Supporting Dynamic Displays Using Active Rules

Oscar Díaz‡, Arturo Jaime‡, Norman W. Paton†, Ghassan al-Qaimari†

Departamento de Lenguajes y Sistemas Informáticos‡
Basque Country University
Apartado 649, 20080 San Sebastián Spain
e-mail: <diaz>@si.ehu.es

Department of Computing and Electrical Engineering†
Heriot-Watt University Edinburgh Scotland
e-mail: <norm>@cee.hw.ac.uk

Abstract

In a graphical interface which is used to display database objects, *dynamic displays* are updated automatically as modifications occur to the database objects being visualised. Approaches based on enlarging either the database system or the interface code to provide the appropriate communication complicate the interaction between the two systems, as well as making later updates cumbersome. In this paper, an approach based on active rules is presented. The declarative and modular description of active rules enables active displays to be supported with minimal changes to the database or its graphical interface. Although this approach has been used to support the link between a database system and its graphical interface, it can easily be adapted to support dynamic interaction between an active database system and other external systems.

1 Introduction

Graphical database interfaces allow some portion of the data stored in the database to be displayed for browsing or manipulation. However, there is no guarantee that while this data is presented on screen the extension of the database will remain unchanged. Thus changes to database objects which are depicted on screen can lead to inconsistencies between the data which is stored and the information which is displayed. *Dynamic* or *active* displays remove such inconsistencies by propagating changes to the state of the database to the different interfaces where the affected data is being displayed.

In an object-oriented database system (OODBS) this propagation could be achieved by the methods responsible for modifying the state of an object: as well as updating the object, such methods could notify appropriate interfaces of the change. However, such an approach makes database objects responsible for maintaining the consistency of all interfaces upon which they may be displayed. Further, any changes to the design of the interface must be taken into account by methods associated with all objects which may be displayed using the interface.

An alternative approach is to make the underlying database system responsible for informing interfaces of changes to the database state, but as with the use of method code, this approach is not readily extensible to support new or revised database interfaces.

To support active displays, it is necessary for the interface to be informed automatically of changes to the state of objects which it is currently displaying. Active databases have been proposed for exactly this kind of activity, in which behaviour must be invoked automatically to enable an appropriate response to some happening [2].

OODBS commonly support activity by event-condition-action rules. The structure of a rule is mainly described by the *event* that triggers the rule, the *condition* to be checked and the *action* to be performed if the condition is satisfied.

Such rule mechanisms are particularly suitable for describing extensions to behavioural properties of a system [7] such as those required to support dynamic displays. This convenience mainly stems from:

- rules having a more declarative description which includes the context where the rule is used. By contract, an extension achieved by enlarging code makes this code and its extensions more cumbersome to maintain.

- rules having a more modular description. Each rule represents a single *chunk* of the extension. By contrast, methods can support different types of extension, and as such are the principal mechanism for defining user-invoked operations on objects. However, the implementation of extensions to the functionality of a class using method code can be jeopardised by the subsequent overriding of the method [7].

In what follows, such an approach is used to support active interfaces using event-condition-action rules (hereafter ECA rules). The idea is borne out by an implementation in which a graphical browser, EVE [9], is used to visualised objects from a DB system, ADAM [8].

The paper is organised as follows. The rule system used, EXACT [3, 4] is presented in section 2. Section 3 shows how rules are used to enhance the interface with dynamic capabilities. Finally, conclusions are presented.

2 The rule system

Briefly described, the function of ECA rule management is to provide timely response through the use of *rules*, to *events* generated by some *system*. Five components can be identified in this process:

- *the event* is an indicator to signal that a specific situation has been reached to which reactions may be necessary [6].

- *the rule* describes both when and how the system reacts to an event.

- *the event generator* can be seen as any system producing events which may need a special response in terms of rule triggering. Events can be generated by the DBMS itself or by any other external system such as a clock or an application program.

- *the event manager* which is in charge of the management of events. This includes the setting up of *alarms* in the appropriate event generator as well as the detection of composite events, i.e. events defined as a conjunction, disjunction or sequence of events [1].

- *the rule manager* where the creation, ordering and numbers of rules to be executed are considered.

The event, the condition and the action are the core components of rules. The condition is a set of queries which checks that the state of the database is appropriate for execution of the action. The action is a set of operations that make some suitable response to the event. The event description depends upon the nature of the event. In an OODB, an important source of internal events is message sending. In EXACT, these events can be described using the following attributes:

- active_method: the name of the method to be monitored.

- when: the point at which the rule should be fired relative to the execution of the active_method - *before* or *after* method execution.

- active_class: the class of the object to which the message has been sent for which the event is to be triggered. In OO systems, methods are not isolated but are part of a class definition. The same method name can be implemented in different ways in different classes, or a method can be overridden by a definition in a subclass. Thus the context in which an event occurs needs to include the class of the object to which a message has been sent.

- active_object: the instances to which the message has been sent for which the event is to be triggered. This is an alternative to the active_class attribute. If an ECA rule is shared by all instances of a class, this commonality should be reflected by defining the rule at the class level. However, sometimes active behaviour is shown by only a few instances of a range of classes. In this case the definition of the event should be at the instance level, to avoid events being triggered for objects which are not of interest.

3 Supporting Dynamic Displays Through ECA Rules

In what follows it is shown how ECA rules have been used to convert a passive graphical interface to an active interface. This is achieved by informing the interface of changes to the displayed objects.

The type of active behaviour which is most appropriate for updating the display depends upon:

- whether the object is displayed as an instance or as a class, and
- the methods used to modify the state of the object. The sending of these methods generates the events which may have to be responded to, by the current displays.

Such differences are analysed in the following subsections.

3.1 Updating Objects As Instances

In ADAM, methods used to modify the state of an object are generated automatically by the system whenever an attribute is created. For example, when the attribute *age* is introduced, the system generates the methods *put_age, delete_age* and *update_age*.

Thus the instances of a class are modified by a set of class-specific methods with fixed prefixes. Rather than generate a separate rule for each method which may update an attribute, the active rule system provides the keyword *modify_method* which can be used as the *active_method* attribute of a rule, to represent any method which directly updates the state of an object. Objects are displayed in their role as instances using windows with a format exemplified in figure 1. The nature of instance browsing is that while there may be a very large number of instances in the database, the number of which are being displayed at any moment is typically very small. Thus monitoring changes to every instance in a class which is known to be represented on screen would be extremely expensive, as changes to instance objects are not uncommon. A further characteristic of instance presentation is that there are often instances from a number of different classes displayed on screen at the same time.

Therefore, the approach taken is to define a rule which fires only when changes are made to objects which are actually on screen (e.g. *11#student, 23#course*). Such a rule has the following attributes:

- active_object: [11#student,23#course]
- active_method: modify_method
- when: after
- condition: true
- action: warn the interface of the change

On the basis of the rule described above, the system will inform EVE of changes to the objects *11#student* and *23#course*, as these are the values of the *active_object* attribute. As a result, the overhead of rule triggering is only paid for instances which are actually being displayed, rather than for every potentially displayable instance.

The next question to be addressed is who makes the active mechanism aware of the objects to be monitored i.e., who updates the *active_object* attribute. Two approaches are possible namely:

- it is the responsibility of the interface to update the *active_object* attribute dynamically as the display is revised to show different objects. This rate of change is constrained by the rate at which the user clicks on the *Next* button illustrated in figure 1. This approach can be supported by enlarging the definition of the corresponding *callbacks* (i.e. behaviour attached to the interface components, such as icons or dialog boxes),
- it is the responsibility of the active mechanism itself. Here, the interface is seen as an event generator which warns the active mechanism but the reaction to this warning is handled within the database. Such reactions can be expressed by another rule which is triggered by these external events and whose action updates the rule described above. Unlike the previous approach, the "communication" with the database interface is described by this rule rather than coded in the *callbacks* within the interface.

The latter approach is supported by the second rule shown in appendix A.

3.2 Updating Objects As Classes

In the ADAM database, all class objects share a common signature which is used to support the operations associated with schema evolution – for

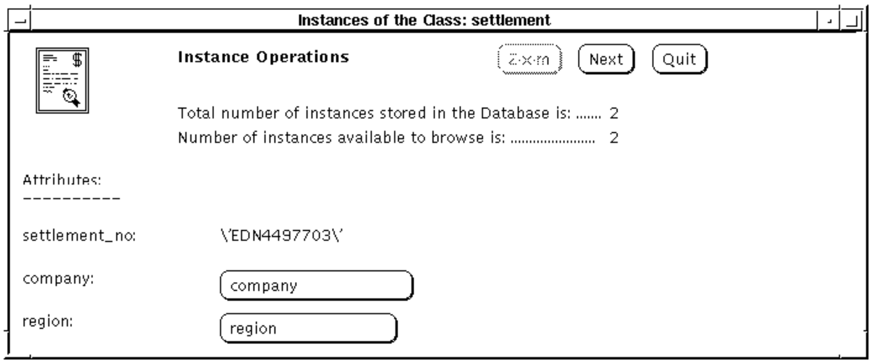

Figure 1: Example default instance window.

example, all classes understand the methods used to create or delete an attribute or method. Typical of these is the method *put_slot*, which is used to introduce the definition of a new attribute. If it happens that the class is being displayed at the time when this method is invoked, the appropriate part of the display has to be updated. Thus for each of the limited number of methods which perform changes to the database schema an ECA rule has been defined which informs the displays of schema changes.

Changes to the schema of the database directly affect the information displayed in EVE using the display shown in figure 2. For example, the creation of a new slot may require an insertion in the list of the slot names of a class, or an extension to the presented fragment of the schema diagram. This is achieved using an active rule with the following characteristics:

- active_class: sm_beh[1]
- active_method: put_slot
- when: after
- condition: true
- action: inform the displays of a new slot

The effect of the rule is to inform each interface every time a new slot is created. Each interface is then able to determine if this requires any immediate action to update parts of its display. The fact that interfaces are informed which need not make any updates to their display is felt to be acceptable because the frequency of schema updates is relatively low.

The fact that it is common for a significant number of classes to be depicted on screen at the same time, combined with the relatively small number of schema change operations, has enabled active behaviour for class objects to be defined for all classes at the metaclass level, rather than separately for each individual class. Thus, in contrast with the previous subsection, rules are defined at the level of the metaclass of which all classes are instances, *sm_beh*, rather at the level of individual objects. Hence, the rule gives a value for the *active_class* attribute, rather than the *active_object* attribute utilised in section 3.1.

It is worth noting that the requirement placed on the interface that it knows how to revise itself to reflect schema changes is not a significant extra burden - as the interface can be used to perform schema modification, the ability to revise its display to reflect schema changes is necessary even where the interface is not dynamic. It is also appropriate that this information be stored along with the interface, thereby adhering to the object-oriented principle that programs are stored with the data to which they relate.

4 Conclusions

Dynamic displays are updated as modifications occur in the objects being displayed. The approach presented here for supporting dynamic displays is based on using active rules to provide such dynamism. The communication required between the interface and the database system is supported us-

[1] All classes are created as instances of some metaclass. Every metaclass is a subclass of *sm_beh*, from which it inherits behaviour associated with schema modification [5]. Thus every class can be considered to be an instance of *sm_beh* in the same way as every student can be considered an instance of person.

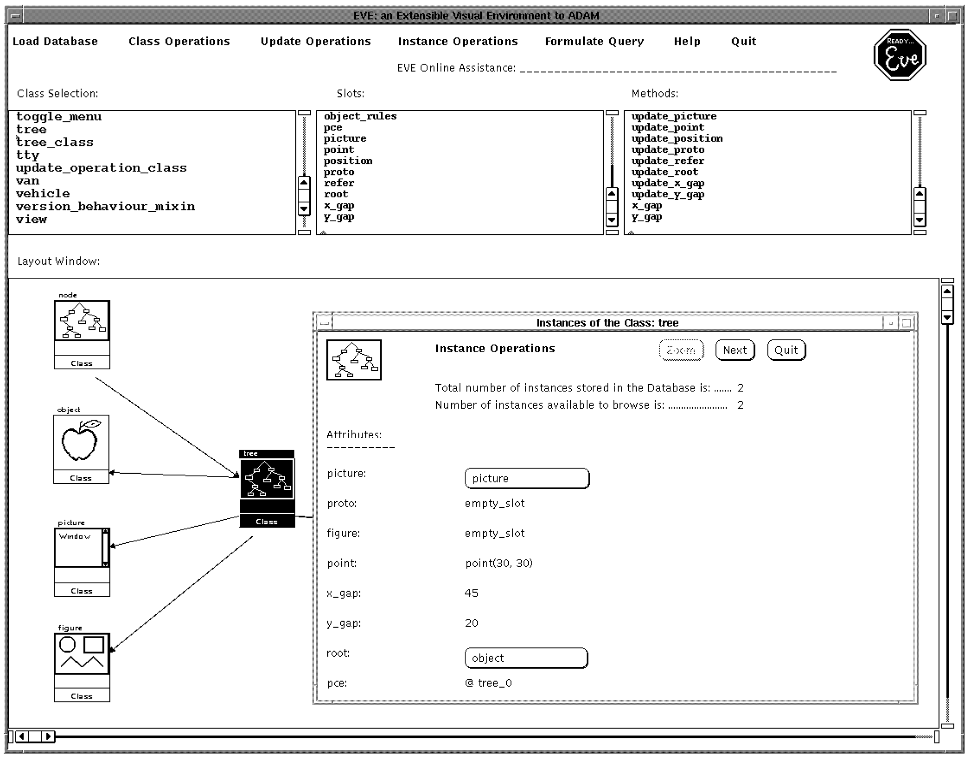

Figure 2: Layout of EVE browser.

ing event-condition-action rules. Such an approach frees both the interface and the database system code from being enlarged with the extra functionality which would implement the communication. Besides, the modularity of rules allows for representing each unit of communication as a separate rule, and hence it enhances the updateability and evolution of the database interface. Indeed, a different interface can be supported by updating the action part of the rules to use the corresponding commands of the new interface, and by converting the interface into an event generator.

References

[1] S. Chakravarthy. Rule management and evaluation: an active *DBMS* perspective. *SIGMOD RECORD*, 18(3):20–28, 1989.

[2] U. Dayal. Active database management systems. *SIGMOD RECORD*, 18(3):150–169, 1989.

[3] O. Diaz, P.M.D. Gray, and N. Paton. Rule management in object oriented databases: a uniform approach. In R. Camps G.M. Lohman, A. Sernadas, editor, *17th Intl. Conf. on Very Large Data Bases, Barcelona*, pages 317–326. Morgan Kaufmann, 1991.

[4] O. Diaz and A. Jaime. EXACT: an EXtensible approach to ACTive object-oriented databases. *Submitted for publication*, 1993.

[5] O. Diaz and N. Paton. Making object-oriented databases extensible through metaclasses: An experience. *To be published in IEEE Software*, 1994.

[6] A.M. Kotz, K.R Dittrich, and J.A. Mulle. Supporting semantic rules by a generalized event/trigger mechanism. In *Advance in Database Technology, EDBT, Venice*, pages 76–91, 1988.

[7] N. Paton, O. Diaz, and M.L. Barja. Combining active rules and metaclasses for enhanced extensibility in object-oriented systems. *Data and Knowledge Engineering*, 10:45–63, 1993.

[8] N.W. Paton. Adam: An object-oriented database system implemented in prolog. In M.H. Williams, editor, *Proc. British National Conference on Databases*, pages 147–161. Cambridge University Press, 1989.

[9] N.W. Paton, G. al Qaimari, and A.C. Kilgour. An extensible interface to an extensible object-oriented database system. In R. Cooper, editor, *Proc. 1st International Workshop on Interfaces to Database Systems*. Springer-Verlag, 265–281, 1993.

Appendix A: Example ECA rules in EXACT notation

This rule maintains the instances that are being displayed on the different browsers:

```
new([RuleOid,[
    active_object([<List of displayed objects>]),
    active_method([modify_method]),
    when([after]),
    is_it_enabled([yes]),
    condition([(true)]),
    action([(
        current_object(DisObj),
        (get_by_displays([DisObj],TheBrowser)
            => adam_browser,
            refresh([DisObj]) => TheBrowser,
            fail ; true
        ))
    ])
]]) => display_rule,
```

The next rule maintains the "active_object" attribute of the above rule. This is used to make active only those instances that are being displayed on any browser at that specific time. It is worth noting that EVE has been integrated within ADAM as an object, i.e. displays are described by attributes and methods. This simplifies the detection of events generated by the interface since its events are normal message-sending events. A similar rule needs to be defined when objects are no longer displayed i.e., a rule triggered by the sending of the method "delete_displays".

```
new([_,[
    active_class([adam_browser]),
    active_method([put_displays]),
    when([before]),
    is_it_enabled([yes]),
    condition([(
        current_arguments([DisObj]),
        not get_by_displays([DisObj],_)=> adam_browser
        )]),
        action([(
        current_arguments([DisObj]),
        put_active_object([DisObj]) => RuleOid
    )])
]]) => display_rule,
```

Seven rules like the one shown below are defined for the events generated by sending messages which update the class definition: put_slot, delete_slot, put_method, delete_method, replace_method, new (introducing a new class) and delete (deleting an existing class). The action is customised depending upon the method.

```
new([_,[
    active_class([sm_beh]),
    active_method([put_slot]),
    when([after]),
    is_it_enabled([yes]),
    condition([( true )]),
    action([(
        current_object(The_class),
        current_arguments([_nam,_vis,_car,_sta,Type]),
        ( object_db(Type,_)
            -> BroMetNam = new_obj_s_screen_refresh
            ; BroMetNam = new_scal_s_screen_refresh),
        BroMet =.. [BroMetNam, [The_class]],
        ( get(Browser)=>adam_browser,
        BroMet => Browser, fail; true
        ) )])
]]) => display_rule,

new([_,[
    active_class([sm_beh]),
    active_method([delete_slot]),
    when([after]),
    is_it_enabled([yes]),
    condition([( true )]),
    action([(
        current_object(The_class),
        current_arguments([_nam,_vis,_car,_sta,Type]),
        ( object_db(Type,_)
            -> BroMetNam = del_obj_s_screen_refresh
            ; BroMetNam = del_scal_s_screen_refresh),
        BroMet =.. [BroMetNam, [The_class]],
        ( get(Browser)=>adam_browser,
        BroMet => Browser, fail; true)
]]) => display_rule.
```