概念ネットワークによる知識表現システム

渡辺正信
（日本電気（株）C&Cシステム研究所）

1. はじめに

人間の知的活動の一部を代行する大規模・複雑なシステムを計算機上に構築するためには、人間の知識を適宜表現・活用できるプログラミングの枠組を整備する必要がある。本稿は、この枠組の最も基本となる知識表現システムの試作を通して，そのあるべき姿を模索した結果について報告するものである。

まず，知識表現システムのポイントは、次の3点にある[1]～[4]。

(i) 表現力：多種多様な知識（例えば，宣言的・手続き的知識，オブジェクト・制御・メタ知識等）を陽に表現できる。この場合、可読性，自然さが重要となる。
(ii) 柔軟性：新しい知識の設定，既存の知識の変更が容易にでき，拡張性に富む（自己記述的である）。このためには、一様な表現枠組と効果的モジュール化力を保持することが必要。又、柔軟性は、知識獲得・学習へつながる。
(iii) 構造化力：知識を適宜管理・維持・活用するためには、その知識が体系的に整理されていなければならない。この体系化を促進する機構の内在がシステムに求められる。汎化階層としての表現・管理機構は、この最も典型的なもののひとつである。

さて，現在，知識表現での代表は、プロダクションルール（以下単にルールと呼ぶ）とフレーム（意味ネットを含む）である[5]～[7]。

ルールの利点は、モジュラリティと一様性にある。しかし，個々のルールが独立しているというモジュラリティの高さは、必然的にルール間の構造化力が弱いということになる。又，制御，メタ・手続き的知識をルールの一様な形式で表現することには無理があり表現力に欠ける。つまり、従来ルールの利点として強調されたことには、知識表現における本質的な問題を内在させていたことになる。

ルールに関する問題の多くは、フレームにより解決される。一方、フレームの追加・更新・削除が他のフレームとの関連により複雑化すること，副作用により状態の管理が複雑化すること等で、柔軟性に欠ける欠点がフレームにある。この欠点にもかかわらず，フレームは、対象世界に内在する概念を体系化・構造化するということから知識表現における基本的枠組を提供する。

本稿は、このフレーム表現を基本にし，その欠点を補うため，概念ネットワーク（C-NET）を使って構造化力・柔軟性を高めた知識表現システム（COMET）を提案する。

COMETは、フレームに基づく知識表現においてスキーマの動的変更を強力に支援するシステムとして設計され以下の3つの特長をもつ。

(i) メタスキーマによるスキーマ管理とメタスキーマの自己管理。
(ii) 逆関係を明示することによりポインタ管理の自動化。
(iii) 副作用を起こす付加手続き管理の強化。データ又は手続きの変更に対する柔軟性が高い。

2. COMETシステム概要

COMETシステムは、図1に示されるように，COMETインタープリターと概念ネットワーク(C-NET)から構成される。以下，これらの概要を示す。

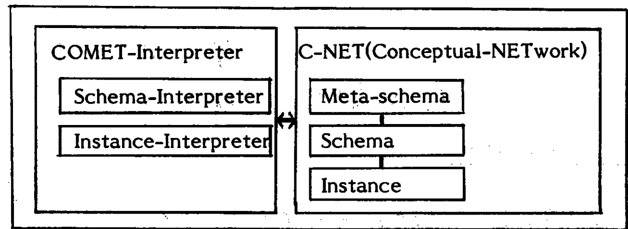


図1. COMETシステム構成

2.1. 概念ネットワーク(C-NET)

スキーマ，インスタンスの考えに基づく知識の構造化に関しては、Davis[8]やSMALLTALK[9]等で既に提案され,その有効性が認識されつつある。COMETにおいても,フレーム表現形式上で，メタスキーマ，スキーマ，インスタンスをノードとする概念ネットワーク(C-NET)を生成し管理する方式をとる。C-NETの基本的考え方は、メタスキーマ，スキーマ，インスタンス階層での上位レベルのものが下位レベルのものの作成，修正，削除処理に関する知識を保持し，利用者と対話的にその処理を実現することである。さらに,この概念ネットワークを矛盾の無い一貫性あるものとして管理・維持し，段階的に発展させていくということである[10][11]。このことは、フレームでの変更に対する柔軟性を高めるため，利用者が提示した変更に従い，他の関連ある処理をシステムが受けもつことを意味する。

尚，本稿でのフレーム表現形式とはFRL[2]と同じく(フレーム，スロット，ファセット，バリュー)の4つ組みのことである。

メタスキーマに関しては、3.で詳述する。

2.2. COMETインタープリター

利用者と対話形式で概念ネットワークを作成・変更していく制御機構としてCOMETインタープリターがある。COMETインタープリターは、以下に示すスキーマ・インタープリターとインスタンス・インタープリターからなる。

(i)スキーマ・インタープリター：メタスキーマに表現されているスキーマの構造，処理手順を参照して利用者と対話形式でスキーマの管理維持(スキーマ作成・修正・削除)を行なう。さらに，後述する核メタスキーマを基に周辺メタスキーマの管理維持を行なう。このため，スキーマインタープリターは、スキーマ作成部，修正部，削除部から構成される。

(ii)インスタンス・インタープリター：スキーマに表現されているインスタンスの構成，処理手順を参照して利用者と対話形式でインスタンスの管理維持を行なう。インスタンス作成部，修正部，削除部から成る。

スキーマ・インタープリター，インスタンス・インタープリター共に同じフレーム表現形式の知識を解釈実行する。従って、その基本的考え方は同じである。しかし，スキーマの下位にはインスタンスがあるが，インスタンスは最下位であり自分より下位には何も無いことから，COMETにおいては、スキーマ・インタープリターとインスタンス・インタープリターを別々に設けた。尚，前者は、4.で詳述する。

3. メタスキーマ

メタスキーマは、COMETシステムを最も特徴づけるものである。メタスキーマは、概念ネットワーク(C-NET)において、スキーマの作成，修正，削除処理手順を提供するだけでなく，メタスキーマ自身の自己管理手順を保持している。すなわち，以下に示す核メタスキーマに基づき，周辺メタスキーマの作成，修正，削除処理手順を提供する。

(i) 核メタスキーマ
- ①主メタスキーマ(SCHEMA)
- ②スロット・メタスキーマ(SLOT)
- ③メタスキーマ共通スロット作成メタスキーマ(例えば，META-TYPE)

(ii) 周辺メタスキーマ
- ①スキーマ共通スロット作成メタスキーマ(例えば，INStance，TYPE SCHema，INStance)
- ②ファセットディフォルト値設定メタスキーマ(Facet-Value-Defaults)

以下，(i)(ii)について説明する。

3.1. 核メタスキーマ

核メタスキーマを図2に示す。スキーマ全体の管理を主メタスキーマが，スロットの管理をスロット・メタスキーマが受け持つ。以下，各々の核メタスキーマについて説明する。

(i) 主メタスキーマ(SCHEMA)

スキーマ管理(作成・修正・削除)処理の主手順を提供するもので，以下のスロットから構成される。

①schema-name：スキーマ名の設定手順(to-fill)，変更手順(if-updated，if-removed)を保持する。但し，図2には付加手続き名(バリュー部)は省略している。

```
SCHEMA
    type
      value (Meta-schema)
    schema-name
      to-fill
      if-updated
      if-removed
    meta-schema-slot
      to-fill
      value (META-TYPE)
    common-slot
      to-fill
      value (INStance-NAME
             TYPE SCHema
             INStance)
    general-slot
      to-fill
      value (attribute-of
             object-of)
    optional-slot
      to-fill
    document
      to-fill
    ending
      to-fill
    potential-schema
      to-fill
    creating-order
      value (schema-name
             meta-schema-slot
             common-slot
             general-slot
             optional-slot
             document
             ending type)

META-TYPE
    value
      to-fill
    type
      value
      (Meta-schema)
    super
      value(SLOT)

SLOT
    slot-name
      to-fill
      if-updated-on
      -frame-name
      if-updated
      if-removed
    value-type
      to-fill
    value
      to-fill
      to-precheck
      if-updated
      if-removed
      if-added
    reverse-relation
      to-fill
      to-precheck
      if-updated
    generating-type
      to-fill
      to-precheck
      restrict(one-of
      (ask auto)
      default(ask)
    to-fill
      to-fill
      to-precheck
    when-filled
      to-fill
      if-updated
    if-updated
      to-fill
    if-updated-on
    -frame-name
      to-fill
    if-removed
      to-fill
    if-added
      to-fill
    deleting-type
      to-fill
    default
      to-fill
      if-updated
    restrict
      to-fill
      if-updated
    to-acknowledge
      to-fill
    type
      value
      (Meta-schema)
    creating-order
```

図2. 核メタスキーマ

②meta-schema-slot：周辺メタスキーマの共通スロット作成主手順とそのスロット名(META-TYPE)を保持する。

③common-slot：スキーマの共通スロット作成主手順とそのスロット名(valueのバリュー部)を保持する。

④general-slot：スキーマの準共通スロット作成主手順とそのスロット名を

保持する。

⑤ optional-slot：各スキーマ固有のスロット作成の主手順を保持しスロット・メタスキーマSLOTの各スロットのto-fill付加手続きを起動する。

⑥ document：スキーマ管理情報（作成，修正者名とその日付等）の管理手順の保持。

⑦ ending：スキーマ作成終了処理手順を保持。例えば，スロットの出力順(printing-order)，作成順(creating-order)削除順(deleting-order)の実行手順。

⑧ potential-schema：概念ネット状態管理情報（未定義スキーマ名）の保持。

⑨ type：メタスキーマ，スキーマ，インスタンスの区別用スロット。

(ii) スロット・メタスキーマ(SLOT)

スキーマの準共通，固有スロットの作成と，スキーマの各スロットの修正・削除処理手順を保持する。従がってスロット・メタスキーマのスロットは，基本的に，スキーマ・フレームのファセットから構成される。slot-name，type，creating-orderスロット等はスロット・メタスキーマ固有のものである。

① slot-name：スロット名の作成・修正・削除処手順の保持。

② value-typeからrestrictまで：スキーマのファセットとなる。スロットの値の型(value-type)，作成型(generating-type)，削除型(deleting-type)，値(value)ディフォルト(default)，値の制限(restrict)逆関係(reverse-relation)，各種付加手続きトークン(to-fill，when-filled，if-updated，if-updated-on-frame-name，if-removed，if-added)からなる。ここで，if-updated-on-frame-nameは，フレーム名の変更時に起動されるトークンである。

③ to-acknowledge：スロット作成確認処理手順を保持する。

```
INStance-NAME
    generating-type
        to-fill
    to-fill
        to-fill
    type
        value
        (Meta-schema)
    super
        value(SLOT)

TYPE
    to-fill
        to-fill
    value
        to-fill
    generating-type
        to-fill
    type
        value
        (Meta-schema)
    super
        value(SLOT)

SCHema
    to-fill
        to-fill
    generating-type
        to-fill
        default(auto)
    if-removed
        to-fill
    if-updated-on
    -frame-name
        to-fill
    type
        value
        (Meta-schema)
    super
        value(SLOT)

INStance
    to-fill
        to-fill
    generating-type
        to-fill
        default(auto)
    type
        value
        (Meta-schema)
    super
        value(SLOT)

Facet-Value-Defaults (Example)
        ∫
    object-of
        reverse-relation (attribute-of)
        generating-type (auto)
        if-removed ((son-delete))
        value-type (schema-name)
    attribute-of
        reverse-relation (object-of)
        if-removed ((link-delete))
        value-type (schema-name)
        ∫
```

図3. 周辺メタスキーマ

(iii) メタスキーマ共通スロット作成メタスキーマ

周辺メタスキーマに必須のスロットの作成手順を保持する。主メタスキーマ(SCHEMA)のmeta-schema-slotのto-fill手続きで起動される。図2のMETA-TYPEメタスキーマがこの例である。このMETA-TYPEは，周辺メタスキーマのTYPEスロットのValue値としてMeta-schemaを設定する。

3.2. 周辺メタスキーマ

周辺メタスキーマを図3に示す。周辺メタスキーマは，核メタスキーマの下で作成，管理される。ここでは，以

下に示すスキーマ共通スロット作成メタスキーマとファセット・ディフォルト値設定メタスキーマから成る。

(i)スキーマ共通スロット作成メタスキーマ

スキーマが共通に保持すべきスロットの作成処理手順を保持するメタスキーマである。主メタスキーマ(SCHEMA)のcommon-slotのto-fill付加手続きで起動される。図3では4つのメタスキーマ例を示している。

① INStance-NAME：スキーマのINStance-NAMEスロットを作成する。このスロットは、インスタンス名の作成・修正・削除処理手順を保持する。
② TYPE：スキーマのTYPEスロットのvalue値にschemaを設定する。
③ SCHema：スキーマのSCHemaスロットを作成する処理手順を保持する。このスロットは、そのスキーマが作成管理するインスタンスのSCHemaスロットにそのスキーマ名を設定し，管理する。
④ INStance：SCHemaの逆関係スロットで，スキーマが管理するインスタンス名の管理手順を保持する。

尚，以上①②のgenerating-typeのdefault値は，SLOTメタスキーマのdefault値(ask)が取られ，③④のそれは(auto)が取られる。

(ii)ファセットディフォルト値設定メタスキーマ(Facet-Value-Defaults)

特定スロットのファセットに対するディフォルト値情報を保持する。スキーマ作成時に，ここに登録されたディフォルト値が仮設定されて作成者の確認を問う。この確認は，SLOTメタスキーマのto-acknowledgeスロットで行なわれる。当然，仮設定値の変更支援機構を保持する。

4. スキーマ・インタープリター

スキーマ・インタープリターは、メタスキーマの知識を利用して，スキーマの作成・管理，および，周辺メタスキーマの作成・管理を実現する制御機構である。以下，スキーマと周辺メタスキーマの作成・修正・削除処理方式を説明する。

4.1. スキーマ作成

前述のメタスキーマの各スロットのto-fill付加手続きを起動することによりスキーマ作成を実現する。尚，スロット作成順序は，メタスキーマ(SCHEMA)のcreating-orderスロットに登録されたスロット順に従う。

4.2. スキーマ変更(修正，削除)

ここでは、スキーマ変更とは、スキーマ名(フレーム名)，スロット名，バリューの変更を意味する。すなわち，ファセットの変更はメタスキーマ(SLOT)の変更を通して行なわなければならない。さらに制御用ファセット(付加手続きトークン)の変更は，インタープリター自身の変更を必要とする。

まず，スキーマ変更に対する一般的処理手順を図4に示す。この図で矩形

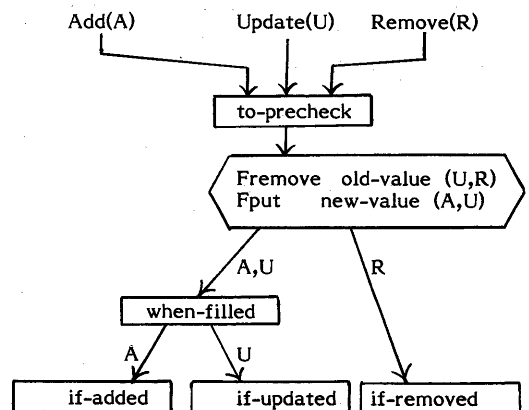


図4. スキーマ変更処理手順

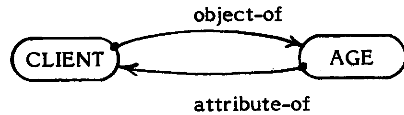
は、変更制御用ファセット(付加手続きトークン)を示す。to-precheckは、変更可能な環境であるかを変更実行前に確認するものである。例えば、インスタンスが既に存在するスキーマを変更する時は、利用者に適宜指示を問う。when-filledは、変更後の値の正当性を自分自身の制約条件の下で検査する。さらに、if-added, if-updated, if-removedにより、必要に応じて他への副作用を起こさせる。

スキーマ変更処理でのポイントは、ポインタ管理と、データと手続きの一貫性管理である。この後者は、4.3.で説明するので、ここでは、ポインタ管理方式を説明する。

▶逆関係によるポインタ管理方式◀

概念間の関係は、一般に方向性をもち、非可逆的である[12]。さらに、フレーム間の関係は、スロットで表現される。そこで、COMETでは、関係表現スロットのファセットにreverse-relationを用け、そのスロットの逆関係を明示する方式を取った。このことにより、変更に対するポインタ管理の自動化が促進された。

この考えを図5を使って説明する。今、2つのスキーマ(CLIENT(顧客)とAGE(年令))が、CLIENTはAGEのオブジェクト(object-of)であり、AGEはCLIENTの属性(attribute-of)であるという関係で結ばれているとする。つまり、object-ofとattribute-ofという2つの関係は、互いに他の逆関係となっている。この時、スキーマ名(CLIENT)を変更すれば、同時にAGEスキーマのattribute-ofスロットのvalue値(CLIENT)も変更しなければならない。COMETのスキーマインタープリターは、CLIENTのobject-ofスロットのvalue値とreverse-relation値に基づき以上の変更処理を自動的に行なう。この原理は、スロット名の変



CLIENT
object-of
value (AGE)
reverse-relation (attribute-of)

AGE
attribute-of
value (CLIENT)
reverse-relation (object-of)

図5. 逆関係の明示例

更に関しても同様に適用できる。

一方、スキーマの削除処理は、そのスキーマのcreating-orderスロット(インスタンス作成用)に登録された順で実行される。この時、削除されるスキーマのすべてのインスタンスも、同時に削除される。又、そのスキーマと関係をもつスキーマの削除は、利用者の確認を取りながら必要に応じて実行される。

4.3. メタスキーマ作成・変更

周辺メタスキーマの作成・変更は、スキーマの作成・変更と同様の処理手順で実現される。又、スキーマでのファセットの追加は、SLOTメタスキーマへのスロットの追加として実現される。ただし、追加したファセットは、利用者の管理下となる。

4.4. 付加手続き管理

COMETでは、概念ネットワーク(C-NET)のノード単位に知識が構造化され、それに付加された手続きのモジュール化が促進されていく。従がって、ノードにどのような手続きが付加されているかということと、その手続きはどのノードを参照又は、更新するかということを、システムとし管理し、ノード又は手続きの変更に対する知識ベ

ースの一貫性を保持することは、段階的にプログラムを構築していく利用者にとって非常に強力な機能となる。

以上の観点から、COMETでは以下の付加手続き管理を実現している。

(i)付加手続き使用場所(フレーム名,スロット名,ファセット名)の管理
(ii)付加手続きの参照,又は,更新場所管理
(iii)定義済付加手続き名管理
(iv)未定義付加手続き名管理

尚、上記(ii)に関しては、利用者が、付加手続きを定義する時、その手続きが参照又は更新する場所(フレーム名,スロット名,ファセット名)を,コメント文として明示する必要がある。このコメント文には、参照文(DEMON-GET)と,更新文(DEMON-PUT)がある。

5. 利用例

ここでは、スキーマのスロット名の変更を行なう場合の対話例を示す。例は、文献[10]で示したスキーマに関するものである。図6は、2つのスキーマFUNCTIONとAPPLY,およびそのインスタンスを示している。FUNCTIONとAPPLYは、APPLY-CHKとFUNCTION-NAME関係で結合されている。この2つの関係は互いに他の逆関係となっている。

図7は、FUNCTIONスキーマのAPPLY-CHKスロット名をAPPLY-CHECKに変更する対話例である。逆関係を利用してAPPLYスキーマのFUNCTION-NAME

```
FUNCTION (スキーマ)
  S
 APPLY-CHK
  VALUE-TYPE : ( SCH-NAME )
  VALUE : ( APPLY )
  REVERSE-REL : ( FUNCTION-NAME )
  GEN-TYPE : ( ASK )
  IF-UPDATED-FN : ( (GENERAL-UPFN-LINK) )
  IF-REMOVED : ( (APPLY-CHK-DEL) )
  IF-ADDED : ( (GENERAL-CREATE-LINK) )
  TO-FILL : ( (F-ASK-APPLY-CHK) )
  IF-UPDATED : ( (APPLY-CHK-UPDATED) )
  REFERRED-BY : ( VAL-CHECK )
 S

APPLY (スキーマ)
  S
 FUNCTION-NAME
  VALUE-TYPE : ( SCH-NAME )
  VALUE : ( FUNCTION )
  REVERSE-REL : ( APPLY-CHK )
  GEN-TYPE : ( AUTO )
  TO-FILL : ( (AUTO-NIL) )
  IF-REMOVED : ( (SON-DELETE) )
  IF-UPDATED-FN : ( (F-APPLY-CHK-UPDATE) )
 S

GREAT (インスタンス)
 TYPE
  VALUE : ( INSTANCE )
 SCH
  VALUE : ( FUNCTION )
 APPLY-TYPE
  VALUE : ( NUM )
  INHERIT : ( F )
 APPLY-CHK
  VALUE : ( GREAT-CHECK )

GREAT-CHECK (インスタンス)
 TYPE
  VALUE : ( INSTANCE )
 SCH
  VALUE : ( APPLY )
 FUNCTION-NAME
  VALUE : ( GREAT )
 NUM
  VALUE : ( (VALUE LT MAX) )
```

図6. スキーマ,インスタンス例(変更前)

```
********************************
***   C O M E T - S T A R T   ***
********************************
                         Ver 1.0

*** What do you do ? ***
*** Please select ! ***
     FL   (FRAME LIST)
     P*   (PRINT ALL)
     CF   (CREATE FRAME)
     CFV  (CREATE FRAME VERIFY)
     EF   (EDIT FRAME)
     EFV  (EDIT FRAME VERIFY)
     DL   (DEMON LIST)
     D*   (DEMON PRINT ALL)
     FIN  (COMET-JOB END)
ANS ? EFV
*** What do you edit ? ***
  ---> M  (Meta-schema)
       S  (Schema)
       I  (Instance)
       E  (END)
ANS ? S
*** What do you do for edit ? ***
  ---> N   (schema-name update)
       L   (slot edit)
       V   (value edit)
       D   (schema delete)
       P   (schema print)
       E   (END)
ANS ? L
*** What do you do ? ***
*** Please select !  ***
  ---> U  (slot-name update)
       A  (slot add)
       D  (slot delete)
       E    (END)
ANS ? U
```

(つづき)

```
*** What is a schema-name with the slot to be updated ? ***
ANS ? FUNCTION
*** What is the slot-name to be updated ? ***
ANS ? APPLY-CHK
*** What is the new slot-name ? ***
ANS ? APPLY-CHECK
*** FREMOVE -- APPLY FUNCTION-NAME REVERSE-REL nil ***

***  FPUT   --- APPLY FUNCTION-NAME REVERSE-REL APPLY-CHECK ***

  APPLY
    FUNCTION-NAME
     VALUE-TYPE : ( SCH-NAME )
     VALUE : ( FUNCTION )
     GEN-TYPE : ( AUTO )
     TO-FILL : ( (AUTO-NIL) )
     IF-REMOVED : ( (SON-DELETE) )
     IF-UPDATED-FN : ( (F-APPLY-CHK-UPDATE) )
     REVERSE-REL : ( APPLY-CHECK )              <=
***  FPUT   --- SCH-MAINTE DEMON-PROMPT VALUE VAL-CHECK ***
*** FREMOVE --- FUNCTION APPLY-CHK nil nil ***

*** Please update this demon ***
       VAL-CHECK                                <=
*** FREMOVE --- SCH-MAINTE DEMON-PROMPT VALUE nil ***

FUNCTION
  APPLY-CHECK                                   <=
   VALUE-TYPE : ( SCH-NAME )
   VALUE : ( APPLY )
   REVERSE-REL : ( FUNCTION-NAME )
   GEN-TYPE : ( ASK )
   IF-UPDATED-FN : ( (GENERAL-UPFN-LINK) )
   IF-REMOVED : ( (APPLY-CHK-DEL) )
   IF-ADDED : ( (GENERAL-CREATE-LINK) )
   TO-FILL : ( (F-ASK-APPLY-CHK) )
   IF-UPDATED : ( (APPLY-CHK-UPDATED) )
   REFERRED-BY : ( VAL-CHECK )
```

図7. スキーマのスロット名変更対話例

```
(defun VAL-CHECK nil
   (DEMON-GET ((FUNCTION APPLY-CHK))) ⇦
        S
(cond
   ((or (null
        (setq FN      (FGET INSNAME
                       (quote FUNCTION)
                       (quote VALUE))))
        S
(cond
   ((or (null
        (setq JINS   (FGET FN
                      (quote APPLY-CHK)
                      (quote VALUE))))
        S
```

図8. 付加手続き内のデモン文例

スロットの reverse-relation 値が更新される。さらに、APPLY-CHK を陽に利用している付加手続き名 VAL-CHECK を表示し、その付加手続きの変更を促す。この手続き名は、referred-by ファセットの値から摘出される。尚、referred-by ファセットへの手続き名の設定は、付加手続きファイルをオープンするとき各手続きのデモン文（参照又は更新文）に基づき実行される。このデモン文の例を図8に示す。

6. おわりに

概念ネットワーク(C-NET)に基づく知識表現システム（COMET）を紹介した。特に、メタスキーマとスキーマインタープリタに関して詳述した。COMETは、フレーム型知識ベースシステムでの変更に対する柔軟性を高めるため、メタスキーマの設定，逆関係明示によるポインタの自動管理，付加手続き管理を実現している。尚，COMETは、LISP上で現在稼働中である。

一方，COMET利用の応用システムとしては文献[11]で発表したMOSレイアウト設計支援システムを現在作成中である。そこでは、設計概念の整理，設計ルールの組込みを行ない，必要に応じてCOMET自身のバージョンアップを段階的に実現している。この作業を通して，COMETシステム自身の柔軟性の高さを確認した。

又，階層関係上での、インヘリタンス機能の実現は、メタスキーマの定義を通して行なわれる。つまり、COMETシステム（特に、インタープリタ）は、より汎用の知識表現システムツールといえる。

今後の課題として，次の3点をあげる。

(i) 付加手続き間呼び出し関係のより効果的管理方式の検討。
(ii) COMETインタープリタ自身のさらなる汎用化。現バージョンでは、作成・変更手順が固定化されている。
(iii) 応用システムを通した検証。

最後に，本研究の機会を与えていただいた当研究所コンピュータシステム研究部箱崎勝也部長，山本昌弘課長，および，有益な示唆をいただいた牧野武則主任にそれぞれ深謝いたします。

REFERENCES

[1] Bobrow,D.G. and Winograd,T.,"An Overview of KRL, A Knowledge Representation Language," Cognitive Science 1,1, 1977, pp.3-46.
[2] Roberts,R.B. and Goldstein,I.P.,"The FRL Mannual," MIT AI-memo 409, 1977.
[3] Stefic,M.J. and Bobrow,D.G.,"The LOOPS Mannual : A Data Oriented and Object Oriented Programming System for Interlisp," Xerox Knowlege-Based VLSI Design Group Memo KB-VLSI-81-1,1981.
[4] Mizoguchi,F.,"Tutorial on Knowledge Representation," Draft of Meeting at Kyoto University, 1982(6/26).
[5] Aikins,J.S.,"Prototypical Knowledge for Expert Systems", Artificial Intelligence 20, 1983, pp.163-210.
[6] Stefic,M.J.,"Planning with Constraints," STAN-CS-80-784, 1980.
[7] 上野，"フレーム理論に基づく知識型システム，"人工知能と対話技法研究会，21-3，1981(6/19).
[8] Davis,R.,"Knowledge Acquisition in Rule-based Systems : Knowledge about Representations as a Basis for System Construction and Maintenance," Pattern-Directed Inference Systems, New York Academic Press, 1977, PP.99-134.
[9] Xerox Learning Research Group,"The Smalltalk-80 System," Byte 6, 8, 1981.
[10] 渡辺，"概念ネットワークを利用した知識の獲得支援，"知識工学と人工知能研究会，28-6，1982(12/8).
[11] 渡辺，"設計支援知識ベースシステムでの制約処理，"情処全大，昭58前，1983，pp967-968.
[12] Griffith,R.L.,"Three Principles of Representation for Semantic Networks," ACM TODS, 7, 3, 1982, pp.417-442.