한국지능정보시스템학회논문지 제9권 제1호 2003년 6월(pp.251~267)

Intelligent Query Processing in Deductive and Object-Oriented Databases

Yang Hee Kim
Division of Liberal Art, Korea National Sport University
(yangh-kim@knupe.ac.kr)

In order to satisfy the needs of an intelligent information system, it is necessary to have more intelligent query processing in an object-oriented database. In this paper, we present a method to apply intelligent query processing in object-oriented databases using deductive approach. Using this method, we generate intelligent answers to represent the answer-set abstractly for a given query in object-oriented databases. Our approach consists of four stages: rule representation, rule reformation, pre-resolution, and resolution. In rule representation, a set of deductive rules is generated based on an object-oriented database schema. In rule reformation, we eliminate the recursion in rules. In pre-resolution, rule transformation is done to get unique intensional literals. In resolution, we use SLD-resolution to generate intensional answers.

Key Words : Object-Oriented Databases, Deductive Database, Intelligent Query Processing, Intensional Answers, SLD-resolution

논문접수일 : 2003년 2월 게재확정일 : 2003년 6월 교신저자 : 김양희

1. Introduction

When querying an object-oriented database, a set of objects (extensional answers) is usually returned to users as an answer set (Bancilhon, 1988). But these objects may belong to different classes within a class hierarchy or they may be different complex objects somehow related to each other. In a certain query, it is necessary to find the set of rules which characterizes the conditions for an object to satisfy in order to belong to an answer to the query. Object-oriented databases do not support such capability. However, in a deductive database, users can get the answer of a query as not only a set of facts but also a set of formulas (intensional answers). Deductive databases can generate a set of first order logic formulas as an answer set for a given query.

Intensional query processing has many advantages. Intensional answers are given as a set of formulas which are independent of particular circumstances in the database. Not only does an

intensional answer represent the answer to the given query in a more compact way, but it can be computed much faster than extensional answers. Most of the time intensional answers can be computed only using the rules without accessing the database. For a detailed description of the intensional query processing, we refer to (Yoon et al., 1994).

In this paper, we introduce a method to apply the intensional query processing techniques of deductive databases to object-oriented databases. By introducing rules into object-oriented database systems and apply the intensional query processing techniques to object-oriented database systems. All the query languages in object-oriented database systems known to us are not able to handle incomplete queries. An incomplete query is a query on the attribute which belongs to a subclass but not to the base class. In this paper, we make it possible to answer incomplete queries by representing systems schema in terms of rules. Conventionally, the answer-set of a query in systems is represented as a set of objects. But, the presence of semantics in systems schema and intensional query processing methodologies enable us to express the answer-set abstractly as names of classes. In this paper, we present an algorithm to obtain abstract representation of a given answer-set. It provides us better understanding of the answer.

Rules which represent object-oriented database schema consist of structural rules and subclassing rules. The structural rules in turn consist of "IS_A"-relationships representing the class hierarchy. The subclassing rules represent characteristic properties of subclasses. We then transform all the rules to non-recursive Horn clauses and get an intensional answer by using SLD-resolution.

We provide some sample queries to show the advantages of an intensional answer to give query in an object-oriented database system over conventional answers.

This paper is organized as follows. In section 2, we discuss several previous works for intensional query processing. In section 3, we review the definition of intensional answers and methods for deriving intensional answers. In section 4, we look at the "IS_A"-relationship in object-oriented database systems. In section 5, we present an approach to convert an object-oriented schema based on class hierarchy into non-recursive Horn clauses and generate intensional answers from logical consequences of non-recursive Horn clauses and a given query. In section 6, we give a detailed example. In section 7, we give conclusions and some remarks for possible extensions of the method in this paper.

2. Related Works

A deductive database is a database in which new facts may be derived from the facts that were explicitly stored by using an inference system. A deductive databases composed of extensional database (EDB) and intensional database (IDB). Cholvy and Demolombe (1986) studied the idea of having a set of formulas as an answer set. Their

answers were a set of formulas defining the conditions that are independent of a particular set of facts. They also developed a method for generating answers using resolution.

Pascual and Cholvy (1988) improved an algorithm based on the research by Cholvy and Demolombe. They dealt with only Horn clauses to avoid many steps for generating the answers. But they did not discussed the detailed steps to remove meaningless answers. Song and Kim (1991) solved this problems. They discussed intensional query processing scheme based on SLD-resolution and discussed an implementation fo an intensional query processing in Prolog.

Pirotte et al. (1991) used integrity constraints to filter out improper answers. Yoon et al. (1995) used only Horn-clauses for intensional databases and use a SLD-resolution which takes advantages of Horn-clauses. They introduced the notions of extended term-restricted rules, relevant literals and relevant clauses to avoid generating certain meaningless intensional answers.

Also, Yoon and Park (1999) introduced a partially automated method for generating intensional answers at multiple abstraction levels for a query. Pontieri et al. (2002) propose a data source integration approach. The proposed approach consists of two components, performing intensional and extensional integration, respectively; these are strictly coupled, since they use related models for representing intensional and extensional information and are synergic in their behaviour.

3. Definition of Intensional Answers

In this section, we give a formal definition of intensional answers. A formal definition of intensional answers is given by Cholvy and Demolombe (1986). We define T as the database theory consisting of a set of facts and rules. Let $Q(X)$ be a query where X is a tuple of free variables. Then, the intensional answer $ANS(Q)$ to a certain query $Q(X)$ is defined as follows:

$$ANS(Q) = \{ans(X) \sim : \sim T \sim \vdash \sim \forall X \sim (ans(X) \rightarrow Q(X))\}$$

where $ans(X)$ is a literal.

However, we want to restrict the answers within a defined domain of interest. Here are some restriction on the intensional answer set.

So, let $DP = \{P_1, \cdots, P_n\}$ be set of predicate symbols either of the IDB or EDB. And let $L(DP)$ be the first order language whose predicate symbols are $P_1, \cdots, P_n$. Then define an intensional answer $ANS(Q, DP)$ to the query $Q(X)$ by :

$$\begin{aligned} ANS(Q, DP) = \; & \{ans(X) \sim : asn(X) \in L(DP) \sim \text{and} \\ & \sim T \sim \vdash \sim \forall X \sim (ans(X) \rightarrow Q(X)) \sim \text{and} \\ & (ans(X) \sim is \sim \backslash not \sim the \sim negation \sim of \sim a \sim tautology) \sim \text{and} \\ & (each \sim ans(X) \sim is \sim \backslash not \sim redundant)\} \end{aligned}$$

We note that :

$$\begin{aligned} & \sim T \sim \vdash \sim \forall X \sim (ans(X) \rightarrow Q(X)) \sim \\ \Leftrightarrow \; & T \sim \bigcup \sim \{\text{not}(\forall X \sim (ans(X) \rightarrow Q(X)))\} \sim is \sim \text{inconsistent} \\ \Leftrightarrow \; & T \sim \bigcup \sim \{\exists X(ans(X) \wedge \text{not}(Q(X)))\} \sim is \sim \text{inconsistent} \end{aligned}$$

Let S be a set of clauses that represent the standard clausal form of T axioms (Chang and Lee, 1973). Then

$$\exists X \sim (ans(X) \wedge \text{not}(Q(X)))$$

leads the standard form $ans(X_0) \wedge \text{not}(Q(X_0))$ where X_0 is a tuple of Skolem constants. So, the above formula is equivalent to

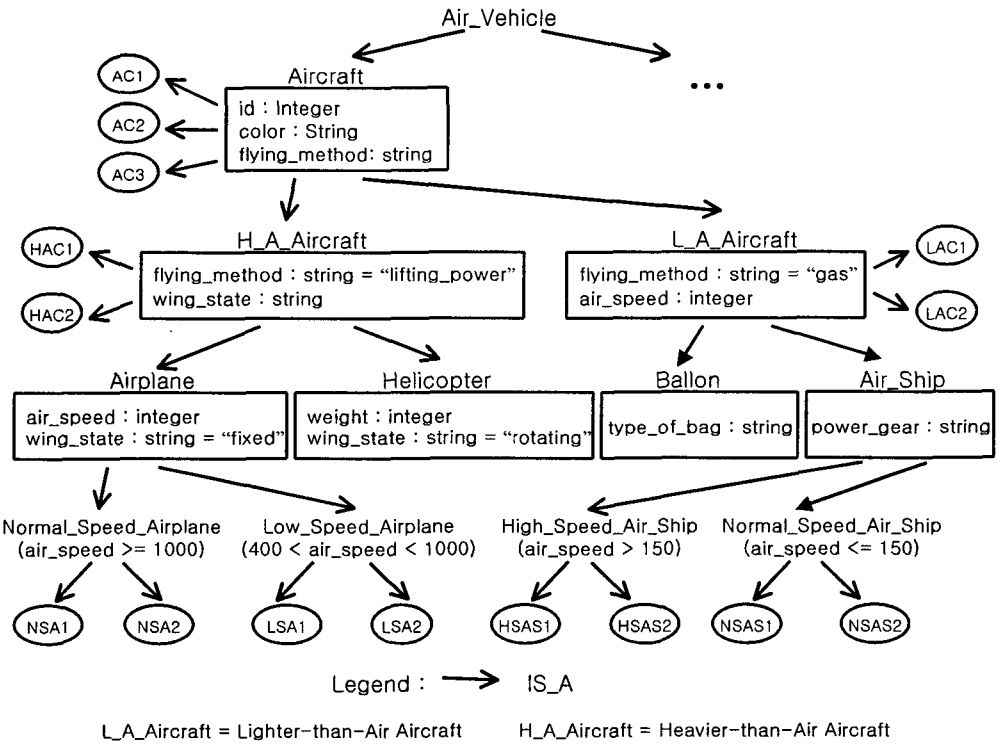
$$S \sim \bigcup \sim \{ans(X_0), \neg(Q(X_0))\} \sim is \sim \text{inconsistent}.$$

Therefore, answer formulas $ans(X)$ are such that resolution on $S \sim \bigcup \sim \{ans(X_0), \neg(Q(X_0))\}$ leads to the empty clause.

However, initially we do not know what $ans(X_0)$ are. So, for the resolution processing, we will start with $S \sim \bigcup \sim \{\neg Q(X_0)\}$. After resolving $S \sim \bigcup \sim \{\neg Q(X_0)\}$ will result in a resolvent $R(X_0)$ and then resolving $R(X_0)$ together with $ans(X_0)$ will result in the empty clause. That means that $ans(X_0)$ must equal to $\neg R(X_0)$.

4. Class Hierarchy

In this section, we look at a class hierarchy, representing "IS_A"-relationships between the different classes. The class hierarchy represented in the <Figure 1> will be our object-oriented example database.



<Figure 1> Class Hierarchy

A typical incomplete query for the class hierarchy in <Figure 1> is the following:

SELECT Aircraft.id
WHERE air_speed > 150

4.1 Conventional Query Processing

In a conventional object-oriented database we would get back a set of aircraft ids where the air speed of the aircraft is greater than 150 kilometers per hour. According to our particular database in <Figure 1>, we get back the following set:

{NSA1, NSA2, ···, LSA1, LSA2, ···, HSAS1, HSAS2, ··· }.

All these objects belong to different subclasses of the base class object. In order to find all the applying objects, the database must provide a technique to search through all the subclasses of aircraft. But there do not exist good query languages for OODB systems which are simple to use the advantages of an object-oriented system.

4.2 Intensional Query Processing

Using intensional query processing methodologies, the answer-set of the same query will be different. The intensional answer will consist of several formulas characterizing the different classes the answer objects come from.

In our example there would be the following intensional answers:

- intensional-answer1 = all normal_speed_airplane
- intensional-answer2 = all low_speed_airplane
- intensional-answer3 = all high_speed_air_ship

In the next section of this paper, we will look at a way to automatically access the desired subclasses without being aware of the exact structure of the class hierarchy.

5. Formalization of Intelligent Query Processing

In this section, we present the intelligent query processing. We divide our approach into four stages : rule representation, rule reformation, pre-resolution, and resolution. The four phases are partitioned into two categories : processing that can be done statically once and processing that has to be performed at run time. Rule representation, rule reformation and pre-resolution belong to the first category and the last phases belong to the second category.

5.1 Rule Representation

In the first phase, we execute two steps. In the first step, we generate a set of deductive rules based on the hierarchy in an object-oriented database. We can classify rules into two different categories : structural rules and subclassing rules. The rules in the first category come out of the class hierarchy of the objects. For example, there is a class Aircraft having a aircraft type, say

H_A_Aircraft as subclass. Then we have the rule: IS_A(H_A_Aircraft, Aircraft). And the rules in the second category represent the semantic knowledge between classes. For example, there are two classes, Airplane and Normal_Speed_Airplane. If the air speed is greater than or equal to 1000, the that airplane belongs to subclass Normal_Speed_ Airplane. This semantic knowledge involves two classes of Airplane and Normal_Speed_Airplane.

The structural rules and subclassing rules of our given example database would be the followings :

· structural rules
- IS_A(H_A_Aircraft, Aircraft)
- IS_A(L_A_Aircraft, Aircraft)
- IS_A(Airplane, H_A_Aircraft)
- IS_A(Helicopter, H_A_Aircraft)
- IS_A(Ballon, L_A_Aircraft)
- IS_A(Air_Ship, L_A_Aircraft)
- IS_A(Normal_Speed_Airplane, Airplane)
- IS_A(Low_Speed_Airplane, Airplane)
- IS_A(High_Speed_Air_Ship, Air_Ship)
- IS_A(Normal_Speed_Air_Ship, Air_Ship)

· subclassing rules
- Normal_Speed_Airplane(X) ← *Airplane(X)* ∧ air_speed(X, Y)∧greatereq(Y, 1000)
- Low_Speed_Airplane(X) ← *Airplane(X)* ∧ air_speed(X, Y)∧greater(Y, 400) ∧ less(Y, 1000)
- Airplane(X) ← *H_A_Aircraft(X)* ∧ wing_state(X, Y) ∧ equal(Y, "fixed")
- Helicopter(X) ← *H_A_Aircraft(X)* ∧ wing_state(X, Y) ∧ equal(Y, "rotating")
- High_Speed_Air_Ship(X) ← *Air_Ship(X)* ∧ air_speed(X, Y) ∧ greater(Y, 150)
- Normal_Speed_Air_Ship(X) ← *Air_Ship(X)* ∧ air_speed(X, Y) ∧ lesseq(Y, 150)
- H_A_Aircraft(X) ← *Aircraft(X)* ∧ flying_method(X, Y) ∧ equal(Y, "lifting_power")
- L_A_Aircraft(X) ← *Aircraft(X)* ∧ flying_method(X, Y) ∧ equal(Y, "gas")

And in the second step, we introduce EDB literals to represent the objects that belong to each class. The EDB literals to represent objects that belongs to our example database would be the followings :

EDB

Aircraft(id, color, flying_method)
H_A_Aircraft(id, color, flying_method, wing_state)
Airplane(id, color, flying_method, wing_state, air_speed)
Normal_Speed_Airplane(id, color, flying_method, wing_state, air_speed)
Low_Speed_Airplane(id, color, flying_method, wing_state, air_speed)
Helicopter(id, color, flying_method, wing_state, weight)
L_A_Aircraft(id, color, flying_method, air_speed)
Ballon(id, color, flying_method, air_speed, type_of_bag)
Air_Ship(id, color, flying_method, air_speed, power_gear)

High_Speed_Air_Ship(id, color,
flying_method, air_speed, power_gear)
Normal_Speed_Air_Ship(id, color,
flying_method, air_speed, power_gear)

5.2 Rule Reformation

In the rule reformation steps, we remove literals that will cause recursion in rules. By limiting non-recursive clauses the algorithm can be terminated, and by Horn clauses efficient algorithm can be used. To compute intensional answers efficiently, subclassing rules should be represented in a proper form. Since testing satisfiability in first-order logic formula is undecidable, adopting first-order logic formula for managing subclassing rules is not desirable. Therefore, we need a subset of first order logic expressions which is powerful enough for expressing subclassing rules and in which the satisfiability problem can be processed efficiently.

Subclassing rules can be represented with the "simple predicates". The BNF of simple predicate abbreviated by SP is as follows:

⟨*SP*⟩ :: = ⟨*SP*⟩ ∧ ⟨*SP*⟩ | ⟨*SP*⟩ ∨ ⟨*SP*⟩ | ¬⟨*SP*⟩ | ⟨*predicates*⟩

⟨*predicates*⟩ :: = ⟨*comparison operator*⟩(⟨*variable name*⟩, ⟨*constant*⟩)
| ⟨*comparison operator*⟩(⟨*variable name*⟩, ⟨*variable name*⟩)
| ⟨*comparison operator*⟩(⟨*variable name*⟩,
⟨*variable name*⟩ + ⟨*constant*⟩)

⟨*comparison operator*⟩:: = *equal* | not_*equal* | *greater*
| *greater_eq* | *less* | *less_eq*

Rosencrantz and Hunt showed that the satisfiability problem of the set of simple predicate is NP-hard (Rosencrantz and Hunt, 1980). But they showed that conjunctive not_equal free predicates (simple predicates that do not contain not_equal and $\vee$) can be solved in polynomial time. We can represent a large class of subclassing rules with conjunctive not_equal free predicates. The following algorithm changes a conjunctive not_equal free predicate to a weighted directed graph (Motro and Yoon, 1990).

Algorithm 1

Input : A conjunctive not_equal free predicate P.
Output : A weighted directed graph.
(v_1 and v_2 stand for variables and c stands for a constant)

1. Convert P into an equivalent predicate P' containing only less_eq comparison literal as follows :
   1.1 Replace $v_1 = v_2$ with $(v_1 \le v_2 + 0) \wedge (v_2 \le v_1 + 0)$.
   1.2 Replace $v_1 < v_2$ with $v_1 \le v_2 + (-1)$.
   1.3 Replace $v_1 \le v_2$ with $v_1 \le v_2 + 0$.
   1.4 Replace $v_1 > v_2$ with $v_2 \le v_1 + (-1)$.
   1.5 Replace $v_1 \ge v_2$ with $v_2 \le v_1 + 0$.
   1.6 Replace $v_1 = c$ with $(v_1 \le 0 + c) \wedge (0 \le v_1 + (-c))$.
   1.7 Replace $v_1 < c$ with $v_1 \le 0 + (c-1)$.
   1.8 Replace $v_1 \le c$ with $v_1 \le 0 + c$.
   1.9 Replace $v_1 > c$ with $0 \le v_1 + (-c-1)$.
   1.10 Replace $v_1 \ge c$ with $0 \le v_1 + (-c)$.
   1.11 Replace $v_1 = v_2 + c$ with $(v_1 \le v_2 + c) \wedge (v_2 \le v_1 + (-c))$.

1.12 Replace $v_1 < v_2 + c$ with $v_1 \leq v_2 + (c-1)$.

1.13 Replace $v_1 \leq v_2 + c$ with $v_1 \leq v_2 + c$.

1.14 Replace $v_1 > v_2 + c$ with $v_2 \leq v_1 + (-c-1)$.

1.15 Replace $v_1 \geq v_2 + c$ with $v_2 \leq v_1 + (-c)$.

2. Convert P' into a weighted directed graph. The graph has a node for each variable and a node for a constant zero. Conversion is as follows:

2.1 $v_1 \leq v_2 + c$ corresponds to an edge from node v_1 to node v_2 with edge weight c.

2.2 $v_1 \leq 0 + c$ corresponds to an edge from node v_1 to zero node with edge weight c.

2.3 $0 \leq v_1 + c$ corresponds to an edge from zero node to node v_1 with edge weight $-c$.

There are two restrictions in the above algorithm. The one is that each variable should be integer valued. The other is that predicates can not have not_equal operators. Fortunately, many of subclassing rules involve integer valued domains such as engine size, price, number of doors, etc. And in this paper, we will deal with not_equal free predicates.

The next algorithm will change a weighted directed graph G with no multiple edges to a conjunctive less_eq predicate(not_equal free predicate that contains only less_eq).

Algorithm 2

Input : A weighted directed graph G with no multiple edges.

Output : A conjunctive less_eq predicate.

(v_1 and v_2 stand for variables and c stands for a constant)

1. An edge from node v_1 to node v_2 with edge weight c corresponds to $v_1 \leq v_2 + c$.
2. An edge from node v_1 to zero node with edge weight c corresponds to $v_1 \leq 0 + c$.
3. An edge from zero node to node v_1 with edge weight $-c$ corresponds to $0 \leq v_1 + c$.

The next algorithm will test comparison literals using a weighted directed graph and return a truth constant or a simplified predicate.

Algorithm 3

Input : A predicate consisting of the conjunction of an old comparison predicate (predicate in resolvent before resolution) and a new comparison predicate (predicate in resolvent after resolution).

Output : Truth constant(TRUE of FALSE) or simplified predicate.

1. Apply algorithm 1 to the conjunction of old and new comparison predicate. And then we get a weighted directed graph G.
2. If G has a negative cycle then
   return FALSE.
   else
   If there is more than one edge from node v_1 to node v_2 then

```
        begin
        retain the minimum weight edge
        and discard the others
          apply algorithm 2 to G and we get
          a conjunctive less_eq predicate
          using
            step 1 of algorithm 1
        if P is the same as new comparison
        predicate then
            return TRUE
        else
            return P
        end
    else
        return old comparison predicate ∧
        new comparison predicate
```

We can use Floyd's all shortest path algorithm to see if the graph has a negative weight cycles. In algorithm 3, the step 1 can be processed in a linear time, if-part of the step 2(Floyd's all shortest path algorithm) takes $O(k^3)$ and else-part of the step 3 can be processed in a linear time where k is a number of node in G.

In the rule reformation step, we need to eliminate literals that will occur recursion. After that elimination, we will get the following reformed rules in our example database.

· reformed subclassing rules
- Normal_Speed_Airplane(X) ← air_speed(X, Y) ∧ greatereq(Y, 1000)
- Low_Speed_Airplane(X) ← air_speed(X, Y) ∧ greater(Y, 400) ∧ less(Y, 1000)
- Airplane(X) ← wing_state(X, Y) ∧ equal(Y, "fixed")
- Helicopter(X) ← wing_state(X, Y) ∧ equal(Y, "rotating")
- High_Speed_Air_Ship(X) ← air_speed(X, Y) ∧ greater(Y, 150)
- Normal_Speed_Air_Ship(X) ← air_speed(X, Y) ∧ lesseq(Y, 150)
- H_A_Aircraft(X) ← flying_method(X, Y) ∧ equal(Y, "lifting_power")
- L_A_Aircraft(X) ← flying_method(X, Y) ∧ equal(Y, "gas")

5.3 Pre-Resolution

In the first and second step some rule reformations should be done in order to get unique intensional literals.

Unique intensional literals means that a literal should either be extensionally or intensionally defined but not both. We can always get rid of this equality of names by renaming the extensional literal to p^* and introduce a new rule $p \leftarrow p^*$. For example, we have the following new rules : Aircraft(X) ← Aircraft*(X), Airplane(X) ← Airplane*(X) and Helicopter(X) ← Helicopter*(X). In doing so, we can handle complete queries as well as incomplete queries for the intensional query processing.

Next we can change "IS_A"-rules to the first-order logic since the semantic of "IS_A" is implication. For example, $IS\_A(X, Y)$ can be changed $Y \leftarrow X$. Now, IDB corresponds to a set

of non-recursive Horn clauses.

5.4 Resolution

In this step, we use SLD resolution to compute intensional answers. The following algorithm will find intensional answers from a set of non-recursive Horn clauses consisting of $EDB \cup IDB$ and a query $Q(X)$.

Algorithm 4

Input : A set of non-recursive Horn clauses consisting of $EDB \cup IDB$ and a query $Q(X)$ where X is a tuple of free variables.

Output : A set of $ANS_I(X)$ of intensional answers.

1. Negate the query and convert it into the clause form
2. Repeat for all branches of a resolution tree
   2.1 Perform resolution using subclassing rules, structural rules or new rules
   2.2 If resolvent contains extensional literal p^* then
      If base literal is not in the attributes of p^* then
         current branch is fail branch and return
            current branch is fail branch and return
         else if result is FALSE then
            current branch is success branch and return
            $ANS_I(X) = selected\ predicate$
         else
            current branch is success branch and return
            $ANS_I(X) =$ *selected predicate* $\wedge$ simplifi*d predicate*

   Until it cannot be further resolved
3. If all success branches contain extensional answers then
      begin
      choose the highest success branch
      generate the intensional answers by negating resolvent
      end
   else
      begin
      ignore success branch containing extensional answers
      return intensional answers $ANS_I(X)$
      end

6. Example

In this section, we introduce an example to show the application of our approach and algorithm introduced in the section above. To show the application of the algorithm introduced in the previous section, we will use our example database given in <Figure 1>.

EDB and IDB schema looks as follows :

EDB

as in section 5.1

IDB

structural rules

subclassing rules

Frist we rename the extensional literal, add new rules in the IDB, remove recursion, and change structural rules to the first order logic:

EDB

Aircraft*(id, color, flying_method)

H_A_Aircraft*(id, color, flying_method, wing_state)

Airplane*(id, color, flying_method, wing_state, air_speed)

Normal_Speed_Airplane*(id, color, flying_method, wing_state, air_speed)

Low_Speed_Airplane*(id, color, flying_method, wing_state, air_speed)

Helicopter*(id, color, flying_method, wing_state, weight)

L_A_Aircraft*(id, color, flying_method, air_speed)

Ballon*(id, color, flying_method, air_speed, type_of_bag)

Air_Ship*(id, color, flying_method, air_speed, power_gear)

High_Speed_Air_Ship*(id, color, flying_method, air_speed, power_gear)

Normal_Speed_Air_Ship*(id, color, flying_method, air_speed, power_gear)

IDB

· structural rules

Aircraft(X) ← H_A_Aircraft(X)

Aircraft(X) ← L_A_Aircraft(X)

H_A_Aircraft(X) ← Airplane(X)

H_A_Aircraft(X) ← Helicopter(X)

L_A_Aircraft(X) ← Ballon(X)

L_A_Aircraft(X) ← Air_Ship(X)

Airplane(X) ← Normal_Speed_Airplane(X)

Airplane(X) ← Low_Speed_Airplane(X)

Air_Ship(X) ← High_Speed_Air_Ship(X)

Air_Ship(X) ← Normal_Speed_Air_Ship(X)

· subclassing rules

Normal_Speed_Airplane(X) ← air_speed(X, Y) ∧ greatereq(Y, 1000)

Low_Speed_Airplane(X) ← air_speed(X, Y) ∧ greater(Y, 400) ∧ less(Y, 1000)

Airplane(X) ← wing_state(X, Y) ∧ equal(Y, "fixed")

Helicopter(X) ← wing_state(X, Y) ∧ equal(Y, "rotating")

High_Speed_Air_Ship(X) ← air_speed(X, Y) ∧ greater(Y, 150)

Normal_Speed_Air_Ship(X) ← air_speed(X, Y) ∧ lesseq(Y, 150)

H_A_Aircraft(X) ← flying_method(X, Y) ∧ equal(Y, "lifting_power")

L_A_Aircraft(X) ← flying_method(X, Y) ∧ equal(Y, "gas")

· new rules

Aircraft(X) ← Aircraft*(X)

H_A_Aircraft(X) ← H_A_Aircraft*(X)

Airplane(X) ← Airplane*(X)
Helicopter(X) ← Helicopter*(X)
Normal_Speed_Airplane(X) ←
  Normal_Speed_Airplane*(X)
Low_Speed_Airplane(X) ←
  Low_Speed_Airplane*(X)
L_A_Aircraft(X) ← L_A_Aircraft*(X)
Ballon(X) ← Ballon*(X)
Air_Ship(X) ← Air_Ship*(X)
High_Speed_Air_Ship(X) ←
  High_Speed_Air_Ship*(X)
Normal_Speed_Air_Ship(X) ←
  Normal_Speed_Air_Ship*(X)

Now let us consider the query "Find a set of aircraft ids where the air speed of the aircraft is greater than 150 kilometers per hour". The query can be written us

$$Q(X) = \text{Aircraft}(X) \wedge \text{air\_speed}(X, Y) \wedge \text{greater}(Y, 150).$$

Thus the goal clause is

(G) ← Aircraft(X), air_speed(X, Y), greater(Y, 150).

After four phases, we get the following resolvents:

(R1) ← Aircraft*(X), air_speed(X, Y), greater(Y, 150)
(R2) ← H_A_Aircraft(X), air_speed(X, Y), greater(Y, 150)
(R3) ← L_A_Aircraft(X), air_speed(X, Y), greater(Y, 150)

(R1-1) ← fail

(R2-1) ← H_A_Aircraft*(X), air_speed(X, Y), greater(Y, 150)
(R2-2) ← Airplane(X), air_speed(X, Y), greater(Y, 150)
(R2-3) ← Helicopter(X), air_speed(X, Y), greater(Y, 150)
(R2-4) ← fail
(R2-5) ← Airplane*(X), air_speed(X, Y), greater(Y, 150)
(R2-6) ← wing_state(X, Y), equal(Y, "fixed"), air_speed(X, Y), greater(Y, 150)
(R2-7) ← Normal_Speed_Airplane(X), air_speed(X, Y), greater(Y, 150)
(R2-8) ← Low_Speed_Airplane(X), air_speed(X, Y), greater(Y, 150)
(R2-9) ← success
(R2-10) ← wing_state(X, Y), equal(Y, "rotating"), air_speed(X, Y), greater(Y, 150)
(R2-11) ← success
(R2-12) ← fail
(R2-13) ← Normal_Speed_Airplane*(X), air_speed(X, Y), greater(Y, 150)
(R2-14) ← air_speed(X, Y), greater_eq(Y, 1000), air_speed(X, Y), greater(Y, 150)
(R2-15) ← Low_Speed_Airplane*(X), air_speed(X, Y), greater(Y, 150)
(R2-16) ← air_speed(X, Y), greater(Y,400),

less(Y,1000), air_speed(X, Y), greater(Y,150)

(R2-17) ← success

(R2-18) ← $ANS^1_1(X)$ = Normal_Speed_Airplane(X)

(R2-19) ← success

(R2-20) ← $ANS^2_1(X)$ = Low_Speed_Airplane(X)

(R3-1) ← L_A_Aircraft*(X), air_speed(X, Y), greater(Y, 150)

(R3-2) ← Ballon(X), air_speed(X, Y), greater(Y, 150)

(R3-3) ← Air_Ship(X), air_speed(X, Y), greater(Y, 150)

(R3-4) ← success

(R3-5) ← success

(R3-6) ← Air_Ship*(X), air_speed(X, Y), greater(Y, 150)

(R3-7) ← High_Speed_Air_Ship(X), air_speed(X, Y), greater(Y, 150)

(R3-8) ← Low_Speed_Airplane(X), air_speed(X, Y), greater(Y, 150)

(R3-9) ← success

(R3-10) ← success

(R3-11) ← air_speed(X, Y), greater(Y, 150), air_speed(X, Y), greater(Y, 150)

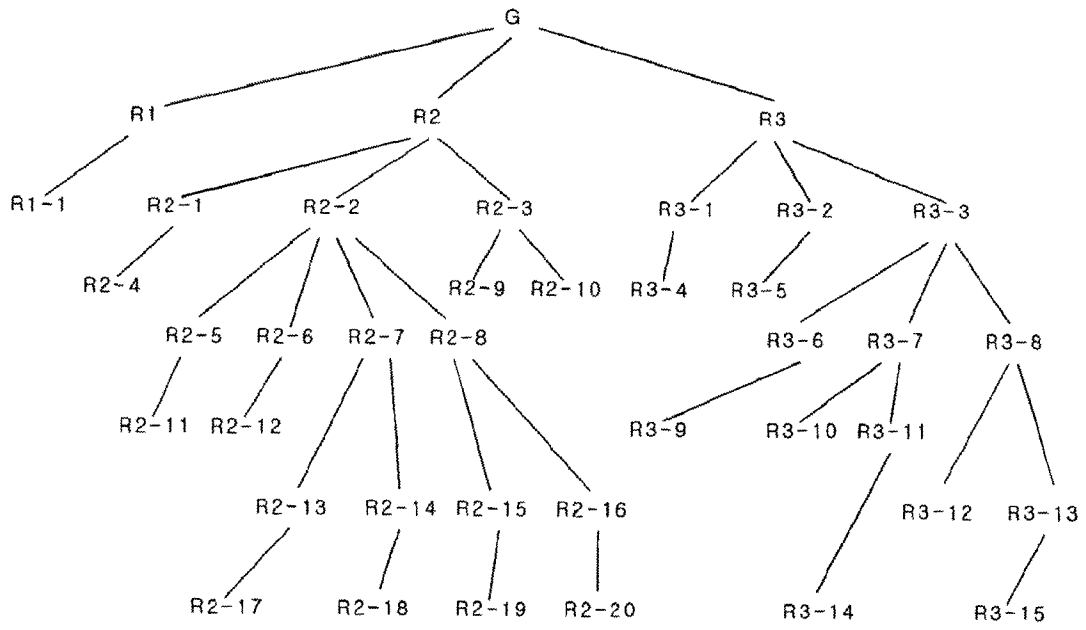
(R3-12) ← success

(R3-13) ← air_speed(X, Y), less_eq(Y, 150), air_speed(X, Y), greater(Y, 150)

(R3-14) ← $ANS^3_1(X)$ = High_Speed_Air_Ship(X)

(R3-15) ← fail

So, the resolution tree is as follows.



<Figure 2> Resolution Tree

Since there are three branches that have intensional answers, we have following intensional answers :

- $ANS_I^1(X)$ = Normal_Speed_Airplane(X)
- $ANS_I^2(X)$ = Low_Speed_Airplane(X)
- $ANS_I^3(X)$ = High_Speed_Air_Ship(X)

7. Conclusion

In this paper, we have presented a method to obtain more meaningful answers (intensional answers) to queries in object-oriented databases using deductive approach. By introducing rules into object-oriented databases and applying the intensional query processing techniques of deductive database to the object-oriented databases systems, we are able to use the advantages of the semantics of object-oriented databases schema.

Our approach consists of four stages: rule representation, rule reformation, pre-resolution, and resolution. In rule representation, a set of deductive rules is generated based on an object-oriented database schema. In rule reformation, we eliminate the recursion in rules. In pre-resolution, rule transformation is done to get unique intensional literals. In resolution, we use SLD-resolution to generate intensional answers. Since our method discovers all of the intensional answers, it is complete.

In this paper, we only obtain intensional answers for a class hierarchy model but not for a class-composition hierarchy. Our approach does not seem to be powerful enough to represent a complex object hierarchy. It is probable that we need more powerful logic for reasoning intensional answers on complex objects.

References

F. Bancilon, *Object-Oriented Database Systems*, Proceedings of Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin, Texas, March 21-23(1988), 152-162.

C. Chang and R. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York and London, 1973.

L. Cholvy and R. Demolombe, *Querying a Rulebase*, Proceedings of the first int'l conference on Expect Database Systems, ed. Kerschberg, L., Charleston, South Carolina, April 1-4(1986), 365-371.

Parke Godfrey and Jarek Gryz, *Overview of Dynamic Query Evaluation in Intensional Query Optimization*, Proceedings of the 5th DOOD, Montreux, Switzerland, December(1997), 425-426.

A. Motro and Q. Yuan, *Querying Database Knowledge*, Proceedings of ACM SIGMOD, Atlantic City, New Jersey, May 23-25(1990), 173-183.

A. Motro, *Using Integrity Constraints to Provide Intensional Answers to Relational Queries*, Proceedings of 15th VLDB Conference, (1989), 237-246.

E. Pascual and L. Cholvy, *Answering Queries Addressed to the Rulebase of a Deductive Database*, Proceedings of 2nd Int'l Conference on Information Processing and Management of

Uncertainty in Knowledgebased Systems, Urbino, Italy, July(1988), 138-145.

A. Pirotte, D. Roelants and E. Zimanyi, "Controlled Generation of Intensional Answers", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No. 2 (1991), 221-236.

L. Pontieri, D. Ursino and E. Zumpano, *An Approach for Synergically Carrying out Intensional and Extensional Integration of Data Sources Having Different Formats*, LNCS 2348 (2002), 752-756.

D. J. Rosenkrantz and M. B. Hunt, *Processing Conjunctive Predicates and Queries*, Proceedings of the Sixth International Conference on VLDB, Montreal, (1980), 64-74.

I-Y. Song and H-J. Kim, "Design and Implematation of a Three-Step Intensional Query Processing in Prolog", *Jounal of Data Administration*, Vol. 2, No.2 (1991), 23-25.

J. Ullman, *Principles of Database and Knowledge-base Systems*, Computer Science Press, 1988.

S. C. Yoon, I. Y. Song and E. K. Park, *Intelligent Query Answering in Deductive and Object-Oriented Databases*, Proceedings of the Third International Conference on Information and Knowledge Management, (1994), 244-251.

S. C. Yoon, I. Y. Song and E. K. Park, *Semantic Query Processing in Object-Oriented Databases Using Deductive Approach*, Proceedings of the Fourth International Conference on Information and Knowledge Management, (1995), 150-157.

S. C. Yoon and E. K. Park, *An Approach to Intensional Query Answering at Multiple Abstraction Levels Using Data Mining Approaches*, Proceedings of the 32nd Hawaii International Conference on System Sciences, (1999).

요약

추론적 기법을 사용한 객체지향 데이터베이스의 지능적인 질의 처리

김양희*

객체지향 데이터베이스에서는 지능 정보 시스템에서 요구하는 것을 만족하기 위하여 보다 지능적인 질의 처리 기법이 필요하다. 본 논문에서는 추론적 기법을 사용하여, 객체지향 데이터베이스에서의 지능적인 질의 처리하는 방법에 대하여 논의한다. 논문에서 제시하는 방법을 사용하여, 객체지향 데이터베이스에서 주어진 질의에 대한 답을 추상적으로 표현하는 지능적인 답을 얻을 수 있다. 본 논문에서 제안하는 지능적인 질의 처리 방법은 규칙 표현, 규칙 재편성, 전 분석, 분석의 네 단계로 구성 된다. 규칙 표현 단계에서는 객체지향 데이터베이스 스키마를 사용하여 추론 규칙을 생성한다. 규칙 재편성 단계에서는 규칙에서 순환을 제거한다. 전 분석 단계에서는 유일한 내포적 문자를 얻기 위하여 규칙변환이 이루어진다. 분석 단계에서는 SLD-분석을 사용하여 내포적 답을 구한다.

* 한국체육대학교 교양과정부 조교수