An Object-Oriented Environment for OIS Applications

O.M. Nierstrasz
D.C. Tsichritzis

Institute of Computer Science
Research Centre of Crete

ABSTRACT

Object-oriented programming environments are increasingly needed for programming OIS applications. A prototype object-oriented language has been implemented, and we are refining the language and its implementation. The environment integrates a number of database and operating system concepts, in particular, abstract data types, database constraints, atomic transactions, data persistency, triggering of events, reliability and crash recovery, and a large virtual memory. We outline the object model, discuss a number of implementation issues, and give some examples of objects useful in an OIS application environment.

1. Introduction

Object-oriented data-modeling and programming techniques have become widely publicized in recent years. We believe these techniques to be especially important to the area of Office Information Systems (OISs). Experience has shown us that OISs typically deal with a variety of long-term objects with an often well-defined behaviour. Office activity is generally event-driven and highly parallel [HaSi80, ElNu80, HaKu80, Morg80]. Objects such as documents, messages and forms generally have a longer lifespan than the transactions that manipulate them, and they may have non-trivial procedures for operating on their contents. Object-oriented approaches directly address many of the needs of OIS programming [ABBH84].

An object-oriented system may be thought of as a database in which the intelligence is associated with the data items, rather than with programs that manipulate them. Objects assume the responsibility for the operations that may be performed upon them. Nevertheless we are concerned with a number of traditional database issues. The system is responsible for enforcing atomicity of events (comparable to database transactions), specified object behaviour (database constraints), and reliable object storage. One accesses information contained in objects through the interface specified by the object's behaviour, however, rather than through a standard database interface. Arbitrary interfaces may be provided, depending upon the application.

A prototype object-oriented programming environment called *Oz* was implemented by Mooney and Twaites at the University of Toronto, and is described in their M.Sc. theses [Moon84, Twai84] and in [NiMT83]. An overview of the system and further speculation are provided in [Nier85].

We are currently refining the ideas behind Oz on three fronts. First, we are refining the concepts; this is most simply stated as "what is the most useful way to think of objects?" The concepts manifest themselves in the semantics of the language constructs. Second, we are investigating efficient ways of implementing objects. Ultimately we would need an operating system geared especially to supporting an object-oriented environment, though for experimental and developmental purposes it is desirable to use an existing operating system such as UNIX† as a base. Finally, we are looking at applications. What are examples of extremely useful objects? How would we program them with objects. These three fronts may be thought of as different levels. Though they are to some extent independent, they clearly influence one another.

In this paper we describe our thoughts in these three areas. We are presently implementing a successor to Oz, called *Son of Oz*. Oz was intended to demonstrate the workability of certain ideas, without necessarily implementing them as efficiently as possible. Son of Oz is intended to be the next step towards an object-oriented programming environment suitable for building OIS applications.

2. Concepts

The objects that we will describe here are comparable to Smalltalk objects [GoRo83, Gold84], Actors [Hewi77, Ther83]], monitors [Hoar74], abstract data types [Gutt77], and modules with persistent data. The word "object" is usually used to describe an entity that encapsulates some data and the allowable operations on it. Smalltalk objects, for example, have a collection of instance variables that contain data, and a number of ***methods*** for manipulating the data.

The key difference between our object model and others in the literature is that ***events*** are automatically triggered when the desired preconditions are met. Events are simply the units of action (atomic transactions) within which participating objects may change state. Since the state changes are visible after events take place, we obtain a powerful event-driven model of computation. (For a different view of atomic transactions with stable storage of objects, the reader is directed towards [LiSc83, Oki83].)

The object database is maintained by the system, giving the illusion that all objects reside in a large, persistent virtual memory. Furthermore, objects may have quite complex operations associated with them. Consequently, rather than having programs that interact with a database management system, we simply have objects interacting with objects, all within the same, homogeneous system.

Since there are many workable notions of "objects" in the literature, we shall first present a brief overview of our model, and then describe some of its features in greater detail, with examples.

2.1. Overview

An ***object*** is an entity with ***contents*** and ***behaviour***. Objects fall into ***classes*** with the same generic specification. Object ***instances*** are distinguished by having (possibly) different contents and unique object identifiers (***oids***). (Oid is a unique identifier for addressing an object at any time, in any place, no matter what happens to the object during its lifetime.) Object classes in the real world would be, for example, ***books***, ***people***, ***bicycle tires***, and so on. Contents could be properties such as ***colour*** and ***weight***, or parts such as ***front derailleurs***. In the latter case, we have complex objects (like bicycles) made up of other objects. Whereas contents constitute the internals of an object, behaviour constitutes its external appearance. It tells how an object may change state and interact with other objects. Books may be opened to an arbitrary page and they may be read, but they can't be pumped up with air (usually).

An object's behaviour is extremely important because it is the mechanism through which it controls its interactions with all other objects. Object creation, destruction and all updates take place under the constraints of the rules specified in the behaviour. In particular, this means that objects cannot be ***forged***. That is, one cannot create an object that masquerades as one of a known class. An object that claims to be of a given class is therefore guaranteed ***by the system*** to obey the behaviour one expects of it.

Every object class has a ***specification*** which describes the contents and behaviour of instances of the class. The specification is not necessarily an object itself, but may be ***represented*** as an object -- a ***text*** object, for example. We will describe the contents and behaviour portions in turn.

Figure 1 shows a partial specification for an imaginary ***percolator*** object. Percolators are ***appliances*** with a certain ***capacity***, that can hold a certain amount of ***grounds***, ***water***, and ***coffee***. They may be in the process of ***brewing***, and when the coffee is ready, there is a ***light_on*** condition. To make coffee, a number of conditions must hold first, for

† UNIX is a trademark of AT&T Bell Laboratories.

example, the percolator must be clean and empty. To tell whether the coffee is ready, there is a *light_on()* rule which must be invoked, since the instance variable *light_on* is internal to the object, and therefore not directly visible from the outside. The only legitimate interface to an object's contents is through its behaviour.

```
percolator : appliance {
    /* instance variables */
    capacity : integer; /* in cups */
    grounds, water, coffee : integer;
    brewing, light_on : boolean;

    /* rules */
    alpha(cups) { /* rule for creation */
        /* creator */
        ~ : hacker;
        cups : integer;

        /* a trigger condition: */
        /* you can't have two pots */
        ~.pot() = nil;
        capacity := cups;
        }

    make_coffee(cups) {
        cups : integer;

        /* trigger conditions */
        /* you want fresh coffee, right? */
        cups <= capacity;
        grounds = 0;
        water = 0;
        coffee = 0;
        grounds := cups;
        water := cups;
        brewing := TRUE;
        light_on := FALSE;
        }

    /* this rule just tells you if the light's on */
    light_on(){
        }(light_on)

    ...
    /* other rules */
    }
```

Figure 1 : A *percolator* object class

2.2. Contents

We call the data portion of an object its *contents*. Data in an object world typically consists of other objects. Ultimately all objects are made up of ***simple objects*** such as *integers* and *strings*. The contents of the *percolator* object are all simple objects. The contents of an object are stored in *instance variables*. Instance variables are not objects themselves, just a place to put them. In the implementation, an instance variable declaration allocates space for objects of a certain class, just as variable declarations do in programming languages like C or Pascal. Instance variables are so named because they store the data that distinguish one object instance from another.

Objects that are part of the contents of some other object are *dependent objects*. Such an object is called the *child* of its containing *parent* object. *Independent* objects have no parents. According to the specification of the objects involved, the death of a parent may or may not precipitate the death of its children. A dependent object whose parent dies would have to be adopted by another parent, or possible attain independent status. Simple objects, like integers, are always dependent, and are destroyed with their parents.

Objects may communicate with each other if they are *acquainted*. Objects are acquainted if they are directly related (parent and child), or if one is in possession of the oid of the other. By convention, whenever an object makes use of another object's oid, its own oid becomes available to the second object. Acquaintance is therefore a reflexive relationship within any interaction, as one would expect.

In *Oz* there was no need for objects to be explicitly acquainted. One would simply specify the class of the objects one wished to have as acquaintances and some trigger conditions to limit the selection. The trigger condition functioned as a query to select the appropriate acquaintance. In *Son of Oz* we expect objects to maintain a set of oids for acquaintances. One can still, as before, select acquaintances from a list, but one must be properly introduced to acquaintances before this can be done.

In addition, oids may be thought of as capabilities, since, being objects, they cannot be forged. The system protects against irregular creation of objects. One could not, therefore, simply generate oids for all objects in the system and attempt to initiate events with them. Objects may exploit this feature by being careful whom they are introduced to.

Since there is an overhead associated with oids, we can allow certain simple objects to do without them. Such objects could only be directly accessed by their parents. Integers, characters and oids themselves are obvious choices for oid-less objects.

2.3. Behaviour

An object's behaviour describes precisely the circumstances under which an object may do something. "Doing something" means changing state or causing some other object to change state. This must happen within an event (discussed below). An object changes state when it is created or destroyed, or when its contents are altered in some way.

The behaviour of an object may be specified by a set of named ***rules*** that describe what happens, and under what circumstances it may happen. If you try to do something inappropriate to an object (i.e. invoke a non-existent rule) then it will simply ignore you.

A rule invocation looks very much like a procedure call, and, in fact, rules themselves look very much like procedures, with instance variables as static data. Every object class has ***alpha*** and ***omega*** rules for the creation and destruction of instances.

An *event* takes place when several objects enter into a mutually satisfactory contract, exchange information and change state. The nature of the contract and the side-effects is completely specified by the rules of the participating objects (***participants***). The execution of an event is very similar to the execution of a program, with one rule, a ***top-rule***, invoking sub-rules of other objects, passing objects as arguments and receiving other objects in return. There are several important differences, however.

The execution of an event is atomic -- if any of the participants is not satisfied, then the event will not take place. Instead, the event *waits* until something happens to allow it to continue. Usually, the unsatisfied object, or the participant it was not satisfied with, must take part in some other event. If the side-effects of the other event cause the unsatisfied object to "change its mind", then the waiting event may continue. Alternatively, if one of the participants is destroyed in another event, then the waiting event can never continue, and it may be cancelled.

In the example, if a hacker tries to make coffee in a dirty pot, the pot will wait until it is cleaned. This is presumably accomplished by some other rule.

Events are guaranteed to be serializable, that is, the participants have read and write locks placed on them during the event to prevent parallel events from putting the objects in an inconsistent state. In this way events are similar to database transactions. Waiting events, however, may need to rollback actions and release locks if a participant is required for some other event. This is necessary since a waiting event is effectively a stalemate between its participants. Some participant must back-out, change state in another event, and resolve the stalemate in the waiting event. It is also a fairness requirement, since there is no guarantee that the awaited condition will ever be met, and the locks released.

Top-rules are similar to sub-rules except that they are not explicitly invoked. Instead, they initiate an event when their trigger conditions become true. This is always a consequence of a side-effect of some other event, so one may think of events triggering one another in a chain-reaction.

Triggering is especially useful for being notified of side-effects in other objects. A ***hacker*** object, for example, can be notified when coffee is ready by including the rule in figure 2.

```
coffee_ready {
        pot.light_on();
        /* notification statements ... */
        }
```

Figure 2 : A top-rule

Rules may send and receive lists of objects as arguments and return values, as is the case with procedure calls. Only top-rules never take arguments or return values. Subrules are passed the oid and class of their invoking acquaintance. The rule may insist that only objects in certain classes may invoke it, or that only certain known acquaintances are acceptable, or even that the rule is to be "private" and only invokable by other local rules. The oid may also be used to query the invoker. This might be done in order to perform authentication procedures, for example.

Rules may also have a set of ***temporary variables*** for manipulating arguments, computing return values, and so on. Temporary variables, like instance variables, are used to hold objects, such as the arguments to a rule, or newly-created objects, but their storage disappears when the rule finishes. As a consequence, the objects they hold must be either destroyed or given a new home before the rule completes. For convenience' sake, certain objects, like integers, may agree to disappear quietly when their storage is deallocated. Note that temporary variables, like instance variables, initially contain no objects at all, and must be filled before they can be used.

Rules are invoked by specifying an object to be addressed, the name of the rule, a list of objects to be sent as arguments, and a list of variables or locations for the returned objects to be stored. One would be able to invoke simple unary and binary rules as though they were operations. For example, one would be able to write "z := x + 1" instead of "z := x.add(1)". A similar approach is taken in Smalltalk.

Failure of a rule within an event need not always cause the event to wait. Instead, the failure may be used to affect control-flow, causing the parent event to seek alternatives. Success or failure of a rule would decide, for example, which branch of an if-then-else to execute next. The only important difference between control-flow here and in a procedure-oriented language, is that if the sub-rule fails, then it can cause no side-effects.

2.4. Other Issues

What we have described above is a very basic object model. There are a number of other issues that are important if an object programming language is to be useful. One such issue is the selection of simple objects available. A tentative list would include integers, floating point numbers, booleans, characters, strings, lists, and arrays. A variety of complex objects, such as oids, object specifications, and compiled object specifications should also be available.

Specialization of objects should be provided along the lines of Taxis [GrMy83]. A specialized object would have at least the contents and behaviour of its superclass, and possibly more. In addition, one may further restrict the rules and contents by respectively adding trigger conditions and by specializing the classes of the instance variables. One must take care that specialized objects do not violate the constraints imposed by the superclass. This may be done by making the instance variables of the superclass private to the original rules. New rules belonging to the specialized objects could then only access the original variables through the old, possibly restricted, rules. This would prevent, say, a programmer from creating specialized oids that could alter the unique identifier they contain. Such an object would not be an oid, but something entirely new.

An issue that has not been addressed in Oz, is flow control within activities. Events, being atomic, will typically be very low-level transactions within more complex activities. Editing a document, for example, may be viewed as a long-term activity composed of a sequence of short, atomic edit events. Programmers will need ways of expressing the possible flows from rule to rule as concisely as possible. The mechanisms that exist so far for describing flow between rules is quite primitive: an instance variable could be used as a "program counter", and rules could set the counter to indicate what rule fires next. An explicit means of indicating flow would also simplify the implementation, since the system could tell immediately which rule to try next.

One obvious construct would be to provide for "programs" of unnamed top-rules, each with a single "main" entry point, where the rules trigger each other in sequence. Rules, then, would function as atomic statements in such a program.

These, and other language issues, are still under investigation.

3. Implementation Issues

The object-oriented approach is a synthesis of many existing ideas, rather than an entirely original creation. In building an object-oriented system we are concerned with many of the same issues that arise when writing an operating system, a database system, a compiler, and so on. There are consequently very few truly new implementation issues here. A wealth of solutions already exist for many of the problems we have. The one difference, perhaps, is that we are concerned with familiar issues at slightly different levels. We provide atomicity of events and stable object storage at the system level rather than the application level. The system need only provide a suitable object environment, not a user environment. The user environment is provided by the object classes that happen to be defined. The system, for example, provides security mechanisms by guaranteeing object integrity through atomic events and privacy of an object's contents. The actual security policies are provided by the objects that exist. We are therefore still concerned with the same old issues, but we must consider them in a different order.

The virtual machine that we provide must give the illusion that all objects are present at any time, and that they are always ready to spring into action according to their prescribed behaviour. Objects must be reliably backed-up onto stable storage, so that a consistent state of the system is recoverable after a disaster (a system crash, power failure, etc.). In order to accomplish these tasks, we require the aid of a number of system objects:

The *event manager* is responsible for scheduling and executing events. Object rules may be compiled or interpreted. In either case, the event manager must keep track of the busy and sleeping objects. When events take place, it must determine whether other events must be run, or whether waiting events must be restarted.

The *object manager* is responsible for reliably backing-up objects, for retrieving them from stable storage, and for keeping track of oids. It should especially provide cheap, rapid access to objects that are already in memory.

The *memory manager* is an object that owns all of virtual memory. Memory is carved up for creating new objects and for executing events. The space is used for instance variables and temporary variables. Object rules, whether compiled or interpreted, are stored as objects too (just as executable programs in other domains are stored as files). It is also responsible for cleaning up deallocated memory.

Aside from these three system objects, it will be necessary to have a number of "device driver" objects for talking to the outside world. "The outside world" includes not only i/o devices but also network connections to other object-oriented systems. Objects are allowed to migrate to other machines that are willing to run them.

Since many of our implementation issues are well-understood problems, we shall not go into too many details here. Rather, we shall concentrate on a few of the problems that are peculiar to objects.

3.1. Object Management and Stable Storage

For events to be serializable and atomic, we must be able to store objects reliably and update the stable versions atomically. Atomic transactions are a familiar notion in database management systems, and we can use similar techniques here. Reliable storage in a similar context is also discussed in [Moss81, Verh78]. We will briefly describe a scheme for managing stable objects.

We first allow ourselves to think of two versions of the object database. The first is the ***stable version*** reflected in stable storage. We assume that stable storage itself is reliable (or rather, we do not concern ourselves with the details at this level). There is a second version, the *current version* of the object database, which resides in virtual memory. The current version only contains a number of *busy* objects which are currently of interest. In the event of a crash, the swap area of stable storage is naturally discarded, and the system uses the stable version of the object database as the last consistent state.

The object manager knows which objects are busy and which are not. Oids for busy objects can be immediately resolved to memory pointers. *Dormant* objects, which exist only in stable storage, must be activated by bringing them into virtual memory. Events may be *running*, *blocked*, *waiting* or *committed*. The objects participating in an event are all busy objects in virtual memory. Changes to objects caused by a running event are made to the busy versions of the object only.

In addition, the side-effects are stored separately in an *event log* for that event. As long as the event is running, the busy objects are locked. Objects not actually altered by an event will only need read locks. The type of locks required can be determined at compile time. If the event is waiting, it may be cancelled, and the side-effects rolled-back. Side-effects should be stored incrementally, so that it will be possible to partially roll-back an event to a desired point. Rather than actually un-doing actions, the simplest way to roll-back time is to restore copies of altered objects saved before the changes were made.

Until the event commits, the altered busy objects are not formally part of the current version of the object database. If the event commits, then the event log is committed, and the locks are released. At this point, the side-effects are part of the current object database. A queue of current logs may be maintained as events commit. Any unlocked busy object can thus be recovered from the stable version plus the current side-effects.

The next order of business is to update the stable version of the object database. We accomplish this by partitioning the stable version into two parts. The first is the ***stable object repository***, and the second part is the ***stable event log queue***. Current event logs must be written out atomically to the stable event log queue. (An atomic write of a single event log is not too difficult, if the hardware will support single atomic block writes.) Once an event log is safely written out, it is no longer a current log, but a stable log. One may then begin updating the objects in the stable object repository. If a crash occurs at any point, recovery begins by updating the stable objects from the logs in the stable event log queue. The updates already made before the crash are simply redundantly performed. The old current logs are lost, and execution continues from the stable version.

Note that the event logs are intended primarily as a recovery mechanism, and need not normally be used for updating objects. Instead one might use the busy version of an object in virtual memory to re-write the stable version (provided that there are no other pending current side-effects on that busy object).

Writing out current logs and performing the actual updates can be done when the processor is otherwise idle. Only the order of updates needs to be preserved. In addition, it is not strictly necessary to perform all the updates that occur in the events. Objects that experience a burst of activity could be updated when the activity dies down. Only the last current version needs to be written out. This corresponds to merging a sequence of current logs together before writing out a single stable log.

3.2. Event Searching

Events are atomic, and take place between a collection of participating objects. The participants must enter into a mutually agreeable contract before the event commits. No side-effects are visible unless the event commits.

It is easier to see how one might implement event searching and execution by using an analogy. Consider that events, while they are executing, take place in a "board-room", behind closed doors. An initiator object with a top-rule starts the event. As sub-rules are successfully invoked, new participants may enter the room, but none may leave (two-phase locking is used to maintain serializability). If at any point in time some rule fails (i.e. a trigger condition fails), then negotiations come to a halt. The event does not abort, but the participants are in a stalemate.

As long as we stay behind closed doors, we cannot break the stalemate. What we need is for at least one participant to leave the room, take part in some ***other*** event that changes its state, then return and attempt to restart negotiations. Since we have, in general, no way of predicting exactly what is needed to break the stalemate, we allow all of the participants of a waiting event to leave the room and do something else. Note that no deadlock can take place, because the stalemate automatically releases -- or rather, "softens" -- the locks held.

An object that leaves a waiting event must, however, be rolled-back to the state it had before the uncommitted, waiting event was started. If it returns to that event, the event itself is rolled-back, and the returning participant re-enters in its new state. Note that waiting events are not actually rolled back until they are restarted, since we do not know in advance when a participant leaves if it will find an event that commits or not.

A clear possibility is that an object may end up in ***several*** waiting events, all corresponding to different possible futures. If all of them would effect some state change on that object, then at most one of them may eventually commit.

For each object, then, we may maintain a queue of waiting events. Every new event that it joins will eventually either wait or commit. If it waits, the event is added to the queue. If the event commits, then the object must return to each of its old waiting events, roll them back to its entry point, and re-evaluate the event from that point.

Running events, including restarted waiting events, may ***block*** if a participant is unavailable. If the desired object is part of waiting event, it is simply reclaimed, of course, but if it is part of some other running event, then the first event blocks. The first event continues when the second either commits or waits. If the second event also blocks, then the process is continued. Obviously, if an event is blocked upon a request of a participant of another blocked event, then we must check that no deadlock exists by following the chain either to a truly running event (no deadlock), or to a blocked event that we have already seen (deadlock). Deadlock can be resolved by rolling back some blocked event far enough to release participants needed elsewhere.

There is, in general, no way of knowing what may cause a waiting event to continue and commit. We do know that it is necessary (but not sufficient) for one of the participants to leave, change state, and return. The implementation could simply attempt to restart waiting events whenever participants change state elsewhere. We can, however, improve on this. We know that the participants of a waiting event are stalemated because a single trigger condition has failed. We can also trace the arguments of the failed condition to all the relevant participants of the event. For example, if the condition "x>y" fails, we can trace the computation of x and y ***within the waiting event*** to simple objects participating in that event. These objects are ***hot***, in the sense that the event can ***only*** continue if one of these hot objects is modified. All other participants are ***cold***, and can never cause the event to continue, even if they are altered in another event.

Hotness is, of course, a necessary, but not sufficient property to determine whether the modification of an object can cause a waiting event to continue An efficient implementation of event-searching would probably have to make use of (1) event analysis along the lines of hotness, or (2) explicit direction from the programmer indicating what events may trigger other events, or (3) some combination of (1) and (2).

4. OIS Applications

The objects we have described are highly structured, persistent and reliable. Objects encapsulate data and operations, and they cannot be forged. Events involving a collection of participating objects are automatically triggered, but remain atomic, requiring the approval of all participants. These properties make objects especially useful for programming OIS applications.

We shall illustrate the use of objects through a number of examples.

4.1. Roles and secure objects

Objects enable one to specify flexible security environments. If one is primarily managing documents, for example, then the documents can be stored as objects. Rather than associating documents directly with users, one may associate them with a set of ***roles*** [TsGi85]. Users play a particular role in order to perform certain functions. Different roles may be associated with the owner of a document, and the allowable readers, copiers and editors. Copies of documents may be associated with the same roles, and thus prevent information leaking "down" to unprivileged roles (this is a crude simplification of the Bell and LaPadua security model as described in [Land81]).

Roles themselves are objects, so it is possible to divorce the intelligence regarding who is allowed to play which role from the actual allowable operations on a document. A role object performs the necessary authorization checks to guarantee that a given user is allowed to play that role. The role may actually store a list of user ids, or it may keep track of what other roles may assume that role. A user playing one role may then ask to play another role. The role objects then either act as a filter for operations on owned objects (such as documents), or, more likely, they authorize users for a single event session, or activity.

Capabilities may also be represented by objects. Since objects are unforgeable, one need only specify precisely which objects (or objects playing what roles) are allowed to create new capabilities. Capabilities may be used as ***currency***, allowing an object to take part in an event or an activity a limited number of times. Capabilities may also be used as licenses to create other capabilities. One could acquire a capability when playing a certain role, and have to give it up when leaving the role.

Foreign objects arriving over a network can be filtered by an "immigration office". Foreign objects initially have no acquaintances and no capabilities. Such objects have to be explicitly sponsored by local objects who can grant them capabilities. A foreign object can then take part in some limited events with a few acquaintances it is introduced to. Alternatively, it can be accepted fully as a naturalized citizen and be granted all the capabilities of local objects.

4.2. Documents

Since objects have a permanent structure associated with them, it is somewhat easier to think of documents as being non-linear. One may exploit the fact that objects have a hierarchical decomposition when designing and implementing document types. Furthermore, since we are specifically concerned with persistent objects, there would be no need to "load in" a linear representation of a hierarchically structured document in order to use it.

There are a number of possibilities which would benefit from an object-oriented implementation. An object-oriented document model would, for example, easily support views. One may view an abbreviated document, or one may ask to see just the main semantic components and their names (chapter and section headings). Documents such as manuals could even dispense with any basic linear interpretation at all. Pointers to articles within the document can be stored at arbitrary locations, so that one may peruse the document in one's order of interest rather than in an order imposed by the author.

One would similarly be able to store several versions of a document as a single object. Different versions can then be called up as separate views. Components of the document would simply need some additional information indicating what versions of the document they apply to. Since the display of a document is not tied to its representation, this information is easily hidden.

Formatting information can also be associated with the various components of the document without having to, say, embed formatting commands in the text. It would also be possible to store the raw text and the formatted portions in the same object. One could then take advantage of the fact that small local changes generally do not affect how the rest of a document is formatted. Usually only the location of page breaks changes. The formatted segments could also be useful in the presentation of ad hoc views.

4.3. Mailing

Mail messages encoded as objects suggest a number of interesting possibilities. First of all, mail messages can package other objects, without having to translate them into ascii text, as is the case in UNIX. A mail message might simply consist of some header information, and a collection of objects to be mailed.

Who is allowed to mail what to whom may be incorporated into the operations of the roles that users are allowed to play. Users playing certain roles, for example, may only be able to mail or read specific kinds of messages. In this way, mail distribution can be controlled and limited [TsGi85].

So-called "intelligent messages", or *imessages* [Hogg85] can be implemented using objects. Messages of this type enter into a dialogue with their recipient when the user attempts to read the mail. The dialogue itself may be modified by the responses received, and the message may automatically forward itself to another user based on the information the user supplies. Such messages may be used either to distribute responsibilities or resources to a set of users, or to track down information accordingly to a dynamically modified plan.

Imessages would be instances of a single imessage object class, rather than each imessage being programmed separately as a different class. Typically, imessages have a list of questions to pose or statements to make, a repository for the responses collected, a set of variables encoding its knowledge, and an eventual return address. When received, it must initiate a dialogue. Each response is evaluated, possibly stored, and an action is taken. The action updates some variables, and causes more information to be displayed, or closes the dialogue and chooses a new recipient.

The dual notion to imessages are automatic procedures that process mail upon its arrival [HoNT85]. In this case we would expect the messages to be passive, and to accept the actions of the automatic procedures. The automatic triggering of these procedures is easily captured by the object triggers monitoring incoming messages. Filing of messages into different "trays" according to subject or sender can be handled this way. Automatic procedures may also alter the contents of messages and possibly mail responses to the original sender. A totally automatic "user" can be specified that takes actions based on the mail it receives. An example would be an inventory administrator that answers queries, accepts orders, notes arrival of new stock, and issues warnings if stock falls below some threshold.

4.4. Knowledge collection

As a final example, we consider *kno*'s [Tsic85]. A kno is cross between an imessage and a *worm program* [ShHu82]. A kno travels through an object system, or a network of object systems, collecting and processing information. Unlike an imessage, it need not be represented as a mail message, nor need it ever make its presence felt to most users. Of course, for kno's to have the power to examine information at various sites, they must be armed with the right capabilities. An unwelcome kno may be turned away (or destroyed) when it arrives at a new site.

Objects, when they die, can disappear entirely, or they may give up their contents to an archive, or database of dead objects. The information contained in the original object is then still accessible, but the former object is stripped of its behaviour. These databases of dead objects constitute one suitable domain for kno's to gather information from.

A kno has the power to sit and wait for some event to happen before it begins to do its work, or decide what to do next. Monitoring of existing objects by a foreign (but welcome) kno can be completely transparent in an event-driven object world.

Kno's may also be thought of as distributed queries. A user query may initiate a number of kno's that travel from site to site (or from database to database) collecting information. Kno's may have the ability to clone themselves, if there is a choice of possible continuations. Like imessages, kno's can travel according to fixed routing, dynamic routing, or even random routing (for sampling purposes). Kno's can automatically return to home when they have gathered a certain amount of information, or they may be "harvested" by other kno's sent out to collect them at an independently decided time. If kno's are never harvested, we may allow them to die of old age or malnutrition.

5. Conclusions

We are currently refining the Oz object model and its implementation. The successor, Son of Oz, is intended to be a general programming system capable of supporting OIS applications. Son of Oz is similar to other object-oriented languages, such as Smalltalk. The main difference is that Son of Oz provides for automatically triggered, autonomous events. The significant features are:

1. Objects provide hierarchically structured abstract data types; an object encapsulates data and the permissible operations on it.
2. Events are automatically triggered when pre-specified conditions become true. It is not necessary to explicitly initiate all events.
3. Events are atomic, being a contract between a collection of participating objects. This is the primary mechanism for ensuring the integrity and security of objects.
4. Objects are persistent data, residing in a stable and reliable virtual memory.

6. References

[ABBH84] M. Ahlsen, A. Bjornerstedt, S. Britts, C. Hulten, and L. Soderlund, "An Architecture for Object Management in OIS", *ACM Transactions on Office Information Systems*, Vol. 2(3), pp. 173-196, July 1984.

[ElNu80] C.A. Ellis and G. Nutt, "Computer Science and Office Information Systems", *ACM Computing Surveys*, Vol. 12(1), pp. 27-60, March 1980.

[Gold84] A. Goldberg, *Smalltalk 80: the Interactive Programming Environment*, Addison-Wesley, 1984.

[GoRo83] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, May 1983.

[GrMy83] S.J. Greenspan and J. Mylopoulos, "A Knowledge Representation Approach to Software Engineering: The Taxis Project", *Proceedings of the Conference of the Canadian Information Processing Society* pp. 163-174, May 1983.

[Gutt77] J. Guttag, "Abstract Data Types and the Development of Data Structures", *Communications of the ACM*, Vol. 20(6), pp. 396-404, June 1977.

[HaKu80] M. Hammer and J.S. Kunin, "Design Principles of an Office Specification Language", *Proceedings of the NCC*, pp. 541-547, 1980.

[HaSi80] M. Hammer and M. Sirbu, "What is Office Automation?", *Office Automation Conference*, Georgia, pp. 37-49, 1980.

[Hewi77] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages", *Artificial Intelligence*, Vol. 8(3), pp. 323-364, June 1977.

[Hoar74] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, Vol. 17(10), pp. 549-557, Oct 1974.

[Hogg85] J. Hogg, "Intelligent Message Systems", pp. 113-134, in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, Springer Verlag, Heidelberg, 1985.

[HoNT85] J. Hogg, O.M. Nierstrasz, and D.C. Tsichritzis, "Office Procedures", pp. 137-166, in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, Springer Verlag, Heidelberg, 1985.

[Land81] C.E. Landwehr, "Formal Models for Computer Security", *ACM Computing Surveys*, pp. 247-278, September 1981.

[LiSc83] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM TOPLAS*, Vol. 5(3), pp. 381-404, July 1983.

[Moon84] J. Mooney, "Oz: An Object-based System for Implementing Office Information Systems", M.Sc. thesis, Department of Computer Science, University of Toronto, 1984.

[Morg80] H.L. Morgan, "Research and Practice in Office Automation", *Proceedings 1980 IFIP Congress*, pp. 783-789.

[Moss81] J. Eliot B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", Ph.D. thesis, MIT/LCS/TR-260, MIT Dept EE and CS, April 1981.

[Nier85] O.M. Nierstrasz, "An Object-Oriented System", pp. 167-190, in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, Springer Verlag, Heidelberg, 1985.

[NiMT83] O.M. Nierstrasz, J. Mooney, and K.J. Twaites, "Using Objects to Implement Office Procedures", *Proceedings of the Canadian Information Processing Society Conference*, Ottawa, pp. 65-73, May 1983.

[Oki83] B.M. Oki, "Reliable Object Storage to Support Atomic Actions", M.Sc. Thesis, MIT/LCS/TR-308, MIT Dept EE and CS, May 1983.

[ShHu82] J. Shoch and J. Hupp, "The Worm Programs - Early Experience with a Distributed Computation", *Communications of the ACM*, Vol. 25(3), pp. 172-180, March 1982.

[Ther83] D.G. Therault, "Issues in the Design and Implementation of Act2", M.Sc. thesis, TR #728, MIT AI Lab, June 1983.

[TsGi85] D.C. Tsichritzis and S.J. Gibbs, Etiquette Specification in Message Systems, pp. 93-112, in *Office Automation: Concepts and Tools, ed. D.C. Tsichritzis, Springer Verlag, Heidelberg, 1985.*

[Tsic85] D.C. Tsichritzis, "Objectworld", pp. 379-398, in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, Springer Verlag, Heidelberg, 1985.

[Twai84] K.J. Twaites, "An Object-based Programming Environment for Office Information Systems", M.Sc. thesis, Department of Computer Science, University of Toronto, 1984.

[Verh78] J.S.M. Verhofstad, "Recovery Techniques for Database Systems", *ACM Computing Surveys*, Vol. 10(2), pp. 167-195, June 1978.