8. An Object-Oriented System

O.M. Nierstrasz

ABSTRACT

Applications in Office Information Systems are often very difficult to implement and prototype, largely because of the lack of appropriate programming tools. We argue here that "objects" have many of the primitives that we need for building OIS systems, and we describe an object-oriented programming system that we have developed.

1. Introduction

One of the great difficulties in implementing office information systems and prototypes for testing new OIS concepts is the unavailability of appropriate programming languages. A great deal of effort is therefore spent "re-inventing the wheel" whenever a new prototype is developed. In this paper, we discuss our efforts to address this problem by developing a simple, object-oriented programming environment. We argue that "objects" are a natural primitive for programming many OIS applications (see the companion paper, "Objectworld"). They are far more appropriate (if they can be implemented efficiently) than a high-level language such as C or Pascal.

In papers such as [HaSi80, ElNu80, HaKu80, Morg80, SSKH82], office behaviour is described as being event-driven and semi-structured. Office activities exhibit a great degree of parallelism and "bursty" behaviour, meaning that activities alternate between running and suspended states. Activities may have to coordinate several documents, or even synchronize themselves with other activities. Messages and documents are sometimes highly structured, especially in the case of forms. Typically these documents also have certain constraints and functional capabilities not generally associated with databases. Many of these issues are addressed directly by object-oriented programming [ABBH84].

Objects bear comparison to abstract data types [Gutt77], actors [Hewi77], and SBA/OBE boxes [deZl77]. Many of the properties of our object model are also exhibited by Xerox' Smalltalk system [GoRo83, Gold84]. Objects combine data and program by allowing the programmer to specify the nature of the data that the object may hold, that is, its *contents*, and also the allowable set of operations valid for those data, that is, the object's *behaviour*. The object construct therefore exhibits several "nice" properties, among them modularity, encapsulation (of data and operation), strong typing, and duration. The last is important, since objects typically have a longer lifetime than the execution of most programs. In addition, our object model allows for specialization of objects, and automatic triggering of the object's *rules* (operations) wherever appropriate. Finally, because the operations are explicitly bound to the data, an extra measure of security is achieved, without any loss of generality. The object model appears to be as powerful as more traditional machine models that separate data and program.

In the following section we shall discuss our abstract object model, and we will demonstrate some of the power of objects. The remaining sections deal with the implementation of our prototype object-oriented programming system, called *Oz*. Specifically, we discuss the user interface, the internal system design, and the details of object management.

2. The object model

In this section we shall describe in some detail what we mean by the term "object", and how it can be used as a programming tool. Specifically, we discuss the relationship between the data and the program elements of objects (called *rules*), and we explain under what circumstances the rules may be executed.

In *Office Automation: Concepts and Tools*, D. Tsichritzis (Ed.), pp. 167-190, Springer-Verlag, Heidelberg, 1985.

2.1. Object classes

Perhaps the key distinguishing characteristic of objects is encapsulation. An object, like an abstract data type, forces us to describe our data, and the operations that manipulate them, together. Once we have completed our specification of an object class, we can be certain that instances of that class will not be abused by anyone's attempt to perform invalid or inappropriate operations on them.

An object is responsible for anything that happens to it. Furthermore, in our object model, we give objects the responsibility of executing that part of their behaviour that is to be automatically triggered whenever pre-defined conditions are met. Any object can therefore become active at any time, if the right conditions arise to cause it to spring into action.

Objects are divided into *object classes*, which are comparable to the notion of types. Any object is an *instance* of some given object class. The classes are characterized by their specifications, and the instances are characterized by their values. As an analogy, we may compare object classes to database schemata, and object instances to values in the database (such as relational tuples).

Objects have both data and "program" components. We refer to these as *contents* and *behaviour*. The contents of an object can be described by a set of *instance variables*. The values of these variables will characterize any given object instance. The behaviour of an object is given by a collection of *rules*. These "rules" resemble the procedures or subroutines of a program, with the exception that there is no "main" program to call them. The rules are invoked by other objects, or *acquaintances*, that the object agrees to deal with.

Rules may contain local (temporary) variables, and executable statements that modify the instance variables, just as a subroutine might, but they may also contain a set of *triggers* or preconditions on the execution of the rule. If any one trigger condition fails, then the rule may not be executed. A common trigger condition is to restrict the allowable object classes of the object invoking the rule, as in the following example.

```
customer : office {
    /* instance variables */
    name, owner : string;

    /* rules */
    set_name (n) {
        /* invoking acquaintance */
        ~ : office;
        n : string;

        /* a trigger condition */
        ~.owner = owner;
        name := n;
        }
    }
```

Figure 1: A simple object specification

In the simple example in figure 1, we define part of a *customer* object. It is defined to be a specialization of an *office* object. *name* and *owner* are instance variables, and *set_name* is a rule. The invoking object (indicated by the special symbol "~") must be an *office* object whose owner is also the owner of the *customer* object. The *name* variable may be manipulated by other rules as well, specifically, it may be initialized at the time of creation.

2.2. Events

If a rule *b* of object *B* is invoked, then there must be an invoking rule *a* of an acquaintance *A*. Rule *b* can fire if and only if both it and rule *a* are completely satisfied, that is, all their trigger conditions are met.

For example, the *ch_name* rule in figure 2 invokes the *set_name* rule of figure 1. Both rules must be satisfied for an event to fire.

```
ch_name {
        c : customer;
        m : memo;

        m.creator = "legal";
        m.oldname = c.name;
        c.set_name(m.newname);
        m.omega;
        }
```

Figure 2: an invoking rule

Furthermore, rules *a* and *b* may invoke other rules in yet other objects. All of these rules must be satisfied before any of them may execute. This is what we call an *event*. If any rule participating in an event has a trigger condition that fails, then the event fails. If all the rules are satisfied, then the event may *fire*, and all rules participating in the event are executed.

A rule is allowed to invoke itself. The trigger conditions within such a rule then monitor instance variables or an acquaintance. The *ch_name* rule in figure 2 is self-triggering, and monitors the arrival (creation) of a *memo* object from the *legal* department, indicating a change-of-name. In this example, the *memo* object could not invoke the *set_name* rule directly, since it is not an *office* object. When an event occurs that alters the instance variables or those of the acquaintance, the trigger conditions must be checked to see if a new event must be fired. The firing of one event may therefore "cascade", and cause other events to (eventually) be fired.

Trigger conditions may dictate the allowable message classes of acquaintances invoking a rule, the type of value passed by a communicating acquaintance, predicates over those values, and predicates over the instance variables of the object itself.

There are two special rules included in the behaviour of any object. The *alpha* rule is used to create new object instances, and the *omega* rule is used to destroy an existing object instance. The *alpha* rule may thus be used to specify the conditions under which objects may be created, whom they may be created by, and what instance variables should be initialized to when they are created. Of course, any side effects of object creation can also be included by causing the *alpha* rule to invoke other rules in acquaintances. The *alpha* rule for the *customer* object might be used to initialize the *name* variable. Once an object is created, other rules in its behaviour may be triggered.

```
omega {
        ~ : user;

        ~.owner = owner;
        }
```

Figure 3: an omega rule

The *omega* rule, given in figure 3, ensures that only the *owner* may destroy the object, and the act must be performed directly by the *user*, not any subordinate *office* object. Another possible use of the *omega* rule is to keep a log of the circumstances under which an object was destroyed.

2.3. Specialization

New object classes may be created from old ones by the process of specialization. A specialized object class is a *subclass* of some parent *superclass*. The subclass may have:

1. more instance variables: the existing instance variables are inherited from the superclass, and new variables are made available to instances of the subclass
2. more rules: the existing rules are inherited from the superclass and new rules are available to instances of the subclass
3. restricted domains: instance variables are inherited from the superclass, but they may assume values only from subdomains
4. restricted rules: rules are inherited from the superclass, but they may have additional trigger conditions to further restrict the cases under which they may fire

The definition of specialization given here is very similar to that used in the Taxis system [GrMy83].

Specialization is important, in that it ensures that new classes derived from some superclass have at least the properties of the superclass. All *office* objects might thus, for example, be defined to have *owner* variables set at creation, and rules that prohibit destruction by anyone other than the current owner. An important open issue is how much alteration of existing behaviour should be allowed in subclasses. If a specialized *office* object has altered behaviour or additional behaviour that completely undermines the behaviour of the unadorned *office* object class, then the fact that it is a "specialization" is virtually meaningless.

2.4. Expressive power

As described in [NiMT83, Moon84, Twai84], Oz objects can easily be used to capture the behaviour exhibited by event-oriented models such as finite automata and Petri nets. The state of an automaton can be easily described using the instance variables of an object, and the rules for changing states can be captured in the general language of the object's rules. In addition, one may associate additional side effects with the state transitions given by the underlying model. A typical application would be to implement *augmented Petri nets*, as described by Zisman in his dissertation [Zism77, Zism78]. In this formalism for specifying procedures, Petri nets are augmented by additional preconditions and actions that refer to the world outside the model.

Office procedures, as described in a companion paper in this book, can also be implemented using objects. Trigger conditions in the office procedures translate directly into trigger conditions of a procedure object, and actions similarly translate into object rule actions.

Objects can also be used to easily capture electronic forms. An electronic form would be represented by a single object class. Form instances would correspond to object instances, with each field of the form being represented by a single instance variable. Additional, "hidden", instance variables might also be used to maintain internal information about a form, such as who created it, or when it was last modified. All form types could therefore be implemented as specializations of a standard *form* object class, with a few minimal properties. Since a form's behaviour is entirely determined by the rules of its object class, there is no danger of corrupting existing forms by adding new applications to a system. These new applications would still be forced to make use of the form interface defined by the object's behaviour.

A wide variety of important field types [Geha82] can be implemented with comparative ease. Some of the possibilities are fields that must be supplied when a form is created, and then may never be changed, fields that must be filled in a particular order, fields that function as locks on other fields, and signature fields that are automatically filled when a particular action is performed. Restricted views can also be implemented, since the identity of an acquaintance must be made available before an object will release any information. Since the language for specifying actions is general-purpose, there is virtually no limit to the kinds of fields that can be implemented.

Intelligent messages, as described in the companion paper, "Intelligent Message Systems", are implementable using objects. In this scenario, messages are objects that not only store information, but carry procedures with them for dynamically altering the content of the message, and for altering or refining their destination. Since arbitrarily complicated procedures can be encoded in the behaviour of an object, intelligent as well as passive messages can be designed using the object formalism. For a discussion of various "flavours" of interesting objects, see the concluding paper of this book.

Finally, objects provide an elegant mechanism for ensuring data security and integrity. *Roles*, as described in the companion paper, "Etiquette Specification in Message Systems", can be implemented with objects. A trivial example of this is the use of the *owner* variable in *office* objects. In the hierarchical object world described in the concluding section of this paper, objects could be equipped with instance variables that are themselves *role* objects. The *role* objects may be arbitrarily complex (or as simple as the *owner* variable), and they may be thought of as authorization currency in object transactions. Since objects cannot be forged in an object world, the possession of a particular kind of *role* object may be used to guarantee certain powers or capabilities.

3. User Interface

Our prototype object-oriented programming system makes use of an explicit *user* object class to represent users. Whenever a user interacts with the universe of objects in any way, he does so under the guise of a *user* object. The system was designed in a highly modular fashion, so that one would not necessarily be forced to use one particular user interface. One interface might be appropriate for system developers and another, more appropriate for naive users who do not do their own programming. We describe here a simple, but general, interface that is adequate for illustrating the power of our system. The material discussed in this section is covered in greater detail in chapter 4 of the M.Sc. thesis by Twaites [Twai84].

3.1. The *user* object class

The user interface is a "back door" into the system that allows us to make instances of the *user* object class appear to spontaneously initiate events. The *user* object class has its own predefined behaviour just as all other object classes do. In addition, there is a special *io* rule to enable us to exchange information with other objects, and a facility to allow users to temporarily create new rules. This latter capability is necessary if we do not wish to limit our actions to what is set out in the *user* specification. Of course, any given implementation of a user interface may choose whether or not to allow arbitrary interactions between *users* and other objects. Programmers might require a general, unrestricted interface such as is provided by Oz, whereas applications might present highly specialized interfaces. The kinds of objects that a user may create and interact with can be explicitly governed by his *user* specification. Furthermore, it is possible to provide a variety of *user* specifications corresponding to a variety of roles to be played by the users of a system. System administrators could thereby control the valid interactions between roles. (See the companion paper, "Etiquette Specifications in Message Systems", for a discussion of roles.)

The predefined *user* behaviour would normally include an *alpha* rule, restricting authorized users to creating new users, as well as rules for keeping track of login passwords, and so on. This predefined behaviour may naturally be specialized to restrict or extend the power of certain users.

The *io* rule is used by objects that require human intervention for the completion of events. As an example, consider the *ok_user* rule in figure 4 that checks whether a password is valid before allowing an object to change state and continue communicating with a *user*.

```
ok_user {
    /* acquaintance must be a user */
    ~ : user;

    /* print message and test response */
    passwd = ~.io ("password: ");

    U := ~; /* remember the user */
    ok := TRUE; /* change state */
    }
```

Figure 4: using the io rule

The *io* rule is used to print a message and retrieve a value. Only if the value returned is acceptable will the rule and the event fire. Since a response must be received before the condition may be tested, *io*

rules are handled in a slightly different manner from other rules. Events including *io* rules must be suspended, pending the user's response. If, in the meantime, anything happens to disable the event (such as the object being destroyed), then the event simply dies.

Temporary rules are used to expand the automatic behaviour that is predefined for *user* objects. This facility is provided because it is not possible to predict everything that a user may wish to do. Users can therefore "tailor" their *user* objects by temporarily adding new rules. Temporary rules may be used, for example, to create new object instances, to query existing objects (through their rules), or to modify objects.

3.2. System commands

The current user interface presents the system through a screen, as shown in figure 5. Commands are entered in the first area. The second area is used to indicate the current mode. The interface message area is used to display messages pertaining to the user interface. The object manager message area is used to display messages from objects using an *io* rule. These messages may or may not require a response. A message that is purely informative requires no response, and is not blocking any waiting event.

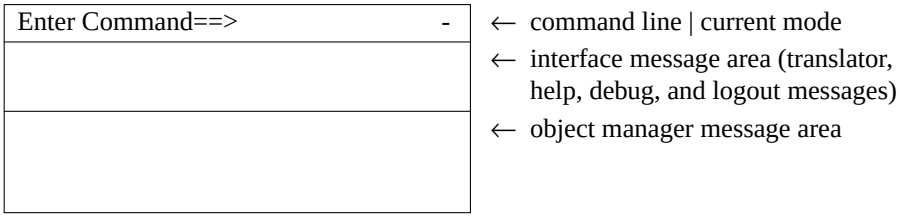

Figure 5: Screen layout

Whenever an object class or a temporary rule is being edited, the user interface screen is replaced by that of the editor, until the editing function is completed.

There are five top-level commands in the system, each with a one-letter name. The commands are all in prefix order, with the operator preceding the operands. The commands are:

h : Help facility.

l : Logout.

m [*message number*] :
The specified message in the object manager message area is displayed, and a response to the *io* rule may be made.

t [[<] *temporary-rule-name*] :
The specified temporary rule is executed. If preceded by a "<", the user is placed in the editor, and the new or existing rule may be edited.

c [*object class*] :
The user may edit the new or existing object class definition. Upon exit from the editor, the definition may be translated and (upon error-free translation) added to the universe of object classes.

A BNF grammar for object classes and rules is presented in the appendix of this paper.

4. System Design

The Oz system is written in the C programming language [KeRi78], and runs on a VAX 11/780 under the UNIX™ operating system. The current implementation consists of under 4000 lines of C code. The VAX was chosen for the UNIX™ program development environment and for its availability rather than its size. The Oz system could easily have been developed on a smaller, stand-alone system such as a Sun workstation (which also runs UNIX™). One of the goals of the project was to allow users to share the same object universe. We decided, therefore, to have one process per user, plus a single process dedicated to object management. The system interface is via the user processes. Requests and commands that affect the object universe are then passed on to the object manager, which updates the database of objects.

Conversely, when events take place that affect users, the object manager notifies the appropriate user processes. The division of labour between system interface and object manager is intended to be transparent to users.

One of the difficulties in using UNIX™ as an environment for implementing Oz is that processes cannot share memory. The communicating processes would be forced either to pass information through temporary files, or to make use of UNIX™ "pipes", which are buffers for passing streams of data. For the sake of speed, the latter approach was chosen.

A related difficulty was that processes may not communicate via pipes unless they are "related", that is, they have some common ancestor. This problem was solved by introducing a special "host" process that babysits the pipes and spawns new user processes. Whenever a user wishes to enter the system, the host process is signaled, and a new user process is created. (Signals may be sent between arbitrary processes, provided they have the same "group id", and the process identification of the receiver is known.) The new process inherits the pipe from the host, and communication with the object manager is enabled. The host is only retired when there are no more user processes connected to the object manager *and* the object manager has exhausted its current list of work to do. The next person entering the system will (transparently) create a new host and a new object manager.

Finally, we had to decide whether to make use of pipes in either direction, between the object manager and each of the user processes, or have just two pipes (one for data traveling in either direction) shared by all the user processes, or use some further variation. For simplicity's sake we decided to use just two pipes. There appeared to be no realizable efficiency gains by having multiple pipes, since the object manager could read messages from ten pipes no faster than from one. Whenever a process places a message on one of the pipes, it notifies the receiving process by sending it a signal. Reads and writes are guaranteed by UNIX™ to be atomic actions, thus ensuring the integrity of the messages. Signals, however, are not queued, so a reading process must always check the pipe after reading a message, to be certain that the pipe is empty.

Since processes are blocked if they attempt to write to a full pipe, write-request, write-ok, and receipt-acknowledgement signals are used to inform processes about the status of a pipe. The object manager makes sures its messages are received before attempting to send new messages to other user processes, and user processes must request a free pipe before they attempt to send a message to the object manager.

Although these considerations may be of interest to someone implementing an object-oriented programming environment, they do not have a direct bearing on the object model as described in the previous section. They do, however, illustrate the gymnastics one must go through in order to implement objects in an environment with an architecture that is better suited to supporting processes, files and stream i/o.

The messages that are sent between the user and object manager processes are all of a standard format. Each message consists of four pieces of data: the message type (represented by a short integer), the process identification of the sender (for acknowledgement purposes), the length of the message (in bytes), and the message body (generally a character string). User processes currently may send the following messages:

login request:
sent if a user wishes to log into the system

change class definition:
sent if a user wishes to add, change or delete an object class

temp rule:
a temporary rule is being sent for immediate execution (and subsequent disposal)

instance manipulation:
the user wishes to manipulate an object instance (currently handled through temporary rules)

reply to message:
the message body is the response to an outstanding *io* rule message

logout request:
sent if a user wishes to exit the system.

There is a corresponding set of messages that may be sent by the object manager

logout:
acknowledges a logout request; the user process may exit, die, and return control to the calling program (usually the UNIX™ shell)

changing user contents:
a change has occurred in the *user* object corresponding to the logged-in user; the user process maintains a consistent version

io rule message:
a message from another object is sent to the *user* object via the *io* rule; a response may be in order

response to previous user message:
a response is given to a previous temporary rule, an object class definition change or a logout

login successful:
used to inform the user process that an attempt to log in has been successful

5. Object Management

The object manager is responsible for storing and retrieving objects, and it must find and execute events. Object storage is divided into two components. Since all objects of the same class share the same behaviour, it is only necessary to store that behaviour once. Object instances of the same class are distinguished only by their contents. Thus only the instance variables are actually stored for each object instance. Object rules and the variable declarations are stored separately, in a structure that supports the notion of object specialization.

5.1. Storing and retrieving objects

In the compilation and translation of object definition, the instance variable names are converted to integers which serve as indices into a table of information about the variables. The correspondence between the variable names and the indices is stored in a symbol table. The information about the variables includes:

1. whether the variable is an instance variable or a temporary variable
2. the type of the variable
3. if the type is *object*, then the object class
4. the location of the value held by the variable

Of course, only the permanent (instance) variables are stored. Temporary variables exist only when events are being fired, and storage for them is provided at that time.

Similarly, a certain amount of processing takes place when rules are translated. Rule statements accomplish four things:

1. they may establish conditions which, if false, cause an event to fail
2. they may assign values to temporary variables
3. they may pass information to an acquaintance
4. they may update the value of an instance variable

The first three of these functions are done while events are being assembled. The last may only be performed if the event does not fail. It is, of course, possible to update an instance variable to some value sent by an acquaintance in a single statement. Statements are therefore decomposed into simpler statements that fall into just one of the above categories, and assignments to instance variables may be translated into two statements: an assignment to a new temporary variable, and reassignment to the instance variable only if the event succeeds. The simplified rule statements are then stored in a list structure and interpreted at run-time.

Specialization of object classes is implemented by storing rules and variable declarations in an *m-way* tree [HoSa76]. Nodes in the tree correspond directly to nodes in the specialization hierarchy. To determine which rules and variables, or which versions of rules and variables apply to a given specialization, one

simply searches *up* the tree to the root. One therefore inherits the closest version of a rule or a variable. If the rule or variable does not apply to the given specialization, then the search ends with failure at the root of the tree.

5.2. Event-searching

Rules may either be explicitly invoked, or they may be self-triggering. The self-triggering rules wait for some condition to become true, and the triggered rules wait to be invoked by another rule belonging to some acquaintance (possibly another rule in the same object).

Event execution begins with self-triggering rules. A depth-first search algorithm is used to build the event. Whenever a call to a rule in an acquaintance is made, a branch is made in the tree, and execution continues at that level. If execution successfully completes at a certain level, control returns to the level above, and eventually to the self-triggering rule. If it does, an event will have been constructed, and the tree will be traversed to update the instance variables.

If at any point a rule fails, the backtracking of the depth-first search algorithm takes effect, and an alternative acquaintance is sought. This process continues until an event is constructed, or all possible acquaintances at some level are exhausted.

In addition, if an *io* rule is encountered, the event is suspended and the event-tree is saved, pending a user response. In this implementation, the objects in the tree are marked, and not used in other events until a response is received.

Since the event construction always starts with a self-triggering rule, a queue is kept of all such rules. The object manager repeatedly attempts to construct events starting with these rules until it succeeds. Although far more efficient schemes were initially considered, the simplicity of this approach made it quite adequate for the purposes of the Oz prototype. An alternative is outlined in the following section.

Note that no synchronization problems ever arose, since the object manager would never attempt to execute more than one event at a time.

6. Observations and Conclusions

The Oz system served primarily to demonstrate that certain ideas about programming with objects were workable. Not only is the object model powerful enough to capture interesting behaviour, but it appears to be quite workable as an implementation language. Our experience with Oz leads us to several conclusions about what is required to produce a useable object-oriented programming language. In addition, there are a number of open questions and philosophical puzzles concerning the proper way to implement such a language (see also the companion paper, "Objectworld").

6.1. Basic requirements

First of all, one would need to get rid of processes and files. They are not only conceptually incompatible with object-oriented programming, but the overhead they introduce could only serve to slow down an implementation by an order of magnitude (say). Instead, one would need a large, permanent, virtual memory. The address space required would certainly be larger than the host computer's primary storage, and would have to include at least all of available secondary storage. Since files will not exist, all of secondary storage will be available for the storage of objects (although "files" could be made available through the object interface). The virtual memory provided could be very simple. There would be no notion of objects associated with pages of memory at this level, although the pages themselves could be viewed as objects.

An object manager would use the virtual memory to permanently store objects. It would have to be able to bring any object into main memory quickly, given a unique object identification (*id*), and it would have to "swap out" inactive objects intact. An important requirement would be always to keep the object versions on disk at least coherent, and as up-to-date as possible. The object manager would also include (or work in tandem with) an event manager that would decide what objects were currently of interest. Requirements for the event manager are discussed the next section.

Certain objects would function as interfaces to device drivers. These objects would include the disk drive, the terminal, a communications network, and so on. A uniform object interface to everything would be desirable, so that even the operating system kernel, and pages of memory, could be dealt with as objects, by privileged programs. This is important if the language is to have any credibility as a systems implementation language.

6.2. The event manager

Events trigger other events. An event that has failed once will always fail, unless something happens to change the state of one of the objects involved in the event. It follows, as a consequence, that it is only necessary to check whether or not events are fireable, when variables mentioned explicitly in trigger conditions are altered by other events.

It suffices, therefore, to keep track of a queue of recently altered (and created) objects. For each object in the queue, one must determine what new events may be triggered as a consequence of the state change, and then attempt to construct an event. If no new event is found, the object is removed from the queue. Otherwise an event is found, and all altered participants are added to the queue. Of course, the queue need not be handled in a strictly sequential fashion. It is only necessary to ensure that all objects in the queue are handled eventually, and preferably before any objects that are added later. True concurrency may be achieved if several events are searched for at once.

To construct events, one would need to keep track of who is acquainted with whom, and determine which objects may initiate an event involving the one in the queue. It is open at this point to what extent one may intelligently choose possible events. To a large extent, this depends on how carefully the language is designed. The event manager should be presented with a clear list of possible acquaintances, to eliminate random searching. The event mechanism should be presented to the object programmer in such a way that it is clear how costly it will be to search for events, depending on how objects are designed.

Obviously, one would save time by checking only events that have a reasonable expectation of succeeding. A good language design can eliminate a great deal of fruitless event-checking, by making it possible at compile-time to note which events might trigger other events.

6.3. Object domains

A more sophisticated way of organizing objects is needed. A flat object universe makes event-searching a horror if there are many objects. One may easily organize objects hierarchically into *domains*. Each object is then an instance variable of some parent object, which is its domain. Conversely, all objects are the domains for their instance variables.

Parents are automatically acquainted with their children, and vice versa. It immediately follows that children can (ultimately) only become acquainted with anything in the outside world — and even with other siblings — through their parents. A parent may access its children through the instance variable names, but all other objects must do so through the children's *id*s. An *id* may thus be thought of as an indirect reference to an object. Once an object becomes acquainted with other objects, however, it becomes a free agent. A parent may choose, of course, to be protective, and always act as a middleman for certain of its children. The only other object that would necessarily be acquainted with *all* objects in the system would be the object manager. System objects or other privileged objects could then learn the identity of any object through the object manager, even when the parent is reluctant to reveal it.

Instance variables save space for objects. This is consistent with our intention that everything be an object. In *Oz*, only primitive objects (strings, etc.) were instance variables, but, in general, instance variables can hold any object. A parent may create a child object by saying to the system, "create an object for me, and put it *here*". If the object is destroyed, the space may be reused. Note that an object may only create another object if it has a place to put it. Otherwise it must find an acquaintance who is willing to be a parent.

Since one does not necessarily know the classes of all objects that one will become acquainted with, there should be some facility for discovering the class of an acquaintance. Similarly, an object would need to be able to discover what rules are valid for that acquaintance, and be able to *dynamically* address an arbitrary rule. Both of these problems may be addressed by supplying default rules to all objects in the

language for revealing class and behaviour information, and for accessing rules dynamically using, say, strings composed of rule names. This is comparable to facilities in languages such as APL, LISP, and Snobol, that allow one to compose strings of commands and execute them through an interpreter.

6.4. Rules and instance variables

A wider variety of instance variables is needed. Instance variables could be primitive objects, such as integers, characters and object *id*s, or they could be complex objects. One would naturally want to have arrays of objects, but it also appears highly desirable to allow for *lists* of objects. A list would be similar to an array, but of unbounded length. Furthermore, whereas an array could contain gaps for nonexistent objects, lists might consist of existing objects only. Lists are important to have if certain objects and domains are to grow without bound. In particular, a *text* object would likely have an instance variable which is a list of characters (a string).

Rules similarly require some re-thinking. Explicit *assert* and *fail* statements appear to be more natural than the present scheme of simply stating conditions. (A *fail* statement is equivalent to *assert*(FALSE), and an *assert*(<cond>) is equivalent to *if*(not <cond>) *then fail*.) The ability to spawn asynchronous events may also be necessary for certain applications, though it is not clear what would happen to an event spawned by another event that fails.

A cleaner notion of event-searching results, if we force objects to provide a list of acquaintances with which they may be interested in communicating within some event. Clearly, the longer these lists get, the more work that must be done to search for events. The cost of event-searching is more directly in control of the object programmer. Ideally, the programmer should be able to tightly specify precisely the circumstances under which event-searching should take place. A carefully designed object would then cause a minimum of unsuccessful event searches. Again, a good language design will make the cost of triggering for alternative object specifications very apparent to the programmer.

6.5. Open issues

There are a number of questions for which it is more difficult to provide adequate answers. Some of these may be religious issues that can be argued a variety of ways. Others may quite significantly affect the function and semantics of the language and the system, but in ways that are not yet obvious. Still others do not seem to yield any appropriate solutions. We shall briefly discuss a few of the more interesting questions.

Are rules objects? Certainly object specifications are themselves objects (possibly *text* objects), and the executable code must be stored as an object, but there appears to be no conceptual justification for viewing rules as objects. Alternatively, it would be very convenient to be able to dynamically create rules, store them as instance variables, and execute them. Temporary rules could be handled in this fashion. Certain objects could then modify their own behaviour, or deal with arbitrary acquaintances in interesting ways. A good example would be a debugging object used to develop new object specifications.

Should objects be allowed to change their own specification by adding variables or rules? If an object has a list of rules, and rules are objects, then an object could just create a new rule and add it to the list. Somehow this seems to run contrary to the principle of an object as a sort of abstract data type. Instead, perhaps one should have to create a new object class and convert old objects to new objects. This would avoid horrendous problems in managing objects that are always changing their own representation. Furthermore, if it is possible to dynamically create and store temporary rules as instance variables, then it is no longer necessary to alter the default behaviour of an object.

Since one does not necessarily know the classes of all objects one may become acquainted with (since objects of new classes will likely become acquainted with old ones), there must be a way to get at the rules of these new objects. The suggestion made earlier was to allow for dynamic invocation of rules. This might be sufficient justification for a *rule* primitive object class which would be used just to store rule identifiers (as opposed to strings containing their names).

Temporary variables present some philosophical problems. Are they objects too? They can hold the same information that permanent objects can, but they come and go with apparent abandon. This may be a religious issue, since one can take the view that events are atomic, and, as a consequence, temporary

variables never really exist.

More seriously, one should consider what is meant by assigning a value to an instance variable. Since instance variables are objects, one should never be able to simply "assign" a value to them. Rather, one should have to invoke a rule in the object, and pass the value to be assigned. Of course, this must eventually stop with primitive objects, so one could consider the notation ":=" as shorthand for invoking an implicit *assign-value* rule. Complex objects must be treated with more respect, however. It follows then that the only "values" appropriate for passing between acquaintances are primitive objects such as integers, characters and object *id*s.

An exception to this rule would be if an object is to change domains. It might be necessary, for example, to send an object from one machine to another. The alternative would be to destroy the original of an object, and to create a "copy" in the new domain. For many object types, however, it might be undesirable to allow the creation of copies in this fashion. Far simpler and much more elegant would be to permit objects to change domains. In the case of primitive objects such as integers, *id*s and strings, it is simpler to make a copy of the object, and pass that when communicating with an acquaintance. If a large object is to be passed (rather than simply its *id*), duplication of the object is likely to be undesirable, for efficiency reasons. In environments where objects represent documents or private communications, it is important to be clear that the actual owner of the object may change, rather than just its apparent owner.

If several object systems are to be connected via a network, and these systems are allowed to exchange objects, then it is important to ensure that all objects have *id*s unique in the entire object universe. All objects on a given machine should therefore be provided with identifiers that somehow indicate the host machine on which they were created (or the object manager should at least be able to handle identifiers for objects originating from a different machine, if they may superficially coincide with local identifiers).

A thorny question is how to handle events taking place between two (or more) machines. A reasonable approach is to appoint *overseer* objects that act as go-betweens for all the objects on a given machine, and those on other machines. The *overseers* would then be the only objects to partake in very simple events limited to exchanging objects between systems. Once an object has moved to a different system, it can take part in more complicated events.

As a final comment, we should point out some of the dangers of muddying the atomicity of events. If an event is allowed to "partially fail", or to fail but spawn another event before failing, then there is a potential for unauthorized information to leak from an object. Atomic events have the desirable property that none of the participants in an event give up any information unless all of them agree to a mutually acceptable contract (consisting of all the trigger conditions). An event, by definition, has no side effects unless it fires. If this definition is relaxed even slightly, then the security of all objects is threatened. Any attempts to do so would therefore have to take this into account by preventing pending events from communicating with external objects or with other events.

7. Appendix: BNF Grammar for the Oz Language

The BNF grammar presented below uses the following meta-symbols and meanings:

::=	shall be defined as
|	alternatively
[x]	zero or one instance of x
{ x }	zero or more instances of x
"xyz"	the terminal symbol xyz
< x >	the non-terminal symbol x where x is a sequence of letters and hyphens beginning with a letter.

<object> ::= <object-class> ":" <super-class> "{"
{ <declaration> ";" }
{ <rule> }
"}"

<object-class> ::= *user*
| <identifier>

<super-class> ::= *object*
| <object-class>

<declaration> ::= <variable> { "," <variable> } ":" <type>

<rule> ::= <rule-name>
["(" [<variable> { "," <variable> }] ")"]
"{" { <statement> ";" } "}"
["(" [<variable-value>] ")"]

<statement> ::= <declaration>
| <condition>
| <send>
| <assignment>
| <function>
| <sub-rule>

<condition> ::= <variable-value> <comparator> <expression>

<comparator> ::= "="
| "!="
| "<"
| "<="
| ">"
| ">="

<send> ::= <send-name> "." <rule-name>
"(" [<variable-value> { "," <variable-value> }] ")"

<assignment> ::= <identifier> ":=" <expression>

<function> ::= <identifier>
"(" [<variable-value> { "," <variable-value> }] ")"

<sub-rule> ::= "{"
<statement> ";"
{ <statement> ";" }
{ "|" <statement> ";" { <statement> ";" } }
"}"

<expression> ::= <variable-value>
| <function>
| <send>
| <arithmetic-expression>
| "(" <expression> ")"

<arithmetic-expression> ::= "-" <expression>
| <expression> <arith-op> <expression>

<arith-op> ::= "*"
| "/"

```
        | "+"
        | "-"

<rule-name> ::= alpha
        | omega
        | io
        | <identifier>

<send-name> ::= <identifier>
        | "~"
        | "*"

<type> ::= integer
        | string
        | <super-class>

<variable-value> ::= <variable>
        | <value>

<variable> ::= "~"
        | <identifier>

<value> ::= <integer-value>
        | <string-value>
        | "*"
        | nul

<identifier> ::= <alpha> { <alphanumeric> }

<integer-value> ::= <numeral> { <numeral> }

<string-value> ::= <double-quote> <character> <double-quote>

<alphanumeric> ::= <alpha>
        | <numeral>

<alpha> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h"
        | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q"
        | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
        | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I"
        | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R"
        | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "_"

<numeral> ::= "0" | "1" | "2" | "3" | "4"
        | "5" | "6" | "7" | "8" | "9"

<double-quote> ::= the double quote character (").

<character> ::= any character - the conventions for non-
        printing characters, single quote and "\" are the
        same as in the C programming language [KeRi78].
```

8. 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*, 2(3), pp. 173-196, July 1984.

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

[Geha82] N. Gehani, "The Potential of Forms in Office Automation", *IEEE Transactions on Communications*, Com-30(1), pp. 120-125, January 1982.

[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. Myplopoulos, "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*, 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*, 8(3), pp. 323-364, June 1977.

[HoSa76] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, 1976.

[KeRi78] B.W. Kernighan and D.M. Ritchie, "The C Programming Language", Prentice-Hall Software Series, 1978.

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

[NiMT83] O.M. Nierstrasz, J. Mooney and K.J. Twaites, "Using Objects to Implement Office Procedures", *CIPS conference proceedings*, Ottawa, pp. 65-73, May 1983.

[SSKH82] M. Sirbu, S. Schoichet, J. Kunin and M. Hammer, "OAM: An Office Analysis Methodology", in *Office Automation Conference 1982 Digest*, pp. 317-330, AFIPS, 1982.

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

[Zism77] M. Zisman, *Representation, Specification and Automation of Office Procedures*, Ph.D. dissertation, Wharton School, University of Pennsylvania, 1977.

[Zism78] M. Zisman, "Use of Production Systems for Modelling Asynchronous Concurrent Processes", *Pattern-Directed Inference Systems*, Academic Press, pp. 53-68, 1978.