Using Objects to Implement Office Procedures[1]

Oscar Nierstrasz
John Mooney
Ken Twaites

Department of Computer Science
University of Toronto
Toronto, Ontario, M5S 1A1

ABSTRACT

Office information systems (OISs) provide facilities for automatically triggering procedures when certain conditions become true or particular events take place such as receipt of mail. Such systems are characterized by a high degree of parallel activity that cooperates with but may run independently of user processes. Traditional high-level programming languages do not readily capture this sort of behaviour. This makes building a customized OIS a painful process. "Objects" are entities with contents and a set of rules describing their use. We believe that objects are a useful primitive for designing and building such systems quickly.

1. Introduction

An *office information system* is a computer system that models the activities of an office. One would expect to find that the objects defined in an OIS parallel the real objects in a physical office such as desks or workstations, memos, forms, telephones, calculators, tables, mail trays and so on. An OIS presents a uniform medium in which to represent the objects in an office thus permitting the automation or partial automation of routine activities and providing the advantage of increased speed of communication.

Automation in an OIS is enabled by permitting events to trigger other events. The receipt of a certain kind of message, for example, may trigger a procedure which automatically reads the message and responds to it. The creation or modification of any object may also trigger an event. By specifying exactly what situations may trigger an event a user passes the responsibility of firing it to the system. What distinguishes an electronic office from a physical office is that in it we may implement "intelligent" office objects. Such objects may be passive and merely constrain their appearance or range of values, or they may be active and initiate more involved office procedures. These procedures need not be explicitly called in all cases. Instead, one might ask for an event to be triggered whenever a particular situation arises, without indicating what other events might possibly be responsible.

"Objects" can be used to model intelligent office objects and office procedures in an easy, natural way. Intelligent forms with the full range of field types and constraints can be specified. Standard form manipulations can also be easily defined. In addition, it is simple to define office procedures that are modelled on sequential scripts, finite automata or even augmented Petri nets.

The term *object* is a familiar one but also a notoriously imprecise one. It suggests at once a collection of similar concepts: abstract data types, intelligent messages, modules, actors [Hewi77], boxes [deJo80], data frames [Embl80], automatic procedures [TRGH81] and Smalltalk objects [BYTE81].

[1] In *Proceedings of the Canadian Information Processing Society Conference*, Ottawa, May 1983, pp. 65-73.

Rather than invent a new word, we will try to present a simple definition of "object" that adequately captures its vernacular usage and is also powerful enough to model real object-based systems. A computer implementation of such objects would allow us to simulate (say) intelligent messages or all of the parts of an office information system.

We are currently implementing an object-based system at the University of Toronto. It is being written in the C programming language on a VAX-11/780 running the UNIX operating system. The design and implementation of the project is divided into two major parts -- the object manager and the user interface. The object manager identifies fireable events, and fires the rules of each of the participants of the event. The user interface enables the user to define new object classes and interact with existing object instances through the use of a special "user object".

2. An object model

An *object-based system* (OBS) enables one to model objects within a computer. It allows users to define classes of objects with contents resembling database relations and behaviours resembling bodies of associated code. A behaviour consists of rules for the creation, transformation and destruction of objects. The difference between this definition of objects and most others in the literature is that we allow an object's rule to be triggered implicitly rather than having to be called explicitly. An object behaves as though it is always watching for a triggering condition to become true.

An OBS enforces the behaviour by detecting conditions that will trigger events. This is the task of *managing* a set of objects. A general OBS could be used as a programming language for building OISs.

A more formal notion of an object is needed before we can begin to discuss the properties of an OBS. The state of the world is characterized by the states of the objects in it. A particular object may take on a number of values in its lifetime, but the changes happen in a restricted way. A chair may change colour if it is painted, but it cannot be changed into a bicycle. We need, therefore, a model that captures the individual characteristics of objects in the same class, and also allows us to describe how these objects may interact. Objects interact by "doing things" to each other that "make sense". You can paint a chair, but not a puddle of water. When two or more objects interact, something "happens", that is, the properties of some of the objects may change. When a chair is painted, one of its properties--its colour--changes.

2.1. Objects

An "object" is a thing -- something with an identity. Objects can be partitioned into groups of similar objects (chairs, desks), and they have a set of properties which makes them unique (wooden classroom chair, comfy chair with leather upholstery). Objects also have a behaviour which captures their relationship to other objects. An object "knows" its behaviour and is responsible for it. You can sit on a chair, but you can't drive it to Montreal. If you try, it simply won't cooperate.

Objects can be partitioned into *classes* on the basis of some reasonable grouping -- chairs, bicycles, presidents. Objects in the same class are referred to as *instances* of that object class. Instances are characterized by their *contents* -- a set of *instance variables* that retain their values between the firing of events. Associated with an object instance is its *behaviour*, a set of *rules* that use the instance variables and make them available to other objects.

A specification for a trivial object class called *secret* follows. It stores a secret message and a key to unlock the message. Anyone who knows the key can obtain the message. When the message is disclosed, the object instance self-destructs.

```
secret: object{
        /* instance variable declarations */
        msg, key : string;
        owner : user;
/* rules */
alpha(m,k){
        /* only users may create secrets */
        ˜ : user;
        m, k : string;

        /* conditions */
        m ≠ "";
        k ≠ "";

        msg ← m;
        key ← k;
        owner ← ˜;
        }

omega(k){
        ˜ : user;
        k : string;

        /* if key matches */
        /* return msg */
        /* then self-destruct */
        k = key;
        }(msg)
}
```

The *secret* object has three instance variables in its contents: *msg*, *key* and *owner*. It has two rules in its behaviour: *alpha* and *omega*. *alpha* and *omega* are reserved words for rules that create and destroy object instances. Any other rules would apply to instances that have been created but not yet destroyed. "˜" is a reserved word that identifies another object invoking that rule. "*" is another reserved word meaning "myself".

A *user* object could automatically obtain these messages with the following rule:

```
getmsg{
        /* no ˜ variable: getmsg is */
        /* triggered--not explicitly called */

        s : secret;

        /* TRUE is a boolean constant */
        ready = TRUE;

        save ← s.omega(key);
        ready ← FALSE;
        }
```

ready, *save* and *key* are instance variables of the *user* object. The *getmsg* rule can only fire if the *ready* variable is set to *TRUE*. This prevents *getmsg* from firing repeatedly and stomping on its saved messages. (There might be another rule that allows users to look at the saved message and reset the *ready* variable.) Whereas the *alpha* and *omega* rules of the *secret* object must be explicitly called, the *getmsg* rule may be fired automatically whenever there is a *secret* object with a matching *omega* rule. For that reason, the *getmsg* rule accepts no parameters and returns no value.

An object's *acquaintance* is another object that it knows about. Acquaintances communicate by invoking rules in each other's behaviours. The acquaintance variable in the *getmsg* rule is *s*. The *alpha* and *omega* rules have only the ˜ acquaintance -- the object invoking the rule. The *getmsg* rule has no invoking acquaintance.

A rule may only fire if all the conditions it contains are true. A rule is *active* if the conditions on its instance variables *alone* are true. *getmsg* is only active if *ready=TRUE*. *alpha* and *omega* in the *secret* object are always active -- there are no conditions on contents, only on values passed by acquaintances.

Specialization can be a useful tool in defining new classes that are derived from existing classes. A sub-class identifies a special case of the existing class which requires some additional attention. A sub-class may be defined by specializing its superclass' contents or behaviour. Contents may be specialized by adding new instance variables or by restricting the domain of existing ones. Behaviour may be specialized by adding new rules or by modifying existing rules.

2.2. Events

An *event* is a collection of object instances and a collection of rules applying to those object instances. The objects in the collection are called the *participants* of the event. All acquaintances mentioned in the rules of the event must belong to the set of participants. From the example in the previous section, the *getmsg* rule and the *omega* rule together with the two objects involved would constitute an event.

A rule in an event is *fireable* if *all* the conditions in the rule are true. Firing a rule consists of executing statements within the rule. Before any rule in an event can fire there must be unanimous agreement. This means that all the rules in the event must be fireable. At this point the event itself may be fired and the contents of the objects participating in the event may be updated according their behaviour. If any rule in an event is not fireable then nothing happens.

3. Office procedures

An office procedure is a program for accomplishing a particular task. Typically such tasks require the manipulation of some set of physical objects (paper forms, books, telephones, pencils), some clearly defined actions, some decision-making, and some information-gathering and waiting. We propose that objects, as they are described above, are a natural choice for implementing office procedures on a computer. High-level programming languages like Pascal, COBOL or LISP, on the other hand, do not provide the primitives needed for easily defining this type of behaviour.

One may define an office procedure to accomplish a simple one-time task like modifying some field on a form, or to handle a large-scale, ongoing task like inventory control. In the first case the office procedure would be handled by a single event, perhaps involving only one rule in a single object. A single event would be appropriate because the task is logically indivisible; it is not composed of smaller subtasks which may succeed or fail. For the same reason the creation of a form is an indivisible task even though it may involve the filling in of several fields. The form is either created or it is not created. It is not possible to have a partially created form. If it is the *intention*, however, to allow the creation of a form with some or all of its field blank, then this task must be broken into its indivisible components. The *first* subtask--the actual *creation* of the form object--would still be indivisible, however.

In the second case, a large-scale office procedure would have to be broken into its indivisible steps. Every step which must succeed entirely or fail entirely would correspond to the firing of a single event. Waiting, information-gathering and decisions that influence the control flow of the procedure are modelled by conditions within the rules that make an event fireable or not. The specification of a procedure is distributed across the rules of the objects involved.

One has the option of building a procedure out of objects that talk directly to one another (order forms talk directly to inventory records and customer records) or creating intermediate objects that coordinate the others (a "fillorder" procedure collects the inventory form, the order form and the customer record, then fills the order). The first approach is preferable if one wants to limit for all time how the objects involved in the procedure are to be used. The procedure is, in a sense, insulated from the rest of the system, because the objects involved are incapable of communicating with any other object classes. The second approach is more flexible if future applications that make use of these objects are anticipated. Each object is provided with a set of rules which a wide set of object classes can access. The objects involved are more passive in nature. Events are brought together by "procedure objects" that collect the other objects and invoke rules within them.

Localized intelligence is the kind that applies to the contents of one object alone, such as the fields of an intelligent form. Objects are capable of modelling a wide variety of important field types. [Geha82] identifies most of the following:

A *required* field must be filled at object creation time. Such a field would be set in an object's *alpha* rule.

An *unchangeable* field may be filled at any time but must never be modified. The condition *field≠null* is simply placed in any rule that modifies its value.

A *virtual* field is one whose value is computed but never stored:

```
total{
        ˜ : object;
        t : integer;
        t ← price × quantity;
        }(t)
```

Since the only access to an object's contents is through its rules, it is absolutely transparent whether a field is virtual or not.

Ordered fields are fields that must be filled in some set order. Here we simply place the condition $field_n \neq null$ in any rule that modifies $field_{n+1}$.

Lock fields are similar: a lock field is set to *true* or *false*, and rules that modify the fields to be locked simply test the lock field.

A *personalized* field is one that is a function of the person performing an operation. An example is a signature field which is automatically filled whenever some other field is filled:

```
quantity=(q){
        ˜ : user;
        q : integer;
        q > 0;
        quantity ← q;
        /* obtain caller's name */
        signature ← ˜.name;
        }
```

Any rule that would require a personalized field or a signature field to be filled would simply fire the appropriate rule in the calling object to obtain the necessary information.

Default fields can be set to constants in the *alpha* rule.

Key fields and *date* or *time* fields can be set by calling a system function. Keys could also be allocated by a special object that keeps track of a counter.

Certain standard operations can also be easily accommodated. *Create* and *destroy* operations are accomplished by an object's initial and final rules. *Retrieval* involves nothing more than restricting one's acquaintances by placing a condition in a rule:

```
**myboss**{
        /* any object can ask */
        /* who your boss is */
        ˜ : *object*;

        m : *manager*;

        /* find the right manager */
        m.name() = boss;
        }(m);
```

If *boss* is a variable containing your boss' name, then the condition *m.name()=boss* guarantees that *m* is the desired *manager* object.

Editing operations have already been discussed. Any rule may specify the conditions under which an instance variable may be modified. The niceties of editing and displaying objects are the job of the user interface, and are not part of the object model.

Mailing and *printing* operations present minor difficulties in that they deal with entities not inherent to the object model, namely machines and printers. Objects limit their scope by placing conditions on their acquaintances. One may model ownership by explicitly including an *owner* or *location* field in every object. The set of all objects would then be partitioned according to workstations or machines. The operation of mailing an object would be accomplished by simply firing a rule that changed that variable. The object manager would have to be smart enough to realize that changing the location variable of an object means that that object must be moved from one machine to another.

Objects are also able to capture large-scale intelligence. An office procedure consisting of a simple sequence of steps can be modelled by an object in which the steps appear as rules and a counter is used to keep track of the next step to be fired.

```
**step**$_n${
        counter = n;
        .
        .
        .
        counter ++;
        }
```

Objects can simulate finite automata:

```
**rule**$_k$
        {
                state = $j$;
                ...
                state ← $j'$;
        |
                state = $l$;
                ...
                state ← $l'$;
        |
                ...
        }
```

A state variable keeps track of the automaton's state. Each rule is composed of a number of alternatives that describe the action to be performed in each state, and the appropriate next state to follow.

Objects can also simulate (augmented) Petri nets [Zism77]:

```
transition_i {
        /* input states */
        state[j] > 0;
        state[k] > 0;
        ...
        /* additional conditions */
        ...
        /* actions */
        ...
        /* outputs */
        state[l] ++;
        state[m] ++;
        ...
        }
```

In all of the above cases, it is the object manager's responsibility to detect when a rule may be fired. Merely defining the objects is enough to guarantee the desired behaviour.

4. The object manager

The task of the object manager in an OBS is to manage the internal representation of objects. This entails the translation of object class definitions (or modifications to them) into their internal representations, and the management of object instances by searching for and firing events. Users are modelled by a special "user object" implemented by the user interface, which presents objects and events to users, interprets user requests, and communicates with the object manager. It will be discussed in greater detail in the next section.

4.1. Translation

The translation of object instances presents no special difficulties. The object manager need only generate the appropriate data structure for storing the instance variables of each object instance. This is similar to the task of implementing relational databases.

Much more difficult is the problem of translating behaviours. A rule may be broken down into several parts. There are conditions and actions. Some conditions may be determined from the values of the instance variables alone. These conditions determine whether a rule is *active* or not. If a rule is not active, it is certainly not fireable. All other conditions depend on values sent by acquaintances, and hence cannot be determined until an event is constructed. It is therefore very useful to keep track of which rules are active in order to simplify the task of constructing events. A simple bit vector can be maintained for this purpose. Whenever a rule is fired that modifies some instance variables, the object manager must check whether any other rules have been activated or deactivated. Every rule must then keep a list of what new rules to check.

The remaining conditions can be divided into conditions on the invoking object and parameters passed by it, and conditions on other acquaintances and values returned by them.

Actions must also be segregated. Some actions set temporary variables to be used in calculations or to be passed to an acquaintance. Others modify the contents of an object. Those statements that modify an object's instance variables must not be executed until a fireable event has been identified, for it is these statements that constitute the permanent side effects of firing the event. All other statements "merely" establish conditions or aid in passing values between objects.

Finally, when an object fires a rule, it may not only activate or deactivate one of its own rules, it may in fact effect a state change that some other object was waiting for. One must therefore also keep track of who may be invoking one's rules. A rule that becomes active becomes available to another object waiting to invoke it.

4.2. Event management

The object manager must guarantee that every fireable event eventually either get fired or become disabled by the firing of some other event. It is not possible to guarantee that all fireable events get fired because two events may overlap in objects that may only participate in one of the two. If, for example, two users have the same *key* for their *getmsg* rule, then a *secret* object with a matching *key* could only disclose its *msg* to one of the two objects since the object would self-destruct after the firing of the first rule. We assume, in cases like this, that the firing of *either* event is equally appropriate for accomplishing the task at hand. What follows is a scenario that informally describes the algorithm that searches for and fires events.

Initially our system is *stable* and free of objects. A *stable* system has no fireable events. User objects may be created through the use of the user interface. Let us consider the case where we have a stable system with some set of object instances in existence. Spontaneously some event fires. This event is presumably the work of a user object, which may "improvise rules" not in its behaviour and may spontaneously change state. A "clock" object may also spontaneously change state, as may any object that speaks to the outside world.

Because an event has been fired, the system may no longer be stable. If some new event has become fireable, it must involve one of the participants of the last event fired. Every object in the event that has undergone a change of contents is placed on a queue and its active rule list is updated. Starting at the head of the queue we take an object and try to construct an event. We take every active rule in turn and try to find acquaintances for it. If the rule needs to be invoked, we must first look for an invoking acquaintance with an active invoking rule. For each acquaintance we must recursively search for its acquaintances. At each step the conditions on values passed between objects must be checked.

If none of the active rules of the object yields a fireable event, we remove the object from the queue. Alternatively, if a fireable event is found, we fire it and add those objects that have changed contents to the end of the queue. This process is continued until the queue is emptied and the system has stabilized. Although there can be no guarantee that the system will ever stabilize we may be certain that every fireable event is found and fired or is deactivated in a finite and reasonable amount of time.

5. The user interface

An OBS might be created in which all interactions among objects take place internally and there is no user input to the system. Objects would be defined at system start-up and all modification of object classes and instances would be done by the system as it runs. While this system may be useful in some applications, it would not be able to effectively model the complex interactions between people and office procedures. A method of allowing people to dynamically manipulate object classes and instances and to send and receive messages in the system is required.

The user interface to an OBS must allow users to define, modify and possibly delete object classes, and it must enable users to create object instances, communicate with them, and destroy them. Since the second function may also be carried out by other objects, it is natural to present this portion of the user interface as though it were an object itself. The behaviour of this "user object" is the code associated with the interface together with the user’s needs and imagination.

In addition to this, a user object may have a set of default rules to handle its automatic behaviour in dealing with other objects that wish to communicate with it. The user interface needn't interrupt the user to satisfy a query about the user's name, for instance.

There are similarities between defining and modifying object classes and creating and interacting with object instances. It may therefore be desirable to model object class definitions as objects themselves, thereby enabling objects to create new object classes. In an existing implementation, this would be a relatively simple matter of linking the code that creates new object instances with the code that creates new classes. The difficulty lies in deciding on an appropriate definition of the "object-class" object class. At present, however, we shall not consider this twist.

5.1. Class manipulation

The manipulation of object classes involves the definition of new object classes and the changing of contents and behaviour in existing object classes. This is done out of view of the object manager and, when complete, the object manager is informed of the manipulation by causing a particular user object rule to become active. When the rule is fired, a token string representing the manipulation is passed to the manager and an object class definition in the system is created or updated.

The user manipulates an object class by editing the class definition. An object class name is entered and, if the class exists, the current definition is edited. If the class does not exist, a new class is created. When the manipulation is complete, the definition is compiled and passed to the object manager.

In creating a class, the user must give a unique class name. If the class is a sub-class of another class, the user must specify the superclass. The contents are then specified as zero or more variables and their types. If initial values are given then each subsequent instance takes on these values when created. If no variables are specified then the class behaves like a function. The behaviour is given by specifying zero or more rules, each with a unique rule name within the object class. Each rule is comprised of an optional set of acquaintances, an optional set of conditions and an optional set of actions. The actions may be assignment of values to variables, sending or receiving of messages, or a set of sub-rules.

The changing of an object class involves altering the class definition. Deletion of a class may require all instances of the class to be first deleted or it may alternatively result in the class definition and all instances being deleted.

5.2. Instance manipulation

Instance manipulation involves the creation and changing of object instances. The user interface must provide the object and rule names and the required parameters. It may guide the user in constructing the event by displaying a template which the user may fill in, or it may expect the user to explicitly indicate the rule to be fired. For example, using the *secret* object described in section 2.1, to create a new instance the user may type *secret.alpha(string, key)*. Alternatively, a template may be presented to the user, who is expected to fill in values for *string* and *key*. When the alpha rule (the initial rule which creates instances) for that object is satisfied, the prepared event is presented to the object manager so that the newly created object can be stored, and other objects may communicate with it.

Changes to existing objects are handled in a similar fashion. The user interface allows the user to choose from the active rules applying to that object to invoke the changes. Object retrieval can be accomplished by indicating conditions to be met and patterns to be matched within the fields of a template. Object deletion is merely a special case of modification. In any case, once the user interface constructs a fireable event that satisfies the user, it presents it to the object manager to fire it.

5.3. The user object

In order to integrate the user interface into the object-based environment, the user is modelled as a special "user object" that is capable of spontaneously communicating with other objects independent of its predefined behaviour. (The alternative, of course, is to design a user object behaviour that captures all possible events that users might ever wish to participate in.) Not every event involving users requires the active participation of a logged-in human being, however. A predefined behaviour enables one to specify certain automatic behaviour. Messages sent to a user object can be automatically answered in some cases and information can be automatically received and stored. Examples of automatic behaviour are the *name* and *newcharge* rules in the simple *user* object described below.

```
user : object {

          /* instance variables */
          charges : integer;
          loginid, pswd, newpswd : string;


          /* rules */
          alpha(id,pw){ /* rule for creating user objects */
                    /* only ’superusers’ can create them */
                    ˜ : superuser;
                    id, pw : string;
                    loginid ← id;
                    pswd ← pw;
                    newpswd ← pw;
                    charges ← 0;
                    }


          name{  /* return loginid to anyone */
                    ˜ : object;
                    }(loginid)


          chgpswd(pw){ /* change user’s password - */
                    /* this is done by specifying the rule   */
                    /* name or by altering the newpswd variable */
                    A : accounting;
                    pw : string;
                    {
                              ˜ : user;
                              ˜ = *;  /* can only talk to self */
                              pw ≠ "";  /* non-null string */
                              newpswd ← pw;
                    |
                              /* newpswd was otherwise altered */
                              pswd ≠ newpswd;
                    }
                    pswd ← newpswd;
                    /* inform accounting object of change */
                    A.pswdupdate(newpswd);
                    }


          newcharge(c) { /* increment charges by c */
                    ˜ : accounting;
                    c : integer;
                    charges ← charges + c;
                    }
          **omega**{ /* only superusers can destroy */
                    ˜ : *superuser*;
                    }
          }
```

In other cases, events initiated by other objects may need to communicate with the user directly. A request for a non-existing rule in the case of a user object is taken to mean that the user is to be prompted (rather than that the request be considered inappropriate). In addition, a customized prompt rule in the user object behaviour may be fired to prompt for user input.

The contents of the user object may be modified without restriction by the user himself (*pswd*), or by other objects (*charges*) through the rules defined. (Values that must not be altered directly by users can be associated with other objects.) The contents may be part of a "virtual screen", a large desk top of which the user interface can present windows to the user [BYTE81, Swin74]. Separate windows would be available for the user to manage different activities, such as defining several new object classes, retrieving and modifying objects and monitoring events. One window could be used, for example, to monitor the mailbox for receipt of fresh mail, and another window might be a clock that displays the current time. Whenever the user logs in, the virtual screen assumes its current incarnation, depending on what events were fired in which the user object was a participant during the user's absence.

A user may define rules that apply specifically to himself. In effect, he becomes a sub-class of the *user* object class. With these rules he may specify automatic replies to prompts or he may customize responses. The user may also, upon request, cause certain rules to become active. These rules could be commands the user wishes to execute and they may be activated by altering a variable in the rule's contents or by specifying the rule name (*chgpswd*). Temporary rules may be specified which fire once and then disappear. (This is equivalent to modifying the class to include the rule, invoking the rule, and then modifying the class to delete the rule.) Temporary rules are useful for defining behaviour that occurs only once.

Useful concepts that are not inherently part of the object model are *ownership* and *location*. Users might, for instance, only be allowed to modify certain kinds of objects that they *own*. This would be accomplished by endowing such objects with ownership fields that are set to a user object's *name* upon the firing of a *mail* rule (in that or some other object). In addition, rules that modify that object's contents could insist that the object modifying it be the *owner* or some other object owned by him.

6. Conclusion

Objects are a natural choice for modelling the behaviour of office information systems. The object model we have presented is capable of expressing many of the components of such systems. Intelligent forms and messages, workstations, office procedures and even users can be defined as object classes. A wide collection of field types can be easily expressed. In addition, several useful frameworks for defining office procedures, such as automata and Petri nets, can be easily simulated.

Although objects do not explicitly model the outside world, it is possible to include special object classes that talk to the rest of the universe. Users communicate with objects in this way. One benefit of this approach is that several different user interfaces could be grafted onto the same system without having to significantly alter the object manager.

Objects naturally capture the three most important aspects of office information systems: information management, communication and partial automation. Information is stored in object instances and managed by rules, communication between objects occur when events are fired, and automation is achieved through rules that may be triggered automatically. Specialization is used for defining special cases of object classes, or for defining superclasses with properties that their specializations have in common.

We are building a prototype OBS at the University of Toronto. Our object model is fairly well developed at this point. We are currently in the design stage of our project, and we expect to have an working version of the system within one year. We have drawn from our experience in building an intelligent message management system [TRGH81]. Our object is to design a programming system in which similar and more powerful systems can be written quickly and painlessly. Existing software can always be accessed through function calls within objects' rules.

Concurrently with the development of this system, we are studying notions of correctness in office procedures. Poorly defined objects may unexpectedly exhibit infinite loops, deadlock and other undesirable behaviour [NiTs82]. Techniques for automatically detecting these anomalies and for presenting users with a view of global behaviour of object systems are being developed.

7. An object grammar

The object grammar that follows is only a skeleton. There are, for example, no special keywords like "alpha" or "null", no built-in arithmetic expressions, and we have not included pattern-matching in our conditions.

```
<object> ::= <object-class> [ ":" <super-class> ] "{"
             { <declaration> ";" }
             { <rule> }
             "}"
<declaration> ::= <variable> { "," <variable> } ":" <type>
<rule> ::= <rule-name> [ "(" [ <variable> { "," <variable> } ] ")" ]
             "{" { <statement> ";" } "}"
             [ "(" [ <variable> { "," <variable> } ] ")" ]
<statement> ::= <declaration>
             | <condition>
             | <send>
             | <assignment>
             | <sub-rule>
<condition> ::= <variable> <comparator> <expression>
<comparator> ::= "="
             | "≠"
             | "<"
             | "≤"
             | ">"
             | "≥"
<send> ::= <acquaintance> "." <rule-name>
             "(" [ <expression> { "," <expression> } ] ")"
<acquaintance> ::= <variable>
             | "<" <object-class> ">"
<assignment> ::= <variable> "←" <expression>
<expression> ::= <value>
             | <variable>
             | <function> "(" [ <expression> { "," <expression> } ] ")"
             | <send>
<sub-rule> ::= "{"
             { <statement> ";" }
             { "|" { <statement> ";" } }
             "}"
<rule-name> ::= <identifier>
<acquaintance> ::= <identifier>
<type> ::= <identifier>
<object-class> ::= <identifier>
<super-class> ::= <identifier>
<variable> ::= <identifier>
```

<function> ::= <identifier>

References

[AtBS79] Attardi, G., Barber, G. and Simi, M., "Towards an Integrated Office Work Station", AI Laboratory, MIT, Cambridge, 1979.

[BaHe80] Barber, G. and Hewitt, C., "Research in Office Semantics", MIT, Cambridge, 1980.

[BySJ82] Byrd, R.J., Smith, S.E and de Jong, P, "An Actor-Based Programming System", SIGOA Conference on Office Information Systems, SIGOA Newsletter Vol. 3 No. 1 and 2.

[BYTE81] Special issue on Smalltalk, Byte Vol. 6, No. 8, Aug. 1981.

[Cook79] Cook, C.L., "Streamlining Office Procedures", System Sciences Lab, Office Research Group, SSL79-10, Xerox PARC, Nov. 1979.

[deJo80] deJong, P., "The System for Business Automation: A Unified Application Development System", IBM Research Report #34539, Yorktown Heights, N.Y., 1980.

[ElNu79] Ellis, C.A. and Nutt, G.J., "Computer Science and Office Information Systems", Xerox PARC, June 1979.

[Embl80] Embley, D.W., *A Forms-based Nonprocedural Programming System*, Department of Computer Science, University of Nebraska-Lincoln, 1980.

[FiHe80] Fikes, R.E. and Henderson, D.A., "On Supporting the Use of Procedures in Office Work", MIT workshop, Cambridge, 1980.

[Fike81] Fikes, R.E., "Odyssey: A Knowledge-Based Assistant", Artificial Intelligence, Vol. 16, No. 3, July, 1981.

[Geha82] Gehani, N.H., "The potential of forms in office automation", IEEE Transactions on Communications, Vol. 30, No. 1, January 1982.

[Gibb82] Gibbs, S., "Office Information Models and the Representation of ’Office Objects’", SIGOA Conference on Office Information Systems, SIGOA Newsletter Vol. 3 No. 1 and 2.

[HaKu80] Hammer, M. and Kunin, J.S., "Design Principles of an Office Specification Language", NCC 1980.

[Hewi77] Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages", Artificial Intelligence, Vol. 8, No. 3, pp. 323-364, June 1977.

[HoNT81] Hogg, J., Nierstrasz, O.M. and Tsichritzis, D., "Form Procedures", in *Omega Alpha*, Technical Report 127, Computer Systems Research Group, University of Toronto, 1981.

[LaTs80] Ladd, I. and Tsichritzis, D., "An Office Form Model", Proceedings NCC, Anaheim, 1980.

[LuYa81] Luo, D. and Yao, S.B., "Form Operation By Example", Department of Computer Science, Purdue University, W. Lafayette, Indiana, 1981.

[MaPa77] Malhotra, A. and Parkman, J., "A System for the Automation of Almost-Routine Functions", IBM Research Report #27769, Yorktown Heights, NY, 1977.

[Morg78] Morgan, H.L., "Control and Tracking of Office Documents", MIDCON Proceedings, Dallas, Texas, 1978.

[NiTs82] Nierstrasz, O.M. and Tsichritzis, D., "Message Flow Modeling", *Alpha Beta*, Technical Report 143, Computer Systems Research Group, University of Toronto, 1982.

[Pete77] Peterson, J., *Petri Nets*, ACM Computing Surveys Vol. 9, No. 3, pp. 223-252, Sept. 1977.

[Pott78] Potts, D., *Specifications Language for Office Procedures Execution*, Thesis, The Wharton School, University of Pennsylvania, Philadelphia, 1978.

[Swin74] Swinehart, D.C., "Copilot: A Multiple Process Approach to Interactive Programming Systems", Stanford Artificial Intelligence Laboratory Memo AIM230, Stanford University, July 1974.

[SSKH81] Sirbu, M., Schoichet, S., Kunin, J. and Hammer, M., "OAM: An Office Analysis Methodology", MIT Office Automation Group Memo OAM-16, Cambridge, 1981.

[TRGH81] Tsichritzis, D., Rabitti, F.,Gibbs, S., Hogg, J., Nierstrasz, O., and Kornatowski, J., "A Message Management System", IEEE Transactions on Communications, Vol. 30, No. 1, January 1982.

[Tsic81] Tsichritzis, D., "Form Management", Communications of the ACM, July 1982.

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