1991 OOPSLA Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming

Computational Reflection in Executions of Knowledge Level Models

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

ABSTRACT

A new type of object-oriented components, called reflectors, is presented for making knowledge level models executable. From a knowledge level perspective, a problem-solving process is viewed as a modeling activity through explicit representation knowledge level models. Certain useful models are currently identified such as task structures and problem-solving methods. However, the critical issue is how to turn a knowledge level description of an application into a working system. The paper describes how to use a computational reflection scheme for the executions of those higher 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 in research projects, 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 a new type of component, called reflector. It is smoothly integrated with other types of knowledge level components, problem-solving methods and task structures. 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 Components of Expertise

A cliche model is designed to uniformly represent knowledge level components. 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. 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, 83a, 83b]. 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>)
```

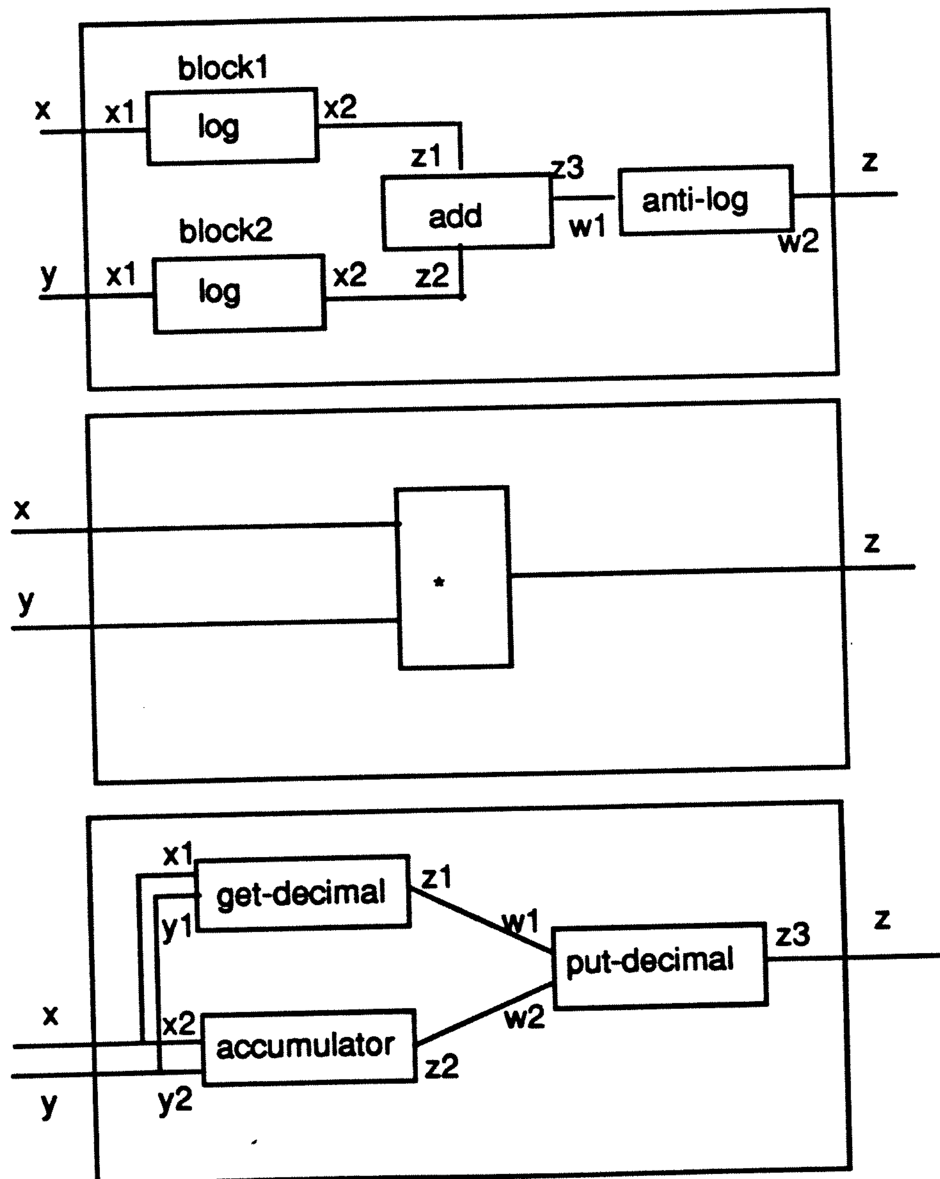
(<*cliche-type specific properties*>)
(<*other properties descriptions*>))

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.

A simple example of the task ***multiplying-two-real-numbers*** is used to illustrate how to instantiate the three types of components.

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-*)))))
```

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.

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.

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.

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) 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.

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. In [Chang, 91], the cliches as the building blocks to construct more complex programs from problem descriptions. The reusable components can be selected from an object-oriented cliche library.

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, 87] 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))))))
```

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 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. Customization of reflectors can be done along the tower. The effects of the reflection influence how the meaning of the cliche is computed.

4. Conclusions

A new type of components, based on computational reflection, is designed for knowledge level modeling. Together with task structures and problem-solving methods, these components are used to model a problem-solving process from a knowledge level perspective. The reflectors are used to make knowledge level models 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 knowledge-level models that could be domain-specific and task-specific and from which they can pick suitable models whenever needed for designing knowledge-based systems.

References

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

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., 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.

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.