Office Object Flow

Oscar Nierstrasz
Dennis Tsichritzis

Department of Computer Science
University of Toronto

ABSTRACT

Office information systems provide facilities for automatically triggering procedures when certain conditions become true or particular events take place such as receipt of mail. When these procedures operate concurrently and independently in a common environment, the overall behaviour of the system may be unexpected. *Firing expressions* are proposed as a tool for describing global behaviour and for detecting unusual properties of the system.

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, 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.

The messages passed between workstations in an OIS may be intelligent forms, procedures, or any appropriate office object. Automation in such systems 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 circumstances may trigger an event a user passes the responsibility of firing it to the system.

The firing of an event may create a situation in which some other event is triggered. All such new events are then also fired. This process continues until the system "stabilizes" and no more events can be fired. Since an event may be triggered implicitly rather than called explicitly, the net effect of firing an event may be far from obvious, in fact, the system may never stabilize if events trigger one another in a cycle or generate new events. This highly parallel activity can be difficult to understand and evaluate unless there is a view of global behaviour available.

The task of describing global behaviour is a general one, independent of the specific OIS application being studied. A model is needed to capture this behaviour without being bound to a particular class of applications. We cannot enumerate and anticipate all possible applications. We approach the problem at a different level by providing the primitives needed to define such systems. Any results obtained from our analysis of global behaviour could then be embedded in a system based upon such a model.

2. Objects

Objects in an OIS can be memos, forms, records, messages or anything else that may be manipulated by a user. The objects that are of greatest interest here are those that are fairly regular in their appearance and in their usage. Such objects may be automatically processed by procedures associated with user workstations that await their arrival, creation or modification. These procedures may look for objects that satisfy certain conditions, or they may coordinate objects that belong together. Different applications and different

---

In *Beta Gamma*, Technical Report no. 150, Computer Systems Research Group, University of Toronto, 1983.

implementations may have different characteristics. There is still enough regularity of information and common triggering of procedures. It is possible to describe these systems with a model. "Objects" are a construct that can be used to model or to program such systems. An electronic form together with its characteristic behaviour could be coded as an object, as could messages, procedures, mail trays and even parts of the user interface. An "object" [BYTE81, DeJo80, NiMT83] is an entity with a set of properties and a *behaviour* that describes how the object may be used and modified. An object "knows" its behaviour and is responsible for it, in a sense.

An *object-based system* enables one to represent 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. An object's rules may 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 object-based system enforces the behaviour by detecting conditions that will trigger events. This is the task of *managing* a set of objects. 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.

Objects interact by "doing things" to each other that "make sense". Certain actions may only make sense for objects in a particular state. One can only fix a chair, for example, if it is in a "broken" state. When two or more objects interact, something "happens", that is, the properties of some of the objects may change.

Objects have both contents and behaviour. Contents consist of *instance variables* which distinguish objects in the same class somewhat like attributes distinguish tuples in the same relation. They may also be compared to the static variables of a module. Throughout an object's lifetime these contents may change. The values of all the objects in the system determine the state of the system. Value changes may occur because certain conditions become true, or they may take place as a consequence of one object interacting with another object. An object's behaviour is a set of *rules* describing the situations under which these value changes may normally occur.

Objects' contents are simply a set of variables, each of which ranges over some set of values. They may be stored in data structures such as those provided by a high level programming language like Pascal or C. A rule is a named body of code. Rules specify trigger conditions, communicate with other objects and modify contents.

The *firing* of an object's rule is the execution of its statements. A rule is *fireable* if its statements may be successfully executed. A rule may only fire if all the conditions in the rule are guaranteed to hold upon firing. This eliminates the need for "postconditions" and the idea of "success" or "failure" of a rule. A rule is *active* if the conditions that refer to its instance variables alone are true.

If for any reason a rule is not fireable, then *none* of its statements are executed.

An *event* can be fired whenever there exists a collection of objects that wish to communicate through rules whose trigger conditions are all true. An event is fired by firing all the rules in that event. We assume that every fireable event eventually either (1) get fired or (2) get deactivated by the firing of some other event. In [NiMT83] we give a more thorough discussion of objects and outline an implementation. In the appendix we give an example of an object.

3. Correctness and global behaviour

Objects as they are described above suggest a number of questions. First of all, there is nothing in an object's behaviour that tells us what an event looks like. We only see an object's immediate acquaintances, not the entire set of participants of an event. We may also not be able to tell from a rule's reference to acquaintances exactly what classes of objects are matched, or precisely what rules within those objects will be matched. This is analogous to not knowing what procedures are ultimately going to make use of a library of functions.

Furthermore, the overall behaviour of the system we have defined may not correspond to what we had in mind. This is analogous to the problem of debugging a program. The problem is more difficult here because we are not defining programs to be run at a single instant in time, but rather a set of cooperating programs that have an extended lifetime. The interactions between existing objects and newly-defined ones may be difficult to determine and comprehend in an OIS of any complexity.

In a running system, it may be desirable to monitor the progress of the objects currently defined. Suspicious behaviour may be detected by identifying those objects which are not proceeding at an expected rate.

Some automatic analysis is required that will (1) provide users with a view of global behaviour and (2) point out any behaviour that is unusual. There can be no *a priori* bad behaviour because an OIS is necessarily not a closed system. What may look like deadlock may in fact be a way of *preventing* certain events from taking place except under very unusual circumstances.

We assume that all objects are meant to be created, go through a series of transformations, and then peacefully die. We may be able to detect the possibility of objects being frustrated in their attempts to fulfill this goal.

The creation of an object is accomplished by firing an *initial rule* (called *alpha*) which brings it from the *initial state* (or *alpha state*) to a state in which its contents assume some value. An object eventually is destroyed when a *final rule* (called *omega*) is fired, bringing it to its *final state* (or *omega state*), which causes the object to be archived, printed, forgotten, or otherwise leave the set of active objects. An object is *stuck* if it is prevented from reaching a final state. An object may get stuck

(1) if it reaches a state in which no rule is active

(2) if it cycles infinitely through the same set of states,
never reaching a final state

(3) if it reaches a state in which it waits forever for an acquaintance to reach a certain state because that acquaintance is stuck or because that state is unattainable or is simply never reached.

An analysis of of objects' behaviours can also tell us

(4) what states are unattainable and

(5) whether there are rules that can never fire because the object can never reach a state in which those rules are active.

Our task is to abstract our object model to a simpler model that captures state changes and firing sequences. This abstraction must then be interpreted to evaluate the global behaviour of an OIS.

4. Object states

We are concerned primarily with the possible life histories of objects, that is, the sequence of states a objects reaches and the sequence of rules fired from state to state. A *state* is some set of values that an object may (or may not) reach. If states are blocks in some partitioning of the range of an object's contents, then we need some criteria for establishing the partition. Since rules are the only means by which objects may change state, an object's state must be related to whether a rule is fireable.

The simplest possible state space would consist of a single state. Since we are interested in life histories, we would naturally add the initial and final states. One might add one state per rule -- an object being in a particular state if that rule is active, but several rules may be active at any one time, and an object may be in only one state at a time.

The first reasonable state space might be to consider 2^n states where there are n rules, since each rule may be either active or inactive. Some states would, however, correspond to no objects since we may expect some rules to be mutually exclusive. In the *inv* object outlined in the appendix, excluding the *alpha* and *omega* rules, either all rules are active or all rules are inactive.

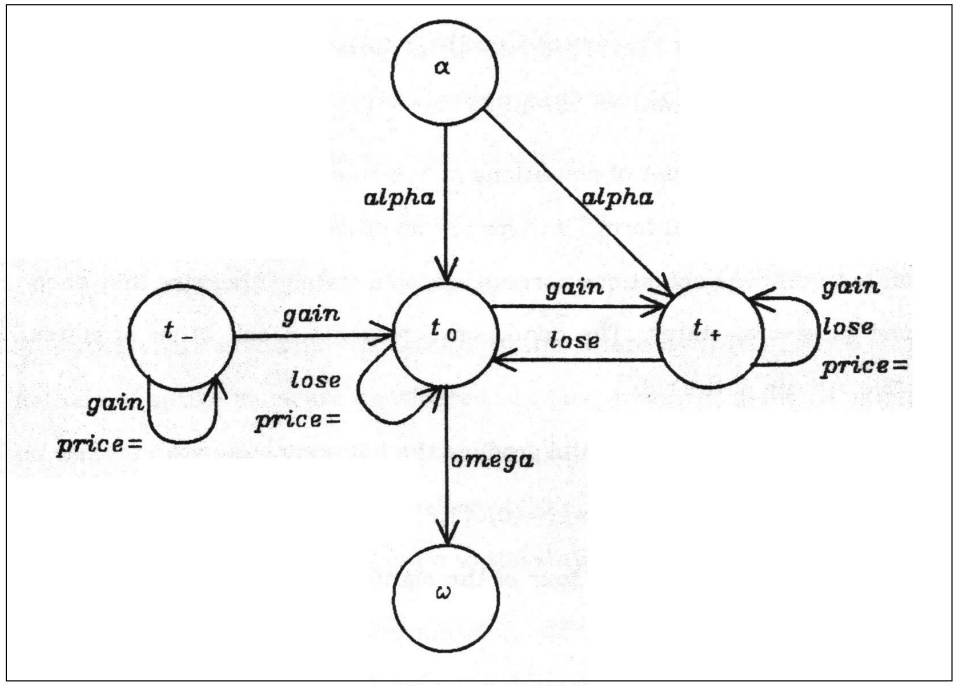
A more reasonable approach would be to consider all the conditions placed on the instance variables of an object. An object is in a given state if it satisfies that set of conditions. The conditions in the

behaviour of the *inv* object are *total*=0 in the *omega* rule, and *total*>0 in the *lose* rule (in the first alternative of the sub-rule, *total*≥*n* where *n*>0). We have, then, the states α, the initial state, t_0 (where *total*=0), t_+ (where *total*>0), t_- (where *total*<0), and ω, the final state.

The significance of states is that they capture what rules may be active for an object. Rules map objects from states to states and consequently cause different rules to become active. An object's *state graph* is a digraph with nodes representing states and arcs representing state transitions. For every arc there must be at least one rule active in the state at the arc's tail which, when fired, may cause the object to reach the arc's head. A state graph may be simplified by combining states. For example, a collection of states with identical out-going arcs may be combined since they share the same next-state set.

The *dicomponents* of the state graph are the sets of mutually reachable states. Since these states are mutually reachable, an object may loop infinitely through the states in a dicomponent *unless* the dicomponent contains only a single state and there is no loop from that state to itself. The dicomponents of the *inv* state graph are $\{\alpha\}$, $\{t_0, t_+\}$, $\{t_-\}$ and $\{\omega\}$. Any dicomponent without outgoing arcs is *terminal*. If this dicomponent is not the omega state, then it is a *dead* dicomponent since it can never reach the omega state. Any dicomponent without incoming arcs is *unattainable*, except the alpha state. ($\{t_-\}$ is an unattainable dicomponent). Any dicomponent whose only incoming arcs are from an unattainable dicomponent is also unattainable. Any dicomponent whose only outgoing arcs are to a dead dicomponent is also dead. Any state in a dead dicomponent is a *dead state*. All other dicomponents are *live*.

A rule is composed of statements. The condition for the rule is the conjuction of all its constituent conditions. A sub-rule may contain alternatives. The condition for the sub-rule (a statement) is the disjunction of its constituents.



A state graph for *inv* objects

eg.

```
{       x > 1;
        x < 5;
        { y ≥ 0; y < 1 | y > 2};
        { z < 0 | z > 10};
}
```

The condition for this rule (which contains no executable statements) is

$$(((x > 1)\wedge(x < 5)\wedge(y \geq 0)\wedge(y < 1)\wedge(z < 0))\vee$$

$$((x > 1)\wedge(x < 5)\wedge(y \geq 0)\wedge(y < 1)\wedge(z > 10))\vee$$

$$((x > 1)\wedge(x < 5)\wedge(y > 2)\wedge(z < 0))\vee$$

$$((x > 1)\wedge(x < 5)\wedge(y > 2)\wedge(z > 10))).$$

We must take a cross product of conditions in sub-rules to obtain our rule condition in disjunctive normal form. If there are no other conditions in any other rule, then each of these conditions corresponds to a state. Otherwise they each correspond to several states. The condition represents a collection of states which is the *domain* of the rule.

In this case our conditions would produce the following state space:

$\{(1,5),(-\infty,1)\cup[5,\infty)\}\times\{[0,1),(2,\infty),(-\infty,0)\cup[1,2]\}\times\{(-\infty,0),(10,\infty),[0,10]\}$

The rule above would be active in four of the eighteen states. Here we end up with too fine a partition of objects' states by considering all combinations of all conditions. It would therefore be desirable to combine certain states if they are in some sense equivalent. Non-empty intersections of rule domains and their complements might determine a better state space. In this case the four states would be combined into one state and the remaining fourteen would be combined to produce another state.

5. Formal abstraction

An object class C has a set of variables $V(C)$, a domain $D_C(v)$ for each variable $v\in V(C)$, and a set of rules $R(C)$. The *instances* of C are the mappings $I(C)=\{\mu|\mu(v)\in D_C(v)\forall v\in V(C)\}$. The rules $R(C)$ are a relation over the instances of C (*i.e.* a rule need neither be well-defined nor totally defined). We model, therefore, whether a rule is active and what states may result from firing it, but we do not (yet) model those firing conditions that depend on outside information (*i.e.* the state of an acquaintance).

The precondition for firing a rule $r\in R(C)$ is that the instance $\mu\in I(C)$ belong to the domain of the rule $p(r)=\{\mu\in I(C)|r(\mu)$ is defined$\}$.

Class C_i is a specialization of C_j if $V(C_i)\supseteq V(C_j)$. We write $C_i \geq C_j$. The base object class is C_0 where $V(C_0) = \varnothing$ and $R(C_0) = \varnothing$. We have therefore $C_i > C_0\forall i$. Specialization is a partial order on object classes.

A state space $S(C)$ may be defined as follows:

$$S(C) = \{s\subseteq I(C)|s = (\bigcap_{r\in X} p(r)) \bigcap (\bigcap_{r\notin X} I(C)\setminus p(r)) \neq \varnothing, X\subseteq R(C)\}$$

This state space would have at most 2^n states where $n = |R(C)|$, since all combinations of active rules are considered. In fact, however, many rules will have no associated conditions, such as the *total* rule, for which $p(total) = I(C)$. Other rules may be mutually exclusive. Many of the states so generated will therefore be empty and so will not contribute to the state space $S(C)$.

A *firing expression* [NiTs82, Shaw78] is a regular expression representing the set of all possible rule firing sequences and state changes for an object. Since objects generally require acquaintances for their rules to fire, one must examine the firing expression of possible acquaintances to decide whether they may ever reach a state in which the matching rule is active. An iterative algorithm may be used to sort through

all firing expressions to decide if the rules in the sequences may actually become fireable by finding a matching rule in another firing sequence.

The firing expression for an object class corresponds to the set of strings accepted by the automaton $A =< S, R, \delta, \alpha, \omega >$ with states S, alphabet R, initial state α, final state ω and state change function δ where $s' \in \delta(s, r) \Leftrightarrow s' \bigcap r(I) \neq \emptyset$. If the firing expression is to include state changes as well as rule firings, then the alphabet is $R \times S$, and $s' \in \delta(s, (r, s')) \Leftrightarrow s' \bigcap r(I) \neq \emptyset$.

The firing expression including state changes for the *inv* object is

$$\alpha\ alpha(t_0 + t_+((gain + lose + price =)t_+)^*\ lose\ t_0)$$

$$((lose + price =)t_0 + gain\ t_+((gain + lose + price =)t_+)^* lose\ t_0)^*\ omega\ \omega.$$

(Since the *item, total* and *price* rules do not cause any state changes, they have been left out.) If state changes are ignored, this expression simplifies to

$$alpha(gain(gain + price =)^* lose + lose + price =)^* omega.$$

Firing expressions may be used to generate *flow expressions*. If certain kinds of objects have an *owner* attribute indicating who may modify it at any given time, then certain states will correspond to ownership by a given workstation. The firing expressions may be used to determine what sequences of workstations may own the object, *i.e.* how the object *flows* through the network.

Non-determinism may enter the model in 3 different ways: (1) The state space may not be fine enough to capture all state transitions accurately (an increment of a variable may or may not cause a state change) (2) Incoming data from acquaintances or functions cannot be predicted (3) A genuinely non-deterministic source (a random number generator or user input) may be used.

If rules require no acquaintances, if conditions are simple comparisons of variables to constants and if firing a rule will only set variables to constants, then firing expressions are completely deterministic. If more general actions are permitted and variables may be set to values returned by functions, then firing expressions become regular expressions, and non-determinism is introduced.

6. Conclusions

OISs can exhibit quite complicated behaviour when automatic procedures are introduced. Global office activity that manages objects, e.g. messages, can be hard to understand when independent local procedures vie for control. Techniques for describing and analysing global behaviour are needed as a means to determining whether these systems behave "correctly". Some kinds of unusual behaviour can be identified. Firing expressions are suggested as an approach to studying global behaviour and detecting peculiarities that are of interest to users and programmers.

7. Appendix

A specification for an object class *inv* follows. Instances are inventory objects used to keep track the number of items in stock. We use the notation outlined in [NiTM83].

The contents of the *inv* object are the variables *item, total, price, auth* and *date*. Its behaviour consists of the rules *alpha, item, total, price, price=, gain, lose* and *omega*. The *inv* object is a specialization of the null object class *object*, which has no contents or behaviour. Every object must have an *alpha* rule and an *omega* rule for creating and destroying it.

The conditions appearing in a rule may refer to the object's contents, or to any value sent by an *acquaintance*, an object interacting with it. The acquaintance variables in the above rules are ~ and *B*. (~ is the object invoking the rule, if there be one.) The rule may also produce values to be sent to acquaintances or to be used in modifying the object's contents. The *alpha* rule above obtains the *name* of its one acquaintance for setting one variable, and it gets the current time by calling a globally defined *time* function.

Rules may be named like the procedures of a module or an abstract data type, but they may also be triggered without having to be explicitly called by another object. Consider the following rule added to the behaviour of *inv*:

```
panic{
        M : manager;
        total = 0;
        auth = M.name();
        M.panic(item);
        }
```

This rule would be triggered whenever the *total* field hit zero. There is no ˜ acquaintance for this rule (and hence no need to name it). This would be useful for separating triggering conditions from the events that cause them. There are, for example, two distinct ways in which *total* could be set to zero. There is no need to check this condition at those points. Note, however, that the panic rule can only be deactivated by the *gain* or *omega* rules, and so would fire repeatedly until one of these other rules fired.

```
inv : object {

        /* instance variables */
        item, total, price : integer;
        auth : string;
        date : time;


        /* rules */
        alpha(i,t,p){ /* rule for creating inv objects */
                /* only managers can create them */
                ˜ : manager;
                i, t, p : integer;
                /* conditions on the input */
                i > 0;
                t ≥ 0;
                p ≥ 0;
                item ← i;
                total ← t;
                price ← p;
                /* get the creator's name by invoking his name rule */
                auth ← ˜.name();
                date ← time();
                }

        item{
                /* return item to anyone who wants it */
                ˜ : object;
                }(item)

        total{
                /* return total to anyone who wants it */
                ˜ : object;
                }(total)
```

```
price{
        /* return price to anyone who wants it */
        ~ : object;
        }(price)

price=(p){
        /* only managers can change the price field */
        ~ : manager;
        p : integer;
        p ≥ 0;
        price ← p;
        auth ← ~.name();
        date ← time();
        }

gain(n){
        /* increment the total number of items */
        ~ : object;
        n : integer;
        n > 0;
        total ← total + n;
        }

lose(n){
        /* fill an order; return the amount filled */
        ~ : object;
        n : integer;
        n > 0;
        {
                total ≥ n;
                /* the usual case */
                k ← n;
                total ← total - n;
        |       /* alternatively: */
                /* can't fill order; create a backorder */
                B : backorder;
                k ← total;
                B.alpha(~,item,n-k);
                total ← 0;
        }
        /* return the size of the order filled */
        }(k)

omega{
        /* only managers can destroy inv objects */
        ~ : manager;
        /* can't have any leftover stock */
        total = 0;
        }
}
```

8. Bibliography and references

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

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

deBy80 deJong, P. and Byrd, R.J., "Intelligent Forms Creation in the System for Business Automation", IBM Research Report, Yorktown Heights, N.Y., 1980.

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

Elli79 Ellis, C.A., "Information Control Nets", Proceedings of the ACM, Conference on Simulation, Measurement and Modification, Boulder, Colorado, Aug. 1979.

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

FiHe80 Fikes, R.E. and Henderson, D.A., "On Supporting the Use of Procedures in Office Work".

Gisc81 Gischer, Jay, "Shuffle Languages, Petri Nets, and Context-Sensitive Grammars", Communications of the ACM, Vol. 24, No. 9, September 1981.

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

Macq79 MacQueen, D.B., "Models for Distributed Computing", INRIA Research Report #351, Domaine de Voluceau, Rocquencourt, Le Chesnay, France, April 1979.

NiMT83 O.M. Nierstrasz, J. Mooney, K. Twaites and D. Tsichritzis, "Using Objects to Implement Office Procedures", Technical Report 150, Computer Systems Research Group, University of Toronto, 1983.

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

Shaw78 Shaw, Alan C., "Software Descriptions with Flow Expressions", IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, May 1978.

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" in *Omega Alpha*, Technical Report 127, Computer Systems Research Group, University of Toronto, 1981.

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

Zloo80 Zloof, M.M., "A Language for Office and Business Automation", IBM Research Report #35086, Yorktown Heights, N.Y., Jan. 1980.