OBJECT-ORIENTED KNOWLEDGE REPRESENTATION FOR EXPERT SYSTEMS

Stephen L. Scott

Hughes Information Technology Company
Washington Engineering Laboratory

Abstract: Object-oriented techniques have generated considerable interest in the AI community in recent years. This paper discusses an approach for representing expert system knowledge using classes, objects, and message passing. The implementation is in version 4.3 of NASA's CLIPS, an expert system tool that does not provide direct support for object-oriented design. The method uses programmer-imposed conventions and keywords to structure facts, and rules to provide object-oriented capabilities.

1. INTRODUCTION

A typical expert system consists of a rules, facts, and an inference engine. Although many types of problems can be addressed using this knowledge representation model, others may require features afforded by logical, network, or frame based models (Mylopoulos and Levesque, 83). Recent interest in object-oriented design has suggested an object-based view of knowledge representation, combining elements of several of these mechanisms (Leung and Wong, 90).

Object-oriented design elements, such as classes, objects, and messages, can be implemented in an expert system that does not directly support these capabilities by using facts containing object-oriented keywords and using rules to manipulate these facts. A fact with an object-oriented keyword will be referred to as a "structured fact". The use of facts that contain keywords in certain fields bears similarity to the use of an IS-A or A-KIND-OF link in a semantic network. The object-oriented keyword technique is used by (Assal and Meyer, 1990) in the implementation of a frame-based knowledge representation mechanism. Before proceeding further with the details of this approach, it may be useful to clarify the terminology, adapted from (Meyer, 88), that will be used in the remainder of this paper.

A *class* is a structure defined prior to run-time that identifies data and procedural characteristics of a program entity. Classes can be implemented in an expert system using a set of structured facts that describe features that characterize the class. An *object* is a run-time instance of a class. An object can be represented in an expert system by a structured fact that uniquely identifies the object. Additional structured facts can be used to store the current feature values of the object. A *feature* is a data characteristic associated with a class. Features can be of any data type, such as integer, real, character, or string. A *method* is a procedural characteristic associated with a class. In an expert system, methods can be implemented using rules. A *message* is a structured fact that contains either a request for the current value of an object's feature, or a request to initiate a method associated with an object. The use of messages is one of the distinguishing characteristics of the object-oriented methodology.

2. IMPLEMENTING CLASS AND OBJECT STRUCTURES

2.1 Class Definition Using Facts

A class is defined by asserting one or more facts of the form

(CLASS <class-name> <parent-class>)

This declaration indicates that <class-name> is a subclass of <parent-class>. By convention, in the first CLASS declaration for a class, the identifiers used for <class-name> and <parent-class> should be the same. This is required by the syntax of the rule used to instantiate objects from class declarations. Note that multiple inheritance is accomplished by allowing multiple class declarations, as in the following:

```
(CLASS      <class-name> <class-name>)
(CLASS      <class-name> <parent-class-1>)
(CLASS      <class-name> <parent-class-2>)
.
.
.
(CLASS      <class-name> <parent-class-N>)
```

A class feature is specified as such

```
(HAS-FEATURE   <class-name> <feature-name>
                            [default-feature-value(s)] )
```

where [default-feature-value(s)] consist of one or more data items that comprise the default values of this feature.

A class method is specified using a fact such as

(HAS-METHOD <class-name> <method-name>)

The actual implementation of the method is not specified here, nor is any parameter information given at this time.

The following example, employing the CLIPS deffacts fact structuring construct, illustrates a typical class declaration. The length and width fields have been assigned arbitrary default values.

```
(deffacts    rectangle-class
     (CLASS          rectangle    rectangle)
     (HAS-FEATURE    rectangle    length      2)
     (HAS-FEATURE    rectangle    width       5)
     (HAS-FEATURE    rectangle    area        10)
     (HAS-METHOD     rectangle    rectangle-area))
```

2.2 Object Instantiation Using Facts

An object is created by declaring the existence of the object and its features. The instantiation is accomplished by asserting a fact of the following form

(INSTANCE <object-id> <class-name>)

where <object-id> is a unique identifier associated with this object, and <class-name> is the name of a parent class.

The features of the object are automatically asserted with appropriate default values by a rule in the expert system that tests for the presence of a newly instantiated object. The rule asserts facts that make the connection between this particular object and its features. For each feature, a fact of the following form is asserted.

(HAS-FEATURE <object-id> <feature>
[default-value(s)])

Similarly, methods associated with an object can be declared by asserting facts such as:

(HAS-METHOD <object-id> <method-name>)

The following example, taken from the rectangle class presented earlier, illustrates a typical object instantiation.

```
(INSTANCE      a-box      rectangle)
(HAS-FEATURE   a-box      length       2)
(HAS-FEATURE   a-box      width        5)
(HAS-FEATURE   a-box      area         10)
(HAS-METHOD    a-box      rectangle-area)
```

3. MANAGING OBJECTS

3.1 Top Level Object Management Facts

While these conventions provide a way to structure classes and objects, it is also useful to provide some "system functions" to manage objects. Some typical operations might include getting and setting object feature values, and creating and destroying objects. To get an object's values, a fact of the following form is asserted.

(GET <object-id> <feature>)

An object's feature is set using a fact such as

(SET <object-id> <feature> <value>)

These facts are used to trigger the GET and SET rules (see 3.2) that bring about the appropriate functionality by reading or updating the feature of the object.

To create an object, a fact of this form is asserted

(CREATE <object-id> <class>)

An object is destroyed by asserting a fact of the following form

(DESTROY <object-id>)

These facts trigger the CREATE-INSTANCE and DESTROY-INSTANCE rules that instantiate or remove the object and its associated features.

It is also useful to provide other functions to return current information about a given object. Listing the features of an object may be accomplished by asserting the following fact

(SHOW-FEATURES <object-id>)

Similarly, listing the parent class or classes of an object may be accomplished by asserting a fact such as this.

```
(SHOW-CLASS      <object-id>)
```

These facts cause the SHOW-FEATURES and SHOW-CLASS rules to display the desired information.

It should be noted that these are top level functions for the benefit of the programmer. Analogues to the top level GET and SET functions are provided at the object level by the messages RETURN-VALUE, REQUEST, and APPLY-METHOD, discussed in further detail below.

3.2 Top Level Object Management Rules

The fact structuring conventions described so far do not provide the functionality needed, although they can be used to trigger rules that do. In the following discussion of the implementation of these rules, it is necessary to introduce the syntax of the CLIPS expert system. A complete presentation of the CLIPS language may be found in (Giarratano, 89) and in (Giarratano and Riley, 89).

The GET rule is used to obtain the current value of a feature of a particular object. It is implemented as follows.

```
(defrule      GET
      (GET                ?object-id     ?feature)
      (INSTANCE           ?object-id     ?class)
      (HAS-FEATURE        ?object-id     ?feature      $?values)
=>
      (printout t     ?object-id " has feature " ?feature)
      (printout t     " with value " $?values crlf))
```

The SET rule is similar, however, it contains additional code to manage the retraction of the old HAS-FEATURE and the assertion of a new HAS-FEATURE fact.

```
(defrule      SET
      (INSTANCE  ?object-id     ?class)
      ?x <- (SET    ?object-id     ?feature      $?new-values)
      ?y <- (HAS-FEATURE       ?object-id     ?feature      $?values)
=>
      (retract ?x ?y)
      (assert (HAS-FEATURE ?object-id ?feature $?new-values)))
```

Object instantiation is accomplished using a two-step process. First, CREATE-INSTANCE recursively traverses the inheritance chain, asserting facts that declare this object to be an instance of each of its ancestor classes. In the second phase, CREATE-FEATURES asserts facts to declare this object's features, and CREATE-METHODS asserts facts to declare this object's methods.

```
(defrule      CREATE-INSTANCE
      (CREATE             ?object-id     ?class)
      (CLASS              ?class         ?parent-class)
=>
      (assert       (INSTANCE ?object-id ?parent-class)))
```

```
(defrule      CREATE-FEATURES
      (INSTANCE          ?object-id      ?class)
      (HAS-FEATURE       ?class
            ?feature     $?default-values)
=>
      (assert
            (HAS-FEATURE      ?object-id      ?feature $?default-values)))

(defrule      CREATE-METHODS
      (INSTANCE          ?object-id      ?class)
      (HAS-METHOD        ?class          ?method-name)
=>
      (assert (HAS-METHOD      ?object-id      ?method-name)))
```

Object deletion is accomplished similarly. Instances of an object and any features and methods of the object must be removed. DESTROY-INSTANCE handles the former, and DESTROY-FEATURES and DESTROY-METHODS the latter.

```
(defrule      DESTROY-INSTANCE
      (DESTROY                   ?object-id)
      ?x <- (INSTANCE      ?object-id ?class)
=>
      (retract ?x))

(defrule      DESTROY-FEATURES
      (DESTROY                   ?object-id)
      ?x <- (HAS-FEATURE         ?object-id ?feature $?values)
=>
      (retract ?x))

(defrule      DESTROY-METHODS
      (DESTROY                   ?object-id)
      ?x <- (HAS-METHOD          ?object-id ?method-name)
=>
      (retract ?x))
```

The SHOW-FEATURES rule displays all the features of a particular object. It is implemented as follows:

```
(defrule      SHOW-FEATURES
      (SHOW-FEATURES ?object-id)
      (INSTANCE          ?object-id      ?class)
      (HAS-FEATURE       ?object-id      ?feature $?values)
=>
      (printout t ?feature " with value " $?values crlf))
```

The SHOW-CLASS rule, listing the parent class or classes of an object, is similar.

```
(defrule      SHOW-CLASS
      (SHOW-CLASS        ?object-id)
      (INSTANCE          ?object-id      ?class)
=>
      (printout t      ?object " is instance of class " ?class crlf))
```

4. MESSAGE PASSING

A message can be either a request for data or the initiation of a procedural action. Both types of messages can be implemented using structured facts and rules.

4.1 Feature Extraction Using Facts

In 3.2, the GET rule was used to print the value of a feature. It is also useful to extract an object's feature value and make the information available for use by other objects. Such a request for data can be made by asserting a fact of the following form:

```
(REQUEST    <calling-object-id>
            <target-object-id>    <feature>)
```

The <calling-object-id> identifies the object that initiated the request, the <target-object-id> shows which object is being queried, and <feature> indicates the feature of interest in the target object. The REQUEST fact is detected by a general "request manager" (see 4.3) rule that removes the REQUEST fact, polls the target object for the requested value, and creates a reply to the message by asserting a fact of the following form:

```
(RETURN-VALUE   <calling-object-id>   <feature>   <value>)
```

4.2 Method Invocation Using Facts

A request to invoke a method is accomplished by asserting a fact such as

```
(APPLY-METHOD   <calling-object-id>   <target-object-id>
            <method> [optional parameters])
```

Again, the <calling-object-id> identifies the object that requested the invocation, the <target-object-id> shows which object is being queried, and <method> indicates the method of interest in the target object. In some method invocations, it may be necessary to pass one or more parameters. Unlike the generalized REQUEST operation above, where a single rule can handle all requests by all objects, each method requires a separate rule. The rule performs the necessary computation, and creates a reply to the message by asserting a fact of the following form:

```
(RETURN-VALUE   <calling-object-id>   <feature>   <value>)
```

4.3 A Rule to Implement Feature Extraction

The rule that manages REQUEST messages is as follows:

```
(defrule    REQUEST-MANAGER
    ?x <- (REQUEST      ?caller-id
                        ?target-id      ?feature)
    (HAS-FEATURE        ?target-id      ?feature      $?value)
    (INSTANCE           ?target-id      ?target-class)
=>
    (retract ?x)
    (assert
        (RETURN-VALUE ?caller-id        ?feature      $?value)))
```

The rule requires the following: a REQUEST fact must exist, the requested feature must be declared as a feature of the target object, and the target object must have been previously instantiated. The (retract ?x) statement removes the REQUEST fact.

4.4 Rules to Implement Method Invocation

Each method defined in a class must be accompanied by an appropriate rule to implement the method. A uniform naming convention, such as the concatenation of the class name, a hyphen, and the method name, can be very useful. The following rule is an example of a method to compute the area of a rectangle.

```
(defrule        rectangle-area
?x <- (APPLY-METHOD     ?caller-id      ?target-id
                rectangle-area)
        (HAS-FEATURE    ?target-id      length  ?length)
        (HAS-FEATURE    ?target-id      width   ?width)
        (INSTANCE       ?target-id      rectangle)
=>
        (retract ?x)
        (assert
        (RETURN-VALUE   ?caller-id      area    =(* ?length ?width))))
```

This rule requires the following: a request to apply the "area" method must exist, the target object must have a length and width feature, and the target object must have been previously instantiated. The RETURN-VALUE fact contains the desired arithmetic result, where the =(* ?length ?width) statement is used to calculate the result of the expression and store the value as a field of a fact. The (retract ?x) statement removes the APPLY-METHOD fact.

4.5 Message Cleanup

In the feature extraction rule (4.3) and the method invocation rule (4.4), statements were explicitly included to remove the message fact that caused the rule to fire. It is useful to consider a general purpose message cleanup method, similar to the "garbage collection" operation performed by most symbolic computation implementations. A low priority rule can be used to remove all messages and interim results after each message processing cycle. This ensures that old messages or return value facts are not accidentally reused or misused at a later time.

In order to simplify the implementation of the garbage collection rule, a fact containing each object-oriented command keyword is used. As command keywords are expected to occur in the first field of a structured fact, any fact containing a command keyword in its first field at the end of a message processing cycle can be assumed to be eligible for garbage collection.

A simple garbage collection rule can be implemented as follows. The list of commands is stored in the COMMAND-LIST fact, and the garbage collection rule tests to ensure that the selected fact contains a command.

```
(deffacts COMMAND-LIST
        (COMMAND-LIST
                RETURN-VALUE
                CREATE          DESTROY
                SET             GET
                SHOW-ATTRIBUTE          SHOW-CLASS))
```

```
(defrule        garbage-collection
        (salience -5)
        (COMMAND-LIST $?cmd-set)
        ?x <- (?cmd&:(member ?cmd $?cmd-set) $?cmd-tail)
=>
        (retract ?x))
```

5. USING OBJECT-ORIENTED KNOWLEDGE REPRESENTATION

The techniques described in this problem were used as the basis for an object-oriented knowledge base that was incorporated in the software prototype of a satellite metatdata information system. This domain is typified by very large quantities of data, suggesting a fundamental need for intelligent query and browse features. More thorough treatments of satellite information systems can be had in (Corey and Carnahan, 90) and (Roeloffs and Campbell, 90).

The project constraints dictated that the expert system component had to be portable, extensible, flexible, and robust. In addition, it had to be developed rapidly, and interface readily with existing C and C++ software on UNIX[1] platforms. The NASA CLIPS tool was virtually ideal for this task, although it lacked support for object-oriented modelling.

One of the lessons learned from this experience was that an object-oriented knowledge base can be fairly easy to integrate with traditional software. The object-oriented approach allowed the existing C programs to be minimally affected. The expert system functioned transparently to the C code, providing services through the use of the assert() and run() function calls.

The expert system software architecture was based on a "state-machine" model, with control states similar in some respects to the "read evaluate print garbage-collect" cycle of a LISP interpreter. The expert system, once initialized, entered a "read" state, where it waited for requests for information. Messages, in the form of facts asserted into the knowledge base by procedural C/C++ code, would either supply information about the constraints of the current query, or cause the system to enter an "evaluation" state.

In the "evaluation" state, the expert system applied rules to find information meeting the criteria dictated by messages. The derived information, typically tokens in the knowledge base that matched the left-hand-side (LHS) of a rule, would be processed on the right-hand-side (RHS) of the rule as parameters to a user defined C function that appended the information to a globally available linked list structure.

After a given request was satisfied, the expert system entered a "garbage-collection" state. All messages, commands, and intermediate facts were eliminated, and the system then returned to the initial "read" state to wait for additional messages.

The application program that invoked the expert system handled the "print" phase. Information was taken from the linked list structure generated by the expert system and used by the C program.

This method made the interfaces between the C/C++ code and the expert system very straightforward. The C programs made use of the expert system by including the <clips.h>

---

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

headers, loading and initializing selected rule and fact files, declaring a global linked list of strings to hold query results, and proceeding with the application processing.

When information was needed from the expert system, a character string containing a syntactically valid data request (typically in the form of a constraint or a command) was assembled and "asserted" into the expert system. When all requests were in place, a call to the run(-1) function was made, activating the expert system rules. When the expert system had finished, the results of the query were available in a linked list. The C program could proceed at this point with its processing, which typically involved presenting the information onto an X Windows[2] dialog box.

6. CRITICAL ASSESSMENT AND PERFORMANCE ISSUES

The use of structured facts and rules as suggested in this work is no different than the use of other fact structuring conventions, such as the Object Attribute Value (OAV) triple or the IS-A or AKO (A-Kind-Of) links in semantic networks. The approach offers more functionality than the OAV model, since procedural and data elements can be associated with an object.

The method may offer certain performance advantages over more sophisticated systems, since expert systems are optimized to process rules and facts, rather than objects and messages. The method is reasonably portable, although rules may need to be recoded in order to accommodate the particular syntax of the tool being used.

Perhaps the greatest advantage of this method is its high degree of flexibility. While a number of expert system shells incorporate object oriented extensions, few if any allow the user to completely redefine the syntax of the object manipulation language at will.

A detrimental performance aspect of this method is that using facts to implement inheritance can cause a great many facts to be asserted when objects are instantiated. If deep inheritance chains are modeled, multiple instantiations of an object far down the chain may begin to pose memory and speed problems on the expert system inference engine.

Other, more subtle problems also exist. The method does not account for feature inconsistencies, hence it is possible for an object to inherit the same feature name from different parent classes if the default values are different. This will result in an inconsistency, since the object will manifest two features with the same name yet different values. Other object-oriented concerns, such as selective inheritance, or precedence of local features or methods over inherited features or methods are not addressed.

7. CONCLUSIONS

This paper has introduced a method for defining classes, objects, and messages in an expert system. The method can be implemented using conventional hardware and very straightforward expert system tools, and does not require a sophisticated run-time object or message manager. It requires some programmer-imposed conventions on facts and rules, and calls for the use of structured facts containing object-oriented keywords. Promising results have been obtained by incorporating an expert system using object-oriented knowledge representation with traditional C code.

---

[2] The X Window System is a trademark of the Massachusetts Institute of Technology.

8. REFERENCES

(Assal and Myers, 90) Assal, Hisham, and Leonard Myers. "An Implementation of a Frame-based Representation in CLIPS." First CLIPS Conference Proceedings, NASA Conference Publication 10049, Volume II, pp. 570-580.

(Corey and Carnahan, 90) Corey, Stephen M., and Richard S. Carnahan. "Knowledge Structure Representation and Automated Updates in Intelligent Information Management Systems." 1990 Goddard Conference on Space Applications of Artificial Intelligence, NASA Conference Publication 3068, May 1990, pp. 271-282.

(Giarratano and Riley, 1989) Giarratano, Joseph C., and Gary Riley. Expert Systems: Principles and Programming. Boston, PWS-Kent Publishing, 1989.

(Giarratano, 1989) Giarratano, Joseph C. CLIPS User's Guide, Artificial Intelligence Section, Lyndon B. Johnson Space Center, NASA. 1989.

(Leung and Wong, 90) Leung, K. S., and M. H. Wong. "An Expert System Shell Using Structured Knowledge." IEEE Computer. March 1990. p38-47.

(Meyer, 88). Meyer, Bertram. Object Oriented Software Construction. New York, Prentice Hall, Inc., 1988.

(Mylopoulos and Levesque, 83) Mylopoulos, John and Hector Levesque. "An Overview of Knowledge Representation." in Brodie, M.L., Mylopoulos, J., and Schmidt, J.V.,(eds.) On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages, Springer-Verlag, New York, 1983.

(Roeloffs and Campbell, 90) Roeloffs, Larry H., and William J. Campbell. "Using Expert Systems to Implement a Semantic Data Model of a Large Mass Storage System." 1990 Goddard Conference on Space Applications of Artificial Intelligence, NASA Conference Publication 3068, May 1990, pp. 253-270.