ソフトウエア工学 34−14
（１９８４．２．１０）

ソフトウェア開発システム（ＴＥＬＬ）における自然言語による仕様記述言語（ＮＳＬ）とその応用例

槙本　肇・米崎直樹・佐伯元司
東京工業大学　工学部

１．はじめに

大規模で複雑なソフトウェアを理解しやすいように定義・記述するためには、ソフトウェアを階層的に分割し、上位の層では下位の層を抽象化して参照し記述していく以外に方法はない。これまでに、そのための手法として、Parnas[1] のinformation hidingに基づくモジュール分割、Guttag[2],Goguen [3] らによる代数的抽象化技法、Jackson[4]らによる入出力のデータ構造に基づく手法、Myers[5]の構造化設計法などが研究されてきた。しかし、これらの方法論と密接に関連したソフトウェア開発システムは、今だに実用になっていない。その理由は、ソフトウェアの自然な詳細化をうながす仕様化技法とそれをサポートする仕様記述言語が提供されていなかったことであろう。

また大規模システムでは、その仕様記述量も多いため、仕様を入力として、一貫性などの検証や文書合成といった意味処理までも計算機で行なえるような形式的意味を持った仕様記述形式が望まれる一方、人間にとって理解の容易さも重要な要求である。

我々は、上記のような問題点を解決するための仕様化技法として、自然言語の語彙分割に基づく形式的仕様化技法[6] を提案した。本報告ではその技法に即した仕様記述言語TELL/NSL(Natural language like Specification Language)とそれによる記述例について紹介する。

２．設計方針

自然言語の語彙分割に基づく仕様化技法では、ソフトウェアシステムの内容を自然言語の文で説明し、その文中に出現した単語やひとまとまりの句の意味を、さらに自然言語文で詳細に次々に定義していくことによって仕様を記述する。我々がシステムを自然言語で説明する場合、無意識のうちに語句を選択し、文章を構成するが、これらの語句には通常まとまった概念が対応している。それが分割されたソフトウェアモジュールに対応していると考えることにより、ソフトウェアの分割を意識することなしに、結果として自然な機能分割が得られる。この手法に基づいた理解しやすい仕様を記述するための道具として、TELL/NSLは、以下のような機能を持つものとして設計された。

１）自然言語による記述

本仕様化技法から明らかなように、記述の単位は、１つの自然言語の文である。ただし、それは通常の文より、構文上限定されており、形式論理によって厳密でかつ、ごく自然な意味が割当られている。一方、その構文上の制限が、語彙分割のガイドを与える。TELL/NSLは、使用する自然言語によって、英語版と日本語版の２種類が設計され、その自動翻訳も考えているが、本報告では例題記述に英語版を使用する。

２）階層的モジュール化のための機能

本仕様化技法では、原則として、自然言語の単語・句が１つの分割されたソフトウェア・モジュールに対応する。従って、語句定義の階層構造が結果として、そのまま記述対象ソフトウェアの分割モジュールの親子関係になる。TELL/NSLでは、語句の定義に、通常のプログラム言語流のスコープ・ルールを設け、語句定義の入れ子構造を許している。

３）対象指向に基づくデータの抽象化機能

ソフトウェアシステムの各機能を抽象化していくのと同様に、システムの入出力となるデータ自身を抽象化することは、非常に重要である。通常自然言語では、普通名詞に対応するソフトウェアモジュールには、関数として捕えられるもの以外に、データがある。このようなデータに対応するNSL による種々の記述は、いわゆる抽象データ型をはじめとして、クラス、メタクラス、サブクラスといった概念を持ったモジュールに対応づけられる。

４）記述のためのガイド

本仕様化技法では、どのような語彙を選択してシステムを説明し、その語彙をまたどのように分割していくかを、人間の自然言語での説明における自然な発想にゆだねており、この点を最大の特徴としている。しかし、全く自由な自然言語文を許したのでは、単語あるいは語句をソフトウェアモジュールに対応させることが困難である。そこで、TELL/NSLでは、自然言語の構文を制限することにより、モジュールに対応する語句のカテゴリを限定し、語彙分割の様々なバリエーションを減らし、結果としてモジュール化の指針を与えている。

さらに、文中に出現した語句がどのようなソフトウェアモジュールと対応づくかは、品詞情報や格情報、語句間の修飾関係といった自然言語上の構造情報を利用する。例えば、主語や目的語となっている普通名詞句は、クラスを記述するモジュールに、述部は、主語や目的語、修飾句中で示された固有名詞を入出力パラメータとするようなモジュールに対応づく。ユーザ（部分的にはシステム）は、自然言語上の情報により、どのような定義形式を用いるかを決定することができる。

５）記述概念の直交性

仕様の書きやすさや言語の学習のし易さという面から、記述のための概念が直交性を持っているということは重要である。TELL/NSLでは、基本的には、簡単な自然言語による記述に統一されており、記述概念が言語の各部によって異なってくるようなことはない。

６）並列実行システムのための記述機能

入出力関係だけが、仕様の本質となるようなシステム（関数型システムと呼ぶ）だけでなく、並列実行プログラムや通信プロトコルなどのように、動作

のタイミングが重要であるようなシステムの仕様も記述できなければならない。ここでは、並列実行システムを語句定義の形式で、並列に動作するプロセスの仕様を記述したものと共有資源に対応する名詞に関する動作の同期条件の記述を附加したものとして記述する。

7）多層構造ソフトウェアの記述機能[7]

ソフトウェアの変更のしやすさと汎用性という観点から、通信プロトコルなどに見られるような多層構造（レイヤ・アーキテクチャ）をしたソフトウェアを定義する機能は不可欠である。このような多層構造によるソフトウェアの階層化は、各階層が厳格な規約であらかじめ規定されているという点で、通常の階層的モジュール化とは、異なっており、1つの層内でも、理解しやすい記述を行なうために、階層化が行なわれる。TELL/NSLには、多層構造のソフトウェアを記述するために、上位層と下位層のインターフェースを記述する機能を組み込む。この部分の記述も、同様に自然言語で行なわれる。

3．言語仕様

仕様の本体となる語句の定義には、機能定義、動作定義、クラス定義、動的クラス定義の4種類がある。これらは、各々関数型システム、並列実行システム、後の2つがオブジェクト（データ）のソフトウェアモジュールに対応する。前者2つが1つの語句の意味定義として記述され、クラス定義および動的クラス定義は、普通名詞の意味を、複数の語句の意味の関係記述として記述する。これらの記述は、ある構文規則のもとで、入れ子にすることができ、入れ子構造をした記述に関して、通常のプログラム言語流のスコープ・ルールが適用され、自然言語での適当なvocabularyの不足に対処している。スコープ・ルールのもとでは、モジュール内で定義された語句は、そのモジュールの外では、参照されない。

3．1　機能定義(functional definition)

機能定義の記述例の一部を図1に示す。これは、8クイーン問題の例である。機能定義は、以下のような構成をしている。

```
<機能定義> ::=
        <defining sentence>
          means that
          <spec body>
          {<機能定義>;}*
          {<クラス定義>;}*
          [<lexicon>]
          end <definition id>
```

（（注）　{A}*は、Aの0個以上の列を表わし、[ ]は、省略可能であることを表わす。また {A}+は、Aの1個以上の列を表わすものとする。）

defining sentence は、定義する語句と、その語句の修飾句となる普通名詞との関係を1つの自然言語文の形で宣言したもので、定義される語句の典型的な使用例になっている。機能定義においては、定義される語句は、名詞か形容詞（句）でなければならず、defining sentence 中では、補語として出現する。従って、機能定義を記述する文の動詞は、すべてbe動詞である。定義される補語が、ソフトウェアのモジュール名に対応し、主語や前置詞句といった補語に対する修飾句中の固有名詞が、対応するモジュールの入出力を表わすと解釈される。図1の場合のソフトウェアの仕様としての解釈では、最上位のソフトウェアモジュールeight queens' solution は、入出力としてXを持つ。普通名詞arrangement は、固有名詞Xに対応する対象の属するクラスを表わすと、ソフトウェアでは解釈される。

機能定義は、その内部に詳細定義として、別の機能定義や 3.2で述べるようなクラス定義を含むことができる。図1の例では、機能定義として、checkingが、クラス定義として、arrangement, queen などが含まれている。checkingの機能定義に対応するソフトウェアモジュールの入出力は、q1,

```
Arrangement X is a eight queens' solution
 means that
    1) Eight queens are placed in X .
    2) No queen is checking against
                  any other queen in X .

    Queen q1 is checking against queen q2
                in arrangement X
     means that
       1) q1 and q2 are on the same row in X or
          q1 and q2 are on the same column in X or
          q1 and q2 are on the same diagonal in X .

      Queen q1 and queen q2 is on the same row
                  in arrangement X
       means that
          1) The X-coordinate of the position of
             q1 in X is the X-coordinate of
                   the position of q2 in X .
      end  on the same row ;
      ...

    end  checking ;

   ....

    Arrangement means that
        sequence of  chessboard's square
    end arrangement ;

    Queen means that
     ...

    end  queen ;
   ....

   lexicon
       p is a position of q in X
            := The index of p in X is q;
       c is a X-coordinate of y
            := X-coordinate[y] ;
       c is a Y-coordinate of y
            := Y-coordinate[y] ;
     ...

end eight queens' solution
```

図1　機能定義の記述例

q2，Xに対応するパラメータで、前置詞 against，inは、実引数と仮引数との対応をとるためのマーカの役割りを果すと解釈される。

spec body は、仕様記述の本体、すなわち機能定義のdefining sentence 部で宣言した固有名詞が表わす対象の満たすべき条件を箇条書き形式の文の集合として記述したものである。文の構文規則を以下に示す。

```
<sentence>::= [前置詞  主辞]
                 主辞  be動詞  補語  |
              <sentence>  接続詞 <sentence> |
               [前置詞  主辞]
                 there be動詞  補語  |
              ／論理式／
補語::= 主辞  |  形容詞句
主辞::= 限量詞  名詞句  |  固有名詞  |
        主辞  等位接続詞  主辞
形容詞句::= 形容詞 {前置詞  主辞}*  |
            形容詞句  等位接続詞  形容詞句
名詞句::= 普通名詞 {前置詞  主辞}*  |
          形容詞句  名詞句  |
          名詞句  形容詞句  |
          名詞句  関係代名詞節
関係代名詞節::= 関係代名詞  be動詞  補語  |
        関係代名詞  主辞  be動詞  補語  |
          such that  <sentence>
be動詞::=  is [ not ]  |  are [ not ]
限量詞::=  a | an | the | no |
          every | any other
関係代名詞::= [ 前置詞 ]  which | who  |
              whose  |  [ 前置詞 ] whom
接続詞::= 等位接続詞  |  →  |  ←→
等位接続詞::= and | or
```

各箇条書き形式の文は、条件として同時に成立するものと解釈する。また、1つの文の共通な部分は、1まとめにして、異なる箇所だけを抽出し、さらに小項目の箇条書き形式にすることができる。接続詞→，←→は、夫々含意、等価の意味を表わすシンボル表現である。if then や if and only if のような語句を用いれば完全な自然言語文となるが、複文になってしまい、了解性が低下するためこのようなシンボル表現を許している。

lexicon は、すでに定義されている語句を、この定義内でのみ有効な別の新しい語句として使用したいとき、その対応を与える。図1のlexicon 部の最初の記述は、index という語（図2で定義される）をpositionという語に言い換えて使用することを述べている。新しい語句の宣言は、defining sentenceを用いて行なう。

3．2　クラス定義

クラス定義は、対象の集合を表わす語句として、記述中に出現した普通名詞句を定義する。これは、ソフトウェアシステムが使用するデータ型を定義することと解釈される。ここでは、共通な性質を持った対象をひとまとまりの集合（普通名詞で表わされる）とし、クラスとして定義する。この普通名詞の属性を表わす語句やインスタンスを表わす語句、さらにはこの普通名詞の修飾を受けるようないくつかの語句も、同時に定義する。例えば、普通名詞sequenceの属性を表わすlength（sequenceの長さ）、インスタンスを表わすempty sequence（空列）、concatenation of sequence A ... のように修飾を受けるconcatenation （列の連結）といった語句は、sequenceのクラス定義でその意味定義も行なう。このような語句は、ソフトウェアではクラスに対する演算であると解釈される。クラスを表わす普通名詞の定義は、そのクラスの性質を代数的仕様化における手法と同様に、同時に定義される語句間の関係を機箇条書き形式の文を用いて記述する。クラスの他に、メタクラス、サブクラスといった概念もあり、その間の関係を単純な自然言語（be動詞を用いた文など）で記述される。

クラス定義の記述例の一部を図2に示す。これは、sequenceという普通名詞をクラスとして定義した例で、これ以外に、empty sequence，length，concatenation，indexという語句も同時に定義される。

クラス定義は、以下のような構文をしている。

```
<クラス定義> ::=
        普通名詞句
        [ associated by <words list> ]
        { means that      }
        { is  普通名詞句  }
        <class spec body>
        {<機能定義>;}*
        {<クラス定義>;}*
        [<lexicon>]
        end  <definition id>
  普通名詞句::= [形容詞句]  普通名詞
                {前置詞  固有名詞}*
```

普通名詞単独のときは、この語は、クラスの概念に対応する。形容詞句で修飾されているときは、この句は、普通名詞に対応するクラスのサブクラスに対応する。前置詞によって修飾を受けているとき

```
sequence of x  means that
   1) Empty sequence is a sequence.
   2) The concatenation of sequence A and x B
                          is a sequence.
   3) The length of sequence A is an integer.
   4) The index of x p in sequence A
                          is an integer.
    ...
 satisfy
   1) The length of  empty sequence is 0.
   2) The length of the concatenation of A and p
        is an increment of the length of A.
   3) The index of p in the concatenation of A
      and p is an increment of the length of A.
   4) p is not q →
        the index of p in the concatenation
        of A and q is the index of p in A.
    ...
end  sequence
```

図2　クラス定義の記述例（1）

は、その普通名詞本体が、修飾句中の固有名詞を他の普通名詞で置き換えたときにできる句に対応するクラスのメタクラスに対応する。例えば普通名詞句 sequence of xの定義があったときに、固有名詞xをinstantiate してできる句 sequence of integer や sequence of real は、クラスに対応するとともに、これらは、普通名詞 sequence に対応したメタクラスのインスタンスでもある。

associated wordsは、このクラス定義中で定義される語句のリストで、この定義で規定される対象に対する演算に対応する。

isの部分は、別のクラスに対応する普通名詞句を用いて定義するときに使用する。このとき新しく定義された普通名詞は、もとの普通名詞の性質をそのまま相続する。これは、定義しようとしているクラスが is によって指定されたクラスのサブクラスになっていることを示している。

class spec body は、このモジュールの仕様の本体で、以下のような構文をしている。

```
<class spec body>::=
    { <class defining sentence>; }+
    { locally <class defining sentence>; }*
    satisfy   <spec body> |
    <type expression>
```

class definig sentenceは、そのクラス定義の中で同時に定義される語句を宣言する。この文は、箇条書き形式であることと、定義する語句が補語だけでなく主語として出現してもよいということを除けば、記述形式は、機能定義のdefining sentence とほぼ同じである。この文の前に予約語locally を付けると、定義される語句は、このモジュール内でしか用いられない。

図２では、empty sequence is a sequenceが、class definig sentenceの１つで、これは、empty sequenceがsequenceの１つのインスタンスであることを述べており、これにより名詞句empty sequenceが定義される。ソフトウェアでは、この句は引数を持たないが値域がsequenceであるような関数と解釈される。同様に、concatenation は、sequence×x→sequenceであるような関数と解釈される。

spec body は、演算と対応づけられる語句の性質を記述する他に、このクラスがあるメタクラスのインスタンスになっているというようなことも記述できる。定義するクラスがあるクラスのサブクラスになっているときは、新しく追加する語句の宣言やその性質記述だけを書けばよい。またメタクラスのインスタンスになっているときは、それを宣言するだけでよく、他の記述は不要である。

type expression は、データ構造を陽に定義したい場合に用いる。これは、データ型という考えで、通常のプログラム言語で使用されている基底型、写像型、直積型、直和型、列挙型、部分範囲型といったデータ型がシンボル表現で定義できるようになっている。type expression の構造に応じて、演算とその意味が自動的に定義される。図３にtype expression を用いて定義した例を示す。クラス chessboard's squareは１～８までの整数のタップルで、第１要素を取り出す演算がx-coordinate、第２要素がy-coordinateである。

```
Queen means that
      [1..8]
end  queen ;

Chessboard's square means that
      (X-coordinate: [1..8] ,
            Y-coordinate: [1..8])
end  chessboard's square
```

図３　クラス定義の記述例（２）

３．３　動作定義

並列処理システムの仕様記述は、共有資源と並列に動くことが可能な複数の動作の定義とに分離して記述する。動作定義は、プロセスの仕様を記述するモジュールと解釈され、共有資源となるクラスの定義を行なうモジュールと解釈されるのが、次節の動的クラス定義である。定義では、固有名詞が表わす対象の静的関係だけでなく、動作に関する排他制御や同期条件といった動的な性質も記述しなければならない。

動作定義の記述例の一部を図４に示す。この例は、ｎ人のproducerとｍ人のconsumerが、容量が１つしかないbufferを共有する問題である。動作定義の構文は、以下のようになっている。

```
<動作定義> ::=
        <dynamic defining sentence>
        means that
        [<config part>]
        [<state pred part>]
        timing
        <dynamic spec body>
        {<動作定義>;}*
        {<機能定義>;}*
        {<動的クラス定義>;}*
        {<クラス定義>  ;}*
        [<dynamic lexicon>]
        [<interface section>]
        end  <definition id>
```

dynamic defining sentence は、定義する語句とその語句の記述形態を宣言するための文である。defining sentence に対し、be動詞だけでなく、一般動詞も使用できることと、主語に it,he,she,Iなどの代名詞が使用できるという点で拡張されている。一般動詞もしくは補語となっている固有名詞は、モジュールとして、目的語や前置詞句中の固有名詞が、モジュールの入出力と解釈される。図４の例では、n-producer and m-consumer problem.

produce, consume, generate などがモジュール名で、xは、 generate の入出力パラメータと解釈される。また、モジュールに解釈される動詞の名詞形もモジュールとして解釈される。例えば、produceの名詞形producerも同じモジュールを表わす。dynamic defining sentence の主語は、機能定義のそれと違って意味は持っていない。

config part は、その語句定義内の動作定義や動

```
N-producer and m-consumer problem
                is the system of
   means that
   Configuration
      1) There are n producers.
      2) There are m consumers.
   Timing
      1) Initially every consumer begins
                              to consume.
      2) Initially every producer begins
                              to produce.
      3) Initially buffer is empty.
      4) Every producer which finishes producing
                    will begin to produce.
      5) Every consumer which finishes consuming
                    will begin to consume.

   He produces means that
     Timing
       1) Initially he begins to generate .
       2) He finishes generating data x
            → he will begin
                 to write data x to buffer.
       3) He finishes writing to buffer
            → he will finish producing.

       He generates data x means that
         ...
       end generate ;

   end produce ;

   He consumes means that
     ...

   end consume ;

   Buffer means that
     ...

   end buffer ;
end n-producer and m-consumer problem
```

図４　動作定義の記述例

的クラス定義で定義される動作やクラスのインスタンスがいくつ存在するかを宣言する。また複数個存在したときに、どのような識別子をつけるかを宣言する。この部分の記述は、箇条書き形式の、thereを形式主語とする自然言語文で行なわれる。<config part> が省略されたときは、この定義内では、その動作、クラスのインスタンスは、唯一で、識別子は、定義された単語そのものとなる。図４の例ではproduce で定義される動作はｎ個、consumeはｍ個、クラスbufferのインスタンスは１個存在する。

state pred part は、この動作の内部進行状態を表わす述語と、それに対応する語句を宣言する。図４では、省略されているが、図５のbuffer定義では、動詞begin と形容詞waiting が同じ述語atに対応づけられている。これは、２つの語が同じ意味であることを述べている。

仕様の本体dynamic spec body は、dynamic defining sentence で宣言した固有名詞を含む箇条書き形式の文dynamic sentenceの集合として記述される。この文は、1)内部の動作の相互関係　2)動き始めたときに、どのような事柄が成立しているか。（動作の初期状態）　3)どんな事柄が成立したとき、その動作は終了するか。（最終状態）を記述している。

dynamic sentenceは、先に定義したsentenceに対し、1)文頭にinitially, finally、文頭もしくは文末に in the next time といった副詞（句）を付けてもよい。 2) be動詞以外に一般動詞や助動詞willも使用できる。 3) 動名詞や to 不定詞も使用できる。4)接続詞として、until,before, after といった時を表わす接続詞が使用できる。といった点で拡張されている。また、記述に際しては、一般動詞を含む文の主語は、dynamic definig sentenceで使用されたものと同じか、動作定義で定義された動詞の名詞形でなければならない。副詞としてinitiallyが付加されると、その文は、初期状態の記述を表わし、finally は、最終状態の記述を表わす。図４の例では、n-producer and m-consumer problem の初期状態は、bufferが空(empty) であり、producer, consumerはすべて動き始めている状態である(begin to produce, begin to consume)。

dynamic lexiconの役割は、3.1 、3.2 節のそれと全く同じで、動詞も再定義できるという点で拡張されているだけである。

interface section は多層アーキテクチャ式のソフトウェアシステムを記述するためのもので、語句定義を下位レイヤのどの語句定義がサポートするかを記述する。この宣言の後に、両語句定義間の状態の対応をこれまでと同様、箇条書き形式の自然言語文を用いて記述する。使用できる文の形式は、dynamic sentenceである。

３．４　動的クラス定義

動的クラス定義は、ソフトウェアでは共有資源を定義することと解釈される。3.2 節のクラス定義に対して、対象の動的な性質が記述できるように拡張されている。記述に使用する自然言語文は、すべて3.3 節で延べたような拡張が施されている点を除くと、クラス定義の場合と全く同じである。図５にクラスbufferを定義した例を示す。

```
<動的クラス定義>::=
        普通名詞句
        [ associated by <words list> ]
        { means that      }
        { is  普通名詞句  }
        <dynamic class spec body>
        {<動作定義>;}*
        {<機能定義; }*
        {<動的クラス定義>;}*
        {<クラス定義> ; }*
        [<dynamic lexicon>]
        [<interface section>]
        end  <definition id>
```

dynamic class spec body が、仕様の本体で、機能定義の記述<spec body> と排他制御や同期条件などの動作に関する記述<dynamic spec body> とから

成る。前者は、クラス定義のそれと、後者は、動作定義のそれと全く同じである。図５では、bufferとともに動詞read, write,普通名詞句 empty buffer という語句が同時に定義される。これらはクラスbufferの演算と解釈される。予約語satisfy に引き続き記述されている式は、これら３つの語句の機能に関する記述である。timing以下が動作に関する記述である。また、定義本体のtype expression を用い、そのtype上で用いられる演算の動作タイミングを附加して記述することも可能である。

```
<dynamic class spec body>::=
        { <dynamic class
              defining sentence>; }+
        { locally  <dynamic class
              defining sentence>; }*
        satisfy    <spec body>
        [[ <state pred part> ]
          timing <dynamic spec body>]  |
         <type expression>
        [[ <state pred part> ]
          timing  <dynamic spec body>]
```

４．おわりに

本報告では、自然言語に基づく仕様記述言語NSLの仕様について述べた。実際に、何人かの人に、toy program 以外にテキスト・エディタ、通信プロトコル、NSL のパーザなどの仕様をNSL で記述してもらった。その結果、ソフトウェアの上位レベルの概念は、容易に仕様化ができ、記述された仕様も他人にも読みやすいものとなった。しかし下位レベルでは、抽象化レベルとしては低いが、記述に用いる概念はより数学的にprimitive なものになり、自然言語では表現が長く煩雑になる。このためNSL は、数学的シンボル表現による記述も併用できるように考慮されている。　仕様を記述するにあたり、我々は、仕様化しようとしているソフトウェアの概念をただちに形式的に記述できなくても、それに対して漠然としたイメージだけは持っている。そのイメージから言語表現を導き、それをNSL で許される文として記述していくための対話形式のツールの開発が、仕様化を容易にするうえで特に重要であろう。

謝辞：本研究に関し、熱心に議論して頂きましたＫＤＤ研究所端末装置研究室の樽松明室長、千葉和彦氏、滝塚孝志氏、および本学、榎本・米崎研究室の諸氏に感謝致します。

参考文献

[1] Parnas,D.L. : On the Criteria to Be Used in Decomposing Systems into Modules, CACM Vol.15, No.12, 1972.
[2] Guttag,J.V. et.al. : Abstract Data Types and Software Validation, CACM, Vol.21, No.12, 1978.
[3] Goguen,J.A. : An Initial Algebra Approach to the Specification of Reliable Software, Boston M.A., 1979.
[4] Jackon,M.A. : Principles of Program Design Academic Press, 1975.
[5] Myers,G.J. : Realiable Software through Composit Design, Manson/Charter Pub., 1975.
[6] 佐伯・米崎・榎本：形式的仕様記述としての自然言語、第２４回プログラミングシンポジウム、1983.
[7] 榎本・米崎・佐伯・荒俣：TELL/NSLによるレイヤ・プロトコルの仕様記述とその検証、第28回情報処理学会全国大会、1984

```
Buffer means that
  1)  I write data x to buffer B
      :=  ¡write[B,x]
              : (buffer,data) → buffer ;
  2)  I read data arg2[read[B]]
              from buffer B
      :=  ¡read[B]
              : buffer → (buffer,data) ;
  3)  Empty buffer is a buffer
      :=  ∅ ;
satisfy
  1) /¡read(¡write(∅,α)) = (∅,α)/.

Timing
  State predicate
      at :: begin [v], waiting  [adj] ;
      in(1) :: l°ing  [adj] ;
      after :: finish  [v] ;
  1) Producer i begins to write data x to B
        and B is not empty
      → producer i is waiting
        to write data x to B until B is empty .
  2) A producer begins to write to B
      and B is empty →
     one of producers which begin to write to B
                    is writing to B.
  3) Consumer i begins to read from B
        and B is empty
     → consumer i is waiting to read from B
              until B is not empty.
  4) A consumer begins to read from B
      and B is not empty →
     one of consumers which begin to read from B
                    is reading from B.
  5) A producer is writing to B
       → there is no consumer
         which is reading from B .
  6) A consumer is reading from B
       → there is no producer
         which is writing to B .
end  buffer ;
```

図５　動的クラス定義の記述例