特　集　「知的プログラミングの基礎」

自然言語理解に基づくプログラム合成

Automatic Program Synthesis based on Natural Language Understanding

豊田　順一*　上原　邦昭*
Jun-ichi Toyoda　Kuniaki Uehara

*　大阪大学産業科学研究所
The Institute of Scientific and Industrial Research, Osaka Univ.

1987年5月18日　受理

Keywords : automatic program synthesis, natural language understanding, verification, explanation, rule-based system.

1. はじめに

通常，ソフトウェアの開発において，要求仕様は自然言語で記述されることが多い．このことから，自然言語理解に基づくプログラム合成システムを開発しようという動きがある．このような研究を自然言語処理の観点から眺めると，非形式性[3]の取り扱いが問題になる．自然言語の非形式性には，曖昧性，漠然性，多様性，部分性，宣言性などがある．非形式性は，人間が自然言語によって要求仕様を記述する際には利点として働くことが多い．しかしながら，計算機によって自然言語の非形式性を取り扱うためには，要求仕様で記述されている内容を全体としてとらえるといった文脈処理のメカニズムを開発する必要がある．また，要求仕様で記述された対象について理解するための知識（Extra-Linguistic Knowledge）をどのようにしてモデル化するか，さらにモデル化された知識（世界モデル）をどのように利用するかといった知識情報処理のメカニズムを開発する必要がある．

一方，プログラム合成システムの観点から眺めると，要求仕様とプログラミング言語間でのセマンティックギャップが問題になる．たとえば，要求仕様では非形式性の利点を活かして曖昧に記述されている箇所も，望むプログラムを合成するためには，厳密にプログラミング言語に変換しなければならない．また，合成されたプログラムの正当性を検証するためのメカニズムを開発しなければならない．さらに，この検証過程を計算機に任せる場合，どのように人間と計算機のコミュニケーションを行うかといった問題がある．

以上の問題点のうち，前者については文献（16）で詳しく議論されている．本稿では，後者の問題点について，特に自然言語処理とのかかわり方に重点をおいて議論する．さらに，プログラム合成システムの今後の動向について検討する．

2. 詳細化パラダイム

プログラム合成の研究において，詳細化パラダイム（構造的アプローチ，段階的詳細化）という概念が提案されている．詳細化パラダイムでは，抽象的な概念で記述された要求仕様に変換ルールが適用され，徐々に詳細な言語要素を用いた記述へと変換される．最終結果は，目標言語で定義された言語要素による完全に具体化されたプログラムになる．

詳細化パラダイムの典型例として，スタンフォード大学で開発されたPSI[6]がある．PSIでは，ユーザが，英語や入出力例を用いて仕様獲得モジュールと対話しながらプログラムの仕様を与える．この仕様は，高級仕様記述言語に変換され，さらに，サブシステムPSI/SYN[4]（コーディングを担当するPECOSとコーディングの効率化を担当するLIBRAからなる）により，最終的なプログラミング言語に変換される．次節では，PSIで詳細化パラダイムを実現しているPECOSについて説明する．

2・1　PECOS

PECOSは，変換ルールとしてコード化されたプログラミング知識の集合（知識ベース）と，タスク指向の制御構造からなる．変換ルールは，以下の3種類のタイプに分けることができる．

①　Refinement rule：要求仕様に現れる概念をさ

らに詳細な概念に変換するためのルールである．詳細化過程で用いられるルールはほとんどがrefinement ruleである．

② Property rule：すでに詳細化の行われた部分に新たな属性を付加するためのルールである．

③ Query rule：すでに詳細化の行われた特定の部分に関する問合せに答えるためのルールである．このルールは，通常，他のルールの適用可能性を決定する過程で用いられる．

これらのルールは，タスク指向の制御構造によって起動される．詳細化の各サイクルで3種類のタスク，refinement, property, queryから，いずれかのタスクが選択され，上記のルール集合の中からそのタスクに対応する適切なルールが適用される．もし複数のルールが適用可能であれば，すべてのルールの適用が試みられる．また，あるタスクの実行中にサブタスクが生成される場合は，これらのサブタスクをアジェンダに登録し，元のタスクを再実行する前にサブタスクを実行する(last-in first-out)ようにしている．

簡単な例を用いてPECOSの詳細化過程を説明する．要求仕様として，

Is X (integer) stored in a location of Y (sequential collection of integers)?

を考える．この仕様は，内部表現として図1(a)のように表される．ただし，四角で囲まれているノードには概念が割り当てられ，リンクは概念に関する属性を示しているものとする．また，すでに述べた変換ルールのうち，refinement ruleは内部表現に新たにノードを追加すること，あるいはノードを別のノードで置き換えること，property ruleはノードにリンクを付加すること，query ruleは内部表現を調べて問合せに答えることに対応している．

上記の内部表現のうち，まずトップノード(1)が選択

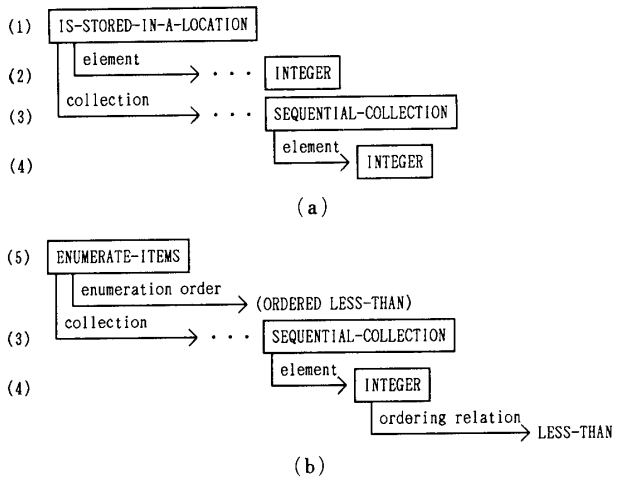


図1 内部表現の詳細化過程

され，タスク(refine 1)が実行される．このタスクを実行するために，PECOSは以下のrefinement ruleを適用しようとする．

もし順次集合体(sequential collection)の要素に順序関係が存在すれば，ある要素が集合体のどこかに存在するかどうかを判定するテストは，その集合体の中での要素の数え上げ(enumerate items)で実現できる．ただし，数え上げの順序は，この順序関係に従っており，数え上げは，目標とする要素が見つかった段階で終了するものとする．

しかしながら，上記のルールを適用するためには，順次集合体Yの要素に順序関係が存在しなければならない．このため，新たなサブタスク(property ordering-relation 4)が生成される．このタスクを達成するのは以下のproperty ruleである．

整数の順序関係は“less than”によって表される．このルールの適用により，ノードINTEGERに新たなリンクが付加される．サブタスクが達成されたので，元のタスク(refine 1)の再実行が試みられ，上のrefinement ruleが適用される．この結果，図1(b)のように内部表現が詳細化される．

2・2 詳細化パラダイムの利点と問題点

前項で述べたように，PECOSの特徴は，プログラミング知識をコード化して，ルールという明示的な表現によって記述可能であることを示した点にある．この結果得られる利点として以下の点がある．

① 各ルールを細かく記述し，詳細化過程の各ステップ間での変換を小さくしているために，ルールがモジュラーになり，設計における決定を探索木の中で明確に反映することが容易になる．

② 変換のためのステップが比較的小さく，ほぼ人間の決定の大きさに対応しているために，ユーザがプログラムの詳細化過程を理解したり，修正したりするのが容易になる．

③ 変換ルールのうち，下位レベルのルール(目標言語に依存しているルール)を他のルールと置き換えれば，目標言語の異なるプログラム合成システムが実現できる．

しかしながら，詳細化パラダイムには以下のような問題点がある[11]．

① 詳細化のレベル(1回のサイクルでどの程度まで詳細化するか)に関する基準が明確でない．したがって，どの程度の詳しさでルールを記述するのが妥当であるかについての基準が明確でない．

② 詳細化の停止条件に関する基準が明確でない．

言い換えると，最終的なプリミティブの大きさが明確でない．たとえば，プリミティブを大きく考えるとサブルーチンやライブラリほどにもなる．一方，プリミティブを小さく考えるとプログラムのオペレーション程度になる．この問題については5節で詳しく議論する．

③ 詳細化された各モジュール，あるいは適用されたルール間の相互関係が明確でない．したがって，本来互いに影響を及ぼしあっているモジュールか，他と独立したモジュールかを明確に区別することができない．

④ 詳細化の過程でプログラムの実行可能性を判定することができない．詳細化パラダイムでは，主にソフトウェア開発における静的な側面をモデル化したものであるために，要求仕様の無矛盾性を保証することはできるが，正当性を保証することはできない．この問題については次節で改めて議論する．

上記③の問題についてもう少し詳しく検討する．例として，以下の要求仕様を詳細化パラダイムに基づいてプログラムに変換する過程を考える[13]．

ファイルのデータを入力して平均値を求めよ．

詳細化の過程を図2に示す．この図は一見するとうまく詳細化されているように見えるが，データの総和を求める部分と，データの総数を求める部分が独立しており，両者の相互関係がうまく表されていない．つまり，通常のプログラミング言語で両モジュールを記述する場合は，ループ構造をひとまとめにするか，あるいは最初のループの終了時にファイルをリワインドし，2番目のループ時にファイルからの再入力が実行できるようにしておかなければならない．詳細化パラダイムでは，各モジュールごとに変換ルールを適用するために，このような相互関係をうまく処理する手段が欠落している．

3．操作的パラダイム

前節で述べた詳細化パラダイムに対して，Balzerらは操作的パラダイム[2]という概念を提案している．操作的パラダイムでは，要求仕様がシステムのシミュレーションモデルのような形式で記述される．詳細化パラダイムがシステムの静的な側面を重視したものに対して，操作的パラダイムはシステムがどのように振る舞うかを動的に記述する点に特徴がある．

操作的パラダイムのもう1つの特徴は，操作的(operational)という語からもわかるように，仕様自身の実行可能性(runnable)という点にある．要求仕様が実行可能であれば，コーディング以前にプロトタイプを作成することができ，仕様レベルで正当性の確認を行うことができる．また，コーディング段階ではプログラムの高速性などのチューニングのみに専念できるという利点がある．

操作的パラダイムの典型例として，南カリフォルニア大学でのGISTプロジェクト[14]がある．GISTは超高級仕様記述言語で，ユーザがどのような(what)振舞いを望んでいるかのみを記述し，どのように(how)その振舞いが計算機上で達成されるかについては記述しなくてもよいようにしている．ユーザがライブラリ(library of transformations)にある適切な変換を選択すれば，後は自動的にシステムがその変換を仕様に適用し，プログラミング言語に翻訳することができる．

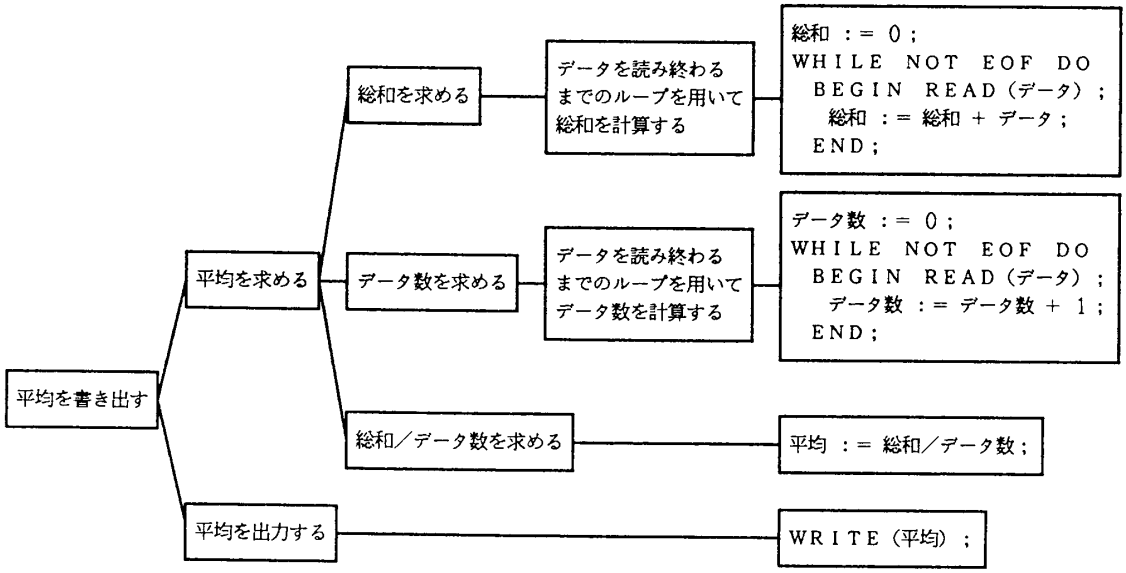


図2 プランの詳細化過程

3・1　自然言語と形式言語の相反性

すでに述べたように，自然言語による仕様の利点は，詳細化パラダイムのように，モジュラリティを意識した静的記述（宣言性）を行う際に際立っている．さらに，自然言語の部分性という特徴を加えると，人間が言葉を使って物事を考える以上，要求仕様を自然言語によって記述することは極めて自然である．しかしながら，システムの動的な側面を自然言語によって記述する場合，その冗長性が問題となる．たとえば，自然言語で，

Load register 5 with contents of memory location 190.

と書くよりも，形式的に，

load : register 5 ←(memory location 190)

と書くほうが簡潔でわかりやすいものになる．GISTも同様の理由で，自然言語の仕様記述能力を持った形式言語という観点から開発されている．

逆に形式言語の問題点として，以下の点が挙げられる．

① 形式言語による仕様は，量が多くなれば読みづらくなり，たとえそれを書いたユーザ自身でも，内容を取り違えて理解することがある．

② 形式言語が高級になるに従って，通常のプログラミング言語では取り扱わないような抽象的データ構造や暗黙の仮定を持っていることが多い．したがって，プログラミングが詳しいユーザにも，形式言語の持つ概念やsyntaxについて新たに教育しなければならない．

③ 仕様が形式言語によって操作的に記述されているために，実行の流れを考慮しながら理解しなければ，仕様中での非局所的な相互作用を把握できない．すなわち，仕様全体を見通して理解することが困難である．

①，②の問題を解決し，形式言語を理解しやすいものとするためには，不慣れな記述形式を自然言語に翻訳して仕様の静的な側面を表す必要がある．このような形式言語から自然言語への変換機能があれば，仕様に新たな視点による読み方を与えることができ，デバッギングの際に有効になる．一方，③の問題を解決するためには，仕様を疑似的に実行し，仕様中に直接示された事実だけでなく，そこから引き出される相互関係といった仕様の動的な側面を表す機能が必要である．

3・2　GISTの枠組み

GISTで記述された仕様が自然言語に変換される過程を図3に示す．この図で仕様の静的な側面は左側の流れで，動的な側面は右側の流れで自然言語に変換される．GIST English Generatorでは，中間表現として格フレーム構造を用いて，以下の3段階のステップにより仕様から自然言語への翻訳を行っている．

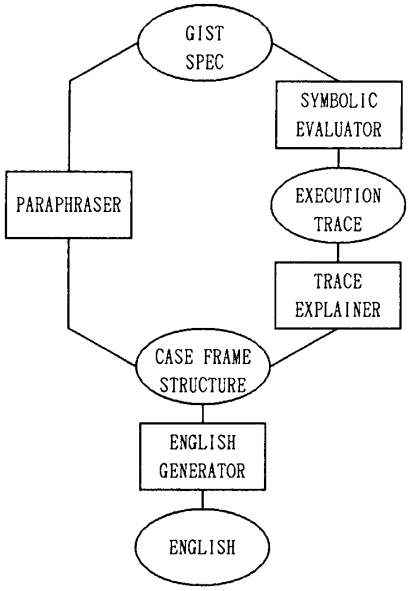


図 3　システム構成図

① GISTの仕様から，Paraphraserによって生成すべき英文の格フレーム構造を抽出する．

② 生成する英文の質を向上させるために，格フレーム構造の相互文変形を行う．たとえば，続く2文間で同一の主語が用いられている場合は，主語の冗長性を取り除くために，単文から重文への変形が行われる．

③ 変形された格フレーム構造から実際の英文を生成する．ここでは，英語特有の言い回しも考慮して，文内部の最適化も同時に行われる．

GISTによる仕様は，タイプ宣言部，アクション部，アクションテスト部からなる．タイプ宣言部では，仕様中での変数の型宣言，属性関係などが示される．これらの中には，①前向き写像（forward mapping）や後向き写像（back mapping）を示すための属性関係，②part-of関係を示すための属性関係，③動詞として扱われるような属性関係がある．

アクション部にはプロセスの流れが記述される．アクションは動詞に対応しており，英語の動詞あるいは複合動詞で表される．アクションの各引数は格文法における格に対応している．アクションテスト部には，アクション部をSymbolic Evaluatorによって疑似実行する際のデータや前提条件などが記述される．図4に，GISTによる仕様のアクション部とその自然言語表現を示す．

Trace Explainerは，Symbolic Evaluatorによって疑似実行された実行トレースから格フレーム構造を

```
begin
                    .
                    .
   agent PackageRouter() where
        action Insert[box]
          definition update :Location of box
                       from input1 to Source1;
        action Set[switch]
          percondition ~$ :Location=switch
          definition update :Selected-outlet
                 of switch to switch :Outlet;
        action Move[box]
          precondition box :Location=Source1 or
                       box :Location=a switch
          definition
              if box :Location=Source1
                 then update :Location of box
                     to Source1 :Source-outlet
                 else update :Location of box
                     to box :Location :Selected-outlet;
                     .
                     .
   end
end
```

```
A package-router can insert a box, set a switch, or
move a box.
   To insert a box:
      Action: The box's location is updated from
      input1 to source1.
   To set a switch:
      Action: The switch's selected-outlet is updated
      to an outlet of the switch.
      Preconditions:
         The switch must not be the location of any
         box.
   To move a box:
      Action:
         If: The box's location is source1,
            Then: The box's location is updated to the
            source-outlet of source1.
            Else: The box's location is updated to the
            selected-outlet of the switch that is the
            box's location.
      Preconditions:
         Either:
         1. The box's location must be source1, or
         2. The box's location must be a switch.
```

図 4　GIST による仕様の一部とその自然言語表現

```
1. A box, call it box1, is inserted.
Result: The new location of box1 is source1.

2. A box is moved.  The box must be box1 since
   2.1 For all boxes except box1, the box's location
   is input1, and
   2.2 The precondition of moving a box requires that
   either:
      2.2.1 The box's location must be source1, or
      2.2.2 The box's location must be a switch.
Result: The new location of box1 is switch1.
```

図 5　Trace Explainer による実行トレースの出力

生成するものである．Trace Explainer には 2 種類の説明方法が備わっている．1 つはトレースに基づく方法 (trace-based method) で，もう 1 つは構造化による方法 (structuring method) である．trace-based method は，トレース中に生じた特定の状況（アクションの起動や事実の正当化）についての説明を行うものである．また，structuring method は，trace-based method から得られた出力を再構成し，より高度な説明にするものである．たとえば，アクション P と Q の間に因果関係がある場合，単純に “P and Q” と並べず，“Q since P” というように構造化された文を生成する．Trace Explainer による出力例を図 5 に示す．

4．プログラムの検証と自然言語処理

従来，プログラム検証 (verification)[1] の分野では，要求仕様とその仕様から合成されたプログラムを与え，定理証明手法を用いてプログラムの正当性を検証しようとする研究が行われてきた．しかしながら，定理証明によるプログラムの検証には困難な問題が多く，実験的なシステムしか開発されていなかった．このような状況に対して，知識工学，特にエキスパートシステムの開発によって得られた知見を援用したプログラム合成システムの研究がある．

このアプローチでは，システムがプログラム合成過程をユーザに説明することによって，合成されたプログラムの正当性を示そうというものである．当然のことながら，このようなアプローチでは，プログラム合成で用いられた知識やシステムの動作についてもユーザがある程度熟知している必要があるが，定理証明によるアプローチと比較して実現の可能性が高いという点で，現在では最良のアプローチだと考えられる．

合成されたプログラムの正当性を示すためには，

① なぜそのルールが適用されたかを示すこと

② 単に適用されたルールを示すだけでなく，その基本原理についても同時に述べ，ユーザにとってわかりやすい説明をすること

③ ユーザの理解レベルに応じた説明をすること

が必要である．本節では，これらの機能を実現したシステムについて説明する．

4・1　ルール提示による説明

〔1〕 TEIRESIAS

TEIRESIAS[5] は，スタンフォード大学の MYCIN プロジェクトの一部として開発された説明・知識獲得支援システムである．TEIRESIAS の説明機能は，MYCIN の 2 大特徴，①ルールに基づく知識表現，②後向き連鎖による単純な推論メカニズム，に依存したものとなっている．

TEIRESIAS では，単純な AND/OR 木を用いた制御構造に基づいて推論が行われ，この枠組みに従っ

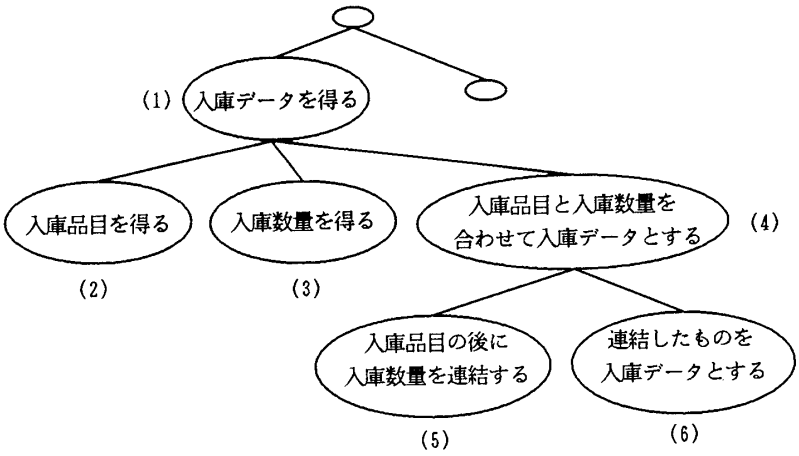


図 6 プログラムの詳細化過程

てHowとWhyという説明用コマンドが実現されている．Howは「どのようにして～に関する推論を行うのか」という質問に相当するコマンドで，説明のためにはAND/OR木を下位レベルにたどることになる．また，Whyは「なぜ～のような質問をするのか」という質問に相当し，AND/OR木を上位レベルにたどることになる．

TEIRESIASの枠組みを用いてプログラム合成過程を説明する例を以下に示す．システムが，プログラムの合成過程において，図6に示す探索木を生成したとする．合成されたプログラムの実行中に，システムからユーザに，図中のノード(2)に関する質問が発せられた状況を仮定する．すなわち，「入庫品目を入力してください」とシステムがユーザに要求している場面である．このとき，ユーザが“Why”と入力すると，システムは次のような説明を行うことができる．

「入庫データを得る」ためです．
「入庫データを得る」ためには，
「入庫品目を得る」，
「入庫数量を得る」，
「入庫品目と入庫数量を合わせて入庫データとする」

を実行すればよいからです．

また，“How (4)”と入力すると，システムは以下のような説明を行うことができる．

「入庫品目と入庫数量を合わせて入庫データとする」ためには，
「入庫品目の後に入庫数量を連結する」，
「連結したものを入庫データとする」

を実行します．

上の例で示したように，TEIRESIASのアプローチをプログラム合成に適用した場合，推論過程で適用されたルールしか明示されず，プログラム合成のように詳細化のレベルが段階的に変化していくような状況では，適切な説明になっていない場合がある．また，推論過程で必要になる操作手順（ワーキングメモリへのファクトの登録，消去など）は，それぞれ対応するルール中に書き込んでおかなければならないが，これらは人間が推論を行う場合，直接関係のないものである．したがって，TEIRESIASのように単純に推論過程を再現するのみでは，これらの操作手順も同時に提示されてしまい，かえってユーザにわかりにくい説明となる可能性がある．

逆に，ルールを記述するために着目した知識が推論過程と関係ないために，ルールから削除されているという場合がある．これは，ルールには単純に表層上の因果関係しか記述せず，ルールを記述するに至った問題領域に関する基本原理（プログラミング原理，データ構造に依存する詳細な制約条件など）は，システムを動作させるためには必要ない，という理由でルールには明記されていないことによる．したがって，ユーザが問題領域の基本原理について理解していない場合は，妥当な説明になっていないことがある．

このような状況においても，ユーザにわかりやすい説明を行うためには，システムが現在どのような推論を行っているかを提示するだけでなく，そのルールを適用した根拠，あるいは問題領域を理解するための基本原理を示す必要がある．このようにすれば，なぜそのルールがその状況で適用されるか（ルールの正当性）を説明することができる．以上のような考え方に基づいて開発されたシステムにXPLAIN[15]がある．

4・2 基本原理による説明

〔1〕 XPLAIN

XPLAINは，MITの臨床意思決定研究グループによって開発された，ジギタリスの投薬に関して医師に助言を与えるシステム，Digitalis Therapy Advisor

の説明システムである．なお，Digitalis Therapy Advisor の対象とする問題領域は医学であるが，合成される手続きが if-then 型のプログラムであるために，本稿では一種のプログラム合成システムと考える．

XPLAIN には，問題領域に関する基本原理として，ドメインモデル (Domain Model) とドメイン原理 (Domain Principle) が用意されている．ドメインモデルは，ジギタリス療法に関するさまざまな因果関係をネットワーク表現したものである．ドメイン原理は，ジギタリス療法に関する手続き的知識を記述したものである．ジギタリス療法決定の手続きは，ドメインモデルにドメイン原理を適用し，抽象的なゴールを詳細化することによって合成される．この手続きを合成する過程を記録したものが詳細化構造 (Refinement Structure，詳細化パラダイムにおける探索木と等しい概念) である．

ドメイン原理は，ゴール (Goal)，ドメイン根拠 (Domain Rationale)，プロトタイプ (Prototype Method) から構成されている．ゴールは，詳細化構造のノードとマッチングをとるために利用される．ドメイン根拠は，ドメイン原理が，ドメインモデルとマッチングするためのテンプレートとして用いられる枠組みである．このマッチングの結果，プロトタイプ中の変数に具体的な値が代入され，さらに具体化されたプロトタイプによって詳細化構造のノードが詳細化される．

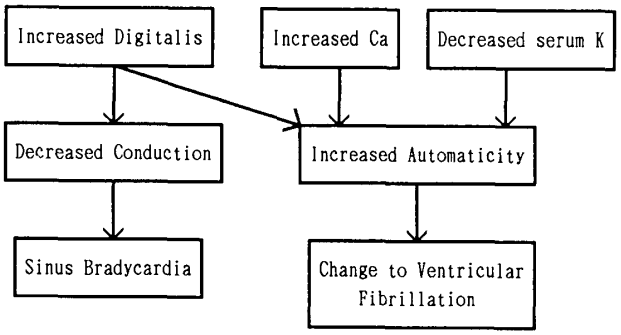
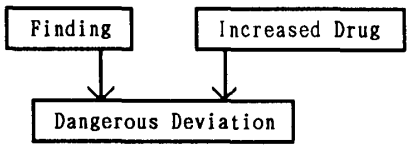


図 7　ドメインモデル

```
Goal: Anticipate Drug Toxicity

Domain Rationale:
```




```
Prototype Method:
    If the Finding exists
    then: reduce the Drug dose
    else: maintain the Drug dose
```

図 8　ドメイン原理

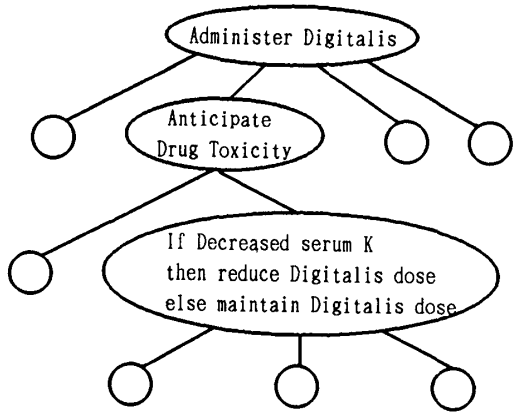


図 9　詳細化構造

XPLAIN の手続きの詳細化過程を具体的に例を用いて説明する．ドメインモデルとして図7，ドメイン原理として図8，詳細化構造として図9を考える．詳細化構造のノード“Anticipate Drug Toxicity”は，ドメイン原理のゴールとマッチングすることができる．さらに，ドメイン根拠とドメインモデルのマッチングが行われ，ドメイン原理の変数“Finding”には“Decreased serum K”あるいは“Increased Ca”のいずれか，“Drug”には“Digitalis”，“Dangerous Deviation”には“Change to Ventricular Fibrillation”が代入される．この結果，詳細化構造のノードはさらに詳細化される．

以上のような手続きの詳細化過程を保存しておけば，ドメイン原理を活用して，合成された手続きの正当性を説明することができる．たとえば，ジギタリス療法の手続きを実行中に，システムが，

Please enter the value of serum K.

という質問を出した状況で，ユーザが“Why”と入力した場合，システムは，

The system is anticipating digitalis toxicity. Decreased serum K causes increased automaticity, which may cause a change to ventricular fibrillation. Increased digitalis also causes increased automaticity. Thus, if the system observes decreased serum K, it reduces the dose of digitalis due to decreased serum K.

と答えることができる．このうち，下線部はドメインモデルから得られる説明，その他の部分は詳細化構造から得られる説明である．

〔2〕 EES

XPLAIN の説明機能は，Swartout らによってさらに拡張され，EES (Explainable Expert System)[10] というシステムが開発されている．EES は，問題領域として Lisp プログラムの効率化を支援するエキス

パートシステムを想定している．EES の改良点には以下の3点がある．

① XPLAIN では専門用語の定義が明確に与えられていなかった．したがって，前述の例のように，“Dose”（投薬）と“Digitalis”（ジギタリス），あるいは“Dangerous Deviation”（中毒症状の兆候）と“Ventricular Fibrillation”（心室細動）といった，異なる概念間でのマッチングメカニズムが明確に示されていなかった．EES では，新たに NIKL と呼ぶ表現形式を開発し，専門用語の定義，概念間の階層性，問題解決知識の階層性が記述できるようにしており，この問題に対処している．

② XPLAIN では，Why や What についてのみ質問に答えるようになっていたが，EES では NIKL で記述された定義を参照することによって，専門用語の意味についても答えることができるようになっている．また，質問の型に応じてそれぞれの戦略を用意しており，取り扱うことが可能な質問の種類を拡充している．

③ XPLAIN では，詳細化構造のノードと直接マッチング可能なドメイン原理がなければ，それ以上詳細化することができなかった．このような場合，EES では問題解決知識を NIKL によって階層的に表現できるために，上位概念の問題解決知識を用いて詳細化構造を再構成し，新たな詳細化を再実行できるようになっている．

4・3 ユーザモデルの導入

プログラム合成システムは，いろいろなユーザに使用される可能性がある．したがって，あらかじめユーザの知識レベルを想定することは不可能であるが，何らかの形でユーザのレベルに応じた応答をすることが望ましい．このような能力を実現したプログラム合成システムは，まだ開発されていないが，UNIX のコマンドに関して説明する UC[17] や，われわれの開発したシステム ASSIST[9] などで提案されたユーザモデルの概念を利用することが考えられる．

〔1〕 UC

UC のユーザモデルでは，UNIX に関するユーザの知識量に応じて4種類のカテゴリを用意している．これはユーザとの対話中に，novice, beginner, intermediate, expert とユーザの熟練度を分類するものである．さらに，UC のコマンド，コマンド形式，用語，その他の関連する知識などを，概念の複雑さ，難解さに応じて4種類に分類している．以下にこの分類例を示す．

〈難易度〉	〈UNIX のコマンド〉
simple	rm, ls, cat, ...
mundane	vi, diff, spell, ls−1, ...
complex	grep, chmod, tset, ...
esoteric	rdist, gremlin, make, ...

UC は，ユーザの問合せから何を知っており，何を知らないかを推論し，上記の分類に従って，ユーザの属するカテゴリを決定する．たとえば，ユーザが simple に属するコマンドを知らない場合，ユーザは expert でも intermediate でもないと推論する．また，ユーザが complex に属するコマンドを知っている場合，ユーザは novice でも beginner でもなく，intermediate である可能性は低く，expert である可能性が高いと推論する．UC では，対話過程でこのような推論を常に実行し，ユーザの知識状態を判断しているために，以下のような対話が可能になっている．

① ユーザの知っていることを省略した説明

② ユーザに的確な説明を与えるための問い返し

③ すでにユーザが知っている事柄との対比による説明

④ ユーザの誤解の指摘

UC のユーザモデルは，ユーザを単にクラス分けしたものに過ぎず，ユーザとの対話を通じて個別的なユーザモデルが動的に構成されていくようにはなっていない．また，UC の説明文はあらかじめカテゴリに応じて用意されたもので，同一レベルのユーザが何度同じ質問を繰り返しても，常に同じ説明しか行うことができないという欠点がある．

以上のような観点から，われわれは動的なユーザモデルとプランニングの手法を用いて，ユーザの理解度に応じた説明文を生成するシステム ASSIST を開発している．

〔2〕 ASSIST

ASSIST では，「対話を介してユーザに説明した内容は，ユーザがすでに知っている」という仮定に基づき，ユーザモデルの一部として既知情報を記録したファクトを利用している．たとえば，ASSIST が新たに何らかの概念をユーザに説明した場合，そのことを表すファクトがユーザモデルに書き込まれる．このようなファクトは，システムとの対話を重ねながら次第に蓄えられてゆき，ユーザの既知情報を考慮して説明内容を決定する際に用いられる．また，システムが信じているユーザの知識状態を表すために推論ルールを導入している．推論ルールは，ユーザが持っていると考えられる知識間での依存関係を表したものである．さらに，ファクト集合と推論ルールを用いて，ユーザ

の既知概念を推論するための推論プログラムを導入している．推論プログラムは，ユーザにとって既知であるかどうかを決定する手順を示したものである．

ユーザの知識の欠落や誤りを検出するためには，さらにユーザモデルの診断プログラムが必要である．このような診断を行うために，Shapiroの提案した，Prologプログラム中のバグを発見するための診断モジュールPDS (Program Diagnosis System) [12]を利用している．PDSは以下の3種類のバグを発見することができる．

(1) incompleteness：プログラム中の述語やファクトの欠如
(2) incorrectness：プログラム中の述語やファクトの誤り
(3) nontermination：プログラムの非停止性

診断されるPrologプログラムをユーザの持つ不完全な知識とみなすと，(1)は「知識の欠落」，(2)は「知識の誤り」，(3)は「知識の適用に関する誤り」と対応づけることができる．

このような考え方に基づき，ASSISTではユーザモデルをPrologプログラムとして表現し，PDSを用いてユーザの「知識の欠落」および「知識の誤り」を検出するという方法を採用している．たとえば「知識の欠落」，すなわちユーザがある知識を持っていないということは，ユーザモデルのファクト集合に欠落があるということになる．したがって，PDSによってファクトの欠落が検出できれば，ユーザに対応する概念を詳しく説明すればよいことになる．また，ファクトが存在する場合は，ユーザがその概念を知っていると仮定して，簡単に説明するようになっている．

5．プログラム部品を用いたプログラム合成

最近，プログラム部品を用いたプログラム合成の研究が盛んに行われている．この手法は，一連の操作を記述したプログラムモジュールなどを，あらかじめ知識ベース内にプログラム部品として蓄えておき，要求が与えられたときに適当な部品を選択し，それらを組み合わせてプログラムとするものである．この手法を用いれば，部品として実現されている部分には詳細化する必要がないために，PECOSなどの手法に比べて準備しなければならない詳細化ルールの数が大幅に減少するという利点がある．さらに，自然言語仕様で記述しにくいプログラムの細部については，あらかじめ部品として知識ベースに蓄えられるために，要求仕様中に記述する必要がなくなるという利点がある．

5・1 ARIES/I

プログラム部品からプログラムを合成するシステムに電力中央研究所のARIES/I [7]がある．ARIES/Iは，企業の業務処理を対象としており，要求は特定の「言い回し」によって与えられる．たとえば，要求仕様は「…に対して…をレポートせよ」というように特定の処理を表す動詞と助詞から構成されている．要求中にどのように「レポート」すればよいのかについての，具体的な記述がなされていないのは，「レポート」する方法に関連したプログラム部品が，ARIES/Iの知識としてすでに用意されているためである．

ARIES/Iの手法を企業の業務処理以外の分野に適用しようとすると，以下の2点が問題となる．

① 特定の「言い回し」だけで要求を記述しなければならない点
② 要求を満足する適当な部品が見つからなければ，システムによるプログラムの合成が不可能になる点

①の問題点は，言葉や言い回しに関する知識が，システムとユーザの間で統一されていないことが原因となっている．したがって，この問題を解決するためには，ユーザの知識をシステム内にモデル化するか，あるいはユーザが持っている知識をシステムが学習するなどの技術が必要となる．また②の問題点は，プログラム部品を単純に組み合わせてプログラム合成を行っていることが原因となっている．

人間がプログラムを作成する場合は，作成しようとするプログラムが難しければ，まず近似解となるプログラムを作成し，そのプログラムを修正するという手段を用いている．自動プログラム合成システムにおいても，要求を満たすプログラムが合成できない場合は近似解を与えるなどして，利用者のプログラム作成を援助する機能を実現すれば，②の問題点は解決できると考えられる．以上のような観点にたって，われわれは，APSS (Automatic Program Synthesis System) [8]を開発している．

5・2 APSS

APSSでは，一度合成したプログラムに新しい言葉を対応づけることができる．たとえば，APSSが合成した「データの合計をデータ数で割る」プログラムに「平均」という言葉を対応づければ，仕様中に現れる「試験の点数の合計を試験を受けた人数で割ったもの」を簡単に「試験の平均点」と記述することができ，先のプログラムが「平均」に対応するプログラム部品として再利用される．

さらに，APSSは，要求を満たす適当なプログラム部品が検索できなかった場合でも，要求と類似した仕様を満たすプログラム部品を選択する機能を有しており，要求しているプログラムの近似解を利用者に示すことができる．この機能は，プログラム部品の仕様と要求仕様間に類似性を定義し，類推の概念を用いてプログラム部品を選択することで実現している．一方，類推の概念を用いて選択された部品には，そのまま利用することができないものがあり，選択された部品の修正が必要である．APSSは，合成したプログラムが近似解である場合には，部品選択時に用いた類似性について説明し，利用者の手による修正を支援するようにしている．

6．ま　と　め

本稿では，プログラム合成において，人間と計算機がどのようにコミュニケーションを行うかという観点から，現状およびその問題点を明確にした．ここで紹介したいくつかの手法の中には，直接プログラム合成と関係のないものもあるが，プログラムの合成が人間の高度な知的生産である以上，単にソフトウェア工学のみならず，自然言語処理，認知科学，知識工学など他分野の研究との融合が，今後ともますます必要となるであろう．

◇参　考　文　献◇

(1) 阿草清滋：要求仕様記述とその検証(bit, 大野豊編), 新しい時代のソフトウェア, Vol. 16, No. 6, pp. 27-33 (1984).
(2) 有沢　誠：ソフトウェアプロトタイピング，ソフトウェア工学ライブラリ, 16, 近代科学社 (1986).
(3) Balzer, R., Goldman, N. and Wile, D. : Informality in program specifications, IEEE Trans. on Softw. Eng., Vol. SE-4, No. 2, pp. 94-103 (1978).
(4) Barstow, D. R. : An experiment in knowledge-based automatic programming, Artif. Intell., Vol. 12, No. 1, pp. 73-119 (1979).
(5) Davis, R. and Lenat, D. B. : Knowledge-Based Systems in Artificial Intelligence, McGraw-Hill (1982).
(6) Green, C., Richard, P. G. and Kant, E. : Results in knowledge-based program synthesis, Proc. of IJCAI-6, pp. 342-344 (1979).
(7) 原田　実，篠原靖志：部品合成によるプログラム自動合成システム ARIES/I, 情報処理学会論文誌, Vol. 27, No. 4, pp. 417-424 (1986).
(8) 今中　武，上原邦昭，豊田順一：類似したプログラムの再利用による自動プログラム合成，Proc. of the Logic Programming Conf. '87, 3-2 (1987).
(9) 槇本英治，垣内隆志，上原邦昭，豊田順一：対話型システムにおける文脈情報を利用した文章生成について，情報処理学会自然言語処理研究会資料，59-8 (1987).
(10) Neches, R., Swartout, W. R. and Moore, J. D. : Enhanced maintenance and explanation of expert systems through explicit models of their development, IEEE Trans. on Softw. Eng., Vol. SE-11, No. 11, pp. 1337-1351 (1985).
(11) 野木兼六：要求定義技術の最新の動向，情報処理学会誌, Vol. 27, No. 1, pp. 21-30 (1986).
(12) Shapiro, E. Y. : Algorithmic Program Debugging, ACM Distinguished Dissertations, The MIT Press (1982).
(13) Soloway, E. : Learning to program=learning to construct mechanisms and explanations, Comm. of ACM, Vol. 29, No. 9, pp. 850-858 (1986).
(14) Swartout, B. : The GIST behavior explainer, Proc. of AAAI-83, pp. 402-407 (1983).
(15) Swartout, W. R. : XPLAIN : A system for creating and explaining expert consulting systems, Artif. Intell., Vol. 21, No. 3, pp. 285-325 (1983).
(16) 辻井潤一，上原邦昭：ソフトウェア工学と自然言語処理，情報処理学会誌, Vol. 28, No. 7 (掲載予定).
(17) Wilensky, R., Arens, Y. and Chin, D. : Talking to UNIX in English : An overview of UC, Comm. of ACM, Vol. 27, No. 6, pp. 574-593 (1984).

著　者　紹　介

豊田　順一（正会員）

昭和36年大阪大学工学部通信工学科卒業．昭和41年同大学院博士後期課程単位取得退学．同年大阪大学基礎工学部助手．昭和44年助教授．昭和57年大阪大学産業科学研究所教授．工学博士．現在，主として，自然言語理解，画像理解，文書画像処理，およびICAIシステム等の研究に従事している．電子情報通信学会，日本認知科学会，情報処理学会各会員．

上原　邦昭（正会員）

昭和53年大阪大学基礎工学部情報工学科卒業．昭和58年同大学院基礎工学研究科博士後期課程退学．同年，大阪大学産業科学研究所勤務．現在，同研究所助手．工学博士．現在，人工知能，特に自然言語理解，および自動プログラム合成の研究に従事している．ACM，電子情報通信学会，計量国語学会，情報処理学会，日本ソフトウェア科学会各会員．