4de

SUID-AFRIKAANSE REKENAARSIMPOSIUM
SOUTH AFRICAN COMPUTER SYMPOSIUM

HOLIDAY INN PRETORIA
JULIE 1 – 3 JULY 1987
Proceedings
of the
4th South African Computer Symposium

Holiday Inn, Pretoria
1 – 3 July 1987

edited by

Pieter Kritsinger
Computer Science Department
University of Cape Town
PREFACE

Computer science is an emerging discipline which is having difficulty in being recognised as a worthy member of the sciences. I will paraphrase John Hopcroft, co-winner of the 1986 Turing Award, when, during a recent interview, he said that the primary reason for the lack of recognition, is the age of our researchers. Probably not one of the researchers who presented their work at this symposium is older than 45. I know of no computer scientist in South Africa who is in a position where (s)he can affect funding priorities. As far as I know we have no representation on any of the committees of the Foundation for Research Development and for our Afrikaans speaking fraternity, none who is a member of the Akademie vir Wetenskap en Kuns. It will take time and conscious effort to establish our presence. The same is true of course for our universities. Again, with one exception, I know of no dean of a science faculty, vice-principal or principal who is a computer scientist. We consequently spend an enormous amount of time trying to explain the needs of computer science and its difficulties. I believe this symposium is a further step towards accreditation by our peers and superiors from the other sciences.

The total number of papers submitted to the Programme Committee for consideration was 34. Each paper was reviewed by three persons knowledgeable in the field it represents. Of those submitted, 23 were finally selected for inclusion in the symposium. As a result the overall quality of the papers is high and as a computer science community in Africa we can be justly proud of the final programme.

This is the fourth in the series of South African computer symposia. This year the symposium is sponsored by the Computer Society of South Africa (CSSA), the South African Institute for Computer Scientists and the local IFIP Committee. The executive director of the CSSA and his staff deserve warm thanks for handling the organisation as well as they have, while the Organising Committee provided Derrick and I with very valuable advice.

Finally I would like to express my sincere appreciation to the authors, to the members of the Programme Committee and particularly the reviewers. Without the kind cooperation of everyone, this symposium would not have taken place.

Pieter Kritsinger
SYMPOSIUM CHAIRMAN: PS Kritzinger, University of Cape Town.
SYMPOSIUM CO-CHAIRMAN: D Kourie, University of Pretoria.
MEMBERS OF THE PROGRAMME COMMITTEE

Judy Bishop, Witwatersrand University
Chris Bornman, UNISA
Hannes de Beer, Potchefstroom University
Gideon de Kock, Port Elizabeth University
Jaap Kies, Western Cape University
Derrick Kourie, Pretoria University
Pieter Kritzinger, Cape Town University
Tony Krzesinski, Stellenbosch University
Michael Laidlaw, Durban Westville University
Peter Lay, Cape Town University
Ken MacGregor, Cape Town University
Theo McDonald, Orange Free State University
Jan Oosthuizen, University of the North
Dennis Riordan, Rhodes University
Alan Sartori-Angus, Natal University
John Shochot, Witwatersrand University
Theuns Smith, Rand Afrikaans University
Trevor Turton, ISM (Pty) Ltd
Gerrit Wiechers, Infoplan.
LIST OF REVIEWERS

BERMAN Sonia
BISHOP Judy
BORNMAN Chris
CAREY Chris
CHERENACK Paul
DE BEER Hannes
DE VILLIERS Pieter
GORRINGE Pen
KIES Jaap
KOURIE Derrick
KRITZINGER Pieter
KRZESINSKI Tony
LAIDLAW Michael
LAY Peter

MacGREGOR Ken
MATTISON Keith
McDONALD Theo
RENNHACKKAMP Martin
RIORDAN Denis
SATORI-ANGUS Alan
SCHOCHT John
SMITH Theuns
TURTON Trevor
VAN DEN HEEVER Roelf
VAN ROOYEN Hester
VON SOLMS Basie
VOS Koos
**TABLE OF CONTENTS**

**Keynote Address**

"An Extensible System and Programming Tool for Workstation Computers." ............................................. 1  
Niklaus Wirth, ETH, Zurich

**Invited Lectures**

"The Relationship of Natural and Artificial Intelligence." .................. not included in Proceedings.  
G Lasker, University of Windsor, Ontario.

"Software Engineering: What Can We Expect in the Future?" ................ not included in Proceedings.  
D Teichrow, University of Michigan, U.S.A.

**Computer Languages I**

"SPS-Algol: Semantic Constructs for a Persistent Programming Language." ................................. 13  
S Berman, University of Cape Town.

"Petri Net Topologies for a Specification Language." .............................. 25  
R Watson, University of the Witwatersrand.

"Towards a Programming Environment Standard in LISP." ...................... 45  
R Mori, University of Cape Town

"ADA for Multiprocessors: Some Problems and Solutions." ................... 63  
J Bishop, University of the Witwatersrand.

**Computer Graphics**

"Polygon Shading on Vector Type Devices." ................................. 75  
C F Scheepers, CSIR.

"Hidden Surface Elimination in Raster Graphics Using Visigrams." ..................... 97  
P Gorringe, CSIR.

**Database Systems I**

"On Syntax and Semantics Related to Incomplete Information Databases." ................................. 109  
M E Orlowska, UNISA.

"Modelling Distributed Database Concurrency Control Overheads." ..................... 131  
M H Rennhackkamp, University of Stellenbosch.

**Operating Systems**

"The Development of a Fault Tolerant System for a Real-time Environment." ................................. 149  
M Morris, CSIR.

"A New General-purpose Operating System." ................................. 161  
B H Venter, CSIR.
Computer Languages II

"The Representation of Chemical Structures by Random Context Structure Grammars." ................ 175
E M Ehlers and B von Solms, RAU.

"A Generalised Expression Structure." ............... 189
W van Biljon, CSIR.

Computer Networks and Protocols I

"An Approximate Solution Method for Multiclass Queueing Networks with State Dependent Routing and Window Row Control." .................. 203
A E Krzesinski, University of Stellenbosch.

"A Protocol Validation System." ....................... 227
J Punt, University of Cape Town.

Computer Networks and Protocols II

"Protocol Performance Using Image Protocols." ......... 251
P S Kritzinger, University of Cape Town.

Artificial Intelligence

"A Data Structure for Exchanging Geographic Information." ................................. 267
A Cooper, CSIR.

"The Design and Use of a Prolog Trace Generator for CSP." ................................. 279
D G Kourie, University of Pretoria.

Database Systems II

"An Approach to Direct End-user Usage of Multiple Databases." .............................. 297
M J Phillips, CSIR.

"A Semantic Data Model Approach to Logical Data Independence." .......................... 329
S Berman, University of Cape Town.

Information Systems

"The ELSIM Language: an FSM-based Language for the ELSIM SEE." .......................... 343
L du Plessis and C Bornman, UNISA.

"Three Packaging Rules for Information System Design." .................. 363
J Mende, University of the Witwatersrand.
"Experience with a Pattern-matching Code Generator." ... 371
M A Mulders, D A Sewry and W R van Biljon, CSIR.

"Set-oriented Functional Style of Programming." ........ 385
C Mueller, University of the Witwatersrand.

Tutorial

The use of Modula-2 in Software Engineering." ............ 399
N Wirth, ETH, Zurich.
### DAY 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>07h30</td>
<td>Registration and Coffee.</td>
</tr>
<tr>
<td>08h45</td>
<td>Welcoming address, President of the South African Institute of Computer Scientists, Dr. G. Wiechers.</td>
</tr>
<tr>
<td>09h00</td>
<td>Invited Lecture, Professor D. Teichrow, University of Michigan.</td>
</tr>
<tr>
<td></td>
<td>Software Engineering, ... What Can We Expect in the Future.</td>
</tr>
<tr>
<td>10h00</td>
<td>COFFEE</td>
</tr>
<tr>
<td>10h15</td>
<td><strong>Computer Languages I.</strong> Chairman: G. Wiechers.</td>
</tr>
<tr>
<td>10h50</td>
<td>S. Berman, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><strong>SPS-Algol: Semantic Constructs for a Persistent Programming Language.</strong></td>
</tr>
<tr>
<td>10h50</td>
<td>A. Watson, University of the Witwatersrand.</td>
</tr>
<tr>
<td></td>
<td><strong>Petri Net Topologies for a Specification Language.</strong></td>
</tr>
<tr>
<td>11h25</td>
<td>R. Mori, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><strong>Towards a Programming Environment Standard in USP.</strong></td>
</tr>
<tr>
<td>11h50</td>
<td>J. Bishop, University of the Witwatersrand.</td>
</tr>
<tr>
<td></td>
<td><strong>ADA for Multiprocessors: Some Problems and Solutions.</strong></td>
</tr>
<tr>
<td>12h30</td>
<td>LUNCH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>14h00</td>
<td><strong>Computer Graphics.</strong> Chairman: D. Kourie</td>
</tr>
<tr>
<td></td>
<td>C. F. Scheepers, CSIR.</td>
</tr>
<tr>
<td></td>
<td><strong>Polygon Shading on Vector Type Devices.</strong></td>
</tr>
<tr>
<td>14h35</td>
<td>P. Gorringe, CSIR.</td>
</tr>
<tr>
<td></td>
<td><strong>Hidden Surface Elimination in Raster Graphics Using Visigraphics.</strong></td>
</tr>
<tr>
<td>15h15</td>
<td>COFFEE</td>
</tr>
<tr>
<td>15h30</td>
<td><strong>Database Systems I.</strong> Chairman: B. von Solms.</td>
</tr>
<tr>
<td></td>
<td>M.E. Orlowska, UNISA.</td>
</tr>
<tr>
<td></td>
<td><strong>On Syntax and Semantics Related to Incomplete Information Databases.</strong></td>
</tr>
<tr>
<td>16h05</td>
<td>M.H. Rennhakkm, Stellenbosch University.</td>
</tr>
<tr>
<td></td>
<td><strong>Modelling Distributed Database Concurrency Control Overheads</strong></td>
</tr>
<tr>
<td>18h00</td>
<td>Cocktail Party in Cullinan Room A.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>14h00</td>
<td><strong>Operating Systems.</strong> Chairman: R. MacGregor.</td>
</tr>
<tr>
<td></td>
<td>M. Morris, UNISA.</td>
</tr>
<tr>
<td></td>
<td><strong>The Development of a Fault Tolerant System for a Real-time Environment.</strong></td>
</tr>
<tr>
<td>14h35</td>
<td>B. H. Venter, CSIR.</td>
</tr>
<tr>
<td></td>
<td><strong>A New General-purpose Operating System.</strong></td>
</tr>
<tr>
<td>15h15</td>
<td>COFFEE</td>
</tr>
<tr>
<td>15h30</td>
<td><strong>Computer Languages II.</strong> Chairman: J. Bishop.</td>
</tr>
<tr>
<td></td>
<td><strong>The Representation of Chemical Structures by Random Context Structure Grammars.</strong></td>
</tr>
<tr>
<td>16h05</td>
<td>W. van Biljon, CSIR.</td>
</tr>
<tr>
<td></td>
<td><strong>A Generalised Expression Structure.</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>18h00</td>
<td><strong>Cocktail Party in Cullinan Room A.</strong></td>
</tr>
</tbody>
</table>
## DAY 2

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
</table>
| 08h30 | **Keynote Address** by Professor Niklaus Wirth, Swiss Federal Institute for Technology, Zurich.  
**An Extensible System and a Programming Tool for Workstation Computers.** |  
**Computer Networks and Protocols I.** Chairman: P.S. Kritzinger. |
| 09h30 | **A.E. Krzesinski, University of Stellenbosch.**  
**An Approximate Solution Method for Multiclass Queueing Networks with State Dependent Routing and Window Flow Control.** |  
**Computer Networks and Protocols I.** Chairman: P.S. Kritzinger. |
| 10h05 | **J. Punt, University of Cape Town.**  
**A Protocol Validation System.** |  
**Coffee** |
| 10h30 | **Computer Networks and Protocols II.** Chairman: A. van der Heever. |  
**P.S. Kritzinger, University of Cape Town.**  
**Protocol Performance using Image Protocols.** |  
**Invited Lecture by Professor G. Lasker, University of Windsor, Ontario.**  
**The Relationship of Natural and Artificial Intelligence.** |  
**Lunch** |
| 14h00 | **Artificial Intelligence.**  
Chairman: G. Lasker.  
**A. Cooper, CSIR.**  
**A Data Structure for Exchanging Geographic Information.** |  
**Information Systems.**  
Chairman: D. Teichrow.  
**L. du Plessis and C. Bornman, UNISA.**  
**The ELSIM Language: an FSM-based Language for the ELSIM SEE.** |  
**Database Systems II.**  
Chairman: C. Bornman.  
**R. I. Newcombe, University of Cape Town and R. Rado, National Library of Medicine, Maryland.**  
**Strategies for Automatic Indexing and Thesaurus Building.** |  
**COFFEE** |
| 14h35 | **A. Cooper, CSIR.**  
**R. I. Newcombe, University of Cape Town and R. Rado, National Library of Medicine, Maryland.**  
**Strategies for Automatic Indexing and Thesaurus Building.** |  
**Computer Languages III.**  
Chairman: N. Wirth.  
**W. van Biljon, CSIR.**  
**Experience with a Pattern-matching Code Generator.** |  
**Database Systems II.**  
Chairman: C. Bornman.  
**R. I. Newcombe, University of Cape Town and R. Rado, National Library of Medicine, Maryland.**  
**Strategies for Automatic Indexing and Thesaurus Building.** |  
**COFFEE** |
| 15h15 | **COFFEE** |  
**Computer Languages III.**  
Chairman: N. Wirth.  
**W. van Biljon, CSIR.**  
**Experience with a Pattern-matching Code Generator.** |  
**Set-oriented Functional Style of Programming.** |
| 15h30 | **M.J. Philips, CSIR.**  
**An Approach to Direct End-user Usage of Multiple Databases.** |  
**Computer Languages III.**  
Chairman: N. Wirth.  
**W. van Biljon, CSIR.**  
**Experience with a Pattern-matching Code Generator.** |  
**Set-oriented Functional Style of Programming.** |
| 16h05 | **S. Berman, University of Cape Town.**  
**A Semantic Data Model Approach to Logical Data Independence.** |  
**Computer Languages III.**  
Chairman: N. Wirth.  
**W. van Biljon, CSIR.**  
**Experience with a Pattern-matching Code Generator.** |  
**Set-oriented Functional Style of Programming.** |
| 16h45 | **Open Forum with professors G. Lasker, D. Teichrow and N. Wirth.**  
Moderator: Dr. D. Jacobson. |  
**Symposium Banquet in Cullinan Room.**  
**Guest speaker, Dr. D. Jacobson, Group Executive: Technology, Allied Technologies Limited.** |
<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>08h00</td>
<td>Registration (Tutorial only).</td>
</tr>
<tr>
<td>08h30</td>
<td>Tutorial.</td>
</tr>
<tr>
<td></td>
<td>The Tutorial will be given by professor Niklaus Wirth, Division of Computer Science, Swiss Federal Institute of Technology, Zurich.</td>
</tr>
<tr>
<td></td>
<td><em>The use of Modula-2 in Software Engineering.</em></td>
</tr>
<tr>
<td></td>
<td>Topics to be covered include:</td>
</tr>
<tr>
<td></td>
<td>What is Software Engineering?</td>
</tr>
<tr>
<td></td>
<td>Data types and structures.</td>
</tr>
<tr>
<td></td>
<td>Modularization and information hiding.</td>
</tr>
<tr>
<td></td>
<td>Definition and implementation parts.</td>
</tr>
<tr>
<td></td>
<td>Separate compilation with type checking.</td>
</tr>
<tr>
<td></td>
<td>Facilities to express concurrency.</td>
</tr>
<tr>
<td></td>
<td>Pompous programming style.</td>
</tr>
<tr>
<td></td>
<td>What could be excluded?</td>
</tr>
<tr>
<td>12h15</td>
<td><strong>Close of Symposium.</strong></td>
</tr>
<tr>
<td>12h30</td>
<td><strong>LUNCH</strong></td>
</tr>
</tbody>
</table>
An Extensible System and a Programming Tool for Workstation Computers

N. Wirth

Introduction

When observing widely used operating systems, one is compelled to conclude that they are only moderately suited to modern workstations which provide a high degree of interactivity by means of their high-resolution displays and pointing devices such as the mouse. Clearly, they are descendants of systems stemming from the era of remote batch processing. Even when multiple processes and multiple windows are considered, this observation cannot be entirely dismissed.

Another observation is that modern systems have emerged as extensions of older generations, the extensions being grafted onto facilities that were not designed with those extensions in mind. A direct consequence is the cancerous growth of their size and complexity. The phenomenal progress in hardware technology has not only enabled but also fostered the emergence of huge software systems. There has been no corresponding new technology to cope with the simultaneous growth of complexity, however. It has unfortunately led to unreliable designs, because the mastering of such difficult designs is beyond the capability of most (perhaps all) programmers. It has also led to systems that are difficult to use, because their specification is usually both incomplete and too voluminous to digest. This holds for almost everything in the domain of modern computing, from specifications of circuits (chips), to that of computer architectures, instruction sets, programming languages, operating systems, text editors, and application packages. The fatalistic acceptance of this deplorable state by the computing community is a sad and astounding phenomenon. I have repeatedly witnessed people not only use but also buy workstations which require no less than half a million bytes of system code in mainstore to execute a null program. More significantly, the customer reaches this remarkable goal only after having learnt a myriad of system specific incantations.

Instead of speculating about reasons – or perhaps more significantly about the driving forces – which led to such monolithic blocks of code and descriptions, and about how future systems ought to be designed, I will report on a project undertaken during the last year with the aim to obtain a reasonably simple, yet powerful workstation. The software was developed by J. Gutknecht and myself, the inherently limiting manpower thus rigorously eliminating any temptation to compete with industrial megalomania.

The paradigm

Severe limitations of available resources have one healthy consequence: they force to concentrate on the essential and to ignore the rest. Our main paradigm was that programming is extending a given system. This means that our goal had to be the design of a conveniently extensible kernel, and to discern those facilities that are fundamental and therefore have to be part of the kernel. It also became evident rather quickly that this kernel (and its later extensions) had to be expressed in terms of a notation supporting the notion of extension in a manner more general than present in existing languages. Inevitable, the project therefore expanded into the area of language design and encompassed the design of a new language and its compiler. Once again, our basic guideline was concentration on the essential and elimination of the rest. As a starting ground, we chose Modula-2. The crucial additions of the derived language are constructs to express a new data type as an
extension of another type that typically is declared in another module, and a construct to discriminate among related types. It thus becomes possible to extend an existing system with modules - separately compiled - that contain not only procedural extensions, but also data type extensions.

It would clearly have defeated our efforts from the start, had we tried to build on top of an existing system. Hence, the hardware, devoid of any special purpose attachments and resident system code, was also tailored to our attitude and built by a small team, H. Eberle and I. Noack. The computer Ceres is based on the NS32032 processor and features a main store of 2 and a disk store of 40 MBytes capacity, a display with 1024 x 800 pixels, and a keyboard and a mouse as input devices.

The software of the new system, called Oberon, was developed using the personal computer Lilith. Downloading to Ceres is accomplished through the modest facilities of a 9600 baud serial link and a 300 byte program resident in the Ceres ROM.

Concurrent in interactive systems

In modern operating systems, concurrent processes are a primary topic: several tasks can be executed (quasi-) concurrently and the processor(s) can be switched from one to another task at any moment. Upon closer inspection, this latter facility does not appear to be essential, at least if one relinquishes the possibility of background tasks. The user certainly wishes to operate on several sets of data (called objects) which need be accessible (and even visible) simultaneously. However, we maintain that it generally suffices to perform one operation on an object x followed by an operation on an object y (possibly different from x). Thus we allow for a much coarser granularity of interleaving of processes and delegate the choice of task switching explicitly to the user who selects the sequence of operations interactively. The unit of granularity is the procedure. It is evident that the elimination of switches occurring at totally unpredictable places in a program offers a substantial simplification of a system's realization.

A natural postulate is that the user may conveniently select possible operands and commands. The former are preferably sets of data that are visible and therefore can be selected on the screen by pointing. This requires a display with multiple viewers and the disconnection of data from particular programs. In the same manner, the user should be able to select commands by simply pointing at a text designating the desired operation. Commands denote procedures (exported from modules), and hence the operating system must be able to activate any such procedure at any time.

Several essential corollaries emerge from this view:

1. The concept of a "main program" has vanished. The unit of action is not represented by a program but by procedures.

2. The double role of modules as loading unit and program has vanished. The module constitutes a collection of procedures and possibly encapsulates a set of variables representing system state.

3. The body of a module does no longer form a program, upon whose termination the space occupied by the code and the data of the module is relinquished. The body rather assumes the modest role of initialization statement for the module's (global) variables. It is executed when a module is loaded.

4. Data objects are generated dynamically and not "owned" by a module or program. No indication is provided by the user signalling that the object will henceforth not be referenced (and if were given, it could not be trusted). Hence, an automatic retrieval system (garbage collection) is mandatory. We envisage that the collector is activated between commands only. This implies that the roots of collectible data structures are restricted to global pointer variables, which simplifies the bookkeeping and accelerates garbage collection significantly.

5. The use of garbage collection requires safety against breaches of the language's type concept. The
reason for this is that the collector must have access to information about type and structure of collectible data. Corruption of this information by programming errors must be impossible. Because of the lack of a guarantee of the absence of programming errors, this implies execution-time checks for validity of array indices and references via pointers. In particular, Modula-2's type transfer functions and variant records have become intolerable constructs.

6. Programming errors detected at execution-time (traps) do not terminate a program and cause the loss of its data and the computed results. A trap merely terminates the command currently being obeyed. Global data remain intact.

**Editing as a central facility**

The established view, namely that the operator selects programs to be executed, reading sets of data and generating new sets of data, is replaced by the view which centers on the data to which sequences of individual commands are applied. Selected data are visible on the screen in various viewers (windows). There must be no hidden state (mode) depending on the history of preceding commands. The questions “where am I”, “where do I come from”, and “what can I do” become irrelevant, and much of the traditional confusion of computer usage disappears [1].

The operations applicable to visible data are basically those of editing: selection, insertion, deletion, copying, and combinations thereof. This applies to textual as well as graphic objects. The important addition is that a selected text may be interpreted as a command. As a consequence, the text (and graphics) editor assumes a distinguished role. Its commands have to be available at any time and to be applicable to any selectable object. This does not imply that it be implemented in any different form than other commands which can be added to a system at any time just as programs can be added to a conventional operating system. But it implies that (some of) its commands may be activated by a more rapid mechanism than by selecting a piece of text for interpretation. An obvious candidate for such a distinguished selection facility are the keys of the pointing device (mouse), that becomes an integral part of such a system. We have adopted the following unifying definition of the meaning of mouse keys:

- **Left key:** positioning of caret
- **Middle key:** interpretation of selected text
- **Right key:** selection of object

A few further commands – particularly for the handling of graphic objects – are available by combination of mouse keys.

A welcome simplification of implementation is achieved by the use of tiling the display into individual viewers instead of allowing them to be partially overlapped, as shown by the Cedar system [2]. Hence, viewers are always totally visible, or not visible at all. For the latter case, a displaced viewer remains accessible by its name in a list of closed viewers, and it can be re-opened upon demand. The screen is divided into so-called tracks, vertical columns of viewers. Just as new viewers can be opened in a given track, new tracks can be opened on the screen. In the third dimension, we may open a new layer on top of the existing set of contiguous tracks. This scheme provides for a large degree of flexibility.

**The Read problem**

The request for the elimination of modes which the system assumes upon certain commands and which determine the interpretation of subsequent commands has profound consequences. For example, the conventional Read statement requesting a keyboard input stands in direct contradiction to this concept, because the system inherently enters a specific state, namely that of waiting for data, and it allows no alternative (re)action than typing a key.

The conventional Read statement mirrors the view of the sequence of keyboard characters like
an input stream from a tape or a disk. Whereas the stream of visible output to a display can indeed be treated in the same manner as a stream to a tape or disk, this analogy seems quite mistaken in the case of input. The reason is that a Read statement expresses the request for an action of the human partner, whereas a Write statement does not. The human operator and the computer are in essence cooperating sequential processes, alternating coroutines. A Read statement should release control of the (cooperative) process to the operator, just as pressing a key (at the keyboard or the mouse) returns control to the computer.

This demand is quite fundamental. Imagine, for instance, that a conventional Read statement requests a numeric parameter for a subsequent computation. The operator may well not immediately know its value, but may wish to calculate it from other data at hand. For this very purpose he needs to have the full capabilities of the computer at his disposal, before the process requesting the parameter can proceed. A more familiar example is the request for a file name; the operator may not remember it correctly and wishes to inspect the file directory. Once again he needs the full flexibility of the computer, before the process requiring the file name can proceed. In both examples, the conventional Read statement, blocking the computer until the fixed request has been honoured, is entirely inadequate.

An obvious solution would seem to be a generalized Read statement, permitting any kind of interposed action until the final reply is provided. However, this solution is unacceptable, because it would leave the issuing process "hanging" in a state, which from the above mentioned considerations is undesirable. A logical consequence would be that the interposed process could again issue a Read request, and so on, leading to a whole chain of processes whose states have to be kept in mind by the operator.

A workable solution is to select parameters of commands from (displayed) text. This text is input in a conventional fashion. The pressing of any key is interpreted as the command to insert a character at the current position of the caret. The character value is seen as the parameter of the insert command and is determined by the identity of the pressed key.

Commands hence accept parameters in the form of selections, they issue output through Write statements, but there is no such thing as a Read statement.

The system's structure

The view of a single computing process consisting of a sequence of procedure activations determined by the (human) operator applied to coexisting objects (data, system state) lead to a relatively unconventional system structure. The core of the Oberon system is a "small" loop in which input events are monitored. Input events are signalled by the pressing of a key at the keyboard or the mouse. Control is then passed to a procedure previously assigned to the object to which the input is directed. The core first determines that addressee. If the input event is the pressing of a mouse key, the addressed object is determined by the current cursor position (i.e. the viewer in which the cursor is located). If it is a keyboard event, the addressee is the current carrier of the input focus, the viewer designated to receive subsequently typed characters. Each viewer record therefore contains a field whose value is a procedure. Assigning of a procedure is called installing the procedure.

Because the caller does not rely on the identity of the installed procedure, but simply activates whatever procedure is present, it has become customary to refer to this kind of procedure activation as sending a message. The caller merely signals the occurrence of a certain event, and "sends a message" by activating an installed procedure supposedly performing the appropriate reaction. The actual parameters of the installed procedure, which supply characteristics of the event (system state) are called messages. This terminology has been introduced in Smalltalk along with the notion of object-orientation [3].

We emphasize, however, that the notions of object, message, and sending correspond to well established concepts of conventional programming, namely dynamically allocated, record typed
variable, procedure parameter, and calling a (formal) procedure. The new terminology merely reflects a different view of system structure (sometimes pompously called inverse programming). We also emphasize that the language Oberon in no way forces this view upon programmers. In fact, many parts of the Oberon system use conventional calls of exported procedures, thus avoiding the overhead intrinsic in message passing, i.e. calling of formal procedures. We use the additional flexibility where needed, but avoid it where it merely causes additional overhead.

The same holds for data: objects are variables that persist, survive individual procedure (program) activations. The dynamic nature of their allocation, the indirectness of referencing them, and the problems of their removal when no longer referenced, cause intrinsic overhead. We avoid it by relying on conventional techniques when they suffice. Messages, which are parameters, are allocated together with local variables in a conventional stack. Retrieval of associated storage space consists of a simple resetting of the stack pointer upon termination of the procedure.

With the elimination of the notion of program, modules assume the role of collections of procedures. Global variables appear to vanish. In fact, they remain, assuming the important role of roots for the data structures consisting of "objects". Typically, there exist only a few such roots, and hence global variables become rather rare. The module body serves to initialize them.

Fig. 1 displays the module structure of the Oberon system after a text editor, a graphic editor, and possibly other "tools" have been loaded. Arrows denote imports. The outer box delineates the collection of modules present after system initiation. It contains the procedures for managing the system's resources for loading modules, creating and closing viewers, writing and selecting text, and for activating commands. The inner box contains the modules required minimally for loading other modules. It is displayed in greater detail in Fig. 2.

The module loader allows both to request modules - to load them from disk unless present in main store - and to call commands exported from modules. Upon loading a module M, all modules imported by M are also referenced, but not necessarily loaded. Actual loading is delayed until one of the procedures is activated for the first time. This facility separates the conventional static interdependence of modules from their common presence in store. (Execution of the null program therefore does not require the actual presence of an entire operating system!) This highly desirable flexibility has become feasible through the use of virtual address mapping provided by the memory management unit of the processor.

The kernel module combines those routines that require special processor instructions or that must be highly efficient. The Kernel and the Bitmap modules are the only parts of the system that are expressed in assembler code. Bitmap raster operations and garbage collection require highest efficiency, whereas storage management routines reside in the kernel, because they use special hardware facilities. Disk sector and memory page reservation tables are protected from access in user mode. The only routines executed in supervisor mode are those of the kernel.

An interesting technical detail concerns the method of getting Oberon loaded and started. When developing an operating system, it is important to retain as much flexibility as possible. This implies to refrain from fixed, built-in facilities, to assume the presence of (almost) no software, to build on top of the bare hardware. In our case, the system is started by activating a minimal boot loader, resident in ROM and loading a boot sequence from either a serial line or from a fixed track of the disk. The modules shown in Fig. 2 are those loaded by the boot loader, cross-compiled and linked on the development tool (a Lilith computer). Control is first transferred from the boot loader to a routine of the kernel which initializes the page and sector reservation tables, the interrupt controller, and the virtual memory manager. Thereafter the loader is activated to load the module Oberon containing the central input loop. As a consequence, all modules imported by Oberon are also loaded. In passing, it may be mentioned that the modules contained in the boot file do not refer to any output device (and were therefore to be developed without the availability of such). Evidently, these modules would be the primary candidates for being located in the ROM (in place of the boot loader).
Type extensions

If the guiding principle in the design of a system is concentration on the essential, then this same principle must also apply to the tools and in particular to the language in which the system is specified. Clearly, a concept so fundamental as extensibility of data types should not be expressed in some contortuous fashion (mis)using available (low-level) facilities, but calls for an appropriate language addition that harmoniously fits into the existing body.

The basis of the data extensibility concept is the record type. Let a record type $T$ be given. Then the declaration

$$T' = \text{RECORD (} T \text{) ... END}$$

defines a new type $T'$ which (directly) extends $T$. $T$ is called the (direct) base type of $T'$. Let, for example, $T$, $T0$, and $T1$ be defined as

$$T = \text{RECORD} \ x, y: \text{INTEGER END};$$
$$T0 = \text{RECORD (} T \text{)} \ z: \text{REAL END};$$
$$T1 = \text{RECORD (} T \text{)} \ b: \text{BOOLEAN}; \ ch: \text{CHAR END}$$

Then $T0$ has a field $z$ and $T1$ has fields $b$ and $ch$ in addition to $x$ and $y$; hence $T0$ and $T1$ extend $T$.

Extended types can be re-extended, and we define extension in more generality as follows:

A type $T'$ extends a type $T$, if $T = T'$ or if $T'$ directly extends an extension of $T$. Accordingly: $T$ is a base type of $T'$, if $T = T'$ or if $T$ is the direct base type of a base type of $T'$.

For example, a type $T00$ can be defined as

$$T00 = \text{RECORD (} T0 \text{)} \ w: \text{LONGREAL END}$$

and hence has the fields $x, y, z, \text{and } w$.

We may conveniently consider a data type $T$ as the set of all instances (variables) of this type. Then the instances of an extension $T'$ are those members of $T$ that carry the additional properties (fields) specified by $T'$. They form a subset of $T$. The relationship between a data type and its extensions is therefore appropriately expressed by set inclusion. Applied to the foregoing example, we obtain

$$T00 \subseteq T0 \subseteq T \quad T1 \subseteq T$$

This view also explains the rule of assignment which we adopt from Modula-2 but must interpret in the light of extended types: The value of an expression $e$ can be assigned to a variable $v$, if the type of $e$ extends the type of $v$ (which includes equality). Given the declarations

$$\text{VAR } v: \text{T}; \ t0: \text{T0}; \ t00: \text{T00}$$

the assignments $t := t0; \ t := t00; \ t := t00$ are admissible, whereas $t0 := t; \ t00 := t; \ t0 := t$ are not. It follows, that for example $t := t0$ must stand for

$$t.x := t0.x; \ t.y := t0.y$$

implying that $t0.z$ remains uncopied.

The genuine value of type extensions appears in conjunction with pointers, because only here the situation may arise where the actual extension type of a variable is not visible from the program text. We naturally extend the above definition to pointer types (bound to records).

The one and only operation genuinely required is a facility to discriminate among related types.
It is called a *type test* and is a Boolean expression of the form $v \IS T$. The symbol $\IS$ is classified as a relation. The full power of this facility emerges from the possibility to use it freely as a factor in logical conjunctions and disjunctions.

For a more complete discussion of extended types and their relationship to other, similar notions, we refer to [1]. Here it may suffice to give an example of an application relevant in the Oberon system, namely that of Viewers. Let a type `Viewer` specify its screen coordinates, width, and height:

```plaintext
TYPE Viewer = RECORD x, y, w, h: INTEGER END
```

This type may then be extended by types such as `TextViewer` or `GraphViewer` with additional properties, including handler procedures constituting operations on the data displayed in the respective viewer.

The crucial point is that extensions can be declared in modules different from that in which the base type is declared. If we regard programming as extending a given (compiled and linked) system, it must be possible to construct new types on the basis of existing ones, and to devise them without regard to possible further extensions (that might be made years later).

**From Modula-2 to Oberon**

For obvious reasons, we based our work on incorporating a type extension facility on Modula-2 and its compiler developed inhouse. Our intimate familiarity with both language and compiler provided ample temptation to grasp the opportunity not only to extend Modula-2 by this indispensable facility, but also to include a few more changes induced by programming experience gained in the last eight years of using Modula [4].

Concentration on the essential inherently implies the elimination of the inessential. The elimination of certain features rarely used is of course an effective method to simplify the effort of constructing a compiler and to make it more reliable. In fact, the decision to eliminate certain language constructs quickly led to a chain reaction, finally leading to a state where it was felt that the result should be given a new name. However, Oberon still clearly shows Modula as being its ancestor. Subsequently we give a list of the features eliminated and a few brief comments about them.

- **Variant records.** They are the genuine sore point providing ample opportunities for breaching the type consistency concept. A system using automatic garbage collection is inherently unsafe in the presence of variant records [1].

- **Local modules.** They proved to be rarely used, yet cause a considerable complication in the language specification and (naturally) also compiler implementation.

- **The FROM clause in import lists.** We consider it as preferable to denote each reference to an imported object $x$ explicitly as $M.x$, instead of allowing a simple identifier $x$ to be globally qualified by $M$ through the import clause `FROM M IMPORT x`. This again simplifies both the language definition and the compiler without loss in substance.

- **Subrange types.** They complicate the language definition and in particular cause a deviation from the rule that the type of any object be visible from the program text and be disjoint from other types. Their benefit is marginal, providing implicit range checks on assignment, but not additional substance.

- **Enumeration types.** The argument against their presence is similar. Their value rests largely upon additional implicit redundancy. However, they cause considerable further problems related to import and export, for example giving rise to the exceptional rule that import of the type identifier implies the import of all associated constant identifiers. Furthermore, they contradict
the desire for extensibility.

- Set types. With the absence of subrange and enumeration types, there is little justification left for a general SET constructor. It is therefore sacrificed as well. However, a single standard type SET is provided in its place, denoting the sets whose elements are the integers between 0 and, say, 31. Operations on sets directly correspond to logical operations present in all instruction sets.

- Array types are restricted to indices which are integers. Furthermore, the lower index bound is fixed to zero. In place of bounds, the array declaration specifies the length of the array. We consider this elimination of an implicit, yet restricted index mapping as quite acceptable in view that no essential expressive power is given up, and that a simplification of the compiler and a gain in program efficiency are obtained.

- The type CARDINAL is eliminated, thus avoiding the confusing type incompatibility problems between CARDINAL and INTEGER expressions. The sacrifice is easily accepted on systems that offer long integers, and where no mapping between addresses and integers is required.

- Type transfer functions (coercions) hide implicit assumptions about data representation and render programs unportable. They defy any effort to achieve a safe system, particularly in the presence of automatic storage retrieval.

- The types ADDRESS, WORD, and BYTE seduce programmers to introduce (often unnecessary) machine dependence, and they are incompatible with the goal of a type-safe programming language.

- For statements are eliminated, because they provide little advantage and (in the form without bound, local control variable) constitute no expressive power beyond that of the other repetitive statements.

- With statements were discarded for similar reasons: the additional complexity in language definition and compiler appear incommensurate with the little advantage gained. Explicit qualification of record fields is in harmony with explicit qualification of imported objects.

Oberon also contains a few facilities not present in Modula-2:

- Multidimensional open array parameters are highly desirable for applications involving matrices. Without this facility, matrices can be passed as parameters, only if an exceptional rule of parameter compatibility rule is postulated. Moreover, the explicit programming of the index mapping function is cumbersome and requires the explicit provision of additional length parameters.

- The types LONGINT, SHORTINT, and LONGREAL pass on to the programmer facilities that are present in many, if not most modern computers.

The reduction of the number of Modula-2's constructs, and the addition of facilities harmoniously fitting into the existing framework results in a language that - without reference to an implementation - can be specified in a concise and precise report. No further elaboration - such as a standardization - should be necessary.

Conclusions

We believe that in order to approach the goal of satisfying the unbounded expectations many people have, future computer systems must become more sophisticated and at the same time simpler. The present Oberon system is presented as an attempt to proceed with this premise in mind. Its development required a small investment of resources and a modest amount of manpower, but
rests upon the willingness to concentrate on essential, fundamental concepts and to eliminate what adds no or little substance. It also required the same willingness in the preparation of the necessary tool and the mental flexibility to adapt to a new environment of programming.

This approach stands in contrast to the development of ever larger systems through grafting onto existing software. We firmly believe that this method will not provide any genuine progress because of the inherent unreliability of the resulting contrivances. Computer systems are no longer limited by processor power or memory size, but only by our (in)ability to construct and use them properly.

Acknowledgement

I wish to express my thanks to J. Gutknecht without whose cooperation and contributions the Oberon system would not have been possible. In particular I am indebted to him for the many fruitful discussions and suggestions, for his uncompromising critical scrutiny, and for his obstinate insistence on getting rid of what turned out to be merely conventional rather than convenient.

References

Fig. 1. Structure of the Core of the Oberon System
Fig. 2. Structure of the Boot-Modules