待ち行列網シミュレーション専用論理型言語について

渡辺 尚† 小林 真也† 山口 高平††
中西 暉† 真田 英彦† 角所 収†† 手塚 慶一†
（†大阪大学 工学部 ††大阪大学 産業科学研究所）

１．まえがき

近年、計算機網やローカルエリアネットワーク等の分散型システムが発達してきた。これらのシステムの性能評価手段としては、待ち行列網理論による解析と計算機上での待ち行列網シミュレーション(ＱＮＳ)による解析がある。しかしながら、対象とする網の規模が大きくなり複雑になると前者の理論解析による方法には限界が生じる。そこで後者のＱＮＳが一層重要視されてきた。

ところが、ＱＮＳの生産性は必ずしも高くない。つまりシステム設計者が想定した待ち行列網システムを計算機上で疑似動作させるためのソフトウエアを書き、その正しさを確かめ、実際に計算機上で動作させ必要なデータ(統計)を得るまでの労力は莫大なものである。

一方、最近の人工知能の研究成果から計算機ソフトウエアを論理型言語によって記述し、プログラム作成を容易にする試みがある。

本研究では、以上のことを踏まえ、ＱＮＳ生産性のうち特に記述性を向上させるＱＮＳ専用論理型言語ＳＩＬＱ(Ｓimulation language based on Logicfor Queueing network)を提案する。ＳＩＬＱは論理型言語Prolog[1]を基礎とし、主としてPrologに時刻概念を導入した言語である。

Prologは事物間の関係を定義した論理式をプログラムとみなす言語であり、プログラムの実行状態を考慮せずにプログラミングが可能である(declarative semantics)という特徴を持つため、高い記述性を有しているとされている。したがって、ＦＯＲＴＲＡＮのような手続き型言語に比べＱＮＳのように内部に多くのプロセスが存在しそれらの関係を記述する場合には有効であると思われる。しかし、Prologの持つ非決定性からどの手続きから実行されるか一意的に決まらないため、シミュレーションに必要不可欠な時刻概念を取り扱えない。Prologに時刻概念を導入する方法としては、Ｉ．Futoらが提案しているＴ－Prolog[2]のような時刻に関する組み込み述語による方法や、Concurrent Prolog[3]のようにread－only変数を用いて実現する方法等が提案されてきた。ところが、これらの方法ではPrologのdeclarative semanticsの利点が損なわれてしまう。そこでＳＩＬＱではシミュレーションの対象をＱＮＳに限り、プロセステンプレートの概念を導入することによって時刻をユーザレベルには見えなくし、なおかつdeclarative semanticsを保ったところに価値がある。

本稿では、まず、ＱＮＳの生産性について述べＱＮＳソフトウエアをPrologで記述する際に生じる問題について言及する。次にＳＩＬＱにおける時刻概念の取り扱いとＳＩＬＱが提供する２０個のプリミティブについて述べる。さらに、他の言語といくつかの例を通して比較することによってＳＩＬＱの記述性を検討する。

２．待ち行列網シミュレーションとその生産性

ＱＮＳは網を通り抜ける客、網を構成するバッファやサーバ等の構造および客が各サーバやバッファ内でどのように扱われるか等の動作から構成される。またシミュレーションに必要な情報も記述しなければならない。まとめるとＱＮＳソフトウエアには以下の項目が記述される。

（１）構造
①ソフトウエア上での客の構造
②サーバ、バッファの数
③サーバ、バッファの接続関係
（２）動作
①バッファ内での客の並び方
②バッファから客の出ることのできる条件
③サーバ内で客の受けるサービス
④経路選択方法
⑤他のサーバ、バッファに客の情報を知らせる事が必要な場合のその方法
など
（３）制御情報
①シミュレーションの開始時刻、終了時刻
②必要な統計量

ＱＮＳの生産性は記述性、検証性、経済性の３つに分けることができる。記述性とはユーザが対象としているシステムの待ち行列網モデルをプログラムで表すときの書きやすさ、読みやすさ、理解しやすさ等、検証性は書いたプログラムの正しさの保証の与えやすさをそれぞれ意味する。経済性は、実際の計算機上で実行し必要とするデータ(統計量)を得るまでの時間と実行に必要な計算機資源量にかかわる要素である。

現在ＱＮＳにはモデルの記述が容易な待ち行列網シミュレーション専用言語ＧＰＳＳや処理速度が速い利点を持つＦＯＲＴＲＡＮなどが用いられている。しかし、これらでは複数のプロセス間の同期を陽に記述しなければならない。従って、ＦＯＲＴＲＡＮはもちろんＧＰＳＳでさえも対象とする網が大規模になり含まれるプロセスが多くなるとそれらの同期をとることが困難になってくる。すなわち、実際に解析が困難な待ち行列網モデルに対しては手続き型言語では記述性の向上が望めない。

ＱＮＳはそれに含まれるイベントおよび構成要素間の関係を記述しさえすればソフトウエアを構成できるという立場から論理型言語Prologを基礎とし、これによってＱＮＳソフトウエアの記述性向上を図る。

3．論理型言語Prologによる待ち行列網シミュレーション

Prologの持つ非決定性によって時刻概念、すなわち、ある事象が生じてから次の事象が発生する順序関係を扱うことができない。しかし、時刻概念は複数のプロセス間の同期に深く関係するため、ＱＮＳには必要不可欠である。そこでPrologに時刻概念を導入する事を考える。

Prologに時刻概念を導入する方法としては

Ⅰ．プロセス間の同期を陽に表すプリミティブを導入する。

Ⅱ．プロセス間の同期をユーザレベルから見えなくする。

Ⅰ．については、Ｔ－Prologのような時刻に関する組み込み述語を用意する方法、またConcurrent Prologにおけるread－only変数を利用する方法が考えられている。

Ｔ－PrologはFutoらが提案しているシミュレーション専用言語であり、Prologに概念上並行に動作する複数のプロセス、システムによって更新される時刻およびリソースの概念を加えたものである。

本研究で特に興味のある時刻に関する組み込み述語としては以下の４つがある。

①hold(T)　：T time units だけプロセスの実行を中断する
②wait_for(M)：メッセージMを受信するまで待つ
③wait(C)　：条件Cが満たされるまで待つ
④send(M)　：メッセージMを送信する

Ｔ－Prologによるプログラミングでは時刻を超えた述語の入れ換えは不可能である。つまり、プログラムを手続き的に記述しなければならず、Prologの利点であるdeclarative semanticsを損なう。

4．ＳＩＬＱにおける時刻概念の取り扱い

本研究は、前節で述べた時刻概念の導入方法のうちⅡの立場にたつ。時刻概念をユーザから隠ぺいする方法として以下に示すプロセステンプレートの概念を提案する。

プロセスはメッセージの受信によって起動されそのメッセージに対する処理を行った後他のプロセスにメッセージを送信する。ただし、各プロセスはメッセージ受信に対しては受動的である。このようなプロセスを入出力関係と時刻消費の有無によって分類し、表現したものがプロセステンプレートである。以下プロセステンプレートについて詳述する。

(a)　入出力関係によるプロセスの分類

一般にプロセスに入ってくるメッセージの経路数とプロセスから出ていくメッセージの経路数によって図１に示すような分類が可能である。つまり、

（ｉ）１出力のみ
（ii）１入力のみ
（iii）１入力１出力
（iv）１入力多出力
（ｖ）多入力１出力
（vi）多入力多出力

(vi)については(iv)(ｖ)の組み合わせで実現できるので特別なものとしては考えない。

(b)　時間の消費によるプロセスの分類

プロセスは予め定めた時間がたたなければメッセージの送信ができないか否かによって２種類に分類できる。前者をtime－consumingプロセス、後者をnon－time－consumingプロセスと呼ぶ。

以上２種類の分類法によって合計１２のプロセスに分類できるがＱＮＳは以下の６つのプロセスによって記述できる。これらのプロセスの表現形式(プロセステンプレート)と状態遷移図を図２に示す。

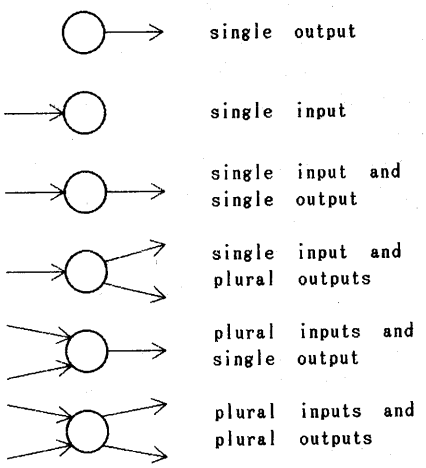

図１　プロセスの分類

各プロセスは以下の動作を繰り返す。

（ｉ）ＡＲＲＩＶＡＬ process

ＡＲＲＩＶＡＬプロセスは、まずある客に対する処理を行う(executing状態)。その後、予め定められた分布に従った時間だけ消費する。つまりこの時間がたつまでこのプロセスは休止している。これが同図におけるholdingの状態である。holding状態からはtimeoutによって解除される。

（ii）ＤＥＰＡＲＴＵＲＥ process

ＤＥＰＡＲＴＵＲＥプロセスは、受信したメッセージで表わされる客に対する処理(例えば統計収集など)を実行し(executing状態)次のメッセージを待つ。(waiting状態)

（iii）ＳＥＲＶＥＲ process

このプロセスは、今受信したメッセージによって表わされる客に対して予め定められた確率分布に従った時間を消

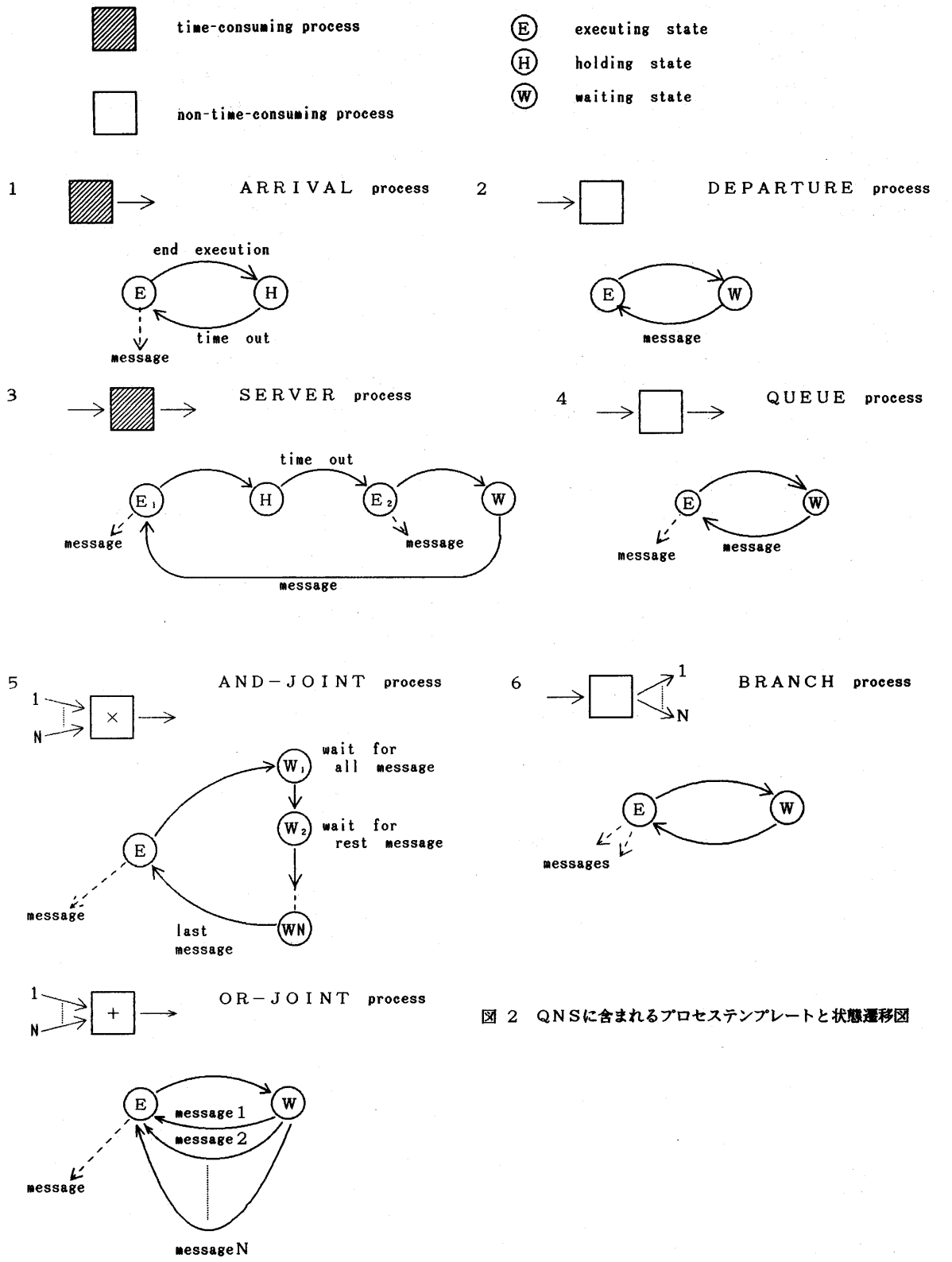

図 2　QNSに含まれるプロセステンプレートと状態遷移図

費する。ここでholding状態となる。holding状態を抜けたプロセスは次のプロセスへメッセージを送信する。その後次のメッセージを待つ。

(iv) QUEUE process

QUEUEプロセスは、受信したメッセージを予め定義された規律に従って待ち行列に並べる。また待ち行列から出る条件が満たされると待ち行列の先頭の客を次のプロセスに送り出す。QUEUEプロセスにおけるメッセージの送信は受動的に可能になっている。

(v) JOINT process

JOINTプロセスには2つのタイプがある。すなわち、AND－JOINTプロセスとOR－JOINTプロセスである。前者はこのプロセスが待つすべてのメッセージを受信するまでは次の処理を開始できない。後者は、待っているメッセージの内1つを受信すれば次の処理(すなわちメッセージの送信)を実行できるプロセスである。実際には後者はあまり意味を持たないがANDに対するJOINTとして加えておく。

(vi) BRANCH process

このプロセスは経路を複数本選択したり、単一のメッセージを複写し放送型の通信を実現するためのプロセスである。

5. シミュレーション専用言語SILQ

SILQではQNSによく現れる同種のプロセスを一括して定義できるようにプロセスクラスとプロセスインスタンスを持っている。プロセスクラスとは前章で述べた6つのプロセステンプレートであり、同じ機能を持つプロセスインスタンスを一括して定義する。実際にはシミュレーション実行中にプロセスインスタンスとなって動く。プロセスインスタンスはクラス名と識別子で表現される。例えば、クラス:QUEUE 識別子:3のプロセスインスタンスは

[queue,3]と表される。

客は以下に示すようにシステムフィールドとモデルフィールドからなるリスト構造をしている。

$$[\underbrace{[[T_1,T_2,\cdots,Tn],[P_1,P_2,\cdots,Pn]]}_{\text{System Field}} \mid \underbrace{[*****]}_{\text{Model Field}}]$$

システムフィールドは客が現在まで通ってきたプロセスインスタンス名Piと、Piを出た時刻Tiのリストの集合から成る。この履歴の管理はシステムが独自に行う。モデルフィールドはユーザが自由に書き込み読み出しができる。モデルフィールドによってユーザがより細かいレベルのシミュレーションが行えるようになっている。

SILQでは、ユーザが待ち行列網をより簡単に定義できるように、20個のプリミティブを用意している。ユーザはこれらのプリミティブを用いて、待ち行列網の構造と動作を別々に定義できる。さらに、シミュレーションの制御情報についてのプリミティブが用意されている。20個のプリミティブは、構造に関するプリミティブ1個、動作に関するプリミティブ17個、制御情報に関するプリミティブ2個である。もちろん、ユーザは必要に応じてプリミティブ以外の述語の定義ができる。

プリミティブには、以下に示すシステムプリミティブとユーザプリミティブと呼ばれる2種類がある。

① システムプリミティブは、システム内部で定義されているものであり、ユーザはホーン節のbody部にのみ用いることができる。システムプリミティブの例としては、各プロセッサの処理履歴やプロセスの現在の時刻等がある。シンタクス的にはsys__で始まる。プリミティブである。

② ユーザプリミティブはユーザ自身がその動作等を定義できるプリミティブである。つまり、ユーザプリミティブはホーン節のhead部に用いる。この例としては待ち行列の規律等がある。

各プリミティブの一覧表を表1に示す。

5.1 構造の定義 (STRUCTURE primitive)

構造に関するプリミティブとしては、connectがある。connectのシンタクスを以下に示す。

connect(Id, P_Id1, P_Id2)

connectは2つのプロセスインスタンスP_Id1とP_Id2を接続するプリミティブである。Idはconnect identifierであり、このconnectによって表される接続を示す指標である。connect identifierによって2つの接続を2つのプロセスの接続と同様に階層的に表すこともできる。この有用性は待ち行列網によく現れる階層的な構造、例えば1つのサーバと1つのキューをまとめてノードとする構造を容易に扱えるところにある。

5.2 動作の定義 (BEHAVIOR primitive)

動作に関しては表1に示すように各プロセスごとに異なるプリミティブと共通なプリミティブを提供している。

5.2.1 プロセスプリミティブ

ここでは、ARRIVALプロセスプリミティブappend__model__fieldについて説明する。append__model__fieldの第一引数はARRIVALプロセスのインスタンス名である。また第二引数はシステムフィールドのみの客、第三引数はモデルフィールドを伴った客を表す。ユーザはシステムフィールドのみを持つ客にユーザの対象とするモデルにあった属性をモデルフィールドに書くことができる。書かれた属性はsys__enter__customerとcustomer__changeを用いることによって各プロセスにおいて自由に参照、書き替えができる。

5.2.2 各プロセスに共通なプリミティブ

これらはプロセスに入ってきた客やこのプロセスがシミュレーションを開始してから現在までの履歴の参照等を行うためにある。また、ユーザがダイナミックにプログラムを変化させることができるプリミティブも用意されている。

5.3 シミュレーション制御(CONTROL primitive)

シミュレーションの開始、および終了の時刻を規定するプリミティブである。

6．ＳＩＬＱの使用例

ここではＳＩＬＱの簡単な使用例としてガソリンスタンドシミュレーションを挙げる。

[例　1]

以下にシミュレーションの仮定を示す。

① 給油設備は1台

② 車は平均3分の指数分布に従って到着する。

③ 給油時間は2～6分の一様分布とする。

④ シミュレーション時間はガソリンスタンドが開店してから閉店するまでの8時間とする。

例1の構造は、図3に示す単一キュー単一サーバの待ち行列網で表現できる。同図ではプロセスクラスとしてARRIVAL、QUEUE、SERVER、DEPARTUREがあり、プロセスインスタンスとしては[arrival,1],[queue,1],[server,1],[departure,1]がある。　ＳＩＬＱによるプログラムを図4に示す。このプログラムは以下の定義からなる。

(i)　ＡＲＲＩＶＡＬプロセス

・到着間隔は平均3分の指数分布に従う

(ii)　ＱＵＥＵＥプロセス

・待ち行列から客を出す条件はサーバに車がいないことである。

・最大待ち行列長は無限大である。

・待ち行列規律は先き入れ先き出しである。

(iii)　ＳＥＲＶＥＲプロセス

・サービス時間は平均2～6分の一様分布に従う。

(iv)　ＤＥＰＡＲＴＵＲＥプロセス

この例では統計量収集を目的としていないので何も定義されていない。

(v)　structureの定義

[arrival,1]と[queue,1],[queue,1]と[server,1],[server,1]と[departure,1]がそれぞれ接続されており、connect　identifierは1、2、3である。

各プロセスは4節で述べたプロセステンプレートをもっている。ここで各プロセスがいかに実行されるかを示すために[server,1]動作について説明する。

①waiting状態からexecuting状態への遷移

プリミティブsys__customerで表される客が現在このプロセスインスタンス[server,1]に入ってきた所である。例1では何も実行されないが、ユーザがなんらかの客に対する処理を実行したい場合はプリミティブcustomer__changeを用いることができる。

②executing状態からholding状態への遷移

プロセスインスタンス[server,1]はservice__timeで表される時間だけ消費する。

③holding状態からexecuting状態への遷移

サービス時間たった後、客はサーバを出る。ここで、[server,1]は客の行き先をconnectを用いて決定し、プロセスインスタンス[departure,1]に客を表すメッセージを送信する。

以上のようにプロセステンプレートに各プロセスインスタンスを割り当て処理を行う。

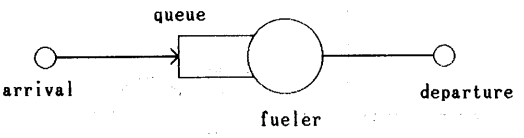

図 3　ガススタンドシミュレーション

```
/*                */
/*  Gas Station   */
/*                */

/*  Arrival  */
interval(1,T):-exp_distribution(3,T).


/*  Queue  */
queue_out_condition(1,T):-
                sys_customers_in_server(1,[]),
                sys_time([server,1],T).

queue_out_selection(1,[Packet|Rest],Packet).

queue_priority(1,[[[T1|X1]|SFP1]|MF1],
                [[[T2|X2]|SFP2]|MF2]):-T1>=T2.

buffer_capacity(1,10000).


/*  Server  */
server_capacity(1,1).

service_time(1,T):-s_random(4,Tr),T is 2 + Tr.


/*  departure  */


/*  connect  */
connect(1,[arrival,1],[queue    ,1]).
connect(2,[queue  ,1],[server   ,1]).
connect(3,[server ,1],[departure,1]).


/*  Start&End Time  */
start_time(0).
end_time(480).
```

図 4　ＳＩＬＱ による記述(例1)

7．ＳＩＬＱと他の言語との比較

ＳＩＬＱで記述したモデルをＴ－PrologとＧＰＳＳで記述し比較することによってＳＩＬＱの記述性を検討する。

7．1　Ｔ－Prologとの比較

[例　2]

記述例は、図5に示すような2ノードがループ状に接続されているパケット交換網である。以下を仮定する。

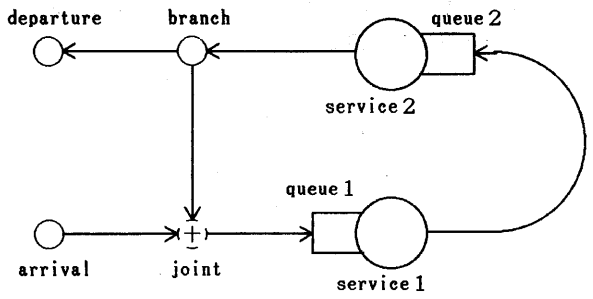

図 5　2ノードネットワークモデル

```
/*  Server 1  */
servicel:-send(queuel,[qout,sl]),
                wait_for(Packet),
                intserv(sl,T),hold(T),
                send(queue2,[qin|Rest]).
```

図 6　T－Prolog による記述(例2)

```
/*  server  */
service_time(Id,T):-member_of(Id,[1,2]),
                    s_random(9,T).

server_capacity(Id,1):-member_of(Id,[1,2]).
```

図 7　SILQ による記述(例2)

①パケットの到着時間間隔は0～20単位時間の一様分布
②サーバのサービス時間は0～9単位時間の一様分布
③サーバ2を出た客が網から退去する確率は0．5
④シミュレーション開始時刻は0、終了時刻は10000単位時間

この例に対するシミュレーションプログラムをT－PrologとSILQで記述したものが図6と図7である。両図はそれぞれのサーバに関する記述部分である。（SILQの全プログラムを付録に示す。）2つのプログラムはサーバ1を例に挙げると以下のように解釈できる。

◎T－Prolog

サーバ1の動作は下に示す4つのステップからなる。

(1)Queue－1に対してパケットの送信要求を出す。
(2)Queue－1からのパケットを待つ。
(3)サービス時間だけパケットを保持する。
(4)パケットをQueue－2に送信する。

◎SILQ

サーバ1の動作は以下のような定義からなる。

(1)パケットがServer－1で費やすサービス時間
(2)Server－1で同時にサービスを受けることのできるパケット数

2つのプログラムを比較した結果、T－Prologでは待ち行列やサーバの記述の他にプロセス間のメッセージ送受信を記述しなければならない。つまり、プロセス間の同期を陽に記述しなければならない。また、T－Prologでは記述順序の入れ替えは不可能であり、Prologを手続き的に利用しているにすぎない。一方、SILQでは各プロセスの動作を定義するだけでプログラムが成り立っている。

7．2　GPSSとの比較

GPSSと比較するためには例1のような簡単な例では顕著な差はでない。そこでエレベータのシミュレーション[8]について比較した。ここでは紙面の都合上結果のみを挙げる。

SILQによる記述はプロセス中心であるが、GPSSによる記述はトランザクション中心の記述である。後者はユーザが自分自身をトランザクションと見なし自分が網の中を動きまわる様子を記述すればよい利点があるが、複数のプロセスの同期についてはユーザが陽に記述しなければならず、繁雑である。

一方、SILQによる記述では、各プロセスに関する定義を行うだけでよいが、エレベータシステムを待ち行列網に分解して考えなければならない。この点については、ある対象から待ち行列網モデルへの変換を支援するシステムを検討する予定である。

8．SILQ処理系

8．1　試験システム

現在、SILQは基本的なプリミティブのシンタクスが決定されたところであり、詳細な機能および統計量に関するプリミティブは検討中である。前に述べたプリミティブの妥当性はSILQの試験システムによって確かめている。

試験システムでは、時刻の取り扱いをT－Prologに類似した組み込み述語をサポートしたT－like－Prologをパーソナルコンピュータ上の逐次処理Prologで構成しプロセステンプレートを作成している。上記の例1～2のSILQで記述されたプログラムは人手によってプロセステンプレートに変換し、実際にT－like－Prolog上でその動作を確かめたものである。

8．2　SILQシステム

経済性を考慮すればマルチプロセッサシステムが望ましいのは当然である。しかし、マルチプロセッサ上にジョブをどのように配分するか、また複数のプロセッサの制御をどのようにするかなどの問題が存在する。これらに対する対策として現在以下を考慮中である。

(1)プロセス割り当て方法

まず、time－consumingプロセスにプロセッサを割り当てる。もし、利用可能なプロセッサの数がtime－

consumingプロセスよりも少ない場合にはいくつかのプロセスをまとめて同一プロセッサに割り当てなければならない。 次にnon－time－consumingプロセスをプロセッサ間通信が最小になるようにプロセッサに割り当てる。以上２つのステップのいずれでもプロセッサの負荷をなるべく均等にする必要がある。

(2)プロセッサ制御方法

ＱＮＳは画像処理などとは異なって確率的要素を含むためプロセス間のメッセージパッシング[5]の形はunknown型となる[6]。unknown型を含むジョブの制御についてはすでにＤ－ＳＳＱ[7]で実験済みである。従って、基本的にはＤ－ＳＳＱのプロセッサ制御方式である先行制御方式を用いる。

９．まとめ

本稿で提案したPrologに基づく待ち行列網シミュレーション専用論理型言語ＳＩＬＱは、６個のプロセステンプレートの概念を用いて、ユーザから時刻概念を隠ぺいし、declarative semanticsを保持した所に大きな特徴がある。ＳＩＬＱには２０個のプロセッサが用意されている。また、他のシミュレーション言語Ｔ－PrologとＧＰＳＳによる記述例と比較し、ＳＩＬＱの記述性の高さを確認した。

今後の課題としては、

(Ⅰ) 統計量に関するプリミティブおよび種々の例についての検討。

(Ⅱ) ＳＩＬＱ試験システムの完成

(Ⅲ) ＳＩＬＱの検証性についての検討

(Ⅳ) マルチプロセッサを用いたＳＩＬＱシステムの作成

(Ⅴ) ＳＩＬＱシステムの時間的および空間的効率の検討

などが残されている。

謝 辞 本研究を行うに当たって熱心なご討論を戴いた同学研究生佐藤 圭氏に感謝する。

文 献

[1] Clocksin,W.F. Mellish,C.S.: " Programming in Prolog"Springer-Verlag 1981

[2] Futo,I. Seredi,J.:"A Discrete Simulation System Based on Artificial Intelligence Methods" Simulation and Related Fields North-Holland,1982

[3] Shapiro,E.Y.:"A subset of Concurrent Prolog and Its Interpreter" Technical Report TR003 ICOT

[4] "AGPS(言語および操作)説明書"DEMOS-E 4151

[5] Yonezawa,A. Matsuda,H.:"Toward Object Oriented Concurrent Programming" ソフトウエア科学･工学の数理的方法 京大数理解析研講究録 5471985

[6] 渡辺､佐藤､山口､中西､真田､角所､手塚:"並行処理待ち行列網シミュレータD-SSQにおけるPrologの記述性について" 情報処理全国大会 1985 3Q-1

[7] 渡辺､佐藤､中西､真田､手塚:"並行処理事象型待ち行列網シミュレータD-SSQについて" 電子通信学会交換研究会1983-8 CS83-102

[8] 小林:"待ち行列網シミュレーション専用論理型言語に関する研究"大阪大学卒業論文 1985

[付録]

```
/*  arrival  */
interval(1,Ti):-s_random(20,Ti).

/*  queue  */
queue_out_condition(Id,T):-
        member_of(Id,[1,2]),
        sys_customers_in_server(Id,[]),
        sys_time([server,Id],T).
queue_out_selection(Id,
        [Packet|Rest],Packet):-
        member_of(Id,[1,2]).
buffer_capacity(Id,10):-
        member_of(Id,[1,2]).
queue_priority(Id,[[T1|X1]|Y1],
                  [[T2|X2]|Y2]):-
        member_of(Id,[1,2]),T1>=T2.

/*  server  */
service_time(Id,T):-
        member_of(Id,[1,2]),
        s_random(9,T).
server_capacity(Id,1):-
        member_of(Id,[1,2]).

/*  branch  */
branch_selection(1,6):-
        s_random(100,R),R<=50,!.
branch_selection(1,7).

/*  joint  */
joint(1,or).

/*  departure  */

/*  connect  */
connect(1,[arrival,1],[joint    ,1]).
connect(2,[joint  ,1],[queue    ,1]).
connect(3,[queue  ,1],[server   ,1]).
connect(4,[queue  ,2],[server   ,2]).
connect(5,[server ,1],[queue    ,2]).
connect(6,[server ,2],[branch   ,1]).
connect(7,[branch ,1],[departure,1]).
connect(8,[branch ,1],[joint    ,1]).

/*  Start&End Time  */
start_time(0).
end_time(10000).

member_of(X,[X|Y]).
member_of(X,[Y|Z]):-member_of(X,Z).
```

表 1　ＳＩＬＱプリミティブ

① STRUCTURE primitive	
connect(C_Id,Ps_Id,Pr_Id)	C_Id:接続識別子 Pr_Id:受信プロセスインスタンス識別子 Ps_Id:送信プロセスインスタンス識別子
② BEHAVIOR primitive	
INDIVIDUAL primitive	
ARRIVAL	
append_model_field(Id,Sys_customer,Customer)	Id:プロセスインスタンスの識別子 Sys_customer:システムフィールドのみの客 Customer:モデルフィールド付きの客
interval(Id,T)	T:メッセージ発生時間間隔
QUEUE	
queue_out_condition(Id,Tc)	Tc:待ち行列からパケットの出る時間の条件
queue_out_selection(Id,List,Customer)	List:待ち行列に並んでいる客のリスト Customer:先頭の客
buffer_capacity(Id,C)	C:バッファの容量
queue_priority(Id,Hp,Lp)	Hp:優先権の高い客 Lp:優先権の低い客
sys_customers_in_queue(Id,List)	List:待ち行列に並んでいる客のリスト
SERVER	
server_capacity(Id,C)	C:同時に同じサービスを受けられる客の最大数
service_time(Id,Ts)	Ts:サービス時間
sys_customers_in_server(Id,List)	List:現在サービスを受けている客のリスト
JOINT	
joint_logic(Id,Log)	Log:jointの論理(and／or)
BRANCH	
branch_selection(Id,List)	List:選択された経路のリスト
COMMON primitive	
sys_time(Id,Tp)	Tp:プロセスインスタンスIdの時刻
sys_customers_record(Id,List)	List:プロセスインスタンスIdを出ていった客のリスト
sys_enter_customer(Id,List)	List:現在プロセスインスタンスIdにいる客のリスト
sys_change(User(Pre),User(Post))	User(Pre):ユーザの定義した述語Userの前の状態 User(Post):Userの後の状態
customer_change(Pre_customer,Post_customer)	Pre_customer:変化する前の客 Post_customer:変化した後の客
③ CONTROL primitive	
start_time(St)	St:シミュレーション開始時刻
end_time(Et)	Et:シミュレーション終了時刻

ただし、大文字で始まるものは変数