Journal of the Japan Society for Management Information
Vol.11 No.1, Jun. 2002, pp.69-89

論　文

Prologをベース言語とするエンドユーザ開発のためのオブジェクト指向データベースの設計と実現

高原　康彦、柴　直樹、高木　徹（千葉工業大学）

要旨：筆者らは，エンドユーザ開発(End User Development、以下 EUD とする)のためのベース言語として，標準的な Prolog を拡張した extProlog を開発している．今回，EUD 実現のための情報技術支援の一環として，extPrologとリレーショナルデータベースを接続することにより，PrologをEUDのためのベース言語とするオブジェクト指向データベースを実現した．本論では，筆者らが意図しているPrologによるオブジェクト指向的なデータベース操作を言語仕様の形で定義し，それの実施方法を正確に記述するために定理証明として表現している．実現されたシステムは期待された機能を果たしている．

キーワード：意思決定支援システム，オブジェクト指向，Prolog，データベース，EUD

A Design and Implementation of OODB for EUD by Prolog

Yasuhiko TAKAHARA ， Naoki SHIBA， Toru TAKAGI (Chiba Institute of Technology)

Abstract： We have engaged in a research of EUD (End User Development) where a standard Prolog is extended as extProlog so that it can be used as a base language of our EUD. In order to make extProlog a satisfactory technological support of our EUD, an object oriented data base connectivity is realized for it using a conventional relational data base system. In this paper the data base manipulation in extProlog is formalized as a specification of the manipulation language and its implementation is presented in a theorem - proof form to be precise. The realized system performs the expected functions.

Keywords：DSS, Object Oriented, Prolog, Data Base, EUD

2000年5月30日 受付， 2002年2月26日 受理

1 はじめに－第4世代システム開発としてのEUD

筆者らは，EU (End User) が，比較的小規模な問題を対象として，それを自分で定式化し，然るべき情報技術支援の下にその定式化をPC或はWS上に実施し，自分で問題解決を行うことをEUD (End User Development) の1つの形と考え研究を行っている．参考文献[1]は，このようなシステム開発を第4世代システム開発と呼んでいる．

正確に言うと，参考文献[1]での第4世代システム開発は，EUが，情報技術の専門家の力を借りずに，その代わりに第4世代ソフトウェアツール (第4世代言語やグラフ処理言語) の力を利用して自分で問題解決のための情報システムを開発することとしている．そして第4世代言語というのは，通常の言語より非手続き的であり，また自然言語的コマンドをより多く含んでいるもので，情報技術の非専門家にも使いやすくなっているとしている．参考文献[1]で考えている第4世代言語というのは，例えばAPLである．

第4世代システム開発研究の必要性は，

1. 経営情報システムにおいて，例えばKDD (Knowledge Discovery in Database) への関心の高まりが示すように，単なるデータ処理ではなく，問題解決の重要性が増加してきたこと．また，問題を本当に知るものはEUであること．

2. 現在システム開発の大部分 (参考文献[1]では90%と述べている) はWSまたはPC上で行われており，従来のライフサイクルアプローチとは異なる開発方法論が必要であること．

3. ビジネス環境の変化速度の増大からRAD (Rapid Development) が重要となっていること．

による[1].

この論文でいうEUDは第4世代システム開発である．参考文献[1]と異なる所は，筆者らの第4世代システム開発では，その概念がより操作化され，第4世代ソフトウェア支援ばかりでなく，問題の定式化方法も考察の対象となっている点である．すなわち，問題の定式化は，数理的一般システム理論を基礎にして参考文献[13]で提案した問題解決のスケルトンアプローチを考え，第4世代ソフトウェア支援も，特にEUDのために拡張されたProlog(extProlog)を採用している[6]．この意味では，この論文で考えているEUは，差し当たりはスケルトンアプローチを使うという意味でシステムズアプローチを理解し，extPrologで自分の問題が記述できるユーザである．Prologが第4世代言語の代表の1つであるのは言うまでもない．

一般にPrologは第4世代システム開発の観点から次の大きな利点を持っている．

1. Prologを使えば，概念が直接表現でき (すなわち数理的一般システム理論の定式化がそのままプログラムに変換され)，概念

のみでプログラムが構成されるという意味からプログラムが読み易く，その長さが，例えばCと比較したとき，大幅に減少する．これはRADに適していると同時にEUDにとって重要である．

2. 非手続き的にプログラムが作れること，再帰的プログラムが容易に作れること，バックトラック機能の存在はEUDに大きな利益となる．

3. 全ての変数がタイプレス且つローカルであるから，プログラムの変更が容易である．タイプレスはプログラムの短縮化にも寄与している．

4. シンボル的処理を自由に行うことができる．

5. リスト処理を容易に行うことができる，すなわち複雑な構造，ダイナミックスのための時間関数が容易に表現できる．

6. 文法が「事実」と「規則」のみで簡単であることから，EUに向いている．

7. Prologの処理がインタープリタ方式であるので，RADに非常に好都合である．

しかし，通常のPrologは問題解決型経営情報システムの開発を考えるとき，次の問題を持っている．

1. 数値処理(特にベクトルや行列の演算)が弱い．

2. GUIを持っていない．

3. モデル構築ツールを持たず，またmodel integrationアプローチができない[7]．

4. オブジェクト指向データベースとの結合がない．

今まで開発を行ったextPrologというのは，1., 2., 3.を問題として通常のPrologを拡張したものである．4.の問題がこの論文のテーマである．また，問題解決のためのモデルは，DBとの結合がなければ用をなさないことは言うまでもない．

現在extPrologの利点から，我々のグループのシステム開発，授業に使うための単純なAI的ゲームから，MRPの実施[12]，データマイニングの実施，CMOT(computational and mathematical organization theory)の実施[10]などほとんど全てのシステム構築にシステムを利用している．

勿論，Prologの最大の難点は，処理速度である．しかし，メモリの価格の低下とCPUの処理速度の向上により，EUDでの問題解決にPrologを使用することは問題のないレベルになっている．例えば，我々のextPrologで線形計画法の処理プログラムを書くとき，変数20，制約式20の問題をPentium-II 300MHzのコンピュータで1秒以内で解くことができる．変数20，制約式20のLPの問題は，EUDの問題としては決して小さくない問題である．

オブジェクト指向データベースの定義は，参考文献 [9] によると，次の 2 つの点がキーとなっている．

1. オブジェクトを取り扱うことができる．
2. 既存の汎用言語から透過的にデータベースへアクセスできる．

問題解決のための EUD でのデータの取り扱いもこの 2 点を満足する必要がある．

問題解決のための EUD で取り扱うデータは，大量の浅い構造のデータというよりは，少量ではあるが深い構造のデータであることが多い．深い構造のデータというのは，データの属性が構造を持ち，データの処理は属性の属性あるいはそのまた属性に対する処理を行なうようなものである．例えばシステムの構造が樹構造であったり，各データ間の関連が密接であるシステムである．我々が取り扱ったものでは MRP システム [12] や人工社会のシミュレーション [16] が典型的な例である．深い構造のデータはオブジェクト指向データベースが目指すもので，我々の EUD のためには，オブジェクト指向のデータベースを構成する必要性が生じている．なお RDB では，データ間の関連を直接記述することはできない．したがって間接的にデータ間の関連が密接であるシステムを記述することになるため，そのデータ操作言語 (例えば SQL) の記述は複雑になる．一方，OODB では，データ間の関連を直接表現できるため，わかりやすい表現で記述できる．

また，問題解決のための EUD が取り扱うデータは，relational calculus が行なうような，ある述語を満足するデータのクラス (部分集合の同定) ではなく，クラス内の個別の要素，即ちオブジェクトが対象となるのが普通である．それを自在に取り扱えるようにデータベースを構成することは，結局オブジェクト指向のデータベースを構成することになる．

オブジェクト指向データベースに対する上記の 2 番目の要請は，データの複雑な処理を自由に行なうためのものである．我々が目指す EUD では，開発されたシステムを長く使うことはなく，システムは状況に応じて絶えず改変される．或は開発の途中でも絶えず改変される．これは，TPS (transaction processing system) とは大いに異なり，簡単に処理方法が変更できることが重要である．この柔軟性のために問題処理を記述するベース言語と，データベースを取り扱う言語が完全にシームレスに接続されていることが望ましい．

この論文の目的は第 4 世代システム開発を実現するために，extProlog の DB 接続を実現することである．そして，その実現をオブジェクト指向的に，且つシームレスに行うことである．

Prolog に代表される論理型プログラミング言語を拡張する試みは，80 年代より数多くなされてきている．特に，わが国における第五世代コンピュータプロジェクトにおいては，Prolog

をベースにした言語拡張の研究が成果を上げている．その目標は，ほぼ次の２点に集約できる．

1. 知識情報処理のための新しい計算機アーキテクチャーにおけるシステム記述言語としての拡張．GHC[8]，KL1[17] など．

2. 演繹オブジェクト指向データベースに代表される知識表現言語のための拡張．Quixote[14] など．

1では，高速並列計算機のハードウェアの開発と並行して，Prolog をベースとした並列論理型言語への拡張がその目的とされた．GHC や KL1 は，ホーン節の集合をプログラムとする一階述語論理をベースとした論理型言語であると言う意味で，従来の Prolog と同じであるが，並行処理される複数のゴール間の同期・通信を実現するためのメカニズム等を組み込むことにより，Prolog を並列処理言語へ拡張している．

2は，Prolog の項を用いて表現される知識／データを，オブジェクト指向に拡張するため，オブジェクト識別性，クラス階層，メソッド実行などのオブジェクト指向概念を項表現に盛り込むことによって，Prolog を拡張するという手法を取っている．

本論文の目指すものは，並列処理を目的とした Prolog の拡張ではない．したがって，上記１の目標のもとで行われてきた Prolog の拡張とは全く異なる．また，問題解決に用いられるデータ/知識をオブジェクト指向的に取り扱うことを目的としているという点では，上記２の目標と共通しているが，意図と実現方法は以下の意味で全く異なっている．

上記２の目標のもとで行なわれた Prolog の拡張は，(Prolog の中で) 知識やデータをオブジェクト指向的に表現することを目的としている．すなわち，拡張された Prolog をデータベースそのものを記述する言語として位置付けている．それに対し，本論の目的は「既存のリレーショナルデータベースを，extProlog という１つの言語上で，オブジェクト指向的に取り扱うための仕組みを実現すること」である．そのために，実装上は extProlog を用いて表現されたデータ構造やデータベース操作と，標準的なリレーショナルデータベースの操作言語とされる SQL との間で，内部的に翻訳を行なうという手法を取っている．データは persistent データとしてデータベースの中に存在しなければならない．したがって本論の理論的骨子は，オブジェクト指向 Prolog の仕様や実現の話ではなく，この翻訳の仕組みの定式化とその十分性の証明である．ただし，この研究の副産物として，オブジェクト指向 Prolog が実現されている．

なお本論での拡張された extProlog の位置付けは，データベースの操作のみならず，問題の表現，求解アルゴリズムの表現など，エンドユーザ開発全般を支援するための「ベース言語」である．extProlog においては，本論で問題とするデータベース操作以外にも，問

題解決を支援するEUDのための言語という観点から，既存のPrologに対し拡張を行なっている[6].

この論文は次のような章立てとなっている．第1章で我々の考えているEUDを議論し，ベース言語として何故Prologを使うかを明らかにする．第2章で，実現したデータベースの操作言語仕様を述べ，第3章でその例示を行い，第4章で実施のメカニズムを定理の形で議論する．

2 操作言語の定式化

この章では，第4章の議論の前提として，開発したオブジェクト指向データベースの操作言語仕様の基本部分を定式化して提示する．実際に使われているものはここで提示する以上の機能を持つものであるが，理論としてはそれらの機能は本質的ではない．定式化の例示と意味解説は第3章で行う．この研究の目標はextPrologという言語のためのオブジェクト指向データベースの構築であるから，仕様は通常のオブジェクト指向データベースの定義とは異なる[9, 4, 18]．またベース言語とデータベース操作言語のシームレスな結合を目標としているので，extProlog自体もオブジェクト指向に拡張されるが，Prologの文法構造は原則的には変えないという方針の下にオブジェクト指向拡張を行っていることから，通常実現されているオブジェクト指向Prologの定義とも異なる[2, 3].

以下の定義で，< *constant symbol* >,< *integer* >,< *Prolog predicate list* >,< *Prolog variable* >,< *Prolog variable list* >,< *Prolog value* >(Prologの変数がとる値)等は，未定義語として使用する．これらの定義は通常使われているものを想定している．

2.1 クラスの定義

クラスは次のようなPrologルールによって定義する．

< *class definition* >::= *class*(< *class name* >,< *attribute list* >);

< *class name* >::=< *constant symbol* >

< *attribute list* >::= [< *attribute list0* >]

< *attribute list0* >::=< *attribute list1* > | < *attribute list1* >,*inherits*(< *class name list* >)

< *attribute list1* >::=< *atomic attribute declaration* > | < *attribute list1* >,< *atomic attribute declaration* >

< *class name list* >::=< *class name* > | < *class name list* >,< *class name* >

< *atomic attribute declaration* >::= " < *attribute name* > < *attribute type* > "

< *attribute name* >::=< *constant symbol* >

< *attribute type* >::= *int*|*float*| < *string* > | < *pointer* > | < *vecpointer* >

< *string* >::= *char*(< *integer* >)

< *pointer* >::=< *class name* >

< *vecpointer* >::=< *pointer* > (< *integer* > , < *integer* >)

2.2 オブジェクトの定義

オブジェクトは次のようなProlog述語によって定義する.

< *object definition* >::=< *object name* >:= *newdbob*(< *db name* >, < *class name* >)

< *object name* >::=< *constant symbol* > |*this*

< *db name* >::=< *constant symbol* >

2.3 メソッドの定義

メソッドは次のような2つのPrologのルールによって定義する.

< *method definition* >::=< *method declaration* >< *method implementation* >

< *method declaration* >::= *methodclass*(< *method name* >,< *method input* >,< *method output* >) : −!;

< *method name* >::=< *constant symbol* >

< *method input* >::=< *class name* >

< *method output* >::=< *class name* > |*undef*

< *method implementation* >::=< *method name* > (< *method argument list* >) : − < *Prolog predicate list* >, *return*(< *return object* >);

< *method argument list* >::=< *Prolog variable list* >

< *return object* >::=< *object name* > | < *Prolog variable* > |*void*

上述の *method declaration* でカットを用いている理由は，述語 methodclass の無駄な unification を回避することにある.

2.4 データの定義

データの定義は次の Prolog 述語によって定義する.

< *data assignment* >::=< *variable form* >:=< *value form* >

< *variable form* >::=< *Prolog variable* > | < *attribute form* >

< *attribule form* >::=< *attribute head* > − >< *attribute name* > | < *attribute head* > − > []| < *attribute head* > − > [< *attribute name list* >]| < *attribute head* > − >< *integer* >

< *attribute head* >::=< *object name* > | < *attribute head* > − >< *attribute name* >

< *attribute name list* >::=< *attribute name* > | < *attribute name list* > , < *attribute name* >

< *value form* >::=< *data form* > | < *action form* >

< *data form* >::=< *Prolog value* > | < *attribute form* >

< *action form* >::=< *object name* > * < *method form* > | < *action form* > * < *method form* >

< *method form* >::=< *method name* > (< *method argument list* >)

なおクラスの定義にある通り，属性のタイプとしてクラスのベクターを定義することができることから，その属性に対してデータを挿入するときに，直接ベクターのインデックスを指定することで任意のインデックスの場所にデータを挿入することを許すために，"< *attribute head* > − >< *integer* >" の記述を可能にしている．これについては 3 章のプログラムの object data assignment 中に例示がある．

プログラムは program head と program body の 2 部からなり，クラスとメソッドは program head で定義され，オブジェクトとデータ (処理) は program body で定義される．実際のプログラムの実行は program body で行われる．

3 操作言語の定式化の適用

2 章の仕様を，参考文献 [9] に記載してある人事データベースに適用し，仕様の例示を行なう．

Figure1 が人事データベースのイメージである．

Figure1 の構造，及び参考文献 [9] に示されている method の一部を，2 章の仕様を使って記述すると次のようになる．

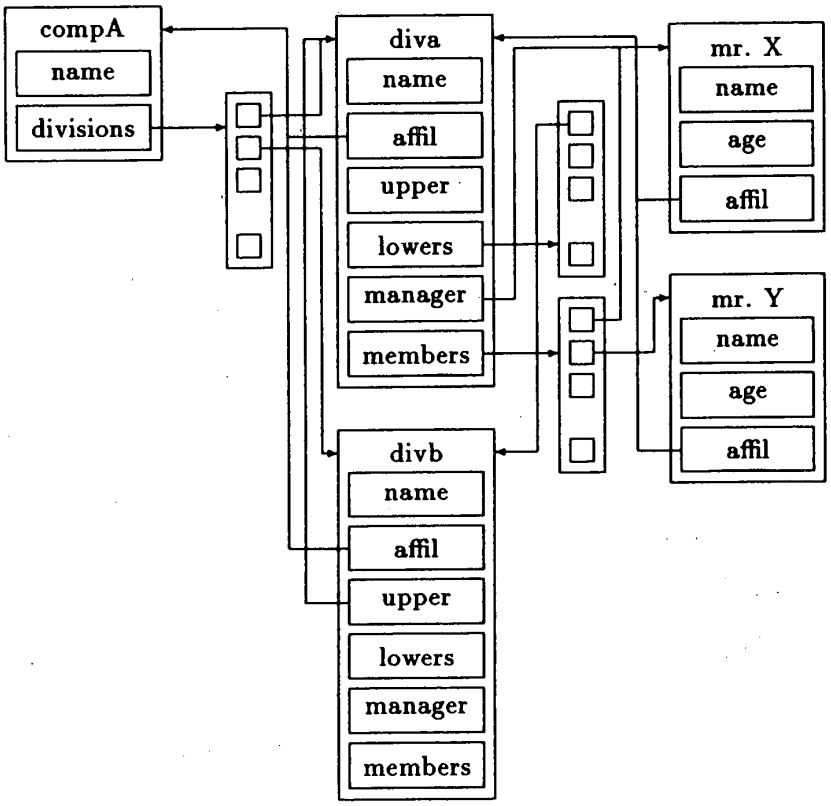

Figure 1 人事データベースの例

```
/*** Personnel Data Base ***/
/** program head **/
/* class definition */
class(company,[
      "name char(40)",
      "divs divisions"]);
class(division,[
      "name char(40)",
      "affiliation company",
      "upper division",
      "lowers divisions"]);
class(divisions,[
      "vecdiv division(0,10)"]);

/* method definition */
methodclass(cre_division,company,
    company):-!;
cre_division(D,N):-
        D:=newdbob(mydb,division),
        N:=newdbob(mydb,division),
        insert(this->divs,N),
        N->upper:=D,
        return(this);

/** program body **/
personnelmain():-

/* object definition */
        compA:=newdbob(mydb,company),
        diva:=newdbob(mydb,division),
        divb:=newdbob(mydb,division),
        divsA:=newdbob(mydb,divisions),

/* object data assignment */
        compA->[]:=["CompA",divsA],
        diva->affiliation:=compA,
        divb->[name,affiliation,upper]
            :=["Divb",compA,diva],
        divsA->0:=diva,

/* usage of method */
        X:=compA*cre_division(divb,
            divc),
        xwriteln(0,
            "current codivision=",
                diva->affiliation->
                    divs);

?-personnelmain();
```

上記のプログラムを使い，仕様の説明を行なう．

3.1 クラスの定義

プログラムでは 3 つのクラス company, division, divisions の定義が例示されている．company は 2 つの属性 name, divs を持ち，それらの属性のタイプがそれぞれ char(40), divisions となっている．char(40) は長さ 40 の string, また属性 divs のタイプ divisions は次に定義されているクラスである．即ち divs の値はクラス divisions のオブジェクトとなる．

仕様でいう < *class name* > は company,
< *attribute list* > は ["name char(40)", "divs divisions"],
< *atomic attribute declaration* > は "name char(40)", "divs divisions" 等にそれぞれ対応している．

クラス divisions の属性 vecdiv のタイプはクラス division のベクトルとなっている．即ち，vecdiv の値はクラス division のオブジェクトを 11 組並べたものになる．システムの内部において，["vecdiv division(0,10)"] は，["vecdiv0 division"], ···, ["vecdiv10 division"] のように展開されて表現されている．

3.2 オブジェクトの定義

プログラムでは 4 つのオブジェクト compA, diva, divb, divsA の定義が例示されている．compA は述語 newdbob により，データベース mydb の中にクラス company のオブジェクトとして定義されている．同様に，diva は mydb の中に division のオブジェクトとして，divb は mydb の中に division のオブジェクトとして，divsA は mydb の中に divisions のオブジェクトとしてそれぞれ定義されている．divsA は

divisionのオブジェクトを要素とするベクトルである.

仕様でいう < *object name* > はcompA, < *db name* > はmydb, < *class name* > はcompany等にそれぞれ対応している.

3.3 メソッドの定義

プログラムでは1つのメソッドcre_divisionの定義が例示されている. まず述語method-classにより, メソッドの名前をcre_division, メソッドを適用するクラスとしてcompany, メソッドの出力のクラスとしてcompanyを定義している.

cre_divisionの引数Nはthis(適用されるオブジェクト)のdivsの中に付加されるオブジェクト, Dは Nの上位となるオブジェクトの変数である. 2つのnewdbobによりD, Nはそれぞれ安全のためにmydbの中にdivisionのオブジェクトとして定義されている. DとNが既に定義されているときは, このnewdbobは無視される.

次にinsertにより, 新たなオブジェクトNがthisの属性divsに加えられる. this − >divsはthisの属性divsを指定する. insertはベクトルを取り扱う特別の述語である. "− >"は属性を指定する作用素である. 次の述語においてオブジェクトNの上位のdivisionをDと指定するためにも使用されている. すなわち, オブジェクトNのクラスは, divisionであることが予想されている. それの属性upperの値をオブジェクトDにすることによって, Dを Nの上位のdivisionとすることができる. なお, この予想が成立しないときはエラーが生じる.

returnはメソッドの処理結果を返すものである.

仕様でいう < *method name* > はcre_division, < *method input* > はcompany, < *method output* > はcompany, < *method argument list* > は(D,N)にそれぞれ対応している.

3.4 データのアサイメント

プログラムでは, 定義されている4つのオブジェクトにそれぞれデータを入れている. compAの全属性の組[name, divisions]を[]で記述し, それに対応するデータを["CompA", divsA]としている. これにより, nameにはCompAが, divisionsにはdivsAが挿入されている.

divaの属性affiliationを特に指定してデータを挿入するため, オブジェクトから直接矢印で属性を指定している. その値としてcompAが挿入される.

divbに対しては, 属性の中のname, affiliation, upperにデータを挿入するため, オブジェクトから直接矢印でデータを挿入する属性を指定している. ここではname, affiliation, upperの順に指定していることから, データもリストの形で順に"Divb", compA, divaと記述することにより各属性に値を挿入している.

divsA は属性のタイプが division のベクトルとなっており，ここでは 0 番目から 10 番目までの 11 組の division を入れることができる．例ではベクトルの 0 番目にオブジェクト diva を入れるために，/ オブジェクトから直接矢印で 0 番目を指定している．したがって挿入するべき値 (オブジェクト diva) をそのまま記述することにより，データを直接挿入している．

仕　様でいう < *attribute form* > は compA− >[] や divsA− >0，< *attribute form* > は diva− >affiliation や divb− >[name, affiliation, upper]，< *data form* > は ["CompA", divsA] や compA などにそれぞれ対応している．

最後に，プログラムはメソッド cre_division の使用例を示している．cre_division はオブジェクト compA に適用され (それの引数は divb, divc)，結果は変数 *X* に代入される．なお，divc は未定義であるので，メソッド cre_division の中で生成される．次の write 文の中で深い構造の操作が例示されている．

4 基本仕様の実施

この章では，2 章の基本仕様がどのようにリレーショナルデータベース上で実現されるかについて論じる．この実現方法は，具体的には Figure2 が示すように，リレーショナルデータベースとして PostgreSQL を使用しているが，一般的な SQL でデータを操作できる全てのリレーショナルデータベースでも適用することができる [5, 15]．

実現方法は extProlog とリレーショナルデータベースの間に変換のメカニズム (インタープリタ) を導入し，extProlog からはリレーショナルデータベースがあたかもオブジェクト指向データベースであるかのように取り扱えるようにする．変換のメカニズムとリレーショナルデータベースの間は SQL で接続されている．

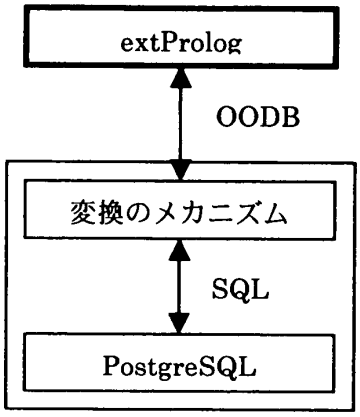

Figure 2 extProlog と PostgreSQL の関係

この変換のメカニズムを実現するために，3 つの構造体とそれらを要素とする 3 つの配列を定義する．

最初に構造体を導入する．

1. クラスの属性を定義するための構造体: A

   この構造体 A の属性は以下の通りとする (括弧内はそれぞれの属性のタイプを示す)．

   - name (string)
   - type (string)
   - start (int)

- end (int)
- link (int)

string は char(n)(n: 自然数) の一般名とする．"type" は < *attribute type* > を，"start" と "end" は属性がベクトル (vec-pointer) のときのインデックスの始まりと終りを表す．"link" は，次の属性を定義する構造体へのポインタである．Figure3 のクラス company が示すように，与えられたクラスを定義する属性群は，この "link" によりリスト構造としてまとめられている．今回の実現では，ポインタの値は構造体 A を要素とする配列 (後述の *Attr*) の中のインデックスの値である．

2. クラスを定義するための構造体: C

この構造体 C の属性は以下の通りとする．

- name (string)
- attribute (int)
- parent (int)
- brother (int)
- child (int)
- link (int)

属性 "attribute" は，このクラスを定義する属性 (構造体 A により定義されるリストの最初のもの) へのポインタである．上述の通り，クラスを定義する属性の集合はリスト構造となっている．"parent"，"brother"，"child" は他のクラスへのポインタで，これにより継承関係を実現する．"link" も他のクラスへのポインタで，これにより構造体 C の集合もリスト構造を持つことになる．後述のように同じデータベースに入っているクラスが 1 つのリストを構成する．

3. オブジェクトを定義するための構造体: O

この構造体 O の属性は以下の通りとする．

- name (string)
- dbname (int)
- classname (int)

"dbname" と "classname" は，このオブジェクトが存在するデータベースとそれが属するクラスへのポインタを表す．

以上の 3 つの構造体を要素とする配列をそれぞれ *Attr*, *Class*, *Obj* と呼ぶことにする．例えば，*Attr*[0] は構造体 A の 1 つ (0 番目) を表わす．また，各構造体 (A, C, O) へのポインタの値は各配列内のインデックスの値である．例えば *Attr*[0] は構造体 A の 1 つの表現 (インスタンス) となるが，それへのポインタは 0 である．

上記の構造体を使うと，3 章で例示したクラス company は Figure3 のように表現できる．

即ち，クラス company は *Class* の i 番目の要素，*Class*[i] として定義されている．

クラス company の最初の属性 "name" は *Attr* の j 番目の構造体 *Attr*[j] で表現されて

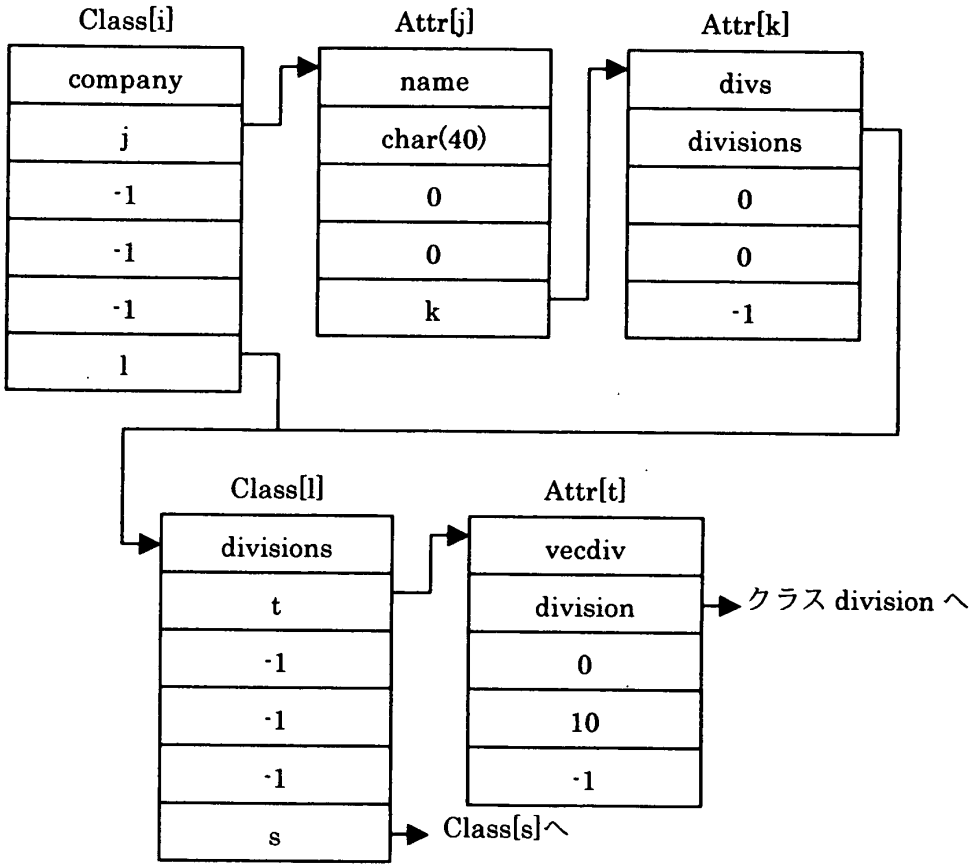

Figure 3 各構造体のポインタの受け渡しの関係

いる．ゆえに *Class*[*i*] の属性 "attribute" の値は *j*，*Attr*[*j*] の "name" には値 name が入っている．"start" と "end" は vecpointer(ポインタのベクトル) を表現するためのもので，現在の属性はスカラーであるから，両者とも 0 となっている．"link" には，(クラス company の) 次の属性 "divs divisions" を定義している *Attr*[*k*] へのポインタが入っている．

属性 "divs divisions" の属性のタイプはクラス divisions へのポインタである．これが company の最後の属性であるから *Attr*[*k*] の "link" には-1(nil) が入っている．

現在，クラス company に対して inheritance hierarchy は定義されていないとしているので，"parent"，"brother"，"child" のポインタは-1 である．*Class*[*i*] の "link" は同一データベースに存在するクラスを 1 つのリストとして表現するためのものである．Figure3 では *Class*[*l*] が次のクラスとしてリンクされている．

vecpointer を持つ例としてクラス divisions を考える．クラス divisions が *Class*[*l*] として定義されているとすると Figure3 のようになる．この場合，*Attr*[*t*] の "name" には属性名 vecdiv が入り，属性は division のベクトルであるので，"type" は division(ポインタ)，"start" は 0，"end" は 10 となる．

Figure4 が示すように，リレーショナルデータベースの中では，クラスはリレーショナル

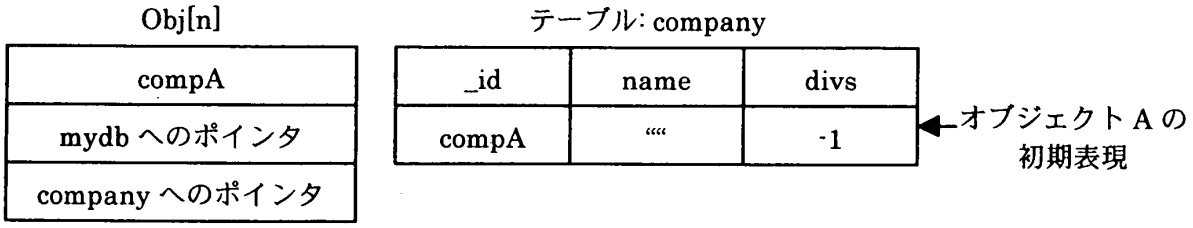

Figure 4 クラスの実現

データベースのテーブル，オブジェクトはテーブルの行として実現される．このとき pointer は整数 (int) として表現する．

compA := *newdbob*(*mydb*, *company*) の述語によって，クラス company の 1 つのオブジェクト *compA* が定義されると，そのことは配列 *Obj* とリレーショナルデータベースのテーブル *company* によって Figure4 のような情報 (データ) として表現される．Figure4 では compA は Obj 中の n 番目の要素となっている．

考察しているデータベースの世界では，オブジェクトの名前は重複していないと仮定しているので，配列 *Obj* 中のインデックスと名前が共にオブジェクト *compA* のキーとなっている．*compA* は mydb というデータベースの中でクラス company のオブジェクトとして定義されているので，"dbname" と "classname" の値は mydb へのポインタと company へのポインタとなる．company へのポインタの値は，Figure3 によると i となるはずである．Figure4 では，オブジェクト *compA* に対して属性 name と divisions にそれぞれ初期値として " "(空) と-1 が割り当てられている．compA の属性にデータが与えられると，それらがテーブル company に代入される．

リレーショナルデータベース中のテーブルには，クラスを定義したときの属性以外に "_id" という属性が自動的に付加され，その属性をキーとしてテーブル中のオブジェクトを識別する．実際には，"_id" にはオブジェクトの名前が入っている．

以上の 3 つの配列を使うことにより次の定理が証明できる (以下の証明で未定義の値は undef という定数で表示する)．

定理 1 第 2 章の仕様は，配列 $Attr, Class, Obj$ と SQL で操作できるリレーショナルデータベースにより実現できる．

証明

最初に以下の集合を定義する．

Obj_n = オブジェクトの名前の集合

C_{name} = クラスの名前の集合

$Attr_n$ = 属性の名前の集合

$Attr_{type}$ = 属性タイプの集合

= $\{int, float, string\} \cup C_{name}$

$DataV$ = リレーショナルデータベース

の各属性がとる値の集合

$= Int \cup Float \cup String$

ただし，$Int, Float, String$ は各々整数の集合，実数の集合，文字列の集合である．

$Attr = Attr_n \times Attr_{type}$

ObV = 2章の仕様の属性がとる値の集合

$= Int \cup Float \cup String \cup Obj_n$

$Int_m = \{0, \cdots, m-1\}$

= オブジェクトのインデックス集合 (オブジェクトが m 個存在すると仮定)

以上の集合に対して，3つの配列から次の関数群が定義できる．

定義 1 $obj : Int_m \rightarrow Obj_n$ ただし，$obj(i) = Obj[i].name$ である．$Obj[i].name$ は配列 Obj の i 番目の要素の属性 $name$ の値を表わす．以下，同じ記法を使う．

定義 2 $obindex : Obj_n \rightarrow Int_m$ ただし，$obindex = obj^{-1}$ である．obj^{-1} の存在はオブジェクトの名前は重複しないという仮定からくる．

定義 3 $c_{name} : Obj_n \rightarrow C_{name}$ ただし，$c_{name}(obj_n) = Class[Obj[obindex(obj_n)].classname].name$ である．

定義 4 $attrnamelist : C_{name} \rightarrow \{\_id\} \times \bigcup_{k=1}^{\infty}(Attr_n)^k$ ただし，$attrnamelist(c_{name})$ はクラス c_{name} の "_id" を含む全属性名をリストとして並べたものである．例えば，$attrnamelist(company) = (\_id, name, divs)$ である．

定義 5 $attrtypelist : C_{name} \rightarrow \{string\} \times \bigcup_{k=1}^{\infty}(Attr_{type})^k$ ただし，$attrtypelist(c_{name})$ はクラス c_{name} の "_id" を含む全属性のタイプをリストとして並べたものである．例えば，$attrtypelist(company) = (string, string, divisions)$ となる．

定義 6 $attrtype : (C_{name} \times Attr_n) \rightarrow Attr_{type}$ ただし，

$$attrtype(c_{name}, attr_n) = \begin{cases} \text{クラス } c_{name} \text{の属性 } attr_n \text{のタイプ} & if\ attr_n \in attrnamelist(c_{name}) \\ undef & o.w. \end{cases}$$

例えば，$attrtype(company, name) = string$ である．

定義 7 $objV : (Obj_n \times Attr_n \times DataV) \rightarrow ObV$ ただし，

$$objV(obj_n, attr_n, d) = \begin{cases} d & if\ attrtype(c_{name}(obj_n), attr_n) \in \{int, float, string\} \\ obj(d) & if\ attrtype(c_{name}(obj_n), attr_n) \in (attrtypelist(c_{name}(obj_n))) - \{int, float, string\}) \& d \in Int \\ undef & o.w. \end{cases}$$

$attr_n$のタイプ $\notin \{int, float, string\}$ のとき，即ち $attr_n$ のタイプがクラスの名前を表わす

ときは，$d \in DataV$ はオブジェクトのポインタを表す整数になっているとする．

例

えば，$objV(diva, affiliation, d) = compA$．ただし，$obindex(compA) = d$ とする．

定義 8 $entryV : (Obn_n \times Attr_n \times ObV) \rightarrow DataV$ ただし，

$$entryV(obj_n, attr_n, v) = \begin{cases} v & if\ attrtype(c_{name}(obj_n), attr_n) \in \{int, float, string\} \\ obindex(v) & if\ attrtype(c_{name}(obj_n), attr_n) \in (attrtypelist(c_{name}(obj_n)) - \{int, float, string\}) \& v \in Obj_n \\ undef & o.w. \end{cases}$$

例えば，$entryV(diva, affiliation, compA) = obindex(compA) = i \in Int$ である．ただし $Obj[i].name = compA$ とする．

定義 9 $attrlist : C_{name} \rightarrow \{(\_id, string)\} \times \bigcup_{k=1}^{\infty}(Attr)^k$ ただし，$attrlist(c_{name})$ はクラス c_{name} の全属性名とそのタイプの組をデータベース内の表現に合わせてリストとして表現したものに属性 $\_id$ を付加したものである．属性のタイプがクラスの名前のときはそれを int に置換する．例えば，クラス $company$ に対しては，

$attrlist(company) = ((\_id, string), (name, string), (divs, int))$ となる．

定義 10 $initialattrV : Obj_n \rightarrow Obj_n \times \bigcup_{k=1}^{\infty}(DataV)^k$ ただし，$initialattrV(obj_n)$ はオブジェクト obj_n の属性の初期値をデータベース内の表現に合わせてリストとして並べたものである．例えば $compA$ がクラス $company$ のオブジェクトとすると，$initialattrV(compA) = (compA, "", -1)$ となる．データベース内の属性の初期値は次のように決める．

int	0
$float$	0.0
$string$	""
$pointer$	-1
$vecpointer$	$(-1, \cdots, -1)$

関数 1, 2, 3 は配列 Obj と $Class$ から，4, 5, 6 は配列 $Class$ と $Attr$ から決定される．7, 8, 9, 10 は以上の関数の組み合わせとして定義される．以上の関数を使い，実施のメカニズムを表現する．即ち，上記のテーブルと関数がインタープリタを構成する．

クラスとオブジェクトを作るための述語 $<$ $object\ name$ $>:=$ $newdbob(<$ $db\ name$ $>, <$ $class\ name$ $>)$ は次の 2 つの SQL の命令に変換されて実施される．

sql_1 : $create\ table\ <\ class\ name\ >\ (attrlist(<\ class\ name\ >))$

sql_2 : $insert\ into\ <\ class\ name\ >\ (attrnamelist(<\ class\ name\ >))values(initialattrV(<\ object\ name\ >))$

sql_1 の $attrlist(<\ class\ name\ >)$ は class の定義に対応している．

この 2 つの命令が実行されると，データベース内に Figure4 のようにクラスを表現するテーブルが作られる．以下 SQL 言明 sql を実行し，

結果 x を得ることを $x = execsql(sql)$ と書くことにする．ただし，$execsql : \{SQL$ のコマンド $\} \rightarrow DataV$ とする．

次に，データのアサイメントを定義する典型的例として次の述語を考える．

$< object\ name > - >< attr_n >:=< object\ name' > - >< attr'_n >$

これは次の 2 つの SQL 言明により実現される．

$sql_3 : select < attr'_n > from\ c_{name}(< object\ name' >) where\ \_id =< object\ name' >$

$x_3 = execsql(sql_3)$ とする．x_3 に対し，$obj_n = objV(< object\ name' >, attr'_n, x_3) \in Obj_n$ として，新しい SQL 言明を

$sql_4 : update\ c_{name}(< object\ name >) set < attr_n >= entryV(< object\ name >, < attr_n >, obj_n) where\ \_id =< object\ name >$

とする．$x_4 = execsql(sql_4)$ とすればクラス $c_{name}(< object\ name >)$ のオブジェクト $< object\ name >$ の $< attr_n >$ にオブジェクト $obj_n =< object\ name' > - >< attr'_n >$ のポインタがデータベースの要求する形で代入される．

2 章の言語仕様によれば，アサイメントの左辺も右辺も一般に

$< object\ name > - >< attr_n 1 > - > \cdots - >< attr_n l >$

の形でオブジェクトが指定できる．この場合は sql_3 の形の SQL を再帰的に実行することで，必要なオブジェクトが求められる．

$< object\ name > - > []$

の場合は，$< object\ name >$ に対応する属性のリスト $[attr_n 1, \cdots, attr_n l]$ が $attrnamelist(c_{name}(< object\ name >))$ によって求められるので，上記の述語 $< object\ name > - > []$ は下記の形の述語にの組に分解できる．

$< object\ name > - > attr_n k$

分解された述語に対し，sql_3 の形の SQL 言明を構成し，繰り返し実行すれば，$< object\ name > - > []$ の実施が行なうことが可能となる．

最後に method の実施を考える．典型的な例として次の形を考察する．ここでは method の名前を m としている．

$X :=< object\ name > *m(-)$

これを実現する上で問題となるのは，m(-) は関数ではなく Prolog の述語であるので，処理結果を X に代入する特別のメカニズムを必要とすることである．また method の定義の中に “this” という定数オブジェクト名が使われている可能性があり，処理の過程では this を正しく $< object\ name >$ で置換する必要がある．この 2 つの問題のために，特別の Prolog 述語 _THIS(-) と _RETURNVALUE(-) が用意されている．

上記の表現はextPrologでは次のような内部表現に変換される.

$$is(X, action(< object\ name >, m(-)))$$

この内部表現に対して次のステップの実行が行なわれる.

ステップ1 述語_THIS(Y)の内容Yを(Cレベルのプログラムの中で読み出し,それをCのサブルーチンの補助変数にコピーして)セーブする.これはメソッドmの定義の中で他のオブジェクトのメソッドm'を使用した場合,他のオブジェクト名がthisとして使用されることと許すためである.extPrologがthisを正しい< *object name* >と読みかえるためには,m'の実行が終了し,mの処理に戻ったときにthisをmの< *object name* >に戻す必要がある.そのためにYをセーブする.

ステップ2 (Cレベルのプログラムにより)_THIS(-)の中に< *object name* >を挿入し,述語_THIS(< *object name* >)を設定する.

ステップ3 extPrologのインタープリタによりメソッドm(-)を実行する.ただし,mの処理過程では,mの定義の中のオブジェクト*this*を_THISの中に保存されているオブジェクト名< *object name* >で置換する.

ステップ4 メソッドm(-)の述語return(V)によって変数Vの値が_RETURNVALUE(-)に挿入され,extPrologのインタープリタは述語_RETURNVALUE(V)を設定する.

ステップ5 _RETURNVALUE(V)の内のVが関数*action*の値として出力される.

ステップ6 _THIS(< *object name* >)の内容が,セーブされているYに置き換えられ,_THIS(Y)となる.

ステップ7 述語*is*によって*action*の出力となったVの値が変数*X*に代入される.

2章の仕様によると

$$< object\ name > *method_1()*\cdots*method_n()$$

が許されるが,この場合は関数*action*が再帰的に実行される.

証明終

5 おわりに

以上の議論の中でinheritanceとpolymorphismの実現の詳細な議論は行わなかった.inheritanceは,継承している属性をクラスの定義内にコピーする方法で実現している.polymorphismは,我々のPrologインタープリタの実施方法に依存している.methodの定義はmethodclassの宣言の直後に置かれるので,methodの参照は必ずそのmethodclassの宣言

後のルールに対して行うようにしている．これによりpolymorphismが実現されている．

この論文は，オブジェクト指向データベースに興味があるので，オブジェクト指向Prologについて論じてはいないが，この論文での実施はオブジェクト指向Prologの実現にもなっている．

オブジェクトの定義 < *object name* >:= *newdbob*(< *db name* >,< *class name* >) の中で，< *db name* > の所にsystemを指定すると，このオブジェクトはPrologの述語としてPrologのワーキングメモリ上に定義される．補助記憶上でのオブジェクトの表現はリレーショナルデータベースの表現方法に従い，またPrologのワーキングメモリ上での表現はPrologの表現方法に従うので，同じオブジェクトでも両者での表現構造は全く異なる．しかし，どちらの形で表現されていようとも，実行は表現に関係なく，EUの観点からは同じ様に行われる．故に全ての < *db name* > をsystemとすると，2章の仕様はオブジェクト指向Prologを定義することになる．

第2章の操作言語の定式化は，データベースをオブジェクト指向的に取り扱うことを記述しているので，Prologの特徴はほとんど表現されていない．ここでは言語仕様については述べていないが，データのタイプの宣言は“exp”とすることで，タイプなしの表現とすることもできる．これはPrologの特徴であるので，使うことによって便利にはなるが，現在のところメモリの大量使用と実行速度に問題がある．

実現されたシステムで，Prologの特徴はmethodの具体的実現にある．これはPrologで書かれているので，当然Prologの特徴が全て適用される．

第1章で述べた目的である「我々の考えるEUD実現のために，extPrologを，オブジェクト指向データベースとシームレスに結合すること」は実現されている．そして，実現されたオブジェクト指向データベースは予定された機能を発揮しているが，実用上問題となったのは，データベースの実行速度である．PostgreSQLのアクセスのかなりの部分が実のデータのアクセスではなく，ポインタ(虚のデータ)のアクセスであり，データベース上のテーブルを忠実にアクセスすると実行速度はかなり低下する．このため，実際の運用では上述の性質(オブジェクトをワーキングメモリに置いてもよい)を使い，プログラムの実行時に要求されたオブジェクトは主メモリにロードし，Prologのワーキングメモリ上で処理することにしている．プログラムの終了と同時に，ロードされたオブジェクトは自動的に補助記憶に戻される．このような操作は全てシステムが自動的に行うため，EUはこのことについて特に関与することはない．

オブジェクトをこのように主メモリにロードするためには，主メモリが大きいことが必要であるが，幸いメモリ価格の低下によりかなりの主メモリが使えること及び問題解決の

各時点で実際に使うオブジェクトの数は必ずしも大きくないことにより，問題解決EUDのためのデータベース処理では，このような方法が可能になっている．少なくとも我々の行っている例題ではこの通りである．

問題解決EUDのデータ処理と，SQLの意図とは基本的に異なるが，参考文献[13]の問題解決スケルドンのデータモデルの処理でSQLと同じ処理が必要となることがある．このような処理に対して，2種類の言語を使うことより1種類の言語で処理できる方がEUDには当然望ましい．現在のシステムが，どのようにSQLの処理を1種類の言語として取り込むかについては，このシステムの応用として既に参考文献[11]で発表されている．

References

[1] Laudon,K. and et. al, "Information system and the Internet - Problem Solving Approach", Dreydon, 1998

[2] McCabe,F. G., "Logic and Objects", Prentice Hall, 1992

[3] Moss,C., "Prolog++", Addison-Welsley, 1994

[4] Pator,N., R. Cooper, H. Williams and P. Trinder, "Database Programming Languages", Prestice Hall, 1996

[5] PostgreSQL Organization, *http://www.postgresql.org/*

[6] Y. Takahara and Y. Liu, "An Extended Prolog for End User Development", Proc. of World Multiconference on Systemics, Cybernetics and Infotmatics, Orlando, pp.83-90, 1999

[7] Takahara,Y., J. Iijima and N. Shiba, "A Model Management System and its Implementation", Systems Science, Vol. 19, 1992

[8] Ueda, K., "Making Exhaustive Search Programs Deterministic", Proc. Third Int. Conf. on Logic Programming, in E. Shapiro (ed.), *Lecture Notes in Computer Science*, Springer-Verlag, pp.270-282, 1986

[9] 石塚圭樹，「オブジェクト指向データベース」，アスキー出版, 1996

[10] 高原康彦，「数理組織論モデルとGarbage Canモデル」，経営情報学会2000年春季全国大会予稿集, 2000

[11] 高原康彦, 柴直樹, 高木徹，「あるオブジェクト指向データベースにおけるSQL機能の実現」，経営情報学会誌, Vol. 8, No. 3, pp.55-70, 1999

[12] 高原康彦, 高木徹,「Object Oriented Prolog と SQL の比較」, 経営情報学会 2000 年秋季全国大会予稿集, pp.382-385, 2000

[13] 高原康彦, 清水明,「問題解決を目的とした EUD 基礎論 -概念枠組-」, 経営情報学会誌, Vol. 5, No. 3, 1997

[14] 東条 敏ほか,「言語情報処理の枠組としての Quixote」, 人工知能学会誌, Vol.9, No.6, pp.863-874, 1994

[15] トップマネジメントサービス,「Linux/FreeBSD によるデータベース構築入門」, Locus, 1998

[16] 服部正太, 木村香代子(訳),「人工社会 -複雑系とマルチエージェントシミュレーション-」, 構造計画研究所, 1999

[17] 松岡 聡,「ICOT におけるプログラム言語とその実装に関するコメント」, 情報処理, Vol.37, No.5, pp.407-410, 1996

[18] メアリー E. S. ルーミス (野口喜洋訳),「オブジェクトデータベースのエッセンス」, トッパン, 1996

高　原　康　彦
所　属：千葉工業大学 社会システム科学部
　　経営情報科学科
連絡先：〒275-0016
　　千葉県習志野市津田沼 2-17-1
電　話：047-478-0349
E-mail：takahara@mis.it-chiba.ac.jp

柴　　直　樹
所　属：千葉工業大学 社会システム科学部
　　経営情報科学科
連絡先：〒275-0016
　　千葉県習志野市津田沼 2-17-1
電　話：047-478-0351
E-mail：shiba@mis.it-chiba.ac.jp

高　木　　徹
所　属：千葉工業大学 社会システム科学部
　　経営情報科学科
連絡先：〒275-0016
　　千葉県習志野市津田沼 2-17-1
電　話：047-478-0416
E-mail：toru@mis.it-chiba.ac.jp