Vol. 27 No. 11 情報処理学会論文誌 Nov. 1986

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†

市川 至†† 蓬莱尚幸††* 佐伯元司††
米崎直樹†† 榎本肇††*

本論文では，自然言語ベースの仕様記述言語 TELL/NSL におけるシステムの静的な仕様記述から，変換により実行可能な Prolog プログラムを得てテストデータによる試験を行うラピッドプロトタイピング手法について述べる．TELL/NSL の仕様記述は，まずその意味となる1階述語論理式へ変換され，次にホーン節型式に変換される．さらに，述語の入出力モードを入出力依存グラフを利用して決定し，その入出力モードに基づき正しい動作をするように Prolog の実行制御を考慮してリテラルや節の並べ換えを行い，実行可能な Prolog プログラムに変換する．これらの変換は，段階的に適応される部分変換からなり，それぞれの変換では健全性が保証され，全体の変換により得られるプログラムは元の仕様記述を満たし部分正当であることが保証される．得られたプログラムを実行しデータ試験を行うことにより，元の仕様記述において不明確な点を発見することが可能となった．

1. まえがき

近年，ソフトウェアの開発コストを低下させ品質を上げるため，仕様記述の段階で可能な限り詳細な部分まで記述を行うことが必要となってきている．さらに，仕様記述自身が完全であるか，無矛盾であるかや仕様記述者の意図したように記述されているかどうかテストを行うための研究や，仕様記述から系統的にソフトウェアを生成する試みがなされている[1),2)]．

特に，仕様が要求定義者の意図したように記述されているかどうかを，仕様記述から人間によりプロトタイププログラムを得て，実行しテストするという，ラピッドプロトタイピングは，その結果により仕様記述者が仕様を修正することが可能となり有用である．

従来のラピッドプロトタイピングでは，プログラムを人間の手により得ている[1)]．このような方法は，得られたプログラムが仕様を満たしているかどうか（プログラムの部分正当性）の論理的根拠がなく，人間が誤りを犯してしまうと，プロトタイプの効果を生かすことができない．

これに対して，仕様記述言語が形式的な意味を持ち，プロトタイププログラムを（半）自動的な変換により得る方法は，その変換によるプログラムが元の仕様を満たしていること（変換の健全性）を保証しさえすれば，得られたプログラムは元の仕様に対して部分正当であるということが保証できる．

榎本らによって提案された自然言語風仕様記述言語 TELL/NSL[3)] では，語彙分割法により自然な形で詳細化が行われるため，人間にとって記述性，了解性に富む記述が可能である．さらにその意味は1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といった各種の性質について検証を行える数学的基盤を持っている．

本論文では，TELL/NSL による静的仕様記述から，その意味として得られる1階述語論理式を介し，プロトタイプとして実行可能な Prolog プログラムを(半)自動的変換により得て，仕様に対するデータ試験が行えるプロトタイピング手法について報告する．

変換は，図1に示した構成をとる．まず，TELL/NSL の記述よりその意味となる1階述語論理式(第2章)を介して，これをホーン節型式に変換する(第3章)．さらに Prolog の実行制御を考慮し，述語の入出力モードを入出力依存グラフを利用して決定し，これに基づきリテラルや節の並べ換えを行い，実行可能な Prolog プログラムに変換する(第4章)．これらの変換についての健全性は保証され，変換により得られるプログラムが解を得れば，その解は仕様を満たしていることが保証される．

図2は，ページのフォーマット問題について自然言語によるユーザからの要求仕様で，文献 4) より引用した．TELL[5),6)] では，このような自然言語による仕様記述を，仕様記述者が図3のような TELL/NSL の

† The Method of Transformation from Natural Language Based Functional Specifications into Prototype Programs by ITARU ICHIKAWA, HISAYUKI HORAI, MOTOSHI SAEKI, NAOKI YONEZAKI and HAJIME ENOMOTO (Department of Computer Science, Faculty of Engineering, Tokyo Institute of Technology).

†† 東京工業大学工学部情報工学科
* 現在 富士通国際情報社会科学研究所

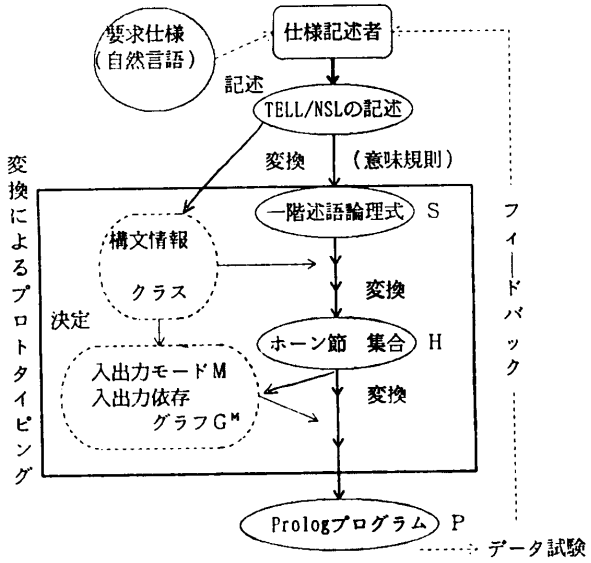


図 1 本手法の概要
Fig. 1 Overview of this prototyping method.

記述にする．この仕様は一見完全な記述であるように見えるが，第5章で指摘するように，この変換により得られたプロトタイププログラムを実行させテストデータによる試験を行うことにより，不明確な記述を含んでいることが判明した．このように，この変換手法によるラピッドプロトタイピングは，仕様記述の不明確な部分を発見することができ，仕様記述の理解の助けとなる．

The input to the procedure is a file of program specifications. The object of the procedure is to paginate the input by separating the input into pages. The output is the paginated text in a file ready for printing. Each page must contain some minimum number of lines, denoted MIN. There is also a maximum number of lines, MAX, which is the physical page length. Between the MIN and MAX number of lines, the most important line of text should be chosen to start the next page. Important lines are defined by the following (in order of importance).

1) **The first line of the input specification.**
2) **Those lines which are preceded by blank lines (indicating the beginning of a new specification). The more blank lines preceding a line, the more important that line is.**
3) **The least indented line (on the assumption that the specifications are written like structured programs with meaningful indenting).**

In the case of ties (e.g., two lines each preceded by four blank lines), the line closest to MAX should be chosen for the next page. In case there are no important lines encountered, the page should be broken at MAX lines.

図 2 自然言語による要求仕様の例
Fig. 2 Example of requirement specification written in natural language.

2. TELL/NSL

TELL/NSL[3),5),6)]は，限定された自然言語（英語）を基本とし，語彙分割技法[7)]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，自然言語の語句に添ってより原始的な語句へと分解していくことにより進めていく．TELL/NSL では is や not 等の一般の語句はその意味が既に定義済みであるとして利用できる．

TELL/NSL には，システムの時間に依存しない関係を記述する静的記述と，システムの動作を記述する動的記述が用意されているが，ここでは静的記述のみを対象とする．

2.1 TELL/NSL の語句定義

TELL/NSL の静的記述には，語句の定義方法として，関数定義とクラス定義の2種類があり，それぞれ構文規則に対応して割り当てられた意味規則によって，意味となる1階述語論理式に変換される．

図2を TELL/NSL により記述した例は図3である．この中で定義する語句には下線が引かれている．

TELL/NSL では，共通な性質を持つ物の集まりは，普通名詞としてクラス定義により定義される．クラス定義では，定義される普通名詞，その属性を表す語句，クラスの要素となる物の構成方法を表す語句や関連する語句も同時に定義する．図3中で定義されている普通名詞は，file, page, line number 等である．

また，直積，直和，集合や列のように，よく使用される概念については TELL/NSL ではあらかじめ用意されており，その記述が利用できる．例えば，図3の line は，文字列として定義している．その他，line に関連する語句としては tail があることも記述されている．

さらに，small-talk[8)] などの一般的な対象指向言語と同様にクラスの階層関係も定義でき，上位クラスに対する定義を下位クラスに継承することができるため，記述量を減らし読みやすくすることができる．

TELL/NSL では，クラスのインスタンス間の関係を規定するような普通名詞・形容詞を，以下の形式の関数定義で記述する．

定義文 **means that** 定義本体 **end**;

図3の最初の関数定義では定義語句は

```
Pages T is pagenated from file F between minimum line number Min
  and maximum line number Max means that
   case 1) F is empty: T is empty.
   case 2) F is not empty:
        2-1) The result of separating a page from F is,
               page P and file F1.
        2-2) Pages T1 is pagenated from F1 between Min and Max.
        2-3) T is the concatenation of P and T1.
  The result of separating a page from file F is page P
    and file F1 means that
     1) Line number N is the most important in F from Min+1 to Max+1.
     2) The result of breaking F at line number N is P and F1.
    The result of breaking file F at line number N is
      file F1 and file F2 means that
       1) F1 is the first half sequence of F with line number N-1.
       2) F2 is the last half sequence of F with line number N.
    end break;
  end separate a page;
  Line number N is the most important in file F from line number I
    to line number J means that
     case 1) I is J: N is J.
     case 2) J is greater than I:
             Let line number M be the most important
               in F from I to J-1,
             case2-1) M is the more important than J in F: N is M.
             case2-2) M is not the more important than J in F: N is J.
    Line number I is the more important line number
      than line number J in file F means that
       Let NI be the number of blank lines in F before I, and
           NJ be the number of blank lines in F before J,
       case 1) NI is greater than NJ .
       case 2) NI is NJ:
            2-1) II is the level of indentation of I th element of F.
            2-2) IJ is the level of indentation of J th element of F.
            2-3) II is greater than IJ .
    end the more important;
    Number N is the number of blank lines in file F
      before line number I means that
       case 1) I is 1: N is 0.
       case 2) I is not 1:
         case 2-1) I - 1 th element of F is not blank line: N is 0.
         case 2-2) I - 1 th element of F is blank line:
              2-2-1) N1 is the number of blank lines in F before I1.
              2-2-2) N is N1 + 1.
    end number of blank lines;
      Number N is the level of indentation of line L means that
         case 1) The first charactor of L is not blank: N is 0.
         case 2) The first charactor of L is blank:
              2-1) N1 is the level of indentation of tail of L.
              2-2) N is N1+1.
       end level of indentation;
    end the most important;
end pagenated;
Number is positive integer.
Line number is positive integer.
Pages is sequence of page.
Page is file.
Line is sequence of charactor associated with tail.
   Line L1 is the tail of line L means that
        1) L is the concatenation of charactor C and line L1.
    end tail;
end line;
File is sequence of line
  associated with first half sequence, last half sequence.
  File F1 is the first half sequence of file F
    with line number N means that
     case 1) N is 0: F1 is empty file.
     case 2) N is not 0:
          2-1) F is concatenation L and F2.
          2-2) F3 is first half sequence of F2 with N -1.
          2-3) F1 is concatenation L and F3.
  end first half sequence;
  File F1 is the last half sequence of file F
    with line number N means that
     case 1) N is 1: F1 is F.
     case 2) N is not 1:
          2-1) F is concatenation L and F2.
          2-2) F1 is last half sequence of F2 with N -1.
  end last half sequence;
end file;
lexicon
Minimum line number := line number.
Maximum line number := line number.
Blank line := empty.
```

図 3 TELL/NSL による仕様記述例
Fig. 3 Example of specification written in TELL/NSL.

pagenated である．定義文では，T, F, Min, Max の関係 pagenated を定義しており，例えば，F は file のクラスを持ち pagenated に対しては前置詞 of を伴って出現することを表している．定義本体では，定義文で定義した関係の満たすべき条件を場合分けや箇条書きの文を用いて記述する．

図3の中で，関数定義で定義された定義語句は，他に separate a page, break, most important 等であるが，separate a page は pagenated の中にネストして定義されている．このような場合には，ネストの中の定義で用いられる語句で，ネスト外の定義文に現れる語句は定義しなくとも参照することができるという文脈機構を採用している．

本論文では，仕様中に現れる語句がすべて仕様記述者により定義され，未定義語句を含まない正しい無矛盾な TELL/NSL の静的な関数的記述を対象とする．

2.2 意味となる論理式

TELL/NSL で記述された関数的仕様記述は，TELL/NSL の意味規則によってその意味となる1階述語論理式に翻訳される．TELL/NSL の構文・意味規則の詳細は，ここでは述べないので，文献3),9)を参照されたい．

図4は図3に対して得られる意味となる論理式である．このように，図3の F, T 等の固有名詞は変数に

```
∀T ∀F ∀Min ∀Max[pagenated(T,F,Min,Max)↔
    pages(T)∧file(F)∧line_number(Min)∧line_number(Max)∧
    ((F=empty∧T=empty)∨
     (~F=empty∧∃F1[file(F1)∧∃P[page(P)∧∃T1[pages(T1)∧
            separate_a_page(P,F1,F,Min,Max)∧
            pagenated(T1,F1,Min,Max)∧T=concatenation(P,T1)]]]))]                    ——— (1)

∀P ∀F1 ∀F[separate_a_page(P,F1,F,Min,Max)↔
    page(P)∧file(F1)∧file(F)∧line_number(Min)∧line_number(Max)∧
    ∃N[line_number(N)∧
      ∃V0[+(V0,Min,1)∧∃V1[+(V1,Max,1)∧most_important(N,F,V0,V1)]]∧
      break(P,F1,F,N)]]

∀F1 ∀F2 ∀F ∀N[break(F1,F2,F,N)↔
    file(F1)∧file(F2)∧file(F)∧line_number(N)∧
    ∃V0[-(V0,N,1)∧first_half_sequence(F1,F,V0)]∧last_half_sequence(F2,F,N)]

∀N ∀F ∀I ∀J[most_important(N,F,I,J)↔
    line_number(N)∧file(F)∧line_number(I)∧line_number(J)∧
    ((I=J∧N=J)∨(J>I∧∃M[number(M)∧
            most_important(M,F,I,J-1)∧(more_important(M,J,F)∧N=M∨
            ~more_important(M,J,F)∧N=J)]))]

∀I ∀J ∀F[more_important(I,J,F)↔
    line_number(I)∧line_number(J)∧file(F)∧
    ∃NI[number(NI)∧number_of_blank_lines(NI,F,I)∧
        ∃NJ[number(NJ)∧number_of_blank_lines(NJ,F,J)∧
           (NI>NJ∨(NI=NJ∧∃II[number(II)∧
                ∃V0[element(V0,F,I)∧level_of_indentation(II,V0))∧
                  ∃IJ[number(IJ)∧
                    ∃V1[element(V1,F,J)∧level_of_indentation(IJ,V1))∧
                    II>IJ]]]))]]]
                    ........

∀V0[number(V0)↔positive_integer(V0)]                                            ——— (2)
∀V0[line_number(V0)↔positive_integer(V0)]
∀V0[pages(V0)↔
    V0=empty∨∃V1[page(V1)∧∃V2[pages(V2)∧V0=concatenation(V1,V2)]]]
∀V0[page(V0)↔file(V0)]
∀V0[line(V0)↔
    V0=empty∨∃V1[charactor(V1)∧∃V2[line(V2)∧V0=concatenation(V1,V2)]]]
∀V0 ∀V1[first_charactor(V0,L)↔
   charactor(V0)∧line(V1)∧∃V2[line(V2)∧V1=concatenation(V0,V2)]]
∀L1 ∀L[tail(L1,L)↔
    line(L1)∧line(L)∧∃C[charactor(C)∧L=concatenation(C,L1)]]
∀V0 ∀V1[first_cahractor(V0,V1)↔
    charactor(V0)∧line(V1)∧∃V2[line(V2)∧V1=concatenation(V0,V2)]]
∀V0[file(V0)↔
   V0=empty∨∃V1[line(V1)∧∃V2[file(V2)∧V0=concatenation(V1,V2)]]]                 ——— (3)
                    ........
```

図4 図3の意味となる論理式
Fig. 4 Meanings of specification in Fig. 3 as logic formulas.

翻訳され，クラスを構成する構成子は関数記号として翻訳されている．file, line 等の普通名詞はクラス述語名に翻訳され，pagenated 等の関数定義で定義された語句は，関係を表す述語記号として翻訳される．また，各語句を定義するモジュールの意味は，語句に対応する述語を定義する固有の形に制限された論理式（以下，定義式と呼ぶ）として得られ，仕様全体の意味は，定義式の集合として得られる．以下，定義式の構文則について定義する．

また，TELL/NSL の記述からは，意味となる論理式のほかに，述語がクラス述語であるかどうかといった情報や，クラスの階層関係などの構文情報も得ることができる．

2.2.1 定義式の構文則

TELL/NSL から得られる定義式の集合 SPEC は，通常の1階述語論理式[10]の部分集合であり，以下その構文を定義する．

[定義 2.1] 定義本体の意味となる論理式：FORM

TELL/NSL の定義本体中に出現する自然言語文の翻訳である論理式 FORM は，等号を含む通常の1階述語論理式に，変数束縛が行われる場合には，必ずクラス述語が伴って出現するという制限が加わる．

すなわち，FV(Φ) を式 Φ 中に出現する自由変数の集合を表すとすると，FORM は以下を満たす最小集合である．

- Φ が変数束縛のない任意の論理式ならば，$\Phi \in$ FORM.
- $\Phi \in$FORM, $x \in$FV(Φ), α がクラス述語名ならば，$\forall x[\alpha(x) \rightarrow \Phi]$, $\exists x[\alpha(x) \wedge \Phi] \in$FORM.

例えば，図4の式(1)のように変数束縛が行われる場合，クラス述語が，束縛変数に対して必ず存在している．

[定義 2.2] TELL/NSL の記述の意味となる論理式の集合：SPEC

i) クラス定義の場合：定義されるクラス名が翻訳されたクラス述語名を α とすると，クラス定義の意味となる論理式は，ある項がそこで定義されるクラスに含まれるかどうかを判定する述語の定義式となり，その形式は以下のとおりである．

$\Phi \in$FORM, FV(Φ)$=\{x\}$ ならば，

$\forall x[\alpha(x) \leftrightarrow \Phi] \in$SPEC

これをクラス述語 α の定義式と呼ぶ．

図4の式(2)は，図3の line number の記述に対する，クラス述語 line_number の定義式であり，図4の式(3)は図3の file の記述に対するクラス述語 file の定義式である．

ii) 関数定義の場合：まず，p が定義語句の翻訳であり，$x1 \cdots xn$ が定義文中に出現する固有名詞の翻訳となる変数であるとすると，定義文の翻訳はリテラル $p(x1, \cdots, xn)$ となる $(n \geq 1)$．さらに，$\alpha_1, \cdots, \alpha_n$ をクラス述語名とし，$\Phi \in$FORM を定義本体の意味となる論理式とすると，関数定義の意味となる論理式は，以下の形式となる．

$$\forall x1 \cdots \forall xn[p(x1, \cdots, xn) \leftrightarrow \Phi \wedge \alpha_1(x1) \wedge \cdots \wedge \alpha_n(xn)] \in \text{SPEC}$$

これを，$\alpha_1 \times \cdots \times \alpha_n$ の定義域を持つ n 引数述語 p の定義式と呼ぶ．

図4の式(1)は，図3の語句 pagenated の記述に対する，述語 pagenated の定義式の例である．

3. ホーン節形式への変換

変換は，先の図1に示した構成をとる．まず，TELL/NSL の記述より意味規則を用いて，意味として得られる論理式（定義式）の集合 S を得る．次に S を，ホーン節の集合 H に変換する．以下この変換について述べる．

3.1 ホーン節形式への変換の健全性

この変換は，論理式の集合に対して多段的に適応されるいくつかの部分変換からなり，すべての部分変換は閉世界仮説の下で健全であることが保証される．

ここで，閉世界仮説[11]の下での健全性を以下に定義する．まず，閉世界仮説の下での反駁とは以下を意味する．

[定義 3.1] 閉世界仮説の下での反駁：

論理式の集合 S と閉論理式 Ω について，まず，完全な証明手続きを行って $S \cup \{\leftarrow \Omega\}$ が反駁されると，S で $\leftarrow \Omega$ が閉世界仮説の下で反駁されたとし，$S \vdash_c \Omega$ と書く．$\leftarrow \Omega$ が反駁できなければ，$S \nvdash_c \Omega$ と書き，S で $\leftarrow \sim \Omega$ が閉世界仮説の下で反駁されるとし，$S \vdash_c \sim \Omega$ と書く．

[定義 3.2] 変換の健全性：

変換 T が，論理式の集合 S を $S' = \{T(\Psi) \mid \Psi \in S\}$ に変換するとする．任意の正リテラル Φ について，ただし T で新しい述語記号 π を導入する場合は π を含まないような Φ について，$S' \vdash_c \Phi$ ならば，$S \vdash_c \Phi$ のとき，この変換 T は閉世界仮説の下で健全であると呼ぶ．

部分変換のいくつかは，次の定義する等価変換であ

る.

[定義 3.3] 等価変換:

変換Tが，任意の論理式 Φ について，

$\models \Phi \leftrightarrow T(\Phi)$

である場合，このTを等価な変換と呼ぶ．ただし，$\models$ Pとは，Pが妥当であることを表す.

また，証明手続きの完全性により等価な変換は健全であることは明らかである.

各部分変換は，以下のように表現される書き換え規則の集まりである.

COND　：適応条件

　適応前パターン ⇒ 適応後パターン

WHERE：新たに導入する記号

ADD　：新しく加える論理式

DEL　：削除する論理式

各書き換え規則は，COND で示される適応条件が成立し適応前パターンにマッチするパターンが存在する限り繰り返し適応され適応後パターンに書き換えられる．適応されると WHERE で示される今までに出現していない新しい記号が導入され，定義式の集合に新しく ADD で示された論理式が加わり，DEL で示された論理式が除かれる．部分変換は，適応できる書き換え規則がなくなると変換を終了する.

各部分変換が健全ならば，部分変換を多段的に並べて構成している全体の変換も健全ということは自明である.

3.2 ホーン節形式への変換

定義ごとの論理式を，そこで定義される定義語句名をそのまま述語記号として，これを頭部とするホーン節に変換する.

[変換 H.1] 定義式の解釈変更:

定義式の "↔" を "←" に書き換える.

COND: $\forall x1 \cdots xn[R(x1, \cdots, xn) \leftrightarrow \Phi]]$ は定義式

$\forall x1 \cdots xn[R(x1, \cdots, xn \leftrightarrow \Phi]]$

$\Rightarrow \forall x1 \cdots xn[R(x1, \cdots, xn) \leftarrow \Phi]]$

なお以降，本体とは $\forall x1 \cdots xn[R(x1, \cdots, xn) \leftarrow \Phi]$ の Φ をさすものとし，本体中とは，この Φ の部分式をさすものとする.

変換前の論理式集合を S とし，変換後を S' とすると，$S' \vdash_e \Phi$ かつ $S \vdash_e S'$ ならば $S \vdash_e \Phi$ である．ここで $S \models S'$ なので，$S \vdash_e S'$ が成立する．よって，この変換は健全である.

[変換 H.2] 本体中の含意および全称限量子の除去:

本体中の含意および全称限量子について，$\Phi \rightarrow \Psi$ を $\sim\Phi \vee \Psi$ で，$\forall x[\Phi]$ を $\sim\exists x[\sim\Phi]$ で書き換えることで除去する等価変換.

[変換 H.3] 本体中の否定の内部移動と二重否定の除去:

ド・モルガンの法則と $\models \sim\forall x[\Phi] \leftrightarrow \exists x[\sim\Phi]$ と $\models \sim\sim\Phi \leftrightarrow \Phi$ を用いて，本体中の否定がもっとも小さな部分式に付くようにする等価変換である．この変換では $\sim\exists x[\Phi]$ を $\forall x[\sim\Phi]$ にすることはない．ここまでの変換を行った結果，本体中では，限量は存在限量子によってのみ行われ，接続詞は $\wedge$ と $\vee$ のみで，否定記号は原子式または存在限量された式にしか用いられていない.

[変換 H.4] 本体中の存在否定の除去:

本体中の $\sim\exists$ (存在否定) を新しい定義式を導入し除去する．($\forall\sim$(否定の全称) は変換 H.2 によりすでに $\sim\exists$ に変換されている).

COND: $\sim\exists x[\Phi]$ は本体の部分式，

　$x1, \cdots, xn$ は Φ で free に出現する x 以外の変数

　$\sim\exists x[\Phi] \Rightarrow \sim\pi(x1, \cdots, xn)$

WHERE: π はこれまでに出現していない新しい述語記号

ADD: $\forall x1 \cdots \forall xn[\pi(x1, \cdots, xn) \leftarrow \exists x[\Phi]]$

この変換は，健全性が保証される．以下にその証明を示す.

[証明]

定義式にこれまでの変換を施した式の集合を S とし，S にこの変換を行った結果を S' とする．π 以外の述語記号からなる任意のリテラルPで，$S' \vdash_e P$ であるとする.

Pの証明において，π を含む式を使用しない場合に，$S \vdash_e P$ は明らかである．以下，Pの証明には π を含むものとし，S' を以下のような式の集合とする．$\sim$P を反駁するには，$\sim$Q が反駁されることが必要であり，$\sim$Q を反駁する際に π を含む式(1)を使用したとする．ここで＊は項の列の省略を表すものとする.

$$\begin{cases} P \leftarrow \cdots Q \cdots \\ Q \leftarrow \cdots \sim\pi(*) \cdots & (1) \\ \quad \cdots\cdots \\ \pi(*) \leftarrow \exists x[\Phi] & (2) \end{cases}$$

さらに，$\sim$Q の反駁では，(1)式より $\sim\sim\pi(*)$ を反駁することになる．S' を節形式にしたときに $\sim\pi(*)$

が出現しないので，閉世界仮説を用いずに直接 ～～π(＊) を反駁できない．また閉世界仮説より，～π(＊) の反駁が失敗すれば，～～π(＊) の反駁ができることになる．以上より，～π(＊) の反駁が失敗するしか，Pを証明する手段はない．S' を節形式にしたときに π(＊) が出現するのは(2)式を節形式にした式しかないので，～π(＊) の反駁が失敗するには ～∃x[Φ] の反駁が失敗するしかない．よって，Pが証明されるならば，～∃x[Φ] の反駁は失敗する． (3)

一方，S' に対して S は以下の形式をしている．

$$\begin{cases} P \leftarrow \cdots Q \cdots \\ Q \leftarrow \cdots \sim \exists x[\Phi] \cdots \\ \quad \cdots\cdots \end{cases}$$

Pを証明するには，～P を反駁し，～～∃x[Φ] が反駁できればよい．さらにこれは，閉世界仮説を用いると，～∃x[Φ] の反駁が失敗すればよい．(3)によりこれは失敗する．よってPは証明され，$S \vdash_e P$ となる．

以上よりこの変換は健全である．■

[変換 H.5] 全称・存在限量の除去：

本体中の存在限量，および，定義式全体にかかる全称限量について除去する．その際に束縛変数 x は新しい変数 y で置き換える．

COND：∃x[Φ] は本体の部分式

$$\exists x[\Phi] \Rightarrow \Phi[y/x]$$

WHERE：y はこれまでに出現していない変数

COND：∀x[Φ] は定義式

$$\forall x[\Phi] \Rightarrow \Phi$$

ここで，$\Phi[\Psi/\Omega]$ は，Φ 中に出現する Ω のすべての出現を Ψ で置き換えることを表す．

例：$\forall x1 \cdots xn[R(x1, \cdots, xn) \leftarrow$

$$\exists y1 \cdots ym[Q(y1, \cdots, ym, x1, \cdots, xn)]]$$

$$\Rightarrow R(x1, \cdots, xn) \leftarrow Q(z1, \cdots, zm, x1, \cdots, xn)$$

変換が健全な変換であることを以下に証明する．

[証明]

ここまでの変換で，本体中では，限量は存在限量子によってのみ行われ，接続詞は ∧ と ∨ のみで，否定記号は原子式にしか用いられていない．また，存在束縛は定義式の本体中のみに出現し，全称束縛は定義式全体にのみに用いられている．

この場合，束縛変数を適当に付け替えることにより，本体中の存在束縛をすべて本体全体に用いるように等価変換でき，また，Φ に y が出現しなければ $\Phi \leftarrow \exists y[\Psi]$ を $\forall y[\Phi \leftarrow \Psi]$ に等価変換できるので，すべての限量を定義式全体の全称束縛になるように等価変換することができる．

また，$\models \forall x[\Phi] \leftrightarrow \Phi$ である．

よって，以上の事実から，この変換は健全な変換である．■

[変換 H.6] 選言標準形化：

本体を ∧∨ の分配則を用いて選言標準形にし，次に，論理式が $\Phi \leftarrow \Psi \vee \Pi$ ならば，$\Phi \leftarrow \Psi$ と $\Phi \leftarrow \Pi$ の二つの論理式に分離する等価な変換．ここまでの変換の結果，本体は，原子式または原子式の否定から構成される連言となる．その際，否定の付いた原子式を頭部に移動することはしない．

[変換 H.7] 等号の簡略化：

本体中に $x=t$ が出現した場合に，その論理式のすべての x に t を代入して式の簡略化を行う等価変換である．

例：$P(\cdots x \cdots) \leftarrow x=t \wedge \cdots \wedge Q(\cdots x \cdots) \Rightarrow$

$$P(\cdots t \cdots) \leftarrow \quad \cdots \wedge Q(\cdots t \cdots)$$

図3の TELL/NSL の記述からここまでの変換で得られたホーン節集合の一部を**図5**に示す．図の中でVで始まる変数は，変換途中で導入された変数である．

4. Prolog プログラムへの変換

以上の変換の結果，ホーン節の集合 H が得られる．これをまず Prolog の記法に変換し，その後各述語の引数に入出力のモードを割り当て，それによりリテラルや節の並び換えを行い，元の仕様を満たす解を得ることができる Prolog プログラムPを得る．

ここで，Prolog プログラム上の変換の健全性と，得られたプログラムの元の仕様に対する部分正当性を次のように定義する．

[定義 4.1] 部分正当性，健全性：

ホーン節の集合 H を介して得られた Prolog プログラムPにおいて，:-Φ. の実行が成功する変数が完全にインスタンシエートされたリテラル Φ の集合をPの解集合と呼ぶ．Pの解集合のすべての Φ について，$H \vdash_e \Phi$ ならば，H に対してPが部分正当であると呼ぶ．

また，変換で得られる Prolog プログラムが常に元のホーン節集合に対して部分正当である場合，そのような変換を健全であると呼ぶ．

4.1 プログラムの記法への書き換えと not に対する条件

これまでの変換で得られた，ホーン節を Prolog プ

```
pagenated(empty,empty,Min,Max) ←
        file(empty)∧pages(empty)∧line_number(Min)∧line_number(Max)
pagenated(concatenation(V02,V03),F,Min,Max) ←
        file(F)∧pages(concatenation(V02,T1))∧
        line_number(Min)∧line_number(Max)∧~(F=empty)∧
        file(V01)∧page(V02)∧pages(V03)∧
        separate_a_page(V02,V01,F,Min,Max)∧
        pagenated(V03,V01,Min,Max)
separate_a_page(P,F1,F,Min,Max) ←
        page(P)∧file(F1)∧file(F)∧line_number(Min)∧line_number(Max)∧
        line_number(V11)∧+(V12,Min,1)∧+(V13,Max,1)∧
        most_important(V11,F,V12,V13)∧break(P,F1,F,V11)
break(F1,F2,F,N) ←
        page(F1)∧file(F2)∧file(F)∧line_number(N)∧
        -(V21,N,1)∧first_half_sequence(F1,F,V21)∧
        last_half_sequence(F2,F,N)
most_important(V31,F,V31,V31) ← line_number(V31)∧file(F)
most_important(V42,F,I,J) ←
        line_number(N)∧file(F)∧line_number(I)∧line_number(J)∧
        >(J,I)∧-(V41,J,1)∧line_number(V42)∧
        most_important(V42,F,I,V41)∧more_important(V42,J,F)
most_important(J,F,I,J) ←
        line_number(N)∧file(F)∧line_number(I)∧line_number(J)∧
        >(J,I)∧-(V41,J,1)∧line_number(V42)∧
        most_important(V42,F,I,V41)∧~(more_important(V42,J,F))
more_important(I,J,F) ←
        line_number(I)∧line_number(J)∧file(F)∧
        line_number(V51)∧number_of_blank_lines(V51,F,I)∧
        line_number(V52)∧number_of_blank_lines(V52,F,J)∧
        >(V51,V52)
more_important(I,J,F) ←
        line_number(I)∧line_number(J)∧file(F)∧
        line_number(V51)∧number_of_blank_lines(V51,F,I)∧
        number_of_blank_lines(V51,F,J)∧line_number(V52)∧
        element(V53,F,I)∧level_of_indentation(V52,V53)∧
        line_number(V54)∧element(V55,F,J)∧
        level_of_indentation(V54,V55)∧>(V52,V54)

                .....

number(V0) ← positive_integer(V0)
line_number(V0) ← positive_integer(V0)
pages(empty)
pages(concatenation(V81,V82)) ← page(V81)∧pages(V82)
page(V0) ← file(V0)
line(empty)
line(concatenation(V91,V92)) ← charactor(V91)∧line(V92)
first_char'(V101,concatenation(V101,V102)) ←
        charactor(V101)∧line(concatenation(V101,V102))∧line(V102)
tail(V111,concatenation(V112,V111)) ←
        line(V111)∧line(concatenation(V112,V111))∧charactor(V112)
file(empty)
file(concatenation(V121,V122)) ← line(V121)∧file(V122)

                .....
```

図 5 変換により得られるホーン節集合
Fig. 5 Example of Horn clauses obtained from the logic formulas in Fig. 4.

ログラムの記法に書き換える（変換P.1）．本論文では，Prologとして，DEC 10-Prolog，C-Prolog[12]を対象とする．ここで，否定（～）に対しては，Prologのメタ述語 not を用いることにする．

ここで対象とするPrologは，次の補題が成立することは明らかである．

[補題 4.2]

ホーン節の集合Hから記法の書き換えで得られたPrologプログラムPにおいて，Hに否定（～）を含まず，Pに not が含まれていない場合には，任意の述語記号pと任意の基礎項aに対して，Pで:-p(a). の実行が成功するならば，$H\vdash_c p(a)$であり，Pで有限で失敗[15]するならば，$H\nvdash_c p(a)$である．

Prologのメタ述語 not は，その引数である述語について反駁動作を行い，その際に行った変数に対する代入を捨て，実行結果の成功・失敗の否定のみを返すという性質がある[14]．そのため，次の定理が成立する．

[定理 4.3]

ホーン節HをPrologプログラムの記法に変換して得られたPrologプログラムP（節やリテラルの順序は任意とする）において，Prologのメタ述語 not が，『その引数に出現する変数をすべて基礎項に代入してから実行されなければならない』（以下これを「notに対する条件」と呼ぶ）という条件を満たして実行されれば，PはHに対して部分正当である．

[証明] ある述語記号pと任意の基礎項aに対して，p(a)がPの成功集合に含まれれば$H\vdash_c p(a)$であり，また，:-p(a). の実行がPで有限に失敗すれば，$H\nvdash_c p(a)$であることを，not の中を実行すると1増え not から戻ると1減るような数を not のネスト数とし，:-p(a). を実行する場合に用いられる not のネストの最大数nについての帰納法で証明する．

① $n=0$ の場合，補題 4.2 より明らか．

② nがk以下の場合に成立すると仮定する．Prologのプログラムでは，本体のリテラル not(p(X)) についてXが完全にaに代入され not に対する条件を満たして実行される場合は，:-not(p(a)). の成功失敗のみを調べることになる．Pで :-not(p(a)). の実行が成功するのは，:-p(a). の実行が有限に失敗する場合である．仮定より，このとき$H\nvdash_c p(a)$であり，閉世界仮説より，$H\vdash_c \sim p(a)$である．Pで :-not(p(a)). の実行が有限に失敗するときも，同様に，$H\vdash_c p(a)$である．よってネストの最大数が一つ増えた$k+1$の場合でも成立する．■

また，実際のPrologの実行は，プログラムの節集合中で上から下，一つの節の中で左のリテラルから右に，かつ，depth-first に反駁動作が行われるために，節およびリテラルの順序によっては反駁動作が無限になり停止しない場合があり，可能な限り停止するように実行させる必要がある．

そのため本論文では，not に対する条件を満たし，かつ，可能な限り停止するように，リテラルや節の並べ換えの変換を行うことにする．こうした並べ換え自体は，プログラムの実行により求まる解集合を変化させるが，not に対する条件を満たしていれば健全な変換である．

4.2 入出力の決定

not の条件を満たす必要があるので，変数がどこで代入されるかを決定できるように，Prologの動作を制限した次の仮定を行う．

[仮定 4.4] 関数的動作仮定：

各述語の実行が成功した場合には，各引数は必ず完全に基礎項に代入されているように動作する．

この仮定は，実行が成功する場合には，変数を含む項が変数に代入されることがないことを意味し，各節の各変数についての代入状態を把握することが可能となる．

実は，TELL/NSL からこれまでの変換で得られるPrologプログラムの特徴として，次の補助定理が成立するので，この仮定は自然である．

[補助定理 4.5]

これまでの変換で得られるPrologプログラムが，not の条件を満たして実行されれば，どの節についても実行が成功した場合には，その引数は必ず基礎項に代入されている．

[証明]

Case1：変換 H.4 で導入された以外の節については，頭部の述語の各引数の項は，必ず本体でクラス述語の引数として出現している．クラス述語の性質として，その実行が成功する場合には，必ずそのクラスに含まれる基礎項に代入が行われる．そのため，この節の実行が成功した場合には，その引数は必ず基礎項に代入される．

Case 2：一方，変換 H.4 で導入された節は，常に not の中から呼ばれるので，not の条件からその引数は常に基礎項に代入されて実行される必要がある．ゆえに，実行が終了したときにはその引数は必ず基礎項に代入されている．■

この条件の下で，Prolog プログラムの各述語の引数に対して入出力[13]を設定する．

[定義 4.6] 入出力モード

入出力モードM：述語記号集合×N→{入力，出力}とは，M(p, i) がプログラム中の述語記号 p について，その i 番目の引数のモード値を表すような関数である．ここで，モード値は，入力か出力のいずれかであり，以下のような意味を持つものとする．

- 入力：その引数が，基礎項に代入がなされて評価しなければならないことを表す．
- 出力：その引数には，そのような制限がないことを表す．

関数的動作仮定の下では，出力に割り当てられた引数は，評価が終われば完全に代入されていることを意味する．また，本論文の入出力モードは，評価が始まる前に出力のモード値の引数が代入されていてもかまわない．

入出力モードを設定することにより，変数に対する代入の状況を把握することが可能となり，not の条件を満たして実行できるリテラルの順序を決定することが可能となる．一方，Prolog には入出力の区別がなく，TELL/NSL でも入出力を特に区別せずに記述するが，通常，ソフトウェアの仕様を記述する場合には，入力，出力の区別のないようなシステムはほとんどなく，入出力が定まっている関数的な記述する例が多いので，実行可能な Prolog プログラムを得るためにこのような入出力モードを導入することは自然である．

4.2.1 モードの候補の選択

プログラムに対する入出力モードは，以下のようにして決定され，リテラルの順序を決定する際に用いられる．

① システムが用意している述語記号についてはその引数のモード値は固定されており，その値を用いる．例えば，メタ述語 not はその引数のモード値は入力である．

② まず，仕様記述の最上位のシステムを定義している定義語句（図3では pagenated）の引数については，人間がそのモード値を設定する．というのは，このモード値は，得られたプログラムのどの引数にデータ値を与える必要があるかということに関係しているからである．

③ ユーザの定義した各述語について，まず，クラス述語については 4.2.2 に示す条件 4.8 を満たすモード値に設定する．

④ それ以外の述語については，4.2.3 に示す条件 4.10，4.11 を満たし，モード値が入力になる述語の引数の総数が最小となるようにモード値を設定する．

例えばある一つの述語のある引数のモード値だけが入力と出力で異なるようなモードの候補が存在する場合は，出力の方の候補を代表として残す．

⑤ ④において，どのようなモードを設定しても条件 4.10，4.11 に矛盾する場合は，そのままではモード付けができないとして，人間の判断にまかせる．

4.2.2 クラス述語に関するモードの条件

まず，クラスについてそれが再帰的かどうかを次のように定義する．

[定義 4.7] 再帰的クラス：

クラス述語の定義が再帰になるクラスを再帰的クラスと呼ぶ．再帰的クラスのクラス述語を定義本体に含むクラスも再帰的クラスである．

クラス述語に関する条件を次のように定義する．入出力モードはこれを満たさねばならない．

[条件 4.8] 入出力モードのクラス述語に関する条件：

再帰的なクラスのクラス述語記号の引数のモード値は常に，入力とし，再帰的でないクラスのクラス述語記号の引数は出力とする．

例えば，file は再帰的なクラスなので，その引数のモード値は入力である．

このクラス述語に関する条件は，再帰的なクラスのクラス述語が未代入変数を含む引数で実行された場合に，バックトラックが生じると実行が無限の再帰に陥る可能性があるためである．例えば，

p(X):−α(Y)，　not(q(X, Y)).

については，最悪の場合どんな α(b) となる基礎項 b についても q(a, b) が成功するならば，α が再帰的クラスの述語のとき，not(p(a)) の実行は停止しない．

4.2.3 代入に関するモードの条件

次に，そのモードで実行すると各節のどの変数も必ずその節の中で代入されるという条件と，モードに反しないようなリテラルの順序が決定可能である条件を，次の入出力依存グラフという二部有向グラフを用いて定める．

[定義 4.9] 入出力依存グラフ：

節Cに対する入出力モードMの下での入出力依存グラフ $G_c^m=\langle N_c, A_c^m\rangle$ を次のように定義する．C が Bo :−B1, ⋯, Bn. ($n\geq 0$) であるとすると，N_c はノードの

集合で，次の2種ノードからなる．

L_c：リテラルの集合，$L=\{Bo, B1, \cdots, Bn\}$

V_c：節中に出現するすべての変数の集合．

$N_c=L_c \cup V_c$，$L_c \cap V_c=\phi$ である．

A_c^m は有向アークの集合で，$\langle a, b\rangle \in A_c^m$ はaからbへのアークを表す．A_c^m は以下の条件を満たす最小集合として定義される．

① Bo が $P(t1, \cdots, tk)$ で，ti に x が出現し $M(P, i)$=入力ならば，$\langle Bo, x\rangle \in A_c^m$，$(1 \leq i \leq k$，$x$ は変数)

② Bo が $P(t1, \cdots, tk)$ で，ti に x が出現し $M(P, i)$=出力，かつ，$\langle Bo, x\rangle \notin A_c^m$ ならば，$\langle x, Bo\rangle \in A_c^m$，$(1 \leq i \leq k$，$x$ は変数)

③ Bj が $P(t1, \cdots, tk)$ で，ti に x が出現し $M(P, i)$=入力ならば，$\langle x, B\,j\rangle \in A_c^m$，$(1 \leq i \leq k$，$1 \leq j \leq n$，$x$ は変数)

④ Bj が $P(t1, \cdots, tk)$ で，ti に x が出現し $M(P, i)$=出力，かつ，$\langle x, B\,j\rangle \notin A_c^m$ ならば，$\langle B\,j, x\rangle \in A_c^m$，$(1 \leq i \leq k$，$1 \leq j \leq n$，$x$ は変数)

このグラフは，変数の出現する引数のモード値により構成されるが，頭部のリテラルのみはモード値を反転して用いる．リテラルに入るアークを持つ変数がすべて代入されると，リテラルの評価が可能となり，関数的動作仮定の下で評価が終わるとそのリテラルから出るアークを持つ変数はすべて代入されることを示している．

入出力モードMは，Prolog プログラム中のすべての節Cに対して，次の変数への代入に関する条件が成立しなければならない．

[条件 4.10] 変数の代入に関する条件：

節Cに対するMの下での入出力依存グラフ $G_c^m=\langle N, A\rangle$ において，Cに出現する変数 x はすべて，$\langle B, x\rangle \in A$ となるリテラルBが存在する．

これは，各節のどの変数も，その節の実行が終了する時には，必ず完全に代入されることを表している．なぜならば，関数的動作仮定の下では，変数に入るアークに出力するいずれかのリテラルによりその変数が代入されることは明らかであるので，入るアークのない変数は，その節の実行によって代入が行われないことを表しており，このような変数は関数的動作仮定の下では認められないからである．

また，入出力モードMは，Prolog プログラム中のすべての節Cに対して，次のリテラルの順序の決定可能性に関する条件が成立しなければならない．

[条件 4.11] リテラルの順序の決定可能性に関する条件：

節Cに対するMの下での入出力依存グラフ $G_c^m=\langle N, A\rangle$ について，以下に示す規則を適応していくことでアークを減らしていくとグラフからループを無くすことができる．

規則：Cを Bo←B1, …, Bn. $(n \geq 0)$ とし，Cに出現する変数 x について，

(1) アーク $\langle x, Bo\rangle$ がAに存在するならば除く．

(2) $i, j \geq 0$，$i \neq j$ となるアーク $\langle B\,i, x\rangle$ とアーク $\langle B\,j, x\rangle$ がAに存在するときは，一方のアーク $\langle B\,j, x\rangle$ を除く．

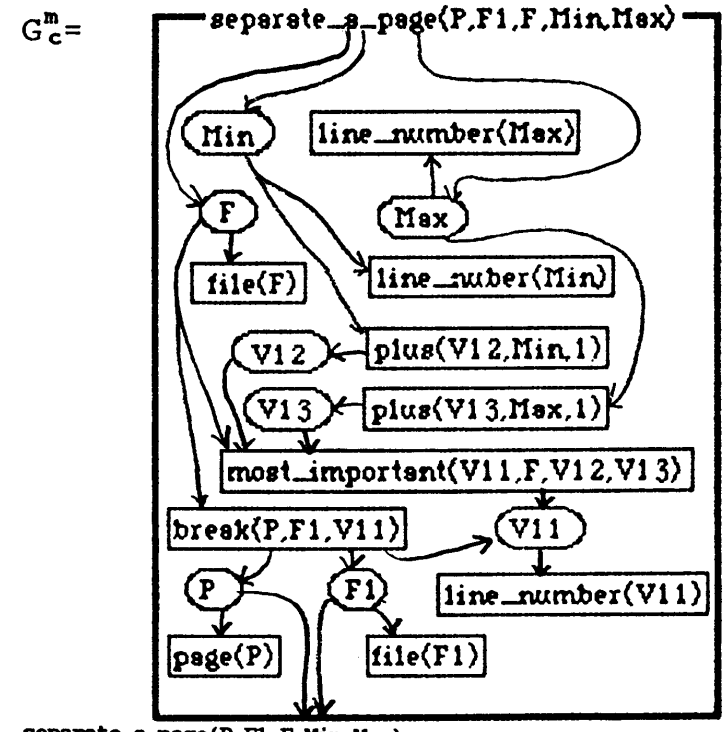
この条件の規則のうち(1)は，頭部のリテラルを含むループがある場合にはそれを除く規則であり，(2)は，変数に最初に代入する本体のリテラルを定める規則であり，選択が複数ある場合にはループがなくなるまですべての可能性を試してみる．この規則に基づいてどのようにアークを除いてもループが残る場合は，リテラルの並べ換えを行う際に，入力のモード値の引数が代入されてから実行する順序付けを求めることができない．なぜならば，入出力依存グラフで，変数 x に対して，$\langle B\,i, x\rangle$ と $\langle x, B\,j\rangle$ のアークがある場合，そのモードでは Bi より Bj が後に実行されるべきであるという入出力依存関係を示している．規則に基づいてどのようにアークを除いてもループが残る場合は，この関係が単一ループを形成している場合であるので，どのようにリテラルを並べ換えても，入力のモード値の引数が代入されてから実行することができないからである．

図6に図5から得られる Prolog プログラムに対する入出力モードと，それに対する入出力依存グラフの例を示す．図6では，入出力モード値について入力を＋で出力を－でそれぞれ表しており，入出力依存グラフは丸が変数のノードを四角がリテラルのノードをそれぞれ表し，矢印で有向アークを表している．頭部のリテラルはグラフを包む外側の大きな四角である．

4.3 リテラルの順序

Prolog では，節本体中のリテラルを左から右の順で実行を行う．入出力モードを利用し，not に対する条件を満たし，可能な限り停止するリテラルの順序を決定し，その順序にリテラルを並び換える．

リテラルの並び換えは，まず，前節で選んだ入出力モードMの下で，各節Cごとに，次の条件 4.12 を満たす本体のリテラルの順序 O_c^m を求める．ここで，$O_c^m(A, B)$ は，AはBより左にあることを示すものと



C= separate_a_page(P,F1,F,Min,Max) ←
page(P)∧file(F1)∧file(F)∧line_number(Min)∧line_number(Max)∧
line_number(V11)∧+'(V12,Min,1)∧+'(V13,Max,1)∧
most_important(V11,F,V12,V13)∧break(P,F1,F,V11)

M=

pagenated(-,+,+,+)	separate_a_page(-,-,+,+,+)
break(-,-,+,+)	most_important(-,+,+,+)
more_important(+,+,+)	number_of_blank_lines(-,+,+)
level_of_indentation(-,+)	number(+)
line_number(+)	pages(+)
page(+) line(+)	first_char(-,+)
tail(-,+)	file(+)
seq_lement(-,+,+)	first_half_sequence(-,+,+)
last_half_sequence(-,+,+)	charactor(-)
positive_integer(+)	add(-,+,+)
sub(-,-,+)	greater(+,+)

図 6 入出力モードと入出力依存グラフの例
Fig. 6 Example of I/O mode and I/O dependency graph.

する．この順序は条件 4.11 を満たすMならば求めることができる．

[条件 4.12] 正しく代入されるリテラルの順序：

節Cに対するモードMの下での入出力依存グラフを〈N, A〉とする．Cが Bo←B1, …, B n. ($n \geq 0$) であるとすると，CのモードMでの本体のリテラルの順序 O_c^m は，次を満たさねばならない．

- 本体のすべてのリテラル B i ($1 \leq i \leq n$) で，$\langle x, Bi \rangle \in A$ となるすべての変数 x は，頭部のリテラル Bo で $\langle Bo, x \rangle$ であるか，または，$\langle Bj, x \rangle \in A$ となりかつ $O_c^m(Bi, Bi)$ となるリテラル B j が存在しなければならない．

この条件は，節の本体の各リテラルにおいて，入力のモード値を持つ引数が代入されてから実行される順序でなければならないことを示している．

すべての節で条件 4.12 を満たす順序でリテラルの実行が行われれば，not の条件を満たすので，条件 4.12 を満たす任意の順序でリテラルを並べ換える変換はすべて健全である．

一方，ほとんどの節には，一つの入出力モードMに対して条件 4.12 を満たす順序が複数個存在する．その中から，なるべく停止解が求まるリテラルの順序を選ぶ必要がある．そこで，再帰のあるリテラルをなるべく左に置くと停止しやすいといった経験則に基づい

て，可能な限り停止して解が求まるリテラルの順序を選び，その順序に並べ換える（変換 P.2）．この変換は健全であることは，以上の議論から明らかである．

4.4 節の順序決定

Prolog では，同一の述語名を頭部に持つ節については，上から下の順で実行を行う．変換 P.2 が行われている Prolog プログラムでは，節の順序を変更しても，本体のリテラルでモード値が入力の引数はすべて代入されてから実行され，not に対する条件は守られるので，元のホーン節に対して部分正当である．しかし，節の順序によっては停止しなくなる場合がある．

そこで，再帰を行わない節が再帰を行う節の前に置かれると停止しやすいといった経験則に基づいて，可能な限り停止して解が求まるような順序に節の並べ換えを行う（変換 P.3）．

図3の TELL/NSL の記述から，これまでの変換をすべて行った結果，図7に示す Prolog プログラムを得た．このプログラムでは，concatenation (A, B) といった記法が試験値の作成に不便なため，便法として

```
pagenated([],[],Min,Max) :-
        line_number(Min),line_number(Max),file(empty),pages(empty).
pagenated([V02|V03],F,Min,Max) :-
        line_number(Min),line_number(Max),file(F),pages([V02|V03]),
        not(F=[]),separate_a_page(V02,V01,F,Min,Max),
        file(V01),page(V02),pagenated(V03,V01,Min,Max),pages(V03).

separate_a_page(P,F1,F,Min,Max) :-
        file(F),line_number(Min),line_number(Min),
        add(V12,Min,1),add(V13,Max,1),most_important(V11,F,V12,V13),
        break(P,F1,F,V11),pages(P),file(F1).

break(F1,F2,F,N) :-
        file(F),line_number(N),sub(V21,N,1),
        first_half_sequence(F1,F,V21),page(F1),
        last_half_sequence(F2,F,N),file(F2).
most_important(V31,F,V31,V31):- line_number(V31),file(F).
most_important(V42,F,I,J):-
        line_number(I),line_number(J),file(F),greater(J,I),
        sub(V41,J,1),most_important(V42,F,I,V41),line_number(V42),
        more_important(V42,J,F).
most_important(J,F,I,J):-
        line_number(I),line_number(J),file(F),greater(J,I),
        sub(V41,J,1),most_important(V42,F,I,V41),line_number(V42),
        not(more_important(V42,J,F)).
more_important(I,J,F):-
        line_number(I),line_number(J),file(F),
        number_of_blank_lines(V51,F,I),line_number(V51),
        number_of_blank_lines(V52,F,J),line_number(V52),
        greater(V51,V52).
more_important(I,J,F):-
        line_number(I),line_number(J),file(F),
        number_of_blank_lines(V51,F,I),line_number(V51),
        number_of_blank_lines(V51,F,J),seq_element(V53,F,I),
        level_of_indentation(V52,V53),line_number(V52),
        seq_element(V55,F,J),level_of_indentation(V54,V55),
        greater(V52,V54).
            . . . .

number(V0) :- positive_integer(V0).
line_number(V0) :- positive_integer(V0).
pages([]).
pages([V81|V82]) :-page(V81),pages(V82).
page(V0) :- file(V0).
line([]).
line([V91|V92]) :- charactor(V91),line(V92).
tail(V111,[V112|V111]):- line(V111),line([V112|V111]),char(V112).
first_char(V101,[V101|V102]):- char(V101),line([V101|V102]),line(V102).
file([]).
file([V121|V122]) :- line(V121),file(V122).
seq_element(V131,[V131|V135],1).
seq_element(V131,[V134|V135],V133) :-
        sub(V136,V133,1), seq_element(V131,V135,V136).

            . . . .
```

図 7 変換により得られる Prolog プログラムの例
Fig. 7 Example of Prolog program obtained by this method.

```
|?- get_file(X),printer(X,"INPUT").

INPUT =[
        "main()",
        "{",
        "    int i;",
        "",
        "    for (i = 0; i<10; ++i)",
        "        printf(i,power(2,i),power(-3,i));",
        "}",
        "",
        "",
        "",
        "power(x,n)",
        "int x, n;",
        "{",
        "    int i, p;",
        "",
        "",
        "    p=1;",
        "    for (i=1;i<=n;++i)",
        "        p=p*x;",
        "    return(p)",
        "}",
        "",
        "",
        ""
]
```

図 8 データ試験用のファイル入力例
Fig. 8 Example of input file for data examination.

C-Prolog のリストの記法 [A|B] などを用いてプログラムを書き換えてある.

5. プロトタイプによる仕様の試験

図2の自然言語で記述された要求仕様に対する TELL/NSL による記述は図3であり，この手法により最終的に得られたプロトタイププログラムは図7である．図7のプログラムにファイル，最大行数，最小行数のテストデータを与え，実行させ試験を行うことにする．ここで，get_file と prine_out という述語を，ファイルの入力と結果の印刷のために用意しておく[16].

図8は今回の試験のためのファイル入力の例であり，ファイルは文字列の列となっており，"" は空白行を表している.

図9は，このファイル入力に対して，最小行数5最大行数8と最小行数5最大行数7の二つの場合で実行した結果である．前者は成功し，結果の出力は文字列の列の列となっており，文字列の列が1ページを表している．これに対して後者は失敗している.

同じ入力ファイルに対して最小／最大行数を変化させて実行させてみると，ファイルから1ページずつ切り出して最後に行が残らなかった場合のみ成功していることが判明した．このことは，図2の要求仕様や図3の記述が完全でなく，記述が不足しており，仕様記述者がこの点を見逃したと考えられる．確かに，ファイルの行数が，最大行数より少ない場合についての記述が述べられてない.

そこで，TELL/NSL の記述で separate a page にファイルの行数が最小行数以下ならばそのまま出力し，最小行数以上ならば改定前と同じ処理をするという条件を追加し，変換を行うと separate a page の部分が図10のように変化した Prolog プログラ

```
|?- Min=5,Max=8,get_file(Input),pagenated(Output,Input,Min,Max),printer(Output).

OUTPUT=[
        [
                "main()",
                "{",
                "    int i;",
                "",
                "    for (i = 0; i<10; ++i)",
                "        printf(i,power(2,i),power(-3,i));",
                "}",
                ""
        ],[
                "",
                "",
                "power(x,n)",
                "int x, n;",
                "{",
                "    int i, p;",
                "",
                ""
        ],[
                "    p=1;",
                "    for (i=1;i<=n;++i)",
                "        p=p*x;",
                "    return(p)",
                "}",
                "",
                "",
                ""
        ]
]

---

|?- Min=5,Max=7,get_file(Input),pagenated(Output,Input,Min,Max),printer(Output).

no.
```

図 9 実行結果例
Fig. 9 The result of execution with faulty specification.

```
pagenated([],[],Min,Max) :-
        line_number(Min),line_number(Max).

pagenated([V02|V03],F,Min,Max) :-
        file(F),line_number(Min),line_number(Max),
        not(F=[]),
        separate_a_page(V02,V01,F,Min,Max),
        pagenated(V03,V01,Min,Max).

/*************************************************************************/

separate_a_page(F,[], F, Min, Max) :-
        file(F),line_number(Min),line_number(Max),pages([]),
        seq_length(V12,F),not(V12 > Max).

separate_a_page(P,F1, F, Min, Max) :-
        file(F),line_number(Min),line_number(Max),
        seq_length(V12,F),V12 > Max,
        add(V13,Min,1),add(V14,Max,1),most_important(V11,F,V13,V14),
        break(P,F1,F,V11),pages(P),file(F1).

/*************************************************************************/

break(F1,F2,F,N) :-
        sub(V21,N,1),
        first_half_sequence(F1,F,V21),
        last_half_sequence(F2,F,N).
```

図 10 プログラムの変更部分
Fig. 10 Program changes after correcting the faulty specification.

ムを得た.

これを先のファイル入力に対して実行してみたところ，最小行数5最大行数8と最小行数5最大行数7のいずれの場合もページングが成功した結果を得た．前者は図9と同じ結果を得たので，後者の結果を図11に示す.

しかしまだ，図9にみられるように，最大行数と最小行数がちょうど空白行が連続している箇所にある場合には，次のページの先頭行が空白行で始まることがある．これは仕様の意図に反すると考えられる．このような現象が発生したのは，ページの先頭の空白行をスキップするような記述が元の要求仕様にないためである.

```
|?- Min=5,Max=7,get_file(Input),pagenated(Output,Input,Min,Max),printer(Output).
OUTPUT=[
        [
                "main()",
                "{",
                "    int i;",
                "",
                "    for (i = 0; i<10; ++i)"
        ],[
                "        printf(i,power(2,i),power(-3,i));",
                "}",
                "",
                "",
                ""
        ],[
                "power(x,n)",
                "int x, n;",
                "{",
                "    int i, p;",
                "",
                ""
        ],[
                "    p=1;",
                "    for (i=1;i<=n;++i)",
                "        p=p*x;",
                "    return(p)",
                "}"
        ],[
                "",
                "",
                ""
        ]
]
```

図 11 変更後の実行結果
Fig. 11 The result of execution of corrected program.

このように，変換で得られたプログラムの実行により試験を行うことで，仕様の不備な点の発見が行え，仕様の理解を助けることができる.

本論文の変換手法では，無限のデータのインスタンスを生成し，それにフィルタをかけて解答を選択することが仕様の本質である記述に対しては，節やリテラルを並べ換えても実行が無限となるので解を得ること

ができない.

また，変換により得られるプログラムは，仕様の無矛盾性を検証する目的には向いていない．しかし，元の仕様が仕様記述者の意図したように記述されているかどうかを，得られたプログラムを実行させることによっていくつかの解を得て試験したり，入力と出力を与えて真となるかどうかを試験する目的には十分であり，こうしたデータ試験の結果により仕様記述者が仕様を修正することが可能となった.

6. むすび

TELL/NSL の仕様記述より，プロトタイプとしての Prolog プログラムを自動的な変化により直接的に得て，仕様に対するデータ試験を行うプロトタイピング手法について述べた.

この手法を用いて，ページのフォーマッティング問題のほか，騎士巡礼問題，電報解析問題，八王妃問題などについても，TELL/NSL の記述から変換により Prolog プログラムを得，テストデータを用いた試験を行い，TELL/NSL で記述された仕様の不備な点を発見でき，仕様を改善することができた.

本論文では，depth-first で直列実行する Prolog の実行制御を考慮し，述語の入出力モードと入出力依存グラフを利用して，リテラルや節の並べ換えを行っているが，並べ換えの変換を他の変換に交換するだけで並列実行が可能な Prolog をターゲットにすることも可能である.

今回の節やリテラルの並べ換えの変換は，経験則に基づいているので，無限の実行に陥ることは完全に防げない．また，得られるプログラムにはいくつかの候補が存在し，最終的には人間によるプログラムの検査やデバッグが必要と思われる.

この変換は，仕様を直接的に変換しているため，仕様が戦略を含んで記述されていない場合は，得られたプログラムは一般的に実行に時間がかかるが，自動的な実行効率の改善は，自動プログラミングを意味することになる.

謝辞 本学，片山卓也教授および片山・米崎研の諸氏に感謝いたします．また，御指導御討論をいただきました，TELL プロジェクトに参加された諸氏に感謝いたします.

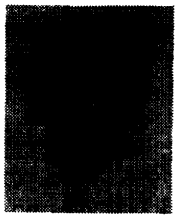
参考文献

1) ソフトウェア要求仕様の検証に使われだすプロトタイピング，日経エレクトロニクス，1983年6月6日号 (1983).
2) Goguen, J. and Mesequer, J.: Rapid Prototyping in the OBJ Executable Specification Language, *ACM SIGSOFT NOTE*, Vol. 7, No. 5, pp. 75-84 (1982).
3) Enomoto, H. and Yonezaki, N. and Saeki, M.: Natural Language Based Software Development System TELL, *Proc. of ECAI-84, Pisa 1984*, pp. 721-731 (1984).
4) Claybrook, B. G.: A Specification Method for the Specifying Data and Procedural Abstructions, *IEEE Trans. S. E.*, Vol. SE-8, No. 5, pp. 449-459 (1982).
5) 榎本，米崎，佐伯，樽松，千葉，滝塚：自然言語に基づくソフトウェア開発システム TELL の概要，第28回情報処理学会全国大会論文集，pp. 449-450 (1984).
6) 榎本，米崎，佐伯：TELL システムにおけるソフトウェア開発，第30回情報処理学会全国大会論文集，pp. 483-484 (1985).
7) 佐伯，米崎，榎本：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. 204-215 (1984).
8) Goldberg, A. and Robson, D.: *Smalltalk-80, The Language and Its Implementation*, Addison-Wesley, Reading (1983).
9) 榎本，米崎，佐伯：TELL/NSL における語句定義構造と意味規則，第30回情報処理学会全国大会論文集，pp. 485-486 (1985).
10) Van Dalen, D.: *Logic and Structure*, Springer-Verlag, Germany (1983).
11) Clark, K. L.: Negation as Failure, Gallaire, H. and Minker, J. (ed.), *Logic and Database*, Plenum, New York (1978).
12) Clocksin, W. F. and Mellish, C. S.: *Programming in Prolog*, Springer-Verlag, Germany (1981).
13) Shapiro, E. Y.: A Subset of Concurrent Prolog and Its Interpreter, Technical Report TR-003, ICOT (1983).
14) Kowalski, R.: *Logic for Problem Solving*, North Holland, Amsterdam (1979).
15) Van Emden, M. H. and Apt, K. R.: Contribution to the Theory of Logic Programming, *JACM*, Vol. 29, No. 3, pp. 841-862 (1982).
16) 市川，蓬萊，佐伯，米崎，榎本：仕様記述言語 TELL/NSL における仕様記述からプロトタイプへの変換，情報処理学会ソフトウエア工学研究会，46-2 (1986).

(昭和61年3月19日受付)
(昭和61年8月27日採録)

市川　至（正会員）

昭和33年生．昭和56年東京工業大学工学部情報工学科卒業．昭和58年同大学院修士課程修了．現在同博士後期課程在学中．プロトタイピング，仕様記述，自然言語などの研究を行い，UNIX ネットワークの発展に努力．電子通信学会会員．



蓬莱　尚幸（正会員）

昭和36年生．昭和59年東京工業大学工学部情報工学科卒業．昭和61年同大学院情報工学専攻修士課程修了．同年富士通(株)入社．現在，国際情報社会科学研究所に勤務．ソフトウェア工学に興味を持つ．

佐伯　元司（正会員）

昭和31年生．昭和53年東京工業大学工学部電気電子工学科卒業．昭和58年同大学院情報工学専攻博士課程修了．工学博士．同年より東京工業大学工学部情報工学科助手．パターン認識，マンマシン・システム，ソフトウェア工学などの研究に従事．

米崎　直樹（正会員）

昭和25年生．昭和47年東京工業大学工学部電気工学科卒業．昭和52年同大学院電子物理工学専攻博士課程修了．工学博士．同年東京工業大学工学部情報工学科助手．昭和58年同大学助教授．昭和60年8月より1年間，英国エディンバラ大学人工知能学科客員研究員．画像処理，プログラミング言語，データベース，ソフトウェア工学などの研究に従事してきた．ソフトウェア工学に対する人工知能応用に興味を持つ．昭和59年度情報処理学会論文賞受賞．電子通信学会，日本ソフトウェア科学会，ACM 各会員．

榎本　肇（正会員）

大正14年生．昭和23年東京工業大学電気工学科卒業．郵政省電波監理局勤務．昭和29年国際電信電話株式会社研究所，昭和42年東京工業大学教授，昭和60年3月東京工業大学名誉教授．現在富士通国際情報社会科学研究所長．この間，短波，マイクロ波のフェージングの研究など電波伝播の研究，放送のサービス領域の研究，衛星通信，電報自動交換の研究など通信システムの研究を行い，情報理論，言語理論，パターン情報処理，ソフトウェア工学など情報工学の基礎分野の研究に従事．電子通信学会，テレビジョン学会，情報処理学会の論文賞各2回，電子通信学会業績賞受賞．