Third International Conference on Tools for AI

Components of Expertise for Knowledge Level Modeling

Ruey-Juin Chang and Gordon S. Novak, Jr.
Artificial Intelligence Laboratory
The University of Texas at Austin

Topics: AI Lanuguage Toolkits, Expert Systems and Environments

ABSTRACT

A new dimension of object-oriented components is presented for knowledge level modeling. From the knowledge level perspective, components are needed for modeling the problems, solution methods and their relationships in a problem-solving process. A problem can often be decomposed recursively by various methods into subproblems until reaching primitive ones. Each primitive problem is confined to a simple computational task, which can be solved by one problem-solving method. The process of problem solving is viewed as a modeling activity through explicit representation knowledge level models that are, then, operationalized. This paper describes design issues encountered when developing such a reusable component library for knowledge level models.

1. Introduction

For the design of reusable component libraries in the past, NIH[Gorlen, 87], Libg++[Lea, 88], InterViews[Linton, 89], Booch Components[Booch, 90], the type of components used mainly is of generic problem-solving methods. However, from a knowledge level perspective, methods only are not sufficient for modeling the process of problem-solving. Many other types of components are currently identified by researchers, for example, KADS, Generic Task and CML[Breuker, 86][Chandrasekaran, 89][Vanwelkenhuysen, 90] respectively. The knowledge level analysis of problems has received increasingly attention. This paper presents certain related types of components from the perspective of knowledge level modeling. The component types of task structure, problem-solving method, and computational reflector, and their relationships, are discussed for modeling the process of problem-solving. A *task structure* is provided to model the problem/subproblem relationships and its applicable methods. A *problem-solving method* is used to describe how to solve a task. There can be multiple methods for a task. A task is executed by invoking one of its applicable methods. The methods can be task-specific since the relationships to a task can be described in the task structure. More efficient methods can be provided for particular task structures. *Reflectors* are used to describe how to operationalize (execute) a component. The executable components can turn the knowledge level models of an application into a working system. In addition, explicit representation of knowledge level components makes the application systems easier to be extended, adapted, and maintained.

2. The Concept of Programming Cliches

A cliche model is described to represent all types of components. AI researchers and expert programmers usually need to use much higher level programming structures than those expressible in current programming languages such as LISP or PROLOG. Such high level programming structures or problem solving techniques have been termed cliches by expert programmers. Cognitive studies[De Groot, 65] also indicate that abstraction levels are often utilized by experts for solving problems. The experts develop and use "chunks" that correspond to the functional units in their domains. Problem solutions can be reformulated into abstract concepts and experts perform better than novices in their domains because of these abstract concepts.

One example of a cliche is *FILTER,* which selects a subset satisfying some predicates from a list of items. This selection involves an iteration over a list of items at the implementation level. Thus, there can be many variations of filter programs depending on the selection method and the data structure of the list. However, they all perform similar functions conceptually. Other abstract operators could be COLLECT, TOTAL, AVERAGE, MAXIMUM, MINIMUM, and so on. This is the original concept of programming cliches for abstracting stereotypical functional units.

2.1. The Cliche Model

Our cliche model goes beyond the original idea of programming cliches. It is designed to uniformly represent components for knowledge level modeling. From a knowledge level perspective, there are at least three types of components currently being identified as important elements; there are problem-solving methods, task structures, and computational reflectors. In fact, the original idea of a programming cliche is only a class of primitive task structures which use a single problem-solving method for solution. Often, the types of task structure and methods are indistinguishable. The class of methods is also known as well-structured algorithms that produce solutions for a problem without any search in the space of alternatives. However, the other types of problem-solving methods could be high-level controls or problem space search methods. These types of methods are also useful for solving tasks, particularly for solving composite tasks. Furthermore, the methods, tasks, and reflectors must be integrated in an open-ended and coherent way. For taking into account all aspects, three essential pieces information are described in our cliche model shown below: (1) template description and composition; (2) template instantiation and interpretation; and (3) meaning function. The cliche model provides a modular way for both *composition* and *interpretation* of cliches.

A cliche is a specialized GLISP structured object description as shown below[Novak, 82b, 83a, 83b, 83c]. A GLISP object description is a description of the structure of an object in terms of named substructures, together with definitions of ways of referencing the object.

```
(<cliche-name>          (<structured-descriptions>)
SUPERS        (<list-of-cliche-names>)
PROP
        ((interpretation <cliche-name>)
         (interpreter <cliche-name>)
         (<cliche-type spècific properties>)
         (<other properties descriptions>))
MSG
        ((meaning <message description for intensional semantics>)))
```

The object description defines the lisp data structure for the object and the optional properties. The SUPERS property describes a list of cliches from which the PROP and MSG properties can be inherited. The PROP and MSG properties specify the computable properties. Each description in the properties has the format:

```
(<name> <response><prop-1><value-1>...<prop-n><value-n>)
```

where <name> is the name of property, <response> can be either a function name or a list of GLISP code to be compiled in place of the property, and the <prop><value> pairs are optional properties that affect the compilation. Both kinds of properties are compiled in a similar fashion. One of the distinguishing GLISP features is its *structured descriptions* of objects, which is different from the generalized record-like descriptions in traditional object-oriented programming languages. These structured object descriptions result in more flexibility to describe cliches. Usages of the following particular GLISP features are discussed in detail.

The GLISP **context reference** capability is quite powerful for describing the dynamic behavior of a cliche's meaning function. In GLAMBDA programs, as shown below, the GLISP compiler organizes the context levels on a stack.

```
(<program-name> (GLAMBDA (<arguments> )
  (PROG( <prog-variables> )
       <code>)))
```

Searching of the context proceeds from the top level of the stack to the bottom level. The bottom level of the stack is composed of the LAMBDA variables of the function being compiled. New levels are added to the context when the following keywords are confronted by the compiler: PROG, FOR, WHILE, and IF[Novak, 1983a]. One major difference between a GLISP program and a common LISP definition is the presence of type declarations for prog-variables, which are in a syntax of the following forms: <variable>:<type> or <variable>:(a <type>) such as declarations in (PROG((X 1) Y:integer Z: (a real)). The type declaration can be any GLISP object description.

The GLISP **transparent** structure description (transparent <type>) provides a convenient way of describing composite objects. When an object of type <type> is composed into a composite object by using the structure description, **transparent**, this substructure object is in *transparent mode*. It means that all fields (data structures) and properties of this substructure can be directly referenced as if they were properties of the object being defined[Novak, 1983a]. When a substructure is a named type and is not declared to be **transparent**, it is assumed to be opaque; that is, its internal structure cannot be seen unless an appropriate access path is explicitly specified. Thus, structure description **transparent** is very useful when being used to describe hierarchical cliches. For instance,

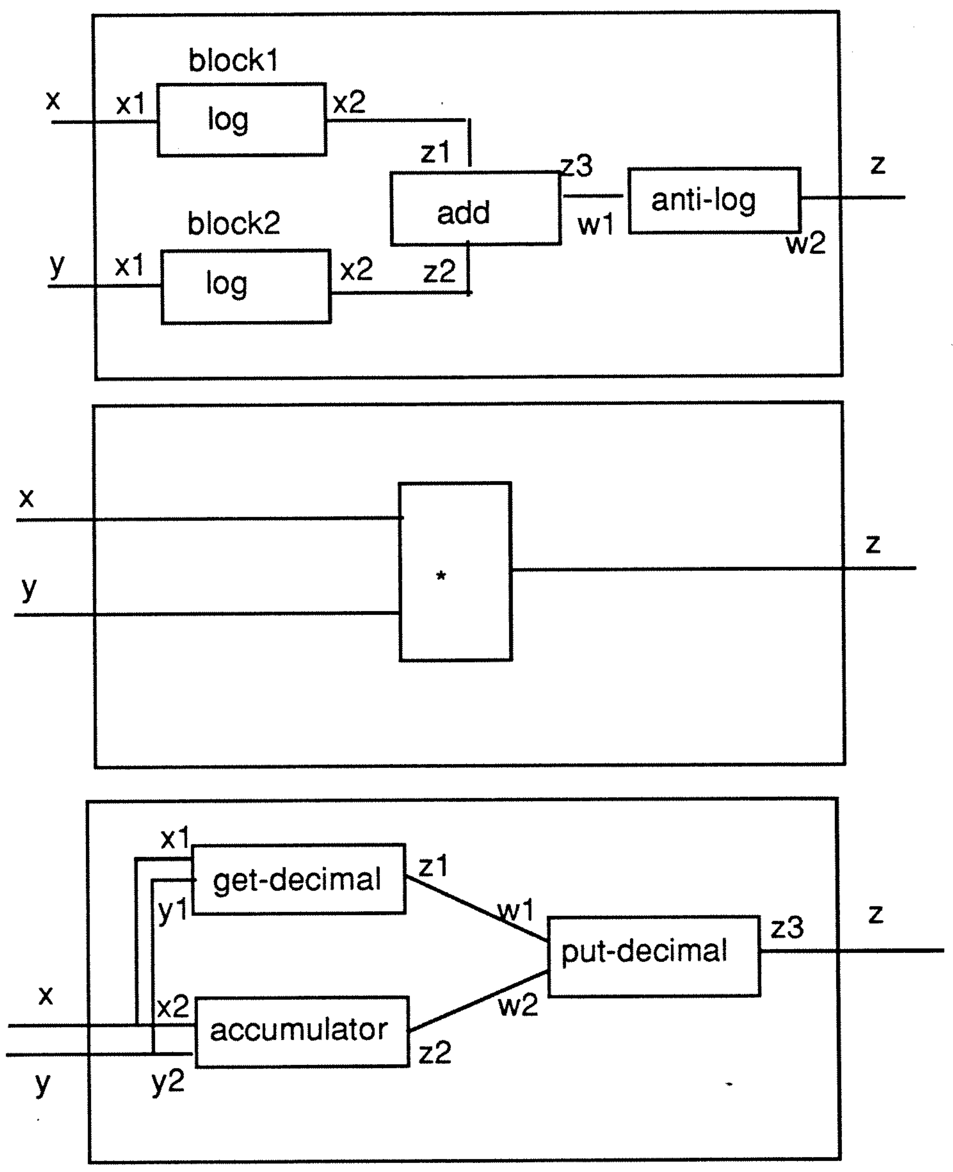
```
(TASK   (listobject      (block1 (transparent  SUBTASK1)
                         (block2 (transparent  SUBTASK2)
                         (INPUT anything)(OUTPUT anything))
PROG          • • •

MSG           • • •)
```

A substructure reference could occur in GLISP instance generation, a GLAMBDA program, or GLISP object descriptions. For example, (A TASK WITH INPUT = ..., FIELD1 =...) generates an instance of structured object TASK. Due to the transparent type, any data structure of SUBTASK1 and SUBTASK2 can be instantiated by directly specifying a value as if it were a field/property of TASK. Note that INPUT is a field name in TASK and FIELD1 is a substructure name in SUBTASK1. In addition, by using **transparent**, the constraints of the subtask connections can be described in terms of any field of subtasks.

2.2. Components of Expertise

These cliches are used to represent knowledge level components. There are three types of components: (1) problem-solving methods, which could be higher level operators, control structures, or problem space search; (2) task structures for modeling the problems; and (3) reflectors for describing computation of components. The cliche descriptions of different types of components will be shown. A simple example of the task *multiplying-two-real-numbers* is used to illustrate how to instantiate the descriptions.

Task:multiply
Decomposed by Methods



Task Structure: for modeling the problem/subproblems relationships and applicable methods.

(*<task-name>* (*<list-of-default-subtasks> <list-of-task-I/O-ports>*)
SUPERS (*<list-of-task-names>*)
PROP
((interpretation *<reflectors>*)
(interpreter *<reflectors>*)
(block *<list-of-subtask-names>*)
(connection *<inter-subtasks-connection>*)
(model *<list-of-task-I/O-port-names>*)

```
          (<task-specific properties>))
MSG
          ((meaning <problem-solving-methods>)))
```

<problem-solving-methods> can be a GLISP expression or function name for describing a *single* method. If there are *multiple* methods that can be applied to the task structure, then *<problem-solving-methods>* is a list of method names. These method names denote the cliches that describe the problem-solving methods.

Example-1 (Task Structures):

```
(multiply     (listobject     (block1 (transparent log))
                              (block2 (transparent log))
                              (block3 (transparent add))
                              (block4 (transparent anti-log))
                              (x real)(y real)(z real))
PROP
          ((interpretation ('PRODUCTION-SEMANTICS))
           (interpreter ('INTERPRETER))
           (block  ('(block1 block2 block3 block4)))
           (connection     ((x (block1 x1))(y (block2 x1))
                            ((block1 x2) z1)((block2 x2) z2)
                            (z3 w1)(w2 z)))
           (model ('(multiply(x y)(z)))))
MSG
          ((meaning        ('(       by-logarithm
                                     by-accumulator
                                     operator-*)))))


(log    (listobject        (x1 real)(x2 real))
PROP
          ((interpretation ('MEANING))
           (interpreter ('INTERPRETER))
           (model ('(log(x1)(x2)))))
MSG
          ((meaning        (log x))))


(add    (listobject        (z1 real)(z2 real)(z3 real))
PROP
          ((interpretation ('MEANING))
           (interpreter ('INTERPRETER))
           (model ('(add (z1 z2)(z3)))))
MSG
          ((meaning        (+ z1 z2))))


(anti-log        (listobject        (w1 real)(w2 real))
PROP
          ((interpretation ('MEANING))
           (interpreter ('INTERPRETER))
           (model ('(anti-log(w1)(w2)))))
MSG
```

```
        ((meaning      (anti-log w1))))
```

Problem-Solving Method: for specifying how a task can be solved. It can be either a primitive problem solution or a decomposition method that describes the problem solving sequence of the subtasks.

```
(<method-name>         (listobject (task atom)(instance anything))
SUPERS        (<list-of-method-names>)
PROP
        ((interpretation <reflectors>)
         (interpreter <reflectors>)
         (condition <predicate>)
         (<properties for method-specific decomposed subtasks>))
MSG
        ((meaning <operator, control, or problem-space-search>)))
```

Method Spectrum: It is worth mentioning that the methods are associated with task structures. That is, the methods are task-specific. A task-specific method can describe how its task is specifically decomposed and, hence,can be more efficient. At the other extreme, the methods can be more generic or domain-specific. They are used for more than one task or domain. The default task decomposition is defined in the task structure.

Method Types: Methods are described as either a message form in a task structure or a method-type cliche. No matter what form a method uses, it describes one of the following functions: (1) high level operators; (2) high level controls, such as Task Sequence Control for data flow task execution sequence, Generic Method Dispatcher for multiple variant methods of the problem, Program Structure for describing program templates to compose the components, and so on; and (3) problem-space search for describing a search space of multiple decompositions of the problem.

Example-2 (Problem-Solving Methods):

```
(by-logarithm          (listobject (task atom)(instance anything))
PROP
        ((interpretation ('MEANING))
         (interpreter ('INTERPRETER))
         (condition ('false)))
MSG
        ((meaning task-sequence)))
```

The above method decomposes the task **multiply** by default. The default decomposition is defined in the task itself as shown in the first example. That is, the subtasks consist of two tasks of **log**, one task of **add**, and one task of **anti-log**. Those subtasks are primitive tasks without further decomposing.

```
(by-accumulator        (listobject (task atom)(instance anything))
PROP
        ((interpretation ('MEANING))
         (interpreter ('INTERPRETER))
         (condition ('true))
         (block (         '((block1 get-decimal)
                           (block2 accumulator)(block3 put-decimal))))
         (connection      ((x x1)(y y1)(x y2)(y y2)
                           (z1 w1)(z2 w2)(z3 z)))
        (model ('(multiply(x y)(z)))))
MSG
        ((meaning task-sequence)))
```

In the method **by-accumulator**, the task **multiply** is explicitly decomposed into three primitive subtasks: (1) **get-decimal** for totaling number of digits after decimal points in both inputs, (2) **accumulator** for accumulating the first input a number of times that is equal to the second input, ignoring the decimal points, and (3) **put-decimal** for getting a value from the first input and replacing the decimal point at that value in a position according to the second input.

```
(operator-*    (listobject (task atom)(instance anything))
PROP
        ((interpretation ('MEANING))
         (interpreter ('INTERPRETER))
         (condition ('false)))
MSG
        ((meaning (* x y))))
```

Reflector: for specifying how the cliche evaluates its meaning.

```
(<reflector-name>        (<formal-arguments>)
SUPERS        (<list-of-reflector-names>)
PROP
        ((interpretation <reflectors>)
         (interpreter <reflectors>)
         (<reflector-specific properties>))
MSG
        ((meaning <a GLISP message form>)))
```

These cliches are **executable** and the executions are based on their **meanings** and **reflectors** in properties *interpretation* and *interpreter* which explicitly describe how the cliche meaning is executed. The default execution of a cliche, by using reflectors MEANING and INTERPRETER, is to send the message *meaning* to the cliche. The reflector "production-semantics" in task *multiply* describes that the method selection from multiple choices is based on the type of production systems and method conditions. For example, we can execute *multiply* with inputs x equals to 4.0 and y equals to 5.0.

```
(DO multiply with x = 4.0 y = 5.0) or
(DO multiply with $Instance (A multiply with x = 4.0 y = 5.0))==> 20.0
```

The task *multiply* is executed by one of the three possible methods depending on conditions of methods. Each method specifies a decomposition of the task. The execution sequences of subtasks shown below can be generated by the same meaning function *task-sequence* from method descriptions of **by-logarithm** and **by-accumulator,** respectively, if they are chosen. In this particular case, the execution program from **by-accumulator** is actually generated since the method condition is "true". Note that the operator DO is recursively called in both execution sequences until reaching the primitive tasks.

The following program is generated to describe the execution sequence of *multiply* by meaning *task-sequence* from the method description **by-logarithm:**

```
(GLAMBDA(self: multiply)
  (PROG()
        (block1:x1 <- x)
        (block2:x1 <- y)
        (DO log with $Instance block1)
        (DO log with $Instance block2)
        (z1 <- block1: x1)
        (z2 <- block1: x2)
        (DO add with $Instance block3)
        (w1 <- z3)
        (DO anti-log with $Instance block4)
        (z <- w2)
        (return z)))
```

The following program is generated to describe the execution sequence of *multiply* by meaning *task-sequence* from the method description **by-accumulator**:

```
(GLAMBDA(self:multiply)
  (PROG(block1:get-decimal block2:accumulator block3:put-decimal)
        (x1 <- x)
        (y1 <- y)
        (x2 <- x)
        (y2 <- y)
        (DO get-decimal with $Instance block1)
        (DO accumulator with $Instance block2)
        (w1 <- z1)
        (w2 <- z2)
        (DO put-decimal with $Instance block3)
        (z <- z3)
        (return z)))
```

There are several interesting points in using this kind of implementation technique for developing such a knowledge level component library:

(1) Task structures can be recursively decomposed by the associated problem-solving methods. That is, a method decomposes its task into subtasks, which are solved by their own respective methods. If subtasks are not the primitives, then they can be decomposed again by associated methods.

(2) There can be multiple ways to describe task decomposition. The way that is used for decomposition at task execution depends on which methods are chosen. The selection of associated methods depends on run-time context and condition of methods. For instance, the above example uses a constant Boolean value in the condition of methods.

(3) The meaning *task-sequence* in the method by-logarithm or by-accumulator first uses its own properties, **model**, **block, connection,** for task decomposition. If unavailable, then the defaults are taken from its task structure. For example, by-logarithm takes the defaults and by- accumulator uses its own decomposition.

(4) Computational reflectors provide user-definable execution aspects of each type of component. For example, execution semantics of multiple methods in a task structure can be sequential, production system, etc.

(5) The problem-solving process can be described in terms of higher level components. The different types of components are smoothly integrated to describe the process.

In addition, a **framework** of a cliche-based environment can be built by a set of cliches that embodies an abstract design for solutions to a family of related problems and supports reuse at a larger granularity than the individual cliche does. The enhanced version of the PROGRAM command in GLISP Display Inspector/Editor (GEV)[Novak, 82a,82b] can use the cliches as the building blocks to construct more complex programs from problem descriptions[Chang, 90]. The reusable components can be selected from an object-oriented cliche library.

2.3. A Process View of Object Design

In an object-oriented programming paradigm, there are two viewpoints of object descriptions: **data view** and **process view**. In data view, objects are used to describe data. In process view, objects are used to model the operation behaviors like program skeletons or generic programs and are used to reuse and extend the behaviors in an object-oriented way. This view of object description was seldom discussed in the past. However, if an operation is (1) a meaningful abstraction, (2) likely to be shared by several classes, or (3) complex enough, then it is worthwhile to model this operation as a separate object. In other words, the computational tasks that are operations for manipulating data can be, and probably should be, modeled as separate objects. This is not contrary to the object-oriented programming paradigm.

Cliches are often touted as promoting software reuse. However, the cliches must be *designed* for reusability. From the process view of objects, cliches can be designed to provide more reusability of the computational aspects, such as how to evaluate the cliches, how to instantiate the cliches, and how to modify existing templates for slightly different applications. These computational aspects are crucial to reusable components for executable knowledge level models.

3. Cliche-Based Reflective Architecture

The reflectors, MEANING, INTERPRETER, or any specialization of these primitive ones, are used to describe the computational aspects of components. These user-definable computations cause closer relationships between knowledge-level models and implementations. In object-oriented programming, the computation can be based on either (1) message sending such as SMALLTALK[Goldberg 83], (2) generic function dispatching such as CLOS[Bobrow, 88a], or (3) referent computing such as KRS[Steels, 88][Marke, 88]. Many object-oriented reflection systems[Briot, 87][Cointe, 87][Graube, 88][Ferber, 89] focus on the process of message sending, i.e., reifying entities during the process of message sending. CLOS/MOP[Bobrow, 88b][des Rivieres, 90] does reflection during the process of dispatching generic function. 3-KRS[Maes, 87a, 87b] is built on KRS. Although it presents a framework of computational reflection during the process of message sending, the reflection cannot dynamically modify its own referent computing. The referent computing of KRS/3-KRS is rather fixed and hardwired in the system. In our system, the meaning computation of a cliche by reflectors is based on reflective referent computing. The reflective meaning computation provides a framework of customizable and user-definable referent computing.

3.1. Self-Representation of the Cliche Interpreter

The interpreter for cliche evaluation must be able to construct an *explicit* representation of the evaluation process and its current status. It is on this base of its self representation that the interpreter is able to reflect, i.e.,to reason about itself and to support actions upon itself. The meta circular interpreter for cliche evaluation is defined as shown below. In each cliche, property *interpreter* and property *interpretation* specify cliche names such as MEANING and INTERPRETER that actually provide the explicit representation of the evaluation process.

```
(DO cliche with p)
        =
(If (cliche.interpreter = 'INTERPRETER)
  then (If (cliche.interpretation = 'MEANING)
          then (send  (a MEANING with class = 'cliche args= '(p))  meaning)
          else (DO cliche.interpretation with class = 'cliche args = '(p)))
  else (DO cliche.interpreter with class = 'cliche args = '(p)))
```

In GLISP, *(a cliche with field1 = value1 field2 = value2 ...)* generates an instance of structured object *cliche*. *Cliche.property* is used to access the value of *property* in *cliche*. Cliches MEANING and INTERPRETER are defined below. They are kernel cliches for default interpretation process. Users can modify the interpretation process by giving different values in property *interpretation* or *interpreter* of the cliche or by inheriting one in an object-oriented way.

```
(MEANING (list (class atom)(args (listof atom)))
 PROP
        ((interpretation ('MEANING))
         (interpreter ('INTERPRETER)))
MSG
        ((meaning (send  (eval `(a ,class with ,@args))  meaning))))

(INTERPRETER (list (class atom)(args (listof atom)))
PROP
        ((interpretation ('MEANING))
         (interpreter ('INTERPRETER)))
MSG
        ((meaning (eval `(DO ,class.interpretation with class= ,(kwote class) args= ,(kwote args))))))
```

We now explain certain key points in this version of meta circular interpreter as follows:

(1) A meta circular interpreter is an **explicit representation** of the **interpretation** and **interpreter** in the **language itself.** This cliche evaluation is based on two default cliches, MEANING and INTERPRETER. These cliches can be specialized in an object-oriented way.

(2) Property *interpretation* in each cliche is used to describe how to realize the abstract cliche by giving an instance of structured description of arguments. The realization of a cliche from the cliche description/composition is established by executing the command **(DO interpretation-cliche with ...)**. The default way is to send the message *meaning* to the cliche instance.

(3) Property *interpreter* in each cliche is used to specify how to interpret the realization of this cliche. The interpretation is performed by executing **interpreter cliche**, which is the value of cliche property *interpreter*. The default way is using the final realization as it is.

(4) The underlying evaluation of a cliche is based on LISP/GLISP interpreter. This means that the evaluation of cliches MEANING and INTERPRETER uses this underlying interpreter.

Two kinds of entities shown here (interpreter and interpretation) can be reified in the entire process of cliche evaluation. The first reification entity is cliche description and composition. The second entity is the cliche interpretation itself and cliche instantiation.

3.2. Reflection on Meaning Computation

The evaluation of our cliche description is based on *intensional semantics*[Steels, 88]. Each cliche is associated with a message *meaning*. The message *meaning* is user-definable to explicitly describe the behaviors of the cliche. The meaning of a cliche is used to *explicitly* model the semantics *inside* the cliche representation rather than *outside* or *implicitly*.

The meaning computation is invoked by executing (DO cliche with ...). It is based on computational reflection. *(DO cliche with ...)* recursively calls *(DO interpreter-1 with ...)* if *cliche.interpreter* is not INTERPRETER. Otherwise, it recursively calls (DO interpretation-1 with ...) if *cliche.interpretation* is not MEANING. If both default cliches, INTERPRETER and MEANING, are used, then it simply executes (send (a MEANING with class = 'cliche args = ...) meaning) or (send *an-instance-of-cliche* meaning). Note that the recursive calls stop when the default interpreter and interpretation are reached. Computational reflection is triggered at recursive calls according to the properties **Interpreter** and **Interpretation** in the cliches. If non-default computation is used, then the reflection will happen at meta levels according to the names in these two properties. The names are either specified by the users or inherited from supers. There can be a reflective computation tower. The effects of the reflection influence how the meaning of the cliche is computed.

Computation in cliche interpreter can be viewed as an infinite tower of interpreters and interpretations all reasoning about the level below as shown in FIG-1. The lowest level reasons about the actual problem domain. The other levels manipulate causally connected representations of the level below. Customization of reflectors can be done along the tower.

Reflective Tower

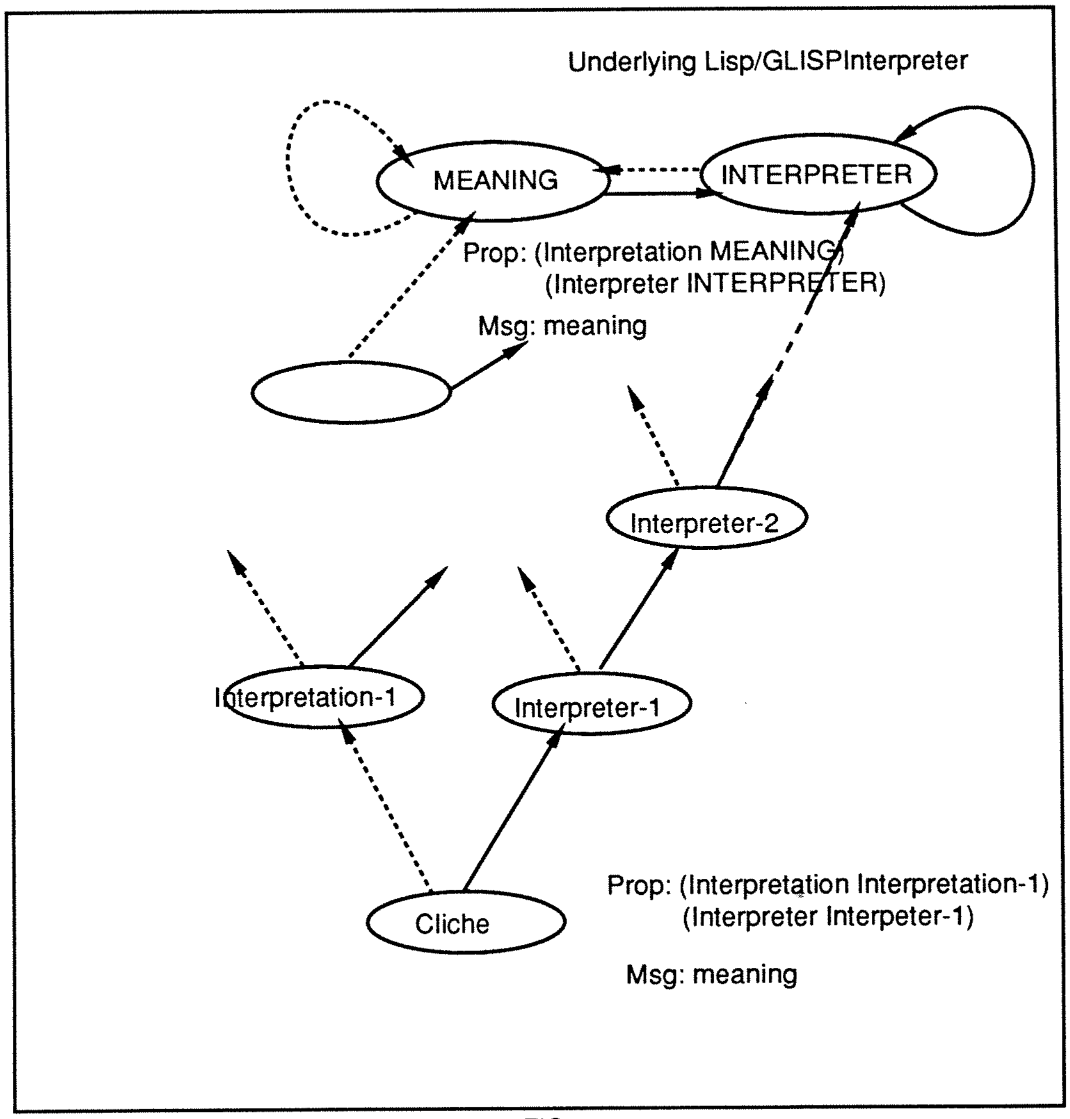

Underlying Lisp/GLISPInterpreter
MEANING
INTERPRETER
Prop: (Interpretation MEANING)
(Interpreter INTERPRETER)
Msg: meaning
Interpreter-2
Interpretation-1
Interpreter-1
Cliche
Prop: (Interpretation Interpretation-1)
(Interpreter Interpeter-1)
Msg: meaning

FIG-1

3.3. Specializations of Reflectors

Reflectors are used to specify how a component is executed and to make knowledge level components executable. Computational reflection architecture provides an open-ended modular evaluation of components. Users can modify or inherit defaults and they can also add their own reflectors. Due to space limitations, the following examples only give a glimpse of our DO interpreter of components modifiable at user level. Complicated examples of reflectors are shown in [Chang, 91].

Example-3 (Instance Generation):

In GLISP, *"a"* can be considered as a specialization of our meta circular interpreter, i.e., *(DO cliche with p)* will result in the same value of *(a cliche with p)* if using A-INTERPRETER for evaluating *cliche.*

```
(cliche           (list (x integer)(y integer))
PROP   ((interpreter ('A-INTERPRETER))
        (interpretation ('MEANING)))
MSG    ((meaning (+ x y)))
```

Note that *cliche* uses A-INTERPRETER instead of default cliche INTERPRETER in the property *interpreter.* This results in an instance generation rather than addition of x and y. Note that A-INTERPRETER itself inherits property values of *interpreter* and *interpretation* from its super INTERPRETER.

```
(A-INTERPRETER     (list (class atom) (args (listof atom)))
SUPERS          (INTERPRETER)
MSG             ((meaning (eval `(a ,class with ,@args)))))

(DO cliche with x = 2 y = 3)
=> (DO A-INTERPRETER with class = 'cliche args = '(x 2 y 3))
=> (send (a MEANING with class = 'A-INTERPRETER args = '(class 'cliche  args '(x 2 y 3)))
        meaning)
=> (send (a A-INTERPRETER with class = 'cliche args = '(x 2 y 3)) meaning)
=> (a cliche with  x  = 2 y = 3 )
=> (2 3).
```

Example-4 (KRS/3-KRS Referent Computing):

```
(REFERENT (list (class atom)(args (listof atom)))
PROP   ((interpretation ('MEANING))
        (interpreter ('INTERPRETER)))
MSG    ((meaning  (eval (DO MEANING with class = class args =  args)))))

(FOO    (list (x real)(y real))
PROP
       ((interpretation ('MEANING))
        (interpreter ('REFERENT)))
MSG
       ((meaning `(+ ,x ,y))))

(DO FOO with x =3 y = 5)
=> (DO REFERENT with class = 'FOO args = '(x 3 y 5))
=> (send (a MEANING with class = 'REFERENT args = '(class 'FOO args '(x 3 y 5))) meaning)
=> (send (a REFERENT with class = 'FOO args = '(x 3 y 5)) meaning)
=> (eval (DO MEANING with class = 'FOO args  ='(x 3 y 5)))
=> (eval (send (a MEANING with class = 'MEANING args = '(class 'FOO args '(x 3 y 5))) meaning))
=> (eval (send (a MEANING with class = 'FOO args = '(x 3 y 5)) meaning))
=> (eval (send (a FOO with x  = 3 y = 5) meaning))
=> (eval (+ 3 5))
=> 8.
```

It is interesting to see that we can simulate KRS/3-KRS referent computing by using cliche REFERENT in *interpreter* property. For comparison, the following case is computed by using default cliche INTERPRETER instead of REFERENT. In fact, many variants of referent computing can be defined in this architecture.

(DO FOO with x = 3 y = 5) => (+ 3 5).

4. Conclusions

A framework of an object-oriented component library for knowledge level modeling is presented. Components are represented as cliches that take the process view of objects. There are three types of components needed for modeling a problem-solving process from a knowledge level perspective. From this perspective, task and problem-solving method components must be smoothly integrated in a coherent way. Reflector components are used to make components executable in a user-definable way. We believe that an object-oriented programming environment should be able to provide programmers with libraries of commonly used components that could be domain-specific and task-specific and from which they can pick suitable components whenever needed for knowledge level modeling.

References

De Groot, A., 1965, *Thought and Choice in Chess*, Mouton, Paris, France.

Bobrow, D., L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon, 1988a, Common Lisp Object System Specification X3J13 Document 88-002R, *Special Issue of SIGPLAN Notices*, **23**(9).

Bobrow, D., and G. Kiczales, 1988b, The Common Lisp Object System Metaobject Kernel: A Status Report, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah.

Booch, G., 1990, The Design of the C++ Booch Components, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications and European Conference on Object-Oriented Programming (OOPSLA/ECOOP-90)*, Ottawa, Canada.

Breuker, J., and B. Wielinga, 1986, Models of Expertise, *Proceedings of the 1986 European Conference on Artificial Intelligence (ECAI-86)*, Brighton, England.

Briot, J., and P. Cointe, 1987, A Uniform Model for Object-Oriented Languages Using The Class Abstraction, *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, Milan, Italy.

Chang, R.J., 1991 Analyst's Workbench: A Cliche-Based Programming Paradigm, The University of Texas at Austin, Ph.D. Thesis(In preparation).

Chandrasekaran, B., 1989, Task Structures, Knowledge Acquisition, and Learning, *Machine Learning*, **4**(3,4).

Cointe, P., 1987, Metaclasses are First Class: the ObjVLisp model, *Proceedings of the 1987 ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA-87)*, New Orleans, Louisiana.

des Rivieres, J., 1990, The Secret Tower of CLOS, *Proceedings of the ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, Ottawa, Canada.

Ferber, J., 1989, Computational Reflection in Class Based Object Oriented Languages, *Proceedings of the 1989 ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA-89)*, New Orleans, Louisiana.

Goldberg, A. and Robson, D. 1983 , Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, MA.

Graube, N., 1988, Reflective Architecture: From ObjVLisp to CLOS, *Proceedings of the 1988 European Conference on Object-Oriented Programming (ECOOP-88)*, Oslo, Norway.

Foote, B., and R. Johnson, 1989, Reflective Facilities in Smalltalk-80, *Proceedings of the 1989 ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA-89)*, New Orleans, Louisiana.

Maes, P., 1987a, "Computational Reflection," Ph.D. Thesis, Laboratory for Artifitial Intelligence, Vrije Universiteit Brussels.

Maes, P., 1987b, Concepts and Experiments in Computational Reflection, *Proceedings of the 1987 ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA-87)*, New Orleans, Louisiana.

Marke, K.V., 1988, "The Use and Implementation of the Representation Language KRS", Ph.D. Thesis, Laboratory for Artifitial Intelligence, Vrije Universiteit Brussels.

Steels, L., 1988, "Meaning in Knowledge Representation" in *Meta-Level Architectures and Reflection*, Maes, P., and D. Nardi (ed.), North-Holland, Amsterdam.

Novak, G., 1982a, "The GEV Display Inspector/Editor," HPP-82-32, Heuristic Programming Project, Computer Science Department, Stanford University.

Novak, G., 1982b, GLISP: A High-Level Language for AI Programming, *Proceedings of the Second National Conference on Artificial Intelligence (AAAI-82)*, Pittsburgh, Pennsylvania.

Novak, G., 1983a, "GLISP User's Manual," TR-83-25, Artificial Intelligence Laboratory, The University of Texas at Austin.

Novak, G., 1983b, Knowledge-Based Programming Using Abstract Data Types, *Proceedings of the Third National Conference on Artificial Intelligence (AAAI-83)*, Washington, D.C.

Novak, G., 1983c, GLISP: A Lisp-Based Programming System with Data Abstraction, *The AI Magazine*, Fall, 1983.

Vanwelkenhuysen, J., and P. Rademakers, 1990, Mapping a Knowledge Level Analysis onto a Computational Framework, *Proceedings of the 1990 European Conference on Artificial Intelligence (ECAI-90)*, Stockholm, Sweden.