TR-381

Integration of Relational Knowledge Bases and Logic Programming Languages

by
H. Monoi, Y. Morita, H. Itoh, T. Takewaki,
H. Sakai and S. Shibayama(Toshiba)

May, 1988

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Integration of Relational Knowledge Bases and Logic Programming Languages

Hidetoshi MONOI, Yukihiro MORITA, Hidenori ITOH

ICOT Research Center[*]

Toshiaki TAKEWAKI, Hiroshi SAKAI, Shigeki SHIBAYAMA

Toshiba R. & D Center[†]

May 1988

Abstract

Relational database systems are very useful and widely accepted as the systems that handle large amounts of data efficiently. However, the expressive power of the data model is limited because of the restriction of the first normal form. This limitation prevents storage of knowledge representing data with a complicated structure to a traditional database system and prevents the realization of applications using large amounts of knowledge. The relational knowledge base (RKB) model was introduced to remove such limitations on the logic programming language. The RKB enables Prolog terms to be stored and manipulated within the framework of the relational model.

This paper introduces a manipulation language for sets of terms stored to the RKB. This manipulation language is embedded in the logical programming language and enables the logic programming language to manipulate large amounts of knowledge represented by terms. Because our manipulation language can manipulate terms using the unification operation, we can manipulate terms stored in the RKB without operational gaps from the unification based logic programming language.

[*]Institute for New Generation Computer Technology, 4-28, Mita 1-Chome, Minato-ku, Tokyo, 108, Japan

[†]1, Komukai-Toshiba-cho, Saiwai-ku, Kawasaki, 210, Japan

1 Introduction

As knowledge information processing technology advances, many knowledge-based systems, such as expert systems, will be made for practical use. The amount of knowledge and the degree of complexity of knowledge representing data used in such systems will increased according to the degree of maturation. Such knowledge will be required to be shared by knowledge-based systems on the analogy of traditional database systems.

In recent years, logic programming languages have become very popular in the design and implementation of knowledge-based systems. However, these systems are established with the premise that all knowledge is included as part of the systems in spite of the fact that they must handle a large amount of knowledge or that they have a knowledge base management system for private use. In such situations, it is necessary to establish a dedicated system which can efficiently manage and store a large amount of knowledge, that is, knowledge representing data with a complex structure, shared by several knowledge information processing systems. We call this dedicated system a *knowledge base system*.

Database systems are very useful and widely accepted as systems to manipulate and share large amounts of data. However, the expressive power of the data model is limited because of the restriction of the first normal form. This limitation prevents the use of traditional database systems as the back end of logic programming languages which use structured terms as the basic data structure. Recently, much research has been conducted towards extending the traditional data model and adding more useful semantics to it [8][10][14]. Especially, for the relational data model, many extensions have been made because of its formally definable semantics.

We are researching a knowledge base system which can be accessed from knowledge information processing systems based on the logic programming language in Japan's Fifth Generation Computer Project. A logic programming language can be regarded as a programming language which manipulates terms as basic data structures. It can manipulate terms with complex structure using a unification operation. Therefore, we intend to establish knowledge base systems that can be store and manipulate large amount of terms efficiently.

We have proposed a *relational knowledge base* (RKB) model as an extended relational data model which can store a set of terms to a table (relation) and manipulate terms by a unification operation [6][13]. The collection of these relations is called the RKB. We have also proposed a set of operations on the RKB, which is a natural extension of relational algebra and can be used to perform inference on stored rules. The operations are called, collectively, retrieval by unification (RBU) operations. Introducing the unification operation as a basic operation of the RKB, the manipulation language for the RKB is naturally embedded in a logic programming language.

The major contribution of this paper is to show how to integrate RKBs and logic programming languages. This integration will provide a very convenient environment for logic programming language when a large amount of knowledge is required. This paper introduces query language in the form of predicates of the logic programming language, which can define the retrieval and updating of sets of terms.

The remainder of this paper overviews the relational knowledge base model and our manipulation language embedded in the logic programming language. Section 2 describes the relational knowledge base model. Section 3 defines queries for the relational knowledge base. Section 4 describes our manipulation language for term relations. Section 5 is a discussion and Section 6 is a summary of this paper.

2 Overview of the Relational Knowledge Base Model

This section overviews the relational knowledge base model. This model enables us to define *terms* as attribute values of the relational scheme and to manipulate these terms by unification operations. Because items stored to each relation are terms, instances of the relational scheme are called *term relations*.

A relational knowledge base is composed of a collection of term relations. An n-attribute term relation is defined as a relation whose domain of each attribute is a set of terms. Here, we define terms as:

1. A variable is a term.

2. A constant is a term.

3. If f is an n-ary function symbol and $t_1, t_2, ..., t_n$ are terms, then $f(t_1, t_2, ..., t_n)$ is a term.

This definition of terms is the same as logic programming languages such as Prolog. Expressing a set of terms, K_i, an n-attribute term relation is defined as a subset of the Cartesian product of sets of terms, $K_1, K_2, ...K_n$. Assuming that T is an n-attribute term relation, T is defined as:

$$T \subset K_1 \times K_2 \times ... \times K_n.$$

In the process of extending the relational data model to the relational knowledge base model, operations of conventional relational algebra, such as join, projection, and restriction, are extended to operations based on unification. We call these extended operations retrieval by unification (RBU) operations. The extension was made by enhancing the equality check between constants to unification operation between terms. The *unification-join* and *unification-restriction* operations are defined as RBU operations [6][13].

Let us consider a simple knowledge base which will be used to illustrate the relational knowledge base model. In the following, we express a tuple of an n-attribute term relation, r, as:

$$r(x_1, x_2, ..., x_n)$$

where each x_i stands for a term defined above. Each tuple with the same relation name and the same number of attributes is stored to the relation with its relation name. Moreover, we assume the conventions of DEC-10 Prolog, such that any word starting with either a capital letter or '_' denotes variables, and other words denote constants.

Allowing terms as attribute values, many expressions are possible. The following is a simple bicycle relation which expresses a module-submodule relation among modules constructing bicycles. Although there are multiple submodules for one module, we can express that relation in just one tuple using a list structure.

Example 2.1 *Simple module-submodule relation of bicycles*

```
assembly(bike,[frame,wheel]).
assembly(frame,[front_fork,diamond_frame]).
assembly(wheel,[tire,rim,spokes,hub]).
```

For example, the first tuple describes that the `bike` module has `wheel` and `frame` as submodules.

Using n-ary functors, we can add more attributes to each item. For example, we can distinguish colors among modules by introducing a color attribute for each item.

Example 2.2 *More complex expressions using functors*

```
assembly(bike(red),[frame(red),wheel]).
assembly(bike(yellow),[frame(yellow),wheel]).
assembly(frame(red),[front_fork(red),diamond_frame(red)]).
assembly(frame(yellow),
                [front_fork(yellow),diamond_frame(yellow)]).
assembly(wheel,[tire,rim,spokes,hub]).
```

This shows, for example, that a *red* `bike` has a *red* `frame` and a *red* `frame` is composed from a *red* `front_fork` and a *red* `diamond_frame`.

By introducing variables, we can express the property inheritance between a module and its submodule. For example, the following tuple expresses that `front_fork` and `diamond_frame` have the same color as `frame`.

Example 2.3 *Expression of property inheritance using variables*

```
assembly(bike(red),[frame(red),wheel]).
assembly(bike(yellow),[frame(yellow),wheel]).
assembly(frame(X),[front_fork(X),diamond_frame(X)]).
assembly(wheel,[tire,rim,spokes,hub]).
```

As stated above, relational algebraic operations are extended to RBU operations. The RBU operations allow unification operation to be used between terms as conditions of relational algebraic operations. $\diamond$ denotes the unification operation as a condition, and many symbols defined in [12] denote relational algebraic operations, i.e., σ, $\bowtie$, and π for restriction, join, and projection, respectively.

Using the unification-restriction operation, we can restrict tuples whose first attribute can be unified with `bike(X)` from the `assembly` relation of Example 2.3. Suppose that the results of this restriction are stored into the new two-attribute relation, `result1`(X_1, X_2), this restriction operation can be expressed as the following equation.

$$result1 = \sigma_{A_1 \diamond bike(X)} assembly(A_1, A_2)$$

Example 2.4 is the result of this operation

Example 2.4 *Result of the unification-restriction operation.*

```
result1(bike(red),[frame(red),wheel]).
result1(bike(yellow),[frame(yellow),wheel]).
```

We can make a list of parts that are necessary for making `bike(red)` and `bike(yellow)` from `assembly` of Example 2.3 and the `result1` relation. First, to extract submodules one by one from the second attribute of the `assembly` relation, we must introduce unique tuple relation, `template`, such as:

$$\texttt{template}(\texttt{X}, \texttt{Y}, [\texttt{X}|\texttt{Y}]),$$

and make the following unification-join operation between the `template` relation and `assembly` relation.

$$temp = \pi_{A_1,A_2}(template(A_1, A_2, A_3) \underset{A_3 \diamond B_2}{\bowtie} assembly(B_1, B_2))$$

We can obtain the `temp` relation below as a temporary result.

```
temp(frame(red),[wheel]).
temp(frame(yellow),[wheel]).
temp(front_fork(X),[diamond_frame(X)]).
temp(tire,[rim,spokes,hub]).
```

Making the unification-join between the `temp` relation and the `assembly` relation, we can obtain the relation of Example 2.5.

$$result2 = \pi_{B_1,B_2}(temp(A_1, A_2) \underset{A_1 \diamond B_1}{\bowtie} assembly(B_1, B_2))$$

`frame(X)` in the first attribute of the `assembly` relation is unified with `frame(red)` and `frame(yellow)` in the first attribute of the `temp` relation and the binding to variable `X` is propagated to terms in the remaining attribute.

Example 2.5 *Result of the unification-join operation*

```
result2(frame(red),[front_fork(red),diamond_frame(red)]).
result2(frame(yellow),
                [front_fork(yellow),diamond_frame(yellow)]).
```

Although other relational algebraic operations, such as aggregate functions, are not described above, the relational knowledge base model includes them with the same operational semantics defined in the relational data model.

3 Queries to the Relational Knowledge Base

This section considers how to access term relations from a logic programming language. A logic programming language can be regarded as a programming language that manipulates terms as a basic data structure. The RKB enables direct storage and manipulation of terms. These functions are effective in managing and storing a large amounts of knowledge represented by terms.

Term relations are manipulated by the relational algebraic operations in the previous section. Those relational algebraic operations are procedural operations. Because a logic programming language is rather declarative, it is necessary to make a manipulation language declarative so that it can be embedded in the logic programming language without operational gaps.

3.1 Relational Calculus for the Relational Knowledge Base

For the relational data model, we already have a declarative manipulation language, called relational calculus. Relational calculus is based on predicate calculus. Because predicate calculus is also a logic programming language base, it is desirable to establish manipulation language for the RKB based on relational calculus.

This section gives an informal definition of the manipulation language for the RKB, based on relational calculus. It can be considered as an implementation of domain relational calculus for term relations. Referencing the definition of domain relational calculus in [12], we extend it for the term relations. The extension is made by extending the domain of each calculus to the set of terms and operations defined between constants to the unification operation between terms.

Expressions in domain relational calculus for the term relations are of the form

$$\{< \mathbf{t_1}, \mathbf{t_2}, ..., \mathbf{t_k} > | \psi(\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_l})\},$$

where each t_i $(1 \leq i \leq k)$ and x_j $(1 \leq j \leq l)$ is a term and ψ is a formula built from terms and atomic formulas defined below. Each t_i can include the same variables used in each x_j so that, when variables in each x_j are instantiated in the evaluation of ψ, bindings are propagated to the corresponding variables in each t_i. Atomic formulas forming ψ are defined as follows.

$\mathbf{r}(\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_l})$: where r is the relation name of a l-attribute term relation and every x_i is a term.

$\mathbf{x}\theta\mathbf{y}$: where x and y are terms and θ is an operator defined between terms.

In the definition of domain relational calculus for the relational data model, each x_j must be a constant or a variable and $r(x_1, x_2, ..., x_l)$ merely asserts that the value of each x_j variable must be selected so that $x_1 x_2 ... x_l$ is in relation r. However, we must extend this definition so that it can manipulate sets of terms.

The first type of atomic formula asserts the following. Suppose that $< y_1, y_2, ..., y_l >$ denotes an arbitrary tuple of relation r, the value of each x_j $(1 \leq j \leq l)$ must be x_j such that $r(x'_1, x'_2, ..., x'_l)$ is the result of unification between $r(x_1, x_2, ..., x_l)$ and $r(y_1, y_2, ..., y_l)$. That is, there is a most general unifier, β, between $r(x_1, x_2, ..., x_l)$ and $r(y_1, y_2, ..., y_l)$ such that

$$r(x'_1, x'_2, ..., x'_l) = r(x_1, x_2, ..., x_l)\beta = r(y_1, y_2, ..., y_l)\beta.$$

For example, suppose `X` and `Y` are variables, we can select all tuples from the `assembly` relation of Example 2.3 using the formula of `assembly(X,Y)`. Using the `assembly(bike(X),Y)` formula, we can select the same tuples as Example 2.4.

The second type of atomic formula, $x\theta y$, asserts that x and y must be terms that make $x\theta y$ true. θ is also extended from an arithmetic relational operator to the operator defined between terms. The next section introduces various kinds of relational operators between terms.

Atomic formulas may be combined by means of logical operators such as $\vee$, $\wedge$, and $\neg$. We define formula ψ recursively as follows.

1. Every atomic formula is a formula.

2. If ψ_1 and ψ_2 are formulas, then $\psi_1 \vee \psi_2$, $\psi_1 \wedge \psi_2$, and $\neg\psi_1$ are formulas. These formulas assert respectively that "ψ_1 or ψ_2, or both are true", "ψ_1 and ψ_2 are both true", and "ψ_1 is not true".

3.2 Relational Operations between Terms

For the relational data model, arithmetic relational operators are sufficient to compare constants. However, these arithmetic operators cannot treat structured data types such as terms. Therefore, it is necessary to add other relational operators which are defined between terms for the RKB. This section introduces several relational operations between terms. These operations are based on unification, unifiability, arithmetic comparison, and generality between terms.

In the following, it is assumed that x and y stand for terms. Each operator is defined as follows:

Unification :

Unification is one of the most necessary operation primitives to manipulate terms. We have defined three kinds of operators relating to unification. They are the operations to check whether two terms are unifiable or not.

When we want to unify two terms or extract substructures from terms, we can do it using the *unification operation*, which is shown by `=`. `x = y` asserts that it is true when two terms are unifiable and `x` is unified with `y`. For example, when the following formulas are used for the `assembly` relation of Example 2.3,

$$\texttt{assembly(X,Y)} \land \texttt{X} = \texttt{bike(A)}$$

we can select the first two tuples, whose first attribute's value is `bike(A)`, of the `assembly` relation and obtain {`red`,`yellow`} as a set of bindings to variable `A`.

Sometimes, it is necessary only to test whether two terms can be unified or not without applying their substitutions. We call this a *unify-check operation* and assign the `<=>` symbol to it. For example, when the above formula is as follows,

$$\texttt{assembly(X,Y)} \land \texttt{X} <=> \texttt{bike(A)}$$

it merely selects the first two tuples, and cannot obtain any bindings for the variable `A`.

Generality :

When we want to know if several terms have the same structure or if several terms may have the same meaning in our semantic definition for terms, neither the unification nor the unify check operation can be used. To enable this, the *generality* of terms should be compared. The generality of terms is defined as follows.

> *Generality*: Between terms `t` and u, `t` is defined as more general than u if and only if there is a substitution, β, such that $\texttt{t}\beta = \texttt{u}$.

According to the above definition, `f(X,Y)` is more general than `f(3,Z)`. This is shown using the `>>` symbol:

```
f(X,Y)>>f(3,Z).
```

When one term is more general than the other and vice versa, the generality of these terms is regarded as equal. For example, `f(X,Y)` and `f(A,B)` is such a case. This is denoted `<<>>`, as follows.

```
f(X,Y)<<>>f(A,B).
```

According to the above definition, if the generality of two terms is equal, they can be made literally identical by appropriately renaming the variables of one term. Note that there are many cases when the generality order is not applicable. For example, we cannot decide the generality between `f(X,3)` and `f(4,Y)`.

Equality :

This is used when we want to test whether the two terms currently instantiating each term are *literally equal*. Especially, variables in equivalent positions in the two terms must be literally equal. Literally equal is denoted `==`. For example, although `f(X,Y)` and `f(A,B)` are equal in generality, they are not equal in the case of literally equal when each variable, `X`, `Y`, `A`, and `B`, is not instantiated. However, if `X=A` and `Y=Z`, then `f(X,Y)==f(A,B)`.

3.3 Query Expressions in the Form of Calculus

As stated in the preceding two sections, queries for the RKB are expressed as a combination of two kinds of atomic formulas. This section gives some query expressions using atomic formulas defined in the preceding sections by examples. Suppose that each t_i and x_j is a term, and a calculus expression is an expression of the form

$$\{< t_1, t_2, ..., t_k > |\psi(x_1, x_2, ..., x_l)\}.$$

The unification-join between relations `r(X,Y)` and `t(Z,W)` is expressed as

$$\{< \mathtt{X}, \mathtt{Y}, \mathtt{W} > |\mathtt{r}(\mathtt{X}, \mathtt{Y}) \wedge \mathtt{t}(\mathtt{Y}, \mathtt{W})\}.$$

Attributes used to join are designated by the same variable name. In this case, `r` and `t` are joined on the second attribute of `r` and the first attribute of `t`. The bindings to each variable of one predicate are propagated to variables with the same variable name in other predicates, the same as the execution of Prolog clauses. We can rewrite this formula using the unification operator, `=`, as follows.

$$\{< \mathtt{X}, \mathtt{Y}, \mathtt{W} > |\mathtt{r}(\mathtt{X}, \mathtt{Y}) \wedge \mathtt{t}(\mathtt{Z}, \mathtt{W}) \wedge \mathtt{Y} = \mathtt{Z}\}$$

Moreover, equijoin can be asserted by the literally-equal operator, `==`, as follows.

$$\{< \mathtt{X}, \mathtt{Y}, \mathtt{W} > |\mathtt{r}(\mathtt{X}, \mathtt{Y}) \wedge \mathtt{t}(\mathtt{Z}, \mathtt{W}) \wedge \mathtt{Y} == \mathtt{Z}\}$$

A restriction operation which obtains tuples in `result1` of Example 2.4 from `assembly` of Example 2.3 is expressed as

$$\{< \mathtt{bike}(\mathtt{X}), \mathtt{Y} > |\mathtt{assembly}(\mathtt{bike}(\mathtt{X}), \mathtt{Y})\}$$

Lastly, unification-join and projection between `result1` and `assembly` to obtain `result2` of Example 2.5 is expressed as follows.

$$\{< \mathtt{Y}, \mathtt{Z} > |\mathtt{result1}(\mathtt{X}, [\mathtt{Y}|\_]) \wedge \mathtt{assembly}(\mathtt{Y}, \mathtt{Z})\}$$

4 Manipulation Language for the Relational Knowledge Base

We have established a manipulation language for the RKB. This manipulation language is embedded in ESP[3], which is a logic programming language with object oriented features. ESP has been developed in ICOT and used to establish the SIMPOS, the operating system of the AI personal work station, PSI.

As described in the previous section, queries for the term relations can be built based on the unification operation defined among terms. Because the unification operation can be regarded as a basic operation of logic programming languages, it is possible to integrate the manipulation language for the RKB into a logic programming language without operational gaps. That is, we can establish a manipulation language with the same operational semantics as a logic programming language.

We have embedded the manipulation language for the RKB by providing special predicates which manipulate term relations. They are used to create term relations, to insert tuples into term relations, or to retrieve tuples from term relations.

The special predicates for accessing the RKB are described below.

Data Retrieval

format :

```
retrieve(Relation,Query)
retrieve(sort(Relation,Attrlist,EqOpList),Query)
retrieve(unique(Relation,EqOp),Query)
retrieve(group(Relation,AttrList,EqOpList),Query)
```

meaning :

The `retrieve` predicate is always executed successfully and creates resultant relations within the knowledge base system as the side effect.

retrieve(Relation,Query) is the simplest predicate for retrieving tuples from term relations in the RKB. In this predicate, Query denotes the condition used to retrieve tuples. For example, in specifying assembly(frame(X),Y) as this Query for Example 2.2, tuples that have the unary functor, frame, in the first attribute are retrieved. Relation denotes the specification for the resultant relation of this retrieval operation. For example, retrieve(color(X),assembly(frame(X),_)) creates a one-attribute relation, color, that consists of two tuples, color(red) and color(yellow). If the relation specified by Relation does not exist in the RKB, it is created before execution of the retrieval operation.

The remaining three predicates are retrieval operations being added aggregate functions. Each predicate corresponds to sorting, to making unique, and to grouping tuples of the resultant relation. AttrList specifies attribute variables used in Relation in the form of a list. Sorting and grouping are performed according to the values of attributes designated in this list. EqOp and EqOpList denote relational operations which are used to compare values to execute each aggregate function.

Tuple reference

format :

```
get(Tuple)
getAsList(List,Number,Tuple)
```

meaning :

As stated above, the retrieve predicate returns no binding values for the tuples in the resultant relation. Therefore, we need a predicate to reference values of each tuple in the term relation. We have two kind of predicates to refer values of each tuple. One refers tuples one by one, and the other refers all tuples of the designated term relation at once.

get(Tuple) predicate is the first kind of predicate. Tuple specifies the relation name and its attribute. For example, we can specify assembly(frame(X),Y) as Tuple for Example 2.2. Tuple is unified with one of the tuples in the designated relation and bindings that are the result of this unification will be returned. We can access all tuples in the term relation designated by Tuple using backtracking. That is, in redoing get, an alternate tuple is unified to Tuple.

The getAsList(List,Number,Tuple) predicate returns the number of tuples specified by Number of the relation specified by Tuple to List in the form of a Prolog list.

For example, we can use get(assembly(X,Y)) for tuple-wise reference or getAsList(List,5,assembly(X,Y)) for reference of all tuples for the assembly relation of Example 2.2.

Tuple insertion

format :

```
put(Tuple)
putList(List,Tuple)
```

meaning :

In the same way as tuple reference, we have two kinds of predicate for tuple insertion into the term relations. They are put(Tuple) for tuple-wise insertion and putList(List,Tuple) for insertion of a set of tuples. Each argument plays the same role as the get and getAsList predicates. However, Tuple and List must be bound before these predicates are executed.

Suppose that we write put(assembly(bike(X),[wheel,frame(X)])) and that variable X is bound to blue before execution of the put predicate. The assembly(bike(X),[wheel,frame(X)]) tuple will then be inserted in the assembly relation of Example 2.2.

We realize *not* by the difference operation between term relations, that is we express the negation of a relation in a relative complement expression. Therefore, we provide a meta-level predicate, such as `dif(A,B,`θ`)` where `A` and `B` are relations with the same attribute number and θ is a relational operator, as a `Query` of the `retrieve` predicate. `dif(A,B,`θ`)` asserts that if `A` and `B` have the same number of attributes, the set of tuples `a` in `A` such that, for arbitrary tuple `b` in `B`, `a`θ`b` is not true. θ must be `==`, `<<>>`, or `<=>`.

5 Discussion

This paper introduced a manipulation language for the RKB. The RKB is an extension of a relational database, which can store and manipulate sets of terms directly. The manipulation language introduced in this paper can provide functions to access large amounts of knowledge stored in the RKB from a logic programming language such as Prolog.

There are many approaches to integrate database systems and logic programming languages, such as deductive database systems[4] and complex object data models[1]. Deductive database systems were discussed on the premise that large numbers of facts are stored in the relational database systems. Complex objects are data models to obtain fuller expressive power, which is sufficient to represent knowledge with a complex structure. Since our approach follows the approach in traditional database systems, we can establish knowledge base systems using many kinds of technologies from the traditional database systems.

As described in [11], the impedance mismatch between the relational query and a logic programming language must be taken into consideration when the method of integrating them is considered. This mismatch originates in the fact that, although the relational queries are based on set-at-a-time semantics, a logic programming language is based on tuple-at-a-time semantics. LDL[11] intends to resolve such problems by extending a logic programming language to be based on set-at-a-time semantics. We provide two special predicates `get` and `getAsList`. These predicates enable a logic programming language to access not only one tuple at a time but also one set at a time. Compared to LDL, our attempt does not resolve the mismatch completely. However, our manipulation language enables a traditional logic programming language to access the RKB without any semantic change.

Our manipulation language cannot express recursive queries for the present. Recursive queries are one of the most important features for realizing deduction on a knowledge base. Because there are many research reports about efficient recursive query processing in deductive database systems[2], it is possible to take their methods in our manipulation language as a future topic of research.

6 Conclusion

Our manipulation language for the RKB can manipulate sets of terms by unification operation. To manipulate terms declaratively, we propose formulas defined on the set of terms and relational operators between terms referencing the definition of domain relational calculus.

The RKB is implemented on an experimental hardware system named *Mu-X*. *Mu-X* is a multiprocessor system with a multiport page-memory[5][7]. We connected a PSI to this system as the host system. The manipulation language introduced in this paper is embedded in the PSI and is used as an interface language between the PSI and *Mu-X*[9].

Acknowledgment

We wish to extend our thanks to members of the VLKBM meeting for many useful discussions.

References

[1] Bancilhon, F., et al., "A Calculua for Complex Objects", in *Proc. ACM Int. Symp. Principles of Database Systems*, March 1986, pp. 53-59

[2] Bancilhon, F., et al., "An Amateuer's Introduction to Recursive Query Processing Strategies", in *Proc. ACM SIGMOD '86*, pp. 16-52 (1986)

[3] Chikayama, T., "Unique Feature of ESP", in *Proc. Int. Conf. Fifth Generation Computer Systems*, 1984

[4] Gallaire, H., et al., "Logic and Data Bases : A Deductive Approach", *ACM Comput. Surv.*, Vol. 16, No. 2, pp. 153-185 (1984)

[5] Monoi, H., et al., "Parallel Control Technique and Performance of an MPPM Knowledge Base Machine Architecture", in *Proc. 4th Int. Conf. Data Engineering*, February 1988, pp. 210-217

[6] Morita, Y., et al., "Retrieval-By-Unification Operation on a Relational Knowledge Base", in *Proc. 12th Int. Conf. Very Large Database*, August 1986, pp.52-59

[7] Sakai, H., et al., "A simulation Studay of a Knowledge Base Machine Architecture", in *Database Machines and Knowledge base Machines*, Kluwer Academic Publishers, pp. 585-598 (1988)

[8] Scholl, M. H. and Scheck, H. J. (ed.), *Proc. International Workshop on Theory and Applications of Nested Relations and Complex Objects*, April 1987

[9] Shibayama, S., et al., "*Mu-X*: An Experimental Knowledge Base Machine with Unification-Based Retrieval Capability", in *Proc. France-Japan Artificial Intelligence and Computer Science Symposium 87*, November 1987, pp. 343-357

[10] Stonebraker, M., "Object Management in POSTGRES Using Procedures", *Proc. 1986 International Workshop on Object-Oriented Database Systems*, Sept. 1986

[11] Tsur, S., et al., "LDL: A Logic-Based Data-Language", in *Proc. 12th Int. Conf. Very Large Data Bases*, August 1986, pp.33-41

[12] Ullman, D.J., "Principles of Database Systems", *Computer Science Press*, Maryland, USA, 1982

[13] Yokota, H., et al., "A model and an Architecture for a Relational Knowledge Base", in *Proc. 13th Annual Int. Sump. Computer Architecture*, June 1986, pp.2-9

[14] Zaniolo, C., "The Database Language GEM", *Proc. ACM-SIGMOD Conference on Management of Data*, San Jose, Ca., May 1983