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.