KNOs: *KN*owledge Acquisition, Dissemination, and Manipulation *O*bjects

D. TSICHRITZIS, E. FIUME, S. GIBBS, and O. NIERSTRASZ
Université de Genève

Most object-oriented systems lack two useful facilities: the ability of objects to migrate to new environments and the ability of objects to acquire new operations dynamically. This paper proposes Knos, an object-oriented environment that supports these actions. Knos' operations, data structures, and communication mechanisms are discussed. Kno objects "learn" by exporting and importing new or modified operations. The use of such objects as intellectual support tools is outlined. In particular, various applications involving cooperation, negotiation, and apprenticeship among objects are described.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**] Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*data types and structures*; H.4.1 [**Information Systems Applications**]: Office Automation; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods.

General Terms: Design, Languages.

Additional Key Words and Phrases: Distributed knowledge, messages, objects, office and application support tools

1. INTRODUCTION

One of the main reasons for the advent of Office Information Systems is the lack of equipment and tools in offices. It is often pointed out that the set of capital equipment at the disposal of an average office worker is inferior to that of an industrial worker. A collection of tools, including electronic mail, word processing, spreadsheets, graphics, and database systems, is slowly changing the office environment. All of these tools provide the office worker with the equivalent of machine tools and power drills, that is, with superior tools for the mechanization of routine aspects of office work. In the meantime, however, manufacturing workers have progressed well beyond power tools. A formidable array of robots is already revolutionizing manufacturing. Office workers are again one step behind. They do not have the equivalent of "white-collar robots" to help them do their daily work.

There are many reasons for this situation. First, the mechanization of office activities came later than that of the factory. Second, office work is far less

Authors' address: Centre Universitaire d'Informatique, Université de Genève, 12, rue du Lac, CH-1207, Genève, Switzerland.

structured and rather difficult to automate. Third, intellectual work is not visible (although it may have visible effects). What is not visible can be difficult to understand and therefore difficult to automate. Finally, office work is considered overhead by many organizations, and it is not receiving as close attention as factory work.

All of these factors are slowly changing. We should therefore start exploring the notion of "automated office assistants" for office workers. Since office work is intellectual, we are not talking about robots performing mechanical tasks. Instead, we are talking about computer tools that perform intellectual tasks. These tools may replace some of the work by office workers, but the techniques used by a person may differ from those used by the automated counterpart. For example, electronic mail does not consist of little robots carrying paper messages. A different medium and transport is used to achieve the same result, that is, to transmit a message from one person to others. In trying to provide office workers with intellectual computer tools, we are faced with the need to isolate the tasks that can later be automated. In the same way, we first establish the need for a particular result, as, for example, a hole, in manufacturing—we instruct a robot to reach out and create that hole.

We can broadly define two categories of tasks. Low-level tasks usually involve repetitive, routine work for which the inputs and outputs are well defined and the procedures followed are relatively clear. This type of task can be automated with the specification of automatic procedures, as has been proposed and implemented in many systems [9, 18, 19].

High-level tasks, on the other hand, are neither clear nor routine. They involve cooperation among many agents, negotiation among parties, confrontation and argumentation, and the ability to set and reach goals. In addition, some high-level tasks cannot be established a priori. We cannot, therefore, rely entirely on an exhaustive study or modeling of offices that defines their procedures. The tools we use should be capable of adapting and learning. If they cannot learn from example (i.e., by extrapolating from specific instances), they should at least be capable of apprenticeship. That is, they should be able to acquire new behavior from other tools and from human beings [13].

Our goal is to build a system in which it is possible to model the following several difficult problems effectively:

(1) problems in which knowledge about an overall system is distributed among several environments (e.g., real-time remote sensing, understanding trends in international finance, distributed office environments);
(2) problems in which information is fuzzy (e.g., language understanding, vision systems);
(3) problems that require negotiation and strategy development and refinement, (e.g., automated bargaining, planning, testing hypotheses);
(4) problems that require learning or adaptation, that is, problems in which information is dynamically changing (e.g., automatically tailoring and optimizing a user interface to a user, adapting to behavioral trends) or incomplete (e.g., high-level office tasks).
(5) problems requiring simulation.

In this paper we concentrate on advanced tools for office environments. We propose tools that can be considered automated office assistants, that is, tools that can take over some of the tasks in offices involving cooperation, negotiation, and learning by apprenticeship. We occasionally consider analogies for such tools. We establish these analogies for two reasons: first, in order to visualize and remember the operation of such tools, and second, because the result of their operation can be explained more easily in human or animal terms. For example, consider a hole in an industrial product. It is easier to design and control the robot that makes the hole if we have a visualization of a person reaching out with a drill and making the hole. This metaphor could hinder us from thinking about a better way to do the same thing (e.g., using a laser beam to make the hole). It helps, however, in defining an initial solution and explaining the operation.

We will call the proposed tools *Knos* (pronounced "nose") for *KN*owledge acquisition, dissemination, and manipulation *O*bjects [15]. Their purpose, loosely speaking, is to manipulate fragments of knowledge with their own rules. They can also negotiate, cooperate, and learn from other objects and from users. Such object behavior is not easily realized by existing object-oriented systems. However, we have identified a number of characteristics that we believe are essential to realizing Knos. These include

—data abstraction,
—dynamic instantiation of objects,
—object autonomy,
—inheritance,
—the ability to acquire new operations (i.e., to "learn") dynamically,
—concurrency,
—a uniform communication mechanism,
—nomadic object migration to other environments

Below, we describe the general structure of a Kno environment, a prototype implementation in Zetalisp, a computational model of Knos, and a planned implementation in terms of an object-oriented language. The application of object-oriented methods to office systems has been explored by a number of researchers [1, 2, 5, 6, 12]. As will be seen, Knos differ from these earlier approaches in their high degree of autonomy, adaptability, and concurrency.

2. MACROSCOPIC VIEW OF KNOS

We first present Knos in a macroscopic way to get an idea of their behavior. In this way the reader can get a feeling for the difference in behavior between Knos and other object-oriented environments. In the next section we describe a Kno's internal structure in more detail.

A set of cooperating Knos exist within a *context*. Knos communicate within a context by reading messages from or writing messages onto a *blackboard*. A Kno can *move* itself to another context (assuming it is aware of the new setting). Typically, a context is physically associated with a workstation. Multiple contexts would then represent several workstations. Presumably, these contexts would be

in some way connected, perhaps by a local area network or telecommunications link. The administration of each context is controlled by an *object manager*. It is the object manager's job to oversee the local blackboard, to accept or reject move requests, and to decide whether or not to acquaint itself with other contexts (or object managers).

The fundamental novelty of Knos lies in the explicit support for two types of move actions:

—the migration of Knos to new contexts,
—the migration of new operations into Knos.

We call the first form of migration a *move* action and the second form a *learn* action. It will be seen later that Kno operations can similarly be forgotten, or "unlearned."

Figure 1 depicts three Kno contexts, S_1, S_2, and S_3. Several Knos can be seen to be moving from one context to another, and a Kno can be seen receiving a new rule from the blackboard. Under our workstation paradigm, a Kno would be transported as a network message. Most object-oriented systems do not have explicit move or learn operations. In other systems, all objects would typically be under the jurisdiction of one object manager. The possibility of multiple contexts either does not exist, or it is hidden from the objects. We cannot accept such an environment. First, Knos are supposed not only to support office activities, but also to generalize message systems. Messaging is closely related to moving and changing a context. Moving is, therefore, very important. Second, networks exist and cannot always be easily ignored. If the Kno paradigm were only intended to support a set of tightly coupled workstations linked by a local area network, we could still try to hide the network. However, Knos should be able to operate within global and even heterogeneous networks. In such cases it is beneficial for the existence of a network to be known explicitly.

Knos are persistent but mortal. A Kno dies as a result of a *die* action within one of its rules. We are currently examining the notion that when Knos die, their contents can be archived. Active Knos would then be able to consult the remains of dead Knos (e.g., for post-mortem debugging, for history mechanisms, audit trails). Hence the notion of a database Kno in Figure 1.

There are two special types of Knos that serve as external interfaces. A *user Kno* is a Kno that can be manipulated by an external user. Other Knos are not directly accessible to users, and in this way, common Knos are quite independent. User Knos, on the other hand, are like marionettes. A human user can, figuratively, hold the strings and can directly control their behavior. A user Kno could provide an animated sequence of images and sound that depicts the internal operation of a Kno environment [4].

An *agent Kno* is manipulated indirectly by a Kno in another context. As representatives of the object manager, agent Knos handle migration of Knos and forwarding of messages to other environments. A context may be created with one or more agent Knos that are acquainted with other contexts. Other agents may be created dynamically.

Knos communicate with one another by posting messages on a blackboard that is managed by the object manager. A Kno can export a message to, or import a

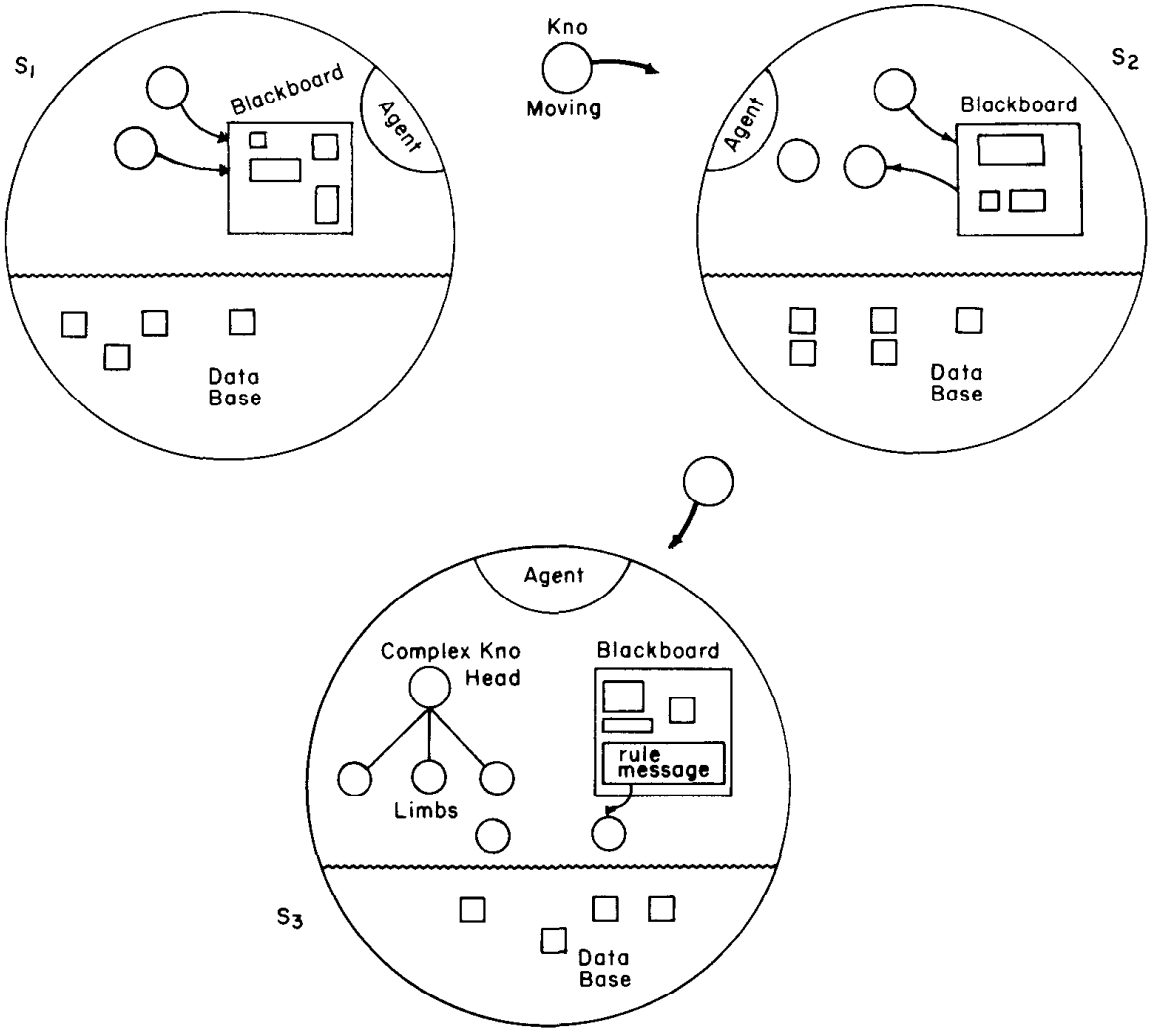

Fig. 1. Kno environment.

message from, the board. A message on the blackboard contains the sender Kno ID and possibly a target Kno ID. It also contains information that is exported from the sending Kno and imported by the receiving Kno.

Knos communicate through the blackboard for reasons of *autonomy* and *resilience*. Knos may travel very far and find themselves in hostile environments. If their only interface is the blackboard, no other Kno can force them to accept or even to look at a message. Each Kno is in complete control of its own operation. It chooses, when it wants, to accept a message from the blackboard and perform some actions as a result. The only acquaintance a Kno has when it migrates to a new context is the local blackboard by way of the basic message-passing operations.

When a Kno is in a friendly or cooperating environment, it may choose to make some of its behavior generally available to other Knos. These behaviors are called *operations*. For one Kno to request an operation from a second Kno, it must know the name of that Kno. An operation is a shorthand for a particular

pattern of message-passing actions between two Knos, using the blackboard for storing messages. Observe that there is at least as much concurrency within a context as there are Knos (the internal operation of a Kno might itself be concurrent).

All Knos are autonomous, except for one particular case. A Kno may grow another Kno as a *limb*. A limb is by definition dependent on the Kno (the *head*) that grew it. Limbs may move away from heads. A head and its limbs can span different object managers. The head is always notified of a limb's move (via an agent). We currently insist that the head remain stationary. Moving heads and limbs independently and arbitrarily gives rise to difficult distributed control problems. A group of one head Kno and its Kno limbs is called collectively a *complex Kno*. Complex Knos can be used for coordinating activities that occur in different workstations. They facilitate the development of "worm programs" in distributed environments [14]. For example, an active message gathering information can be implemented as a complex Kno [8, 16].

A Kno system is constructed from a group of object managers and a Kno name server. The object managers provide support for simple Kno behavior and allocate processing resources to Knos. The name server is used to locate Knos within object managers and object managers within the network.

3. KNO STRUCTURE

In this section we outline an initial Kno specification based on the LISP dialect known as Zetalisp [17]. The reason for choosing Zetalisp is that it provides an object-oriented programming facility (the flavor system) that can be used to prototype Knos and illustrate their basic features. Knos as objects can be implemented on top of a regular programming environment; for example, active messages in I-mail were implemented with C and UNIX[1] [8]. It is, however, more appropriate to implement messages in terms of an object-oriented environment, since it will allow us to take advantage of features common to both Knos and objects. Some details of the Kno structure presented in this section will change as we migrate toward a concurrent, distributed object-oriented implementation. The computational basis for this planned implementation is described in Section 5.

Each Kno is an instance of one or more Kno classes. Kno *classes* specify the (initial) structure and behavior of Knos. Kno *structure* refers to the instance variables contained within the Kno, while a Kno's *behavior* is determined by the operations it can perform. In our Zetalisp implementation, each Kno *operation* is composed of a set of production rules, described shortly. Typically, each Kno is monitor-like, in that only one operation may be active at a time [7]. However, it will be possible to provide a custom *Kno handler* (see Section 5) that can either strengthen this assumption (e.g., each operation is indivisible) or weaken it (e.g., several operations within a Kno can execute concurrently). The (inherited) default is a single-threaded execution within a Kno, but with operations not necessarily indivisible. The general (Zetalisp) form of a Kno class and operation

[1] UNIX is a trademark of AT&T Bell Laboratories.

specification is

```
(kno-def ⟨Kno class name⟩
  (⟨instance variable list⟩)
  (⟨inheritance list⟩)
  (kno-op (⟨Kno class name⟩ ⟨Kno operation name⟩)
    (⟨parameter list⟩)
    (⟨body⟩))
  .
  .
  .
)
```

Different Kno classes can be specified. However, there is a predefined Kno class that provides basic structure and behavior common to all Knos. This predefined class is called *basic-kno* and is specified as

```
(kno-def basic-kno
(kno-name kno-rules kno-limbs) ( )
  (kno-op (basic-kno :kno-init) ( ) (. . .))
  (kno-op (basic-kno :kno-learn) (kno-op) (. . .))
  (kno-op (basic-kno :kno-unlearn) (kno-op) (. . .))
  (kno-op (basic-kno :kno-die) ( ) (. . .))
  (kno-op (basic-kno :limb-death) (limb) (. . .))
)
```

In this case the basic-kno class has three instance variables. These are used for the unique name of the Kno, the rules possessed by the Kno, and the names of any limbs of the Kno. It is important to note that a Kno's rules and operations are kept as part of its instance data structure and thus may be modified by the Kno.

In the above schema five operations are listed (the full specification is not shown). These particular operations are used internally by the object manager and are not normally invoked by Knos. The first operation, :kno-init, initializes the instance variables of a newly created Kno. The operations :kno-learn and :kno-unlearn modify a Kno's set of operations. The :kno-die operation is invoked when a Kno dies; the :limb-death operation is invoked when one of a Kno's limbs dies.

A particular class defines a specific set of operations. When a Kno is created, its class and hence its operations are initially defined. Kno operations are similar to methods in languages such as Smalltalk or Zetalisp. However, as we said above, operations essentially denote certain patterns of message-passing behavior through a blackboard. It is possible, for example, that when one Kno invokes an operation from a second Kno, the two Knos need not be in the same context; the two Knos may actually be on different machines. Communication between two different contexts is handled by an agent Kno for the remote Kno.

Kno classes may inherit both structure and operations from other classes. In this prototype the Kno inheritance mechanism derives directly from the Zetalisp flavor system. Knos belonging to a certain Kno class would inherit all the instance variables and operations of that class, together with those of the class basic-kno.

In our Zetalisp implementation of Knos, we chose to code operations as production rules. There are many reasons for choosing production rules for Knos.

For example, production rules are simple to describe and read. On the other hand, control flow can be difficult to understand. Knos are intended for applications in which the entities of interest modify their behavior according to interactions between themselves and the environment. Production rules are well suited for such applications. In our planned object-oriented implementation, however, our target language will support features such as triggering [11], and we are therefore considering an alternative, block-structured approach to specifying Kno operations.

Kno production rules are of the form:

```
(rule ⟨rule-name⟩
  (trigger ⟨trigger condition⟩)
  (action ⟨action series⟩))
```

The ⟨trigger condition⟩ is a Boolean-valued expression that must be satisfied before the rule can be executed.

The ⟨action series⟩ is a series of ⟨actions⟩. All actions are executed when the rule is executed. A rule is indivisible.

Kno *actions* are the atoms of Kno behavior. Actions are of five different types:

(1) local actions,
(2) communication actions,
(3) existential actions,
(4) learning actions,
(5) limb actions.

Table I lists specific Kno actions according to their type.

A local action involves inspecting or changing an instance variable. Local actions affect the instance variables of only one Kno: the one performing the action.

3.1 Communication Actions

Communication actions allow Knos to pass messages among themselves using the local blackboard as an intermediary. Each blackboard message consists of three parts: a message type, a message body, and an expiry part. The message type is simply a name given to the message by the Kno creating the message. Typically, a set of reasonable message types would be placed in a library. The message body is an arbitrary expression; it is the responsibility of the message recipient to interpret the body. The expiry part indicates how long the message can remain on the blackboard. A message may remain on the blackboard indefinitely or be removed after it has been read or a certain time period has elapsed.

The *export* action allows one Kno to request an operation from a second Kno. In order to issue an *export*, the first Kno should know the name of the second, the name of the operation, and the parameters required by the operation. Export is of the form

```
export[KnoID | any | *] [wait] MessageType MessageBody ExpiryTime MessageBody =
  (OperationName parameters)
```

The target Kno of an export may be a specific Kno or any Kno willing to accept the message; or the message may be directed to all (nonagent) Knos in the

Table I. Kno Actions According to Type

Type	Actions
Local	Put, get
Communication	Import, export
Existential	Spawn, die, move, freeze, unfreeze
Learning	Act, learn, unlearn
Limb	Grow, kill, ship, teach, unteach

context. The optional "wait" subcommand states that the sender blocks until the message is consumed. Otherwise, export does not block. If the KnoID specified is that of an agent Kno, then the message can be forwarded by the agent to another context.

The *import* action allows the receiving Kno to retrieve one or more messages from the blackboard. It is of the following form:

import[KnoID | *] [OperationName | *] [1 | $ | *] [wait].

All operands are optional. The object manager returns to the requesting Kno a queue of all messages matching the import specification. An asterisk in the specification matches any KnoID or OperationName. It can ask for the (chronologically) first message matching the pattern (by specifying "1"), the last message (by specifying "$"), or every message (by specifying "*"). The default is "1". Finally, if no messages match the specification, the Kno can instruct the object manager that it would like to wait until a matching message arrives.

3.2 Existential Actions

Some Kno actions seriously affect Kno existence. They are related to moving and generating other Knos. They are called existential actions. We discuss them separately to emphasize their operation. There are five existential actions: move, spawn, die, freeze, and unfreeze.

The *move* action specifies a new context for the Kno. The execution of the action will move the Kno as a whole to the context of a new object manager. When a Kno moves, its entry is updated on the Kno name server.

The *spawn* action creates a new Kno of a particular class. The spawned Kno is from this point on like any other Kno. It then executes independently.

The *die* action is used by a Kno to terminate execution voluntarily. When a Kno dies, all its limbs (see below) are killed.

The final two existential actions, *freeze* and *unfreeze*, are used to convert a Kno to and from a static representation known as a Kno description. Using this action, all information contained within a particular Kno can be embedded in the structure of a second Kno. In this manner, it is possible to build Knos that interpret other Knos.

Another use of freeze and unfreeze occurs in the implementation of the move action. When a Kno moves from one context to another, it moves in a state of suspended animation. Knos are frozen by the sending object manager and unfrozen by the receiving object manager.

3.3 Learning Actions

Knos can pass operations to one another in the various ways described earlier. A receiving Kno can then add a new operation to its own list of operations if it so desires. One can say that a Kno *adapts* or *learns* new operations. Observe that our primitives support learning from other Knos, but not the potentially more difficult notion of learning by example.

Knos can also pass Knos as values among themselves (as produced by the freeze action). We have two mechanisms for enriching a Kno with rules and descriptions. One is a mechanism for acting, and the other is for learning. Acting implies the execution of foreign operations, but under constant monitoring by the acting Kno. Essentially, the *act* action invokes a desired operation and traces its effect on the instance variables of the Kno. This trace can then be presented to the Kno for consideration.

Learning implies the incorporation of foreign operations in a Kno's own body. The learn action is used for this purpose. It takes an imported operation and adds it to the Kno's operation list. The imported operation subsequently has the same status as the indigenous Kno operations. It is also possible, through the use of the *unlearn* action, for a Kno to remove an operation from itself.

A Kno can use the learn action in a very simple fashion to add any new operation it encounters to its operation list. It is also possible to specify more complicated Kno learning behavior. An example would be a Kno that applies various selection criteria to the operations it is asked to learn. The selection criteria themselves may be specified using general operations. In this case, the act action may be used to test out new operations and generically test the outcome with Kno-specified selection criteria. We have applied this notion to our Zetalisp prototype to create a generic operation-testing mechanism that can be used by all Knos.

3.4 Limb Actions

We wish to model situations in which knowledge about a concept is distributed throughout several environments. Kno movement operations allow a Kno to explore various environments and gather information. An alternative approach is for a Kno to dispatch representatives throughout the universe of environments and gather information concurrently. We have provided the following set of actions to accomplish this task: *grow*, *kill*, *ship*, *teach*, and *unteach*.

The *grow* action generates a limb Kno. A limb Kno is similar to other Knos, but there are two important differences. First, if it dies, a message is automatically sent to the Kno that grew it. Second, the head Kno can control it to a certain extent (this is the only case in which a Kno can force an action on another Kno). The remaining Kno limb actions are those used by a head Kno to control its limbs. The *kill* action kills a limb. *Ship* is used to force a limb to move to a new context, and *teach* and *unteach* allow the head to modify the operation list of the limb.

4. EXAMPLE APPLICATION

The following application has been implemented in the Kno Zetalisp package. We omit the details of this implementation and present instead an outline to illustrate the way in which Knos can be used.

We consider a part of an application that models stock market activity. Each object manager represents a different stock market and contains information about the local stock prices and accounts among buyers and brokers. In this example we define two Kno classes, buyer-kno and broker-kno:

```
(kno-def buyer-kno
  (state worth portfolio broker)
  (basic-kno))

(kno-def broker-kno
  (state worth clients credit open-fee interest commission)
  (basic-kno))
```

Both buyer-kno and broker-kno inherit from the basic-kno class. Each has an instance variable that records its internal state (e.g., states of buyer are 'looking-for-brokers' and 'investing'). Buyers have instance variables that record their net worth, a list of stocks owned, and the name of their broker. Broker instance variables are used for a client list and for charging information. Some of the operations supported by these Kno classes are

```
(kno-op (broker-kno :accept-client) (kno-name)
; accept a new buyer as a client
)

(kno-op (broker-kno :buy-stock) (kno-name amt)
; buy a stock for a client
)
```

For example, broker Knos can perform operations called :accept-client and :buy-stock. The :accept-client operation takes two parameters: the name of the client (a buyer Kno) and the fee being paid by the client. The parameters to :buy-stock are the name of a buyer Kno and the name and amount of a stock.

Some of the rules used in this application are

```
(rule find-broker-rule
  (trigger(eq (kno-get 'state) 'looking-for-broker))
  (action
  ; see if any brokers on blackboard
  ; if so register as a client
))

(rule buy-stock-rule
  (trigger (eq (kno-get 'state) 'investing))
  (action
  ; choose a stock and buy it
))

(rule find-client-rule
  (trigger (eq (kno-get 'state) 'looking-for-clients))
  (action
  ; put an advertisement on the blackboard
))
```

The find-broker-rule and buy-stock-rule are used by buyer Knos, and the find-client-rule is used by broker Knos. For example, when a broker Kno is created, it is in the state 'looking-for-business'. When this Kno is allowed to execute, it will fire its find-client-rule. This exports a message containing the name of the broker to the blackboard. At some later time a buyer Kno will inspect the

blackboard, find the name of the broker, and send it an accept-client message. Once the buyer and broker know each other, the broker can start serving the buyer's :buy-stock requests.

Several interesting additions to this framework follow:

—Buyers can monitor their portfolios and try new brokers if their assets are not improving.

—Buyers can also move to new markets (object manager contexts) or send representatives (i.e., limbs) to these markets.

—One can add advisor Knos that sell buyer Knos information. An advisor Kno would advertise its presence over the blackboard. When contacted by a buyer (sending the appropriate fee), the advisor returns a rule that replaces the buyer's initial buy-stock operation (or just the buy-stock-rule). This leads naturally to the problem of learning operations. For instance, one can add metarules that monitor the performance of a buyer Kno's choices and decide when to seek a new strategy for buying stock.

—One may wish to unlearn strategies. If a broker has been applying a strategy for buying oil stocks that, for example, depend on a particular political situation, then a change in that situation (such as the sudden departure of an influential oil minister) could invalidate that strategy.

5. A COMPUTATIONAL MODEL FOR KNOS

The Zetalisp implementation of a Kno application provided a convenient test bed for our ideas. A number of fundamental concepts, such as concurrency, were missing from this environment, however. In this section we show how Knos can be readily mapped to a concurrent object-oriented environment in which objects are the active entities.

Hybrid is an object-oriented programming language that we are currently developing [11, 12]; it attempts to unify a number of object-oriented concepts into a single framework that incorporates concurrency and distributed environments. In particular, the following object-oriented ideas are fundamental to Hybrid:

—data abstraction with instantiation,

—multiple inheritance,

—aggregation,

—dynamic (i.e., run-time) object binding,

—active objects,

—persistence,

—message-passing,

—distributed environments.

Objects in Hybrid are like objects in Smalltalk or other object-oriented systems to the extent that they have an interface, which is a collection of methods (i.e., operations or procedures), each of which may be invoked, and a hidden representation that encodes the state of the object. Objects are classified into types according to their interface. There may be any number of instances of an

object type. Object types are objects as well, thus providing a mechanism for introducing new types within the framework. Furthermore, the instance variables of an object's state are also objects and thus provide a natural object-structuring mechanism (i.e., aggregation). Multiple inheritance [3] is available so that new types may inherit methods from multiple supertypes.

Hybrid differs from Smalltalk, and most other object-oriented systems, in that objects are *active*. In fact, the only active entities are objects, since there is no notion of "files," "programs," or "processes." Passive objects are simply active objects that only do something when they are told to do so.

To understand how much concurrency this gives us, suppose that at a "lower level" we have actors [2] (i.e., message-passing processes) and passive data objects. We may then map a "top-level" Hybrid object, together with its instance variables, to one (or more) actors. We have, then, exactly as much concurrency as there are actors. Furthermore, each actor provides a granularity of mutual exclusion for the objects "on top of" it. A top-level object may be "compiled into" actors that handle the messages for that object and (recursively) all its subobjects.

Truly active objects are those that respond to events through a triggering mechanism. There are many kinds of triggering events, including user input events, clock ticks, object updates, and anything that might cause a constraint to be violated.

Hybrid is designed with distributed environments in mind. A collection of Hybrid objects exists in a local environment under the supervision of a single "system object," which deals with scheduling, message passing, creation of new instances, and so on. The environment of a system object is a "virtual workstation." Objects may also pass messages to objects in other environments, if both objects are acquainted. Although objects may use location information explicitly to move themselves to another environment, for example, it is convenient for the system objects to provide some distributed functionality by transparently handling message passing and transactions across environments.

The notion of object persistence means that we can do away with files. Message passing becomes the *only* paradigm for communication. The system object is responsible for ensuring that objects are reliably saved in stable storage.

We now show how the concept of Knos maps to a Hybrid environment.

5.1 Contexts and Object Managers

A Kno context is illustrated in Figure 2. Each Kno context is mapped to a specialized Hybrid environment. The *object manager* (OM) is a Hybrid object that is the basic context manager. It handles all blackboard transactions. Some of the work that is done by the object manager in the Zetalisp implementation is subsumed by the Hybrid system object. (In particular, object scheduling is handled by the system.) All Knos are acquainted with the object manager.

Agents are also Hybrid objects that communicate with the object manager and the outside world. When a new context is created, it may be initialized with agents that are aware of other contexts. At the very least, each context is created with an agent that knows the address of a *name server*. It, in turn, knows the addresses of other contexts and may allow one context to become acquainted with others. Knos are a complex of Hybrid objects and are described next.

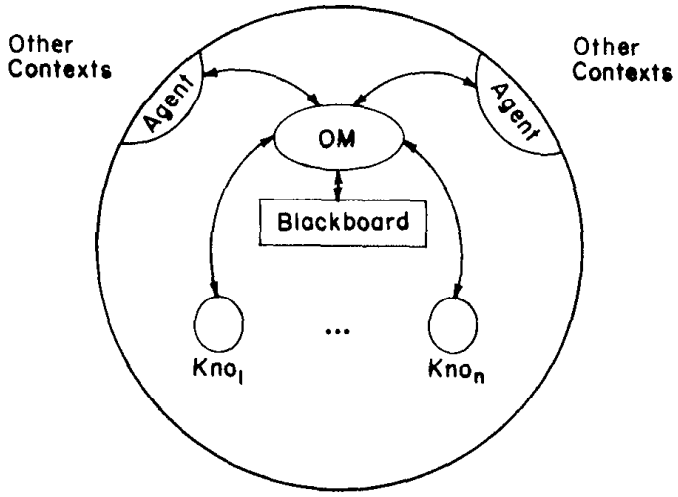

Fig. 2. Hybrid model for contexts and object managers.

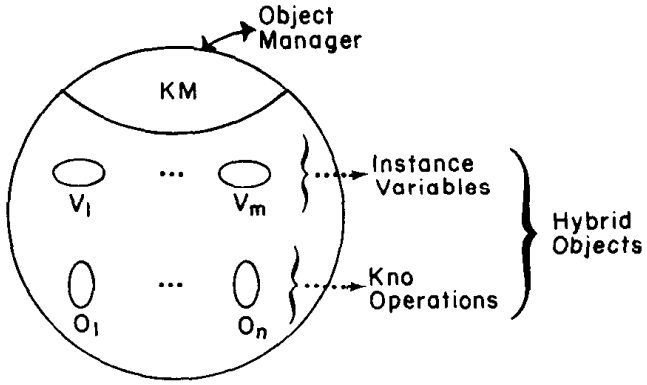

Fig. 3. Hybrid model for Knos.

5.2 Kno Representation

A Hybrid model for a given Kno is illustrated in Figure 3. Each Kno is realized by several objects: a *Kno manager* (KM), an object for each instance variable (publicly addressable by all operations), and an object for each operation. A Kno manager synchronously asks the object manager pertinent messages and dispatches operations accordingly. It decides when the operations it manages are to be triggered. The default Kno manager enforces monitor-like semantics on internal operations in that it ensures that, at most, one operation is active at a time within its domain. Programmers may wish to override this default to enhance or restrict concurrency further by providing their own Kno manager objects. At some point, we may provide a choice of Kno managers from a library.

5.3 Moving and Learning

Under the model illustrated above, moving and learning actions can be captured in a straightforward way. Assuming a Kno manager has chosen to accept a new operation, the manager simply unfreezes the operation by creating a new

operation instance from the imported value, updating its list of valid operations, and making the operation executable within its domain. Hybrid's dynamic binding enables the Kno manager to deal with operations of a previously unknown (object) type, as well. Moving a Kno to a new context is an analogous operation that requires the cooperation of object managers in the source and destination contexts.

6. CONCLUDING REMARKS

In this paper we have outlined the facilities for a powerful object-oriented environment. Knos (*KN*owledge acquisition, dissemination, and manipulation *O*bjects) are active, nomadic, and adaptive objects and can be used to model many situations and provide general applications support tools. An example Kno implementation was outlined using Zetalisp, and a basic skeleton of the Kno environment was represented as Hybrid objects.

An environment based on Knos can be useful for sophisticated application support tools that take over user operations. We are currently investigating the properties of useful Kno species. That is, families of prepackaged Knos that can perform useful jobs in an office information systems context. Knos provide behavior that is rather different from an environment based on automatic procedures and passive messages and data. Office tools based on active agents like Knos are extremely important. We are severely limiting the scope of office information systems if we only consider mechanization of office work or provide procedures for repetitive office work. Office tools should provide the option of performing more difficult tasks, such as information acquisition, negotiation, cooperation, or apprenticeship.

The following outlines other interesting operations that can be provided by Knos:

(1) A Kno can consider learning, even in a potentially hostile environment, by testing out possible new operations on a representative of itself (i.e., a limb) that it knows exists in a safe environment. The result of the operation may be evaluated in the safe context, and the choice as to whether or not to accept the operation in the other context can be a more informed one.
(2) A Kno can "do recursion" by leaving messages for itself on the blackboard.
(3) A complex Kno can cut off or multiply its limbs that may reside in foreign contexts. In this way, it can react appropriately to hostile or benevolent environments.
(4) A Kno can export its whole body, or a part of it, to another "doctor" Kno. Later on, it can reimport all its body or the exported part. In this way, the Kno can let the "doctor" Kno "fix" some of its rules or data structures, do garbage collection, optimize operations, and so forth.

The environment provided by Knos is somewhat strange by programming language standards. It is not, however, strange in terms of user experiences, for, if the goal of an object-oriented system is to allow one to structure a system in direct correspondence with reality, then surely Knos bring us closer to that goal. Programming problems may change considerably when looked at in a Kno

environment. What is lacking at present is a methodology for programming with concurrency, migration, and dynamic operation acquisition.

The notion of *security* in most systems and programming languages is based on "walls" of security protecting passive objects (e.g., files, records, messages [7, 10]). Knos are active, so they can defend themselves. Not only can a Kno fend off attempted intrusions, but it can erase itself if threatened. A Kno can even provide "disinformation" by giving information that differs from what it is carrying.

The notion of *consistency* is absent in a Kno population. A Kno needs to be consistent with itself. It does not need to be consistent with other Knos. As a matter of fact, the inconsistency possible among Knos is extremely interesting. Each Kno can carry a different opinion or belief. The problem of consistency gives way to a problem of *reconciliation*. Suppose many Knos presenting different opinions appear within the context of an object manager. We need to study ways of exchanging information among them to provide a consensus rather than enforcing consistency.

Resource management does not appear at the Kno level. Knos live in an environment of seemingly unlimited resources. On the other hand, we cannot escape the fact that if useless Knos proliferate, they can adversely affect the operations of useful Knos. The problem of resource management is translated into a problem of "pest" control. It is not clear, however, how this can be realized without violating the notion of Kno autonomy.

The evolution of Knos de-emphasizes static class structure and inheritance. Traditional static class inheritance becomes less important if Knos can change their operations and data structures during their lifetimes. Class inheritance should provide only generic, prototypical behavior. The rest, which is, in effect, dynamic inheritance and class evolution, can be "learned" by Knos. The problem of class inheritance gives way to the problem of *adept learning*, that is, providing Knos with the capability of separating useless or dangerous new operations from those that are useful. What is required is the development of a programming methodology to provide guidelines for the disciplined use of the powerful features of Knos.

The behavior of Knos is not always easy to understand. Formal tools and a rigorous semantics will help (and are planned), but alternative ways of understanding object behavior are also required. For that reason, we are working on a facility to allow the behavior of Knos to be illustrated using computer animation tools. This facility is interesting in itself, since it includes a temporal scripting language for animated objects (that are themselves Knos) and an interactive environment for defining graphic objects and specifying their motion [4]. Expressions of the scripting language are directly executable, and thus the execution of Kno operations can be used to generate animation expressions dynamically. Moreover, the tools can be used for other purposes. The environment for specifying graphic objects and their motion can be used to create libraries of prepackaged animation. They can then be instantiated and coordinated using the scripting language to create complex multimedia Kno objects, which can be transmitted to other contexts and/or stored on compact disks. Last, the temporal scripting language can be used not only to generate behaviors satisfying a

temporal specification, but also to specify the desired coordination of object activities.

ACKNOWLEDGMENTS

The research described in this paper is partially supported by an FNRS grant from the Swiss Federal Government.

REFERENCES

1. AHLSEN, M., BJORNERSTEDT, A., BRITTS, S., HULTEN, C., AND SODERLUND, L. An architecture for object management in OIS. *ACM Trans. Off. Inf. Syst. 2*, 3 (July 1984), 173–196.
2. BYRD, R. J., SMITH, S. E., AND DE JONG, S. P. An actor-based programming system. In *Proceedings of SIGOA Conference on Office Information Systems* (Philadelphia, June 21–23). ACM, New York, 1982, pp. 67–78.
3. CURRY, G., BAER, L., LIPKIE, D., AND LEE, B. TRAITS: An approach to multiple inheritance subclassing. In *Proceedings of SIGOA Conference on Office Information Systems* (Philadelphia, June 21–23). ACM, New York, 1982, pp. 1–9.
4. FIUME E., TSICHRITZIS, D., AND DAMI, L. A temporal scripting language for object-oriented animation. Submitted for publication.
5. GOLDBERG, A., AND ROBSON, D. *Smalltalk 80: The Language and its Implementation.* Addison-Wesley, Reading, Mass. 1983.
6. GUTTAG, J. Abstract data types and the development of data structures. *Commun. ACM 20*, 6 (June 1977), 396–404.
7. HOARE, C. A. R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549–557.
8. HOGG, J. Intelligent message systems. In *Office Automation: Concepts and Tools*, D. C. Tsichritzis, Ed. Springer-Verlag, Heidelberg, 1985, pp. 113–134.
9. HOGG, J., NIERSTRASZ, O. M., AND TSICHRITZIS, D. Office procedures. In *Office Automation: Concepts and Tools*, D. C. Tsichritzis, Ed. Springer-Verlag, Heidelberg, 1985, pp. 137–166.
10. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 381–404.
11. NIERSTRASZ, O. M. HYBRID: A unified object-oriented system. *IEEE Database Engineering* (Dec. 1985), 49–57.
12. NIERSTRASZ, O. M., AND TSICHRITZIS, D. C. An object-oriented environment for OIS applications. In *Proceedings of Conference on Very Large Data Bases* (Stockholm, Aug.), Morgan Kaufmann Publishers, Inc., Mountain View, Calif., 1985, pp. 335–345.
13. RICH, C., AND SHROBE, H. E. Initial report on a LISP programmer's apprentice. *IEEE Trans. Softw. Eng. SE-4*, 6 (1978), 456–467.
14. SHOCH, J., AND HUPP, J. The worm programs—Early experience with a distributed computation. *Commun. ACM 25*, 3 (Mar. 1982), 172–180.
15. TSICHRITZIS, D. C. Objectworld. In *Office Automation: Concepts and Tools*, D. C. Tsichritzis, Ed. Springer-Verlag, Heidelberg, 1985, pp. 379–398.
16. VITTAL, J. Active message processing: Messages as messengers. In *Proceedings of The International Symposium on Computer Message Systems, IFIP TC-6* (Ottawa, Ontario, April, 1981). North-Holland, Amsterdam, 1982, pp. 175–195.
17. WEINREB, D., AND MOON, D. *The Lisp Machine Manual.* Symbolics Inc., 1981.
18. ZISMAN, M. Representation, specification and automation of office procedures. Ph.D. dissertation, Wharton School, Univ. of Pennsylvania, Philadelphia, Pa., 1977.
19. ZLOOF, M. M. QBE/OBE: A language for office and business automation. *IEEE Comput. 14* (May 1981), 13–22.

Received June 1986; revised September 1986; accepted September 1986