LOOPS is a Data and Object Oriented Programming system for Interlisp. It includes: 1) A programming environment to support object oriented programming. 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 [13]. In a more extensive paper, [19], 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 pattern 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 object 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 FL system [1, 8]. This layered knowledge base contains alternative views of a design world, and allows users to define their own incremental modifications to community knowledge bases.

2. Object Oriented Programming

Classes and Inheritance. A fundamental introduction to object oriented programming is found in the Byte magazine special issue on Smalltalk, particularly in [9]. We will only summarize the basic 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 (variable) structures. 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.

Properties and Property Lists. LOOPS supports two kinds of variables - class variables and instance variables. Information 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. This default value is also provided as an extension feature for other classes, and 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 superclasses. Each superclass 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 provided 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 involves the general method even when the specialized method invokes +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 LISP itself which has a rich repertoire of control mechanisms already available.

3. Data Oriented Programming

The data oriented programming metaphor allows the invocation of a procedure as a side effect of accessing data in LOOPS. 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 side effects that can change the 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:

(Modeling the local state)
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 #(Post NIL UpdateDisplay)
  displayObjects (DispObj DispObj DispObj DispObj) doc (* position of car in x-y coordinate system*)
  (speed 2)] --
```

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 then update the view of the window 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 an immediate parent of a variable. Furthermore, such nested active values activate the putFn 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 Automobile1. The active value for position is then:

```
[ #(Post NIL UpdateDisplay) GettingTracedVar SettingTracedVar]
```

An attempt to set position would cause SettingTracedVar to be called with the new value at 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 functions. 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 Interloop 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 systems (e.g., KRL [1] and its Unis [12]). Some problem-solving systems work by actively maintaining and propagating constraints between values [12]. 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 between the designer and the objects of design. This environment 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 export 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, each layer containing the information necessary to facilitate inferences 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 by making an 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 integrate 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 project for several months. It has been a valuable addition to our programming tools in Interloop. 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 Interloop. LOOPS can be ported to any machine running Interloop; we have run our programs both on Xenon 1100 Scientific work stations [8] and on the FDP-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 [13] and more extensive paper [12] are available.

**Acknowledgements:** 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 Gereau and Gordon Novak for helpful bug reports and suggestions. To Richard Finn, Adel Wassef, Danny Hills, Dan Ingalls, and Chia Tung for comments on earlier drafts of LOOPS's manuscripts. We are grateful to Larry Manaster and the Interloop-D group for stimulating discussions, and to Lynn Conway for encouraging this work and to the Xerox Corporation for providing the intellectual and computing environment in which it could be done.

**References**