ソフトウェア基礎論　３３－３
プログラミング言語　２３－３
（1989. 12. 8）

言語NAIVEによる並列処理システムの記述

日野 克重

富士通㈱ 情報システム事業本部

あらまし　ソフトウェア危機が言われて久しいが，いまだ解決から遠い。これに対するアプローチとして，「実行可能な仕様記述言語」が有望であると思う。そこで筆者らは，実行可能な仕様記述言語の一つとして言語NAIVE を開発した。当言語は，①自然語に近い表現が可能である，②計算機の上で実行可能である，③動的かつ複雑なシステムの記述が簡単にできる，④内包論理に基礎をおくなどの特徴をもつ。すでにその処理系も実現されており，Ｃ言語で数Ｋ行規模でかつ並行処理もおこなう動的プログラムを約３０個自動的に導出し，かつ実行した実績がある。その記述・実行実験の結果から，当言語を使用することにより，従来に比して格段に速くかつ正確にソフトウェアが開発できることがわかっている。本稿では，当言語について特にその並列処理言語としての側面に焦点を当てつつ述べる。

How can Concurrent Systems be described and executed in the Language NAIVE ?

Katsushige HINO

COMPUTER SYSTEMS GROUP, FUJITSU LIMITED

140 MIYAMOTO, NUMAZU-SHI SHIZUOKA, 410-03 JAPAN

Abstract　Long time has passed since software crisis was warned of. The author thinks that the approach of specification language which is close to natural language and also executable on computer is promosing.

We have developed an executable specification language:NAIVE. The language NAIVE has following characteristics: ①NAIVE is close to natural language in description manner, ②NAIVE is executable on computer, ③NAIVE can describe dynamic and concurrent systems easily, ④NAIVE has its logical foundation in Intensional Logic.

Until now, about 30 example programs have been derived automatically through NAIVE system, and executed correctly. Through the experiment, we confirmed that software development can be made much easier work, by using the language NAIVE.

1．はじめに

ソフトウェア危機が言われて久しいが，いまだ解決から遠い。これまでにも，古くはジャクソン法やプログラム検証から最近ではCASEツールや知識工学の応用に至るまで，さまざまの技法・ツールが提案されているが，いまだソフトウェア産業界の要求を十分満足させるものはない。

この点で，筆者は，より人間のレベルに近い言葉でソフトウェアを記述でき，しかもそれが計算機上でも実行可能な「仕様記述言語」のアプローチが有望であると考える。特に，以下に示すような条件を満足する仕様記述言語が実現されれば，ソフトウェア開発のおかれている状況は大きく好転することが期待できる：

①自然な表現：自然語（もっと強くいえば日常語）に近い言葉や概念で仕様を表現できること。特定のモデル概念や計算機に関する概念は不要であること。ただし，問題自体が数学的なものであるときには，数式も一種の自然語と考えてよいだろう。

②実行可能性：実行可能性の裏付けもあること。また，ささいな開発「支援」ツールでないのがよい。

③動的かつ複雑な仕様の記述：オペレーティングシステムやシミュレーションに代表されるような動的かつ複雑な問題を表現できること。

④論理的基礎：論理的基礎がしっかりしていること。ただ単に，高級な表現ができるというだけではいけない。

本稿で述べる言語NAIVE は，以上のようなことを念頭において開発された言語であり，すでに，本言語で書いて数十行規模の仕様記述から，C言語で１K行～２K行になるプログラムをいくつか自動的に導出し，かつ実行した実績がある。

以降，この言語NAIVE について，その並行処理言語としての側面に重点をおきながら述べる。したがって，当言語の実現方式などについては，必要最小限に言及するに止める

2．言語NAIVE の概要

2．1　言語NAIVE の特徴

言語NAIVE の特徴を以下に列挙する。

(1) 言語NAIVE は，自然な日常語のレベルで問題をとらえ，それを表現できる言語であろうとしている。したがって，たとえば，オブジェクト，メッセージ，あるいはその他の特定のモデル表現概念は前提としない。

(2) いわゆる手続き的表現と宣言的表現を融合した言語である。手続き的表現を導入したのは，オペレーティングシステムに代表される動的システムを記述するとき，通常の時間論理ではそれを自然に表現できない；むしろ人間の思考様式にそぐわない表現になってしまうと考えたからである。なお，当言語における手続き的表現は，後述するようなある論理体系の中にきちんと位置づけられている。

(3) WHEN（待機），WHILE（継続）などの接続詞が導入されている。これにより，動的あるいは並行システムの作成時に不可欠なプロセス間の同期・通信処理が簡潔に記述できるようになっている（厳密には通信を意識する必要もない）。

(4) 限定詞all,some,many がある。また，これらの限定詞は，命題式（事実の判定式）と行為式（世界状態の変更式）のいずれにも使うことができる。これにより，従来型言語では繰り返し処理として記述せざるを得なった集合操作が，簡潔に記述できる。

(5) 指示詞の機能を果たす限定詞the がある。

(6) 「～を作り出す（～がある）」および「～をなくす（～がない）」という概念を表現する実在詞beおよびnil がある。

(7) by,to,at,in,ofなどの基本的な関係詞があり，その意味が言語の中に組み込まれている。

(8) いわば能動的動作主体というものを記述することができる。これはシミュレーション記述などで有用である。

(9) レジスタ，データ構造，キュー，プロセス，モジュール，および割り込みなどの従来のプログラミング上の概念を，プログラマには意識させない言語である。

(10) 言語NAIVE で書かれた仕様記述は，その処理系によって自動的に論理合成され，目的言語（たとえばC言語）のプログラムに展開される。その後，ただちに実行させてみることもできる。したがって，従来型言語で開発したときに経験する論理ミスによるシステムの異常終了やコンパイルエラーなどは，一切発生する心配がない。

(11) 相互排除の対象および区間の検出ならびにその他のソフトウェア技術上の諸問題が，プログラムの文面を解析することにより，自動的に解決できる潜在的可能性をもった言語である。こうした可能性は，言語NAIVE の語彙（基本詞）が，自然言語と同様に，「意味」を内包させられ得るものであることから来ている。

(12) 内包論理（モンタギュー意味論）に論理的基礎をおいた言語である。ただし，その論理体系は，

手続き表現の導入のため少し変更が加えられている。

2．2　言語NAIVE による仕様記述法

言語NAIVE には，自然語のうちのきわめて基本的な語彙が，その意味とともに基本詞としてとりいれられている。言語NAIVE で書くということは，自然語で書くということに近い。したがって，自然語でものごとを記述するときに特別の記述法がないのと同様に，言語NAIVE による特別の仕様記述法というものはない。言語NAIVE がもつ基本詞をうまく使って適切に表現するというだけである。

ここで，言語NAIVE による仕様記述の実例を挙げておく。いまだ構文と意味を述べていないので読みにくいだろうが，おおざっぱな感じをつかんでいただきたい。ただ，あらっぽく言って，“Ａ＝Ｂ”は，「ＡはＢである」または「ＡをＢであるようにする」（どちらであるかは，それがIFやWHENなどの接続詞の条件部にあるか否かという文脈による），“名詞Ａ..名詞Ｂ”は「ＡのＢ」，“修飾句Ａ．名詞Ｂ”は「ＡであるＢ」，“all.名詞Ａ”は「すべてのＡ」，”some．名詞Ａ”は「あるＡ」”，”many< 数>.名詞Ａ”は「～個以上のＡ」と読む。また“名詞Ａ:x”は，「そのＡ，それをｘとすると，それは」というふうに読めばよい。

図－１は，酒屋の在庫管理システムの言語NAIVE での仕様記述である。このプログラムの読みを以下に示す。

まず，このプログラム全体は次のような構成になっている：

```
'倉庫生成 :s' ≡　Ｐ１
'コンテナ生成 :c' ≡　Ｐ２
'酒ビン収納 :c :m :n'≡　Ｐ３
'コンテナ搬入 :c :s' ≡　Ｐ４
'顧客注文 :k :m :n'≡　Ｐ５
'空コンテナ自動搬出' ≡　Ｐ６
```

これは，次のように読むべきである：「酒屋の在庫管理システムというものは，（引用符でくくられた）６つのコマンドが処理できなければならない。そして，それらのコマンドの意味は，それぞれＰ１，Ｐ２，・・・，Ｐ６である。」

なお，引用符の中の文字列は，操作者が後に投入すべきコマンド文字列自体であり，“：変数”は，そのコマンドのパラメタを示している。

次に，各コマンドの記述部分については，たとえば，倉庫生成コマンドのところは，「倉庫生成コマンドというのは，ｓ で指定された名前をもつ倉庫を１つ生成する（be）ものである。」と読む。コンテナ生成コマンドも同様である。

また，酒ビン収納コマンドのところは，「酒ビン収納コマンドというのは，指定された銘柄の（“of-the．銘柄:m　”）酒ビンを，指定された本数（“many<n>”）だけ，指定されたコンテナの中に収納する（“in-the．コンテナ:c”）ものである。」というぐあいに読む。

さらに，顧客注文のところは，「顧客注文コマンドというのは，指定された銘柄の酒ビンがどこかの倉庫の中に存在するのを待って（“WHEN some.(of-the.銘柄:m).ビン:b=in-some．倉庫:s”），それを指定された顧客に渡し（“ the．ビン:b=in-the.客:k”），そのビンはその倉庫の中にはなくなる（“the.ビン:b=not-in-the.倉庫:s”）ということを，指定された本数分だけ繰り返す（“REPEAT n”）ものである。」と読む。

最後の空コンテナ自動搬出コマンドは，「当コマ

〔問題：酒屋の在庫管理システム〕
酒屋の在庫を管理するシステムを作る。このシステムは，以下の状況に対処できるも　のでなければならない。
－各種銘柄の酒ビンが任意本数入ったコンテナが時々倉庫に搬入される。
－顧客からの注文に応じて，指定された銘柄および本数の酒ビンを，当顧客に届ける。ただし，当酒店に，その銘柄の在庫が不足していた場合は，在庫分だけとりあえず届けて，残りは，新たな搬入があったとき，敏速に追加送付する。
－コンテナが空になると，ただちに倉庫から搬出する。

〔言語NAIVE での記述〕

```
'倉庫生成 :s'≡
    DO(the.倉庫:s=be)
'コンテナ生成 :c'≡
    DO(the.コンテナ:c=be)
'酒ビン収納 :c:m:n'≡
    DO
      many<n>. ビン:b=be
       WHERE
         the.ビン:b=of-the.銘柄:m  and
         the.ビン:b=in-the.コンテナ:c
    END
'コンテナ搬入 :c:s'≡
    DO(the.コンテナ:c=in-the.倉庫:s)
'顧客注文 :k:m:n'≡
   DO
     REPEAT n
       WHEN some.(of-the.銘柄:m).ビン:b
                    =in-some．倉庫:s
          (the.ビン:b=in-the.客:k  and
           the.ビン:b=not-in-the.倉庫:s)
   END
'空コンテナ搬出' ≡
   DO
     WHILE  true
       WHEN  for some.(in-some.倉庫:s).コンテナ:c
                  all. ビン=not-in-the.コンテナ:c
          the.コンテナ:c=not-in-the.倉庫:s
   END
```

図－１　酒屋の在庫管理システムのNAIVE での記述

ンドが投入された後は永久に("WHILE true") あることを行い続ける。つまり，どこかの倉庫の中のいずれかのコンテナの中にビンが1本もなくなったら(" WHEN for some.(in-some. 倉庫:s).コンテナ:call.ビン=not-in-the.コンテナ:c")，そのコンテナをその倉庫から搬出する("the.コンテナ:c=not-in-the. 倉庫:s")」と読む。

この仕様記述をNAIVE の処理系に投入すれば，自動的に実行可能なコードが生成され，その後コマンドを投入すれば実行される。

この例題プログラムの場合，特に重要なはたらきをしているコトバは，接続詞WHENと関係詞inである。接続詞WHENは，そこに書かれた条件が満足されない場合は，それが満足されるまで待つ（待機する）という意味をもっている。また，関係詞inは，一種の推移性をもつ関係としてはたらいている(in が推移性をもつ関係であることは，NAIVE の中に組み込まれている）。たとえば，あるビンをあるコンテナの中に収納したのち，そのコンテナをある倉庫の中に搬入すると，そのビンも当然その倉庫の中にあるようになるはずだとか，あるビンをどこかの倉庫から取り出して顧客に渡せば，そのビンは当然，倉庫内のどのコンテナの中にもないはずだということをNAIVE はこの関係詞inの推移性から推論する。

特に，本例題の場合，「指定された銘柄の在庫が不足していた場合は，その銘柄がその後搬入されたとき敏速に追加送付する」という条件があるが，通常の言語でこれを行おうとした場合は「不足が生じたら，その旨を記録しておき，新たにコンテナが搬入されると，その中に入っている酒ビンの銘柄を調べ，その銘柄を追加送付すべき顧客があれば，不足本数だけ追加送付する」というような記述が必要であるが，NAIVE ではその必要はない。それにもかかわらず，おのずとその処理が行われるのは，この接続詞WHENと関係詞inのはたらきに負っている。詳細には次のようなメカニズムでそれが遂行される：まず，コンテナが倉庫の中に搬入されると，そのコンテナの中の酒ビンもすべて倉庫に入る（入ったことになる）。一方，ある銘柄の追加送付が必要な顧客がもしあれば，ある顧客注文のコマンドプロセスが，"WHEN some.(of-the. 銘柄:m).ビン:b=in-some. 倉庫:s" で待ち状態になっているはずである(NAIVEシステムでは1つのコマンドは1つのプロセスとして実行される）。いま新たに倉庫の中に入った酒ビンの中にその銘柄のものがあれば，このWHEN文の待機条件が満足されるので，その次に書かれている"the.ビン:b=in-the.客:kが実行される。これはその銘柄のビンを追加送付することに他ならない。

このようにNAIVE では，コトバのもつ意味を有機的に活用することにより，先に示したようなさりげない記述だけで，結構複雑なこの酒屋の在庫管理システムが表現できることになる。

3．論理体系

言語NAIVE は，いわば「自然語の論理」に基礎をおいている。正確には，内包論理（モンタギュー意味論）に少し変更をくわえた論理体系（以降これを論理NAIVE とよぶ）に依拠している。この変更は，基本的に，言語NAIVE に手続き表現を導入したことからくる。ここでは，この論理体系の細部にわたる形式的記述は避け，通常の内包論理に比して特徴的な点に重点をおきながら簡単に述べる（〔注〕内包論理の詳細についてはたとえば，参考文献Dowty〔2〕あるいはGallin〔5〕を参照されたい）。

3．1　構文論についての特徴

(1)タイプ：基本タイプとして，タイプe(個体),タイプt(命題）にくわえて，タイプp(行為）がある。タイプp 式とは直感的にはプログラムのことである。また，各タイプのもとにサブタイプを任意に定義できる（これは手続き表現の導入とは無関係である）。サブタイプとは直感的には普通名詞のことである。たとえば，自然語における「人間x」という表現の場合，「人間」は変数x のサブタイプを示しているといえる。なお，論理NAIVE に組み込みのサブタイプとしては，" 物(THING)","数(NUMBER)"," 関数(FUNC)" などがある。それぞれタイプe,タイプc,タイプ<cc>と等しい（ここでc は自然数のタイプのこととする）。

(2)原始記号：≡，λ，↑（内包），↓（外延）がある。これは内包論理の最も本質的な部分であり，論理NAIVE においても変わらない。

(3)整合式：整合式の形成規則も通常と変わらないが，参照の便のため，示しておく。

論理NAIVE におけるタイプa の整合式の集合W_a は次のように帰納的に定義される。

① すべてのタイプa の変数x_a について，
$x_a \in W_a$。

② すべてのタイプa の定数C_a について，
$C_a \in W_a$。

③ $A \in W_{<a,b>}$ かつ $B \in W_a$ ならば，
〔AB〕$\in W_b$。

④ $A \in W_b$ でかつx がタイプa の変数ならば，
$\lambda x.\ A \in W_{<a,b>}$。

⑤ A，B∈W_a ならば，〔A≡B〕∈W_t 。

⑥ A∈W_a ならば，↑A∈$W_{<sa>}$。

⑦ A∈$W_{<sa>}$ ならば，↓A∈W_a 。

3．2 意味論についての特徴

(1)フレーム：指標集合Ｉは自然数と同じ離散的全順序構造をもつものとしている（直感的には時刻集合）。そして，論理NAIVE では整合式の解釈はすべて（時点でなく）時区間で行う。なお，タイプｐ式（行為式）の解釈領域はタイプｔ式（命題式）の解釈領域と同じく，真値（true:1）と偽値（false:0）の２要素からなる集合である。

(2)解釈：整合式の解釈を時区間で行うことから，タイプe（個体）やタイプt（命題）の整合式の時区間での解釈にどんな直感的意味があるかの疑念が生じるであろう。本論理体系では，タイプｅ式とタイプｔ式の '時区間' での解釈には固有の意味はなく，その区間の開始時点での解釈と一致するものとした。

(3)変数への割り当て：通常とほぼ同じ。ただし，REPEATやWHILE などで作られる入れ子構造と変数の関係を形式的に示すために，割り当て関数αに関する部分関数の概念を導入する必要がある。

3．3 論理NAIVE 固有の定数とその意味

論理NAIVE には，定数として，◎<tp>，⇨<p<pp>>，IF<t<p<pp>>>,REPEAT <c<pp>> ,WHEN <t<pp>> ，WHILE <t<pp>> ，☆<e<sp>> ，Π<e<t<tt>>>，Σ<e<t<tt>>>，Θ<c<e<t<tt>>>>，$\underline{\Pi}$<e<t<pp>>>，$\underline{\Sigma}$<e<t<pp>>>，$\underline{\Theta}$<c<e<t<pp>>>>，and<p<pp>> ，or<p<pp>> ，# <e<tc>> ,not<<e*t><e*t>>,be <et>，nil <et> ，by<e<et>> ，to<e<et>> ，in<e<et>> ，at<e<et>> ，of <e<et>> ,sum <e<t<cc>>>が導入されている。 以下，これらの直感的意味を示す。また，その形式的意味が不明と思われるものについてはそれを付す。なお，これら以外の常識的な定数（たとえば∧や∨や→などの論理演算記号や算術演算記号や真偽定数など）はもちろん通常の意味をもつものとして含んでいる。 注）ここでタイプ記法<a*b>は<a<a..<ab>..>の略記とする（以降同じ）。

(1) ◎ ：命題式から行為式をつくりだす定数である。たとえば，信号s を青く(blue)するという行為式は「◎(blue(s))」と書く。形式的意味を次に示す。

$$Vi,j,\alpha\,〔◎A_t〕=1 \quad iff$$
$$Vj,j,\alpha\,〔A_t〕=1$$

（ここで，Ｖは，整合式に値を対応させる付値関数であり，αは変数に値を割り当てる割り当て関数である。直感的には" Vi,j,α〔W〕＝a "は「Wの時区間<i,j> での意味はａである」と読む。以降同じ。）

(2) ⇨ ：通常のプログラム言語における連接を表現する。たとえば，Ａを行って次にＢを行うという行為式は「A⇨B」と書く。形式的意味は次の通り。

$$Vi,j,\alpha\,〔A_p⇨B_p〕=1 \quad iff$$
$$i<j'<i'<j \quad なる$$
$$j',\ i'\in I \quad があって,$$
$$Vi,j',\alpha\,〔A_p〕=1 \quad かつ$$
$$Vi',j,\alpha\,〔B_p〕=1$$

(3) IF ： 直感的には，通常のプログラム言語におけるIFと同じ。

(4) REPEAT ： ある動作を何回か繰り返すことを表現する。

(5) WHEN ： ある条件が成り立つのを待ってなにかをおこなうということを表現する。形式的意味を次に示す。

$$Vi,j,\alpha\,〔WHEN\ \ A_t\ B_p〕=1 \quad iff$$
$$Vi,j,\alpha\,〔A_t〕=1 \quad ならば$$
$$Vi,j,\alpha\,〔B_p〕=1$$
$$Vi,j,\alpha\,〔A_t〕=0 \quad ならば$$
$$Vi+1,j,\alpha\,〔WHEN\ \ A_t\ B_p〕=1$$

(6) WHILE ：ある条件が成り立っている間なにかを継続しておこなうということを表現する。

(7) ☆ ： いわばそれぞれの物が有している本質的活動傾向を意味する（たとえば，「猿というものはこんなふうに行動するものである」というようなことを表す）。これはシミュレーション記述における能動的行動主体を表現するために導入した。形式的意味は次のとおり。

$$Vi,j,\alpha\,〔↓(☆x_A)〕=1 \quad iff$$
$$Vi,j,\alpha\,〔be(x_A)〕=1$$

具体的には，プログラムの中で，なにかを実在させる（beにする）という行為を行うと，その直後から新たなプロセスが発生し，そのもとでそれに対応するプログラムが動作しはじめる。なお，ここでいう「対応」関係は，仕様記述上では，

the.普通名詞: 補助詞≡行為式

という文がそれを定義していることになる。

(8) Π ： パラメタとしてタイプ付変数と２つの命題式をとって，あらたな命題式を形成する論理定数。直感的には「すべての～である～は～である」ということを表現するのに用いる。言語NAIVE では論理的限量を含む表現はすべてこのような文型で用いるよう制限するために導入した。形式的意味を次に示す。

$V_{i,j,\alpha}〔\Pi\ x_A\ B_t\ C_t〕=1$ iff

x_A に対する割り当てを除いてαと同じすべての割り当てα'について，

$V_{i,j,\alpha'}〔be\ x_A〕=1$ かつ

$V_{i,j,\alpha'}〔B_t〕=1$ ならば

$V_{i,j,\alpha'}〔C_t〕=1$

(9) Σ： 直感的には「ある～である～は～である」を表す。その他は，Πと同様。

(10) Θ： 直感的には「いくつ以上の～である～は～である」を表す。その他，パラメタとして，数もとることを除いてΠと同様。

(11) Π，Σ，Θ： 直感的には，Πが「すべての～である～は～である」（命題）を表すのに対し，Πは「すべての～である～を～する」（行為）を表す。ΣおよびΘについても同様に命題と行為の違いがそれぞれΣとΘとの間にあるだけである。

(12) and：直感的には，同時実行を表す。形式的意味を次に示す。

$V_{i,j,\alpha}〔A_p\ and\ B_p〕=1$ iff

$V_{i,j,\alpha}〔A_p〕=1$ かつ

$V_{i,j,\alpha}〔B_p〕=1$

(13) or ：直感的には，選択実行を表す。形式的意味はand のそれから類推される。

(14) # ：直感的には，ある条件を満たす個体の個数を表す。

(15) not：関係の否定を表す。

(16) be,nil： 直感的には，それぞれ，「実在している」および「実在していない」ということを表現する。nil はまた，あるものがnil であれば，そのものは，他のすべてのものと何の関係ももっていないという意味も含んでいる。

(17) by,to,in,at,of：by は直感的には隣接関係を意味し，toはその隣接関係に基づく到達関係を意味している。ただし，byとtoに独立に意味があるのでなく，相互にいわば帰納的なメタ関係があるだけである。inは典型的には包含関係を意味し，したがって，推移性をもつ関係である。atは，典型的には「あるものがある場所にある」という関係を表すものである。したがって，「あるものがある場所にあれば，それは他のいかなる場所にもあるはずがない」という意味を含んでいる。また，ofは，典型的には，ある集合への所属関係のようなことを表す。したがって，所属要素ができたとき集合もでき，所属要素が一つもなくなれば，その集合もなくなるという意味を含む（このofの意味は，前章で述べた酒屋の在庫管理システムの記述でも使われている）。

(18) 算術演算子：+（和），－（差），＊（積），／（商），abs(絶対値)，sign(正負)，sum(合計)

(19) 真偽定数：true(恒真)，false (恒偽)

3．4 プログラミング概念との関係

通常のプログラミング上の基礎概念は，それぞれ以下のように，この論理体系の中に位置づけることができる。

(1) 論理式：タイプt の整合式（命題式）

(2) プログラム：タイプp の整合式（行為式）。

(3) プログラム名：タイプ<e*p> あるいはタイプp の定数。

(4) 仕様記述（プログラムシステムの記述）：次のような形式のタイプt の整合式。

$(\uparrow C_p \equiv \uparrow D_p) \wedge \cdots \wedge (\uparrow E_t \equiv \uparrow F_t) \wedge \cdots$

(5) 時刻i でのコマンド“C”投入：モデル制約 $V_{i,j,\alpha}〔C〕=1$ を与えること。ただし，Cは，タイプ<e*p> あるいはタイプp の定数であり，コマンドのパラメタを指定するということは，割り当てαの一部を決めることに相当する。

(6) プログラムシステムの実行：与えられたモデル制約のもとで，与えられた仕様記述（公理；理論）に対するモデル（解）を発見すること。

(7) 並列処理： 与えられたモデル制約の中に $V_{i_1,\ j_1,\ \alpha_1}(C_1)=1$ と $V_{i_2,\ j_2,\ \alpha_2}(C_2)=1$ が含まれていて，時区間$〔i_1,\ j_1〕$と$〔i_2,\ j_2〕$が重なりをもつ場合，これらのモデル制約のもとで仕様記述に対するモデルを発見するときのプログラムの実行形態のこと。

4．言語NAIVE の構文と意味

言語NAIVE の構文は，3章で述べた論理NAIVE のそれに単に構文糖衣を施したものである。その構文規則と意味規則を付録1に示す。なお，BNF 記法では複雑になるので，付録1には入れなかったが，where 構文にはさらに種々の簡略表現が許される。たとえば，次に挙げる2つの表現は，等価である。

① for all.ペン:x where the.ペン:x=red
for some.机:y where the. 机:y=tall
the.ペン:x on-the.机:y

② all.red.ペン:x-on-some.tall.机:y

5．実現方式

言語NAIVE のシステム構成とその用法を図－2に示す。

言語NAIVE システムは，おおきく，翻訳系，論理

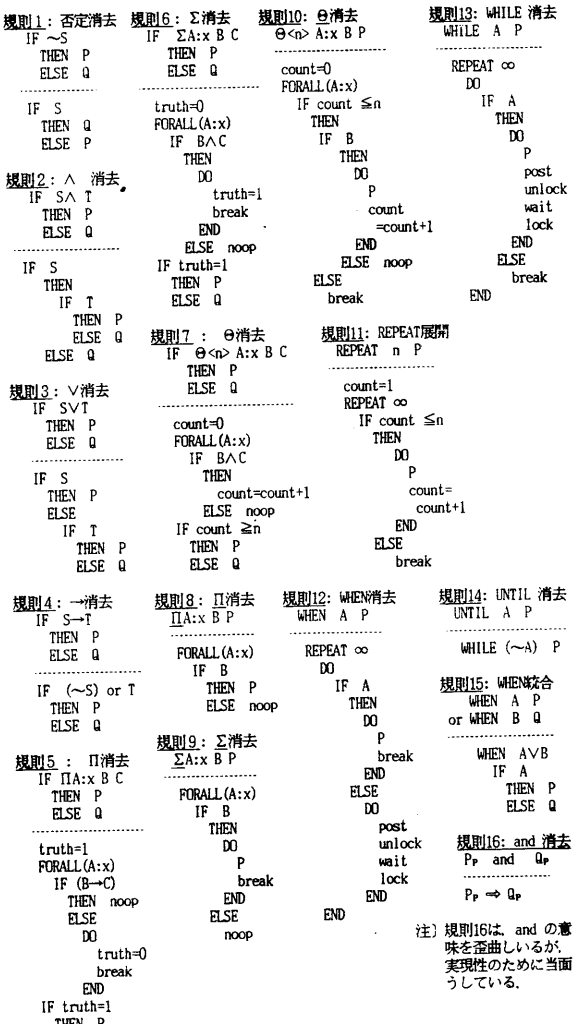
合成系，目的言語展開系，および実行系の各コンポーネントから成る。言語NAIVE で記述したプログラムをこのシステムに投入すると，最終的には，目的言語（たとえばC言語）で書かれたプログラムがデータ宣言部も含めて自動的に導出される。この後コンパイルすれば，ただちに実行可能になる。以下これらの各コンポーネントについて簡単に述べる。

翻訳系：言語NAIVE の構文で記述された仕様記述を内部表現に変換する。この内部表現は，論理NAIVE の整合式の形（関数表現）になる。これにより，後続の論理合成系における項書き換えのように，仕様記述を形式的処理の対象とできるようになる。

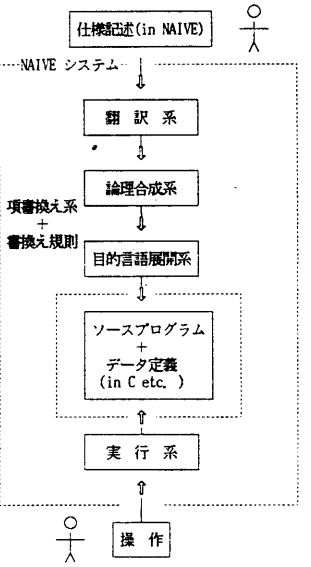
論理合成系：内部表現された仕様記述を項書き換えのメカニズムで変換し，通常のプログラム言語とOSの上で実現可能な中間構文子と中間命令だけからなるプログラムに変換していく。このとき，項書き換えの規則としては，①言語NAIVE の基本詞（接続詞，論理演算詞，限定詞，および，by,to,in,at,of などの関係詞）の意味にもとづく書き換え規則と②各仕様記述の中にあらわれる基底命題定義式と基底行為定義式から得られる書き換え規則とを使用する。そして，その書き換え操作が停止すると，最後に基底命題式と基底行為式を，対応する中間命令に書き換える。書換え規則例を図－３に示す。

目的言語展開系：先に述べたように，論理合成の結果出力されるプログラムは，通常のプログラム言語およびOS上で実現可能な中間構文子と中間命令だけからなっている。さらに，その目的プログラムでは個体や個体間の関係をどのようなテーブル形態で表すか，を統一的に決めておけば，そのプログラムが必要とするデータ宣言部も自動的に生成できる。このようにして，目的言語展開系は，目的プログラムの手続き部とデータ宣言部を同時に出力する。

実行系：仕様記述に対応する目的プログラムを計算機上にロードし，その動作環境を整える。その後コマンドが



図－３　基本詞に関する書き換え規則例



図－２　言語NAIVE システムの構成と用法

投入されると，そのつど新たにプロセスを生成し，そのプロセスのもとで，そのコマンドに対応するプログラム部分を起動する。そのとき同時に入力パラメタを引き渡す。

6．並列処理システムの記述例

言語NAIVE による並列処理システムの記述例を３題示す（図－４，５および６参照）。これらの記述はそのままで実行可能でもある。以下若干注釈をくわえる。

これらいずれの問題の中にも（明示的あるいは暗黙に）含まれている，スケジュールの公平性に関する記述が，仕様記述の中にはないが，それは，NAIVE 実行系の「より先に待ち状態になったプロセスを先にディスパッチする」という単純なしくみによっておのずと実現されている。これらの問題のように，スケジュールの優先度について「自然な条件しか付いていない」問題の場合には，本記述のように，それに関する記述の必要がないのがむしろ望ましいであろう。

また，先に言語NAIVE では手続き記述を導入したと述べたが，NAIVE が許す手続き記述は，あくまで，本質的に手続き的なものについてのみであり，たとえば，for all やfor someに代表される全称的処理，あるいは，なにかの合計を求めるというような，本質的に宣言的な事柄は宣言的に記述する。

〔ジョブスケジューラ〕
問題： 小さなオペレーティングシステムがあり，そこにジョブがどんどん投入される。それらのジョブには，その実行優先度と所要メモリ量が指定されている。全体のメモリ量には制限があるので，ジョブスケジューラは，各ジョブの実行優先度と所要メモリ量とを考慮しながら，効率的にスケジュールする。

言語NAIVE での記述

```
'スケジューラ' ≡
    WHILE true
      WHEN sum<running. ジョブ:x,
                the. ジョブ:x..メモリ量> <500
           some.(not-start). ジョブ:j=start
           WHERE (the. ジョブ:j=enoughsmall and
                the. ジョブ:j..優先度
                    ≧all.(not-start).ジョブ:k..優先度)
                      where the.ジョブ:k=enoughsmall)

 the.ジョブ:j≡
    WHEN the.ジョブ:j=start
      DO
        UNTIL the.ジョブ:j=end
          the.ジョブ:j=running
        the. ジョブ:j=nil
      END
 'ジョブ生成 :j :n :p' ≡
    DO
      the.ジョブ:j=be and
      the.ジョブ:j..メモリ量=the. 数:n and
      the.ジョブ:j..優先度=the. 数:p
    END
 'ジョブ終了 ;j' ≡DO(the. ジョブ:j=end)
 (the.ジョブ:j=enoughsmall) ≡
    (the.ジョブ:j..メモリ量≦500 − sum<running. ジョブ:x,
                          the. ジョブ:x..メモリ量>)
```

図－５　ジョブスケジューラのNAIVE による記述

〔洗車問題〕
問題　：洗車場に車が次々に到着し，何人かの洗車係がこれを洗っていく。車が到着する間隔はランダムで，各洗車係は固有のスピードで洗車を行うものとする。洗車係への車の割り当ては，次の規則に従って行われるものとする。
・車は洗車場に着いた順番に洗い始められる。ただし，同時刻に複数の車が到着した場合は，どれから洗い始めても構わない。
・洗車係は，前の車を洗い終えた順に次の車を洗い始める。ただし，同時刻に複数の洗車係が洗車作業を完了した場合は，どの洗車係から次の洗車を始めてもかまわない。
この洗車作業のシミュレーションを行なえ。

言語NAIVE での記述

```
the.洗車係:w≡
    WHILE the.洗車係:w=be
      WHEN  some.(not-clean).(not-washed). 車:c
                          =in-the.洗車場:PLACE1
        DO
          UNTIL the.車:c=clean
            the.洗車係:w=washing-the. 車:c
          the.洗車係:w=not-washing-the. 車:c
        END
the.車:c≡
    DO
      the.車:c=in-the.洗車場:PLACE1
      WHEN the.車:c=clean
        the.車:c=not-in-the.洗車場:PLACE1
    END
'洗車場生成' ≡
    DO(the. 洗車場:PLACE1=be)
'洗車係生成 :w' ≡
    DO(the. 洗車係:w=be)
'車生成 :c' ≡
    DO
      the.車:c=be     and
      the.車:c=not-clean
    END
'きれいになった :c' ≡
    DO(the. 車:c=clean)
(the.車:x=washed) ≡
    (some. 洗車係=washing-the. 車:x)
```

図－４　洗車問題のNAIVE による仕様記述

7．並列処理言語としての特性

本章では，他の並列言語あるいは動的論理と比較しつつ，言語NAIVE の並列処理言語としての特性について述べる。

(1)　近接的相互作用モデル　－CSP などとの比較－

Occam やAda などの多くの並行処理言語の基礎になっているHoare のCSP(Communicating Sequential Processes)やオブジェクト指向モデルにおけるプロセス間相互作用のモデルは，基本的に，メッセージ通信による，いわば１対１の遠隔作用のモデルであり，メッセージ送信側，受信側とも，通信の相手および通信そのものを明示的に意識するものとなっている。

これに対してNAIVE では，いわば場を介した近接作用によって同期・通信が行われる（正確には通信の概念もない）。すなわち，NAIVE では，各プロセスは，それらに共通に唯一存在する世界（場）の状態を変更・参照しながら互いに独立に動作し，他のプロセスや通信そのものを意識することがない。渡辺他〔13〕にも同様の指摘があるように，NAIVE も，

`飲酒する哲学者`
問題　：哲学者達が飲酒する。哲学者の傍には何個かのテーブルが置ける。テーブルの上には、いろんな種類の酒ビンが置かれている。哲学者達は、一度に何種類かの酒を飲みたくなり、そのときには、それらのすべての種類の酒ビンが飲めるようになると飲酒を始める。渇きが癒えたら、それらの酒ビンをすべて元のテーブルに戻す。
なお、哲学者達は、自分の傍にあるテーブルの上の酒しか飲めない。
この現象のシミュレーションを行なえ。

言語NAIVE による記述

```
the.哲学者:p≡
    WHILE the.哲学者:p=be
       DO
         UNTIL  the.哲学者:p=thirsty
            the.哲学者:p=tranquil
         WHEN  for all.need(the.哲学者:p,*).酒種:m
                 some.(of-the.酒種:m).(near-the.哲学者:p).ビン
                                                  =not-drunk
           DO
             FOR all.need(the. 哲学者:p,*).酒種:m
                the. 哲学者:p=drinking-some.(of-the. 酒種:m).
                      (near-the. 哲学者:p).(not-drunk).ビン
             WHEN  the.哲学者:p=not-thirsty
                the. 哲学者:p=not-drinking-all.ビン
           END
       END
'飲みたい :p :x :y :z' ≡
       DO
         the.哲学者:p=thirsty  and
         the.哲学者:p=need-the.酒種:x  and
         the.哲学者:p=need-the.酒種:y  and
         the.哲学者:p=need-the.酒種:z
       END
'飲みたくない :p'≡
       DO
         the.哲学者:p=not-thirsty  and
         the.哲学者:p=(not-need)-all.酒種
       END
'哲学者生成 :p ' ≡DO(the. 哲学者:p=be)
'テーブル生成 :t'≡DO(the. テーブル:t=be)
'配置 :a :b' ≡DO(the. 物:a=by-the.物:b)
'ビン生成 :m :t' ≡
       DO
         some. ビン:b=be  and
         the.ビン:b=of-the.酒種:m  and
         the.ビン:b=on-the.テーブル:t
       END
 (the. ビン:b=drunk) ≡(some.哲学者=drinking-the.ビン:b)
 (the. 哲学者:p=tranquil)≡(the. 哲学者:p=not-thirsty)
 (the. ビン:b=near-the.哲学者:p) ≡
       (for some.テーブル:t
          (the.ビン:b=on-the.テーブル:t  and
            (the.テーブル:t=by-the.哲学者:p  or
              the.哲学者:p=by-the.テーブル:t)))
```

図－6　飲酒する哲学者のNAIVE による記述

「並列処理にとっては１対１遠隔作用モデルはむしろ不自然で，場を介した近接作用モデルの方がより自然かつ本質的である」という立場をとった。

実際上も，これまで，OSのような並行処理システムは１対１遠隔作用モデルのもとで記述されることが多かったが，まさにそこがOSの設計および検証を困難にするところでもあった。あえて比喩的にいえば，自然な並列処理記述にとっての１対１のプロセス間通信モデルは，ちょうど，構造化プログラミングにおけるGOTO文に相当するものであり，回避するか隠蔽すべきものではなかろうか。

(2)　内部状態の非明示性　　時間論理および状態遷移図（表）との比較－

記述の宣言性および自然語との親近性の点で，Temporal Prolog [3] やRACCO [11] などで採用されている時間論理には共感を覚える。しかし，時間論理では，人間が通常は意識することのないプロセスの内部状態を，明示的に記述することが強いられることが多い。同じことは状態遷移図（表）にも典型的に現れる。このことは，少し複雑な問題を記述しようとすればただちに顕在化する欠点である。

NAIVE では，手続き的記述の積極的導入により，内部状態を明示的に記述することなく自然に動的システムが記述できる。明示的に記述することが不自然に感じられるような内部状態は，手続き的記述では，ごく自然な文脈の形で表現されることが多い。

(3)　ミクロ排他制御容易性　　－従来型プログラム言語との比較－

その他，並列処理システムの実際のインプリメント時には必ずつきまとうミクロ排他制御の問題の自動的解決が容易な点も，従来型言語に比しての当言語の特長である。関係型でも論理型でもない従来型プログラム言語では，そもそも関係処理（ex．２人の人を互いに配偶者関係にする）や全称的処理（ex．すべての～を～する）などの単位性を表現し得ないため，この単位性を保証するためのミクロ排他制御を自動化するのは容易でない。

（注）ミクロ排他制御：たとえば前章で示した洗車問題や飲酒哲学者の問題もそれ自体が洗車や飲酒に関する排他制御の問題を扱うものであるが，ここでいうミクロ排他制御とは，そうした排他制御のさらに一段低位に位置するもので，たとえば，あるプロセスが飲酒可能な酒ビンを自分のものにしようとしているちょうどそのときに，他のプロセスでも同様なことをする，というような不都合を回避するための排他制御のことをいっている。

(4)　形式的意味論としての特性　　－Harel の動的論理との比較－

同じく手続き型プログラムの形式的意味論を与えるものとして，言語NAIVE の論理体系（論理NAIVE ）とHarel の動的論理（DL）とを比較しておく。

Harel の動的論理の骨子は次の４点に要約される：①Kripke的可能世界意味論の一種である；②プログラムの意味は２つの世界からなる対の集合として定式化される；③プログラムそのものは様相記号としてとらえられる；④代入文(x:=τ）および判定文(P?)を基本とする言語を想定して構築されている。

このうち，①と②については，３章に示したとおり，論理NAIVE も本質的に同様である。しかし残る③と④については異なる。③についていえば，論理NAIVE では，その接続詞は一種の様相記号であるが，

プログラムそのものはあくまで基本タイプp の整合式（行為式）と位置づけられている。このことは，論理NAIVE の方がより本格的にプログラム（あるいは行為）というものを取り扱っていることを示唆するものである。また直感的にもわかりやすい。
さらに④については，論理NAIVE は，（行為式を含めて）述語論理的表現を基本とする言語を対象するものであり，適用領域が，より自然でかつ広い。

8．おわりに

本稿では，仕様記述言語NAIVE について，その概要，論理体系，実現方式，仕様記述例，および並列言語としての特性などについて述べた。本言語の処理・実行系が完成して以来，本稿でも示したような約30題の例題の記述・動作実験を行ってきたが，「従来型言語でのプログラミングに比して格段に速くかつ正確にプログラム開発ができる」というのが実感である。今後はさらに実用規模のソフトウェアの記述実験を通して，当言語の実用性を高めていきたい。その他，並列知識情報処理への応用なども魅力ある課題である。

参考文献

1) Chandy,K.M and Misra,J : The Drinking Philosophers Problem,ACM Transactions on Programming Languages and Systems,Vol.6, No.4,pp.632-646(1984)
2) Dowty,D.R.,Peters,S and Wall,R.E. : Introduction to Montague Semantics
3) 榎本肇編著：ソフトウェア工学ハンドブック，オーム社,(1986)
4) 二村，雨宮，山崎，淵：新しいプログラミング・パラダイムによる共通問題の設計，情報処理，Vol.26,No.5,pp.458-459(1985)
5) Gallin,D :Intensional and Higher-Order Modal Logic ,North-Holland (1975)
6) Harel,D : First-Order Dynamic Logic, LNCS 68 (1979)
7) 日野克重：言語NAIVE による実行可能な仕様記述，情報処理学会論文誌投稿中(1989)
8) Hoare,C.A.R: Communicating Sequential Processes,CACM,Vol.21,No.8,pp666-677(1978)
9) 村上昌己，稲垣康善：相互通信逐次型プロセス系の部分的正当性検証体系，電子通信学会論文誌，Vol.J68-D,No11,pp.1846-1853(1985)
10) 桜川貴司：Temporal Prolog,コンピュータソフトウェア』,Vol.4,No.3,pp.15-27 (1987)
11) 桜川，竹中，中島，新出，服部： RACCO:実時間プロセス制御システムのモデル記述のための様相論理プログラム言語，コンピュータ・ソフトウェア,Vol.5,No.3,pp.22-33(1988)
12) 柴山悦哉，米澤明憲：並列オブジェクト指向言語ABCL/1，bit ,Vol.20,No.8,pp940-954(1988)
13) 渡辺，原田，三谷，宮本：場とイベントによる並列計算モデル - Kamui88,コンピュータソフトウェア,Vol.6,No.1,pp.41-55(1989)

付録1　構文規則と意味規則

<仕様記述>::=<定義式列>
<定義式列>::=<定義式>｜<定義式列:A><定義式:B>
　　$A_t \wedge B_t$
<定義式>::='<プログラム名:A>' ≡<行為式:B>｜
　　$(\uparrow A_p) \equiv (\uparrow B_p)$
　　the.<普通名詞:A>: <補助詞:x>≡<行為式:B>｜
　　$(\star x_A) \equiv (\uparrow B_p)$
　　<基底命題式:A>≡<命題式:B>｜<基底行為式:A>≡<行為式:B>
　　$(\uparrow A_t) \equiv (\uparrow B_t)$　　$(\uparrow A_p) \equiv (\uparrow B_p)$
<プログラム名>::=<文字列>｜<プログラム名><プログラム名断片>
<プログラム名断片>::=<文字列>：<補助詞>
<命題式>::=<単純命題式>｜～(<命題式:A>)｜<命題式:A>and <命題式:B>｜
　　$\sim A_t$　　$A_t \wedge B_t$
　　<命題式>or<命題式>｜<命題式>→<命題式>
　　$A_t \vee B_t$　　$A_t \vee B_t$
<単純命題式>::=<基底命題式>｜
<単純行為式>　for <限定句:E>. <普通名詞:A>: <補助詞:x> where<命題:B>
　　<命題:C>
　　$E_{<<t<t<t>>>>}(x_A, B_t, C_t)$
<基底命題式>::=true｜false ｜<対象:xA ><叙述詞><関係:R>｜
<基底行為式>　　$R_{<<t>}(x_A)$ または　$\sim(R_{<<t>}(x_A))$　叙述詞による
　　<関係:R>（<対象:xA >）｜
　　$R_{<<t>}(x_A)$ または　$\sim(R_{<<t>}(x_A))$
　　<対象:xA ><叙述詞><関係:R>－<対象:yB >｜
　　$R_{<<t<t>>}(x_A, y_B)$ または　$\sim(R_{<<t<t>>}(x_A, y_B))$
　　<関係:R>（<対象:xA >，<対象:yB >）｜
　　$R_{<<t<t>>}(x_A, y_B)$
　　<関係:R>（<対象:xA >，<対象:yB >，<対象:zC >）｜
　　$R_{<<t<t<t>>>}(x_A, y_B, z_C)$
　　<対象:xA ><比較詞:R><対象:yA >｜
　　$R_{<<t<t>>}$
<行為式>::=<単純命題式:A>｜
　　A_p
　　DO（<単純命題式:S_t >）｜
　　◎(S_t)　または　(Sが基底命題式のとき)
　　$E_{<<t<t<t>>>>}(x_A, B_t, ◎(C_t))$　(S=E(x_A ,B_t ,C_{t1}) のとき)
　　BE（<命題式:A_t >）｜DO<行為式列:A>END ｜
　　◎(A_t)
　　REPEAT<数:C><行為式:B>｜WHEN<命題式:A><行為式:B>　｜
　　REPEAT$_{<<p>>}$ (C_t ,B_p)　　WHEN$_{<<t<p>>>}$ (A_t ,B_p)
　　WHILE <命題式:A><行為式:B>｜UNTIL <命題式:A><行為式:B>｜
　　WHILE $_{<<t<p>>>}$ (A_t ,B_p)　　WHILE $_{<<t<p>>>}$ (～A_t ,B_p)
　　IF<命題式:A>THEN<行為式:B>｜
　　IF $_{<<t<p<pp>>>>}$(A_t, B_p , ◎(true))
　　IF<命題式:A>THEN<行為式:B>ELSE<行為式:C>｜
　　IF$_{<<t<p<pp>>>>}$(A_t, B_p ,C_p)
　　<行為式:A>and <行為式:B>　｜<行為式:A>or<行為式:B>
　　A_p and B_p　　A_p or B_p
<行為式列>::=<行為式:A>｜<行為式列:A><行為式:B>
　　A_p　　$A_p \Rightarrow B_p$
<対象>::=the.<普通名詞:A>: <補助詞:x>｜the.<普通名詞:A>: <固有名詞:C>
　　x_A　　C_A
　　<対象:xA >..<関数名詞:F>｜<関数名詞:F><<対象:xA >>
　　$F_{<ab>}$ x_A　　$F_{<ab>}$ x_A
<関係>::=<関係詞>｜not-<関係詞>
<限定句>::=all　　some
　　$\Pi_{<<t<t<t>>>>}$または$\Pi_{<<t<p>>>>}$　　$\Sigma_{<<t<t<t>>>>}$または$\Sigma_{<<t<p>>>>}$
　　many. <<数:C>>
　　$\Theta_{<<t<t<t<t>>>>>}$ (C_t)または$\Theta_{<<t<t<t<t>>>>>}$ (C_t)
<数>::=<数詞>｜the.数：<補助詞>｜
　　#（<普通名詞:A>: <補助詞:x> where<命題:B>）｜
　　# $_{<<t<t>>}$ (x_A, B_t)
　　<数>+<数>｜<数>－<数>｜<数>＊<数>｜<数>／<数>｜
　　sign<<数>>｜abs <<数>>
<叙述詞>::= ＝｜≠
<比較詞>::= －｜≠｜≦｜<
<限定詞>::=all ｜some｜many｜the
<普通名詞>::=物｜数｜関数｜関数2　（その他任意に使用可能）
<関係詞>::=by｜to｜in｜at｜of　（その他任意に使用可能）
<数詞>::=0～9の数字の列
<固有名詞>::=（先頭2文字以上が英字の文字列）
<補助詞>::=　（1文字の英字または先頭1文字が英字でそのあとが数字の文字列）