Access-Oriented Programming for a Multiparadigm Environment

Mark J. Stefik
Daniel G. Bobrow
Kenneth M. Kahn
Intelligent Systems Laboratory
Xerox Palo Alto Research Center
Palo Alto, Ca. 94304
(415) 494-4000

Abstract: In access-oriented programming, the fetching or storing of data causes user defined operations to be invoked. Annotated values, a reification of the notion of storage cell, are used to implement active values for procedural activations and properties for structural annotation. The implementation satisfies a number of criteria described for efficiency of operation, and non-interference with respect to other paradigms of programming. The access-oriented programming paradigm has been integrated with the Loops multi-paradigm knowledge programming system which also provides function-oriented, object-oriented and rule-oriented paradigms for users.

The Loops knowledge programming system contains a number of integrated paradigms of programming. It builds on the function oriented programming of Interlisp-D [Sanella83], and adds the familiar paradigms of rule-oriented and object oriented-programming. Its most unusual contribution is the addition of an access oriented programming paradigm not found in most systems.

In access-oriented programming fetching or storing data can cause procedures to be invoked. In terms of actions and side-effects this is dual to object-oriented programming. In object-oriented programming when one object sends a message to another, the receiving object may change its data as a side-effect. In access-oriented programming when one object changes its data, a message may be sent as a side-effect.

Access-oriented programming is based on an entity called an *annotated value* which associates annotations with data. These annotations can be installed on object variables and can be nested recursively. In Loops there are two kinds of annotated values: property annotations and active values. *Property annotations* associate arbitrary extendible property lists with data. *Active values* associate procedures with data so that methods are invoked when data are fetched and stored. Active values are the basic computational mechanisms of access-oriented programming.

Access-oriented programming in Loops went through several stages of development. From the beginning Loops provided one level of property values for annotating object variables. Active values were added shortly thereafter. The unification of these two ideas and their representation as objects was proposed after several years of experience, and was under development at the time of this writing.

Access-oriented programming has historical roots in languages like SIMULA and Interlisp-D, which provide ways of converting record accesses into a computations for all records of a given type. It is also related to the virtual data idea in some computer architectures. For example in the B5000, a tag bit associated with data caused data access to be converted into a procedure invocation. More immediate predessors are the ideas of procedural attachment from frame languages like KRL (Bobrow77), FRL (Roberts77) and KL-ONE (Brachman78). Attached procedures are programs that are associated with object variables and which are triggered under specific conditions. Access-oriented programming in Loops is intended to satisfy a somewhat different set of purposes than attached procedures and this has led to a synthesis of ideas with some important differences. Although a thorough historical review is beyond the scope of this paper, we will occasionally return to attached procedures to show how language features in Loops diverge from that work.

In the access-oriented paradigm, programs are factored into two kinds of parts: parts that compute and parts that monitor the computations. Figure 1 shows this kind of factoring for a traffic simulation program. The traffic simulation program has two parts called the *simulator* and the *display controller.* (This example was inspired by related work in Smalltalk on the partitioning of some programs into a *model* and *a view controller*). The simulator represents the dynamics of traffic. It has objects for such things as automobiles, trucks, roads, and traffic lights. These objects exchange messages to simulate traffic interactions. For example, when a traffic light object turns green, it sends messages to start traffic moving. The display controller has objects representing *images* of the traffic and provides an interactive user interface for scaling and shifting the views. It has methods for presenting graphics information. The simulator and the display controller can be developed separately, provided there is agreement on the structure of the simulation objects. *Access-oriented programming provides the "glue" for connecting them at run time.* The process of gluing is dynamic and reversible. When a user tells the display controller to change the views, it can make and break connections to the simulator as needed for its monitoring.

To illustrate this example, suppose that the simulator is running and the next event is that a traffic light turns green. The traffic light object could then send a *Go* message to each of the stopped vehicles. One of the vehicles, say *Car37*, receives the message and computes its initial velocity and position. When the method in *Car37* updates its *position* instance variable, that triggers an active value which then sends an *Update* message to the display object *StreetScene13*. *StreetScene13* may then make a change to the computer display screen so that an image representing *Car37* appears to move. This sequence of events shows how the updating of the computer display is a side-effect of running the simulator.

1. Basic Concepts of Access-Oriented Programing

In Loops there are two kinds of annotated values: property annotations and active values. *Property annotations* associate arbitrary extendable property lists with data. *Active values* associate procedures with data so that methods are invoked when data are fetched and stored. Annotated values have several important characteristics:

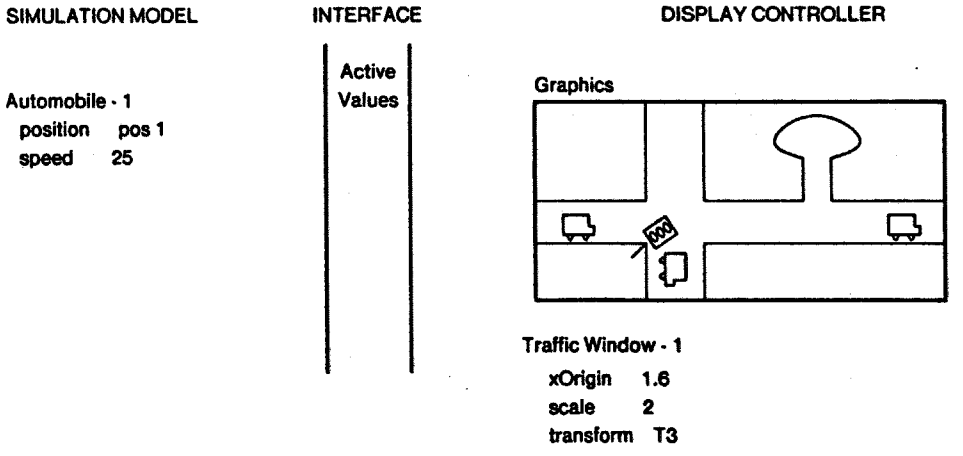

Figure 1. Traffic Simulation. *This shows an interactive graphical simulation system for city traffic. The traffic simulator has objects like automobiles, city blocks, emergency vehicles, and traffic lights that exchange messages to simulate traffic interactions. The display controller has objects for traffic icons, viewing transformations, and windows that display different parts of the city connected to the simulation objects by active values. A user of the complete system could interact strictly with the display controller in order to change what he sees on the display.*

- *Annotations are invisible to programs that are not looking for them.*

This is the first invariant of access-oriented programming. Adding and removing annotations are common changes to programs in this paradigm. Making these changes to programs does not cause the programs to stop working, unless the programs use the annotations. New annotations do not interfere with old programs that do not refer to them. For active values, this claim ultimately depends on the condition that the user-defined procedures have non-interfering side-effects.

- *Annotations have a low computational overhead when there are no annotations and also when there are annotations that programs do not reference (e.g. unreferenced properties.)*

This characteristic takes non-interference a step further. Not only are annotations invisible to programs that ignore them, but also they do not slow things down much. Accesses can either be a function call or compiled open, and the Loops implementation reduces the overhead to a single type check which has microcode support.

- *Annotations are recursive -- they too can be annotated.*

This extends the main invariant above to cover multiple annotations, that is, adding annotations to data that are already annotated. This characteristic enables the creation of *descriptions of descriptions* in the case of nested properties, and multiple independent side-effects in the case of nested active values.

- *Annotations can be efficiently accessed starting from the annotated object.*

This constraint demands that annotations can be accessed quickly and in a standard way. This characteristic is important for programs that reference annotations explicitly (e.g. using the value of a particular property), and also for programs that test annotations implicitly (e.g. automatic testing for and triggering of active values). It distinguishes annotated values from *ad hoc* data structures for annotation that point to their data. Such annotations would have the other characteristics, but would require that programs either *search* for pointers to data or else provide idiosyncratic arrangements for storing and indexing the annotations.

- *Annotations are objects that can be specialized and used with standard protocols.*

This characteristic comes from the features of object-oriented programming to simplify the creation of new kinds of annotations.

- *Active values have their own variables for saving state.*

This characteristic comes for free since active values are objects. As will be shown later, this feature removes a potential path of interaction between active values that are intended to be independent.

1.1 Active Values. Active values convert a variable reference to a method invocation. They can be installed on the value of any object variable or property. When an active value has been installed, *any* part of the program that accesses the variable will trigger the computation. This is the major lever for elision in access-oriented programming. Elision is an important mechanism in the support of a programming paradigm. By eliminating excess verbiage, the programmer can focus on the essentials, having less opportunity for mistakes, and more easily understood programs. Active values as an implementation of access-oriented progamming eliminate the need to use a variable-specific functional interface for *all* of the places that access a particular variable to allow specialization of that access.

Active values appear in slots of objects. Each active value contains a *localState* to hold the value that should appear in that slot. Since an active value is an object, it can also contain additional information in other slots of that active value object . To allow users to easily view the contents of an active value, an active value is shown as:

#[*<activeValueClass> localState otherSlot1*$_1$ Value$_1$ ...]

The class of the active value *<activeValueClass>* determines the behavior of the active value on access. Below is an example of an active value installed in a Loops object. In this example, an active value is the interface between objects in a simulator and a display controller. The object *Automobile-1* represents an automobile in a traffic simulation model. The *xPosition* instance variable represents the position of the car in the simulation world. The value of *xPosition* has been made into an active value to connect *Automobile-1* with objects in the display controller.

```
Automobile-1
    (speed 25)
    (xPosition #[InformDisplayController 50
                displayControllerObjects
                (<DispObj1> <DispObj2>)] )
```

When the simulation stores (that is, puts) a value into *xPosition*, the Loops access functions will recognize the active value and will send it a *PutWrappedValue* message. The protocol for the *PutWrappedValue* message is defined by the *InformDisplayController* class for the active value. In this case *Update* messages will be sent to appropriate objects in the display controller (DispObj1, DispObj2). These objects respond by updating the views in windows of the display.

InformDisplayController is a class for active values for updating objects in a display *controller*. It has a special method for storing and uses *viewObjects* to hold a list of objects to be informed in the display *controller*. Like other classes in Loops, classes for active values are organized in the inheritance lattice. The class *ActiveValue* defines a standard protocol for putting and getting values. This protocol is specialized in each kind of active value to describe the particular side-effects of getting or putting a value. Most subclasses of *ActiveValue* specialize either the *GetWrappedValue* protocol to specify side-effects when data are fetched or the PutWrappedValue protocol to specify side-effects when data are stored. The default behavior for *GetWrappedValue* is to return the value in the *localState*. The default behavior of *PutWrappedValue* is to store the new value in the *localState*. As is discussed later, accessing data in the *localState* may trigger additional side-effects if active values are nested.

1.2 Property Annotations. The second kind of annotated value in Loops is *property annotation*. Property annotations can be installed on the value of any object variable or property. Properties are useful for describing the relationship between the value of an object variable and the object itself. They also provide a mechanism for storing "derived" values that can be cached locally.

The idea of property annotations in Loops was motivated by several applications to knowledge programming. Property annotations provide a way of attaching extra descriptions to data for guiding its interpretation. The example below shows several kinds of annotations that have been used in different knowledge engineering applications.

```
Truck-37
    owner PIE              doc (* owner of truck)
    highway I66            doc (* Route number of the highway.)
                           prevHighway I9
    milePost 276           doc (* location on the highway)
                           dataType Integer
    totalWeight 10         doc (* Current weight of cargo in tons.)
                           upperLimit 12
    stoppingPlace FortWorth
                           reason AuditRecord12
    arrivalTime 1400
                           certaintyFactor .8
```

One is to keep a history of values for a variable. The variable *highway* has a property *prevHighway* to record the previous value for the variable. Properties can also be used to save data type information for dynamic checking of programs. This is illustrated by the variable *milePost*. Properties can be used to save constraints on values, such as the *upperLimit* property of *totalWeight*. In some knowledge engineering applications it is important to save a record of program inferences. For example, the *certaintyFactor* property of *arrivalTime* records a measure of the confidence in the estimate of arrival time, and the *reason* property of *stoppingPlace* saves a record of the reasoning step that led to this choice of a place to stop.

These properties have no special significance to the Loops kernel. Loops just provides a way of associating the property lists with data so that application programs can find them starting with the data and interpret them appropriately. Like active values, the idea of extendible property lists for variables has its historical roots in frame languages. Alternative ways of annotating values have been tried in actor and constraint languages (e.g. Steele, 80). The major variations in Loops are that property annotations are objects which are created on demand, can be recursively nested, and share much of the implementation of active values.

1.3 Recursive Annotations. Annotated values can be nested. Nested property notations enable what we loosely call "descriptions of descriptions". Nested active values enable the programming of multiple independent side-effects. Both kinds of annotated values have an instance variable conventionally named *localState*, which is used to hold the data. When annotated values are nested, they are arranged in a chain such that the outermost annotated value points to an inner one through its *localState*, and the innermost annotated value contains the ultimate datum.

When active values are nested, then *GetWrappedValue* methods or *PutWrappedValue* methods are procedurally composed. Getting a value eventually causes all of the *Get* methods to be invoked. Similarly, putting a value causes all of the *PutWrappedValue* methods to be invoked.

Loops allows programmatic control of the order of access to localState of an activeValue and other operations. Different common applications require variations in the time ordering of the side-effect and accessing the *localState*. For applications like checking a constraint, it is appropriate to check the constraint *before* the new value is stored. For applications such as computing the sum of the new *localState* and other data (say to maintain a derived value for the sum of a set of figures), one first gets data and then does the summation. A more extensive set of examples is summariazed in the following chart:

Operation	Order of Side-effect	Side-effect for Application
Put	Side-effect first.	Check a constraint.
Put	Side-effect second.	Update a Gauge.
Get	Side-effect first.	Check access privileges.
Get	Side-effect second.	Combine with other data.

Language support for attached procedures has historically aspired to provide a discipline for controlling interactions among multiple attached procedures. To this end two issues have been addressed -- classifying the kinds of trigger conditions and specifying the order and conditions of execution for multiple procedures. For example, FRL provided event categories for triggering such as *if-needed, if-added,* and *if-removed.* In Loops the only triggering events are the fetching and storing of data. Specialization of fetching and storing must be programmed in the access methods of the active values. Other "events", such as object creation, are handled by methods on objects which can be specialized (thus taking advantage of the integration with the object oriented paradigm).

Frame languages have taken various approaches to specifying the order of invocation for multiple procedures. KL-ONE distinguished between three types of triggers -- *pre, post,* and *after.* Other frame languages have used an ordered list to specify the order, with exceptions indicated by the use of special tokens returned by the procedures as they are executed. We have found these sublanguages unneccessary for the common case of composed procedures and too weak for the exceptional cases of complex ordering anyway. Loops avoids introducing a sublanguage for control by using nesting of active values for the common case of functional composition, and using the control structures of LISP inside the methods for the complex cases.

Programming languages like Flavors and Smalltalk support specialized access methods for variables. However, these methods are applicable to all instances of a class. No language support is provided for dynamically attaching (and detaching) methods to individuals.

2. Applications of Access-Oriented Programming

Access-oriented programming in Loops started out not as a "programming paradigm", but rather as a minor variation of an implementation for attached procedures. As we have tried new applications, we have looked for ways to change Loops that would simplify our applications. These changes and simplifications led to the development of the access-oriented paradigm. This section presents a sampling of applications that have shaped the development of the paradigm.

2.1 Gauges. When a technician fixes a broken piece of electronic equipment, he brings to the task a collection of measuring tools, such as voltmeters and oscilloscopes. These tools enable him to observe the behavior of a circuit as registered by a *probe* that he attaches to the circuit paths. An analogous set of "instruments" is available in Loops that uses active values as probes for data. Figure *TruckGas* illustrates the attachment of a fuel gauge in a traffic simulator. An active value from the class *GaugeProbe* has been installed on the *fuel* instance variable of a truck object in the simulator. This active value connects the truck object to a gauge object. Whenever the simulator changes the value of *fuel* in the truck, the gauge object updates the image on the screen. The use of a gauge in Loops is very much like taking a meter off the shelf and attaching its probe to a circuit. One simply creates an instance of the appropriate gauge and sends it a message telling it to attach itself to the desired object variable. Figure 2 is a picture of the object hierarchy for the set of gauges standardly available in the Loops, and their screen representation.

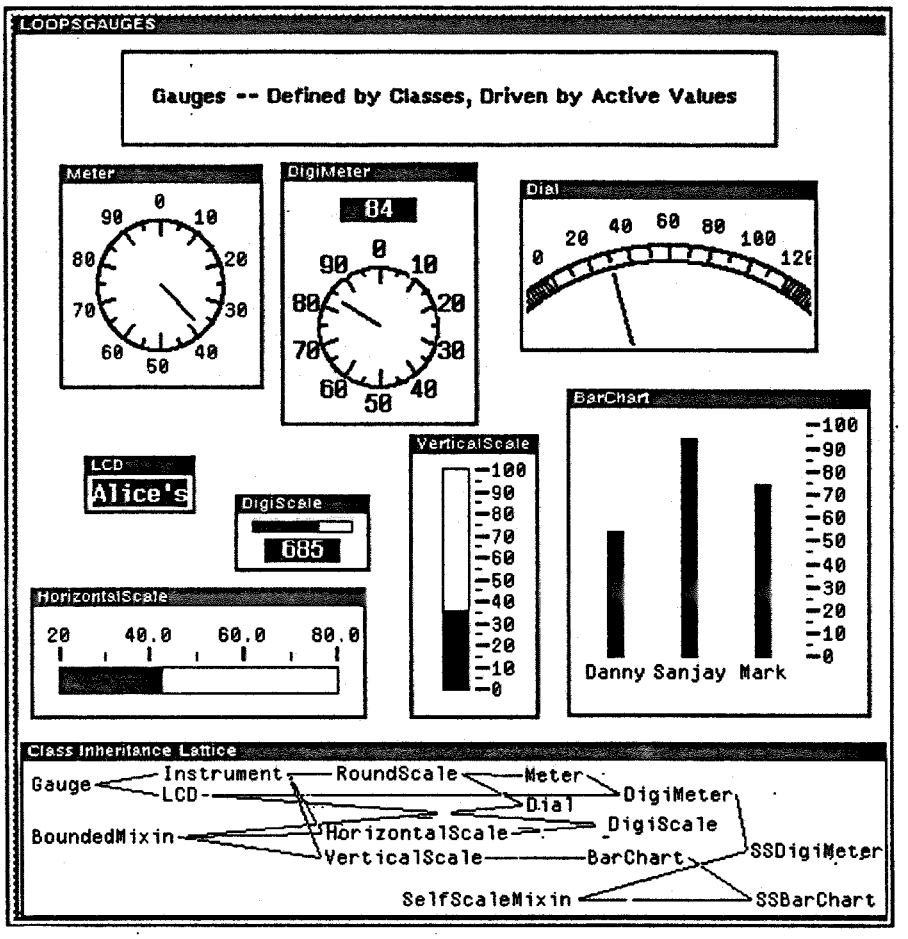

Figure 2: Gauges. *These gauges display the value of the vairable to which they are attached, and are updated when the variables are changed. Some like the veritcal scale on the left can rescale themselves if the value exceeds the gauge limit.*

A gauge is connected to the monitored variable through an active value, an instance of the *GaugeProbe* class. The important features of *GaugeProbe* are its variable *myGauge* and its specialized *PutWrappedValue* method. Its variable *myGauge* is analogous to the wire that connects a physical probe to its display instrument.

This arrangement also works for attaching *multiple* gauges on a single datum. In an earlier implementation of gauges, the active values used fixed properties of the monitored variables to save state, such as pointers to their associated gauges. In that arrangement special precautions were necessary to keep multiple gauges from interfering with each other. We now recognize that multiple gauges have *independent* purposes and should have independent resources. By representing *GaugeProbes* as Loops objects and using theirown state for storing state information, we eliminate an unwanted path of interaction among multiple probes. This principle simplifies the correct implementation of independent monitoring processes. It is one of the most important differences between active values and attached procedures.

We use gauges as a tool for instrumenting programs. Although gauge-like displays have been used in computer programs for years, their special attraction in Loops is that they can be attached to data in arbitrary programs *without changing the program.* To instrument data in most programming languages, it is necessary to find the all of the places in the source program that can change the data and then add code at each place to invoke a display package. Access-oriented programming makes it possible to annotate the program in only one place -- the variable to be monitored. This makes gauges considerably more

practical as debugging aids. They are also superior in their display technique to inserted print statements whose verbose output scrolls off a screen. Gauges provide a more focused way of monitoring program states.

2.2 Traps for Variables. Access traps are another application of active values that is generally useful for debugging. They are used to suspend program execution when some variable is referenced. The usual action wanted in a trap is an invocation of a debugging executive, such as the Interlisp BREAK package. Such traps are an important tool for identifying the conditions in a large program when some data are erroneously changed.

Loops provides several kinds of traps:

- *GetTraps* which detect when a program fetches a value.
- *PutTraps* which detect when a program stores a value.
- *AccessTraps* which detect both stores and fetches.
- *ConditionalTraps* which perform a trap operation only if an auxiliary condition is satisfied.

The example below shows the annotation of the object *Truck-37* with two traps. *Truck-37* has traps on the value of *stoppingPlace* and on the *upperLimit* property of *totalWeight*. When a program tries to change the value of *stoppingPlace* a break will be unconditionally invoked. The *ConditionalTrap* on *upperLimit* illustrates the use of an auxiliary condition for determining when to invoke a break. In this case the break is invoked only if the new value for the property is greater than 15.

```
Truck-37
  totalWeight #[ConditionalTrap 12
        when (GREATERP newValue
            (@ containingObject totalWeight::upperLimit))]
        upperlimit 15
  stoppingPlace #[PutTrap FortWorth]
```

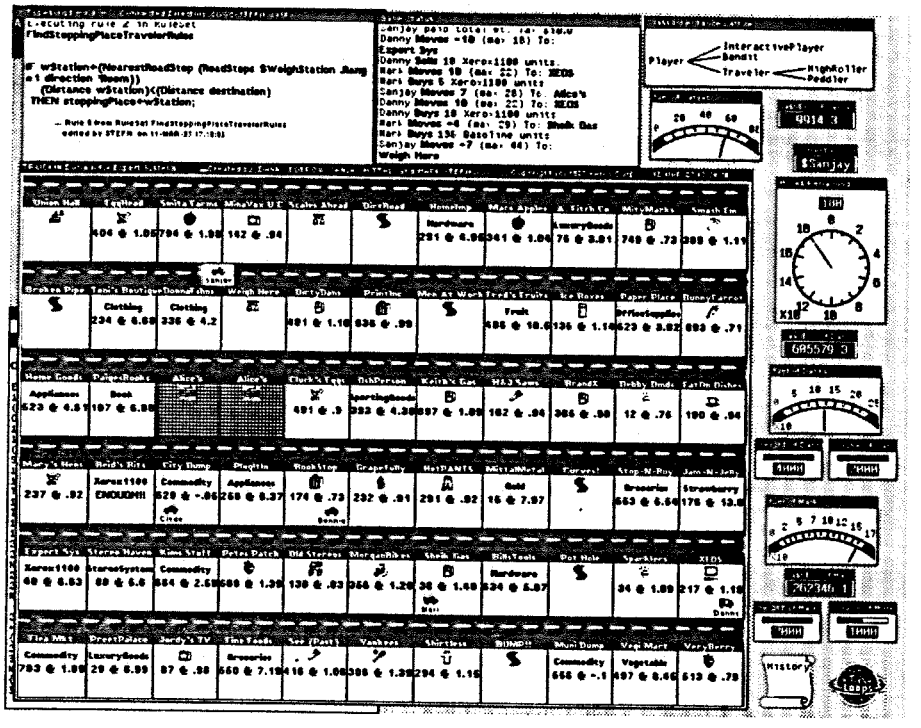

Figure 3. Truckin: *This is a display from the Truckin' game. Trucks are controlled by user programs, must buy and sell goods, and must avoid the bandits in black cars. The display is maintained by active value probes into the simulation.*

Historically access traps have mostly been used with computers that have a special built-in trap register. Active values bring this capability to a high-level language and allow multiple variables to be monitored simultaneously.

A generalization of the *trap* idea is to check new data against constraints before it is stored. An important kind of constraint is a check of the type of data being stored. This kind of specification is of central importance in "strongly typed" computer languages, but implemented as a run-time constraint with active values rather than as a compile-time check of the source.

2.3 Truckin'* and the *Track Announcer. One of the largest programs written in Loops is the *Truckin'* knowledge game that we use in teaching courses about Loops (Stefik83). The *Truckin'* program and related programs make extensive use of access-oriented programming.

Truckin' is a board game inspired in part by Monopoly. Figure 3 shows a snapshot of a *Truckin'* game board. The board has road stops arranged along a highway. (The highway implicitly loops along the edges of the board.) The players in the game drive trucks around buying and selling commodities. Their goal is to make a profit. The game is based on a relatively complicated simulation, which includes such things as road hazards, perishable and fragile goods, bandits, gas stations, and weigh stations. An unusual feature of *Truckin'* is that the "players" are actually computer programs developed by the students taking our course. We use the game as a rich and animated environment for teaching principles of knowledge programming.

One of the design issues in implementing *Truckin'* was to find a way to ensure that the picture of the game board is always up to date with the underlying simulation. Several people were involved in writing the *Truckin'* simulator and there were many places where the values in the road stops could be changed either by direct action of the game master or by the necessity of maintaining some constraints. In analogy with the simulator/display controller example cited earlier, our approach was to connect the screen image of each road stop to the underlying object variables. Each road stop image in the display is a sort of gauge monitoring part of the simulator.

A related program is the *Truckin' Track Announcer*, written by Martin Kay. The visual display of *Truckin'* changes quite rapidly during a competition. One of the

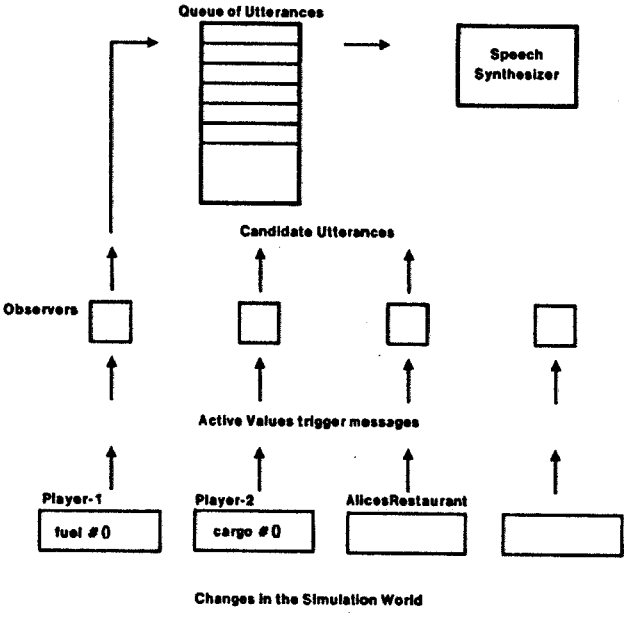

Figure 4. Martin Kay's Track Announcer: *Observer demons watch for interesting patterns in the Truckin' game that can be converted to utterances. A new observer can be inserted at any time using an active value probe.*

Roberts, R. Bruce, and Ira P. Goldstein, "The FRL Primer", Report AIM-408, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA July 1977

Sanella, M. The Interlisp-D Reference Manual, Xerox Corporation, 1983

Steele, G. "The definition and implementation of a computer programming language based on constraints" AI-TR-595, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1980

Stefik, M., D. Bobrow, S. Mittal, and L. Conway. "Knowledge Programming in Loops: Report on an Experimental Course", *The AI Magazine*, Fall 1983.

Weinreb, D. and Moon, D., *Lisp Machine Manual*, Symbolics Corporation, 1984