LA-UR -90-3541

CONF-9010211 - 2

LA-UR--90-3541

DE91 001957

TITLE A CLOS IMPLEMENTATION ON TOP OF KEE

AUTHOR(S) HARRY W. EGDORF, A-7

SUBMITTED TO AI IN DOE CONFERENCE, OCTOBER 1990

Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

A CLOS Implementation on top of KEE

by

H. W. Egdorf
Los Alamos National Laboratory
October 11, 1990

Abstract

IntelliCorp's Knowledge Engineering Environment (KEE)[1] is a popular AI shell running on top of Common Lisp. KEE implements a message-passing style of object-oriented programming. Common Lisp has adopted the different style of the Common Lisp Object System (CLOS) as the standard object-oriented programming style.

This paper describes an implementation of a subset of CLOS on top of KEE that uses the CLOS-style of programming to construct and manipulate KEE objects. The subset has been used to construct a moderate-sized discrete-event simulation of some facilities at the Rocky Flats Plant in Colorado. Experiences constructing this model will be discussed.

Several reasons are examined for use of such a system. Programmers may have to maintain and extend an existing KEE application, but wish to do so using the newer CLOS style of programming while staying within the KEE environment. A requirement may exist for support of a mixed CLOS/KEE system where CLOS is not available. Programmers may wish to begin developing in CLOS with a view toward migration from KEE when CLOS becomes more available.

1. KEE Strengths and Weaknesses

IntelliCorp's Knowledge Engineering Environment (KEE) is one of the more full-featured and long-lived AI shells available commercially. It has been the basis for many important projects at Los Alamos in the area (amongst others) of large discrete-event simulations.

KEE provides a frame-based knowledge representation facility that allows many different styles of use and many different features. It provides an excellent rapid prototyping environment with a mouse-and-menu style of interface and good graphics support. The object-oriented programming paradigm provided is built upon the frame system. This object system is a message-passing system in the style of Smalltalk or the original Flavors system.

KEE is not, however, without problems. The system is beginning to be phased out by IntelliCorp in favor of a new series of products. While the end-of-life for KEE has not yet been announced, the latest versions are available on a more limited number of platforms than previously and IntelliCorp is providing a number of migration aids to its new Kappa series of products.

It has been determined that for some projects it is more appropriate to use a source-code based programmatic interface to KEE rather than the more traditional rapid-

[1] KEE is a trademark of IntelliCorp, Mountain View, CA.

prototyping interface emphasized by the IntelliCorp documentation and training. The use of this programmatic interface leads to some implementation difficulties. KEE provides a single level of programming interface to the features of the system. This interface is at a uniform low level. No higher-level toolkits or programming abstractions are provided upon this low-level programming interface.

2. Programming Toolkit Requirements

The issues recognized above contribute to the requirements driving the design of a high-level programming toolkit for use with the object-oriented features of KEE. The users of such a toolkit are the designers and programmers who wish to build systems upon the KEE package. Those users have certain requirements that arise from the intended use of the KEE programming environment.

The user-level requirements fall roughly into the following categories.

(1) Programmatic requirements dictate the use of KEE for development, but there is a desire to migrate from the proprietary package to the new Common Lisp Object System (CLOS) in the future. This migration should take the form of an evolution of the design rather than a revolution requiring a complete re-design and implementation.

(2) An existing KEE system needs to be maintained and extended, but the programmers wish to use the more modern CLOS programming style rather than the older message-passing style of KEE along with a more directed object creation system than the (sometimes too general) low-level unit creation facilities of KEE.

(3) The designers of a system wish to use CLOS for object-oriented facilities and KEE for other knowledge-representational facilities, but need a bridge for development until solid CLOS implementations are available along with KEE.

(4) Programmers familiar with KEE wish to have a system for learning CLOS while maintaining their familiar environment.

The toolkit described here addresses these issues. It provides one high-level object-oriented abstraction for use by programmers within KEE while in no way limiting any other use of KEE by programmers or users.

3. Programming Toolkit Design

The design of this toolkit arose from three observations.

(1) The work and experience that has gone into development of CLOS has been extensive and productive in gathering a most complete and up-to-date view of those facilities that should be in an object-oriented programming system. The toolkit should provide a programming interface compliant with CLOS.

(2) The object-oriented programming facilities provided by KEE, particularly the method inheritance and method combination mechanisms, are very similar to those provided by CLOS.

(3) The general frame mechanism upon which KEE's object system is based combined with the single fully-general procedure provided for creation of such frames leads to a system where programmers can easily violate principles of object-

oriented design by inappropriate use of the full generality of the frame system.

The design of this toolkit primarily provides a CLOS-compatible syntax for object-oriented programming operations while allowing KEE to provide all functionality involved with method combination and inheritance. The decision to utilize KEE's mechanisms in these areas rather than develop them independently leads to the major limitations on just how much of CLOS can be implemented within this toolkit. These limitations reflect those places where KEE's object-oriented programming system is incompatible with CLOS. The rational for this decision is that if a more complete implementation independent of KEE is desired then use should be made of either a vendor's CLOS implementation or a public CLOS package such as PCL from Xerox.

The rest of this section describes the details of the toolkit's design. Familiarity with both KEE and CLOS is assumed.

Meta-Objects

A small part of the proposed Meta-Object protocol is implemented. Meta-Objects are implemented in a KEE knowledge base called *CLOS-On-KEE*. Two Meta-Objects are implemented.

Standard-object is a superclass of every class except itself. Any class created by the user with no defined superclasses is made a direct subclass of *standard-object*. A primary method for the generic function *initialize-instance* is inherited from this class by every instance created in the system. *Initialize-instance* may be modified to provide specific initialization at object creation time.

Every class created by the system is not only a subclass of its superclasses, allowing the normal inheritance of slots and the other functions normally associated with classes in an object-oriented system, but is also an *instance* of some class. The purpose for this parent (known as the metaclass) is to allow capabilities to be associated with the class rather than just the instances of the class. Every class is an instance of the metaclass *standard-class*. Subclasses of *standard-class* may be defined and specified as the metaclass of a class.

Primary methods for the generic functions *make-instance* and *allocate-instance* are inherited by every class from *standard-class* but none of the rest of the instance creation structure defined in the CLOS specification is implemented.

Class Creation

Classes are implemented as KEE units with particular parent links.

Defclass creates a class. The class is implemented as a KEE unit with class-parent links to the superclasses specified in the *defclass* form (or to *standard-object* if none are specified) and a single member-parent link to the specified metaclass (or *standard-class* if none is specified). Slots defined by the *defclass* form are implemented as member slots in the KEE unit.

Some CLOS defclass slot options are not supported at all or are only partially supported.

- *:allocation :class* is not supported due to there being no KEE inheritance mechanism that maps to this particular type of storage.
- *:initargs* is not implemented. This lack does not reflect any fundamental limitation of KEE, rather no mechanism for storage and use of the initargs has yet been added to the system.
- The functions defined by *:reader :writer,* and *:accessor* are implemented as defuns rather than generic functions. The reason for this is that these functions map into KEE *put.value* and *get.value* calls and it was believed that the overhead of method invocation would be unacceptable. The main limitation is that method modification cannot be performed on the accessor functions. If this is desired, two facilities can be used. The designer may use KEE active values for the desired behavior modification, or a separate generic function can be defined that performs the slot access and that can be wrapped.

Slot-value and its associated setf method are implemented to allow the programmer the ability to define true (wrapable) generic functions as slot accessors.

One CLOS defclass class option is not implemented.

- The *:default-initargs* class option is not implemented for the same reasons described above regarding the *:initargs* slot option.

Instance Creation

The generic function *make-instance* is provided by the metaclass *standard-class* and is used to create an instance of a class.

This generic function may be modified in the normal way with wrappers, but does not implement the detailed behavior of the CLOS specification.

Generic Function and Method Creation

Generic functions are created by *defgeneric.* These generic functions are defuns that simply perform an appropriate KEE *unitmsg* call. Generic functions are very simple due to the use of KEE for all method combination.

Unlike CLOS, this package will not automatically create the generic function if it does not exist at method definition time. The use of *defgeneric* is not required if the programmer wishes to use *unitmsg* calls directly. However, experience with this package indicates that the use of a functional style rather than use of *unitmsg* is preferred.

Only the following defgeneric options are allowed.

- *:documentation* provides the appropriate documentation string to the generic function.
- *:method* may be used to include methods.

Methods of generic functions are defined with *defmethod.* Methods simply add Lisp forms to method slots of a KEE unit representing the class upon which the first method argument is specialized. This leads to the following limitations.

- Only the first parameter may be specialized. This reflects the KEE notion that methods are attached to classes. Multiple specialized parameters can be implemented within KEE as a class that inherits from the proper classes to specialize all the desired parameters, with the method specializing upon this new class. Experience with both KEE and CLOS indicates that the CLOS multiple-parameter specialization is a nicer structure within which to work.

- *Eql* specializers are not supported. This limitation reflects the fact that KEE does not support all Lisp objects. It would not be difficult to add method modification to specific KEE instances, but the common use of *eql* specializers is to specialize on symbols or numbers rather than just instances of CLOS user-defined classes.
- *(setf symbol)* methods are not supported. This reflects both the fact that accessors on slots are implemented as defuns rather than generic functions, and the fact that the *(setf symbol)* notation is not yet a part of delivered Common Lisp Implementations.
- *Call-next-method* may not pass modified arguments to the next method, may not have a return value, and may only be used in *:around* methods rather than in primary methods as allowed by CLOS. These limitations reflect the implementation of *call-next-method* by use of the KEE *wrapperbody* form.

4. Experience

Despite the limitations of this CLOS subset, (with the lack of *:initargs* being the most severe,) the experience with the system has been positive.

The system succeeds in providing a programming toolkit that appropriately limits the full use of the KEE frame system to a subset of features needed just for object-oriented programming. This limitation of features is accomplished without either losing the features needed to develop object-oriented programs within KEE or limiting in any way other use of KEE from outside the package.

Code developed within this package is easily moved to a true CLOS implementation. While this has only been done for simple test cases, experience indicates that use of this package should provide an evolutionary bridge to a CLOS environment.

A major disadvantage of this package is that very little of the CLOS specification regarding class and instance modification and redefinition have been implemented. In particular, a change to a method cannot be incrementally reloaded because there is no mechanism to determine just what portions of a KEE method the new changed code should replace. This is due to a difference in KEE's method combination vs. CLOS's method combination where KEE builds the final method at the time the method is defined rather than the time at which the method is invoked. This means that incremental changes to source code during development require reloading of all of the source code onto a clean environment. In practice, this has not proven to be a problem. It does, however, prevent use of some Lisp development environment's incremental compilation and loading features.

The greatest unexpected benefit of the use of this package has been in the reduction of code size due to both the simpler programming constructs and the use of accessor functions and their associated setf methods. KEE normally uses two reader and writer functions, *get.value* and *put.value,* for slot access. The ability to make use of all of the normal Common Lisp functions that operate with setf methods (as provided by the *:accessor* slot option) such as *push, pop, incf,* and *decf* has drastically improved the code of systems built with this package compared to the normal KEE slot accessors.

5. Conclusions

The use of the CLOS package with KEE will continue. The benefits of programmer productivity and evolution away from KEE far outweigh the limitations of having only a subset of CLOS (and the programmer training required to recognize that subset) and the loss of certain incremental development tools in the Lisp development environment.