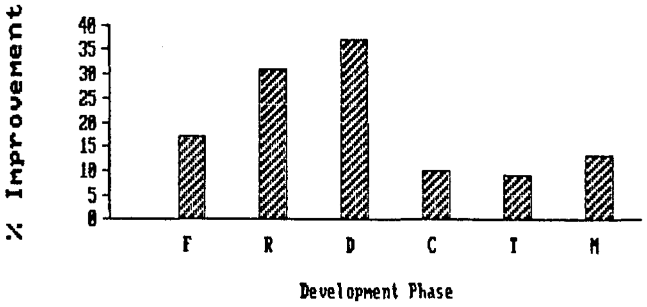


Fig. 6. Productivity improvement in particular system development phases (F - M) when using the CASE tool Excelerator (Data froma a user survey).

Legend: F = Feasibility study; R = Requirements specification; D = Analysis and design; C = Coding; T = Testing; M = Maintenance.

TRENDS OF COMPUTER PROGRESS

Keywords: trends, computer development, technology, programming

Matjaž Gams[1], Karmen Žitnik[1, 2]
[1] Institut »Jožef Stefan«, Ljubljana
[2] Iskra Unitel, Blejska Dobrava

ABSTRACT:
In the last decades computers are one of the leading factors influencing the progress in many areas of human activities from theoretical sciences to applied technologies. Analyses of computer history show several distinctive laws indicating that a) computer progress is very constant over last 50 years b) this progress is at least one order of magnitude faster than progress in other sciences and technologies c) there are no reasonable limits for this progress in near future and d) we can expect computers to greatly enhance their impact on everyday life by becoming dominant over human capabilities in several important fields such as massive memory.

POVZETEK:
V zadnjih nekaj desetletjih so računalniki med najpomembnejšimi vzpodbujevalniki razvoja človeških dejavnosti - od teoretičnih znanosti do uporabnih tehnologij. Analize računalniške zgodovine odkrijejo nekaj značilnih zakonitosti, med njimi a) razvoj računalnikov je zelo stanoviten v zadnjih 50. letih b) ta razvoj je vsaj za red velikosti hitrejši kot pri drugih znanostih in tehnologijah c) v bližnji bodočnosti ni videti pomembnejših omejitev razvoja računalnikov in d) lahko pričakujemo, da se bo vpliv računalnikov na vsakdanje življenje ljudi bistveno povečal s tem, ko računalniške sposobnosti prehitevajo človeške na nekaterih pomembnih področjih, npr. pri kapaciteti masovnih pomnilnikov.

1 Introduction

Several authors have tried to make unbiased analyses of computer history, their implications on everyday life and prognoses of computer progress (see all references). Some of these authors belonge to established industry, some are recognized scientists of different branches - from sociologists to computer researchers. But, most widely published and read opinions are mainly those belonging to popular or semi-scientific group. Publications of this kind often tend to exaggerate or even deform certain facts in order to attract a common reader. On the other hand, some of the laws presented in our article are well known in the computer community. Still, it might be interesting to throw some new light upon that especially because public opinion and certain governmental designers seem to be unaware of these mostly indisputable facts. And finally, our view might clarify the position of the fifth and the sixth computer generation.

2 Evolution of the basic computer component

A switch can be regarded as a basic computer component and by making evident it's progress we can understand the basis of computer progress.

In Table 1 we see that roughly each 10 years a switch was produced in a different technology. In the first 10 years electronic computers were based on valves, in the next 10 years on transistors and somewhere around 1964 chips emerged as a new product integrating transistors into one integrated circuit. The following 10-year periods were based on new technologies that achieve higher and higher density of transistors: LSI - Large Scale Integration, VLSI - Very Large Scale Integration and finally ULSI - Ultra Large Scale Integration.

10-YEAR PERIOD	SWITCH
1	VALVE
2	TRANSISTOR
3	CHIP
4	LSI
5	VLSI

Table 1: The progress of the basic component of computers - a switch - can be roughly grouped into five 10-year periods.

In each 10-year period the switch was produced in a new technology, it became much smaller, faster, more reliable and with longer exploration period. The improvement of overall efficiency is especially noticeable when compared to the price of one switch.

The technological progress of the basic computer component largely influenced the progress of computer hardware, architecture, peripherals and also software.

3 Software evolution

Software progress can again be schematically represented by the development of the basic meaningful part of programming languages (see Table 2).

10-YEAR	PROG. UNIT	TYPE OF LANGUAGE
1	010110100	MACHINE LANGUAGE
2	LOAD I	ASSEMBLY LANGUAGE
3	DO I=1,10	PROCEDURAL HIGH-LEVEL LANGUAGE (FORTRAN)
4	procedure invert(var A:aT);	MODULAR HIGH-LEVEL LANGUAGE (PASCAL)
5	succ(X,Y) :- parent(X,Z), succ(Z,Y).	DECLARATIVE LANGUAGE (PROLOG)

Table 2: Evolution of software through the progress of the basic unit of programming languages again enables structuring the history into 5 roughly equidistant 10-year periods.

First programmers had to code in sequences of 0's and 1's in machine code. This awful task was substituted by programming in simple instructions that were still strongly hardware oriented - for example, LOAD I means that the value of I is loaded into a specialized register - accumulator. However, these assembly languages represented a large step ahead in the direction of simplified human-computer communication. The next generation of procedural high level programming languages enabled programming in terms of hardware independent statements. For example, the presented loop in Fortran in Table 2 means that the following statements are to be executed 10 times. These statements remained only slightly changed also in newer languages like Pascal, Modula and Ada, but they enabled new concepts - a subroutine, a process and a module together with structural programming. And in the last 10 years there are two additional lines of software progress: a) declarative Prolog-like languages which enable programming by declaring logical theorems whereby the computer does the proving task and b) specialized tools such as advanced relational data base management systems, program environments with editors, debbugers, program verifiers and tools for automatic code generation from specifications.

Each new software generation was more adapted to a human way of communicating meaning more and more work for translation into machine code was left to computers. But, while the price of computer work was very rapidly declining, the price of human work remained more or less constant. Therefore, the technological progress of computer capabilities directly dictates more and more human-like communication.

4 The 5th and the 6th generation

In the light of our presentation of computer history one might wonder where to put the 5th and the 6th computer generation. The heavily published project of the 5th generation started in Japan and attracted attention of Western scientific and industry community. The high goals of the project which started in the land of economic and technological miracle triggered several advanced projects in USA and Western Europe. But unlike the 5th generation project in Japan these projects used very different terms which are more consistent with our view. The basic confusion comes from the fact that our denotation of the fifth generation represents computers in the late eighties which don't have the intended properties of the Japanese 5th generation. Further more, in order to come to the fifth generation should we wait for the computers to become similar to those defined by the Japanese 5th generation programme? The problem is similar with the 6th generation: from the trends of computer progress it is possible to prognosticate the properties of the future generations but only the history is the unbiased judge which records where and what happened.

5 Practical implications of computer progress

Practical implications of computer progress are influenced by technological development and as well as several other factors, with market being one of the most important.

Market break-throughs happen whenever a new and better type of computer product reaches technological maturity. Since this is advanced product with preferable cost/benefit ratio, it causes great commercial attraction. All products of this type are bought by customers and the supply/demand ratio is preferable for any producer capable of mastering new technology. The production grows very rapidly and gradually overfills the market - the supply becomes greater than the demand. A hard competition ruins many producers but on the horizon there is another technological break-through which rolls the circle once again (see Figure 1).

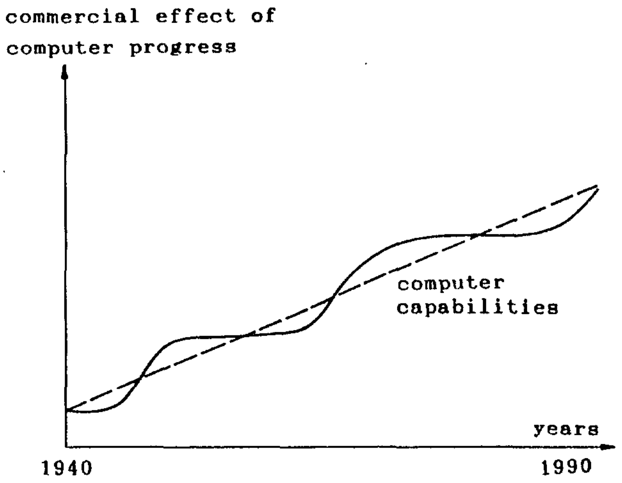


Figure 1: Commercial break-throughs in the computer market happen in supply/demand circles following technological advances.

The widely known example is a sequence of break-throughs of microcomputers: home computer (Spectrum, Commodore) with a boom in computer games and children education, personal computer (Apple, IBM PC) with a boom in small business and smaller tasks and workstation (PS2, SUN) with more complex applications. Workstations are in the stage of technological progress and we are just witnessing the technologically underdeveloped state of our country measured by the number of workstations which have already ousted personal computers in many areas in the developed countries. An interesting product of this kind could also be Steven Jobs's NeXT. It qualifies as a state of the art in the areas of vision, sound, memory, software and cost/benefit ratio. The high-resolution black&white monitor shows exactly what will come out of the laser printer. It records and plays back music and voices with the fidelity of compact disc. The removable, erasable optical disk drive can hold enough information to fill hundreds of books. In a certain way this is a rather successful competition with the products of the Japanese 5th generation programme.

We can roughly estimate the importance of computers today. To replace the work of all computers today we would need something like 1000 times more people than the whole human population. Despite the fact that not all of these work would have to be replaced by "brute" human work we can still claim that we would need more people for the replacement of all computers than for the replacement of all other machines! This tells us how vast is the amount of computer work.

In certain fields of everyday work like commerce and technology more than every second worker in USA has an equivalent of an IBM PC. And in respect to all USA population this relation is one IBM PC equivalent per 8 men. On the other hand, in the Soviet Union only every seventieth man has a PC equivalent.

6 Trends of computer progress

The development of several computer characteristics is very steady over the last 50 years, among them the density of components on one square millimeter. Each 10 years the density of components increases 100 times (Figure 2).

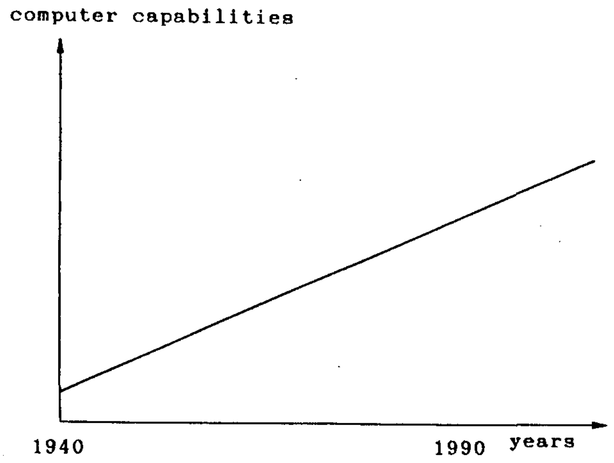


Figure 2: The development of many basic computer characteristics is constant over the last 50 years. The speed of progress is at least one order of magnitude greater than in other technologies.

Greater density enables other improvements, e.g. shorter connections, faster computing and as the overall result computers become approximately million times more capable each 10 years. This progress is beyond any comparison with other technologies which are usually much more binded by the natural limits. For example, in only a couple of years with such a tempo of progress we should get a super family car capable of reaching the speed of an airplane with a consumption of only a few drops of gasoline per 1 km. This is definitelly illusional, because natural limits for cars are obvious and unavoidable, for example air friction.

Considering the present state of computer technology we can see possible limits in chip capacity, computing speed, overheating of components, etc. But it was the same in the past and each computer was actually produced within the limits of existing computer technology. There were no reasonable unavoidable limitations for the progress of computers in the past and there aren't any in the near future. In fact, it has always turned out that the cause of the limit was our knowledge or rather the lack of it!

There are several potential break-throughs in the near future such as erasable optical disks, communication in natural language and through graphical interfaces; further away there could be superconductive fully intelligent computers. And even when we reach the limits of these technologies, there are plenty of others. For example, when we reach the limits in computing or chips posed by the electron (e.g. the tunneling effect), we might try with photons.

7 Massive memory - a possible break-through

First electronic computers were useful because they outperformed humans in the speed and precision of numerical computing thus becoming valuable assistants. Similarly, important break-throughs happen when computers outperform people in other areas and so far there seems to be no end to this. The only question is when and in which area it will happen.

One of the important areas where capabilities of computers are quickly approaching human performances is massive memory (see Figure 3).

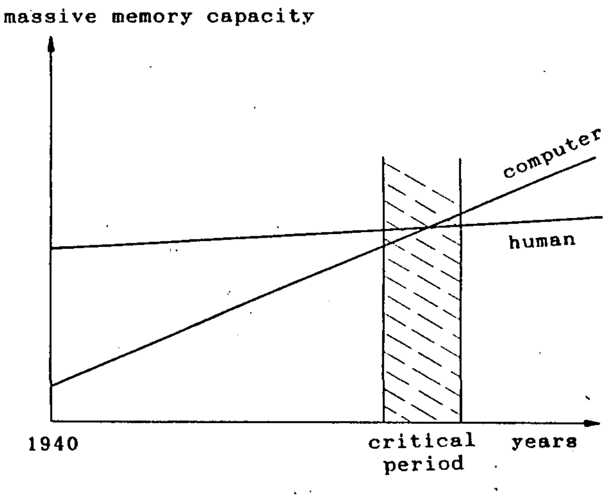


Figure 3: The progress in the capacity of computer massive memory is constant and promises to reach the effective massive capacity of human brain in the near future.

The progress presented in Figure 3 is rather constant. Typical examples in Table 3 show the possibility of another break-through.

TYPE	CAPACITY	LENGTH OF BOOKS	TIME FOR TYPING
first home c.	20K	3 mm	0.002 years
IBM PC	20M	3 m	2 years
next pers. c.	20G	3 km	2000 years

Table 3: Jumps in the capacity of massive memory from first home microcomputers to IBM PC and next personal microcomputers. Memory capacity of the computer is presented in the second column, in the third column we see the corresponding length of books on a book shell and in the last column years needed for typing that amount of information.

The first home microcomputers had around 20K of memory. This amount of information could be typed by a typist in 0.002 years on a 3 mm wide book. IBM PC has a disk with 20M which corresponds to 3 m of books and 2 years of typing. One of the next generation personal computers will reach 20G corresponding to 3 km of books and 2000 years of typing. This certainly represents a qualitative leap since one person needs for his work about 100-200M.

To understand the importance of a qualitative leap ahead let us make a comparision with a car. If the first prototype of a car reached only 8 m/h and the second 8 km/h, we would achieve a leap of 1000. Although this would be a large jump ahead, these cars would remain economically uninteresting. But one jump more by the factor of 1000 would radically change the commercial outcome: we'd have a car with the maximal speed of 8000km/h and a clear market boom.

A qualitative leap ahead in massive computer memory will obviously enable us to store lots of books and lots of encyclopedias on one disk. Communications through picture and sound will become feasible and maybe even the rudiments of the human recognizable artificial intelligence. But how will it influence everyday life, e.g. the learning process and the access to (computer) libraries? Therefore, while we are able to see that something important is going to happen, we are not able to see when and especially exactly how it will happen.

8 Areas of computer applications

Areas of computer applications are presented in Figure 4. The circles represent the growing use of computers by mastering more tasks.

In the last years we are already witnessing the shift from classical programming to new tools such as relational data base management systems and knowledge engineering techniques. Today's tasks are typically ill-structured and systems require knowledge from human experts.

For the next qualitative shift we need to overcome the parallel processing barrier and the artificial intelligence barrier. Then the systems will be able to intelligently handle massive amounts of knowledge, they will be able to learn from experience and modify their knowledge during the running. Somewhere around that time the computers might become aware of themselves.

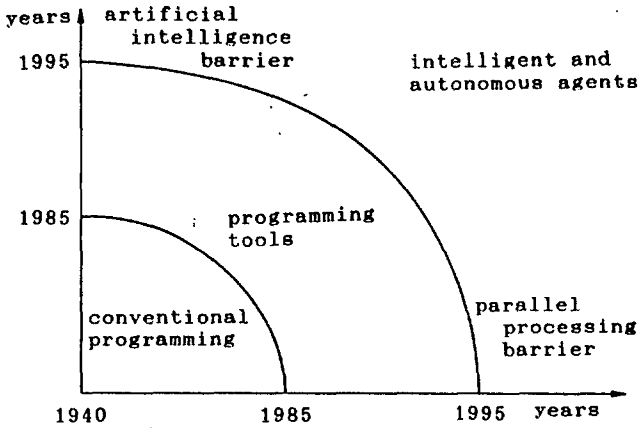

years
artificial intelligence barrier
1995
intelligent and autonomous agents
1985
programming tools
conventional programming
parallel processing barrier
1940
1985
1995
years

Figure 4: In these years conventional programming is being superseded by programming tools with the emphasis on knowledge engineering. The next stage is programming by intelligent and autonomous agents.

9 Summary

Computer progress is practically constant over the last 50 years. The speed of this progress is beyond comparision with other sciences and technologies alike and is still very far from the limits that could threaten to slow it down. Even today a computer is the most important single machine invented by human and it will tend to largely enhance its influence on everyday life by becoming dominant over people in more and more categories. The designers of our research and industry directions should pay more attention to that!

Literature

Dreyfus H. L. (1979): What Computers Can't Do, Harper Colophon Books.

Dreyfus H. L., Dreyfus S. E. (1986): Mind Over Machine: The Power of Human Intuition and Expertise in the Era of the Computer, The Free Press, Macmillan, New York.

Laurie P. (1984): The Joy of Computers, Cankarjeva založba, Ljubljana.

McDermott D. (1985): Artificial Intelligence Meets Natural Stupidity, Mind Design: Philosophy, Psychology, Artificial Intelligence, The MIT Press, Cambridge.

Michie D., Johnston R. (1984): The Creative Computer, Viking.

Newsweek, Time, Byte.

Winograd T., Flores F. (1986): Understanding Computers and Cognition: A New Foundation for Design, Ablex Publ. Corp, Norwood.

Železnikar A. P. (1987): Research of Computers and Information in the Next Decade, Informatica 11, No. 2, pp. 57-59.

Železnikar A. P. (1988): On the Developmental Iniciative of the Computer Industry, Informatica 12, No. 2, pp. 3-11.

DETECTION OF THE INTERSECTION OF TWO SIMPLE POLYHEDRA

Dušan Šurla, Zoran Budimac
Institute of Mathematics,
dr I. Djuričića 4, 21000 Novi Sad

Keywords: detection algorithm, polyhedron intersection, modula-2

ABSTRACT. An algorithm is described for detecting the intersection of two simple polyhedra. The corresponding programme, implemented in Modula-2, is essentially based on a procedure developed to test the intersection of the given segment and simple polygon. The basis for this procedure is the relations between a point, a straight line and a plane, expressed in the vector form.

1. INTRODUCTION

One of the fundamental problems in computational geometry is detection of the intersection of two polyhedra. The problem is directly related to linear programming, hidden surface elimination, computer vision, motion planning and robotics.

Of the numerous publications devoted to this subject we shall mention only those dealing with the problem of intersection [5,6,9] and detection of the intersection [1-4] of two polyhedra. Some of the authors have considered the computational complexity of the algorithms used for solving these problems [3-6,9,10].

In [7] and [8] we have described an algorithm and the corresponding programme for determination whether a given point belongs to the interior domain of a simple polyhedron, as well as for determination of the intersection of a straight line and a simple polyhedron. The basic procedures were formed on the basis of the relations (given in the vector form), between a point, a straight line and a plane.

The present article is a continuation of the above studies in which our considerations are being extended onto the problem of detection of the intersection of two simple polyhedra.

2. THE ALGORITHM

Let be given two simple polyhedra P and Q. Their possible relations may be as follows:

$$P \cap Q = C, \quad C \neq \emptyset, C \neq P, C \neq Q \qquad (1)$$
$$P \cap Q = P, \quad (P \subseteq Q) \qquad (2)$$
$$P \cap Q = Q, \quad (Q \subseteq P) \qquad (3)$$
$$P \cap Q = \emptyset \qquad (4)$$

If the intersection of at least one edge of P (resp. Q) and at least one facet of Q (resp. P) is not an empty set, then condition (1) is fulfilled. If condition (1) is not fulfilled, then P and Q intersect provided that one arbitrary vertex of P (resp. Q) belongs to the interior of the polyhedron Q (resp. P), i.e., condition 2 (resp. 3) is fulfilled. Obviously, if conditions (1)-(3) are not fulfilled, then case (4) holds, i.e., P and Q do not intersect.

Thus, testing condition (1) is reduced to the multiple use of the function for detecting the intersection of the segment (polyhedron edge) and the simple polygon (polyhedron facet). This function can be formed on the basis of the relations given in their vector form.

2.1. BASIC RELATIONS

An arbitrary point $Z \in \mathbb{R}^3$, considered as a vector of the same coordinates, we shall denote by $\vec{Z}$.

Let be given the points $A \in \mathbb{R}^3$ and $B \in \mathbb{R}^3$, and the plane α in $\mathbb{R}^3$. Let us form the following expression:

$$D_1 = (\vec{A} - \vec{X}) \circ \vec{N} \qquad (5)$$
$$D_2 = (\vec{B} - \vec{X}) \circ \vec{N} \qquad (6)$$
$$D = D_1 \cdot D_2 \qquad (7)$$

where $\vec{N}$ is a vector perpendicular to α and $X \in \alpha$. The mark "$\circ$" denotes the scalar product of vectors. If $D < 0$ (resp. $D>0$), then the points A and B are on different (resp. on the same) side of the plane α. For $D_1 = 0$ and $D_2 \neq 0$ point A belongs to the plane α, and for $D_1 \neq 0$ and $D_2 = 0$ point B belongs to the plane α. If $D_1 = 0$ and $D_2 = 0$ then the segment AB belongs to the plane α.

Let us form the expressions:

$$\vec{E}_1 = (\vec{G} - \vec{U}) \times (\vec{V} - \vec{G}) \qquad (8)$$
$$\vec{E}_2 = (\vec{H} - \vec{U}) \times (\vec{V} - \vec{H}) \qquad (9)$$
$$E = \vec{E}_1 \circ \vec{E}_2 \qquad (10)$$

where the points G, H, U and V belong to the same plane. The mark "x" stands for the vector product of vectors. If $E > 0$ (resp. $E < 0$), then the points G and H are on the same (resp. on different) side of the straight line determined by the points U and V. For $E_1 = 0$ (resp. $E_2 = 0$) point G (resp. H) lies on the straight line determined by the points U and V.

On the basis of relations (8)-(10) it can be determined whether the segments GH and UV intersect. Namely, if the points G and H are on the different sides of the straight line UV and the points U and V are on the different sides of the straight line GH, the two segments intersect, otherwise not.

2.2. DETECTION OF THE INTERSECTION OF A SEGMENT AND A SIMPLE POLYGON

Let us denote the vertices of an edge of the one polyhedron by A and B, and by S a facet of the other polyhedron. Facet S is a simple polygon. Let the plane α be determined by the polygon S. Let us suppose the values in expressions (5)-(7) are as follows: $D<0$; $D_1=0$ and $D_2\neq0$; $D_1\neq0$ and $D_2=0$. In these cases the intersection of the segment AB and the plane α is a point. Let us denote this point by R. If R belongs to the interior region or of the hull of the polygon S, then the intersection of the segment and polygon S is not an empty set. In the case when the segment AB belongs to the plane α, then detection of the intersection of the segment AB and polygon S consists in the following. The intersection of the segment AB and all the edges of S is tested on the basis of relations (8)-(10). If this intersection is an empty set, then it is necessary to test additionally if at least one of the points A and B belongs to the interior region of S. If it does, the intersection of the segment AB and polygon S is not an empty set.

Therefore, detection of the intersection of the segment AB and polygon S is reduced further to solving the following task.

Given a simple polygon S in α and the point $R\in\alpha$, determine if the point R belongs to the interior region of S.

Let r be an arbitrary straight line lying in the plane α and passing through the point R. Let us introduce the following definitions.

Definition 1. The intersection point between r and the hull of P is a piercing point if at this point r passes from the interior into the exterior domain of P, or vice versa.

Definition 2. The edge of the simple polygon S, lying on the straight line r is a piercing edge if one vertex of this edge borders upon the internal and the other on the external region of S.

Then, the following theorem holds.

Theorem. Point R belongs to the interior region of S if on the same side of the point R lying on the straight line r, the sum of piercing points and piercing edges is an odd number.

Proof. Let us suppose the point Y moves along the straight line r from infinity to the point R. Then Y belongs to the exterior domain of S until it reaches the first piercing point, or piercing edge. After passing through the first piercing point / piercing edge, the point Y enters the interior domain of S and remains in it until reaching the second piercing point / piercing edge. Afterwards, the point Y comes again to the exterior domain, and so on. Therefore, if point Y coincides with R after passing through an odd number of the sum of piercing points and piercing edges, then R belongs to the interior domain of polygon S.■

Let us denote an edge of S by $V_{i-1}V_i$. The straight line r and the edge $V_{i-1}V_i$ may have one of the following relations:

(i) $r \cap V_{i-1}V_i = T$;
T is different from V_{i-1} and V_i,

(ii) $r \cap V_{i-1}V_i = V_i$ or $r \cap V_{i-1}V_i = V_{i-1}$,

(iii) $r \cap V_{i-1}V_i = V_{i-1}V_i$,

(iv) $r \cap V_{i-1}V_i = \emptyset$

In case (i) T is a piercing point. Let in case (ii) the intersection be the vertex V_i. Then V_i is a piercing point if the vertices V_{i-1} and V_{i+1} are on different sides of the straight line determined by the points V_i and R, i.e. if the following condition is fulfilled:

$$((\vec{V}_{i-1}-\vec{V}_i)x(\vec{R}-\vec{V}_{i-1})) \circ ((\vec{V}_{i+1}-\vec{V}_i)x(\vec{R}-\vec{V}_{i+1})) < 0 \tag{11}$$

In case (iii), the edge $V_{i-1}V_i$ is a piercing edge if the vertices V_{i-2} and V_{i+1} are on different sides of the straight line r, i.e. if the following condition is fulfilled:

$$((\vec{V}_{i-2}-\vec{V}_{i-1})x(\vec{V}_i-\vec{V}_{i-1}))\circ((\vec{V}_{i+1}-\vec{V}_{i-1})x(\vec{V}_i-\vec{V}_{i+1}))<0 \tag{12}$$

Obviously, in case (iv), the edge $V_{i-1}V_i$ is not a piercing edge and there are no piercing points on it.

Figure 1 shows an illustrative example of the intersection of the straight line r and polygon S. The corresponding intersection points are K_1, K_2, K_3, K_4 and V_{16}, and the piercing edge is $V_{10}V_{11}$. Additionally, the edges V_2V_3 and V_6V_7 and the vertex V_{13} are lying on the straight line r. On the basis of these data, it is possible to determine if a point $R\in r$ is on the hull of S, or it belongs to the interior / exterior region of S. First, if the point R coincides with one of the piercing points K_1, K_2, K_3, K_4 and V_{16}, or with the vertex V_{13}, or it belongs to one of the edges V_2V_3, V_6V_7 and $V_{10}V_{11}$, then R is on the hull of S. Second, let us suppose that the point R is between K_3 and V_{10}. Then, on the one side of this point are found the piercing points K_1, K_2 i K_3, and on the other side, the piercing edge $V_{10}V_{11}$ and the piercing points K_4 and V_{16}. Obviously, on the basis of the given theorem, in both cases point R belongs to the interior region of S.

Let us consider now the segment RR_∞, where the point R_∞ is chosen to belong to the exterior region of S, which is easily achieved by taking that absolute values of the coordinates of the point R_∞ are large. The algorithm for determining if R belongs to the interior region of S, can be formed as a Modula-2 function procedure Internal in the following way:

```
PROCEDURE Internal(R, S): BOOLEAN;
[ Procedure Internal returns TRUE if point R
belongs to the interior region or to the hull of
the simple polygon S, whose vertices are denoted
by V_i, i=1, 2, ..., n, where it is assumed that
V_n=V_0, V_n+1=V_1, V_n+2=V_2.
K is the sum of piercing points and piercing
edges. ]
```

```
BEGIN
  K := 0;
  Determination of point $R_\infty$;
  FOR i := 1 TO n DO
    IF ($R \in V_i V_{i+1}$) THEN
      RETURN TRUE
    ELSIF ($V_i V_{i+1} \subset RR_\infty$) AND
    ($V_{i-1}, V_{i+2}$ are on different sides of the
     straight line $RR_\infty$) THEN
      INC(K)
    ELSIF ($V_i \in RR_\infty$) AND
    ($V_{i-1}, V_{i+1}$ are on different sides of the
     straight line $RR_\infty$) THEN
      INC(K)
    ELSIF ($V_i V_{i+1} \cap RR_\infty \neq \emptyset$) THEN
      INC(K)
    END
  END
  RETURN K<>0 AND ODD(K)
END Internal;
```

In the given algorithm, the relations between two segments, and between a point and a segment are determined on the basis of relations (8)-(10).

The algorithm for determining if an edge and a facet intersect is the auxiliary one, and will be used in the final step. It is formed as the Modula-2 function procedure Intersect.

```
PROCEDURE Intersect(E, F): BOOLEAN;
{ Procedure Intersect returns TRUE if the edge E
and the facet F intersect, otherwise it returns
FALSE }

BEGIN
  IF (E $\cap$ plane(F) $\neq \emptyset$) THEN
    IF (E $\subset$ plane(F)) THEN
      IF (E $\cap$ hull(F) $\neq \emptyset$) THEN
        RETURN TRUE
      ELSE
        R is one of the vertices of E
        IF Internal(R, F) THEN
          RETURN TRUE
        END
      END
    ELSE
      R := E $\cap$ plane(F)
      IF Internal(R, F) THEN
        RETURN TRUE
      END
    END
  END;
  RETURN FALSE
END Intersect;
```

2.3. PROCEDURE FOR DETECTING THE INTERSECTION OF TWO SIMPLE POLYHEDRA

Let us denote by EP_i, i = 1,2,...,|EP| and FP_i i = 1, 2, ..., |FP| the edges and facets of the simple polyhedron P, and by EQ_i, i = 1,2,...,|EQ| and FQ_i i = 1, 2, ..., |FQ| the corresponding edges and facets of the polyhedron Q. Then, the algorithm for detecting the intersection of P and Q may be presented in the form of Modula-2 function procedure PolyhedraIntersection.

```
PROCEDURE PolyhedraIntersection(P, Q): BOOLEAN;
{ Procedure PolyhedraIntersection returns TRUE
if Polyhedra P and Q intersect, otherwise
returns FALSE }

BEGIN
  FOR i := 1 TO |EP| DO
    FOR j := 1 TO |FQ| DO
      IF Intersect($EP_i$, $FQ_j$) THEN
        RETURN TRUE
      END
    END
  END
  FOR i := 1 TO |EQ| DO
    FOR j := 1 TO |FP| DO
      IF Intersect($EQ_i$, $FP_j$) THEN
        RETURN TRUE
      END
    END
  END
  IF (any vertex(P) $\in$ Q) OR { $P \subseteq Q$, cond. (2) }
     (any vertex(Q) $\in$ P)    { $Q \subseteq P$, cond. (3) }
  THEN
    RETURN TRUE
  ELSE
    RETURN FALSE
  END;
END PolyhedraIntersection;
```

The modules for testing whether any vertex of one polyhedron belongs to the interior domain of the other polyhedron has been given in [8].

3. TEST EXAMPLE

Data structure of the simple polyhedra P, Q and R is given in Tables 1.-3.

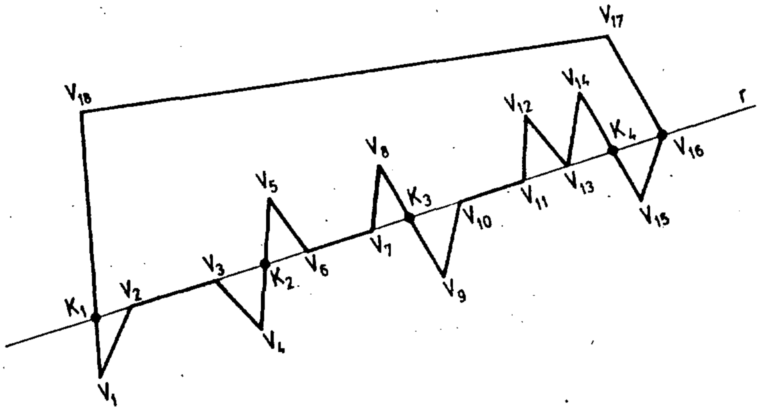


Figure 1

Table 1

Ordinal no. of vertex	Polyhedra P	Q	R
1	(0,0,0)	(1,1,1)	(0,0,6)
2	(5,0,0)	(6,1,1)	(5,0,6)
3	(3,2,0)	(4,3,1)	(3,2,6)
4	(4,4,0)	(5,5,1)	(4,4,6)
5	(2,2,5)	(3,3,6)	(2,2,11)

Table 2

Ordinal no. of edge	Edge determined by vertices
1	1, 2
2	2, 3
3	3, 4
4	3, 5
5	4, 5
6	4, 1
7	1, 5

Table 3

Ordinal no. of facet	Facet determined by ordered vertices
1	1, 2, 5
2	2, 3, 5
3	3, 5, 4
4	4, 1, 5
5	1, 2, 3, 4

$P \cap Q \neq \emptyset$ and $P \cap R = \emptyset$.

4. CONCLUSION

On the basis of the relations derived in vector form, a function can be easily formed for testing of the intersection of a given segment and a simple polygon. The multiple use of this function can serve for detecting the intersection of two simple polyhedra P and Q for the cases when $P \cap Q = C$, $C \neq \emptyset$, $C \neq P$ and $C \neq Q$. The introduced vector relations may be suited for solving other problems in computational geometry.

REFERENCES:

1. J. W. Boyse, Interference Detection Among Solids and Surfaces, *Communications of the ACM* 22(1), 3-9(1979).

2. G. Davis, Computing Separating Planes for a Pair of Disjoint Polytopes, *Proc. ACM Symposium on Computational Geometry*, 8-14(1985).

3. D. P. Dobkin, D. G. Kirkpatrick, A Linear Algorithm for Determining the Separation of Convex Polyhedra, *Journal of algorithms* 6, 381-392(1985).

4. D. P. Dobkin, D. G. Kirkpatrick, Fast Detection of Polyhedral Intersection, *Theoretical Computer Science* 27, 241-253(1983).

5. S. Hertel, M. Mäntylä, K. Mehlhorn, J. Nievergelt, Space Sweep Solves Intersection of Convex Polyhedra, *Acta Informatica* 21, 501-519(1984).

6. K. Mehlhorn, K. Simon, Intersecting Two Polyhedra One of which is Convex, *FCT*, 535-542(1985).

7. D. Surla, The Relation Between a Point and a Simple Polyhedron, *Informatica* 12(2), 65-68(1988).

8. D. Surla, Lj. Jerinic, An Algorithm for Determining the Relation Between a Straight Line (Point) and a Simple Polyhedron, *The Third International Conference on Computer Graphics*, Dubrovnik, Yugoslavia, 1988, in press.

9. M. Szilvási-Nagy, An Algorithm for Determining the Intersection of Two Simple Polyhedra, *Computer Graphics Forum* 3, 219-225(1984).

10. A. C. Yao, R. L. Rivest, On the Polyhedral Decision Problem, *SIAM J. Computing* 9(2), 343-347(1980).

APPENDIX

IMPLEMENTATION OF THE ALGORITHM

A.1. PRELIMINARIES

For representing a simple polyhedron, the following data structure has been adopted:

```
Point      = ARRAY [1..3] OF REAL;
Edge       = RECORD
               F, S: Point
             END;
Polygon    = RECORD
               V  : ARRAY [1..100] OF Point;
               No : [1..100]
             END;
Aux        = ARRAY [1..20] OF INTEGER;
Polyhedron = RECORD
               NoP      : CARDINAL;
               Vertices: ARRAY [1..100] OF Point;
               NoE      : CARDINAL;
               Edges    : ARRAY [1..50] OF Aux;
               NoF      : CARDINAL;
               Facets   : ARRAY [1..50] OF Aux;
               NoOfV    : ARRAY [1..50] OF INTEGER;
             END;
```

A vertex is represented by the array *Point* of three real numbers, i.e. coordinates of the point. An edge is represented by the record *Edge* of two points, and a polygon is represented by the record *Polygon* i.e. by the array *V* of *No* points.

A polyhedron is represented by the record *Polyhedron* i.e. by its vertices (array *Vertices* of *NoP* points), edges (array *Edges* of *NoE* vertex indices - pointers to array *Vertices*) and facets (array *Facet* of *NoF* vertex indices - pointers to array *Vertices*). The i-th element of the array *NoOfV* contain the information on the number of vertices of the i-th polyhedron facet.

There are two operations on data structures representing polyhedron. The first one (implemented as the function procedure *Edg(P: Polyhedron; i: CARDINAL): Edge* selects the i-th edge of the polyhedron P. The other one (implemented as the function procedure *Fac(P: Polyhedron; i: CARDINAL): Polygon* selects the i-th facet of the polyhedron P. Both procedures return their values in appropriate data structures, i.e. *Edge* and *Polygon*, respectively.

```
PROCEDURE Edg(P: Polyhedron; i: CARDINAL): Edge;
VAR E: Edge;
BEGIN
  WITH P DO
    E.F := Vertices[Edges[i,1]];
    E.S := Vertices[Edges[i,2]];
  END;
  RETURN E
END Edg;
PROCEDURE Fac(P: Polyhedron; i: CARDINAL):
                                        Polygon;
VAR S: Polygon;
    j: CARDINAL;
BEGIN
  WITH P DO
    S.No := NoOfV[i];
    FOR j := 1 TO S.No DO
      S.V[j] := Vertices[Facets[i,j]]
    END;
  END;
  RETURN S
END Fac;
```

We will cite without a source code some procedures for basic vector operations, which we need for implementation of the algorithm:

```
PROCEDURE ScalarMul(V1,V2:Point): REAL;
                              [ ScalarMul=V⃗1∘V⃗2 ]
PROCEDURE VecEqual(V1,V2:Point): BOOLEAN;
                              [ VecEqual=(V⃗1=V⃗2) ]
PROCEDURE VecAdd(V1,V2:Point): Point;
                              [ VecAdd=V⃗1+V⃗2 ]
PROCEDURE VecScMul(A:REAL; V1:Point): Point;
                              [ VecScMul=A*V⃗1 ]
PROCEDURE VecSub(V1,V2: Point): Point;
                              [ VecSub=V⃗1-V⃗2 ]
PROCEDURE VecMul(V1, V2: Point): Point;
                              [ VecMul=V⃗1xV⃗2 ]
```

A.2. PROCEDURE Internal

The procedure determines if the given point belongs to the interior domain of the simple polygon. It uses additional procedures *OppSides* and *Between* , which are based on relations (8)-(10).

If points A and B are on different sides of the straight line determined by C and D, then function procedure *OppSides* returns TRUE, otherwise it returns FALSE.

```
PROCEDURE OppSides(A, B, C, D: Point): BOOLEAN;
VAR E1, E2: Point;
BEGIN
  E1 := VecMul(VecSub(A,C), VecSub(D,A));
  E2 := VecMul(VecSub(B,C), VecSub(D,B));
  RETURN ScalarMul(E1,E2) < 0.0
END OppSides;
```

If the point R is on the segment V_1V_2, then the function procedure *Between* returns TRUE, otherwise it returns FALSE.

```
PROCEDURE Between(R, V1, V2: Point): BOOLEAN;
  PROCEDURE Opposite(): BOOLEAN;
  BEGIN
     RETURN ScalarMul(VecSub(R,V1),
                      VecSub(R,V2)) <= 0.0
  END Opposite;
  PROCEDURE SameLine(): BOOLEAN;
  VAR ZeroVec, E: Point;
  BEGIN
     ZeroVec[1] := 0.0;
     ZeroVec[2] := 0.0;
     ZeroVec[3] := 0.0;
     E := VecMul(VecSub(R,V1), VecSub(R,V2));
     RETURN VecEqual(E, ZeroVec)
  END SameLine;
BEGIN
  RETURN SameLine() AND Opposite()
END Between;
```

If the point R belongs to the interior domain of the simple polyhedron S, then function procedure *Internal* returns TRUE, otherwise it returns FALSE.

```
PROCEDURE Internal(R: Point; S: Polygon):
                                         BOOLEAN;
CONST Inf = 200.0;
VAR i, K: CARDINAL;
    RInf: Point;
    PROCEDURE NextV(i: CARDINAL): Point;
    BEGIN
      RETURN S.V[(i MOD S.No) + 1]
    END NextV;
    PROCEDURE Next2V(i: CARDINAL): Point;
    VAR Ind: CARDINAL;
    BEGIN
      Ind := (i MOD S.No) + 1;
      RETURN S.V[(Ind MOD S.No) + 1]
    END Next2V;
    PROCEDURE PrevV(i: CARDINAL): Point;
    BEGIN
      IF i=0 THEN
         RETURN S.V[S.No]
      ELSE
         RETURN S.V[i-1]
      END
    END PrevV;
BEGIN
 K := 0;
 WITH S DO
  RInf := VecAdd(VecScMul(Inf, VecSub(R,V[1])),
                V[1]);
  FOR i := 1 TO No DO
   IF Between(R, V[i], NextV(i)) THEN
      RETURN TRUE
   ELSIF Between(V[i], R, RInf) AND
         Between(NextV(i), R, RInf) AND
         OppSides(PrevV(i),Next2V(i),R,RInf) THEN
              INC(K)
   ELSIF Between(V[i], R, RInf) AND
         OppSides(PrevV(i),NextV(i),R,RInf) THEN
              INC(K)
   ELSIF OppSides(V[i], NextV(i), R, RInf) AND
           OppSides(R, RInf, V[i], NextV(i)) THEN
              INC(K)
   END
  END;
 END;
 RETURN (K<>0) AND ODD(K)
END Internal;
```

A.3. PROCEDURE PolyhedraIntersection

The procedure determines if two simple polyhedra intersect. Additional procedures *InterExists* and *Sameplane* are based on relations (5)-(7).

If the intersection between the segment E and the plane determined by polygon S is not an empty set, then the function procedure *InterExists* returns TRUE, otherwise it returns FALSE.

```
PROCEDURE InterExists(E: Edge; S: Polygon):
                                         BOOLEAN;
VAR N: Point;
    E1, E2: REAL;
BEGIN
  WITH S DO
    N := VecMul(VecSub(V[1], V[2]),
                VecSub(V[2], V[3]));
    E1 := ScalarMul(VecSub(E.F, V[1]), N);
    E2 := ScalarMul(VecSub(E.S, V[1]), N);
  END;
  RETURN E1*E2 <= 0.0
END InterExists;
```

If the segment E belongs to the plane determined by polygon S, then the function procedure *SamePlane* returns TRUE, otherwise it returns FALSE.

```
PROCEDURE SamePlane(E: Edge; S: Polygon):
                                        BOOLEAN;
VAR N: Point;
    E1, E2: REAL;
BEGIN
  WITH S DO
    N := VecMul(VecSub(V[1], V[2]),
                VecSub(V[2], V[3]));
    E1 := ScalarMul(VecSub(E.F, V[1]), N);
    E2 := ScalarMul(VecSub(E.S, V[1]), N);
  END;
  RETURN (E1 = 0.0) AND (E2 = 0.0)
END SamePlane;
```

If the intersection between the segment E and the hull of S is not an empty set, then the function procedure *HullIntersect* returns TRUE, otherwise it returns FALSE. Procedure is based on mutual application of procedure *OppSides*.

```
PROCEDURE HullIntersect(E: Edge; S: Polygon):
                                        BOOLEAN;
VAR i: CARDINAL;
   NV: Point;
BEGIN
  WITH S DO
    FOR i := 1 TO No DO
      NV := V[(i MOD No)+1];
      IF OppSides(V[i], NV, E.F, E.S) AND
         OppSides(E.F, E.S, V[i], NV) THEN
           RETURN TRUE
      END;
    END;
  END;
  RETURN FALSE
END HullIntersect;
```

Function procedure *CrossingPoint* returns the piercing point between the segment E and the plane determined by polygon S. The procedure is called only when E is piercing the plane.

```
PROCEDURE CrossingPoint(E: Edge; S: Polygon):
                                        Point;
VAR R: Point;
    Aa, Bb, Cc, Dd, L: REAL;
BEGIN
  WITH S DO
    Aa:= (V[2,2]-V[1,2])*(V[3,3]-V[1,3])
       - (V[3,2]-V[1,2])*(V[2,3]-V[1,3]);
    Bb:= (V[2,3]-V[1,3])*(V[3,1]-V[1,1])
       - (V[2,1]-V[1,1])*(V[3,3]-V[1,3]);
    Cc:= (V[2,1]-V[1,1])*(V[3,2]-V[1,2])
       - (V[3,1]-V[1,1])*(V[2,2]-V[1,2]);
    Dd:= -V[1,1]*(V[2,2]-V[1,2])*(V[3,3]-V[1,3])
         -V[1,3]*(V[2,1]-V[1,1])*(V[3,2]-V[1,2])
         -V[1,2]*(V[2,3]-V[1,3])*(V[3,1]-V[1,1])
         +V[1,3]*(V[3,1]-V[1,1])*(V[2,2]-V[1,2])
         +V[1,1]*(V[3,2]-V[1,2])*(V[2,3]-V[1,3])
         +V[1,2]*(V[2,1]-V[1,1])*(V[3,3]-V[1,3]);
    L:= (Aa*E.F[1]+Bb*E.F[2]+Cc*E.F[3]+Dd) /
        (Aa*(E.S[1]-E.F[1])+Bb*(E.S[2]-E.F[2])+
          Cc*(E.S[3]-E.F[3]));
   END;
   R[1]:=E.F[1]-L*(E.S[1]-E.F[1]);
   R[2]:=E.F[2]-L*(E.S[2]-E.F[2]);
   R[3]:=E.F[3]-L*(E.S[3]-E.F[3]);
   RETURN R
END CrossingPoint;
```

If the intersection between the segment E and the polygon S is not an empty set, then the function procedure Intersect returns TRUE, otherwise it returns FALSE. The procedure is based on afore-mentioned procedures and the algorithm described in section 2.2 of the paper.

```
PROCEDURE Intersect(E: Edge; S: Polygon):
                                        BOOLEAN;
BEGIN
  IF InterExists(E, S) THEN
     IF SamePlane(E, S) THEN
        IF HullIntersect(E, S) THEN
           RETURN TRUE
        ELSE
           RETURN Internal(E.F, S) OR
                  Internal(E.S, S)
        END
     ELSE
        RETURN Internal(CrossingPoint(E, S), S)
     END;
  END;
  RETURN FALSE;
END Intersect;
```

If the intersection of simple polyhedra P and Q is not an empty set, then the function procedure *PolyhedraIntersection* returns TRUE, otherwise it returns FALSE.

```
PROCEDURE PolyhedraIntersection(P, Q:
                          Polyhedron): BOOLEAN;
VAR i, j: CARDINAL;
BEGIN
  FOR i:=1 TO P.NoE DO
    FOR j:=1 TO Q.NoF DO
      IF Intersect(Edg(P,i), Fac(Q,j)) THEN
         RETURN TRUE
      END
    END
  END;
  FOR i:=1 TO Q.NoE DO
    FOR j:=1 TO P.NoF DO
      IF Intersect(Edg(Q,i), Fac(P,j)) THEN
         RETURN TRUE
      END
    END
  END;
  RETURN FALSE
END PolyhedraIntersection;
```

KONVEKSNI OPTIMIZACIJSKI PROBLEMI Z LINEARNIMI OMEJITVAMI

Keywords: convex function, convex polyhedron, local optimization, linear boundaries

Tjaša Meško
Fakulteta za elektrotehniko in računalništvo

POVZETEK. Za konveksno funkcijo na konveksni množici je značilno, da je vsak lokalni minimum tudi globalen. Zato lahko uporabimo postopek lokalne optimizacije. V tem članku bomo obravnavali iteracijsko metodo iskanja minimuma konveksne funkcije na konveksnem poliedru. Najprej poiščemo možno rešitev konveksnega poliedra, ki je tudi začetna točka v prvi iteraciji. V iteracijskem postopku poiščemo dopustno smer, v kateri funkcija pada, nato poiščemo minimalno vrednost funkcije pri pogoju, da rešitev ostane možna, pri tem pa se pomikamo v dopustni smeri. To je začetna točka naslednje iteracije. Postopek je končan, ko najdemo minimum, ali ko se minimumu dovolj približamo.

ABSTRACT. For a convex function on a convex solution set it holds that every local minimum is also global. In this case the local optimization methods are used. In the following article we are dealing with the iteration method of searching a minimum of a convex function on a convex polyhedron. First a feasible solution of the polyhedron is found, which is also the starting point for the first iteration. In the iteration process we find a feasible direction in which function is decreasing then we find the minimum of the function in this direction considering the constraint that the solution has to stay feasible. This is the starting point for the next iteration. The process is terminated when we find the minimum or when we approach the minimum close enough.

1. Formulacija naloge

Problem konveksne optimizacije pri linearnih omejitvah zapišemo v obliki

$$\min f(x) \qquad (1.1)$$

pri pogojih

$$a_i'x = b_i \qquad i \in E \qquad (1.2)$$

$$a_i'x \geq b_i \qquad i \in N \qquad (1.3)$$

kjer je f povsod zvezno odvedljiva konveksna funkcija, E je množica indeksov omejitev v obliki linearnih enačb in N množica indeksov omejitev v obliki linearnih neenačb [6] Morebitni pogoji nenegativnosti spremenljivk so podani v obliki neenačb (1.3), zato niso posebej navedeni. Neenačb z neenačajem ≤ ne bomo uporabljali, saj jih je mogoče pretvoriti v neenačbe z neenačajem ≥, kar dosežemo z množenjem leve in desne strani neenačbe z -1. Funkcija

$$f: R^n \rightarrow R$$

je konveksna, če velja

$$f(pa + (1-p)b) \leq pf(a) + (1-p)f(b)$$

kjer sta $a \in R^n$ in $b \in R^n$ poljubno izbrana, $p \in R$ pa zadošča pogoju

$$0 \leq p \leq 1$$

Rešiti dano nalogo pomeni poiskati tiste vrednosti spremenljivk $x_1, x_2, \ldots, x_n$, to je komponent vektorja $x \in R^n$, pri katerih ima funkcija (1.1) najmanjšo vrednost, pri čemer morajo spremenljivke oziroma vektor x zadoščati pogojem (1.2) in (1.3).

Nalogo rešujemo tako, da najprej poiščemo možno rešitev $x \in B$, ki zadošča pogojem (1.2) in (1.3) neglede na vrednost namenske funkcije, in indekse omejitev, za katere velja enačaj pri začetni rešitvi. Takim omejitvam rečemo aktivne omejitve. Nato prehajamo z iteracijskim postopkom od dane k ugodnejši rešitvi. V vsaki iteraciji najprej poiščemo dopustno smer, v kateri v bližini dane možne rešitve funkcija f pada, nato poiščemo minimalno vrednost funkcije f pri pogoju, da rešitev ostane možna, pri tem pa se pomikamo v izbrani dopustni smeri. Ta nova rešitev je začetna rešitev v naslednji iteraciji. Iskanje prekinemo, ko najdemo tako možno rešitev, pri kateri ni mogoče najti dopustne smeri, v kateri funkcija f pada. V tej točki ima funkcija lokalni minimum. Ker pa je funkcija konveksna, smo na ta način našli globalni minimum. V večini primerov do take točke ne pridemo, zato se zadovoljimo s približno rešitvijo.

2. Potek iteracije

Vzemimo, da imamo pred pričetkom k-te iteracije znano možno rešitev $x^k \in B$ in množico indeksov aktivnih omejitev I^k, da za vsak $i \in I^k$ velja

$$a_i' x^k = b_i \qquad (2.1)$$

nato pa v vsaki iteraciji rešitev izboljšamo [1]. Poskusimo najti dopustno smer, ki leži na premici

$$x = x^k - pu \qquad (2.2)$$

kjer je $u \in R^n$ vektor, ki kaže v dopustni smeri. Izberimo $i \in I^k$, da za možno rešitev x^k velja (2.1). Zapišimo najprej pogoj, da premica z enačbo (2.2) leži v hiper ravnini z enačbo

$$a_i' x = b_i \qquad (2.3)$$

Vektor u mora biti takšen, da za vsako točko premice, torej pri poljubnem p drži (2.3), če namesto x vstavimo (2.2). Zahtevamo, da bo

$$a_i'(x^k - pu) = b_i \qquad (2.4)$$

Če p ni enak Ø, zaradi (2.1) iz (2.4) sledi

$$a_i' u = Ø \qquad (2.5)$$

Vzemimo sedaj $i \in N \cap I^k$ in ugotovimo, kateremu pogoju morajo zadoščati komponente vektorja u, da bo v (1.3) pri pozitivnem p veljal pravilen neenačaj, če namesto x vstavimo (2.2). Iz (1.3), (2.1) in (2.2) sledi, da mora vektor u pri $p > Ø$ zadoščati neenačbi

$$a_i' u \le Ø \qquad (2.6)$$

Za vsak $i \in E$ mora seveda vedno veljati (2.5).

Raziščimo, kako velik sme biti p, da bo spremenjena rešitev ostala možna. Lahko se namreč zgodi, da prekoračimo katero izmed neaktivnih omejitev, če je p prevelik. V ta namen vzemimo neaktivno omejitev in ugotovimo, kakšnemu pogoju mora zadoščati p, da leva stran neenačbe ne bo manjša od desne v (1.3). Namesto (2.4) dobimo za neaktivno omejitev pogoj

$$a_i'(x^k - pu) \ge b_i$$

ali

$$a_i' x^k - b_i \ge p a_i' u$$

Ker je leva stran pri neaktivni omejitvi zaradi (1.3) pozitivna, pri pogoju (2.6) pri poljubnem pozitivnem p v pripadajoči omejitvi velja pravilen neenačaj. Samo v tistih neaktivnih omejitvah, za katere ne velja (2.6), lahko pri pozitivnem p pride do prekoračitve. Če za vsako neaktivno omejitev, ki ne zadošča pogoju (2.6), p zadošča neenačbi

$$Ø \le p \le \frac{a_i' x^k - b_i}{a_i' u} \qquad (2.7)$$

bo spremenjena rešitev (2.2) možna. Zato p ne sme biti večji od najmanjšega pozitivnega količnika (2.7) pri neaktivni omejitvi.

Vzemimo poljubno smer, ki jo določa vektor t. Zapišimo ga kot vsoto vektorja u, ki za vsak $i \in I^k$ zadošča pogoju (2.5), in vektorja, ki je linearna sestava vektorjev a_i, $i \in I^k$. Koeficiente te linearne sestave zaznamujmo z v_i. Vektor t torej zadošča pogoju

$$t = u + \sum_{i \in I^k} v_i a_i \qquad (2.8)$$

To vektorsko enačbo lahko zapišemo za komponente. Denimo, da imamo h aktivnih omejitev. Zaradi lažjega pisanja naj bodo indeksi omejitev urejeni tako, da imajo aktivne omejitve indekse od 1 do h. Tedaj namesto (2.8) dobimo enačbe

$$u_j + a_{1j} v_1 + a_{2j} v_2 + \dots + a_{hj} v_h = t_j$$
$$j = 1, \dots, n$$

Če vektor u zadošča pogojem (2.5) za vsak $i \in I^k$, ima dopustno smer. Tudi skalarni produkt $a_i' u$, ki nastopa v (2.5), je mogoče zapisati na dolgo in namesto (2.5) dobimo

$$a_{i1} u_1 + a_{i2} u_2 + \dots + a_{in} u_n = Ø \qquad i \in I^k$$

Tako imamo sistem n+h linearnih enačb z n+h neznankami, kjer h pomeni moč množice I^k, to je število aktivnih omejitev. Dokazati je mogoče, da je tako dobljen sistem enačb vedno enolično rešljiv, če polieder možnih rešitev B ni degeneriran. To pomeni, da v nobeni točki normale aktivnih omejitev niso odvisne. Ta zahteva je v praktičnih nalogah le redko izpolnjena. Ker pa pa mora navedeni sistem biti vedno rešljiv, pri pisanju računalniškega programa po potrebi degeneracijo odpravimo z majhnimi spremembami stalnih členov.

Če levo in desno stran (2.8) skalarno množimo z u in upoštevamo (2.5), dobimo

$$t'u = u'u + \sum_{i \in I^k} v\, a_i' u_i = u'u = \sum_{j=1}^{n} u_j^2 \ge Ø \qquad (2.9)$$

Enačaj velja samo, če je $u = Ø$, to je, če je

$$u_1 = u_2 = \dots = u_n = Ø$$

Zaznamujmo z A_h matriko

$$A_h = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{h1} \\ a_{12} & a_{22} & \cdots & a_{h2} \\ \vdots & \vdots & & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{hn} \end{bmatrix}$$

z A_h' transponirano matriko in z v vektor s komponentami $v_1,\dots,v_h$. Tedaj lahko enačbi (2.8) zapišemo v obliki

$$u + A_h v = t \qquad (2.1Ø)$$

Če levo in desno stran te enačbe z leve množimo z $A_h'v$, dobimo

$$A_h'u + A_h'A_h v = A_h't$$

Zaradi (2.5) je prvi sumand enak Ø. Zato po množenju z leve z inverzno matriko matrike $A_h'A_h$ dobimo enačbo

$$(A_h'A_h)^{-1}(A_h'A_h)v = (A_h'A_h)^{-1}A_h't$$

od koder sledi, da je

$$v = (A_h'A_h)^{-1}A_h't \qquad (2.11)$$

Če v (2.1Ø) upoštevamo (2.11), dobimo

$$u = t - A_h(A_h'A_h)^{-1}A_h't \qquad (2.12)$$

Uporabljena inverzna matrika obstaja natanko takrat, ko so normale na hiper ravnine, ki pripadajo aktivnim omejitvam, neodvisne ć3ë.

Vzemimo, da je zadnja aktivna omejitev dobljena iz neenačbe in naj ima indeks h. Naj bo

$$v_h < 0 \qquad (2.13)$$

Ker je $h \in I^k$, je

$$a_h'x^k = b \qquad (2.14)$$

Če upoštevamo, da je $a_h'a_h > 0$, pri pogoju (2.13) iz (2.14) sledi

$$a_h'(x^k - v_h a_h) - b_h = -v_h a_h'a_h > 0$$

Komponenta $v_h a_h$ torej pri pogoju (2.13) ne povzroči prekoračitve h-te omejitve. Od tod sklepamo, da smemo v (2.8) opustiti sumand $v_h a_h$. Zato razstavimo vektor $v_h a_h$ na dve komponenti. Prva naj bo izražena v obliki linearne sestave vektorjev $a_1,\dots,a_{h-1}$, druga pa pravokotna na prvo. To dosežemo tako, da rešimo sistemu (2.5), (2.1Ø) analogen sistem enačb

$$a_i'y = 0 \qquad i=1,\dots,h-1 \qquad (2.15)$$

$$y + A_{h-1}z = v_h a_h \qquad (2.16)$$

Če v h-to omejitev namesto x vstavimo

$$x = x^k - p(u + y)$$

ter upoštevamo (2.14), (2.5), (2.16), (2.15) in (2.13), pri pozitivnem p dobimo

$$a_h'(x^k - p(u + y)) = b_h - pa_h'(u + y) =$$

$$= b_h - pa_h'y = b_h - \frac{p}{v_h}(y' + A_{h-1}z)y =$$

$$= b_h - \frac{p}{v_h}y'y > b_h$$

V h-ti omejitvi torej pri pozitivnem p in pri pogoju (2.13) dobimo dopustno odstopanje od enakosti. Smer vektorja u+y je ugodnejša od smeri vektorja u [3].

Vzemimo namesto vektorja t gradient funkcije f v točki x^k. Funkcija f narašča, če se pomikamov smeri, pri kateri je skalarni produkt vektorja, ki kaže v opazovani smeri, in gradienta, pozitiven [5]. Če torej v (2.12) namesto t vstavimo gradient funkcije f v točki x^k, zaradi (2.9) vektor u kaže v smeri, v kateri funkcija f narašča. Funkcija f v smeri u+y narašča hitreje kot v smeri u [3]. Zato je pri pogoju (2.13) pri iskanju minimuma funkcije f ugodneje vzeti premik v smeri -(u+y) kot v smeri -u.

Vektor u+y lahko dobimo tudi tako, da rešimo sistem enačb, ki ga dobimo, če tvorimo enačbe (2.5) za i=1,...,h-1 in enačbe (2.8), kjer h-ta omejitev ne spada v množico I^k. Ugodnejšo smer od smeri, ki jo podaja vektor u, lahko dobimo vedno, kadar obstaja v izrazu (2.8) vsaj za eno aktivno omejitev, ki ima obliko neenačbe, negativen v_j. Ni pa nobenega zagotovila, da bo smer ostala dopustna, če obstajata dva negativna koeficienta in iz množice aktivnih omejitev pri tvorbi (2.8) izločimo dva indeksa. Zato bomo v k-ti iteraciji iz množice aktivnih omejitev izločili tisti indeks neenačbe, pri katerem najdemo negativen in po absolutni vrednosti največji koeficient, če tak koeficient obstaja. Če nobeni neenačbi ne pripada negativen koeficient, je smer vektorja -u najugodnejša dopustna smer.

Zaznamujmo odslej z u^k vektor, ki ga dobimo kot rešitev sistema enačb oblike (2.5), (2.8) neglede na to ali smo v k-ti iteraciji katero aktivno omejitev spremenili v neaktivno ali ne. Tedaj iščemo novo možno rešitev, pri kateri ima funkcija f manjšo vrednost kot pri rešitvi x^k, v obliki

$$x^{k+1} = x^k - pu^k$$

kjer je potrebno p določiti tako, da bo

$$f(x^{k+1}) < f(x^k)$$

če je to mogoče. Nova rešitev bo možna, če p zadošča pogoju (2.7). Videti je, da je primerno iskati minimum funkcije

$$g(p) = f(x^k - pu^k) \qquad (2.17)$$

Reševanje take naloge pa je lahko zamudno. Upoštevati moramo, da poteka pri vsaki funkciji reševanje take naloge drugače, zato pripadajočega računalniškega programa ni primerno vključiti v splošen računalniški program. Po drugi strani pa postopek pogosto hitreje konvergira, če je p nekoliko manjši od tistega, pri katerem ima funkcija (2.17) minimalno vrednost. To je mogoče tudi pričakovati, saj je v točki, v kateri ima funkcija (2.17) minimum, v splošnem najugodnejša smer pravokotna na smer v predhodni iteraciji. Če hočemo zagotoviti hitrejšo konvergenco opisanega postopka, moramo s pomočjo drugih odvodov ali kako drugače poiskati smer, v kateri funkcija ne pada tako hitro kot v najugodnejši smeri, vendar pa ni pravokotna na smer v prejšnji iteraciji. Pojav, da je zaporedje smeri sestavljeno iz pravokotnih smeri, poznamo kot cik-cak. S tovrstnimi problemi se ne bomo ukvarjali, raje bomo izvajali večje število iteracij, postopek v iteraciji pa bomo izvedli čim bolj preprosto.

Ker se moramo ozirati na parcialne odvode funkcije f, moramo pri sestavljanju računalniškega programa predvideti, kako jih bomo računali. Vrednosti parcialnih odvodov potrebujemo le v posameznih ¸točkah, zato jih je primerno računati tako, da za vsakega izračunamo diferenčni količnik, pri čemer vzamemo diferenco argumentov tako majhno, da ne trpi natančnost rezultata. To je mogoče doseči, saj se napake, ki jih naredimo pri računanju parcialnih odvodov, ne prenašajo iz iteracije v iteracijo, tako kot na primer pri reševanju sistemov linearnih enačb. Namesto parcialnega odvoda funkcije f na spremenljivko x_j zazanamujmo ga z f_j, smemo vzeti

$$f_j = \frac{1}{d}(f(x_1,\dots,x_j+d,\dots,x_n) -$$
$$- f(x_1,\dots,x_j,\dots,x_n))$$

kjer je d dovolj majhno pozitivno število, na primer 0,0001.

3. Konvergenca na primerih

Poiščimo minimum funkcije

$$f(x) = 2x_1^2 - x_1x_2 + 4x_2^2 - 5x_1 - x_2 + 3$$

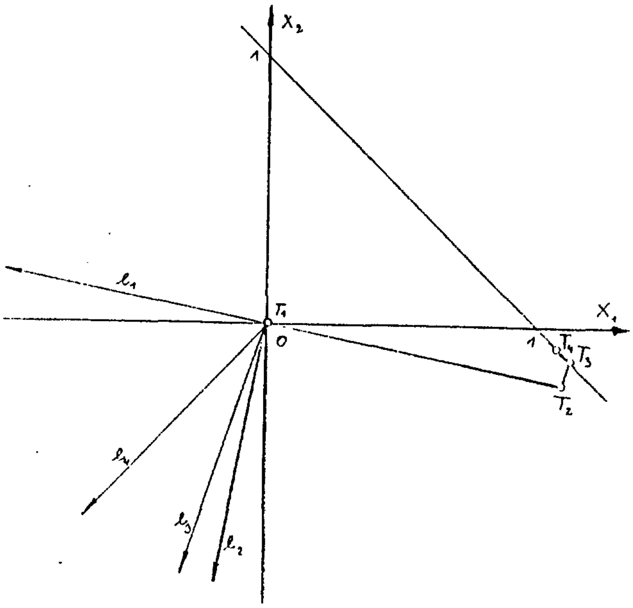
pri pogoju

$$x_1 + x_2 \leq 1$$

Funkcija je eliptični paraboloid in je konveksna. Računalniški program v pascalu je sestavljen tako, da vsebuje proceduro za računanje funkcijske vrednosti, ki jo uporabljamo tudi pri računanju gradienta. Po vsaki iteraciji program izpiše številko iteracije, komponenti gradienta, kompomnenti vektorja u in vektorja x po premiku in pripadajočo funkcijsko vrednost ć3ë. Če preiskusimo postopek na zgoraj podanem primeru, vidimo, da potrebujemo le 6 iteracij in absolutna vrednost projekcije gradienta je manjša od 10^{-5}. Iz slike 1 je razvidno, da je kot mad gradientom in premico

$$x_1 + x_2 = 1 \qquad (3.1)$$

iz koraka v korak večji, zato se projekcija gradienta manjša, s tem pa se manjša tudi dolžina premika. V prvih dveh iteracijah se premaknemo v gradientu nasprotni smeri, v preostalih iteracijah pa to ni več mogoče, ker gradientu nasprotna smer ni več dopustna. Zato se premeknemo po nasprotni smeri projekcije gradienta na premico (3.1). Namesto gradientov, so na sliki 1 enotski vektorji, ki imajo smeri gradientov. Enotske vektorje smo zaznamovali z e_1, e_2, e_3 in e_4.



Slika 1. Prikaz postopka

Če je funkcija konveksna, postopek v splošnem hitro konvergira. Če pa ni konveksna, to ni zmeraj res. Če ima funkcija več lokalnih minimumov, pa ne moremo trditi, da bomo našli globalni minimum. Preiskusimo postopek na nekonveksni funkciji

$$f(x) = ((x_1 - 1)^2 + (x_2 - 1)^2 - 1)^2 - 0.1x_1$$

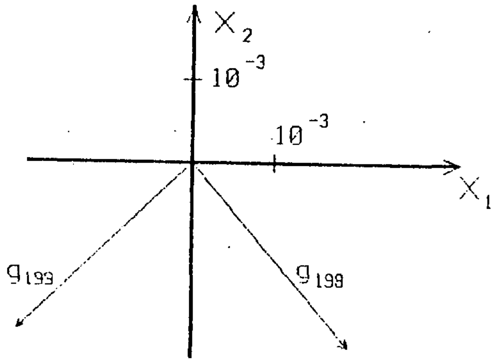
ki ima samo en lokalni minimum. Na krožnici z enačbo

$$(x_1 - 1)^2 + (x_2 - 1)^2 = 1$$

ima prvi člen vrednost 0, povsod drugod pa je pozitiven. Drugi člen je v teh točkah najmanjši pri $x_1 = 2$, $x_2 = 1$. Zato ima funkcija f(x) v v bližini te točke globalni minimum. Množica možnih rešitev naj bo neomejena. Minimum te funkcije bomo iskali v dveh primerih. V prvem primeru bomo natančneje iskali minimum funkcije (2.17) kot v drugem. Pri natančnem iskanju minimuma dobimo:

```
Začetna rešitev
X[1] = 0.0000      X[2] =  0.0000

  K    U[1]     U[2]     X[1]     X[2]     F(X)

  1  -4.0999  -3.9999  0.2939   0.2867  -0.02966
  2  -0.1202  -0.0204  0.3130   0.2900  -0.03072
  3  -0.0338   0.0683  0.3249   0.2659  -0.03246

198   0.0019  -0.0022  2.0117   0.9784  -0.20059
199  -0.0021  -0.0020  2.0122   0.9789  -0.20059
200   0.0018  -0.0021  2.0118   0.9794  -0.20059
201  -0.0020  -0.0019  2.0122   0.9799  -0.20059

322   0.0000  -0.0000  2.0122   0.9991  -0.20061
323  -0.0001  -0.0000  2.0122   0.9991  -0.20061
324   0.0000  -0.0000  2.0122   0.9992  -0.20061

Minimalna funkcijska vrednost -0.2006 je dosež-
ena v točki X[1] = 2.0123     X[2] = 0.9992
```

K v tabeli pomeni zaporedno številko iteracije. Izpisali smo prve tri iteracije, iteracije od 198. - 201. in zadnje tri iteracije. Ker ni omejitev, je gradient enak vektorju $\dot{u}$. Če pogledamo gradiente v iteracijah 198. - 201., vidimo, da so paroma pravokotni. Na sliki 2 sta predstavljena gradienta iteracij 198 in 199. Tudi skalarni produkt je v vseh treh primerih reda 10^{-7}. V zadnjih treh iteracijah pravokotnosti gradientov ne moremo preverjati, ker je natančnost izpisa premajhna.



Slika 2. Predstavitev gradientov

V drugem primeru minimuma funkcije (2.17) ne računamo natančno in dobimo manj iteracij.

```
Začetna rešitev
X[1] = 0.0000      X[2] = 0.0000

  K    U[1]     U[2]     X[1]     X[2]     F(X)

  1  -4.0999  -3.9999  0.1295   0.1264   0.02674
  2  -1.9131  -1.8197  0.1779   0.1724  -0.00624
  3  -1.2858  -1.1937  0.2429   0.2328  -0.02255

187  -0.0042  -0.0024  2.0118   0.9744  -0.20058
188  -0.0012  -0.0024  2.0120   0.9748  -0.20058
189   0.0004  -0.0024  2.0110   0.9802  -0.20059

244  -0.0002  -0.0001  2.0122   0.9989  -0.20061
245  -0.0001  -0.0001  2.0122   0.9990  -0.20061
246  -0.0000  -0.0001  2.0122   0.9990  -0.20061

Minimalna funkcijska vrednost -0.2006 je dosež-
ena v točki X[1] = 2.0123     X[2] = 0.9990
```

Gradienti v drugem primeru paroma niso pravokotni, zato je konvergenca boljša. Podobne primere najdemo v literaturi [4].

4. Zaključek

Postopek, ki je opisan v prejšnjih poglavjih, pride v poštev pri optimizaciji poslovanja [2]. Poslovno odločitev je mogoče kvantificirati z n odločitvenimi spremenljivkami, ki pomenijo iskane količine proizvodnje. Zaradi omejenih količin elementov poslovnega procesa in zaradi omejenih potreb trga ni mogoče izbrati vrednosti odločitvenih spremenljivk poljubno, ampak morajo zadoščati pogojem, ki jih lahko zapišemo v obliki linearnih enačb in neenačb. Med pogoji so običajno tudi pogoji nenegativnosti odločitvenih spremenljivk.

Če so razpoložljive količine elementov in potrebe trga odvisne od slučajnih vplivov, jih ne moremo z gotovostjo napovedati. Ker pa zadovoljimo določeno potrebo šele ob koncu poslovne akcije, za katero se moramo vnaprej odločiti, moramo odločitve o izvajanju poslovne akcije sprejeti preden poznamo realizacijo poslovnih pogojev.

Pri znanih poslovnih pogojih lahko potrebe po i-tem elementu poslovnega procesa izrazimo kot linearno funkcijo odločitvenih spremenljivk in jih primerjamo z razpoložljivo količino. Če je razpoložljiva količina i-tega elementa, zaznamujmo jo s t_i, enaka potrebam, dobimo enačbo

$$t_i = r_{i1}x_1 + r_{i2}x_2 + \ldots + r_{in}x_n \qquad i=1,\ldots,k \qquad (4.1)$$

kjer r_{ij} pomeni potrebno količino i-tega elementa za proizvodnjo ene enote j-tega izdelka in x_j odločitveno spremenljivko, ki pomeni iskano količino proizvodnje j-tega izdelka. Zaradi slučajnih vplivov, ni mogoče vnaprej določiti vrednsti odločitvenih spremenljivk tako, da bi veljale enačbe (4.1). Tedaj je tudi odstopanje od enakosti

$h_i = t_i - r_{i1}x_1 - r_{i2}x_2 - \dots - r_{in}x_n$

slučajna spremenljivka. Taka odstopanja povzročijo dodatne stroške, zato tovrstne stroške upoštevamo v namenski funkciji. Matematično upanje take namenske funkcije je konveksna funkcija. Če imamo poleg pogojev, ki so odvisni od slučajnih vplivov, še običajne linearne pogoje in pogoje nenegativnosti odločitvenih spremenljivk, dobimo nalogo konveksne optimizacije.

Postopek lahko uporabimo tudi, ko iščemo minimum nekonveksne funkcije, ki ima samo en minimum na območju, ki je podano z omejitvami. Primeri kažejo, da pri taki funkciji postopek ne konvergira tako dobro.

Literatura

1. E. Blum in W. Oettli: Mathematische Optimierung. Springer-Verlag, Berlin-Heidelberg-New York 1975.
2. I. Meško: Computerprogramm fUr konvexe und stochastische lineare Optimierung. Operations Research Proceedings 1981, Springer-Verlag, Berlin-Heidelberg 1982, str. 595-601.
3. T. Meško: Konveksni optimizacijski problemi z linearnimi omejitvami. Diplomska naloga, Fakulteta za elektrotehniko in računalništvo, Ljubljana 1989.
4. K. Schittkowski: More Test Examples for Nonlinear Programming Codes. Springer-Verlag, Berlin-Heidelberg 1987.
5. I. Vidav: Višja matematika II. Državna založba Slovenije, Ljubljana 1975.
6. S. Zlobec in J. Petrić: Nelinearno programiranje. Naučna knjiga, Beograd 1989.

DIJAGRAM TOKA PODATAKA ZA PROCES PROJEKTIRANJA I UVODJENJA INFORMACIJSKIH SISTEMA

INFORMATICA 1/90

Keywords: software engineering, process modelling, data flow diagram, data modelling, entity-relationship

Eligio Drandić*

ABSTRACT:
"THE DATA FLOW DIAGRAM FOR INFORMATION SYSTEM DEVELOPMENT"
This paper describes an software engineering approach in information system development based on relational software.
Using a data flow diagrams, the decomposition of the information system development process in 20 subprocesses is given.
The advantages and the key moments of the project approach are emphasized.

KLJUČNE RIJEČI: inženjerijski pristup, modeliranje procesa, dijagram toka podataka, modeliranje podataka, model objekti-veze.

SAŽETAK: U članku je prikazan inženjerijski pristup u izgradnji informacijskih sistema baziranih na relacijskom softveru.
Koristeći dijagram toka podataka izvršena je dekompozicija procesa razvoja informacijskih sistema na 20 podprocesa.
Naglašene su prednosti i ključni momenti u procesu projektiranja.

1. NEOPHODNOST INŽENJERIJSKOG PRISTUPA

Iako je informatička znanost izrazito okrenuta ka budućnosti i koristi najsuvremenija dostignuća, ipak ona zaostaje za drugim znanostima kada je u pitanju primjena inženjerijskih metoda u razvoju informacijskih sistema. Te su metode zadnjih godina u naglom razvoju, posebno njihova automatizacija (CASE alati), a pokrivaju uglavnom:

- strateško planiranje,
- modeliranje procesa,
- modeliranje podataka i
- strukturno projektiranje.

Neophodnost inženjerijskog pristupa u projektiranju informacijskih sistema (i aplikacija), jednaka je kao i neophodnost projektiranja kuće, broda, aviona. To pokazuje i IBM studija o visini troškova za korekciju grešaka na raznim nivoima procesa projektiranja i uvođenja informacijskih sistema (Sl. 1).

Uobičajena greška (uvjetovana neznanjem) koja se čini pri projektiranju i uvođenju informacijskih sistema je preskakanje strateškog planiranja, logičkog i fizičkog projektiranja te direktan ulazak u fizičku realizaciju, testiranje i primjenu.

Na slici su uz engleske nazive pojedinih procesa date i odgovarajuće metode koje se uzimaju kao standardne a pomoću kojih se ti procesi realiziraju:

- za nivo strateškog planiranja: IBM-ovo planiranje poslovnih sistema (BUSINESS SYSTEM PLANING, skraćeno BSP),
- za izradu modela procesa: strukturna sistem analiza (SSA) sa dijagramom toka podataka (DTP) kao osnovom,
- za izradu modela podataka: dijagram objekti-veze (DOV), odnosno ENITITY-RELATIONSHIP (ER),
- za projektiranje programa: strukturno projektiranje.

Budući da je u sadašnjem trenutku relacijski softver najrazvijeniji sistemski softver za upravljanje bazama podataka (i na njemu se

* Elektronski računski centar
Brodogradilište "3. maj"
JNA 13, 51000 Rijeka

razvija najviše informacijskih sistema), proces projektiranja i uvođenja informacijskih sistema prikazan je upravo za relacijski softver.

Prednosti koje nam donosi inženjerijski pristup jesu:

- planiranje razvoja informacijskih sistema (od strateškog plana ka realizaciji),
- uvođenje integriranih informacijskih sistema, uvođenje po podsistemima koji se integriraju u zajednički sistem,
- optimalno korištenje svih postojećih resursa, u prvom redu softvera i hardvera (normalizacija podataka ...),
- usklađivanje potreba i mogućnosti (metoda troškovi-efikasnost),
- povećanje produktivnosti pri izradi softvera, olakšano održavanje postojećeg softvera (npr. korekcija),
- postupnost, sistematičnost i sveobuhvatnost u snimanju realnog sistema, bolja analiza korisničkih zahtjeva, bolja komunikacija sa korisnikom i prilagođenje njegovim potrebama,
- kvalitetniji softverski proizvodi, potpuna dokumentiranost,
- izrada logičkih modela koji su nezavisni od hardvera i softvera (dakle projekata koji se mogu koristiti za bilo koji hardver i softver),
- pravilnost u postavljanju granica ručnog i automatskog izvođenja procesa,
- pravilno definiranje i provođenje reorganizacije realnog sistema radi prilagođenja automatiziranom informacijskom sistemu,
- uočavanje grešaka u ranijim fazama procesa projektiranja, smanjenje grešaka na minimum,
- mogućnost automatizacije razvoja informacijskih sistema kao posljedica formalizacije i standardizacije (CASE alati),
- uvođenje standarda u razvoju informacijskih sistema,
- znatno pojeftinjenje informacijskih sistema (uvođenje, održavanje).

2. MODEL PROCESA

U nastavku je pomoću dijagrama toka podataka prikazan proces projektiranja i uvođenja informacijskih sistema.

Na prvom nivou dekompozicije proces se sastoji iz 6 osnovnih podprocesa, a na drugom iz 21 (tabela 1).

Dijagram konteksta dat je slikom 2, prvi nivo dekompozicije slikom 3, a drugi nivo dekompozicije slikama 4 do 9.
Datoteke koje imaju oznaku i naziv dat izvan grafičkog prikaza, znači da se prvi put pojavljuju, dok datoteke koje imaju oznaku i naziv unutar grafičkog prikaza, znači da su se već ranije pojavile.
Isto vrijedi i za interfejse (izvore i ponore).

Naravno, moguća je i detaljnija dekompozicija.

Svaki konkretni informacijski sistem uvodi se na temelju već definiranog strateškog plana. Dakle, mora prethodno biti definiran plan metodom BSP.

U ovom je radu uzeto da je strateški plan napravljen te se projektiranje nastavlja na njega.

Prikazanim dijagramom toka podataka dobivamo odgovarajuću uzročno-posljedičnu vezu između podprocesa, sve ulazne i izlazne tokove podataka te posebno uočavamo neke presudne momente:

- kada vršiti izbor procesa za automatizaciju,
- kada i na osnovu kojih ulaznih podataka vršiti izbor resursa (hardver, softver,...) za budući informacijski sistem,
- kada se uključuju korisnik i rukovodstvo,
- kako su uspostavljene povratne veze, odnosno petlje unutar kojih tražimo odgovarajuća (optimalna) rješenja,
- kada dolaze do izražaja organizacijske promjene koje donosi novi informacijski sistem,
- kada i kako utječe izbor gotovog softvera na projektiranje informacijskog sistema itd.

Prikazani dijagram toka podataka predstavlja opći proces projektiranja i uvođenja informacijskih sistema, što znači da nastoji zadovoljiti bilo koju moguću varijantu (izbor gotovog softvera, izrada vlastitog, itd.), te za konkretan slučaj treba izbaciti određene podprocese ukoliko za njih nema potrebe.
Za potpuni opis procesa bilo bi potreno koristiti još neku od metoda radi boljeg prikaza resursa, kao npr. mrežni dijagram (gantogram) za vremensku analizu.

Za dodatni opis vrlo je pogodno koristiti tabelarni prikaz (tabela 2) gdje se može dati dodatni opis procesa i svih korištenih resursa (metode, sredstva, izvršioci, vrijeme, itd.).

3. LITERATURA

- Meilir Page-Jones: "The practical Guide to Structured System Design", Prentice-Hall, Inc.,Englewood Cliffs, New Yersey 1980.

- B. Lazarević, V. Jovanović, M. Vučković: "Projektovanje informacionih sistema", I i II deo, Naučna knjiga, Beograd 1986.

- Slavko Tkalec: "Relacijski model podataka", Informator, Zagreb 1986.

- James M. Kerr: "The Data-driven Harvest", Database Programming and Design, May 1989.

- Suad Alagić: "Relacione baze podataka", Svjetlost, Sarajevo 1984.

- James Martin, Carma McClure: "Structured techniques for computing", Prentice-Hall, Inc., Englewood Cliffs, New Yersey 1985.

- Richard Fairley: "Software Engineering Concepts", McGraw-Hill 1985.

- mr. Anton Hauc (redakcija): "Upravljanje projektima" , Informator, Zagreb 1975.

TABELA 1. DEKOMPOZICIJA PROCESA PROJEKTIRANJA I UVOĐENJA INFORMACIJSKOG SISTEMA

1. OPIS POSTOJEĆEG STANJA

1.1. IZRADA MODELA PROCESA POSTOJEĆEG STANJA
1.2. IZRADA MODELA PODATAKA POSTOJEĆEG STANJA
1.3. IZRADA MODELA POSTOJEĆIH RESURSA
1.4. IZRADA ANALIZE POSTOJEĆEG STANJA

2. OPIS BUDUĆEG STANJA

2.1. IZRADA MODELA PROCESA BUDUĆEG STANJA
2.2. IZRADA MODELA PODATAKA BUDUĆEG STANJA
2.3. ANALIZA I IZBOR VARIJANTE REALIZACIJE PROJEKTA
2.4. PRILAGOĐENJE LOGIČKIH MODELA
2.5. IZRADA MODELA POTREBNIH RESURSA
2.6. KOREKCIJA I USVAJANJE PREDLOŽENOG RJEŠENJA

3. REALIZACIJA RESURSA

3.1. IZRADA PLANA REALIZACIJE POTREBNIH RESURSA
3.2. REALIZACIJA POTREBNIH RESURSA

4. FIZIČKO PROJEKTIRANJE I REALIZACIJA BAZE PODATAKA

4.1. PRILAGOĐENJE MODELA PODATAKA
4.2. PREVOĐENJE MODELA OBJEKTI-VEZE U RELACIJSKI
4.3. FIZIČKA REALIZACIJA BAZE PODATAKA

5. FIZIČKO PROJEKTIRANJE I REALIZACIJA PROGRAMA

5.1. IZRADA OPISA LOGIKE PROCESA
5.2. PISANJE PROGRAMSKOG KODA

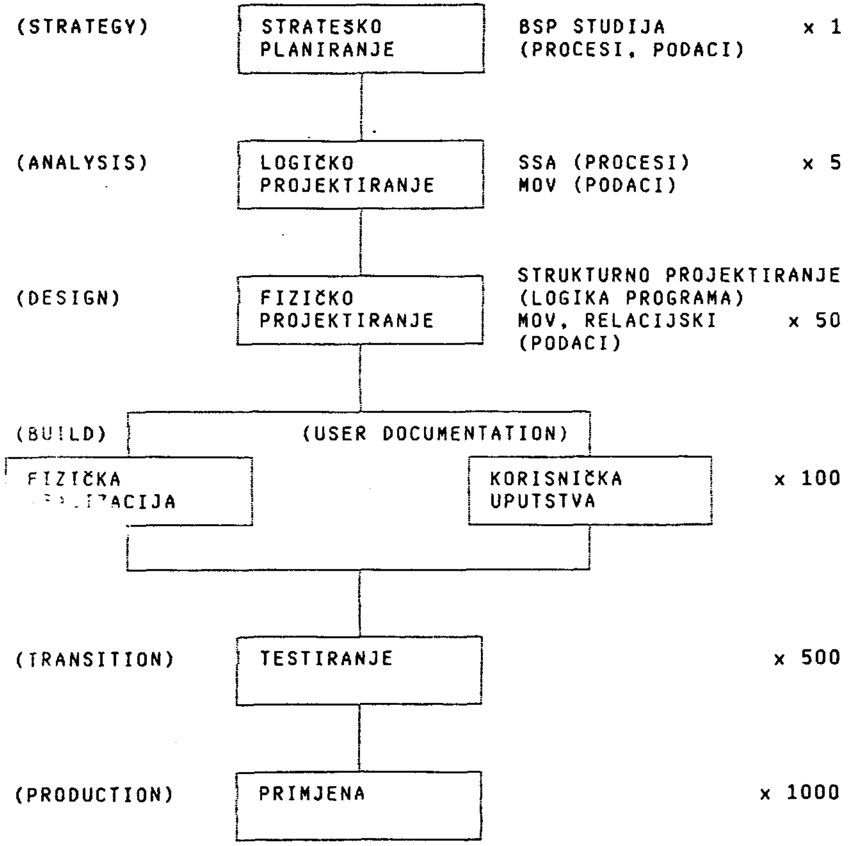
6. IMPLEMENTACIJA I TESTIRANJE

6.1. IZRADA UPUTSTVA ZA KORIŠTENJE
6.2. TESTIRANJE I KOREKCIJA APLIKACIJE
6.3. IZVRŠENJE PRIMOPREDAJE
6.4. IZRADA IZVJEŠTAJA O REALIZACIJI PROJEKTA

TABELA 2. TABELARAN OPIS PROCESA

OZNAKA PROCESA	NAZIV I OPIS PROCESA	ROK	IZVRŠIOCI
	4. FIZIČKO PROJEKTIRANJE I REALIZACIJA BAZE PODATAKA		
4.1.	PRILAGOĐENJE LOGIČKOG MODELA PODATAKA		
	a) Zbog specifičnosti konkretnog relacijskog softvera na kojem ćemo dalje razvijati projekt, treba izvršiti prilagođenje modela objekti-veze. Voditi računa: - koje koncepte modela objekti-veze podržava, a koje ne podržava konkretni relacijski softver (entitete, veze, mješovite entitete ...), - da model bude optimalan za konkretni RSUBP sa stanovišta pristupa podacima (potrebne izmjene tabela, veza, atributa, ključeva ...). Izvršiti analizu da li model odgovara na sve upite koji mu se postavljaju sa stanovišta modela procesa budućeg stanja.		
	b) Ukoliko sa novodefiniranom bazom podataka samo nadograđujemo već postojeću, onda treba ispitati poklapanje dviju struktura (po tabelama, vezama, atributima ...) te novu strukturu prilagoditi već postojećoj.		

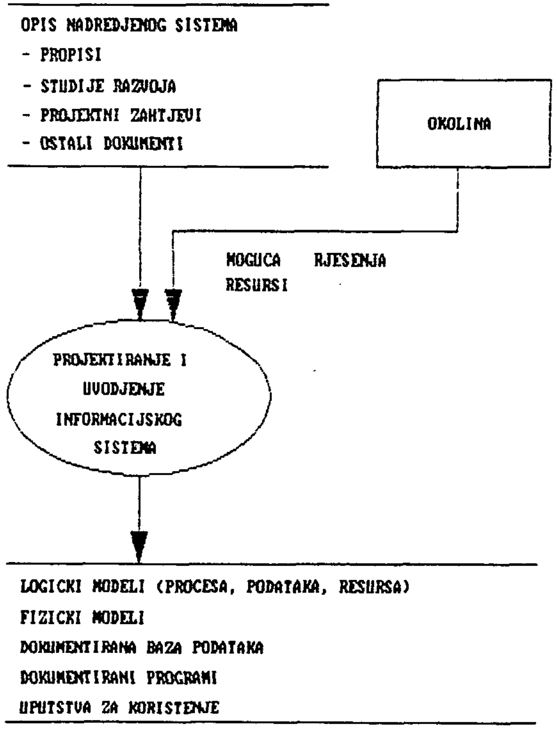
SLIKA 1. TROŠKOVI KOREKCIJE GREŠAKA
(IBM STUDIJA)



Skraćenice:

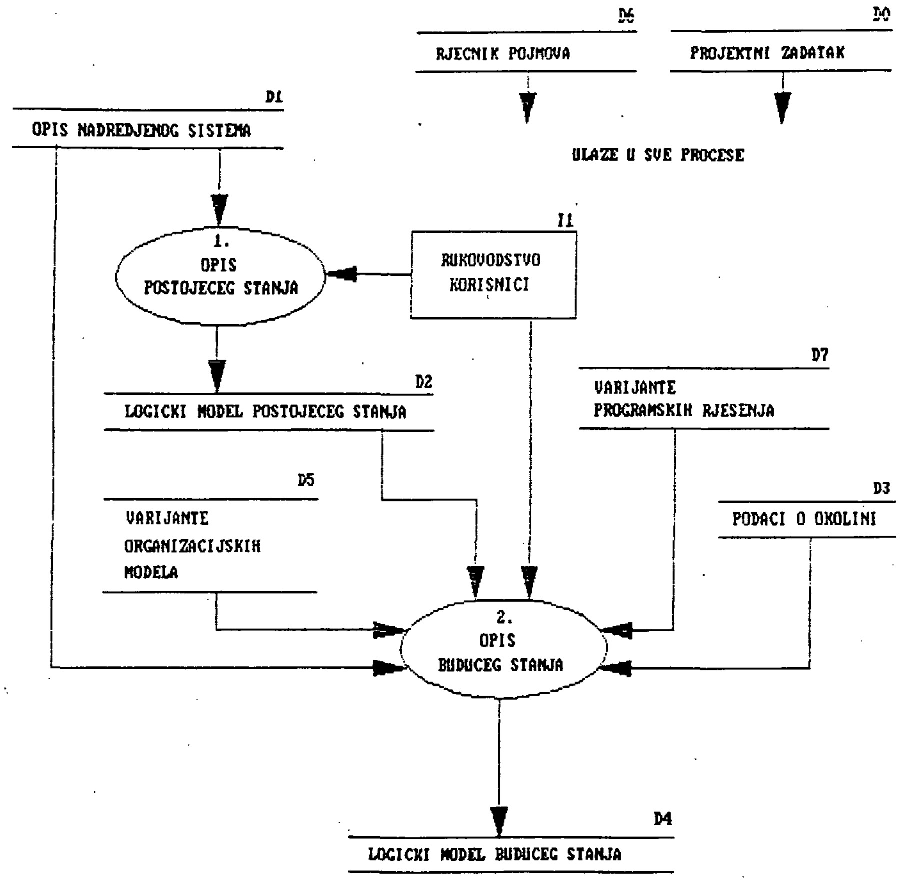
BSP - BUSINESS SYSTEM PLANING (PLANIRANJE POSLOVNOG SISTEMA)
SSA - STRUKTURNA SISTEM ANALIZA
MOV - MODEL OBJEKTI-VEZE

Slika 2. DIJAGRAM KONTEKSTA
(PROJEKTIRANJE I UVODJENJE INFORMACIJSKIH SISTEMA)



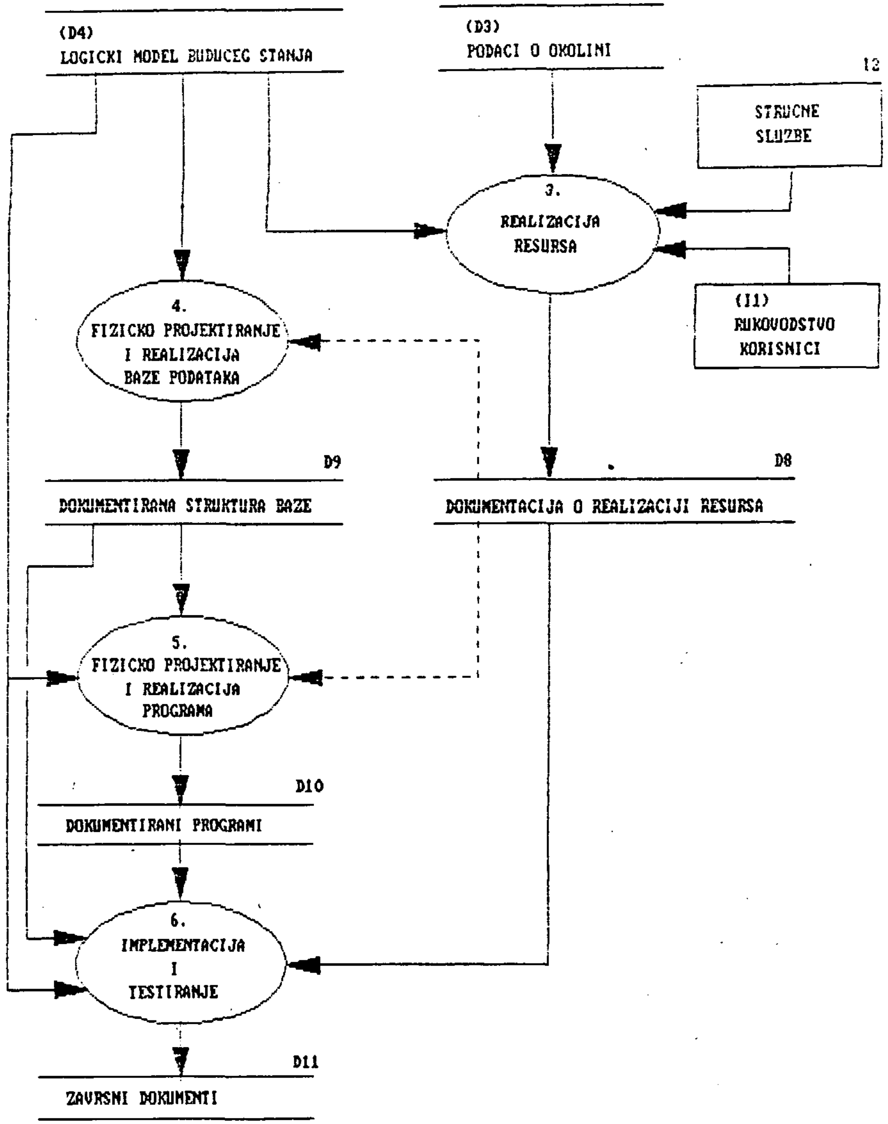
SLIKA 3. PROJEKTIRANJE I UVODJENJE INFORMACIJSKIH SISTEMA

- DIJAGRAM TOKA PODATAKA (1. NIVO) -



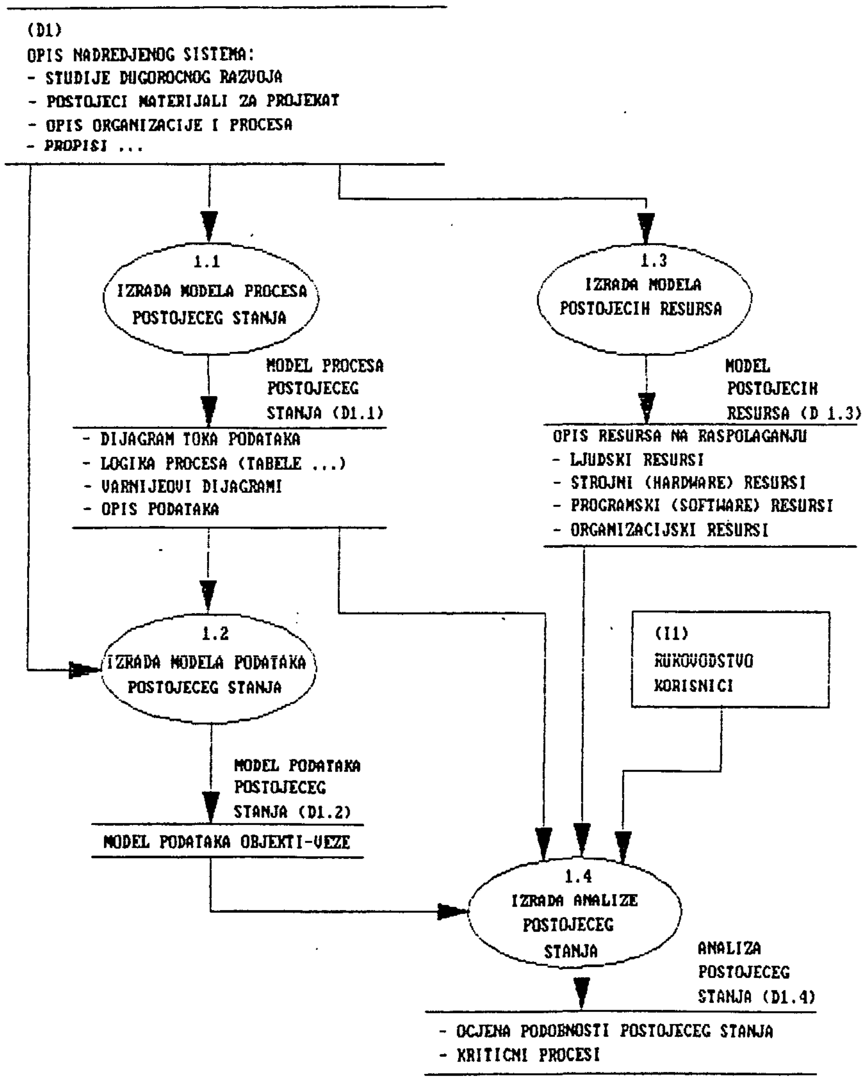
(NASTAVLJA SE)

(NASTAVAK SLIKE 3)



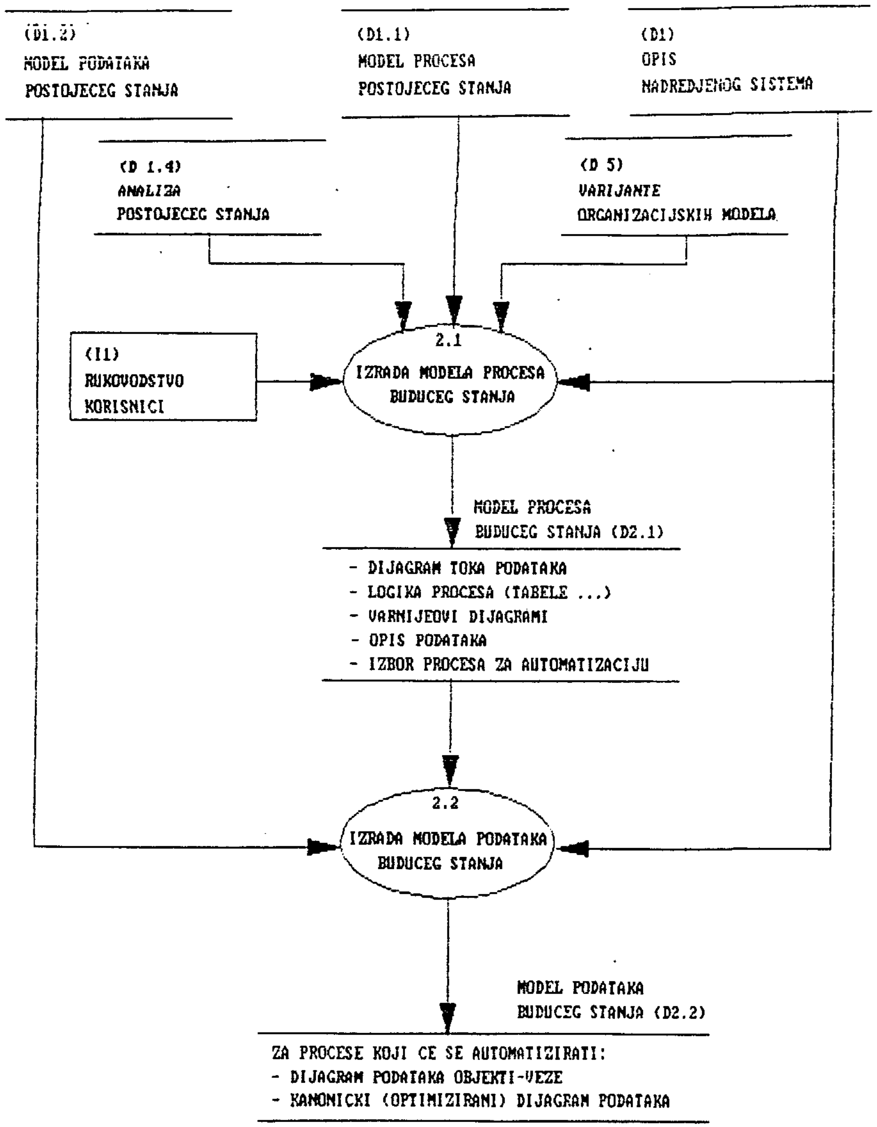
SLIKA 4. 1. OPIS POSTOJECEG STANJA

- DIJAGRAM TOKA PODATAKA (2. NIVO) -



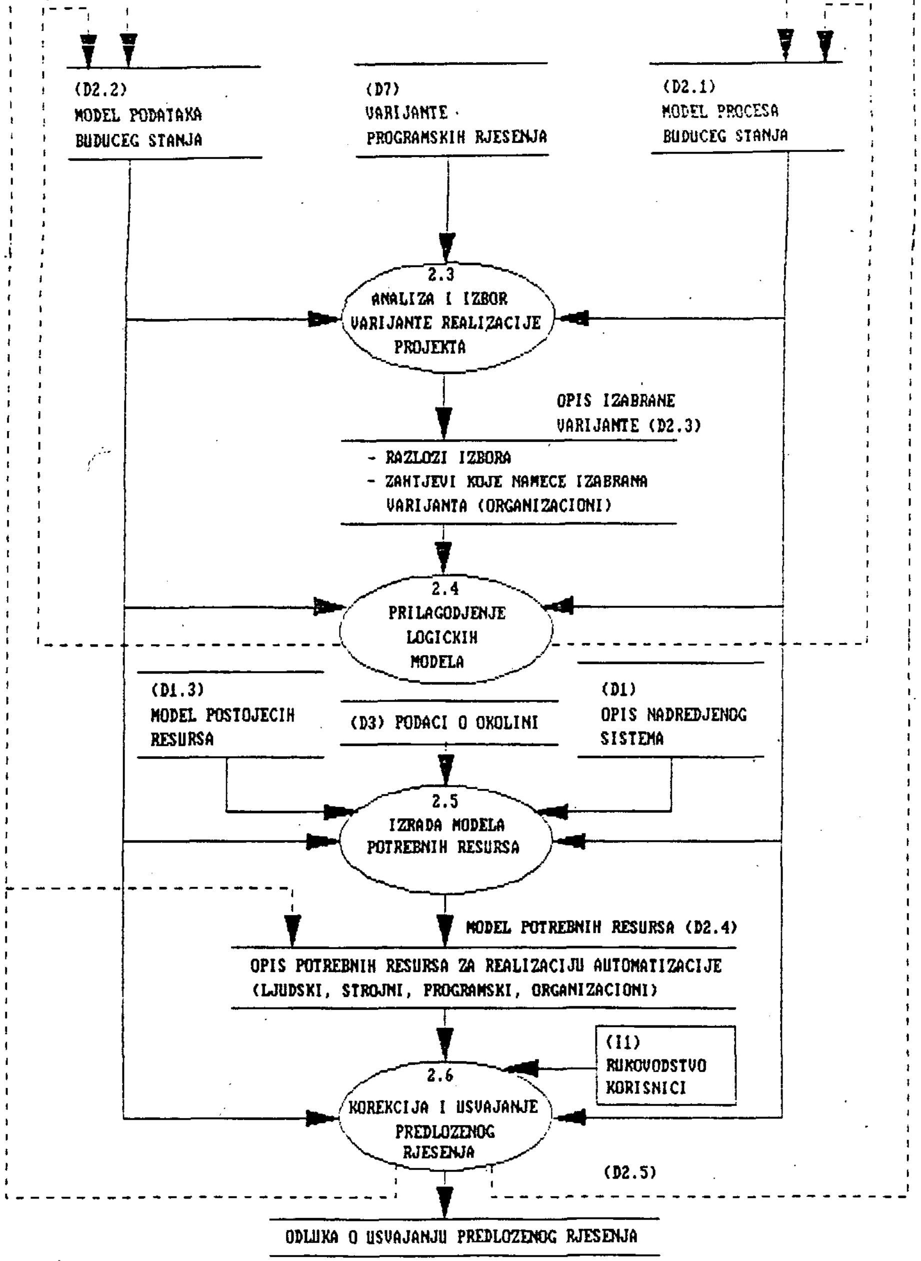
SLIKA 5. 2. OPIS BUDUCEG STANJA

- DIJAGRAM TOKA PODATAKA (2. NIVO) -



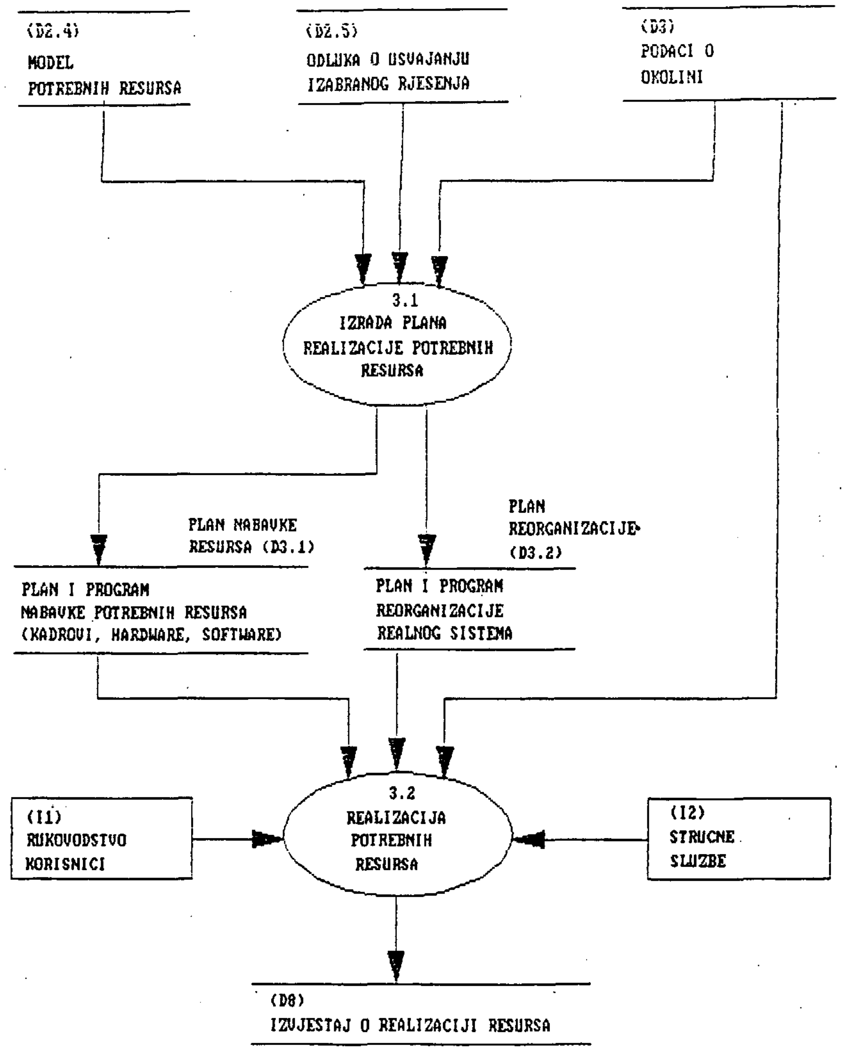
(NASTAVLJA SE)

(NASTAVAK SLIKE 5)

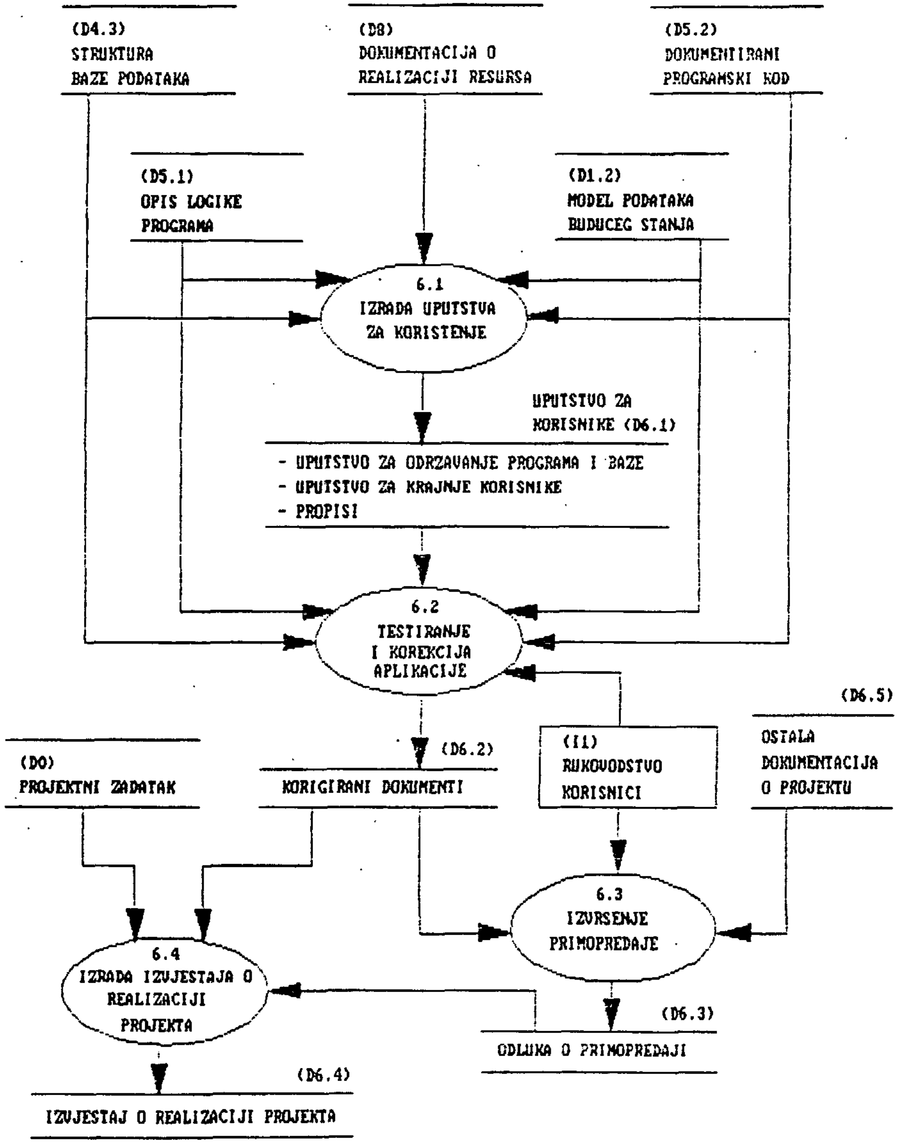

(D2.2) MODEL PODATAKA BUDUCEG STANJA
(D7) VARIJANTE PROGRAMSKIH RJESENJA
(D2.1) MODEL PROCESA BUDUCEG STANJA
2.3 ANALIZA I IZBOR VARIJANTE REALIZACIJE PROJEKTA
OPIS IZABRANE VARIJANTE (D2.3)
- RAZLOZI IZBORA
- ZAHTJEVI KOJE NAMECE IZABRANA VARIJANTA (ORGANIZACIONI)
2.4 PRILAGODJENJE LOGICKIH MODELA
(D1.3) MODEL POSTOJECIH RESURSA
(D3) PODACI O OKOLINI
(D1) OPIS NADREDJENOG SISTEMA
2.5 IZRADA MODELA POTREBNIH RESURSA
MODEL POTREBNIH RESURSA (D2.4)
OPIS POTREBNIH RESURSA ZA REALIZACIJU AUTOMATIZACIJE (LJUDSKI, STROJNI, PROGRAMSKI, ORGANIZACIONI)
(I1) RUKOVODSTVO KORISNICI
2.6 KOREKCIJA I USVAJANJE PREDLOZENOG RJESENJA
(D2.5)
ODLUKA O USVAJANJU PREDLOZENOG RJESENJA

SLIKA 6. 3. REALIZACIJA RESURSA

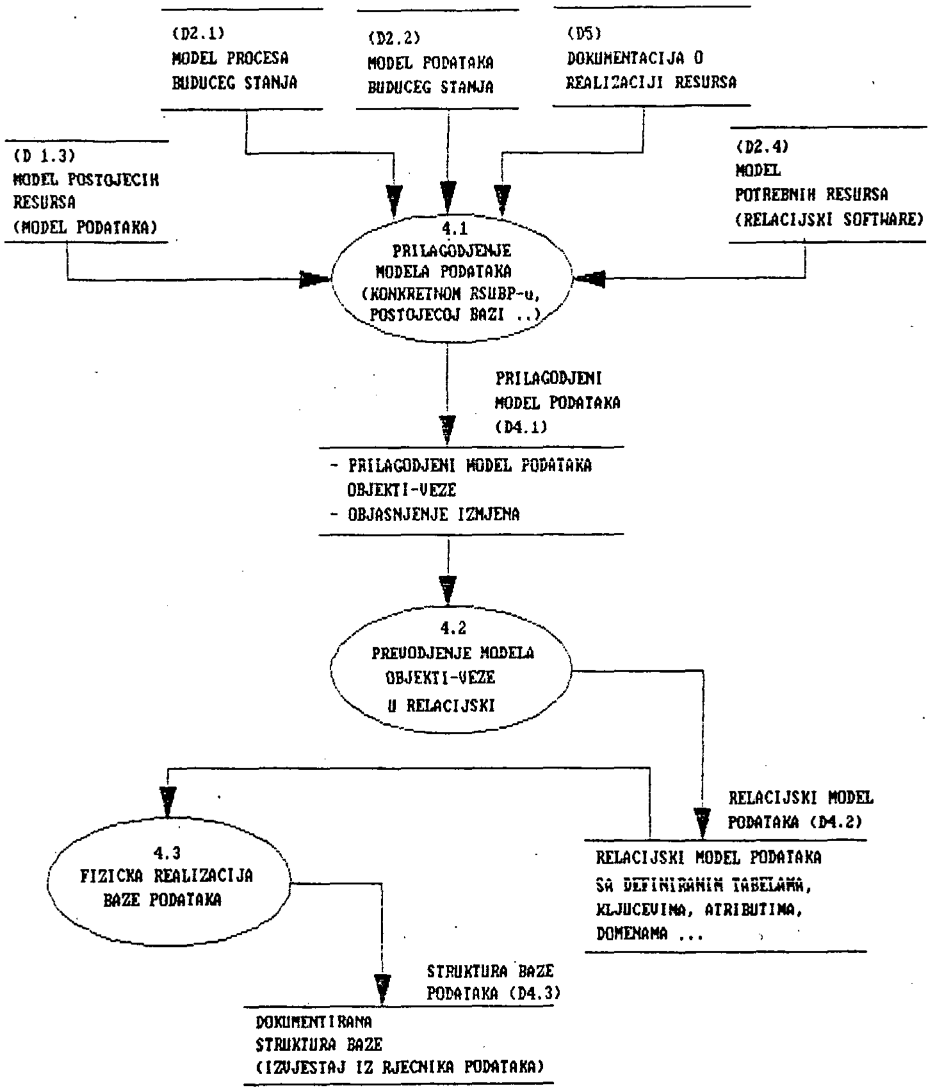
- DIJAGRAM TOKA PODATAKA (2. NIVO) -


(D2.4) MODEL POTREBNIH RESURSA
(D2.5) ODLUKA O USVAJANJU IZABRANOG RJESENJA
(D3) PODACI O OKOLINI
3.1 IZRADA PLANA REALIZACIJE POTREBNIH RESURSA
PLAN NABAVKE RESURSA (D3.1)
PLAN REORGANIZACIJE (D3.2)
PLAN I PROGRAM NABAVKE POTREBNIH RESURSA (KADROVI, HARDWARE, SOFTWARE)
PLAN I PROGRAM REORGANIZACIJE REALNOG SISTEMA
(I1) RUKOVODSTVO KORISNICI
3.2 REALIZACIJA POTREBNIH RESURSA
(I2) STRUCNE SLUZBE
(D8) IZVJESTAJ O REALIZACIJI RESURSA

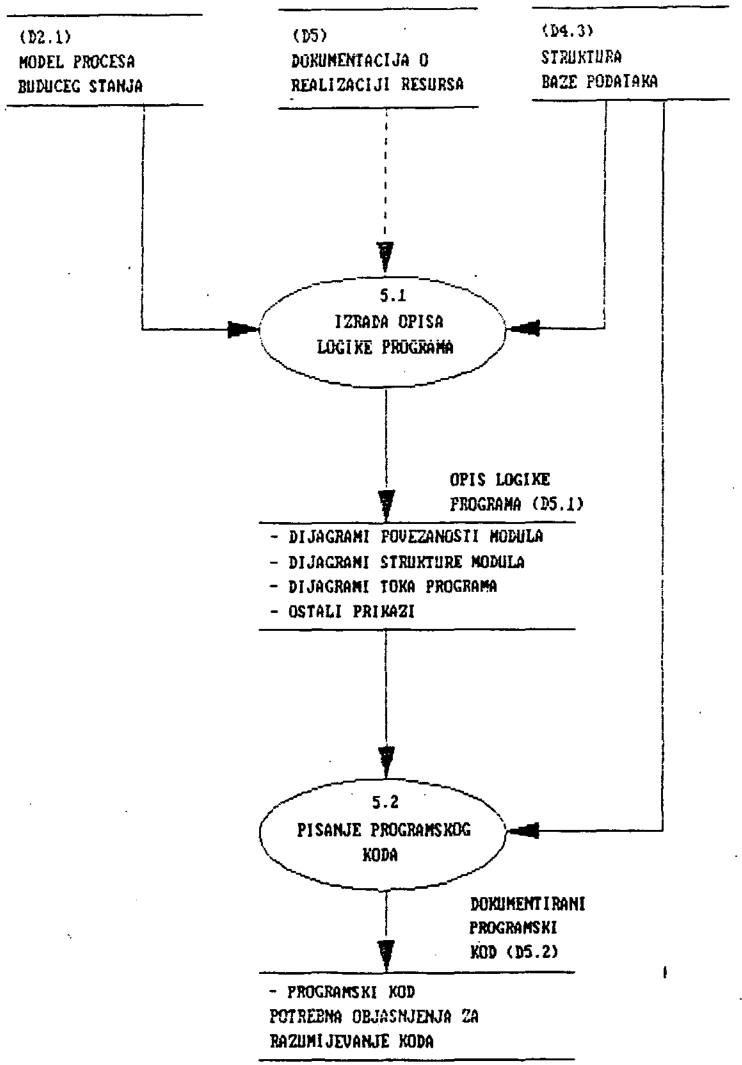
SLIKA 9. 6. IMPLEMENTACIJA I TESTIRANJE
- DIJAGRAM TOKA PODATAKA (2. NIVO) -



SLIKA 7. 4. FIZICKO PROJEKTIRANJE I REALIZACIJA BAZE PODATAKA
- DIJAGRAM TOKA PODATAKA (2. NIVO) -



SLIKA 8. 5. FIZICKO PROJEKTIRANJE I REALIZACIJA PROGRAMA
- DIJAGRAM TOKA PODATAKA (2. NIVO) -



POTPUNE DEFINIČIJE SINTAKSE NAREDBI PASCAL JEZIKA I POZIVA STANDARDNIH PROCEDURA READ I WRITE

INFORMATICA 1/90

Keywords: programming language, syntax, Pascal, semantic rules

Marjana Ivanović
Institut za matematiku dr Ilije Đuričića 4
21000 Novi Sad

ABSTRACT: Definisanje sintakse programskih jezika moguće je skoro u potpunosti izvršiti korišćenjem kontekstno-slobodnih gramatika. Međutim, ponekad je potrebno adekvatno izraziti određene semantičke zahteve. To je moguće uraditi uvođenjem kontekstno-zavisnih restrikcija unutar kontekstno-slobodne definicije sintakse jezika. Koristeći kontekstno-zavisne restrikcije potpuno je definisana sintaksa naredbi i poziva standardnih procedura Pascal jezika.

COMPLETE SYNTAX DEFINITIONS OF PASCAL STATEMENTS AND CALLS OF STANDARD PROCEDURES READ AND WRITE: Defining syntax of programming languages is almost entirely possible using context-free grammars. However, sometimes it is necessary to express various semantic rules more adequate. It is possible by introducing context-sensitive restrictions into context-free definition of the language. Using these restrictions syntax of Pascal statements and procedure calls of READ and WRITE is completely described.

1. UVOD

Definisanje sintakse programskih jezika moguće je, skoro u potpunosti, korišćenjem kontekstno-slobodnih gramatika. Naime, veći deo sintakse programskih jezika može se izraziti pravilima kontekstno-slobodnih gramatika. Međutim, primenom pravila kontekstno-slobodnih gramatika mogu se formirati sintaksno ispravne, ali semantički besmislene konstrukcije jezika.

Semantički neispravne konstrukcije, jasno, nemaju nikakvu funkciju u jeziku, te je potrebno onemogućiti njihovo formiranje. To se može uraditi jednom od sledećih metoda:

- odvojenim definisanjem semantike jezika,

- uvođenjem kontekstno-zavisnih restrikcija unutar kontekstno-slobodne definicije sintakse jezika.

Javlja se dilema koju metodu izabrati i na koji način isključiti mogućnost formiranja semantički neispravnih konstrukcija.

S jedne strane kontekstno-slobodne gramatike zauzimaju značajno mesto u definisanju i prevođenju programskih jezika zbog razvijenog formalizma i niza efikasnih algoritama sintaksne analize.

S druge strane još uvek nije razvijen dovoljno efikasan formalizam za izražavanje semantike jezika.

Stoga, je jednostavnije i pogodnije semantička pravila izraziti kontekstno-zavisnim restrikcijama (unutar kontekstno-slobodne definicije sintakse jezika).

Kontekstno-zavisne restrikcije se uglavnom odnose na kontrolu tipova promenljivih, kontrolu tipova i broja argumenata u pozivima standardnih procedura i sl.

Na primer, standardni način opisivanja izraza jezika Pascal [3] podržava sintaksno ispravne i sematnički besmislene izraze, jer ne vodi računa o semantičkim zahtevima jezika, tj. o tipovima operatora i operanada u izrazu.

Ovaj nedostatak moguće je izbeći definisanjem pravila gramatike tipova izraza [2] koja vodi računa o slaganju tipova operatora i operanada i razlikuje sledeće kategorije izraza:

```
<izraz> ::= <celi-izraz>         |
            <realni-izraz>       |
            <logički-izraz>      |
            <znakovni-izraz>     |
            <skalarni-izraz>     |
            <pokazivački-izraz>  |
            <niskovni-izraz>     |
            <skupovni-izraz>
```

Uvođenjem ove gramatike pruža se mogućnost za potpuno definisanje i nekih naredbi i poziva standardnih procedura jezika, sa specifikacijom odgovarajućih tipova.

U standardnoj gramatici jezika Pascal se za definisanje sintakse naredbe dodeljivanja, naredbi odabiranja (IF, CASE), naredbi ponavljanja (REPEAT, WHILE, FOR) i poziva standardnih procedura (READ i WRITE) koriste metalingvističke promenljive <identifikator>, <izraz>, <CASE-grana>, <promenljiva>, <početni-izraz>, <krajnji-izraz>, <i-izraz> pri čemu se ne vodi računa o njihovim tipovima.

2. NAREDBA DODELJIVANJA

Osnovna naredba jezika ja naredba dodeljivanja, čijim se izvršavanjem novoizračunata vrednost izraza pridružuje promenljivoj.

Sintaksa naredbe dodeljivanja u standardnoj gramatici Pascal jezika definiše se, pomoću Bekusove notacije, na sledeći način:

```
<naredba-dodeljivanja> ::=
                    <identifikator> := <izraz>

<identifikator> ::= <promenljiva>   |
                    <ime-funkcije>
```

Koristeći gramatiku tipova izraza, potpuna definicija sintakse naredbe dodeljivanja je:

```
<naredba-dodeljivanja> ::=
 <c-ident>   := <celi-izraz>                |
 <r-ident>  := ((realni-izraz> | <celi-izraz>)|
 <l-ident>   := <logički-izraz>             |
 <z-ident>   := <znakovni-izraz>            |
 <sl-ident> := <skalarni-izraz>             |
 <p-ident>   := <pokazivački-izraz>         |
 <sk-ident> := <skupovni-izraz>

 <sk-ident> ::= <sk-promenljiva>

 <p-ident>   ::= <p-promenljiva>    |
               <ime-funkcije-pokazivačkog-tipa>

 <sl-ident> ::= <sl-promenljiva>  |
                <ime-funkcije-skalarnog-tipa>

 <z-ident>   ::= <z-promenljiva>    |
                <ime-funkcije-znakovnog-tipa>

 <l-ident>   ::= <l-promenljiva>    |
                <ime-funkcije-logičkog-tipa>

 <r-ident>   ::= <r-promenljiva>    |
                <ime-funkcije-realnog-tipa>

 <c-ident>   ::= <c-promenljiva>    |
                <ime-funkcije-celog-tipa>
```

3. NAREDBE ODABIRANJA

IF-naredba uslovljava izvršavanje naredbe iza ključne reči THEN, ukoliko izraz koji sledi ključnu reč IF ima vrednost tačno. Ukoliko izraz ima vrednost netačno izvršava se naredba, ako postoji, iza ključne reči ELSE.

IF-naredba se, u standardnoj gramatici Pascal jezika definiše, pomoću Bekusove notacije na sledeći način:

```
<IF-naredba> ::= IF <izraz> THEN <naredba>
                            [ELSE <naredba>]
```

Koristeći gramatiku tipova izraza potpuna definicija sintakse IF-naredbe je:

```
<IF-naredba> ::=
   IF <logički-izraz> THEN <naredba>
                     [ELSE <naredba>]
```

CASE-naredba se sastoji od izraza-selektora i liste naredbi. Svaka naredba odgovara jednoj ili više konstantnih vrednosti koje su tipa selektora. Na osnovu tekuće vrednosti selektora određuje se naredba koja se izvršava. Nakon izvršenja selektovane naredbe kontrola izvrešenja se prenosi na kraj CASE-naredbe.

Sintaksa CASE-naredbe u standardnoj gramatici Pascal jezika se definiše na sledeći način:

```
<CASE-naredba> ::=
  CASE <izraz> OF
   <CASE-grana> {;<CASE-grana>}
  END

<CASE-grana> ::=
    <lista-CASE-obeležja> : <naredba>

<lista-CASE-obeležja> ::=
    <CASE-obeležje> {,<CASE-obele`je>}

<CASE-obeležje> ::= <konstanta>
```

Izraz je prostog tipa koji nije realni. Potpuna definicija sintakse CASE-naredbe je:

```
<CASE-naredba> ::=
     CASE <celi-izraz> OF
        <c-CASE-grana> {;<c-CASE-grana>} |
     CASE <logički-izraz> OF
        <l-CASE-grana> {;<l-CASE-grana>} |
     CASE <znakovni-izraz> OF
        <z-CASE-grana> {;<z-CASE-grana>} |
     CASE <skalarni-izraz> OF
        <sl-CASE-grana>{;<sl-CASE-grana>}

 <c-CASE-grana> ::=
   <c-konstanta> {,<c-konstanta>} : <naredba>

 <z-CASE-grana> ::=
     <z-konstanta> {,<z-konstanta>} : <naredba>

 <sl-CASE-grana> ::=
     <sl-konstanta>{,<sl-konstanta>}:<naredba>

 <l-CASE-grana> ::=
     <l-konstanta> {,<l-konstanta>} : <naredba>
```

4. NAREDBE PONAVLJANJA

Naredbe ponavljanja uslovljavaju izvršavanje naredbe sa unapred poznatim brojem ponavljanja (FOR naredba) i sa unapred neizvesnim brojem ponavljanja (WHILE-naredba i REPEAT-naredba).

Sintaksa WHILE i REPEAT naredbe se, u standardnoj gramatici Pascal jezika definiše na sledeći način:

```
<WHILE-naredba> ::=
    WHILE <izraz> DO <naredba>

<REPEAT-naredba> ::=
    REPEAT <naredba> {;<naredba>}
    UNTIL <izraz>
```

Izraz iza ključne reči WHILE je logički, izračunava se svaki put pre izvršavanja naredbe koja sledi iza ključne reči DO (naredba se ne izvršava ili se izvršava više puta). Ako je vrednost logičkog izraza tačna naredba se izvršava, inače se kontrola izvršenja prenosi na kraj WHILE-naredbe.

Izraz iza ključne reči REPEAT je logički, izračunava se posle izvršenja naredbe između ključnih reči REPEAT i UNTIL (naredba se izvršava bar jedanput). Izvršavanje naredbe se nastavlja sve dok logički izraz iza ključne reči UNTIL ne postane tačan.

Koristeći gramatiku tipova izraza Pascal jezika, potpune definicije naredbi WHILE i REPEAT je:

```
<WHILE-naredba> ::=
    WHILE <logički-izraz> DO <naredba>

<REPEAT-naredba> ::=
    REPEAT <naredba> {;<naredba>}
    UNTIL <logički-izraz>
```

U standardnoj gramatici Pascal jezika FOR-naredba se definiše na sledeći način:

```
<FOR-naredba> ::=
   FOR <promenljiva> := <poč-iz> <reč> <zav-iz>
   DO <naredba>

<poč-iz> ::= <izraz>

<reč> ::= TO | DOWNTO

<zav-iz> ::= <izraz>
```

FOR-naredba uslovljava izvršavanje naredbe koja sledi ključnu reč DO, povećavanjem kontrolne promenljive u opsegu (<poč-iz>, <zav-iz>). Početni i završni izraz se izvršavaju samo jednom pre početka izvršenja naredbe. Kontrolna promenljiva ne sme biti izmenjena naredbom iza ključne reči DO.

Kontrolna promenljiva, početni izraz i završni izraz su istog, prostog tipa, koji nije realni.

Koristeći gramatiku tipova izraza, sintaksa FOR naredbe se definiše na sledeći način:

```
<FOR-naredba> ::=
   FOR <c-promenljiva> :=
       <celi-izraz> <reč> <celi-izraz>
   DO <naredba>                                  |
   FOR <z-promenljiva> :=
       <znakovni-izraz> <reč> <znakovni-izraz>
   DO <naredba>                                  |
   FOR <l-promenljiva> :=
       <logički-izraz> <reč> <logički-izraz>
   DO <naredba>                                  |
   FOR <sl-promenljiva> :=
       <skalarni-izraz> <reč> <skalarni-izraz>
   DO <naredba>                                  |

<reč> ::= TO | DOWNTO
```

5. POZIV STANDARDNIH PROCEDURA READ I WRITE

Sintaksa poziva standardne procedure za ulaz podataka se, po standardnoj gramatici Pascal jezika, definiše na sledeći način:

```
READ[LN] ([<ime-u-teke>,] <lista-prom>);   |
READLN [(ime-u-teke>)];

<ime-u-teke> ::=
    <ime-standardne-ulazne-datoteke>

<lista-prom> ::=
    <promenljiva> {,<promenljiva>}
```

Procedura READ prihvata različit broj parametara. Parametri mogu da budu celog, realnog i znakovnog tipa. Ako je parametar celog ili realnog tipa READ prihvata niz karaktera koji formiraju označeni broj. Parametru se dodeljuje taj broj.

Koristeći gramatiku tipova izraza potpuna definicija liste parametara u pozivu standardne procedure READ je:

```
<lista-prom> ::=
  <ul-promenljiva> {,<ul-promenljiva>}

  <ul-promenljiva> ::=
           <c-promenljiva>        |
           <r-promenljiva>        |
           <z-promenljiva>
```

Sintaksa poziva procedure izlaza podataka je, po standardnoj gramatici Pascal jezika, definisana na sledeći način:

```
WRITE[LN] ([<ime-i-teke>,] <lista-i-izr>);  |
WRITELN [(<ime-i-teke>)];

<ime-i-teke> ::=
   <ime-standardne-izlazne-datoteke>

<lista-i-izr> ::=
   <i-izraz> {,<i-izraz>}

<i-izraz> ::=
   <i-izraz>[:<dužina-polja>[:<duž-dec-dela>]]

<dužina-polja> ::= <izraz>

<duž-dec-dela> ::= <izraz>
```

Izraz određuje vrednost koja se izdaje. Tip izraza je celi, realni, logički, znakovni ili niskovni. Dužina polja određuje broj znakova od kojih se sastoji vrednost koja se izdaje. Ako je dužina polja izostavljena podrazumeva se standardna vrednost. Dužina polja i dužina decimalnog dela su celi izrazi. Dužina decimalnog dela se isključivo navodi pri izdavanju realnih vrednosti.

Koristeći gramatiku tipova izraza, potpuna definicija izlaznog izraza poziva standardne procedure WRITE je:

```
<i-izraz> ::=
  <realni-izraz>  [:<duž-p>[:<duž-d-dela>]] |
  <ostali-izrazi> [:<duž-p>]

<ostali-izrazi> ::= <celi-izraz>        |
                    <logički-izraz>     |
                    <znakovni-izraz>    |
                    <niskovni-izraz>

<duž-p> ::= <celi-izraz>

<duž-d-dela> ::= <celi-izraz>
```

6. ZAKLJUČAK

Potpune definicije naredbi i poziva standardnih procedura u gramatici Pascal jezika koja vodi računa o tipovima operanada i operatora, su neznatno komplikovanije u poređenju sa standardnom gramatikom, ali su sistematičnije i pogodnije za učenje i primenu.

Literatura:

1. Alagić S., *Principi programiranja*, Svjetlost, Sarajevo, 1976.
2. Ivanović M., Budimac Z., *Gramatika tipova izraza Pascal jezika*, Informatika (sv. 3), Beograd, 1986.
3. Jensen K., Witrh N., *PASCAL user manual and report*, Springer-Verlag, New York Inc.,1985.
4. Stojković V., Tošić D., Stojmenović I., *Programski jezik Pascal*, Naučna knjiga, Beograd, 1985.
5. Tremblay J.P., Sorenson P.G., *The theory and practice of compiler writing*, McGraw-Hill, 1985.

SOME EXPERIENCES IN TEACHING THE PROGRAMMING PRACTICUM FOR UNDERGRADUATE AND GRADUATE STUDENTS

INFORMATICA 1/90

Keywords: programming practicum, teaching, graduate, undergraduate

Davor Bonačič
Faculty of Engineering, Maribor

ABSTRACT The article focuses on the difficulties arising during the programming practicum course for undergraduate and graduate students at the Faculty of Technical Science at the University of Maribor. Similar problems in computer science teaching are not uncommon even in recent years at numerous Universities abroad, also in the most highly developed countries. Measures employed to overcome this difficulties are described along with some cases from other countries.

1. INTRODUCTION

At the University of Maribor, Faculty of Technical Science, Department of Electrotechnics, Computing and Informatics section as well as at the Faculty of Electrotechnics in Ljubljana, the Programming Practicum course is taught to undergraduate and graduate students. The course takes place in the winter semester of the second year for the undergraduates and in the summer semester of the third year for the graduates, that is toward the end of their studies. It does not comprise any lectures and amounts to sixty hours' practical work on the computer.

2. CONTENT OF THE PROGRAMMING PRACTICUM

The aim of the course Programming Practicum is to enable students to get programming experience with larger and more complicated programs based on previously acquired programming skills. Among the proposed programming examples are the text editor, the simple bookkeepings and the data base system. Emphasized is placed on the correlation with other courses preceding Programming Practicum: Programming I, Forming and Developing of Programs, Algorithms and Data Structures.

Each student has a computer terminal at his disposal and has to make his program independently. The exam consists of presenting and defending the finished program before the lecturer.

3. EXPERIENCES WITH THE SUBJECT

3.1 Introduction

Already at the beginning of the first of Programming practicum course the following two things became clear:

- undergraduate and graduate students had different previous knowledge of programming
- neither group had the essential knowledge required

The most evident difference between both groups was that graduate students did not have the required theoretical and practical knowledge of the methods for developing and designing computer programs. The required subject matter forms a part of the course Modeling and Developing Programs which, for no know reason is in the curriculum of undergraduate students only and contains:

- strategical and tactical programming
- functional and procedural access to a problem
- specification requirements
- bottom-up and top-down designing
- program modularity
- structured design technics (HIPO, Jackson...)
- life cycle of the programming project
- verification, documentation and testing
- formal methods in program design

As a rule, students of both groups were not used to adhering to basic principles of software engineering. They all principally concentrated on writing code and their bad habits, like coding while designing or inserting comments last were firmly entrenched and difficult, if not impossible, to eliminate.

Also, their knowledge of the computer and the environment in which to develop their programs were not in the least adequate. The students, with rare exceptions, had very little knowledge of:

- the computer operating system, its commands and error messages
- programming tools (editor, debugger ...)
- using system libraries
- making their own libraries
- advanced capabilities and instructions of high level language in which they are programming (I/O procedures, error recovery, modularity and linking of the modules...)

3.2 How to carry out with the subject Programming Practicum

It was no use deploring what was wrong or had not been done before, the only possibility was to impart to the students in the condensed form the knowledge they were lacking, during Programming Practicum course itself, with the result that many hours intended for practical exercises on the computer were used up to that purpose.

In the next few years, the author of this article has been trying to transfer as much missing knowledge as possible to the students earlier time in their studies, during computer courses, preceding the Programming Practicum.

3.3 Experiences

In the next years there have been noticeable results of all these efforts to give the students more of the required skills and knowledge before the Programming Practicum course. But at the same time, it has become quite evident that the problem cannot be solved without more or less extensive changes in the teaching of computer programming - i. e. in the content and arrangement of most computer courses during the whole studies.

All the previously mentioned bad habits in programming are still manifest to a greater or lesser extent with nearly all the students. Especially the best of them seem to find it only a necessary nuisance to write specifications, comments and otherwise document their programs. "But to me the program is clear enough without all this", is the usual explanation of why they don't follow the rules for software engineering which they have been taught.

And so far the group of graduate students of course still has, as before, a wide gap in their knowledge concerning methods of developing and designing computer programs.

In the meantime, with increasing computer and terminal capacities, extensive computer exercises have been introduced in almost all other partly- or non-computering courses with very varying approaches to the subject of quality programming and software engineering. It is not a rare event that the students have had to make similar or the same programs in several different courses. A typical example is the text editor, which has become the most popular programming exercise in at least three different courses. All this has resulted in a waning of students' interests in their programming exercises. The Programming Practicum, comming toward the end of the studies when exams are in full swing, has become increasingly "popular" among the students.

3.4 Results

All the results of such largely inadequate teaching of computer programming become the most evident in the students' graduation theses; in more extensive programming works and projects, done by graduating students for the first time without any assistance. hat is when all their difficulties concerning specification, designing, testing and verification of their programs surface. A typical example is a student who at the time of defending his thesis still was not aware that it is possible to write programs in several modules in the programming language in which he has done his work. Or the another one who, when defending his thesis was most confused at the question about the reliability of the software he had produced.

4. CONCLUSIONS

If various curriculums and reports about computer science studies are studied (/1/, /2/, /3/), it becomes obvious that mostly all other learning institutions of that kind in our country and abroad have had to cope with more or less the same problems as the ones outlined in the present article. Other than that, there is a most significant basic difference in the distribution and at the same time in the content of the computer programming teaching between our faculties on the one hand and foreign curriculums on the other.

In the USA and in most other European countries a student of the undergraduate or graduate program in computer science learns at the beginning of his studies theoretically and practically all about quality programming and software engineering. After that he ought to be capable to implement this knowledge in all other subjects which do not deal directly with computer programming. Most universities abroad follow the recommended ACM curriculum in computer science or the IEEE computer engineering curriculum, but recently there have been a lot of reports and criticism that they are not quite up to date as well as a lot of efforts to change and modernize them (/2/,/3/,/4/,/5/,/7/ ...). Both curriculums introduce two basic courses in programming methods for computer science students, named CS1 and CS2. As an example let me list the objectives of these two courses as stated in the reports of Elliot B. Koffman of Temple University in Philadelphia in 1984 (/8/, /9/):

Objectives of CS1

- To introduce a disciplined approach to problem solving methods and algorithm development.
- To introduce procedural and data abstraction.
- To teach program design, coding, debugging, testing, and documentation using good programming style.
- To teach a block-structured high-level language.
- To provide a familiarity with the evolution of computer hardware and software technology.
- To provide a foundation for further studies in computer science.

Objectives of CS2

- To continue developing a disciplined approach to design, coding, and testing of programs.
- To teach the use of data abstraction using as examples data structures other than those normally provided as basic data types in current programming languages; for example, linked lists, stacks, queues, and trees.
- To provide an understanding of the different implementations of these data structures.
- To introduce searching and sorting algorithms and their analysis.
- To provide a foundation for further studies in computer science.

Studying computer programming along the listed guidelines eliminates most of the problems, discussed in this article. However, there has been a number of the recent reports from other developed countries about similar or the same difficulties in computer programming teaching as at our universities.

Let us mention just some of them.

From the University of Minnesota comes a report about chronically bad habits of their students at computer programming (/4/). The situation did not improve till 1988 when it was finally required all along through both courses CS1 and CS2 to use strict documentation, methodical step-wise development of the programs in high-level pseudocode, structure diagrams with data flow, detailed test descriptions, testing results, and a description of all known. bugs.

At the University of Texas at Austin (/5/) they try to teach students quite at the beginning quality programming along with the methods for the design and development of computer programs and introduce a lot of project and team work together with other group activities in their software engineering exercises.

The department of Computer Science University of Virginia (/6/) reports on a new approach to the teaching of introductory computer programming and computer science based on the teaching of skills in program analysis along with the traditional emphasis on skills in ' program synthesis with the new element of program state introduced as the key element in understanding programs.

From all these reports it follows that problems in teaching quality programming are not at all limited to the University of Maribor. And, taking into account the given respect to different conditions in comparison with leading countries in computer science, similar approaches how to cope with the problems involved have been tried in recent years.

REFERENCES:

/1/ - CURRICULUM '78, Recommendations for the Undergraduate Program in Computer Science, Communications of ACM, March 1979, Vol.22, No.3, p.147-164

/2/ - Anton P. Železnikar: UNIVERZITETNI POUK RACUNALNISTVA I, II, Ljubljana, Informatuca 2/198Ø, str.32-42 in 3/198Ø, str.67-88

/3/ - Mary Shaw: THE CARNEGIE-MELLON CURRICULUM FOR THE UNDERGRADUATE COMPUTER SCIENCE, Springer-Verlag New York, Berlin, Heidelberg, Tokyo, 1985

/4/ - Linda L. Deneen, Keith R. Pierce: DEVELOPMENT AND DOCUMENTATION OF COMPUTER PROGRAMS IN UNDERGRADUATE COMPUTER SCIENCE COURSES, ACM Ø-89791-256-X/88/ØØØ2/ØØ17 p.17-21

/5/ - Laurie Honour Werth: INTEGRATING SOFTWARE ENGINEERING INTO AN INTERMEDIATE PROGRAMMING CLASS, ACM Ø-89791-256-X/88/ ØØØ2/ØØ54 p.54-58

/6/ - Terence W. Pratt: TEACHING PROGRAMMING: A NEW APPROACH BASED ON ANALYSIS SKILLS, ACM Ø-89791-256-X/88/ØØØ2/Ø249 p.249-253

/7/ - Dale A. Brown: REQUIRING CS1 STUDENTS TO WRITE REQUIREMENTS SPECIFICATIONS: A RATIONALE, IMPLEMENTATION SUGGESTIONS, AND A CASE STUDY, ACM Ø-89791-256-X/88/ØØØ2/ ØØ13 p. 13-16

/8/ - Koffman, E. B., Miller, P. L., and Wardle, C. E.: RECOMMENDED CURRICULUM FOR CS1, 1984, Communications of ACM 27, 1Ø, p. 998-1ØØ1.

/9/ - Koffman, E. B., Stemple D., and Wardle, C. E.: RECOMMENDED CURRICULUM FOR CS2, 1984, Communications of ACM 28, 8, p. 815-818.

MOŽNOST IMPLEMENTACIJE ZANESLJIVE BAZE PODATKOV V MS-DOS OKOLJU

INFORMATICA 1/90

Keywords: data base, reliability, MS-DOS environment

Ivan Pepelnjak ml., Jernej Virant,
Nikolaj Zimic
Fakulteta za elektrotehniko in računalništvo v Ljubljani

Članek predstavlja možnosti implementacije zanesljive baze podatkov v MS-DOS okolju. Predstavljeni so problemi, na katere naletimo ob implementaciji zanesljive baze podatkov s poudarkom na operaciji povrnitve (recovery). Opisane so znane strojne in programske rešitve tega problema in programski produkti za podporo baz podatkov v MS-DOS okolju skupaj z oceno njihove zanesljivosti in možnosti programskega nadzora transakcij. Predstavljen je sistem za vodenje dnevnika povrnitve in potrebne operacije za implementacijo povrnitve v okviru baze podatkov ter ob restartu sistema. Definiran je pojem podtransakcije in prikazana uporabnost takšnega koncepta v operacijskem sistemu, ki načelno ni transakcijsko orientiran.

The paper presents a reliable data base implementation in MS-DOS environment. The problems encountered when implementing reliable data base are discussed along with existing software and hardware solutions in general and in MS-DOS environment. Existing software products under MS-DOS are evaluated according to level of reliability they provide. A complete recovery system for a one - transaction data base is presented. We define the idea of subtransaction and its function in parent transaction together with useful hints on its usage in non transaction oriented operating system.

UVOD

Jasno je, da noben računalniški sistem ne deluje ves čas popolnoma zanesljivo. Ta enostavna ugotovitev pa ima daljnosežne posledice za konstrukcijo samih računalniških sistemov in predvsem baz podatkov, ki jih uporabljamo v računalniških sistemih.[1] Baza podatkov mora namreč v vsakem trenutku vsebovati veljavne in zanesljive informacije, ki jih lahko aplikacije uporabljajo brez dvoma v njihovo veljavnost oz. pravilnost. V članku se ne bi ukvarjali z problemi hkratnega dostopa in popravljanja baze podatkov s strani več programov, kar lahko v slabo implementiranem sistemu za delo z bazo podatkov hitro privede do nepravilnosti podatkov v sami bazi, bolj nas bo zanimala odpornost baze podatkov na nezaneslji vost delovanja računalniškega sistema. Definirajmo najprej zanesljivost sistema za delo z bazo podatkov:

Definicija :

> Sistem za delo z bazo podatkov je zanesljiv, če lahko po vsaki napaki strojne ali programske opreme obnovi podatke v bazi podatkov v stanje, za katerega se ve, da je pravilno.

Zanesljivost delovanja računalniškega sistema lahko dosežemo npr. z redundanco v strojni opremi. Za popolnoma zanesljivo delovanje računalniškega sistema bi morali podvojiti vse njegove elemente (procesno enoto, kanale oz. vodila, diskovne enote, napajalnike), a kljub temu bi obstajala verjetnost, da bi ob tako veliki redundanci prišlo do napake pri delovanju sistema. Poleg tega je takšna redundanca draga, navadno se je ne da optimalno izkoristiti (predvsem diskovnih enot, kjer moramo skrbeti za popolno kopijo podatkov na dveh diskovnih enotah, drugače je pri procesnih enotah, ki lahko opravljajo različna opravila v času, ko vse pravilno delujejo). Ob reševanju zanesljivosti sistema za delo z bazo podatkov se torej ne moremo opreti na aparaturne redundance, problem bo potrebno rešiti s programsko opremo.[2]

Drug element, ki nas navaja na rešitev problema s programsko opremo, je problem zanesljivosti samih aplikacij. Vsaka aplikacija ima lahko v sebi skrite programske napake, ki privedejo tudi do nenormalnega konca delovanja programa. V takšnem primeru program navadno zapusti bazo podatkov v neveljavnem stanju in potrebna je akcija s strani sistema za delo z bazo podatkov, ki ponovno spravi podatke v bazi podatkov v veljavno stanje.

Ne nazadnje bi želeli v aplikacijskem programu (posebno interaktivnem) možnost za lep izhod iz „zagat" [3], ki nam jih pripravi uporabnik med vnosom. Želeli bi si torej programsko možnost, da sredi dela obupamo in uničimo vse, kar smo doslej opravili.

Preden se resneje lotimo obravnavanja problema si definirajmo pojem *transakcije* :

> Transakcija je najmanjša, nedeljiva enota dela z bazo podatkov. Vsaka transakcija (oz. spremembe v bazi podatkov, ki jih je aplikacijski program izvršil v okviru ene transakcije) se izvrši v celoti ali pa se sploh ne izvrši.

Za reševanju zgoraj navedenih zahtev bomo imeli v aplikacijskih programih na razpolago tri dodatne funkcije sistema za delo z bazo podatkov :

(1) V članku bomo pod pojmom „baza podatkov", ki ima sicer lahko mnogo pomenov, smatrali skupek datotek, ki vsebujejo podatke, kot sistem za delo z bazo podatkov (*database system*) pa programsko opremo, ki skrbi za dostop aplikacijskih programov do baze podatkov na disku. Ta programska oprema je lahko zbrana v operacijskem sistemu ali v aplikaciji kot poseben proces, program ali kot skupek podprogramov v aplikacijskem programu

- BEGIN_TRANSACTION začne transakcijo. Vsi dostopi do datotek od tega klica dalje so knjiženi kot enotna transakcija, ki lahko vsa uspe ali pa je sploh ni bilo.

- COMMIT klic potrdi transakcijo. Vse spremembe, ki jih je transakcija izvršila se zapišejo v datoteke baze podatkov.

- ROLLBACK klic prekine delo, povrnitveni modul v sistemu za delo z bazo podatkov uniči vse spremembe, ki jih je transakcija izvršila, s stališča podatkov v datotekah se transakcija ni začela niti izvajati.[4]

Obstoječi sistemi za delo z bazami podatkov na velikih računalnikih rešujejo zgoraj navedeno problematiko s standardnimi postopki za vodenje dnevnika transakcij (*transaction logging*) in dnevnika posegov v podatke (*recovery log*). Na podlagi enega ali drugega dnevnika lahko operacija povrnitve (*recovery*) ob nenormalnem koncu delovanja posameznega programa ali izpada celotnega sistema obnovi stanje baze podatkov v znano pravilno stanje.

Za razliko od okolja velikih računalnikov, kjer so sistemi za delo z bazami podatkov standarden del sistemske programske opreme, je stanje v mikroračunalniškem okolju milo rečeno katastrofalno. Niti eden od mikroračunalniških sistemov, ki so v svetu široko razširjeni (CP/M, MS-DOS oz. PC-DOS, PC-MOS, XENIX, celo nekatere verzije UNIX operacijskega sistema), nima rešenega problema zanesljivosti dostopa do baze podatkov v sistemski programski opremi. Tako je reševanje tega problema prepuščeno posameznim aplikacijam, programskim jezikom oz. sistemom za delo s podatkovno bazo v posameznem programskem jeziku [5].

Tudi na aplikativnem področju je stanje katastrofalno. Niti eden od standardnih paketov, ki naj bi omogočal delo z bazo podatkov (oz. z indeksiranimi datotekami, kar je najboljši približek za bazo podatkov v MS-DOS okolju) pa naj bo to Turbo Pascal Database Toolbox (Borland), dBASE III plus (Ashton Tate), Clipper (Nantucket), Oracle (Oracle), Paradox (Borland), kakor tudi takoimenovani jeziki četrte generacije (Focus, Magic II ipd.) ne ponujajo zadovoljive rešitve tega problema. Različne verzije dBASE programa in Clipper prevajalnika sploh ne poznajo transakcij. Paradox in Oracle poznata transakcije, povrnitev pa ni zadovoljivo rešena. Magic II ima (po zatrdilih proizvajalca) povrnitev implementirano, nima pa možnosti programskega ROLLBACK-a. Edina svetla izjema je Novell s svojim Advanced Netware operacijskim sistemom, ki v določenih konfiguracijah podpira tudi vodenje transakcij in povrnitve v primeru izpada posameznega vozlišča v mreži ali posameznega programa. Seveda je tudi tukaj uporaba teh možnosti in implementacija zadovoljive baze podatkov odvisna od aplikacije.

Pričakovati bi bilo, da bo takšno stanje skrbelo proizvajalce programske opreme ter da bodo poskušali to pomanjkljivost bolj ali manj uspešno odpraviti. Žal se izkaže, da je gonja za hitrim in enostavnim zaslužkom tudi tu opravila svoje, večina "specialistov" za programiranje aplikacij v mikroračunalniškem okolju na problem zanesljivosti delovanja računalniškega sistema kar pozabi oz. odpravi tečnega uporabnika, ki ga vendarle skrbi varnost njegovih podatkov s "standardnim" nasvetom : "Skrbite za arhiviranje podatkov, če bi se že slučajno kaj pripetilo, kar je pri naših programih skoraj nemogoče, lahko restavrirate podatke z diskete".[6] Izkaže se, da takšen pristop skoraj zanesljivo privede do frustriranih uporabnikov, ki so zadnje arhiviranje opravili pred tednom dni, in do njihovega razumljivega besa, ko morajo ponovno vnašati vse transakcije zadnjega tedna. Rezultat takšnega odnosa razvijalcev aplikacij do kupcev ne nazadnje vodi do nezaupanja v mikroračunalniške sisteme in do nezanesljivih implementacij informacijskih sistemov [7].

Možnost implementacije zanesljivega sistema za delo z bazo podatkov v MS-DOS okolju

Zanesljiv sistem za delo z bazo podatkov je, kot smo ugotovili v predhodnem razdelku, sistem, ki v vsakem primeru (izpad programa ali računalnika) ohrani v datotekah baze podatkov zanesljivo kopijo podatkov. Zanesljivost baze podatkov dosežemo v dveh stopnjah

- Z rednim in avtomatskim arhiviranjem podatkov na več kompletov (spet avtomatsko določenih) disket dosežemo pravočasno arhiviranje podatkov in možnost vzpostavitve delujočega stanja tudi v primeru okvare strojne opreme (trdi disk) ali v primeru fatalne in nepopravljive napake v samem sistemu za vodenje baze podatkov.

- Z vodenjem dnevnika dostopov na disk oz. dnevnika transakcij, ki omogoča, da popravimo bazo podatkov v veljavno stanje pred izpadom programa oz. nenormalnim koncem izvajanja programa. Spet je zaželjeno, da je povrnitev iz dnevnika avtomatska in da se izvede takoj po ponovnem vklopu sistema oz. takoj po nenormalnem koncu izvajanja programa

V probleme arhiviranja podatkov in pravočasnega zagona programa za povrnitev baze podatkov v pravilno stanje se sedaj ne bi spuščali, zanima nas le implementacija sistema za delo z bazo podatkov, ki vodi dnevnik svojega dela in postopki, ki omogočajo, da z recovery programom popravimo bazo podatkov v veljavno stanje.

Zahteve za vodenje dnevnika v sistemu za delo z bazo podatkov

MS-DOS operacijski sistem in samo mikroračunalniško okolje postavlja za sistem za delo z bazo podatkov specifične zahteve in omejitve, ki olajšajo oz. otežkočajo pisanje programske opreme za sistem, ki naj bi bil kolikor toliko zanesljiv. Te zahteve in omejitve lahko združimo v naslednjih nekaj točk

- V sistemu je naenkrat aktivna samo ena transakcija

- Ker so aplikacijski programi navadno usmerjeni programsko in ne transakcijsko (čim večji programi, ki v sebi združujejo čimveč funkcij), je dobro definirati tudi podtransakcije, ki se navzven obnašajo ravno tako kot celotna transakcija [8] (za definicijo glej dalje v članku).

(2) Tipičen primer, da lahko z aparaturnostjo učinkovito povečamo zanesljivost baz podatkov je stroj baze podatkov (*data base machine*), vendar je to preveč specifičen in predrag prijem, da bi se v našem primeru odločili zanj.

(3) Primer zagate : uporabnik si sredi funkcije, ko je nekaj podatkov že knjiženih, premisli in hoče uničiti vse kar je dotlej opravil.

(4) Pripomniti velja, da se ob prvem dostopu do baze podatkov navadno izvrši implicitni BEGIN TRANSACTION, ob normalnem koncu programa implicitni COMMIT in ob nenormalnem koncu programa implicitni ROLLBACK.

(5) Tako ima npr. jezik C kar več izvedenk sistema za dostop do indeksiranih datotek, kar seveda še ni prava baza podatkov, vendar je največ, kar lahko s C jezikom v MS-DOS okolju dobimo

(6) Poudariti moramo, da obstajajo tudi v našem okolju aplikativne skupine, ki resno razvijajo aplikacije vsaj z najboljšo dostopno tehnologijo (Paradox, Oracle), večina aplikacij je še vedno pisana v dBASE III in Clipperju, ki ne nudita prav nobene zaščite.

(7) Da niti ne omenjamo danes vse bolj prisotnega problema računalniških virusov, ki se ravno tako pojavljajo predvsem v MS-DOS okolju, večinoma zaradi neresnega pristopa uporabnikov do računalnika, ki ponekod meji že na pravi računalniški kriminal.

- Sistem za delo z bazo podatkov mora o spremembah voditi čim manjši dnevnik, da ne bi po nepotrebnem tratili prostora na diskih

- Zaradi relativno počasne diskovne opreme mora sistem za delo z bazo podatkov optimalno zapisovati podatke na disk, da se prepreči preveliko premikanje bŕalno / pisalne glave po disku.

- Operacijski sistem močno omejuje število hkrati odprtih datotek (v verzijah do 3.30 samo največ 20 odprtih datotek v posameznem programu, kar je za večjo bazo podatkov zelo huda omejitev).

- Operacijski sistem nima nobene definirane poti, kako zapisuje podatke nam disk. Sistem za delo z bazo podatkov mora kljub vsemu ohraniti načelo, da morajo biti informacije v dnevniku transakcij dostopne programu za izvedbo povrnitve preden zapišemo spremenjene podatke na disk.

Od vseh zgoraj naštetih točk nam je v korist samo prva točka - obstoj ene same transakcije. Le - ta nam omogoča vodenje enostavnega dnevnika transakcije in enostavnejše povrnitvene operacije.

Osnoven sistem za delo z bazo podatkov

Za osnovo našega sistema smo vzeli Turbo Pascal Database Toolbox, ki rešuje probleme dostopa do indeksiranih datotek, vendar nima vgrajene možnosti povrnitve. Z izbiro tega sistema smo se izognili delu, ki z našim namenom v bistvu nima veliko skupnega - implementacija B* drevesne strukture, delujoč programski paket pa nam je poleg tega omogočil še, da smo se osredotočili samo na zagotovitev zanesljivosti delovanja podatkovne baze.

Turbo Pascal Database Toolbox (v nadaljevanju Database) ima implementirani dve vrsti datotek :

- Podatkovne datoteke so v bistvu datoteke z relativnim dostopom, zapisi s stalno dolžino, dostop do zapisov imamo po številki zapisa. Database vodi za takšne datoteke tudi dodatne atribute, ki jih navadne datoteke z relativnim dostopom nimajo, tako lahko v podatkovni datoteki dodajamo in brišemo zapise, Database pa sam skrbi za optimalno izkoriščenost podatkovne datoteke (novi zapisi izkoriščajo prostor, ki so ga nekoč zasedali zbrisani zapisi)

- Indeksne datoteke predstavljajo povezavo med podatkovnimi datotekami in ključi. Indeksna datoteka je po strukturi B* drevo, ki ga uporabnik v aplikaciji uporablja kot imenik : za dano vrednost ključa mu imenik vrne številko zapisa v podatkovni datoteki, ki mu ustreza dana vrednost ključa. Ob dodajanju zapisa je postopek obraten - najprej dodamo zapis v podatkovno datoteko, dobimo številko novega zapisa in v imenik vnesemo zapis z željeno vrednostjo ključa in novo dobljeno številko zapisa. Podobno je implementirano tudi brisanje ključa.

Globoko znotraj Database Toolbox programskega paketa so tudi indeksne datoteke implementirane kot podatkovne datoteke, kar nam prihrani eno stopnjo varovanja - poskrbeti moramo samo za zanesljiv pristop do podatkovnih datotek.

Implementacija zanesljivega sistema za delo z bazo podatkov

Ugotovili smo torej, da moramo za implementacijo zanesljivega sistema za delo z bazo podatkov v MS-DOS okolju ob uporabi Database Toolbox ISAM datotek realizirati zanesljiv pristop do datotek z relativnim dostopom. Ker je v sistemu hkrati aktivna samo ena transakcija, smo se odločili za najenostavnejši sistem vodenja dnevnika dela, za tim. senčenje (*shadowing*), ki temelji na zelo enostavnem konceptu :

- Za vsak zapis, ki ga zapišemo v bazo podatkov, mora biti v dnevniku zapis s predhodno vrednostjo tega zapisa v bazi.

Ta koncept je zelo enostaven za implementacijo, prav tako je načelno dokaj enostaven program za povrnitev baze podatkov : program čita dnevnik v obratnem vrstnem redu in popravlja vrednosti zapisov v bazi. Ko pride program do začetka dnevnika, je stanje podatkov v datotekah veljavno.

Opozoriti velja, da ta koncept, čeprav enostaven, zadostuje tudi zelo trdim zahtevam, ki jim mora ustreči program za povrnitev baze podatkov - med samim delom programa lahko pride do ponovnega izpada računalniškega sistema in dvojna ali celo večkratna delna ali popolna povrnitev z istim dnevnikom mora na koncu privesti do veljavnega stanja podatkov v datotekah.

Vodenje podatkov, potrebnih za delovanje povrnitvenega programa

Sistem za delo z bazo podatkov mora torej ob vsakem pisanju na disk najprej v dnevnik zapisati predhodno vrednost zapisa, ki se bo zapisal na disk. Poleg tega mora sistem za delo z bazo podatkov v dnevnik zapisati tudi imena vseh datotek, ki jih odpre aplikacijski program ter dodatne atribute, ki jih vodi Database Toolbox (lista zbrisanih zapisov in število zbrisanih zapisov v celotni datoteki) ob vsaki spremembi teh atributov (brisanje in dodajanje novega zapisa).

Zaradi hitrega delovanja povrnitvenega programa zahtevamo, da mora vsak zapis v dnevniku vsebovati kazalec na predhodni zapis v dnevniku in kazalec na predhodni zapis za isto datoteko, kar omogoča selektivno povrnitev samo nekaterih datotek. Zaradi različne dolžine zapisov v posameznih datotekah, mora glava zapisa v dnevniku vsebovati tudi dolžino podatkovnega dela trenutnega zapisa. Za kontrolo pravilnosti zapisov v dnevniku mora vsak zapis vsebovati checksum ali magično številko (kombinacija bitov, ki se v podatkovnem delu zelo redko ponovi, njena prisotnost naj z veliko verjetnostjo potrdi pravilnost zapisa). Magična številka ali checksum se mora nahajati na koncu glave in podatkovnega zapisa, da lahko povrnitveni program preveri veljavnost celotnega zapisa. Omenjene zahteve nas privedejo do naslednje definicije zapisa v dnevniku :

Imena posameznih polj so dovolj opisna, polje Trans_ID je namenjeno za kasnejše razširitve z več hkrati aktivnimi transakcijami (PC-MOS ali Novell implementacija), SHD_Length je skupna dolžina glave in podatkovnega dela in služi za hitrejše delo povrnitvenega programa v prvem delu, ko mora čim hitreje priti do konca dnevnika. File_ID je interna številka Database Toolbox-a, ki služi za identifikacijo podatkovne datoteke v internih tabelah.

Zapisi atributov posamezne datoteke oz. informacije o odpiranju datoteke uporabljajo enak format, le da je v prvem primeru RECORD_ID = 0, v drugem pa RECORD_ID = -1. Podatkovni del so v takšnih zapisih atributi datotek oz. polno ime datoteke.

Zanesljivo vodenje dnevnika na disku

Če hočemo, da bo povrnitveni program zanesljivo opravil svoje delo neodvisno od trenutka, ko pride do izpada računalniškega sistema [9], moramo zagotoviti, da bodo vse informacije o predhodni vrednosti zapisa gotovo dostopne preden zapišemo novo vrednost zapisa na disk.

(8) tako kot transakcija je tudi podtransakcija izvršena v celoti ali pa ni izvršena. Razmerje podtransakcije do nadrejene transakcije je opisano kasneje v članku.

```
Type SHD_Header = Record
                    SHD_Length   : Integer ;
                    Prev_Ptr     : LongInt ;
                    Trans_ID     : Integer ;
                    Back_Ptr     : LongInt ;
                    File_ID      : Integer ;
                    Record_ID    : LongInt ;
                    Record_Len   : Integer ;
                    HDR_Magic    : LongInt ;
                  End ;
```

Definicija glave zapisa v dnevniku

MS-DOS operacijski sistem hrani informacije o posamezni datoteki na več mestih :

- V FAT tabeli hrani informacijo o lokaciji posameznih sektorjev, ki sestavljajo datoteko.
- V imeniku (directory) hrani velikost datoteke in prvi vnos v FAT tabeli, kar nam omogoča, da preko verižne strukture najdemo naslednje vnose.
- Kjerkoli na disku hrani podatke te datoteke.

Preden zapišemo podatke posameznega zapisa na disk, moramo torej zapisati vse podatke o stari vrednosti tega zapisa v dnevnik ter po potrebi (če se je velikost dnevnika povečala) še v FAT in imenik.

O načinu, kako MS-DOS piše podatke na disk, ni znano prav mnogo. Dostopna dokumentacija navaja le sledeče podatke :

- Ob zapiranju datoteke se zapišejo vse informacije o tej datoteki na disk. V verziji 3.30 nam je na razpolago tudi COMMIT FILE funkcija, ki izvede iste operacije kot CLOSE, vendar ne zapre datoteke.

Trivialna implementacija dnevnika

Najenostavnejša implementacija dnevnika je sledeča : ob vsakem pisanju v datoteke baze podatkov se predhodna vrednost zapisa zapiše v dnevnik in naredi se CLOSE oz. COMMIT dnevnika. V tej implementaciji se ob vsakem zapisu v katerokoli datoteko glava trdega diska premakne na začetek diska (zapisovanje FAT tabele), na imenik (zapisovanje nove velikosti datoteke), na zapis v dnevniku in končno na zapis v datoteki. Jasno je, da je takšna implementacija strahovito počasna.

Optimizacija 1 : Uporaba glavnega imenika

En premik glave na disku v predhodni implementaciji lahko prihranimo na dokaj enostaven način : dnevnik transakcije vodimo vedno v glavnem imeniku diska, ki je blizu FAT tabeli. Če v glavnem imeniku nimamo veliko datotek, je ta imenik celoten vsebovan v istem cilindru kot FAT tabela, torej ne pride do premika diskovne glave ob zapisovanju imenika, kar je očiten prihranek.

Optimizacija 2 : Prealociranje dnevnika

Bistven prihranek lahko dosežemo z dokaj enostavnim trikom : ob prvem pisanju v dnevnik zapišemo vanj dovolj dolg prazen zapis, ki zasede prostor za naslednjih nekaj'zapisov v dnevnik. Zasedenost dnevnika oz. logični konec dnevnika lahko sedaj določimo samo s pomočjo magičnih številk v glavi in podatkovnem delu zapisa v dnevniku. Ta optimizacija prihrani vpis v imenik in FAT tabelo v 95% vseh zapisov (ta procent je seveda odvisen od velikosti vnaprej dodeljenega prostora v dnevniku in povprečne velikosti zapisa v datotekah baze podatkov). Še vedno pa moramo ob vsakem zapisovanju podatka na disk fizično zapisati zapis v dnevnik, kar povzroči preskakovanje diskovne glave med dnevnikom in podatkovno datoteko.

Implementacija dnevnika s podatkovnim vmesnim pomnilnikom

V prejšnji implementaciji dnevnika smo videli, da je bil bistveni omejitveni faktor hitrosti zahteva, da moramo vsak zapis v dnevnik tudi fizično zapisati na disk. Če hočemo v MS-DOS operacijskem sistemu zagotovo zapisati en zapis pred drugim je to tudi edina možnost. Možnost hitrejšega dela z dnevnikom moramo torej iskati drugje.

Ker je bil naš sistem za delo z bazo podatkov napisan v Turbo Pascalu, je razumljivo pričakovati, da bodo tudi aplikacije napisane v Turbo Pascalu. Izkaže se, da je zaradi dokaj velike učinkovitosti prevajalnika za Turbo Pascal povprečna velikost aplikativnega programa relativno majhna (v primerjavi z npr. Clipper prevajalnikom za dBASE III). Poleg tega lahko dosežemo majhne programe z razdelitvijo aplikacije v več programskih modulov. Vidimo torej, da imamo v vsakem aplikativnem programu ob danes standardni konfiguraciji sistema (640K osnovnega pomnilnika) na razpolago veliko količino dinamičnega pomnilnika, ki bi ga veljalo koristno izrabiti. V sistemu za delo z bazo podatkov moramo torej ta dinamični pomnilnik čim koristneje izrabiti za hitrejše delovanje aplikativnih programov.

Za uporabo dinamičnega pomnilnika smo uporabili sledeči pristop :

- Zapisi v dnevnik se zapisujejo kot v običajno datoteko (brez CLOSE in COMMIT zahtev).
- Zapisi v podatkovnih datotekah se hranijo najprej v dinamično dodeljenem glavnem pomnilniku (Turbo Pascal funkcije GetMem in FreeMem), kjer sistem za delo z bazo podatkov z njimi tvori vmesni pomnilnik (*Cache*).
- V vmesni pomnilnik vstopajo tudi vsi zapisi, ki jih aplikacija samo bere.
- Ob COMMIT ali ROLLBACK zahtevi se izvede CHECKPOINT - zapre se dnevnik, nato se vsi zapisi vseh podatkovnih datotek zapišejo na disk, datoteke se zaprejo, da so vsi podatki na disku tudi resnično poknjiženi, dnevnik se nato zbriše.

[9] V tem delu nas zanima v glavnem izpad računalniškega sistema, ob nenormalnem koncu programa smo lahko prepričani, da bo operacijski sistem do konca zapisal vse podatke na disk
- MS-DOS hrani podatke v svojem delu pomnilnika v vmesnikih, ki so organizirani po cca. LRU sistemu ob upoštevanju prioritet (najprej se iz vmesnikov odstranijo podatki, nato direktoriji, nato FAT), torej obstaja velika verjetnost, da bo FAT in imenik zapisan na disk šele kasneje kot posamezni podatki.

- Kadar je dinamično dodeljevan pomnilnik poln, sistem za delo z bazo podatkov najprej sprosti zapise v vmesnem pomnilniku, ki niso bili spremenjeni (read-only); ko bi sistem moral sprostiti spremenjen zapis naredi delni CHECKPOINT - dnevnik se zapre, vsi zapisi se zapišejo na disk, podatkovne datoteke se NE ZAPREJO.

S tem pristopom smo dosegli kvantitetno in tudi kvalitetno spremembo v delovanju sistema - hitrost dostopa do diska se je povečala cca za faktor 10. Tako opremljen sistem zaradi implementiranega vmesnega pomnilnika deluje celo hitreje kot originalni Database Toolbox brez možnosti povrnitve. Zastoj v delovanju sistema se pojavi samo ob CHECKPOINT operaciji, ko se zapišejo na disk vsi čakajoči zapisi (v klasični aplikaciji, napisani v Turbo Pascalu je lahko v dinamičnem pomnilniku tudi do 300K podatkov). Zaradi zmanjšanja tega zastoja smo omejili maksimalno dovoljeno količino podatkov v cache pomnilniku. Tako sta za uporabo cache pomnilnika dva omejitvena faktorja : velikost dinamično dodeljenega pomnilnika in velikost samega cache pomnilnika.

Ker obstaja možnost, da bi sistem za delo z bazo podatkov za svoje delo zasedel celoten razpoložljiv dinamično dodeljevani pomnilnik, smo za implementacijo cache pomnilnika uporabili posebno funkcijo, ki jo Turbo Pascal podpira v svoji sistemski knjižnici. Funkcija HeapError se izvede kadar Turbo Pascal ob New ali GetMem klicu ne najde zadostne količine dinamično dodeljevanega pomnilnika. To funkcijo uporablja sistem za delo z bazo podatkov, da po potrebi sprosti del vmesnikov v cache pomnilniku in tako omogoči aplikacijskemu programu nemoteno delo. Dinamično dodeljevani pomnilnik je tako popolnoma izkoriščen - če ga aplikativni program ne uporablja, ga zasede sistem za delo z bazo podatkov, drugače ga lahko aplikativni program izkoristi do konca.

Po evaluaciji zgoraj navedenih možnosti smo se odločili, da je implementacija z uporabo vmesnikov v dinamično dodeljevanem pomnilniku daleč najboljša. S pomočjo te implementacije smo dalje razvijali še naslednje cilje, ki naj jim zadosti zanesljiv sistem za delo z bazo podatkov v mikroračunalniškem okolju : kompresija dnevnika in podtransakcije.

Kompresija dnevnika

Kompresija dnevnika naj dodatno optimizira delo sistema. V dnevnik se morajo zapisovati samo tisti zapisi, ki so neobhodno potrebni za povrnitev datotek v primeru izpada računalniškega sistema ali programa. Kadar uporabljamo dnevnika z metodo senčenja lahko dosežemo kompresijo dnevnika z dvema pristopoma:

- Zapisov, ki smo jih v tej transakciji dodali v bazo podatkov ni potrebno voditi v dnevniku, saj nimajo stare vrednosti.
- Za zapise, ki jih v transakciji večkrat spremenimo, zadošča v dnevniku samo prva vrednost, saj se bodo ob operaciji povrnitve naslednje vrednosti zanesljivo izgubile.

Če hočemo ugotoviti, kateri zapisi so v bazi podatkov novi, moramo ločevati dve možnosti :

- Zapis je prekril enega od obstoječih zbrisanih zapisov.
- Zapis je popolnoma nov in predstavlja razširitev podatkovne datoteke.

Vodenje evidence zbrisanih zapisov je zahteven proces, ki ne prinese rezultatov, ki bi opravičevali njegovo uporabo, poleg tega ga lahko vgradimo v naslednji proces - kompresija zapisov spremenjenih vrednosti. Veliko enostavneje je ugotoviti, kateri zapisi so bili dodani na konec datoteke - enostavno primerjamo velikost datoteke na začetku transakcije z velikostjo datoteke med transakcijo. Zapisov, ki smo jih dodali preko originalne velikosti datoteke, ni potrebno zapisovati v dnevnik.

Za kompresijo zapisov v dnevniku, ki vsebujejo podatke o spremenjenih zapisih, uporabimo cache pomnilnik. V tem pomnilniku označimo zapise, ki so bili spremenjeni in zapise, ki so že bili zapisani v dnevnik. Ko hočemo ponovno zapisati v dnevnik staro stanje zapisa, ki je že bil zapisan v dnevnik, to operacijo preskočimo.

Zanimivo je, da sta zgoraj omenjeni operaciji po uporabi v realni aplikaciji dokaj strogo ločeni. Navadno program ne popravlja istega zapisa večkrat, prav tako se novi ključi v indeksni datoteki navadno dodajajo v že obstoječe strani. Kompresija dnevnika z vodenjem evidence novih zapisov je torej uporabna večinoma za podatkovne datoteke. Po drugi strani se s spremembami ključev pogosto spremeni več zapisov v indeksni datoteki (zapis, ki vsebuje ključ in po potrebi še nekaj zapisov, ki so v hierarhični strukturi nad njim), kar pomeni, da kompresija dnevnika ob uporabi indeksnih datotek prinese opazne prihranke.

Omeniti velja, da je izvajanje kompresije dnevnika omejeno na število podatkovnih zapisov, ki jih je sposoben držati cache pomnilnik. To število ni tako veliko, da bi lahko pri podatkovni datoteki izvajali opazno količino kompresije dvakrat spremenjenih zapisov, medtem, ko je to število zadosti veliko, da pokrije kar precejšen del drevesne strukture indeksne datoteke, kar pomeni, da so zaporedne spremembe ključev, ki so v drevesni strukturi blizu eden drugemu, za ta način kompresije ugodni. Še ugodnejše je seveda sekvenčno polnjenje baze podatkov, kjer so si zadnji uporabljeni ključi navadno zelo blizu, kar prinese visok faktor kompresije.

Podtransakcije

Podtransakcije so koncept, ki ga v velikih sistemih za delo z bazami podatkov navadno ne srečujemo. Ti sistemi so orientirani na transakcijsko vodene aplikacije, kjer vsaj ob interaktivnem delu velja en program - ena transakcija.

V MS-DOS okolju, kjer je nalaganje in startanje programa zamudna operacija, posebej zato, ker je potrebno ponovno odpreti vse datoteke baze podatkov, s katerimi program deluje[10]. Koristno je torej, da je čim večji del aplikacije zajet v enem programu.

[10] Ne smemo pozabiti, da je sistem za delo z bazo podatkov po zasnovi del aplikacijskega programa in ne operacijskega sistema. Zato ima vsak program svojo kopijo tega sistema, torej ni možno ohranjanje odprte baze podatkov preko izvajanja več programov. Vsak program mora torej inicializirati bazo podatkov od začetka.

Če hočemo zajeti čimveč aplikacije v enem programu, se znajdemo v naslednji dilemi

- Program je lahko razdeljen na transakcije, tako npr. vsaka funkcija v glavnem menuju izvede svoj klic BEGIN_TRANSACTION in COMMIT oz. ROLLBACK. Na ta način je program zgubil svojo identiteto kot celota in v primeru izpada računalniškega sistema bo del opravil, ki jih je program opravil, pustil svojo sled v podatkih, drug del opravil pa bo izginil. Navadno takšno obnašanje ni zaželjeno.
- Program mora biti enovita transakcija. V tem primeru zgubimo elegantno možnost, da posamezno funkcijo programa prekinemo in enostavno popravimo vse podatke z enim samim ROLLBACK klicem.

Če hočemo torej združiti učinkovitost in lepoto programiranja z omejitvami, ki jih postavlja MS-DOS okolje, smo prisiljeni definirati transakcijo rekurzivno, kar nas avtomatsko privede do definicije podtransakcij :

Definicija :

> (Pod)transakcija je enota dela z bazo podatkov. (Pod)transakcija je omejena s klici BEGIN_TRANSACTION in ROLLBACK oz. COMMIT. COMMIT klic potrdi vse spremembe v bazi podatkov, ki so se zgodile od ustreznega BEGIN_TRANSACTION klica dalje[11]. ROLLBACK klic povzroči, da se vzpostavi v bazi podatkov stanje enako stanju tik pred ustreznim BEGIN_TRANSACTION klicom.[12]

Tako definiran sistem podtransakcij se je izkazal kot izredno uporaben v interaktivnih obdelavah z večstopenjskim sistemom vnosa podatkov. V vsaki stopnji obdelave lahko uporabnik prekine vnos samo na tisti stopnji, vsi že vnešeni podatki na višjih stopnjah se ohranijo, kljub temu jih lahko kasneje uniči (primer : vnos naročila za proizvodnjo, kosovnice v naročilu, elementi v kosovnici).

Implementacija povrnitvenih procedur v sistemu podtransakcij, kjer uporabljamo za dnevnik sistem senčenja je enostaven :

- Za vsak nivo transakcij hranimo pozicijo v dnevniku, ki ustreza poziciji ob BEGIN TRANSACTION klicu.
- Ob ROLLBACK klicu podtransakcije uničimo spremembe do označene pozicije v dnevniku
- COMMIT podtransakcije naredi samo CHECKPOINT (gl. prejšnji razdelek), COMMIT glavne transakcije naredi CHECKPOINT in zbriše dnevnik, saj so sedaj ti podatki brezpredmetni

Implementacija podtransakcij nam prinese še dodatno komplikacijo. V sistemu brez podtransakcij smo lahko optimizirali zapise v dnevnik (gl. razdelek o komprimiranju), česar sedaj ne moremo več delati. Podtransakcija lahko namreč spremeni novo dodan zapis (njegove vrednosti ne hranimo v dnevniku), nato naredi ROLLBACK in podatki v datotekah baze podatkov so neveljavni. Kompresija dnevnika s pomočjo ignoriranja novo dodanih zapisov sedaj ni več možna.

Večkratno ažuriranje že obstoječih zapisov je še vedno razlog za kompresijo dnevnika, vendar moramo ob vsakem BEGIN TRANSACTION klicu izbrisati vse oznake, da je zapis že bil zapisan v dnevnik. Tako lahko vsaka podtransakcija z ROLLBACK klicem uniči vse svoje spremembe.

Ne nazadnje omenimo še popravljanje podatkov ob izpadu računalniškega sistema. V tem primeru povrnitveni program smatra celoten dnevnik kot enotno transakcijo, saj so zanj meje med podtransakcijami nepomembne. Ker mora povrnitveni program uničiti vse spremembe, ki jih je naredila glavna transakcija, COMMIT podtransakcije ne sme krajšati dnevnika temveč mora v njem pustiti vse spremembe.

Zaključek

Zgoraj opisani sistem smo uspešno vgradili v Turbo Pascal Database Toolbox. Testiranje sistema v realnih aplikacijah je pokazalo, da sistem zagotavlja zanesljivo delovanje sistema v vseh pogojih, tako ob simuliranem izpadu sistema (nasilni reset s pomočjo programske ali strojne opreme) kakor tudi v realnih pogojih izpada električne energije.

Opisani sistem je osnova za razširitve sistema v večuporabniško okolje, kjer je lahko hkrati aktivnih več transakcij (PC-MOS in Novell mrežno okolje), senčenje podatkov lahko v takšnih sistemih uporabimo kot osnovno varovanje celotne baze ob izpadu računalniškega sistema, medtem, ko moramo za potrebe varovanja aplikacij med seboj upeljati sistem zaklepanja zapisov, dnevnika transakcij, ki nam omogoča selektivni ROLLBACK ene same transakcije ob nenormalnem koncu izvajanja programa ali na programsko zahtevo.

Izkaže se, da je najprimernejša implementacija takšnega sistema v smislu database strežnika (*serverja*), torej kot posebnega programa, ki skrbi samo za delo z datotekami baze podatkov in nudi svoje usluge aplikacijskim programom.

Naslednja možnost razvoja zadeva razširitev standardnih produktov (npr. Clipper, dBASE III Plus, dBASE IV) z možnostmi povrnitve, ki bi jih dodali kot dodatno funkcijo operacijskega sistema ali kot dodatno možnost samega jezika (Clipper). Na ta način bi dosegli visoko zanesljivost dela že obstoječih aplikacij in podaljšali njihovo življenjsko dobo.

(11) Vse nadaljnje transakcije na istem nivoju ne morejo uničiti teh sprememb, lahko jih samo dodatno spremenijo, uniči jih lahko le ROLLBACK klic na višjem nivoju (gl. naslednjo opombo).

(12) ROLLBACK klic uniči torej tudi vse spremembe, ki so jih izvedle morebitne podtransakcije zadnje transakcije, čeprav so bili te spremembe potrjene z ustreznimi COMMIT klici.

NEKAJ PRAKTIČNIH VIDIKOV UPORABE CASE ORODIJ

INFORMATICA 1/90

Keywords: case tools, application, computer control

Marjan Rihar
Institut »Jožef Stefan«, Ljubljana

V članku so opisane vrste in značilnosti današnjih CASE orodij, njihova uporabnost v razvojnih fazah projektov ter nekaj naših izkušenj pridobljenih z dosedanjim delom na tem področju.

Pri izdelavi in izvedbi projektov računalniške avtomatizacije, s katerimi se ukvarjamo na našem odseku, predstavlja enega izmed zahtevnejših delov programska oprema. Ta mora biti sposobna reševati kompleksne operacije in delovati v realnem času. Za izdelavo programske opreme večkrat nimamo na voljo zadostnih specifikacij od naročnika ali izdelanega vzorca za načrtovanje sistema. Menimo, da pri reševanju tovrstnih problemov, lahko izdatno izkoristimo vse prednosti, ki jih nudi pristop k izdelavi projektov s pomočjo metodologij programskega inženiringa in CASE orodij.

SOME PRACTICAL ASPECTS OF USING CASE TOOLS. This paper describes types and characteristics of today s CASE tools, their applicability to project development phases and some our experiences acquired during our work in this domain.

Software represents one of the more important parts of elaboration and accopmlishment of computer control projects we are concerning with in our department. It must be able to solve very complex tasks which operate in the real - time environment. We often do not have sufficient specifications from end user or we also do not have existent patern to make such software. We believe, that we can significantly take advantages of approaches to making projects with the assistance of Software Engineering methodologies and CASE tools to solving such problems.

1. UVOD

Z obsežnostjo in zapletenostjo sodobnih avtomatskih in informacijskih sistemov, v katerih ključno mesto zavzema programska oprema, se težišče razvoja vse bolj prenaša s faze kodiranja v fazo sistemske analize in načrtovanja. Kompletni tovrstni projekti obsegajo še ocenjevanje stroškov, časovno porazdeljevanje posameznih faz projektov, porazdeljevanje in sklajevanje nalog med izvajalci, ocenjevanje uspešnosti itd. Vsa našteta področja so bila že dalj časa teoretično temeljito obdelana v sklopu inženiringa programske opreme (Software Engineering), vendar se je celovit pristop k projektom začel šele v sedemdesetih letih s prodorom zmogljivejših in uporabniško prijaznejših računalnikov. Pravi razmah je to področje doživelo z razcvetom osebnih računalnikov. Začel se je razvoj in tudi že uporaba številnih programskih orodij (CASE orodja), ki v znatni meri avtomatizirajo posamezne faze projektov, poleg tega pa skušajo do nedavnega navidezno različne faze, kot na primer vodenje projektov in razvoj pripadajoče sistemske opreme, združiti v enotnem okolju imenovanem CASE (Computer Assisted Software / System Engineering).

2. ORODJA UPORABLJENA V ŽIVLJENJSKEM KROGU PROJEKTOV

Za celovit pregled nad projekti se poslužujemo modela življenjskega kroga projektov. Izmed večjega števila znanih modelov smo izbrali strukturni model/2/. Na sliki 1.je predstavljen kot diagram pretoka podatkov (Data Flow Diagram). Opisi v krogcih predstavljajo dejavnosti posameznih faz projektov, opisi v pravokotnikih izvajalce teh dejavnosti, usmerjeni tokovi pa prehajanje informacij med izvajalci oziroma dejavnostmi.

Na kratko si bomo ogledali obseg dejavnosti v vseh fazah projektov in orodja, katerih uporabo terja sodoben pristop k izdelavi projektov.

Faza planiranja obsega vse aktivnosti v zvezi z določanjem ciljev in sistemskih zahtev uporabnika, časom trajanja posameznih faz projekta, izbiro izvajalcev in razvojne opreme. Skratka, gre za iskanje optimalnega razporeda in obsega vseh naštetih dejavnosti glede na minimalen čas in minimalne stroške, maksimalno produktivnost ter maksimalno kvaliteto opravljenega dela. Modelna orodja, ki jih uporabljamo v tej fazi, so lahko diagrami pretoka podatkov (Data Flow Diagrams), diagrami poteka (Flow Charts), diagrami razmerij med entitetami (Entity Relationship Diagrams), izredno koristni pa so se izkazali na primer PERT in Gantt-ovi diagrami. Ti poleg povezav med dejavnostmi v času trajanja projekta omogočajo tudi prikaz časovnega zaporedja izvajanja posameznih opravil ter ugotavljanje kritičnih točk projekta /2/.

Nadaljne faze življenjskega kroga projekta bomo obravnavali s stališča razvoja programske opreme /10/.

Tudi za programsko opremo lahko rečemo, da ima svoj življenjski krog, ki je podmnožica življenjskega kroga projekta. Začne se s sistemsko analizo in nadaljuje preko načrtovanja, izvedbe, do vzdrževanja. Vse faze razvoja programske opreme so med seboj izredno prepletene. Sosledje faz in čvrstost povratnih povezav sta odvisna od pristopa, ki je lahko radikalen, konservativen ali ustrezna kombinacija obeh /2/.

V *fazi sistemske analize* se poslužujemo številnih metodologij /4/, ki so namenjene graditvi modela zahtev sistema. Izbor primerne metodologije je med drugim odvisen od obsežnosti in vrste sistema. Vsaka metodologija podpira eno ali več orodij za modeliranje. Tako je naprimer za podatkovno intenzivne sisteme najprimernejše modelno orodje diagram razmerij med entitetami P. P. Chena. Za modeliranje sistemov, kjer je glavni povdarek na zahtevnem preoblikovanju razmeroma enostavnih podatkov, so najprimernejši diagrami pretoka podatkov, ki jih v različnih enačicah vsebujejo pristopi avtorjev DeMarco-a, Gane in Sarson-a, Yourdon-a, Hatley in Pirbhai-a. Za sisteme, kjer je poleg strukture podatkov in funkcij važno tudi časovno obnašanje, to je v sistemih realnega časa, so primerni diagrami prehajanja stanj (State Transition Diagrams), tabele akcij (Action Tables) in tabele stanj (State Tables). Ta modelna orodja nastopajo kot razširitve nekaterih metodologij kot so na primer Yourdon-ova, Hatley-Pirbhai-jeva, DARTS /2,4/. Poleg naštetih grafičnih modelnih orodij vsaka metodologija uporablja še vrsto tekstovnih. Za vse elemente, ki nastopajo v obeh vrstah modelov, metodologije predvidevajo centralno shrambo podatkov običajno imenovano slovar podatkov (Data Dictionary).

V *fazi načrtovanja* modele, ki smo jih dobili med sistemsko analizo, pretvorimo v module bodoče programske opreme. Pri tem se poslužujemo dveh temeljnih strategij, to je transakcijske in transformacijske analize /1,2/. Osnovno grafično orodje v tej fazi so strukturni diagrami (Structure Charts), ki prikazujejo hierarhično zgradbo programske opreme v obliki modulov. Module pojasnjujejo tekstovne specifikacije (Module Specifications).

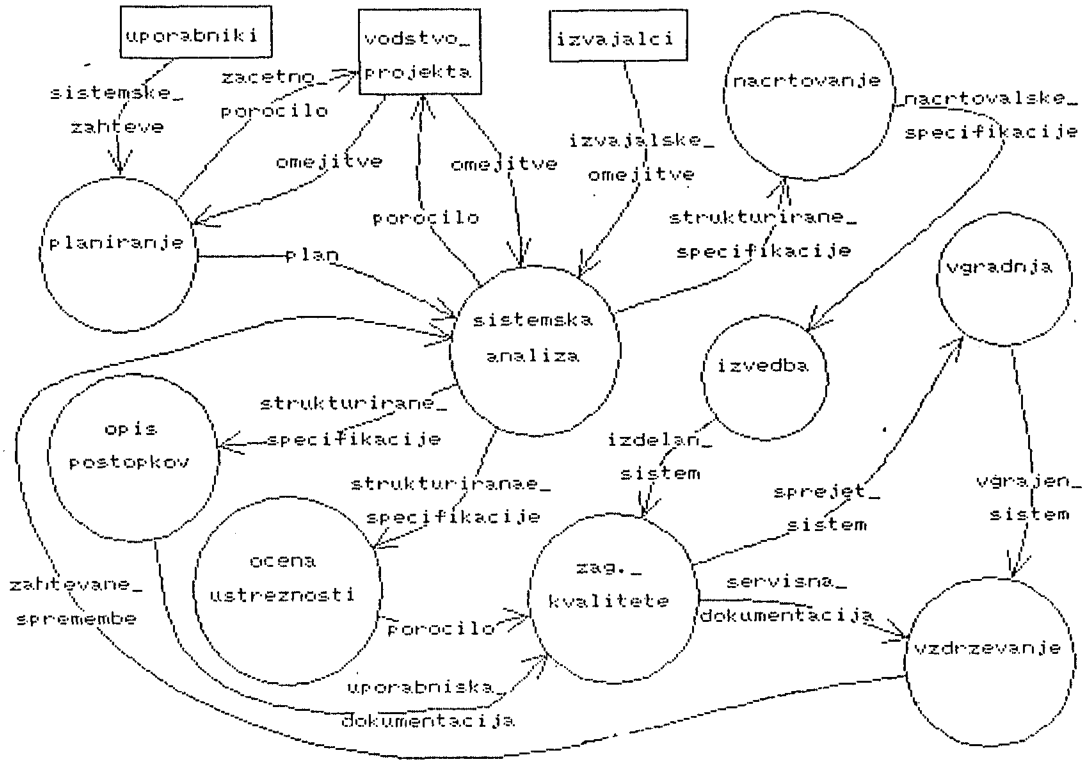
Faza izvedbe obsega izdelavo modulov v ustreznem programskem jeziku, posamično preiskušanje, združevanje programske opreme z materialno in nato preiskušanje kompletnega sistema.

Vzdrževanje obsega dopolnjevanje in spreminjanje programske opreme. Sem spadajo popravki pozno odkritih napak, spremembe ali dopolnitve uporabnikovih dodatnih zahtev, pomoč uporabnikom pri pravilni uporabi in navodila za vzdrževanje sistema. Vsako spremembo je potrebno tudi sproti dokumentirati. Iz tega sledi, da v tej fazi uporabljamo in tekoče vzdržujemo kompletno dokumentacijo, nastalo v predhodnih fazah. S tem je življenjski krog programske opreme sklenjen.

Poleg do sedaj opisanih faz, kjer omenjena orodja bistveno vplivajo na njihovo izdelavo, se vzporedno z njimi odvijajo še naslednje važne dejavnosti:

- vzporedno s sistemsko analizo in načrtovanjem poteka *faza ocenjevanja ustreznosti* s stališča uporabnika,

- *zagotavljanje kvalitete* mora biti vgrajeno v vsak projekt. Kvalitetno morajo biti izdelane



Slika 1.: Strukturni model življenjskega kroga projektov

posamezne faze kakor tudi celotna integracija projekta,

- cilj *opisa postopkov* je dokumentacija o tem na kakšen način se vključujejo avtomatizirani postopki v neavtomatizirano okolje projektov,

- *faza vgradnje* je lahko časovno zelo kratka in neza htevna za preposte sisteme, pri obširnih sistemih oziroma projektih pa lahko dolgotrajna in postopna, združena s številnimi dodatnimi aktivostmi uporabnikov in izvajalcev.

3. AVTOMATIZIRANA ORODJA

Vsa v posameznih fazah našteta grafična in tekstovna orodja so v CASE okolju deloma ali popolnoma avtomatizirana. V splošnem lahko rečemo, da so glavne komponente CASE orodij naslednje /2/:

ORODJA ZA VODENJE IN SPREMLJANJA PROJEKTOV pomagajo osebju za vodenje projektov usmerjati aktivnosti v času trajanja projekta. Podatke o trenutnem stanju projekta črpajo iz centralne shrambe podatkov. Nekatera tovrstna orodja imajo možnost vgraditve metodologije po želji naročnika.

ORODJA ZA RISANJE DIAGRAMOV omogočajo risanje diagramov, sprotno preverjanje skladnosti (consistency) z vgrajeno metodologijo in shranjevanje vseh informacij o elementih diagramov v centralno shrambo podatkov.

ORODJA ZA OBLIKOVANJE TEKSTOV so urejevalniki besedil, poleg tega pa tudi orodja za izdelavo različnih tabel in vnašanje podatkov v tabele.

ORODJA ZA PREVERJANJE SKLADNOSTI IN POPOLNOSTI preverjajo popolnost (completness) diagramov in tekstov ter skladnost med podatki,ki nastopajo v diagramih, tekstih in v slovarju. Orodja za operiranje s centralno shrambo podatkov omogočajo pregled, dodajanje in nadaljnje strukturiranje podatkov. Običajno imajo vgrajen mehanizem, ki zagotavlja varnost podatkov.

GENERATORJI KODE omogočajo iz sistemskih specifikacij avtomatsko generacijo strukture programskih modulov za standardne programske jezike, graditev programske knjižnice, avtomatsko uporabo ustrezne kode iz programske knjižnice, nekateri pa celo avtomatsko prevedbo specifikacij v izvršljivo kodo.

REGENERATORJI KODE služijo za predelave obstoječe kode. Kodo, ki je bila napisana v nizkem ali pa v nestrukturiranem programskem jeziku, prevedejo v kodo, ki je dobro strukturirana in napisana v izbranem jeziku tretje generacije.

ORODJA ZA PREISKUŠANJE KODE so najrazličnejše vrste programskih orodij za odkrivanje napak, generatorji testnih podatkov, analizatorji učinkovitosti izvršljive kode, emulatorji itd.

V literaturi /2,6/ se običajno pojavlja delitev CASE orodij glede na možnost njihove uporabe v življenjskem krogu projektov. Tako zasledimo naslednje izraze:

- *višja CASE orodja (upper CASE)*: Uporabljena so v fazi planiranja. To so orodja za vodenje in spremljanje projektov,

- *srednja CASE orodja (middle CASE)*: Uporabljena so v fazi analize in načrtovanja. To so orodja za risanje diagramov,oblikovanje tekstov, preverjanje skladnosti in popolnosti z metodologijo, orodja za vodenje pri postopkih transformacije specifikacij v strukturne diagrame, orodja za optimizacijo in preverjanje strukture, orodja za operiranje s centralno shrambo podatkov,

(za višja in srednja CASE orodja zasledimo tudi skupen izraz *Front-end*)

- *nižja CASE orodja (lower CASE, tudi Back-end CASE)*: Uporabljena so v fazi izvedbe in vzdrževanja. To so generatorji kode, orodja za orodja za preverjanje skladnosti kode s specifikacijami, orodja za reorganizacijo kode, orodja za dokumentacijo kode,

- *integrirana CASE orodja (ICASE ali IPSE - Integrated Project Support Environment)*: Ta predstavljajo zaokroženo skupino programskih orodij, ki z enotno metodologijo podpirajo vse faze življenjskega kroga projekta. To pomeni, da naj bi bila ta orodja sposobna avtomatsko prevesti zahteve uporabnika v izvršljivo kodo. Današnja CASE orodja tega v celoti še niso sposobna. Razkorak obstaja predvsem med srednjimi in nižjimi CASE orodji, in je tem večji, čim večja je zahtevnost projekta.

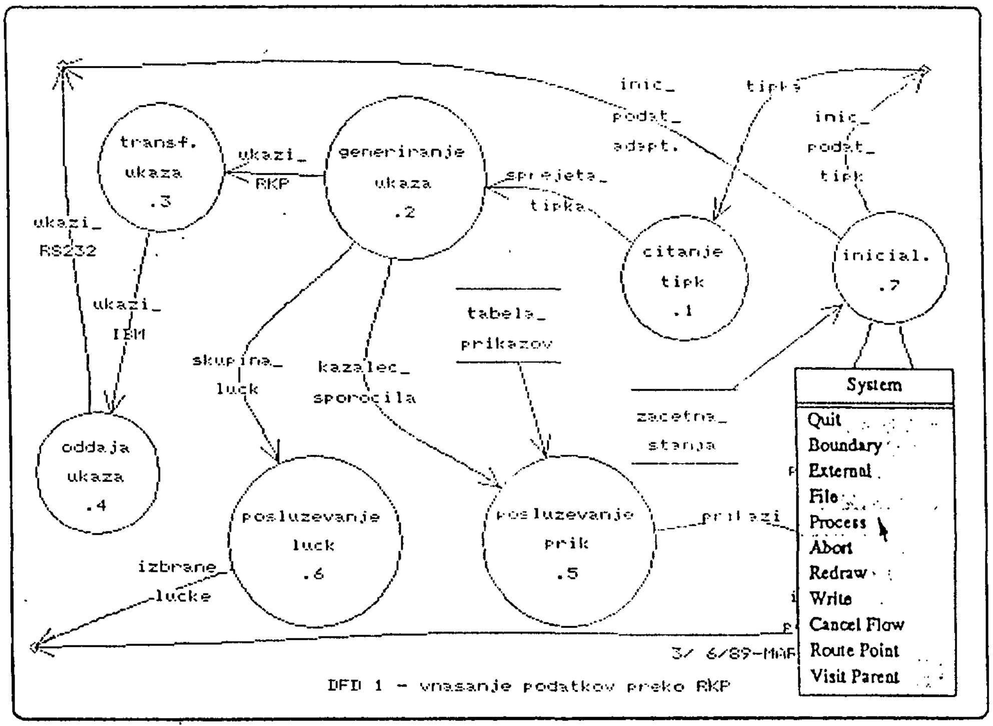
Cene današnjih CASE orodij so razmeroma visoke. Cena je odvisna predvsem od zmogljivosti orodja in od tega, za kakšen računalnik so namenjena. Najcenejša so orodja namenjena za delo na računalnikih tipa PC XT/AT. Običajno podpirajo le eno fazo življenjskega kroga projekta. Zaradi relativne cenenosti so ta orodja zelo primerna za začetno spoznavanje s CASE. Za obsežne projekte niso primerna predvsem zaradi počasnosti in skromnih možnosti, ki jih nudi ustrezna materialna oprema.

CASE orodja, ki dejansko podpirajo vse faze življenjskega kroga projekta, so izredno draga. Namenjena so za računalnike razreda VAX/VMS in tudi večje. Za uspešno delo je potrebno predvideti še veliko število grafičnih delovnih postaj in komunikacijsko omrežje, ki omogoča dostop do centralne shrambe podatkov in medsebojno izmenjavo podatkov.

4. NAŠ PRISTOP K UVAJANJU CASE

Delo na vse zahtevnejših projektih in dosedanje iskušnje iz vodenja projektov računalniške avtomatizacije nas je napotilo, da smo tudi na inštitutu Jožef Stefan začeli razmišljati o CASE. Zavedamo se namreč, da lahko zahtevnejše projekte uspešno izvedemo samo ob dodatni podpori metodologij, ki nas vodijo k skrbnejšemu planiranju projektov, izdelavi ustreznejših specifikacij, učinkovitejšemu programiranju in boljši dokumentaciji. Prvi koraki so bili storjeni z iskanjem primerne metodologije za sistemsko analizo, ki bi bila primerna za tovrstne projekte. Pri tem smo pričeli s proučevanjem strategije avtorjev Hatley-a in Pirbhai-a /3/.

Nekako v tem času smo se seznanili s prvim CASE orodjem in sicer s Structured Analysis tools (SA tools), ki so ga izdelali pri Tektronixu in ga trži Mentor Graphics. Orodje podpira le fazo sistemske analize in je precej togo vezano na De Marco-vo strukturno analizo. S tem orodjem smo poskušali razdelati začetne



Slika 2.: Izgled diagrama pretoka podatkov in del nabora ukazov SA Tools na zaslonu

specifikacije nekaterih vzorčnih nalog. Izgled prikaza enega izmed teh diagramov, izdelanega s pomočjo SA tools, prikazuje slika 2. Na sliki je poleg diagrama pretoka podatkov, ki je glavno modelno orodje SA tools, prikazan tudi eden izmed naborov, s katerimi izbiramo dovoljene operacije.

Orodje je dokaj uporabniško prijazno, vendar se je kmalu izkazalo, da za naše področje ni primerno. Glavna pomankljivost je v tem, da prav tako kot De Marco-va strukurna analiza, ne omogoča prikaza časovnega zaporedja izvajanja posameznih funkcij sistemov. Zasilno rešitev smo poiskali tako, da smo uporabljali pristop Hatley-a in Pirbhai-a, od SA tools pa smo izkoristili zmožnost funkcijske dekompozicije modelov sistemov in avtomatsko preverjanje skladnosti med diagrami pretoka podatkov, mini specifikacijami in podatkovnim slovarjem. Problem predstavitve časovnega poteka izvajanja funkcij smo rešili tako, da smo diagrame ročno dopolnili na papirju ter napisali potrebne kontrolne specifikacije. S tem smo si pridobili občutek za uporabo CASE orodij, spoznali zahteve, ki jih mora tako orodje izpolnjevati, ter naslednje izkušnje:

- CASE orodje, ne glede na to, da v celoti ne podpira vseh uporabnikovih zahtev, uvaja določen red v izvajanje projektov,

- za reševanje problemov računalniške avtomatizacije potrebujemo orodje, ki podpira metodologije primerne za predstavitev v realnem času delujočih sistemov,

- sprotno nastajajočo dokumentacijo moramo stalno dopolnjevati, če želimo, da je uporabna,

- v začetku izgubimo precej časa, če ne poznamo dovolj dobro ustrezne metodologije - vrstni red mora biti najprej poznavanje metodologije, potem uporaba orodij,

- začetna počasnost zaradi neveščega rokovanja z orodjem je zelo hitro presežena,

- izkušeni programerji, ki so spočetka dosledno zavračali CASE so postali dovzetni za njegove prednosti.

Naslednji velik korak je bil storjen z možnostjo dela s CASE orodjem Yourdon Analyst/Designer Workbench (YA/DW). Iz izkušenj smo vedeli, da poznavanje metodologije bistveno za uspešno delo z orodjem, zato smo precej časa posvetili študiju literature /2/. Prehod iz SA tools na YA/DW ni bil težak, ker imata mnoga skupna izhodišča.

Orodje smo preiskusili na manjšem vzorčnem projektu, sicer skromnem po obsegu, vendar dovolj značilnem za naše področje. Ugotovili smo, da orodje zadovoljuje do neke mere, ne nudi pa tistega, kar smo na osnovi reklamnega gradiva od njega pričakovali (YA/DW Fall 88 release). Kljub temu smo se odločili, da bomo orodje uporabili tudi v bližnjem dokaj zahtevnem projektu z utemeljitvijo "bolje nekaj kot nič".

V naslednjih vrstah bomo na kratko predstavili Yourdon-ov pristop k strukturni analizi in načrtovanju ter orodje YA/DW. Orodje spada po svojih karakteristikah v generacijo srednjih CASE orodij. Sestoji iz naslednjih programskih paketov:

- CORE , ki omogoča risanje predstavitvenih diagramov. Nabor simbolov vsebuje standardne označbe za elemente materialne in programske opreme. Predstavitveni diagrami so uporabni za risanje diagramov poteka in shem računalniške arhitekture.

- ANALYST/DESIGNER, za opredeljevanje zahtev sistema in načrtovanje modulov programske opreme.

- COMPOSE , ki omogoča oblikovanje diagramov in tekstov v enoten dokument.

Proizvajalec za nadaljnje izvedbe orodja predvideva še več dodatnih programskih paketov, ki bodo orodje naredile še bolj univerzalno. Tu je predvsem možnost vgraditve metodologije po zahtevah naročnika za reševanje točno določenih problemov.

Programski paket ANALYST/DESIGNER pokriva fazo sistemske analize in načrtovanja. Temelji na pristopu avtorjev Yourdona in Constantina, ki je zelo univerzalen in kot tak omogoča sistemsko analizo in načrtovanje praktično vseh vrst sistemov. Pristop k sistemski analizi temelji na treh postavkah:

1.Dobro modeliranje : za modeliranje funkcij sistemov so uporabljeni diagrami pretoka podatkov in tekstovne specifikacije, za modeliranje strukture podatkov diagrami razmerij med entitetami ter za modeliranje časovnega izvajanja fukcij sistemov diagrami prehajanja stanj.

2.Zaporedno izboljševanje predstavitve sistemov: modele je potrebno v fazi sistemske analize in načrtovanja neprenehoma izboljševati in sicer tako dolgo, da se tesno približajo željam naročnika.

3.Dobro razgrajevanje (dekompozicija) predstavitve sistemov, kar nas vodi od splošnih lastnosti k podrobnostim. Od lastnosti sistema je odvisno ali je poudarek na razgrajevanju podatkov, funkcij ali časovnega izvajanja funkcij. Smer razgrajevanja, ki je lahko od zgoraj navzdol, od spodaj navzgor ali kombinirana /2/, je odvisna predvsem od obsežnosti sistemov.

Orodje YA/DW bistveno vpliva na realizacijo vse treh postavk z naslednjimi poglavitnimi zmožnostmi:

- z izborom simbolov za posamezne elemente diagramov nas vodi, da na pravi način uporabljamo le tiste, ki so skladni z metodologijo,

- z izborom ukazov dovoljuje izvedbo le določenega nabora operacij,

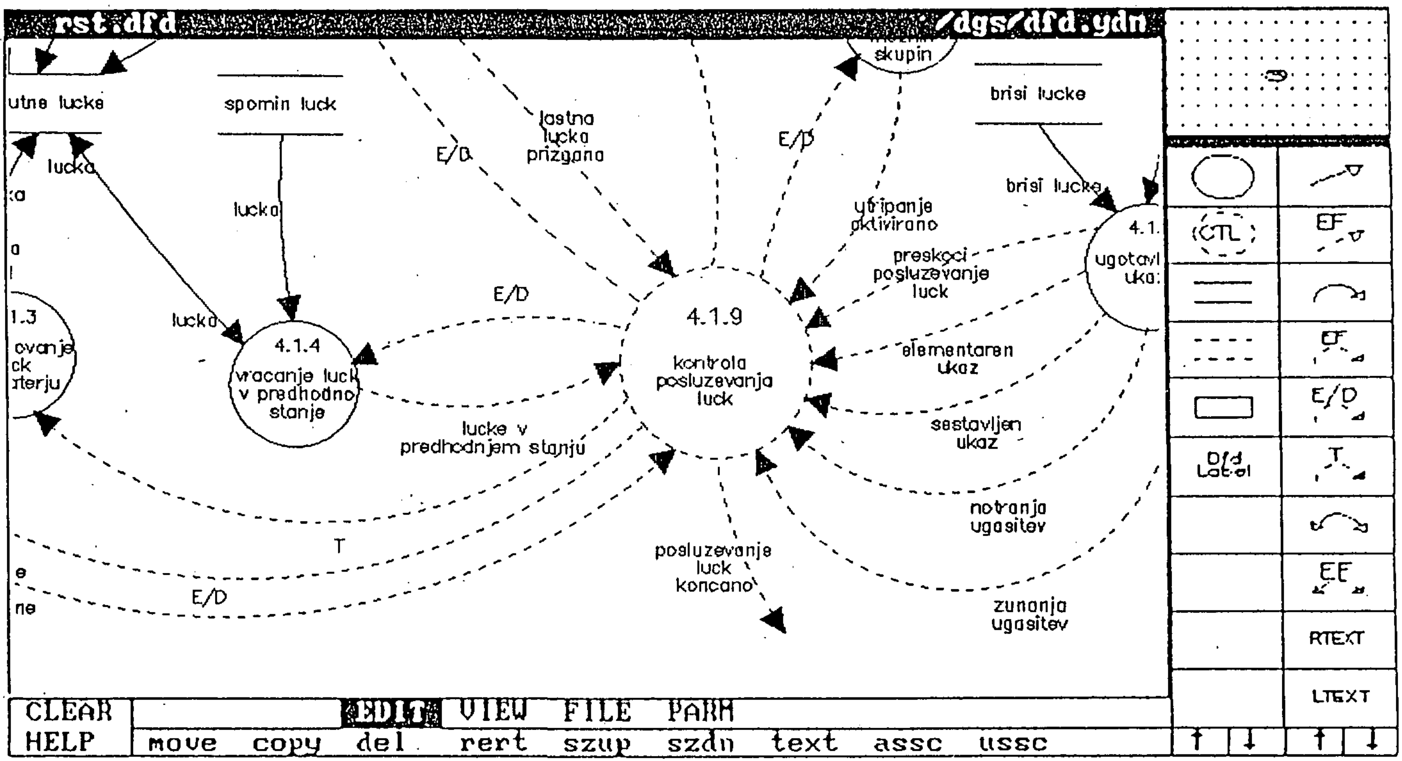
- omogoča hitro razvrščanje, kopiranje, povečave, pomanjšave tako posameznih simbolov, kot tudi samo dela ali pa celotnih diagramov.

- zbira vse podatke v slovarju,

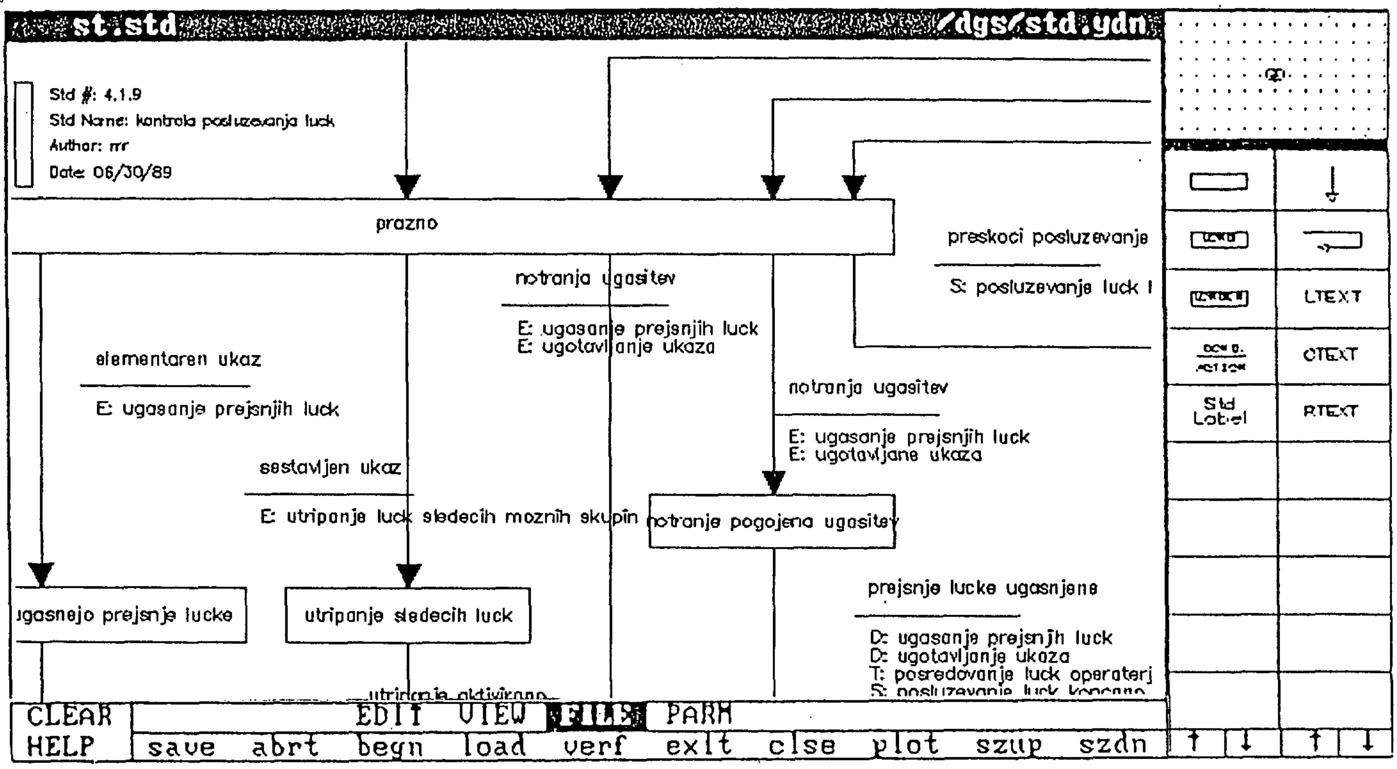
- omogoča interaktivno prehajanje med diagrami, tekstovnimi specifikacijami in slovarjem,

- samostojno preverja skladnost z metodologijo

- omogoča povezavo podatkov z zunanjo podat-



Slika 3.: Izgled diagrama pretoka podatkov in del pripadajočega nabora ukazov YA/DW na zaslonu



Std #: 4.1.9
Std Name: kontrola posluzevanja luck
Author: rr
Date: 06/30/89

Slika 4.: Izgled diagrama prehajanja stanj zaslonu

kovno bazo,

- omogoča risanje in izpis na papir.

Izgled prikaza diagramov in enega izmed izborov dovoljenih operacij YA/DW na zaslonu računalnika, prikazujeti slika 3 in slika 4.

Prva prikazuje diagram pretoka podatkov iz našega vzorčnega projekta. Elementi, ki nastopajo na diagramu, so tipični za sisteme delujoče v realnem času.

Na sliki 4 pa je prikazan primer specifikacij kontrolnega procesa 4.1.9 s prejšnje slike z diagramom prehajanja stanj.

Glavna pomankljivost tega orodja je, da nima mehanizma za preverjanje kompletne skladnosti. Skladnost preverja le med diagrami pretoka podatkov, procesnimi specifikacijami in slovarjem. Pri diagramih prehajanja stanj, diagramih razmerij med entitetami in strukturnih diagramih preverja le notranjo skladnost, to pomeni skladnost samih diagramov z metodologijo. To je bistvena pomankljivost tega orodja za izdelavo specifikacij sistemov realnega časa, kamor spada tudi področje računalniške avtomatizacije. Prav tako ne preverja skladnosti modelov iz faze strukturne analize in faze strukturnega načrtovanja.

Druga slaba stran orodja je v izredni počasnosti preverjanja, predvsem pri avtomatskem preverjanju skladnosti in kompletnosti ter urejevanju podatkov celotnega projekta.

5. ZAKLJUČEK

Uporaba primernih CASE orodij v znatni meri vpliva na rok, ceno in kvaliteto projektov, zato so se v svetu že močno uveljavila. Obstajajo že nekatera podjetja, ki s pomočjo CASE orodij dosegajo skoraj popolnoma avtomatiziran postopek izdelave programske opreme.

Pri nas sledimo temu razvoju zaenkrat le kot redki uporabniki. Skopa uporaba je po našem mnenju predvsem posledica relativno slabega poznavanja svetovnih dosežkov na tem področju. Še vedno je precej prisotno mišljenje, da je glavno opravilo izdelave programske opreme kodiranje. Drugi vzrok pa se nahaja v specifičnosti našega okolja. CASE izhaja iz urejenega okolja in je takemu okolju tudi namenjen. Metode in orodja sicer vsebujejo postopke, ki olajšujejo pri nas tako značilne spremembe v kasnejših fazah projekta. Kljub temu pa napake, storjene v zgodnjih fazah, močno podaljšujejo čas trajanja projekta, lahko pa so zanj celo usodne.

Prepričani smo, da v našem okolju ni mogoče uvajati CASE pristopov s slepim posnemanjem tujih izkušenj. Prava pot je v tem, da skušamo najprej kritično preveriti obstoječe možnosti ob izdelavi manjših projektov. Pri tem bomo imeli možnost prilagajanja metodologij specifičnim domačim razmeram in projektom. Šele v naslednji fazi lahko z uporabo CASE orodij na večjih projektih dokažemo vse koristi, ki nam jih razvoj na tem področju obeta.

DODATEK:

V zvezi s CASE se pojavljajo angleški izrazi od katerih nekateri še nimajo ustreznega prevoda v slovenščino, drugi prevodi pa so že uveljavljeni. Zbirka takih izrazov uporabljenih v tem članku, predstavlja spodnji slovarček. Podčrtani izrazi predstavljajo naš predlog za prevod.

Data Flow Diagram - diagram pretoka podatkov

State Transition Diagram - diagram prehajanja stanj

Entity Relationship Diagram - diagram razmerij med entitetami

Structure Chart - strukturni diagram

Action Table - tabela akcij

State Table - tabela stanj

Data Dictionary - slovar podatkov

Structured Analysis - strukturna analiza

Structured Design - strukturno načrtovanje

Konsistency - skladnost

Completness - popolnost

Decomposition - razgrajevanje

LITERATURA

1. T. De Marco, Structured Analysis and System Specification, Prentice-Hall, Englewood Cliffs, 1979

2. E. Yourdon, Modern Structured Analysis, Prentice- Hall, Englewood Cliffs, New Jersey, 1989

3. D. Hatley, I. Pirbhai, Strategies for Real - Time System Specificatin, Dorset House Publishing Co., New York, 1987

4. J. Černetič, Suitability of CASE methods and tools for computer control, Informatica, 1989

5. J. Voelcker, Automating Software: Proceed with Caution, IEEE , Spectrum, July 1988, 25-27

6. K. L. Gibson, The CASE Philosophy, Byte, April 1989, 209-218

7. C. McClure, The CASE Experience, Byte, April 1989, 235-244

8. D. Shear, CASE shows Promise but Confusion still exsists,EDN, December 8, 1988

9. E. J. Chikofsky, B. L. Rubenstein, CASE - Reliability Engineering for Iformation Systems, Software, March 1988, 11-16

10. CASE, Tektronix katalog, 198811. Yourdon - Software Engineering Workbench User Guide, 1988
12. SA Tools User Manual, 1987

novice in zanimivosti

news

EUUG Conference Report

Milan Palian, Iskra Delta Computers

mpalian@idcyuug.uucp

1. Introducing the EUUG

The European UNIX system User Group (EUUG) is a non-profit organization formed in 1976 to further the exchange of information about the UNIX operating system and related technologies. It has since grown into a large organization that coordinates the activities of national user groups in most European countries.

The bulk of EUUG activities are organized by the national group, which include:

UUGA (Austria)
BUUG (Belgium)
DKUUG (Denmark)
FUUG (Finland)
AFUU (France)
GUUG (Germany)
HUUG (Hungary)
ICEUUG (Iceland)
IUUG (Ireland)
i2u (Italy)
NLUUG (Netherlands)
NUUG (Norway)
PUUG (Portugal)
EUUG-S (Sweden)
UKUUG (UK) and
YUUG (Yugoslavia)

Membership in the above organizations provides automatic membership in the EUUG. Both institutions and individuals can become members. There are currently over 3000 members throughout Europe. The services provided include:

UNIX Network (EUnet)

An electronic mail and news service, EUnet, is available to members. With over 900 connected sites it is currently the largest network in Europe. The mail service extends to other networks giving access to more than 8000 sites in the world.

The news service is an electronic bulletin board which provides access to 250 discussion topics. On the average, about 40 Mbytes of technical information are posted every month.

The popularity of this network is due to the fact that the connection costs are extremely low, requiring only a modem for use on a dialup telephone line. Larger institutions are connected using X.25 or leased lines for better performance. The other reason is that both academic and commercial institutions can join, provided they are members of the EUUG or a national group.

A publication called „The European R&D E-Mail Directory„ is published by EUnet and is available from the EUUG.

Newsletter

There is a quarterly technical Newsletter that is included in the membership price. Additional copies can also be purchased.

Software

Software distributions are organized as a centralized service run from CWI, Amsterdam. Some national groups also provide this service. The distribution tapes contain well hardware consists of a 3 COM 505 Ethernet board and a OST PCSnet ISDN S0 board.

Resource Management System for UNIX Networks

D. Schenk, G. Reichelt, A. Pikhard

Describes the UNISYS Remote Management System in which a relational database is used to provide centralized information about all the resources in a network. The system provides an orderly method for adding users, line printers and installing software products in a network of computers.

Porting Applications to the XVIEW Toolkit and the OPEN LOOK Graphical User Interface

Nannette Simpson

The XVIEW toolkit is an X11 windows implementation of the SunView user interface toolkit. The OPEN LOOK interface is an application front end that has been developed by AT&T and Sun and proposed as the standard „Look and Feel„ for windowed applications in UNIX. The idea is that every application should have an interface similar enough for users to learn it very quickly. This is what has made the Macintosh such a success.

The paper deals with portations of existing tools into this new environment. It is also interesting as a confirmation that such applications are becoming available to the general public.

The applications covered include a file manager (similar to Apple's finder), a calendar and an appointments tool.

X Display Servers: Comparing their Functionality and Architectural Differences to Diskless Workstations

Timothy L. Ehrhart

X11 windows is the standard graphical interface for UNIX workstations. It has received wide industry approval and is used by everyone from IBM and DEC to SUN and AT&T. It requires a bitmapped display and can operate in a network. In other words, the graphical application can be on a remote machine and the same workstation screen can be used to control several such applications on various remote machines. The interface has been ported to all UNIX systems, VMS and MS-DOS.

However, as workstations are expensive, various solutions were investigated with the objective of wiping out alphanumeric terminals from the face of the earth. The diskless workstation was the first such means.

Recently, another even less expensive approach is emerging. X display servers are closed systems that execute only the X display code, they have an Ethernet TCP/IP connection and fonts stored in RAM. The software solution is also available that turns an IBM PC AT compatible into an X display server. Several such stations can be connected to an Ethernet instead of using alphanumeric terminals. The attractiveness of such devices is the low price $1000 - $4000 and the fact that they place less load on the LAN than diskless workstations.

Performance Analysis for Shared Oracle Database in UNIX Environment

C. Boccalini

Benchmarking is used to select systems in a purchase bid for Oracle RDBMS platforms which are to be used to implement a booking system for doctors.

Efficient implementation of low-level synchronization primitives in the UNIX-based GUIDE kernel

E. Paire, M. Riveill

Investigates performance aspects of synchronization in the UNIX environment. Describes the implementation of semaphores that do not require system calls to be executed for P and V operations. The performance analysis compares the standard semaphores with a pseudo driver based implementation and finally a predominantly user space implementation. In the last case, performance is increase by approximately two orders of magnitude.

4. The exhibition

The exhibition was fairly large, what follows is a selection. It was primarily attended by Austrian representatives of big vendors. To illustrate how big, let us start with the biggest there is:

IBM

Has obviously decided to give credibility to its claim of entering the UNIX market. Displayed were machines running the AIX operating system on various hardware platforms. This ranged from IBM AIX/370 to IBM AIX/PS 2. The offering was what one might expect OSF/Motif, X windows, TCP/IP, Ethernet. The salesman refused to be baited into discussing the performance of IBM AIX/370.

Data General

Displayed the AViiON Motorola 88100 based worksta-

6. The Andrew Toolkit

Nathaniel Borenstein

ITC CMU

A toolkit for building complex interactive programs in graphics windowing environment.

7. OSI

Colin l'Anson

Hewlett Packard

A practical user introduction to ISO/OSI with emphasis on higher layers.

The program for the evening of the first day included a reception by the Mayor of Vienna. The event proved to be interesting as many well known UNIX expert made their first public appearance wearing neckties, this caused them to be confused with people employed by computer vendors.

Technical Programme

The technical session consisted of 3 gruelling days of coffee talk coffee talk lunch coffee talk coffee talk dinner coffee. Predictably, after all these coffees, sleep was no longer necessary requiring larger quantities of coffee in the days that followed.

A total of 36 papers were presented. This section gives a brief comment on the talks that I found interesting. It's sole aim is to provide information for individuals that have not heard of this event. The abstracts are usually published in the EUUG Newsletter and the full 300 page conference proceedings are available from the EUUG.

The keynote speaker, Professor Ahmed Elmagarmid opened the technical part of the conference with a talk on „Heterogeneous Distributed Computing - A Database Prospective„. An interesting conclusion was that database experts will be called operating systems experts, in the not too distant future.

Are Standards the Answer ?

Dominic Dunlop

This thought provoking talk showed how „species„ of products evolve, merge and die. Recording media was used to provide examples of standards that were ignored, superior candidates that did not become standard and the like. Finally, it was demonstrated that standardization does not really decrease diversity, as new technologies evolve before the old ones can be standardized.

Engineering a (Multiprocessor) UNIX Kernel

Michael H Kelley, Andrew R. Huber

A description of software engineering techniques used by Data General engineers in the implementation of the DG/UX operating system for the Eclipse and AViiON family of computers. The kernel was organized as a hierarchy of subsystems, modules and functions. Information hiding was introduced by separating the interface of functions from its implementation. These well known techniques are not currently used in traditional UNIX kernels which makes them very difficult to maintain. On the bright side, one must admit that this sorry state of affairs has generated a big demand for all of us with UNIX kernel hacking skills.

An Implementation of STREAMS for a Symmetric Multiprocessor UNIX Kernel

Philippe Bernadat

A discussion on the placement of locks and semaphores in a multiprocessor UNIX kernel. The kernel was based on standard AT&T, BSD and SUN components and uses global memory only for the storage of data structures shared by all processors. The implementation did not require the UNIX System V STREAM modules to be reprogrammed, although this would improve performance.

Developing Writing Tools for UNIX Workstations

Martin D. Beer, Steven M. George, Roy Rada

A description of an authoring system that runs on the ATARI-ST under the GEM operating system is followed by a report of the implementation of this system in various environments. The environments included the Andrew Toolkit (ATK) from the Carnegie Mellon University, the Emacs editor and the X11 widgets (all available from the EUUG software distributions). Predictably, ATK was found to be the most appropriate tool (see software above).

Teaching a Spreadsheet how to Access Big Databases

Michael Haberler, Martin Ertl

A public domain spreadsheet (available from the EUUG), running on a PC, was modified to allow SQL queries to be entered and executed on a remote machine. The remote UNIX system and its associate database software returns the result of the query as a table that is incorporated into the spreadsheet. TCP/IP ethernet was the transport facility. The use of Sun RPC (available from the EUUG) was investigated for use at the session level. The primary bottleneck was found to be sending the result table row by row.

UNIX in German speaking countries

Wolfgang Christian Kabelka

The International Software Information Services (ISIS) provide reports that were used to analyze the UNIX market. It is interesting that UNIX already holds 16.6% of the market and that it is leading in the UNIX vs PROPRIETARY war. It is widespread throughout the business community with the exception of insurance companies.

Modelling the NFS service on an Ethernet LAN

Floriane Dupre-Blusseau

The Queuing Network Analysis Package (QNAP) is used to model Sun's Network File System in a local area network. The maximum number of active workstations is predicted and the feasibility of remote file backup is investigated.

RISC vs CISC From the Perspective of Compiler Instruction Set Interaction

Daniel V. Klein

An extremely interesting analysis of MIPS R2000, Motorola 88100, SUN Sparc, DEC VAX, Motorola 68020 and Intel 80386 instruction sets. The author set out to prove CISC was better and ended up understanding why RISC is superior. It was found that CISC instruction sets were littered with instructions and addressing modes which compiler writers found impossible to use in practice. These instructions were designed by assembler programmers, a class of now almost extinct craftsmen. These instructions and addressing modes do not come free because they take up silicon on the chips and prevent one instruction per cycle design. The author concludes the major effect of RISC is making pipelining really work.

SPARC - Scalable Processor Architecture

Martin Lippert

The papers presents Sun's RISC architecture. The SPARC architecture has been licensed to 50 companies. Sun makes a very strong case for the argument that this is the most promissing and open architecture currently on the market. The argument is that Sun licenses not only the chip but the complete architecture including the SunOS operating system. As SunOS is binary compatible with AT&T UNIX System V 4.0, this means that the technology is really available to anyone. Sun is helping hardware vendors produce SPARC machines that range from PC level, VME boards and multiprocessors. Software vendors are similarly encouraged and 500+ applications are already available.

The processor itself is claimed to be superior because it can be implemented in various technologies, using varying numbers of registers making the design scalable. This is combined with one cycle execution, AI and multiprocessor support to give 10-1000 MIPS machines in the $5000 to $5.000.000 range. Current systems are 60 MIPS, while 100 MIPS and 200 MIPS machine are already in development.

Some analysts doubt that Sun will be able to compete against the companies that have licensed its technology. Regardless of Sun's fate, this technology has the technical potential to provide a better alternative to the IBM PC AT and PS/2 architectures.

Processable Multimedia Document Interchange using ODA

Jaap Akkerhuis, Ann Marks, Jonathan Rosenberg, Mark S. Sherman

A report on the work done within the Experimental Research in Electronic Submission project at the Carnegie Mellon University. It investigates techniques for processing documents that consist of text in different fonts, tables, drawings and the like in such a manner that they can be editing on a large number of different systems. This work is extremely interesting because it is based on ISO 8613 and the Andrew Toolkit. In fact, the software will be made available as part of the new Andrew Toolkit.

TCP/UDP Performance as Experienced by User-Level Processes

Josef Matulka

Interesting conclusions from this research include the measurement of the effect of CPU load on communication performance. It would seem that file servers should not be used as execution servers in the TCP/UDP environment.

Interconnection of LANs, using ISDN, in a TCP/IP architecture

Philippe Busseau

A description of a router based on an IBM PC AT running SCO XENIX that can be used to connect two Ethernet TCP/IP networks using ISDN. The additional known packages such as:

MMDFIIb (Mail system)

GNU (Emacs editor, C++ compiler ...)

NFS (Sun's public domain implementation)

ISODE (ISO Development Environment)

X11R3 (X11 Window System)

ATK (The Andrew Toolkit from CMU)

and many others.

Conferences

Conferences are organized every spring and autumn in different European cities. They include technical sessions, tutorials and an exhibition. Attendance varies from 300 to 600 people. The planned venues are:

Munich (Spring 1990)

Nice (Autumn 1990

Tromso (Spring 1991)

Budapest (Autumn 1991)

A conference in Yugoslavia is expected in 1993.

2. YUUG

The Yugoslav UNIX User Group was started as part of a joint UNIX development project by several manufacturers:

Iskra Delta, Ljubljana

Hermes, Ljubljana

IRCA, Sarajevo

IRIS, Sarajevo

Nikola Tesla, Zagreb

which is partially funded by the federal government. Currently 25 institutions have joined. The YUUG provides access to the network, distribution of public domain software and organization of meetings.

3. EUUG Autumn 1989 Conference

The conference venue was the Wirtschaftsuniversitat, Vienna, Austria, which has good conferencing facilities. The following is a description of the program:

Tutorials (Monday and Tuesday)

These are whole-day or half-day events for smaller groups that are scheduled prior to the conference itself. What follows is a short description, which should give a feeling for the technical level. Tutorials are not include in the conference fee but provide good value for money because of the quality of the tutors.

1. UNIX Network Programming

Richard Stevens

Health Systems International

Network programming, oriented towards „sockets„ running over TCP/IP protocols. Comparison with the UNIX System V, ISO/OSI conformant interface, called TLI.

2. RISC

Jean Wood and Ashis Khan

MIPS Inc.

Discussion of alternative RISC architectures and their impact on compilers and operating systems.

3. System V IPC

George Philippot

NCR Norway

Programming of the UNIX System V Interprocess Communication (IPC) functions, with include shared memory, semaphores and messages.

4. Make

George Philippot

NCR Norway

The standard UNIX utility for automatic recompilation and testing of programs.

5. Beyond 4.3 BSD

Mike Karels

Kirk McKusic

The improvements in future Berkeley Source Distributions, which is an influential UNIX variant. These include ISO/OSI communications support and the POSIX 1003.1 interface.

tions and multiprocessor server. They claim binary compatibility with Motorola BCS. There was a nice demo, where the mouse is used to pick information of interest.

Motorola

The centerpiece of the display was an 88100 processor, VME bus system. All the other systems were 680x0.

Olivetti

Their star was the Intel 80386, 33 MHz system (the leaflets they handed out said 25 MHz). The system runs SCO XENIX and all the associated software.

Bacher

This is the local Sun dealer. They demonstrated the use of all these SPARC stations. See article above.

NCR

Displayed a range of machines, the smallest being a Tower the size of a overlarge book. Very interesting was their X display station, the NCR TOWERVIEW. This is a Motorola 68020 with memory and an Ethernet controller. The X11 software is in ROM. There is also an RS232 connection that can be used, but nobody would admit having tried it (Yes - it must be slow, they said).

Intergraph

For all you 27-inch monitor, 2-megapixel colour fans, this is it. Based on the Clipper processor, it runs UNIX and will connect to anything (emulates IBM PC too, including EGA).

There were many more that deserve to be mentioned, among them the Macintosh IIx and the Amiga 2500 UX look much more serious than they used to.

5. Information

To promote the exchange of information, common lunches were organized in the university cafeteria. There was also a conference dinner with record attendance. The daily conference newsletter was there, but proved to be a disapointment. Finally, for the first time, electronic mail was delivered at the conference. Delegate could send and receive mail, as well as brousing through news. The trial was successful and will be expanded at the next conference.

Kaos in informacija

Poročilo **Petra Tomaža Dobrile** (Telex, 23. novembra 1989, 50-51) s Štajerske jeseni med 14. 10 in 19. 11. 1989 v Gradcu (Avstrija), ki je bila namenjena temam, kot so Kaos kot umetnost, znanost in magija, Virtualne arhitekture, Računalniška delavnica pa še koncertom, teatru, filmu itn., sega s svojo problematiko tudi v področje filozofije in tehnologije informacije oziroma v okvir sodobnega pojmovanja informacijskega. Verjetno je značilnost Telexovega zapisa prav v tem, da prihaja naposled do čedalje vidnejšega, že kar produktivnega prepletanja do nedavnega nezdružljivih znanstvenih disciplin (matematike, fizike, kemije, biologije, umetne inteligence, sociologije, ekologije itd.) na eni ter umetnosti (filma, literature, filozofije, slikarstva, glasbe, paranaturalistike itn.) na drugi strani. Med vidnimi predstavniki strok in umetniških smeri velja omeniti imena **C. G. Langton** (Center za nelinearne študije, Los Alamos, avtor knjige Umetno življenje), **R. H. Abraham** (Matematični oddelek Univerze Santa Cruz, Kalifornija s temo Prostorsko-časovni vzorec dinamike kaosa), **S. Addiss** (umetnostni zgodovinar Univerze v Kansasu), **G. Altner** (ekonomija in ekologija na Visoki šoli v Porenju), **C. Bahn** (Center za računalniško glasbo, Brooklin College, New York), **L. Bec** (Znastveni inštitut za raziskavo paranaturalnih pojavov, Sorguesa, Francija), **J. P. Crutchfield** (Oddelek za fiziko, Kalifornijska univera, Berkeley), **R. Eisler** (Center za raziskave na področju medčloveških odnosov, Camel, ZDA), **M. J. Figenbaum** (Matematično-fizikalni oddelek, Rockfellerjeva univerza, New York), **K. Stockhausen** (pionir moderne glasbe) in **V. Globokar** (komponist, Pariz).

Markantne misli Štajerske jeseni so bile npr.: Ako bo videno in doživeto ponazorjeno v govorici logike, se ukvarjamo z znanostjo (A. Einstein). Virtualne arhitekture in fraktali so osnovni delci in celota (vprašanja tehnologije). Življenje ni le lastnost materije, ni nekaj kar je materiji inherentno, temveč je rezultat organizacije materije, značilnost forme (C. G. Langton). Ni nobenega kaosa in nobenega reda (S. Addiss). Tehnologija poskuša producirati naprave, ki spadajo v enostavni tip: ideja se realizira ... (M. J. Figenbaum). Totalna značilnost sveta je v vsej večnosti kaos - ne v smislu pomanjkanja nujnosti, ampak v smislu pomanjkanja reda, razvrstitev, forme, lepote, modrosti, in kakorkoli že je drugih imen za estetski antropomorfizem (Friedrich Nietzsche: Vesela znanost). Itd.

Sinteza živega in kulturnega postaja tako čedalje bolj informacijska tudi na področju zavesti. Tako sta lahko tudi kaos in red le posebni informacijski entiteti. Prepletenost informacijskega je vesoljna. *A. P. Železnikar*

Prva mednarodna konferenca o Moduli-2

V dneh od 11. do 13. oktobra 1989 je potekala na Bledu v organizaciji Instituta Jožef Stefan iz Ljubljane »Prva mednarodna konferenca o Moduli-2« skupaj s predkonferenčno delavnico z naslovom »Izmenjava izkušenj pri uporabi Module-2«.

Konferenco je spremljalo 116 udeležencev iz 14 držav, 60% jih je prišlo iz visokošolskih in znanstvenih ustanov, ostali pa iz industrije. Delavnice se je udeležilo 42 slušateljev, dobra polovica jih je prišla iz industrije.

Programski odbor je od 37 prispelih izbral 25 referatov iz 12 držav. Povabljeni predavatelji so bili: prof. Niklaus Wirth, prof. Gustav Pomberger, prof. Roger Henry, John Souter in Albert Meier.

Častni gost in avtor uvodnega predavanja na konferenci je bil prof. Niklaus Wirth. Prof. Wirth je od leta 1968 predavatelj na ETH v Zürichu (Švica) in vsem dobro poznan kot avtor programskih jezikov Pascal in Modula-2. Sodeluje tudi pri razvoju računalniških sistemov. Njegova znana dosežka sta osebni računalnik Lilith in sistem Oberon. Za svoje raziskovalne dosežke je prof. Wirth prejel leta 1983 Priorejevo nagrado, ki jo podeljuje IEEE, naslednje leto pa Turingovo nagrado, ki jo podeljuje ACM.

Prof. Niklaus Wirth med predavanjem »Modula-2 in objektno usmerjeno programiranje«, ki je naletelo na velik odziv med udeleženci blejske konference.

Modula-2 prodira tudi na področje programirnega inženirstva, zato je programski odbor povabil na konferenco prof. Gustava Pombergerja z univerze v Linzu (Avstrija), avtorja knjige Software Engineering and Modula-2. Glede na to, da se standardizaciji Module-2 posveča velika pozornost, je programski odbor uvrstil v program povabljeno predavanje prof. Rogerja Henryja z univerze v Nottinghamu (Velika Britanija), priznanega strokovnjaka s tega področja in člana delovne skupine Modula-2 pri britanskem institutu za standardizacijo. S tega instituta prihaja tudi John Souter, ki je na konferenci osvetlil položaj Module-2 med pomembnejšimi programskimi jeziki (Ada, C, Pascal). Albert Meier je predsednik in lastnik podjetja A+L AG iz Grenchena (Švica), ki širom po svetu distribuira prevajalnike, knjižnice in programska orodja za Modulo-2. Na konferenci je v okviru povabljenega predavanja podal zgodovinski pregled in perspektive Module-2.

V nekaj naslednjih stavkih bomo skušali strniti vsebino konference.

V dveh sekcijah o objektno usmerjenem programiranju je bilo poleg referata prof. Wirtha predstavljenih še pet referatov. Francesco Tisato z univerze v Calabriji (Italija) je govoril o objektno zasnovanem programiranju sistemov za delo v realnem času. Profesor s sarajevske univerze Suad Alagić je predstavil okolja za programiranje objektno orientiranih baz podatkov. Martin Odersky in Beat Heeb z ETH v Zürichu (Švica) sta v dveh prispevkih opisala jezik Modula-90, ki izhaja iz Module-2 in poudarja značilnosti objektno usmerjenega programiranja. Način definiranja objektov v Moduli-2 je v svojem prispevku podala Letizia Leonardi z univerze v Bologni (Italija).

Prispevek našega instituta je bil predstavljen v okviru sekcije Jezikovne razširitve. Referat o polimorfnih funkcijah sta pripravila Andrej Brodnik in prof. Boštjan Vilfan s Fakultete za elektrotehniko in računalništvo v Ljubljani. Gabriella Dodero z univerze v Genovi (Italija) pa je podala možnost razširitve s permanentnimi moduli.

Uporabo Module-2 v paralelnih sistemih so predstavili prispevki v sekciji Sočasnost. Gerard Padiou z ENSEEHIT-a v Toulousu (Francija) je govoril o uporabi Module-2 v porazdeljenem okolju. Vključitev Hoarejevega koncepta CSP (Communicating Sequential Processes) v Modulo-2 je predstavil Jürgen Vollmer z univerze v Karlsruheju (ZRN), ki je tako razširjen jezik poimenoval Modula-P. Na Moduli-2 temelji tudi programski jezik M2PLUS, namenjen paralelnemu programiranju, ki ga je v svojem referatu opisala Viera Šipková z Instituta za tehnično kibernetiko v Bratislavi (ČSSR).

V večerni sekciji prvega dne sta o perspektivah Module-2 govorila že prej omenjena povabljena predavatelja John Souter in Albert Meier.

Prof. Gustav Pomberger je bil uvodničar v sekciji Programirni inženiring. Programska orodja za IBM OS/2 operacijski sistem, ki so implementirana z JPI Modulo-2, je predstavil Timothy Line iz IBM ASD Nordic laboratorija v Lidingu (Švedska). O integraciji abstraktnih podatkovnih tipov v

Modulo-2 je govoril Herbert Klaeren z univerze v Tübingenu (ZRN).

V sekciji Programska okolja so udeleženci spremljali sledeče referate: Walter Bischofberger z univerze v Linzu (Avstrija) je predstavil orodje za konstrukcijo sistemov, imenovano SCT. Referat o uporabniškem vmesniku v programskih okoljih za Modulo-2 je podal Tomas Felner, predstavnik norveškega podjetja Modula-2 CASE Systems iz Osla. Sekcijo je zaključil Libero Nigro z univerze v Calabriji (Italija), ki je predstavil kontrolni mehanizem (treads), s pomočjo katerega postanejo programi, pisani v Moduli-2, neodvisni od run-time podpore in operacijskega sistema.

V sekciji Programiranje sistemov za delo v realnem času je bilo kar pet prispevkov, kar kaže na to, da zavzema Modula-2 pomembno mesto med programskimi jeziki, ki so namenjeni za delo v realnem času. Samir Al-Khayatt z univerze v Loughboroughu (Velika Britanija) je predstavil na Moduli-2 zasnovana programska okolja za večprocesorske »embedded« sisteme. O operacijskem sistemu za delo v realnem času XMOD je govoril Gunter Reińig z univerze v Dresdenu (NDR). Primer uporabe Module-2 za programiranje komunikacijske kartice v delovni postaji, povezani na digitalno omrežje z integriranimi storitvami (ISDN), je opisal Rolf-Dieter Klein z univerze v Augsburgu (ZRN). Brian Kirk iz britanskega podjetja Robinson Associates je predstavil razvoj obsežnega programskega okolja v Moduli-2 za krmiljenje industrijskih obdelovalnih strojev. Modula-2 je našla mesto tudi v sistemih, ki zahtevajo veliko zanesljivost, o čemer priča projekt nadzora pariškega metroja, ki ga je na konferenci predstavil Dominique Couturier iz Matra Transport v Parizu (Francija).

Modula-2 v izobraževanju je bil naslov sekcije, v kateri so bile podane izkušnje pri uporabi Module-2 pri učenju tako osnovnih metod programiranja, programiranja sistemov v realnem času kakor tudi sistemskega ter paralelnega programiranja. V sekciji so sodelovali: Franco Mercalli s Centra »A. Volta« v Comu (Italija), Juan Jose Moreno Navarro z univerze v Madridu (Španija), John Dyke s politehnične univerze v Walesu (Velika Britanija) in Gerard Padiou z ENSEEHIT-a v Toulousu (Francija).

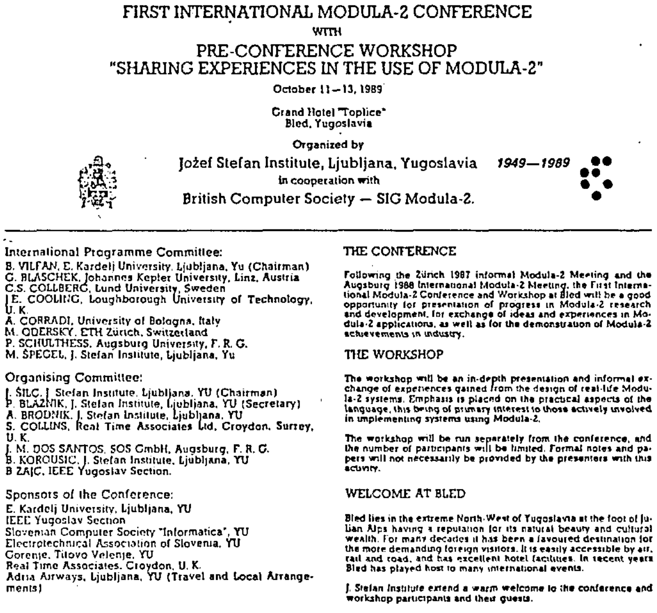
V okviru zaključne sekcije z naslovom Standardizacija je prof. Roger Henry podal »state-of-the-art« na področju standardizacije Module-2.

Vsi prispevki, vključeni v program konference, so zbrani v zborniku. Nekaj izvodov je še na voljo na Odseku za računalništvo in informatiko na Institutu Jožef Stefan.

V enodnevni predkonferenčni delavnici je bilo predstavljenih šest evropskih razvojnih projektov. Jim Cooling z univerze v Loughboroughu, Niall Cooling iz Ready Systems, Paul Curtis iz Rowley Associates in Barry McGibbon iz Robinson Associates so predstavili britanske izkušnje s tremi projekti: operacijski sistem na samostoječih računalnikih za delo v realnem času, načrtovanje prevajalnikov za Modulo-2 in uporaba Module-2 v telemetriji. O norveškem projektu razvoja sistema za programirno inženirstvo v Moduli-2 je govoril njegov avtor Frode Odegård iz Modula-2 CASE Systems v Oslu. Elmar Henne iz podjetja pl GmbH v Münchnu je predstavil zahodnonemški projekt za računalniško načrtovanje. Slovenske izkušnje s programiranjem v Moduli-2 je predstavila sodelavka našega instituta Barbara Koroušić s poročilom o domačem razvojnem sistemu ROMUL-2 za programiranje samostoječih računalnikov v Moduli-2 in o uporabi tega okolja pri razvoju krmilnikov lokalne mreže.

Polona Blaznik in Jurij Šilc

Živahna diskusija po uspešni predstavitvi slovenskih izkušenj z Modulo-2, ki jo je podala Barbara Koroušić.



FIRST INTERNATIONAL MODULA-2 CONFERENCE
WITH
PRE-CONFERENCE WORKSHOP
"SHARING EXPERIENCES IN THE USE OF MODULA-2"
October 11–13, 1989

Grand Hotel "Toplice"
Bled, Yugoslavia

Organized by
Jožef Stefan Institute, Ljubljana, Yugoslavia 1949—1989
in cooperation with
British Computer Society — SIG Modula-2.

International Programme Committee:
B. VILFAN, E. Kardelj University, Ljubljana, Yu (Chairman)
G. BLASCHEK, Johannes Kepler University, Linz, Austria
C.S. COLLBERG, Lund University, Sweden
J.E. COOLING, Loughborough University of Technology, U. K.
A. CORRADI, University of Bologna, Italy
M. ODERSKY, ETH Zürich, Switzerland
P. SCHULTHESS, Augsburg University, F. R. G.
M. ŠPEGEL, J. Stefan Institute, Ljubljana, Yu

Organising Committee:
J. ŠILC, J. Stefan Institute, Ljubljana, YU (Chairman)
P. BLAŽNIK, J. Stefan Institute, Ljubljana, YU (Secretary)
A. BRODNIK, J. Stefan Institute, Ljubljana, YU
S. COLLINS, Real Time Associates Ltd, Croydon, Surrey, U. K.
J. M. DOS SANTOS, SOS GmbH, Augsburg, F. R. G.
B. KOROUSIC, J. Stefan Institute, Ljubljana, YU
B ZAJC, IEEE Yugoslav Section.

Sponsors of the Conference:
E. Kardelj University, Ljubljana, YU
IEEE Yugoslav Section
Slovenian Computer Society "Informatica", YU
Electrotechnical Association of Slovenia, YU
Gorenje, Titovo Velenje, YU
Real Time Associates, Croydon, U. K.
Adria Airways, Ljubljana, YU (Travel and Local Arrangements)

THE CONFERENCE

Following the Zürich 1987 informal Modula-2 Meeting and the Augsburg 1988 International Modula-2 Meeting, the First International Modula-2 Conference and Workshop at Bled will be a good opportunity for presentation of progress in Modula-2 research and development, for exchange of ideas and experiences in Modula-2 applications, as well as for the demonstration of Modula-2 achievements in industry.

THE WORKSHOP

The workshop will be an in-depth presentation and informal exchange of experiences gained from the design of real-life Modula-2 systems. Emphasis is placed on the practical aspects of the language, this being of primary interest to those actively involved in implementing systems using Modula-2.

The workshop will be run separately from the conference, and the number of participants will be limited. Formal notes and papers will not necessarily be provided by the presenters with this activity.

WELCOME AT BLED

Bled lies in the extreme North-West of Yugoslavia at the foot of Julian Alps having a reputation for its natural beauty and cultural wealth. For many decades it has been a favoured destination for the more demanding foreign visitors. It is easily accessible by air, rail and road, and has excellent hotel facilities. In recent years Bled has played host to many international events.

J. Stefan Institute extend a warm welcome to the conference and workshop participants and their guests.

International Conference on

Organization and Information Systems

Bled, September 13-15, 1989

Anton P. Železnikar
Iskra Delta Computers

Ladies and gentlemen, dear guests and participants, Mr. chairman:

It is a great pleasure for me to welcome you on behalf of the general sponsor of this international congress dedicated to the topics of information systems and organization. **Iskra Delta Computers,** the Yugoslav information system industry is aware of the importance of professional international meetings which support the usage of information machinery and information methodology in solving problems of management, decision making, administration and organization within various enterprises of the modern epoch. Since the world's congress of International Federation of Information Processing, the IFIP Congress, held in Ljubljana in 1971, the present one is the biggest, most important, and best organized professional meeting in this part of the world, concerning relational information systems and enterprise organization. It links already substantially the conceptualism existing between the business and organization area within information processing; however, it links not only two politically and economically different worlds, from the western and the eastern parts of the Continent, but also ideas how to improve information systems and organization in developed and developing countries.

Inspecting the program of your congress I have come across some crucial topics which cannot be ignored anymore in the information management and organization. For instance, management consulting cannot ignore the strategy of a company; design of a management information system has to consider the organizational structure and vice versa; engineering of information systems and organization concepts can hardly be separated; informational and organizational sciences are on the unique way to make a developing enterprise successful. Both information systems and organizational concepts are parts of economic systems which will have to take new shapes, particularly in the countries of Eastern Europe.

Information and organization have to be planned on a strategic level, using the existing and new strategic and intelligent concepts by which enterprises adjust their behavior on the market, in society, in international affairs, and particularly in the globalization of their activity. Creating strategies and developing strategic plans are highly intelligent informational processes which depend essentially on another informational category, called the informational ability of human actors. At this point, satisfactory strategies delivering management, organization, and behavior of success, confront with existing and new informational methodologies, performing the metalevel of knowledge, intelligence, and art of organization.

On the basis of adequate information, information systems and organization of various structures can determine, condition, and enable the control of vital enterprise activities in various domains, from strategic thinking to efficacious organization. Information crucially impacts organization and organization by itself signals deviations, changes, and facts as information. Here the cycle closes: and that becomes the subject of this congress.

In the next few years, European countries will have to adjust their organization on the enterprise level according to the new political and economic paradigm of the internal market of the European Communities. This accommodation will call for restructuring and reorganization of management and business of European and other enterprises which will participate in the common European market. Thus, information systems for newly shaped organizational models of enterprises will have to be restructured and innovated as well. From this indispensable task new business and organization strategies will arise, calling for more efficient, advanced, intelligent, and globally structured information. In this way, all these facts will contribute to the rise of new methodologies for shaping the information systems of the next century. In the next years, the European economic, technologic, and cultural integration will hopefully bring new informational and organizational harmony to our everyday life. So, let this harmony be also the hope for a prosperous advance of information systems and organization.

At the end I wish you a pleasant staying in Bled and in this part of Slovenia, for which I believe, is friendly and picturesque.

Thank you!

O računalniški industriji v Sloveniji

V razdobju gospodarske krize se postavlja tudi vprašanje, ali je računalniška industrija v Sloveniji smiselna, mogoča oziroma izvedljiva in če je, v katerih mednarodnih, nacionalnih in korporativnih okvirih. To vprašanje glede na razvojno relevantnost in razvojne možnosti nacionalne populacije seveda ni zanemarljivo, saj se vanj vmešča vprašanje o tem, kaj bo ta populacija zmogla sama in kakšna bodo njena izhodišča v skupni evropski poslovni areni.

Nekatera uradna in pa tudi alternativna politična izhodišča se nagibajo v prepričanje, da bi bilo najbolje razprodati obstoječa domača podjetja na zunanjih trgih, saj je realsocialistična zgodovina teh podjetij pokazala, da sama niso sposobna doseči evropske ravni proizvodnje in njene kakovosti. Za bistveno tržno preusmeritev teh podjetij je namreč preostalo premalo časa oziroma so bile že izčrpane kapitalske možnosti. To bi seveda pomenilo, da bi se morala tudi t.i. drobna računalniška podjetja neogibno navezati na zunanje partnerje oziroma tuje trge in industrijo - seveda pod pogoji evropske konkurenčnosti. Kaj bi to lahko pomenilo za avtohtono kulturo populacije zlasti na njenem informacijskem področju, je seveda mogoče tudi predvidevati.

Politično lažji sestop v mednarodno razprodajo računalniške industrije in njenega trga je seveda lahko le posledica določene nesposobnosti, ki ni le individualna in populacijska, temveč je tudi upravljavska (vodstvena, politična, odločitvena). Alternativa temu sestopu bi lahko bilo široko industrijsko povezovanje v evropskem prostoru, ki začasno lahko nosi izkoriščanost v pogojih določenih polkolonialnih razmerij, vendar v svoji perspektivi stremi v lastno zmogljivost in proizvodno kakovost (pogojno samostojnost) po preteku določenega časa. V tej zvezi so prav gotovo poučni primeri prav računalniških industrijskih podjetij Daljnjega vzhoda. Strategija takšnega razvoja pa bi morala biti jasno začrtana tako na državni kot korporativni ravni.

Drugo vprašanje, ki se postavlja pa je tudi, kakšne bi bile prednosti dobro organizirane in mednarodno uspešne domače računalniške industrije. Ta industrija bi seveda vezala nase t.i. drobna računalniška podjetja v fleksibilen poslovni sistem, ki se prilagaja pogojem razvitega trga. Ta fleksibilnost pomeni, da se v razdobjih konjunkture veže na vodilne industrijske konglomerate večje število drobnih podjetij, da narašča tudi zaposlenost. V razdobjih krize pa se navezanost drobnih podjetij in zaposlenost zmanjšujeta, tako da industrijski konglomerati lahko preživijo in kasneje zopet povečujejo navezanost in zaposlenost.

V novi paradigmi industrijskega razvoja bi morali na podoben način obravnavati zlasti nacionalne razvojne projekte in prek njih financirana raziskovalna in razvojna podjetja. Bazna nacionalna industrija, v katero se uvršča tudi računalniška industrija, mora biti v prihodnje povezana v enovit kompleks mednarodne trgovine in industrije, tj. v posebno vladno ministrstvo, ki organizira mednarodno trgovino (zlasti izvozno) in pripadajočo industrijo skozi vrsto ukrepov, državnih in industrijskih konzorcijev in seveda z aktivnostjo v mednarodni politiki. To pa je seveda nekaj povsem drugega, kot je bila državna administracija vajena doslej. Ko danes politično premišljamo o lažjem sestopu v razprodajo industrije, bi morali enako kritično razmisliti tudi o tem, kako bomo zagotovili kakovost tistih državnih organov, ki bodo sposobni spremljati dogajanja in ustrezno ukrepati ob izzivih, ki se bodo pojavljali na evropski industrijski sceni.

Nacionalna računalniška industrija naj bi bila tedaj trdno vmeščena v evropsko razvojno shemo

mednarodni (evropski) pogoji - državna strategija - korporacija - mednarodna in domača računalniška industrija - razvojna in drobna računalniška podjetja,

hkrati pa vpeta v državno koordinacijo zunanje trgovine, bržkone predvsem ali vsaj zaenkrat pretežno v evropskem prostoru. V sedanjem trenutku je tedaj potrebno tako prilagajanje industrije kot državnega (trgovinskega) instrumentarija skupaj s finančnimi in tržno-raziskovalnimi in tehnološko-raziskovalnimi potenciali.

Anton P. Železnikar

Razmislek o Iskri kot korporaciji

Anton P. Železnikar

Zakaj sodelovanje javnosti?

Pred tedni me je znanec, ekonomist-finančnik prepričeval, kako je vsakršnja javna polemika o konkretnih podjetjih in podjetništvu nepotrebna in škodljiva, ker da vpliva z javno besedo lahko le škodljivo na trenutno kritični tržni položaj podjetja. Izkušnje potrjujejo, da je v obstoječih upravljavskih trdnjavah podjetniške administracije izredno težko premikati okostenelo in nekonkurenčno miselnost, ki se oklepa prikritih privilegijev in nesposobnosti skupaj znešenih vodstvenih ekip. Cilj teh ekip tudi v tem gospodarskem trenutku ni kaj več kot načrt obvladovanja reformnih podjetniških tokov v prihodnosti tako, da se bodo ti ujeli s prikritimi načrti razvojno preseženе ekipne oblasti. Principu stagnacije se podrejajo največkrat tudi razvojne študije, naročene pri domačih elitnih svetovalnih organizacijah, ki se zavestno poslovno prilagajajo kroju-interesu naročnika.

Iskra je v razdobju zadnjih desetletij veljala za prvo slovensko državno industrijsko podjetje. Začudo pa država tega svojega podjetja, ki je lahko veljalo za državno že zaradi znatnega prelivanja kapitala brez posebnih pravic in odgovornosti vanj, ni urejala mandantno-upravljavsko, npr. v smeri razvoja v sodobno korporacijo razvitega sveta. Korporativna miselnost Iskre, ki je v zgodnjih letih njenega razvoja še nastajala, se je pod različnimi pritiski in politično vsiljenimi (represivnimi) in že sposobnostno erodiranimi vodstvenimi ekipami razkrojila in je danes brezciljna, zameglena ali nična. Ekipe, sestavljene iz poslušnih in neinventivnih uradnikov, so tako v glavnem trošile le svoj delovni čas, si nabirale leta na imenitnih položajih in nikoli niso poskušale prebijati nesmiselnih političnih blokad. Saj tega zares tudi niso zmogle, tako kot so bile kvalificirane, nepovezane ali celo skregane z ekonomskim, finančnim, inženirskim in pravnim intelektom v Iskri in izven nje.

Naštevanje vsega tega, ki ni novo in ni za nikogar več presenetljivo, seveda razen poslovnega spomina nekega podjetja, nima spodbudnega razvojnega pomena, lahko pa ohranja pedagoško zavest. Kaže pa na to, da je javna razprava o slovenskem industrijskem razvoju nujna, podobno kot ekološka, saj naj bi bila Iskra to, kar velja za ekološko sprejemljivo na področju elektronike in v okviru elektronike za telekomunikacije, računalništvo in avtomatizacijo. Zadušeni in neaktivirani poslovni, ekonomski, finančni, tehnološki in pravni intelekt Iskre naj bi zaživel tudi skozi možnosti javnega izražanja, Iskra pa bi ga morala z vsemi sredstvi oživljati in spodbujati, tudi v svojih vodstveno cenzuriranih javnih občilih. Tudi zato, da se ne bi ponovila javnosti neznana afera, ko je bil zaradi povsem upravičene javne polemike doktor elektrotehniških znanosti in v strokovnih krogih mednarodno uveljavljen strokovnjak kazensko premeščen v skladišče z nalogo raznašalca materiala svojim strokovno manj podkovanim kolegom, ki so ta absurd cinično prezrli in se v spopadu z direktorjem varno potuhnili. Tako temu skromnemu in razgledanemu strokovnjaku ni preostalo kaj drugega, kot da je Iskro zapustil in s tem oslabil že tako pičlo maso kritičnega javnega uma Iskrinih inženirjev.

Zakaj naj bi Iskra postala korporacija?

Korporacija je združenje, ki ščiti pravice svojih članov- podjetij, zastopa njihove interese in uresničuje njihove poslovne cilje; hkrati pa je tudi oblika **državne gospodarske aktivnosti** in oblika upravljanja z javno lastnino. Status korporacije imajo tista podjetja, ki v okviru svoje gospodarske dejavnosti zastopajo tudi interese države, so pod nadzorom državnih organov in javnosti in so le v tem okviru podjetniško samostojna. Korporacija je posledica nacionalizacije podjetij in sodobnih državnih kapitalskih razmerij. Skozi to je korporacija vselej tudi instrument državne gospodarske politike in raznovrstne (naložbene, kapitalske, davčne, poslovne) intervencije. Korporacija je tedaj posledica praktičnih potreb populacije in ne predvsem posameznika, ima svojsko socialno uravnoteženost in smiselnost.

Člani-podjetja Iskre se že danes organizirajo kot holding ali krovna podjetja po vzoru Iskri podobnih industrijskih podjetij v svetu (Philips, Hewlett-Packard, Siemens). Krovno podjetje ima v svoji aktivi le akcije drugih podjetij (delniških družb), opravlja pa finančne operacije in izvaja kontrolne in upravljavske naloge svojih podjetij na področju finančne, razvojne, proizvodne in komercialne aktivnosti. Seveda pa krovno podjetje omogoča tudi delničarstvo različnim slojem prebivalstva. Krovno podjetje plasira svoje deleže v vsako svoje podjetje, kjer razpolaga s t.i. kontrolnim paketom delnic, ki pa npr. ne presega petine skupne vrednosti delnic delniškega podjetja. Tako krovno podjetje kontrolira kapital svojih podjetij in denarne prihranke prebivalstva, ki so vloženi v delnice. Bistveni kazalci uspešnosti krovnega podjetja postanejo tako dobiček, prihodek na zaposlenega, donosnost iz prometa, donosnost poslovnih sredstev in kazalci rasti (povečevanje ali upadanje vrednosti naštetih kazalcev).

V delniški družbi, ki je v okviru krovnega podjetja, je kapital delničarjev združen z namenom, da se dosegajo nekateri učinki, ki na ravni individualnega podjetja niso mogoči: dejavnosti, ki zahtevajo velika sredstva za proizvodnjo; dejavnosti, s katerimi se proizvodnja posodablja in povečuje njena produktivnost iz sredstev skupne akumulacije; obstajajo tudi prednosti pri najemanju bančnih kreditov zaradi zanesljivejšega plasmaja in možnosti vračanja kreditov. Delniško podjetje temelji na prostovoljni centralizaciji raznovrstnega kapitala, s tem pa tako ali drugače podružblja tudi poslovanje delniške družbe.

Podobno kot krovno podjetje združuje dogovorjene skupne funkcije svojih delniških družb, združuje korporacija strateške in poslovne funkcije svojih krovnih organizacij. Danes je domala vsaka pomembnejša korporacija tudi multinacionalna, tako da se v njen prepletajo kapitali njenih nacionalno različnih podjetij in je večkrat v njej prisotna tudi udeležba državnega kapitala. Mešanje nacionalnih, podjetniških in individualnih kapitalov je mogoče na vseh podjetniških ravneh, tj. v korporaciji, krovnem in podkrovnem podjetju.

V prid korporativnemu načinu organiziranja Iskre govorijo preprosto primeri organizacije podobnih podjetih v razvitem svetu. Izmišljevanje nekih subkulturnih oblik podjetniške organiziranosti nas ne more pripeljati prav daleč. In prav v tej točki se postavlja vprašanje kvalificiranosti dosedanjega vodstva Iskre, da vztraja pri sedanji ali neki svojeglavi organizaciji, s katero se podjetja, ki jo konstituirajo, npr. ne strinjajo. Tu pa se seveda začenja tudi vprašanje, zakaj si intelektualci Iskre želimo, da bi imeli na vseh nivojih vodenja podjetij mednarodno uveljavljene standarde kakovosti, in sicer tako za način vodenja in seveda za kvalifikacijo samih vodstvenih delavcev. Leto kakovosti 1989 naj bi tako uvajalo tudi kakovost na področju poslovanja, organiziranja in vodenja. Z mednarodnimi standardi kakovosti vodenja pa bi se moral sprijazniti tudi tisti del politike, ki je bil doslej prepričan, da mora v podjetja represivno filtrirati nesposobnostno preverjene ekipe. Saj se menda ne moremo več sprenevedati ob nuji, da tekmovanje v evropski poslovni areni postaja naš lasten imperativ preživetja.

Sodobni principi vodenja podjetij

Vprašanje, ki ga je smiselno postaviti, je, kateri principi vodenja so lahko odločilni pri vodenju Iskre kot korporacije in v njo vključenih holding podjetij. Ti principi veljajo tako za predsednika korporacije kot za generalne direktorje holding podjetij.

Določene načelne ali principske in posebne skrbi za poslovanje in razvoj podjetja v slovenski elektronski industriji nismo negovali. To bržkone tudi ni bilo potrebno, saj je bilo mogoče zasilno shajati s povsem drugimi, v glavnem nesposobnostnimi, tudi koruptnimi prijemi. V novih pogojih poslovanja pa postaja skrb za vzdrževanje in vrednotenje podjetniških principov podjetja tudi pri nas čedalje nujnejša in generalni direktorji se ji ne bodo mogli več izogibati. Kakšni so ti principi, o katerih razpravlja harvardski profesor A. E. Pearson, bivši predsednik PepsiCo v Harvard Bussiness Review (julij/avgust 1989)?

Dober športni trener poudarja vselej specifična osnovna pravila, npr. temeljne spretnosti in igro, ki omogočajo njegovemu moštvu, da zanesljivo zmaguje. Dober generalni direktor (GD) postopa podobno. Zaveda se, da nenehne, nadpovprečne zmogljivosti ni mogoče vzdrževati s kratkotrajnimi izboljšavami, kot so prestrukturiranje, vztrajno zniževanje stroškov ali reorganizacija. Seveda je mogoče tudi tovrstne radikalne posege izvajati takrat, ko postanejo ti potrebni in zaželeni. Nujno pa je, da GD preprečuje, da bi do takih radikalnih situacij sploh prišlo. To preprečevanje temelji na uporabi šestih principov, ki oblikujejo delovno osnovo oziroma razvid dela vsakega GD:

oblikovanje delovnega okolja,

postavljanje strategije,

razporejanje virov,

usposabljanje direktorjev,

graditev organizacije in

pregled nad operacijami.

Ta seznam naj ne bi bil presenetljiv; pri vsem tem naj bi bila delovna vsebina generalnega direktorja več ali manj znana stvar. Ti principi ne bi smeli biti pozabljeni, saj pomenijo okvir dela GD, določajo prednostne naloge in videnje medsebojnih odnosov med naštetimi delovnimi področji. Pri nas pa smo imeli primere, ko generalni direktorji praktično niso bili oblikovalci delovnega okolja sozda ali ozda, niso bili postavljalci poslovne in tehnološke strategije, so spontano ali sproti razporejali vire, zanemarjali šolanje direktorjev in samih sebe, podirali organizacijo ali jo praviloma nenačelno in pogosto spreminjali in tudi niso zmogli, kljub računalniškim pripomočkom, pregleda nad finančnimi, proizvodnimi in tržnimi operacijami.

Vsako podjetje ima svoje posebno delovno okolje, svojo legalnost iz preteklosti, ki določa, kako reagirajo direktorji na probleme in priložnosti. Neglede na velikost podjetja je delovno okolje podedovano iz preteklosti in njegovo oblikovanje - ali preoblikovanje - je kritično pomembna naloga generalnega direktorja.

Trije elementi določajo delovno okolje podjetja:

veljavni (sedanji) standardi zmogljivosti, ki pogojujejo korak in kakovost naporov uslužbencev;

poslovni koncepti, ki opredeljujejo, kaj pod jetje je in kako deluje; in

koncepti in vrednote uslužbencev, ki so prevladujoče in določajo način dela.

V teh treh elementih so standardi zmogljivosti najpomembnejši, saj določajo kakovost naporov, ki jo izkazuje organizacija podjetja. Če ima GD visoke standarde, bodo imeli take standarde tudi ključni direktorji. Če so standardi generalnega direktorja nizki in megleni, tudi podrejeni ne bodo imeli dosti boljših standardov. Visoki standardi so tako osnovni pripomoček, s katerim GD vpliva in vzpodbuja talente uslužbencev v okviru celotnega poslovanja podjetja.

Če tedaj podjetje ali divizija podjetja že ima zahtevne standarde - in nekatera podjetja in divizije to že imajo - bo največji prispevek GD k trenutnim rezultatom in dolgoročnemu uspehu podjetja prav v dvigu zmogljivostnega pričakovanja, tako pri uslužbencih kot pri samemu sebi. To pa pomeni zavestno odločitev o naslednjih vprašanjih:

katera otipljiva merila pogojujejo večjo zmogljivost;

v kakšnem položaju se podjetje nahaja v danem trenutku; in

ali je GD pripravljen na energične pozive in potrebne korake, da izvleče podjetje iz nastale situacije.

Seveda je eden izmed najpomembnejših standar-dov GD oblikovanje ciljev podjetja. Dober GD oblikuje cilje in z njimi vpliva na organizacijo, da te cilje prilagaja in jih tudi dosega. Ti cilji tako niso naključni in nerealistični, tako da usihajo in ne motivirajo, temveč so takšni, da opozarjajo, kako neizprosna je konkurenčna arena.

Visoki standardi lahko izhajajo le iz več kot zahtevnih ciljev. Vrhunski trenerji, vojaški vodje ali dirigenti simfoničnih orkestrov, vrhunski direktorji vselej postavljajo sebe kot primer, in sicer po neomejenem delovnem času,

s svojim prepričanjem v uspeh in

z **nepretrgano kakovostjo** svojih ukrepov.

Tako GD postavlja in krepi visoke standarde z malimi koraki, ki se hitro povečujejo.

V slovenskem prostoru prav gotovo imamo takšne predsednike in generalne direktorje. Njihova energija, askeza, racionalnost in sposobnost je za direktorje in uslužbence zgledna. Ko sem pred nedavnim opozarjal naše politike na nujnost delegiranja takih GD tudi v Iskro, sem imel pred seboj referenco Gorenja, ki se je v zadnjih letih razvilo v evropsko podjetje in je na poti v Evropo. Nobena skrivnost ni, da je Iskra tu zaostala, saj je bilo njeno vodenje spontano, neambiciozno in takšno, kot so ga lahko oblikovale njene vodilne ekipe in predvsem njeni predsedniki. Vesel bi bil dokaza, ki bi to moje prepričanje spodbijal, namreč, da Iskra kot de facto državno podjetje ni imela Gorenju podobnih možnosti in svoje lastne razvojne moči.

Česa se niso naučili iz preteklosti?

Ko danes analiziramo preteklost Iskre in njeno sedanjo neperspektivnost, ko vodilna ekipa Iskre priporoča stečaje svojih ozdov in pri tem ne pomišlja na odstop, kot da za stečajne trende, ki se vlečejo iz preteklosti, ni odgovorna, ko veliko bolj uvidevno odstopajo vodstva njenih ozdov zaradi neprimernega poslovanja, se sprašujem, zakaj se vodstvena neodgovornost in neprimernost korporativne Iskre nadaljuje in se lahko prav v tem negativnem procesu načrtuje njena prihodnja podjetniška učinkovitost in organiziranost. Siemens je npr. v letu 1988 zaradi neznatnega upadanja poslovnih kazalcev, poslovne razmehčanosti in perspektiv enotnega evropskega trga po letu 1992 sprožil največji pretres svojega menagementa, s ciljem ekspanzije v mednarodne operacije in posodobitve programa s podporo multinacionalnih partnerjev. Pridobivanje novih poslov, združevanje in skupni posli so dejavnosti, ki so se uvrstile v vrh programa novega Siemensa. Stisnjeno v ožajoče pogoje, spodbujeno s trenutno donosnostjo in ujeto v tržne situacije, ki postajajo čedalje bolj specifične, je italijansko podjetje Olivetti v letu 1988 sprožilo masivni program prestrukturiranja in odpoklicalo svoje poslovneže in organizatorje domov (npr. odpoklic V. Cassonija iz ameriškega AT&T). Podjetje Groupe Bull, v katerem ima francoska vlada 92% holding, je realiziralo globalno posest francoskega ljudstva nad prvim ameriškim računalniškim podjetjem Honeywell. Vsi ti signali - notranji in zunanji - so razen redkih odstopov ohranili monolitnost Iskrine vodstvene ekipe in s tem preprečevali pozitivne trende, na katere smo inženirji uradnike Iskre v zadnjih letih opozarjali. Javnost - politična in strokovna - naj tedaj razsodi, česa se niso naučili in kaj jim je sedaj storiti.

Opomba

Članek je bil z neznatnimi spremembami objavljen v Delu pod naslovom **Sposobni direktorji namesto nesposobnostno preverjenih ekip,** dne 30. septembra 1989, Sobotna priloga, stran 30.

Edward Feigenbaum

Interviewed for Expert Systems

by Kenneth Owen

... He is a cofounder of TeKnowledge and of IntelliGenetics (now Intelli-corp), and co-author of several books, including *The Fifth Generation* (1983) and *The Rise of the Expert Company* (1988).

... a spin-out group from ICOT called EDR (Electronic Dictionary Research Institute) in Japan which is funded by the government. The view of the Japanese researchers was: we can't make progress on natural language processing by programming the syntax of language in Prolog. That's not the issue, the issue is: understanding implies knowledge; where's the knowledge? So their project is to produce a semantic network of five hundred thousand words and seven hundred and fifty thousand concepts. And they have one hundred million dollars and seven years to do it.

... However, there is a strong emotional pull to the work of neural networks, and scientists are moved by emotion just like other creative poeple.

These are only some of communication and information systems activities offered to you from the

Iskra Delta Development Division™

Stegne 15C, Ljubljana, Yugoslavia

Phone: (+38 61) 57 45 54
Telex: 31366 yu delta
Fax: (+38 61) 32 88 87 and (+38 61) 55 32 61
E-mail: rri@idc.uucp

This Division can become your main profitmaking growth engine in the field of communication and information systems (C&IS) and related services. For needs of your company and together with you we can

– minimize risk-growth through R&D in C&IS domain within corporate culture
– extend your current C&IS technology
– be your holding company for C&IS ventures
– be your joint R&D venturer through minority investments and aquisition
– create new C&IS business for you
– modernize your C&IS technology and penetrate it into new markets
– become your subsidiary dedicated to your C&IS products and services diversification
– bring new C&IS business into your operative divisions
– reach quality and quantum jump by break-through C&IS research
– develop for you C&IS technology penetration strategy
– discover your C&IS capability without burdening you

Iskra Delta Development Division™ has 150 skilled professional researchers (PhD's, MSc's), market strategists, engineers, economists, and lawyers at disposals for your company to extend your business operations in the field of communication and information systems.

Prof. Dr. Anton P. Zeleznikar

Ljubljana, October 12, 1989

Executive Manager

Informatica

Editor – in – Chief

ANTON P. ŽELEZNIKAR
Iskra Delta Computers
Production Center
Stegne 15C
61000 Ljubljana

PHONE: (+38 61) 57 45 54
TELEX: 31366 yu delta
FAX: (+38 61) 32 88 87 and
(+38 61) 55 32 61
ELECTRONIC MAIL:
ike@idcyuug.uucp

Associate Editor

RUDOLF MURN
Jožef Stefan Institute
Jamova c. 39
61000 Ljubljana

Informatica *is published four times a year in Winter, Spring, Summer and Autumn by the Slovene Society Informatika, Iskra Delta Computers, Stegne 15C, 61000 Ljubljana, Yugoslavia.*

Volume 14 Number 1 January 1990 YU ISSN 0350 – 5596

Informatica

A Journal of Computing and Informatics

CONTENTS