SPECIAL ISSUE

7th SA COMPUTER RESEARCH SYMPOSIUM
The South African
Computer Journal
An official publication of the South African
Computer Society and the South African Institute of
Computer Scientists

Die Suid-Afrikaanse
Rekenaartydskrif
'n Ampelike publikasie van die Suid-Afrikaanse
Rekenaarvereniging en die Suid-Afrikaanse Instituut
vir Rekenaarweenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
Email: dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof John Schochot
University of the Witwatersrand
Private Bag 3
WITS 2050
Email: 035ebrs@witsvma.wits.ac.za

Productio.. Editor
Prof G de V Smit
Department of Computer Science
University of Cape Town
Rondebosch 7700
Email: gds@cs.uct.ac.za

Editorial Board

Professor Gerhard Barth
Director: German AI Research Institute

Professor Pieter Kritzinger
University of Cape Town

Professor Judy Bishop
University of Pretoria

Professor F H Lochovsky
University of Toronto

Professor Donald Cowan
University of Waterloo

Professor Stephen R Schach
Vanderbilt University

Professor Jürg Gutknecht
ETH, Zürich

Professor S H von Solms
Rand Afrikaanse Universiteit

Subscriptions
Annual Single copy

Southern Africa: R45,00 R15,00
Elsewhere $45,00 $15,00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685
7th Southern African Computer Research Symposium

Karos Indaba Hotel, Johannesburg
1 July 1982

PROCEEDINGS

Guest Editor: Judy M Bishop

Persetel

Organised by the SA Institute of Computer Scientists
in association with the Computer Society of SA

Sponsored by Persetel and the FRD
When the first SA Computer Symposium was held at the CSIR in the early eighties, it was unique. There was no other forum at the time for the presentation of research in computer science. In the intervening decade, conferences, symposia and workshops have sprung up in response to demand, and now there are several successful ventures, some into their third or fourth iteration. Each of these addresses a specific topic - for example, hypermedia, expert systems, parallel processing or formal aspects of computing - and attracts a specialised audience, well versed in the subject and eager to learn more. For the main part, the proceedings are informal, and certainly not archival.

SACRS, though, is still unique, in that it deliberately covers a broad spectrum of research in computing, and in addition, seeks to provide a lasting record of the proceedings. To achieve the second aim, we negotiated with the SA Institute of Computer Scientists for the proceedings to form a special issue of the SA Computer Journal, and the copy you have in front of you is the result. The collaboration between the symposium committee and the journal’s editorial board placed high standards on the refereeing and final presentation of the papers, to the symposium’s benefit, while we were still able to maintain a fresh, audience-oriented approach to the selection of papers.

This is SACJ’s first such special issue, and the largest issue (at 145 pages) to date. We hope that it is only the beginning of future such collaborations.

In all 29 papers were received, all were refereed twice, and 19 were chosen for presentation by the programme committee. All the papers were thoroughly revised by the authors on the basis of the referee’s comments, and the committee’s suggestions aimed at making the material more accessible to a broadly-based audience. Papers had to be new, and not to have been presented elsewhere, a requirement that is still unusual within the SA conference round.

A third goal of SACRS has been to invite keynote speakers, usually from overseas. This year, we are fortunate to present Dr Vinton Cerf, the father of the Internet and a world-renown expert on computer networks. Although his paper is not available for this special issue, it will appear later in SACJ. Through the good offices of Professor Chris Brink of UCT, we also have three other speakers from Germany, Canada and the US adding interest to the event, and two of their papers appear in this issue.

The programme committee originally devised a theme for the symposium - "Computing in the New South Africa". We received several queries as to the meaning of this theme, but unfortunately few papers that addressed it directly. One prospective author went as far as to enquire whether computer research would survive in the new South Africa. Another felt that his work was definitely not in the theme, as it was genuine, old world, basic, theoretical science! Nevertheless, there are two papers that consider one of South Africa’s key issues, that of language. Others look at the success we have achieved in applying technology to mining, and the future of low-cost operating systems. In all, the mix of papers represents a balance between the theoretical and the practical, the past and the future, all firmly based in the computing of the present.

Organising the symposium has involved the hard work of several people, and I would like to thank in particular

• Derrick Kourie, my co-organiser, and the editor of SACJ for his invaluable advice and hard work throughout the planning and implementation stages;
• Riel Smit, the production editor, for attaining such a high standard in such a short time for so many papers;
• Gerrit Prinsloo and the staff at the CSSA for their efficient and quite delightfully unfussy organisation;
• Persetel for their very generous sponsorship of R25000, and Tim Schumann for taking a genuine interest in our events;
• the Foundation for Research Development for sponsoring Vint Cerf’s visit;
• and finally the Department of Computer Science of the University of Pretoria for providing the ideal working conditions for undertaking ventures of this kind, and especially Roelf van den Heever for his unfailing encouragement and support.

Judy M Bishop
Organising Chairman, SACRS 1992
Guest Editor, SACJ Special Issue
Referees

The journal draws on a wide range of referees. The following were involved in the refereeing of the papers selected for this special issue. Their role in certifying the papers and their contribution to enhancing the quality of papers is sincerely appreciated.

John Barrow
Ronnie Becker
Danie Behr
Sonia Berman
Liesbeth Botha
Theo Bothma
Chris Brink
Peter Clayton
Ian Cloete
Antony Cooper
Elise Ehlers
Quintin Gee
Andy Gravell
Wendy Hall
David Harel
Scott Hazellhurst
Derrick Kourie
Willem Labuschagne
Doug Laing
Dave Levy
Graham McLeod
Hans Messerschmidt
Deon Oosthuizen
Ben Oosthuysen
Neil Pendock
D Petkov
Martin Rennhackkamp
Cees Roon
Jan Roos
John Shochot
Morris Sloman
Riel Smit
Pat Terry
Walter van den Heever
Lynn van der Vegte
Herman Venter
Herna Viktor

UNISA
University of Cape Town
University of Pretoria
University of Cape Town
University of Pretoria
University of Pretoria
University of Cape Town
University of Cape Town
Stellenbosch University
CSIR
Rand Afrikaanse University
DALSIG
University of Southampton
University of Southampton
The Weizmann Institute of Science
University of the Witwatersrand
University of Pretoria
UNISA
ISM
University of Natal
University of Cape Town
University of the Orange Free State
University of Pretoria
University of Pretoria
University of the Witwatersrand
University of Natal
Stellenbosch University
University of Pretoria
University of Pretoria
University of the Witwatersrand
Imperial College, London
University of Cape Town
Rhodes University
University of Pretoria
University of Pretoria
University of Fort Hare
Stellenbosch University
A Model For Object-Oriented Databases

Michael M Brand
Peter T Wood
Department of Computer Science, University of Cape Town, Rondebosch, 7700

Abstract

Unlike the relational model, there is no universally accepted object-oriented data model. This has led to systems being defined in an ad-hoc manner, with the resultant difficulty of determining exactly what features are provided in any particular system as well as how these features are interpreted. We define an implementation-level data model which supports all the essential features required of any object-oriented database in a uniform, consistent and extensible way. User-level data models can be implemented on top of such a system, or can be compared in terms of it.

Keywords: Compilation, generic types, meta-types, methods, object-orientation, subtypes, type inference
Computing Review Categories: D.3.4, E.1, H.2.1

Received March 1992, Accepted April 1992

1 Introduction

While relational database systems have proved to be successful in modelling traditional business applications, they have been found to be inadequate for modelling emerging applications such as CAD/CAM, office automation, scientific databases, and hypertext, for example. Object-oriented database systems are claimed to provide the necessary modelling power for such applications, while also offering superior performance to relational systems [4].

One advantage of relational systems, however, is their dependence on a single, simple, formal data model, which means that specific implementations of relational database systems can be relatively easily compared against this model as well as another. No such model exists for object-oriented databases; indeed, authors such as Maier have argued that there cannot be such a model [7].

Nevertheless, a number of researchers and developers have attempted to define the major features which any object-oriented database system must possess [1]. We have incorporated most of these features in an implementation-level data model in a uniform and consistent way, allowing extensibility for other features which may be considered important for particular application domains. The intention is that such a system can be used both to implement specific user-level data models, and to compare existing data models in terms of their translation to our model.

In the Object-Oriented Database System Manifesto [1], the mandatory features for an object-oriented database system are listed as: complex objects, object identity, encapsulation, types or classes, type or class hierarchies, overriding, overloading and late binding, computational completeness, extensibility, persistence, secondary storage management, concurrency, recovery and reliability, and an ad-hoc query facility. At the present stage of our system, the last four features have not yet been included. Optional features in the manifesto are given as: multiple inheritance, type checking and type inferencing, distribution, design transactions, and versions. We provide the first two of these, while the last three may only be required in specific application domains, in which case they can be added to the system.

By defining everything as an instance through the use of complex data structures and metatypes, the model is uniform and allows for dynamic extensibility. The latter is also provided for by the creation of new types, generic types, and extensible base types. The correctness of the system is achieved by typing all instances and type checking all operations.

The rest of the paper is organised as follows. In the next section, we describe the principal features of our model. This is done by means of a number of examples modelling a publication database. The fundamental features of the model include types and subtypes, instances, methods, metatypes, and generic types. These are then used to define a particular implementation model, a so-called exemplar. In Section 3, we briefly describe the implementation of the model. Our conclusions are presented in Section 4.

2 The Data Model

The model is based on three components: a structure system, an object system, and a type system. The structure system defines the various forms of values which may occur in the database. The object system provides the mechanism for modelling entities in the real world by allowing the persistence of objects as well as a means of uniquely referencing and manipulating them. A type models a concept in the real world. Each type has a set of entities associated with it, whose structure and behaviour are defined by the type. These three systems form the foundations for defining the data model. They are used in an exemplar to define a realization of the concepts specified in the type system.

Figure 1 illustrates the structure of the model. At the bottom level are the foundations which specify the general requirements and constraints of an object-oriented system. The exemplar is defined by making use of the constructs and
The concept of a conference proceedings is modelled in the system by a type. A type defines the structure for its instances and the operations which may be performed on them. For example, type Proceedings is defined as follows:

```
(Type,
 (name:='Proceedings',
  state:={Proc1, ...},
  objects:={Proc1, ...},
  contents:List((work:=>Paper,
                pages:Product(Int, Int)),
               issn:String,
               conf-add: String, conf-date: Date))
```

The type has a name, 'Proceedings', which is used to identify it symbolically. The state of the type is used to define the structure of its instances. The state makes use of a structure which is defined in the structure system. These structures are similar to data types found in programming languages, with '\>' denoting an object reference. The data structures have been based on those of Schmidt [8], who defines record, product, disjoint union, and function constructors, along with assembly and disassembly operations. Some of these constructors have also been defined by Cardelli and Wegner [3], with the addition of subtyping rules. These rules form the basis for the subtyping rules used in the structure system. Ordinal types, ranges and enumerations are based on Ada, Pascal and [3]. The array and method constructors have been adapted from record and function constructors respectively.

The Proceedings type has a set of methods defined for it, which can be used to perform operations on its instances. The methods are stored in the behaviour attribute of the type. The objects attribute stores references to objects which are instances of a conference proceedings, as described in the next section.

The concept of a type captures the properties of an abstract data type: it is an intentional specification of its instances through the use of a data structure and is used in the type checking process. Classes are not defined in the system, but the extensional specification of a class is found in the type, in that each type stores a reference to each of its objects. The system provides extensibility by allowing the user to define new types at will, which have the same status as the system-defined types. The system also allows new
base types to be added which also share the same status as other types.

As can be seen in the previous example, a type also stores references to its subtypes and supertypes. Subtyping is a mechanism provided by the system that allows us to model specialization of concepts. For example, the general concept of an edited publication is some form of printed matter which has a title, a date when it was published, an organization that published it, an editor and a list of articles which appear in it. The concept of a conference proceedings is a specialization of this concept, since it has all of the properties of an edited publication, as well as additional properties such as the conference address and date. The Edited Publication type is defined below.

\[
\text{(Type,}
\begin{array}{l}
\text{name:- 'Edited Publication',} \\
\text{state:- Product(\{Type,} \\
\text{\textbf{title:- String,}} \\
\text{\textbf{date:- Date,}} \\
\text{\textbf{publisher:- \Rightarrow Organization,}} \\
\text{\textbf{editor:- \Rightarrow Person,}} \\
\text{\textbf{contents:- List((work:- \Rightarrow Paper,}} \\
\text{\textbf{\textit{pages:- Product(Int, Int))}}}) \\
\text{\textbf{behaviour:- \{\ldots,}} \\
\text{\textbf{objects:- \{\ldots,}} \\
\text{\textbf{subtypes:- \{Edited Book, Proceedings,}} \\
\text{\textbf{Journal, Magazine},}} \\
\text{\textbf{supertypes:- \{Publication\}\\})}
\end{array}
\]

The Edited Publication type stores a reference to each of its subtypes in the 'subtypes' attribute. Likewise, the Proceedings type stores a reference to each of its supertypes. As can be seen, Proceedings is a subtype of both Edited Publication and Periodical. The subtyping relationship between types gives rise to the subtyping hierarchy, which depicts the inheritance that occurs between types. In Figure 3, we illustrate subtyping between the various types which are used to model the concepts of publications, reference lists and articles.

The term subtyping is in keeping with the use of types and type checking by the system. The various forms of inheritance identified by [1] are all supported by the subtyping mechanism. The data structure used in a subtype must be a subtype of the data structure used in the type. This allows the methods defined in the types and supertypes to be inclusion polymorphisms as defined by [3]. The uniformity of the system is also evident in subtyping, since all types, including base types and generic types, may have subtypes defined for them.

**Instances**

While a type models a real world concept in the system, instances are used to model specific cases of those concepts. The following SIGMOD proceedings is an instance of the conference proceedings concept.

\[
\text{(Proceedings,}
\begin{array}{l}
\text{\textbf{title:- 'Proc. 1990 ACM SIGMOD Int. Conf.}} \\
\text{\textbf{on Management of Data',}} \\
\text{\textbf{date:- 1/6/90,}} \\
\text{\textbf{publisher:- Pub1,}} \\
\text{\textbf{editor:- Person2,}} \\
\text{\textbf{contents:- \{\textbf{work:- Paper1,}} \\
\text{\textbf{\textit{pages:- (1, 35)), \ldots,}} \\
\text{\textbf{issn:- '2367-34578', \ldots \}}}
\end{array}
\]

The instance stores all of the information relating to the publication in the real world which it is modelling. This information is stored in a value which is defined by the structure in the type's state. For example, the title of the proceedings is stored in the attribute 'title' and has the string value 'Proc. 1990 ACM SIGMOD Int. Conf. on Management of Data'.

In accordance with the terminology for types, we have termed the concept of an object which is found in the literature, an instance in our model. Instances are split into two disjoint groups: objects and values. An object is an instance of a type's data structure, may exist as an independent entity, and contains information regarding its type.

The object system is used to capture the basic properties of an object and its operations. The concept of object identity and the object operations which are used in the model are based on those defined by Khoshafian and Copeland [6]. An object in the system corresponds directly to the entity it is modelling in the real world. The object reference structure is used to reference objects, to create relationships (weak references) between objects, and to
share components among objects as discussed by Zdonick and Maier [9]. An object's identity is used as a means of referencing it. The Proceedings type stores the identities of all of its objects, in the 'objects' attribute. The 'publisher' and 'editor' attributes also contain object references, to 'Organization' and 'Person', respectively. The references must be consistent and the concept of Nil is defined to avoid dangling references.

Objects are the only persistent structures: if something is to persist, it must be defined as an object. There are also temporary objects which enjoy the property of identity but do not persist. Once an object has been declared as persistent, the system ensures that the object is loaded from and saved back to disk.

A value, on the other hand, is dependent on the context in which it is used, such as a variable or as a component in a larger structure. Since the information regarding a value's type can always be obtained from its context, this information need not be stored in the value. A value is also defined by the data structure in its type, but omits the portion of information regarding its type.

Not all instances used in the system require the modelling capabilities of an object. The distinction between objects and values is made to allow for values which do not require persistence or an identity, and hence the additional overhead.

In the publication example, the entities in Figure 2 are modelled in the system by the objects depicted in Figure 4. Each object stores the information regarding the entity in the real world which it is modelling and also stores the identities of other objects to which it is related. The Paper1 object stores the identity of ACM proceedings Procl in its 'appearsIn' attribute. This models the relationship which exists between these two objects.

Methods

The operations defined for an instance are specified in its type as methods. To use a method, we pass a message to the object specifying that it must perform the operations defined in a specific method. For example, a Print method is defined for type Proceedings in our method language as follows. It makes use of the conditional if statement and also relies on Print methods which have been defined in other types, viz., String, Date, Person, Organization, etc.

(Method[Owner:- Proceedings, Param:- Boolean, Result:- Nil],
   (name:- 'Print',
    source:- ' 
    /* 
    \nProceedings: '.Print;
    this.title.Print;
    \nDate of Publication: '.Print;
    this.date.Print;
    \nPublished by: '.Print;
    this.publisher.Print;
    \nConference Address: '.Print;
    this.conf-add.Print;
    \nConference Date: '.Print;
    this.conf-date.Print;
    if (param) |
    | \nEditor: '.Print;
    this.editor.Print;
    \n: ISSN: '.Print;
    this.issn.Print;
    \n: Papers: '.Print;
    /|
    | \n'.Print;
    /|
    code:- ...));

The Print method is defined for instances of type Proceedings and requires a boolean parameter. The parameter is used to specify either a full printout or a brief printout. This method is used by passing the Print message to an object, say Procl, which performs the operations specified in the method. The following message pass requests the Procl object to perform a full printout according to the actions of the Print method which is defined in its type.

Procl.Print(true)

A method is defined by a method structure which forms part of the structure system. Since a method is defined by a structure, it is a value and may be treated like all other values. The methods in the system are defined as objects of a special type called Method. The reason for
doing this is to make use of the same constructs which are defined for instances, thereby providing a uniform means of dealing with both methods and instances. An object is the only persistent construct in the system and by making the methods objects, we can store them in the system. By using the identity provided by objects, methods can be referenced from the types in which they are defined and do not need to be stored as part of the type structure. Identity also allows us to reference specific methods, even if there are other methods with the same name (overloading). Since methods are first-class values, there are also operations, such as compile or edit, which can be performed on them.

The operations of the method language are specified either by an operator symbol or by the name of an operation. All named operations which are defined for structure values as well as messages defined for instances make use of the message passing syntax, i.e. the instance or value, a dot, the name of the operation or method and an optional parenthesized parameter. The dot operator is also used for every value in a set or list, yielding a set or list containing the result is a hierarchy of types and instances. An object is the only persistent construct in the system and only one form of instantiation, such as, between type and instance, metatype and type, meta-metatype and metatype, and so on. The metatype concept allows us to have various levels of identity and associated referencing mechanism. An object stores its typing information by storing the identity of its type. The subtyping relationship between types is specified by each type storing the identity of other types in either its 'subtypes' or 'supertypes' attribute. The name of a type is used as a symbolic reference to a type and forms part of a database with object identity. The metatype concept provides the system with a reflective means of definition, giving it the power and flexibility to adapt to changes and new requirements as they arise. The metatype concept also allows the system to treat types as objects.

Metatypes allow us to achieve uniformity in another respect. All types in the system are defined as objects. Like methods, they are also stored as objects and may be referenced by an identity. Thus, there is only one persistent construct in the form of an object and only one form of identity and associated referencing mechanism. An object stores its typing information by storing the identity of its type. The subtyping relationship between types is specified by each type storing the identity of other types in either its 'subtypes' or 'supertypes' attribute. The name of a type is used as a symbolic reference to a type and forms part of the name space. But essentially all types are referenced by their identities.

The type concept gives rise to the instantiation mechanism, whereby an instance is instantiated from its type. The metatype concept allows us to have various levels of instantiation, such as, between type and instance, metatype and type, meta-metatype and metatype, and so on. The result is a hierarchy of types and instances.
Generic Types

The idea of a generic type originated from the generic template defined by Ada, while some of the typing structures are based on the parametric polymorphism structures defined in [3]. The model provides structures for defining generic parameter lists and hence generic types.

The generic types can be viewed as a means to extend the set of constructors provided by the system. A constructor may be applied to any set of types and yields a structure with certain basic properties that are independent of the component structures. A constructor also has a set of operations to manipulate its values, once again independently of the component structures.

A stack can be defined as a generic type by internally using a list. The values stored in the stack (the list) are defined in terms of a generic parameter called Data. The generic type Stack is defined as follows:

```
(TypeGen,
 (name: 'Stack',
 state:- Product(=>TypeGen,
 (value: List(Data))),
 behaviour:- {Pop, Push, Empty, ...},
 objects:- (Stack1, Stack2, Stack3),
 subtypes:- {},
 supertypes:- {InstGen},
 generic:- [Data: Struct]))
```

The structure of an instance is defined by a list of values which have the Data parameter. The operations such as Pop and Push which are used to manipulate a stack are also defined using the generic parameter. The resulting generic type can be conceptualized in the same manner as the list constructor, and may be used in the same way. For example, a list of people and a stack of people are denoted as follows:

```
List(=>People)
Stack(Data: =>People)
```

The generic parameter Data is replaced by the object reference to people throughout the type. With the use of subtyping and since the generic types are treated basically like any other type, it is possible to define a subtype of a generic type. If we view generic types as defining constructors, then it is possible for us to specialize constructors in this way.

Exemplar

The structure and object systems provide the constructs for defining an object-oriented database model. The type system specifies the constraints which must be satisfied in order for the model to be object-oriented. We have defined such a data model which we term an exemplar, since it is not unique (there are many other models which can be defined that satisfy the constraints).

The concept of a type is defined in the system by the metatype Type, which defines the structure of types and the messages which may be passed to them. Since Type is a metatype and since we treat all types as instances, there must exist some other type of which Type is an instance. Since Type defines all types, we have defined it as its own instance. Thus, Type forms the root of the instantiation hierarchy. The model also requires that everything be an instance, to which end we have defined a type Instance. This defines the general structure of an instance and forms the root of the subtyping hierarchy, since every type is required to produce instances. Because Instance is a type, it is an instance of the type Type. Type is also required to produce instances, therefore it is also a subtype of Instance.

From these principal two types, the other types used in the model are defined. These include generic types, method types, base types and the types used specifically for the application, such as type Proceedings. The instantiation and subtyping hierarchies are depicted in Figure 5. The nodes in the graph are objects, while each plane contains different classes of objects. The Metatype plane contains only metatypes. The Types plane contains ordinary types, which excludes metatypes. The Instances plane contains all ordinary instances and thus excludes all types. A dotted arrow denotes instantiation and is read “X is an instance of Y,” while a solid arrow denotes subtyping and is read “X is a subtype of Y.”

In summary, the system provides uniformity because types and methods are treated as objects. Thus there is only one form of identity and one mechanism for persistence. Everything is referenced and stored in the same way, which makes the system simple in this respect. Through the use of metatypes, the system is flexible and may be extended by defining new base types and ordinary types. The set of constructors for the system can also be extended by defining new generic types.

3 Implementation

We have implemented the following data structures and their corresponding values: base, record, product, set, list, method, reference structure, and higher-order structures. Enumerations, ordinals, sums and arrays have not been implemented. There are also operations for comparing values for equality as well as structures for equivalence.
and subtyping. The operations defined in the model for each structure have been implemented. These structures and values, together with their operations, form the kernel of the system.

For each constructor in the structure system, we define a memory and a disk representation for its structures and corresponding instances. The structure system is data type complete, which means that any structure can be used as a component in any other structure. Thus the memory representation of a structure is usually in the form of a tree. Unlike the memory representation, the disk representation of a structure is stored in a sequential format, which corresponds to a depth-first traversal of the memory representation.

The object system implements the identity and persistence of objects. It consists of an object table structure for storing the objects and an interface for manipulating them. The object table consists of a hash table onto which balanced binary trees are chained. The hash function is used to split the identities into disjoint sets. Each set of identities is represented by a balanced binary tree and is accessed from the hash table. Each node in the tree contains the identity of an object and a pointer to the value for the object. Thus, an object in the model is implemented by this identity and the value to which it is pointed.

There is also a name table and manager which together implement the name space associated with the database. The name table stores unique identifiers and corresponding object identities. These names may then be used in methods to access the objects. In the present implementation, the name space has been limited to the names which are used for types.

When a session begins, a particular database is selected and loaded into memory. This process involves reading the identities of the objects and their values, and generating the memory representations of object values. As each object is generated, so the associated node in the binary tree is also generated. At the end of the session all objects are transferred back to disk with their identities. Obviously, a more sophisticated mechanism would be required in a production database system.

The method language allows all of the operations defined in the model as well as the formulation of complex data values and simple control statements. A compiler type checks the method language and then generates code for an abstract machine. Compilation occurs in the context of a method structure and the database. The method structure is used to type check the recipient of the message, the parameter value, and the result of the method. The database is used to check the types used in the method and messages which are passed. Type checking ensures that the operations are correct; when static type checking cannot be performed, dynamic type checks are generated. Also included is a type resolution process which is used to infer the typing of untyped values.

The purpose of type resolution is twofold: first is the inference of typing structures, and the second is the checking of expressions for type correctness. Due to the relative simplicity of the method language syntax, the resolution of typing is far more complex. The typing process makes use of typing structures to store the typing information generated by the expressions as they are parsed. These structures are then used by the type checking and resolution processes to determine the correct typing of values. For example the following operation must be resolved to determine the structure of the components in the list.

\[ J_1, J_2, J_3, J_4, \ldots \] <= .Date

Once this structure has been determined, the appropriate Date message can be located if it exists. From this, the structure of the resulting list can be determined.

The compiler has been written with the aid of Lex and Yacc and comprises approximately 12 000 lines of C code. The compiler produces code which is the value associated with a method structure. This code is defined for an abstract machine which operates on the database. The abstract machine defines the primitive operations for data structures, control structures, message passing and special operations which may be linked to the system. The abstract machine is a stack machine with six-address code that is an extended form of three-address code. The primitive operations which are defined in the model are implemented generally by instructions or templates of instructions in the machine code. The complete abstract machine is defined in [2].

The implementation of our model shows that it is possible to provide all the data model features of an object-oriented database system in a uniform manner, while allowing for extensibility. The system has been developed on a 386 machine and presently comprises approximately 40 000 lines of C code.

4 Conclusion

In this paper we have presented a model for an object-oriented database and its associated implementation. The model provides a powerful set of structures and extensible types which can be used to store data in a manner which corresponds naturally with the entities being modelled. The system is easier for the user to conceptualize because of this direct correspondence between objects in the system and entities in the real world. Through its extensibility, it can adapt to keep pace with changes as they occur. By making use of higher-order and meta constructs, such as higher-order structures in the structure system, metatypes and generic types, we have specified a data model that is reflective and uniform. The model is totally extensible, since new types may be defined at will, base types may be implemented outside the system and then linked into the system, metatypes may be used to extend the types in the model, and generic types can be used to extend the constructors which the model provides.

The model supports modularity and encapsulation which make the maintenance of an application simpler. A type contains the implementation of its instances. All operations are performed by methods defined in the type's behaviour; thus all access to the internal structure of an instance is localized to its type.

SACJ/SART, No 7, 1992
The implementation has demonstrated the suitability of the constructs in the model. It has provided a means of representing the values defined in the structure system, both in memory and on disk. A simple language has been defined to express the operations of the model. The constraints of the type system and the validity of operations are checked by type checking and type resolution processes. The compiler performs these operations and produces code for an abstract machine, which has been defined with operations to manipulate the database and to perform dynamic type checks.

Clearly there are a number of features omitted from the system which form natural extensions to it. The database features of reliability, concurrency and recovery are missing and are essential for turning this system into a full database management system. The model can also be extended with a number of the features found in other systems, such as versions, composite objects and histories.

As far as the implementation is concerned, future work first involves the implementation of all the abstract machine instructions. The next step is to implement all of the structures which are defined in the model. A user-friendly interface is also required on top of the system to access databases and to perform operations such as queries. The system also lends itself to a graphical browser which can be used to navigate from one object to another while inspecting its contents. Finally, a debugger for the abstract machine will be essential once more complicated code is written.

References

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines:

- Use wide margins and 1 ½ or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets [1, 2, 3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) \LaTeX style file(s), either on a diskette, or via e-mail/ftp – a \LaTeX style file is available from the production editor;
2. In camera-ready format – a detailed page specification is available from the production editor;
3. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

Further instructions on how to reduce page charges can be obtained from the production editor.

4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typsetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for \LaTeX or camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 3 and 4 above will be sent to the author to ensure that typsetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisements

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

## Contents

| Forward .................. | i |
| Referees .................. | ii |

| Machine Translation from African Languages to English | DG Kourie, WJ vd Heever and GD Oosthuizen | 1 |
| Automatically Linking Words and Concepts in an Afrikaans Dictionary | PZ Theron and I Cloete | 9 |
| A Lattice-Theoretic Model for Relational Database Security | A Melton and S Shenoi | 15 |
| Network Partitions in Distributed Databases | HL Viktor and MH Rennhackkamp | 22 |
| A Model for Object-Oriented Databases | MM Brand and PT Wood | 27 |
| Quantifier Elimination in Second Order Predicate Logic | D Gabbay and HJ Ohlbach | 35 |
| Animating Neural Network Training | E van der Poel and I Cloete | 44 |
| HiLOG - a Higher Order Logic Programming Language | RA Paterson-Jones and PT Wood | 53 |
| ESML - A Validation Language for Concurrent Systems | PJA de Villiers and WC Visser | 59 |
| Semantic Constructs for a Persistent Programming Language | SB Sparg and S Berman | 65 |
| The Multiserver Station with Dynamic Concurrency Constraints | CF Kriel and AE Krzesinski | 75 |
| Mechanizing Execution Sequence Semantics in HOL | G Tredoux | 81 |
| Statens - an Alternative Modelling Mechanism for Performance Analysis | L Lewis | 87 |
| Galileo: Experimenting with Graphical User Interfaces | R Apteker and JM Bishop | 99 |
| Accessing Subroutine Libraries on a Network | PH Greenwood and PH Nash | 117 |
| A Multi-Tasking Operating System Above MS-DOS | R Foss, GM Rehmet and RC Watkins | 122 |
| Using Information Systems Methodology to Design an Instructional System | BC O’Donovan | 126 |
| Managing Methods Creatively | G McLeod | 131 |
| A General Building Block for Distributed System Management | P Putter and JD Roos | 141 |