GIS/LIS '95

Annual Conference
& Exposition
November 14-16, 1995
Nashville Convention Center
Nashville, Tennessee

Proceedings
Volume 1
DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.
IMPLEMENTATION OF A RULE-BASED FRAMEWORK FOR MANAGING UPDATES IN AN OBJECT-ORIENTED VPF DATABASE

David K. Arctur, GeoPlan Center
431 Arch Building, University of Florida
Gainesville, FL 32611-2004
e-mail: arctur@ufl.edu

Co-authors:
Eman Anwar, University of Florida
Sharma Chakravarthy, University of Florida
Maria Cobb, Naval Research Laboratory
Miji Chung, Naval Research Laboratory
Kevin Shaw, Naval Research Laboratory
John F. Alexander, University of Florida

ABSTRACT

This paper presents an approach for supporting reactive capability in an object-oriented GIS database, through the use of an event interface comprising an event generator and rule objects. This interface supports specification of events spanning sets of geographic-feature objects, and detection of primitive and complex events. Rules can be specified to apply either at a class level (i.e., to all instances of a given geographic-feature class) or at an instance level. In addition we allow evaluation of both pre- and post-conditions on changes to a feature. This approach is relevant in three distinct situations: (1) immediate mode, to execute rules immediately before or after some state change; (2) deferred mode, to execute rules at the end of several changes; and (3) detached mode, to perform rule-based actions separately from the state changes. GIS presents a rich set of problems for which this approach can be useful. This paper outlines the key elements of the rule-based approach employed in an object-oriented framework used for viewing and editing Vector Product Format (VPF) source data.

INTRODUCTION

A joint project of the U.S. Naval Research Laboratory (NRL) and the University of Florida during the past two years has resulted in the development of an object-oriented prototype application for viewing and managing Vector Product Format (VPF) geographic databases (DMA 1993), using the Smalltalk language on a Unix platform. As defined by the Defense Mapping Agency (DMA), the VPF specification uses a relational database approach for storing metadata, feature attributes and geo-spatial information. However, the current project uses an object-oriented representation of VPF (OVPF) to facilitate complex, user-interactive operations with the database (Arctur 1995a,b, Shaw 1994).

Building on the literature of recent developments in rule-based frameworks for object-oriented databases, we have implemented such a framework in our Smalltalk OVPF viewer/editor application to help in managing application-dependent constraints on changes to geographic features (e.g., preventing the user from creating or moving a land feature over water). We first present background on the rule-based framework, then introduce our implementation and provide an example of usage.
CONCEPTUAL BACKGROUND

Active Objects and Databases
During the past years database management systems (DBMS) have undergone dramatic changes as a result of the increasing requirements of modern day applications. Conventional record-oriented database systems are subject to the limitations of a finite set of data types and the need to normalize data. These limitations have led to the evolution of a new paradigm, namely object-oriented database management systems (ODBMS), which offer increased modeling power, flexible abstract data-typing facilities and the ability to encapsulate data and operations via the message metaphor. Despite the ability to model complex objects and relationships, these ODBMSs still lack some of the requirements of a large class of new applications, specifically those requiring monitoring of situations and the ability to respond automatically, possibly subject to timing constraints.

Active databases have been proposed to meet some of the requirements of non-traditional applications (Chakravarthy 1994a). Active ODBMSs extend the normal functionality of ODBMSs with support for monitoring user-defined situations and reacting to them without user or application intervention. These ODBMSs continuously monitor situations to initiate appropriate actions in response to database updates, occurrence of particular states, or transition between states, possibly within a response-time window. The emergence of this trend of active ODBMSs serves a large variety of applications such as GIS, AM/FM, computer integrated manufacturing (Honnavalli 1994), process control, battle management, and network management. Furthermore, active databases provide an elegant means for supporting integrity constraints, access control, maintenance of derived data, and materialized views and snapshots.

Active behavior is by no means a new notion. However, it has been used to connote different behavior in various contexts within computer science. Morgan used the term "active database," perhaps for the first time (Morgan 1983), to describe a system that supports automatic update of views and derived data as base data are updated. In the artificial-intelligence community the term "active object" is used either for active knowledge representation and inference mechanisms or for achieving intelligent behavior and concurrent computation. The programming-language community uses the term "active object" in order to structure concurrent applications in an object-oriented programming language. Ishikawa used the term "active object" to distinguish real-time objects from others which have timing constraints (Ishikawa 1990). In summary, the term "active" has been used to convey concurrency, synchronous behavior, and parallelism of active objects, intelligent behavior of agents/actors, or active capability of a system. In other literature similar notions are elaborated without using the term active explicitly.

The key distinction we draw between an active and a passive object lies in an active object’s ability to monitor its state and take pre-defined actions that are based on the state changes. This is in contrast to a conventional object which responds to a message with a
predefined effect (of course based on the state), but the object cannot monitor its or other objects' status. This concept (of activeness), to some extent, is present in the actor model of Agha, et al. (Agha 1986).

Rules

Rules, also referred to as triggers and alerters (Chakravarthy 1989, Dayal 1988, Dittrich 1986, Su 1988), have been proposed to provide active functionality in ODBMSs. Rules, in the context of an active DBMS, consist primarily of three components: an event, a condition, and an action. An event is an indicator of a happening (either simple or complex). Events are recognized by the system or signalled by the user. For example, events such as the creation of an instance, the change of an attribute's value, and accessing an attribute's value are detected by the ODBMS. The condition specifies an optional predicate over the database state which is evaluated when its corresponding event occurs. The conditions to be monitored may be arbitrarily complex and may be defined not only on single data values or individual database states, but also on sets of data objects, transitions between states of materialized/derived objects, trends and historical data. Actions are the operations to be performed when an event occurs and its associated condition evaluates to true. Actions can be programs whose execution may in turn cause other events to occur. Once rules are specified declaratively to the system, it is the system's responsibility to monitor the situations (event-condition pairs) and execute the corresponding action when the condition is satisfied without any user or application intervention. The advantage of using rules as a means of providing active behavior is the freedom from explicitly hard-wiring code which checks the situations being monitored in each program that updates the database.

Events

Incorporation of rules in any system entails identifying what constitutes an event, developing an expressive event specification language, constructing an event detection mechanism, and identifying how to represent conditions and actions. Our framework is based on the classifications in Snoop (Chakravarthy 1994b,c) which defines semantics for various events and event operators in an object-oriented environment. An event is defined as something that happens at a point in time. In an object-oriented context, the events of interest are concerned with changes to an object's state. An object's state changes as the result of an update operation. Update operations occur when an object receives a message. Therefore, we view each message sent to an object as a potential event. Considering messages sent to objects as events per se is ambiguous; it is not clear whether the event is raised before or after the execution of the update message. To resolve this ambiguity, the pre and post clauses are introduced. The "pre" clause indicates the signalling of an event before the message is executed, while the "post" clause indicates the signalling of an event after the execution of the message.

Events are categorized as being either primitive or complex. Primitive events are those signalled at the beginning or at the end of execution of a single specific message. The term beginning refers to the point before the receipt of the message and end refers to the point after executing all operations within the method including the return statement. It is important to note that messages sent to objects are considered as primitive events regardless of the type of operations performed by the method.

Many applications are not well served by primitive events alone. Complex events (also called composite events) provide a simple and powerful mechanism for expressing the conjunction, disjunction and sequence of either primitive or other complex events. In our
framework an event can be defined as the conjunction of two events E1 and E2 which is signalled when both E1 and E2 occur, regardless of the order of execution. The disjunction of two events would be signalled when either E1 or E2 occurs. The sequence event would be signalled by completion of the sequential occurrence of a set of events. Further details on the event specification language, as well as descriptions of temporally-complex events may be found in Chakravarty (1994c).

RULE-BASED FRAMEWORK IMPLEMENTATION

The foregoing discussion provides a basis for introducing our implementation of the rule-based framework. As events are the most central part of the design, we will first describe event objects, then describe rule objects, and present an example of usage. We will then describe the outer framework in which event detection and rule firing take place.

Event Objects

Events are first-class objects in this framework as they have significant state and behavior. The PrimitiveEvent class in Figure 1 defines an eventMsg attribute which is inherited by all its subclasses. For each new instance of any event, this attribute is assigned the name of the message for which the event is raised. The ComplexEvent class defines further attributes used by its own subclass.

The key method for each of the event classes is notify. This method takes only one argument which specifies the name of the message which causes an event to be raised. The event is raised when the object(s) associated with the event object receives that message. For PrimitiveEvents the notify method simply compares the argument to its own eventMsg attribute value and returns true if they match. For each ComplexEvent subclass, the notify method also examines a particular combination of the status of its other attributes, before returning true or false.

![Event Class Hierarchy Diagram]

An event instance is typically created at the time of rule creation. We will describe the Rule object and then present an example of usage.
Rule Objects

Rule objects have the structure shown in Figure 2. A single class suffices for defining all rules. The feature attribute may be assigned a pointer to either a single geographic-feature instance, such as a road or lake; or to a feature class, such as the defining class for roads or lakes. In the former, the rule will be applied only to a particular instance whereas in the latter, the rule will be applied to all instances of the defining class. The event attribute is assigned a pointer to a specific event instance (introduced above), which could be either a PrimitiveEvent or a ComplexEvent. The condition attribute is assigned the name of a method to be executed at the time the event is signalled, which will return true if the condition is met and false otherwise. The action method is then executed if the condition evaluates to true. The preOrPost attribute specifies the relative timing for execution of the condition method with respect to the message raising the event. The condition may be evaluated either before the event message is executed, or upon completion and return from the event message execution. The actionPriority attribute value is used to help mediate in situations where multiple rules fire at the same time.

Example Rule and Event Objects

An example of a Rule to prevent any BuildingPoint geographic features from being created over water is shown in Figure 3. In this case, the Rule is associated with the BuildingPoint class and thus will be applied to all instances of that class. Alternatively, the

![Figure 2. Structure of a Rule Object](image)

![Figure 3. Example Rule and Event Objects](image)
user may associate the Rule with a particular BuildingPoint instance. Due to the setting of
the preOrPost attribute, the condition method onWater is evaluated before the Event's
eventMsg (the newPoint method) is carried out. If the condition method onWater returns
true, the action method stopCreateFeature will then be executed, which will prevent the
eventMsg method newPoint from being performed. The actionPriority setting ensures
this action will have highest priority among any other Rules which may also fire.

FeatureConstructor Objects
At this point we need to introduce the rest of the framework in which Events are
detected and Rules are fired. In the OVPF viewer/editor tool, all changes to geographic-
feature objects are handled through the use of FeatureConstructor objects (see Figure 4a).
OVPF uses a construction-script framework with a state machine, supporting
asynchronous events for flexibility in working with runtime-dependent constraints on
changes to a given feature. This framework also is capable of extending its own semantics
at runtime. It is beyond the scope of this paper to fully describe the FeatureConstructor
framework, thus a very simplified portion of it is shown here for discussion.

![FeatureConstructor Class Hierarchy]

(a) Partial FeatureConstructor Class Hierarchy

![VPFFeature]

(b) Partial Feature Class Hierarchy

Figure 4. Key Components of Event Detection Framework

With reference to our example for creating a new BuildingPoint feature, we assume a
Rule-Event pair has already been created (for checking if a new point feature is over
water) and stored in VPFFeature's RuleBase (Figure 4b). This rule base is actually a
persistent collection held in the ODBMS; its reference in VPFFeature is provided for
convenient access at runtime. The following sequence of events could then take place at
the user's initiation (step numbers correspond to those in Figure 5):

1. The user chooses the appropriate OVPF menu option to add a new geographic
   feature, and selects BuildingPoint from a list of available feature classes.

2. The OVPF graphical user interface (GUI) creates a PointFeatureConstructor.
Action Summary

1. User chooses menu option to add a new feature
2. GUI creates a constructor for the new feature object
3. Constructor creates a default instance of BuildingPoint, and requests coordinate point from GUI
4. GUI returns user-defined location coordinates for new feature
5. Constructor sends message -- feature notify: 'newPoint:'
   argList: (point)
   preOrPost 'pre'
   from: self
6. Feature scans rule base for rules with event message 'newPoint:'
7. Feature finds rule and evaluates condition message --
   constructor perform: 'onWater';
   constructor then queries ODBMS and returns true or false
8. If condition evaluates true, feature sends message --
   constructor perform: stopCreateFeature
9. If constructor has to stopCreateFeature, then constructor assigns 'stop' value to its nextAction attribute
10. If constructor's nextAction is 'stop' it discards the new feature
11. If constructor's nextAction was not 'stop' then it sends the message --
    feature newPoint: point
    and finally inserts the new feature in the quadtree.

Direction of Messages

- User
- OVPF GUI
- PointFeatureConstructor
- OVPF GUI
- PointFeatureConstructor
- BuildingPoint
- RuleBase
- PointFeatureConstructor
- ODBMS
- BuildingPoint
- Spatial Quadtree

Figure 5. Flow of Control and Behavior For Rule-Event Example
3. The Constructor creates a default BuildingPoint feature object, and initiates a request to the GUI for a user-selected location coordinate point, to be returned via the point1 message.

4. On instruction from the GUI, the user chooses a location on the map with the mouse, and the GUI returns it as the argument in the point1 message to the Constructor.

5. Within its point1 method, the Constructor notifies the new BuildingPoint feature instance of an impending Event via the parametrized notify.argList: preOrPostFrom: message.

6. The new BuildingPoint object executes the inherited notify.argList: preOrPostFrom: method, which checks the rule base for all Rule-Event pairs whose eventMsg matches the notify argument, in this case newPoint.

7. If a matching Rule-Event pair is found, then the Rule's condition value (onWater:) is sent as a message to the Constructor to perform. The Constructor's onWater: method checks the database for any water-related features within a given tolerance of the user-selected coordinates, and returns true or false. By user's preference, this check can be performed either on just the features currently being displayed, or on features from all coverages in the ODBMS.

8. If the onWater: method returns true (coincident water feature was found), the Rule's action message is then sent to the Constructor. In this case if water features were found, the message stopCreateFeature would be the action message sent to the Constructor. Note that in the present framework, all applicable conditions are evaluated before any actions are performed. If multiple conditions return true, their action messages are sent to the PointFeatureConstructor in order of decreasing actionPriority.

9. If the Constructor receives the message stopCreateFeature, it will set its nextAction attribute to 'stop'.

10. Upon completion of all applicable conditions and actions, the new BuildingPoint object returns from executing the notify.argList: preOrPostFrom: method. The thread of control reverts to the Constructor's point1 method, which then checks its nextAction setting. If it is 'stop' then the new default BuildingPoint feature is discarded, and control returns to the user with a descriptive dialog message.

11. If the nextAction is not 'stop' then the Constructor sends the newPoint: message to the new BuildingPoint, inserts it in the spatial quadtree, and presents the user with a dialog window to fill in any BuildingPoint feature attributes needed.

**SUMMARY AND PROSPECTS**

The GIS field presents a rich set of problems for which this approach can be useful. It can be used in three distinct situations: (1) immediate mode, to execute rules immediately before or after some state change; (2) deferred mode, to execute rules at the end of several changes; and (3) detached mode, to perform rule-based actions separately from the state changes. Furthermore, it has the advantage over traditional inference engine approaches...
in that it will work with an arbitrarily-large database of persistent objects, rather than being limited to those objects which can fit in memory.

The rule-event framework and procedures were surprisingly simple to implement. The FeatureConstructor classes, together with a single supporting method in VPFFeature class (notifyArgListPreOrPostfrom()), provide a simple and flexible event detection and rule processing system. While it introduces some processing overhead, all but the spatial query to the ODBMS in step 7 are very fast operations. An important benefit of this object-oriented framework is the potential for direct reuse by other FeatureConstructors of condition checks and actions such as the onWater and stopCreateFeature methods. Furthermore, with this system provision can be made for adding and changing rules at runtime.

The design presented here is easily extended to trigger on any kind of change (create, modify, delete) to geographic-feature objects, as well as to specific feature attributes and spatial coordinates of a given feature object. This could be a significant advantage over the triggers supported by many commercial relational and even hybrid object-relational DBMSs. These DBMSs can typically trigger only on insert, update or delete of a complete feature record, rather than being able to discriminate on changes made to a single feature attribute.

There are a number of issues related to supporting rules that have not been addressed in this paper. We mention these for completeness as well as to provide directions for further areas of research. The number of rules defined in an application can become very large and may be defined by various users at different points in time. This can lead to the problem of having inconsistent or conflicting rules present within the application. For example, user A may define a rule R1 whose action may trigger rule R2 defined by user B. Suppose rule R2's action results in triggering rule R1, thereby yielding an infinite loop. From this scenario, it is evident that a mechanism for establishing the consistency or correctness of rules must be an inherent part of any active system. This involves writing algorithms which statically detect rule conflicts as well as algorithms which dynamically detect problems such as infinite rule triggering. We are in the early stages of pursuing these next steps.

ACKNOWLEDGMENT

We wish to thank the U.S. Defense Mapping Agency for supporting this work.

REFERENCES


This paper presents an approach for supporting reactive capability in an object-oriented GIS database, through the use of an event interface comprising an event generator and rule objects. This interface supports specification of events spanning sets of geographic-feature objects, and detection of primitive and complex events. Rules can be specified to apply either at a class level (i.e., to all instances of a given geographic-feature class) or at an instance level. In addition we allow evaluation of both pre- and post-conditions on changes to a feature. This approach is relevant in three distinct situations: (1) immediate mode, to execute rules immediately before or after some state change; (2) deferred mode, to execute rules at the end of several changes; and (3) detached mode, to perform rule-based actions separately from the state changes. GIS presents a rich set of problems for which this approach can be useful. This paper outlines the key elements of the rule-based approach employed in an object-oriented framework used for viewing and editing Vector Product Format (VPF) source data.