DECEMBER 1985 VOL. 8 NO. 4

a quarterly bulletin
of the IEEE computer society
technical committee
on

Database Engineering

Contents

Special Issue on Object-Oriented Systems

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Memoership in the Database Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Letter from the Editor

This issue of **Database Engineering** is on "Object-oriented Systems". Object-oriented systems are receiving wide attention these days in the areas of office systems, data base systems, programming languages, and artificial intelligence. Each of these areas has something to contribute to the design and implementation of an object-oriented system. The term "object", for the purposes of this issue, is interpreted fairly loosely. However, certain aspects of an object-oriented system appear to be common across the papers presented in this issue. These are:

- abstraction of data;
- inheritance of properties;
- persistency of data;
- encapsulation of data and operations;
- automatic triggering of operations.

We start the issue by looking at some applications of object-oriented systems. The first paper *Object Species* by Dennis Tsichritzis uses analogies from the animal world to illustrate what types of objects might be useful to end users. The next two papers, *Task Management for an Intelligent Interface* by Bruce Croft and *An Object-Based Approach to Modelling Office Work* by Carson Woo and Fred Lochovsky, discuss two different approaches to using objects for supporting tasks in offices.

The next six papers describe systems for defining, storing, managing, and using objects. We will refer to such systems as *Object Management Systems (OMS).* The paper by Stan Zdonik *Object Management Systems for Design Environments* defines what an OMS is and describes Encore, an OMS being developed at Brown University. The next paper by Matts Ahlsen et al. *OPAL: An Object-Based System for Application Development* presents an OMS being developed at the University of Stockholm. *An Object-Oriented Protocol for Managing Data* by Stephen Weiser describes the evolution of an OMS developed at the University of Toronto. The paper by Oscar Nierstrasz *Hybrid: A Unified Object-Oriented System* details the design of an OMS being developed at the University of Geneva. The design of this system is also an evolution of the system described by Stephen Weiser. David Maier et al. in the paper *Object-Oriented Database Development at Servio Logic* discuss the development of GemStone, a commercially available OMS that marries the Smalltalk-80 programming language with an advanced data base management system. In *Some Aspects of Stored Operations in an Object-Oriented Database,* Nigel Derrett et al. describe the IRIS OMS being developed at Hewlett-Packard with particular emphasis on data abstraction facilities and operations.

Finally, the last two papers discuss more specific aspects of an object-oriented environment. Gul Agha in *A Message-Passing Paradigm for Object Management* describes the Actor model of computation emphasizing the concurrency issues and how they are resolved in the model. Dennis McLeod and Surjatini Widjojo in their paper *Object Management and Sharing in Autonomous, Distributed Data/Knowledge Bases* conclude the issue by outlining issues that need to be addressed when dealing with distributed aspects of object management and in particular communication and sharing of information among objects.

I would like to thank all the authors for accepting my invitation to contribute to this issue. I have enjoyed working with them and learning about their research and development efforts in object-oriented systems. I hope that the reader will find this issue both informative and stimulating.

Fred Lochovsky

December, 1985.
Toronto, Canada

Object Species

D. Tsichritzis

Université de Genève

ABSTRACT

This paper outlines a framework for end-user-oriented objects. We are interested in the specification and implementation of complex objects which have a simple external behaviour. Users can visualize the external behaviour through analogies. We plan to use this environment in the context of Office Information Systems in general and sophisticated Message Systems in particular.

1. INTRODUCTION

This paper discusses a conceptual framework for end-user-oriented objects which can be useful within an Office Information System. Our objects will be based on a particular object-oriented environment [NIER83, NIER85a, NIER85b]. However, given suitable facilities, our conceptual model can be implemented on top of different object-oriented systems. The objects that we will use in this paper are related to Smalltalk objects [GOLD83, GOLD84], Actors [HEWI77, THER83], monitors [HOAR74] and abstract data types [GUTT77].

We will explain the concepts by using analogies from the animal world. The analogies serve two purposes. First, they provide a user model for the behaviour of the system [LEE85]. Second, they illustrate the design choices and the implementation difficulties. We hope that the reader will not be distracted by the analogies and lose sight of the technical nature of our discussion.

We start by defining the two most essential concepts in our object world *light* and *matter.* Light corresponds to information and emanates from the users. Matter corresponds to data and is the encoding of information within the system. The interplay of light and matter according to the rules of creation produces life. The interplay of user information and system data according to the rules of the application produces knowledge. Life is perceived in terms of living objects. Knowledge in our environment is encapsulated as objects which will be called *KNO*wledge acquisition, dissemination and manipulation objects, in short *kno's* [TSIC85].

Kno's contain *data* (matter) and *rules* (soul). *Kno*'s interact with users directly, or through other defined *kno*'s. *Kno*'s can be *alive* or *dead.* They are alive if they can participate in events orchestrated by an object manager. When they are dead both their data and rules become part of the (local) *data base* (mother earth). *Kno*'s can move around between environments. An *environment* is a collection of *kno*'s under the jurisdiction of an *object manager.* The object manager controls events between *kno*'s and it

Author's address: Université de Genève, Centre d'Informatique, 24, rue General Dufour, CH 1211 Genève 4, Switzerland (22/20.93.33).
UUCP: decvax!mcvax!cernvax!cui!dt.

oversees the birth and death of *kno*'s. Object managers play, in essence, the role of god.

To summarize, we can fantasize about and represent our world, the *objectworld*, as a galaxy of stars. Each star is under the jurisdiction of its object manager (the local god). Each star consists of *matter*, the local data base, and *kno*'s, its living objects. Objects can move from star to star. They behave according to their own rules and the local conditions. The general terms under which they live are enforced by the object manager.

2. OBJECT SPECIFICATION

We are interested in defining objects which can be potentially useful to users. The objects are defined in terms of an object specification language [NIER85b]. We expect these objects to be defined by application programmers using the object specification language, and then made available to users. We do not require, therefore, that they be simple, or easy to define. We expect them, however, to be easy to describe to users, possibly by using physical analogs. We also expect them to have wide use.

A user-oriented *Kno* consists of: a *head*; a *body*; and a *behaviour*. The *head* contains the *kno*'s identifier and information on acquaintances and past history. Detailed contents of the head are not visible outside a *kno*. The head's contents are needed mainly by the object manager for housekeeping purposes.

The *body* of a *kno* consists of a set of named relations using a set of named attributes. The *kno* can also have other local variables used in its rules. The body is the part of its data structures which are visible outside the *kno*. We expect that the attributes of a *kno*'s body conform to a universally known set of domains. In this way the basic properties of such attributes are known across *kno*'s. One way to achieve this requirement is to assume that a *kno* is always an instance of a given *kno* class. In this case the body types of allowable *kno* classes are known and understood by other *kno*'s and eventually users.

The behaviour of a *kno* consists of a set of rules. These rules govern how the *kno* reacts to stimulus from the outside world. Some of the rules are visible from the outside world, in the sense described above, while others are invisible, in the sense that they do not affect directly other *Kno*'s, users, or the data base. These invisible (internal) rules, while not affecting the *Kno*'s external behaviour, are important for its operation.

Rules have the form:

<cause> ==> <effect>

and are indivisible. They operate potentially in parallel, but in a serializable fashion. Rules are performed when the <cause>-part is satisfied completely. They perform what is prescribed in the <effect>-part.

The <cause>-part is the *conjunction* of a number of clauses. Each clause is an instance of the following types:

1. <user clause>

   This is like an oracle coming from outside the object world. It implies the desire of a user to perform the rule represented by the behaviour, provided the other <cause>-clauses are also satisfied *concurrently*.

2. <object clause>

   A specified object, an *acquaintance*, has to perform a corresponding rule at the same time. The corresponding rules are fired together in one event provided all their clauses are satisfied.

3. <data base clause>

   A Boolean condition of data base selections is specified in terms of the data base available in the *kno*'s environment (local data base). This feature enables the deposition of values in the data base by *kno*'s to be used later to invoke rules of other *kno*'s.

4. <body clause>

   A Boolean condition of selections and comparisons is specified between attributes of the *kno*'s body and local data base attributes. This feature allows sequencing of rules by modifying internal variables used in the <cause>-parts of other rules.

The <effect>-part of a rule is a *series* of actions which are performed in order. The rule is not complete (and it should be rolled back) unless all the actions are performed. The actions can be of the following types:

1. <body action>

   Results in the evaluation of an expression, and the assignment of its result to one of the *kno*'s body attributes. Using this feature, a *kno* can obtain information from the local data base.

2. <data base action>

   An assignment of an expression of body attributes and local data base attributes to the local data base. With this feature a *kno* can deposit information to the local data base.

3. <object action>

   This action allows the triggering of a rule in an acquaintance, including the possibility of exporting some *body* values to another *kno*. This action presupposes that the other *kno* is available and willing to operate its corresponding rule. If not the action rule cannot be performed.

4. <user action>

   This action comes as an alarm to the user. It also presupposes that the user is there to receive it, or else the rule is not performed.

3. EXISTENTIAL RULES

There are a number of rules which are special. We will call them *existential rules* because they affect crucially the existence of objects. Their <cause>-parts are the same as any other rule. We would actually expect them to be triggered by simple body or local data base flags to make their firing conditions very clear. Their actions are very important to other objects, other environments, and themselves. We will enumerate the actions of special existential rules.

1. <move action>

   The object performing this rule is moved to another environment as specified. The object is removed from the environment of its current object manager. The move has ramifications in terms of disabling rules waiting to fire and has to be handled carefully.

2. <die action>

   The object commits suicide by firing this rule. In essence, it is like a move to nowhere. The body of the object falls back as part of the local data base. We would expect a data base type corresponding to each object class. The rules are also stored (but they need be stored only once for each object class).

3. <export action>

   The object exports by copying a rule or a part of its body to another object. We require that the other object fires a corresponding rule with an import action. Exporting-importing are parts of an event. The importing object stores the imports in a special place. Imported rules and bodies do not augment the existing rules and bodies. Imported bodies and rules can only be read and passed over to children objects. The importing object does not change class.

4. <grow action>

   A subset of an object's rules and body are copied in a new, subservient object. The new object does not operate independently but behaves like a *limb* of the original object. Limbs cannot grow other limbs. Only ***heads*** can grow limbs. In this way there is centralized control of all the limbs. Limbs however, can move to other environments. Each time a limb moves the head is automatically notified.

5. <spawn action>

   A subset of an object's rules and body are copied into a new and ***independent*** object. Imported bodies and rules can also take part in the formation of the new object. This is really the way that an object can endow its offspring with more rules than it has. An existing object can not become more "intelligent" in terms of rules. It can, however, gather rules and pass them to its children. Before an offspring is born we need to check for rule consistency (genetically doomed offspring cannot be born).

4. KNO SPECIES

Using the rules outlined in the previous sections, we can specify many different kinds of *kno*'s. Some of them may even be odd or demonstrate unsocial behaviour. It is important to identify and obtain classes of useful *kno*'s and eventually even categories of similar classes. We started with an analogy between *kno*'s and living objects in the real world. To continue the analogy, we will establish correspondences between categories of useful *kno*'s and categories of living objects. Most categories of living objects are defined in terms of the way they breed, what they eat, and how they move. We will see that *kno*'s fall into categories in a similar way.

A *plant kno* is a *kno* which has many restrictions on its use of existential rules.

1. A plant *kno* cannot move (i.e., its rules cannot have a <move action> to another environment). A plant *kno* can have limbs, but its limbs remain local.
2. A plant *kno* interacts with users, the data base and other kinds of kno's, but not with other plant *kno*'s. In this way, the plant *kno*'s "grow" independently.

There are many examples of useful plant *kno*'s. Views in data bases are examples of plant *kno*'s. They provide a way of selecting and transforming data for a particular use. A knowledge base is another example of a plant *kno*. Rules of inference are defined based on the data of a data base. The rules give a way of interpreting the data and making some deductions. Note that the information and constraints inside different plant *kno*'s may be different. In that way, plant *kno*'s can provide different opinions based on the same information. Consistency is not enforced across *kno*'s, but only within them.

Any *kno* which moves will be called an *animal kno*. There are obviously many different cases. We will distinguish them by the way they breed, move, and eat.

An animal *kno* can be simple or may have a head and limbs. An animal *kno* may have limbs residing in different environments. The coordination of such limbs may become a problem, especially if the communication facilities are unreliable or intermittent.

A simple animal *kno* without limbs can perform move actions in which case it will hop to another environment. An animal *kno* with limbs moves in a far more complex manner. Each limb can move independently. We expect, however, the moves to be serialized and controlled by the *head*. An interesting case is an animal *kno* which keeps its head stationary but grows and moves its limbs. In this way it can grow objects in different environments while the control remains centralized. An example of such *kno*'s are intelligent messages as in I-mail [HOGG85].

An animal *kno* can produce children by using a <spawn action> in its rules. If the child *kno* inherits a subset of the rules of the parent, it is not that interesting. The interesting case arises with the export of rules from one *kno* to another. The second kno can then produce a child which is augmented by the imported rules. In this way, many animal *kno*'s can participate in the birth of a new animal *kno*. Each one deposits rules in one "mother" *kno* which then spawns a child with a combination of all the rules deposited.

Animal *kno*'s can "eat" by importing data and rules. They can import from the data base, from other *kno*'s and from the users. If they import information from plant *kno*'s rather than the data base, they can get them preselected and possibly transformed rather than as raw data from the data base. Animal *kno*'s importing from plants are like herbivores. They "eat" plants. Animal *kno*'s can import from other animal *kno*'s either by copying or by copying and then destroying. In the latter case, they operate as predators. Predator *kno*'s can be very important for population control. For instance, if we want to destroy a free roaming *kno* we can do so by releasing a predator *kno* to chase it. In this way we decide its destruction dynamically. Additionally, we may want to destroy a *kno* of which we have lost track. In this case, only a predator *kno* can find it and destroy it [SHOC82].

There are many simple cases of extremely useful animal *kno*'s. A ***carrier kno*** is a simple *kno* which moves around and can carry a message. It is destroyed when the message is delivered or deposited. A ***bee kno*** is a simple *kno* which can hop around among plant *kno*'s transmitting information from one to another. A ***shepherd kno*** is a *kno* which can trigger move actions in other *kno*'s and influence them to assemble in the same environment. One of the most interesting *kno*'s is an ***actor kno*** whose rules allow it to interpret imported rules. In this way it can act out any rules [HEWI77]. A ***universal actor kno*** is a *kno* which can act as any other *kno* if it is provided with the appropriate specification.

We expect *kno*'s to be prepackaged, that is prespecified and ready to use. A user can then specify some parameters on its behaviour and give it life through an object manager. Hopefully, the analogy with the real world can help the user visualize immediately the behaviour of the *kno* he is using.

5. IMPLEMENTATION

We are embarking on an implementation of the framework discussed in this paper. The implementation consists of four distinct aspects closely coordinated. First, we are implementing a new object-oriented system as outlined in a companion paper in this issue [NIER85c]. Second, we are working on the specification of many "useful" *kno*'s using our object-oriented specification language. Third, we are trying to put together an appropriate user interface for visualization and external manipulation of such *kno*'s. Finally, we are implementing a sophisticated message system where both messages and user roles are objects with *kno* behaviour.

The purpose of the project is to explore interesting ways for object birth, growth, death, inheritance, and cooperation. We are interested in exploring the possibility of specifying very complex *kno*'s while retaining some control on *kno* population behaviour.

6. EVOLUTION

To complete our analogy, we will end with a parallel between world and *kno* evolution. As mentioned in the first section, at the beginning there were users and light (information). Then there was matter (data) which was the equivalent of light, but in more concrete form. Programming languages, including object-oriented systems, give us the tools to construct any living cell (program). The problem is always to combine the primitive cells to achieve an overall purpose.

In traditional programming environments, the cells are put together in a very careful way to construct a rather unwieldy animal which resembles a dinosaur. We call it an application system. As dinosaurs, application systems are all powerful and they do well a particular job. However, they are difficult to deal with and the users are like pygmies running around them with bows and arrows. Every now and then the pygmies get sick and tired and they kill the dinosaur. However, another one sooner or later takes its place.

This form of life was particularly successful until now, but we see emerging a new environment for the following reasons. First, we cannot easily distribute dinosaurs in little pieces. Second, users are getting sophisticated. They shoot down dinosaurs at a very fast pace. Third, the environment of a system's operation changes very rapidly and dinosaurs adapt with great difficulty.

What we need is animals which are smaller, adapt quickly, move freely, and are dispensable. In terms of our conceptual framework, we need *kno*'s which move around, can import data and rules from their environment, and can multiply rapidly. In such an environment, new problems become important. The problem is not how to design a dinosaur from non-existing or shaky specifications. Rather, the emphasis is on how to control the object population, how to make objects adapt in foreign environments, how to make them cooperate with each other, and how to relate them to users. There is no such thing as the perfect *kno*, or perfect animal for that matter. The perfection lies in the harmony between *kno*'s or animals. This harmony is obtained through a reasonable cooperation between imperfect objects, and imperfect users, in a fast changing environment.

REFERENCES

[GOLD84] Goldberg, A., *Smalltalk 80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA, 1984.

[GOLD83] Goldberg, A. and Robson, D., *Smalltalk 80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[GUTT77] Guttag, J., "Abstract data types and the development of data structures", *Comm. ACM* **20**(6), 1977, pp. 396-404.

[HEWI77] Hewitt, C., "Viewing control structures as patterns of passing messages", *Artificial Intelligence* **8**(3), 1977, pp. 323-364.

[HOAR74] Hoare, C.A.R., "Monitors: an operating system structuring concept", *Comm. ACM* **17**(10), 1974, pp. 549-557.

[HOGG85] Hogg, J., "Intelligent message systems", in *Office Automation: Concepts and Tools*, Tsichritzis, D.C., ed., Springer-Verlag, Berlin, 1985, pp. 113-134.

[LEE85] Lee, A. and Lochovsky, F.H., "User interface design", in *Office Automation: Concepts and Tools*, Tsichritzis, D.C., ed., Springer-Verlag, Berlin, 1985, pp. 3-20.

[NIER85a] Nierstrasz, O.M., "An object-oriented system", in *Office Automation: Concepts and Tools*, Tsichritzis, D.C., ed., Springer-Verlag, Berlin, 1985, pp. 167-190.

NIER85c] Nierstrasz, O.M., "Hybrid: a unified object-oriented system," *IEEE Database Engineering* **8**(4), 1985.

[NIER83] Nierstrasz, O.M., Mooney, J., and Twaites, K.J., "Using objects to implement office procedures", *Proc. Canadian Information Processing Society Conf.*, 1983, pp. 65-73.

[NIER85b] Nierstrasz, O.M. and Tsichritzis, D.C., "An object-oriented environment for OIS applications", *Proc. 11th Int. Conf. on Very Large Data Bases*, 1985, pp. 335-345.

[SHOC82] Shoch, J. and Hupp, J., "The worm programs—early experience with a distributed computation", *Comm. ACM* **25**(3), 1982, pp. 172-180.

[THER83] Therault, D.G., *Issues in the Design and Implementation of Act2*, M.Sc. Thesis, Tech. Rep. 728, The Artificial Intelligence Lab., MIT, Cambridge, MA, 1983.

[TSIC85] Tsichritzis, D.C., "Objectworld", in *Office Automation: Concepts and Tools*, Tsichritzis, D.C., ed., Springer-Verlag, Berlin, 1985, pp. 379-398.

Task Management for an Intelligent Interface

W. Bruce Croft

University of Massachusetts

ABSTRACT

An intelligent interface assists users in the execution of their tasks. To do this, the system must be able to represent tasks and the objects that are manipulated. The intelligent interface described in this paper uses an object management system to manage object and task instantiations and the relationships between them. The object management system is viewed as an implementation of a data model that emphasizes the modeling of operations.

1. INTRODUCTION

In interactive computing environments, the users play a dominant role in determining the operation of the system by selecting the services or tools that are required for their tasks. A task is simply a sequence of activities, some of which are performed on the computer, which taken together accomplish users' goals. Examples of this type of environment are office information systems, software development environments and CAD/CAM systems. In such systems, the interaction with the user is typically viewed as an unpredictable series of tool invocations, rather than as the execution of tasks which are at a higher level of abstraction. The lack of knowledge of user tasks severely limits the role of the system during the interaction. To address this limitation, we define an ***intelligent interface*** as a subsystem that provides a means of describing and supporting the typical interactions users have with the computing environment. The primary function of the intelligent interface is to provide a wide range of assistance to users in the execution of their tasks.

The characterization of a user's interaction with a system presents a number of problems that cannot be addressed with conventional programming languages. The following features of task description are particularly important:

1. Tasks involve user actions as well as executable code. Often they are nondeterministic.
2. Tasks must be able to be specified by users with widely varying computer experience.
3. Task descriptions are often incomplete. The description of a task must be able to change as the user's understanding of the task changes.
4. Task descriptions represent only typical actions involved in carrying out a task. Exceptions to these typical patterns are very common.

This research was supported in part by the Rome Air Development Center and by Digital Equipment Corporation.
Author's address: Computer and Information Science Department, University of Massachusetts, Amherst, MA 01003 (413/545-0463). CSNET: croft@umass.

The POISE system [CROF84] was designed to address the problems of task definition and support. In this system, tasks are specified as underconstrained plans [COHE82, Ch. 15]. A task is described in terms of subtasks, associated objects, local variables, the preconditions for the task and the effect of carrying out the task. It is underconstrained in the sense that the exact ordering of subtasks is often not specified or only partially specified. The primitive tasks in a task hierarchy are either the operations provided by the tools or application programs. No further breakdown of these operations is necessary to execute them. Not all of the lowest-level tasks in a task hierarchy need be primitive tasks; they may currently only be specified at an abstract level or they may correspond to actions that occur outside the system (e.g., making a telephone call).

As the user specifies more details about a task, or as the system learns more about a task, the task descriptions are further constrained by the addition of rules that affect the ordering of subtasks or the relationships of objects or variables used by subtasks. New subtasks representing more detailed actions may be added. Examples of these added constraints are

- a rule specifying when step A must come before step B
- a rule specifying that the object used in step B is the same as the object in step D.

In this way, the system builds up detailed plans for tasks that are initially specified at a higher level of abstraction by the users.

The system uses the task descriptions to predict user actions (as well as automating aspects of the task). When an *exception* to the predicted action occurs, the system is alerted to the fact that its task description is inadequate and it can then take appropriate action. The emphasis on acquiring knowledge through exceptions is also found in Borgida's work [BORG85]. Many types of exceptions can occur including, for example, different orderings of subtasks, missing subtasks, subtasks activated with preconditions not satisfied, and object constraint violations.

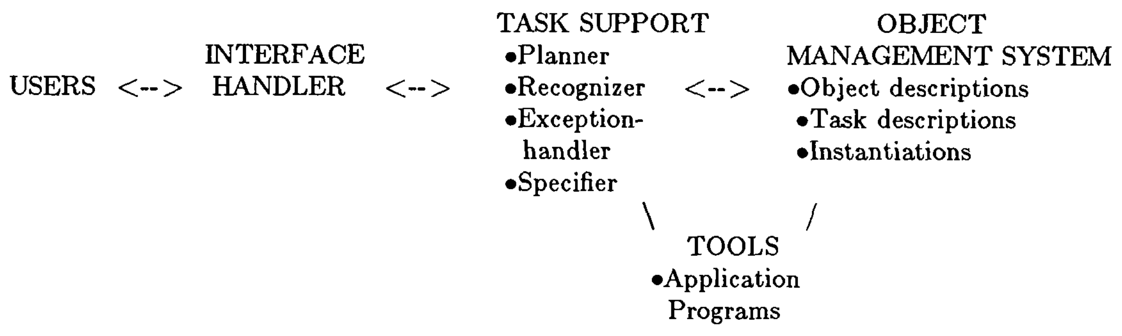


Fig. 1. The POISE system.

The basic architecture of the system incorporating task definition and support is shown in Figure 1. The interface handler is responsible for presenting to the users an integrated view of the tasks, tools, and objects that are available. Users can invoke tasks or manipulate objects directly with the tools. The task support module "understands" the user actions and choices, records them, and takes appropriate actions. This module has four major components. The *planner* executes plans (task descriptions). This includes predicting user actions and propagating constraints from one task step to another. The *recognizer* is used to recognize plans that the user is following without having been specifically invoked. This includes the recognition of exceptions. Recognition of plans in ambiguous situations requires sophisticated control and backtracking mechanisms [CARV84]. The *exception handler* is used to update task descriptions in response to specific user actions. The *specifier* provides the means for users to specify tasks. This specification is done through a graphical interface and requires the user to describe tasks in terms of subtasks, relationships between subtasks, and objects that are manipulated.

The object management system provides facilities for describing objects and tasks and for managing their instantiations. Tools can be viewed as a special class of application program that manipulates the objects stored in the object management system. For example, in an office system, the tools would include an editor, a forms package, a spreadsheet, and a mail facility. In this paper, we shall describe how the object management system can be considered to be an implementation of an extended data model.

TASK: Purchasing

$$\begin{tikzcd}
 & & \text{REQUISITION} \arrow[d, dash] & & \\
 & & \text{Fill-out-requisition} \arrow[dr, dash] & & \\
\text{Receive-purchase-request} \arrow[r, dashed] \arrow[dd, dash] & {} \arrow[ur, dash] \arrow[dr, dash] & & {} \arrow[r, dashed] & \text{Complete-purchase} \arrow[dd, dash] \\
 & & \text{Fill-out-order-form} \arrow[ur, dash] \arrow[d, dash] & & \\
\text{REQUEST} & & \text{ORDER FORM} & & \text{PAYMENT FORM}
\end{tikzcd}$$

Fig. 2. An example task.

A simple example of the operation of this system in the office environment is given by the purchasing task shown in Figure 2. This shows the task at the highest level of abstraction. The description of the purchasing task, its subtasks and the associated objects (such as the order form) reside in the object management system. The task description contains a constraint that a request for a purchase must occur before an order form or a requisition can be filled out. It also specifies that either one of those steps must occur before completing the purchase. The other form of constraint relates the contents of the request, the order form and the payment form. At this level of abstraction, the task description will look very similar to an ICN specification [ELLI82]. Some of the steps in the task description will be specified at a greater level of detail. For example, the "Fill-out-order-form" subtask may contain a detailed description of how this step is accomplished. Other steps, such as "Fill-out-requisition", may be only partially specified. It is the responsibility of the task support module to monitor the user's interactions with the system, recognize when a requisition is being used and to gather information that will further specify this step. Once the purchasing task has been specified by the user, it is presented by the interface handler as one of the "tools" available to the user. When a particular purchase is required, the user would invoke this task and the system would create instantiations of the purchasing task and related objects such as the order form.

2. DATA MODELS AND EXTENSIONS

Data models provide a means of defining the structure of objects in a particular environment, constraints on those objects, and operations that may be performed on them [TSIC82]. Much of the research in this area has concentrated on the static aspect of object description, rather than the dynamic aspect. To support the intelligent interface, however, we are forced to look at the task descriptions and ask how they are related to application programs, transactions and the data manipulation languages provided in conventional database systems. We define an extended data model as consisting of a means to describe objects, a means of describing operations and a means of describing the connections between objects and operations. Constraints are specified as part of both the object and operation definitions.

Object definitions are accomplished using a data model such as that described in Gibbs [GIBB84], which allows non-first normal form objects, generalization hierarchies and constraints defined using domain specifications and trigger procedures. For example, in an office application, an order form that contains a variable number of ordered items may be defined as a specialization of a general form object. The order form may inherit a constraint from the general form object that the form number should be between 1 and 99999. A specific constraint, that the total field should be the sum of the costs of the items, could also be defined.

The operations that can be defined include tasks, application programs, tools and transactions. The primitive operations, which are provided in the data manipulation language in database systems, are predefined and apply to all objects. These operations include creating, updating, deleting and retrieving objects. A containment hierarchy of operations, as shown in Figure 3, results from the observation that operations higher in the hierarchy are described in terms of operations that are lower in the hierarchy.

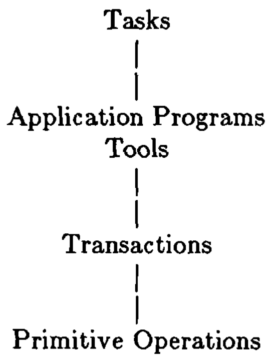


Fig. 3. Operation hierarchy.

A distinction can be drawn between atomic and non-atomic operations. The primitive operations and transactions are atomic in the sense that they are indivisible from the user's point of view. On the other hand, the steps involved in tasks and application programs can be visible to the users and may require user input. Task concurrency and constraint checking thus cannot be handled in the same manner as transactions. Delaying constraint checking until the end of a task, for example, is not possible because the intermediate states are visible to the users. The fact that tasks can be suspended indefinitely also requires that locking does not occur as it would for a transaction. These points lead to the conclusion that transactions can only be defined for the very low level operations from the user's point of view. The maintenance and checking of task instantiations in order to provide a consistent view of the system's operation to the user is entirely the responsibility of the task support module. For example, the task support module can assist the user in "undoing" the steps of a task and can check constraints whenever new information becomes available.

The main advantages of introducing task, application program, and tool operations into the data model are that the connections between user-level operations and objects can be made explicit and a common framework is provided for describing and managing the static and dynamic aspects of a system. Generalization hierarchies of operations, multiple instantiations of operations, and inheritance of operations through specialization of object types can all be described. For example, it is possible to describe a "Fill-out-form" task that is connected with a general form object. We could then describe a "Fill-out-order-form" task as a specialization of the more general task that includes more steps and constraints. An order form, which is a specialization of the general form object, would have a connection to the "Fill-out-order-form" task but would also inherit operations connected to the the general form, such as "Get-form-number".

In contrast to the Smalltalk view [GOLD83], where objects are defined through the operations that are attached to the object, our view is that the operations and the structure of the objects are both of interest and have separate descriptions, but are tightly connected (a kind of "marriage of equals"). An alternative description of the extended data model, which is more object-oriented in nature, would view tasks and objects as two subclasses of a more general object class. Operations that are attached directly to objects are atomic whereas task "objects" describe user-level operations that typically are non-atomic and manipulate a number of other objects. The extended data model is closely related to the model described by Stemple and Sheard [STEM82, SHEA85].

The description of the operations vary according to the operation type. Task descriptions were mentioned in the last section. The programming languages and data manipulation languages used to describe application programs and transactions in conventional database systems are the major part of the description of these operations, but other information is needed. From the point of view of the intelligent interface, the most important information about these operations is the name, the functionality, and the input/output characteristics. That is, given a task step, the POISE system has to know what lower-level operations can carry out that step, how these operations can be invoked, and what information is required.

As mentioned previously, the description of operations involves a definition of constraints. These constraints, either explicitly defined in task descriptions, or implicit in the application program code, define allowable *transitions* of the object instantiations and the operation instantiations. It has been recognized that static and transition constraints are not independent and that redundant specifications are not uncommon [SHEA85]. POISE is designed to use either form of a constraint during planning and recognition. For example, a task description constraint may specify that if an order amount is less than $500, the step "Fill-out-order-form" is appropriate, otherwise "Fill-out-requisition" should be used. In the description of objects, the same constraint could be specified by allowing only values less than $500 in the amount field of the order form. By allowing users to specify this constraint in either way, POISE simplifies the task description process.

3. OTHER MANAGEMENT ISSUES

A number of other problems arise in the management of the object and operation instantiations for the intelligent interface. One of these is that in this type of system it is essential to know which people or more accurately, "agents", can carry out tasks. The description of agents and the "roles" that they take has been the subject of previous research [ELLI82]. In the system described in this paper, agents would be represented as a class of objects with connections to both tasks and other objects.

During the process of planning and recognition, the intelligent interface must keep track of assumptions that are made in order to backtrack should a mistake be made or if the users change their actions. Part of this record keeping involves version histories of the objects [ZDON84]. However, in the intelligent interface, histories of operation instantiations are also required. This situation is further complicated by the fact that there may be multiple interpretations of a single user action, only one of which may turn out to be valid. The process of planning also requires the propagation of constraints into "predicted" versions of the objects. The interpretations in this system are similar to *contexts* used in some systems developed for artificial intelligence research [BARR82, p. 35].

By representing operations and objects in a single framework, the management problem is considerably simplified. A task instantiation can have a set of object instantiations associated with it. These object instantiations can be either "base" objects or "constraint" objects. Base objects record the state of the objects as seen by the users. Constraint objects are used as placeholders for propagating constraints and making predictions. The definition of a constraint object is a "relaxed" version of the base object definition. For example, a particular field in a base object may be specified as containing an integer in the range 1 to 100. The constraint object version of the field has to be able to hold values such as "$20<x<60$" to allow for symbolic propagation of constraints.

The object management system is partially implemented using a frame-based language [WRIG83]. At this level, both the operations and objects are represented as *frames*. Facilities such as generalization hierarchies and triggers are typically supported in these languages. The *slots* of the frames can hold any type of information, including code, and can therefore be used for the complex datatypes and constraints used in the extended data model. The planner and recognizer have previously been implemented as independent modules and are currently being reimplemented to take advantage of the object management system.

ACKNOWLEDGMENTS

The author benefited from many discussions with David Stemple. POISE was designed jointly with Victor Lesser.

REFERENCES

[BARR82] Barr, A. and Feigenbaum, E.A, eds., *The Handbook of Artificial Intelligence, Vol. 2*, William Kaufmann, Los Altos, CA, 1982.

[BORG85] Borgida, A. and Williamson, K., "Accommodating exceptions in databases and refining the schema by learning from them", *Proc. 11th Int. Conf. on Very Large Data Bases*, 1985, pp. 72-81.

[CARV84] Carver, N.F., Lesser, V.R., and McCue, D.L., "Focusing in plan recognition", *Proc. AAAI Conf.*, 1984, pp. 42-48.

[COHE82] Cohen, P. and Feigenbaum, E.A., eds., *The Handbook of Artificial Intelligence, Vol. 3*, William Kaufmann, Los Altos, CA, 1982.

[CROF84] Croft, W.B. and Lefkowitz, L.S., "Task support in an office system", *ACM Trans. on Office Information Systems* **2**(3), 1984, pp. 197-212.

[ELLI82] Ellis, C.A. and Bernal, M., "Officetalk-D: an experimental office information system", *Proc. ACM SIGOA Conf. on Office Information Systems*, 1982, pp. 131-140.

[GIBB84] Gibbs, S., *An Object-Oriented Office Data Model*, Ph.D. Thesis, Dept. of Comp. Sc., Univ. of Toronto, Toronto, Canada, 1984.

[GOLD83] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[SHEA85] Sheard, T., *Proving the Consistency of Database Transactions*, Ph.D. Thesis, Comp. and Inf. Sc. Dept., Univ. of Massachusetts, Amherst, MA, 1985.

[STEM82] Stemple, D., *Generalized Type Specifications for Database Systems*, Tech. Rep. 82-15, Comp. and Inf. Sc. Dept., Univ. of Massachusetts, Amherst, MA, 1982.

[TSIC82] Tsichritzis, D.C. and Lochovsky, F.H., *Data Models*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[WRIG83] Wright, M. and Fox, M.S., *SRL 1.5 User Manual.* Intelligent Systems Lab., Robotics Inst., Carnegie-Mellon Univ., Pittsburgh, PA, 1983.

[ZDON84] Zdonick, S., "Object management system concepts", *Proc. ACM SIGOA Conf. on Office Information Systems*, 1984, pp. 13-19.

An Object-Based Approach to Modelling Office Work

C. C. Woo
F. H. Lochovsky

University of Toronto

ABSTRACT

Building office systems to support office work is often very difficult and time consuming. Some researchers argue that this is due to the lack of appropriate programming tools, and they believe that "objects" are a natural primitive for programming many office applications. However, object systems developed so far still lack the appropriate tools to program non-procedural office work. We believe that by putting the right facilities into objects we can have very powerful, yet uniform, tools for describing different types of office work. In this paper, we describe such a model and use an example to show how it can be used to support office work.

1. INTRODUCTION

Office activities involve both data and procedures. It has long been recognized that information (i.e., data) is a corporate resource. The need for companies to handle greater amounts of information has led to the use of technologies such as data base management systems to manage their data. However, office procedures are also a corporate resource (i.e., how you do something is as important as the information you need to do it). Currently, this knowledge of office procedures resides entirely with the office worker and is best known to the person who performs the tasks. As the size and scope of an organization increase, there is a need to manage this knowledge of office procedures in the way we handle information using the computer.

In constructing a model to represent office procedures, we must be able to take into account the following characteristics of office work:

Office activities are parallel, distributed and interactive. Offices and their activities are distributed both in space and time. They involve the continuous coordination of parallel activities in many locations [HAMM80, HEWI84, ZISM78]. Rarely are problems solved, or goals realized, except by the interaction of several persons in a back-and-forth exchange of work, ideas and commitment. Therefore, communication is vital in office work.

Domain knowledge is open-ended. For both new office tasks as well as existing ones, situations will arise for which there is no a priori domain knowledge [FIKE80, LOCH83]. Under this situation, the office worker, who is in charge of a particular task, will have to handle it using some common sense knowledge

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada under grant A3356.
Authors' address: Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, M5S 1A4 (416/978-6986). CSNET: carson.toronto@csnet-relay.

about the world.

There is a wide range of types of office work. Many researchers [GORR71, LOCH83, PANK84] have recognized that office work falls on a continuous spectrum, with one end of the spectrum representing structured office work and the other end of the spectrum representing unstructured office work. Structured (also known as type I) office work is that of a routine nature for which a prescribed step by step solution is known. Unstructured (also known as type II) office work is novel and nonrepetitive, and must be solved with creativity, initiative, and originality. For this type of office work, we cannot build systems to replace existing office workers. Rather, we should build systems to support them in defining their goals and strategies [FIKE80, BARB83, LOCH83].

In finding a representation for supporting office work, we have concluded that by putting the right facilities into "objects" we can model the office characteristics mentioned above very nicely. Another major advantage of using "objects" is that other researchers have found them useful to model office data, office procedures, and office communication (e.g., models in [GIBB84, HOGG84, TWAI84, TSIC85]). In our approach, we merely extend the object world to handle more complicated office work and thus provide a uniform representation to model the office. In this paper, we will describe what we mean by objects in office work and discuss how we can use them to support office work.

2. OVERVIEW OF THE MODEL

We use objects in our model to describe some meaningful entities in the office that can be used to perform office work. Some examples of objects are office workers, office procedures, and file cabinets. Objects have their own data, rules, and behaviour. We will define an object type in office work to consist of [TWAI84]:

```
<object type> : <super class> {
    acquaintances
    variable declarations
    ALPHA rule
    rule_1
        .
        .
        .
    rule_n
    OMEGA rule
}
```

The format for a rule is as follows:

```
<rule name> (parameters) {
    acquaintances
    variable declarations
    preconditions
    actions
} (a value returned to the caller)
```

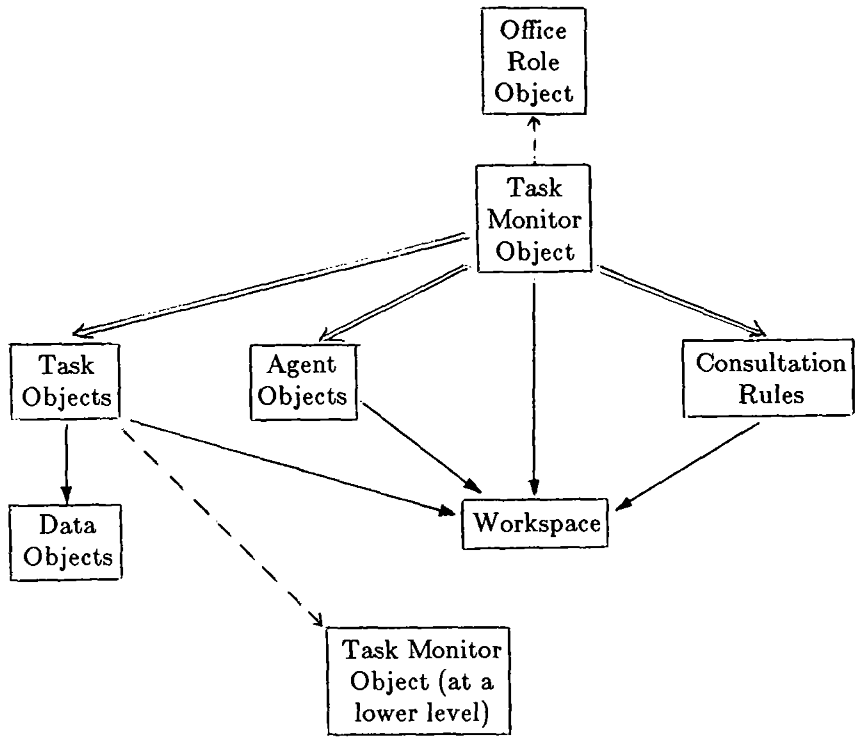
"Acquaintances" is a list of other object types that can communicate with this object type. Variables are used to provide storage and data structures for the object type. The instances of an object type have the same set of acquaintances and rules, but different values for variables. There are two special rules, called ALPHA and OMEGA, which are used to create and delete an instance of the object type respectively.

There are five different classes of object types in our model: office role objects, data objects, task objects, agent objects and task monitor objects. Each one of them plays a distinct role in our model. The relationships among them are shown in Figure 1.

Office role object

An office role is the set of actions and responsibilities associated with a particular office function [WOO84]. An office worker performs office work in our model by playing the correct office role. An office worker can be associated with more than one office role and vice versa. "Chief Programmer", "Director of CSRI", and "Secretary" are all examples of office roles. The notion of an office worker playing an office role provides logical independence in specifying the capabilities of office workers with respect to system resources and facilities.

Since office workers play an important role in our office work model, they are represented as office role



$\Longrightarrow$ means supervises the execution of the objects pointed to.

$\longrightarrow$ means allowed to access the information in the objects pointed to.

$\dashrightarrow$ means allowed to pass control to the object pointed to.

Fig. 1. Objects used in our model.

objects. There is a special rule in an office role object called the *io-rule* which allows office workers to communicate with other object types in the system [TWAI84].

Data objects

Data objects store inactive information. They serve the same purpose as file cabinets in a traditional office. However, they are more "intelligent" than file cabinets. The rules in data object types are used to trigger events when certain conditions become true. For example, on the tenth day of each month, if the manager of a sales division has not placed his last month's sales figures in the "sales" data object, a reminder can be sent to him by a rule in the "sales" data object.

Task objects

We use task objects to model office procedures. This is done by modelling a task done by an office worker as a task object. If this task is not routine enough to be expressed procedurally, a task monitor object will be attached to it so that problem solving can be done with the help of consultation rules and the user. Otherwise, the task object should be able to perform the task by itself. Communication among task objects is done using a message passing mechanism. Enough information is given to each task object such that it is possible for them to coordinate themselves to accomplish a particular task.

In some applications, the sequence of executing the task objects is known or partially known. This is handled in our model by ALPHA and OMEGA rules. As an example, let us consider the task in Figure 2 which requires five task objects. The arrows in the figure are used to indicate the execution sequence for the task objects. In this example, an instance of obj_1 is created by one of the following: (1) a user, (2) a task object instance in another workstation, or (3) an instance of a data object type which has rules to trigger events. When obj_1 finishes its job, its OMEGA rule will wake up task objects obj_2 and obj_3 at the same time and pass them relevant data. obj_4 can be awakened by both obj_2 and obj_3. However, obj_4 will go back to sleep if one of obj_2 or obj_3 has not finished its job yet. This is handled by the ALPHA rules in obj_4. obj_5 is the last task object to be awakened in this example before the task is done.

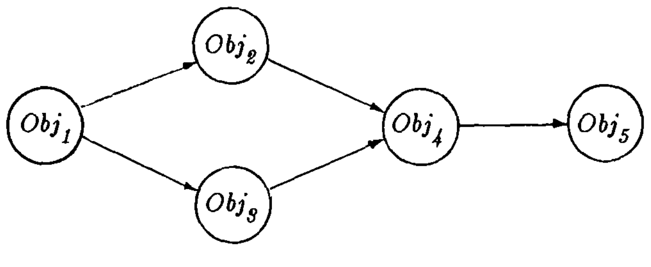


Fig. 2. A task that requires 5 task objects.

In more complicated situations, the OMEGA rule can provide alternatives if the task objects it wakes up fail to accomplish their jobs. However, situations might arise for which all of the given alternatives fail to accomplish their jobs. Furthermore, it is also possible that the OMEGA rule contains no information about what task objects to awaken. Under these situations, the task monitor object will take over and control the execution of the task objects. Again, the task monitor object handles the execution with the help of the user and the consultation rules.

Task monitor object

The organization of an office is normally partitioned into logical workstations. These logical workstations form the structure (either a hierarchy or network) of the company. In our model, a task monitor object corresponds to a logical workstation in an office. Objects such as task objects and data objects are therefore associated with a task monitor object. The dotted arrow in Figure 1 shows how task monitor objects are connected together.

The function of a task monitor object is to control the execution of task objects and agent objects, and the use of consultation rules (see Figure 3) within its jurisdiction. It contains an agenda which is used to record the execution history of tasks. This agenda is useful for debugging conflicting rules specified in the task objects and/or consultation rules.

There are three different communication methods used in our model. If a task object knows exactly which other task object it wants to talk to (i.e., routine and procedural office work), it will use the message passing mechanism to communicate with that task object. However, in some situations, a task object might want to talk to another, unknown task object (i.e., non-routine or special case office work). In this case, if the two task objects reside within the same task monitor object, the *workspace* serves as the message dissemination medium. Otherwise, an agent object is used to set up the communication (see below).

The workspace is thus the temporary storage of the task monitor object. It differs from variables in an object type in two ways. First, variables can be added or deleted from the workspace easily during execution. Second, a variable in the workspace can contain multiple values, unless it is explicitly declared to be unique.

Consultation rules are a special set of rules in the task monitor object. They are only used to assist the task monitor object. They do not perform any office work such as computation or communicating with other objects. Most of these rules are one of the following types:

1. Control rules (e.g., selection criteria when more than one task object is awakened and they cannot be executed concurrently).

```
while goal not achieved do
    if no task objects trigger then
        if no consultation rules can give advice then
            if no agent objects trigger then
                notify user
                user performs some action
            else
                communicate with foreign objects
                update workspace
            end if
        else
            update workspace
        end if
    else
        if more than one task objects trigger and
            some consultation rules can give advice then
                put some task objects to sleep
        end if
        perform task
        update workspace
    end if
end while
```

Fig. 3. The control flow of task monitor object.

2. Default rules which include equivalent information, simple implicit knowledge, common sense knowledge and so on. Some examples are:
   - (miscellaneous expenses) = (other expenses)
   - taken CSC364 ===> knows about "NP-Complete"
   - December 25th is a holiday.
3. Reduction rules (e.g., to compute Y, we must first compute X_1, ..., X_n).

Agent objects

An agent object is a representative sent by a task monitor object to communicate with an object (call it *X*) located in another task monitor object. Hence the name "agent". If *X* is a data object, then an agent object can also be viewed as a tool to perform distributed queries. If *X* is a task object, then an agent object is used to join up necessary information (i.e., information required to accomplish a task) in different workspaces of two or more task monitor objects. Finally, if *X* is a user, then an agent object is similar to Imail [HOGG84].

Due to the uncertainty of what *X* is and/or where it is located, possible locations of the information can be given to the agent object before it starts its journey. The agent object will visit locations with a higher certainty factor first. When visiting a particular location, if the information is not found, the agent object can modify its possible-locations list based on the information available at the visiting location. If the agent object successfully brings back the information requested, it will be placed in the workspace to be shared by all task objects associated with the sender task monitor object. On the other hand, if the agent object fails to bring back the required information after some pre-set time limit or after visiting all possible locations, it will return to the sender task monitor object and put a failure message in the workspace. An agent object will die by "committing suicide" after it has completed its function.

3. AN EXAMPLE

We use the following example to demonstrate how our model can be used to model office work:

> *The XXX Computer Company has several research laboratories located in different geographical locations. Each lab has its own Human Resource Department that is responsible for screening, evaluating, and hiring personnel at the lab. A research assistant currently working in* **lab A** *wishes to transfer to* **lab B**.

Figure 4 shows the task monitor objects (TMO) used to handle this particular application. The dotted arrows are used to show the passing of control to the object pointed to.

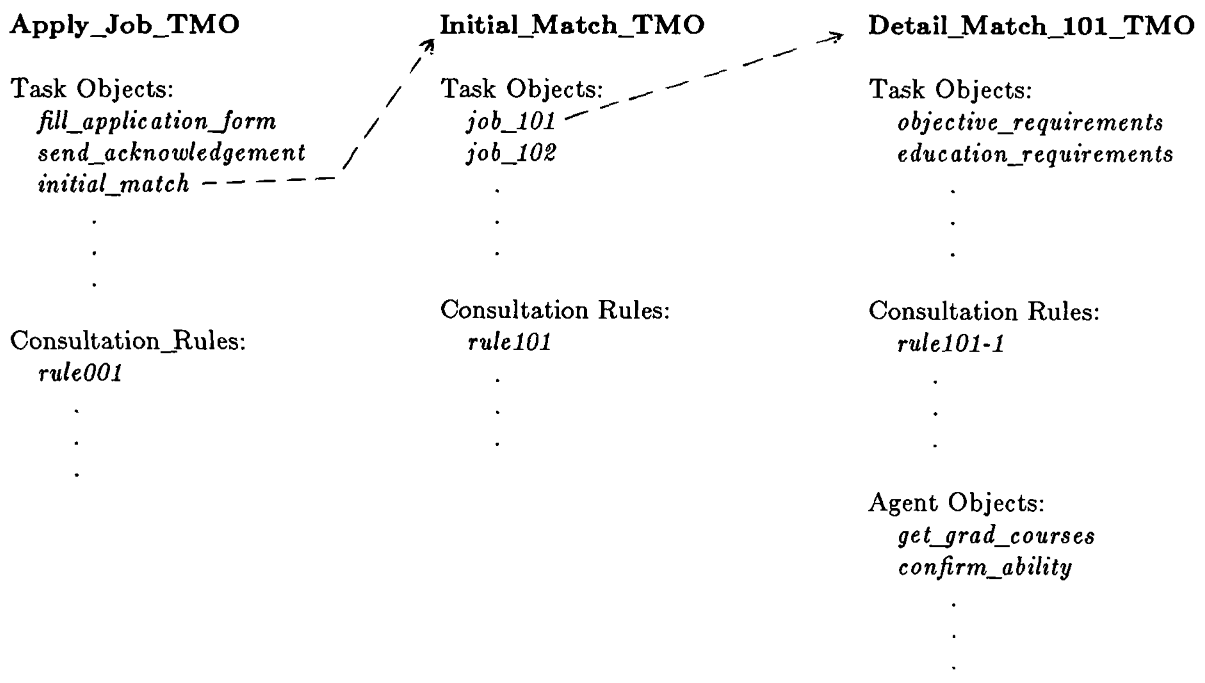


Fig. 4. Task monitor objects used in the example.

The functions of task objects such as *fill_application_form* and *send_acknowledgement* in **Apply_Job_TMO** are self-evident.

Task objects in **Initial_Match_TMO** are unfilled positions in the lab. The responsibility of **Initial_Match_TMO** is to try to assign an unfilled position to the applicant without worrying about the detailed job requirements. For example, to qualify for *job_101*, the applicant must hold a master degree in Electrical Engineering or Computer Science, and be interested in working in **lab B** as a system programmer. Consultation rules are very useful in this monitor. For example, if the applicant qualifies for more than one unfilled position, a consultation rule can inform the task monitor object to pick the oldest unfilled position.

The task monitor object **Detail_Match_101_TMO** is associated with the manager who has requested such an employee to work for him. It should have all the detailed requirements for the unfilled position. Task objects in this monitor are used to compute results that would lead to a decision (i.e., interview or reject the applicant). The consultation rules, on the other hand, provide information that would help the computations. These facilities will be illustrated in the rest of this section using the example in Figure 5.

When the task object *job_101* invokes the task monitor object **Detail_Match_101_TMO**, all

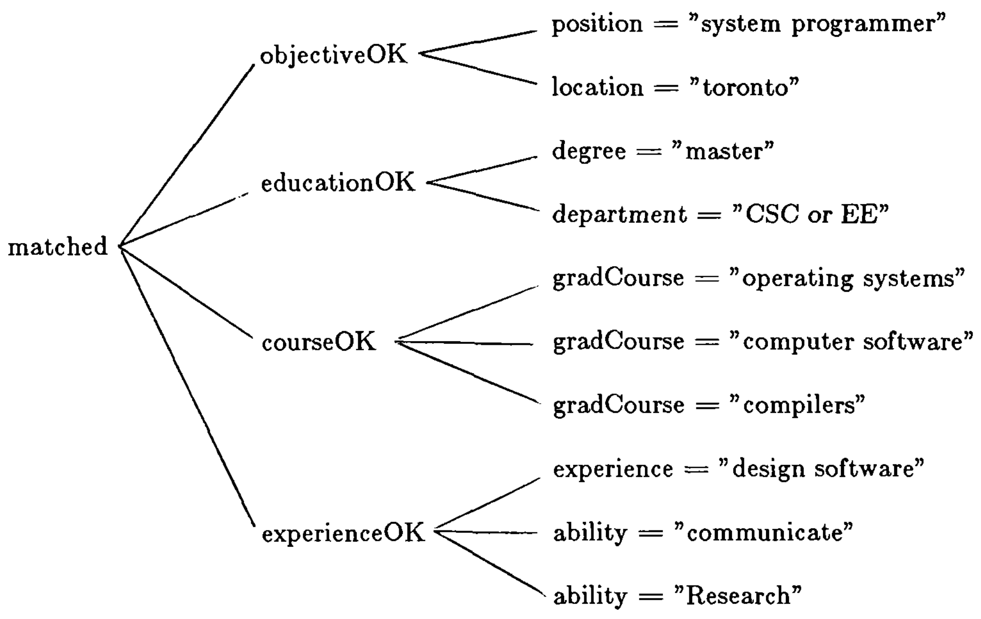


Fig. 5. Requirements of an unfilled position decomposed into a hierarchy of sub-requirements.

information about the applicant will be put into the workspace of **Detail_Match_101_TMO** (see Figure 6(a)). For the purpose of illustration, we have simplified the application form into four form fields: position, location, degree, and department. The goal of this task monitor object is to find a value for the variable "matched". The "?" sign in the figure means the variable does not have a value yet. Since the information in the workspace is not sufficient to compute "matched", no task objects will trigger. However, it is clear from Figure 5 that if we know the value for the variables "objectiveOK", "educationOK", "courseOK", and "experienceOK", we can compute "matched". Hence, a consultation rule (say *rule101-1*) puts this information into the workspace (see Figure 6(b)). Now, since the value for the variables "position" and "location" are available in the workspace, the task object *objective_requirements* triggers and fills in the value for "objectiveOK". Note that a task object can only fill in a value for an existing variable, it cannot create a new variable. Using similar information (see Figure 5), we can also compute the value for "educationOK". The result is shown in Figure 6(c).

Due to the open-ended nature of job requirements, the application form cannot capture all the required information to compute "matched". For example, the application form provides no information about graduate courses taken by the applicant. Yet the manager has imposed some requirements related to graduate courses taken. As a result, neither task objects nor consultation rules can carry out the computation in Figure 6(d). In this case, the task monitor object permits the triggered agent object *get_grad_courses* to perform its job. After *get_grad_courses* returns, information about graduate courses is put into the workspace as shown in Figure 6(e). Note that "computer software" is not the same as "software engineering". Therefore, none of the task objects, agent objects, nor consultation rules can continue the computation at this stage, and the user is informed. One possible action the user can take is to create a new consultation rule: "*computer software is the same as software engineering*", and use it to get to the result in Figure 6(f). Now there is enough information to provide a value for "courseOK". Similarly, to compute "experienceOK", the agent object *confirm_ability* can be sent to people who have supervised this applicant before to gather his work abilities. Finally, knowing the value for the variables "objectiveOK", "educationOK", "courseOK", and "experienceOK", we can compute "matched" and return the result to the caller task object (i.e., *job_101*).

(goal)	matched = ?
	position = "system programmer"
	location = "toronto"
	degree = "master"
	department = "CSC or EE"

Figure 6(a).

(goal)	matched = ?
	position = "system programmer"
	location = "toronto"
	degree = "master"
	department = "CSC or EE"
(new)	objectiveOK = ?
(new)	educationOK = ?
(new)	courseOK = ?
(new)	experienceOK = ?

Figure 6(b).

(goal)	matched = ?
	position = "system programmer"
	location = "toronto"
	degree = "master"
	department = "CSC or EE"
(new)	objectiveOK = true
(new)	educationOK = true
	courseOK = ?
	experienceOK = ?

Figure 6(c).

(goal)	matched = ?
	position = "system programmer"
	location = "toronto"
	degree = "master"
	department = "CSC or EE"
	objectiveOK = true
	educationOK = true
	courseOK = ?
	experienceOK = ?
(new)	gradCourse = ?

Figure 6(d).

(goal)	matched = ?
	position = "system programmer"
	location = "toronto"
	degree = "master"
	department = "CSC or EE"
	objectiveOK = true
	educationOK = true
	courseOK = ?
	experienceOK = ?
(new)	gradCourse = "operating systems"
(new)	gradCourse = "software engineering"
(new)	gradCourse = "compilers"

Figure 6(e).

(goal)	matched = ?
	position = "system programmer"
	location = "toronto"
	degree = "master"
	department = "CSC or EE"
	objectiveOK = true
	educationOK = true
	courseOK = ?
	experienceOK = ?
	gradCourse = "operating systems"
	gradCourse = "software engineering"
	gradCourse = "compilers"
(new)	gradCourse = "computer software"

Figure 6(f).

Fig. 6. Various stages of the workspace in **Detail_Match_101_TMO**.

4. CONCLUSIONS

We have briefly outlined a model that can be used to support office work. We believe that the notion of "objects" is very useful in describing office work. We have demonstrated this through an office example. Future research includes looking into the expressive power of the consultation rules. Implementation is another area to be investigated. Finally, we would like to explore the usefulness of this model in real offices.

REFERENCES

[BARB83] Barber, G.R., "Supporting organizational problem solving with a workstation", *ACM Trans. on Office Information System* **1**(1), 1983, pp. 45-67.

[FIKE80] Fikes, R.E. and Henderson, D.A., "On supporting the use of procedures in office work", *Proc. 1st Annual AAAI Conf.*, 1980, pp. 202-207.

[GIBB84] Gibbs, S.J., *An Object-Oriented Office Data Model*, Ph.D. Thesis, Dept. of Comp. Sc., Univ. of Toronto, Toronto, Canada, 1984.

[GORR71] Gorry, G.A. and Scott Morton, M.S., "A framework for management information systems", *Sloan Management Review* **13**(1), 1971, pp. 55-70.

[HAMM80] Hammer, M. and Sirbu, M., "What is office automation?", *Proc. 1980 Office Automation Conf.*, 1980, pp. 37-49.

[HEWI84] Hewitt, C. and de Jong, P., "Open systems", In *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages,* Brodie, M.L., Mylopoulos, J., and Schmidt, J.W., eds., Springer-Verlag, New York, 1984, pp. 147-164.

[HOGG84] Hogg, J. and Gamvroulas, S., "An active mail system", *Proc. ACM SIGMOD Conf.*, 1984, pp. 215-222.

[LOCH83] Lochovsky, F.H., "A knowledge-based approach to supporting office work", *IEEE Database Engineering* **16**(3), 1983, pp. 43-51.

[PANK84] Panko, R.R., "38 offices: analyzing needs in individual offices", *ACM Trans. on Office Information Systems* **2**(3), 1984, pp. 226-234.

[TSIC85] Tsichritzis, D.C., "Objectworld", in *Office Automation: Concepts and Tools,* Tsichritzis, D.C., ed., Springer-Verlag, Berlin, 1985, pp. 379-398.

[TWAI84] Twaites, K.J., *An Object-based Programming Environment for Office Information Systems,* M.Sc. Thesis, Dept. of Comp. Sc., Univ. of Toronto, 1984.

[WOO84] Woo, C.C. and Lochovsky, F.H., "Authorizations in a computer-based office information system", *Proc. IEEE 1st Office Automation Conf.*, 1984, pp. 81-90.

[ZISM78] Zisman, M.D., "Use of production systems to model asynchronous, concurrent processes", in *Pattern Directed Inference Systems,* Waterman, D. and Hayes-Roth, F., eds., Academic Press, New York, 1978, pp. 53-68.

Object Management Systems for Design Environments

Stanley Zdonik

Brown University

ABSTRACT

Object management systems can be distinguished from their more conventional counterparts (i.e., DBMS's) by their ability to deal with arbitrary object types in an environment that is constantly changing. They are particularly suited for environments such as office information systems, electrical CAD, and programming environments. All of these environments deal with the process of design.

This paper discusses the design of a particular object management system named ENCORE. It is based on the object-oriented paradigm and incorporates several other special-purpose facilities that are essential to the support of a design environment.

We describe a particular experimental approach to the construction of programming environments that is currently under investigation. We also mention the design of an interface for this style of database that has been developed at Brown.

1. INTRODUCTION

In the 1970's the field of database management developed out of a need to integrate and carefully manage the data that supported all of the applications within an organization. Database management systems (DBMS) were a technical response to this need. In the 1980's, workstations and workstation applications are becoming more wide-spread. There is a similar need to integrate these applications, but the database management tools from the 1970's are not particularly well-suited to a large class of workstation applications.

We envision the spread of a new generation of DBMS that addresses these needs. In order to distinguish these new tools from their predecessors, we will use the term *object management system (OMS)*. It should be noted that an object management system is a natural evolution of conventional database technology. It starts from the same basic philosophical underpinnings and moves in the direction of a broader application base.

One of the principal distinguishing capabilities of an OMS is its ability to deal with objects of arbitrary type. A conventional DBMS is in general limited to objects which are a parameterization of the type record. The term *type* has been used in many different ways in the computer science literature. We take it to mean a template that defines the operations that may be performed on the instances of that type and the properties that instances of that type may possess. This is similar to the definition used by

Author's address: Department of Computer Science, Brown University, Providence, RI 02912 (401/863-2364).
CSNET: sbz.brown@csnet-relay.

CLU [LISK77] and Smalltalk [GOLD83].

We define an ***object management system*** as a shared storage facility for objects of arbitrary types. It must, therefore, support the definition of types within the framework of the OMS. These type definitions must also be sharable.

Another cornerstone of an OMS is its ability to deal with change. We expect an OMS to be able to support applications in which evolution is the norm. An OMS should be able to deal with change at all levels of description. It should be able to handle the evolution of individual objects as well as the evolution of the types that define these objects.

The OMS should be powerful enough to capture knowledge about these dynamic environments and respond to change in intelligent ways. We expect that the system can help applications developers deal with the complexities that are introduced by continual change. For example, it can keep track of what changes have occurred, manage the execution of concurrent, unreliable transactions, automatically produce any side-effects that are required by a particular change, and maintain the integrity of the object store as a whole.

2. THE DATA MODEL

ENCORE (Extensible and Natural Common Object Resource) is an OMS that is being developed at Brown University. Currently, a basic prototype has been implemented to which we are adding additional functionality and tuning for increased performance. The ENCORE data model is described in this section. It is a high-level semantic model that is in the same spirit as several previous systems [CHEN76, HAMM81, MYLO80, SMIT77 ,TSIC82, ZDON84].

In ENCORE, a type has a set of explicitly-defined operations and properties, may inherit additional operations and properties from its supertypes, and has an associated class to which objects of the type automatically belong. A type T is introduced in the data definition language by a specification of the following form:

```
Define Type T
Supertypes: <list of supertypes>
Associated Class: <class-name>
    Operations:
        <list of operations>
    Properties:
        <list of properties>
```

The inheritance mechanism is similar to that of Smalltalk, but ENCORE allows multiple inheritance. The idea of a class that contains all declared instances of the type provides a feature not available for Smalltalk objects that is useful in performing retrieval on objects of any type. ENCORE endows objects with many of the attributes possessed by objects in Smalltalk-like languages and provides objects with additional explicitly-specified behaviors like version history as well as implicitly-specified behaviors like persistence and class membership. It is these additional attributes, orthogonally possessed by all objects, that differentiate object-oriented database languages from object-oriented programming languages.

The most general object type in ENCORE is type *Entity*. All other types are either directly or indirectly subtypes of type *Entity*. All instances of type *Entity* are by definition persistent. That is to say, any entity is automatically retained in a permanent store that is separate from the address space of the process that created it. It is placed in the proper classes which provide access paths to this object for future references.

The following definition of the type *File* has a supertype "Entity", an associated class "Files", three operations, and one property.

Define Type File
Supertypes: Entity
Associated Class: Files
    Operations:
        FILECreate-file ( ) **returns** (F : File)
        FILEOpen (F : File) **returns** (F : File)
        FILEClose (F : File) **returns** (F : File)
    Properties:
        Filename: String

The list of operations contains for each operation a specification of the types of its arguments and its return value. The list of properties follows, and for each property, its name followed by a colon and a description of the set of legal values for that property is given. The *Filename* property can have any string as a legal value.

A type T determines the operations that can be performed on instances of T. Each instance has an internal state that is represented by an object of some other type. This representation object is not visible outside of the *type manager* code (i.e., the code that supports the operations of the type). For example, an object of type *set* might be represented by an array. At any point in time, a set S will be represented by an array containing all members of S. It is not appropriate to include this information in the schema, since the schema specifies only the external behavior of the type and the representation should not be externally visible.

Our database system supports a separate notion of *class*. A class is a set of objects, as opposed to a type which is a description of the semantics of objects of that type. Every type T has an associated class C that holds all current instances of T (i.e., C = {x | type-of (x) = T}). For example, the header lines in the above file example indicate that the type *File* has an associated class named *Files*. Each time a new file object is created, it is automatically added to this class. Classes are related to each other with the *subset-of* property. It is possible to define additional classes to be subsets of existing classes by means of set selection operations (i.e., based on a predicate).

Also, a type T specifies the *properties* that instances of T can have. A property is an object that relates two database objects. For example, a source program S is related to its object code O by means of a property that might be called *compiled-version-of*. A property can be thought of as a function that maps the object that possesses the property to the object or set of objects that are the values of the property. We, therefore, have compiled-version-of (S) = O.

Types are objects, and as such, they can have the behavior of objects in general. This includes the ability to have properties. A type object has several properties including *properties-of* and *operations-of*. *Properties-of* has as a value the set of all the property types that can be associated with an object of the given type. Similarly, the property *operations-of* of a type T expresses the set of operations that can legally be applied to instances of T.

For example, in the definition of type *File* given above, the type *File* object has two properties *properties-of* and *operations-of* that it gets from being an instance of type *Type*. The *properties-of* property has as a value the set {Filename} and the *operations-of* property has as a value the set {create, open, close}. The schema definition that was given above is simply syntactic sugar for the proper operations to set up this structure.

Another important example of a property of a type is the *is-a* property. If we say that a type S is related to a type T by means of the *is-a* property (i.e., is-a (S) = T), then all instances of type S are also instances of type T. All of the semantics (i.e., all properties and operations) defined for type T must also pertain to type S. An instance of type S will have all operations and properties defined on T as well as any additional ones defined on S. We call this behavior *inheritance*. A file directory (similar to that used in UNIX) might be defined as follows:

```
Define Type Directory
Supertypes: File
Associated Class: Directories
    Operations:
        DIRECTORYenter-file (D : Directory, F : File)
                returns (D : Directory)
        ...
```

Here, the *Directory* type inherits all the properties and operations from the type *File* since *File* is defined to be a supertype. A type can have multiple supertypes and can, thereby, inherit properties and operations from several types. We call this *multiple inheritance.*

We treat properties as objects. It is, therefore, possible to have properties of properties. This is very useful for precise modeling of real world situations. Consider the property *compiles-into* that relates a source code module to its object code counterpart. If we wanted to express the fact that the compilation produced one warning message, we might very naturally wish to attach this fact to the property itself. It's not really a property of either the source module or the object module, but, rather, of the act of compiling. Perhaps, with a different compiler this message would not have occurred. We can define this as follows:

```
Define Type Source-Code-Module
...
    Properties:
        Compiles-into: Object-Code-Module

    Define Type Compiles-into
    Supertypes: Property
    Associated Class: Compiles-into-properties
        Properties
            Warning-Messages: Set of String
        Operations
            Set-Property-Value (C : Compiles-into, O : Object-Code-Module)
            Get-Property-Value (C : Compiles-into)
                    returns O : Object-Code-Module)
```

We also view operations as objects. This leads to several interesting effects. First, much as with properties, it becomes possible to assert things about operations. Second, in an object-oriented approach to system building, a new application is constructed by building new higher level types out of previously defined types. All code is associated with the operations of some type. Since the operations are also objects, all new applications are automatically subsumed by the database. The code (i.e., operation objects) persists by the general object storage mechanisms. Third, it is possible to use some of the more advanced features of the system like version control to manage the operations of an evolving application.

The structure of the basic data model is represented in Figure 1 using its own notion of type to describe itself. The basic types in the system are related to each other by means of the *is-a* property type. All boxes in this picture are instances of the type *type* (including the type *type* itself). The root of the system is the type *entity* of which all objects in the system are instances. To say that an object is an entity, is to say that it is a part of our database.

Notice that all entities are broken down into four basic subtypes, *operation* which describes the active elements in the system, *property* which describes a type of entity that is used to relate objects, *type* which describes those things that can be dynamically instantiated, and *aggregate* which describes those things that group entities together. Notice that the type *class* is a subtype of *aggregate* since classes are homogeneous collections of entities.

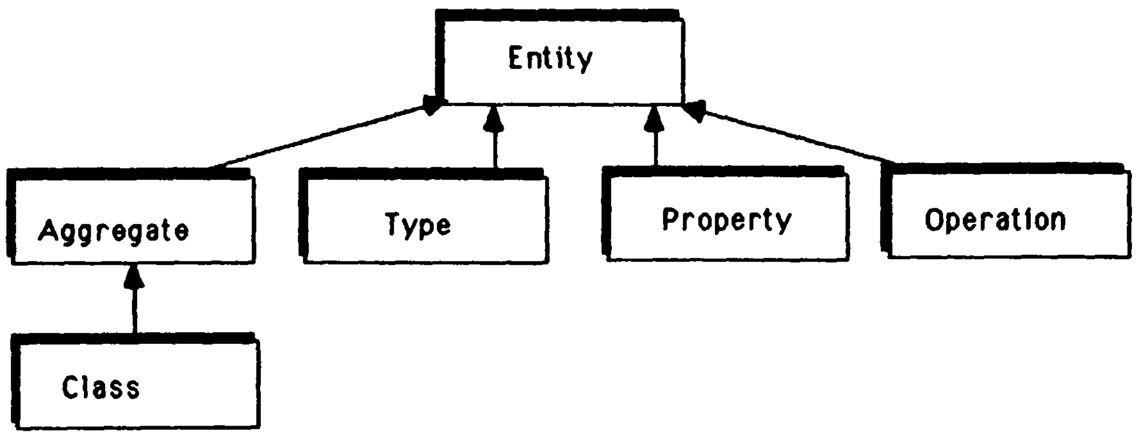


Fig. 1. The basic type hierarchy.

3. VERSION CONTROL

Any design environment is characterized by the evolution of objects over time. A programming environment is no exception. Programs change throughout all phases of the software life-cycle. A version control system is very important for a software library. We have a version control mechanism built into our database management system at the level of the model. Applications builders can describe to the system the behavior that they desire for new versions of an object.

A new object type T can be specified to be a subtype of the type *History-Bearing-Entity*. T then inherits the semantics of our version mechanism. *History-Bearing-Entity* defines the operations and properties needed for version control. All history-bearing-objects participate in version sets. That is, when a change is made to an object, it causes a new version to be added as the latest version of its version set. These changes are installed when the transaction that created them commits. Old versions cannot be changed; they can only have new versions inserted into their associated version set. Version sets are partially ordered. If one adds a new version v2 to an object x that already has a new version v1, then v1 and v2 are said to be *alternatives*. Inserting an alternative into a version set has the effect of starting a new branch in the partial order.

As we stated earlier in this section, a type may be defined to be a subtype of several other types including the type *History-Bearing-Entity*. This new type inherits the properties *previous-version*, *succeeding-version*, and *member-of-version-set*. The first two of these properties express the partial order of version sequence, and the third relates each of the individual versions to its containing version set.

Thus version control need not be an inherent built-in property of all entities but merely a set of programmer-defined attributes transmitted to subtypes through the inheritance mechanism. Moreover several different forms of version control can easily coexist in the database, although we provide one such mechanism in the base-level system.

4. TRANSACTIONS

There are several things that distinguish transactions in this context from that of the more traditional notion of a transaction. Normally, a transaction is an atomic, recoverable piece of work. All integrity constraints are true before and after a transaction executes.

We are looking into ways to relax this requirement on integrity. A more appropriate notion for a design environment is one in which there is a set of constraints that are guaranteed to be true before a transaction executes (a pre-condition) and a set of constraints that must be true after a transaction executes (a post-condition). The pre-condition need not be the same as the post-condition. For example, when one writes a paper, he often writes down the section headings first to act like an outline. These sections are then slowly filled in as the paper begins to materialize. The constraint which says that all sections must have bodies should not be checked all of the time. Instead, it should only be checked, when the paper is sent out for review. In this case, we would like the additional constraint to join the post-condition of the transaction that sets the *status* attribute to *reviewable*. We call this behavior *partial consistency*.

In order to deal with partial consistency, a powerful language is needed to express integrity constraints and when they should be enforced. A type definition carries with it a set of constraints that pertain to objects of that type. The applicability of each constraint is governed by a predicate that is based on the current state of the object and its environment. We are designing a language for describing these relationships.

5. PROGRAMMING ENVIRONMENTS

We are interested in applying our database ideas in real design environments. A prime example is that of a programming environment. Here, our view is that the database contains all objects that are used during the software life-cycle. It provides the glue that ties together all of the disparate tools and objects that are created or used during system development.

We are involved in two projects to test these ideas. First, we are using ENCORE as a platform for the construction of the GARDEN system, a descendent of the visual programming environment, PECAN [REIS84]. GARDEN provides users with multiple views of a program. All program pieces such as variables, expressions, and statements are objects that will be managed by the ENCORE OMS.

The second project involves the design of a general approach toward software environment construction [ZDON85]. This approach is based on a *databased programming language*, a type of language that integrates object-oriented database facilities with an object-oriented programming language.

In integrating databases and programming languages into a single programming system we may start from either a programming language or a database perspective.

1. Programming language perspective: Programming languages subsume databases.
   Augment conventional programming languages with persistent data structures and database operators [ATKI84]. Programs may access and manipulate databases as external resources, but are not themselves a part of the database.

2. Database perspective: Databases subsume programming languages.
   The database contains all objects including application programs, system programs, libraries, and environments. It provides general-purpose operations for specifying, creating, manipulating, transforming, and retrieving objects.

These two perspectives may be combined by including database facilities in the programming language subsumed by the database. In this case program modules in the database may be manipulated as database entities and may in turn create and manipulate programmer-defined database objects. If the database language supported by the programming language is the same as that in which programs are subsumed, then the programming language also subsumes the database and the two approaches are simultaneously realized. We shall refer to a system that realizes this symbiotic relation between databases and programming languages as a "databased" programming language. A databased programming language is essentially a language with a database facility sufficiently powerful to describe its own environment.

Databased programming languages are similar in concept to programming languages that can simply specify their own interpreters, such as LISP. However, databased languages are more ambitious in aiming to describe not just their interpreters but the complete multi-language program development environment in which they are embedded. They need not have only primitives for interpreting program structures, such as list processing primitives, but also primitives for describing, storing, and retrieving persistent database entities.

6. USER INTERFACES

As a part of this overall effort, we are also investigating the use of graphical interfaces to a database of the kind that we have been describing. We feel that generalized facilities for visually accessing an OMS will enhance the use of these systems. This also adds another dimension to the overall "visual programming" effort at Brown. Several other projects [CATT80, FOGG84, HERO80, KING84, MCDO75] have looked at the visual dimension to database access, but not in the context of an object-based model or in a system that integrates several database functions.

We have built a prototype system that provides visual access to a database that is in the same spirit as ENCORE. It does not as yet encompass all of the mechanism that we provide, but it is an important first step in this direction.

The system is called ISIS [GOLD85] and runs on an Apollo Domain 300 workstation. It allows users to define and extend schemas, to browse the database at both the type and the instance level, and to formulate queries using only a high-resolution graphics display and a mouse.

We are currently extending this facility to include a graphical way of dealing with version histories. This system has a notion of an interactive transaction, the boundaries of which the user can define. This kind of transaction would be useful during an editing session with the database. The user periodically presses the *End Transaction* key which has the effect of committing all changes that were made since the last *End Transaction*. This amounts to adding visible new versions to all of the respective version sets.

It will then be possible to view objects (as well as types) in the database and see graphical representations of the fact that versions of these objects exist. One can point to a version V in the history, in which case V is extracted from the sequence and the state of all objects in the database is rolled back to a state that is consistent with V.

7. CONCLUSIONS

We have briefly described our approach to building software engineering environments. It is based on an object-oriented database system that has been tightly coupled with a programming language. The database provides the common linguistic base that ties together all of the many objects that occur as a part of a large programming effort. In order to apply database techniques to this class of problems, we will need to make some progress in developing the next generation of data manager. We call systems that take an object-oriented approach to this problem object management systems.

We see many challenges for future research within this area. For example, we are exploring issues related to dynamic references and naming schemes for long-lived objects, specification methods for describing object lifetimes, the packaging of system facilities (e.g., version control, persistence) such that they can be easily inherited, a query language for an object-oriented database, and better ways to manage change to an evolving type-hierarchy.

REFERENCES

[ATKI84] Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J., and Morrison, R., "Progress with persistent programming", in *Databases—Role and Structure*, Stoker, P.M., Gray, P.M.D., and Atkinson, M.P., eds., Cambridge Univ. Press, Cambridge, UK, 1984.

[CATT80] Cattell, R.G.G., "An entity-based database user interface", *Proc. ACM SIGMOD Conf.*, 1980, pp. 144-150.

[CHEN76] Chen, P.P.S., "The entity-relationship model: towards a unified view of data", *ACM Trans. on Database Systems* **1**(1), 1976, pp. 9-36.

[FOGG84] Fogg, D., "Lessons from 'living in a database' graphical query interface", *Proc. ACM SIGMOD Conf.*, 1984, pp. 100-106.

[GOLD85] Goldman, K., Goldman, S., Kanellakis, P., and Zdonik, S., "ISIS: interface for a semantic information system", *Proc. ACM SIGMOD Conf.*, 1985, pp. 328-342.

[GOLD83] Goldberg, A. and Robson, D, *Smalltalk-80: The Language and Its Implementation,* Addison-Wesley, Reading, MA, 1983.

[HERO80] Herot, C.F., "Spatial management of data", *ACM Trans. on Database Systems* **5**(4), 1980, pp. 493-514.

[HAMM81] Hammer, M. and McLeod, D., "Database description with SDM: a semantic database model", *ACM Trans. on Database Systems* **6**(3), 1981, pp. 351-387.

[KING84] King, R., "Sembase: a semantic DBMS", *Proc. IEEE 1st Int. Workshop on Expert Database Systems*, 1984.

[LISK77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction mechanisms in CLU", *Comm. ACM* **20**(8), 1977, pp. 564-576.

[MYLO80] Mylopoulos, J., Bernstein, P.A., and Wong, H.K.T., "A language facility for designing database-intensive applications", *ACM Trans. on Database Systems* **5**(2), 1980, pp. 185-207.

[MCDO75] McDonald, N. and Stonebraker, M.R., "CUPID—the friendly query language", *Proc. ACM Pacific Conf.*, 1975, pp. 127-131.

[REIS84] Reiss, S., *Graphical Program Development with PECAN Program Development System,* Tech. Rep. CS-84-04, Dept. of Comp. Sc., Brown Univ., Providence, RI, 1984.

[SMIT77] Smith, J.M. and Smith, D.C.P., "Database abstractions: aggregation", *Comm. ACM* **20**(6), 1977, pp. 405-413..

[TSIC82] Tsichritzis, D.C. and Lochovsky, F.H., *Data Models,* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[ZDON84] Zdonik, S.B., "Object mangement system concepts", *Proc. ACM SIGOA Conf. on Office Information Systems*, 1984, pp. 13-19.

[ZDON85] Zdonik, S.B. and Wegner, P., *A Database Approach to Languages, Libraries, and Environments,* Tech. Rep. CS-85-10, Dept. of Comp. Sc., Brown Univ., Providence, RI, 1985.

OPAL: An Object-Based System for Application Development

Matts Ahlsen
Anders Bjornerstedt
Christer Hulten

University of Stockholm

ABSTRACT

Design and implementation of information systems in which requirements on flexibility, system evolution and high-level task support are emphasized, require suitable design and run-time support. Office systems are one class of systems where this need is apparent today. This paper gives a brief overview of OPAL, a combined run time and application development system, currently being designed at the University of Stockholm. The paper describes the environment which OPAL is intended for, the overall architecture and the basic system concepts including the Object-based model. This is followed by a short discussion of how this model can be used for application development. The paper ends with a summary and status of the OPAL project.

1. INTRODUCTION

1.1. Objectives for OPAL

OPAL is a combined run time and application development system[1]. Its purpose is to provide support for information management needs in organizations. It is aimed at supporting administrative tasks, which may be either the unstructured or loosely structured kind, common in creative work, or the more structured and routine kind, commonly associated with clerical work. On a very high level, the functions of OPAL are to provide communication, storage, retrieval and generation of information for office workers. These functions are provided in a way which gives:

- Flexibility in the evolution of office tasks.
- Adaptation to local requirements.
- A uniform user interface.
- Autonomy to different units of the organization.

This work is supported by the National Swedish Board for Technical Development (STU).
Authors' address: SYSLAB, Department of Information Processing & Computer Science, University of Stockholm, Stockholm, Sweden.
USENET: decvax!mcvax!enea!suadb!ch.
1. The OPAL system described here should not be confused with another, very similar system, which unfortunately has been given the same name [COPE84].

- Security of information.
- Reliability and robustness.

OPAL is an Object-based system, meaning that it builds on some central concepts common to the class of Object-based/oriented systems [AHLS84, GOLD83, ZDON84, NIER85, BYRD82, BIRT73, COPE84]. The development aspect includes a programming environment with language concepts for object management. The programming language is used both for defining different kinds of object *types*, and as an interactive command language [SNOD83]. This allows for programming both by end-users for personal customization, and designers for more "hard wired" or basic tasks. The "objects" in OPAL may represent programs (sequences of instructions), data or both. The target applications generated with OPAL are entirely built of object structures stored in a database. A user interacts with OPAL through a customizable interface. The user interface will not be further elaborated in this paper.

1.2. The Intended Environment

We see OPAL as a general development tool well suited for, but not specifically geared at "Office Systems". An OPAL application is typically interactive, multi-user, and decentralized. Applications basically consist of a set of activities (operations) to be performed, together with some constraints on the order in which they are performed, (c.f. Information Control Nets [ELLI80]). There is also a need for information sharing and gradual development of applications.

The tasks to be supported range from well-structured applications which may be fully automated, to more loosely structured applications which are difficult, impossible, or otherwise unsuitable to (fully) automate. The latter can be characterized by a high degree of interaction, meaning that the initiative or control over decision points in the application lies with the user. In many situations the system can know what *can* be done, even if it has not been designed to decide what *should* be done. OPAL provides for the automation of routine tasks, and as far as possible, assists the user in handling the less structured tasks [BROV85, CROF84]. The mechanisms for handling exceptions have been designed to cater to the unforeseen events typical of office routines.

For the more loosely structured tasks the information needs of users cannot be predicted. For this reason facilities for "ad hoc" operations (querying, browsing) are provided.

Tasks will need to be activated both on the basis of explicit demands from objects, and on the basis of expected conditions on objects. For the later kind of invocation OPAL has a trigger mechanism [ZLOO82, NIER85].

Another characteristic of this environment is that it often requires objects with a complex (data) structure, built out of sophisticated data types. The Object model of OPAL includes a variety of data types and facilities for composing "nested" data structures.

Most applications will typically involve several users operating on a common set of data. An activity, or transaction, could extend over a longer period of time than what is usually the case in a database system. Each step in the transaction may involve different users, and users may be assigned to steps dynamically. Due to the possibly long duration of transactions, the interactive nature of the processing done, and the number of users involved, restarts from scratch after failures may be unacceptable. This problem is also found in engineering design applications [KIM84]. OPAL transaction management works in terms of *nested transactions* [MOSS81, REED78]. This makes it possible to backout a subtransaction without aborting the whole transaction. It also has advantages in making applications more modular in the aspect of concurrency. A new application can incorporate an old one as a subtransaction without the need of redesigning the old transaction.

OPAL as a system will usually be decentralized in the sense of control and distributed in the physical sense. By decentralized we mean that the execution of an application (task) may be controlled by several, more or less autonomous, authorities. To handle concurrency and reliability and in general to keep applications consistent over both hardware and administrative (control) boundaries, we have adopted the architecture of ***atomic objects*** [OKI83, WEIH85a, WEIH85b]. All objects in OPAL are instances of one or more types. These types define operations (or functions) which may be invoked in the context of the object. The ***only*** way of manipulating an object (i.e., its data structure) is through the use of one of the available operations defined for an object. Thus the objects in OPAL are a form of ***abstract data types*** [LISK77]. Furthermore, the invocation of an operation may correspond to a subtransaction. A transaction is an operation which is serializable, with respect to concurrent operations, and recoverable. It either completes (once), or fails without altering the state of the objects involved.

In many applications access to past states of the system is useful. For this and other reasons OPAL uses ***version management*** to keep track of the evolution of objects. The possibility to access previous versions of objects adds a "temporal dimension" to the system which may be used by both the system and applications. Changes to existing applications and additions of new ones will be a frequent requirement in office systems. Because object types are maintained as objects and thus subject to version management, there is a means of reducing the problem of keeping the applications consistent in the face of changes on the type level. This is of particular relevance in a decentralized (or distributed) system (explained further below).

2. THE SYSTEM ARCHITECTURE OF OPAL

We can characterize OPAL as a decentralized system in terms of a network, where each node corresponds to an autonomous authority in an organization, and edges correspond to communication paths between nodes (see Figure 1). The function of a node is to provide an information processing environment for an organizational unit requiring authority over processing and data. It is of course application dependent to decide what should be regarded as an "autonomous" unit in an organization. If a unit was completely autonomous, then it would not be part of the organization. On the other hand one can usually identify units which have quite different means of control or different information needs. If an organization can be divided into relatively autonomous units, it can be provided with an OPAL system which is similarly divided with respect to control.

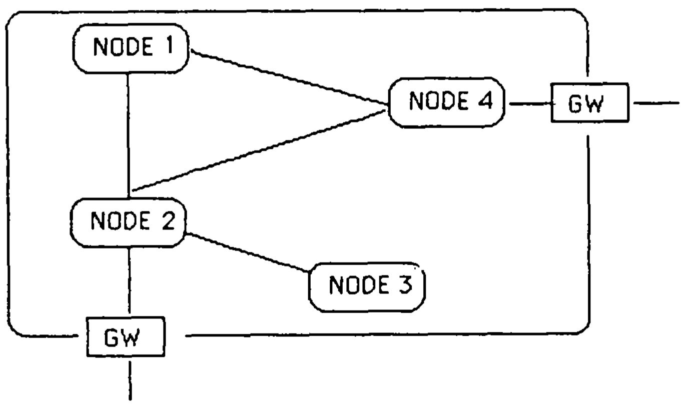


Fig. 1. An OPAL system with four nodes connected by logical communication paths. A node may have a gateway (GW) for external communication.

Authentication of users and protection of data is handled within a node. A node (and the "authority" behind it) is responsible for the data and operation within the node, but not outside it. >From the point of view of protection and authority, an OPAL node regards other nodes as practically "different systems". What unites a set of OPAL nodes into a system is the adoption of:

- a common identification scheme for all objects.
- a common representation form for all objects.

- a communications protocol between nodes.
- application dependent knowledge the nodes have of each other.

Communication between nodes should then not be any problem for application designers. Nodes which are to cooperate in some applications must have a certain degree of "trust" of each other. The important point here is that this "trust" is not built into the OPAL system. Rather, the degrees of freedom in the communication of a node with other nodes is specified in that node, and may be altered by something like "negotiations" between the nodes [HEWI84].

A node does not necessarily correspond to a physical machine ***host***. Nodes are distinct in terms of control and not necessarily in terms of physical location. Several nodes may be implemented on the same host.

The OPAL node interfaces with the authority it belongs to in terms of a set of ***users***. To affect the operation of a node, a user must ***logon*** to the node. On each node there is a supervisor process called the ***monitor*** which administers user authentication and functions as a clearinghouse [OPPE83] in the communication with the environment of the node. The environment consists of ***communication paths*** to other nodes and a number of ***devices*** through which users communicate with the node (see Figure 1). For a user to be able to logon to a node, there must exist an associated ***client*** object at that node. A client is an entity internal to an OPAL node, which mediates between the OPAL node and the user. A user can be represented by clients at more than one node. A node may in fact be aware of this (e.g., for the purpose of routing mail). However, authorization in a node is tied to clients and not to users. A client object is permanently tied to one node (i.e., it cannot be moved between nodes). Different nodes may recognize a user as such and the fact that the user corresponds to certain clients on other nodes.

An OPAL ***node*** is a centralized environment. The monitor is the overall controlling entity. Every node also has a component which we call the ***information base*** of the node. The function of the information base is to provide reliable storage and retrieval of objects. The information base provides what is called a ***single level store*** for applications executing on a node. The meaning of this is that there is no explicit distinction between different levels of memory. There is only a single address space for objects. The information base provides for the mapping between different levels of storage and caching of accessed objects.

In the following we discuss OPAL in terms of a single node.

3. THE OBJECT MODEL

OPAL can be characterized as an object-based system. This means that we built the system on a model which includes a few, fundamental concepts which are made explicit in the system. The most important of these are [AHLS84]:

- Modularity
- Encapsulation
- Instantiation
- Property Inheritance

Modularity refers to the ability of keeping logically related things together; an important structuring criterion. Encapsulation, or information hiding, is a means for limiting the visibility of details in modules (i.e., details not relevant for their abstract properties) which enables safe implementations. Instantiation allows us to discriminate between types and instances of objects. Property Inheritance is a convenient way of structuring type descriptions so as to allow specialization. We do not claim that these concepts alone "define" an object-based system, but we see them as fundamental.

3.1. Packets, Types and Instances

In OPAL, any "object" which may have an independent existence is represented by a *Packet*.

We distinguish between packet types and packet instances. Every packet is an instance of one or more packet types. The packet type according to which an instance is created is called the *base type* of the packet. A packet type defines the structure and behavior of a packet in terms of a data structure and operations defined on this data structure. The data structure of a packet represents the internal state of the packet, and is implemented by a set of attribute values. Each attribute is declared on some *data type*. A data type is similar to a packet type in that it defines a data structure and a set of operations. However a data type is not a packet type, it is a "second class" object type. The difference is that instances of a packet type are packets, while the instances of a data type (data type values) are not packets. Among other things, packets are the units of protection, locking and recovery in OPAL. Data type values are used for representing the state of packets and all packet states are disjoint. Thus, a data type value cannot be a part of more than one packet. However packets can *reference* other packets by using values of the *packet reference* data type. A packet may thus be shared by several other packets by reference, and the reference data type provides the "bridge" between packet types and data types. The reference data type is a parameterized data type because packet references are always qualified to some packet type (i.e., strong typing is used in packet references).

The state of a packet can only be changed by a packet operation. Thus, a packet has a strict operational (functional) interface. In packet operations, parameters and results (data type values) are passed by *value* to ensure encapsulation of attributes. However, local operations in the implementation of a packet type may pass data type values by *reference*.

An operation on a packet is invoked from another packet by a call using a reference to the packet in question.

Packet types are maintained as packets in OPAL (i.e., they are also packet instances of some (meta) packet type). Data types (note: definitions, not data type values!) may also be maintained as packets. This makes it possible to share and reuse old data types when defining new packet- or data- types.

A (packet- or data-) type definition consists of a *specification* and an *implementation*. The specification part *declares* public and private operations. Public operations are operations available for external invocation. Private operations are only available for use in the implementation. The implementation *defines* all operations and data structures of the type.

3.2. Property Inheritance

The set of instances that belong to a packet type is called a class. A packet may belong to several classes. In fact this is usually the case since a packet is not only an instance of its base type, but also an instance of *supertypes* of the base type.

The Property Inheritance mechanism allows a type to inherit the specification of another type (see Figure 2). The new type will be a specialization (a subtype) of the inherited one (the supertype) since it will introduce additional operations and data structures. Both public and private operations declared in the specification of a supertype are available for use in the implementation of a subtype. The public operations of a supertype also logically become public operations in the subtype. Matching of operation names is performed bottom up in the inheritance chain, and left to right in the inheritance prefix order (depth first). If a subtype introduces an operation which overrides (by name) an operation in one of its supertypes this just means that instances of the subtype will have alternative operations with the same name. Which operation will be used depends on the type qualification of the reference used. There is no selectivity in the inheritance of operations, the complete specification of a type is always inherited. A type may have several supertypes as well as several subtypes.

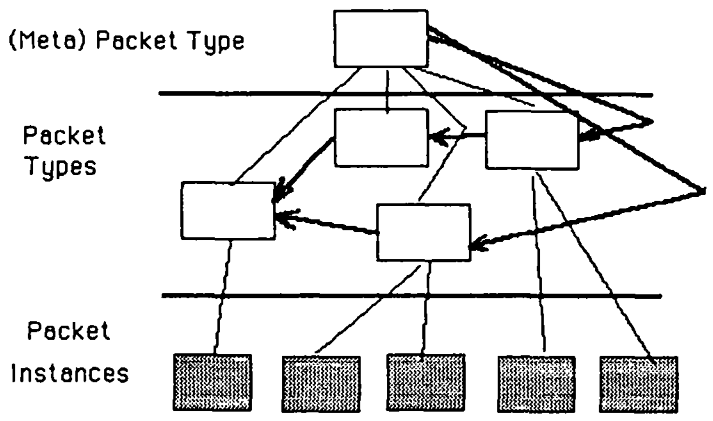


Fig. 2. Instantiation (thin lines) and Property Inheritance (bold lines).

3.3. Active Packets

Thus far we have only considered packets as "passive" objects with operations invoked from other packets in a procedure call-oriented way. A packet type may contain, in its implementation a sequence of instructions called a ***statement body***. Instances of the type can be ***executed*** according to this sequence. A packet instance will always be in a certain state of execution (i.e., unexecuted, executing, suspended, or terminated). Packets may be executed by other packets, or by the monitor of the node as a result of trigger conditions. The execution of a packet generates a new process. A data type does not have a statement body and hence data type values may not be executed.

A packets' statement body may contain an "interact" statement, which when executed will set the packet in interactive mode. This will give some user access to the packet *scratchpad*, which is a special attribute present in every packet instance. The purpose of the scratchpad is to provide an interactive "work space" in which a user can dynamically create data structures[1] and execute operations.

3.4. Packet references

All packets are designated a packet identifier (Pid) when created, which is unique over all nodes in an OPAL system. A reference to a packet (an instance of the reference data type) is basically implemented by a pair of pids (type-pid, instance-pid) containing the pids of a packet type and some instance of this type which is to be accessed. Protection of packets is thus achieved by a capability approach[2] [LEVY84, JONE76]. The type-pid of a reference to a packet need not be the pid of the base type. The type-pid could be the pid of any supertype of the base type or the pid of a *view* packet. A view packet is similar to a type packet except that it does not define any attributes, only operations. A view packet is associated with another type or view packet, called its domain type. While a sub-type adds capabilities (operations) to the ones inherited from the supertype, a view can only alter the capabilities with respect to its domain type, and in a technical sense only weaken them. To implement a new set of public operations, the implementation section of a view may only use the public operations of the domain type to operate on instances[3]. Thus, anything accomplished with an instance by using a reference with a type-pid that of a view, could always be accomplished with a reference to the same instance, but with a type-pid that of the domain type of the view. Even if the operations defined in a view could perform more elaborate computations than the operations of the domain type, they are still never more "powerful". A view packet may also include a class subset condition, which may reduce the set of instances that can be

1. These data structures are only accessible in the scratchpad and have a different status from the attributes implementing a packet. Thus, there is no violation of the normal operation interface to packets.
2. Higher level authorization is achieved in OPAL by adding a kind of authorization list mechanism to the low level capability oriented access control.
3. Property inheritance between views is possible but outside the scope of this paper.

operated upon from the view, as compared with the class of the domain type.

3.5. Aggregates of Packets

Packet references are used to build aggregates of packets. In fact the whole information base of a node is one large and complex aggregate of packets. OPAL has several data types, based on the reference data type, for managing packet references. An example is the *folio* data type, which is an ordered collection of (optionally) named references. Folio's are used to implement various kinds of *naming contexts*, much like file directories in traditional operating systems, but more general.

3.6. Packet Versions and Nested Transactions

Every packet operation which alters the state of a packet (i.e., is not a read only operation) generates a new *version* of the packet. Furthermore, an operation may label the version it generates as being of any of a number of application defined *version categories*. Associated with every version category is a specification of the number (of the latest), or the maximum age, of versions which are to be retained.

A version generating operation is a subtransaction in a nested transaction. When the operation is finished, the subtransaction commits and the version becomes visible to the supertransaction. This continues all the way up to the top transaction which is endless and associated with all executing packets in a node. If an operation does not label the version it generates, then the version will be discarded as soon as a new version of the packet is committed in a supertransaction. A time stamping technique is used to prevent interference between transactions [REED78].

Version management is used as the basis for concurrency and recovery [PAPA84], for higher level application needs [ZDON84, AHLS84], and to provide for flexible and smooth evolution of types [AHLS83]. For instance, read-only transactions on a packet can usually proceed without delay, even if there is currently a write transaction on the same packet, since the read operation is simply provided with the latest visible version of the packet in the enveloping transaction. It will always be possible to technically abort a subtransaction, since the previous version is kept at least until the subtransaction commits. The use of version categories makes it possible to easily keep track of different aspects of the evolution of a packet.

4. APPLICATION DEVELOPMENT

4.1. The Packet Language

The model concepts described in the previous section are manipulated using PAL (the PAcket Language) [AHLS85], which is a high-level programming and command language. The specification and implementation of the packets in an application are written in PAL by the designer. When a user executes a packet in the interactive mode, PAL statements can be used as a command language. The details of the language are however outside the scope of this paper.

4.2. Designing Packets

A designer works with OPAL using a number of design functions. As such, the designer is a "user" from OPAL's perspective. The packets which the designer operates on are in principle the various descriptive packets (packet type packets, view packets, data type definition packets, etc.), and the functions used are packet operations on these.

New packet types are constructed by specialization of existing types. In general, there will be a set of "system defined" packet types holding various generic definitions common to all packets or useful in a large class of them. As a simple example, to create a new packet type of which the instances should represent the staff of programmers employed in a company, the existing packet type Employee is specialized by inheritance to a new type Programmer (see Figure 3).

```
employee
PACKET TYPE programmer
    PUBLIC
     <public operations>
    PRIVATE
     <private operations>
    IMPLEMENTATION
     <data structures>
     <bodies of public and private ops>
     <hidden operations not subject to inheritance>
     <statement body>
END programmer
```

OPAL has "data dictionary"-like facilities which provide information about existing types and how these are related. The designer may create instances of a new packet type or of a new version of an old packet type in order to try out the design, before making it visible by a (top level) commit. If a packet is to be moved between nodes, then there must exist a *copy* or proper packet type versions at both nodes.

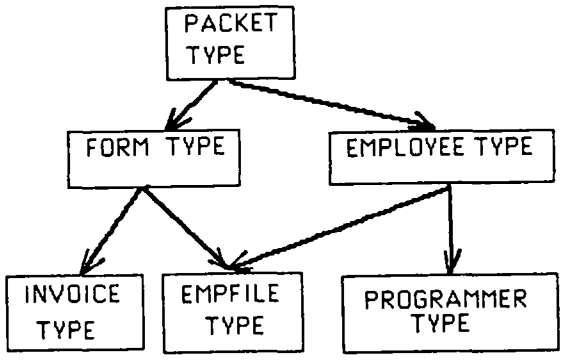


Fig. 3. Inheritance graph (DAG). The new packet type PROGRAMMER is a specialization of EMPLOYEE. The EMPLOYEE type multiply inherits the types FORM and EMPLOYEE.

4.3. Incremental Development

It is important to allow applications to undergo changes in a smooth way (i.e., we should provide support in the development system which assists the designer in modifying existing applications). The inheritance and view facilities provide this to a certain extent, since new types can be built on the basis of existing ones and alternative operational interfaces can be provided to the instances of existing types through Views. Some changes to applications can be managed in this way, but it may also be desirable to make changes to the structure of existing packet types and Views. This kind of change may cause problems since the packets in an application are kept in the database of a node for long periods of time, and other applications and users are dependent on them. If a packet type is updated, this would require restructuring of the instances, which may be difficult to perform due to the nature of the tasks in the system.

Versions of packet types and views may be categorized and retained to reflect the changes made. A version of a type packet may have associated mapping functions so as to enable instances of earlier versions to be processed according to the new version, and conversely, to allow instances of a new version to be processed according to an earlier one. If complete compatibility between versions is achieved, a type change can be made transparent to the dependent applications if desirable, but this may not always be possible due to the nature of the changes [AHLS83].

Applications may be modified by generating successive versions of the corresponding type packets, allowing processing to continue according to the old versions, while testing the new ones. Once the designer decides to "commit" a new version, it will become the current version. Any new instances will be created according to the current version. The old version is left intact.

5. SUMMARY AND STATUS

The aim of the OPAL system is to provide development and run-time support for administrative applications. The system is characterized by the use of concepts commonly adopted in object-based/object-oriented systems. The primary structuring facility in OPAL is the Packet, which is used to represent both "programs" and "data". Packet Types as well as Packet instances are maintained in a database residing on each node in an OPAL system. Packet types may be specialized using Property Inheritance, and Views may be defined on packet types so as to allow alternative interfaces to instances of types. Version management is used to handle updates to packet types and packet instances.

Project status: A first draft of the functional specification of OPAL has been completed, in which the basic requirements of the system are outlined. Detailed design of the major system components is currently in progress.

REFERENCES

[AHLS83] Ahlsen, M., Bjornerstedt, A., Britts, S., Hulten, C., and Soderlund, L., "Making type changes transparent", *Proc. of IEEE Workshop on Languages for Automation*, 1983. (Also available as Syslab Report No. 22.)

[AHLS84] Ahlsen, M., Bjornerstedt, A., Britts, S., Hulten, C., and Soderlund, L., "An architecture for object management in OIS", *ACM Trans. on Office Information Systems* **2**(3), 1984, pp. 173-196.

[AHLS85] Ahlsen, M., Bjornerstedt, A., Britts, S., Hulten, C., and Soderlund, L., *PAL Specification*, SYSLAB Working Paper 96, SYSLAB, Univ. of Stockholm, Stockholm, Sweden, 1985 (in preparation).

[BIRT73] Birtwistle, G.M., Dahl, O-J., Myhrhaug, B., and Nygaard, K., *Simula Begin*, Auerbach, Philadelphia, PA, 1973.

[BROV85] Broverman, C.A. and Croft, W.B., "A knowledge-based approach to data management for intelligent user interfaces", *Proc. 11th Int. Conf. on Very Large Data Bases*, 1985, pp. 96-104.

[BYRD82] Byrd, R.J., Smith, S.E., and de Jong, S.P., "An actor based programming system", *Proc. ACM SIGOA Conf. on Office Information Systems*, 1982, pp. 67-78.

[COPE84] Copeland, G. and Maier, D., "Making Smalltalk a database system", *Proc. ACM SIGMOD Conf.*, 1984, pp. 316-325.

[CROF84] Croft, W.B. and Lefkowitz, L.S., "Task support in an office system", *ACM Trans. on Office Information Systems* **2**(3), 1984, pp. 197-212.

[ELLI80] Ellis, C.A. and Nutt, G.J., "Office information systems and computer science", *ACM Computing Surveys* **12**(1), 1980, pp. 27-60.

[GOLD83] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[HEWI84] Hewitt, C. and de Jong, P., "Open systems", in *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, Brodie, M.L., Mylopoulos, J., and Schmidt, J.W., eds., Springer Verlag, Berlin, 1984, pp. 147-164.

[JONE76] Jones, A.K. and Liskov, B.H., "A language extension for controlling access to shared data", *IEEE Trans. on Software Engineering* **SE-2**(4), 1976, pp. 277-285.

[KIM84] Kim, W., Lorie, R., McNabb, D., and Plouffe, W., "A transaction mechanism for engineering design databases", *Proc. 10th Int. Conf. on Very Large Data Bases*, 1984, pp. 355-362.

[LEVY84] Levy, H.M., *Capability-Based Computer Systems*, Digital Press, 1984.

[LISK77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction mechanisms in CLU", *Comm. ACM* **20**(8), 1977, pp. 564-576.

[MOSS81] Moss, J.E., *Nested Transactions: An Approach to Reliable Distributed Computing*, Tech. Rep. MIT/LCS/TR-260, Lab. for Comp. Sc., MIT, Cambridge, MA, 1981.

[NIER85] Nierstrasz, O. and Tsichritzis, D., "An object-oriented environment for OIS applications", *Proc. 11th Int. Conf. on Very Large Data Bases*, 1985, pp. 335-345.

[OKI83] Oki, B.M., *Reliable Object Storage to Support Atomic Actions*, Tech. Rep. MIT/LCS/TR-308, Lab. for Comp. Sc., MIT, Cambridge, MA, 1983.

[OPPE83] Oppen, D.C. and Dalal, Y.K., "The clearinghouse: a decentralized agent for locating named objects in a distributed environment", *ACM Trans. on Office Information Systems* **1**(3) 1983, pp. 230-253.

[PAPA84] Papadimitriou, C.H. and Kanellakis, P.C., "On concurrency control by multiple versions", *ACM Trans. on Database Systems* **9**(1), 1984, pp. 89-99.

[REED78] Reed, D.P., *Naming and Synchronization in a Decentralized Computer System*, Tech. Rep. MIT/LCS/TR-205, Lab. for Comp. Sc., MIT, Cambridge, MA, 1978.

[SNOD83] Snodgrass, R., "An object-oriented command language", *IEEE Trans. on Software Engineering* **SE-9**(1), 1983, pp. 1-7.

[WEIH85a] Weihl, W. and Liskov, B., "Implementation of resilient, atomic data types", *ACM Trans. on Programming Languages and Systems* **7**(2), 1985, pp. 244-269.

[WEIH85b] Weihl, W., "Atomic data types", *IEEE Database Engineering* **8**(2), 1985, pp. 26-33.

[ZDON84] Zdonik, S.B., "Object management system concepts", *Proc. ACM SIGOA Conf. on Office Information Systems*, 1984, pp. 13-19.

[ZLOO82] Zloof, M.M., "Office-by-example: a business language that unifies data and word processing and electronic mail", *IBM Systems Journal* **21**(3), 1982, pp. 272-304.

An Object-Oriented Protocol for Managing Data

Stephen P. Weiser

University of Toronto

ABSTRACT

Many researchers believe that object-oriented languages are well suited for some of the programming tasks associated with the building of an office information system (OIS). To lend support to this thesis, we shall concentrate our attention on an object-oriented programming environment, named Oz, which has been effectively employed to capture certain aspects of OISs more simply and naturally than with conventional languages. After pointing out some of the limitations of Oz, we introduce additional facilities into it which further enhance its capabilities, especially with respect to the management of office data.

1. INTRODUCTION

One of the means of evaluating the utility of a programming language is to measure the effort associated with the programming of particular applications. It has been argued that by this standard, object-oriented languages are appropriate for the implementation of OISs [NIER85]. A straightforward way to defend such a proposition is to demonstrate that essential characteristics of OISs can be captured more readily by the *object protocol* of a given object-oriented language than by the constructs associated with conventional programming languages.

This was the impetus for developing *Oz*, a prototype object-oriented programming environment implemented at the University of Toronto [NIER83, MOON84, TWAI84]. While Oz bears comparison to general purpose systems such as Smalltalk, it is distinguished by features which reflect its intended use as a tool for building OISs. These features in turn reflect the designers view of what an OIS is. This requires some elaboration.

In the office place of today, an OIS has come to refer to an aggregation of software often including word processing, graphics, electronic mail, database management and spreadsheets. In the more sophisticated of these systems, such as Lotus 1-2-3 and Symphony, a certain level of integration is achieved by allowing data flow among the constituent programs.

Research in OIS is directed towards more than just the development of integrated software tools with increased functionality and ease of use. These tools assist the office worker in performing his tasks. However, they are passive in that they do not initiate or control the processing of office tasks [LOCH83,

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada under grants G1359 and G1360.

Author's address: Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada, M5S 1A4 (416/978-6610).
CSNET: weiser.toronto@csnet-relay.

WOO85]. To increase office productivity, an OIS should be able to capture, manage, and perform office activities [LOCH84].

Office activities have been described in the literature [HAMM80, MORG80, SIRB81] as being event-driven and semi-structured. They exhibit a high level of parallelism requiring synchronization and coordination. They alternate between active and suspended states which are distributed in time and space. They frequently involve the manipulation of highly structured documents which possess certain constraints and functional capabilities not generally associated with databases [NIER85]. The focus of attention in Oz is the automation of these office activities.

It has been shown elsewhere [NIER85] that Oz accomplishes what it set out to do. In this paper, we try to indicate some of what Oz *doesn't* do, or at least, doesn't do well. Our attention is focussed on the representation and handling of office data, which is achieved in a cursory manner in Oz. We present an enhanced implementation of Oz and illustrate its effectiveness.

2. OZ

For those not familiar with Oz, we offer a brief overview. Oz objects are entities composed of *contents* (data) and *behaviour* (program). The contents of an object are composed of an aggregate of *instance variables*. These variables have values of type *string, integer* or *pointer* (these are unique object instance *id* values). The behaviour of an object consists of a set of *rules*.

Oz object instances are organized into *classes*. The members of a class have the same behaviour but are distinguished by the values of their contents. Classes are organized as nodes in an m-ary tree structure, and inherit instance variable definitions and rules from parent nodes.

A class definition for *employee* objects could take the form:

```
employee : person{  /* class - employee, superclass - person */

        /* instance variables */
        emp-no : integer;  /* employee number */
        s-visor : supervisor;  /* pointer to an employee's supervisor object */
        status : string;  /* current status */
        .
        .
        /* rules */
        .
        .
    }
```

An *employee* object might inherit such instance variables as *name, birth-date, address, phone-no,...* from the *person* superclass as well as the rules governing the manipulation of these variables.

Oz objects communicate by passing messages which attempt to invoke rules. An Oz message specifies the *id* (all Oz objects have unique system generated *ids*) and class of the sender as well as the class, rule name, rule parameters, and (optionally) the *id* of the receiver. If this *id* is not specified, the message *finds its way* to an instance of the receiver's class that allows for the formation of an *event* (*events* are discussed shortly). An invoked rule may return a value to the sender.

Rules may be invoked by rules within the same object or within other objects. Rules consist of *conditions* and *actions*. The conditions must all be true before the actions of a given rule can be performed. Conditions can specify the acceptable classes of objects invoking the rule (these classes are referred to as the rule's *acquaintances*), the state of the object (the value set of its variables) containing the rule, and the state of other objects. Actions correspond to "program" components. Associated with each object class are two rules which have all the characteristics of other rules in addition to the following special functions. The *alpha* rule when invoked will cause an object instance to be created. The *omega* rule will cause an object instance to be destroyed. These rules are necessarily the first and last rules invoked in the lifetime of an Oz object.

The Oz code fragments below illustrate how the state of other objects is ascertained. The *get-super* rule finds an unspecified *available* member of the *supervisor* class. The *get-super-name* finds the name of the specified supervisor.

```
employee : person {
        emp-no : integer;
        s-visor : supervisor;
        .
        .
        /* get a supervisor rule */
        get-super(){
            /* only administrator can invoke rule */
            ~ : administrator;
            /* supervisor object temporary variable */
            s : supervisor;
            /* supervisor must be available */
            s.available = "yes";

            /* assign supervisor */
            s-visor := s;

        }()
        .
        .
        /* get a supervisor's name rule */
        get-super-name(emp-num){
            ~ : administrator;
            /* employee no. */
            emp-num : integer;
            /* temporary variable */
            name : string;
            /* looking for employee with */
            /* employee number emp-num */
            emp-no = emp-num;

            /* get name from supervisor */
            name := s-visor.give-name();

        /* return name */
        }(name)
        .
        .
```

```
supervisor : person{
        .
        .
        /* instance variable - availability */
        availability : string;
        .
        .
        /* availability rule */
        available(){
            /* only an employee can invoke rule */
            ~ : employee;
        /* return availability */
        }(availability)
        .
        .
        /* name rule */
        give-name(){
        }(name)
        .
        .
```

If no acquaintances are specified in the conditions of a rule, the rule will be invoked when its conditions become true. This gives Oz objects a kind of autonomy not found in other object-based systems [NIER85]. Another feature of Oz that is somewhat unique is the way in which it forms *events*. Even when the conditions of a rule are true, its *state changing* actions will not be performed unless all the conditions of its invoking acquaintance (if it has one) are true. This requirement is applied recursively to each acquaintance. As each rule may have many conditions, each of which may invoke rules in other objects, an m-ary tree of associated objects is formed (potentially). Only when the conditions in all these rules are true will all the state changing actions be performed simultaneously. This is the fundamental unit of change of state in the object universe (rather than the firing of individual rules). Thus Oz offers a powerful event-driven model of computation [NIER85].

3. ENHANCEMENTS TO OZ

The ability to model real world structures *naturally* is a hallmark of object-oriented systems [GIBB84]. *Naturally* in this context implies a simple mapping from user conception to object representation. Oz however, offers only a primitive method of representing office structures.

The contents of an Oz object resemble database relations. The correspondence of object class to relation, object contents to tuple, and attributes to instance variables is immediately apparent. Both the relational model and the Oz object model require that attributes and instance variables, respectively, have simple data values. It should be clear that the *encoding* problems associated with relational models are all present in Oz. These problems can be illustrated with an example.

Consider a university which must keep information on its students which includes the courses they have taken and the marks received. A student record can be represented as:

student(*stu-no*, stu-name, (course, grade),...,(course, grade))

A consistent first normal form (1NF) relational schema is:

student(*stu-no*, stu-name)
grades(*stu-no, course*, grade)

We note the following:

1. The loss of the "object" nature of the student record (its information content has been distributed into two relations).
2. The "flattening" of a set-valued field into multiple tuples.
3. The introduction of an attribute that is artificial in the sense that it doesn't reflect an attribute of the entity under consideration but only establishes tuple relationships (the *stu-no* in the grades relation).

Not only does this encoding require a *translation* effort by the programmer, but it also increases the operational complexity associated with record manipulation. Record creation and deletion are no longer associated with a single record but rather with two relations and multiple tuples. Queries and updates are similarly affected. There is an existence dependency relationship of *grades* on *student* (a set of grades must be associated with an existing student, though the converse is not true). The relational representation does not reflect this dependency, whereas it is intrinsic to the structure of a student record. In general, increased encoding requires an increase in integrity constraints [MAIE84].

With Oz, the analogous problems are more critical. Not only would the data associated with a student record be distributed in two object classes, but the operations associated with this data would be as well. It has been shown that this kind of distribution of operations leads to enormous increases in Oz programming effort [WEIS85].

In response to these considerations Oz has been modified in the following manner. Objects are allowed to aggregate not only any number of *simple types (string, integer, pointer)* but other objects as well, each of which in turn may do the same. Simple types and objects may have set occurrences. An Oz student object might now have the syntax:

```
student {
    stu-no   : int
    stu-name: string
    grades  {
        course   : string
        grade    : int
        }*
    parent-names: string*
     . . .
```

The * indicates a set occurrence. Repeating groups (such as *grades*), which occur commonly in office data, are directly representable as contained objects. In general, Oz now allows for the hierarchical representation of data within objects. This is significant in that a very common office structure—the electronic document—is hierarchical in nature.

For purposes of clarity, we shall refer to those objects contained within an object class contents definition as *contained* objects (i.e., *grades* is a contained object). The hierarchical structure of an object's contents may be thought of as a tree; the root corresponding to the object itself, the intermediate nodes to contained objects and set occurrences of simple type, and the leaf nodes to simple type variables. A set of operations must be provided that allow the manipulation of the data contained in this tree. The current version of Oz provides a primitive set of operations that allows for traversal of this tree along with node creation, deletion, and updates. Future versions of Oz will provide more sophisticated operations [WEIS85]. (These operations are not detailed here as they are the familiar ones associated with hierarchical databases.)

Contained objects may be defined in terms of *existing* object class definitions. The contained object thus defined inherits the contents structure *and rules* of the named object class. (The "existence restriction" on object classes removes the possibility of either direct or indirect recursions in object definitions.) Contained objects which inherit class definitions may not have set occurrences and may not be themselves contained within other contained objects. Without these restrictions, the interpretation of inherited rules becomes extremely complex [WEIS85]. Note that by this mechanism, we are providing Oz with multiple inheritance capabilities. Ambiguous rule names are resolved by choosing the first rule encountered in a breath-first search of the class inheritance network.

Text is introduced as a simple data type. This is a step in the direction of representing all common office data types (textual, graphical, audio, etc.) in a uniform manner within Oz objects and providing a set of operations to manipulate them.

While object containment offers a method of "building" object structures out of other objects, it is not suitable for modelling object relationships. *Relationship* here has the specialized meaning of one object being able to communicate *directly* with some other particular object. In Oz, this can only be accomplished by possession of that object's unique *id*. Pointer types hold such *ids* in Oz. In our enhanced version of Oz, pointer types can be sets. However, the restriction that all the *ids* of a set of pointers belong to objects of the same class is enforced. In this way we can partition classes of objects on various criteria. For example, suppose that we have a class of employee objects and a class of department objects. Pointer sets in the department objects would relate all the employees in each department to the appropriate department object. Thus a department object has direct communication privileges with its employee objects. In the original version of Oz, such relationship were not possible. Operations involving the employees of a given department would involve a search of all employee class objects to find the desired ones. This would represent a substantial processing time overhead when the number of objects in the class was great. In addition, if the relationship between departments and employees was other than 1:N (i.e., if employees could be in more than one department), a new *class* would be needed whose purpose would only be to establish the N:M department to employee relationship. The enhanced version of Oz eliminates the need for such artificial constructs.

Methods are being investigated for enforcing 1:1 and 1:N relationships between object classes in Oz, though these have not yet been implemented.

A more sophisticated notion of object state has been introduced into Oz. Objects exist in either a *passive* or *active* state. A passive object is one that has been stripped of its rules and whose data contents have been stored as a contained object in a special *database object* associated with each class. Passive objects are not considered in events (as they have no rules to invoke). Active objects have both contents and rules. An old office memo kept in a file and a currently circulating memo correspond to passive and active objects respectively.

In a very large object universe, it is likely that only a small percentage of objects need be active at any give time, the rest residing as passive objects in their database object containers. Thus database objects may hold vast numbers of objects associated with a class. A set of passive objects may correspond to different versions of the same *conceptual* object, such as a form at various times in its history. Such a set of passive objects are distinguished from all other objects by possession of the same object *id*. The objects of this set are distinguished from one another by a *time-stamp* (*ids* and *time-stamps* are provided for all passive and active objects by the system). Database objects in Oz have been implemented in such a way as to provide a rich set of querying capabilities on their contents. The contents structure of an Oz object is represented by a set of relational tuples generated by an algorithm similar to the one found in [GIBB84]. A standard relational DBMS can then be used to manage these tuples. Database object rules can be "built" rather easily in terms of the relational operators associated with the DBMS.

By replacing each of the simple type values (*integer, string, text* and *pointer*) in an object's contents by a vector, a set of time-stamped versions of a particular conceptual object can be represented with a great saving of space. Each element of the vector is an ordered pair consisting of a data value and the time of its last update. The elements of the vector are ordered by increasing time. This is the method in which version sets are implemented in Oz, although this fact is transparent at the object level; database objects "see themselves" as containing only distinct passive objects. Note that the underlying relational DBMS makes it particularly easy to implement these vectors (they correspond to sets of 2-tuples).

A passive object can be created from an active object by invoking the *omega-db* rule (which replaces the omega rule in the original version of Oz) associated with an object class. Invocation of this parameterized rule may result in one of the following:

1. The storage of the contents of the active object as a time-stamped passive object followed by the destruction of the active object. In addition to their own object *ids*, all active objects carry the *id* of the passive object from which they were created (unless, as explained later, they were not created from a passive object). Thus active objects are returned to their version sets.
2. The storage of the contents of the active object as a time-stamped passive object without the destruction of the active object (version retirement).
3. The destruction of the active object without storage as a passive object (object contents will not be needed at a future time).

The *alpha-db* rule creates an active object from a passive one by providing the converse capabilities of the omega-db rule. These are:

1. The creation af an active object using the contents of a specified passive object which is then destroyed. Specification is provided by passing a passive object *id* to the alpha-db rule. By default, the *newest* member of a version set is used. A selection query on the database object would be the likely method of obtaining a particular *id*. For example, an *administrator* might select a contained object in the *student* database object with a particular student number and then invoke the *student* class alpha-db rule with the selected *id*.
2. The creation of an active object from a member of a set of time-stamped passive object versions. The *id* as well as the *time-stamp* which specify the passive object would likely be obtained by selection of a passive object based on a *time-sensitive* query. Possible time related selection criteria include *oldest* and *newest* members of a version set as well as *closest* to a given time.
3. The creation of an active object whose contents are not obtained from a passive object. The objects contents would be initialized by the alpha-db rule itself (i.e., the actions of the rule would include

instance variable initialization).

Objects created by (3) are newly "born" as opposed to objects in (1) and (2) which are "reincarnations" [TSIC85]. Objects may also "pop" into existence in a passive state. These objects would be created by subverting normal object protocol. One might wish to initialize an object universe by loading database objects with passive objects, as opposed to starting a system up with an empty object universe. Many examples can be found where this is the appropriate method of doing things, though care must be taken to assure that the active counterparts of these objects will not produce inconsistent or fatal system states [WEIS85].

Active object management involves the storage and retrieval of active objects, as events must be found and executed. Since objects of the same class share the same behaviour, it is only necessary to store that behaviour once [NIER85]. As objects in a class are distinguished by their contents, the contents of each object instance must be stored.

The behaviour of a class will usually include inherited rules. As these rules already exist, they can be referenced rather than copied in the class that inherits them. This elimination of "code" redundancy can result in substantial space savings because of the multiple inheritance capabilities of Oz.

At any point in time, the set of all active objects can be partitioned on the basis of current participation in the formation of an event. While those objects participating in event formation must be in primary memory, those not participating may conveniently reside in secondary memory. This is of interest, as there will always be some bound on the number of active objects that can exist in primary memory (we are assuming that primary memory is large enough to hold the objects involved in the formation of a given event and that secondary memory is sufficiently large to hold the entire object universe). In the original implementation of Oz, this issue is masked by the reliance on the virtual memory support of an underlying operating system (UNIX[1]). There are many reasons why Oz should provide its own virtual memory support [TWAI84, NIER85, WEIS85]. Towards this end, the current version of Oz implements the following active object memory management policy.

A copy of the contents of each active object resides in secondary memory. The location of a particular object's contents can be generated by a table lookup based on the object's unique *id*. When it is determined that an object is needed for event formation, its contents are copied into primary memory, unless a copy of its contents already exist there. If an event occurrence induces changes in the state of this primary memory copy, the copy in secondary memory is updated to reflect these changes. The primary memory copy is not deleted until space is needed to bring in other objects for other events (this saves recopying the object in from secondary memory if it participates in an event in the near future). In this manner secondary memory remains coherent and as up-to-date as possible [NIER85]. (Even if primary memory is wiped out by a system crash, a consistent object universe state remains in secondary memory.) Furthermore, primary memory is well utilized, and the amount of object content copying between primary and secondary memory is reduced.

4. CONCLUSIONS

We have demonstrated how complex data structures can be represented and manipulated within objects. This is a significant step in the direction of making Oz an effective programming tool.

By allowing objects to be moved back and forth between passive and active states, we allow the user to assist the object manager in partitioning the object universe on the basis of object activity. This is an important consideration since any practical system will have bounds on primary storage space and event processing time. The object manager can now consider objects on the basis of their "activity level" in forming events, whereas previously it could not differentiate objects on this basis and was required to consider them all equivalently.

In addition to this, querying on the passive object contents of the object world equivalent of databases

1. UNIX is a trademark of Bell Labs.

can be performed quite effectively using analogs of the relational calculus [WEIS85].

Other areas of current research on Oz include improvements in the efficiency of the tasks performed by the object manager: event management, and object storage and retrieval. Design criteria for a sophisticated user interface for Oz are also being developed.

REFERENCES

[GIBB84] Gibbs, S.J., *An Object-Oriented Office Data Model*, Ph.D. Thesis, Dept. of Comp. Sc., Univ. of Toronto, 1984.

[HAMM80] Hammer, M. and Sirbu, M., "What is office automation?", *Proc. 1980 Office Automation Conf.*, 1980, pp. 37-49.

[LOCH83] Lochovsky, F.H., "A knowledge-based approach to supporting office work", *IEEE Database Engineering* **16**(3), 1983, pp. 43-51.

[LOCH84] Lochovsky, F.H., Tsichritzis, D.C., Mendelzon, A.O., and Christodoulakis, S., "An office procedure manager", Working paper, Comp. Systems Res. Inst., Univ. of Toronto, Toronto, Canada, 1984.

[MAIE84] Maier, D. and Price, D., "Data model requirements for engineering applications," *Proc. IEEE 1st Int. Workshop on Expert Database Systems*, 1984, pp. 759-765.

[MOON84] Mooney, J., *Oz: An Object-based System for Implementing Office Information Systems*, M.Sc. Thesis, Dept. of Comp. Sc., Univ. of Toronto, Toronto, Canada, 1984.

[MORG80] Morgan, H.L., "Research and practice in office automation", *Proc. IFIP Congr., Inf. Processing '80*, 1980, pp. 783-789.

[NIER85] Nierstrasz, O.M., "An object-oriented system", in *Office Automation: Concepts and Tools*, Tsichritzis, D.C., ed., Springer-Verlag, Berlin, 1985, pp. 167-190.

[NIER83] Nierstrasz, O.M., Mooney, J., and Twaites, K.J., "Using objects to implement office procedures", *Proc. CIPS Conf.*, 1983, pp. 65-73.

[SIRB81] Sirbu, M., Schoichet, J., Kunin, J., and Hammer, M., *OAM: An Office Analysis Methodology*, Memo OAM-16, Office Automation Group, MIT, Cambridge, MA, 1981.

[TSIC85] Tsichritzis, D.C., "Objectworld", in *Office Automation: Concepts and Tools*, Tsichritzis, D.C., ed., Springer-Verlag, Berlin, 1985, pp. 379-398.

[TWAI84] Twaites, K.J., *An Object-based Programming Environment for Office Information Systems*, M.Sc. Thesis, Dept. of Comp. Sc., Univ. of Toronto, Toronto, Canada, 1984.

[WEIS85] Weiser, S.P., *Using Object-Oriented Techniques for the Development of Office Information Systems*, M.Sc. Thesis, Dept. of Comp. Sc., Univ. of Toronto, Toronto, Canada, 1985.

[WOO85] Woo, C. and Lochovsky, F.H., "An object-based approach to modelling office work", *IEEE Database Engineering* **8**(4), 1985.

Hybrid: A Unified Object-Oriented System

O.M. Nierstrasz

Research Centre of Crete

ABSTRACT

Hybrid is a data abstraction language that attempts to unify a number of object-oriented concepts into a single, coherent system. In this paper we give an overview of our object model, describe a number of the language constructs, and briefly discuss the issue of object management.

1. INTRODUCTION

Hybrid is a programming language that attempts to unify a number of concepts that we consider to be integral to the "object-oriented" paradigm. These include:

1. data abstraction with classification, aggregation, and specialization
2. atomic events
3. concurrency control à la monitors
4. automatic triggering
5. object persistency
6. location-transparent object addressing

Our goal is to present these concepts in a natural, consistent manner in a small, high-level language. The language should be general-purpose, yet provide the programmer with adequate mechanisms for controlling low-level issues, such as process-switching, concurrency, and locking granularity, when these are required. There should be very few assumptions about the data classes and operations supported, so that the language can be truly extendible.

We intend our system to be useful to those who wish to rapidly develop distributed applications such as office forms systems [TSIC82], "intelligent mail" systems [HOGG85], and so on. A useful system should have a rich selection of powerful object classes, such as multimedia documents with version control, user interface objects with arbitrary key-binding and window management, role objects for encapsulating security policies, and so on, as outlined in [NIER85b]. Ideally, programming of new applications would often be a simple matter of putting existing objects together in new ways (much as one can put together "scripts" of tools in an environment like UNIX[1]).

Author's address: Institute of Computer Science, Research Centre of Crete, Box 157, Heraklion, Crete, Greece.
Author's current address: Université de Genève, Centre d'Informatique, 24, rue General Dufour, CH 1211 Genève 4, Switzerland.
UUCP: decvax!mcvax!cernvax!cui!oscar.

1. UNIX is a trademark of Bell Laboratories.

In this paper we present an overview of Hybrid, and a brief discussion of some of the more interesting language and implementation issues. A draft language definition is nearing completion, and we will be starting the implementation of an interpreter written in the C programming language [KERN78]. Future generations will be written in Hybrid itself.

2. OVERVIEW

The notion which is most fundamental to the design of Hybrid is that of data abstraction. Simply taken, it means that the representation of a data object, and the implementation of the operations it supports should be hidden. The only legitimate interface to the object should be through its operations. Programming languages that support some form of data abstraction are growing in number quite rapidly, and are becoming difficult to enumerate. Some of these are: Simula [BIRT73], Smalltalk [GOLD83], CLU [LISK77], Argus [LISK83], Zetalisp [WEIN81], C++ [STRO84], OOPC (Objective C) [COX83], Galileo [ALBA85], Modula [WIRT83], Oz [NIER85a], Taxis [MYLO80], OPAL [AHLS84], Smallworld [LAFF85] and BETA [KRIS83].

In Hybrid, ultimately everything is an object. The initiative to do things is also encapsulated in objects, consequently we do not have "objects" and "programs that manipulate objects", but just objects, and sometimes these objects attempt to communicate with one another. Naturally, the most active objects take their orders from the outside world (i.e., users, etc.).

An *object* is an *instance* of an *object class*. An object class is the specification of an abstract data type. The specification includes a *contents* portion, which describes the *instance variables* that distinguish individual object instances, and a *behaviour* portion, which describes the code shared by all instances of the same class.

An *integer* object, for example, might contain a single instance variable, namely a 32-bit value. Various instances of this class would each be represented by such a 32-bit value, and all the instances would share the code implementing the operations that integers support.

Instance variables are names for *acquaintances*, other objects that can be communicated with. An object that is properly contained in the contents (i.e., is stored locally) is a *dependent*, or *child*. Children are stored in *dependent variables*. Other acquaintances are known through *reference variables*, which reference more distant objects. Syntactically, there is little difference in the way the two kinds of variables are used. Internally, however, reference variables store *object identifiers*, or *oids*, which the system uses to keep track of objects.

In the case of integers, the 32-bit values would undoubtedly be dependent variables. A *mailbox* object, on the other hand, would probably know the messages it "contains" as external acquaintances.

Any object can thus be seen as an aggregate of its children plus a number of (non-child) acquaintances. Top-level objects (those with no parents) are called *independent objects*. The *descendents* of an object are its children and, recursively, the children of its descendents. A *domain* is a collection containing a single independent object, called the *domain parent*, and all its descendents. Conceptually it is most convenient to think of a domain as being a single object, namely the domain parent. For complex objects, such as documents, however, one may wish to deal with individual components (tables, figures, paragraphs) as objects in their own right.

Across a network, the universe of objects can be partitioned into a number of object *environments*, each of which is a logical node. Each environment has a single *object manager* that performs the duties of an operating system. The environments will normally correspond to physical nodes, but this is not a strict requirement. An object wishing to communicate with an acquaintance in another environment can do so through an *agent*, a local representative of that acquaintance. Objects that actually move from one environment to another leave a local agent behind so that they can be located, if necessary. Oids are globally unique, so there can never be confusion between a local object, and one that has immigrated.

3. LANGUAGE

An object specification consists of header, contents, and behaviour sections. The header contains the class name, a list of parameters (optional) and the name of the superclass whose contents and behaviour are inherited by the class being defined.

The contents portion consists purely of instance variable declarations. These are mapped to the representation of an object instance. In addition to the dependent and reference variables mentioned above, we have *sequence* variables, which are useful for manipulating sequences of references. (Sequences are a language construct rather than an object class. In order to effectively manipulate a sequence, it should be assigned to an appropriate class, such as a *list*, or an *array*.) In the example below, *talktime* is a dependent variable of class *date*, *who* is an acquaintance of class *mailer*, and *msgseq* is a sequence of references to messages:

```
var talktime : date;
ref who : mailer;
seq msgseq : message;
```

The behaviour section describes the processes that live in an object instance. All processes are part of some object's behaviour. This is consistent with the idea that there are no external "programs" manipulating objects, but only objects talking to each other.

The behaviour of an object consists of a single main procedure (labeled **start**) and a number of procedures and operations. The **start** procedure may split into a number of concurrent blocks. For a given object, each of these blocks is assigned its own process. Typically most of these processes are idle, waiting for requests from other objects. An object whose processes are all idle is normally retired to secondary storage.

Objects communicate by sending messages. A common type of message is a request to perform an operation. Two other message types are used to return from an operation (with an optional return value), and to report an exception (error) in the execution of the operation.

Depending on the situation, most message exchanges resemble procedure calls or remote procedure calls. The difference lies in the possibility of concurrent requests. Normally an object will queue requests as they come in, but a high-priority request may require immediate attention (especially in the case of real-time applications). Within a domain, where there is no real concurrency, internal requests can be handled as straight procedure calls.

The procedures of an object's behaviour are for that object's use alone. The interface available to acquaintances is the set of *operations* supported by that object. An object announces its readiness to perform an operation with the **accept** statement:

```
accept init ;
```

causes the running process to block, waiting for an acquaintance to call the *init* operation. A list of operations may be given, or the keyword **any** can be used to state that any operation can be invoked. The **select** statement can be used to specify follow-up actions after performing an operation, or, with the aid of an **else** clause, to prevent the process from blocking:

```
select
    accept opA ;
    /* followup activity for opA */ ;
or
    accept opB ;
    /* followup activity for opB */ ;
else
    /* no requests, so do something else */ ;
end
```

When no **else** clause exists, the **select** blocks. A **select** statement resembles Dijkstra's guarded commands [DIJK75]. The **accept** functions as a guard, resembling the input commands of Hoare's communicating sequential processes [HOAR78]. The **select accept** combination is also used in Ada [ANDR83].

Automatic triggering of processes is made possible with the **while await** statement:

while (mbox.empty) **await** (mbox.insert) ;

If the boolean expression after the **while** evaluates to *true*, then the running process blocks until one of the operations following the **await** is performed on the specified object. When notification is received, the expression is re-evaluated. The **while await** statement is atomic, in the sense that the system guarantees that the awaited operations cannot be "lost" in between the evaluation of the condition and the actual waiting.

The onus is on the programmer to indicate what operations may affect the condition. This is perhaps a nuisance, but the degree of control has its benefits:

while (x < y) **await** (x.incr, y.decr) ;

is surely preferable to:

while (x < y) **await** ((x,y).**any**) ;

The **while await** statement is important for detecting changes to acquaintances without having to either poll them, or modify their behaviour to perform the necessary notification. It may be used as a guard for a **select**.

Operations with alphabetic names are invoked by indicating an object, the operation, and a sequence of arguments:

mbox.insert(msg);

Operations may also be invoked using non-alphabetic operator names. Operator-overloading is important if one is to be able to program with objects in a concise, natural manner. The scheme used in Hybrid is to declare operators explicitly as **prefix**, **postfix**, or **infix**. The precedence rules dictate that prefix operators always bind more closely than postfix, and postfix more than infix. In addition, there can be one **index** operation, which is invoked by using (square) brackets. Together with the "dot" notation for invoking operations with alphabetic names, the precedence is as follows:

prefix >> { postfix, index, dot } >> infix

An expression such as:

*x++ - y[n].total

would be evaluated as:

((*x)++) - ((y[n]).total)

regardless of the semantics of the various operations. What remains is to identify operators appearing in a cascade between two expressions. This can be especially troublesome when an operator has multiple interpretations as, say, both prefix and infix, or perhaps even postfix as well. The rule used here, in lieu of disambiguating parentheses, is to start at the end of the cascade, going backwards, assuming prefix operators up to, but not including, the earliest possible infix. The remaining operators (if any) must be postfix. So:

x ++ += - y

would parse as:

(x ++) += (- y)

As an extreme example, suppose "@" has all three interpretations. Then:

```
x @ @ @ @ @ y
```

would parse as:

```
x @ ( @ ( @ ( @ ( @ y ))))
```

that is, one infix, and the rest prefix.

Object specifications can be parameterized. This is useful for defining objects that are to serve mainly as "containers" for other objects. The class of the contained object is listed as a parameter in the header of the specification:

stack (paramclass : *paramsuper*) : *object*

Elsewhere in the *stack* specification, the *paramclass* parameter can be used as though it were an actual class. The only operations that are allowed, however, on *paramclass* objects, are those inherited from the class *paramsuper*. Similarly, instances of stacks cannot assign a class to *paramclass* that is not a specialization of *paramsuper*. When a variable of class *stack* is declared, the parameter is given as, for example:

var jobstack : *stack* **of** *job*;

Although strong-typing is enforced in Hybrid, there are mechanisms for handling objects whose (precise) class is not known at compile-time. Suppose, for example, that mail messages can serve as containers for arbitrary objects. A clever mail-handler could unpackage certain kinds of messages containing objects of known classes. In the specification of the mail-handler, all one knows for certain is that message contents are of the generic class *object*. The specific class can be determined at run-time by using the **class** statement:

```
class {
thing : document =>
    folder.file(thing);
thing : meeting =>
    calendar.enter(thing);
thing : object =>
    /* default, if others fail */ ;
}
```

The *file* operation requires a *document* argument, but it is known to be invoked only if *thing* is verified (at run-time) to be of that class. Type-checking for the code within the various **class** cases can be done at compile-time.

Assignment in Hybrid is performed as follows:

```
x ← 5 ;
```

means, "the variable x is now a name for the object 5". This is different from:

```
x := 5 ;
```

which means, "apply the infix operation := to the object named by x, and send it the argument 5". In the first case, a new object is named; in the second, an existing object is modified. When an object is assigned to a variable, either the object itself is moved, or an oid is created and copied. This depends on whether the variable is a dependent or a reference variable.

Objects can be assigned several at a time:

```
(x,y) ← circle.centre ;
```

Furthermore, a sequence of unspecified length may be passed as an argument to an operation, or as a return value from an operation. *Sequence* variables are used to name these objects. If s is a sequence variable, then:

```
(x,s) ← s ;
```

assigns the head of *s* to *x*, and the remainder back to *s*. Generally a more satisfactory solution is to use the sequence to initialize a *list*, or some other suitable container class, and then use the *list* operations to access the elements of the sequence. Alternatively, one may iterate through a sequence with a **for** statement:

```
for x in (s) {
    /* do your stuff */ ;
}
```

Sections of code can be made "atomic" by declaring them as an *event* block. Before entering an event, the states of the objects used within the event (i.e., its *resources*) are sampled and saved. If the event *commits*, the saved states are discarded. If it *aborts*, the saved states are restored. During the event, the intermediate states are not visible to non-participants. Events may be nested, in the fashion described by Moss in his Ph.D. dissertation [MOSS81]. A parent event, detecting the failure of a child, may either choose to abort itself, or may attempt some other action.

Events are especially important when negotiating a transaction across a network or updating stable storage. In either case, interruption of the event, due to a crash, for example, could leave objects in inconsistent states. Events can be used to make such transactions atomic.

4. OBJECT MANAGEMENT

Objects in Hybrid are persistent. That is, once created, an object continues to exist until it is explicitly destroyed. In order to maintain a consistent view of objects in a given environment, we distinguish between stable and volatile storage. Stable storage contains a complete, consistent representation of all objects in the environment at some point in time. Volatile storage contains "working copies" of currently active objects. In the event of a crash, the contents of volatile storage (i.e., virtual memory) is discarded. Stable storage must therefore be updated very carefully, incrementally creating new, consistent views of the environment. Stable storage can be thought of as the permanent object database.

Depending on the requirements of the applications, the frequency with which stable storage is updated can vary. This means that many objects can be created and destroyed in between such updates, so that these objects are never written to stable storage. Where an atomic event is involved, it is, of course, important to update stable storage in a fashion that preserves the integrity of the event. For example, an event that spans several object environments, and several physical nodes, must, if it commits, have its effects observed reliably at all the nodes. Updating the stable storage of these several environments must also take place as a single, atomic event. Two-phase locking is typically used to implement this sort of behaviour [MOSS81, VERH78].

The object manager is also responsible for bringing needed objects into memory and resolving object identifiers to actual memory addresses. Object identifiers can be thought of as capabilities for addressing objects [FABR74]. (Oids are actually capabilities for sending messages (i.e., for performing operations). One thus distinguishes not merely between, say, read and write operations, but between all of the various operations supported by an object. This is the same philosophy adopted in the Hydra operating system [WULF74].)

A hash table is maintained for the objects currently in memory. The first time an active object attempts to address an acquaintance, the object manager does a lookup to see if the required object is already in memory. If not, it must be brought in. That acquaintance is then marked as being "open" for communication, and the oid is translated into a memory address. As long as the connection stays open, no further lookups are needed. Connections can be closed when an object becomes inactive (i.e., when there are no more outstanding messages for it) and all its processes are blocked. Inactive objects may be retired to stable storage. In fact, objects that have been inactive for only a brief period may quite reasonably become active again in a short time, so it makes sense only to retire objects that have been inactive for a while. (This is analogous to paging policies in operating systems.)

Part of the object manager's task is to keep track of the correspondence between the representations of instances, and the code that implements their behaviour. Since the latter is shared, care must be taken when modifications are made to the specification of an object class. In fact, a fair degree of intelligence should be built into the *class* meta-class. In particular, some sort of version-control is required to protect existing object instances when new versions of classes are installed. Furthermore, some degree of cleverness is needed to handle objects that move from one environment to another, since the class definitions may not be identical.

The object manager must negotiate concurrent accesses to objects. The problem is somewhat simplified by insisting that, for any given domain, there is never more than one process active. Domains are, in this respect, similar to monitors [HOAR74]. The arrival of a message (a request for service) interrupts the domain, which then decides whether it can handle the request immediately or not. Since the process issuing a request blocks if its request is not handled immediately, there is always a possibility of deadlock. When an object is participating in an atomic event, it is, strictly speaking, not permitted to handle any external requests, since the intermediate states of the event must not be visible. If deadlock occurs between several concurrent events, a "victim" must be chosen, and restarted. There is a substantial body of literature on this problem, and many schemes exist for handling it [BERN81, KOHL81].

A final issue of interest is that of garbage collection. The approach taken in Hybrid is that objects themselves are ultimately responsible for their own existence. A request to an object for it to commit suicide can well be ignored by that object, if it decides that the time is simply not right. Similarly, it is not up to the object manager to decide when objects are "garbage". On the other hand, if a passive object is no longer referenced by any other object (except by system objects, such as the object manager), then this is probably a good hint that the object has become garbage. The object manager therefore maintains a reference count for objects, and when that count drops to zero, the object in question is notified. It may then decide whether to commit suicide, or possibly initiate some other action.

5. CONCLUSIONS

Hybrid is a system for programming with abstract data types. There is a uniform view of all objects, so we do not distinguish between "objects" and "programs"—instead, every process is an integral part of the behaviour of an object. An idle object can be activated by sending it a message (i.e., invoking an operation). Objects can also be automatically triggered into action by having them wait for a precondition with a **while await** statement.

Objects are persistent, meaning that the system automatically saves consistent states of the object environment. Objects that are idle are normally kept in the stable storage object database. They are brought into memory only when they are activated by a message or a trigger condition.

Objects are referenced by unique object identifiers. The complete object universe consists of many object environments, each with its own object manager, communicating over a network. To protect the integrity of objects taking part in inter-node transactions (where failure of some component of the network is possible), atomic events are available as a programming construct. Events are useful for preventing inconsistent states of the object environment from being seen by objects contending for the same resources.

A draft language specification is nearly complete. A prototype implementation written in C will be starting shortly.

REFERENCES

[AHLS84] Ahlsen, M., Bjornerstedt, A., Britts, S., Hulten, C., and Soderlund, L., "An architecture for object management in OIS", *ACM Trans. on Office Information Systems* **2**(3), 1984, pp. 173-196.

[ALBA85] Albano, A., Cardelli, L., and Orsini, R., "Galileo: a strongly-typed, interactive conceptual language", *ACM Trans. on Database Systems* **10**(2), 1985, pp. 230-260.

[ANDR83] Andrews, G.R. and Schneider, F.B., "Concepts and notations for concurrent programming", *ACM Computing Surveys* **15**(1), 1983, pp. 3-43.

[BERN81] Bernstein, P.A. and Goodman, N., "Concurrency control in distributed database systems", *ACM Computing Surveys* **13**(2), 1981, pp. 185-221.

[BIRT73] Birtwistle, G.M., Dahl, O-J., Myhrhaug, B., and Nygaard, K., *Simula Begin,* Auerbach, Philadelphia, PA, 1973.

[COX83] Cox, B.J., "The object oriented pre-compiler", *SIGPLAN Notices* **18**(1), 1983, pp. 15-22.

[DIJK75] Dijkstra, E.W., "Guarded commands, nondeterminacy, and formal derivation of programs", *Comm. ACM* **18**(8), 1975, pp. 453-457.

[FABR74] Fabry, R.S., "Capability-based addressing", *Comm. ACM* **17**(7), 1974, pp. 403-412.

[GOLD83] Goldberg, A., and Robson, D., *Smalltalk 80: The Language and Its Implementation,* Addison-Wesley, Reading, MA, 1983.

[HOAR74] Hoare, C.A.R., "Monitors: an operating system structuring concept", *Comm. ACM* **17**(10), 1974, pp. 549-557.

[HOAR78] Hoare, C.A.R., "Communicating sequential processes", *Comm. ACM* **21**(8), 1978, pp. 666-677.

[HOGG85] Hogg, J., "Intelligent message systems", in *Office Automation: Concepts and Tools,* Tsichritzis, D.C., ed., Springer-Verlag, Berlin, 1985, pp. 113-134.

[KERN78] Kernighan, B.W. and Ritchie, D.M., *The C Programming Language,* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[KOHL81] Kohler, W.H., "A survey of techniques for synchronization and recovery in decentralized computer systems", *ACM Computing Surveys* **13**(2), 1981, pp. 149-183.

[KRIS83] Kristensen, B.B., Madsen, O.L., Moller-Pedersen, B., and Nygaard, K., "Abstraction mechanisms in the BETA programming language", *Proc. 10th ACM Symposium on the Principles of Programming Languages,* 1983, pp. 285-298.

[LAFF85] Laff, M.R. and Hailpern, B., *SW 2—an object-based programming environment,* Tech. Rep., IBM Thomas J. Watson Res. Ctr., Yorktown Heights, New York, NY, 1985.

[LISK77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction mechanisms in CLU", *Comm. ACM* **20**(8), 1977, pp. 564-576.

[LISK83] Liskov, B. and Scheifler, R., "Guardians and actions: linguistic support for robust, distributed programs", *ACM Trans. on Programming Languages and Systems* **5**(3), 1983, pp. 381-404.

[MOSS81] Moss, J.E.B., *Nested Transactions: An Approach to Reliable Distributed Computing,* Ph.D. Thesis, MIT/LCS/TR-260, Dept. Elec. Eng. and Comp. Sc., MIT, Cambridge, MA, 1981.

[MYLO80] Mylopoulos, J., Bernstein, P.A., and Wong, H.K.T., "TAXIS: a language facility for designing database-intensive applications", *ACM Trans. on Database Systems* **5**(2), 1980, pp. 185-207.

[NIER85a]	Nierstrasz, O.M., "An object-oriented system", in *Office Automation: Concepts and Tools*, Tsichritzis, D.C., ed., Springer-Verlag, Berlin, 1985, pp. 167-190.
[NIER85b]	Nierstrasz, O.M. and Tsichritzis, D.C., "An object-oriented environment for OIS applications", *Proc. 11th Int. Conf. on Very Large Data Bases*, 1985, pp. 335-345.
[STRO84]	Stroustrup, B., *Data Abstraction in C*, Computing Science Tech. Rep. 109, AT&T Bell Lab., Murray Hill, NJ, 1984.
[TSIC82]	Tsichritzis, D.C., Rabitti, F., Gibbs, S.J., Nierstrasz, O.M:, and Hogg, J., "A system for managing structured messages", *IEEE Trans. on Communications* **Com-30**(1), 1982, pp. 66-73.
[VERH78]	Verhofstad, J.S.M., "Recovery techniques for database systems", *ACM Computing Surveys* **10**(2), 1978, pp. 167-195.
[WEIN81]	Weinreb, D. and Moon, D., *The Lisp Machine Manual*, Symbolics Inc., 1981.
[WIRT83]	Wirth, N., *Programming in Modula-2*, Springer-Verlag, Berlin, 1983.
[WULF74]	Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F., "HYDRA: the kernel of a multiprocessor operating system", *Comm. ACM* **17**(6), 1974, pp. 337-345.

Object-Oriented Database Development at Servio Logic

David Maier

Servio Logic Development Corporation
and Oregon Graduate Center

Allen Otis
Alan Purdy

Servio Logic Development Corporation

ABSTRACT

We describe the development of the GemStone object-oriented database system, which supports objects and messages similar to those in the Smalltalk-80 language. We summarize the functional requirements of the system. We discuss key decisions made and technical challenges encountered during the development.

1. INTRODUCTION

This paper reports on the development of an object-oriented database system named "GemStone". The goal of our work has been to merge object-oriented programming language technology with database technology. GemStone combines the powerful data type definition and code inheritance properties of Smalltalk-80 with permanent data storage, multiple concurrent users, transactions, and secondary indexes.

For the past two years we have been developing a commercial database product running in a VAX/VMS environment with IBM-PC front-end interfaces. The product is aimed at application developers desiring flexible data modeling on which to build their next generation of applications. The system is currently being tested, with first customer shipments planned for the first quarter of 1986.

This paper outlines the requirements that guided our work and gives an overview of key decisions made during the development of GemStone. We identify major problems encountered during development, including some which remain as topics for further research.

Authors' address: Servio Logic Development Corp., 15025 S.W. Koll Parkway #1A, Beaverton, OR 97006 (503/644-4242).
Author's address: Department of Computer Science and Engineering, Oregon Graduate Center, 19600 N.W. von Neumann Drive, Beaverton, OR 97006 (503/690-1154).
CSNET: maier@Oregon-Grad.

2. GOALS AND REQUIREMENTS

Our overall goal is to provide a tool for the solution of data management and information modeling problems not easily solved using relational or network systems. Following are more specific requirements that guided the product development work.

2.1. Data Modeling Requirements

GemStone must provide the following modeling capabilities [MAIE84]:

- Allow the application developer to define a data model matching the structure of information in his problem.
- Support new data types defined by the user, rather than constraining the user to a fixed set of data types. New types must share the same syntax and semantics as the system-supplied types, for the purposes of application programming.
- Model the behavior of entities in the real world, not just their structure. For example, GemStone should facilitate storage of rules and actions as part of the data and allow queries to be stored as data. The system should facilitate rapid prototyping of solutions to information management problems and provide for handling of unexpected data values.
- Package behavior with structure to create new data types.
- Provide direct support for variable length data structures. Fields within a record may be variable in length; in addition a variable number of records may exist in an array or a set.

We distinguish between data types and data structures in defining these requirements. A *data type* is a collection of operators—the *protocol* for operating on a particular structure. *Data structures* are made up of atomic values (such as integers, strings, or Booleans) plus constructors (such as record, set, or array). In conventional systems, there is a one-to-one correspondence between data types and data structures. Atomic values typically have a fixed set of operations such as arithmetic and comparison. Constructors also have a fixed set of operations such as "set field", "get field" for records; or "add record", "delete record" for relations. These restrictions limit the data modeling capabilities of such a system. The lack of nested application of constructors means that certain real-world relationships cannot be modeled directly with conventional systems. For example, it is not possible to define a relation of relations to model projects for employees of a department. The inability to add new operations limits the new types that can be added to a system. For example, an ordered list can be represented with a relation by including an "order" field, but it is not possible to add list operations, such as "insert after", to the query language. Such an operation must be implemented via a sequence of database calls from a general purpose language.

For GemStone to get around the constraint of "data types = data structures", its data model must provide the ability to define new operations on a data structure, rather than supplying only a fixed set associated with each constructor. To get reasonable performance, the collection of constructors must be rich enough that most data types have direct implementations. In particular, we should be able to capture many-to-many relationships, collections, and sequences directly. For an easily usable system, we should be able to nest the structuring operations to arbitrary levels, and use previously defined data types as building blocks for other types.

2.2. Data Management Requirements

The following data management functions must be provided:

- multiple concurrent users
- data location transparency
- security, to allow users to keep data private and declare degrees of sharing for public data

- concurrency transparency (as far as possible) and enforcement of serializability of transactions
- user-selectable replication of data, so that sensitive data can survive a single media failure with no loss of committed transactions
- implementation of transactions with careful update of data and atomic commit/abort operations to ensure no loss of committed data on power failure
- schema definitions to support creation of secondary indexes, system maintenance of indexes, and index transparency

In addition, the object-oriented features of the GemStone require the system to maintain object identity and manage object storage space.

2.3. User Interface and Environment Requirements

The following user interfaces are required:

- an interactive interface for definition of new data types, and for direct execution of ad-hoc queries in GemStone's OPAL language
- a procedural interface to conventional languages, such as C and Pascal, to allow connection of existing applications to the database, and to allow OPAL code to be mated to user I/O functions
- a windowing package on which to build user interfaces for applications

3. DEVELOPMENT DECISIONS

Several key decisions were made during the transition from research to product development. In this section we define some terms and then describe some of these decisions.

3.1. Background

Figure 1 presents the elements of GemStone. All data items are ***objects,*** either ***atomic*** (numbers, characters, booleans) or ***structured.*** All structured objects are divided into slots called ***instance variables,*** each of which has a value that is another object. Structured objects come in three flavors based on the types of their instance variables. *Named instance variables* resemble attribute names in a relational tuple. *Indexed instance variables* are consecutively numbered starting at 1, much like elements of an array. *Anonymous instance variables* are used to form collections where only membership counts, and order is immaterial, such as bags and sets.

The behavior of an object is described via *messages*, which are commands sent to an object (by another object) to update its state or report on the current value of its state. Each message is implemented by a *method*, which is a procedure written in OPAL. *Classes* are the encapsulation mechanism for bundling behavior with structure to form new types. A group of objects with the same structure, messages, and methods have that common information factored out and stored in a *class-defining object* which all those objects reference. Objects in such a group are called *instances* of the class. A class-defining object can restrict the type of object that can be the value of an instance variable in any of its instances. Further, the classes are organized into a class hierarchy through which structures and methods are inherited.

All objects in the system reside in a disk-based object space which is divided into ***repositories.*** A repository represents a dismountable partition of the object space and is implemented as a direct access disk file on the underlying operating system. Repositories are further divided into non-overlapping regions called ***segments*** for purposes of authorization and concurrency control. A segment is a chunk of object storage that is owned by a particular user, who can store objects in it and can grant read or write access

Approximate Equivalences

GemStone	Conventional
object	record instance, set instance
instance variable	field, attribute
instance variable constraint	field type, domain
message	procedure call
method	procedure body
class-defining object	record type, relation scheme
class hierarchy	database scheme
class instance	record instance, tuple
collection class	set, relation

Fig. 1. Approximate equivalence of GemStone elements to conventional data base elements.

to other users. Segments expand to accommodate the objects stored in them. While repositories and segments are partitions of physical storage, their behavior is accessible via OPAL objects that represent them. Thus we mask the physical implementation of repositories and segments while still allowing control of them from within OPAL.

3.2. Language

We decided to use an existing object-oriented language, Smalltalk-80 [GOLD83], as the basis for the syntax and semantics of our own language, OPAL. Such a computationally complete and extensible language meets our data modeling requirements. It also provides one integrated language for data model definition, data manipulation, and system control. In addition, most of the logic of an application (except for user interface I/O) can be written directly in this language. The Smalltalk basis also provides an existing literature and market base which reduces our development and marketing expense. We have extended Smalltalk in the area of associative access support for queries.

3.3. OPAL Implementation

The OPAL language makes it possible to write a large part of the system in OPAL itself. We used this feature extensively to get our first prototype running quickly [COPE84]. Test results from the first prototype helped determine which functions of the product needed to be implemented as primitive operations and which could be written in OPAL and executed interpretively.

We implemented OPAL by writing our own object storage manager, OPAL compiler, and interpreter. This approach was required to provide a multi-user, disk-based system, as opposed to a single-user, memory-resident Smalltalk system. We also developed our own class hierarchy and virtual image, which provide a set of system-supplied data types tailored to database functions.

3.4. Data Management

GemStone's transaction control uses an optimistic approach that gives read-only transactions priority over read/write transactions when they request a commit. We assume that read-only transactions are more frequent than read/write transactions. Authorization and concurrency are controlled at the granularity of segments, since control on individual objects would incur unacceptable performance penalties.

Repositories may be replicated on the disk for resiliency to media failures. Replication was chosen over the traditional transaction log file. Replication is more feasible to implement within the other constraints of our object storage manager.

3.5. GemStone's Environment

The OPAL language and storage management software runs in the DEC VAX/VMS environment. The interactive user interfaces include an OPAL class browser, source code workspace, and bulk loader/dumper. These interfaces constitute the OPAL Programming Environment (OPE). The OPE operates under the Microsoft Windows environment on IBM-PCs networked to a VAX, as shown in Figure 2. This user interface environment was chosen to provide a cost-effective means to offload some of the screen I/O processing from the VAX, and to allow the use of inexpensive, purchased window management tools.

4. DESIGN CHALLENGES

Obviously, in designing a system to meet this set of requirements we encountered problems. Following are summaries of the most challenging problems.

4.1. Indexes and Constraints

Integration of secondary indexes with a Smalltalk-like language is an area of little published research. We had to decide whether secondary indexes were associated with the class of an object or with instances of a class. At the source code level, we wanted to have minimal effect on the syntax and semantics of Smalltalk. In addition, the object storage space supports multiple connectivity; thus we have to provide the necessary functions at runtime to ensure that indexes are maintained consistent with the data as updates to the internal states of objects are performed. Associating an index with a class makes it easy to detect when the state of an object changes. However, unlike a relational system in which all records of a given type are in a single relation, instances of a single class can be stored in multiple OPAL collections which might belong to different applications. A given application should not have to bear the cost of updating indexes for another application. In addition, a user may not have authorization to access all of the instances of a class. To overcome these problems, we chose to create indexes on collection objects rather than on classes.

Efficient implementation of indexes required the addition of class constraint semantics to Smalltalk. We found it necessary to constrain both the class of objects stored as elements in a collection, and the class of objects stored as instance variable values in those elements. For example, if a collection of employees is to be indexed by their last name, we must be assured that every Employee object has a "name" instance variable, and that each "name" in turn has a "lastname" instance variable. The value of "lastname" must be constrained to be of a known class. The following excerpt of OPAL code illustrates specification of these constraints.

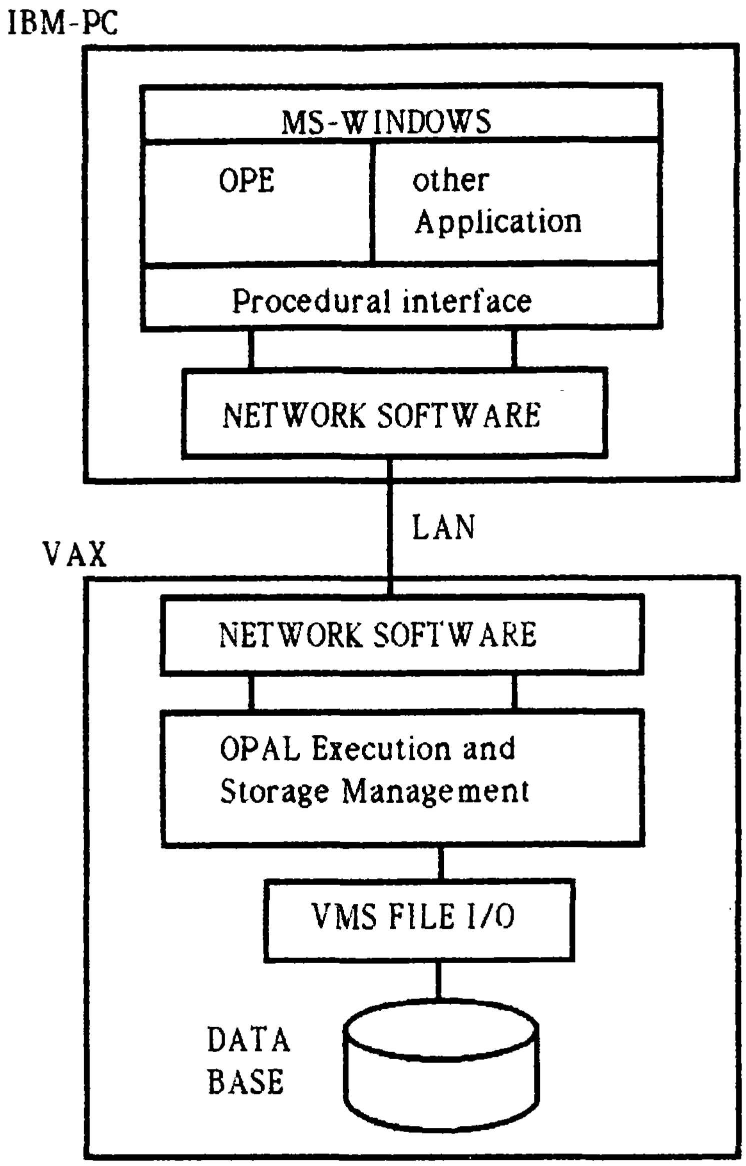


Fig. 2. GemStone environment.

```
Object subclass: #PersonName
    instVarNames: #('firstName' 'lastName')
    constraints: #[ #[ #firstName, InvariantString],
                    #[ #lastName, InvariantString] ].

Object subclass: #Employee
    instVarNames: #('name' 'taxIdNumber')
    constraints: #[ #[ #name, PersonName ],
                    #[ #taxIdNumber, Integer ] ] .

Set subclass: #EmployeesSet
    instVarNames: #()
    constraints: Employee .
```

4.2. Stable Storage and Transactions

Storage management in an object-oriented system can be computationally expensive for the functions of storage allocation, object identity maintenance, garbage collection, and variable-size object management. The many small objects, and the small number of very large objects, must be handled efficiently in both storage space and access time. Much of our effort has been devoted to developing memory management and buffering techniques which provide efficient object management in a disk based environment. In addition, such features as transactions, access control, and concurrent access further complicate the problem. In the case of concurrency and access control, individual objects are too fine a granularity for acceptable performance, so we control these functions at the *segment* level.

Because repositories of the object space can be dismounted, techniques must be provided to preserve consistent object identity when information is taken offline and later brought back online (possibly at a different site or on a different machine). Users must be given meaningful error messages if an object's repository is temporarily dismounted.

We are just beginning to accumulate benchmarks on physical access patterns to the object storage space. More work remains in the area of algorithms for clustering objects to optimize specific queries. Physical clustering of objects is complicated if the data for an application exhibit multiple connectivity. Improvements in this area will make GemStone more competitive in a production environment.

4.3. Documentation and Training

Object-oriented programming is still a very new field. Because there is not a large body of programmers experienced in Smalltalk-like languages, we are planning to offer formal customer training courses in addition to the normal user manuals.

GemStone does not make complex application domains simpler, but does allow more direct modeling with less encoding than other data models. It also allows capturing more of the information's semantics in the database. A formal object-oriented database design methodology for complex domains does not exist yet. We plan to offer such assistance in our training material.

5. SUMMARY

GemStone is designed to provide flexible data modeling capabilities for the application developer. GemStone provides an object-oriented, disk-based storage management system, with a matching object-oriented language, OPAL. The OPAL language is a descendant of Smalltalk-80 and is the language for data definition, data manipulation, and computation functions of GemStone.

ACKNOWLEDGEMENTS

The authors' colleagues at Servio have all played a vital role in the development of GemStone. We are indebted to their perseverance and dedication to the project.

TRADEMARKS

REFERENCES

[GOLD83]	Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.
[COPE84]	Copeland, G. and Maier, D., "Making Smalltalk a database system", *Proc. ACM SIGMOD Conf.*, 1984, pp. 316-325.
[MAIE84]	Maier, D. and Price, D., "Data model requirements for engineering applications", *Proc. IEEE 1st Int. Workshop on Expert Database Systems*, 1984, pp. 759-765.

Some Aspects of Operations in an Object-Oriented Database

Nigel Derrett
William Kent
Peter Lyngbaek

Hewlett-Packard Laboratories

ABSTRACT

The concept of *data abstraction* is a desirable feature of a data model, and is a central feature of a so-called "object-oriented" data model. This requires that operations on data must be stored in, and executed by, the database management system. This, in turn, means that the designers of an object-oriented data model must make some fundamental decisions about the nature of database operations and how they are to be represented, stored, and executed. We discuss some of the decisions which have been made in the design and implementation of the Iris data model.

1. INTRODUCTION

In this paper we shall discuss some aspects of database operations in the Iris DBMS prototype, under development at Hewlett-Packard Laboratories. We shall give a very brief introduction to the Iris data model, and then go on to discuss some specific aspects of this model. Before we start, however, it seems appropriate to state what we mean when we say that the Iris data model is "object-oriented".

The words "object-oriented" seem to have as many meanings as some other well-used phrases such as "block-structured" and "structured programming". We do not wish to enter or provoke a debate about what the words *ought* to mean, we will merely note that when *we* say that the Iris data model is "object-oriented" we mean the following:

- An Iris database is a collection of abstract data types [LISK74], with operations defined on the types, and with objects which are instances of the types.
- Objects can be accessed and manipulated only by invoking operations defined on their types. Data structures and details of implementation are hidden.

In other words, for us the two central features of an object-oriented data model are the entity concept and data abstraction. Both of these features are lacking in, for example, the pure Relational data model [CODD70], although an attempt has been made to add the entity concept to it [CODD79].

Authors' address: Hewlett-Packard Laboratories, P.O. Box 10151, Palo Alto, CA 94303-0866 (415/857-8729).
CSNET: hplabs!derrett@csnet-relay.

2. AN INTRODUCTION TO THE IRIS DATA MODEL

2.1. Objects

Objects in the data model are atomic items. Objects may serve as arguments to operations, and may be returned as results of operations. Each object is associated with one or more types; these types act as constraints, determining which operations the object may serve as an argument to.

A distinction is made between *literal objects*, such as character-strings and numbers, and *non-literal objects*, such as persons and bank-accounts. Literal objects are directly representable, whereas non-literal objects are representable only by surrogate identifiers. The identity of a non-literal object does not depend on the values of its properties—indeed, an object can exist without having any property values at all. The object is referenced internally in the database by its surrogate; it may be referenced from an application program either in terms of its property values (e.g., *the person with name "Jones"*) or in terms of its relationships with other objects (e.g., *the manager of the sales department*).

2.2. Types

Types are organized in a directed graph structure which supports generalization and specialization [SMIT77]. A given type may have multiple immediate subtypes and multiple immediate supertypes. Any object which belongs to the type also belongs to all of its supertypes. The type **Object** is an ancestor type of all other types, and therefore contains all objects.

An object may have more than one type at any time, and it may gain and lose types during the course of its existence. Thus, for example, an **employee** object in a company database might also be a **customer** object at some time, and will become a **retiree** object one day.

2.3. Operations

All Iris operations are functions, that is to say each operation returns a collection of results. (This collection may be empty.) Therefore we will use the words "function" and "operation" synonymously.

Properties of objects and relationships between objects are expressed in terms of (possibly multi-valued) functions, which are defined over their types. For example the function **Department_of** may be defined on objects of type **employee**:

Department_of: employee $\rightarrow$ department

Department_of(Smith) will return the department to which Smith is currently assigned.

A function can express properties of several objects. For example, the function

Marriage_date: person $\times$ person $\rightarrow$ date

defined on pairs of persons, should return the date on which the persons were married (if any).

Functions may return multiple and complex results. For example, the function

On_Hand: part $\rightarrow$ warehouse $\times$ quantity

returns the set of all warehouses in which a particular part is stored, together with their quantities-on-hand. This set may be empty.

Upper- and lower-bound constraints may be placed on the cardinality of parameters and results of operations.

An operation which accepts parameters of type **t** will also accept parameters which belong to subtypes of **t**. Thus, for example, an operation defined upon **person** objects will also apply to **employee** objects (assuming here that **employee** is a subtype of **person**). This means that subtypes inherit all of the properties of their supertypes.

2.4. Querying the Database

The database can be queried by the **FIND** command, which returns all combinations of objects which satisfy some predicate. The syntax of a query is

```
FIND variables WHERE (FORSOME variables ) predicate
```

For example, the query

```
FIND e/employee
    WHERE (FORSOME m/employee, d/department)
    d = Department_of(e) AND m = Manager_of(d) AND Salary(e) > Salary(m)
```

finds all employees who earn more than their managers. This query could have been formulated more succinctly as follows:

```
FIND e/employee WHERE Salary(e) > Salary(Manager_of(Department_of(e)))
```

The effect of the two query formulations is the same.

Predicates in queries may contain function calls, constants, variables, comparison operators, and logical operators.

Meta-data is modeled as a collection of objects, and queries about the database schema are made just like queries about user data.

2.5. Updating the Database

Properties of objects can be modified by changing the values of functions. For example:

```
SET Salary(Smith) = $30000.00
```

A multi-valued function can be changed by the **ADD** and **REMOVE** commands. For example:

```
ADD Employees_in(Sales) = Smith
```

2.6. Operation Definitions

A new operation is defined in two steps: first by specifying the types and cardinalities of its parameters and results and then by specifying how it is implemented. The current Iris prototype only allows the database definer to create operations which are functions without side-effects. Ways of defining and implementing more complex operations are discussed in the last section of this paper.

There are two ways to specify the implementation of a function in Iris: the graph of the function (i.e., the set of all input values and their corresponding results) may be stored explicitly in a database table, or the function may be derived from other functions. In the second case the implementation of the function is expressed as a **FIND** statement. For example, a **Supervisor_of** function may be defined, and its implementation specified as follows:

```
DEFINE Supervisor_of: employee → employee

DERIVE Supervisor_of(e) =
    FIND s/employee WHERE s = Manager_of(Department_of(e))
```

3. SOME ASPECTS OF OBJECTS AND OPERATIONS IN THE IRIS DATA MODEL

In this section we will discuss briefly some of the decisions which were made during the design of the Iris data model.

3.1. Active vs. Passive Objects

Any designer of an object-oriented system must decide from the start whether objects are to be regarded as active or as passive things.

Active objects, such as SIMULATION class objects in Simula [BIRT73], or entities in Beta [KRIS81], may be thought of as processes, each with its own script of actions waiting to be activated.

Passive objects, such as those in Smalltalk [GOLD83], do not have any ongoing activity associated with them per se, but each object has associated with it a sort of "subroutine library" of operations which can be called by an active process.

The Iris data model provides only passive objects, largely for reasons of simplicity and ease of implementation. It may be necessary to review this decision later on, if the model is to be extended to include triggers or active monitors.

3.2. To Whom do the Operations Belong?

Having chosen that database objects are passive, it is still necessary to determine the nature of the association between objects and the operations which may be applied to them.

One approach would be to say that each object has its own, possibly unique, set of operations belonging to it. This approach may be contrasted with that of Smalltalk, where each object belongs to a type, and operations are associated with the types. A third approach may be seen in Modula-2 [WIRT83] where objects and types have no actions belonging to them at all; objects are merely tokens which can be passed as parameters to operations.

The difference between these three approaches is a subtle one, but it pervades the whole design of an object-oriented data model. The Iris data model follows the Modula-2 approach—Iris objects can be thought of as surrogates. Properties of an object can be accessed and manipulated by passing the object as a parameter to an operation, but operations do not belong to the objects or to their types. Types are used as a constraint mechanism to determine whether an object can be passed as a parameter to a particular operation.

This enables the model to deal gracefully with operations which act upon multiple objects or multiple types. Consider, for example, the operation **assign_employee_to_department(e, d)** which takes an employee object and a department object and makes the employee a member of the department. Should this operation be associated with the employee type or with the department type?—neither seems more appropriate than the other. In the Iris data model, such an operation can be thought of as *free-floating*—it does not belong to any single type or object.

The database designer is allowed to hide implementation details of operations by creating *modules.* A module contains declarations of data structures and operations. Objects and operations within the module may be selectively exported to other modules or to application programs. Objects and operations which are not explicitly exported are hidden.

3.3. Properties of Objects

Two very common activities in database systems are reading and updating "properties" (attributes) of objects. Examples of properties are the name and salary of an employee. The designer of a data model must decide therefore how properties are to be treated, in particular whether accessing properties is syntactically different from calling functions. In a data model which supports functions, such as the Functional data model [SHIP81], and the Iris data model, there seems to be no good reason to make a distinction at the application program level between those properties whose values are stored in records on a disk and those whose values require some computation, since a property may change from one to the other during the life of a particular database. Therefore the Iris data model treats all properties as functions. This provides flexibility and a high degree of data independence. The action required to implement a particular function may be as simple as reading a field on disk or as complex as invoking a set of rules.

3.4. The GET and SET Commands

In a similar vein, conventional programming languages usually distinguish between functions which are implemented by storing the graph of the function (vectors and records) and those which are implemented as the result of a computation (subroutines). The former are typically updatable, whereas the latter are not. This distinction does not seem appropriate in a database management system where a combination of stored-graph functions and computed functions is required. The difference between these two sorts of functions is, of course, significant to the database management system itself, but it should not be visible to the application program. Relational database management systems deal with this problem by making computed functions (views) look like stored ones (relations). The Iris model has taken the opposite approach—all functions look like computed functions.

A function value may be accessed by the GET command and its value may be changed by the SET command:

c = GET F(a, b)

SET F(a, b) = newval

The word "GET" is omitted in our examples and in Iris queries, since the use of the function name alone can be interpreted as an implied **GET**. Thus, each function corresponds to a "load/update pair" of operations, where either member of the pair may be missing. (In other words, the function may or may not be gettable, and it may or may not be settable.) It would have been possible to require two operation names (e.g., **Get_F(a, b)** and **Set_F(a, b, newval)**) for the **GET** and **SET** operations on **F**, but this creates rather a lot of operation names in a database design, and the semantic interdependence of the pairs of operations is lost.

3.5. Relationships, Symmetry of Access, and Inverses of Functions

One potential weakness of a data model based on functions is that relationships are not supported well. This is in contrast to the Relational model, which supports relationships well but which supports functions poorly. The relationship between employees and their departments is reflected in a functional model by the function **Department_of(employee)**. A desire for symmetry in queries leads us to introduce the inverse function **Employees_in(department)**. These two functions must be kept synchronized, since they both reflect the same relationship. The problem becomes much worse if the user wishes to model a ternary, or n-ary, relationship, such as parts, warehouses, and quantities. Here there are eight interrelated functions, and the notion of simple inverses does not allow us to specify how they are interrelated.

Relationships are modeled in the Iris data model as predicate functions, for example:

Storage: part $\times$ warehouse $\times$ quantity $\rightarrow$ boolean

The predicate function evaluates to **true** if and only if the parameter objects are in the specified relationship. Functions such as

Quantity_in_warehouse: part $\times$ warehouse $\rightarrow$ quantity

may be defined in terms of these base predicates, and in this way families of related functions may be defined.

We note that a database designer may wish to group the operations in his or her design in either of two ways:

1. Grouping according to argument types (the traditional object-oriented approach), giving the sense of defining the properties of objects.
2. Grouping by relationships (the traditional relational approach), giving the sense of defining families of semantically-related operations.

Either of these ways of grouping operations together is valid in its context, and the Iris data model does not insist on one or the other.

3.6. Binding of Variables in Function Calls

The syntax of a function call in Iris distinguishes between two types of parameters—sometimes called *input parameters* and *result parameters*. For example, in the function call

n = Name(p)

the variable **p** is an input parameter and **n** is a result parameter. In some traditional programming languages, there is a rule which says that all input parameters must have values before the function call is executed, and that result parameters receive a value as a result of the call. An exception to this is Prolog [CLOC81], and the Iris data model has taken a similar approach. Although there is a syntactic distinction between input and result parameters in a function call, there is no semantic distinction made between them. Input and result parameters in a function call may or may not be thought of as having values before the call, and as becoming bound to values as a result of the call.

For example the queries

FIND p/person WHERE Name(p) = "Jones"

FIND n/string WHERE Name(jones) = n

are both valid Iris queries, one of which has the effect of binding the variable **p** to the set of all persons whose name is the string "Jones", and the other of which binds the variable **n** to the string which is the name of the person represented by the variable **jones**. In fact, a function call in a **FIND** clause really acts as a filter, which restricts the combinations of possible objects returned by the **FIND**.

4. SPECIFYING AND STORING OPERATIONS

How to specify and store operations is the biggest technical difficulty which must be faced by the designer of an object-oriented database management system. In effect, a programming language is needed to specify operations, and this language must be implemented as part of the DBMS. Various possible approaches to this problem have been identified, and several of them are being investigated in the Iris project.

We must start by making a distinction between the language in which an operation is specified, the form in which it is stored and implemented, and the language from which it is called. To draw an analogy with a traditional programming environment, a subroutine might be specified in FORTRAN, stored and implemented as machine code, and called from a Pascal program as if it were a Pascal procedure. The same thing can happen with database operations. This means that an object-oriented database management system must support one or more operation-specification languages, one or more storage-and-implementation languages, and one or more operation-calling languages, which may or may not be the same as the specification languages. In this section we will discuss specification languages, and how operations are stored and implemented.

The first approach is to use a special-purpose database language for specifying operations. The TAXIS system [MYLO80] uses this approach. An operation written in the database language is compiled and optimized into some internal form which is stored in the database and later interpreted, in much the same way that compiled queries are stored and executed in relational database management systems today. This approach has the advantage that the language can be tailored to the DBMS, but has the disadvantage that one ends up in the long run designing and implementing yet another programming language. Furthermore, if the database designer is also the applications programmer, then she must learn two programming languages—one for specifying operations on database objects, and the other for specifying operations on programming-language objects.

The second approach is to use an existing programming language and its implementation for defining and implementing database operations. The advantages are obvious—the database designer does not need to learn a new language, and the DBMS implementers do not have to write a compiler and/or interpreter for it. However, few conventional programming languages have the constructs needed to reference and manipulate sets of data in a database (two exceptions are Pascal/R [SCHM77] and Modula/R [REIM84]), and database optimization of operation bodies is not possible without writing a new compiler for the programming language in question.

If one takes this second approach, one may decide to link the code for each database operation into the database management system itself, or one may link it to the application program by providing a subroutine library of database operations. There is a practical problem of dynamic linking if new operations are to be added to a running database management system, and there is also a potential problem of reliability and security if users are allowed to link their own subroutines into the DBMS. It therefore seems preferable to link operations to the application programs, although the code for the operations may be stored and managed by the DBMS. This is particularly easy when applications are written in a language like Lisp, where operations can be stored in textual form in the database and loaded dynamically when needed.

The third approach is to use a small subset of an existing programming language, but to write a compiler which compiles operation bodies written in this subset into a form which can be interpreted by the database management system. This internal form can be the same as that used in the first approach. Additional operators may be added to the language subset if needed, in order to access database objects. This approach is really the same as the first one, except that existing programming-language constructs are used instead of inventing new ones.

All three of these approaches are being investigated, and Iris project members are currently concentrating on the first two: a special-purpose definition language and its compiler, and a way of specifying operations in an existing programming language (Lisp). It may be hoped that by providing the database designer with the full power of a programming language when she needs it, it will be possible to keep the special-purpose database language simple: most operations can be written in the simple language, but Lisp is available for implementing those which cannot. The database designer pays a cost for using this full power of Lisp for defining an operation—namely that no global optimization is performed on the database calls in the operation body, that intermediate results must be shipped from the database to the application program, and that the operation may not be callable from an application written in a different programming language.

5. CONCLUSION

Conventional data models, such as the Relational model, do not support well some important concepts, such as entities, types, constraints, actions, and data independence. In particular, it seems to be necessary to allow database operations to be specified and stored in the database management system. Several commercial relational database management systems do, in fact, already provide such stored operations (as stored queries and views), but in a rather ad-hoc way. The designers of the Iris data model have tried to reexamine the concept of a database operation, and to integrate it into the data model. In this paper we have discussed some of the issues which arose during the design of the model.

A prototype implementation of the Iris data model is currently nearing completion.

ACKNOWLEDGEMENTS

Among the persons who have contributed to the Iris project are: Dushan Badal, David Beech, Henry Cate, Ey-Chih Chow, Timothy Connors, Michael Creech, James Davis, Michael Epstein, Daniel Fishman, Karen Hall, James Kempf, Brom Mahbod, Robert Marti, Joseph Mohan, Thomas Ryan, Allan Shepherd, Alan Snyder, and Arun Swami.

The Iris project is a continuation of the IDM project at Hewlett-Packard Laboratories [BEEC83].

REFERENCES

[BEEC83] Beech, D. and Feldman, J.S., "The integrated data model: a database perspective", *Proc. 9th Int. Conf. on Very Large Data Bases*, 1983, pp. 302-308.

[BIRT73] Birtwistle, G.M., Dahl, O-J., Myhrhaug, B., and Nygaard, K., *Simula Begin*, Auerbach, Philadelphia, PA, 1973.

[CLOC81] Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, Springer-Verlag, Berlin, 1981.

[CODD70] Codd, E.F., "A relational model of data for large shared data banks", *Comm. ACM*, **13**(6), 1970, pp. 377-387.

[CODD79] Codd, E.F., "Extending the database relational model to capture more meaning", *ACM Trans. on Database Systems*, **4**(4), 1979, pp. 397-434.

[GOLD83] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[KRIS81] Kristensen, B.B., Madsen, O.L., Moeller-Pedersen, B., and Nygaard, K., *A Survey of the BETA Programming Language*, Norwegian Computing Center, Oslo, Norway, 1981.

[LISK74] Liskov, B.H. and Zilles, S.N., "Programming with abstract data types", *SIGPLAN Notices*, **9**(4), 1974, pp. 50-59.

[MYLO80] Mylopoulos, J., Bernstein, P.A., and Wong, H.K.T., "A language facility for designing interactive database-intensive applications", *ACM Trans. on Database Systems*, **5**(2), 1980, pp. 185-207.

[REIM84] Reimer, M., "The implementation of the database programming language Modula/R on the personal computer Lilith", *Software Practice and Experience*, **14**(10), 1984, pp. 945-956.

[SCHM77] Schmidt, J.W., "Some high level language constructs for data of type relation", *ACM Trans. on Database Systems*, **2**(3), 1977, pp. 247-267.

[SHIP81] Shipman, D., "The functional data model and the data language DAPLEX", *ACM Trans. on Database Systems,* **6**(1), 1981, pp. 140-173.

[SMIT77] Smith, J.M. and Smith, D.C.P., "Database abstractions: aggregation and generalization", *ACM Trans. on Database Systems,* **2**(2), 1977, pp. 105-133.

[WIRT83] Wirth, N., *Programming in Modula-2,* Springer-Verlag, Berlin, 1983.

A Message-Passing Paradigm for Object Management

Gul Agha

Massachusetts Institute of Technology

ABSTRACT

We discuss the *actor model* which has been proposed as a suitable basis for exploiting large-scale parallelism. Actor systems use message-passing to realize different control structures; communication is thus fundamental to computation in such systems. To organize tasks, allocate resources, and provide debugging tools, all computation in actor systems can be structured in terms of *transactions*. Although actors are a general purpose programming paradigm, several concepts from *distributed databases* are relevant to actor systems. Besides transactions, such issues include the notions of *consistency*, *concurrency control*, and *deadlock*. Actors provide a general means for easily implementing the usual solutions in these areas. The underlying actor architecture also provides support for distributed databases.

1. INTRODUCTION

The actor abstraction has been developed to exploit message-passing as a basis for concurrent computation [HEWI77a, HEWI77b]. The actor construct has been formalized by providing a mathematical definition for the behavior of an actor system [AGHA85b]. Essentially, an actor is a computational agent which carries out its actions in response to accepting a communication. The kinds of actions that are carried out are:

- Send communications to itself or to other actors.
- Create more actors.
- Specify the *replacement behavior*.

In order to send a communication, the sender must specify a mail address, called the *target*. The *mail system* buffers the communication until it can be delivered to the target. However, the order in which the communications are delivered is nondeterministic (thus permitting the dynamic routing of messages without substantial overhead). The mail address abstraction provides a mechanism for dynamic reconfigurability in a system.

A basic difference between actors and the entities used in classical databases is that actors are active

The work described in this paper was supported by a grant from the System Development Foundation.
Author's address: The Artificial Intelligence Laboratory, NE43-809, Massachusetts Institute of Technology, 545 Technology Sq., Cambridge, MA 02139 (617/253-5871 x5875).
CSNET: Agha@MIT-XX.

objects rather than passive data to be externally operated upon by a procedure. Declarative and procedural information is thus encapsulated in a single actor. Actors are also intrinsically parallel—different actors carry out their activities concurrently. The only constraint on the concurrency is the causal data dependencies inherent in the computation.

Actors may be implemented on a variety of architectures. One proposal involves using a network of multi-processors [HEWI80]. Another feasible implementation would be a fine-grain architecture based on MIMD machines linked in a hyper-cube of high-dimensionality. In any case, an actor architecture must support actor creation, real-time garbage collection, load balancing, and migration to maintain locality of reference. These features mean that a database manager using actors would be free from considerations of how to optimally distribute the information in the system. The actor model also provides a transaction oriented view of computation, reconfigurability and extensibility, and freedom from low-level syntactic deadlock.

One area of research in actors is the development of *description systems* for knowledge representation. New techniques developed in description and reasoning are likely to have a profound effect on future database systems since database systems will need to incorporate reasoning techniques for intelligent processing of queries. The ability to reason, in turn, requires the storage of intermediate results so that computational resources are conserved. The life-time of objects incorporating intermediate results is likely to be much shorter than that of the usual database records. In actor systems, most actors are extremely short-lived. Thus techniques developed for the management of actors will have applicability in intelligent database systems. An actor may be described by specifying:

- its mail address, to which there corresponds a sufficiently large *mail queue*; and,
- its *behavior*, which is a function of the communication accepted.

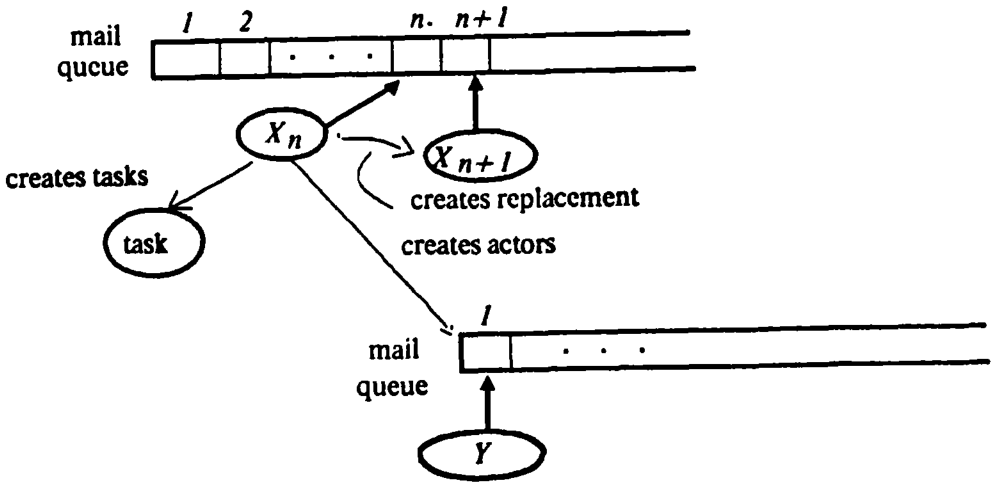


Fig. 1. An abstract representation of transition.

Abstractly, we may picture an actor with a mail queue on which all communications are placed in the order in which they arrive and an *actor machine* which points to a particular cell in the mail queue. The mail queue represents *serialization* of incoming communications. When an actor machine X_n accepts the n^{th} communication in a mail queue, it will create a new actor machine, X_{n+1}, which will carry out the replacement behavior of the actor. This new actor machine will point to the cell in the mail queue in which the $n+1^{st}$ communication is (or will be) placed. The two actor machines X_n and X_{n+1} will <u>not</u> affect each others behavior. This can be pictorially represented as in Figure 1.

An event-based picture for computation in actors uses *lifelines* which are shown in Figure 2. Each actor has an order of acceptance of communications which is linear. The events in the life of an actor are recorded in the order in which they occur: the further down the line, the later in local time. Activations (causal ordering of events) are indicated by the lines connecting two different actors with the arrow on the line indicating causal direction. Finally, each lifeline is labeled by the pending communications (i.e., the communications that have been received but not processed). [CLIN81] used collections of life-lines to provide a fixed-point semantics for actors. The resulting pictures are called the ***actor event diagrams***.

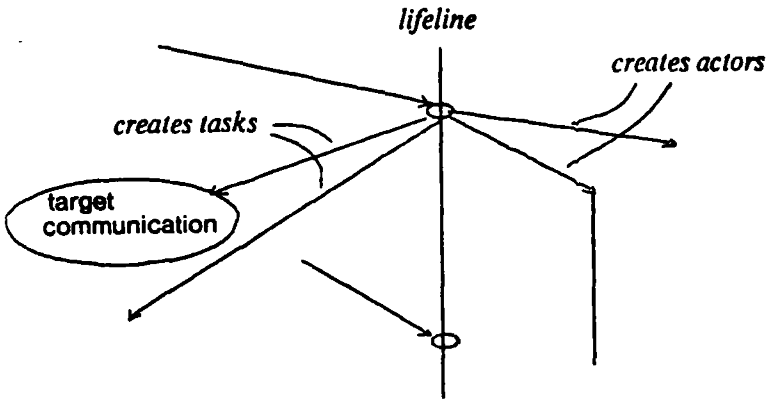


Fig. 2. Actor event diagrams. Each vertical line represents the events occurring in the life of an actor. The arrows represent causal links.

2. TRANSACTIONS

In higher-level actor languages, computation is structured in terms of transactions. Every communication is either a *request* or a *response*. The events between a request and its corresponding response are considered a *transaction*. However, note that if the response is a complaint then, in the usual sense of the term, the transaction has been *aborted*. The example below shows the implementation of a bank account in the actor language *Act3* [AGHA85c].

We use the keyword **Is–Request** to indicate a handler for a *request* communication. The request must come with the mail address of the *customer* to which the *reply* is to be sent. The customer is used as the target of the **reply**. A *request* also specifies a mail address to which a *complaint* can be sent should the request be unsuccessful. From a software point of view, providing independent targets for the complaint messages is extremely useful because it allows the error-handling to be separated from successfully completed transactions. Note that the expression in the *become* command specifies the replacement behavior.

```
(define (Account (with Balance ≡ b ))
    (Is-Request (a Balance ) do (reply b ))
    (Is-Request (a Deposit (with Amount ≡ a )) do
        (become (Account (with Balance (+b a ))))
        (reply (a Deposit-Receipt (with Amount a ))))
    (Is-request (a Withdrawal (with Amount ≡ a )) do
        (if (> a b )
            (then do (complain (an Overdraft )))
            (else do
                (become (Account (with Balance (-b a ))))
                (reply (a Withdrawal-Receipt (with Amount a )))))))
```

In a distributed system, it is not always possible to revert transactions without severely constraining the amount of concurrency in the system. This is because transactions are nested in each other, and sub-transactions may be shared between different transactions. The only way to guarantee the ability to revert any aborted transactions is to record the history of all computations. While it is theoretically possible to this (by associating an ***event recorder*** with each actor), the process is prohibitively expensive. Allowing ***complaint handlers*** permits us to tailor the corrective measures to the particular application domain.

In general, various means of concurrency control can be implemented in actors. In the next section, we present an example which frequently arises in actor systems and involves concurrency control to preserve the structure of transactions. The example is taken from [AGHA85a].

3. INSENSITIVE ACTORS

When an actor accepts a communication and proceeds to carry out its computations, other communications it may have received must be buffered until the replacement behavior is computed. However, the desired replacement for an actor may depend on communication with other actors. For example, suppose a checking account has overdraft protection from a corresponding savings account. When a withdrawal request results in an overdraft, the balance in the checking account after processing the withdrawal would depend on the balance in the savings account. Thus the checking account actor would have to communicate with the savings account actor, and more significantly the savings account must communicate with the checking account, before the new balance (and hence the replacement behavior) is determined. The relevant communication from the savings account can not therefore be buffered until a replacement is specified!

Essentially, the problem is of *locking* the actor to avoid anomalous interaction between independent transactions. However, an important characteristic of our solution is that it does not rely on external control over an actor's behavior. We deal with the problem simply by defining the concept of an *insensitive* actor which processes a type of communication called a ***become communication.*** A become communication tells an actor its replacement behavior. The behavior of an insensitive actor is to buffer all communications until it receives a communication telling it what to become.

First, consider what we would like the behavior of a ***checking account*** to be: if the request it is processing results in an overdraft, the checking account should request a withdrawal from its ***savings account.*** When a reply to the request is received by the checking account, the account will do the following,:

- Reply to the customer of the (original) request which resulted in the overdraft; and,
- Process requests it subsequently received with either a zero balance or an unchanged balance.

Using a *call expression*, where the value of expression is bound to the identifier in the *let expression,* we can express (in pseudo-code) the fragment of the code relevant to processing overdrafts as follows:

```
let r = (call my-savings [ withdrawal, balance - amount ])
    { if r = withdrawn
        then become checking-acc (0, my-savings)
        else become checking-acc (balance, my-savings)
    reply [r] }
```

To show how a call expression of the above sort can be expressed in terms of our kernel, we give the code for a bank account actor with overdraft protection. We give the code for illustrative purposes. A bank account with an overdraft protection is implemented using a system of four actors. Two of these are the actors corresponding to the checking and savings accounts. Two other actors are created to handle requests to the checking account that result in an overdraft. One of the actors created is simply a buffer for the requests that come in to the checking account while the checking account is ***insensitive.*** The other actor created, an *overdraft process,* is a customer which computes the replacement behavior of the checking account and sends the reply to the customer of the withdrawal request. We assume that the code for the savings account is almost identical to the code for the checking account and therefore do not

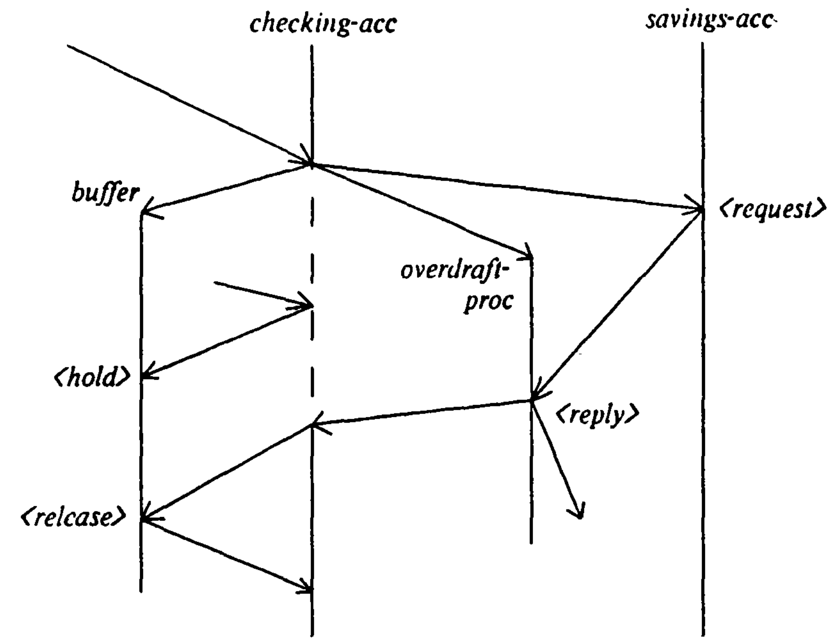


Fig. 3. Insensitive actors. During the dashed segment the insensitive checking account buffers any communications it receives.

specify it here. The structure of the computation is illustrated by Figure 3 which gives the actor event diagram corresponding to a withdrawal request causing an overdraft.

The behavior of the checking account, when it is not processing an overdraft, is given below. When the checking account accepts a communication which results in an overdraft, it becomes an insensitive account.

```
checking-acc (balance , my-savings ) [<request >]
    if <deposit request >
        then become <checking-acc with updated balance >
        send <receipt > to customer
    if <show-balance request >
        then send [balance ] to customer
    if <withdrawal request > then
        if balance ≥ withdrawal-amount
            then become <checking-acc with updated balance >
                send <receipt > to customer
            else let b = new buffer
                and p = new overdraft-proc
                {become new insens-acc (b , p )
                send <withdrawal request with customer p > to my-savings }
```

The behavior of an "insensitive" bank account, called *insens-acc*, is quite simple to specify. It is given below. The insensitive account forwards all incoming communications to a buffer unless the communications is from the overdraft process it has created.[1] The buffer can create a list of communications, until it receives a communication to forward them. It then forwards the buffered communications and becomes a forwarding actor so that any communications in transit will also get forwarded appropriately.

1. Due to considerations such as *deadlock*, one would program an insensitive actor to be somewhat more "active." Good programming practice in a distributed environment requires that an actor be *continuously available*. In particular, it should be possible to query an insensitive actor about its current status.

```
insens-acc (buffer, proxy) [request, sender]
    if request = become and sender = proxy
        then become <replacement specified>
        else send <communication> to buffer
```

Finally, we specify the code for a customer to process overdrafts. This customer, called ***overdraft-process*** receives the reply to the withdrawal request sent to the savings account as a result of the overdraft. The identifier *self* is bound to the mail address of the actor itself. The response from the savings account may be a *withdrawn, deposited,* or *complaint* message. The identifier *proxy* in the code of the insensitive account represents the mail address of the over-draft process. The proxy is used to authenticate the sender of any *become* message targeted to the insensitive actor.

```
overdraft-proc (customer, my-checking, my-savings, checking-balance) [<savings-response>]
    send [become, self] to my-checking
    send [<savings-response>] to customer
    if <savings response is withdrawn>
        then become checking-acc (0, my-savings)
        else become checking-acc (checking-balance, my-savings)
```

4. DEADLOCK

One of the classic problems in concurrent systems which involve resource sharing is that of *deadlock*. A *deadlock* or *deadly embrace* results in a situation where no further evolution is possible. One strategy is to limit access to shared resources in order to avoid the possibility of deadlock. The difficulty with *deadlock avoidance protocols* is that the mechanisms for avoiding deadlock have to be tailored using advance knowledge about how the system might deadlock. Furthermore, the need for centralized control in such protocols implies a serious bottleneck to the throughput in the system. However, this is the only sort of solution in systems relying on synchronously communicating sequential processes. In fact in languages using synchronous communication, deadlock has been defined as a condition where no process is capable of communicating with another [BROO83].

In actor systems, as in concurrent database systems [DATE83], deadlock avoidance is often unrealistic. The reasons why deadlock avoidance is not feasible in concurrent databases can be summarized as follows:

- The set of *lockable objects* is very large—possibly in the millions.
- The set of lockable objects varies dynamically as new objects are continually created.
- The particular objects needed for a *transaction* must be determined dynamically (i.e., the objects can be known only as the transaction proceeds).

The actor model addresses this problem in two ways. First, there is no *syntactic* (or low-level) deadlock possible in any actor system, in the sense of it being impossible to communicate (as in the Brookes' definition above). All actors must designate a replacement and that replacement can *respond* to any further communications. Thus a deadlock can be broken by a time-out mechanism without compromising the encapsulation of an actor's behavior.

An actor is also free and able to figure out a deadlock situation by querying other actors as to their local states and constructing "wait-for" graphs to detect cycles in the graphs, similar to what has been suggested for database systems [KING73]. We would carry out the process of breaking the deadlock in a completely distributed fashion. A concern about deadlock detection is the cost of removing deadlocks. Experience with concurrent databases suggests that deadlocks in large systems occur very infrequently [GRAY80]. The cost of removing deadlocks is thus likely to be much lower than the cost of any attempt to avoid them.

A system of actors is best thought of as a community [HEWI84]. Message-passing viewed in this manner provides a foundation for reasoning in open, evolving systems. Deadlock detection is one particular application of using message-passing for reasoning in an actor system: Any actor programmed to be sufficiently clever can figure out why the resource it needs is unavailable and, without remedial action, about to stay that way. To solve this sort of a problem, *negotiation* between independent agents becomes important. In open and evolving systems, new situations will arise and thus the importance of this kind of flexibility is enormous. Another consequence of "reasoning" actors is that systems can be easily programmed to *learn*.

5. CONCLUSIONS

New technology is providing us with ever increasing computational power. To cope with the added complexity inherent in larger systems, we need the ability to subdivide the large systems and provide tools for incremental development. Actor-based architectures provide an ideal means for parallel realization of open, evolving systems. Many of the principles used in actor languages are related to concepts first developed for database systems. Research in actors is likely to interact productively with research in distributed databases.

ACKNOWLEDGEMENTS

The author acknowledges helpful comments from Carl Hewitt, Peter de Jong, Carl Manning and Tom Reinhardt.

REFERENCES

[AGHA85a] Agha, G., "Semantic considerations in the actor paradigm of concurrent computation", in *Seminar on Concurrency*, Springer-Verlag, Berlin, 1985, pp. 151-179.

[AGHA85b] Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems*, Tech. Rep. 844, The Artificial Intelligence Lab., MIT, Cambridge, MA, 1985.

[AGHA85c] Agha, G. and Hewitt, C., "Concurrent programming using actors: exploiting large-scale parallelism", *Proc. 5th Conf. on Foundations of Software Technology and Theoretical Computer Science*, Springer-Verlag, Berlin, 1985 (forthcoming).

[BROO83] Brookes, S.D., *A Model for Communicating Sequential Processes*, Tech. Rep. CMU-CS-83-149, Comp. Sc. Dept., Carnegie-Mellon Univ., Pittsburgh, PA, 1983.

[CLIN81] Clinger, W.D., *Foundations of Actor Semantics*, Tech. Rep. 633, The Artificial Intelligence Lab., MIT, Cambridge, MA, 1981.

[DATE83] Date, C.J., *An Introduction to Database Systems*, Addison-Wesley, Reading, MA, 1983.

[GRAY80] Gray, J., *Experience with the System R Lock Manager*, Tech. Rep., IBM San Jose Res. Lab., San Jose, CA, 1980.

[HEWI77a] Hewitt, C.E., "Viewing control structures as patterns of passing messages", *Journal of Artificial Intelligence* 8(3), 1977, pp. 323-364.

[HEWI80] Hewitt, C.E., "Apiary multiprocessor architecture knowledge system", *Proc. of the Joint SRC/Univ. of Newcastle upon Tyne Workshop on VLSI Machine Architecture and Very High Level Languages*, Tech. Rep., Univ. of Newcastle upon Tyne Computing Lab., 1980, pp. 67-69.

[HEWI77b] Hewitt, C.E. and Baker, H., "Laws for communicating parallel processes", *Proc. IFIP Congr., Inf. Processing '77*, 1977, pp. 987-992.

[HEWI84] Hewitt, C.E. and de Jong, P., "Open systems", in *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, Brodie, M.L., Mylopoulos, J., and Schmidt, J.W., eds., Springer-Verlag, Berlin, 1984, pp. 147-164.

[KING73] King, P. and Collmeyer, A., "Database sharing: an efficient mechanism for supporting concurrent processes", *Proc. NCC*, 1973, pp. 271-275.

Object Management and Sharing in Autonomous, Distributed Data/Knowledge Bases

Dennis McLeod
Surjatini Widjojo

University of Southern California

ABSTRACT

This paper describes an experimental distributed object management system. A simple model for object management is presented, including a set of primitive manipulation and retrieval operations. A model and mechanism to allow controlled object sharing among multiple data/knowledge bases is specified. A prototype implementation of this system, currently under development, is reviewed.

1. INTRODUCTION

An important current trend in information management is from a record-based to an object-based orientation [AFSA84, BROD84, KENT79, LYNG84a, TSIC82]. In particular, existing record-oriented database management systems fulfill many of the requirements of traditional database application domains, but they fall short of providing facilities well-suited to applications in office information systems [GIBB83, LYNG84a], design engineering databases [AFSA85, BATO85, EAST80, KATZ82], and artificial intelligence systems [KERS84]. In an object-oriented system: information units of various modalities, levels of granularity, and levels of abstraction have individual identity; semantic primitives for object classification and inter-relation are explicitly part of the system; and objects can be active as well as passive.

The purpose of the research project described here is to devise and experimentally test concepts, techniques and mechanisms to support a distributed object management system, termed the *Distributed Personal Knowledge Manager (DPKM).* DPKM is an adaptive tool for the non-computer expert; it is intended to allow end-users to define, manipulate, and evolve collections of information. DPKM handles various forms of information/knowledge in an integrated manner; this includes symbolic data, meta-data, derived data (rules), behavioral information (procedures), constraints, and mixed modality information. An individual DPKM also serves as an access port to other (external) information resources.

This research specifically focuses on the following issues:

This research was supported in part by the Joint Services Electronics Program through the Air Force Office of Scientific Research under contract F49620-85-C-0071, by the National Science Foundation through its Computer Engineering Program under grant ECS-8310774, and by the Defense Advanced Research Projects Agency under contract MDA903-81-C-0335.
Authors' address: Computer Science Department and Information Sciences Institute, University of Southern California, Los Angeles, CA 90089-0782 (213/743-8302).
CSNET: McLeod@USC-CSE.

- an information model to support the integrated specification of various forms of knowledge [AFSA85];
- an end-user interface providing a layered view of knowledge, multi-media information input and output, and prescriptive user guidance [AFSA85, LyMc84b];
- an efficient mechanism for internally organizing and evolving knowledge bases [AFSA85];
- a multi-level networking/communication mechanism to support inter-DPKM information exchange, sharing, coordination, and access control [HEIM85, LYNG84a];
- an approach to (rudimentary) machine-initiated, user-assisted acquisition of knowledge [BORG85].

This research is by definition interdisciplinary in that it must draw on concepts and techniques in the areas of: knowledge representation and engineering; database models, interfaces, and distributed databases; computer networking and message systems; information security and protection; and applied machine learning. An experimental prototype implementation of DPKM is under development, based on an interconnected network of personal workstations and computers (AT&T 3B2s and 3B20s). It is intended that the DPKM project will examine applications in a variety of domains, but it will principally focus on the researcher and design engineer (viz., the VLSI designer) for the purposes of initial experimental application and testing.

This short paper focuses on two of the essential aspects of the DPKM system: the object model and manipulation operations of DPKM; and techniques to support communication and sharing of information objects among DPKM data/knowledge bases.

2. CONTEXTS, OBJECTS AND MAPPINGS

In the approach taken in this research, each data/knowledge base is a logical *context* (node) in a logical *network*. Associated with each context is a collection of information *objects* and *mappings*. The objects model units of potentially shareable information at different levels of abstraction; these are: symbolic (or atomic) objects, abstract objects, object classifications via enumeration (enumerated sets) or via selection predicate (predicate sets), constraints on inter-object relationships (generic mappings), and behavioral objects (procedures). Mappings specify relations among instances of objects. It is significant to note that information objects classically distinguished in database terms as "schema" and "data" are treated here within a uniform framework. A primitive set of operations supports the manipulation and sharing of objects.

In what follows, we first describe the different *flavors* of objects in our model and the mappings in more detail. Following this we consider operations on objects in a single context. Next we see how these operations must be modified or constrained to permit sharing of objects across contexts and introduce additional operations necessary for this purpose.

3. OBJECTS AND MAPPINGS

The different flavors of objects supported by DPKM can be characterized as follows:

- *Symbolic/atomic objects* correspond to nondecomposable units of information. Examples of symbolic objects are the name "Jane Smith" and the phone number "743-5501". We denote symbolic/atomic objects by strings in double quotes in this paper. Each symbolic/atomic object can be referenced by such a string.
- *Abstract objects* correspond to things and concepts that are described by their relationships with other objects. An example abstract object is the person **Jane Smith**. We denote abstract objects by mnemonic object-names in boldface. This model would relate **Jane Smith** to such objects as her name ("Jane Smith"), her phone number ("743-2747"), her employer (**USC-ISI**), etc.
- *Object sets* are used to classify a set of objects which are similar according to some criteria. There are

two different types of sets; the *enumerated set* and the *predicate set*. The instances of an object set are identified either by enumeration (enumerated set) or via a *selection predicate* (predicate set) that specifies the instances relative to one or more other set objects. An example object set is **Person** whose members include specific abstract objects representing person objects.

- *Behavioral objects* embody operations/procedures. Predefined primitive behavioral objects are provided for creating, manipulating, destroying and sharing objects. The user can also define new behavioral objects and modify existing ones.
- *Generic mapping objects* specify a mapping template from one object set to another. A mapping template which describes the general category of mappings consists of a mapping name, an inverse mapping, the domain and range of the mapping, and simple constraints on inter-object relationships. An example of a generic mapping object is **Presenter of/Talk of** which forms the template for the mapping of **Persons** to **Talks**. The mapping can also be constrained (e.g., "unique" and/or "single-valued").

Mappings embody binary relationships among objects (functions). A mapping is represented as a 3-tuple <x,y,z>, where y is a previously defined *generic mapping object*, x is a member of the *domain object set* of y and z is a member of its *range object set*. An example of a mapping and its inverse would be (**President Reagan**, **Presenter of**, **Tax reforms**) and (**Tax reforms**, **Talk of**, **President Reagan**). Mappings also represent "meta" data (e.g., (**Talk-of, has-domain, Talks**)).

4. OBJECT MANIPULATION OPERATIONS

A collection of primitive operations is provided to support object manipulation and retrieval. For expository purposes, the operations are described as primitives that are embedded in a host programming language. It is assumed that the host programming language supports the data types *object-id* and *set of object-id*. Variables of type object-id contain unique, user-specified or system-generated object-identifiers. An *object-reference* (abbreviated *object-ref*) is a handle on (a pointer to) an object. An object-reference can be a user-specified identifier, system-generated object identifier, or a variable of type object-id holding an object-identifier. The primitive operations are themselves stored in the database as behavioral objects.

4.1. Operations for a Single Context

A collection of primitive operations for manipulating and retrieving objects within a given context are described here. The CREATE operation generates a new object of defined flavor. If object-name N is specified in the parameter of the CREATE operation, the object is assigned the identifier N. The system will generate a unique object-id for the object created. For example:

```
CREATE("abstract", "John Smith")
CREATE("enumerated-set","students")
CREATE("generic-mapping","has-employer")
CREATE("behavioral", "get-cs-students")
jane:=CREATE("abstract")
```

The above operations, when executed, will create an abstract object named "John Smith", an (empty) enumerated set "students", a generic-mapping object "has-employer", a behavioral object "get-cs-students", and an abstract object with its system-defined object-id stored in the variable jane. In the last case, if the variable jane is reassigned a new value, the reference to the abstract object created will be lost. Note that symbolic/atomic objects are not created. They exist universally and cannot participate in any database relationships except via a "has-name" relationship with other objects. The DELETE operation removes a given object from the database. Deletion is allowed only if the object is not participating in any mapping instance. For instance, the operation DELETE("John Smith") would remove the object **John Smith** from the database only if it is not related to any other objects in the database. The operation IS-OBJECT queries for the existence of an object. After the execution of the above CREATE operation, the operation IS-OBJECT("John Smith") would return the value true, but, if the above are the only CREATE operations that have been performed, the operation IS-OBJECT("Mary") would return the value false.

For behavioral objects, we need to be able to relate the object to a procedure and to invoke that procedure. The operation DEFINE-PROCEDURE-BODY relates behavioral object N to its executable body P and specifies the expected input and output parameters of the behavioral object when invoked. For example, DEFINE-PROCEDURE-BODY(**get-cs-students**, procedure-body, I, O) relates the procedure body to the object **get-cs-students** and defines the input and output parameters to the procedure. The INVOKE operation invokes the specified behavioral object on a given input and/or output parameter. In the example given above, INVOKE(get-cs-students, I:students, O:cs-students) will execute the procedure body associated with the object get-cs-students, which will take as its input the object set **students** and the results will be stored in the object set **cs-students**.

The DEFINE-GENERIC-MAPPING operation creates a generic mapping object and defines its inverse, domain, range, domain constraint, and range constraint. For example, the operation DEFINE-GENERIC-MAPPING(has-employer, is-employer-of, students, employers, many, many) defines a many to many mapping **has-employer** from students to employers and a many to many mapping **is-employer-of** from employers to students. Other mapping constraints (e.g., total, onto, etc.) can be similarly defined.

There are two types of object sets in DPKM. The DEFINE-SELECTION-PREDICATE operation defines the objects that form the members of the object set. If the predicate set object **Age** has been created, and Z is a set of integers, DEFINE-SELECTION-PREDICATE(Age, $Z>0$ and $Z<100$) defines Age to be all $x \in Z$ such that $x>0$ and $x<100$. The semantics of the selection predicate is beyond the scope of this paper (see [AFSA85, AFSA84, LYNG84b]). The DEFINE-ENUMERATED-SET-DOMAIN operation defines the domain of an object set. For example DEFINE-ENUMERATED-SET-DOMAIN(cs-students, students) defines cs-students to be a subset of students. The ADD-TO-SET operation adds an object to the set specified. This operation results in an error if the object to be added does not belong to the domain defined for that set. The REMOVE-FROM-SET operation removes a given object from the enumerated set object. The operation would result in an error if the object to be removed is not a member of the enumerated set. The GET-MEMBERS operation returns all the members of the specified object set.

Relationships among objects are created via the RELATE operation. This operation creates a mapping from object D to object R via generic mapping M. An inverse mapping is also created, since the inverse is known when M is defined. Following the above example, RELATE("John Smith", "has-employer", "USC") would create the following 3-tuples: <**John Smith**, **has-employer**, **USC**> and <**USC**, **is-employer-of**, **John Smith**>. Relationships are removed via DETACH. This operation removes the relationship <D,M,R> and its inverse from the database. For example, DETACH("University of Southern California" ,"is-employer-of","John Smith") would delete the two 3-tuples created above.

Queries to the database can be done via the SELECT and CHOOSE operations. The SELECT operation is used to retrieve objects from the database. It returns a set of objects satisfying a predicate specified by three parameters, D, M, and R. Each parameter is either a question mark ("?") or a set of objects. The question mark denotes the objects in question. The don't care symbol "*" is the set containing all the objects in the database. Selection predicates can also be used in the parameters to add to the power of SELECT. The CHOOSE operation is similar to the project operation in the relational data model. It operates on a set of mapping instances and returns a set of domain, generic mapping, or range objects. For example, to get the set of cs-students employed by USC:

SELECT(CHOOSE (D, SELECT (*, "has-employer", "USC"), "is-member-of", "cs-students")).

Note that selection predicates may be used in the parameters to the above operations. Further, a high level end-user DPKM interface is being constructed which supports, among other functions, a stepwise prescriptive approach to predicate formulation.

5. SHARING AMONG AUTONOMOUS DATA/KNOWLEDGE BASES

A very important current trend is towards an environment consisting of a network of personal computers, connected also to larger-scale mainframes. In such an environment, support is needed to manage local (personal) data/knowledge bases and to facilitate sharing and coordination among them. The structure and content of such collections of data is typically highly dynamic, with the end-user serving as definer, evolver, and accessor. While a good deal of research has been conducted on techniques and mechanisms for "distributed databases" [CERI84, LIEN78, ROTH77, STON77], these approaches fail to support an environment in which multiple autonomous databases coexist, in which only partial data integration and coordination are appropriate, and in which information sharing patterns are highly dynamic [HEIM85, LYNG84a]. Of particular importance to the focus of this research is our prior work on logically distributed databases. This prior research has focused on identifying the problems involved in supporting information sharing among loosely-coupled databases and on the general architecture of a system to support sharing in such an environment. This research has introduced a specific architecture, termed "federated databases" [HEIM85], and has examined applications in the office information environment [LYNG84a]. This initial work has led to the identification and partial realization of the desired sharing capabilities which are described briefly below.

5.1. Object Sharing Functions

Information objects are distributed and inter-related/coordinated among the contexts of a network. The objects that reside at a context C1 are said to be *owned* by C1, and other objects in the network not owned by C1 are said to be *remote* to C1. *Object sharing* in a network of such inter-connected contexts involves a spectrum of capabilities. Examining these from the perspective of a given context (C1) in the network, it is possible to identify the following functions that C1 may wish to accomplish:

- Context C1 can access (examine, modify) information objects remote to C1. Such access, as well as other kinds of remote functions, is subject to access control constraints, as described below.
- Context C1 can copy or migrate to it (destructively copy) objects remote to C1.
- Context C1 can copy or migrate objects owned by it to other contexts. This may be used, for example, to propagate changes C1 has made in objects it owns to other contexts.
- Context C1 can establish, delete, or modify inter-context relationships between objects C1 owns and remote objects. Note that such inter-context relationships may potentially span many contexts.
- Context C1 can cause remote objects to be activated; this is in effect a remote procedure call on a behavioral object.
- Context C1 can determine if a local object is equivalent (according to some equivalence criteria) to remote object(s).
- Context C1 can find (determine the existence and location of) remote objects that satisfy some selection predicate; this predicate may simply be an object name or it may describe remote object(s) in terms of their properties.
- Context C1 can specify a constraint, which expresses a predicate that must hold true. A constraint can involve local and remote objects, and can include a specification of a (behavioral) object that is to be invoked if the constraint is violated at a time when it should be satisfied.
- Context C1 can selectively permit other contexts to perform the above kinds of functions on its local objects. It may also be desirable to allow optionally the receiver of an access right to in turn pass that right to other contexts.

5.2. Required Object Sharing Mechanisms

In order to support the above spectrum of object sharing functions, several object sharing mechanisms must be supported. These mechanisms are described briefly below, with an aim toward indicating our approach to their realization:

- An inter-context communication mechanism is required to allow messages specifying the sharing functions described above to be directed from a context C1 to other specific contexts, to a specific set of contexts, and broadcast to all other contexts. For this purpose, contexts are assigned a *context-id*, which is unique with respect to all contexts in the network. A response acknowledging the success or failure of a requested function along with returned information is provided.
- An object naming technique is required. Within a given context, each object has a unique, internal (system-generated) *object-id*. An object can also have one or more user-specified (or system generated) *object-names*, which are strings that uniquely identify objects within a given context. The combination of a context-id and an object-name provides a network-wide unique reference to a particular object.
- A mechanism for object classification and interrelation is needed; this is provided by a direct extension of the single context model described above.
- It must be possible to specify the scope of an object. Since objects are inter-connected by what amounts to a graph of inter-relationships, it is necessary to delimit an *object unit* as a (potentially sharable) package.
- It must be possible to determine if two or more objects are *(relative) equivalent*, with respect to some criteria. This issue is, of course, an extremely complex one since objects can be equivalent at a given level of abstraction and not equivalent at another; the presence of behavioral objects further complicates the problem, with, for example, the notion of versions relating to relative equivalence. A related issue is that of an object *copy*. Equivalence of objects can be based upon equal object-names, upon equivalence of object units, or upon equivalence of behavior. In any case, it is minimally necessary for users to specify when objects are (relative) equivalent at some level of abstraction.
- A mechanism to support (dynamic) object ownership is required. For example, objects created by a context C1 are initially owned by C1; C1 may later transfer ownership of such objects to other contexts by migrating the objects.
- A mechanism is required to check and enforce inter-context consistency constraints. There is a spectrum of consistency that is desirable, ranging from ensuring that multiple copies of objects are identical, to maintaining global constraints among objects. A complex aspect of this problem concerns the propagation of behavioral effects of constraint violation action.
- A mechanism to support and enforce access control (object security) rights is needed. This mechanism allows a context to specify which other contexts are to have which kinds of access privileges to the objects it owns, and checks and enforces distributed activities for appropriate access rights.

6. CONCLUSIONS AND RESEARCH DIRECTIONS

This paper has presented a simple database model for modelling of objects and relationships in a logical network of data/knowledge bases. A single context model was described and the desired sharing capabilities of the model was discussed. A prototype implementation of DPKM is currently being developed with research emphasis in the areas of: control and coordination of the different modes of sharing; the prescriptive user interface; access control; and integrity constraints and deduction rules.

REFERENCES

[AFSA84] Afsarmanesh, H. and McLeod, D., "A framework for semantic database models", *Proc. NYU Symposium on New Directions for Database Systems*, New York University, New York, NY, 1984.

[AFSA85] Afsarmanesh, H., Knapp, D., McLeod, D., and Parker, A., "An extensible, object-oriented approach to databases for VLSI/CAD", *Proc. 11th Int. Conf. on Very Large Data Bases*, 1985, pp. 13-24.

[AHAD85] Ahad, R. and McLeod, D., *An Approach to Semi-Automatic Physical Database Design and Evolution for Personal Information Systems*, Tech. Rep., Comp. Res. Inst., Univ. of Southern California, Los Angeles, CA, 1985.

[BATO85] Batory, D. and Kim, W., "Modelling concepts for VLSI CAD objects", *ACM Trans. on Database Systems* **10**(3), 1985, pp. 322-346.

[BORG85] Borgida, A. and Williamson, K., "Accommodating exceptions in databases and refining the schema by learning from them", *Proc. 11th Int. Conf. on Very Large Data Bases*, 1985, pp. 72-81.

[BROD84] Brodie, M.L., Mylopoulos, J., and Schmidt, J.W., eds., *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, Springer-Verlag, Berlin, 1984.

[CERI84] Ceri, S. and Pelagatti, G., *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, 1984.

[EAST80] Eastman, C.M., "System facilities for CAD databases", *Proc. 17th Design Automation Conf.*, 1980, pp. 50-56.

[GIBB83] Gibbs, S. and Tsichritzis, D., "A data modelling approach for office information systems", *ACM Trans. on Office Information Systems* **1**(4), 1983, pp. 299-319.

[HEIM85] Heimbigner, D. and McLeod, D., "A federated architecture for information systems", *ACM Trans. on Office Information Systems* **3**(3), 1985, pp. 253-278.

[KATZ82] Katz, R., "A database approach for managing VLSI design data", *Proc. 19th Design Automation Conf.*, 1982, pp. 274-282.

[KENT79] Kent, W., "Limitations of record-oriented information models", *ACM Trans. on Database Systems* **4**(1), 1979, pp. 107-131.

[KERS84] Kerschberg, L., ed., *Proc. IEEE 1st Int. Workshop on Expert Database Systems*, 1984.

[LIEN78] Lien, Y.E. and Ying, J.H., "Design of a distributed entity-relationship database system", *Proc. IEEE Int. Computer Software and Applications Conf.*, 1978, pp. 277-282.

[LYNG84a] Lyngbaek, P. and McLeod, D., "Object sharing in distributed information systems", *ACM Trans. on Office Information Systems* **2**(2), 1984, pp. 96-122.

[LYNG84b] Lyngbaek, P. and McLeod, D., "A personal data manager", *Proc. 10th Int. Conf. on Very Large Data Bases*, 1984.

[ROTH77] Rothnie, J.B., Jr. and Goodman, N., "A survey of research and development in distributed database management", *Proc. 2nd Int. Conf. on Very Large Data Bases*, 1977, pp. 48-62.

[STON77] Stonebraker, M.R. and Neuhold, E., "A distributed database version of INGRES", *Proc. Berkeley Workshop on Distributed Data Management and Computer Networks*, 1977, pp. 19-36.

[TSIC82] Tsichritzis, D.C. and Lochovsky, F.H., *Data Models*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

Advance Registration

Fifth Symposium on Reliability in Distributed Software and Database Systems

January 13-15, 1986
Los Angeles Mariott Hotel, California
SPONSOR—IEEE Computer Society

Technical Committee on Distributed Processing
and
Technical Committee on Fault-Tolerant Computing
in cooperation with the ACM
and IFIP Working Group 10.4

The theme of this symposium is reliability in distributed systems, including distributed applications, distributed operating systems, and distributed databases.

TOPICS TO BE COVERED

- **Reliable Distributed Software Systems**
  - Protocols for reliable distributed computing
  - Techniques for non-stop operations
  - Decentralized control
  - Software fault tolerance
  - Distributed operating systems
  - Performance studies of reliability techniques
  - Security in distributed applications
- **Reliable Database Systems**
  - Integrity and consistency
  - Robust concurrency control
  - Fault tolerant distributed databases
  - Experiences with testbeds and real-world distributed databases
  - Performance studies of reliability techniques
  - Security in databases

TUTORIAL

J. K. Gallant, G. Lidor and E. N. Shipley:
The impact of DBMS on distributed systems and AI Applications

ORGANIZERS

Symposium Chairman: Herbert Hecht, SoHaR, Inc.
Administrative Chairman: Raif Yanney, TRW
Tutorial Chairman: David Cohen, Teknecon Interswitch
Program Committee:
Algirdas Avizienis, UCLA (Co-Chairman)
Ronald Rutledge, US Dept. of Transportation (Co-Chairman)
Jean-Serge Banino, INRIA, Le Chesnay, France
Bharat Bhargava, Purdue University
Flaviu Cristian, IBM Research, San Jose, CA
Mary C. Chruscicki, IITRI, Rome, NY
Yves Deswarte, LAAS, Toulouse, France
Edwin C. Foudriat, NASA Langley Research Center
David Gelernter, Yale University
Jack Goldberg, SRI International
Per Gunningberg, Uppsala University, Sweden
John P. J. Kelly, ORDAIN, Inc., Torrance, CA
K. H. Kim, University of South Florida
Ming T. Liu, Ohio State University
Nancy Lynch, MIT
John F. Meyer, University of Michigan
David W. Mizell, Office of Naval Research
Radu Popescu-Zeletin, Hahn-Meitner Institute, W. Germany
Lorenzo Strigini, IEI-CNR, Pisa, Italy

INVITED DISTINGUISHED SPEAKERS:

Jim Gray, Tandem:
Why do computers fail? What can be done about it?
H. Garcia-Molina, Princeton U:
Replicated data management
Ray Strong, IBM:
Problems in fault-tolerant distributed systems
Gerard LeLann, INRIA:
Issues in fault-tolerant real time LAN

ADVANCE REGISTRATION FORM

Send this form and check (payable to Symposium on RDSDS) to:
IEEE Computer Society
1730 Massachusetts Avenue, N.W.
Washington, D.C. 20036-1903

	Preregistration			*After Dec. 23, 1985*	
	Member	*Nonmember*	*Student*	*Member*	*Nonmember*
Tutorial	$110	$140	$15	$125	$160
Tech program	$110	$140	$15	$130	$160

Please circle one of the above catagories.

Name: ______________________

Company: ______________________

Company Address: ______________________

____

Home Phone ( ) __________ Bus. Phone ( ) __________

IEEE or ACM Member Yes______ No______ If yes, Member No.______

Hotel reservation must be made by December 27, 1985.
Los Angeles Mariott Hotel
5855 West Century Blvd.
Los Angeles, CA 90045
(213) 641-5700
Special conference rate:
Single $80
Double $95

PRELIMINARY CALL FOR PAPERS AND PARTICIPATION

THIRD INTERNATIONAL WORKSHOP ON STATISTICAL AND SCIENTIFIC DATABASE MANAGEMENT

22nd – 24th July 1986, Grand-Duchy of Luxembourg

Sponsored by :

The Government of Luxembourg
Centre for Population, Poverty and Policy Studies

Commission of the European Communities
Statistical Office (EUROSTAT)
Task Force for Information Technologies

Lawrence Berkeley Laboratory
University of California

Supported by :

CR International A/S, Denmark
ESRC Centre in Economic Computing, England

In cooperation with (requested) :

International Association for Statistical Computing (IASC)
Association of Computing Machinery
Special Interest Group on Management of Data (ACM-SIGMOD)

General Chairman :

Roger CUBITT
Statistical Office of the European Communities

Workshop programme :

The purpose of this limited attendance workshop is to bring together *theoreticians, researchers and practitioners* in the field of statistical and scientific database management to discuss current work and problems. There will be a variety of paper presentations, panel discussions and plenary sessions, as well as time for informal exchange of information. The workshop will be preceeded by a one day tutorial programme which will aim to define the context of the workshop itself.

Topics :

Like its predecessors, this workshop seeks to identify and address research and implementation issues including, but not limited to, the following areas:

Applications : - scientific experiments, medical data, economic data, telemetry data, data analysis, statistical use of transaction and business data, material and substances applications, expert systems applied to statistical and scientific data.

User Interfaces : - languages, Software tools interactive requirements, graphics, use of intelligent workstations.

Storage and access : - data structures, compression methods, security and privacy, distributed databases, analysis management.

Meta Data : - *conceptual models, schema definition, data dictionaries,* self-describing files, data integrity and quality.

Hardware : - database machines, storage technology, display devices, distributed architecture.

Participation by invitation :

The programme committee will invite 50 to 100 people to participate in the workshop on the basis of written proposals for presentations. Presentations will include papers of up to 5000 words and extended abstracts (including reports on research in progress) of up to *2000 words. All contributions should be submitted in English and must* include a 100 word abstract. Send five (5) final copies by 1st February 1986 to either :

- *for the American continent :*

Prof. Gultekin OZSOYOGLU
Department of Computer Engineering and Science
Case Western Reserve University
CLEVELAND, Ohio 44106

- for all other countries :

Dr B. E. COOPER
European programme chairman
ESRC Centre in Economic Computing
The London School of Economics and Political Science
Houghton Street
LONDON WC2A 2AE

Authors will be notified of acceptance by 14th April 1986.

Location and Accommodation :

The workshop will be held at the site of the Luxembourg Study Centre *just outside Luxembourg city. Luxembourg airport* is about 20 minutes away with regular flights to a number of European cities and the U.S. The cost will be approximately $200 (12000 Luxembourg Francs) per person for the workshop including all documentation, meals, a workshop reception and dinner. The tutorial cost will be approximately $50 (3500 Luxembourg Francs) per person including midday meal. A limited amount of student style accommodation is available via the Centre at very reasonable rates; this will be allocated on a first come first served basis. Additional hotel accommodation will be arranged at a reduced rate as required.

Important dates :

Submission Deadline	:	1st February 1986
Acceptance Notification	:	14th April 1986
Final Version due	:	31st May 1986
Tutorial	:	21st July 1986
Workshop	:	22nd - 24th July 1986.

RESPONSE SLIP

To: Roger CUBITT, General Chairman,
Unit for Data Processing Management
EUROSTAT
B.P. 1907
L-1019 LUXEMBOURG
EUROPE

Please send me registration materials for the Third International Workshop on Statistical and Scientific Database Management.

Name: .... Telephone: ....

Organisation: ....

Address: ....

City, State, Post Code,
and Country: ....

Please indicate all of the following that apply:

- I intend to submit a paper (up to 5000 words)
- I intend to submit an extended abstract (up to 2000 words)
- I would like to help organise a panel discussion
- I am not sure I can participate but please keep me informed.

Subject of paper or abstract: ....

CALL FOR PAPERS

12th International Conference on Very Large Data Bases

KYOTO, JAPAN

August 25-28, 1986

THE CONFERENCE

VLDB Conferences are intended to identify and encourage research, development and applications of database systems. The Twelfth VLDB Conference will bring together researchers and practitioners to exchange ideas. We are eager for papers on new concepts, new ideas and new research results having to do with databases and knowledge bases. We not only solicit, but seek and encourage, papers describing work in which an implemented system embodies a new concept. All submitted papers will be read by the Program Committee.

TOPICS

Major topics of interest include, but are not limited to:

Data Models
Database Theory
Database Design Methodology and Tools
Distributed Databases
Query Optimization
Concurrency Control
User Interfaces
Database Hardware
Data Organization
Performance
Security Integration of Logic and Database
Knowledge-Base System
Object-Model Representation
Engineering Databases
Office Information Systems
Multi-media Databases

SPONSORS:
Very Large Data Base Endowment
IFIP
INRIA
Information Processing Society of Japan

TO SUBMIT YOUR PAPERS

Five copies of double-spaced manuscript in English up to 5000 words should be submitted by February 15, 1986 to one of the Program Committee Chairpersons.

Setsuo Ohsuga
University of Tokyo
4-6-1, Komaba, Meguro-ku
Tokyo 153
Japan

Wesley Chu
Computer Science Dept.
UCLA
Los Angeles, CA 90024
USA

Georges Gardarin
INRIA
Domaine de Voluceau Rocquencourt
B.P. 105-78153 Le Chesnay Cedex
France

IMPORTANT DATES

PAPERS DUE:	February 15, 1986
NOTIFICATION OF ACCEPTANCE:	April 30, 1986
CAMERA READY COPIES DUE:	May 30, 1986

CALL FOR PAPERS

ER APPROACH

The 5th International Conference on ENTITY-RELATIONSHIP APPROACH

November 17-19, 1986 Dijon, France

Organized by **Afcet** (France)
with the requested cooperation of ACM and IEEE Computer society

Conference Chairman
François Bodart
University of Namur, Belgium
Program Committee Chairman
Stefàno Spaccapietra
University of Dijon, France
Tutorial Chairman
André Flory
University of Lyon, France
Organizing Committee Chairman
Yves Tabourier
Gamma International, France
American Coordinator
Adarsh K. Arora
Gould Inc., USA

Program Committee

Adarsh K. Arora	USA
Carlo Batini	Italy
Mokrane Bouzeghoub	France
Janis Bubenko	Sweden
Alejandro Buchmann	Mexico
Alfonso F. Cardenas	USA
S. Misbah Deen	UK
Barbara Demo	Italy
Ramez A. Elmasri	USA
James P. Fry	USA
Antonio Furtado	Brazil
Igor T. Hawryszkiewyiz	Australia
Hannu Kangassalo	Finland
Leslie Hazelton	USA
Peter J.H. King	UK
Isamu Kobayashi	Japan
Michel Leonard	Switzerland
Tok Wang Ling	Singapore
Fred Lochovsky	Canada
Sal March	USA
Bernard Moulin	Canada
Erich Neuhold	Austria
Antoni Olivé	Spain
Christine Parent	France
Alain Pirotte	Belgium
Colette Rolland	France
Hirotaka Sakai	Japan
Hans J. Schek	West Germany
Gunter Schlageter	West Germany
Amilcar Sernadas	Portugal
Arne Solvberg	Norway
John F. Sowa	USA
Kazimierz Subieta	Poland
Yves Tabourier	France
Hubert Tardieu	France
Reino P. van de Riet	Netherland
Herbert Weber	West Germany

Major Theme : Ten years of experience in ER modelling

ER Conferences are intended to identify and encourage research, development and applications of database and information systems based on the use of the entity-relationship approach. Ten years after Peter Chen's original paper in TODS first issue, this conference wishes to offer a checkpoint on the usability of the ER approach for the design process and on its use as an operational tool, as well as an insight into the theory of the ER model and into development perspectives

Major topics of interest include, but are not limited to :

* database and information systems design
* data models and data modelling techniques
* data manipulation languages and user interfaces
* database dynamics and integrity
* formal definitions within the ER approach
* significant applications, experiments and implementations
* multi-media databases
* knowledge-based systems
* extensions for special purpose systems (OIS, engineering DB, CAD/CAM,...)
* experiences in training and teaching

Submission of papers :

* five copies of original double-spaced manuscript in English, up to 5000 words, should reach by **March 20, 1986** the Program Committee Chairman
  Prof Stefano Spaccapietra
  Université de Bourgogne - IUT
  B. P 510
  21014 Dijon Cedex
  France
* a one page document (including name, address and affiliation of authors, title of the paper, short abstract and keywords) should be sent to the same address by **March 1st, 1986**, to help in the allocation to referees
* notification of acceptance or rejection will be sent to authors by **June 20, 1986**
* for inclusion in the conference proceedings, the final camera-ready copy must be received by the Program Committee Chairman by **August 20, 1986**

For further information: Stefano Spaccapietra, France, tel 80664611
Adarsh K. Arora, USA, tel (312) 640-4712

The conference will be held at the Palais des Congrès (Conference Hall) in Dijon. Dijon is a delightful old city, located 300 km south of Paris, from which it may be reached in 1h40 using the TGV, the fastest train in the world. Capital of the Burgundy region, Dijon is famous for its history, but also for its gastronomy : mustard, gingerbread, black-currant liqueur, snails, .. and of course the marvelous red and white Burgundy wines Also, plan to attend the very famous auction sale of the wines of the Hospices de Beaune, 35 km from Dijon, to be held on November 18, 1986

IMPORTANT DATES

PAPERS DUE :	MARCH 20, 1986
NOTIFICATION OF ACCEPTANCE :	JUNE 20, 1986
CAMERA READY COPIES DUE :	AUGUST 20, 1986

IEEE COMPUTER SOCIETY

Administrative Office

1730 Massachusetts Ave., N.W.
Washington, D.C. 20036–1903
U.S.A.

Non-profit
Organization
U.S. Postage
Paid
Silver Spring, MD
Permit No. 1398