A technique for customizing object-oriented knowledge representation systems, with an application to network problem management

Lisiane Goffaux and **Robert Mathonet** *
CIG-Intersys Systems
7, avenue Lloyd George - 1050 Brussels - Belgium

Abstract

Over the last few years, object-oriented techniques have gained an increasing recognition both in software engineering and in AI. Object-oriented systems present undisputable advantages and provide features that make them really suitable to represent knowledge. However, our practical experience in using these techniques for knowledge representation led us to discover that object-oriented systems also present serious drawbacks, essentially due to their lack of expressive power. These drawbacks can really be felt when modelling domains characterized by a wide variety of knowledge. This paper introduces the notion of Representation Cluster, which allows to provide any object-oriented system with customizable knowledge representation formalisms. These formalisms enable to express and handle diverse kinds of knowledge using only their natural high-level properties. Such an approach speeds up knowledge bases development, makes them clearer, natural, and avoids a great deal of code writing. Its advantages are illustrated on two realistic examples extracted from the knowledge representation system of an expert system shell dedicated to real-time network troubleshooting.

1 Introduction

As a tool for representing knowledge, object-oriented techniques present a well known interest. They are particularly suited to express knowledge in systems that are inherently model-based.

That is why we intensively used object-oriented techniques to model the knowledge in our Dantes network troubleshooting expert system. (For a description of Dantes, see [Mathonet *et al.*, 1987]). We learnt, the "hard way", that a growing knowledge base can become very difficult to understand and to maintain. This led us to identify some drawbacks in the use of object-oriented techniques for representing knowledge; they are explained in section 2, after a brief overview of the principles of object-oriented techniques in the contexts of programming languages and knowledge representation.

* This research was supported by IRSIA/IWONL.

To overcome the lack of expressive power of the initial Dantes object-oriented system, we enhanced it with the notion of *representation cluster*. This concept is perfectly general and can be applied to any object-oriented system. Its definition can be found in section 3, together with the description of an enhanced object-oriented system (called DOOS).

For illustration purposes, section 4 is devoted to two realistic uses of the representation cluster concept in the domain of computer network troubleshooting. Finally, section 5 stresses the gain of expressive power brought by representation clusters to object-oriented systems.

2 Object-oriented systems

This section discusses the advantages and disadvantages of object-oriented systems. As the term *object-oriented system* can have slightly different meanings depending on the context where it is used, we first specify the object-oriented systems considered in this paper. For a complete discussion, see [Stefik and Bobrow, 1986].

2.1 Object-oriented programming languages

Object-oriented programming is based on the concepts of object, class, inheritance and message passing. An object is "an entity that combines the properties of data and procedure" [Stefik and Bobrow, 1986]: an object has a local memory (variables) and procedures (called methods) attached to it. The interaction between objects is done via message passing. An object responds to a message by executing one of its methods. Objects are organized into classes, i.e. in sets of objects sharing the same properties and the same behaviour. The objects belonging to a class are called the class instances. Classes can be organized in an inheritance lattice. Some object-oriented programming languages also provide metaclasses. A metaclass is a class of classes. As stressed in [Danforth and Tomlinson, 1988], the primary justification to introduce such a feature is the desire to treat everything as an object.

Compared with more traditional approaches, object-oriented programming languages have several advantages, mainly due to a clean data and procedure organization [Stefik and Bobrow, 1986]. They support data abstraction, increase the modularity of programs, and, due to property inheritance, avoid redundant declarations or specifications. Therefore, object-oriented programming languages encourage the development of better organized, more easily modifiable, and thus more robust software.

2.2 Object-oriented concepts and knowledge representation

2.2.1 Object-oriented systems for knowledge representation

In AI, object-oriented concepts have been used for knowledge representation purposes. Object-oriented representation systems can be related to knowledge representation systems based on the frame idea [Minsky, 1975], such as KRL [Bobrow and Winograd, 1977] or KEE [Fikes and Kehler, 1985]. The term *frame-based systems* designates a large family of systems which are all based on the same fundamental principle: they organize knowledge into structures that group both data and procedures; these structures are related by some inheritance mechanism. The similarity with the object-oriented concepts in programming is obvious.

In the rest of this paper, *object-oriented system* refers to a system relying on the basic principles of object-oriented programming and possibly providing specific features for knowledge representation, such as viewpoints [Bobrow and Winograd, 1977] or variable annotations. Object-oriented systems enable the treatment of some common sense reasoning aspects and especially of exceptions and defaults. They provide general and flexible techniques that are ideally suited to express knowledge in model-based systems.

2.2.2 Limitations

Object-oriented systems offer a relatively low level of abstraction and represent knowledge in a uniform way.

Low level of abstraction means that there is an important semantic gap between knowledge as perceived in the real world and its representation. Knowledge is complex in essence: it generally involves subtle and rich concepts mixed together or interlinked by a large amount of various relationships. This contrasts with the simplicity of the features provided by object-oriented systems: object, class lattice, inheritance... As a consequence, the representation of any knowledge requires the decomposition of this knowledge into primitive and simplified elements which fit into the framework imposed by the object-oriented system. As knowledge does not directly match the features provided by the object-oriented system, the knowledge base (KB) developer must force the matching, often by adding procedural code to the knowledge representation. For example, suppose that some device D is represented by an object O. Expressing that an instance variable of O represents a relationship between D and other devices must be done by providing the object O with procedures which handle the constraints and treatments required by the interpretation of the variable as a relationship. In real-life situations, the resulting knowledge bases often consist of a few class definitions and a large amount of procedural code where recognizing the represented knowledge is very difficult. Representing knowledge becomes a very difficult programming task that requires a lot of time and effort and a very sound methodology.

Uniformity of representation means that every concept in any knowledge is directly represented using the *same* basic techniques. Still, knowledge is not uniform but is by nature very different from one domain to another: just consider medicine, oil drilling or plant control. Futhermore, within a given domain, knowledge can also present broad differences. Except for very simple domains, it clearly and naturally divides into various *kinds of knowledge*. Each of these kinds of knowledge is characterized by *intrinsic properties* of its own. These intrinsic properties depend of course on the knowledge itself but also on the treatments to be performed by the knowledge-based system. However, within an object-oriented system, all the different kinds of knowledge must be expressed by means of class lattice, variables, methods..., whatever the differences between them.

These drawbacks can make an object-oriented knowledge base unnatural and very difficult both to understand and to maintain. Moreover, these problems do not facilitate the concurrent development of a knowledge base by several persons. As each developer can have his own ideas about the best representation strategy, the KB risks to be totally unreadable.

2.2.3 Example

To illustrate the notions of kind of knowledge and of intrinsic property, let us consider the Dantes application domain: network troubleshooting. Network troubleshooting involves, at least, two kinds of knowledge: knowledge about the components constituting the network (e.g. nodes, lines, peripheral processors) and knowledge about the events (the alarm notifications) issued by the network. Each of these kinds of knowledge has its own characteristics.

All network components have *individual properties* such as an identifier, a location, a current status. Network components are interlinked by various *relationships* (for example, "processor P123 is connected to node BOSTON by line 96"). The intrinsic properties of network components include thus their individual properties and their relationships.

Events emitted by the network are also characterized by some individual properties: the identifier of the issuing component, a specific code, the time at which they have been issued, ... Moreover, they have a *format*. Events are issued by the network as byte strings; the format of an event is the description of the corresponding string. Note that events are not linked by any relationship. The intrinsic properties of the event kind of knowledge include the individual properties and the format information.

3 An enhanced object-oriented system: DOOS

3.1 The concept of Representation Cluster

The limitations discussed in section 2.2.2 should not lead designers to forget the advantages of object-oriented systems, which make them a very sound basis for knowledge representation. However, we believe that these systems should be enriched with mechanisms allowing to customize them to the representation requirements of each particular kind of knowledge. These requirements include an expressiveness tailored to the intrinsic properties and the automatic generation of general treatment procedures associated with these kinds of knowledge. Such customization mechanisms lead to two object-oriented levels: the knowledge representation level and the underlying object-oriented system level; the former constitutes an abstraction layer on top of the latter.

To represent and handle knowledge using directly its intrinsic properties, a *Representation Cluster* (RC) is associated with each kind of knowledge. A RC groups all the knowledge representation level classes modelling the

corresponding kind of knowledge and is characterized by two properties: a *class definition mechanism* and a *class instantiation mechanism*. These mechanisms specify the syntax and the semantics of the formalisms that must be used to respectively define and instantiate the classes belonging to the RC.

This approach is still object-oriented and satisfies the basic principles exposed in section 2; domain entities are represented by classes and instances. However class definition and instantiation formalisms are no longer fixed as in classical object-oriented systems but can be adapted to the natural high-level properties of the represented knowledge. Therefore, a RC constitutes a knowledge representation environment specifically suited for a given kind of knowledge.

3.2 DOOS an object-oriented system embedding the RC concept

The RC concept has been implemented in DOOS, the Dantes Object-Oriented System. DOOS provides an underlying object-oriented system with the representation cluster concept. The underlying system is based on Flavors [Weinreb and Moon, 1985] and provides the features described in section 2: classes, multiple inheritance, class and instance variables, class and instance methods, variable annotations and viewpoints. In DOOS, a RC is defined by its name, its possible super RC (representation clusters can be organized in a tree allowing inheritance between them) and the declaration of its class definition and instantiation mechanisms. The following subsections present these mechanisms in DOOS.

3.2.1 RC class definition mechanism

In DOOS, a class definition is always handled using a DEFINE Lisp form. Which arguments must be given to this DEFINE form is specified by the class definition mechanism of the RC to which the class belongs. These arguments are thus entirely adaptable to the kind of knowledge related to the RC, and are chosen to allow the explicit expression of all its intrinsic properties.

Most generally, the first three arguments will be the RC name, the class name and the possible superclass name(s) (used to establish the class lattice within the RC). Other arguments can, for example, specify the class and/or instance variable names and annotations, class properties like its relationships with other classes, class behaviours like the way a network component propagates its status changes to other components.

The RC class definition mechanism also specifies the function that will interpret the definition of the classes belonging to the representation cluster. This function will generate code to implement any action implied by the class definition. These actions can include a class definition in the underlying object-oriented system, the introduction of mixins for this class, the management of the class name property-list, ... The interpretation function can also induce the automatic generation of functions or methods which will be used to manipulate the represented knowledge.

3.2.2 RC class instantiation mechanism

A class instantiation is handled by a CREATE Lisp form. The class instantiation mechanism specifies the arguments of the CREATE form and the function that interprets this form. The arguments of the class instantiation form can explicitly refer to the intrinsic properties of the considered kind of knowledge. The interpretation function will take any action necessary when a class will be instantiated. These actions will most generally involve the effective creation of an instance but also any treatment related to the "birth" of a new instance (for example, the modification of other instances linked to the new one by some relationship or the automatic creation of other instances related to the new one).

3.2.3 Defining class definition and instantiation mechanisms

In DOOS, the classes belonging to a RC and the RC intrinsic properties are mapped into features directly provided by the underlying object-oriented system.

For example, one can interpret the definition of a class C belonging to a particular representation cluster by defining a class C' in the underlying object-oriented system. Each intrinsic property specified in the class C definition can be implemented using a large variety of techniques, including the following:

- as a class of the underlying object-oriented system, related in some way with C';
- as a mixin mixed to C';
- as a (set of) class variables of C';
- as a (set of) instance variables of C';
- as a method defined on C'.

This mapping is specified in the class definition mechanism. At the present time, the language used to define the class definition mechanism associated with a RC is the language provided by the underlying object-oriented system (and Lisp). In the future, DOOS should be provided with a higher level formalism for defining that mechanism. This formalism would allow to declaratively express the mapping from classes belonging to a RC and the RC intrinsic properties to the features provided by the underlying object-oriented system.

As an example, suppose that a RC, named R, has three intrinsic properties: P1, P2, P3. The definition of the class definition mechanism of this RC could be:

```
(defmeca  :for R
          :define-form-args
            (RC-name C P1 P2 P3)
          :define-form-interpretation-function
            :C  (class :name C')
            :P1  (mixin :name P1' :mixed-in C'
                        :inheritance superclasses-first)
            :P2  (class-variable :name P2' :of C' :init-value nil)
            :P3  (method :name P3' :of P1' :parameters ()
                         :body (...))  )
```

The :define-form-args declaration indicates that the DEFINE forms used to define classes belonging to R should have five arguments: the RC name, the class name (C), and the values ot the intrinsic properties P1, P2, P3.

The :define-form-interpretation-function declaration specifies how a class C belonging to R (and its intrinsic properties) should be mapped into features of the underlying object-oriented system:

- a class, named C', would be defined in the underlying object-oriented system,
- a mixin, named P1', would be defined and introduced as a superclass of C' (the inheritance priority could be specified),

- a class variable, named P2', would be introduced in the class C'; its initial value would be nil,
- a method, named P3', with no arguments and the specified body, would be defined on the mixin P1'.

The interpretation of such a declarative definition would automatically generate the effective class definition mechanism associated with the RC named R.

The class instantiation mechanism associated with R could be defined in a similar declarative way.

4 Using Representation Clusters to customize knowledge representation

To illustrate how representation clusters can be used for developing dedicated knowledge representation systems, let us consider examples drawn from the Dantes knowledge representation system, developed using DOOS. Dantes is a tool to build expert systems for real-time network problem management. Such expert systems receive events issued by the network components or reported by the persons who operate on the network. They interpret and correlate these events to detect and diagnose problems in the network with the aim to identify faulty components. As mentioned in section 2, two kinds of structural knowledge are of particular interest in this domain: the network components and the network events. Both have quite different properties. The subsections below are devoted to these kinds of knowledge and to their representation.

4.1 Network components

4.1.1 Background

Computer networks are composed of various interconnected components (e.g. nodes, trunks, lines, processors). A network component is characterized by some specific individual properties (name, location, identifier, ...) and by some behaviours such as the way its status variations are propagated to its related components. Network components are interlinked by various relationships. For example, if the hardware constituents of a node are a memory, a bus, a control processor, and some line processors, there exists a relationship *constituents* linking the node and its hardware constituents. The intrinsic properties related to network components include thus their individual properties, their behaviours and their relationships.

The structural knowledge about a network can then be divided into the following:

- The network model. This includes the description of the various network component types, their individual properties, their behaviours, and the relationships that exist between them.
  Considering our preceding example, if the network has nodes with four types of hardware constituents (memory, bus, control processor, and line processors), the network model will include the definition of the following component types: *node, memory, bus, control-processor, line-processor*. The network model will also include the definition of the relationship *constituents* linking the *node* component type and the *memory, bus, control-processor* and *line-processor* component types.
- The network configuration. It describes all the components physically present in the network and their relationships. For example, the network configuration specifies the nodes present in the network (e.g. three nodes: New-York, Boston, and Washington), together with their related hardware constituents.

The network configuration generally includes thousands of components. During their activities, the reasoning process and the network operators need to access specific components according to various and complex criteria. Therefore, an efficient retrieval mechanism must be provided. This mechanism makes an intensive use of the relationships between network components. For example, if one has to access all the components constituting the hardware of a switching node, what is needed is all components linked to the switching node by the *constituents* relationship.

4.1.2 Before DOOS

In the initial version of Dantes, knowledge was directly represented with Flavors. Each component type was represented by a flavor. Its properties were expressed by instance variables. Its behaviours were explicitly attached as mixins to the flavor. The large variety of behaviours often led to a flavor and mixins "soup", hardly maintainable, and with a functionality very difficult to predict. The Flavors representation of relationships was really low-level. A relationship was represented, in the network component flavor, by an instance variable containing the list of the related components. In this approach, relationship management had to be explicitly expressed with procedural code attached to each network component flavor having instance variables representing relationships. This code and the resulting retrieval mechanism depended on the network representation and had to be adapted each time a relationship type was modified or added or when a new type of component was defined.

The creation of the network configuration consisted in making instances of the network component flavors with correct initial values for their instance variables including those representing relationships. The configuration creation also depended strongly on the network representation and any modification to the relationships induced modifications to the configuration creation mechanism.

A great deal of the problems above came from the inadequate level of the relationships representation. This resulted in a knowledge representation that contained too much sensitive code which had to be adapted or revised whenever the network model was modified.

4.1.3 What DOOS provides

The following RCs are defined using DOOS: *network-component* and *network-relation*.

Network-relation

We will only describe the features of the *network-relation* formalism which are necessary to understand the use of relationships in the definition, creation and treatments of network components. A relationship existing between network components is represented by two Dantes relations. These relations must be understood in the common mathematical sense. They are binary and inverse of each other. For example, the *constituents* relationship existing between the nodes and their hardware constituents is represented by a relation from the node class to the node constituent classes (this relation is called *has-constituents*)

and by the inverse relation from the node constituent classes to the node class (it is called *is-constituent-of*). (As explained below, in Dantes, network component types are represented by classes.)

Both relations modelling a relationship are defined together by a single class belonging to the *network-relation* representation cluster. A *network-relation* class definition involves the specification of the network component classes linked by the two defined relations. For example, to represent the *constituents* relationship, a class *constituents* will be defined. It describes both *has-constituents* and *is-constituent-of* relations and specifies the network component classes (nodes, ...) linked by these relations.

Network-component

Each network component type is represented by a class belonging to the *network-component* RC.

The name of a component type and the possible superclasses specification constitute the first two arguments of a *network-component* class definition. Afterwards, class and instance variables, with their possible annotations, can be declared. Finally, the relations linking the defined component class to other ones must be expressed and some behaviours can be specified.

Let us consider nodes as an example. Each node has an identifier, which is represented by an instance variable like all network component individual properties. A node is interlinked to its hardware constituents by the *constituents* relationship (represented by the *has-constituents* relation). Moreover, it is linked to some peripheral processors by a relationship represented by the *linked-to* relation. The specific behaviours associated to a node are the following: a status variation of a node influences the status of the peripheral processors connected to it and the node can record deductions about itself and its hardware constituents.

The Dantes definition of the node class is as follows:

```
(define  network-component
         node                          ; component name
         ((identifier :type symbol ))
          ;; instance variables, identifier has the symbol type
         (backbone-object)
          ;; superclass from which node inherits
         :relations
          ;; node relationships with other component types
           ((has-constituents
                  :type (or memory bus control-processor
                            line-processor))
            ;; node has a has-constituents relation with
            ;; these network component types
           (linked-to :type peripheral-processors))
            ;; node has a linked-to relation with its
            ;; connected peripheral processors
         :behaviours                   ; node specific behaviours
           ((update-peripheral-processors-status)
           (record-deduction))  )
```

Note that the behaviours attached to a *network-component* class (such as update-peripheral-processors-status and record-deduction) must have been predefined. They belong to the *behaviour* RC also defined in Dantes, but not detailled in this paper.

The class definition mechanism associated with the *network-component* RC handles a series of actions including the definition of a class in the underlying object-oriented system, the attachment of the declared behaviours to this class and the implementation of the specified relations. Moreover, it automatically builds the retrieval mechanism that will be used to access specific components in the network configuration. Indeed, the relations specified in each component class definition are used by the class definition mechanism to generate all the possible paths existing between components classes. (A path links two component classes; it is a sequence of relations that must be applied to an object of the first class to access an object of the second class.) By interpreting each network component class definition, the *network-component* class definition mechanism progressively builds a precompiled access graph, linking all the component classes by all possible paths existing between them. This access graph allows a very fast retrieval, in the network configuration, of any component object accessible from a given one and satisfying some given conditions.

The class instantiation mechanism provided by the *network-component* RC specifies the arguments of the network component creation form:

- the *network-component* class that must be instantiated;
- the values of the component individual properties;
- the specification of the network component objects that are linked by some relation with the network component object being created. If some specified component does not exist, it is automatically created.

The following form creates a node with a line-processor as constituent:

```
(create  'network-component 'node :identifier 'Detroit
         :has-constituents '(line-processor :id 5))
```

Note that the inverse relations linking existing component objects to the new one are automatically updated. The network configuration is generated from a series of such CREATE forms. (In practice, these CREATE forms are automatically generated from the codified network description available in each network environment.)

4.2 Network events

Network events are byte strings, called event strings, issued by some network components and received by Dantes. Such events trigger Dantes' reasoning process. Event strings have various formats depending on the kind of event they represent. For each format of event string, there is a well known splitting of the string into various fields. The value of some of these fields determines the specific event type to which the event string corresponds.

As for network component types, network events types are represented by classes. These classes are naturally organized into a tree. They belong to the *network-event* RC.

The main problem with network events is their creation. Dantes indeed receives a byte string from the controlled network and must transform this string into an event object that the reasoning process will be able to treat. As a consequence, the transformation process must determine the network event type (more precisely the network event class) corresponding to the incoming event.

In the previous version of Dantes, the transformation process was implemented explicitly for a specific set of received event strings and a specific tree of event types. Its code explicitly involved conditions on the received strings and referred to the event types. It was thus totally dependent on the particular controlled network. The code had to be rewritten for each network. Moreover, it had to be updated whenever the event string formats were changed or a new event type was considered.

To solve these problems, the present *network-event* class definition formalism allows to declaratively express the information needed to generate automatically the transformation process. This information consists of a *named-fields* property and a *discrimination-condition* property.

The named-fields property allows to associate symbolic names to some fields of the event string, i.e. to some substrings of the event string.

The discrimination-condition associated with an event type ET is a condition defined on the event string. If a received event string does not verify this condition, it certainly does not correspond to an event of type ET. Otherwise the subtypes of ET are recursively checked. The process ends when a leaf of the event type tree is reached and all encountered discrimination-conditions are satisfied: the event type corresponding to the received event string has been found. (Note that backtracking takes place when the process reaches an event type with unsatisfiable discrimination-condition.)

For example, consider that network events belong to two large categories, each having its own format: the *backbone-events* and the *peripheral-events*, which respectively begin with the character N or P. The definition of the corresponding classes can be:

```
(define network-event
        backbone-event              ; event name
        ()                          ; no instance variable
        ()                          ; no superclass
        :named-fields
          ((event-category (first-character *event-string*)))
          ;; the symbol event-category is bound with the
          ;; first character of the event string (stored in
          ;; global variable *event-string*)
        :discrimination-condition
          (equal event-category "N")
          ;; to correspond to a backbone-event, the event
          ;; string must begin with N
        :behaviours  ((frequency-analysis))  )
          ;; frequency analysis can be made on backbone
          ;; events

(define network-event
        peripheral-event            ; event name
        ()                          ; no instance variable
        ()                          ; no superclass
        :named-fields
          ((event-category (first-character *event-string*)))
          ;; to correspond to a peripheral-event, the event
          ;; string must begin with P
        :discrimination-condition
          (equal event-category "P") )
```

From the definition of all event classes and especially from the named-fields and discrimination-condition declarations, the class definition mechanism associated with the *network-event* RC automatically builds a very efficient transformation process. This process is specifically suited to the various formats of the received event strings and to the defined event classes. Therefore, adding or removing event classes or changing the format of event strings only requires to change the declarative representation of events (changing named-fields and/or discrimination-condition properties, defining new event classes or removing obsolete ones). The automatic generation of the transformation process relieves the knowledge engineer from any further programming and removes the need to write any code specifically bound to the event formats of the specific network. Furthermore, it is worth noticing that both properties used for this automatic generation are natural and intrinsic properties of events.

5 Discussion

5.1 Advantages of the Representation Cluster approach

The examples of section 4 show very clearly the advantages of an object-oriented system involving the representation cluster concept.

First of all, such a system preserves the well-known qualities of an object-oriented approach for model-based applications.

Furthermore, it suppresses the major drawbacks of object-oriented systems concerning their lack of high expressive power. Representation clusters indeed allow to dedicate a high-level specific formalism to each kind of knowledge. These dedicated formalisms allow to express knowledge in a natural way, providing adequate levels of abstraction for modelling (sub-) domains. The intrinsic properties that characterize the represented knowledge are directly expressed in the formalism and no longer dispersed in object-oriented features like variables, methods, mixins and so on. This drastically increases the knowledge base readability. Moreover, such dedicated formalisms enable to privilege the important concepts of a kind of knowledge.

The interpretation function of these formalisms can avoid a lot of programming effort by automatically generating functions and methods to be used for the treatment of the represented knowledge. This considerably clarifies the knowledge representation system. Indeed, as soon as the relevant information about the domain has been expressed, all that can be derived from this information is automatically generated. This makes knowledge bases much easier to maintain.

Another point to stress is that all the formalisms of the representation system can provide the same class definition or creation framework and therefore can present a clean uniformity of style. For example, in DOOS, it is very easy to adopt a sound convention for the order of the DEFINE and CREATE arguments. If such conventions are established, the various kinds of knowledge of a domain will be expressed in a uniform style but with their specific characteristics.

Finally, the representation system resulting from the various formalisms is very adaptative. If a formalism does not fit exactly the requirements of the knowledge to which it is dedicated, one simply needs to modify the class definition

or creation mechanism of the corresponding representation cluster.

5.2 Relationships with other concepts

It is worth noticing that metaclasses and RCs are quite different. They lie in two different conceptual levels, respectively the object-oriented system (OOS) level and the knowledge representation (KR) level. Metaclasses are OOS classes whose instances are OOS classes, introducing an additional repetitive layer of classes and objects. They allow to treat everything, at the object-oriented system level, as an object. Now, RCs are introduced to provide an abstraction layer on top of the object-oriented system. Futhermore RCs allow to customize object-oriented knowledge representation formalisms to various kinds of knowledge. They group *knowledge representation* classes and not classes of the underlying object-oriented system. A KR class does not exist by itself in the underlying object-oriented system but corresponds to several features of this object-oriented system (classes, methods, ..., and, why not, metaclasses).

It is also interesting to position our approach with respect to CLOS (Common Lisp Object System) [Bobrow *et al.*, 1988]. CLOS does not yet provide a standard for metaclasses and metaobjects. However the trends have been described in [Bobrow and Kiczales, 1988]. According to these trends, the CLOS interpreter is metacircular, i.e. the CLOS implementation can be viewed as an object-oriented program which is itself written in CLOS. This lets users extent or modify the standard interpreter behaviour by redefining some protocols (i.e., some sets of generic functions). Typically what can be customized is the inheritance mechanism, the allocation and slots access of instances, the updating of subclasses when a class has been modified, and the method combination mechanism. All these features concern internal properties of objects and classes; they stand in the object and class world. They do not concern the formalisms used to define or instantiate classes. In CLOS, these formalisms seem to have been predefined once and for all. Therefore the purposes of representation cluster and CLOS metaobject concepts are fundamentally different.

6 Conclusion

This paper presented the Representation Cluster concept. Each RC is composed of a class definition and a class instantiation mechanism that can be customized to the different kinds of knowledge a knowledge-based system has to deal with. These mechanisms allow to define and instantiate classes using formalisms that are exactly adapted to the particularities and intrinsic properties of the represented knowledge. This approach helps solving the major problems of the object-oriented systems coming from their low level of abstraction and uniformity of expression. Its advantages are:

- it preserves the well-known advantages of the object-oriented systems;
- it allows to express knowledge in a high-level natural way, which facilitates the development and maintenance of knowledge-based systems;
- it relieves knowledge base developers from a tangible amount of programming.

DOOS implements the RC concept upon an underlying object-oriented system based on Flavors. It has been used to develop the Dantes knowledge representation system tailored to the network troubleshooting domain. However, the RC feature could be added to most object-oriented systems: to implement it, the user simply needs to rely on the features provided by his object-oriented system.

Acknowledgements

We are very grateful to Suzanne Galand and Pierre-Joseph Gailly for their helpful comments on drafts of this paper. We also thank the anonymous referees for their remarks.

References

[Bobrow *et al.*, 1988] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common Lisp Object System Specification. X3J13 standards commitee document 88-002R", *ACM SIGPLAN Notices*, vol 23, special issue, September 1988.

[Bobrow and Kiczales, 1988] D. Bobrow and G. Kiczales. The Common Lisp Object System Metaobject Kernel. A Status Report. *Proceedings of the ACM Conference on Lisp and Functionnal Programming*, Snowbird Utah, 1988, pp. 309-315.

[Bobrow and Winograd, 1977] D. Bobrow and T. Winograd. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1:1, pp 3-46, 1977.

[Danforth and Tomlinson, 1988] S. Danforth and C. Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys* 20:1, 1988, pp 29-72.

[Fikes and Kehler, 1985] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28:9, pp. 904-920, 1985.

[Mathonet *et al.*, 1987] R. Mathonet, H. Van Cotthem, and L. Vanryckeghem. DANTES, an expert system for real-time network troubleshooting. *Proceedings of IJCAI-87*, 1987, pp. 527-530.

[Minsky, 1975] M. Minsky. A Framework for Representing Knowledge. In P. Winston (ed), *The Psychology of Computer Vision*, MacGraw Hill, New York, 1975.

[Stefik and Bobrow, 1986] M. Stefik and D. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine*, 6:4, 1986, pp 40-62.

[Weinreb and Moon, 1985] D. Weinreb and D. Moon. *Lisp Machine Manual*. Symbolics Inc., 1985.