Using Polymorphism to Improve Expert Systems Maintainability

John Yen, member, ieee

Hsiao-Lei Juang

Department of Computer Science

Texas A&M University

College Station, TX 77843

(409) 845-5466

Yen@CSSUN.TAMU.EDU

Robert MacGregor

USC / Information Sciences Institute

4676 Admiralty Way,

Marina del Rey, CA 90292

Appeared in *IEEE Expert, Vol. 6, No. 2, pp. 48 - 55, April, 1991.* The research described in this paper was supported by Engineering Excellence Fund at Texas A&M University. Part of the research was conducted at USC/Information Sciences Institute and was supported by DARPA under Contract No. MDA903-87-C-0641. Views and conclusions contained in this

Abstract — One of the major problems in maintaining large rule-based expert system is that the functions performed by rules usually are not well specified, which makes rules difficult to comprehend and modify. Polymorphism in object-oriented programming suggests a promising approach for separating the function of a rule from its implementation details, which can be described by methods. However, there are two major difficulties in integrating methods and rules. First, methods in conventional object-oriented systems can not describe complex conditions regarding their applicability in an expert system. Second, method dispatching does not provide the flexibility of control that is often desirable for an expert system. To alleviate these difficulties, we have generalized methods in two ways. First, the situation about a method's applicability is described by a conjunctive pattern. Second, each generic operation could specify its own control strategy for the selection of methods. To enable specificity-based method dispatching, we have developed an algorithm for computing a well-defined specificity relation among the generalized methods. Based on these enabling technologies, we have developed a production system where the function of each rule is merely invoking a generic function (i.e., sending a message). Furthermore, we use an advanced knowledge representation language for defining classes. Our approach improves the maintainability of expert systems in several important ways. First, it enhances the modularity of rule-based systems because rules and methods can be easily grouped based on their functions. Second, the automatic classification capability of the knowledge representation language assists the user in maintaining a consistent class taxonomy. Finally, it improves the predictability of rules because implicit, heuristic conflict resolution strategies are replaced by explicit control strategies and a principled specificity measure computed by the system.

paper are those of the authors, and should not be interpreted as representing the official opinion or policy of the sponsoring agencies.

I. Introduction

Even though expert system technology has found many successful applications in the industry, the problem of maintaining them has become increasingly difficult due to the increased size of their knowledge bases. For example, Digital Equipment Corporation (DEC) has used a rule-based expert system, XCON, for configuring computer systems. Through the years, XCON has not only grown to a total of 6200 rules, but also required frequent changes to a large portion of its rules. Even though the performance of XCON is satisfactory, the maintenance problem has been such a nightmare that DEC decided to develop a new rule-based language (RIME) for improving the maintainability of XCON[1]. One of the major difficulties in maintaining large rule-based expert system is that the functions and roles performed by rules usually are not well specified, which makes rules difficult to comprehend and modify[1].

Object-oriented programming has been a promising paradigm for the design, development, and the maintenance of large scale software systems. Polymorphism, the ability for different classes of objects to respond to the same set of messages, is one of the key features of object-oriented programming [2]. Polymorphism enables object-oriented systems to separate a generic function from its implementation. Hence, it suggests a promising approach for separating the function of a rule from its implementation details. A rule has two major components: a condition and an action. A rule performs its action when its condition is satisfied by facts in the knowledge base. One way to introduce polymorphism into the rule-based paradigm is to replace the right hand side action of a rule by a generic operation, which states the *function* of the rule's action. Various ways to implement the generic operations in different situations can be described by a set of methods. However, there are two major difficulties in this integration of conventional methods and rules. First, conventional object-oriented systems can not express complex situations under which a method of an expert system is applicable. For instance, it may be desirable to describe the following knowledge as one of the methods that an agent can use to climb up at a location.

```
For an agent to climb up at a location
   If there exists a ladder at the location
   Then 1. empty the agent's hands
        2. ask the agent to climb on to the ladder
```

Since the applicability of methods in object-oriented programming is usually determined by the type of a message's recipient (i.e., the type of the first argument of the method)[1] the situation under which the method above applies can not be described in conventional object-oriented systems. Second, method dispatching does not provide the flexibility of control that is often desirable in an expert system. For instance, the dispatching of methods in object-oriented programming usually selects only one method for responding to a message. However, it may be desirable to select multiple methods in an AI system. For example, to display the status of a car, we may write several methods, each one of them checks a potential problem of the car:

```
To display the status of a car
    If the pressure of a tire is low
    Then display the low-tire-pressure warning

To display the status of a car
    If the brake-fluid level is low
    Then display the low-brake-fluid-level warning
```

It is thus desirable, in this case, to select multiple methods so that the user can be informed about all warning situations.

In this paper, we first describe our approach to address these difficulties based on an implemented production system, CLASP, that integrates methods, production rules, and terminological definitions for classes [3]. Next, we discuss the benefits of our approach regarding the maintainability of expert systems using a simple example. A comparison of our work with previous work in the area of integrating rule-based programming and object-oriented programming follows. Finally, we summarize collateral research issues raised by the work.

[1] CommonLoops and CLOS has an extended notion of methods, where the applicability of a method can be described by the types of all its arguments. Our approach can be viewed as a further generalization of their works.

II. The CLASP Approach

CLASP is an implementation of a high-level AI programming paradigm that integrates an advanced knowledge representation language with production rules and methods in object-oriented programming. Our discussion here will be focused on three areas that are mostly related to the object-oriented programming aspect of the system: (1) using a frame language with a well-defined semantics, which is called *term subsumption language*, for defining classes, (2) methods and their relationships to productions, (3) method dispatching based on a principled specificity measure.

To alleviate the difficulties in introducing polymorphism into rule-based reasoning, we have generalized the notion of methods in two ways. First, the situation about a method's applicability is described by a conjunctive pattern in CLASP. Second, to achieve the flexible control required by expert systems, each generic operation could specify its own control strategy for the selection of methods. More importantly, we have developed an algorithm for computing a well-defined specificity relation among CLASP's methods for performing specificity-based method dispatching.

A. *Using a Term Subsumption Language for Defining Classes*

CLASP offers a term subsumption language (LOOM[4]) for defining characteristics of classes and various relationships between them. *Term subsumption languages* refers to knowledge representation formalisms that employ a formal language, with a formal semantics, for the definition of terms (more commonly referred to as concept or classes), and that deduce whether one term subsumes (is more general than) another [5]. These formalisms generally descend from the ideas presented in KL-ONE [6]. Term subsumption languages are a generalization of both semantic networks and frames because the languages have well-defined semantics, which is often missing from frames and semantic networks.

A term subsumption language enables a knowledge engineer to specify defining characteristics of *concepts* and *relations*. Generally speaking, a concept represents a class of objects, and a relation describes a relationship between two concepts. For instance, the class of male can be represented by a concept `Male` and the relationship between parents and their children can be represented as a

`Child` relation. Because the notion of relation is closely related to the notion of *slots* in frame-based systems, we will use the two terms interchangeably. A concept/relation whose definition can not be fully described is called a *primitive concept/relation*, otherwise it is called a *defined concpet/relation*. Most term subsumption languages allow a concept to be defined in several ways: (1) forming a conjunction of several superconcepts using `:and` constructs, (2) restricting the types of a slot value using `:all` construct, and (3) restricting the cardinality of a slot value using `:atmost` and `:atleast` constructs. This can be illustrated by the following example. Suppose **Successful-father** is defined as a father whose children are all college graduates. This can be expressed as follows using a term subsumption language.

```
(defconcept Successful-Father (:and Father (:all Child College-Graduate)))
```

<primitive-concept-definition> ::= (defconcept C :is (:and $C_1 ... C_n$:primitive))

<defined-concept-definition> ::= (defconcept C :is (:and <concept-forming-expr>$^+$))

<concept-forming-expr> ::= C | (:all R C) | (:atleast k R) | (:atmost k R)

<primitive-relation-definition> ::= (defrelation R :primitive [(:domain C)] [(:range C)])

<defined-relation-definition> ::= (defrelation R :is (:and <relation-forming-expr>$^+$))

<relation-forming-expr> ::= R | (:domain C) | (:range C)

Figure 1: A Partial Description of LOOM's Grammar

Since we use the syntax of LOOM knowledge representation system to define concepts and relations in this paper, we list the relevant part of LOOM's syntax in Figure 1 where k is a non-negative integer, C and R denotes a concept name and a relation name respectively. A further discussion about the LOOM system can be found in [4].

The major strength of term subsumption systems is their reasoning capabilities offered by a *classifier*. The classifier is a special purpose reasoner that automatically infers and maintains a consistent and accurate taxonomic lattice of logical subsumption relations between objects in the knowledge base. Two major advantages of using a term subsumption language to define classes are (1) the consistency of the class taxonomy is improved by the classifier, (2) the classifier enables the system to automatically

infer the membership of objects that can be deduced from their descriptions. We will use the following example to illustrate these benefits. For example, suppose we define an `empty-handed-monkey` as a monkey that is not holding any objects:

```
(defconcept empty-handed-monkey :is (:and monkey (:at-most 0 hold)))
(defrelation hold :is :primitive (:domain animal) (:range object))
```

Suppose we inform the system about a monkey whose hands are empty. The classifier will successfully infer that the monkey is an instance of `empty-handed-monkey`, even though the user did not mention so explicitly. Moreover, suppose the following new concept `empty-handed-animal` has been introduced into the system.

```
(defconcept empty-handed-animal :is (:and animal (:at-most 0 hold)))
```

The classifier will now infer that `empty-handed-monkey` is a subclass of `empty-handed-animal`. Hence, using a term subsumption language to define classes enables the classifier to help system builders in developing and maintaining a consistent class taxonomy.

B. Operators, Methods, and Rules

An *operator* in CLASP is a generic function that can be invoked by procedure calls or by production rules. In fact, the action of a production rule is always performing a specific operation. For instance, a rule that causes the status of a car to be displayed whenever its driver's door is open can be described using a `Display-Status` operator:

```
(defrule Trigger-Display-Status
    :when (:and (Car ?c)
                (Has-driver-door ?c ?d)
                (Open ?d))
    :perform (Display-Status ?c))
```

Hence, the function and the role of a rule is captured explicitly by the operator to which the rule's action refers, and is separated from the implementation details of the operator, which are described by *methods*. In addition to being invoked by rules, operators can be invoked procedurally through the function `perform`. For instance, the function call `(perform (Display-Status chevy-1))` displays the status of the car instance `chevy-1`.

In addition to the associated operator, a CLASP's method has two additional components: a situation part and an action part. The situation part describes a conjunctive condition under which the method applies. In addition to testing types and relations of the method's arguments, the condition could also introduce free variables. For instance, the situations of the methods for `Display-Status` operator may include additional free variables that refer to the trake fluid level, the tire pressure, and other measures related to the safety condition of an automobile. A method is *applicable* to an instantiation of its operator if its situation condition is satisfied. The action part is the procedural body of the method. Because the action part of a method can invoke other operators, methods can be used to decompose a high-level task into lower-level subtasks.

Each operator in CLASP specifies its own control strategy for the selection of methods using a sequence of *filters*[2]. The set of applicable methods passes through each filter in the sequence. Each filter removes some methods from the candidate set. The final set of methods generated by the last filter is executed. Some examples of filters are `:most-specific`, `:select-one`, `:select-all`, `:preferences`, and `:last-one`. The default filter sequence is `(:preferences :most-specific :last-one)` . The `:preference` filter eliminates methods which have been declared (in explicit preference statements) to be less preferable than some other candidate method. The `:most-specific` filter eliminates any method whose pattern is specialized by some other candidate method's pattern. The `:last-one` filter chooses the method which was most-recently defined. The `:select-all` filter chooses all the remaining candidate methods, and the `:select-one` filter randomly chooses one method from the candidates. This filter mechanism offers the flexibility in method dispatching that is useful for many AI applications. For instance, the operator `Display-Status` could use the filter sequence `(:most-specific :select-all)` for selecting all methods that display messages about warning situations such as low tire pressure, low

[2]The PRISM system, developed by Langley and Ohlsson at UC Irvine, employs a similar notion of filtering within a rule-based setting.

brake fluid, low power in battery, etc. The filter sequence of an operator is specified as part of the operator's declaration, e.g.,

```
(defoperator Display-Status :filters (:most-specific :select-all) ).
```

C. Retrieving Applicable Methods

Because the situation condition and the action of a method may consist of free variables that are not in the argument list, applicable methods in CLASP are retrieved using a RETE-style pattern matcher (CONCRETE). CONCRETE is integrated with LOOM's classifier for performing pattern matching based on the semantics of classes [3]. We will illustrate this using the method M1 in Figure 4. Suppose the system has been told about the following facts:

```
John is a Successful-father
Angela is a child of John
Angela has a car named Corolla-1
```

These facts do not directly match the condition of the method; however, they do match M1 (with the variable bindings ?x = John, ?y = Angela) if we also consider the semantics of classes `Successful-father` and `Car-owner`. CONCRETE is able to match the facts with M1 because it is informed about the following deduced facts by LOOM's classifier:

```
John is a Person
Angela is a College-graduate and a Car-owner
```

To retrieve applicable methods, CLASP compiles the situation condition of a method as if the operator and its arguments are part of the condition. For this purpose, a special RETE node (called *the operator node*) is created for each operator. For instance, the method M1 is compiled into a RETE network corresponding to the following pattern.

```
(:and (op ?x)
```

```
(Person ?x)
(College-graduage&Car-owner ?y)
(Child ?x ?y))
```

The RETE nodes corresponding to the situation pattern are updated, as usual, when the database is changed. When an operator is invoked, CLASP sends a token, which includes the arguments of the operator invocation, to the corresponding operator node. This causes other nodes to be updated. Eventually, the terminal nodes of applicable methods will generate method instantiations, which are collected into a list of method instantiations applicable to the operator invocation.

The major difference between retrieving methods and matching rules lie in the way instantiations are generated. Methods retrieval is always initiated by invoking operators; while rule matching is initiated by changes in the database. Even though changes in the database update RETE networks corresponding to a method's situation, they will never generate method instantiations because operator nodes do not have a memory (i.e., they do not store any previous operator instantiations).

D. Computing the Specificity of Methods

Specificity is one of the most important criteria for selecting methods in conventional object-oriented systems. It provides a convenient way for a system to describe general methods that can be applied to many situations (e.g., multiple subclasses of a class) as well as specific methods for handling exceptional cases. A class taxonomy in an object-oriented system often serves as a specificity lattice for methods. Hence, computing the specificity of methods was rarely an issue in object-oriented programming [3]. Computing the specificity of CLASP's methods is complicated for two reasons. First, the situation part of a CLASP method is a conjunctive pattern with extra free variables. Second, classes in CLASP capture a semantics that is richer than that of object-oriented systems (see Section A). As a result, a situation pattern may contain many implicit conditions that can be deduced from the semantics of classes. We will first define a principled specificity relation among CLASP's methods. Then, we outline an implemented algorithm for determining whether a method is more specific than another.

[3] Systems with an extended notion of methods (e.g., CommonLoops and CLOS) have a slightly more sophisticated way for computing methods' specificity

A method m_2 is *more specific than* another method m_1 if and only if (1) they are associated with the same operator, and (2) for any invocation of the operator, if m_2 is applicable then m_1 is also applicable. Our approach for testing the specificity of methods is based on a theorem in [7] that states the sufficient and necessary condition for a conjunctive pattern (i.e., a conjuction of non-negated literal) to be more specific than another one:

> Suppose p_2 and p_1 are two conjunctive patterns, p_2 is more specific than p_1 if and only if there exists a substitution that replaces variables in p_1 by variables or constants in p_2 such that p_2 implies p_1 under the substitution.

The substitution is equivalent to a *mapping* that maps each variable in p_1's condition to a variable or a constant in p_2's condition. Based on the theorem, we have shown that an algorithm for testing the specificity of rules needs to search for a desired mapping between variables of two rules [7]. The specificity test between methods introduce one additional constraint to the mapping: the arguments of one method has to map to corresponding arguments in another method. Intuitively, it is easy to see that the existence of such a mapping is a sufficient condition that m_2 is more specific than m_1 because, for any instantiation of m_2, we can construct an instantiation of m_1 from the mapping. Thus, m_1 is applicable for an invocation of its operator whenever m_2 is applicable. For instance, consider methods M1 and M2 in Figure 3. Suppose `Successful-father` is defined as a father whose children are college graduates, and `Car-owner` is a person who owns at least one car (see Figure 2). CLASP will be able to determine that M2 is actually more specific than M1 because M2's situation implies M1's situation by replacing variables ?x and ?y by ?z and ?w respectively.

Since the number of all possible mappings between variables of two methods is an exponential function of the number of free variables in the methods, an efficient algorithm for the specificity test has to reduce the number of mappings it needs to consider using additional information. This is achieved in CLASP through several steps. First, implicit conditions logically implied by the semantics of classes are made explicit. We will refer to this process as the *normalization* of methods. Second, arguments of one method is constrained to map to corresponding arguments of another method. Third, CLASP uses subsumption links between classes and relations to establish further constraints on the mapping of a method's variables.

Finally, CLASP performs a dependency-directed backtracking to search for a mapping that satisfies all the constraints. We briefly describe each step below. A more detailed discussion about steps 1, 2 and 4, which are also used for testing the specificity of rules, can be found in [7].

```
(defconcept Person (:primitive))
(defconcept Male (:and Person :primitive))
(defconcept Female (:and Person :primitive))
(defconcept College-graduate (:and Person :primitive))
(defconcept Female-College-graduate (:and Female College-graduate))
(defrelation Child (:and :primitive (:domain Person) (:range Person)))
(defrelation Daughter (:and Child (:range Female)))
(defconcept Father (:and Male (:at-least 1 Child )))
(defconcept Successful-Father (:and Father (:all Child College-graduate)))
(defrelation Has-car (:and :primitive (:domain Person) (:range Vehicle)))
(defconcept Car-owner (:and Person (:at-least 1 Has-car)))
```

Figure 2: An Example of Class and Relation Definitions

As mentioned earlier, the rationale behind normalizing methods is to reduce the search space of possible mappings. Without the normalization process, the search for a mapping would have to consider the possibility that a condition in a method's situation is implied by a conjunctive subpattern of another method's situation. For example, consider methods **M1** and **M2** in Figure 3. The condition `(College-graduate ?y)` in M1 is implied by the subpattern `(Successful-Father ?z)` $\wedge$ `(Daughter ?z ?w)` of M2's situation by replacing ?y in M1 by ?w in M2. Having deduced the conditions implied by these conjunctive subpatterns during the normalization process, the specificity test only needs to consider pairs of conditions with the same number of arguments. Figure 4 shows the situation-sides of M1 and M2 after they have been normalized where the class `College-graduate&Car-owner` is the intersection of `College-graduate` and `Car-owner`. It is easier to see that M2 is actually more specific than M1, which was not obvious prior to normalization.

The second step ensures that the arguments of a method are mapped to the corresponding arguments

```
(defmethod op( ?x )
   :title ``M1''
   :situation (:and (College-graduate ?y)
                    (Child ?x ?y)
                    (Car-Owner ?y))
   :action     ... )
(defmethod op( ?z )
   :title ``M2''
   :situation (:and (Successful-Father ?z)
                    (Daughter ?z ?w)
                    (Has-Car ?w ?c))
   :action    ... )
```

Figure 3: An example of two methods before normalization

of another method. For instance, the argument ?x in M1 is first mapped to the argument ?z in M2. The third step attempts to reduce the search space of possible mapping by considering the subsumption relationship between concepts and relations (i.e., unary and binary predicates). Normally, the situation pattern of a method consists of several different predicates, only a small percentage of which are subsumed by a predicate in another method's situation. Using the subsumption relationships between predicates, which are precomputed by LOOM's classifier when classes and relations are defined, we can reduce the candidates a variable in M1 can map to, which in turn prunes the search space for finding a mapping. For instance, `Daughter` is the only predicate in M2 that is subsumed by `Child` in M1. We can infer that any mapping that proves M1 subsumes M2 has to map variable `?x` to `?z`, variable `?y` to `?w` because this is the only way that condition `(Child ?x ?y)` can be implied by M2's situation condition. The last step uses a dependency-directed backtrack to search a mapping that satisfies the constraints generated by the second and the third step.

```
(defmethod op( ?x )
   :title ``M1''
   :situation (:and (Person ?x)
                    (Child ?x ?y)
                    (College-graduate&Car-Owner ?y))
   :action    ... )
(defmethod op( ?z )
   :title ``M2''
   :situation (:and (Successful-Father ?z)
                    (Daughter ?z ?w)
                    (Female-College-graduate&Car-Owner ?w)
                    (Has-Car ?w ?c))
   :action   ... )
```

Figure 4: Two methods after normalization

III. An Example

We will use a simple problem, the monkey-bananas problem in [8], to illustrate how the integration of object-oriented programming and rule-based reasoning in CLASP improves the maintainability of expert systems. The monkey-banana problem is to write a set of rules such that a monkey in a room will follow a sequence of commands in order to grab a bunch of bananas on the ceiling. Objects in the room include a ladder, a sofa, and other furnitures (e.g., a shelf). Figure 5 contains a partial listing of rules for solving the problem. Figure 6 shows an OPS5 implementation of these rules. A major function of these three rules is to help the monkey to climb up at the location where the bananas are hung from the ceiling. However, this function is implicit in the rules, and is difficult for the programmer to capture, even for this simple application. One way to capture the functions of rules and to separate them from various ways to implement the function is to define `climb-up` as an operator. Various ways to climb up in different situations (e.g., whether there exists a ladder at the location or not) may be described by

several methods. This can be implemented in CLASP as shown in Figure 7. A partial LOOM knowledge base for the monkey-banana problem is described in Figure 8. The following sections use this example to illustrate three major benefits of our approach regarding the maintainability of expert systems: (1) improving the modularity and the reusability of the rule base, (2) supporting the development of a more consistent and homogeneous knowledge base, and (3) enhancing the predictability of rules.

```
Rule 1:
     If the goal is to grab an object on ceiling then
        create a goal to move a ladder to the location of the object.
Rule 2:
     If the goal is to grab an object on ceiling and
        the ladder is already at the location of the object then
            create a goal to get on the ladder.
Rule 3:
     If the goal is to grab an object on ceiling and
        the ladder is already at the location of the object and
        a monkey is already on the ladder then
            create a goal for the monkey to empty its hands.
```

Figure 5: Several Rules for the Monkey-bananas Problem

A. Improving the Organization and the Reusability of Rules

It is very difficult to locate relevant pieces of knowledge for modification unless the knowledge base is well-organized. In OPS5-like systems, users often use *context elements* to cluster rules[8]. CLASP's rule base is more modular than that of conventional rule-based systems for two major reasons. First, the function of a production rule is explicitly represented by an operator. Second, operator-triggering rules are separated from operator-implementation rules. Hence, methods that are intended to achieve the same function can be grouped together and form a natural "functional module". Using the monkey-

```
(p monkey-banana-1
   (goal status active type holds object <w>)
   (object ^name <w> ^at <p> ^on ceiling)
-->
   (make goal ^status active ^type move ^object ^ladder ^to <p>))

(p monkey-banana-2
   (goal ^status active ^type holds ^object <w>)
   (object ^name <w> ^at <p> ^on ceiling)
   (object ^name ladder ^at <p>)
-->
   (make goal ^status active ^type on ^object ladder))

(p monkey-banana-3
   (goal ^status active ^type holds ^object <w>)
   (object ^name <w> ^at <p> ^on ceiling)
   (object ^name ladder ^at <p>)
   (monkey ^on ladder)
-->
   (make goal ^status active ^type holds ^object nil))
```

Figure 6: A Partial OPS5 Implementation for the Monkey-bananas Problem

bananas example, suppose the programmer wants to extend the system to consider the case that there is a shelf in the room and the bananas are on the shelf. The programmer can easily find out the relevant operators (i.e., **achieve-hold-goal**) and its associated methods (i.e., **hold-obj-on-ceiling-method**). All he/she has to do is to add the following method for the **achieve-hold-goal** operator:

```
(defmethod achieve-hold-goal (?g)
```

```
(defconcept Real-World-Object :is :p)
(defconcept Ladder :is (:and :p Real-World-Object))
(defrelation At :is (:and :p (:domain Real-World-Object) (:range Location)))
(defconcept Location :is :p)
(defrelation On :is (:and :p (:domain Real-World-Object)
                             (:range Real-World-Object)))
;;; Concept and Relations related to MONKEY
(defconcept Animal :is (:and :p Real-World-Object))
(defconcept Monkey :is (:and :p Animal))
(defrelation holds :is (:and :p (:domain Monkey)
                                (:range Real-World-Object)))
(defconcept Empty-handed-animal :is (:and Animal (:at-most 0 holds) ) )
;;; Goal-related concepts and relations
(defconcept goal-status :is (:the-set active achieved failed))
(defconcept goal-type :is (:the-set hold on ))
(defconcept Goal :is (:and :p (:at-most 1 status) (:at-most 1 type)))
(defrelation status :domain goal :range goal-status)
(defrelation type :domain goal :range goal-type)
(defrelation agent :domain goal :range animal)
(defrelation object :domain goal :range real-world-object)
(defconcept Active-Hold-Goal :is (:and Goal
                                       (:all status (:the-set active))
                                       (:all type (:the-set hold))))
```

Figure 7: A Partial LOOM Knowledge Base for the Monkey-bananas Problem

```
(defrule mab-trigger
   :when (:and (Active-Hold-Goal ?g) )
   :perform (achieve-hold-goal ?g))
(defmethod achieve-hold-goal (?g) :title "hold-obj-on-ceiling-method"
   :situation (:and (agent ?g ?agent)
                    (object ?g ?o)
                    (on ?o ceiling)
                    (at ?o ?l))
   :action ( (perform (climb-up ?agent ?l))
             (perform (grab ?agent ?o))
             (perform (announce-goal-achieved ?g)))
(defmethod climb-up (?agent ?location) :title "climb up using an existing ladder"
   :situation (:and (Ladder ?ladder)
                    (At ?ladder ?location) )
   :action ( (perform (empty-hand ?agent))
             (perform (climb-on-to ?agent ?ladder)) ))
(defmethod climb-up (?agent ?location)
   :title "climb up by carrying a ladder to the location"
   :situation (:and (Ladder ?obj)
                    (:NOT-TRUE (At ?obj ?location) )
                    (At ?obj ?ladder-location) )
   :action ( (perform (empty-hand ?agent))
             (perform (move-self-to ?agent ?ladder-location))
             (perform (carry-to ?agent ?obj ?location))
             (perform (empty-hand ?agent))
             (perform (climb-on-to ?agent ?obj)) ))
```

Figure 8: A Partial CLASP Implementation of the Monkey-bananas Problem

```
   :title "hold-obj-on-shelf"
   :situation (:and (agent ?g ?agent)
                    (object ?g ?o)
                    (on ?o shelf)
                    (at ?o ?l)
   :action ( (perform (move-self-to ?agent ?l))
             (perform (empty-hand ?agent))
             (perform (climb-on-to ?agent shelf))
             (perform (grab ?agent ?o))
             (perform (announce-goal-achieved ?g)))
```

For an OPS5 implementation, the programmer needs to manually search for relevant productions. For example, the following two rules can be created by copying and editing rules `monkey-banana-1` and `monkey-banana-3` in Figure 6.

```
(p monkey-banana-4
  (goal ^status active ^type holds ^object <w>)
  (object ^name <w> ^at <p> ^on shelf)
-->
  (make goal ^status active ^type move ^object monkey ^to <p>))

(p monkey-banana-5
  (goal ^status active ^type holds ^object <w>)
  (object ^name <w> ^at <p> ^on shelf)
  (monkey ^at <p>)
-->
  (make goal ^status active ^type holds ^object nil))
```

The first rule moves the monkey to the location of the object, and the second rule empties the mon-

key's hands so that it can climb on to the shelf. In fact, more than these two rules need to be added for handling the situation that the target object may be on a shelf, rather than on the ceiling. In contrast, the CLASP implementation of the problem only needs to add one method to deal with this extension (i.e., the hold-obj-on-shelf method in page A). Moreover, the method is able to reuse many existing operators and methods without any further modification (e.g., `move-self-to`, `empty-hand`, `climb-on-to`, `grab`, and `announce-goal-achieved`). Actually, the newly added method does not need to introduce any new operators or methods at all. The reusability of OPS rules is low because the condition that triggers an operation is entirely mixed with the condition that determines the implementation of the operation. By separating these two kinds of conditions, CLASP is able to reuse many of its operator-implementation knowledge when its operator-triggering situation has been extended or modified.

The example above demonstrates two important benefits of CLASP's approach to integrating object-oriented programming and rule-based programming paradigms. Using operators to explicitly state the function of rules, CLASP is able to organize the knowledge base such that related rules and methods can be easily located and modified. By separating the functions of rules from the implementation of those functions, CLASP significantly improves the reusability of its knowledge.

B. Supporting the Development of Consistent and Homogeneous Knowledge Bases

When a programmer wants to augment an existing body of code in a software maintenance task, the problem that he/she sometimes encounters is in knowing how and where to add the augmentation so as not to disturb the rest of the code[1]. Maintaining the consistency and homogeneity of knowledge therefore is very important when modifying the system. As we have discussed in Section A, the automatic classification capability of CLASP's knowledge representation language (LOOM) assists the user in maintaining a consistent class taxonomy. If the definition of a class is inconsistent, the system is able to detect inconsistency and inform the knowledge engineer about possible causes of the inconsistency. Furthermore, different types of knowledge are represented by different forms in CLASP. In particular, methods are used exclusively for describing how-to knowledge, while rules are used merely for expressing

when to trigger operations. This improves the consistency and homogeneity of the knowledge base because it facilitates similar representations for similar types of knowledge.

C. Enhancing the Predictability of Rules

One of the major difficulties in maintaining rule-based systems lies in the unpredictability of the rule firings. This is because the execution sequence of rules is implicitly controlled by a complicated conflict resolution strategy. Selection of rules and methods in CLASP is based on (1) a well-defined specificity measure and (2) control knowledge explicitly expressed using filters of an operator. The former allows the system programmer to verify the anticipated specificity relationships and detect unexpected ones during rule compilation time. The latter makes explicit the selection criteria used in method dispatching. Together, they improve the predictability of the system's behavior. For example, suppose we want to extend the monkey-bananas application for dealing with the case where the object to be grabbed is fixed to the ceiling (e.g. a fan), and thus the monkey can not grab the object from the ceiling. To do this, we can add the following method to CLASP's knowledge base:

```
(defmethod achieve-hold-goal (?g)
 :title ''goal-cannot-be-achieved''
 :situation (:and  (agent ?g ?agent)
                   (object ?g ?o)
                   (fixed-on ?o ceiling)
                   (at ?o ?l))
 :action (perform (announce-goal-failure ?g)))
```

where the relation **fixed-on** is defined as a specialization of **on** relation :

```
(defrelation Fixed-On :is (:and On :primitive))
```

CLASP will infer that the new method is more specific than the method "hold-obj-on-ceiling-method." Since the operator **achieve-hold-goal** uses the default filter sequence, which includes the **:most-specific** filter, the programmer can predict that the new method will be executed when the object

to be grabbed is fixed to the ceiling. In the case there are multiple most specific methods, the programmer can change the filter sequence of **achieve-hold-goal** operator to `(:most-specific :select-all)` to ensure the firing of the newly added method. For the OPS5's implementation, the programmer may add the following production for dealing with the situation that the object is fixed to the ceiling.

```
(p monkey-banana-1-fail
  (goal status active type holds object <w>)
  (object ^name <w> ^at <p> ^on ceiling ^fixed t)
-->
  (make goal ^status fail ^type))
```

However, the system does not provide any feedback to help the programmer in predicting the firing sequence of rules. In fact, even though this newly added production is semantically more specific than production **monkey-banana-1**, it will not be recognized as so because specificity in OPS5 is determined by syntactic information (i.e., the number of condition elements). As a result, it is very difficult to predict which one of the two rules will be fired when they both match. In CLASP, even if the programmer codes the new method incorrectly such that it is not more specific than the original method **hold-obj-on-ceiling-method**, he/she could detect the error during comiplation time by browsing the subsumption lattice of methods. In summary, CLASP's approach improves the predictability of rules by using explicit control knowledge and by providing feedback about the subsumption relationships between methods to the programmers.

IV. Related Work

Loops[9] is one of the earliest efforts in integrating rule-based paradigm and object-oriented programming. Rules in Loops are grouped into RuleSets, which can be invoked by message sending. The information for controlling the firing of rules within a RuleSet is also explicitly specified. Rules in KEE can also be grouped into rule classes, which can be invoked by methods or demons [10]. Therefore, rules in Loops and KEE could play a role similar to that of CLASP's methods. Our approach differs, however,

from their approaches in three important ways. First, the right hand side action of a rule is strictly invoking a generic function. Second, classes are defined using a term subsumption language. Third, method dispatching is based on a principled specificity measure computed by the system during rule compilation time. The priority of rules in Loops is determined by their ordering. Thus, it is the system developer's responsibility to manually place specific rules before more general ones. CLASP's approach relieves this burden from the system developer, and enhances the maintainability of expert systems.

V. Current Status and Future Work

CLASP has been implemented using CLOS and Common Lisp. Although it was originally developed in Lisp Machines (i.e., TI Explorers and Symbolics machines), it can be easily ported to general purpose workstations (e.g., SUN workstations). Based on our preliminary experiments using the monkey-banana problem, the run time performance of CLASP seems acceptable. More specifically, for the monkey-banana application, the system takes 20.6 seconds to compile and create 10 concepts, 10 relations, 2 rules, and 14 methods. However, it takes only 3.9 seconds (including the time spent for creating instances) to solve the monkey banana problem. Thus, most of the performance overhead is introduced during compile time rather than run time.

From object-oriented programming point of view, the major limitation of CLASP is a lact of support for encapsulation. Properties about objects in CLASP are globally accessible and modifiable through LOOM's retrieval and tell facilities. Thus, there is no way to hide information about an instance from other objects. Our future research includes incorporating encapsulation into CLASP's framework, and empirical evaluation of the maintainability of medium to large size expert systems developed using our approach. We are currently reimplementing part of R1 (R1-Soar[11]) for a further assessment on the maintainability of expert systems developed using CLASP.

VI. Summary

In this paper, we have described an approach to address the maintenance issue of large expert systems by incorporating the notion of polymorphism in object-oriented programming into the rule-based programming paradigm. To enable the knowledge engineer to describe a complex situation under which a method applies, we have generalized a method's applicable situation to a conjunctive condition that may contain free variables. To obtain flexibility of control that is often desirable in an expert system, we have generalized method dispatching to allow each generic function to specify its method selection criteria using a set of filters. To enable specificity-based method dispatching, we have developed an algorithm for computing a principled specificity relation among methods.

Based on a monkey-bananas example, we have demonstrated that our approach offers several important benefits regarding the maintainability of expert systems. First, it enhances the modularity of rule-based systems because rules and methods can be easily grouped based on their functions. It also increases the reusability of how-to knowledge because they are separated from rules. Second, the automatic classification capability of the term subsumption language assists the user in maintaining a consistent class taxonomy. Finally, it improves the predictability of rules because implicit, heuristic conflict resolution strategies are replaced by explicit control strategy and a principled specificity measure computed by the system.

VII. Acknowledgements

We wish to thank the referees for their comments on an earlier draft of the paper.

References

[1] E. Soloway, J. Bachant, and K. Jensen, "Assessing the maintainability of xcon-in-rime: Coping with the problems of a very large rule-base," In *Proceedings of AAAI-87*, pp. 824–829, Seattle, Washington, August 1987.

[2] G. S. Blair, J. J. Gallagher, and J. Malik, "Genericity vs inheritance vs delegation vs conformance vs ...," *Journal of Object-Oriented Programming*, vol. 2, no. 3, pp. 11–17, September/October 1989.

[3] J. Yen, R. Neches, and R. MacGregor, "CLASP: Integrating term subsumption systems and production systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, no. 1, , March 1991.

[4] R. M. MacGregor, "A deductive pattern matcher," In *Proceedings of AAAI-88*, 1988.

[5] P. F. Patel-Schneider, B. Owsnicki-Klewe, A. Kobsa, N. Guarino, R. MacGregor, W. S. Mark, D. McGuinness, B. Nebel, A. Schmiedel, and J. Yen, "Term subsumption languages in knowledge representation," *AI Magazine*, vol. 11, no. 2, pp. 16–23, 1990.

[6] R. Brachman and J. Schmolze, "An overview of the KL-ONE knowledge representation system," *Cognitive Science*, vol. 9, no. 2, pp. 171–216, August 1985.

[7] J. Yen, "A principled approach to reasoning about the specificity of rules," In *Proc. National Conf. on Artificial Intelligence*, pp. 701–707, Boston, August 1990.

[8] L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, 1985.

[9] M. Stefik and D. G. Bobrow, "Object-oriented programming: Themes and variations," *AI Magazine*, vol. 6, no. 4, pp. 40–62, 1986.

[10] R. Fikes and T. Kehler, "The role of frame-based representation in reasoning," *Communication of the ACM*, vol. 28, no. 9, , September 1985.

[11] P. S. Rosenbloom, J. E. Laird, J. McDermott, A. Newell, and E. Orciuch, "R1-soar: An experiment in knowledge-intensive programming in a problem-solving architecture," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, no. 5, pp. 561–569, September 1985.