代数的仕様と時制論理によるリアルタイム SA と オブジェクト指向設計の融合手法†

本位田 真一†† 大須賀 昭彦†† 内平 直志††

本論文では形式的仕様記述手法が実用に供せられるための種々の課題に対して以下の解決案を示している．まず，分析フェーズにはリアルタイム SA，設計フェーズにはオブジェクト指向設計を採用し，さらにフェーズ間の流れをできる限り自然にすることを試みている．その際に，データや機能に関する記述には代数的仕様，同期やイベントの記述には時制論理を適用する．また，代数的仕様による記述内容の検証には，項書換え系を採用し，正しさの保証されたプログラムを生成するために時制論理の定理証明系を採用する．以上の過程を経て，最後に Ada のタスクを生成する．

1. はじめに

種々のソフトウェアの仕様化技術の中で，形式的手法は，それによって得られるソフトウェアの品質や信頼性を高める手法として広く認められている．しかし，リアルタイムシステムを対象とした場合，産業界では十分に活用されているとは言い難いのが現状である．形式的仕様が実用に供せられるためには，少なくとも次の要件に対して何らかの形で応ずる必要がある．まず，所望のシステムの性質をできる限り多く記述するための形式的手法を確立することである．これは，実務ソフトウェアに適用するにあたって最も要求される項目である．第二としては，形式的手法を用いてどのように概略仕様からより詳細な仕様に展開するかといった，仕様展開プロセスを確立することである．形式的な記述に習熟するにはかなりの訓練と時間を必要とするため，仕様展開プロセスが確立していることが望ましい．また，これによって，複数の人に同一の概略仕様を与えた時，得られるプロダクトにおいて，できる限り差異が生じないようにすることが可能となる．第三には，形式的手法での記述内容を簡便な手法で検証・確認する手段が必要である．記述された仕様のバグを容易に見つけることのできる手段も，合わせて必要である．ここで第二と第三の要件は，実務ソフトウェアにおいて，形式的仕様に経験の少ない人にとっては特に必要となる．第四には，対象とする実務ソフトウェアの規模が大きい場合には，全体を見渡すことのできる手段も必要である．第五に形式的手法で記述された仕様から，プログラムを生成できることが要求される．いかに厳密に仕様を記述できたとしても，それがプログラムに直接結びつかないのであるならば，実務現場での受け入れ可能性は極めて低くなる．

これらの要求項目に応ずるために従来より多くの仕事がある．たとえば第一の要求を満たすために，複数の形式的仕様を効果的に組み合わせる数多くの手法の提案がある[4),7),13),14),17),19)]．これは，単独の形式的手法によって所望のシステムの性質をすべて記述することは，現実的に困難であることによる．しかしながら，それらの提案が第二〜第五までの要求をすべて満たしているとは言い難い．そこで本論文では，並行プロセスを有するリアルタイムシステムを対象として，これらの要求項目にどのように対処するかについて述べる．

まず第一に複数の形式的手法として，代数的仕様と時制論理の組み合わせを採用する．代数的仕様によって静的な側面である機能やデータに関する性質を記述し，時制論理によって動的な側面である並行性，同期，排他に関する性質を記述するものである．第二に，リアルタイムシステムにおいて広く利用されているリアルタイム SA (Structured Analysis) を採用し，仕様展開プロセスを規定する．リアルタイム SA においては，DFD (Data-Flow Diagram) の記述に代数的仕様を適用し，制御スペックの記述に時制論理を適用する．第三の課題に対しては，代数的仕様，時制論理には様々な体系が存在するが，できる限り自動検証可能な範囲に限定させる．第四の課題に対しては，

† An Integration Method of Real Time SA and Object Oriented Design Using Algebraic and Temporal Logic Specifications by SHINICHI HONIDEN, AKIHIKO OHSUGA and NAOSHI UCHIHIRA (Systems and Software Engineering Laboratory, Toshiba Corporation).

†† (株)東芝 システム・ソフトウェア技術研究所

DFD での階層関係に従って，上位の DFD から下位の DFD へ，またはその逆というように自由に記述世界を traverse できるようにし，全体を見渡すことを可能とする．第五に対しては，オブジェクト指向設計 (OOD：Object-Oriented Design) の概念[1]を導入する．

以上のように形式的手法を実用に供するための種々の課題に対する解決案を示す[10]と同時に，システム化した結果を合わせて論じるのが，本論文の目的である．

2. 概　要

本章では，提案手法の概略とそれを支える要素技術について述べる．

まず，仕様記述の核となっているリアルタイム SA を明らかにする．本手法で扱うリアルタイム SA は，DeMarco の SA[3] をリアルタイム向けに拡張しているが，その拡張部分は DFD に現れる“平行線”（ここではストアと呼ぶ）に対する複数の“円”（ここではバブルと呼ぶ）からのアクセスにおいて，そのアクセス順序を状態遷移図によって記述することである．したがって Ward や Hatley のリアルタイム SA をすべてカバーしているわけではない[8),20]．Hatley 手法の範囲では，状態遷移図で記述する制御スペックの対象範囲をストアの回りのアクセス順序に限定していることになる．

提案手法においては入力仕様はリアルタイム SA のコンテクスト・ダイアグラムであり，出力形態は Ada のタスクであるが，次の５つの phase から構成される(**図１**参照)．

- phase 1：DFD を展開することによるオブジェクトの生成
- phase 2：オブジェクトの実現レベルでの記述
- phase 3：項書換え系に基づく検証手続きによる検証
- phase 4：時制論理による制御スペックの生成
- phase 5：Ada のタスクの生成

以下，各 phase ごとに簡単な例題も交えて説明する．この例題は，エレベータの制御システムであり，各階のフロアおよび“かご”からの呼び出しに対処することを目的としている[18]．

<phase 1>

まず，与えられた仕様のコンテクスト・ダイアグラムを記述することから始まるが，このコンテクスト・

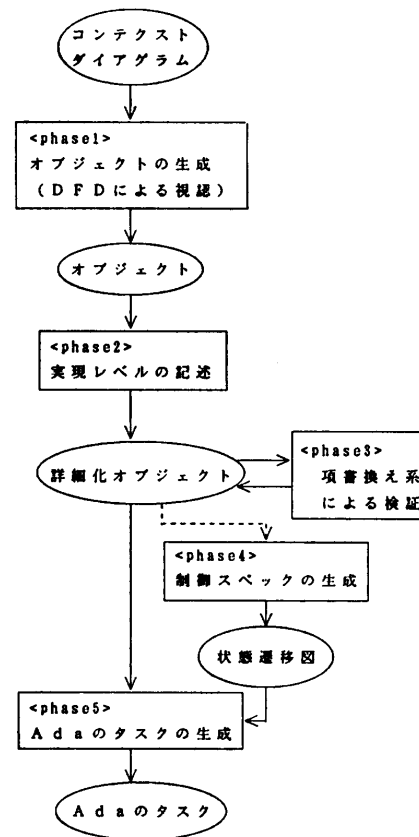


図１ 全体の流れ
Fig. 1 System overview.

ダイアグラムは単一のオブジェクトに相当する．このオブジェクトにおける入出力データ（これが対象システムの外界とのインタフェースを規定）について，代数的仕様を用いて記述する．次に，このコンテクスト・ダイアグラムを DFD に展開する．この時，各バブルの機能を代数的仕様によって記述する．さらに DFD の各バブルを，終了条件を満たすまで展開基準に従って展開していくことを繰り返す．本手法では展開基準と終了条件を次のように規定している．

（展開ルール１）

出力データ１つに対して機能が１つ対応するように分割する．ただし分岐は１つとみなす．（一般に代数的仕様では，機能に対応する演算は副作用のない関数の形で定義されるため）

（展開ルール２）

等式の右辺の記述に基づき展開する．バブル内において等式表現によってそのバブルの機能の性質を表現する．この等式の記述されている式は，機能の働きを表現していると同時に機能の展開を表しているとみな

```
タスク仕様::=
      task  タスク名  [  is
        {  エントリ宣言  }
      end  [  タスク名  ]]
タスク本体::=
      task  body  タスク名  is
        [  宣言部  ]
      begin
        loop
          select
            select文選択肢
        {   or
            select文選択肢  }
          end  select;
        end  loop;
      end  [  タスク名  ];
select文選択肢::=
      [  when  条件  =>  ]
            accept文
条件::=  ノード番号変数  =  整数
accept文::=
      accept  エントリ名  [(エントリ添字)] [仮パラメタ] [do
        文の列
      end  [エントリ名]];
```

図 2　Ada タスクの出力形態の基本形
Fig. 2 Basic format in Ada task.

すことができる.

（展開ルール 3）

ストアはオブジェクトとしてまとめる.

（展開ルール 4）

各バブルはひとつのストアのみをアクセスするように展開する.

（終了条件）

バブル内の等式表現の右辺が primitive function (登録済みの function) あるいは recursive function で構成された時.

phase 1 では，上の終了条件を満たすまで上の展開ルールによって機能展開を繰り返しながら，最終的な出力である Ada のタスクの骨格（**図 2** 参照）を構築する．また，その過程において，代数的仕様におけるシグネチャや等式による記述内容は，随時 DFD に変換し視認することを可能としている．phase 1 において機能展開を繰り返す手順については文献 5) で既に示している．本論文ではその手法を発展させ，OOD を導入し，Ada のタスクの骨格を作る手法についても述べる.

OOD における設計プロセスは

①オブジェクトの抽出

②オブジェクトの属性の定義

③オブジェクトにおけるメソッドの定義

④オブジェクト間のインタフェースであるメッセージの定義

といった大きな流れになる．ここで②と③は逐次的処理ではなく，同時にあるいは交互に処理されるのが現実的である．この流れに沿って手順をまとめると以下のとおりである.

①′ DFD で現れるストアを中心として，展開ルール 3 に従いオブジェクトとしてまとめる．このオブジェクトが生成すべきタスクの基礎となる.

②′③′ ストアの回りのバブルにおいて機能展開を繰り返すことによって，ストアと関係が深い function (DFD ではバブル，OOD ではメソッドに対応)，言い換えれば，ストアを直接アクセスする function を洗い出す．同時に function の洗い出しに必要なデータ（代数的仕様ではソート，OOD では属性に対応）も合わせて洗い出す．洗い出した function 群はオブジェクトに登録される．等式があるオブジェクトに宣言されたデータを直接アクセス（参照，書込み，更新）するならば，そのオブジェクトにシステムが自動的にデータをアクセスする function として登録する．このようにしてオブジェクトのデータに関わる function を抽出していく．最下位層まで機能展開が完了した時点では，ストアを直接アクセスする function はオブジェクトとしてまとめられている．展開ルール 4 により，各 function は高々ひとつのストアしかアクセスしないため，まとめられるべきオブジェクトは一意に決まる．一方，ストアを直接アクセスしない function は，いずれのオブジェクトにも属していないことになる．そこでそれらの function をいずれかのオブジェクトに登録しなければならない．登録のための手順は次のとおりである.

（a）オブジェクトに属している function “x” に対して直接，データの受け渡しをする（データを与える）function “y” を選択する.

（b）function “y” を function “x” の属しているオブジェクトに登録する.

（c）該当する function “y” が存在しなくなるまで（a），（b）を繰り返す.

（d）オブジェクトに属している function “x” から直接データを受けとる function “y” を選択する.

(e) function “y” を function “x” の属しているオブジェクトに登録する.

(f) 該当する function “y” が存在しなくなるまで(d), (e)を繰り返す.

(g) (f)が完了した時点において，いずれのオブジェクトにも属していない function は次の場合である.

- 外界の入力データからどのオブジェクトを経由することなく外界の出力データに至るデータフローのパス上に存在する.

この場合には，そのパス上の function はパスごとにひとつのオブジェクトとしてまとめる.

④′オブジェクト間のインタフェースは，2つのオブジェクトA，Bに各々属す function “a”, “b” の間のデータ受け渡しに対応する. function “a”, “b” の間のデータの受け渡しに関する情報は，機能展開の過程で作成する DFD より得られる. すなわち，ここでの function はバブルに対応するが，DFD はバブル間のデータフローを表しているためである.

例として，**図3**に，phase 1 の入力であるコンテクスト・ダイアグラムおよび**図4**に出力である LIFTstate オブジェクトを中心としたオブジェクト群を示す. (コンテクスト・ダイアグラムからの展開プロセスについては既に文献5)で示しているので，ここでは省略する.) **図5**に LIFTstate オブジェクトの記述内容を示す.

<phase 2>

phase 1 において DFD を展開し，オブジェクトを構築したが，この中での等式記述はあくまでも外部仕様レベルである. 言い換えれば，オブジェクト内のデータ構造を意識しない記述である. phase 2 においてはオブジェクトが含むデータの構造をまず決定する. その際には，対象とするアプリケーションを意識することも必要である. phase 1 で生成したオブジェクトにおいて登録されている function は，データの構造や実現レベルの内容を含んでいない. その理由は，それらの function はオブジェクトの外側からデータをとらえており，抽象データ型に対する function であるとみなすこともで

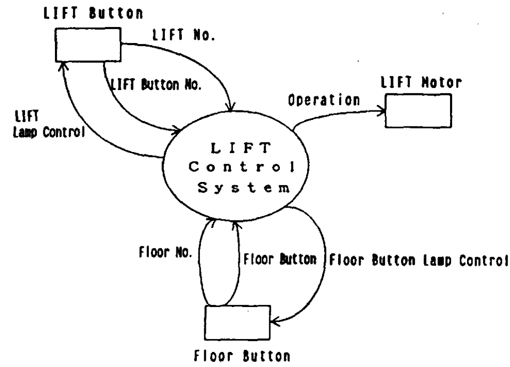


図3 コンテクスト ダイアグラム
Fig. 3 Context diagram.

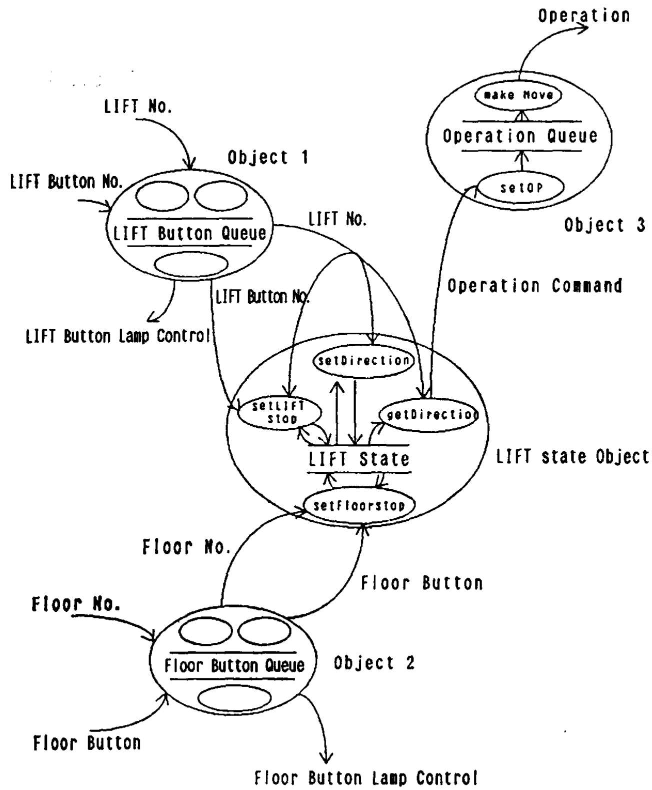


図4 オブジェクト群の例
Fig. 4 Objects example.

```
object: LIFTstate
sort:   LIFTstate, LIFTno., OperationCommand, OperationStatus,
        Direction,LIFTposition, ..........
opns:   setDirection: LIFTNo., LIFTstate -> LIFTstate
        getDirection: LIFTNo., LIFTstate -> OperationCommand
        setLIFTstop: LIFTNo., LIFTButtonNo., LIFTstate -> LIFTstate
        setFloorstop: FloorNo., FloorButton, LIFTstate -> LIFTstate
eqns:   setLIFTstop(L,F,S)=setLIFT(writeStopFloor(F,getLIFT(L,S)),L,S)
                  .
                  .
                  .
```

図5 LIFTstate オブジェクト
Fig. 5 LIFTstate object.

```
e1:  setLIFTstop(A,B,C)=setLIFT(writeStopFloor(A,getLIFT(B,C)),B,C)
e2:  writeStopFloor(A,'ELE'(B,C,D,E))='ELE'(A,B,C,setSF(D,E))
e3:  setSF('1F','SF'(A,B,C))='SF'('STOP',A,B)
e4:  setSF('2F','SF'(A,B,C))='SF'(A,'STOP',B)
e5:  setSF('3F','SF'(A,B,C))='SF'(A,B,'STOP')
e6:  readStopFloor(A,B,C)=readSF(A,getLIFT(B,C))
e7:  readSF(A,'ELE'(B,C,D,E))=getSF(A,B)
e8:  getSF('1F','SF'(A,B,C))=A
e9:  getSF('2F','SF'(A,B,C))=A
e10: getSF('3F','SF'(A,B,C))=A
e11: getLIFT('L1','SYS'(A,B,C))=A
e12: getLIFT('L2','SYS'(A,B,C))=A
e13: setLIFT(A,'L1','SYS'(B,C,D))='SYS'(A,B,C)
e14: setLIFT(A,'L2','SYS'(B,C,D))='SYS'(A,B,C)
```

図 6 LIFTstate オブジェクトの setLIFTstop の詳細記述
Fig. 6 Detailed description for setLIFTstop in LIFTstate object.

きるためである．そこで phase 2 では，オブジェクトに対してデータ構造を踏まえて等式を用いて詳細に記述する（これを詳細化オブジェクトと呼ぶ）．phase 1 までの記述内容は function に対する外部仕様であり，一方 phase 2 の記述は内部仕様である．

例として，図5に示す LIFTstate オブジェクトの構成要素の1つである setLIFTstop に対して，詳細化した記述例を図6に示す．この時，LIFT の構成（何階建てのビルでリフトが何台あるか），LIFT 状態をどのように表現するかをまず決定し，それを反映して記述する．

<phase 3>

phase 2 で記述した等式による詳細な記述内容は，recursive function 等の表現形式になり DFD に変換し視認することが困難になる．そこで他の手法によって検証することが必要となる．検証するためには，その記述内容に対して意味を与える必要がある．抽象データ型の代数的仕様の意味定義法としては始代数（initial algebra）や終代数（final algebra）などがあるが，記述内容の性質を調べる手続きが不十分である．そこで代数的仕様の計算モデルとして手続きが比較的整備されている項書換え系を採用する[16]．項書換え系は等式で記述された仕様に対して操作的な意味を与え，等式を左辺から右辺への書換え規則とみなして計算を行うモデルである．この項書換え系においては，与えられた項書換え規則の集合が停止性，合流性を満たすように新たな書換え規則を生成する手続き（完備化手続き）が存在する．この手続きによって代数的仕様の無矛盾の検証が可能となる．ここでの検証項目は大きく2つである．ひとつは項書換え系の完備化手続きによって停止性，合流性を保証するものである．もうひとつは，ユーザが任意の検証項目を等式で与えることにより，詳細化オブジェクトの記述内容（等式）が検証項目を満たしているかどうかの吟味に相当する．いずれの場合においても完備化手続きが止まらない場合がある．その場合には検証不能である．したがって検証可能範囲は，完備化手続きが停止する場合に限定される．

例として，図6に示す記述例に対して，項書換え系に基づく検証手続きを適用する際に与えた検証項目を図7に示す．図7で与えた検証項目に対する項書換え系に基づく検証手続きの実行ログを図8に示す．図6の記述例が図7の検証項目を満たすことを示している．図8において，左の window は図6の等式記述から完備化手続きによって変換した完備な項書換え規則を示している．

<phase 4>

項書換え系に基づく検証手続きによって検証が完了した詳細化オブジェクトは，あくまでも演算間の順序関係が正しく遂行されることを前提としている．外部からの要求がランダムに発生するために，演算間の実行順序に関する制御が必要となる場合がある．ここで，生成するのは Ada のタスクであるため，ある function の実行中には同一タスク内の他の function の実行は suspend される．しかし，同一タスク内の function 間にまたがる排他制御（たとえば，ある function "a" の実行後には function "b" を実行するまで function "c" を実行してはならない）に対しては，通常のタスク機能では対処できない．そこで function 間の実行順序を何らかの形で規定する必要があるが，リアルタイム SA の世界ではこれを制御

検証項目
　任意のリフトにおいて任意のフロアに対するボタンを押せば，そのフロアを停止すべきフロアの一覧に加える．
　　readStopFloor
　　　$(L, F, \mathrm{setLIFTstop}(L, F, S)) = \mathrm{STOP}$
　　　L：リフト番号変数
　　　F：リフトボタン番号変数
　　　S：リフトの状態変数

図 7 検証項目
Fig. 7 Verification item.

```
EQUATIONS
    No equations.

RULES
r1:  readStopFloor(A,B,C)->readSF(B,getLIFT(A,C))
r2:  readSF(A,ELE(B,C,D,E))->getSF(A,E)
r3:  writeStopFloor(A,ELE(B,C,D,E))->ELE(B,C,D,setSF(A,E))
r4:  getSF(1F,SF(A,B,C))->A
r5:  getSF(2F,SF(A,B,C))->B
r6:  getSF(3F,SF(A,B,C))->C
r7:  getLIFT(L1,SYS(A,B,C))->B
r8:  getLIFT(L2,SYS(A,B,C))->C
r9:  setLIFT(A,L1,SYS(B,C,D))->SYS(B,A,D)
r10: setLIFT(A,L2,SYS(B,C,D))->SYS(B,C,A)
r11: setLIFTstop(A,B,C)->setLIFT(writeStopFloor(B,getLIFT(A,C)),A,C)
r12: setSF(1F,SF(A,B,C))->SF(STOP,B,C)
r13: setSF(2F,SF(A,B,C))->SF(A,STOP,C)
r14: setSF(3F,SF(A,B,C))->SF(A,B,STOP)
r15: readSF(A,getLIFT(B,setLIFT(writeStopFloor(A,getLIFT(B,C)),B,C)))!
->STOP
r16: readSF(A,writeStopFloor(A,B))->STOP
r17: getSF(A,setSF(A,B))->STOP

(R-GEN) r7
(R-GEN) r8
(P-GEN) SYS << setLIFT
(R-GEN) r9
(R-GEN) r10
(P-GEN) getLIFT << setLIFTstop
(P-GEN) writeStopFloor << setLIFTstop
(P-GEN) setLIFT << setLIFTstop
(P-GEN) setSF << setLIFTstop
(P-GEN) SYS << setLIFTstop
(P-GEN) ELE << setLIFTstop
(R-GEN) r11
(P-GEN) STOP << setSF
(P-GEN) SF << setLIFTstop
(P-GEN) SF << writeStopFloor
(P-GEN) SF << setSF
(P-GEN) STOP << setLIFTstop
(P-GEN) STOP << writeStopFloor
(R-GEN) r12
(R-GEN) r13
(R-GEN) r14

Your system is COMPLETE .
Equations: 0 generated, 0 asserted.
Rules:     14 generated, 14 asserted.
Runtime:   35 sec.


[METIS] -> pROVE INDUCTION
<< verify properties >>

> readStopFloor(A,B,setLIFTstop(A,B,C))='STOP'.

(R-GEN) r15
(R-GEN) r16
(R-GEN) r17

   ######## PROVED ########

Equations: 8 generated, 2 asserted.
Rules:     17 generated, 17 asserted.
Runtime:   11 sec. (INDUCTION)

                screen                                   spool
USER : ohsuga   editor_buffer_window/58
SIMPOS Version 5.2                   09-May-90 Wednesday 15:18:08
```

図 8 フェーズ3における項書換え系による検証過程
Fig. 8 Verification process by term rewriting system in Phase 3.

スペックである状態遷移図で記述する．この状態遷移図は，人手で作成するために正当性は保証されていない．そこで本手法では，正当性の保証された状態遷移図を生成するために，時制論理の定理証明系をプログラム生成に適用した Wolper らの手法[23] を採用する[9]．すなわち，Ada における内部データに対する function のアクセス順序を，時制論理で記述した仕様から定理証明系によって生成するものである．Wolper の手法はタブロー法に基づいている．タブロー法とは与えられた論理式が恒真であるかどうかを証明するために，状態遷移図を生成する．タブロー法では，与えられた論理式が恒真でなければ，空の状態遷移図を生成する．すなわち，得られた状態遷移図は，与えられた論理式に対するすべてのモデルとなっている．状態遷移図においては，アーク上の論理式はアークの元にあるノードにおいて真となることを示している．したがって，論理式に何を割り付けるかによって状態遷移図の解釈が異なってくる．ここでは，論理式に対して，データをアクセスする function を割り付ける．それにより状態遷移図は，function の実行順序を示していることになる．

例として，LIFTstate オブジェクトを構成する function 間における実行順序に関する制御要求を，図9に示すように時制論理で示す．たとえば，最初の文

□ (Set LIFT stop ⊃ (¬ set Direction U get Direction ）
□ (Set Floor stop ⊃ (¬ set Direction U get Direction ）
□ (Set LIFT stop ⊃ ○ ◇ get Direction ）
□ (Set Floor stop ⊃ ○ ◇ get Direction ）

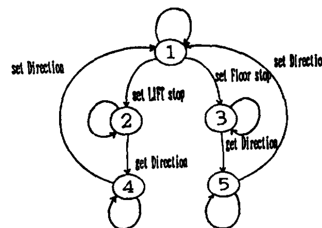


図 9 時制論理および対応する定理証明系による状態遷移図
Fig. 9 Temporal logic specification and generated state transition diagram by theorem prover.

```
procedure main is
task LIFTstate is
    entry setLIFTstop     (L: in LIFTNo.; B: in LIFTButoonNo.;
                           S: in LIFTstate; S: out LIFTstate);
    entry setFloorstop    (--);
    entry getDirection    (--);
    entry setDirection    (--);
end LIFTstate;
function setLIFT (NS: in LIFTNo.state; L: in LIFTNo.; S: in LIFTstate)
        return LIFTstate;
function writeStopFloor   (B: in LIFTButtonNo.; NS: in LIFTNo.state)
        return LIFTNo.state;
function getLIFT (L: in LIFTNo.; S: in LIFTstate)
        return LIFTNo.state;
task body LIFTstate is
    N:integer:=1    --node variable in state transition diagram
begin
    loop
        select
            when N=1    =>
                accept setLIFTstop (L: in LIFTNo.; B: in LIFTButoonNo.;
                                    S: in LIFTstate; S: out LIFTstate) do
                    NS:=getLIFT(L,S);
                    NS:=writeStopFloor(B,NS);
                    S:=setLIFT(NS,L,S);
                    N:=2;    --update node variable
                end setLIFTstop;
        or
            when N=1    =>
                accept setFloorstop (--) do
                    --
                    N:=3;
                end setFloorstop;
        or
            when N=2 or N=3 =>
                accept getDirection (--) do
                    --
                    N:=N+1;
                end getDirection;
        or
            when N=4 or N=5 =>
                accept setDirection (--) do
                    --
                    N:=1;
                end setDirection;
        end select;
    end loop;
end LIFTstate;
function setLIFT (NS: in LIFTNo.state; L: in LIFTNo.; S: in LIFTstate)
        return LIFTstate is
begin
    --
end setLIFT;
function writeStopFloor   (B: in LIFTButtonNo.; NS: in LIFTNo.state)
        return LIFTNo.state is
begin
    --
end writeStopFloor;
function getLIFT (L: in LIFTNo.; S: in LIFTstate)
        return LIFTNo.state is
begin
    --
end getLIFT;
begin
    --main program
end main;
```

図 10 Ada タスクの出力例
Fig. 10 Output example in Ada task.

は，「setLIFTstop を実行した時には，getDirection を実行するまでは setDirection を実行してはならない」ことを示している．図9で示す時制論理仕様から，定理証明系によって得られた状態遷移図を図9に示す．ユーザの立場では，図9に示すような function におけるすべての順序関係を，直接，状態遷移図を用いて記述するよりは，2〜3 の function 間の順序関係で与えた方が，仕様記述作業が容易である．

ここで，代数的仕様と時制論理仕様との関係について述べておく．時制論理の記述対象である function は，代数的仕様から得られたものである．ユーザはこれらの function を見ながら，時制論理仕様を与えることになる．すなわち，両仕様を同時に与えているわけではないので，両者の間に矛盾を生じることはない．

<phase 5>

詳細化オブジェクトにおけるデータと，それに関わる function および phase 4 で作成した状態遷移図から，Ada のタスク骨格を自動的に生成する．DFD のストアは，このタスクに情報隠ぺいされている内部データとなり，ストアに対する各 function は，タスクの各 accept 文となる．すなわち，複数の外部からの呼出し (entry-call) に対して，条件付きの accept 文で待機している形態をとる．when 文の後の条件式が表す数値は，状態遷移図のノード番号を示す．accept 文の最後で，状態遷移図に従ってノード番号を更新する．また，等式表現は Ada の function 形式に変換される．なお，function としての出力データがオブジェクトの外部に出る時は，そのデータを引数として entry-call の形式にする．(該当するデータを入力とする function が属す他のオブジェクトにおいて，accept 文として対応する．どの function と，どの function が，どのデータを受け渡すかという function 間のデータの受け渡し関係は，DFD より得られる.)

例として，LIFTstate オブジェクトを対象に，図9に示す状態遷移図の情報を付加して生成した Ada のタスク形式を**図 10** に示す．図 10 において，when の文の後の条件式が表す数値は，図9の状態遷移図中のノード番号に対応する．

3. 評　　価

本手法の制約として次の項目があげられる．

①バブルの出力をひとつとしていること．

②制御スペックの記述をタスク内の function の順序関係に限定していること．

③検証系からみた制約として，代数的仕様に関しては検証する際に完備化手続きが停止しないことがあ

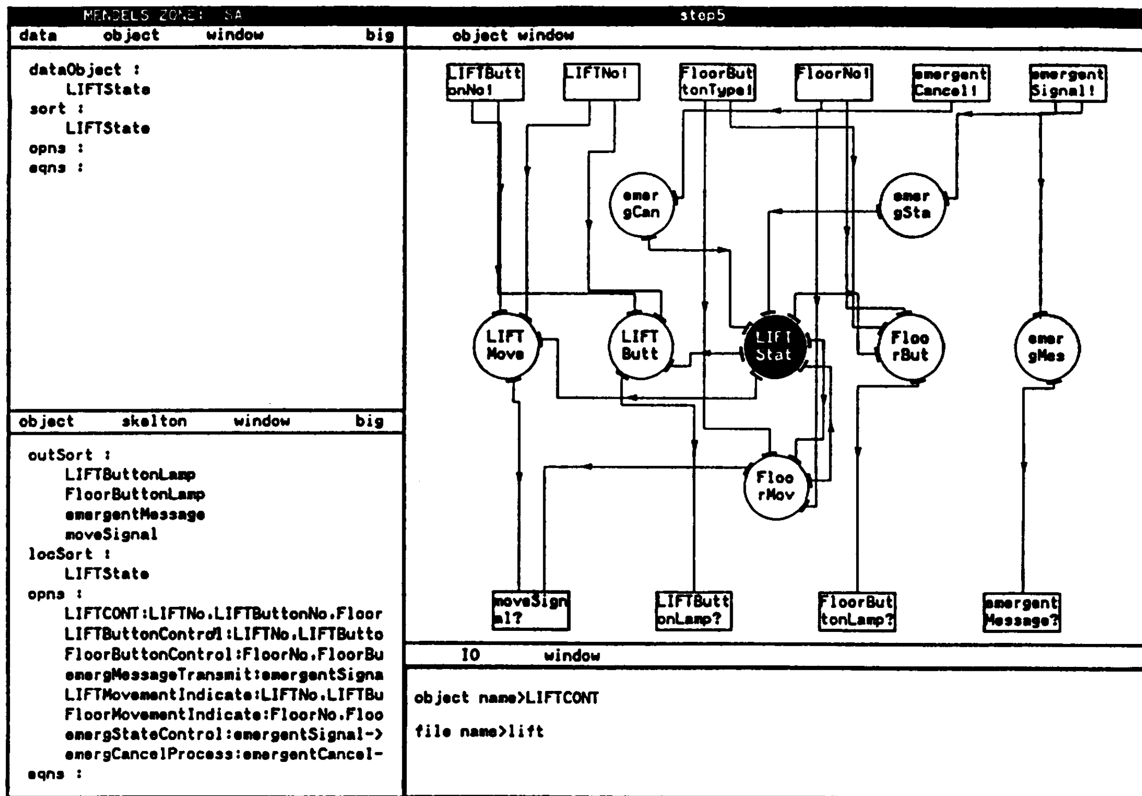


図 11 フェーズ1における MMI
Fig. 11 MMI in Phase 1.

る．この時には検証不能となる．(DFD によって視認可能な場合もある.)

なお，本論文で述べた手法は現在実装中であるが，**図 11** に示すように DFD を表示する window と代数的仕様によって記述する window がある．この2つの window の情報は一貫性を保っており，ユーザがどちらかを修正すればシステムが自動的に片方の window の表示を修正することになる．また，ストアや中間データ，外部との入出力データの詳細を記述する window があり，ストアの回りのバブルを展開することによって得られた等式による記述内容のなかで，ストアを直接アクセスする function は，システムによって自動的にこの window に登録される．形式的仕様記述を実レベルに適用する場合，しかも一般ユーザに利用させるには支援環境の充実が必須である．すなわち，実務現場での受け入れ可能性の観点では，形式的仕様において経験の少ない人にとって，どの程度，使い勝手の良い支援ツール，環境が整備されているかが，評価のポイントになる．この観点で述べるならば，きめの細かい設計プロセスを支援環境が誘導 (navigate) しており，図式仕様記述との併用も，ユーザにとっては有効である．すなわち，DFD において，出力が2つ以上になったバブルをシステムが選択し，そのバブルを展開することをユーザに対してシステムが誘導する．したがってユーザは，システムの誘導に従って，バブルの仕様記述を行うことによって仕様の詳細化を進めることができる．同時に，システムがオブジェクトとして，詳細化した仕様をまとめることによってユーザの設計活動を支援している．また，代数的仕様におけるシグニチャや等式をシステムが DFD に変換することにより，仕様記述内容の視認を可能としている．

さらに，実務現場での受け入れ可能性を高めるためには次の項目の整備が要求される．

①効率の良い仕様デバッガ

②仕様ライブラリの充実

③検証条件生成系の実現

④性能評価系の実現

特に①については，著者らは，時制論理の仕様について，デバッギング手法を既に提案している[15]が，今

後その手法も統合化していく予定である．

4. 他の研究との比較

本章では次の4点から他の研究との比較を行う．形式的アプローチの統合化手法，分析フェーズから設計フェーズへの橋渡し手法，OOD と並行プログラム，支援環境についてである．まず，形式的アプローチの統合化手法としては[4),7),13),14),17),19)]などがある．いずれの手法においても，機能・データの記述に VDM や代数的仕様を用いており，同期，並行性の記述には，時制論理，ペトリネット，CCS, CSP を用いている．このように，種々の手法が採用されているが，必ずしも検証手段，仕様展開プロセス，プログラム生成系が整備されているとは言い難い．また **LOTOS** は言語として両面をサポートしているが，検証系については今後の課題である[12)]．

次に，分析フェーズと設計フェーズの橋渡しについて述べる．JSD[2)] は設計プロセスの中に implementation stage が存在する．DARTS[6)] では DFD からいくつかの基準（たとえば同じ stimulus によって trigger される transformation は同一タスクにする）によってタスク分割を行っている．リアルタイム SD[8),20)] においては，central transformation の考え方でモジュール構造を実現している．本手法では OOD を採用して，データとそれに関わる function をオブジェクトして抽出している．Ward は，リアルタイム SA の枠組で OOD を支援することが可能であると述べている[21)]．すなわち，最下位層の DFD においてストアを内部データとし，ストアを中心として回りのバブルを function とすることによってオブジェクトとみなそうとするものである．本手法とは近いが，本手法では機能展開を繰り返すことによって体系的に function を抽出していく点が異なっている．

OOD を用いて Ada のタスクを生成する研究には OOSD[22)], HOOD[11)], OOD[1)] などがある．これらの研究では，生成される Ada のタスクにおいて，function 間の排他問題を扱っていないことが本手法との相違点である．JSD から Ada プログラムを生成する研究がある[2)]．これは JSD におけるネットワーク仕様（network specification）から Ada のタスクを生成するものであるが，タスク内の function 間の順序関係は規定していない．

また，形式的仕様記述支援ツールの点においては，構文エディタ仕様，ライブラリ，デバッガ，検証条件生成系，直接実行系など数多くの支援ツールに関する研究がある．本手法で提供している支援ツールの中で，他にはないツールとしては，代数的仕様と DFD の両方向一貫性支援ツールおよび DFD の機能展開に関する誘導（ナビゲータ）の2つをあげることができる．

5. む す び

本論文では，形式的仕様記述を実レベルの問題に適用する際のいくつかの課題を解決することを試みた．その時のポイントは

①時制論理と代数的仕様を組み合わせたこと．
②リアルタイム SA から OOD への自然な流れを規定したこと．
③きめの細かい設計プロセスを規定したこと．
④OOD の考え方により並行プログラムを生成したこと．

である．きめの細かい設計プロセスにおける中間生成物の形式化においては見通しを得ることができたが，作業項目の形式化，知識化は未解決であり，今後の課題である．作業項目も知識化できれば，自動プログラミングへ一歩大きく近づくことになる．また，設計プロセス自身の再利用も可能になる．もうひとつの課題としては，実際のシステムに適用することにより，人間の知的活動が必要な部分と計算機で対処可能な部分を明確に分類することにある．

形式的仕様記述の支援環境としては効率的なデバッギング・ツール，仕様ライブラリなど充実しなければならない項目が残っている．

謝辞　本研究の一部は第5世代コンピュータプロジェクトの一環として行われた．

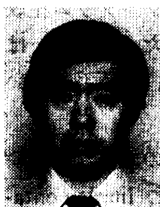
日頃，ご指導いただく（株）東芝　システム・ソフトウェア技術研究所の西島誠一所長，河野毅部長，大筆豊部長に感謝いたします．

参 考 文 献

1) Booch, G.: Object-Oriented Development, *IEEE Trans. Softw. Eng.*, Vol. SE-12, No. 12 (1986).
2) Cameron, J.: *JSP & JSD: The Jackson Approach to Software Development*, IEEE Computer Society (1989).
3) DeMarco, T.: *Structured Analysis and System Specification*, Yourdon, New York (1978).
4) Folkjar, P. and Bjorner, D.: A Formal Model of a Generalized CSP-like Language, *IFIP '80* (1980).
5) 古川，本位田，大須賀，津田：代数的仕様記述

と図式仕様記述の相補的役割について—複眼的システムモデル—，情報処理学会論文誌，Vol. 31, No. 2, pp. 182-193 (1990).
6) Gomaa, H.: A Software Method for Real-Time Systems, *Comm. ACM*, Vol. 27, No. 9 (1984).
7) Hankley, W. and Peters, J.: Temporal Specification of Ada Tasks, *HICSS-23* (1990).
8) Hatley, D.: The Use of Structured Methods in the Development of Large Software Based Avionics Systems, *AIAA/IEEE Sixth Digital Avionics Systems Conf.* (1984).
9) Honiden, S. et al.: An Application of Structural Modeling and Automated Reasoning to Real-Time Systems Design, *The Journal of Real-Time Systems*, Vol. 1, No. 3 (1990).
10) 本位田，大須賀，内平：代数的仕様と時制論理によるリアルタイムSAの形式的支援，情報処理学会ソフトウェア工学研究会資料，90-73 (1990).
11) Hood Working Group: Hood Reference Manual, European Space Agency (1989).
12) ISO: Information Processing Systems—Open Systems Interconnection—LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, ISO 8807 (1989).
13) Kramer, B.: SEGRAS—A Formal and Semi-graphical Language Combining Petri Nets and Abstract Data Types for the Specification of Distributed Systems, *9th ICSE* (1987).
14) Meiling, E.: A Spreadsheet Specification in RSL—An Illustration of the RAISE Specification Language, *ESPRIT '87 Achievements and Impact*, pp. 466-479, North-Holland (1987).
15) 西村ほか：時制論理に基づく仕様記述とそのデバッグ環境，第38回情報処理学会全国大会論文集 (1989).
16) 大須賀，坂井：項書換えシステムに基づく帰納的定理証明，電子通信学会コンピュテーション研究会，COMP 87-36 (1987).
17) Pletat, U.: Algebraic Specification of Abstract Data Types and CCS: An Operational Junction, *Protocol Specification, Testing and Verification*, Elsevier Science Publishers (1986).
18) Problem set, *Proc. of Fourth International Workshop on Software Specification and Design*, CS Press, Los Alamitos, Calif. (1987).
19) Vautherin, J.: Parallel Systems Specifications with Coloured Petri Nets and Algebraic Specifications, *LNCS 266* (1987).
20) Ward, P.: The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing, *IEEE Trans. Softw. Eng.*, Vol. 12, No. 2 (1986).
21) Ward, P.: How to Integrate Object Orientation with Structured Analysis and Design, *IEEE Softw.*, Vol. 6, No. 2 (1989).
22) Wasserman, A. I. et al.: The Object-Oriented Structured Design Notation for Software Design Representation, *IEEE Comput.*, Vol. 23, No. 3 (1990).
23) Wolper, P. et al.: Synthesis of Communicating Processes from Temporal Logic Specification, *ACM TOPLAS*, Vol. 6, No. 1 (1983).

(平成2年9月18日受付)
(平成3年12月9日採録)



本位田真一（正会員）
1953年生．1976年早稲田大学理工学部電気工学科卒業．1978年同大学院理工学研究科電気工学専攻修士課程修了．工学博士．同年(株)東芝入社．現在，同社システム・ソフトウェア技術研究所主任研究員．早稲田大学非常勤講師．主として，ソフトウェア工学，人工知能の研究に従事．ソフトウェアの基礎理論に興味をもつ．1986年情報処理学会論文賞受賞．著書「ソフトウェア開発のためのプロトタイピング・ツール」(共著)，「KE養成講座② エキスパートシステム基礎技術」(共著)，「オブジェクト指向システム分析」(共訳)．人工知能学会，日本ソフトウェア科学会，IEEE, AAAI 各会員．

大須賀昭彦（正会員）
1958年生．1981年上智大学理工学部数学科卒業．同年(株)東芝入社．1985〜89年(財)新世代コンピュータ技術開発機構に出向．現在(株)東芝システム・ソフトウェア技術研究所研究主務．定理の自動証明，形式的仕様・プログラムの検証などに興味を持つ．1986年度情報処理学会論文賞受賞．日本ソフトウェア科学会，EATCS 各会員．

内平　直志（正会員）
1959年生．1982年東京工業大学情報科学科卒業．同年(株)東芝入社．現在同社システム・ソフトウェア技術研究所勤務．時相論理の実用的応用システムを開発中．並列プログラミング言語に興味を持つ．1986年度情報処理学会論文賞受賞．ソフトウェア科学会会員．