

# LOOPS -- Data and Object Oriented Programming for Interlisp

Daniel G. Bobrow and Mark J. Stefik

Xerox Palo Alto Research Center, Palo Alto, California 94304, USA

**Abstract.** LOOPS is a programming system integrated into Interlisp. It includes: 1) object oriented programming with non-hierarchical class structure; 2) user extendible property list descriptions of the classes, their variables and their methods (e.g. for documentation, defaults, and constraints); 3) composite objects - a way of defining templates for related objects that are instantiated as a group; 4) data oriented programming using active values - a way of having a procedure invoked when the value of a variable is set or read; 5) a knowledge base facility providing long term storage of shared knowledge bases, support for the exchange of incremental updates (layers), and the representation of multiple alternatives. This paper describes the features of LOOPS, how they facilitate implementation of a core of knowledge representation features, and our experience using it.

## 1. Introduction

LOOPS is a system we have implemented in the Interlisp [14] programming environment to support knowledge representation experimentation. LOOPS contains both object oriented and data oriented programming features. For object oriented programming we have adapted and extended ideas from Smalltalk [7, 9] and the Flavors package [6] of MIT Lisp Machine environment [15]. In a more extensive paper, [2], we compare LOOPS to these and other systems, and provide the rationale for our design decisions and extensions.

In this paper we summarize the features of LOOPS and indicate how they support different knowledge representation features. Section 2 describes object oriented programming features in LOOPS. Section 3 discusses a general mechanism which allows data-oriented programming. In this programming paradigm, access to object variables can cause invocation of a procedure. The mechanism of "active values" which implements the data oriented programming style is the major new technical contribution in LOOPS. Section 4 describes the LOOPS layered knowledge base which is similar in intent to the PIE system [1, 8]. This layered knowledge base contains alternate views of a design world, and allows users to keep their own incremental modifications to community knowledge bases.

## 2. Object Oriented Programming

**Classes and Instances.** A good introduction to object oriented programming is found in the Byte magazine special issue on Smalltalk, particularly [7]. Here we will only summarize the those features for the LOOPS system. A LOOPS class is a (partial) description of one or more similar objects. An instance is an object described by a particular class. Every object within LOOPS is an instance of exactly one class. All instances have similar structure (variables). A class specifies the behavior of its instances in terms of their response to messages. The class associates selectors (LISP atoms) with methods, the Interlisp functions that respond to the messages. All instances of a class use the same selectors and methods.

**Variables and Property Lists.** LOOPS supports two kinds of variables - class variables used to contain information that is shared by all instances of the class, and instance variables, which contain the information specific to an instance. Both kinds of variables have names and values. In the class, there is a default value specified for each instance variable. LOOPS also provides user-extendible property lists for classes, their variables, and their methods. A property list on a variable can be used to store additional information about the variable and its value, e.g. documentation. Property lists and default values are features of LOOPS not found in Smalltalk or Flavors.

**Inheritance.** LOOPS classes exist in a network of classes. An object inherits its instance variable description and message responses from a class within this network. A particular class is defined by its local declarations and the declarations that it inherits from a list of super classes. Each super class gives a partial description of the behavior and structure of the class, as in Flavors. Slightly different classes can be made up by combining slightly different sets of these partial descriptions with local overriding. All information in a class is inherited by its subclasses unless the information is overridden in a subclass. This is implemented by a runtime search for the information, looking first in the class, and then depth-first recursively in the classes named in the class's *supers list*.

**Method Combination.** For specializing and combining methods, LOOPS provides two special message invocations, *+Super* and *DoMethod*. *+Super* searches the inheritance network for a method above the class from which the current method was inherited. In this way, *+Super* provides a form of relative addressing; it invokes the next more general method even when the specialized method invoking *+Super* is inherited over a distance. *DoMethod* is used for combining methods where one wants to compute either the name of the selector, or the class in which it is to be found. The key feature of *+Super* and *DoMethod* is that they access code indirectly. This indirection provides the same isolation and delimited sharing as ordinary subroutine calls in systems without message passing. More complicated combination makes use of LISP itself which has a rich repertoire of control mechanisms already available.

**Composite Objects.** LOOPS extends the notion of objects to make it recursive under composition, so that one can instantiate a group of related objects as an entity. This is especially useful when relative relationships between members of the group must be isomorphic (but not equal) for distinct instances of the group. Structural templates are classes which describe composite objects having a fixed set of parts. Parameters are parts of a template that are recursively instantiated when the template is instantiated. Multiple references to the same parameter are always replaced by references to the same instantiated instance.

Because the templates are classes, all of the power of the inheritance network is automatically available for describing and specializing composite objects. This contrasts with template data structures that are merely copied to yield composite objects. To make specialization convenient, an editor can create a new set of templates such that each template in the new set is a specialization of a template in the old set. One can selectively edit the new templates describing the new composite object. Unchanged portions of a template will inherit default values from the parent composite object. To add parameters, one creates references to new templates. Conversely, one can make a parameter into a constant by overriding an inherited variable value with a non-template in a subclass.

**Applications to Knowledge Representation.** There is a core of features common to frame based knowledge representation languages (e.g. KRL [3], UNITS [11]) which are supported by objects. This core, which has evolved in response to experience in building problem-solving systems, includes an inheritance network of nodes, slots with properties and default values. Inheritance enables the easy creation of objects that are "almost like" other objects with a few incremental changes. Inheritance avoids copying redundant information and simplifies updating, since information that is inherited need be changed in only one place.

Default values reflect the idea that much reasoning takes place in the absence of specific information and that many of the assumptions that are made in problem solving can be represented as default values. Reasoning with assumptions often requires the ability to revise earlier decisions (belief revision) and this requires remembering how decisions were made. Such bookkeeping amounts to keeping *dependency records*, which can be conveniently be stored on the property list of variables. Property lists can be used to store such information as *support* (i.e., reasons for believing a value), *certainty factors* (i.e., numeric assessments indicating degree of belief), *constraints* on values, and *histories* (i.e., previous values). Finally, expert systems which create large structures (e.g., in VLSI design) need easy ways to create complex sets of objects with defined interrelationships. This is supported by our composite object facility which supports use of structural templates in an inheritance network.

## 3. Data Oriented Programming

The data oriented programming metaphor allows the invocation of a procedure as a side effect of accessing data. It is supported in LOOPS by the use of a specialized data structure called an *active value*. When the value of a variable is an active value, a specified implicit access procedure is invoked when the value is read or set. This mechanism is dual to the notion of messages; messages are a way of telling objects to perform operations that can change their variables as a side effect; active values are a way of modifying access to variables so that messages are sent as a side effect. The two mechanisms provide important complementary ways of factoring programs to provide modularity and to control interactions.

The following notation for an active value illustrates its three parts:

*#(localState getFn putFn)*

The *localState* is a place for storing data. The *getFn* and *putFn* are the names of functions that are applied with standard arguments when a program tries to get or put the value of a variable. Active values enable one process to monitor another one. For example, we have developed a LOOPS debugging package that uses active values to trace and trap references to particular variables. Another use is a graphics package that updates views of particular objects on a display when their variables are changed. This is described in more detail in the model-viewing example below. In both cases, the monitoring process is invisible to and isolated from the monitored process. No changes to the code of the monitored object are necessary to enable monitoring.

**Model Viewing Example.** Suppose that we want a program that simulates the flow of traffic in a city and displays selected parts of the simulation on a screen. Active values enable us to divide the programming of this example into two parts: the traffic model and the viewer. The traffic model consists of objects representing automobiles, traffic lights, emergency vehicles, and so on. These objects exchange messages to simulate traffic interactions (e.g., when a traffic light turns green, it would send *Move* messages to start cars moving). The viewer contains information about how the objects are to be displayed, for

example, through windows into different parts of the city. A user wants to be able to move these windows around to change the view. To view a particular automobile, an active value is created for its *position* variable:

```
[Automobile-1
 (position # (Pos) NIL UpdateDisplay)
 displayObjects (DispObj1 DispObj2 DispObj3)
 doc (" position of car in traffic coordinate system)
 (speed 25)) ...]
```

This active value is the interface between the object in the simulation model and the viewer. Whenever a method in the simulation model changes the value of this *position* variable, the active value *putFn* procedure *UpdateDisplay* is invoked. *UpdateDisplay* updates the local value and sends a message to each of the view objects in the list stored as a property of *position*. These objects respond by updating the view in the windows on the display screen. This example shows how a viewer can be invoked as a side effect of running the simulation. The view can be changed without affecting any programs in the simulation model. To change the set of simulation objects being monitored, only the interface to the viewer needs to be changed by adding active values.

**Embedded Active Values.** Sometimes it is desirable to associate multiple implicit access functions with a variable. For example, we may want more than one process to monitor the state of some objects (e.g., a debugging process and a display process). To preserve the isolation of these processes, it is important that they be able to work independently. LOOPS uses nested active values as a way of composing these functions. An active value can be stored in the *localState* of another active value that is the immediate value of a variable. Put operations to a variable with such nested active values activate the *putFns* from the outermost to the innermost. Get operations work in the opposite order. Each *getFn* sees only the value returned by a nested *getFn*, and the innermost *getFn* sees the value stored in its *localState*.

For example, suppose we wanted to trace access to the *position* variable of *Automobile-1*. The active value for *position* is then:

```
# ( # (Pos) NIL UpdateDisplay) GettingTracedVar SettingTracedVar)
```

An attempt to set *position* would cause *SettingTracedVar* to be called with the new value as one of its arguments. *SettingTracedVar* would run and call the LOOPS function *PutLocalState* to set its own *localState*. This would activate the inner active value causing *UpdateDisplay* to be invoked.

Just as inheritance from multiple super classes works most simply when the super classes describe independent features, active values work most simply when they interface between independent processes using simple functional composition. Any more sophisticated control than composition through nesting is seen as overloading the active value mechanism. More complex cases combine the implicit access functions using Interlisp control structures to express the interactions.

**Application to Knowledge Representation.** Active values provide a mechanism for implementing attached procedures, an idea which has been used in several knowledge representation languages (e.g. KRL [3] and Units [11]). Some problem-solving systems work by actively maintaining and propagating constraints between values [10]. The active value mechanism can be used to invoke constraint maintenance machinery.

#### 4. Knowledge Bases

LOOPS was created to support a design environment in which there is interaction in a community of designers. This is facilitated by use of shared community knowledge bases to which people can add incremental updates. We have chosen the term *knowledge base* instead of *data base* to refer to our long term storage facility to emphasize the intended application of LOOPS to expert systems. In expert systems, knowledge bases contain inference rules and heuristics for guiding problem solving in addition to libraries of previously designed subsystems. This also contrasts with tabular files of facts usually associated with data bases.

**Layers.** Knowledge bases in LOOPS are files that are built up as a sequence of layers, where each layer contains changes to the information in previous layers. A user can get the most recent version of a knowledge base (that is, all of the layers) or any subset of layers. The second option offers the flexibility of being able to share a community knowledge base without necessarily incorporating the most recent changes. It also provides the capability of referring to or restoring any earlier version.

**Community Knowledge Bases.** LOOPS partitions the process of updating a community knowledge base into two steps. Any user of a community knowledge base can make tentative changes to a community knowledge base in his own (isolated) environment. These changes can be saved in a layer of his personal knowledge base, and are marked as associated with the community knowledge base. In a separate step, the person acting as knowledge base manager can copy such layers into a community knowledge base. This separation of tasks is intended to encourage experimentation with proposed changes. It separates exploring possibilities from the responsibility of maintaining consistent and standardized knowledge bases for shared use by a community. The same mechanisms can be used by two individuals using personal knowledge bases to work on the same design. They can conveniently exchange and compare layers that update portions of a design.

**Environments.** A user of LOOPS works in a personalized *environment*. An environment provides a lookup table that associates unique identifiers with objects in the connected knowledge bases. In an environment, users indicate dominance relationships between selected knowledge bases. When an object is referenced through its unique identifier, the dominance relationships determine the order in which knowledge bases are examined to resolve the reference. By making personal knowledge bases dominate over community knowledge bases, a user can override portions of community knowledge bases with his own knowledge bases.

#### 5. Current Status

LOOPS has been used for Palladio, our Knowledge Based VLSI Design [4] project for several months. It has been a valuable addition to our programming tools in Interlisp. It has allowed us to use data and object oriented programming styles where it suited the application with all of the previously developed tools in Interlisp. LOOPS can be ported to any machine running Interlisp; we have run our programs both on Xerox 1100 Scientific work stations [5] and on the PDP-10.

The LOOPS facilities have been most useful in designing a simulator for a high level hardware design language (LMA for Linked Module Abstraction [13]). In this application, objects are used to represent runnable parts of a digital system, described abstractly as a token and data passing network. We wanted to be able to show that the running behaviors of these objects corresponded exactly to the set of possible behaviors of the (possibly parallel) elements in the formal LMA description. We used the multiple inheritance network to organize the inheritance of methods describing behavior and variables describing buffers and control state. We found that the ability to inherit code and variables greatly reduced the total amount of code, and thereby made it easier for us to check the correctness properties of the code. LOOPS has also been used in our project to implement a graphics package. A LOOPS manual [12] and more extensive paper [2] are available.

**Acknowledgments:** Thanks to the other members of the KBVLSI project for suggestions on LOOPS and for enduring the wait for it to come into existence while so many things have been pressing. Thanks especially to Phil Gerring and Gordon Novak for helpful bug reports and suggestions. Johan de Kleer for extensive discussions of design issues, and to Richard Fikes, Adele Goldberg, Danny Hillis, Dan Ingalls, and Chris Tong for comments on earlier drafts of LOOPS's manuscripts. We are grateful to Larry Masinter and the Interlisp-D group for unfailing support and encouragement. Thanks also to Lynn Conway for encouraging this work and to the Xerox Corporation for providing the intellectual and computing environments in which it could be done.

#### References

1. Bobrow, D. G., & Goldstein, I. P. Representing design alternatives. *Proceedings of the AISB Conference*, Amsterdam, 1980.
2. Bobrow, D. G., & Stefik, M. Data-Oriented and Object-Oriented Programming, New Metaphors for Lisp. Knowledge-Based VLSI Design Group Memo KB-VLSI-81-14, August 1982.
3. Bobrow, D. G., & Winograd, T. An overview of KRL, a knowledge representation language. *Cognitive Science* 1:1, 1977, pp 3-46.
4. Brown, H. A., Palladio, An Expert Assistant for Integrated Circuit Design (Submitted to AAAI82).
5. Burton, R. R., Kaplan, R. M., Masinter, L. M., Sheil, B. A., Bell, A., Bobrow, D. G., Deutsch, L. P., Haugeland, W. S. Papers on Interlisp-D. *Cognitive and Instructional Sciences Series CIS-5*, Xerox Palo Alto Research Center, September 1980.
6. Cannon, H. I. Flavors: a non-hierarchical approach to object-oriented programming. *personal communication*, 1982.
7. Goldberg, A. Introducing the Smalltalk-80 System, *Byte* 6:8, August 1981.
8. Goldstein, I. P., & Bobrow, D. G. Extending object oriented programming in Smalltalk. *Proceedings of the Lisp Conference*, Stanford University, 1980.
9. Ingalls, D. H. The Smalltalk-76 programming system: design and implementation. *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978.
10. Steele, G. L. Jr., The Definition and Implementation of a Computer Programming Language Based on Constraints AI-TR 595 MIT Artificial Intelligence Laboratory 1980
11. Stefik, M. An examination of a frame-structured representation system. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Tokyo, Japan, August 1979, pp. 845-852.
12. Stefik, M. & Bobrow, D. G. The LOOPS Manual: A data oriented and object oriented programming system for Interlisp. Knowledge-Based VLSI Design Group Memo KB-VLSI-81-13, August 1981.
13. Stefik, M., Bobrow, D. G., Bell, A., Brown, H., Conway, L., and Tong, C. The partitioning of concerns in digital system design. *Proceedings, Conference on Advanced Research in VLSI*, Artech House, January 1982., pp. 43-52.
14. Teitelman, W. *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.
15. Weinreb, D., & Moon, D. *Lisp Machine Manual*, Massachusetts Institute of Technology, 1981