Description/Introduction

In many knowledge-based systems, it is useful to represent knowledge as interconnected sets of instances. A virtual copy mechanism allows a network of instances to be viewed as a prototype which can be copied. The copy of the prototype is virtual in that the contents of each instance is not completely copied at creation time. Instead, it inherits default values from the prototype (also called the original), thus continuing to share the parts not modified in the copy. The copied network is virtual also in the sense that only those instances needed in the processing are copied.

A virtual copy of an object in the prototype network has the following properties:

- It responds to at least the same set of messages as the prototype object and in the same way; that is, a copy has the same procedural behavior that is defined for the prototype.
- A copy inherits variables and their values from the prototype, and continues to do so until an explicit change is made in the copy. At that point, the new value is stored in the copy and it stops tracking the prototype for that variable. A fetch operation on a value that is not stored locally either finds or creates a virtual copy of the value obtained from the prototype.

Installation/Loading Instructions

The implementation of virtual copies is contained in the file LOOPSVCOPY.LCOM. No other files are necessary.

Application /Module Functionality

A network of instances is tied together through the values of instance variables within each of the instances. Assume an object A has an instance variable x, the value of which is the object B. A virtual copy of A will also have an instance variable named x. The value of x in the copy will point to B if B is a shared object, or x may point to a copy of B if it is to be virtual. Changing the value of x in the copy will not change the value in the original.

Overview of Operation

By default, virtual copies share instance variables. This means that changing the value of an instance variable in the original will be tracked by the copy.

Virtual copies are implemented with two additional classes:

- **VirtualCopyMixin**

  The class **VirtualCopyMixin** is a subclass of Tofu which contains two instance variables:

- **% copyMap%**
- **% copyOf%**

(These unusual names are used to avoid conflicts with any other instance variable names users may create.) This class contains several methods, most of which are required to implement virtual copies and are not used by a programmer.

Printing a virtual copy instance is a specialization of how regular instances are printed. All instances print as #,($& <class-name> UID). The class of a virtual copy is a dynamic mixin of the class **VirtualCopyMixin** and the class of the original object (see the *Xerox LOOPS Reference Manual* for more information on mixins). The virtual copy print function adds the name or unique identifier (UID) of the original object. For example,

#,($& (VirtualCopyMixin Container1) (JFW0.0X:.aF4.R>8 . 3) c1)

is a copy of the object named **c1**.

- **VirtualCopyContext**

  The class **VirtualCopyContext** has no methods and only one instance variable, **copyMap**. Instances are used as an argument for calls to **MakeVirtualMixin**.

Since copies can be made of copies, you often need to determine the original object of a chain of copies with the **UltimateOriginal** function.

Operands

This section describes the functions, methods, class variables, and instance variables that operate on virtual copies.

VirtualIVs [Class Variable]

Purpose/Behavior: Helps specify a class whose instances may be made into virtual copies. The value of this class variable should be either the symbol ALL, or a list of instance variables contained within instances of the class. If the value is ALL, all objects pointed to by any of the instance variables will be copied. If the value is a list of instance variables, only the instance variables on this list will have their values copied. Other instance variable values will be shared between the copy and the original.

(**MakeVirtualMixin** *x copyContextObj*) [Function]

Purpose: Creates a virtual copy of an object.

Behavior: *C*reates a dynamic mixin class combining the classes **VirtualCopyMixin** and the class of *x*. An instance of this resulting class is created and it is returned.

Arguments:

x — An object to be copied*;* must have the class variable **VirtualIVs** as described above.

copyContextObj — Usually NIL; used internally by **MakeVirtualMixin** when it calls itself. It can be an instance of **VirtualCopyContext** if you are creating an instance that is intended to be part of a currently existing network of copies starting from another entry point. See description in **Limitations** below for a further explanation of this point.

Returns: An object that is a copy of *x*.

Example: Refer to the section, "Example."

% copyMap% [Instance Variable of VirtualCopyMixin]

Purpose/Behavior: A mapping of original nodes (which are objects) in a network to the copied nodes. This map is stored in an instance of the class **VirtualCopyContext**.

% copyOf% [Instance Variable of VirtualCopyMixin]

Purpose/Behavior: Within an instance that is a copy, the value of this instance variable is a pointer to the object that was copied.

(← self **VirtualCopy?**) [Method of VirtualCopyMixin]

Purpose: Determines if an object is a virtual copy.

Returns: *self*

Categories: Object, VirtualCopyMixin

copyMap [Instance Variable of VirtualCopyContext]

Purpose/Behavior: The value of this instance variable is a list of dotted pairs. The CAR of each pair is the original; the CDR, the copy.

(**UltimateOriginal** *self*) [Function]

Purpose: Determines what an object is ultimately copying.

Behavior: If *self* is not a virtual copy, *self* is returned.

If *self* is a virtual copy, this recurses through the value of the instance variable **% copyOf%** until it finds the original and returns it.

Arguments: *self* A Xerox LOOPS object.

Returns: *self* or what is at the top of *self*'s copy chain.

Example

Create a class called **test** and edit it as shown.

```
44_(_($ Class) New 'test)
#,($C test)

45_(ED 'test)
```

```
SEdit #,($C test) Package: INTERLISP
((MetaClass Class Edited%:                ;Edited 11-Dec-87
                                          ;16:50 by
 )
 (Supers Object)
 (ClassVariables (VirtualIVs (atomCopy listCopy objCopy)))
 (InstanceVariables (atom NIL)
       (atomCopy NIL)
       (list NIL)
       (listCopy NIL)
       (obj NIL)
       (objCopy NIL))
 (MethodFns))
```

Create an instance called **t0** of this class and inspect it.

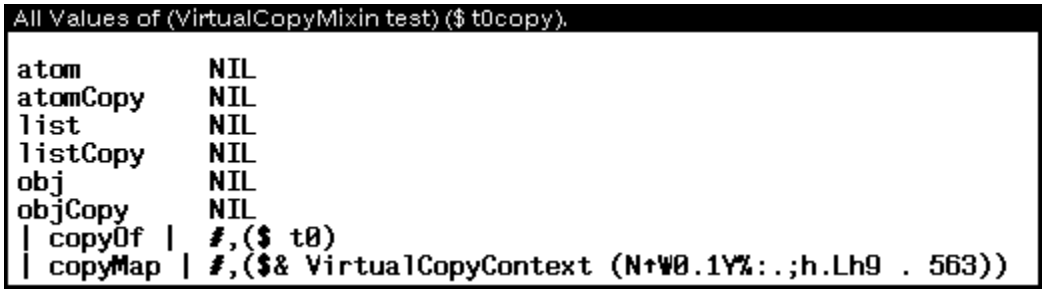
```
46_(_($ test) New 't0)
#,($& test (N↑W0.1Y%:.;h.Lh9 . 556))


47_(_ ($ test)
      NewWithValues
      (BQUOTE ((atom 1)
                  (atomCopy 2)
                  (list (a b c))
                  (listCopy (A B (\, (_ ($ test) New (QUOTE t1)))))
                  (obj (\, (_ ($ test) New (QUOTE t2))))
                  (objCopy (\, (_ ($ test) New (QUOTE t3)))))))
#,($& test (N↑W0.1Y%:.;h.Lh9 . 560))

48_(_ IT SetName 't0)
#,($& test (N↑W0.1Y%:.;h.Lh9 . 560))

49_(INSPECT IT]
{WINDOW}#52,51234
```

```
All Values of test ($ t0).
atom     1
atomCopy 2
list     (a b c)
listCopy (A B #,($ t1))
obj      #,($ t2)
objCopy  #,($ t3)
```

Make a copy called **t0copy** and inspect it.

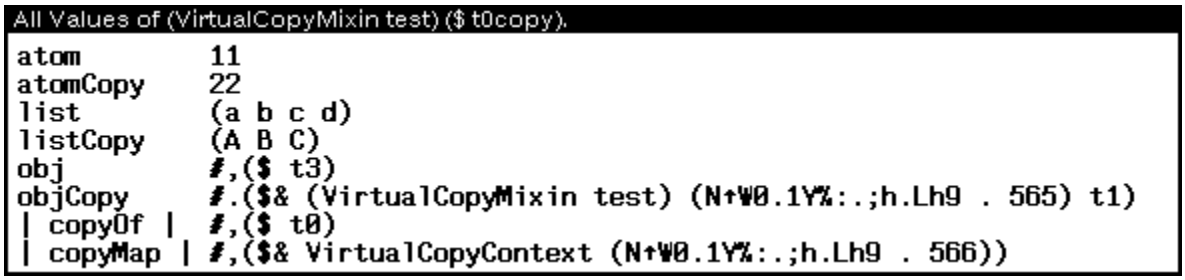
```
57_(_ (MakeVirtualMixin ($ t0))
   SetName
   (QUOTE t0copy))
#,($& (VirtualCopyMixin test) N↑W0.1Y%:.;h.Lh9 . 562)
```

```
58_(INSPECT IT)
{WINDOW}#53,10150
```



Make the following changes to **t0** and then reinspect **t0copy**.

```
60_(for iv in '(atom atomCopy list listCopy obj objCopy)
as val in (LIST 11 22 '(a b c d) '(A B C) ($ t3) ($ t1))
do (PutValue ($ t0) iv val]
NIL

61_ (INSPECT IT)
{WINDOW}#53,10152
```



The copied instance variables have not changed since they do not track changes in the original object.

Limitations

Some subtle issues are involved in building and using prototype structures so that the structure is preserved in the copied network. These involve how the network is typically traversed.

A general constraint is that all the links to any shared node in the prototype either all be marked as virtual variables, or none of them are. If they are all marked, then a single copy will be made and used. If none are, then the original object from the prototype will be used. Sharing with the prototype can be useful if this object is a repository for standard information that is independent of context. However, if this constraint is violated, the topology of the virtual copy will be different from that of the prototype.

In the simplest situation the network has a single entry node. In this case, a copy-map (see the section "Operands") can be created when the entry node object is first copied. After that all values are copied using this copy-map. The mechanism works well in this situation, even if there is sharing and there are cycles within the network.

At the other extreme, networks can have arbitrary connectivity, including multiple entries from outside the network, for example, from other networks or

non-objects. In this case, the following constraints are necessary to ensure correctness of the virtual copy mechanism.

The first constraint states that all access to the network must start through a copy of one of the nodes in the prototype. This condition is necessary because the criteria for copying are contained in the links from one object to another, not in the objects themselves, and a shared node could not specify a link to a node to be copied. This constraint ensures that all accesses from the outside will be copied if and only if that object would have been copied because of an internal link. Otherwise, an analogous situation would occur in which you could either reach a copy or the original node of the prototype itself depending upon which path you follow when the paths lead to the same node in the prototype.

The final constraint requires that all entries to the network should be passed the same copy-map if they are to share structure. The underlying concern in imposing these constraints is that a network be always copied the same way to maintain its topology regardless of where you start.

Suppose you want to make a virtual copy of a virtual copy, that is, to use a virtual copy of a network as a prototype itself. This is very useful if you are using a network to hold the state of a partial design and you want to try two alternative continuations of the design. Some hidden costs are associated with such multiple-level virtual copies.

Suppose further that a network N1 is used as a prototype and you make a virtual copy, N1-VC. Furthermore, N1-VC-VC is defined to be a copy of N1-VC. Values missing from N1-VC-VC are found in the corresponding object of N1-VC. If the value is missing there, the process recurs, and N1 is examined. If the value is to be a virtual copy, then this process will add a virtual copy in N1-VC, and then a second level copy in N1-VC-VC. This is necessary to preserve the semantics presented, but implies that many levels of virtual copy cannot easily do inexpensive incremental searches of a network.

References

Mittal, S. , Bobrow, D. G., and Kahn, K. *Virtual Copies, Between Classes and Instances*. ACM OOPLSA-86 Conference Proceedings, Portland, Oregon, 1986.