ソフトウェア工学 68－4
（１９８９．９．２６）

リフト共通問題の代数的仕様記述

田中哲雄

(株)日立製作所 システム開発研究所

代数的手法による並列処理の仕様記述法を提案する。並列処理の具体例としてリフト共通問題を取り上げ、筆者等が設計した代数的仕様記述言語Æqlによる記述例を示す。また、この種の問題に対する代数的手法の有効性を考察する。

並列に動作する複数のリフト(エレベータ)やそれらに付随するボタンの同期をとるために関数Proceedを定義した。この関数をクロックとみなし、全てのリフト状態がこのクロックに同期して同時に変化するとして仕様記述を行なった。また、リフトシステムを一つの抽象的順序機械として記述することにより、全ての状態について出力値が唯一つ定義されていること等の確認が容易となった。

Specifying the Lift Problem with an Algebraic Method

Tetsuo TANAKA

Systems Development Laboratory, HITACHI, Ltd.
1099 Ohzenji, Asao-Ku, Kawasaki, Japan

abstruct: We propose how to specify concurrent problems with an algebraic method. As an example, we have specified the Lift Problem by using the algebraic language Æql designed by our research group. Our experiment will also show us advantages and disadvantages of Æql. We introduce the function "Proceed" that controls synchronous actions of the lift-system. The function "Proceed" is a clock for concurrent actions of all lifts and buttons in the system. The lift system and its components (lifts and buttons) are described as an abstruct sequential machine. The description-style makes specifications consistent and complete.

1．はじめに

ソフトウェアの仕様を、厳密にかつ形式的に記述する方法として、代数的仕様記述がある。これは、ソフトウェアにおける操作の対象を抽象データ型としてとらえ、抽象データ型の構造を等式によって定義することにより、ソフトウェアの仕様を記述する方法である。筆者等は、代数的仕様記述言語Æql(AlgebraicEquational Language：イー・キュー・エル)、及び、その処理系のプロトタイプを開発し、その評価を行なってきた。

本稿では、具体例としてリフト共通問題[1]を取り上げ、それをÆqlにより記述し、並列処理の代数的仕様記述法を提案する。リフト共通問題の記述例として、CSP[2]やLOTOS[3]によるものがある。CSP による記述は、複数のリフトから非決定的に一個のリフトを選んで一単位ずつ動作させるという形で書かれているため、複数のリフトが同時に動作することが記述されていない。Æqlによる記述では、クロックに相当する関数 Proceedを定義することにより、全てのリフトがクロックに同期して、同時に動作するという形で記述できた。

また、CSPやLOTOSでは起りうるイベントの集合をいくつかのプロセスに分けて記述するので、仕様に抜けがないことの確認が困難である。この点代数的仕様は、仕様が網羅的に記述されているか否かを確認しやすい、という特徴を持つ。リフト共通問題のÆqlによる記述を通し、並列処理の仕様記述における上述の有効性を確認した。

以下、２.でÆqlの概要を述べ、３.でリフト共通問題とそのÆqlによる記述例を示す。次に、４.でÆqlの評価を述べる。

2．Æql概要

準備としてÆqlの構文、及び、特徴を説明する。

2．1　構文

代数的仕様記述では、操作の対象を抽象データ型として捉える。操作の対象が複数種類ある場合、それらを区別するために名前をつけその名前をソートと呼ぶ。また、抽象データ型における操作のことを関数又はオペレーションと呼ぶ。以下、代数的仕様記述法における仕様の定義を述べる（仕様記述により定められる項の集合、及び、その上での合同関係等、代数的仕様記述の意味については文献４）または５）を参照されたい）。また、Æqlにおける書式を図２.１に示す。

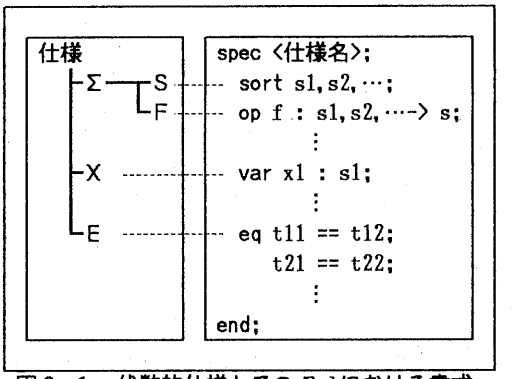

図２.１　代数的仕様とそのÆqlにおける書式
ただし s,siはソート名，fは関数名，
xは変数名である。またtijは項である。

１）仕様は、シグネチャΣ、変数集合族Ｘ、及び等式集合Ｅの組＜Σ，Ｘ，Ｅ＞である。

ここで、ΣとＸは項の集合を定義し、Ｅは項の集合上の合同関係を定義する。直観的には、Σで関数の構文を定義し、Ｅでその意味を定義していると見ることができる。

２）シグネチャΣはソートの集合Ｓと関数記号領域Ｆの組＜Ｓ，Ｆ＞である。

３）関数記号領域Ｆは集合族$<F_{w,s}>_{w\in S^*,s\in S}$であり、関数名，定義域，値域を定義する。

４）変数集合族Ｘは$<X_s>_{s\in S}$であり、等式集合で使われる変数とそのソートを定義する。

５）等式集合Ｅは項の集合上の合同関係を定める。

2．2　言語Æqlの特徴

Æqlは、仕様の読みやすさ・書きやすさ・再利用のしやすさの向上を目的とした、（１）仕様の階層性、（２）基本関数、（３）パラメータ付き仕様（４）ソートの包含関係の４個の機能を具備している。

（1）仕様の階層性

仕様を記述する際、他の仕様記述で既に定義済みの関数を、その仕様中で使いたい場合がある。また、大きな仕様を記述する際、仕様をモジュール化しそれらを組合せた方が読みやすく、かつ書きやすい。これを実現する手段として、Æqlに他の仕様記述（1つのモジュールに相当）を自分自身に取り込む機能includeを設け、仕様の階層性を実現した。

仕様Aで仕様Bを取り込みたいときは図2.2のように取り込む側の仕様に取り込まれる仕様名を予約語includeのあとに続けて書く。

（2）基本関数

仕様を記述する際よく使われるソート（整数、文字列、論理値、配列、リスト等）及びその上で定義される関数（+、−、&、cons等）は仕様記述言語の組込として、ユーザが定義せずに使えると便利である。そこで基本的なソートおよび関数をそれぞれ基本ソート、基本関数と呼びÆqlの組み込みとした。

基本ソート、基本関数は仕様Primitivesに記述されているとし、これらを他の仕様で用いるときは include Primitives によってその仕様に取り込む。

（3）パラメータ付き仕様

例えば整数のソーティングと文字列のソーティングの2個の仕様を書く場合、ソーティングしたい要素のソートのみが異なり他はほとんど同じ仕様になる。このようなとき、そのソートをパラメータ化できれば便利である（仕様が汎用的になり部品として使いやすくなる）。そこで、Æqlにパラメータつきスペックを記述できる機能を設けた。

例えば、ソーティングの仕様を記述する場合は、ソーティングしたい要素の仕様をパラメータ化して図2.3のように書くことができる。

図2.3において、SP,Elm,orderは仮パラメータであり、SPという仕様のElmというソートの要素をorderという順序でソーティングすることを表しているとする。例2.1に示すように仮パラメータに実パラメータを代入することにより整数

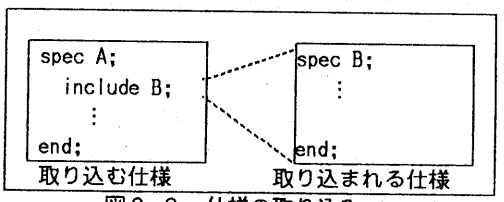

図2.2　仕様の取り込み

```
spec SORTING[SP,Elm,order];
    include SP;
        ⋮
end;
```
図2.3　パラメータ付き仕様

や文字列などのソーティングの仕様が得られる。

[例2．1]

整数を降順にソーティングする

SORTING[INT,Integer,greater]

例2.1においてSORTING[INT,Integer,greater]は、仕様SORTING[SP,Elm,order]中の全ての SP,Elm,orderという文字列をそれぞれINT,Integer,greaterという文字列で置き換えた仕様であることを表す。Æqlのパラメータ機能では、文字列の置換で様々な表現が可能だが、他の仕様との整合性は、記述者の責任でとらなければならない。

（4）ソートの包含関係

代数的仕様記述では、整数の加法と実数の加法のように本来同じ意味をもつ関数でも引数のソートが異なると別の関数を用いなければならない。またエラーを表す項を導入したとき、その項を含むソートと含まないソートは、それ以外の要素が同じでも異なる名前にしなければならない。この問題を解決するためにソートの包含関係を明示する機能を設けた。

ÆqlではソートAがソートBに含まれることを subsort A ＜ B で表す。

[例2.2]

```
subsort Int < Real
```

例2.2の宣言はソートIntがソートRealに含まれることを表わしている。このとき、Add,Subなどの関数ををReal上で定義すれば、それらはRealの要素、Intの要素両方を引数にできる。

3．リフト共通問題とその代数的記述

3．1　リフト共通問題

本稿で取り上げるリフト共通問題を図３．１に記す[1)]。以下、図３．１の記述を自然語仕様と呼ぶ。自然語仕様には不備と思われる点が幾つかある。本稿では、便宜的に以下のように定め、代数的仕様記述を行なった。

1）ドアの開閉についてほとんど記されていないので、ドアについては記述しない。

2）運行停止を解除したときの状態は運行停止前の状態とする。

3）フロアボタンを押したとき、リフトがそのフロアに居るならば、ボタンは点灯しない。

4）要求はあるがその方向に既に他のリフトが向かっているときは、要求がなくなった時点で（既に向かっているリフトが到着した時点で）停止する。

5）リフトはある階に到着してから一定時間停止して、次の要求に対してサービスを行なう。また、出発してから一定時間後に次の階に到着する、又は次の階を通過する。

尚、これら仕様の不備は、Æqlによって形式的に記述してみて初めて発見されたものである。

また、自然語仕様の２の“The algorithm to decide which to service first should minimize the waiting time for both requests.”については、どちらの要求に応じれば両者の待ち時間が最小になるか、その時点では決定できないので、この要求は無視した。

3．2　記述方針

文献２）及び、３）では、複数のリフトから、非決定的に１個のリフトを選んで、１単位動作させるという形で記述されている。しかし、各リフトは入力（ボタンの押下）とは非同期にかつ同時に動作（到着・出発）するので、リフトの動作のトリガとして関数 Proceedを定義し、全ての状態遷移は Proceedと同期して起こるとしてリフトシステムを記述する。

An n lift system is to be installed in a building with m floors. The lifts and the control mechanism are supplied by the manufacturer. The internal mechanisms of these are assumed (given). The problem concerns the logic to move lifts between floors according to the following constraints:

1-Each lift has a set of buttons, one for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. the illumination is canncelled when the corresponding floor is visited by the lift.

2-Each floor has two buttons (except ground and top floor), one to request an up-lift and one to request down-lift. These buttons illuminate when pressed. The illumination is canncelled when a lift visits the floor and is either moving in the desired direction, or has no outstanding request. In the latter case, if both floor request buttons are pressed, only one should be canncelled. The algorithm to decide which to service first should minimize the waiting time for both requests.

3-When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests.

4-All requests for lifts from floors must be serviced eventually, with all floors given equal priority.

5-All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel.

6-Each lift has an emergency button which, when pressed causes a warning signal to be sent to the site manager. The lift is then deemed 'out of service'. Each lift has a mechanism to cancel its 'out of service' status.

図３．１　リフト共通問題

また、代数的仕様の記述スタイルに抽象的順序機械がある。これは、仕様を網羅的に記述しやすいという特徴を持つ。リフト共通問題をこのスタイルで記述し、並列問題の仕様記述での有効性を

考察する。

抽象的順序機械とは、次の条件を満たす代数的仕様の記述スタイルである。

＜条件１＞記述の対象を表すデータタイプを仮りにSTATEで表すと、

（１）引数にデータタイプSTATEを２個以上もつ関数は存在しない。

（２）等式の左辺の形は f(x1,…) または f'(g(x1,…),…) である。ただし、gの値域はSTATEである。また、変数 x1,… はすべて異なる。

（３）同じ左辺をもつ等式は複数個存在しない。また、左辺が f(g(x1,…),…) の形の等式があるとき、左辺が f(x1,…) や g(x1,…) の形の等式はない。 □

＜条件１＞において、データタイプSTATEの元は順序機械の状態に対応する。また、値域がSTATEの関数を状態遷移関数と呼び、それ以外の関数を出力関数と呼ぶ。前述の関数Proceedは、この抽象的順序機械のクロックとして位置付ける。

３．３　代数的記述

抽象的順序機械型の記述スタイルでは、①「状態遷移のタイミング」、②「状態遷移による出力関数の値の変化」の２点を記述する。①は処理の順序を規定するスケジューラの動作であり、②はリフトやボタンの個々の動作である。①のスケジューラの記述を図３．２に、②のリフトの個々の動作の記述を図３．３に示す。以下それぞれの記述について順に説明する。

３．３．１　スケジューラの記述

スケジューラの仕様記述を図３．２に示す。

リフトシステムは、ｎ基のリフト、ｎ組のリフトボタン、及び、ｍ組のフロアボタンからなる。それぞれをモジュール化し、抽象的順序機械として別々に記述する。図３．２では、リフトやボタンの仕様、及び、入力イベント・リフトの運行方向に関する仕様は既に、別に記述してあるものとしてその仕様内に取り込む(include文)。そして、

```
spec LiftSystem(M,N,P,C);
include Primitives,Lift(M,P,C),Button,
        Event,Direction;
sort SYSTEM, EVENT_SEQ : sequence of EVENT;
op Main       : SYSTEM,EVENT_SEQ -> SYSTEM;
   Proceed    : SYSTEM,EVENT -> SYSTEM;
   FloorButton: SYSTEM,INT,DIRECTION ->BUTTON;
   LiftButton : SYSTEM,INT,INT -> BUTTON;
   Lift       : SYSTEM,INT -> LIFT;
var s : SYSTEM;
    in : EVENT_SEQ;
    e : EVENT;
    l,f : INT;
eq

Main(s,in) ==
  if Null(in) then End(s)
  else Main(Proceed(s,Head(in)),Tail(in));

FloorButton(Proceed(s,e),f,d) ==
  if SomeLiftArrived(Proceed(s,e),f,d) then
        TurnOff(FloorButton(s,f,d))
  else if FloorBottonWasPushed(e,f,d) then
        TurnOn(FloorButton(s,f,d))
  else
        FloorButton(s,f,d);

LiftButton(Proceed(s,e),l,f) ==
  if LiftArrived(Proceed(s,e),l,f) then
        TurnOff(LiftButton(s,l,f))
  else if LiftButtonWasPushed(e,l,f) then
        TurnOn(LiftButton(s,l,f))
  else
        LiftButton(s,l,f);

Lift(Proceed(s,e),l) ==
  if InService(Lift(s,l)) then(
    if LiftMoving(Lift(s,l)) & … then
      Arrive(Lift(s,l),l,LiftFloor(s,l),Nil)
    else if LiftMoving(Lift(s,l)) & … then
      Arrive(Lift(s,l),l,LiftFloor(Lift(s,l))
                  LiftDirection(Lift(s,l)))
    else if LiftMoving(Lift(s,l)) then
      Pass(Lift(s,l),l)
    ⋮
end;
```

図３．２　スケジューラの仕様

Proceedをリフトシステム全体の状態を変化させる状態遷移関数とし、include文で取り込んだリフト・ボタンの状態遷移関数を用いて、状態遷移Proceed後（１クロック後）の各リフト、各フロアボタン・リフトボタンの状態の変化をボタンやリフトの状態遷移関数を用いて記述する。

システムへの入力はフロアボタンを押す、リフトボタンを押す、運行停止ボタンを押す、運行停止状態を解除する、というイベントの列である。システムの状態はこのイベント列によって変化するので、これをmainの引数とする。

イベントは１クロックごとにシステムに入力される。また、一回で入力されるイベントは上記４種類のイベントの０個以上の組である。

３．３．２　リフトの記述

リフトの動作のÆqlによる記述例を図３．３に記す。Æqlによる仕様記述は図３．４の手順で行なう。以下、この手順に沿って図３．３の仕様を説明する。

（１）ソートの宣言

リフトの状態を表すソートLIFTを宣言する。尚、INT,BOOL,DIRECTIONは仕様Primitives,Directionで定義されている。

（２）出力関数の宣言

リフトの各状態における、状態要素の値を取り出す関数(出力関数)を宣言する。言い替えれば、出力関数の値の組がリフトの状態であり、出力関数はその組から要素を取り出す射影関数である。出力関数は、運行状態にあるか否か(InService)、リフトの方向(LiftDirection)、リフトがいる階(LiftFloor)、リフトが動いているか否か(LiftMoving)、リフトが階の間のどこにいるか(LiftPosition)、停止時間(Counter)、の６個である。

（３）状態遷移関数の宣言

リフトの動作を表すものとして状態遷移関数を宣言する。状態遷移関数は、リフトがある階に到着したことを表すArrive、リフトがある階を出発したことを表すLeave、リフトがある階を通過したことを表すPass、１クロック停止することを表すStop、運行停止ボタンが押されたことを表すPuushEmergencyButton、運行停止状態が解除されたことを表すPutBackInService、の６個である。

（４）変数の宣言

関数の意味定義に用いる変数を宣言する。

```
spec Lift(M,P,C);
include Primitives,Direction;
sort LIFT;
op
/* 出力関数 */
  InService      : LIFT                -> BOOL;
  LiftDirection  : LIFT             -> DIRECTION;
  LiftFloor      : LIFT                -> INT;
  LiftMoving     : LIFT                -> BOOL;
  LiftPosition   : LIFT                -> INT;
  Counter        : LIFT                -> INT;
/* 状態遷移関数 */
  Arrive         : LIFT,INT,DIRECTION -> LIFT;
  Leave          : LIFT,DIRECTION     -> LIFT;
  Pass           : LIFT               -> LIFT;
  Stop           : LIFT               -> LIFT;
  PushEmergencyButton : LIFT          -> LIFT;
  PutBackInService    : LIFT          -> LIFT;
var
  l: LIFT;
  f: INT;
  d: DIRECTION;
eq

  InService(Arrive(l,f,d)) == InService(l);
  LiftDirection(Arrive(l,f,d)) == d;
  LiftFloor(Arrive(l,f,d)) == LiftFloor(l);
  LiftMoving(Arrive(l,f,d)) == False;
  LiftPosition(Arrive(l,f,d)) == 0;
  Counter(Arrive(l,f,d) == 0;

  InService(Leave(l,d)) == InService(l);
  LiftDirection(Leave(l,d)) == d;
  LiftFloor(Leave(l,d)) ==
      if = Up then LiftFloor(l) + 1
      else LiftFloor(l) - 1;
  LiftMoving(Leave(l,d)) == True;
  LiftPosition(Leave(l,d)) == 1;
              :
end;
```

図３．３　リフトの仕様

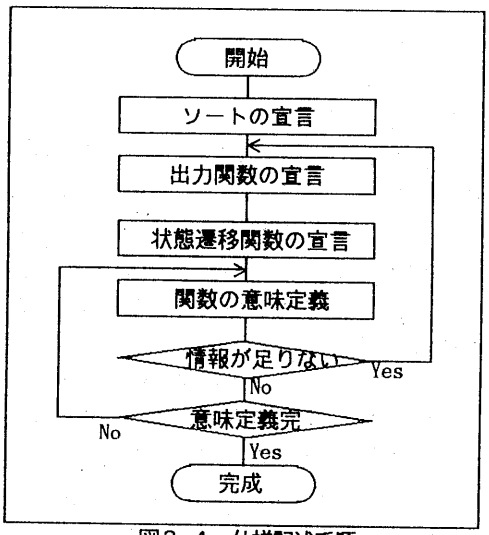

図3.4　仕様記述手順

（5）各関数の意味定義

各状態遷移後の出力関数の値を

f(g(l,…),…) == h(l,…)

の形で定義する。但しｆは出力関数、ｇは状態遷移関数、ｌはリフトの状態を表す変数である。

上記（2）では、出力関数を６個挙げているが、例えばシステムの設計者は、これらの成分を仕様を書き始める前から全て知っているわけではない。図３.４ に示すように、足りない情報が有れば、その情報を状態から取り出す出力関数を宣言し、あとは全ての状態遷移関数に対して、状態遷移後の出力関数の値を定める。従って、状態遷移による出力値の変化に関しては、網羅的な仕様記述が可能である。

また、（5）の記述法により、「全ての状態に対して出力関数の値が全て定義されているか（完全性）」、及び、「唯一つ定義されているか（無矛盾性）」の確認が容易になる。

尚、リフトボタン／フロアボタンの仕様も同様に記述する。

３.４　仕様の拡張例

３.４.１　ドアの開閉の記述

Æqlによって記述された仕様の拡張性を評価するために、リフトのドアの開閉を例に取り、図３.３の仕様を拡張する。

仕様を次のように拡張することを考える。

1）出発のとき開いていれば閉じる。

2）到着のとき、乗り降りの要求が有れば開き、そうでなければ閉じたままとする。

3）フロアボタンを押したときそのフロアにリフトがいれば、その内の一つのドアを開く。

仕様の拡張も、仕様の新規作成と同様に図３.４の手順で行なう。以下、その詳細を述べる。尚、「閉じかけ」という状態はないとする。

まず、ドアが開いていることを表す出力関数LiftDoorOpenを宣言し、次に定義済みの各状態遷移関数に対して、この出力関数の値を次のように定義する。即ち、PushLiftButton,PushEmergencyButton,PutBackInService,Passの各状態遷移のときLiftDoorOpen値は変わらない。状態遷移PushFloorButtonのとき、あるリフトについて、リフトがフロアにいて、ドアが閉じていれば開く。状態遷移Arriveの場合、要求がないときは閉じたまま到着し、要求があるときは到着してから開く。状態遷移Leaveの場合は閉じる。

３.４.２　ドアの開閉の詳細化

仕様を更に以下のように詳細化する。

1）ドアが閉じかけているとき、フロアのボタンを押すと、ドアは開く。

2）押しつづけているとき、ドアは閉じない。

この詳細化の手順も図３.４と同様である。

まず、ドアの開閉の状態を表すソートDOOR（元は「Open(開いている)」、「Close(閉じている)」、「Closing(閉じかけ)」の3個）を宣言する。次に、ドアの開閉の状態、ボタンが押されているか否かを表す出力関数Door、Button、及び、「ボタンを離す」、「ドアが閉じ始める」ことを表す状態遷移関数 ReleaseFloorButton、CloseDoorを宣言し、

3．4．1と同様に等式を追加する。更に、状態遷移の条件を決定し、図3．2のスケジューラの仕様中の関数Proceedの定義を変更する。

4．評価

（1）スケジューラの記述について

元来、代数的言語は宣言的であり、同期制御の方法を明確にすれば、並列処理の記述は容易である。本稿では、3．3．1に述べたように、並列に動作する複数のリフトやボタンの同期を取るために関数Proceedを導入した。

文献2）3）では、クロックの概念を用いずに「リフトが非決定的に選択されてそのリフトが一単位動作する」という形で記述されている。Æqlでも非決定的選択のオペレーションを導入することにより、この方法で記述することができるが、これは、本来同時に動作している複数のリフトを1個ずつ動作させることになり明らかに不自然である。今回は、より人間の思考に近いと考える前述の方法を採用した。

（2）リフトの動作の記述について

リフトを抽象的順序機械として記述することにより、網羅性(設計者の意図が抜けなく記述されている)、完全性(全ての関数が定義されている)、無矛盾性(関数の値は高々一つだけ定義されている)に優れた仕様記述が可能になった。実際、3．1で述べた仕様の不備は、Æqlで形式的に記述してはじめて発見されたものである。

（3）仕様の拡張について

仕様の拡張は、状態遷移後の出力値の変化に関しては、元の仕様をほとんど変更せずに、いくつかのソート、関数、及び等式を追加するだけ行なうことができた。3．4．1の例では出力関数LiftDoorOpenと、8つの等式と追加しただけで元の仕様は変更していない。しかし、状態遷移のタイミングに関しては、遷移条件の場合分けが変化するので、元の仕様を変更しなければならなかった。

（4）仕様の読みやすさについて

Æqlによる記述は、記述量が多くなるにつれ煩雑になり読みにくくなる。今回の記述では、スケジューラの記述で、ｉｆ式の条件部が複雑になり、読みにくくなった。

5．おわりに

リフト共通問題を例として、並列処理の代数的仕様記述の一手法を提案した。並列に動作する複数のリフトやボタンの同期を取るために、関数Proceedを導入した。この関数をクロックとみなし、全てのリフトやボタンの状態が同時に変化するものとして記述した。

また、記述スタイルとして抽象的順序機械を採用することにより、状態遷移による出力値の変化に関して、網羅性の優れた仕様記述ができた。

反面、Æqlによる記述は、記述量が多くなるにつれ煩雑になり読みにくくなるという欠点をもつ。この欠点を解消することが今後の課題である。

謝辞　熱心に御討論頂いた(株)日立製作所システム開発研究所 山野紘一主任研究員、田中厚研究員に感謝する。

参考文献

1)Problem Set For the 4th International Workshop on Software Specification and Design, IEEE, Proc. of the 4th International Workshop on Software Specification and Design, p.x,April 1987.

2)M.D.Schwartz and N.M.Delisle,"Specifying A Lift Control System with CSP,"Proc. of the 4th International Workshop on Software Specification and Design, pp21-27,April 1987.

3)大蒔，二木："図書館の問題とエレベータの問題のLOTOSによる仕様記述"，情処研報，SE64-12，平01-02.

4)嵩，谷口，杉山："代数的言語の設計と処理系"，榎本編：ソフトウェア工学ハンドブック，オーム社，昭61-06.

5)稲垣，坂部："抽象データタイプの代数的仕様記述法の基礎"，情報処理，Vol.25，No.1，pp47-53，昭59-01.