PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Objects, Properties, and Modules in *QUIXOTE*

Hideki Yasukawa, Hiroshi Tsuda, and Kazumasa Yokota

Institute for New Generation Computer Technology (ICOT)
21F. Mita-Kokusai Bldg., 1-4-28 Mita, Minato-ku, Tokyo 108, JAPAN

e-mail: yasukawa@icot.or.jp, tsuda@icot.or.jp, kyokota@icot.or.jp

Abstract

This paper describes a knowledge representation language *QUIXOTE*. *QUIXOTE* is designed and developed at ICOT to support wide range of applications in the Japanese FGCS project.

QUIXOTE is basically a deductive system equipped with the facilities for representing various kind of knowledge, and for classifying knowledge.

In *QUIXOTE* , basic notions for representing concept and knowledge are *objects* and thier *properties*. Objects are represented by extended terms called *object terms*, and thier properties are represented by *subsumption* constraints over the domain of object terms.

Another distinguished feature of *QUIXOTE* is its concept of *modules*. Modules play an important role in classifying knowledge, modularizing a program or a database, assumption-based reasoning, and so on.

In this paper, concepts of objects, properties, and modules are presented. We also present how modules work with objects and their properties.

1 Introduction

Logic programming is a powerful paradigm for knowledge information processing systems from the viewpoint of knowledge representation, inference, advanced databases, and so on.

QUIXOTE is designed and developed to support this wide range of applications in the Japanese FGCS project. Briefly speaking, it is a constraint logic programming language, a knowledge representation language, and a *deductive object-oriented database* language.

In *QUIXOTE*, basic notions for representing concepts and knowledge are *objects* and their *properties*. An object in *QUIXOTE* is represented by an extended term called an *object term*, and its properties are defined as a set of *subsumption* constraints.

Another distinguished feature of *QUIXOTE* is its concept of *modules*. A module corresponds to a part of the world (situation) or the local database. In *QUIXOTE*, its module concepts play an important role in classifying knowledge, modularizing a program or a database, assumption-based reasoning, and so on.

In this paper, concepts of objects, properties, and modules are presented. We also present how modules work with objects and their properties, for example, in classifying or modularizing them.

Other features of *QUIXOTE* and the formalism appear in other papers[20, 12, 21].

Section 2 shows how objects and their properties are treated in a simple version of *QUIXOTE*. Section 3 shows how complex objects are introduced in *QUIXOTE*, and how they are used to deal with *exceptions* in *property inheritance*. Section 4 describes deductive rules in *QUIXOTE*, and the overview of deductive aspects of *QUIXOTE*. Section 5 describes module concepts with some examples. Section 6 describes the facilities for relating modules, especially to import or to export rules among modules. Section 7 describes queries in *QUIXOTE*, which provides the facilities to deal with modifications of a program, or assumption-based reasoning. Finally, Section 8 describes brief comparison with related works.

2 A simple system of objects and their properties

Object-oriented features are very useful for applying logic programming to 'real' applications. *QUIXOTE* is designed as a logic programming language with features such as: object identity, complex objects, encapsulation, inheritance, and methods, which are also appropriate for deductive object-oriented databases and situation theoretic approaches to natural language processing systems.

An *object* is a key concept in *QUIXOTE* to represent concepts or knowledge. In knowledge representation applications, it is important to identify an object or to distinguish two distinct objects, as in the case of object-oriented language.

Object identity is the basic notion for identifying objects.

QUIXOTE precisely defines object identity, where extended terms are used as *object identifiers*. In this sense, extended terms in *QUIXOTE* are called *object terms*.

In this section, simplified treatment of objects and their properties are presented. That is, the case of every object term is *atomic*. In the next section, the system of object terms is extended to non-atomic and complex cases, including the non-well-founded (circular) case.

2.1 Basic Objects

At the first approximation, we assume that each object has a unique atomic symbol as its identifier.

The important thing, here, is that objects are related to each other. There are some relations to be considered, such as *is_a*-relations, *part_of*-relations, and so forth. In *QUIXOTE*, *subsumption* relations over objects are used to relate objects.

First, a set BO of atomic symbols called *basic objects* is assumed. BO is partially ordered by the *subsumption* relation (written $\sqsubseteq$), and $(BO, \sqsubseteq, \top, \bot)$ is a lattice with $\top$ as its maximum element and $\bot$ as its minimum element.

A basic object is used as an object identifier (an object term) in this simple setting.

An example of the lattice is

$$BO^* = (\{animal, mammal, human, dog\}, \sqsubseteq, \top, \bot)$$

where the following holds:

$$mammal \sqsubseteq animal,$$
$$human \sqsubseteq mammal,$$
$$dog \sqsubseteq mammal.$$

2.2 Attribute Terms and Dotted Terms

In addition to the basic objects, we assume a subset L of BO, called *labels*. Labels are used to define the attributes of objects.

An attribute of the object o is represented by the triple (o, l, v) where l is a label and v is an object. The following example shows that John has the attribute of his age being 20: $(john, age, 20)$.

A property of an object is represented by a set of the pairs of a label and its value, that is, a set of attributes. Thus, John's having a property of being 20 years old and being a male is represented by $\{(john, age, 20), (john, sex, male)\}$.

Formally, a label l is interpreted as a function:

$$l : BO \to BO.$$

The syntactic construct for representing an object and its properties is the *attribute term*. An attribute term is of the form:

$$o/[l_1 = v_1, \ldots, l_n = v_n]$$

where o is an object term, l_i's are labels, and v_i's are objects. The syntactic entity $[l_1 = v_1, \ldots, l_n = v_n]$ is called the *attribution* of the object term o. It specifies a property of the object. In what follows, we say that o has the attributes $[l_1 = v_1, \ldots, l_n = v_n]$ when there is no confusion.

For example, the following is an attribute term representing that John has the property of being 20 years old and being a male:

$$john/[age = 20, sex = male].$$

Notice that an object identifier and its property (attribution) are separated by "/".

It is useful to regard an attribute of an object as a concept. For example, John's age can be seen as a concept. In *QUIXOTE*, this kind of concept is represented by *dotted terms*. A dotted term is defined as a pair of an object term and a label, and has the following form:

$$o.l$$

where o is an object term and l is a label.

For example, John's age is represented by the following dotted term:

$$john.age = 20.$$

A dotted term is treated as a *global variable* ranging over the domain of object terms, and interpreted as an object term. The following holds for dotted terms:

$$o_1 = o_2 \Rightarrow o_1.l = o_2.l$$
$$o.l = o_1, o.l = o_2 \Rightarrow o_1 = o_2$$
$$o.l_1 = o_1 \Rightarrow o.l_1.l_2 = o_1.l_2.$$

2.3 Properties as Subsumption Constraints

It is often the case that an object has certain attribute while its value is not fully specified.

$$john/[age \to positive\_integer]$$

The above attribute term represents that John has the property of his age being subsumed by positive integer. In this case, John's age is not specified but constrained as being subsumed by *positive_integer*.

The constraint in simple *QUIXOTE* is a subsumption constraint over basic objects. As mentioned before, the domain of basic objects is a lattice under the subsumption relation. Thus, the rules of subsumption constraints are simply defined as follows:

- $x = x$,
- if $x = y$ then $y = x$,
- if $x = y$ and $y = z$ then $x = z$,
- $x \sqsubseteq x$,

- if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$,
- if $x = y$ and $x \sqsubseteq z$ then $y \sqsubseteq z$,
- if $x = y$ and $z \sqsubseteq x$ then $z \sqsubseteq y$,
- if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$,
- if $x \sqsubseteq y$ and $x \sqsubseteq z$ then $x \sqsubseteq (y \downarrow z)$, and
- if $y \sqsubseteq x$ and $z \sqsubseteq x$ then $(y \uparrow z) \sqsubseteq x$

where $(x \downarrow y)$ is the infimum (meet) and $(x \uparrow y)$ is the supremum (join). Note that $x = y$ is equivalent to the conjunction of $x \sqsubseteq y$ and $y \sqsubseteq x$, that is, the following holds:

$$x = y \stackrel{def}{\equiv} x \sqsubseteq y \wedge x \sqsupseteq y.$$

A set of subsumption constraints is solvable if and only if it does not contain $a = b$ for two distinct basic objects a and b with respect to the above rules[20].

Based on the subsumption constraint, the attribute term above is defined as a pair of the basic object *john* and a subsumption constraint $john.age \sqsubseteq positive\_integer$. Sometimes, this kind of pair is written as:

$$john/|\{john.age \sqsubseteq positive\_integer\}.$$

This is just the opposite of the description of the attribute term shown above.

The general form of an attribute term is as follows:

$$o/[l_1\ op_1\ v_1, \ldots, l_n\ op_n\ v_n]$$

where $op_i \in \{=, \rightarrow, \leftarrow\}$.

For each label that is not explicitly specified, we assume that its value is constrained, that is, subsumed by $\top$. This assumption states that the property of an object can be partially specified.

In addition to the subsumption constraint over basic objects, the subsumption constraint over sets of basic objects is used in *QUIXOTE*. For example, the following attribute term represents that cooking and walking are John's hobbies:

$$john/[hobby \leftarrow \{cooking, walking\}],$$

that is, the dotted term *john.hobby* is a set subsuming the set $\{cooking, walking\}$.

The subsumption relation $\sqsubseteq_H$ over the domain of sets of basic objects is defined as Hoare-ordering over $\sqsubseteq$-ordering as follows:

$$s_1 \sqsubseteq_H s_2 \Rightarrow \forall x \in s_1 \exists y \in s_2\ x \sqsubseteq y.$$

It should be noted that the domain of sets of basic objects is classified by the equivalence relation defined by the Hoare-ordering. In *QUIXOTE*, any set is interpreted as the representative element of the equivalence class that is defined by the following rule:

$$[s] \stackrel{def}{\equiv} \{x \in s \mid \neg \exists y \in s\ x \neq y \wedge x \sqsubseteq y\}.$$

Under this definition, $\sqsubseteq_H$-ordering becomes a partial ordering, and can be used as an equivalence relation.

For example, the following holds:

$$[\{1, integer, \text{"}abc\text{"}, string\}] = \{integer, string\},$$

provided that $1 \sqsubseteq integer$ and "abc" $\sqsubseteq string$.

2.4 Property Inheritance

It is natural to assume that properties are inherited from object terms with respect to $\sqsubseteq$-ordering. For example, consider the following example:

$$swallow \sqsubseteq bird.$$
$$bird/[canfly \rightarrow yes].$$

Since *swallow* is a kind of (subsumes) *bird* and *bird* has the attribute $[canfly \rightarrow yes]$, *swallow* comes to have the same attribute by default.

The rule for inheritance of properties between objects is:

Definition 1 (Rule for inheritance)

$$o_1 \sqsubseteq o_2 \Rightarrow o_1.l \sqsubseteq o_2.l.$$

If $o_1 \sqsubseteq o_2$ holds, then the following holds according to this rule:

- if o_2 has the attribute $[l \rightarrow o']$, then o_1 also has the same attribute,
- if o_1 has the attribute $[l \leftarrow o']$, then o_2 also has the same attribute.

Notice that the attribute $[l = o']$ is the conjunction of $[l \rightarrow o']$ and $[l \leftarrow o']$.

As mentioned before, an attribute term is defined as the pair of an object term and a set of subsumption constraints, thus, property inheritance can be considered as constraint inheritance.

3 Complex Objects

The simplified approach shown in the previous section lacks the capability to represent the complex objects required in actual applications, such as trees, graphs, proteins, chemical reactions, and so forth.

A complex object has certain "structures" intrinsic to its nature. It is important for knowledge representation languages to represent such complex object structures in its description, that is, as the object identifier in *QUIXOTE*.

Thus, it is important to give a facility to introduce complex object terms to *QUIXOTE*.

3.1 Intrinsic vs. Extrinsic Properties

The approach adopted in *QUIXOTE* is a natural extension of the simplified language given in the previous section.

An object has the property, that is, a set of attributes, which are *intrinsic* to identifying that object. Thus, the properties of an object are separated into two, the intrinsic property and the other extrinsic properties. Similarly, the attributes of an object are divided into two, intrinsic attributes and extrinsic attributes. In *QUIXOTE*, the intrinsic attributes are included in the object term representation but not in the attribution of an attribute term representation.

For example, the concept of *red apple* is represented by the following complex object term:

$$apple[color = red].$$

Note that the difference between this object term and the attribute term $apple/[color = red]$ that represents the concept of *apple* with the attribute $[color = red]$ as its extrinsic property.

Let o be a basic object, $l_1, l_2, \ldots$ be labels, and $o_1, o_2, \ldots$ be object terms.

- Every basic object is an object term.
- A term $o[l_1 = o_1, l_2 = o_2, \ldots]$ is an object term if it contains only one value specification for each label.
- A term is an object term only if it can be shown to be an object term by the above definition.

For an object term $o[l_1 = o_1, l_2 = o_2, \ldots]$, o is called the *principal object* and $[l_1 = o_1, l_2 = o_2, \ldots]$ is called the *intrinsic property specification*. The intrinsic property specification of an object term is the set of intrinsic attributes of the object term, and interpreted as the indexed set of object terms indexed by the labels. Thus, an object term is interpreted as the pair of its principal object o and the indexed set s, and is written as:

$$(o, s).$$

Let $BO = \{human, 20, 30, int, male, female\}$, $20 \sqsubseteq int, 30 \sqsubseteq int$, $L = \{age, sex\}$. The following terms are object terms in *QUIXOTE*:

$$human,$$
$$human[age = 20, sex = male].$$

These two object terms are interpreted as $(human, \{\})$ and $(human, \{(age, 20), (sex, male)\})$.

The object term $\top[l_1 = v_1, \ldots]$ is described as $[l_1 = v_1, \ldots]$ for convenience.

By the definition of complex object terms, the following holds:

$$o[\ldots, l_i = v_i, \ldots].l_i = v_i.$$

For example, $human[age = 20].age = 20$ holds.

It is possible to have object terms containing variables ranging over ground object terms as follows:

$$human$$
$$human[age = X, sex = Y].$$

3.2 Extended Subsumption Relation

Given subsumption relations $\sqsubseteq$ among basic objects, the relations can be extended into subsumption relations among complex object terms. The extended subsumption relations preserve the ordering on basic objects, and also constitute a lattice.

Though the precise definition of a extended subsumption relation is given in [20], intuitive understanding will suffice at this point. Intuitively, $o_1 \sqsubseteq o_2$ (we say o_2 subsumes o_1) holds between two complex object terms o_1 and o_2 if and only if:

(1) the principal object of o_2 subsumes the principal object of o_1,

(2) o_1 has more labels than o_2, and

(3) the value of each label of o_2 subsumes the value of each label of o_1.

For example, the following holds:

$$human[age = 20, sex = male] \sqsubseteq animal[age = integer],$$

because the principal object of $animal[age = integer]$ ($animal$) subsumes the principal object of $human[age = 20, sex = male]$ ($human$), the object term $human[age = 20, sex = male]$ has more labels than $animal[age = integer]$, and $20 \sqsubseteq integer$ holds.

Similarly,

$$human[age = 20] \sqsubseteq animal[age = int]$$

holds, but $human[age = 20]$ and $human[sex = male]$ can not be compared with respect to $\sqsubseteq$-ordering over complex object terms.

In this extended subsumption relation over object terms, the object term $\top$ is the largest among all the object terms. In *QUIXOTE*, the object term $\perp$ is the smallest of all, that is, $\perp$ is used as the representative element of the class of object terms that are smaller than $\perp$[1].

The semantic domain of object terms is a set of labeled graphs as a subclass of hypersets with urelement[2, 13].

[1] From the definition of object terms and subsumption relation over them, it is possible to have an object term of the form:

$$\perp[l_1 = v_1, \ldots].$$

The reason such a domain is adopted is to allow object terms with infinite structure. Subsumption relations corresponds to *hereditary subset relations*[2] on that domain.

The rules for extended subsumption constraints are those listed in 2.3 plus the following:

- if $(o_1, s_1) \sqsubseteq (o_2, s_2)$ then $o_1 \sqsubseteq o_2$ and for each $(l, v_2) \in s_2$ there exists (l, v_1) such that $v_1 \sqsubseteq v_2$,

- if $(o_1, s_1) = (o_2, s_2)$, then $o_1 = o_2$ and for each $(l, v_1) \in s_1$ there exists $(l, v_2) \in s_2$ such that $v_1 = v_2$ (the symmetric condition follows).

These two rules correspond to the *simulation* and *bisimulation* relation in [2, 13], where the bisimulation relation is an equivalence relation.

3.3 Exception on Property Inheritance

By introducing complex object terms in terms of intrinsic-extrinsic distinction, it becomes possible to define the notion of *exceptions* on inheritance of properties in a clear way.

Intuitively, the intrinsic property of an object is the property that distinguishes that object from others, and such properties should not be inherited.

In addition to the rule for property inheritance given in 2.4, the rule for exception is defined as follows:

Definition 2 (Rule for exception)
The intrinsic attributes of an object term override the attribution inherited from the other object terms, and any of the intrinsic attributes is not inherited to the other object terms.

In sum, the intrinsic attributes are out of the scope of property inheritance.

For example, consider the attribute of the object term $bird[canfly \rightarrow no]$ with respect to the following database definition:

$$bird/[canfly = yes],$$
$$bird[canfly = no].$$

The object term $bird[canfly = no]$ inherits the attribute $[canfly \rightarrow yes]$, by the rule for inheritance. However, $bird[canfly = no]$ contains the intrinsic specification on the label $canfly$. Thus, $bird[canfly = no]$ has the attribute $[canfly \rightarrow no]$ as its property by the rule for exception.

Thus, given in Section 3.1, the following holds even if property inheritance occurs:

$$o[l_1 = x_1, \ldots, l_n = x_n]/[l_1 = x_1, \ldots, l_n = x_n].$$

4 Deductive Rules

It is important for knowledge representation languages to provide facilities for certain types of inferences, namely deductive inference.

The deductive system of $\mathcal{QUIXOTE}$ is defined by *deductive rules* (rules, for short).

4.1 Rules in $\mathcal{QUIXOTE}$

First, a literal (atomic formula) of $\mathcal{QUIXOTE}$ is defined to be an object term or an attribute term.

The rules of $\mathcal{QUIXOTE}$ are defined as follows:

(1) a literal H,

(2) $H \Leftarrow B_1, \ldots, B_n$ where $H, B_1, \ldots, B_n$ are literals.

H is called the *head* and the "$B_1, \ldots, B_n$" is called the *body* of the rule.

The rules of the form (1) are sometimes called an *unit rule* or a *fact*[2].

Rules of the form (2) are called *non-unit rules*.

A fact H is shorthand for the non-unit rule whose body is empty, that is, the rule $H \Leftarrow$. When there is no confusion, non-unit clauses are simply called rules.

A *database* or a *program* is defined as a finite set of rules.

A fact specifies the existence of an object and its property. The following is an example of facts:

$$john;;$$
$$john/[age = 20];;$$

The former fact specifies that the literal $john$ holds (or is true), that is, the database has the object $john$ as its member. In addition to that, the latter additionally specifies that $john$ has the property of $[age = 20]$.

The informal meaning of the rule $H \Leftarrow B_1, \ldots, B_n$ is as usual, that is, if $B_1, \ldots, B_n$ holds then H holds.

As mentioned in Section 2.3, properties are interpreted as subsumption constraints. Thus, a rule is defined as a triple (H, B, C) of the object term H in its head, the set of object terms B in its body, and the set of constraints C. The elements of B are called *subgoals*. Thus, any rule can be represented by the following form:

$$H \Leftarrow B \parallel C.$$

This form of rule is called *constraint-based* form.

It is possible to associate constraints other than those corresponding to attributes with a rule as follows:

$$john/[daughter \leftarrow \{X\}] \Leftarrow$$
$$X/[father = john] \parallel \{X \sqsubseteq female\}.$$

[2]Sometimes, a fact is defined to be a unit-rule having a *non-parametric* object term as its head. In that case, the set of facts corresponds to a extensional database in usual deductive databases.

Precisely speaking, the set of constraint C of a rule is classified into two, the constraints in the head of the rule (*head constraints*) and the constraints in the body (*body constraints*). For example, the rule

$$o/[l_1 = o_1, l_2 = o_2] \Leftarrow$$
$$p/[l_3 = o_3], q/[l_4 = o_4]$$

has $\{o.l_1 = o_1, o.l_2 = o_2\}$ as its head constraints, and $\{p.l_3 = o_3, q.l_4 = o_4\}$ as its body constraints.

In the context of object-oriented language, the attributes in the head of a rule correspond to the *methods* and the body of the rule can be seen as the corresponding *implementation*, as in F-logic[8].

4.2 Derivations and Answers

Compared to the usual notion of the derivation of goals and answers in logic programming language like Prolog, two points must be explained in the case of *Quixote*.

The first point is concerning the role of object terms as object identifiers. The value of an attribute of an object is unique, because the label corresponding to the attribute is interpreted as a function.

The second point is concerning the fact that the attributes of an object can be partially specified and they are interpreted as subsumption constraints.

Consider the following database:

Example 1

$$o[l = X]/[l_1 \rightarrow a, l_2 = b] \Leftarrow X \parallel \{X \sqsubseteq c\};;$$
$$o[l = X]/[l_1 \rightarrow d, l_3 = e] \Leftarrow X \parallel \{X \sqsubseteq f\};;$$
$$p;;,$$

where both $p \sqsubseteq c$ and $p \sqsubseteq f$ hold.

In this case, $o[l = p]/[l_1 \rightarrow a, l_2 = b]$ holds by the first rule and $o[l = p]/[l_1 \rightarrow d, l_3 = e]$ holds by the second rule. Thus, by combining these two, the object term $o[l = p]$ gains $[l_1 \rightarrow (a \downarrow d), l_2 = b, l_3 = e]$ as its attribute.

This process is done by *merging* the attributes of the derived subgoals equivalent to each other.

The merging process becomes complicated if we take into account the partiality of the attributes of an object.

Consider the following example:

Example 2

$$o/[l_1 \rightarrow a] \Leftarrow p/[l_2 \rightarrow b];;$$
$$o/[l_1 \rightarrow c] \Leftarrow p/[l_2 \rightarrow d];;$$
$$p;;.$$

The subgoal p of the first rule holds with attribute $[l_2 \rightarrow b]$, which is not defined in the database. This is because the fact $p;;$ in the example does not specify the value of its l-attribute. Similarly, the subgoal p of the second rule holds with $[l_2 \rightarrow d]$. If these two attributes are inconsistent, the two rules cannot be applied together, that is, the derivations given by the two rules must not be merged.

Definition 3 (Derivation of a goal)
A derivation of a goal G_0 by a program is defined as the 5-tuple (G, R, Θ, HC, BC) of a sequence G $(= G_0, G_1, \ldots)$ of goals, a sequence R $(= R_1, \ldots)$ of the renaming variants of the rules, a sequence Θ $(= \theta_1, \ldots)$ of most general unifiers[3], the two sets of constraints HC and BC of all the head constraints and all the body constraints of the rules in P, such that each G_{i+1} is derived from G_i and R_{i+1} using θ_{i+1}, and $(HC \cup BC)\Theta$ is solvable.

Definition 4 (Assumed constraint set)
The assumed constraint set of a derivation D $(= (G, P, \Theta, HC, BC))$ is defined as the set of all constraints in BC that are not satisfied by HC with respect to the substitution Θ.

The assumed constraint set of a derivation is the set of attributes of objects which are assumed to derive the goal. This is because some attributes of objects in a database are partially defined.

Each derivation has its own *derivation context* defined as the consequence relation $(\vdash_C)$ between its assumed constraint set and its head constraints. A derivation context $A \vdash_C B$ of a goal represents that the goal is derived by assuming A, and as a consequence, B holds.

The notion of an *refutation* is defined similary as usual: a finite length derivation that has the empty goal as the last goal in its goal sequence.

In Example 2, the two refutations of the goal o have the following derivation contexts, :

$$p.l_2 \sqsubseteq b \vdash_C o.l_1 \sqsubseteq a,$$
$$p.l_2 \sqsubseteq d \vdash_C o.l_1 \sqsubseteq c.$$

To deal with merging of attributes discussed above, of a goal must be merged into the other refutation of the same goal if the derivation contexts of the two refutations have some relation to each other, that is, if the assumed constraint set of one refutation holds in the assumed constraint set of another refutation. This means that the condition holds in a weaker assumption also holds in a stronger assumption.

For example, in Example 2, if $b \sqsubseteq d$ holds, then the second refutation is merged into the first one. As a consequence, A new refutation is given instead of the first refutation, whose derivation context is follows:

$$p.l_2 \sqsubseteq b \vdash_C o.l_1 \sqsubseteq (a \downarrow c).$$

Moreover, if $b \sqsubseteq d$ and $c \sqsubseteq a$, then the context of both refutation becomes as follows:

$$p.l_2 \sqsubseteq d \vdash_C o.l_1 \sqsubseteq c.$$

[3]The most general unifier of two object terms is defined similarly to the usual one, except for the definition of terms.

This means that the first derivation is absorbed to the second with respect to the merge, because $(a \downarrow c) = c$ holds.

After merging all possible pairs of refutations, the notion of an answer to a query is defined as follows:

Definition 5 (Answer)
An answer to the query is defined as a pair of the answer substitution and the derivation context of a refutation.

Thus, the following two answers are given to the query $?\!-\!o/[l = X]$ to the database shown in Example 2:

$$(\{\}, p.l_2 \sqsubseteq b \vdash_C o.l_1 \sqsubseteq a),$$
$$(\{\}, p.l_2 \sqsubseteq d \vdash_C o.l_1 \sqsubseteq c),$$

if no condition is given among a, b, c , and d.

The *QUIXOTE* interpreter returns all answers at once, that is, it employs the top-down breadth-first search strategy.

5 Modules in *QUIXOTE*

In this section, a module concept is introduced into *QUIXOTE*.

5.1 Need for Modules in Deductive System

The goal of knowledge representation is to provide a facility for *reasoning* about a problem by using given knowledge in the way that ordinary people do: we call this everyday-reasoning, or human-reasoning.

Such reasoning systems can be defined as the pair (R, A) of a set of deductive rules and an algorithm for extracting all consequences from the rules.

For simplicity, fix A, and think of R as the knowledge in a reasoning system.

- R is neither consistent nor complete, even though its fragments may be consistent in themselves,
- reasoning is situation-dependent, i.e., some fragment of R is relevant or meaningful in a certain situation,
- reasoning usually requires some assumption.

One way to deal with such an aspect of reasoning is to associate an index to each literal and each rule in R.

Indexes can be used:

(1) to define a fragment of rules (a chunk of knowledge) which can be used in a certain situation, and

(2) to clarify which assumption (set of rules) is used.

(1) defines our conception of a *module* as a set of rules with same index. Thus, if we regard an index as the identifier for a context or a situation, the set of rules can be seen as the chunk of knowledge relevant for that context or situation.

As the result of introducing indexes, each literal has come to have the form:

$$m : A$$

where m is an index called *module identifier*, and A is an object term or an attribute term.

As a consequence, the usual consequence relation between formulas should be replaced by:

$$m_1 : A_1, \ldots, m_i : A_i \vdash m : A$$

Intuitively, this means that A holds in m with reference to parts $m_1, \ldots, m_i$ of the database. In obtaining the answer, the choice of parts of the database can be seen as the assumptions.

In *QUIXOTE*, an object term is used as a module identifier. The use of object terms as module identifiers enables the user to treat modules as objects, and provides meta-like programming facilities.

5.2 Rules with Module Identifiers

Corresponding to the constraint-based form of a rule given in Section 4, a modularized rule has the following form:

$$m_0 :: o_0 \Leftarrow m_1 : o_1, \ldots, m_n : o_n \parallel C$$

where $o_0, o_1, \ldots, o_n$ are object terms, $m_0, m_1, \ldots, m_n$ are module identifiers, and C is a set of constraints[4].

This rule specifies the following two things:

(1) this rule is in (or is accessible from) the module with module identifier m, and

(2) if each subgoal $m_i : o_i$ holds with respect to a variable assignment and constraints C then $m : o$ holds.

Generally, the modules and their rules are defined as follows:

$$\{m_1, \cdots, m_n\} :: \{r_1;; \cdots;; r_m\},$$

where $r_1, \cdots, r_m$ are rules. Note that it is possible for modules to be nested in multiple.

Thus, it is easy to have a set of rules in a module as the set of all rules with the module identifier of that module. The set of rules in the module with m as its identifier

[4] Precisely, this form represents the rule

$$o_0 \Leftarrow m_1 : o_1, \ldots, m_n : o_n \parallel C$$

with index m_0.

is written as Σ_m[5]. In general, a module identifier may be parametric, that is, an object term with variables in its description. The variables appearing in a rule are interpreted as universally quantified, thus the parametric module identifiers that are equivalent with respect to variable renaming are regarded as the same.

In *QUIXOTE*, it is assumed that each module is consistent. It is an important feature of modules to represent inconsistent knowledge where inconsistency arises from differences in situations or context. For example, consider the situation of John's believing that Mary is 20 years old, when she is actually 21 years old. The following database shows the treatment of such a problem:

$$johns\_belief :: mary/[age = 20];;$$
$$real\_world :: mary/[age = 21];;$$

In this case, the database is consistent as a whole unless the two modules are related to each other.

The following example shows the use of parametric module identifiers to describe so-called *generic* modules. A parametric module identifier can be used to pass *parameters* to the rules in the module.

Example 3 (Generic Module)

```
sorter[cmp = C] :: {
  sort[l = [], sorted = [], cmp = C];;
  sort[l = [A|X], sorted = Y, cmp = C] ⇐
    split[l = [A|X], base = A, cmp = C, l1 = L1, l2 = L2],
    sort[l = L1, sorted = Y1, cmp = C],
    sort[l = L2, sorted = Y2, cmp = C],
    list : append[l1 = Y1, l2 = Y2, l = Y];;
    ...};;
  ⋮
less_than :: {
  compare[arg1 = A, arg2 = B, res = yes] ||
    {A < B};;
  compare[arg1 = A, arg2 = B, res = no] ||
    {B < A}};;
```

Module $sorter[cmp = C]$ has the definition of a quick-sorting procedure using argument C for the comparator, and module *less_than* has the definition of a comparator, where the relation $<$ is used as constraint relation for comparing two objects.

In processing the query:

$$?\!-sorter[cmp = less\_than] :$$
$$sort[l = L, sorted = R, cmp = C],$$

the module identifier *less_than* is passed to the rules in the sorting module, and used to compare two elements of list L. It is possible to give module identifiers other than *lt* for using different comparator in the sorting procedure.

The next example shows the treatment of *state transitions* by using modules for representing states.

Example 4 (State Transition)

```
m :: {
  a/[on = nil];; b/[on = a];;
  c/[on = nil];; d/[on = c]};;
sc[sit = M, op = move[obj = A, fr = B, to = C]] :: {
  C/[on = A] ⇐
    M : A/[on = nil],
    M : B/[on = A],
    M : C/[on = nil];;
  B/[on = nil] ⇐
    M : A/[on = nil],
    M : B/[on = A],
    M : C/[on = nil];;
  A/[on = nil] ⇐
    M : A/[on = nil],
    M : B/[on = A],
    M : C/[on = nil]};;
```

In the initial state m, block a is on top of block b, and block c is on top of block d. $move[obj = A, fr = B, to = C]$ represents the operation of moving A from the top of B to the top of C.

Module $sc[sit = M, op = OP]$ defines how the state of M is changed by operation OP. At the same time, the module identifier shows the history of state transitions.

For example, the following answers are obtained:

$$?\!-sc[sit = m, op = move[obj = a, fr = b, to = c]] :$$
$$X/[on = a].$$
Answer :$X = c$.

In this case, the module that represents the state after an operation is not included in the given program, it is possible to create new modules by adding a program to a query (Section 7) and by issuing a *create_module* command.

Concerning modifications made by the sequence of queries
and *create_module* commands, *QUIXOTE* employs transaction logic with special commands, *begin_transaction*, *end_transaction*, and *abort_transaction*. If some modules are created in one transaction, they are incrementally added to the program unless the transaction ends with *abort_transaction*.

---

[5]Precisely, Σ_m should be defined as the set of rules that are *properly* in m. Taking rule inheritance into account, the set of rules in a module is the union of the proper set and sets of rules imported from the other modules.

6 Relating Modules

It is important to relate some modules in defining the database and when reasoning.

Two ways of relating modules should be considered, that is, referring to other modules and importing/exporting rules from other modules.

As shown above, a rule of *QUIXOTE* has a subgoal of the form $m : A$ in its body. This subgoal specifies the external reference to the module with m as its identifier. In such a case, module m can be seen as *encapsulated*, because no rule is imported to it.

6.1 Simple Submodule Relationship

Sometimes, it is useful to define databases by providing a facility to import/export among modules as in typical object-oriented languages.

In *QUIXOTE*, importing/exporting rules are done by *rule inheritance* defined in terms of the binary relation $\sqsubseteq_S$ over modules called the *submodule* relation. The submodule relation is similar to the subsituation relation in PROSIT[15][6]. Basically, rule inheritance is defined as follows:

Definition 6 (Rule Inheritance)
If $m_1 \sqsupseteq_S m_2$ *then module* m_1 *inherits all the rules of* m_2, *that is, all the rules in* m_2 *are exported to* m_1.

Under this definition, the set of rules of m_1 is $\Sigma_{m_1} \cup \Sigma_{m_2}$.

The right hand side of $\sqsupseteq_S$ in a submodule definition may be a formula of module identifiers with set-theoretical union, intersection, or difference. For example, if we have

$$m_1 :: \{r_{11}, \cdots, r_{1i}\},$$
$$m_2 :: \{r_{21}, \cdots, r_{2j}\},$$
$$\{m_2, m_3\} :: \{r_{31}, \cdots, r_{3k}\},$$
$$m_1 \sqsupseteq_S m_2 - m_3$$

then m_1 has the set of rules

$$\{r_{11}, \cdots, r_{1i}, r_{21}, \cdots, r_{2j}\}.$$

Taking rule inheritance into account, a special module identifier *self* is also introduced as in most object-oriented programming languages. For example, consider the following example:

$$m_1 :: o \Rightarrow o_1 \parallel C.$$

The subgoal o_1 is interpreted as $self : o_1$. In this context, $self$ is evaluated as m_1. If $m \sqsupseteq_S m_1$, then m has the rule $m :: o \Rightarrow o_1 \parallel C$, and $self$ is evaluated as m in this case.

[6]Considering a module as a class, $m_1 \sqsupseteq_S m_2$ means that m_2 is a super-class of m_1.

6.2 Controlling Rule Inheritance

To treat various rule inheritance phenomena, two orthogonal concepts, *local* and *overriding*, are introduced into *QUIXOTE*. Each rule may have these modes, which control how each rule is inherited according to submodule relations.

If a rule is *local*, then it is not inherited to other modules. If a rule is *overriding*, then it overrides the other rules inherited from other modules, that is, the inheritance of some rules is canceled.

There are several possibilities on what rules are to be canceled by an overriding rule. Currently, the inheritance of a rule is canceled if its head has object terms with the same principal object and its labels are same as the one of the head of overriding rule. This is similar to the 'retract' predicate of Prolog.

Each rule has an *inheritance mode*. The value of the inheritance mode is (o), (l), or (ol), if explicitly specified. (o) means 'overriding, (l) means 'local', and (ol) means 'local and overriding'. If a rule has no inheritance mode, the rule is regarded as having 'non-local and no-overriding' as its default mode setting.

Consider the following example.

Example 5 (Exception by Inheritance Mode)

$$bird :: canfly/[pol = yes];;$$
$$penguin :: (ol)\ canfly/[pol = no];;$$
$$super\_penguin :: \{\ldots\};;$$
$$bird \sqsubseteq_S penguin \sqsubseteq_S super\_penguin;;$$

The inheritance of the rule of the module *bird* is canceled in the module *penguin* by its 'overriding' rule, whereas the module *super_penguin* gains $canfly/[pol = yes]$, because the rule in *bird* is inherited to it.

By introducing local and overriding modes for rule inheritance, it is possible to relate subsumption and submodule relations closely as follows:

$$penguin \sqsubseteq bird \supset penguin \sqsupseteq_S bird,$$

where rules in m_1 should be overridden.

6.3 Links between two Modules

Sometimes, a facility for representing changes of state is required as shown in the example in Section 5.2.

The relation between the two states before and after an operation is represented by a special form of object terms. However, simpler and more sophisticated treatment may be required for general treatment of state transitions or changes of states. The problem is how to *relate* modules and objects.

Another kind of relations called *links* are provided as follows:

$$m_1 \xrightarrow{L} m_2,$$
$$o_1 \xrightarrow{L} o_1,$$

where m_1 and m_2 are module identifiers, and o_1 and o_2 are object identifiers. L is called the name of a link relation. Notice that a link relation is defined over module identifiers and object terms. The former link is called *module-link* and the latter link is called *object-link*.

The link definition above obeys the following rule:

$$m_1 \xrightarrow{L} m_2 \ \wedge \ o_1 \xrightarrow{L} o_2 \ \wedge \ m_1 :: o_1 \quad \supset \quad m_2 :: o_2.$$

This rule shows how module-links and object-links colaborate. According to this rule, a pair of a module-link definition and an object-link definition can be transformed as follows:

$$m_2 :: X \Leftarrow m_1 : Y,\ m_1 \xrightarrow{L} m_2,\ Y \xrightarrow{L} X.$$

The following is an example of link usage:

$$m_1[agt = a] \xrightarrow{\text{turn-back}} m_2[agt = a]$$
$$to\_the\_right\_of[obj = b] \xrightarrow{\text{turn-back}} to\_the\_left\_of[obj = b].$$

This example means that b is to the right of an agent a in a module m_1, while b is to the left of a in m_2 after a turns back.

By traversing the used links, one can keep track of the stages of reasoning. This feature is especially important in assumption based reasoning and plan-goal based reasoning.

Most of the links appearing in the semantic network can be represented by labels in an attribute term, while some of the links accompanying inference are represented by the pairs of a module-link and an object-link.

7 Programs and Queries

As mentioned before, a *database* or a *program* is defined as a finite set of rules. More precisely, some additional information is associated with the definition of a database or a program.

A definition of a *QUIXOTE* program concept is defined as a 4-tuple (E, M_H, O_H, R) of the environment part E of the definition of macros and information on program libraries, the module part M_H of the definition of the submodule relation, the object part O_H of the definition of the lattice of basic objects, and a set of rules R[7]. The following is an example of a program definition.

$\&b\_pgm;;$
  $\&b\_env;;\ldots;;\&e\_env;;$
  $\&b\_obj;;$
    $\&subsum;;bird \sqsupseteq penguin,\ldots;;$
  $\&e\_obj;;$
  $\&b\_mod;;$
    $\&submod;;penguin \sqsupseteq_S bird,\ldots;;$
  $\&e\_mod;;$
  $\&b\_rule;;$
    $bird :: canfly/[pol = yes];;$
    $penguin :: color[arg = black\_white];;\ldots;;$
  $\&e\_rule;;$
$\&e\_pgm.$

[7] Precisely, M_H contains the definition of module-links, and O_H contains the definition of object-links.

A query is defined as a pair (A, P) of a set of attribute terms A and a program definition P $(=(E, M_H, O_H, R))$.

The purpose of this query is to find the answer to A in the context of adding P. Thus, a query (A, P) to a program P' $(=(E', M'_H, O'_H, R'))$ is the same as a query $(A, [])$ to a program $(E \cup E', M_H \cup M'_H, O_H \cup O'_H, R \cup R')$.

To deal with the modification of the program, a new transaction begins just before a query is processed and ends just after the process is terminated. *QUIXOTE* transactions can be nested, and the user can specify whether the modifications or updates done in each transaction are valid for successive processes or not.

This feature of adding a program fragment in a query extends the ability of the assumption-based reasoning in *QUIXOTE*, as shown in the following query, to the program above.

$?\!-\!super\_penguin : canfly/[pol = X];;$
$\&b\_pgm;;$
  $\&b\_mod;;\&submod;;$
    $penguin > -super\_penguin;;$
  $\&e\_mod;;$
  $\&b\_rule;;$
    $penguin :: (ol)canfly/[pol = no];;$
  $\&e\_rule;;$
$\&e\_pgm.$

8 Related Works

8.1 Objects and Properties

Beginning with Aït-Kaci's work on ψ-terms, there are a number of significant works on the formalization complex terms and feature structures [16, 13, 1, 3, 4]. Formalization of the object terms and attribute terms of *QUIXOTE* is closely related to and influenced by those works, especially the work done by Mukai on CIL [14] and CLP(AFA) [13].

Compared to those works, the unique point of *QUIXOTE* is its treatment of object identity that plays an important role in introducing object-orientedness into definite clause constraint languages.

As for object-orientation, Kifer's F-logic is closely related to *QUIXOTE*, although the treatment of object identity and property inheritance is quite different. In F-logic, object identity is not defined over complex terms but over normal first-order terms. The approach taken in QUIXOTE is more fine-grained than that of F-logic.

8.2 Modules

As module concepts are very important in knowledge representation as well as programming, several related works have been done [9, 10, 11, 15]. First, a brief comparison of the language features of these works is presented.

From the viewpoint of knowledge representation, modularization corresponds to the classification of knowledge. In such this sense, the ability to relate modules flexibly is important. QUIXOTE provides a number of ways to do this, for example, by specifying the nesting of modules. QUIXOTE supports multiple module nesting by allowing set-theoretical operators to relate modules, which are also used for exception handling, while other languages do not mention to it.

QUIXOTE also provides a facility for dealing with exceptions on exporting/importing rules by using the combination of modes associated with each rule (*local* and *overriding*). This covers the features described in [9, 10, 11].

Furthermore, as in most object-oriented languages, QUIXOTE introduces the special module identifier *self* which can be seen as a meta-level variable and plays an important role in rule inheritance, while other languages do not.

On the contrary, other languages have introduced the notion of side-effects mainly to make computation efficient. This is because the others are essentially designed as programming languages. This feature, including database updates, will be enhanced in the next version of QUIXOTE.

Concerning the semantics of modules and reasoning with modularized formulas, Gabbay [6] proposes a proof-theoretic framework for extending normal deductive systems called the Labeled Deductive System (LDS). In LDS, each formula is *labeled*, in the form of $t : A$, where t is a symbol called *label* and A is a logical formula. The consequence relation is replaced by:

$$t_1 : A_1, \ldots, t_n : A_n \vdash s : B.$$

In his *concatenation logic*, the following inference rule is the key to relating labeled formulas:

$$s : a, t : a \supset b \vdash (t + s) : b.$$

This means that b is obtained by using s first and then by using t. The label $(t + s)$ indicates the order of label use. This corresponds to the notion of links in QUIXOTE, explained in Section 6.3.

It is worthwhile investigating the relationship between LDS and QUIXOTE, namely to give a proof theory for QUIXOTE. This is work to be done in the future.

9 Concluding Remarks

Version 1.0 of QUIXOTE written in KL1 (designed by ICOT as a parallel language for parallel inference machines PIM), has been completed. It is used for several application systems, such as legal reasoning systems[19], natural language processing systems[18], and molecular biological databases[17]. Through those experiences, the usefulness of the features of QUIXOTE are being examined.

We are now working with the new version of QUIXOTE for more efficient representation and processing. In the new version, the following features are introduced:

1) Relation between Subsumption and Submodule
   This feature is discussed briefly at the end of Section 6.2.

2) Updates
   In Sections 5.2 and 6.3, we show a simple example of state transition. However, such problems are closely related to updates of databases or programs. Currently, only facts can be added or deleted. In the next version, the facility for adding or deleting non-unit clauses will be provided. The point is how to deal with those updates in a parallel processing environment without causing semantic problems.

3) Meta-Rule
   Meta-rules are useful both in programming languages and knowledge representation languages. They provide a facility to describe schemata to define generic procedures or knowledge.

   For example, in HiLog[5], the following general transitive closure rule can be written:

$$tc(R)(X,Y) :- R(X,Y).$$
$$tc(R)(X,Y) :- tc(R)(X,Z), tc(R)(Z,Y).$$

   In QUIXOTE, new variables corresponding to the principal objects of object terms must be introduced to support such a function.

Acknowledgement

We would like to express our gratitude to the members of the third laboratory of ICOT, and the members of the QUIXOTE project for their discussions and cooperation.

We are grateful to the members of the working groups of ICOT, STS (Situation Theory and Semantics) and NDB (New-generation DataBases) and IDB (Intelligent DataBases), for their stimulative discussions and useful comments.

We also would like to thank Dr. Kazuhiro Fuchi, Dr. Koichi Furukawa, and Dr. Shunichi Uchida of ICOT for their continuous encouragement.

References

[1] S. Abiteboul and S. Grumbach, "COL: A Logic-Based language for Complex Objects", *Proc. EDBT*, in *LNCS*, 303, Springer, 1988

[2] P. Aczel, *Non-Well-Founded Set Theory*, CSLI Lecture Notes No. 14, 1988.

[3] F. Bancilhon and S. Khoshahian, "A Calculus for Complex Objects", *Proc. ACM PODS*, 1985

[4] W. Chen and D. S. Warren, "Abductive Reasoning with Structured Data", *Proc. the North American Conference on Logic Programming*, pp.851-867, Cleveland (Oct., 1989).

[5] W. Chen, M. Kifer, and D. S. Warren, "HiLog as a Platform for Database Language", *Proc. the Second International Workshop on Database Programming Language*, Gleneden Beach, Oregon, 1989.

[6] D. Gabbay, "Labeled Deductive Systems, Part 1", CIS-Bericht-90-22, CIS, Universitat Munchen, Feb., 1991.

[7] M. Höhfeld and G. Smolka, "Definite Relations Over Constraint Languages", LILOG report 53, IBM Deutschland, Stuttgart, Germany, Oct., 1988.

[8] M. Kifer, G. Lausen, and J. Wu, "Logical Foundations for Object-Oriented and Frame-Based Languages", Technical Report 90/14 (revised), June, 1990.

[9] D. Miller, "A Theory of Modules for Logic Programming", *The International Symposium on Logic Programming*, 1986.

[10] L. Monterio and A. Porto, "Contextual Logic Programming", *The International Conference on Logic Programming*, 1989.

[11] L. Monterio and A. Porto, "A Transformational View of Inheritance in Programming", *The International Conference on Logic Programming*, 1990.

[12] Y. Morita, H. Haniuda, and K. Yokota, "Object Identity in *QUIXOTE*", *Proc. SIGDBS and SIGAI of IPSJ*, Oct., 1990.

[13] K. Mukai, "CLP(AFA): Coinductive semantics of horn clauses with compact constraints", In J. Barwise, G. Plotkin, and J.M. Gawron, editors, *Situation Theory and Its Applications, volume II*. CSLI Publications, Stanford University, 1991.

[14] K. Mukai, "Constraint Logic Programming and the Unification of Information", *PhD thesis*, Department of Computer Science, Faculty of Engineering, Tokyo Institute of Technology, 1991.

[15] H. Nakashima, H. Suzuki, P-K. Halvorsen, S. Peters, "Towards a Computational Interpretation of Situation Theory", *The International Conference on Fifth Generation Computer Systems*, 1988.

[16] G. Smolka, "Feature logic with subsorts", Technical Report LILOG Report 33, IWBS, IBM Deutschland GMBH, W. Germany, 1989.

[17] H. Tanaka, "Protein Function Database as a Deductive and Object-Oriented Database", *The Second International Conference on Database and Expert System Applications*, Berlin, Apr., 1991.

[18] S. Tojo and H. Yasukawa, "Temporal Situations and the Verbalization of Information", *The Third International Workshop on Situation Theory and Applications (STA3)*, Oiso, Nov., 1991.

[19] N. Yamamoto, "TRIAL: a Legal Reasoning System (Extended Abstract)", *France-Japan Joint Workshop*, Renne, France, July, 1991.

[20] H. Yasukawa and K. Yokota, "Labeled Graphs as Semantics of Objects", *Proc. SIGDBS and SIGAI of IPSJ*, Oct., 1990.

[21] K. Yokota and H. Yasukawa, "*QUIXOTE*: an Adventure on the Way to DOOD (Draft)", *Workshop on Object-Oriented Computing '91*, Hakone, Mar., 1991.