データベースシステム 114－18
（1998. 1. 20）

複数の包摂関係の矛盾解消のための候補解の生成と環境

中井康裕　三宅孝典　能勢隆雅　横田一正　劉渤江

岡山県立大学情報工学部
{nakai,c06044,nose,yokota}@c.oka-pu.ac.jp
〒719-11 総社市窪木 111

岡山理科大学総合情報学部
liu@penguin.mis.ous.ac.jp
〒700 岡山市理大町 1-1

包摂関係は知識情報処理だけでなく演繹オブジェクト指向データベース (DOOD) などの高度データベースシステムにおいて重要な役割を果たしている。DOOD パラダイムを分散環境に適用するには、複数の包摂関係間の一貫性を維持することが不可欠である。包摂関係は一意的な制約解消のために束を構成しているが、複数の束のマージは前順序にしかならない。しかし半順序生成、束生成のアルゴリズムは、オブジェクトの同定に強い仮定を置いており、分散環境では必ずしもふさわしくない。本稿では、ユーザとの対話によって複数の包摂関係間の矛盾を除去するために、その候補解を求めるアルゴリズムについて述べる。このアルゴリズムは DOOD に基づいたメディエータシステム QUIK で実装されている。

Generation of Candidates for Rosolving Inconsistency among Multiple Subsumption Relations

Yasuhiro Nakai　Takanori Miyake　Takamasa Nose　Kazumasa Yokota

Okayama Prefectural University
{nakai,c06044,nose,yokota}@c.oka-pu.ac.jp
Soja, Okayama 719-11

Bojiang Liu

Okayama University of Science
liu@penguin.mis.ous.ac.jp
Ridai-cho, Okayama 700

Subsumption relation play an important role not only in knowledge information processing systems but also in knowledge-based database systems such as deductive object-oriented database (DOOD) systems. For applying a DOOD paradigm to distributed applications, it is indispensable to maintain the consistency among multiple subsumption relations. Without loss of generality, a subsumption relation is considered as a lattice to solve its constraints, however, merging multiple lattices results in a pre-ordered set. Conventional algorithms for generating a partially ordered set and a lattice assume strong object identification. In this paper, we describe an algorithm, by which we can get some candidate answers for reducing inconsistencies among subsumption relation by interacting with users. The algorithm is implemented in a DOOD-based mediator system, called QUIK.

1 はじめに

データベースシステムをさまざまな知識処理応用に適用させるための高機能化の研究の中で、包摂関係 (is_a 関係) は重要な役割を果たしている。オブジェクト指向アプローチではクラス階層や型階層が中心的な位置を占めているが、そのような汎化関係の取り扱いの形式的な研究は、論理プログラミングの拡張 [1] や演繹オブジェクト指向データベース (DOOD) の研究 [4, 6] の中で行なわれてきた。包摂関係は属性継承と密接に関係しており、多重継承でその解を一意的に求めるなどのためには、包摂関係を束をして、制約解消の対象とすることによりその計算としての扱いが容易になる。F-logic [4] ではこれを行なっていないために多重継承において非決定的な推論が行なわれている。*Quixote*[6, 7] では制約論理プログラミングのスキームで包摂関係を制約解消の対象とすることにより、この問題を解決している。

本稿では、DOOD 言語 *Quixote* を分散環境に拡張した言語 QUIK[9, 5] での束の扱いを説明する。QUIK は、分散環境での DOOD を含む高度な情報源をメディエータシステムとして利用することを目的としており、情報源間の共通モデルやメディエータ仕様記述も QUIK で記述する。しかし束をマージすると前順序集合の性質しかもちえず、そのままでは属性継承で矛盾を生じ利用することができないことがある。それを防ぐために関連するメディエータ間で共通の包摂関係 (束) を生成している。前順序集合から半順序集合の生成手続きの中で、同値類から代表元をとりそれらオブジェクトを同一視するが、分散環境ではこのオブジェクトの同定は強過ぎる仮定である。そこで QUIK では、利用者との対話を通してオブジェクトの同定を行なったり、包摂関係を修正するための、束生成時の矛盾 (同値関係) 除去のための候補解の生成を行なっている。

まず2節では現在実装している、前順序集合から束の生成アルゴリズムを概説する。3節ではそれを分散環境に適用する際の問題点をまとめ、矛盾除去のための候補解生成のアルゴリズムを述べる。4節では実行例を紹介する。

2 束の生成アルゴリズム

現在 QUIK 用に実装している束生成アルゴリズムを説明する。本節では基本的なアルゴリズムを述べ、次節ではそれを分散環境に適用する。

まず入力として包摂関係の集合 $(S, \preceq)$ を考える。上限にトップ $\top$、下限にボトム $\bot$ しかもたない要素は束の生成に影響しないので、ここでは一般性を損なうことなく、入力されないものとして無視することにする。ただし 3 節での分散環境では、結果としてこのノードが階層を持つこともある。

入力から、ノードの配列 node と包摂関係の配列 edge を作成する。edge の各要素は (i,j) (i≠j) の形式をしている。(i,j)、node[i]=a, node[j]= b ならば、これは $a \preceq b$ であることを示している。たとえば $a \preceq b$, $b \preceq c$, $a \preceq c$ の場合、node が [a,b,c] ならば、edge は [(1,2),(2,3),(1,3)] となる。node のノード数を NODENUM とする。

まず包摂関係の推移閉包をとる。これは edge から path を求める再帰関数で、ルールでは一般的には以下のように表現できる。

```
path(X,Y,_) ← edge(X,Y).
path(X,Y,_) ← edge(X,Z),path(Z,Y,_).
```

path の第3引数は後でセットされるノード間の距離のためのものである。このプログラムを closure(edge,path) とする。使用しているアルゴリズムは、上昇評価のセミナイーブ法である [3]。距離の設定は束の生成で変化しうるので後で設定される。

一般的な束生成アルゴリズム (たとえば文献 [2]) では、順序関係に関して同値類を作成しその代表元をとることによって、前順序関係を半順序関係に変換するが、ここでは 3 節の分散環境での束生成を考慮しているために異なったアプローチをとっている。詳細は 3 節で述べ、本節では次に半順序集合から束の生成を考える。

すべての2つのノードに対して、その下限の2つの極大元を求め、最大下限 (glb) を生成する。

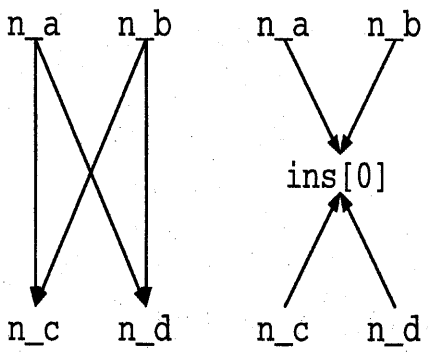
```
generate_glb(node,path)=
  begin;
  for i=1 to NODENUM, j=1 to NODENUM, i ≠ j;
    begin:
    set_glb(i,j,node,path);
    end;
  end

set_glb(i,j,node,path)=
  begin;
    max:=null;
    for k=1 to NODENUM;
      begin;
      if (k,i,_) ∈ path and (k,j,_) ∈ path then
        begin;
        if max=null
        then max:=k;
```

```
        else
          begin;
          if (max,k,_) ∈ path
          then max:=k;
          else
            if ¬ ((k,max,_) ∈ path)
            then new_node(node,path)
          end;
      end;
  end

new_node(node,path)=
  begin;
  NODENUM:=NODENUM+1;
  node[NODENUM]:=node(i,j);
  insert (node(i,j),i,1), (node(i,j),j,1)
    into path;
  insert (k,node(i,j),1), (max,node(i,j),1)
    into path;
  end;
```

node(i,j) が新しく生成されたノードである。これは図示すると、たとえば図1では (a) から (b) のように新たなノードを挿入している。複数の極大元があったときには新しいノードがどこに作られるかは非決定的である。



(a) 束でない例　　(b) 新ノード追加で生成された束

図 1: 束の生成

この generate_glb によって同時に最小上限 (lub) も作成されており、結果は束となっている。

次に 2 つのノード間の距離を設定する。これは ?? 節で述べる分散環境での複数の束の融合によって生じた巡回 (ループ) をなくすために使用する。問題は、推移律によって得られるものの扱い、ループ時の距離をいかに設定するかである。

1) 推移律によって得られないエッジを基本エッジとする。たとえユーザによって与えられたものであっても、推移律によってえられるものは基本エッジの合成とする。
2) 複数回ループする場合を除外する。つまり距離が無限大に大きくなることはない。
3) ループのルートは以下の条件を満たしているもの (複数)
   - ループ外のノードからの基本エッジがある。
   - ループ内の他ノードからの基本エッジですべて下位にはない。

   からひとつが選ばれる。

これは基本的には以下の再帰ルールで表現できる。

```
path(X,Y, ∞) ← path(X,Y,D),X=Y.
path(X,Y,D) ← path(X,Z,D1),path(Z,Y,D2),
                X ≠ Y,X ≠ Z,Z ≠ Y,D=D1+D2.
```

このルールの評価を停止させるために、第3引数を無視した形での、セミナイーブ評価 [3] を行なっている。これによって上記 2 を保証する。これによって 2 つの同一のノードに複数の距離が設定されるが、その中の最大のものをノード間の距離として残しておく。これによって 1を保証している。この関数を set_distance(edge) とする。

これで距離付きの束の生成が完了した。

文献 [2] では n ノードの包摂関係に対して、n×n の行列 M を作成し、推移閉包は $\sum_{i=0}^{\infty} M^i$ で求め、glb は行列の各列または行の論理演算によって生成するアルゴリズムを提案している。計算量の観点からは Aït-Kaci のアルゴリズムの方が優れているが、現在考慮している応用ではそれほど大規模の束は必要としないので、上記のような簡単なアルゴリズムと実装の容易さを優先している。今後データ量が増加が予想されればその変更を予定している。

3 分散環境における束の生成

3.1 束生成時の問題

分散環境の複数の情報源が包摂関係を持っていた場合、それら情報源を融合するためには、包摂関係の一貫性を保証する必要がある。そのために関連する情報源の包摂関係をマージし、大域的な束を必要に応じ生成する必要がある。しかし束のマージでは前順序集合しか得られないので、

- 半順序集合化での同値類の扱い
- 束化での新ノードの扱い

を明確にしておかなければならない。

文献 [8] で議論したように、複数の情報源 m_1, m_2 の包摂関係 (束) をマージした結果、包摂関係 $a \sqsubseteq b, b \sqsubseteq a$ (ただし a, b は基本オブジェクト) が生じたとき、以下の可能性 (組合せを含む) がある。

1) 同じオブジェクトで、m_1, m_2 において $a = b$ である。
2) 名前の置換で、$m_1.a = m_2.b \wedge m_1.b = m_2.a$ である。
3) 異なったオブジェクトであり、新たな識別子 a', b' によって $m_1.a = a'$ または $m_2.b = b'$ として区別する。

1) は半順序集合自動生成時の扱いであるが、他の情報源を更新することはできないので、データディクシナリで変換を行なう機能を持つ。2) は情報源によって用語が異なっている (この場合反転している) ときである。3) はスペルミスの場合も含まれる。一般的に、分散環境での情報源は必ずしも完全なものを期待することはできず、データベースのデバッグを含む試行錯誤的な環境が必要である。そこで QUIK では上記のような巡回を発見したとき、いくつかの可能性を候補解として利用者に提示する。

3.2 矛盾解消の基本的方針

3.1節で指摘したように、束生成時に矛盾 (ループ) が生じたときにはいくつかの解消の可能性がある。どこが矛盾しているかをユーザに提示するのは当然として、さらにどこを修正すべきの候補をユーザに提示することを考える。

前順序関係の集合 $(S, \preceq)$ に対して、パスの集合 $C = \{n_1 \preceq n_2, \cdots\}$ が与えられ、$(S, \preceq) \setminus C$ が束になるとき、C を束の矛盾解消のための候補解という。前順序関係には推移律が成り立つので、C の順序関係は path の距離がすべて 1 ($(n_1, n_2, 1) \in path$) のものだけを考える。候補解 C_1, C_2 があったとき、$C_1 \subseteq C_2$ ならば C_2 は冗長な候補解と呼ぶ。集合 S の要素数 (濃度) を $\overline{\overline{S}}$ で表現するとき、非冗長な候補解 C_1, C_2 が $\overline{\overline{C_1}} < \overline{\overline{C_2}}$ であったとき、C_1 は C_2 より高い可能性を持った候補と呼ぶ。

複数の包摂関係 (束) をマージして前順序関係になったとき、実際には何が問題かを自動的に検出するには不可能であるので、ここでは非冗長な候補解を可能性の高い順番にユーザに提示することにする。

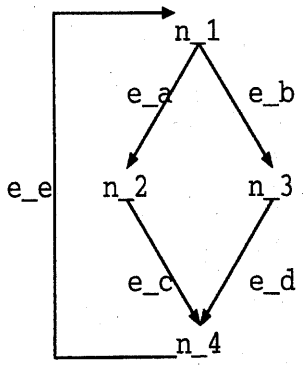
たとえば以下の例を考えよう。

- 入力例 1

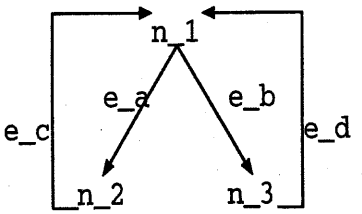
  $\{e_a : n_2 \preceq n_1, e_b : n_3 \preceq n_1, e_c : n_4 \preceq n_2, e_d : n_4 \preceq n_3, e_e : n_1 \preceq n_4\}$
- 入力例 2

  $\{e_a : n_2 \preceq n_1, e_b : n_3 \preceq n_1, e_c : n_1 \preceq n_2, e_d : n_1 \preceq n_3\}$

ここで e_x はエッジで、n_i はノードである。この入力データより得られる包摂関係をそれぞれ図 2(a),(b) に示す。



(a) 入力例 1



(b) 入力例 2

図 2: 巡回する包摂関係の例

図 2(a),(b) での非冗長な候補解はそれぞれ

$$C_a = \{C_1 = \{e_e\}, C_2 = \{e_a, e_b\}, C_3 = \{e_a, e_d\}, C_4 = \{e_c, e_b\}, C_5 = \{e_c, e_d\}\}$$

$$C_b = \{C_1 = \{e_a, e_b\}, C_2 = \{e_a, e_d\}, C_3 = \{e_c, e_b\}, C_4 = \{e_c, e_d\}\}$$

である。C_b に関しては可能性について順位を付けることはできないが、C_a については C_1 がもっとも高い可能性を持っていると判断している。

2節では $\top$, $\bot$ との包摂関係しか持たないノードは無視したが、マージによって他のノードと関係を持つ可能性がある

ので、ここではすべてのノードを対象とする必要がある。また、束のマージによって生じる可能性のある矛盾 (ループ) は必ずしもひとつではないので、各矛盾についてそれぞれ上記のような処理を行なう。

3.3 候補解生成のアルゴリズム

束生成の全体のアルゴリズムは以下のように構成されている。

```
start
入力処理;
closure(edge,path);          % 推移閉包を求める
detect_loop(path);           % 矛盾検知
ユーザとの対話;              % 候補解を表示し修正
generate_glb(node,path);     % 束の生成
set_distance(path);          % 距離の設定
end
```

本節では矛盾検知のアルゴリズムを説明する。以下が概要である。

1) set_distance で設定したノード間の距離の中で距離が (n_i, n_i, ∞) となっているノード n_i を探す。

2) n_i から自分に戻るすべてのルート $r_1, r_2, \cdots, r_n$ を求める。

3) すべてのルート $r_1, r_2, \cdots, r_n$ を分断するため、各ルートにできるだけ共通するエッジ (距離が 1 のパス) を探し、分断する組合せを求める。

4) 3) で求めた組合せ例を濃度の小さい順に並べ替える。

すべてのルートを求めるときに、同じルートを何度も計算する危険性がある。そこで一度通ったルートは (演繹データベースの下降評価と同様に) lemma として記憶し、重複計算を避けている。1) で得られたノードを start とする。

ここでいう start とは、 再帰関数 get_edge を開始するためのノードを指している。以下にこの start を求めるためのアルゴリズムを示す。このアルゴリズムでは再帰関数を用いている。ここでの、start はループしているノードの中からランダムに選択している。

```
search_start(parent)
  begin;
  for i=1 to NODENUM;
    begin;
    if path(i,parent,1) ∈ path and
       ¬ (path(i,parent,1) ∈ lemma)
    then
      begin;
      insert path(i,parent,1) into lemma;
      if i ≠ start then
        begin;
          start_point[i]:=start_point[i]+1;
          search_start(i);
       end;
      end;
    end;
  end
```

この関数 search_start では start_point にノードの枝の数を設定している。これによりどのノードで木が分岐しているか知ることができる。方法は前に述べた再帰関数と同じで、すべてのエッジをたどってノードから複数のエッジが出ていたときに、そのノードから分岐している枝の数を配列 start_point に格納する。終了の仕方はすべてのエッジをたどるか、start (入力時は i) とたどってきたノードが一致したときとする。

```
search_end(parent)
  begin;
  for i=1 to NODENUM;
    begin;
    if path(parent,i,1) ∈ path and
       ¬ (path(parent,i,1) ∈ lemma)
    then
      begin;
      insert path(parent,i,1) into lemma;
      if i ≠ start then
        begin;
          end_point[i]:=end_point[i]+1;
          search_end(i);
       end;
      end;
    end;
  end
```

search_end では、処理の内容は search_start と同じだが、エッジをたどる時に、上からでなく下からたどることによってそのノードにいくつ親があるかを数えて、配列 end_point に格納している。終了の仕方は search_start と同じである。こうすることにより、この二つの関数で分岐の始点と終点を求めることができる。次に unite_start_point では search_start と search_end で得たデータを利用して、最

も距離の長い分岐点を二点求める。これは、分岐の中に分岐がある場合、最も分岐の範囲が大きなものを用いるようにするためである。もし、そうしなければ、分岐の始点と終点がかみあわず、後の処理において誤りを生じてしまう。以下にunite_start_point のアルゴリズムを示す。ここでの fast は search_start や search_end で用いた start と同じである。これは、もし分岐点が存在しない時には始めに選択した start でも不都合が生じないためである。

```
unite_start_point(fast)=
  begin;
  unite_start[0]:=0;
  unite_start[1]:=fast;
  for i=1 to NODENUM;
    begin;
    if start_point[i] ≠ 1 then
      begin;
      for j=1 to NODENUM;
        begin;
        if end_point[j] ≠ 1 then
          begin;
          let path(i,j,X);
          if unite_start[0]<X then
            begin;
            unite_start[0]:=X;
            unite_start[1]:=i;
            end;
          end;
        end;
      end;
    end;
    return unite_start[1];
  end;
```

この関数では、分岐の始点と終点の距離が最も大きなものを探し、そのエッジの始点を返している。こうすることにより start が求まり、このデータを用いて次の関数の処理の始点としている。

```
get_edge(parent,count,start,lemma)=
  begin;
  count:=count+1;
  for i=1 to NODENUM;
    begin;
    if path(i,parent,1) ∈ path and
       ¬ (path(i,parent,1) ∈ lemma)
    then
      begin;
      insert path(i,parent,1) into lemma;
      if i ≠ start
        then get_edge(i,count,start,lemma);
      end;
    end;
  end
```

この関数は start から下の (ループを含めた) すべてのエッジを重複することなく求めている。

次にこの lemma からループするルートを抽出する。ループしているので、start から逆に上方向に lemma のエッジをたどることによって、ループしているエッジのみを求めることができる。

```
get_route(start,lemma,route)=
  for i ∈ S such that path(start,i,1) ∈ lemma
    begin;
    set_route(path(start,i,1),route);
    if i ≠ start then get_route(i,lemme,route);
    end
```

ここで set_route は枝別れの数だけそれまでのルートをコピーし、異なったルートとして登録することを表している。

route には各ルートがエッジの配列として入っているので、それを適当に並べ替えることによって、分断可能なエッジ、つまり候補解を可能性の高い順番に得る。

4 実行例

ここでは、実際に例を挙げて、矛盾の処理を行なってみる。入力例として以下のものを使用する。

$$\{e_a: n_2 \preceq n_1, e_b: n_3 \preceq n_1, e_c: n_4 \preceq n_2, e_d: n_5 \preceq n_2,$$
$$e_e: n_4 \preceq n_3, e_f: n_5 \preceq n_3, e_g: n_6 \preceq n_4, e_h: n_6 \preceq n_5,$$
$$e_i: n_1 \preceq n_6\}$$

まず始めに、図3のウインドウから、束の包摂関係を入力するために PROGRAM WINDOW を開く。

束の包摂関係を入力したら、PROGRAM WINDOW の SAVE ボタンをクリックして、データをセットする。なお、DELETE ボタンは データ を消去するボタンである (図4参照)。

次に、図3のウインドウから、LATTICE WINDOW を呼びだし、show ボタンを押すことで、包摂関係を視覚的に表示する。もし、ここでデータに矛盾があった場合は、ダイアログが表示されるので、図5のウインドウの表示に従って、消去する EDGE を選択する。

入力例では e_i だけが矛盾の候補解となるので、1 を入力する。ここまでの処理で入力例の包摂関係が LATTICE

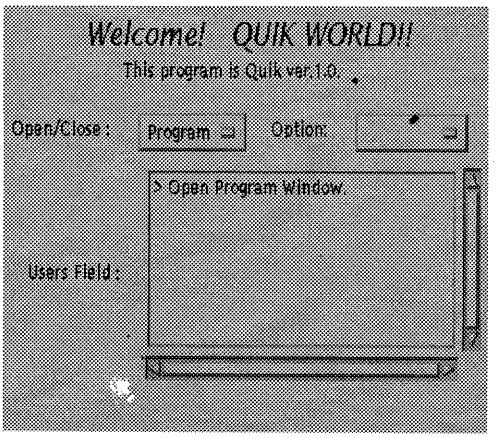


図 3: MAIN WINDOW

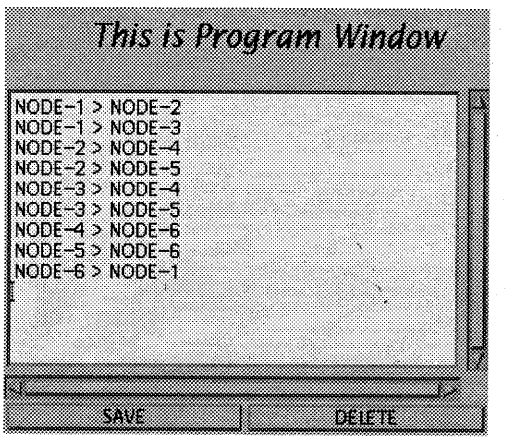


図 4: PROGRAM WINDOW

WINDOW に表示される (図 6参照)。

5 おわりに

本稿で述べた束生成のプログラムは Java で実装され、DOOD に基づいたメディエータシステム QUIK で使用されている。今後の検討点としては

- データディクショナリ (あるいは現在検討を進めているオントロジー) とより関連させた、半順序集合生成時のオブジェクトの同定機構、およびユーザとの対話に基づく修正情報のデータディクショナリへの反映。
- 必要に応じて動的に生成するのは効率がよくないので、生成した束を実体化ビューとして保存し、差分による更新を可能にすること。
- Aït-Kaci アルゴリズムの評価によって、データ量の増加に対応したアルゴリズムの改良。
- ループチェックやユーザインタフェースは部品階層などにも適用可能なので、その適用範囲の拡張の検討。

などを考えている。本プログラムは QUIK と一緒にフリーソフトウエアとしてリリースする予定である。

謝辞

分散環境での束生成のプログラムを最初に Java で実装した京都大学中西英之氏、および日常的に議論して頂いている岡山県立大学の QUIK の研究開発グループに感謝する。

参考文献

[1] H. Aït-Kaci, "An Algebraic Semantics Approach to the Effective Resolution of Type Equations," *Theoretical Computer Science*, no.45, 1986.

[2] H. Aït-Kaci, P. Boyer, and P. Lincoln, "Efficient Implementation of Latice Operation", *TOPL*, vol.11, no.1, pp.115-146, 1989.

[3] F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", *SIGMOD'86*, 1986.

[4] M. Kifer and G. Lausen, "F-Logic — A Higher Order Language for Reasoning about Objects, Inheritance, and Schema," *Proc. ACM SIGMOD International Conference on Management of Data*, pp.134-146, Portland, June, 1989.

[5] 緒方啓孝、堤慎一郎、横田一正 "QUIK メディエータでの問合せ機能の拡張", 情報処理学会第 55 回全国大会、4AA-10, 1997 年 9 月.

[6] K. Yokota and H. Yasukawa, "Towards an Integrated Knowledge-Base Management System — Overview of R&D on Databases and Knowledge-Bases in the FGCS Project", *Proc. Int. Conf. on Fifth Generation Com-*

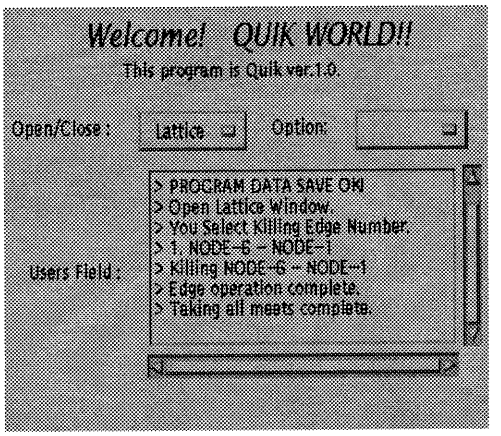


図 5: MAIN WINDOW

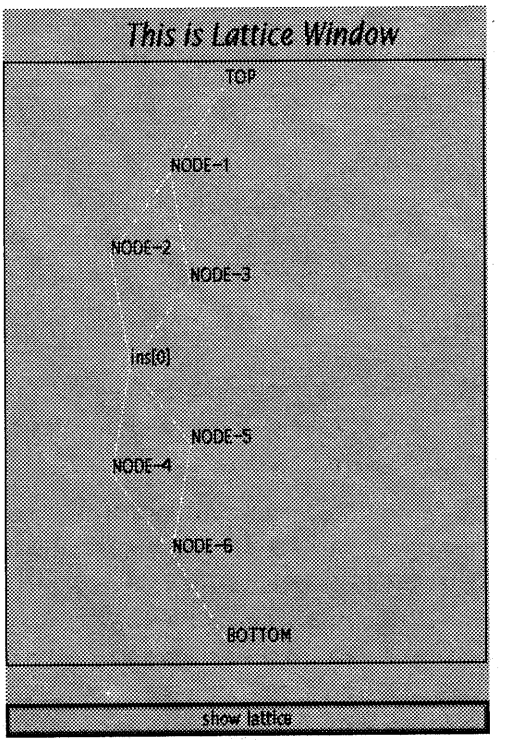


図 6: LATTICE WINDOW

puter Systems (FGCS'92), pp. 89-112, Tokyo, June, 1992.

[7] K. Yokota, T. Nishioka, H. Tsuda, and S. Tojo, "Query Processing for Partial Information Databases in *Quixote*", *6th IEEE International Conference on Tools with Artificial Intelligence (TAI'94)*, pp.359-365, New Orleans, Nov., 1994.

[8] 横田一正、萬上裕、黒田崇、國島丈生、"メディエータにおけるオブジェクト再考"、情報処理学会データベースシステム研究会&電子情報通信学会データ工学研究会合同ワークショップ, 97-DBS-113-54, pp.323-328, 札幌、Jul., 1997.

[9] Kazumasa Yokota, Yutaka Banjou, Takashi Kuroda, and Takeo Kunishima, "Extensions of Query Processing Facilities in Mediator Systems", *Proc. International Workshop on Knowledge Representation Meets Databases (KRDB'97)*, pp,17.1-8, Greece, Aug. 30,, 1997.