Information and Software Technology 1995 **37** (4) 203–211

MGA: Rule-based specification of active object-oriented database applications

Pete Sawyer and Ian Sommerville

Computing Department, Lancaster University, Lancaster LA1 4YR, UK

e-mail:sawyer@uk.ac.lancs.comp

This paper describes a model for developing applications of active object-oriented databases based on three orthogonal concepts; methods, guards and actions and a tool which implements these ideas called MGA. MGA objects are active; they have guards which constrain the object's state, actions which maintain database integrity when the object's state changes, and methods through which the object provides other services. These active components are implemented as (collections of) rules and rules are in turn modelled as objects.

Keywords: active databases, object-oriented databases, rule-based, forms

This paper describes MGA (Methods, Guards and Actions); a tool for the declarative specification of applications of an active object-oriented database (OODB). In MGA, an application is defined as an object class encapsulating attributes and methods. Application classes are active in the sense that they embody procedural semantics through methods and through actions triggered by write operations on attributes. In addition, guards constrain the conditions under which attributes can be accessed. Methods, guards and actions are all rule based and MGA allows an application class, including its behavioural semantics to be defined by form-filling.

Methods, guards and actions, along with the notion of inheritance, are the major concepts which a user of MGA needs to know. MGA conceals the low-level details of the data model. It contains libraries of method, guard and action classes which can be instantiated and composed to form applications. Because applications are modelled as classes, and because classes are themselves objects (instances of meta-classes) they are stored in the database, making them easily retrievable and tractable to reasoning.

In the remainder of this paper we present a brief overview of active OODBs, followed by an introduction to the MGA programming model and a detailed examination of a small application's implementation. We conclude with a brief discussion of some of the issues surrounding the integration of rules and OODBs.

Deductive OODBs

Deductive databases are those in which data has the ability to react to stimuli. One of the most useful features of deductive databases is the ability to define integrity constraints for entities and to have these maintained dynamically in response to changes to the data. Such a mechanism can be used to augment typing schemes (perhaps by defining a constraint which imposes a subrange on an entity's type) and is also capable of expressing relationships such as one-to-many mappings between entities[1]. Active[2] databases provide the additional capacity to define arbitrary dynamic behaviour triggered by events such as state changes.

This reactive behaviour is often encoded as rules of the form <predicate, body> where the predicate represents some event and the body specifies some action to be taken in response to (triggered by) that event. For example, a change to one item of data may automatically propagate changes to other items of data to maintain a specified relationship between them. Another example might be where an attempted write operation is aborted if the condition specified by the predicate is not satisfied.

An attractive feature of deductive and active databases is that the semantics governing evolution of data are part of the data. This contrasts with passive databases where access rights to data are mediated only through the standard application interface of the DBMS with any additional semantics having to be defined in all the applications which access the data. As such applications are necessarily external to the DBMS, change to the data model and/or the application(s) can easily lead to mutual inconsistency.

The general problem of capturing data semantics within

the database is one of the motivations behind the development of OODBs[3]. Here, data is modelled as objects with access to objects and their components strictly prescribed by the interface offered by that object. Object-orientation is promoted as the answer to many of the problems in 'traditional' DBMSs such as the impedence mismatch between application and database data models. However, the maintenance of relationships between objects or between object properties is supported in a quite different way than through database integrity rules.

Access to objects' state, to read or write an attribute, is provided in most OODBs by attribute access methods. In those systems which emphasize information hiding (such as Vbase[4] and its successor, Ontos), the mechanism used to constrain an attribute's access is to overload the standard attribute access method by writing a new method specific to that class (and its subclasses). This is a low-level mechanism which contrasts with the philosophy of declaratively specified database rules monitoring database events and being fired whenever their predicate parts become true.

Consider the case of a derived value method[5] which returns a result calculated from the value(s) of an object's attribute(s). The method will calculate and return this value on receipt of the appropriate message. What is lacking in most OODBs is a mechanism for generating that message internally as a result of a change to one of the attributes from which its value is derived. The integrity relationship is one-way only; the attributes are unaware of any dependency upon them. This poses problems for OODB applications which access data through fine-grained transactions but which nevertheless require their copy of the data held in memory to be consistent with the data in the database at all times.

For example, an application which displays an object of the type just described and allows users to interact with it may, on receipt of a user-generated event, forward a message to the object resulting in a change to the state of an attribute upon which the derived value method depends. The application's problem is that it does not know that the change to the attribute renders the user's representation of the object inconsistent because the value returned by the method when it was invoked *before* the change to the attribute is now wrong.

Part of the problem is the perceived mismatch between the object-oriented and rule-based or declarative programming paradigms. There are, however, OODBs which provide declarative database programming languages, for example Adam[6], Kiwi[7,8] and Mokum[9]. Each one of these systems is deductive in that they have the notion of rules and internal events. In addition, Medeiros[10] has implemented integrity rules in O_2[11]; an OODB whose database programming language is procedural (a flavour of C). It can be seen therefore that active rules can co-exist with object-orientation in database systems. In addition, Adam and Medeiros's work demonstrate that the object model can be exploited by modelling rules as objects.

This is the approach taken by MGA where we also distinguish between attribute access methods and other, general methods. General methods are define as collections of rule objects and are invoked by the receipt of the appropriate message. Attribute access methods are fixed but may be augmented by rules acting as pre and post conditions; guards to filter out illegal update attempts, and actions propagate consistency updates on the successful completion of an update.

The MGA programming model

An MGA application is an object which the user parameterizes by assigning values to its attributes and invokes by sending it messages. The programmer specified the application as an object class which encapsulates a set of services (methods invoked on receipt of a message) and a state space (attributes). Write access to attributes is constrained by guards which intercept requests to assign or delete values and disallow the update if some specified condition is not satisfied. Actions are methods triggered by the successful assignment or deletion of attribute values. Methods, guards and actions specified for an object class are inherited by subclasses but may be overloaded. MGA applications are first-class objects and are therefore persistent so application states can be stored in the database.

An example MGA application: *bbe*

Before describing methods, guards and actions in more detail, we illustrate MGA by describing how an interactive application called *bbe* (browse-by-example) has been implemented. *Bbe* is an object class which mimics, in an object-oriented manner, by-example tools of the QBE[12,13] family. *Bbe* allows users to retrieve and manipulate database objects. The user supplies the name of the class in which they are interested and *bbe* returns the set of all the instances of that class. The user can then further constrain this set by filling in a template of the class and *bbe* will find all the objects which match the template. For example, *bbe* allows a user to find all objects of the *expenses* class whose *payee* attributes have the value 'F.Smith'.

Our MGA implementation uses a dynamic forms user interface metaphor (described in Sawyer and Sommerville[14]) in which objects are represented as graphical forms. Attribute update messages are invoked by typing values into fields labelled with attribute names, and other methods are invoked by 'pressing' buttons labelled with method selectors. 'Set' attributes (those which accept multiple values) are indexed with the total number of values and the index of that currently displayed, indicated numerically beneath the attribute name. Additionally, a set of values may be viewed in a scrollable pop-up window (invoked from a pull-down menu attached to the attribute name).

As shown in Figure 1, *bbe* consists of the following components:

Attributes

Bbe has three attributes which allow the user to supply the required class name and which return references to instances of the class:

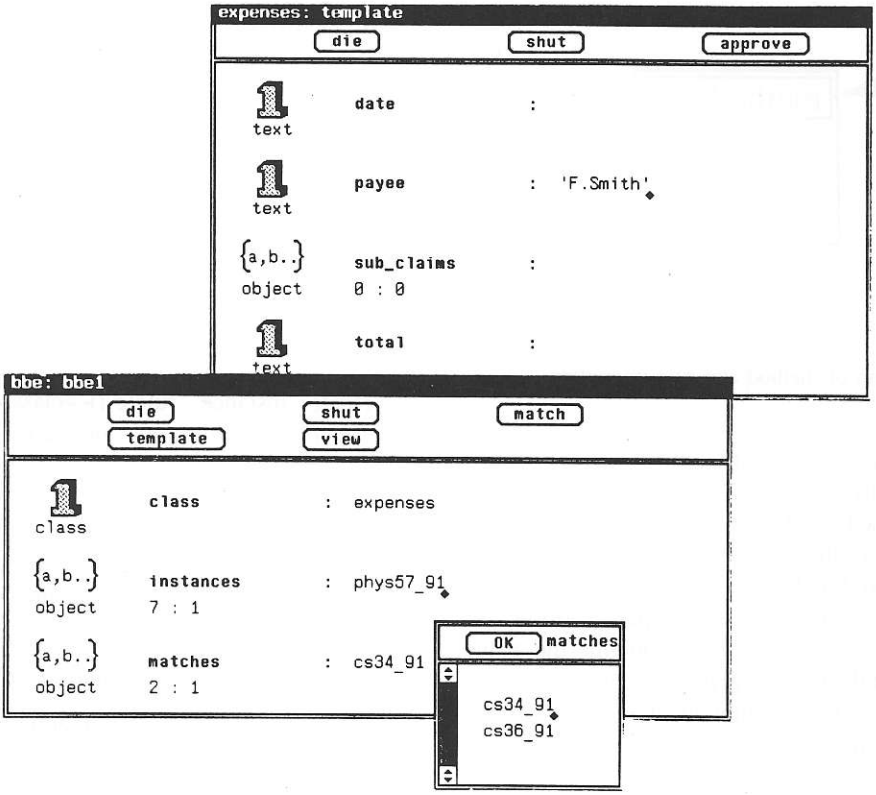

Figure 1 Interacting with *bbe*

- *class:* Accepts the name of an object class. The user has write access to this attribute but a guard rejects values which are not class names. When a valid class name is assigned an action automatically generates values of *instances*.
- *instances:* Accepts references to every object which is an instance of the class represented by the value of *class*. Guards restrict external access to be read only and constrain values to be references to instances of the class specified in *class*. The action attached to *class* finds all the instances of the class and assigns the set of (references to) these to *instances*.
- *matches:* Accepts references to instances of the class represented by the value of the *class* attribute but which also match a 'template' object of the same class (created by the *template* method). The set of values is a subset of those of the *instances* attribute. Again, write access is strictly local. Values are assigned by the *match* method.

Methods

Bbe has six methods which are used for browsing and manipulating the objects referenced by the attributes. The first three methods listed below are common to all objects with which the user may interact directly:

- *die:* Deletes the instance of *bbe* from the database.
- *open:* Displays the *bbe* object at the user interface.
- *shut:* Removes the *bbe* object from the user interface. The object persists within the database.
- *template:* Creates an instance of the object class specified in the *class* attribute. For example, if *class* contains the value *expenses*, then template will create a new instance of *expenses* to which the name 'template' is assigned and whose attributes contain default values.
- *match:* Once a template object has been created by the *template* method, the user can use it to supply 'example' values to its attributes in order to select the set of objects which have identical values. An unassigned attribute in the *template* is taken as a 'don't care' condition. The *match* method performs the comparison between the template and its sibling objects, assigning references to matching objects to the *matches* attribute in *bbe*. For example, if the user is interested in all expenses claims from F.Smith, then *match* will return references to all instances of expenses whose *payee* attribute contains the value 'F.Smith'.
- *view:* Sends an *open* message to a selected object referenced by the *matches* attribute.

Using *bbe* typically entails the following order of actions: A class name is typed into the *class* field and references to all its instances are returned via the *instances* field. The *template* method is invoked and a template object is created. The user types example values into this object's fields and invokes *bbe*'s *match* method. The subset of objects which match the template are returned via the *matches* field and these can be opened using *bbe view* method as required.

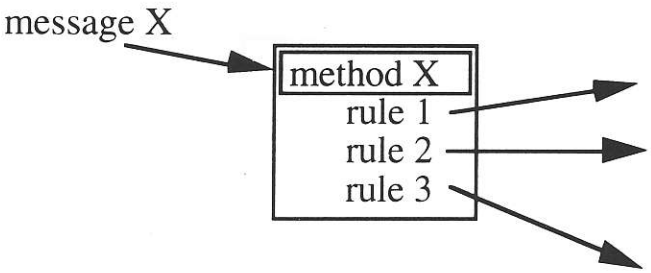

Figure 2 General form of method and rule invocation

Figure 1* illustrates an instantiation of *bbe* (forms are labelled with the corresponding object's id, in this case *bbe:bbe1*) being used to browse *expenses* objects. A template for *expenses* objects (labelled *expenses:template*) has been parameterized and *bbe*'s *match* method has been invoked to select the two instances of expenses (*cs34_91* and *cs36_91*) whose *payee* attribute contains the value 'F.Smith' from a total of seven *expenses* objects.

Having illustrated *bbe*'s implementation, we now describe the three main components of MGA's programming model.

Methods

MGA adopts a conventional view of methods but its method implementation mechanisms are novel. In most object-oriented languages, methods are indivisible sequences of instructions executed on receipt of a message. Methods in MGA are composed of discrete, individually executable components. They allow applications to be prototyped using a small set of high-level building blocks.

MGA views methods as sets of <predicate,action> pairs called rules. The set of a method's rules is ordered in a *script*. The first rule's predicate is a message requesting the method's invocation. Subsequent rules' predicates are defined as the successful termination of the preceding rule's execution. Because rules are modelled as objects, they are typed (as classes); the programmer builds a rule script by parameterizing instances of rule classes and appending them to the script. When a message is received by an object, the appropriate method script is executed, each rule being evaluated in order. Figure 2 illustrates the general form of a method invocation where arrows represent messages.

For example, *bbe*'s *template* method contains two rules. The first rule is of a type which creates a new object (by sending a *new* message) and the second is of a type which sends a programmer-specified message. These rules are parameterized by the programmer to specify the class and name of the object to create, and the selector and 'target' object of the message to send, respectively. When *template* is executed a new 'template' object is created which is then sent an *open* message.

Method rules and their effects currently defined in MGA are as follows. Where:

c:i represents an object id of the form <class>:<instance>.
class:c means the object representing the class c; an instance of the *class* meta-class.

The syntax

<target object id>.<message selector>(parameters*)

is used for MGA messages.

create_instance_rule(c,i)->class:c.new(i)
create a new instance i of class c

delete_instance_rule(c,i)->c:i.die()
delete the object c:i

assign_value_rule(c,i,a,v)->c:i.value(a,v)
assign value v to attribute a in object c:i

delete_value_rule(c,i,a,v)->c:i.delete(a,v)
delete value v from attribute a in object c:i

send_message_rule(c,i,s)->c:i.s()
send message s to object c:i

Methods are represented as objects of class *method_detail*, one multi-valued attribute of which (called *rules*) represents the rule script. Rules are also objects whose attributes act as formal parameters. Rules' formal parameter attributes encode the literal value or the source of the actual parameters. Where parameters are not specified by literals, the actual parameter value is given by a reference to an attribute from which it is evaluated at run time. Hence method parameters may be primitive values or object identifiers.

For example, the *create_instance_rule* (described below) belonging to *bbe*'s *template* method creates a new object of a class specified by *bbe*'s *class* attribute. The fact that this is the source of the actual parameter value is encoded by the value of the rule's *class_domain* attribute.

The effect of executing a rule is to generate a message. In most cases, the generated message is fixed and bound to the rule type - a *create_instance_rule* generates a *new* message, for example. *Send_message_rules* differ because the name of the message to send is supplied by the programmer as a parameter.

Figure 3 shows the message *template* being received by the object *bbe:bbe1*. The *class* attribute contains the name of the *expenses* class and the *instances* field contains references (indicated by the arrow) to all extant instances of *expenses*. On receipt of the *template* message, the *template* method is invoked. Figure 4 illustrates how the method rules are evaluated and executed.

The *template* method's rule script contains the two rules mentioned above. The first rule is the *create_instance_rule*, and the second is a *send_message_rule*.

Execution of the method results in the evaluation of the rules as follows:

*Note that a button for the *open* method is absent — *open* is an idempotent operation so an object which is already 'open' cannot be re-opened.

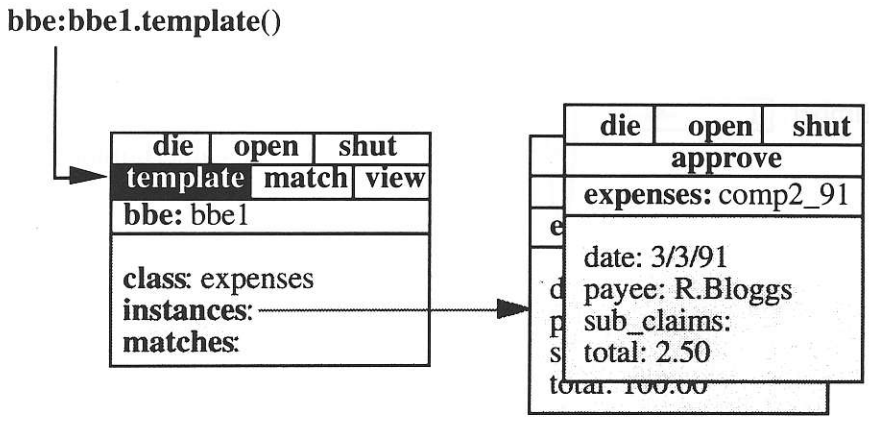

Figure 3 An example method invocation

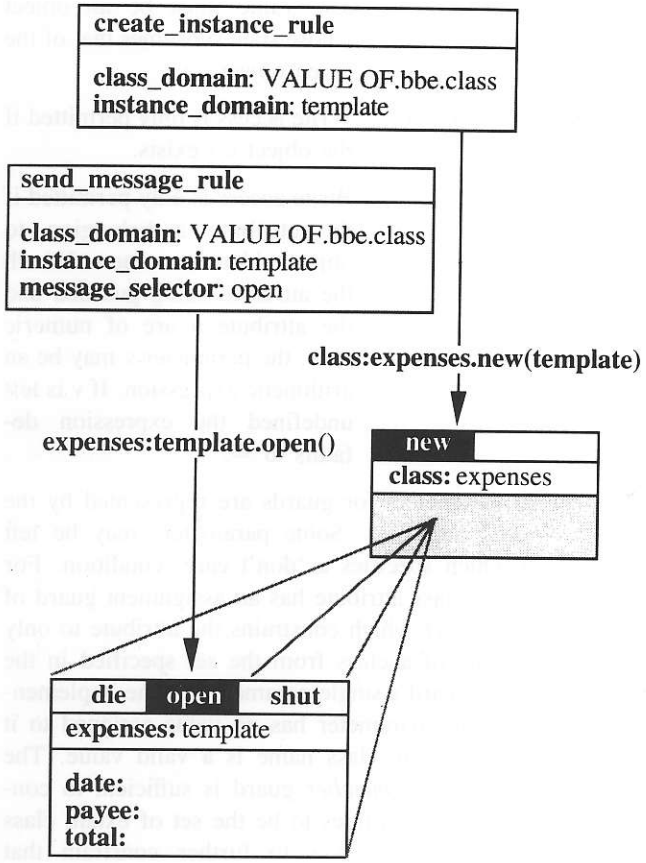

Figure 4 Method rule evaluation

Rule 1: create_instance_rule
The rule's two parameter attributes, *class_domain* and *instance_domain* specify the class of the new object to create and the name to be assigned to the new object. Their actual parameters are derived as follows:

class_domain parameter: The actual parameter value is derived from the formal parameter value *VALUE OF.bbe.class*. This specifies that the actual parameter is the value of *bbe*'s *class* attribute, *expenses* (the syntax of rule parameters includes tests for equality, set and subrange membership and basic arithmetic relationships).

instance_domain parameter: The formal parameter value, *template*, is the literal value of the actual parameter.

Following evaluation of the rule's actual parameters, the new object *expenses:template* is created by sending a *new* message to the *expenses* class object (an instance of the meta-type *class*). Here the *class_domain* parameter specifies the target of the message and the *instance_domain* parameter is the single parameter required by the *new* message.

Rule 2: send_message_rule
The rule's three parameter attributes specify the message to generate and the object which is to be the message's recipient.

class_domain parameter: Again, the actual parameter is *expenses* derived from *bbe*'s *class* attribute.
instance_domain parameter: As with the *create_instance_rule*, the formal parameter value, *template*, is the literal value of the actual parameter.

Taken together the *class_domain* and *instance_domain* specify the target of the message; the newly created object *expenses:template*.
message_selector parameter: The formal parameter value, *open*, is the literal value of the actual parameter and represents the selector of the message to be generated.

Once the *send_message_rule* has been evaluated, the message *open* is sent to the new *expenses:template* object.

Guards

Write access to objects' state is provided by two attribute access methods *value* and *delete*. Guards are a mechanism for filtering and constraining *value* and *delete* messages on a per-attribute basis. Guards range from blanket protection from deletion or assignment, through statically defined sets of acceptable values, to specifications of the context, relative to the state of other attributes and objects, under which an update may or may not occur.

Guards are boolean predicates which act as filters on attribute update messages. As with method rules, guards

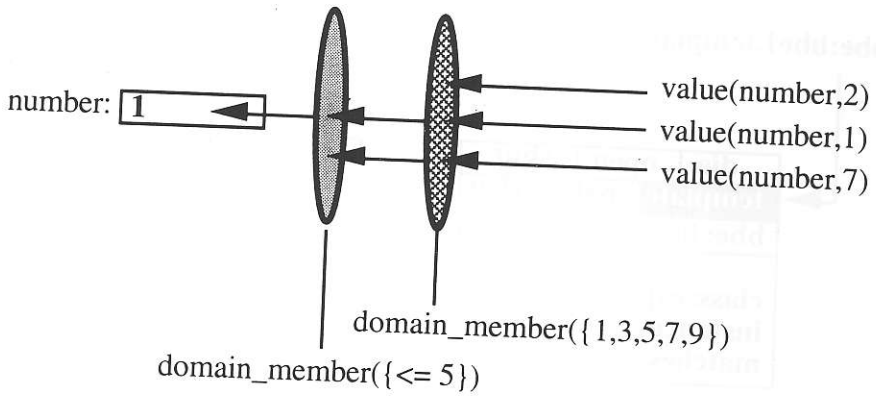

Figure 5 Guards as filters

are modelled as objects with one object class corresponding to each guard type. A similar approach is employed by Medeiros and Pfeffer[10] in their constraint mechanisms for the O_2 OODB. They also model constraints as objects but their processing of constraint specifications differs because it includes an analysis phase which searches the database schema, identifies all methods which update the attribute in question and generates a separate constraint object for each method. MGA generates a single guard object per constraint which remains ignorant of the source of attempted updates. Messages are routed via the guard object which forwards the message if it does not conflict with the constraint.

Guard types implemented in MGA are:

inaccessible()	Write access to the attribute is strictly local, being performed only by one of the object's methods or by an action attached to a sibling attribute.
domain_member({v*})	The value (which may be of any type) must be a member of the given set of values {v*}. If the attribute is of a numeric type, one of the arithmetic expressions =, >, >=, <, or <= may be used to specify the set.
class_member({c*})	The value must be a member of the given set of class names.
instance_member(c,{i*})	The value (of form c:i) must be an object of class c whose instance id is in the give set of instance names.
val_member(c,i,a,v)	The value must be a member of the set of values held by the given object's (c:i) attribute (a). If both the attribute being guarded and the attribute a are of numeric type, the parameter v may be an arithmetic expression. If v is left undefined the expression defaults to =.
fits_template(c:i)	The value must be an object whose state subsumes that of the given object c:i.
instance_dependent(c,i)	Write access is only permitted if the object c:i exists.
value_dependent(c,i,a,v)	Write access is only permitted if the attribute a belonging to object c:i has the value v. If both the attribute being guarded and the attribute a are of numeric type, the parameter v may be an arithmetic expression. If v is left undefined the expression defaults to =.

The formal parameters for guards are represented by the guard objects' attributes. Some parameters may be left unassigned which specifies a 'don't care' condition. For example, *bbe*'s class attribute has an assignment guard of type *class_member* which constrains the attribute to only accept the name of a class from the set specified in the *class_member* guard's single parameter. In the implementation of *bbe*, this parameter has no value assigned to it which means that any class name is a valid value. The existence of a *class_member* guard is sufficient to constrain the set of valid values to be the set of extant class names, its parameter serves to further constrain that set.

Every guard can be negated, so for example, a ¬ class_*member* guard constrains updates to occur only if the values fall outside the given set of class names. Guards are invoked on assignment or deletion. Attributes therefore have two guard scripts; one containing guards which filter assignments (*value* messages) and one for filtering deletions (*delete* messages).

Figure 5 illustrates attempts to assign a value to an attribute *number* being filtered by two guards of type *domain_member* which simply specify the set of values which the attribute may accept. When a request to update an attribute is received, each relevant guard (depending on whether the request is to add or delete a value) is evaluated and only if all are satisfied may the update occur.

The assignment guards attached to *bbe*'s *matches* at-

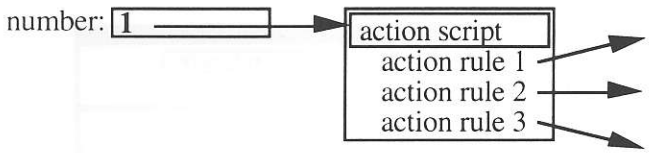

Figure 6 Actions firing on assignment of an attribute value

tribute illustrate the implementation of MGA guards. There are two guards, an *instance_member* and a *fits_template* guard. These provide the bulk of the functionality for the *match* method. Recall that this method's job is to copy (references to) all the objects in the *instances* attribute to the *matches* attribute excluding those which do not match the *template* object. In fact the job performed by the *match* method is much simpler, it simply attempts to copy every value in *instances* to *matches* by generating a *value* message for each. The filtering of objects against the template is performed by the *fits_template* guard and the *instance_member* guard guarantees that no objects which do not appear in the *instances* attribute are copied.

Actions

The objects involved in the execution of an application must remain mutually consistent. Actions are a trigger mechanism which permit dependencies between attributes to be specified. They are simply scripts of method rules, but unlike the methods described earlier actions are invoked only by the attribute update messages *value* and *delete*. Actions may invoke methods of any kind.

One useful function of actions is to maintain consistency where values are derived from attributes whose states may be subject to change. As discussed above, the use of a method which needs to be explicitly invoked in order to calculate such a derived value fails to ensure that such consistency is maintained. This is because an external stimulus is needed to invoke the method while the change of attribute value is an internal event. Instead, an action can be attached to the attribute which, on the occurrence of a value or delete message, automatically causes the derived value to be recalculated. This is analogous to value changes propagating from a change to a cell in a spreadsheet. Here, the user does not need to laboriously discover and recalculate the values of all affected cells when one cell's value changes because an 'action' embodies the relationship(s) and automatically recalculates related cells' values.

Action rule types in MGA are the same as those for methods and hence also implemented as objects. As with guards, action scripts may be specified for both attribute assignment and deletion. An action script is executed after the successful evaluation of an attribute's guards and the value assignment/deletion has taken place. Following the assignment of the value 1 to attribute *number* in Figure 5, for example, the actions in the assignment rule script for *number* are invoked in order of definition (Figure 6).

An example action is the one bound to *bbe*'s class attribute which assigns values to the *instances* attribute. On assignment of a valid class name to *class*, references to all the instances of that class are automatically assigned to the *instances* attribute. Because the action is bound to *class*, it is only invoked following a successful assignment, so the class name must have satisfied all of the guards bound to *class* before the action is invoked.

Implementation

The current implementation of MGA includes a tool called the form editor in which forms are the direct visual mappings of the objects comprising a class definition.

Figure 7 shows part of *bbe*'s definition. The *form_editor:form_editor_5* form contains the declaration of *bbe* and includes fields containing *bbe*'s attribute names and method signatures (these include attributes and methods inherited from the class declared in the *superclass* field). To the top right of this, the *method_detail:'bbe.template'* form contains the method's rule script in the 'set' attribute *rules* (there are three rules; the *create instance* rule and *send message* rule as described earlier, and a delimiting rule which simply marks the end of the script), and the *create_instance_rule: 12* form below contains the *create_instance_rule*'s formal parameters.

The use of a one-to-one mapping between forms and the underlying MGA objects enables the user interface to exploit the use of methods, guards and actions to guide the programmer. For example, a new method is declared simply by typing its selector into the form editor's *methods* field. An action attached to this field instantiates a *method_detail* object and displays it to the user, prompting definition of the rule script.

Another feature is dynamic help obtained by selecting the label of a field with the mouse cursor; a pop-up menu of possible values derived at run-time from the field's type and its assignment guards is presented from which the user selects a value to assign to the field. For example, consider the parameterization of the *assign_value_rule* object for the assignment action bound to *bbe*'s *class* attribute. The help system is able to infer that *INSTANCES OF.bbe.class* is a possible value for the *value_domain* parameter from the facts that:

(a) the *class* attribute has a guard constraining its value to be a class name; and
(b) the target of the action (*bbe*'s *instances* attribute) has a guard constraining it to accept only instances of the class in the *class* attribute.

A list of such possible values is presented to the programmer in a dialogue box. The programmer can either select one of these 'suggestions' or assign a different value to the field.

Discussion

We use the terms guards and actions rather than constraints

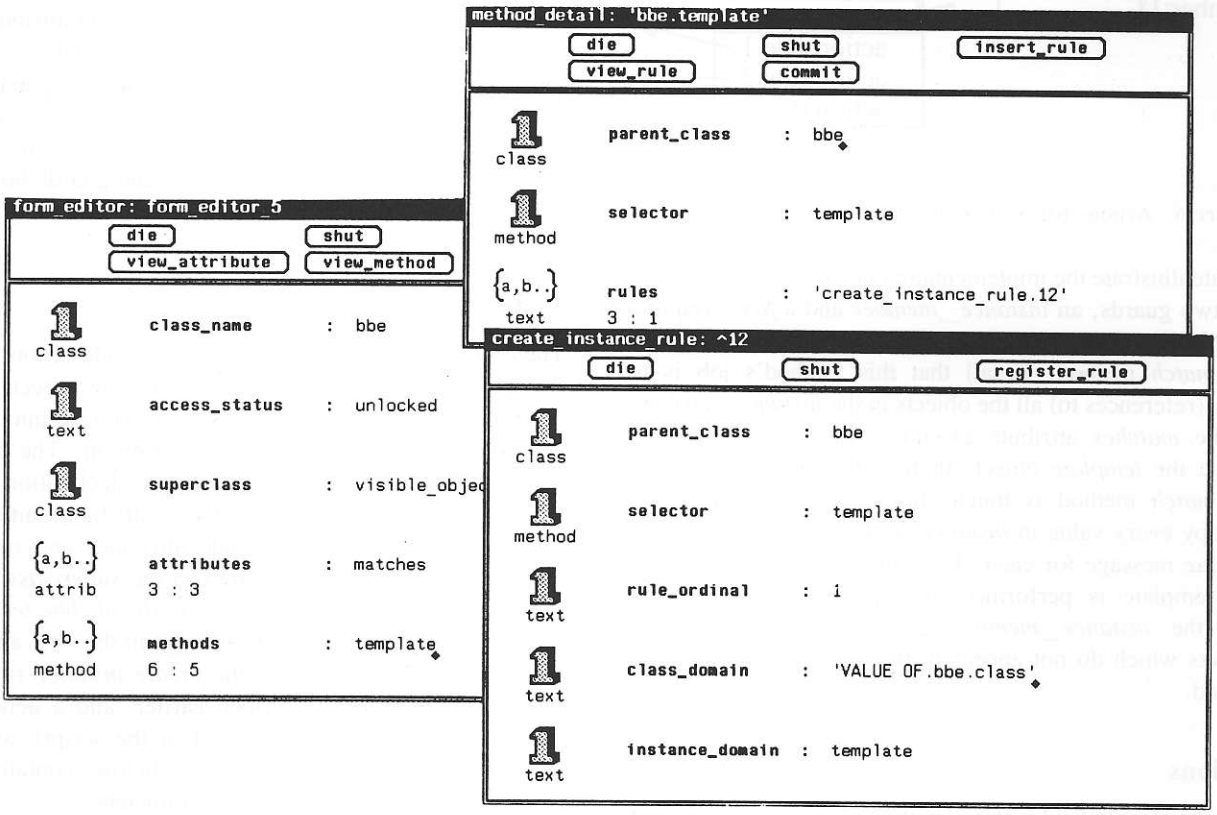

Figure 7 The form editor programming tool

and triggers because they are special cases of constraints and triggers as defined by Bloom and Zdonik[15]. A trigger is a predicate and a body; given some event execute the body. A constraint is defined as a special case of a trigger where the body is an exception to raise. In MGA, an action is always triggered by one of two events; the assignment to or deletion from a given attribute of a value. An MGA action is therefore a trigger with only two possible predicates. With an MGA guard, the guard types permit a wide range of possible predicates (for example the *val_member* type is a filter expressed as the conjunction of an update event with a set of acceptable values) but a default exception is enforced—the update is simply disallowed.

Bloom and Zdonik identify two issues raised by constraints and triggers in OODB programming languages which need to be addressed by a system like MGA. The issues are:

- *Implementation*. Triggering an action from some event can be viewed as a side-effect. They argue that this potentially makes application programming error-prone because an assumption that a trigger exists may prove to be false, or conversely, a trigger may be implemented twice under the mistaken assumption that none exists.
- *Exception handling*. The desirable property of being able to declaratively define exception handling once for an attribute through a constraint causes potential conflicts when individual operations require specific exception handling.

We do not claim that MGA completely resolves either problem, but the fact that guards and actions in MGA are restricted forms of constraints and triggers does localize the programmer's problems. The principle virtue of MGA's guard and action mechanisms is simplicity. Decoupling guards from actions by making the execution of the latter dependent on the satisfaction of the former restricts the set of operations which can have side-effects and makes exception handling more predictable. In so doing we have restricted the implications of the mismatch between good practice in database and programming language design identified by Bloom and Zdonik.

As described, guards in MGA implement static constraints (those expressed on the state of the data at a given time). Because we model guards as objects we could trivially add new predicate classes providing two-stage constraints (a restricted class of dynamic constraint) on single attributes. Evaluating a two-stage constraint requires that the state before and after a transaction be compared, for example to ensure that a new attribute value is greater than its existing value. Guards simply intercept a request to update an attribute so a two-stage constraint *greater_than* would have both the existing value and intended value available to carry out the comparison. The more general class of dynamic constraint, where a sequence of updates is described, is far more difficult as it requires a history mechanism and we have no plans to support them.

Conclusions

MGA is a prototype system designed to explore the poten-

tial of rules to embody object-oriented databases with active properties. We have adopted an orthogonal programming model where all objects' functional properties are modelled as rules and rules are modelled as objects. This both augments OODBs with an active capacity and exploits the object model for the rule mechanisms' implementation. In addition, graphical or form-based browsing tools can be used to browse not only data but also objects' active components.

Methods and actions are implemented as scripts of rules while guards are implemented as boolean predicates constraining write access to objects' attributes. Method/action rules and guard predicates are modelled as objects whose attributes represent parameters. New object classes are specified as named collections of methods and attributes with appropriately instantiated rule and predicate objects defining the object class's functionality. Rules/predicates are typed and the set of types is extensible by adding appropriate new rule/predicate object classes to the system. This is not an application programming task, however, but one which may be used to tailor MGA to a particular application domain.

References

1 Orman, I 'Constraint maintenance as a data model design criterion' *Computer J.* Vol 34 No 1 (1991) pp 73–79

2 Dayal, U 'Active database management systems' *SIGMOD Record* Vol 18 No 3 (1989) pp 150–169

3 Zdonik, S B and Maier, D (eds) *Readings in object-oriented database systems*, Morgan-Kaufmann (1990)

4 Andrews, T and Harris, C 'Combining language and database advances in an object-oriented development environment' *Proc. OOPSLA 87* (October 1987) pp 430–440

5 Kim, W *Introduction to object-oriented databases* MIT Press (1990)

6 Diaz, O and Paton, N 'Sharing behaviour in an object-oriented database using a rule-based mechanism' *Proc. 9th BNCOD*, Butterworth-Heinemann (1991) pp 17–37

7 The Kiwi team 'A system for managing data and knowledge bases' *Proc. 1988 Esprit Technical Week* (1988) pp 594–603

8 Laenens, E, Staes, F and Vermeir, D 'Browsing à la carte in object-oriented databases' *Computer J.* Vol 32 No 4 (1989) pp 333–340

9 van de Riet, R 'MOKUM: An object-oriented active knowledge base system' *Data and Knowl. Eng.* Vol 4 (1989) pp 21–42

10 Medeiros, C B and Pfeffer, P 'Object integrity using rules' *Proc. ECOOP '91* Geneva (July 1991) pp 219–230

11 Deux, O *et al.* 'The O_2 system' *Comm. ACM* Vol 34 No 10 (October 1991) pp 34–48

12 Zloof, M M 'Query-by-example: a database language' *IBM Systems J.* Vol 21 No 3 (1977) pp 324–343

13 Ozsoyoglu, G and Wang, H 'Example-based graphical database query languages' *IEEE Computer* Vol 26 No 5 (May 1993) pp 5–38

14 Sawyer, P and Somerville, I 'Direct manipulation of an object store' *Soft. Eng. J.* Vol 3 No 6 (1988) pp 214–222

15 Bloom, T and Zdonik, S 'Issues in the design of object-oriented database programming languages' *Proc. OOPSLA '87 Orlando* (October 1987) pp 441–451