Integrating Access-Oriented Programming into a Multiparadigm Environment

Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn, Xerox Palo Alto Research Center

The Loops knowledge programming system integrates function-oriented, object-oriented, rule-oriented, and—something not found in most other systems—access-oriented programming.

The Loops knowledge programming system contains a number of integrated paradigms of programming. It builds on the function-oriented programming of Interlisp-D 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 that 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.

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 modules, called the simulator and the display controller. (This example was inspired by related work in Smalltalk on the partitioning of some programs into models, views, and controllers.)

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.

A shorter version of this article appears in Conf. Record HICSS-19, Hawaii International Conference on System Sciences, January 8-10, Honolulu.
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 runtime. The process of gluing is dynamic and reversible. When a user tells the display controller to change the views, the controller 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 a traffic light turning 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, it triggers an active value that 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.

**Basic concepts of access-oriented programming**

In Loops there are two kinds of annotated values: property annotations and active values. Property annotations associate extendible sets of arbitrary properties 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:

- **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 noninterfering side effects.

**History 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 this article was written.

Access-oriented programming has historical roots in languages like Simula and Interlisp-D, which provide ways of converting record accesses into a computation for all records of a given type. It is also related to the virtual data idea in some computer architectures. For example in the Burroughs B5000, a tag bit associated with data caused data access to be converted into a procedure invocation.

More immediate predecessors are the ideas of procedural attachment from frame languages like KRL, FRL, and KL-One. Attached procedures are programs that are associated with object variables and that are triggered under specific conditions.

Access-oriented programming in Loops is intended to satisfy a somewhat different set of purposes than attached procedures. This has led to a synthesis of ideas with some important differences. Although a thorough historical review is beyond the scope of this article, we occasionally return to attached procedures to show how language features in Loops diverge from that work.

**References**


Annotations have a low computational overhead when there are no annotations and also when there are annotations that programs do not reference (such as unreferenced properties). This characteristic takes noninterference a step further. Not only are annotations invisible to programs that ignore them, but they also 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 allows the creation of descriptions of descriptions in the case of nested properties and of 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 (for example, using the value of a particular property). It is also important for programs that test annotations implicitly (for example, automatic testing for and triggering of active values).

It distinguishes annotated values from ad hoc data structures for annotations 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, because 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.

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 the ability to state concisely and without redundancy what is intended. This is a hallmark of appropriate language support for a paradigm. By eliminating excess verbiage, the programmer can focus on the essentials, having both less opportunity for mistakes and more easily understood programs.

An alternative approach would be to provide a functional interface for changing each variable that needs to be monitored. Active values eliminate the need for functional interfaces since they can be installed on each variable as needed.

Active values appear in variables of objects. Each active value contains a localState to hold the value that should appear in that variable. Since an active value is an object, it can also contain additional information in other variables of the active value object. So users can easily view the contents of an active value, an active value is shown as
#
{< activeValueClass > localState otherSlot1 Value1 . . . }

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 viewObjects (<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, DispObj1 and DispObj2, in the display controller. These objects respond by updating the views in windows of the display.

InformDisplayController is an active value class that updates objects in the display controller. Its special method for storing informs all elements contained in its instance variable viewObjects.

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 either specialize the GetWrappedValue protocol to specify side effects when data are fetched or specialize 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 Put-
WrappedValue 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.

**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. Instance Truck-37, shown below, contains several kinds of annotations that have been used in knowledge engineering applications.

<table>
<thead>
<tr>
<th>Truck-37</th>
<th>doc (*owner of truck)</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner PIE</td>
<td>doc (*Route number of the highway.)</td>
</tr>
<tr>
<td>highway 166</td>
<td>prevHighway 19</td>
</tr>
<tr>
<td>milePost 276</td>
<td>doc (*location on the highway)</td>
</tr>
<tr>
<td>totalWeight 10</td>
<td>doc (*Current weight of cargo in tons.)</td>
</tr>
<tr>
<td>stoppingPlace FortWorth</td>
<td>reason AuditRecord12</td>
</tr>
<tr>
<td>arrivalTime 1400</td>
<td>certaintyFactor .8</td>
</tr>
</tbody>
</table>

Properties can be used 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.

We have also considered an implementation in which the properties are stored on an object-wide basis and are separated from the annotated value. This has the disadvantage that the collection of properties of a variable cannot be manipulated as a single entity. The advantage is that variable access is not slowed down by the presence of properties.

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 (Steele3 is one example). The major variations in Loops are that property annotations are objects that are created on demand, can be recursively nested, and share much of the implementation of active values.

**Recursive annotations.** Annotated values can be nested. Nested property annotations enable what we loosely call descriptions of descriptions. Nested active values allow the programming of multiple, independent side effects. Both kinds of annotated values have an instance variable, conventionally named localState, used to hold the data.

When annotated values are nested, they are arranged in a chain so the outermost annotated value points to an inner one through its localState and so 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 GetWrappedValue methods to be invoked. Similarly, putting a value causes all of the PutWrappedValue methods to be invoked.

The first step in this sequence is that a PutWrappedValue message is sent to the outermost active value. The outer PutWrappedValue method then performs a computation and at some point needs to store the data in its localState.

This step involves checking whether the localState contains an annotated value. If not, the data is stored directly in localState. If yes, a PutWrappedValue message is sent to the nested annotated value using the same protocol as the outermost active value.

The code for performing this test and storing the data is inherited from AnnotatedValue and is invoked using —Super as shown below. The PutWrappedValue method invoked by the —Super is inherited from the class ActiveValue.

```
[LAMBDA (self newValue object varName path type)
  (* This is a specialized
  PutWrappedValue method
  for . . .)
  (* Specialized side effect code
  invoked before nesting goes here.)
  . . .
  (* This —Super stores localState
  or invokes a nested active value.)
```

January 1986
For multiple active values, the net effect is that all of the specialized PutWrappedValue methods for all of the nested active values will be invoked, and the data will ultimately be stored in the localState of the innermost one.

GetWrappedValue methods are based on a similar procedural template. Depending on how these procedural templates are filled out, a PutWrappedValue method or GetWrappedValue method can perform its side effects before accessing data in localState, after accessing data in localState, or in some combination of the two.

Language support for attached procedures has historically been used 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 that 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 unnecessary for the common case of composed procedures—and too weak for the exceptional cases of complex ordering anyhow. Loops avoids introducing a sublanguage for control by using nesting of active values for the common case of functional composition and by 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.

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.

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. For example, one can attach a fuel-gauge active value to the contents of the fuel tank of some truck in the traffic 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 pictures the object hierarchy for the set of gauges standardly available in the Loops and the set’s screen representation.

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Order of Side Effect</th>
<th>Side Effect for Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Side effect first</td>
<td>Check a constraint</td>
</tr>
<tr>
<td>Put</td>
<td>Side effect second</td>
<td>Update a gauge</td>
</tr>
<tr>
<td>Get</td>
<td>Side effect first</td>
<td>Check access privileges</td>
</tr>
<tr>
<td>Get</td>
<td>Side effect second</td>
<td>Combine with other data</td>
</tr>
</tbody>
</table>
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 their own 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 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. Gauges provide a more focused way of monitoring program states, with independent views of different aspects.

**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, and
- **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 to determine when to invoke a break. In this case, the break is invoked only if the new value for the property is greater than 15.

```plaintext
Truck-37
  totalWeight #fConditionalTrap 12
    when (GREATERP newValue
           (@ containingObject
            totalWeight: upperLimit))

  upperLimit 15
    stoppingPlace #fPutTrap FortWorth
```

Historically, access traps have been used mostly 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.

**Checking data types and constraints.** 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, where types are checked at compile time.

However, in a late-binding exploratory programming environment, it is necessary to check data types at runtime. This is done with **Access Traps**, which detect both stores and fetches. 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 to determine when to invoke a break. In this case, the break is invoked only if the new value for the property is greater than 15.

```plaintext
Truck-37
  totalWeight #fConditionalTrap 12
    when (GREATERP newValue
           (@ containingObject
            totalWeight: upperLimit))

  upperLimit 15
    stoppingPlace #fPutTrap FortWorth
```

Figure 2. These gauges display the value of the variable to which they are attached. The gauges are updated when the variables are changed. Some, like the vertical scale on the left, can rescale themselves if the value exceeds the gauge limit.

January 1986
vironment, type checking must be done at runtime and is easily implemented using active values. The example below illustrates some of the type checking notations in Loops.

In this example, the instance variable milePost has a property datatype with the value Integer used by various graphics programs, interpreters, and compilers. Since there is no active value on milePost, it will not be automatically checked during an arbitrary store operation. In contrast, the variable totalWeight has a CheckDataType active value.

During a store operation, this active value will detect an error (or trap) if the new value is not an integer. The case for the arrivalTime instance variable is similar, except that the property datatype has an IntegerSpecification as its value.

These specifications are special objects put on the property list of the variable. These objects have methods for checking more complicated kinds of constraints, such as numeric ranges. The stoppingPlace instance variable has a specification that checks the class lattice to verify that the value is an instance of a city in Texas. These specification objects are put on the property list (not inside the active value) because they are also used for other purposes, such as intelligent editing of the instance Truck-43.

Truck-43
milePost 276 datatype Integer

... totalWeight [CheckDataType Integer] datatype Integer
arrivalTime [CheckDataType Integer]
datatype #IntegerSpecification289

... stoppingPlace [CheckDataType FortWorth] datatype #CityInTexasSpecification

Indirection. Some computers have a way of tagging a memory location as containing an indirect address. Any attempt to fetch data from such a location causes data to be fetched instead from the indirect address. Storing data works analogously.

Active values provide an implementation for this idea so that fetches and stores on data cause references to some other object. The IndirectReference class of active values provides for referencing data indirectly in another object. It stores an access path to the real storage location.

This indirection example is fundamentally different in an important way from the gauge examples cited earlier. For gauges, when there are multiple active values, the order of nesting doesn’t matter much. Side effects are independent and it is enough to ensure that they are all carried out.

The case is different for IndirectReference. These active values do not expect other active values to be more deeply nested. Their correct operation requires that they be the most deeply nested values so they will have the last GetWrappedValue or PutWrappedValue operation.

To support this need, active values follow a protocol of informing nested active values when they are installed. This enables active values whose placement is critical to adjust the order of nesting.

Truckin’ and the Track Announcer. One of the largest programs written in Loops is the Truckin’ knowledge game. 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 (which 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 that 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 a course on knowledge programming in Loops.6 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. There are many places where the values in the road stops could be changed either by direct action of the game master or by the need to maintain some constraints.

As it was in 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 ideas that came up during the Loops courses was to augment the display with a sort of radio announcer, an automatic program that would generate interesting spoken commentary about the competition as it unfolded.

The design of the Track Announcer had some of the same constraints as the game display. It needed to be informed about relevant changes in the progress of the game. Furthermore, it needed to be developed separately from Truckin’ at a time when Truckin’ itself was still evolving. Figure 4 illustrates the basic architecture of the design. Active values were used to connect the Track Announcer to key variables in the players and road stops. These active values sent messages to a collection of objects called observers.
Observers are responsible for detecting interesting patterns of change in the game and then generating comments about them. These comments are placed on a queue of candidate utterances. A priority scheme is then used to select the next utterance to, say, send it to a speech synthesizer device.

The observers in the Track Announcer are like demons in the Planner language. Demons are programs triggered whenever specified conditions become satisfied. A key implementation consideration for demons is finding an efficient way to monitor the demon conditions.

While Loops does not currently support demons, the Track Announcer example suggests how active values could be used in their implementation. For example, a demon

![Image of Truckin' game display]

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

![Image of observer demons]

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.

**A merger of Lisp and object-oriented programming**

Researchers at Xerox PARC, drawing on their experience with object-oriented languages such as Smalltalk and Loops, have proposed an object-oriented programming standard for Common Lisp. The proposed extension, CommonLoops, also allows integration of the access-oriented programming paradigm.

CommonLoops is compatible with Lisp's functional programming style. Message sending uses the same syntax as function calls. Object-oriented capabilities are simple extensions of Common Lisp structure defining operations, while object space is defined as a natural extension of the type-space in Common Lisp.

CommonLoops uses the same style of names and syntax as Common Lisp to provide a simple and direct path into object-oriented programming for those accustomed to Common Lisp. Thus programs written in Lisp can become incrementally object-oriented.

CommonLoops's small kernel is easy to integrate into Common Lisp implementations and has fewer special features than most object-oriented languages. The use of metaclasses facilitates specialized object representations and variations of multiple inheritance.

The kernel provides several extensions to object-oriented programming, including method lookup based on the class of more than one argument (a multimethod) or the identity of a particular argument.

A version of the CommonLoops proposal was aired at the August 1985 meeting of the Common Lisp Objects Committee at the International Joint Conference on Artificial Intelligence. CommonLoops and the other proposals for the standard are reviewed in "Object-Oriented Programming: Themes and Variations" by Mark J. Stefik and Daniel G. Bobrow in the winter 1986 issue of *AI Magazine*.

Xerox is building a portable implementation of CommonLoops in Common Lisp for community experimentation.

—Daniel G. Bobrow, Kenneth M. Kahn, Gregor Kiczales, Larry Masinter, and Mark J. Stefik, Xerox PARC
compiler could convert a source description of demon conditions and generate an appropriate set of active values installed on the necessary variables.

Access-oriented programming is based on annotated values that associate annotations with data. These annotations can be dynamically attached to and removed from object variables and can be nested recursively. Loops provides three language features that accommodate the dynamic addition and deletion of annotations.

The first feature is that annotated values are invisible to programs that are not looking for them. Annotations can be added to programs without causing them to stop working.

The second feature is that annotated values are recursive. They can be added to data that are already annotated. Nested properties support the notion of descriptions of descriptions. Nested active values enable multiple, independent side effects on variable access.

The third feature is that annotated values are objects and have their own independent local variables for saving state. This contrasts with the technique of using variables or properties of the monitored object to save state. This language feature removes a path of potential interference among multiple active values intended to be independent.

These language features support the access-oriented paradigm. They provide concise expression of intent and facilitate program evolution.

References
5. A. Goldberg and D. Robson, Smalltalk-80—The Language and Its Implementation, Addison-Wesley, Reading, Mass., 1983.

Mark J. Stefik is a principal scientist in the Knowledge Systems Area in the Intelligent Systems Laboratory of the Xerox Palo Alto Research Center. His current research includes work on expert systems for design and languages for knowledge programming. He recently started the Colab project at Xerox PARC, which is exploring computer support for collaboration and problem solving in face-to-face meetings.

Stefik received his PhD from Stanford University in 1980 for his work on the Molgen program.

He is the book review editor for the journal Artificial Intelligence. Stefik serves as a councilor for the American Association for Artificial Intelligence.

Daniel G. Bobrow is a research fellow in the Knowledge Systems Area at Xerox PARC's Intelligent Systems Laboratory. His recent work includes design and implementation of CommonLoops, a proposed extension of Common Lisp for object-oriented programming.

He received his PhD in mathematics from the Massachusetts Institute of Technology with a dissertation on artificial intelligence.

Bobrow is editor-in-chief of the journal Artificial Intelligence and has been an editor of the Communications of the ACM. He was chair of the governing board of the Cognitive Science Society and served on the executive council of the American Association for Artificial Intelligence.

Kenneth M. Kahn is a research staff member in the Knowledge Systems Area at Xerox PARC's Intelligent Systems Laboratory. His work there includes designing and implementing CommonLoops, a proposed extension of Common Lisp for object-oriented programming.

Kahn received his masters degree and PhD in computer science from the Massachusetts Institute of Technology. After a year on the staff of MIT's AI laboratory, he became a visiting professor at the University of Stockholm in Sweden.

The authors' address is Intelligent Systems Laboratory, Xerox Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304.