Proceedings of the Fourth British National Conference on Databases (BNCOD 4)

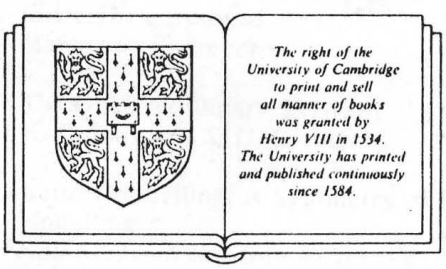
1985

Edited by A. F. GRUNDY

Proceedings of the Fourth British National Conference on Databases (BNCOD 4)

University of Keele, 10–12 July 1985

Edited by A. F. GRUNDY
Department of Computer Science, University of Keele



Published by
CAMBRIDGE UNIVERSITY PRESS
on behalf of
THE BRITISH COMPUTER SOCIETY
Cambridge
London New York New Rochelle
Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
32 East 57th Street, New York, NY 10022, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

First published 1985

Printed in Great Britain at the University Press, Cambridge

Library of Congress cataloguing data available

British Library cataloguing in publication data

British National Conference on Databases (*4th: 1985: University of Keele*)
Proceedings of the Fourth British National Conference on Databases (BNCOD4): University of Keele, 10–12 July 1985.–
(The British Computer Society workshop series)
1. Data base management 2. File organization (Computer Science)
I. Title II. Grundy, A. F. III. British Computer Society IV. Series
001.64'42 QA76.9.D3

ISBN 0 521 32020 8

Conference committees

Programme committee

T. Bourne, CACI Inc. International
T. Crowe, Thames Polytechnic
A. F. Grundy, University of Keele
R. G. Johnson, Birkbeck College
P. J. H. King, Birkbeck College
J. Longstaff, Leeds Polytechnic
M. J. R. Shave, University of Liverpool (Chairman)
P. M. Stocker, University of East Anglia

Organising committee

A. F. Grundy, University of Keele
J. Longstaff, Leeds Polytechnic
A. E. Robinson, North Staffordshire Polytechnic

Production editor

P. Hammersley, Middlesex Polytechnic

THE BRITISH COMPUTER SOCIETY WORKSHOP SERIES

EDITOR: P. HAMMERSLEY

The BCS Workshop Series aims to report developments of an advanced technical standard undertaken by members of The British Computer Society through the Society's study groups and conference organisation. The Series should be compulsive reading for all whose work or interest involves computing technology and for both undergraduate and post-graduate students. Volumes in this Series will mirror the quality of papers published in the BCS's technical periodical *The Computer Journal* and range widely across topics in computer hardware, software, applications and management.

Some current titles:

Data Bases: Proceedings of the International Conference 1980
Ed. S. M. Deen and P. Hammersley

Minis, Micros and Terminals for Libraries and Information Services
Ed. Alan Gilchrist

Information Technology for the Eighties BCS '81
Ed. R. D. Parslow

Second International Conference on Databases 1983
Ed. S. M. Deen and P. Hammersley

Research and Development in Information Retrieval
Ed. C. J. van Rijsbergen

Proceedings of the Third British National Conference on Databases (BNCOD 3)
Ed. J. Longstaff

Research and Development in Expert Systems
Ed. M. A. Bramer

Proceedings of the Fourth British National Conference on Databases (BNCOD 4)
Ed. A. F. Grundy

Forthcoming

People and Computers: Designing the Interface
Ed P. Johnson and S. Cook

Preface

This volume presents the Proceedings of the Fourth British National Conference on Databases organised by the University of Keele in association with The British Computer Society.

The aim of this series of conferences (the first was held in 1981) is to provide a forum for British research workers in databases and database practitioners to present the results of their work. However, in order to broaden the base of the conference two non-British experts were also invited to present papers.

Each of these invited contributions deals, albeit in very different ways, with shortcomings of the relational model. J. W. Schmidt (Chapter 1) discusses the extension of the relational and network model to cater for nested and recursive predicates and data structures. G. Pelagatti (Chapter 10) describes a synthesis of the relational and network models uniting these in a single system designed to provide some of the benefits of each.

The ten papers selected by the Programme Committee, after independent refereeing, for presentation at the conference are all included in these Proceedings. The order in which the papers appear in the Proceedings is the order of scheduled presentation. The subjects these selected papers cover fall into four categories.

Access and Concurrency Control (Chapters 2, 3)
Models and Mapping (Chapters 4, 5, 6, 7)
Database Management Systems and their users (Chapters 8, 11)
Database Management Systems in use (Chapters 9, 12)

In conclusion I should like to acknowledge those who have assisted in the organisation of the conference. I thank, first, the members of the Programme Committee, particularly Professor Mike Shave for his guidance as chairman, and members of the Organising Committee; Dr Jim Longstaff for the benefit of his experience and Mrs Anne Robinson for her help with publicity. Colleagues in both the Department of Computer Science and the Computer Centre at Keele have contributed valuable advice and assistance, especially Mrs Brenda Banks and Mr David Sherwood who have given detailed advice on

design and presentation, also Miss Jayne Doherty who has done the typing. Thanks must also go to Mr Peter Hammersley and Mrs June Carr at Middlesex Polytechnic for their considerable contribution in preparing the Proceedings for publication.

Finally I must thank the BCS for providing publicity and the BCS Database Specialist Group for financial assistance, which was particularly valuable in the early stages of the organisation.

Frances Grundy
Conference Organiser.

Contents

Higher Level Relational Objects*

Joachim W. Schmidt and Volker Linnemann
Fachbereich Informatik, Johann Wolfgang Goethe-Universitat, Dantestrasse 9, D-6000 Frankfurt am Main 1, West Germany

Relations have been accepted as a data structure adequate for a wide variety of data-intensive applications. On the one hand, this is due to the relatively complete query languages that come with relations, on the other it results from additional technical services such as query optimization, integrity and concurrency control as well as user-friendly interfaces provided by relational systems.

A decade of practical experience has demonstrated, however, that there are at least two basic deficiencies of the relational model. At first, there exist restrictions in the use of predicates for relation definition ("linear" predicates only); they turn out to be too restrictive for rule-intensive applications as, for example, in decision support systems and expert systems. At second, there are limitations in the use of structures for element definition ("linear" structures only); they are inappropriate in particular for object-intensive applications as, for example, in CAD/CAM or in office modelling.

The presentation outlines current research that generalizes the relational approach in both directions by allowing recursive structured definition as well as recursive rule definition. This research aims for a basis for future information models while trying to maintain the essential benefits of current relational technology.

1. Introduction

The relational model for data bases, as introduced by E.F. Codd [5], has been accepted as a framework for the solution of many of the problems with data-intensive applications. Its main advantages over traditional models for data files as founded, for example, on C.A.R. Hoare's approach to record handling [13] originate, in essence, from the different ways both approaches deal with object identification. Files refer to records by references, an implementation-oriented notion on a low conceptual level,

***This paper is based on work from [26], [17]**

known to be error-prone and hard to generalize. Relations identify their elements by distinguished attribute values or combinations thereof, ie, by keys, a notion that leads itself, as we shall see, to predicative set definition and set selection.

Practical experience with the relational model has demonstrated, however, that it has at least two basic deficiencies. On the one hand, it restricts the use of predicates for relation definition ("linear" predicates only), and the traditional relational query facilities turn out to be too restrictive for rule-intensive applications, as, for example, decision support systems and expert systems. On the other hand, the relational model limits the use of structures for element definition ("linear" structures only), and the flat relational tuple structures are inappropriate in particular for object-intensive applications as, for example, CAD/CAM or office systems modelling.

The main purpose of this paper is to outline current research that extends the relational approach by generalizing both, relation and element definition, essentially by allowing nested and recursive predicate and structure definitions. Section 2 presents an overview over the use of types, relations and predicates in database programming. In section 3, the principles of rule-based and recursive relation construction are outlined while section 4 generalizes the flat structure of the relational model to what is called a recursive database model. Finally, the two directions of research are compared and related with emphasis on future work in information modelling.

2. Types, Relations, and Predicates

The impact of logic on computing – from early data processing in the fifties to modern computer science – can hardly be overestimated.

In the field of programming, logic marks the step from machine-oriented coding to algorithmic programming. High level languages provide conditional statements and boolean expressions, use propositions for data type definition, and depend crucially on predicates for the specification of language semantics and for reasoning about programs [10], [12].

In the area of data modelling, the degree to which predicates are utilized allows a distinction between early reference-oriented data models and those that capture more of the relationships defined by the application semantics.

2.1 Data Types and Predicates

If "a type is a precise characterization of structural and behavioural properties which a collection of entities (actual or potential) all share ..." [7],

the formalism by which those properties can be characterized decides upon the power of a type calculus.

Currently prevalent programming languages only allow type definitions based on restricted propositional logic. Take, as an example, the following PASCAL-like subtype definition:

```
partidtype IS RANGE  1..100,
```

which is equivalent to the domain predicate ($1 \leq p$ AND $p \leq 100$) and which defines the domain set

```
partidtype { EACH p IN integer: 1 ≤ p AND p ≤ 100 } .
```

The expressiveness of the type calculus in high level languages corresponds closely with that of the expression and statement part of these languages. As a consequence, any action to be taken to assure type properties can be expressed directly in the language. A type checker can produce run time code in the source language to assure, for example, type correctness of an integer expression, ix, which is to be assigned to a variable, p, of partidtype:

```
IF (1 ≤ ix) AND (ix ≤ 100)
THEN p:=ix
ELSE <exception> .
```

Programmers reduce the possibility of run-time exceptions by acquiring sufficient information on rhs-expressions through inductive reasoning about assignment chains and subtype definitions (and so do clever compilers).

Approaches to programming that are more concerned about correctness allow for the definition of additional program properties by so-called annotations. ADA annotations, for example, can be specified in the meta language ANNA [20], and ADA programs can be proven formally correct w.r.t. their specification. The meta language ANNA allows full first order assertions, while the object language ADA is restricted to propositional logic. An ADA subtype definition, for example, primetype, can be fully specified by the following ANNA annotation [20]: defining the domain set

```
primetype IS integer ;; WHERE p IN primetype ==>
                         ALL n IN integer
                          ((1<n AND n<p) ==> p MOD n ≠ 0),
```

defining the domain set

```
primetype { EACH p IN integer: ALL n IN integer
                ((1<n AND n<p) ==> p MOD n ≠ 0)} .
```

2.2 Predicates in Database Languages

Database models, as, for example, the relational model are very concerned about data integrity; therefore they go beyond programming languages in the sense that they provide the expressiveness of first order logic directly through relational languages.

On the expression level, the request for "relational completeness" of query languages is essentially met by allowing full first order predicates, p(r,...), as selection predicates in relational expressions:

```
reltype {EACH r IN rel: p(r,...)}.
```

On the type or schema level, the role of predicates can be exemplified best by comparing a PASCAL-like set type definition

```
settype = SET OF elementtype,
```

with a relation type definition.

The legal values of a relation are also sets of elements; they have to meet, however, the additional constraint that some attribute (or a collection of attributes) serves as a key, ie, has a unique value amongst all the elements of a relation:

```
reltype = SET OF elementtype !!
              WHERE rel IN reltype ==>
                ALL r1,r2 IN rel ( r1.key=r2.key ==> r1=r2 ) .
```

The key constraint is essential to relational data modelling since only unique keys can serve as element identifiers as required, for example, for the construction of higher relationships between elements. Therefore, relational languages directly support the above class of annotated set type definitions by a data structure **relation** that allows for type definitions equivalent to the previous one:

```
reltype = RELATION key OF elementtype.
```

For each assignment of a relational expression, rex, to a variable, rel, of reltype, the relational type checker has to perform a test equivalent to

```
IF ALL x1,x2 IN rex ( x1.key=x2.key ==> x1=x2 )
THEN  rel:=rex
ELSE  <exception> .
```

2.3 Predicative Support for Relations: Selectors and Constructors

The key constraint is, of course, not the only condition one would like to have maintained automatically on a database. Take, for an example, some objects related by the fact that one object is in front of another.

```
TYPE objecttype = ... (* full object description,
                         e.g. by object record *) ... ;
     parttype   = ... (* representative object description,
                         e.g. by object key *) ... ;

     objectrel  = RELATION part OF objecttype;
     infrontrel = RELATION ... OF
                         RECORD
                              front,back: parttype
                         END;
VAR Objects: objectrel;
    Infront: infrontrel .
```

Since the attributes, front and back, of the Infront relation are supposed to relate objects, they have to refer to elements in the Objects relation. The corresponding referential integrity constraint can be expressed by annotating the type of the Infront relation:

```
VAR Infront: infrontrel !! WHERE r IN Infront ==>
                             SOME r1,r2 IN Objects
                                (r.front=r1.part AND
                                 r.back=r2.part).
```

In a relational language such a constraint can be enforced by a conditional which controls assignment to the Infront relation:

```
IF ALL x IN rex (SOME r1,r2 IN Objects
                 (x.front=r1.part AND x.back=r2.part) )
THEN Infront:=rex
ELSE <exception>.
```

Expecting frequent use of relations in such "conditional patterns", the database programming language DBPL [27], [23] provides an abstraction mechanism for such patterns through the notion of a **selector**. Referential integrity on relations of type infrontrel, for example, can be maintained by

```
SELECTOR refint FOR Rel: infrontrel(): ...;
BEGIN  EACH r IN Rel: SOME r1,r2 IN Objects
                      (r.front=r1.part AND r.back=r2.part)
END refint.
```

An assignment to a selected relation variable, for example,

```
Infront[refint] := rex,
```

is defined to be equivalent to the above conditional assignment to the full relation variable Infront.

In summary, selectors "factor out" conditions on relations, represent them uniformally, and make them available to all database system components that have to reason about programs and data (such as query optimizer, concurrency manager, and integrity subsystem). The selector concept is illustrated in Fig. 1.

While selectors provide support when data elements are to be **excluded** from a relation there is also a need for supporting the contrary – when additional derived data objects are to be **included** into a relation.

For an example, a relation, Ahead_2, can be defined that relates – based on the data in relation Infront – two objects if and only if they are separated by at most two steps.

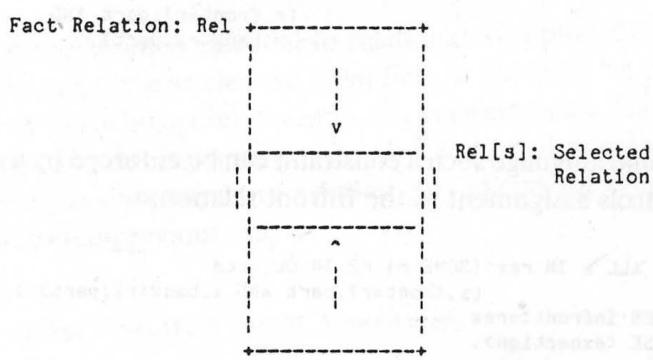


Fig. 1 Selectors and Relations

Starting with type

```
TYPE aheadrel = RELATION ... OF  RECORD
                                    head,tail: parttype
                                 END;
```

an annotated definition of relation Ahead_2 would read as follows:

```
VAR Ahead_2: aheadrel {| WHERE (r IN Infront ==> r IN Ahead_2)
                          AND  (f,b IN Infront ==>
                                 (f.back=b.front ==>
                                  <f.front,b.back> IN Ahead_2)) .
```

In a relational language the value of such a relation, Ahead_2, can be denoted by a query expression in terms of predicates and the Infront relation:

```
aheadrel { EACH r IN Infront: TRUE,
           <f.front,b.back> OF EACH f,b IN Infront:
                               f.back=b.front  }.
```

Expecting frequent use of relations in such "expressional patterns", we have defined an abstraction mechanism for such patterns based on the notion of a **constructor** [17].

As an example, the Ahead_2-relationship based on relations of type infrontrel can be constructed by

```
CONSTRUCTOR ahead_2 FOR Rel:infrontrel (): aheadrel;
BEGIN   EACH r IN Rel: TRUE,
        <f.front,b.back> OF
            EACH f,b IN Rel: f.back=b.front
END ahead_2.
```

The value of a constructed variable, for example,

```
Infront [ahead_2]
```

is defined to be equal to the above relational expression of type aheadrel.

In the same sense as selectors isolate the constraints imposed on selected relations, constructors factor out the rules that define the elements in constructed relations.

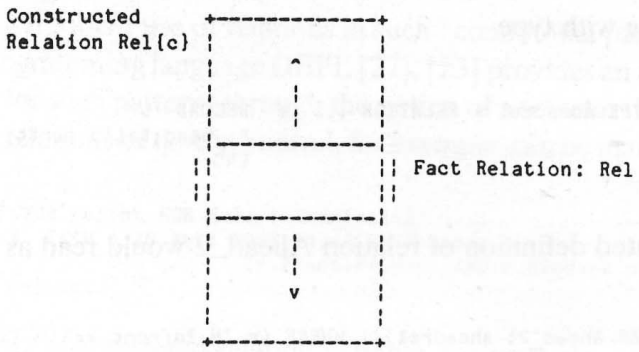


Fig. 2 Constructor and Relations

In the subsequent section, the basic issues of constructor semantics are outlined with emphasis on recursive constructor definition.

3. Relation Constructors

Combining the semantic capabilities of rule-based knowledge representation and reasoning systems with the efficiency-oriented mechanisms for query result construction and transaction processing in large shared DBMS has been the focus of much current research [8], [19].

The use of rules follows a similar pattern. Certain base facts are stored for which it is known that a certain rule holds. Other facts for which the rule also holds are not stored explicitly but can be derived by a possibly recursive (deduction) rule. The deduction rule may depend on the existence of other facts (parameters), which are, however, not necessarily part of the result. Constructors allow the definition of such deduction rules in DBPL. The idea is illustrated in Fig. 2.

In this section, we discuss the notion of a constructor in more detail by providing some examples based on the relations introduced in section 2.

3.1 Recursive Constructors

The above simple constructor, ahead_2, representing all object pairs separated by at most two steps, can be generalized to a sequence of constructors, ahead_n, representing all pairs of objects separated by at most n steps:

```
CONSTRUCTOR ahead_n FOR Rel:infrontrel(): aheadrel;
BEGIN   EACH r IN Rel: TRUE,
        <f.front,b.tail> OF EACH f IN Rel,
                            EACH b IN Rel {ahead_n-1} :
                              (f.back=b.head)
END ahead_n.
```