解　説

ソフトウェア工学と自然言語処理†

辻井潤一††　上原邦昭†††

1. はじめに

自然言語による仕様記述の不完全さ，非形式性への反省から，これまでにも，多くの形式的な仕様記述に関する提案が行われ，その一部は，すでに現実のソフトウェアの生産現場で使用されはじめている．しかしながら，ソフトウェア開発の現場では，現在なお，自然言語による記述がソフトウェア発注者と作成者，あるいは，作成者間の意志伝達の基本的手段として使われているのが現実である．実際，自然言語は，不正確な情報の伝達という欠点をもつ反面，記述の簡明さ，人間にとっての直観的な理解容易性といったほかの表現手段では代用できない利点ももっている．

一方，人工知能や知識情報処理の一環としての自然言語理解の研究は，「言語と理解」,「言語と知識」をめぐっての研究に重点を置き，かなりの進歩と技術の蓄積を得てきている．このような自然言語理解研究の成果を活用して，現在ソフトウェアの生産現場に氾濫する自然言語ドキュメントを計算機管理したり，あるいは，自然言語による非形式的な仕様記述を形式的な仕様記述に変換したりできないかと考えることは，ごく自然なことである．

本稿では，**1.** と **2.** で自然言語をユーザ・インタフェースに適用する場合の一般的な問題を述べ，**3.** 以降で，自然言語を仕様記述に使った事例的な研究の紹介，およびその技術的な問題点について論じる．

2. 自然言語の非形式性

仕様記述に自然言語を使う枠組みは，データベースアクセスを自然言語で行うといった自然言語をユーザフロントエンドとして使う場合の一般的な枠組みで考えることができる．**図-1** は，このようなユーザフロントエンドとしての自然言語処理の一般的な枠組みを示している．ここでは，単語列（あるいは文字列）としての文（文章）を，応用システムによって意味が定められる形式的記述に変換することが自然言語処理の役割となる．

図-1の内部表現は，自然言語によるデータベースアクセスの場合には SQL のようなデータベースアクセス用の人工言語となるし，自然言語による仕様記述の場合には，仕様記述用の形式言語表現となるか，あるいは，実際のプロトタイプ的なプログラムとなる．

自然言語の記述が基本的に WHAT 型の記述であるのに対して，ソフトウェア開発の最終プロダクトとしてのプログラムはこの WHAT を実現するための HOW 型の記述となる．この二つの記述レベル（自然言語とプログラム）の差は，データベースアクセスなどのほかの自然言語応用システムよりもはるかに大きい．

したがって，自然言語による仕様記述からプログラムに至るまでの処理は，概念的には，

（1） 自然言語処理：自然言語の記述を形式的記述-1 に変換する処理．

（2） プログラム合成的な処理：WHAT 型の形式的記述-1 を HOW 型のプログラムへ変換する処理

の二段階に区分して考えることができる(**図-2**)．

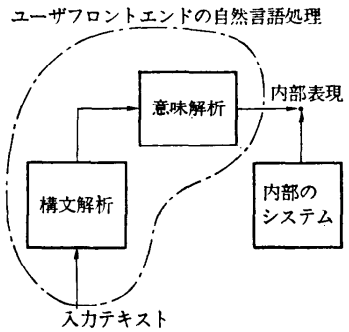

図-1 自然言語処理の枠組

† Software Engineering and Natural Language Processing by Jun-ichi TSUJII (Department of Electrical Engineering, Kyoto University) and Kuniaki UEHARA (The Institute of Scientific and Industrial Research, Osaka University).

†† 京都大学工学部電気工学第二教室
††† 大阪大学産業科学研究所

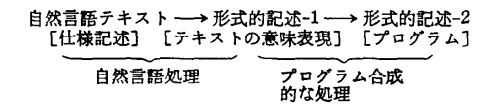

図-2 自然言語による仕様記述

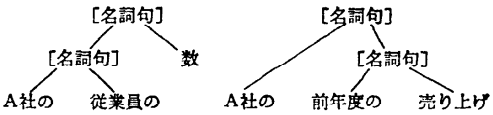

図-3 自然言語のあいまいさ

実際には，形式的記述-1 をどのようなレベルに設定するかで，二つの処理の役割分担が変わる．また，(2)のプログラム合成的な処理にも何段階かの形式変換が必要であり，形式-1から最終のプログラム形式までの全段階が，初期の自然言語による記述の「意味」を明確化する過程であると考える立場もある．この立場からは，上記の二つの段階はあくまで便宜的なものであり，すべての形式変換に，仕様記述者の記述意図を明確化する自然言語処理的な側面があることになる．このような種々の立場は，次章以降で議論するとして，本章では，単語（文字）列としての自然言語を形式的な記述に変換する際に生じる一般的な問題をまず整理しておく．

（1） 曖昧さ (Ambiguity)：自然言語の問題点として最も多く指摘されることであるが，一つの文に対して一般に複数個の意味記述が対応し得る．たとえば，「A社の従業員の数」と「A社の前年度の売り上げ」が，「名詞の名詞の名詞」という同じ品詞の並びであるにもかかわらず，図-3のような異なった句構造をもつ，といった構造的な曖昧さがその代表であるが，仕様記述という立場からは，普通はあまり気が付かない解釈上の曖昧さも多くある．たとえば，

(例) すべての項目に対してファイルを設けて，…

では，「ファイルを」が量限定詞 (quantifier)「すべての」のスコープの中にあると解釈するか，外にあると解釈するかで，すべての項目ごとに別個のファイルを用意するのか，すべての項目に共通の一個のファイルを用意すべきかが変化する．

（2） 漠然性 (Vagueness)：日本語では，これもまた「曖昧さ」と訳されることがあるために，(1)の ambiguity と混同されやすいが，技術的に別に考えるべきものである．この漠然性は，自然言語が本来「正確に」物事を表現できないことに由来する．たとえば，

(例) できるだけ速く応答を返す

の「できるだけ」がどのような時間範囲を表現しているかについては，無限の可能性が考えられる．

（3） 多様性 (Diversity)：同じことを表現したり同じものを指示するのに，非常に多様な表現が使われる．たとえば，

(例) 各顧客ごとにファイルを用意し，この顧客表に……．また，顧客ファイルには……．

では，同じ対象を記述するのに表面上は異なった表現が使われている．

（4） 部分性 (Partiality)：自然言語によるコミュニケーションでは，相互に自明な事柄は表現されない．すなわち，自然言語のテキストは，相手が自分と同じ「知識」をもつことを前提に記述されている．たとえば，

(例) スタックにデータを入れて，……

では，スタックが LIFO の記憶場所であるとの了解のもとでの記述なのかどうか，また，読み手がその知識をもっているかどうかで受け取られる「意味」が変わる．このような誤解をさけるために，スタックとは何かを定義すればよいが，この定義を自然言語で行うことは記述を複雑にするだけである．自然言語による記述は，一般に，非常に多くの暗黙の前提を含んでいる．

（5） 宣言性 (Declarativeness)：前述のように，自然言語の記述は WHAT を記述する．手続き的に記述している場合でも，特定の処理機構を想定しているわけではないので，必ずしも，記述の順序が処理の順序には対応しない．

以上のような現象は，これまで多かれ少なかれ，形式的な記述と比較した場合の自然言語の欠点としてあげられてきた．しかし，逆に，自然言語がわれわれ人間にとって理解が容易であるのは，自然言語がこのような性質をもっているからでもある．上記のような不完全性を排除した自然言語としては，法律や契約書での言語があるが，これらは必ずしも「理解が容易な」言語ではない．

自然言語を仕様記述などのソフトウェア作成過程に組み込むためには，自然言語のもつ上記の性質を残しつつ，形式的記述へと変換していく機能をシステムにもたせる必要がある．たとえば，自然言語のある種の漠然性は，一定の形式的な記述レベルを設定することから生じる問題であり，「できるだけ速く応答を返す」を，「1 msec 以内に応答を返す」といったより漠然性

の少ない表現に変えることは，必ずしも，望ましいとは言えない．むしろ，前者の表現も，仕様記述者の意図を記述するより抽象的なレベルでの記述と考え，全体的なシステムの管理対象と考えるほうが自然である．また，自然言語の部分性も，記述の中で重要なことは何かを自然な形で伝える利点がある．

以上のような自然言語の特質は，処理の面からみると，次の二つの要素をどのようにシステム的に取り扱うかの問題となる．

(A) 文脈依存性 (Context Dependency)：自然言語のテキストは全体として一つの記述を与える．用語の定義が，その用語の現れた後で出現したり，処理の手順が前後して記述されることも多い．自然言語のテキストが記述する内容を全体として捉えて，そこから必要な情報を抽出し，次の段階の形式的記述に変換する必要がある．また，自然言語の文は，省略や各種の代名詞や指示表現を多く含み，一文単位では処理ができないことも多い．

(B) 知識依存性 (Knowledge Dependency)：曖昧さを解消して，解釈を一意に決定する場合，あるいは，漠然とした表現からより精密な表現へと変換する場合に，自然言語のテキストが記述する対象についての知識は不可欠である．また，自然言語の部分性に対処するためには，システム側が対象分野についての知識をもつことが不可欠である．

もちろん，上記(A)(B)の問題を完全に解決することは現時点で不可能であるし，また，望ましいことでもない．むしろ，システムと人間が共同して，初期の不完全な仕様記述からより厳密な仕様記述を作り上げてゆく過程，あるいは，仕様記述からプログラムへと変換してゆく過程の中で，両者の自然言語によるコミュニケーションをどう位置付けるかが重要である．この応答の過程で，人間側の意図を確認しながら，自然言語のもつ曖昧さや漠然性を解消してゆくことが重要である．

3. 要求定義技術と自然言語処理

通常，プログラムの開発は段階的に行われる．まず，ユーザがどのようなプログラム (What) を開発して欲しいかという要求仕様を記述する．要求仕様には，ユーザの要求，知識，実世界の事実といった抽象的レベルの高い概念が記述される．次に，要求仕様を分析（要求分析）し，そのプログラムのもつ機能 (How) を機能仕様としてまとめ上げる．さらに機能仕様に基づいて，そのプログラムをどのように実現するかを決定（システム設計）し，外部設計仕様を作成する．以降は，段階的詳細化によってモジュール分割，コーディングへと続いていく．

このようなプログラム開発過程において，要求定義技術と自然言語処理の接点は，ユーザからの要求仕様の獲得過程と，要求仕様から機能仕様への変換過程にある．前者では，ユーザから要求仕様自体を獲得する際に，ユーザの記述した要求仕様を解析する手段として，あるいは対話を通じて仕様の欠落部分を補いながら完全な仕様に変換する手段として，自然言語処理が利用される．後者では，要求仕様から機能仕様への変換時に，変換過程をユーザに説明したり，仕様の誤りをユーザに指摘するために自然言語処理が利用される．本章では，前者の過程で生じる自然言語処理特有の問題点がどのように取り扱われているかについて，いくつかの事例を紹介する．また，次章では後者の過程で生じる問題点について検討する．

3.1 対話による仕様の獲得

[PSI]

対話を通じて，ユーザから要求仕様を獲得するプログラム合成システムに PSI[9,10] がある．PSI は，緊密に相互作用しあうプログラムモジュール（エキスパートと呼ぶ）からなる，知識依存型のシステムである．PSI では，システム全体の動作を仕様獲得フェーズとプログラム合成フェーズに分けて考えている．仕様獲得フェーズでは，まずパーサ／インタプリタ・エキスパートが入力文を解析し，構文解析木を解釈してプログラムネットと呼ぶネットワーク型データ構造を作成する．パーサ／インタプリタ・エキスパートは約 70 のプログラミング概念と約 175 語の語彙に関する知識をもっている．プログラミング概念のなかには，データ構造，制御構造，プリミティブな演算，複雑なアルゴリズムの概念などが含まれている．

プログラムネットに欠落情報がある場合は，問題領域エキスパートが応用分野に関する知識を活用して，パーサ／インタプリタ・エキスパートが欠落情報を補完するのを援助する．また，問題領域エキスパートによっても補完できない欠落情報は，対話管理エキスパート[26]を通じてユーザに問い合わされる．たとえば，仕様中の曖昧な箇所，厳密に定義されていない箇所はユーザに問い合わす必要がある．また，会話が進むにしたがって，新たな箇所に対話の焦点を移動し，ユーザに適切な文や質問を提示する必要がある．こ

のような処理を行うものが対話管理エキスパートである．

対話管理エキスパートは，パーサ／インタプリタ・エキスパートが作成したプログラムネットを参照し，まず現在の話題の焦点を決定する．次に，ヒューリスティックルールを用いてプログラムネットを辿り，新たにユーザに問い合わすべき質問や話題を決定するようにしている．また，対話過程でシステムとユーザのどちらが主導権をとるか，どんな対話をどのようなレベルで行うかを決定する．対話管理エキスパートが選択した話題や質問は，説明エキスパートに渡され，問題領域に依存した用語を用いて自然言語に変換される．

最後に，プログラムモデル構築エキスパートがプログラムネットを完全で整合性のとれたプログラムモデルに変換する．プログラム合成フェーズでは，コーディング・エキスパート（PECOS）と効率化エキスパート（LIBRA）が相互作用を行いながらプログラムモデルを目標言語に変換する[5,14]．PSI の主な目標言語は LISP や SAIL であるが，PASCAL などのブロック構造化言語にも拡張されている．

[KIPS]

ユーザとの対話によって，要求仕様の論理的な曖昧さの解決や欠落情報の補完を行うシステムにKIPS[24,25]がある．KIPS はいくつかの典型的な事務処理パターンから構成される事務処理プログラムを問題領域としており，目標言語として Hyper COBOL を選んでいる．また，対象世界のモデルとして，意味モデル，プログラムモデル，コードモデルを導入している．意味モデルは要求仕様の単語や文の意味を，プログラムモデルはユーザが意図しているプログラムを，コードモデルは自動生成されるプログラムコードをモデル化したものである．これらのモデルはオブジェクト指向で表現され，自然言語の理解やプログラムの自動生成に必要な知識は各モデルを構成するオブジェクトにもたせている．言い替えると，KIPS における要求仕様の獲得とは，ユーザの意図するプログラムを表現しているプログラムオブジェクトのスロット値を求めることになる．たとえば，要求仕様に現れる多義性や多様性については，オブジェクト内部にそれぞれの場合に応じたメソッドをもたせることによって解決している．また，要求仕様中の情報の欠落はデフォルトとデモンの概念を用いて解決している．

曖昧性の解決については，KIPS が要求仕様の曖昧な箇所を発見した時点で，ユーザインタフェースモジュールにメッセージとしていくつかの可能性ある候補を送り，出力された表示をユーザに直接選択してもらうようになっている．この過程で，KIPS が理解した要求仕様をユーザに知らせるために，プログラムチャートによる図形表現を表示するようにしている．このようなパラフレーズの概念は，自然言語理解の研究でもよく用いられており，ユーザが意図していることと要求仕様として記述したことの差異を明確化し，仕様の誤りを発見する上で有効なものである．

3.2 部分的な仕様と完全な仕様

一般に，自然言語による要求仕様からプログラムを自動合成するアプローチには，二つの考え方がある．一つのアプローチは，計算機を高度な知的推論マシンとみなし，計算機の側から人間に歩み寄らせようとするものである．前節で述べたシステムはいずれもこの立場に立つものである．このようなシステムでは，世界モデルや問題領域に関する知識をあらかじめ用意しておき，推論によって仕様中の不完全な箇所（部分的な仕様）を補う必要がある．たとえば，ユーザが素人の場合は，要求仕様に不完全な部分が多く現れるので，システムが情報の欠落を，デフォルト，問題領域の知識などによって補完しなければならない．また，プログラミングについて何も知らないユーザを対象とする場合は，システム側から積極的にユーザに問いかけ，システム自身が能動的に仕様獲得を行う機能も必要になる．これらの機能は，システムがどのようなレベルのユーザを対象とするかによって異なり，それにともない，要求仕様の記述レベルやシステムが必要とする知識の量も大きく異なってくる．

もう一つのアプローチは，計算機を単純なルーチンマシンとみなし，人間の思考を計算機の論理に歩み寄らせようとするものである．たとえば，自然言語ないしは疑似自然言語で書かれた要求仕様を形式的仕様に置き換え，その後はプログラムの等価変換や文法指向型変換によって低位仕様に変換するものがこの考え方に当たる．このアプローチでは，できるだけ要求仕様のレベルで厳密に記述する（完全な仕様）ことが必要になる．また，問題領域に関する知識は要求仕様内で定義されているので，システムはプログラム変換に関する知識（目標言語が定まっている場合，このような知識は問題領域が異なっても書き換える必要がない）のみをもてばよいという利点がある．本節では両者のアプローチに基づくそれぞれのシステムについて概説

する.

3.2.1 部分的な仕様

自然言語の特質はその不完全性にある．この不完全性によって要求仕様の読み手と書き手の注意を当面の問題にのみ向けさせることができる．また，注意の集中している部分のみを要求仕様に記述することによって，要求仕様の量を圧縮することができる．このことが要求仕様の可読性と理解容易性を高める原因となっている．さらに，要求仕様の不完全性を許すことによって，書き手の訓練（トレーニング）を少なくすることができ，より幅広いユーザが使用可能になるという利点がある．また，合成されたプログラムの保守が容易になるという利点がある．このような観点から仕様の不完全性を取り扱ったシステムに SAFE[3,4] がある．

[SAFE]

SAFE の目標は，要求仕様の中に現れる部分的な記述を完全にして，形式的仕様を生成することにある．SAFE によって合成されるプログラム（形式的な仕様）は，AP2 と呼ぶ実行可能な（runnable）超高級仕様記述言語で表現されている．SAFE では，自然言語で書かれた要求仕様とその意味的に等価な形式的仕様の相違点は，自然言語による仕様では完全な記述の代わりに部分的な記述が用いられている点にあるという仮定をおいている．さらに，これらの部分的な記述は，まわりの文脈から得られる情報によって完全な記述に補完することができると仮定している．

さらに，SAFE は部分的記述の意味的な問題に重点を置いたもので，構文解析上の問題が生じないように，あらかじめ解析された要求仕様を入力されると仮定している．したがって，自然言語に固有の構文上の曖昧性，多義性などはすでに解決済みとみなしている．SAFE で取り扱っている部分的な記述のいくつかを以下に示す．

（1） partial sequencing：要求仕様は必ずしもプログラムの実行順序に従って記述されていない．

（2） missing operand：プログラムの操作対象が欠落していることがある．

（3） incomplete reference：プログラムの操作対象が厳密に指定されていないことがある．言葉の言い回しによって言外にプログラムの実行方法を示すことがある．

（4） scope of conditional：自然言語では，条件文の範囲指定が明確に行われず，その後の文が条件文に含まれているかどうかの決定には，文脈情報を利用しなければならない．

SAFE は言語（Linguistic），計画（Planning），メタ評価（Meta-Evaluation）という三つの段階から構成されている．それぞれの段階が逐次的に起動されて，部分的な仕様からプログラムが合成される．一般に，部分的な記述からは複数個の解釈が生じるために，各段階ではバックトラックの状況が生じる．しかしながら，SAFE では段階間での相互干渉を許していないために，曖昧性が生じた場合，それぞれの段階で部分的に解決するか，次の段階まで解決を延期するかのいずれかの処理を行うようになっている．

これらの解決で用いる制約条件（well-formedness criteria）は三つのグループに分けることができる．

1. static flow criteria

この制約条件は，部分的記述のうちの partial ordering を解決し，プログラム中での演算の全般的な順序付けを決定するために，計画段階で利用される．この制約条件の中には「情報は出力（消費）される前に入力あるいは計算（生産）されなければならない」などがある．このような制約条件を用いた要求仕様の静的な解析は producer/consumer 分析と呼ばれ，演算順序の入れ替え，使用されていない演算や順序付けされていない演算の発見などに用いられる．この処理によって，部分的記述にあらわれる各事象間の静的な順序関係が決定される．

2. dynamic state-of-computation criteria

この制約条件は，プログラムの動的な流れを決定し，部分的記述を完全にするために，メタ評価段階で利用される．たとえば，部分的記述のうちの scope of conditional は「条件文の両方の分岐はいずれも実行可能でなければならない」といった制約条件によって決定される．メタ評価段階では，実際のデータを用いてプログラムの動的な流れを決定することはできないので，記号化された疑似データが利用される．この処理は，「プログラムの動作状況」という動的な文脈において，各事象が互いにどのような相互依存関係にあるかを決定していることに相当している．

3. global reference criteria

この制約条件は，プログラム全体における名前の使用法について調べ，手続き内で参照されている対象が既知のものかどうかを決定するために用いられる．このような決定は，プログラムの順序関係が確定した後でなければ行えないので，制約条件はメタ評価段階の

後で用いられる．この処理は，部分的記述における照応指示の解決に相当している．

[ARIES/I]

SAFE のプログラム合成手法は，与えられた要求仕様に各種の知識を適用しながらプログラムを合成するというものであった．これに対して，ルール化された知識を用いて必要なプログラム部品を検索し，それらを組み合わせてプログラムの合成を行うシステムにARIES/I[11] がある．ARIES/I では，目標言語として KIPS と同様に Hyper COBOL を用いている．ARIES/I の手法は，一般企業での業務要求は，業務用語を用いた特定の表現形式（言い回し）によって言い表されることが多いという仮定に基づいたものである．この言い回しは自然言語解析によって要求モデルに変換される．要求モデルは，入力された要求文に現れる（概念）項目とそれらの間の関係を，関係と実体からなるフレームのネットワークとして表現したものである．要求モデルは，さらに実際のデータ構造を表現したデータモデルを参照しながら，導出モデルに変換される．導出モデルは，オペレーション名，ファイル名，データ項目名など，データ処理に特有の表現を用いたプログラム合成に十分な情報をもったモデルである．最終的に，それぞれの言い回しに対応するプログラム部品が検索され，選択されたプログラム部品を用いて新たなプログラムが合成される．

[REDA]

SAFEと類似のアプローチをとるものにREDA[20),21)] がある．REDA は JRDL と呼ばれる日本語要求記述言語を用いた要求仕様 JRD を，概念要求記述言語 CRDL による概念要求記述 CRD に変換するシステムである．JRDL は REDA の開発を容易にするために，自然言語にいくつかの制限を加えたものとなっている．

（1） 要求文は単文を原則とする

（2） 代名詞の使用ができない

（3） 格の省略ができない

CRDL は Schank の概念依存理論に基づいた内部表現用の記述言語で，6個のエンティティ・タイプ，17個の primitive な動詞を用意している．また，それぞれの primitive にはあらかじめ格フレームを定めており，7個の格要素を用意している．要求記述中の抜け（必須格の欠如）や名詞の意味的な制約の不適格性は，この格フレームによって検出できるようになっている．

REDA の特徴は，要求記述中の必須機能の抜け（意味的な抜け）を検出するために，機能フレームと呼ばれる枠組みを用意している点にある．たとえば，「在庫管理システム」の要求仕様を考えると，「在庫のリストアップ」には必ず「ファイル検索機能」に関する記述がなければならない．もしこのような記述がなければ，この要求仕様の必須機能に抜けがあるという．このような問題は，ファイル操作に関連するひとまとまりの処理を機能フレームという形式で用意しておき，要求文中でその機能の一部しか言及されていない場合，ユーザにそれを指摘することで解決できる．必須機能の抜けは一文単位の構文解析からは発見することができず，機能フレームによって一種の文脈処理を実現していることになっている．

REDA のもう一つの特徴は，CRDL という要求仕様記述言語や目標言語に依存しない普遍的意味表現の存在を仮定することによって，同一の表現から複数の要求モデルが導出できるということにある．これを複合要求モデルと呼ぶ．複合要求モデルは，機械翻訳における Pivot 方式と同様の考え方で，各要求モデルと CRD の間の変換プログラムさえ用意すれば複数の要求記述が定義できるという利点がある．

3.2.2 完全な仕様

広い意味でのプログラマを対象とするシステムでは，ユーザがある程度プログラミングの困難さを知っているために，仕様記述の段階で可能なかぎり詳細な部分まで記述することを要求できる．さらに，このようにすれば仕様記述自身の完全性，無矛盾性を検証するための手がかりができる．また，システムがあらかじめ用意しなければならない，問題領域に依存した知識の量を減らすことができる．このような観点から開発されたものに TELL/NSL[8),13),23)] がある．

[TELL/NSL]

TELL/NSL は自然言語風仕様記述言語で，限定された自然言語（英語）を基本とし，語彙分割法と呼ばれる詳細化手法を用いて，問題固有の語句の定義を，自然言語の語句に沿ってより原始的な語句へと分解して定義しながら仕様を記述するものである．すでに述べた自動プログラム合成システムでは，要求仕様と問題領域に関する知識が分離され，要求仕様はユーザが，問題領域に関する知識はシステム管理者が用意するという暗黙の了解があったが，TELL/NSL ではすべてをユーザが定義するところに特徴がある．また，TELL/NSL で記述された要求仕様は，その意味とな

る一階述語論理式に変換され，最終的に Prolog プログラムに変換される．したがって，要求仕様の意味は一階述語論理式として形式的に得ることができ，無矛盾性やプログラムの正当性などの検証を行うための数学的基盤をもっているという特徴がある．

4. プログラム合成過程の説明機能

自然言語による非形式的な要求仕様は，その記述が容易になる反面，検証について考えると大きな障害となる．これまでに提案された検証法として，レビュ，文法チェック，シミュレーション，プロトタイプなどがある[1]．レビュは複数の人間が客観的に要求仕様を評価するもので，現在もっとも多く使われている方法であるが，計算機援用という立場をとると実現困難なものである．文法チェックは，要求仕様記述言語の文法にしたがって要求仕様を検査するものであるが，文法的に正しくても意味的に間違っていることも多く，あまり多くの成果は期待できない．最近注目を浴びている検証方法は，シミュレーション，プロトタイピングによる手法である．これは，実行可能な要求仕様言語を定義し，その上で記述された要求仕様を Symbolic Evaluator などで模擬的に実行するものである．このようなアプローチを用いたものに GIST[27),28)] がある．

[GIST]

GIST はソフトウェアの設計と保守に費やす負担をできるだけ低減しようという目的から開発されており，ユーザにどんな振舞いを望んでいるか (What) を記述させ，それがどのように達成されるか (How) については記述しなくてもよいようにした超高級仕様記述言語である．GIST による仕様は，システムの妥当な振舞いの形式的記述で，状態の集合，状態間の遷移，および状態や状態遷移に対する制約の集合から構成されている．GIST は，タイプ宣言，条件文，等式などのプログラミング言語に特有の概念が残されていること，仕様記述が宣言的よりもむしろ操作的 (operational)[2),19)] であることなどの特徴がある．したがって，自然言語による要求仕様よりも読みづらくなっている．このため，GIST には English Generator と Behavior Explainer と呼ばれる自然言語変換ツールが用意されている．English Generator は GIST で記述された要求仕様を英文に翻訳するものである．GIST で記述された要求仕様は Paraphraser によって格フレーム構造に変換され，その後に English Generator によって英文に翻訳される．Behavior Explainer は GIST で記述された要求仕様を Symbolic Evaluator で実行し，そのトレース過程を英文で説明するものである．

GIST の方法では，形式的記述が自然言語で表現されるために，たとえ GIST に精通しているユーザにとっても，デバッギングの際の支援になるという利点がある．しかしながら，合成されたプログラムがユーザの望んでいたものと等価なものかどうかが，仕様の動的な振舞いとして検査されるために，仮にプロトタイピングでバグが発見されたとしても，そのバグが (ユーザの要求分析が不完全であったために) 要求仕様に含まれたものか，実行可能な形への変換過程で (システムの知識が不完全あるいは欠落があるために) 新たに生じたものか判断できないという問題がある．

[AutoPro][30)]

われわれは，このような問題に対処するために，エキスパートシステムにおける説明機能を利用して，プログラム合成過程を説明する手法を開発している[15)]．通常，プログラム合成において最も多く使われている手法は，段階的詳細化[2)] と呼ばれるものである．段階的詳細化とは，まず大まかに述べられた要求仕様をいくつかの小さな問題に分割し，さらにこれらの問題をその意味を変えずにもっと細かな問題へと分割していくという手法である．この段階的詳細化の過程で，ユーザに変換の理由付けを説明する機能がプログラム合成システムの説明機能である．

プログラム合成に利用するルールの形式を

If A then B, C, D.

で表すものとする．このルールを用いて段階的詳細化の1ステップが行われると仮定する．このとき，ユーザの質問「変換Bを行うのはなぜですか」に対して「変換Aを行うためです．変換Aを行うためには，変換B, C, Dを行えというルールがあるためです」と説明するのが説明機能の基本原理である．

しかしながら，この説明機能には段階的詳細化に由来する大きな問題点がある．つまり，段階的詳細化では，変換過程が実世界に依存した抽象レベルから目標レベルに依存した詳細レベルまでいくつかの階層に分かれている．したがって，変換過程のあるレベルで用いられたルールのみをユーザに説明しても，そのルールがどのように実世界と対応づけて用意されたものか分からないために，変換過程そのもの，あるいは変換過程で用いられた知識がプログラム合成に適切なもの

かどうか理解できない.

このような問題に対処するために，われわれはなぜそのルールを準備したのかを表す「意図」を各ルールに付随させ，説明時にルールと同時に出力するようにしている．また，システムが変換過程の状況を管理しており，説明時にどのような状況でそのルールが適用されたかを出力するようにしている．

同様の考え方をもつものに，エキスパートシステムの XPLAIN[29]，その後継システム EES[18] がある．これらのシステムでは，通常のルールのほかに，いわゆる深い知識という問題領域特有の基本原理をもっている．これらのシステムが説明文を生成する際には，両者の知識を合わせて自然言語に翻訳しているために，ユーザがプログラムの変換過程について理解を深めるのに役立っている．このように，深い知識を問題領域における基本原理，通常のルールを問題領域からプログラムへの写像とみなすと，問題領域と関連づけてプログラムの合成過程を説明することができ，プログラムの正当性を検証する上で有効になると考えられる．

5. 今後の課題

本報告では，自然言語処理とソフトウェア工学・知識処理の観点から，自然言語を仕様記述に用いた事例的な研究について検討した．今後，これらのシステムが解決すべき問題点については以下のものがある．

非形式的な仕様記述の問題点の第1として，不完全な仕様を計算機が誤って理解するということがあげられる．この誤解の中には，一つの記述から複数の解釈が得られるということ，有効な解釈が得られず仕様中で不整合が生じていること，間違った解釈が得られることなどがある．

この結果，非形式的な仕様から形式的な仕様への翻訳に信頼性が欠如するという第2の問題点が生じる．このような問題を解決するために，たとえば KIPS のようにフローチャートを示すとか，システムが非形式的な仕様を別の表現で言い換えたり，曖昧性を解消するためにユーザに問い返すといった手段が考えられる．しかしながら，このようなユーザとのやりとりが新たな問題を生じさせる原因となる．つまり，頻繁なユーザとのやりとりがユーザを煩わしくさせ，逆に非形式性の長所を失わせることになる．

一方，完全な要求仕様では，要求仕様全体にわたって詳細で暗黙的に言及される情報に関しても明示して記述しなければならず，結果的に，要求仕様に必要な記述量は従来の形式的な仕様記述言語と基本的に差がなく，仕様記述が複雑になるという問題がある．また，関連する情報が仕様中の至るところに散らばることになり，記述が困難になるという問題が残されている．さらに，このことが要求仕様を改訂し，修正しようとするときに大きな問題となる．

第3の問題点として，プログラム合成の適用範囲を拡大するために，受容可能な要求文を増やすと同時に，対応する問題領域知識やプログラム変換知識を追加しなければならないという点があげられる．要求文の解析については，自然言語処理の研究で得られた成果を生かすとしても，プログラミング言語は高度に制約を受けた言語なので，問題領域からプログラムへの写像に必要な知識を一対一に用意できるかどうかについては今後十分に検討する必要がある．

第4の問題点として，従来のプログラム合成システムでは，問題領域に依存した簡単な仕様（単文あるいは重文からなる文の集合）についてのみ検討を行っているが，要求内容が複雑になるにつれて，要求仕様を独立した文集合で表現することが困難になるということがあげられる．この問題に対処するためには，仕様記述を文単位から文章単位に拡張すること，ならびに文章として記述された要求仕様を解析可能とするために，文脈処理メカニズムを導入することが必要である．

最後の問題点として，要求仕様と目標言語の間にどのような中間表現を用意すればよいかという点があげられる．普遍的意味表現を仮定して，機械翻訳における Pivot 方式のようなプログラム合成方式が実現できれば，各種のプログラミング言語によるプログラムが合成可能であるが，このような普遍的意味表現が果たして存在するかどうか，現在は分かっていない状況である．また，仮りに存在すると仮定しても，要求仕様と目標言語のセマンティックスを完全に吸収した普遍的意味表現を定義することが今後の課題である．

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) 有澤 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, Proc. of 5th IJCAI, pp. 389-397 (1977).

4) 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).
5) Barstow, D. R.: Knowledge-Based Program Construction, North-Holland, New York (1979).
6) Barstow, D. R., Shrobe, H. E. and Sandewall, E.: Interactive Programming Environments McGraw-Hill (1984).
7) Borgida, A., Greenspan, S. and Mylopoulos, J.: Knowledge Representation as the Basis for Requirements Specifications, IEEE Computer, Vol. 18, No. 4, pp. 82-91 (1985).
8) 榎本 肇, 米崎直樹, 佐伯元司: 情報の構造と自然言語による仕様記述, bit, 大野 豊編: 新しい時代のソフトウェア, Vol. 16, No. 6, pp. 34-43 (1984).
9) Green, C.: A Summary of the PSI Program Synthesis System, Proc. of 5th IJCAI, pp. 380-381 (1977).
10) Green, C., Richard, P. G. and Kant, E.: Results in Knowledge Based Program Synthesis, Proc. of 6th IJCAI, pp. 342-344 (1979).
11) 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, 情報処理学会論文誌, Vol. 27, No 4, pp. 417-424 (1986).
12) Hobbs, J. R.: What the Natural Language Tells Us about How to Make Natural-Language-like Programming Languages More Natural, Proc. of the Symposium on Artificial Intelligence and Programming Languages, ACM SIGART Newsletter, No. 64, pp. 85-93 (1977).
13) 市川 至, 蓬莱尚幸, 佐伯元司, 米崎直樹, 榎本肇: 自然言語に基づく静的システムの仕様のプロトタイププログラムへの変換手法, 情報処理学会論文誌, Vol. 27, No. 11, pp. 1112-1128 (1986).
14) Kant, E. and Barstow, D. R.: The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis, IEEE Trans. on Softw. Eng., Vol. SE-7, No. 5, pp. 458-471 (1981).
15) 松元貴志, 上原邦昭, 豊田順一: プログラム合成システムにおける知識の適用状況を用いた説明, 第1回人工知能学会大会, 9-2 (1987).
16) 松本吉弘: ソフトウェア工学における知識情報処理技術の応用, 電子通信学会誌, Vol. 69, No. 11, pp. 1176-1180 (1986).
17) Mostow, J. (ed.): Special Issue on Artificial Intelligence and Software Engineering, IEEE Trans. on Softw. Eng., Vol. SE-11, No. 11 (1985).
18) 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).
19) 野木兼六: 要求定義技術の最新の動向, 情報処理, Vol. 27, No. 1, pp. 21-30 (1986).
20) 大西 淳, 阿草清滋, 大野 豊: 要求モデルに基づいた要求定義支援手法, 情報処理学会, 「プロトタイピングと要求定義」シンポジウム, pp. 19-28 (1986).
21) 大西淳, 阿草清滋, 大野 豊: 要求定義のための要求フレーム, 情報処理学会論文誌, Vol. 28, No. 4, pp. 367-375 (1987).
22) Partsch, H. and Steinbruggen, R.: Program Transformation Systems, ACM Comput. Surv., Vol. 15, No. 3, pp. 199-236 (1983).
23) 佐伯元司, 米崎直樹, 榎本 肇: 自然言語の語彙分割による形式的仕様記述, 情報処理学会論文誌, Vol. 25, No. 2, pp. 204-214 (1984).
24) 杉山健司, 秋山幸司, 亀田雅之, 牧之内顕文: 対話型自然言語プログラミングシステムの試作, 電子通信学会論文誌, Vol. J 67-D, No. 3, pp. 297-304 (1984).
25) 杉山健司, 秋山幸司, 亀田雅之, 牧之内顕文: 対話型自然言語プログラミングシステムにおける自然言語の理解, 情報処理学会, 「自然言語処理技術」シンポジウム, pp. 35-40 (1983).
26) Steinberg, L. I.: Question Ordering in Mixed Initiative Program Specification Dialogue, Proc. of AAAI-80, pp. 61-63 (1980).
27) Swartout, B.: GIST English Generator, Proc. of AAAI-82, pp. 404-409 (1982).
28) Swartout, B.: The GIST Behavior Explainer, Proc. of AAAI-83, pp. 402-407 (1983).
29) Swartout, W.: XPLAIN: A System for Creating and Explaining Expert Consulting Systems, Artif. Intell., Vol. 21, No. 3, pp. 285-325 (1983).
30) 上原邦昭, 藤井邦和, 豊田順一: 自然言語による仕様からの自動プログラム合成, コンピュータソフトウェア, Vol. 3, No. 4, pp. 55-64 (1986).

(昭和62年6月15日受付)