4th

SUID-AFRIKAANSE REKENAARSSIMPOSISIUM
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
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  MacGREGOR Ken
BISHOP Judy  MATTISON Keith
BORNMAN Chris  McDonALD Theo
CAREY Chris  RENNHAACKKAMP Martin
CHERENACK Paul  RIORDAN Denis
DE BEER Hannes  SATORI-ANGUS Alan
DE VILLIERS Pieter  SCHOCCHOT John
GORRINGE Pen  SMITH Theuns
KIES Jaap  TURTON Trevor
KOURIE Derrick  VAN DEN HEEVER Roelf
KRITZINGER Pieter  VAN ROOYEN Hester
KRZESINSKI Tony  VON SOLMS Basie
LAIDLAW Michael  VOS Koos
LAY Peter
# 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-Algo·l: 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


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

Computer Networks and Protocols I


Computer Networks and Protocols II


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.

Computer Languages III

"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>
</tbody>
</table>
| 09h00 | Invited Lecture. Professor D. Teichrow, University of Michigan. 
*Software Engineering, ... What Can We Expect in the Future.* |
| 10h00 | **Computer Languages I.** Chairman: G. Wiechers.                                           |
| 10h15 | S. Berman, University of Cape Town. 
*SPS-Algo: Semantic Constructs for a Persistent Programming Language.* |
| 10h50 | A. Watson, University of the Witwatersrand. 
*Petri Net Topologies for a Specification Language.* |
| 11h25 | A. Mori, University of Cape Town. 
*Towards a Programming Environment Standard in USP.* |
| 11h50 | J. Bishop, University of the Witwatersrand. 
*ADA for Multiprocessors: Some Problems and Solutions.* |
| 12h30 | LUNCH                                                                                     |
| 14h00 | **Computer Graphics.** Chairman: D. Kourie                                                |
| 14h35 | C. F. Scheepers, CSIR. 
*Polygon Shading on Vector Type Devices.* |
| 15h15 | **Operating Systems.** Chairman: K. MacGregor.                                             |
| 15h30 | M. Morris, UNISA. 
*The Development of a Fault Tolerant System for a Real-time Environment.* |
| 16h05 | P. Gorringe, CSIR. 
*Hidden Surface Elimination in Raster Graphics Using Visigraphics.*                  |
| 16h05 | **Computer Languages II.** Chairman: J. Bishop.                                             |
| 17h30 | M.E. Orlowska, UNISA. 
*On Syntax and Semantics Related to Incomplete Information Databases.*               |
| 18h00 | **Database Systems I.** Chairman: B. von Solms.                                            |
| 16h05 | W. van Biljon, CSIR. 
*A Generalised Expression Structure.* |
<p>| 18h00 | <strong>Database Systems II.</strong> Chairman: B. von Solms.                                            |
| 18h00 | Cocktail Party in Cullinan Room A.                                                        |</p>
<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>08h30</td>
<td>Keynote Address by Professor Niklaus Wirth, Swiss Federal Institute</td>
</tr>
<tr>
<td></td>
<td>for Technology, Zurich.</td>
</tr>
<tr>
<td></td>
<td>An Extensible System and a Programming Tool for Workstation Computers.</td>
</tr>
<tr>
<td></td>
<td><strong>Computer Networks and Protocols I.</strong> Chairman: P.S. Kritzinger.</td>
</tr>
<tr>
<td>09h30</td>
<td>A.E. Krzesinski, University of Stellenbosch.</td>
</tr>
<tr>
<td></td>
<td>An Approximate Solution Method for Multiclass Queueing Networks</td>
</tr>
<tr>
<td></td>
<td>with State Dependent Routing and Window Flow Control.</td>
</tr>
<tr>
<td>10h05</td>
<td>J. Punt, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td>A Protocol Validation System.</td>
</tr>
<tr>
<td>10h30</td>
<td><strong>COFFEE</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Computer Networks and Protocols II.</strong> Chairman: A. van der Heever.</td>
</tr>
<tr>
<td>11h00</td>
<td>P.S. Kritzinger, University of Cape Town.</td>
</tr>
<tr>
<td>11h35</td>
<td>Invited Lecture by Professor G. Lasker, University of Windsor, Ontario.</td>
</tr>
<tr>
<td></td>
<td><strong>The Relationship of Natural and Artificial Intelligence.</strong></td>
</tr>
<tr>
<td>12h30</td>
<td><strong>LUNCH</strong></td>
</tr>
<tr>
<td>14h00</td>
<td><strong>Artificial Intelligence.</strong></td>
</tr>
<tr>
<td></td>
<td>Chairman: G. Lasker.</td>
</tr>
<tr>
<td></td>
<td>A. Cooper, CSIR</td>
</tr>
<tr>
<td></td>
<td><strong>A Data Structure for Exchanging Geographic Information.</strong></td>
</tr>
<tr>
<td>14h35</td>
<td>A. I. Newcombe, University of Cape Town and A. Noda, National Library</td>
</tr>
<tr>
<td></td>
<td>of Medicine, Maryland.</td>
</tr>
<tr>
<td></td>
<td><strong>Strategies for Automatic Indexing and Thesaurus Building.</strong></td>
</tr>
<tr>
<td>15h15</td>
<td><strong>COFFEE</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Database Systems II.</strong></td>
</tr>
<tr>
<td></td>
<td>Chairman: C. Bornman.</td>
</tr>
<tr>
<td>15h30</td>
<td>M.J. Philips, CSIR</td>
</tr>
<tr>
<td></td>
<td><strong>An Approach to Direct End-user Usage of Multiple Databases.</strong></td>
</tr>
<tr>
<td>16h05</td>
<td>S. Berman, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><strong>A Semantic Data Model Approach to Logical Data Independence.</strong></td>
</tr>
<tr>
<td>16h45</td>
<td><strong>Open Forum with professors G. Lasker, D. Teichrow and N. Wirth.</strong></td>
</tr>
<tr>
<td></td>
<td>Moderator: Dr. D. Jacobson.</td>
</tr>
<tr>
<td>19h30</td>
<td>Symposium Banquet in Cullinan Room.</td>
</tr>
<tr>
<td></td>
<td>Guest speaker, Dr. D. Jacobson., Group Executive: Technology,</td>
</tr>
<tr>
<td></td>
<td>Allied Technologies Limited.</td>
</tr>
<tr>
<td></td>
<td><strong>Information Systems.</strong></td>
</tr>
<tr>
<td></td>
<td>Chairman: D. Teichrow.</td>
</tr>
<tr>
<td></td>
<td>L. du Plessis and C. Bornman, UNISA.</td>
</tr>
<tr>
<td></td>
<td><strong>The ELSIM Language: an FSM-based Language for the ELSIM SEE.</strong></td>
</tr>
<tr>
<td>14h00</td>
<td>J. Mende, University of the Witwatersrand.</td>
</tr>
<tr>
<td></td>
<td><strong>Three Packaging Rules for Information System Design.</strong></td>
</tr>
<tr>
<td>15h15</td>
<td><strong>COFFEE</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Computer Languages III.</strong></td>
</tr>
<tr>
<td></td>
<td>Chairman: N. Wirth.</td>
</tr>
<tr>
<td>15h30</td>
<td>W. van Biljon, CSIR</td>
</tr>
<tr>
<td></td>
<td><strong>Experience with a Pattern-matching Code Generator.</strong></td>
</tr>
<tr>
<td>16h05</td>
<td>C. Mueller, University of the Witwatersrand.</td>
</tr>
<tr>
<td></td>
<td><strong>Set-oriented Functional Style of Programming.</strong></td>
</tr>
</tbody>
</table>

**Symposium Banquet in Cullinan Room.**
<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. 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><strong>The use of Modula-2 in Software Engineering.</strong></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>LUNCH</td>
</tr>
</tbody>
</table>
SPS-ALGOL:
SEMANTIC CONSTRUCTS IN A PERSISTENT PROGRAMMING LANGUAGE

Sonia Berman
University of Cape Town

ABSTRACT

Persistent data is defined as that which survives on secondary store after program execution terminates. In a persistent programming language there is no distinction whatever between persistent data and other objects. In contrast to conventional systems, both the data structures and the operations applicable to "permanent" data are identical to those used for short-term data. PS-Algol is the only implemented persistent language at present. This paper describes a proposed extension to this language, SPS-Algol, which incorporates more of the concepts and tools of databases and semantic data models. In particular transactions, tasks, type hierarchies, association, data derivation, semantic integrity constraints and subschemas are proposed as useful extensions to the programming language, and some of the implementation issues involved are discussed.
1. INTRODUCTION

Databases and programming languages have developed almost independently of one another until very recently when the need for an integrated system for programming and data management was recognised. All large systems involve some form of "permanent" data, so programming language design should meet the need for persistent data.

The word persistence describes that property of data that determines how long it should be kept. It is an orthogonal property of data in that any data item may exist for an arbitrarily long time. To illustrate, imagine students writing B-tree type declarations and manipulation routines in Pascal. Unfortunately this code serves only as a model for a real implementation. It cannot be used to exploit secondary storage, since the only persistent data type in Pascal, the file, will not accommodate this structure.

PS-Algol[2] treats all data uniformly irrespective of its longevity. It is thus a "persistent programming language", where the data types and operators associated with ordinary main-store variables are used for persistent data as well. However it is still a conventional programming language, lacking in the semantic integrity features and data model abstractions that are essential in a sophisticated database environment.

This paper describes SPS-Algol, an extension to the language in which improved semantics and abstractions will facilitate the ease and reliability of data manipulation. Sections 2 and 3 serve as a brief introduction to persistence and PS-Algol respectively. The next section describes those semantic database features which are included in the extended version. Thereafter the resulting language SPS-Algol is illustrated and some of the issues involved in its implementation are discussed. In the conclusion, other proposals for persistent languages are outlined [1,10,12] along with outstanding problems which must be solved before persistent languages can entirely replace database management systems.

2. ALTERNATIVE APPROACHES TO PERSISTENCE

In procedural languages the types that persist after program execution are a small subset, or even different from, the types available in shorter time scales. In most languages the only "permanent" objects are files, parameterisable by certain types only (eg Pascal will not allow a file of pointers). Some languages provide a different sublanguage for their long term data - an embedded database language - while others extend the range of persistence by having a mechanism for storing their workspaces.
The simplest form of persistence, provided by interpreters of Lisp, Prolog and APL, is some form of checkpoint-restart instructions. By using these a programmer may save the current state of the environment and resume it later. This is a simple form of all-or-nothing persistence, inadequate for database work especially when sharing of databases is required. It also provides no way for a user to exploit modularity to save parts of his workspace.

Database "languages" are standard programming languages in which database-routine calls are embedded. The embedded and host languages have different naming conventions, notations and types; and it is impossible to check that programs behave consistently.

Traditional programming languages require some file I/O or database management system for their persistent data. Two views of data result from this: short term data manipulated by programming language facilities, and long term data manipulated by the file or database system. The mapping between the two types of data is usually done in part by explicit user translation code. A program typically has 30% of its code concerned with transferring data to and from files or a database [4], e.g. to flatten and rebuild trees or graphs. The data type protection and structure that might have been exploited to aid comprehension is neither apparent nor protected and is soon lost. A persistent programming language seeks to eliminate the difference between the long-term storage and programming language models of data.

In a persistent language the programmer is not constrained to specific, pre-defined data types and structures. This approach takes existing constructs within a programming language and allows them to persist, enabling the programmer to manipulate this "database" in exactly the same way as he processes any other data. Explicit transfers between stores can and should be inferred from the operations on the data. A programmer is distracted from his task when inserting transfer statements. Usage of a database or heap should be transparent to the programmer.

3. PS-ALGOL

PS-Algol is the only implemented language that provides persistence uniformly at present. It is a strongly typed procedural language providing data type completeness (whereby all types have equal status in the language), allowing data of any type to persist. This results in flexibility in the structures that can be constructed and the operations that can be associated with them. The principles on which it is based are [3]:

i) Persistence is a property of objects, not types.
ii) All objects have the same rights to persistence.
iii) While an object persists, so does its description.
iv) The procedure is a first class data object with the usual rights to persistence. This allows programs and data to be bound together and permits sophisticated data hidings and mappings to be implemented. (In PS-Algol a procedure can optionally return a value.)

PS-Algol was developed by adding persistence to S-Algol[7], with minimal changes to the original language. Associating persistence with a variable or type in its declaration would force a programmer to be conscious of whether he is working with a persistent object or not. Thus the system itself automatically decides what data is retained beyond program execution, using reachability as its criterion. Every database has a root object which always persists, along with all other objects reachable by following pointers from this root. The root thus serves as the origin for the transitive closure of (pointer) references. When a database is opened the root object is returned, from which all the persistent data can be found.

For efficient access to major data structures, an indexing method is available in the form of "tables". Tables provide a mechanism for associative lookup and are implemented as B-trees. A table can be visualised as an ordered set of (key, value) pairs, where the value is a pointer to the object associated with that key. The root object of any database is always a table. In keeping with the principle of orthogonality, procedures can be fields of structures, and can also persist. Generally a table is used as a procedure library.

PS-Algol is implemented as a number of functional extensions to S-Algol: OPEN.DATABASE, CLOSE.DATABASE, table manipulation procedures and COMMIT. The latter stores on disk all changes made since program initiation or the last commit. Not COMMITting before program termination is equivalent to aborting.

Experience with PS-algol has shown that both code and coding time are reduced by a factor of three, compared with conventional database systems. Maintenance is also easier and indications are that overall speed of execution increases. Figure 1 is an example of a PS-Algol program.

! A database of people with a list of gradings.
! This program inserts one person with one grading.
STRUCTURE person (STRING name, phone; PNTR grade)
STRUCTURE grading (STRING course; INT mark; PNTR nextgrade)
LET root = OPEN.DATABASE ("gradedb", "Sonia", "write")
IF root IS ERROR.RECORD THEN WRITE "Cannot open database"
ELSE
    BEGIN
        WRITE "Name?"; LET aname = READ.A.LINE()
        WRITE "Phone?"; LET aphone = READ.A.LINE()
WRITE "Course?"; LET acourse = READ.A.LINE()
WRITE "Mark?"; LET amark = READ.A.LINE()
! construct the records
LET agrading = grading(acourse,amark,NIL)
LET aperson = person(aname,aphone,agrating)
! lookup gradetable in root table
LET tablptr = S.LOOKUP ("gradetable",root)
IF tablptr = NIL DO
    BEGIN tablptr := TABLE()
        S.ENTER ("gradetable",root,tablptr)
    END
    ! enter aname with its pointer aperson in gradetable
    S.ENTER (aname,tablptr,aperson)
    COMMIT()
END

Figure 1. A PS-Algol insertion program. The chain root ->
tablptr -> aperson -> agrading ensures all data persists.

4. ADDITIONAL CONSTRUCTS FOR SEMANTIC SUPPORT

In comparison with conventional programming languages, PS­
Algol considerably improves the treatment of persistent
data. However, in the context of database technology, a
number of shortcomings are evident. Thus there is a need for
PS-Algol to be extended to incorporate more database con­
cepts and tools, particularly those recently proposed in
semantic data models.

The following are outstanding constructs which will need to
be incorporated in SPS-Algol to better accommodate
persistence:
(i) transactions to encapsulate a sequence of operations
that behave as a single operation;
(ii) the data abstractions of conceptual data models;
(iii) semantic modelling primitives;
(iv) privacy and access control;
(v) long-term control and use of declarations.

Data models provide three types of data abstraction - aggre­
enables several constituent objects to be viewed collect­
evatively as a compound object - its analogy in Pascal is the
in a type-subtype hierarchy, and is analogous to the is-a
relationship. Thus eg. Person can be defined as a general­
isation of Student which may in turn be a generalisation of
PostGrad, etc. Association (or grouping) defines a set of
objects as another, higher-level object - eg a set of Stu­
dents might be seen as a Committee object. Aggregation is
inherent in PS-Algol; it needs to be extended to support
generalisation and association.

The semantics incorporated in data models are of three
types: integrity constraints; derivations that specify how certain data values are to be computed from others; and tasks that are standard procedures for manipulating objects. None of these is available in PS-Algol to any appreciable extent, and even traditional database languages support only limited forms of constraints and derivations and no notion of tasks.

5. SPS-ALGOL

The constructs identified in the previous section are incorporated in SPS-Algol in the following ways:

(i) Transactions are introduced as blocks of code delimited by the reserved words TBEGIN and TEND. A transaction is a sequence of statements seen as a single logical unit for the purpose of integrity and consistency maintenance. It enables integrity checking to be suspended while a series of mutually dependent operations are executed. Such a situation occurs for example when a new structure is created and its mandatory fields are assigned values in subsequent steps; a valid state can only be reached once all these operations have been performed.

(ii) Association is introduced by extending the type constructors of the language to include SET. To maintain the principle of orthogonality, any variable or field can be a set and the elements of a set can be any type. This gives rise to sets of structures, sets of sets, sets of procedures, etc. As in the relational algebra, set operations are: subset, union("+"), intersection("/"), difference("-"), Cartesian product("*"), desetting("&"), test for membership("IN") and tests for containment ("<", ">", "<="", ">="). In a programming language we also require the ability to reference the current element of a set, for example when iterating through its members. If A is a set then

WHILE A (%A < 10) DO %A := %A * 1.2;

is a loop executed once for each element in the subset A(%A < 10) with %A denoting the current element.

I := &A

is valid only if A is a singleton set, and I is of a type compatible with the element of A.

Generalisation permits subtypes (or "special cases of" types) to be defined (see figure 2). When applied to structure types it is similar to Pascal's variant record, but less restrictive: no tag field is required and the different cases need not be disjoint (eg Staff is-a Person and Student is-a Person, and some Persons are both). Instead of forcing a tag field to distinguish subtype participation, subtypes can be precisely defined via a boolean condition on their fields, according to relationship participation, or as the union, intersection or complement of other subtypes.

Generalisation allows the strict requirement of type equal-
ity to be replaced by the more flexible notion of type compatibility. Type X is compatible with type Y if X is-a Y. An X object can be the actual argument to a procedure that expects a parameter of type Y, as all operations defined for the supertype are defined for the subtype. Similarly, a set or array defined on a type Y can include elements of type X, since every X is-a Y. The latter is analogous to a Pascal array of variant records.

(iii) Data derivations [9] specify a value is to be computed by applying a procedure or an arithmetic expression to some other data. In addition derived arrays and sets can be defined as the subset, union, difference or intersection of other arrays/sets. Derived data is indicated by ACTUAL followed by the expression with which to calculate its value. Consistency is maintained by altering the derived data whenever any item in the expression is changed. If this is too costly VIRTUAL can be used instead of ACTUAL. Such a value is not stored, but computed whenever it is referenced.

```
STRUCTURE student ISA person(INT Year; STRING Degree)
STRUCTURE undergrad ISA student WHERE Year < 4
  (STRING SET coursespassed;
   INT numcredits ACTUAL COUNT(coursespassed);
   INT yearsmore VIRTUAL (10 - numcredits) / 3)
```

Figure 2. SPS-Algol declarations with subtypes and derived data.

In SPS-Algol any declaration can be followed by constraints on that variable or type using CHECK ... END immediately after the declaration. These constraints are expressed as follows:

```
CHECK READPWD x; WRITEPWD y; UNIQUE <fields>;
MANDATORY <fields>; KEY <field groups>;
TOTAL <field> : <structure> ...;
RULES <boolean expressions>;
FDS <fields> -> <fields> ...;
DELETE <fields> CASCADES ...;
DELETE <fields> NULLIFIES ...
END
```

Figure 3. Integrity and security constraints in SPS-Algol.

READPWD and WRITEPWD protect an object - since any program with write access also has read permission, only one of the two need be specified. UNIQUE x and KEY y,z are applicable to arrays and sets, and imply that the field x (or field combination y,z) must be unique over all elements. TOTAL Supplier:SUPPS means that every element of data structure SUPPS must be referenced by some Supplier field, while FDS list functional dependencies that must hold for the elements
of a set or array. MANDATORY fields cannot be left blank (zero, nil) and RULES constrain values so that they are consistent with other information in the environment. There are two alternative deletion constraints that can be specified. These indicate the action required when an object pointed to is deleted, to prevent "dangling references". Either "cascades" - delete the object referencing it as well; or "nullifies" - change the reference to nil.

As conventional systems cannot include code in a database, direct support for tasks is impossible. SPS-Algol with its persistent procedures lends itself to this facility. The idea could be extended so that the only permissible operations on structures are task invocations, resulting in abstract data types. However the ADT definition would have to include every possible way in which such entities might be manipulated, which would be impossible in a database context [3]. In SPS-Algol tasks represent established, standard procedures for manipulating objects that are directly available to programmers; but they are not the only ways in which the data can be used. If necessary a programmer can be limited to task application by giving him only read access to the structure, and read (execute) access to the tasks.

(iv) In keeping with the objective of data type completeness, access control in the form of READPWDS and WRITEPWDS can be defined for types, variables or procedures. Alternative views of data are supported to protect sensitive components of data structures and to permit greater flexibility in the naming of objects. Thus, unlike PS-Algol, a structure with say four fields A, B, C and D can be accessed by a programmer who declares this structure with fields A, B and C only; the remaining field will be transparent to him. He can also name these fields PARTA, PARTB and PARTC if he so desires, as long as the types of the fields are compatible with those in the database. Note that compatible means that if A is of type X, he may declare PARTA of type Y as long as Y is-a X. To provide complete data independence, the order of field declarations should also be immaterial: the effect of this flexibility on performance, however, seems to be too great.

(v) A major problem with PS-Algol is that any information must be declared in a program before it is referenced; including all the persistent types it uses. This generally involves a considerable amount of text. The database solution to this inconvenience can be adopted here: namely, a subschema facility whereby a program need simply INCLUDE <subschema-names> and the declarations are "copied in". This is similar to the C #INCLUDE statement and to database subschemas but allows declarations of both transient and persistent objects. A programmer may INCLUDE a subschema (if he can supply its READPWD) yet not be aware of everything it contains. The ability to access such metadata is
provided by standard functions such as Strucset(X) - all structures in subschema X, and Fieldset(Y) - all fields in structure Y. (A function returning the type of an object already exists in PS-Algo.) In keeping with the principal of orthogonality, subschemas can be persistent or transient, and can be used in function and type constructors.

6. IMPLEMENTATION ISSUES

To support integrity constraints and derivations extra code must be inserted in the compiled program to check or compute values, as the semantics can seldom be handled at compile time. Only the MANDATORY constraint can be verified at compile time by detecting any assignment which would cause a violation. Constraints on sets or arrays are checked only on COMMIT, otherwise performance would degrade sharply. The remaining constraints are checked only when necessary; at the end of a transaction only constraints involving objects altered during that transaction are verified.

Instead of checking functional dependency violations on COMMIT, normalisation theory is adopted to decompose arrays and sets into 3rd Normal Form. Thus if say an array of structures is defined for which A -> B (where A is not a key field), a separate structure array with fields A and B will be created to store these values, and B removed from the original. This is made transparent to the programmer, in the same way as protected fields of a structure are hidden from him.

Dependency information is maintained to determine what must be checked when an object is altered. Rules, deletion constraints and total relationships require checking objects distinct from the object actually altered. These objects are found using "dependency lists" (described below).

Actual storage of derived data causes redundancy in the database, so the system must ensure consistency is always maintained - if A is derived from B, C and D it must be altered whenever B, C or D is changed.

This would be prohibitively expensive for derived sets or arrays. For example if A is defined as a subset of B (where some condition holds) actual storage would require checking A whenever any value in this condition is altered. On the other hand if A is virtual, its value can be derived when necessary by applying the subset condition to B during execution. Similarly, where A is defined as the union, complement or intersection of B and C, it is always more efficient to keep A virtual and construct its value when a reference is encountered, thereby also saving a considerable amount of space.

If a subtype is defined by means of a predicate, such as
"undergrad . ISA student WHERE Year < 4 ", the programmer requires the system to ensure that no object takes on the role of an undergrad unless this condition is satisfied. This presents no difficulties because only fields of the object itself are involved. Verifying participation in a subtype defined as the intersection, complement or union of other subtypes is achieved by checking any conditions on these other subtypes and then ANDing, negating or ORing the results. A subtype defined in terms of relationship participation, such as RiskyStudent "VALUE-OF Incident.Student-Involved" has its membership checked by maintaining reverse pointers (eg from student to incident) which may not be nil.

Dependency information is kept to enable derived data to be maintained consistently. Whenever the value of any object A is altered by the user, the objects dependent on A are determined from a dependency list attached to it, and their values recomputed. This process is recursive, so that if B is dependant on A and C is derived from B, changing A will cause first B and then C to be altered.

Dependencies are complicated by indirect references. Suppose the Percent of a Student is defined as (Mark / Project-Done.Maximum). If a student's Mark or Project-Done is altered, his Percent must be changed accordingly. Furthermore, if any Project's Maximum is changed, all students with the corresponding Project-Done value must have their Percent adjusted. Thus if A is defined in terms of B.C, A is dependent on both B and C.

To maintain consistency "reverse" pointers must sometimes be kept. In the above example altering a Maximum requires changing the Percent of all Students who did that project; hence it is necessary to be able to obtain these students directly by following a reverse pointer. For this reason, each node in a dependency list contains a (reverse) pointer to the actual object to be altered and another to the derivation involved. (For the many derivations involving only fields of one structure, eg Percent ACTUAL Mark / 2, no reverse pointer is needed.) Semantics such as constraints, derivations and subtype definitions are stored in an encoded form so that they can be efficiently accessed by the system.

Sets are implemented as tables for fast access and manipulation. Operations such as SUBSET, UNION etc. utilise their B-tree organisation for more efficient execution. Sets of INT, REAL or STRING do not require any pointer entry in the table; however it is still well worth using this construct even in these cases. This is useful not only for the sake of consistency but also because PS-Algol tables have two important features ideal for sets - their size is not fixed and they are automatically maintained in ascending key order.
7. CONCLUSION

Other languages with objectives similar to SPS-Algol have recently been proposed and partially implemented, namely Taxis[10], Galileo[1] and Adaplex[12]. Adaplex is an extension of Ada which incorporates most of the constructs of Daplex[13], a functional data model. It includes a form of generalisation and provides for transactions and limited integrity constraints, but derivations are not allowed and persistence is associated only (and implicitly) with "entity classes". Taxis is based entirely on the concept of generalisation (with a consequent increase in complexity) and encompasses transactions and certain semantic constraints as well. In contrast to Adaplex, there is very little data indeed which does not persist in Taxis. Galileo also incorporates an extremely flexible form of generalisation and permits certain derivations and constraints. It has no notion of a transaction, and data can only be made to persist by being defined in a database or explicitly added to it at some point.

There is still considerable scope for future work on persistent languages. SPS-Algol and the other conceptual programming languages above do not yet provide for concurrency or for shared usage of persistent data. The problem of support for data evolution also needs to be studied, so that data structures (declared in one program can change in time and be utilised in new ways by other programs. Utilities for data monitoring, recovery and reorganisation must be provided by persistent systems, probably via routines which can operate on arbitrary structures. Such "universal application routines" are currently being investigated by the PS-Algol developers[8].

A counter argument to providing persistence is that it is difficult to find good engineering techniques to support arbitrary persistent structures. Certainly, perhaps because of the research effort expended, the mechanisms for some types such as relations are better understood at present. Efficient methods of managing other persistent structures are now receiving much research attention and the results are encouraging. A computer architecture to support persistent languages is also being investigated [6]. These languages clearly provide a number of research opportunities for both the programming language and database communities. SPS-Algol is one such project aimed at providing the semantic capabilities essential for the persistent environment of the future.

REFERENCES

on Database Syst. 10(2) (1985).


