

リフレクション機構を持った仕様記述言語 RLOTOS と その応用

広井 武 佐伯 元司

東京工業大学 工学部

形式的仕様記述言語 LOTOS に Reflection の概念を導入した言語 RLOTOS (Reflective LOTOS) がある。RLOTOS はオブジェクトレベルとメタレベルの2つの概念を持っている。メタレベル上にはオブジェクトレベルの記述を解釈実行するインタプリタが存在し、このインタプリタにアクセスすることでオブジェクトレベルの動作を変更することができる。本論文では、RLOTOS の概要を述べ、次に RLOTOS を用いて通信プロトコルをはじめとした種々のシステムの形式的な仕様を記述し、リフレクション機構を持った言語を仕様記述に用いることの利点を議論する。

Reflective Language RLOTOS and Its Application to Formal Specifications

Takeshi Hiroi Motoshi Saeki

Dept. of Electrical and Electronic Engineering, Tokyo Institute of
Technology, Japan

We have proposed the formal specification language RLOTOS (Reflective LOTOS) which is an extension of LOTOS. It has *reflective computation* mechanism. Similary to other reflective languages, RLOTOS has two level architecture — *object level* and *meta level*. On the meta level, an interpreter which executes the object level description exists, and we can change the object level behaviour by controlling the interpreter. In this paper, firstly, we present the outline of RLOTOS, and then we formally specify various kinds of systems using RLOTOS. Finally, we discuss the benefits of reflective languages to be used for constructing formal specifications.

1 まえがき

LOTOS(Language of Temporal Ordering Specification)は、その形式的意味をプロセス代数と多ソート代数におく形式的仕様記述言語であり、OSI(Open System Interconnection)の参照モデルに基づいた通信プロトコルを記述するために開発され、1988年にISOで国際標準化がなされた[1]。LOTOSは、並列性や非決定性、同期などの記述が行なえるという豊富な記述力を持っているだけでなく、ラベル付き遷移規則によって操作的な意味が与えられており、LOTOS記述を実行させることもできる。そのため、通信プロトコルだけでなく他のシステムの記述にも用いられ[2, 3]、その適用性が検討されている。

リフレクション(自己反映計算)は、そのプログラムの実行途中に、そのプログラム自身が実行過程や実行状態をアクセスし、かつ変更することを可能にするためのメカニズムでLISPをはじめ[4]、オブジェクト指向型言語[5]や論理型言語[6]へ導入した例が報告されている。我々は、LOTOSの実行可能性に注目し、LOTOS言語にリフレクション機構を導入した言語RLOTOSを提案した[7]。他のリフレクション言語と同様に、RLOTOSもメタレベル、オブジェクトレベルの2つの概念を持ち、メタレベルでの記述がオブジェクトレベルの記述の実行を制御する。我々のLOTOSへのリフレクション機構導入の目的は、LOTOSの記述能力の向上ではなく、オブジェクトレベルとメタレベルに分離して記述することにより、理解性のよい形式的仕様が記述できるようにすることにある。本稿では、RLOTOSを用いて種々のシステムの形式的仕様の記述し、その仕様の理解性について議論する。

2 RLOTOS のリフレクティブ機構

2.1 LOTOS

LOTOSによる記述は、動的部分を記述するbehavior expression(動作式)の定義と静的データ部分を記述する抽象データ型の定義からなる。前者はプロセス代数、後者は多ソート代数によってそれぞれ意味付けされている。behavior expressionはLOTOSのプロセスを規定しており、外部から観測可能なイベント列として定義される。また、イベントを用いてデータを受け渡すこともできる。データの仕様は抽象データ型言語ACT-ONEで記述される。表1はLOTOSで使用できる主なオペレータを表している。

2.2 RLOTOS

RLOTOSはLOTOSの動的記述を行なうbehavior expression部分にリフレクション機構を導入した言語である。図1にRLOTOSの構造を示す。メタレベル上にはLOTOSのプロセスとして記述されたインタプリタが存在し、オブジェクトレベルのLOTOS記述を解釈実行する。メタレベル上のインタプリタは、オブジェクトレベルの記述を、LOTOSで扱うことのできる抽象データ型言語ACT-ONEの項表現に変換したものを入力とする。オブジェクトレベルとメタレベルの動作は互に対応付けられており、

表 1: 基本的な LOTOS オペレータ

オペレータ	機能
$a;B$	a を実行して B へ
$B1 \mid B2$	$B1$ か $B2$ を選択
$B1 \parallel B2$	$B1$ と $B2$ が同期
$B1 \parallel\!\!\!\parallel B2$	$B1$ と $B2$ が独立
$B1 \llbracket g_1, \dots, g_n \rrbracket B2$	ゲートについて同期
$B1 \rangle B2$	$B1$ の途中に $B2$ へ移る
stop	inaction

a : イベント
 $B, B1, B2$: behavior expression
 $g_1, \dots, g_n$: ゲート

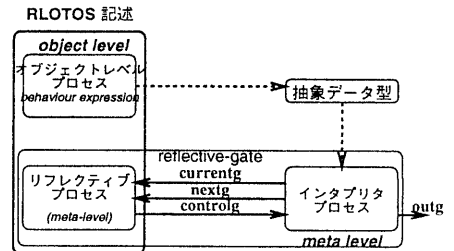


図 1: RLOTOS の構造

RLOTOSではオブジェクトレベルの動作はメタレベル上のインタプリタの出力ゲート $outg$ を通じて出力される。

RLOTOSでは、リフレクティブプロセスとリフレクティブゲートという特別のプロセスとゲートを用いてメタレベルへのアクセスを行ない、リフレクティブ計算を行なう。リフレクティブプロセスからのインタプリタへのアクセスは、LOTOSの同期の概念を用いたプロセス間の通信によって行うことができる。

ラベル付き遷移システムに基づいたLOTOSの操作的意味論に従って、RLOTOSではリフレクティブゲートとして図1に示した3種類のゲートを持っている。ユーザはこれら3種類のリフレクティブゲートを用いてインタプリタを制御できる。以下に3種類のリフレクティブゲートで扱うデータを示す。

currentg 現在のbehavior expressionの状態。

nextg 次に起こり得るイベントとそのイベントが起こった場合の次状態との対のリスト。

controlg 次に起こるイベントとそのイベントが起こった場合の次状態との対。

リフレクティブ計算を行なう際、リフレクティブプロセスではこれら3種類のリフレクティブゲートを以下のプロトコルに従って使用する。

1. インタプリタより $currentg$ を通じてbehavior expressionの現在の状態(ソート $Bexp$)を受け取る。
2. インタプリタより $nextg$ を通じて次に起こり得るイベント(ソート Act)とそのイベントが起こった場合の次

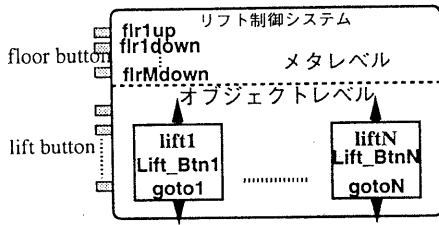


図 2: Lift システム

状態 (ソート Bexp) との対 (ソート ActPair) のリスト (ソート ActPairList) を受け取る。

3. controlg を通して次に起こるイベントとそのイベントが起こった場合の次状態との対 (ソート ActPair) をインタプリタへ送る。

インタプリタは controlg を通して渡されたイベントをオブジェクトレベルで起こるイベントとして outg を通して出力し、次の状態について上記のプロトコルを繰り返す。

3 記述例

3.1 Lift の制御問題

IWSSD-4 [8] で共通問題として取り上げられた問題の一つに Lift の制御問題がある。この問題で扱っている Lift システムは N 台の lift で構成されている。各 lift は各階に対応した lift button を持っており、lift button が押された各階を順番に訪れる。またこの建物は M 階からなっており、各階には lift を呼び up, down のそれぞれの方向へ進むことを要求する 2 つの floor button が付いている。図 2 は Lift システムを示している。Lift システムは、個々に独立して動作する N 台の lift と N 台の lift を制御する部分とに分離して考えることができる。RLOTOS でこの問題を仕様化する際は個々の lift がそれぞれ独立に動作する仕様をオブジェクトレベルのプロセス、N 台の Lift を制御する部分の仕様をメタレベルのプロセスとして記述することで個々の lift の仕様とシステムの制御部分の仕様とを分離して記述することが可能である。以下に lift 問題の RLOTOS による記述を示す。簡単のために lift システムには非常 button はついていないものとしてある。

```

specification Lift[Lift_Btn1, goto1, ..., gotoN, ..., flrMdown]:noexit
library Boolean, NaturalNumber, Set
endlib
type      direct is
sorts     dir
ops       up, down, idle :-> dir
endtype  (* direct *)
type      BtnSet is NaturalNumber, direct, Boolean
sorts     Bset
ops       Get_flg      : dir, Nat, Bset -> Nat
          Get_dir      : dir, Nat, Bset -> dir
          Get_set       : dir, Nat, Bset -> Bset
eqns
endtype  (* BtnSet *)
behaviour
  lift1[Lift_Btn1, goto1](Succ(0), up, {}))
  ||| liftN[Lift_BtnN, gotoN](Succ(0), up, {})
where
  process lift1[Lift_Btn1, goto1](flr:Nat, drct:Dir, DB:BSet):noexit:=
    Lift_Btn1?x:Bset;
    goto1?flr:Get_flg(drct, flr, Insert(x, DB)); Get_dir(drct, flr, Insert(x, DB));
    lift1[Lift_Btn1, goto1](Get_flg(drct, flr, Insert(x, DB)),
      Get_dir(drct, flr, Insert(x, DB)), Get_set(drct, flr, Insert(x, DB)))
  endproc  (* lift1 *)

process meta_level[currentg,nextg,controlg]:noexit:=

```

```

lift_controller[currentg,nextg,controlg]({})
where
  process lift_controller[currentg,nextg,controlg](MDB:Cset):noexit:=
    currentg?x:Bexp;nextg?y:ActPairList;
    lift_controller_i[currentg,nextg,controlg](x,y,MDB)
  where
    process lift_controller_i[currentg,nextg,controlg]
      (x:Bexp,y:ActPairList,MDB:Cset):noexit:=
      choice a:Act,e:Bexp[]
      [[IsIn(pair(a,e),Select(y,MDB))]->
        (controlg?pair(a,e):lift_controller[currentg,nextg,controlg](MDB))
      []controlg?pair('flrUp',x);
        lift_controller[currentg,nextg,controlg](Ins(1up,MDB))
      []controlg?pair('flr2down',x);
        lift_controller[currentg,nextg,controlg](Ins(2down,MDB))
      []controlg?pair('flrMdown',x);
        lift_controller[currentg,nextg,controlg](Ins(Mdown,MDB))]
      [][NotIn(pair(a,e),Select(y,MDB))]->
        lift_controller_i[currentg,nextg,controlg](x,y,MDB)
      endproc
    type      Ex_Lift is Bexp,ActSetSet
    type      Select,EventSelect : ActPairList,Cset -> ActPair
    ops       Reachable : Act,Cset -> Bool
              MoveEventSelect : ActPairList -> ActPairList
              ChangeEvent : Act,Cset -> Act
              ChangeBexp : Act,Bexp,Cset -> Bexp
    eqns      forall i,NextPairList:ActPairList,ci,restDB:Cset,e1:Act,b1:Bexp
              ifsort
                Select(i,ci) = EventSelect(MoveEventSelect(i,ci));
                EventSelect(<,ci) = <;
                Reachable(e1,ci) =>
                  Reachable(e1,b1),NextPairList(ci)
                EventSelect(Insert(pair(e1,ci),NextPairList),ci)
                  = Insert(pair(ChangeEvent(e1,ci),ChangeExp(e1,b1,ci)),
                    EventSelect(NextPairList,ci));
                not(Reachable(e1,ci)) =>
                  EventSelect(Insert(pair(e1,b1),NextPairList),ci)
                  = Insert(pair(e1,b1),EventSelect(NextPairList,ci));
                ...
    endtype
  endproc
endspec

```

オブジェクトレベルのプロセスは lift の個々の動作を示す lift1, lift2, .. のプロセスをインタリーブオペレータを用いて記述する。1 台目の lift の中で lift button が押され、その階へ移動する要求が出ることは lift1 プロセス中で Lift_Btn1 イベントが生じることに対応する。そして lift はその要求に従って順番に lift 内の lift button によって要求された各階を訪れる。目的階への到着は goto1 イベントが生じることに対応している。lift プロセスの flr, drct, DB の各変数はそれぞれ現在いる階、進んでいる方向、要求がまだ処理されていない階の集合を扱っており、lift の状態を表している。また、flr, drct, DB について ACT-ONE の関数 Get_flg, Get_dir, Get_set を用いてそれぞれの次状態を求める。

メタレベル上のインタプリタを制御するリフレクティブプロセスには N 台の独立に動作する lift の制御部分を記述する。各階の floor button からの要求は制御部分であるリフレクティブプロセスで管理する。メタレベル上の lift_controller プロセスでは、常に各階の floor button から要求を出すイベントを実行可能にする。例えば 1 階の up button からの要求の場合は flrUp イベントに対応している。これら各階からの要求は変数 MDB で管理する。また N 台の lift の動作を監視しており、floor button からの要求を満たすことができる lift の中から非決定的に選択し lift の動きを変更して要求のある階に止めることができる。lift_controller プロセスは nextg より受けた次に起こり得るイベントと次状態の対のリスト y から関数 Select を用いて各階からの要求を考慮した際の、次起こり得るイベントと次状態の対のリスト Select(y,MDB) を求める。Select から呼ばれる MoveEventSelect はオブジェクトレベルで起こり得るイベントの中から goto イベントを取り出し、Reachable は goto イベントが各階からの要求を満たし得るかどうかを判定する。もし満足していれば、ChangeEvent, ChangeExp を用いてそれぞれその lift の動作と動作を変更した lift プロセスの状態を変更する。これ以降、わかりやすさのために、例えばイベント name?:type_{la} の ACT-ONE

表現を 'name?s:type!a' , behavior expression $B1 \parallel B2$ の ACT-ONE 表現を ' $B1 \parallel B2$ ' と記す。

3.2 踏みきり

高信頼性が要求されるシステムの一つに踏みきりシステムがある。この種のシステムでは fault が起こった際にシステムが危険な状態に陥るのを回避する fault-tolerant 機能が重要である。Petri net に時間の概念を導入した Time Petri net を用いて fault-tolerant 機構を持った踏みきりシステムの仕様の記述が行なわれている [9]。この踏みきりシステムは、コンピュータに fault が起こって電車が踏みきりを通過し終る前に遮断機が上がり危険な状態になると、遮断機を下ろして安全な状態に回避する fault-tolerant 機能を含んでいる。このようなシステムの記述を行なう際、fault-tolerant 機構の記述がシステム本来の仕様に埋め込まれ、可解性が悪くなるという欠点がある。

このシステムを RLOTOS を用いて記述する場合 fault が起こらない通常のシステムの動作と、fault が起こった際にシステムが行なう fault-tolerant 動作とを分離して記述することができる。以下に踏みきりシステムの RLOTOS による記述を示す。

```
specification crossing[down,up,info,signal,sensor,ndone]:noexit
type
  Signal_color is
  sorts
    color
  opns
    RED, GREEN      : -> color
  endtype
  type
  sorts
    Signal_val is
  opns
    val
  endtype
  type
  sorts
    Sensor_val is
  opns
    APPROACH, OUT   : -> sense
  endtype
behaviour
  (Computer[sensor,info,signal]
  [[info,signal]] Cross[down,up,info,signal]
  [[sensor,signal]] Train[sensor,signal]
  where
  process Computer[sensor,info,signal]:noexit :=
    Sensor[sensor, info]
    [[Signal[signal]]
  where
  process Sensor[sensor,info]:noexit :=
    sensor!APPROACH;info!DOWN;sensor!OUT;info!UP;stop
  endproc
  process Signal[signal]:noexit :=
    signal!GREEN;Signal[signal]
  endproc
  process Cross[down,up,info,signal]:noexit :=
    info!DOWN;down;signal!GREEN;info!UP;up;Cross[down,up,info,signal]
  endproc
  process Train[sensor,signal]:noexit :=
    sensor!APPROACH;signal!GREEN;sensor!OUT;stop
  endproc
  process meta_level[currenttg,nexttg,controlg]:noexit :=
    comp_fault[currenttg,nexttg,controlg](.up,.up.off)
  where
  process comp_fault[currenttg,nexttg,controlg]
    (cmp,crs,flt:Symbol):noexit :=
    currenttg?x:Bexp;nexttg?y:ActPairList;
    fault_tolerant_i[currenttg,nexttg,controlg](cmp,crs,flt,x,y)
  where
  process comp_fault_i[currenttg,nexttg,controlg]
    (cmp,crs,flt:Symbol,x:Bexp,y:ActPairList):noexit :=
  choice a:Act,e:Bexp[]
  [[isIn(pair(a,e),y)]->
    (([a<>'info:DOWN']->controlg!pair(a,e);
      comp_fault[currenttg,nexttg,controlg](.dwn,crs,flt)
    [a<>'info:UP']->controlg!pair(a,e);
      comp_fault[currenttg,nexttg,controlg](.up,crs,flt)
    [a<>'down']->controlg!pair(a,e);
      comp_fault[currenttg,nexttg,controlg](cmp,.dwn,flt)
    [a<>'up']->controlg!pair(a,e);
      comp_fault[currenttg,nexttg,controlg](cmp,.up,flt)
    [cmp=.dwn]->
      ([crs=.up]->
        ([a<>'down']->controlg!pair('down',stop);stop
        [a<>'down']->controlg!pair('down',x);
          comp_fault[currenttg,nexttg,controlg](cmp,.dwn,flt)))
      [flt=on]->
        ([crs=.dwn]->
          ([a<>'up']-> controlg!pair('up',stop);stop
          [a<>'up']-> controlg!pair('up',x);
            comp_fault[currenttg,nexttg,controlg](cmp,.up,flt))
          [crs=.up]->controlg!pair('ndone',x);
            comp_fault[currenttg,nexttg,controlg](cmp,crs,end))
        [flt=off]->controlg!pair('info:UP',x);
          comp_fault[currenttg,nexttg,controlg](cmp,crs,on)(*fault *)
        [([a<>'info:DOWN']and(a<>'info:UP')and(a<>'down')and(a<>'up'))->
          controlg!pair(a,e);
            comp_fault[currenttg,nexttg,controlg](cmp,crs,flt))
    ]]
```

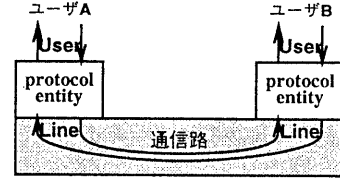


図 3: protocol entity(abracadabra-protocol)

```
[] [NotIn(pair(a,e),y)]->
  comp_fault_i[currenttg,nexttg,controlg](cmp,crs,x,y)
endproc
endproc
endproc
endspec
```

オブジェクトレベルのプロセスとして通常の踏みきりシステムの動作を記述する。踏みきりのシステムは遮断機の制御を行なう Computer プロセスと遮断機の状態を表す Crossing プロセス、電車の通過状態を表す Train プロセスからなる。この場合、遮断機の制御部分を表している Computer プロセスはゲート sensor によって電車の接近を感知したら (sensor!APPROACH が生じる) ゲート info から遮断機に DOWN 命令を送る。Crossing プロセスは DOWN 命令を受けとり遮断機は down する。同様にして Computer プロセスは電車が踏みきりを出ることを感知したら (sensor!OUT が生じる) 遮断機を up させる。また Crossing プロセスは Computer プロセスを通して、遮断機が down すると電車に対して青信号を送る (signal!GREEN が生じる)。

一方、メタレベルのリフレクティブプロセスでは fault が起こり、それを回避する fault-tolerant 部分を記述する。但し、ここでは fault は高々 1 回のみ起こるものとしている。リフレクティブプロセスでは踏みきりシステムの状態としてコンピュータの遮断機の制御状態、遮断機の実際の状態、fault の出現状態を表す変数 cmp, crs, flt を持つ。cmp, crs は up, down 状態としてそれぞれ .up, .down を持つ。flt は fault が起こっていない状態、fault が起こっている状態、fault-tolerant が行なわれた状態としてそれぞれ off, on, end を持つ。メタレベルの comp.fault プロセスはコンピュータが down 命令を送っていて遮断機が降らない場合は遮断機を降ろし、fault が起こって遮断機が降りている時は遮断機を上げる。

3.3 通信プロトコル

通信プロトコルの一つに abracadabra-protocol がある。abracadabra-protocol は実際に LOTOS による記述が行なわれている [10]。また、拡張 LOTOS を用いて記述された例もある [11]。図 3 は abracadabra-protocol の protocol-entity を示したものである。abracadabra-protocol ではユーザは protocol-entity を通して他のユーザと通信を行なう。この際、通信路における情報の誤りや喪失等に対するエラー訂正はすべて protocol-entity で行なわれ、ユーザには隠蔽されている。abracadabra-protocol では、以下の方法で通信を行なう。protocol-entity はユーザよりコネクション要求 (ConnReq) を受けると通信路を通して相手にコネクション要求 (CR) を送る。コネクション要求を受けた protocol-entity はユーザにそのことを知らせる (ConnInd)。知らさ

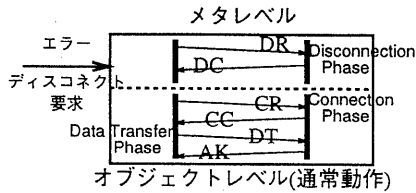


図 4: abracadabra protocol

れたユーザからコネクション応答 (ConnResp) を受け取ると、通信路を通してコネクション要求者にコネクション確認 (CC) を送る。コネクション要求者はこれを受け取ってユーザに ConnConf を送り、両者はコネクション状態に入る。コネクション状態の protocol-entity はユーザよりデータ転送要求 (DataReq) を受けると通信路にデータ転送 (DT) を送る。相手の protocol-entity は DT を受け取ると acknowledge (AK) を返し、ユーザにデータ (DataInd) を送る。また、ユーザは protocol-entity に DiscReq を送ることでディスコネクション要求を出することができる。DiscReq を受けた protocol-entity はディスコネクション要求 (DR) を送る。これを受けた相手の protocol-entity はユーザに DiscInd を送りディスコネクション確認 (DC) を送り返す。これによりディスコネクトが成立する。

一般に通信プロトコルの記述は、割り込みやエラー等の例外処理の記述により複雑になる。このため、通常時 (エラーがない場合) の動作の把握が困難になる。RLOTOS では、図 4 に示したように通常時の処理とディスコネクトによる割り込み処理や通信路における情報の喪失等のエラー処理を分離して記述することが可能である。オブジェクトレベルのプロセスとしてコネクション要求を出して通信するプロトコルを記述し、メタレベルのリフレクティブプロセスとして割り込み処理とエラー処理を記述することができる。以下に RLOTOS による abracadabra-protocol における protocol-entity の記述を示す。

```
specification abracadabra(User,Line,tout):noexit
type ServicePrimitives is Data
sorts Prim
opns ConnReq,ConnInd,ConnResp,ConnConf :> Prim
DataReq,DataInd : Data-> Prim
DiscReq,DiscInd :> Prim
endtype
type ProtocolDataUnits is
sorts PDU
opns CR,CC,DC,DR :> PDU
DT : Prim,SeqNum-> PDU
AK : SeqNum-> PDU
endtype
behaviour
protocol[User,Line]
where
process protocol[User,Line]:noexit :=
ConnPhase[User,Line]
where
process ConnPhase[User,Line]:noexit :=
UserConnection[User,Line] [] RemoteConnection[User,Line]
endproc
process UserConnection[User,Line]:noexit :=
(*Connection Request Phase*)
User!ConnReq;Line!CR;
Line!r:PDU;User!ConnConf;DataPhase[User,Line]
endproc
process RemoteConnection[User,Line]:noexit :=
.....
endproc
process DataPhase[User,Line]:noexit := (*Data Phase*)
.....
endproc
endproc
process meta_level[currenttg,nextg,controlg]:noexit := (*meta_level*)
Meta_abra[currenttg,nextg,controlg]
where
process Meta_abra[currenttg,nextg,controlg]:noexit :=
currenttg!x:Bexp;nextg!y:ActPairList;
Meta_abra_i[currenttg,nextg,controlg](x,y)
where
process Meta_abra_i[currenttg,nextg,controlg]
(x:Bexp,y:ActPairList):noexit :=
```

```
choice a:Act,e:Bexp[]
[isIn(pair(a,e),y)]->
([a='User!ConnReq']->controlg!pair(a,e);
Meta_ConnReq[currenttg,nextg,controlg](Max)
[] [a='Line!r:PDU']->
([r=CR]->
controlg!pair(a,e);Meta_ConnInd[currenttg,nextg,controlg]
[] [r=DR]->
controlg!pair(a,x);currenttg!x:Bexp;nextg!y:ActPairList;
currenttg!x:Bexp;nextg!y:ActPairList;
controlg!pair('Line!DC',InitBExp);Meta_abra[currenttg,nextg,controlg]
[] [r=CC or r=DT or r=AK]->
controlg!pair(a,x);Meta_DiscPh[currenttg,nextg,controlg](Max)))
[] [NotIn(pair(a,e),y)]->Meta_abra_i[currenttg,nextg,controlg](x,y)
endproc
process Meta_ConnReq[currenttg,nextg,controlg](n:Nat):noexit :=
currenttg!x:Bexp;nextg!y:ActPairList;
Meta_ConnReq_i[currenttg,nextg,controlg](x,y,n)
where
process Meta_ConnReq_i[currenttg,nextg,controlg]
(x:Bexp,y:ActPairList,n:Nat):noexit :=
choice a:Act,e:Bexp[]
[isIn(pair(a,e),y)]->
([n=0]->
([a='Line!r:PDU']->
([r=CC or r=CR]->
controlg!pair(a,e);Meta_ConnReq[currenttg,nextg,controlg](n)
[] [r=DR]->
controlg!pair(a,x);Meta_DiscAck[currenttg,nextg,controlg]
[] [r=DT or r=AK or r=DC]->
controlg!pair(a,x);Meta_DiscIn[currenttg,nextg,controlg]
[] controlg!pair('tout','Line!CR;x');
Meta_ConnReq[currenttg,nextg,controlg](n-1))
[] [n=0]->Meta_DiscIn_i[currenttg,nextg,controlg](x)
[] [a='User!ConnConf']->
controlg!pair(a,x);Meta_DataPh[currenttg,nextg,controlg]
[] [a='Line!CR']->
controlg!pair(a,e);Meta_ConnReq[currenttg,nextg,controlg](n)
[] controlg!pair('User!DiscReq',x);
Meta_DiscPh[currenttg,nextg,controlg](Max))
[] [NotIn(pair(a,e),y)]->Meta_ConnReq_i[currenttg,nextg,controlg](x,y,n)
endproc
endproc
process Meta_ConnInd[currenttg,nextg,controlg]:noexit :=
.....
endproc
process Meta_DataPh[currenttg,nextg,controlg]:noexit :=
.....
endproc
process Meta_DiscIn[currenttg,nextg,controlg]:noexit :=
controlg!pair('Use!DiscInd',x);
Meta_DiscPh[currenttg,nextg,controlg](Max)
endproc
endproc
process Meta_DiscPh[currenttg,nextg,controlg](n:Nat):noexit :=
currenttg!x:Bexp;nextg!y:ActPairList;
([n=0]->
(controlg!pair('Line!DR',x);
currenttg!x:Bexp;nextg!y:ActPairList;
([a='Line!r:PDU']->
([r=DC or r=DR]->
controlg!pair(a,InitBExp);Meta_abra[currenttg,nextg,controlg]
[] [r<DC and r<DR]->
controlg!pair(a,x);Meta_DiscPh[currenttg,nextg,controlg](n))
[] controlg!pair('tout',x);
Meta_DiscPh[currenttg,nextg,controlg](n-1)))
[] [n=0]->controlg!pair('Line!DR',InitBExp);
Meta_abra[currenttg,nextg,controlg])
endproc
process Meta_DiscAck[currenttg,nextg,controlg]:noexit :=
currenttg!x:Bexp;nextg!y:ActPairList;controlg!pair('User!DiscInd',x);
currenttg!x:Bexp;nextg!y:ActPairList;controlg!pair('Line!DC',InitBExp);
Meta_abra[currenttg,nextg,controlg]
endproc
endspec
```

メタレベルのプロセスではディスコネクト要求があった場合 (User!DiscReq) や、通信路 (Line) より通常と異なる情報が届いた場合は、ディスコネクトを行なうプロセスを呼ぶ。Meta_DiscIn, Meta_DiscPh, Meta_DiscAk プロセスはディスコネクト処理を行なう。これらのプロセスで扱われている ACT-ONE の項 InitBExp はオブジェクトレベルのプロセス protocol のデータ表現である。また、通信路における情報喪失が起こった場合はタイムアウトイベント tout が起こり、それにより再コネクション要求を出す。abracadabra protocol では再コネクション要求の回数は制限されており、ACT-ONE の項 Max は再コネクション要求の最高回数を表している。コネクション要求の回数は Meta_ConnReq, Meta_sub.ConnReq プロセスの変数 n を用いて管理される。同様にディスコネクト要求の回数も Meta_DiscPh プロセスの変数 n で管理される。

3.4 LOTOS-T

仕様記述経験をもとに、不備な箇所を抽出し、LOTOS を拡張しようという動きがある [11]。その一つが実時間システムが記述できるように、時間概念を導入した言語 LOTOS-T である。LOTOS-T では、各々のイベントに対し、起こる時刻を明示する手法で、時間の経過を表す。例えば、 $a\{t\}$; B は、 t 単位時間経過した後にはちょうどイベント a が起こり、その後 behavior expression B を行なうことを表している。時刻を表すパラメータが付加されていない場合、つまり a ; B は、 a の生起が可能ならばできるだけ早く a を起こし、その後 B を行なう。

以上のような時間概念を導入したことにより、実行のやり方も通常の LOTOS とは異なる部分がある。LOTOS-T では、起こり得るイベントは直ちに起こす (ASAP : As Soon As Possible) という原則で実行を行なう。この原則に従った実行を行なわせるために、ラベル付き遷移規則には、イベントの生起に関して、イベントが起こる時刻よりも前に起こり得る内部イベントがないという条件が付加されている。これにより、例えば、

$(i\{3\}; a\{0\}; \text{stop}) ||| (i\{5\}; b\{0\}; \text{stop})$

で、 $i\{5\}$ が $i\{3\}$ よりも先に起こることが無くなり、デッドロックに陥ることがなくなる。従って、LOTOS-T の記述を実行させるためには、次に生起が可能なイベント集合を求め、その中から上記の条件、それよりも先に起こさなければならない内部イベントがないようなイベントのみを選択する機構が必要である。この機構をメタレベルのプロセス (asap) として、RLOTOS を用いて記述すると以下のようになる。

```
specification LOTOS-T : noexit
(* 抽象データの定義 *)
behavior
(* LOTOS-T による記述 *)

process meta-level[currentg, nextg, controlg] : noexit :=
  currentg?z:Bexp;nextg?x:ActPairList;
  asap[currentg,nextg,controlg](z,x)
where
  process asap[controlg](z:Bexp,x:ActPairList):noexit:=
    choice y:Act,nextbexp:Bexp[]
    [isin(pair(y,nextbexp),Select(x))]->
      ((z='A1|[Glist]|A2')and(nextbexp='nextA1|[Glist]|nextA2')->
        [(A1 = nextA1)->
          controlg'pair(y,'Age(timepart(y),A1)|[Glist]|nextA2');
          meta-level[currentg,nextg,controlg]
        ]
        [A2 = nextA2]->
          controlg'pair(y,'nextA1|[Glist]|Age(timepart(y),A2)');
          meta-level[currentg,nextg,controlg]
        ]
        [not(A1=nextA1)and(not(A2=nextA2))]->
          (*otherwise-synchronizations*)
          controlg'pair(y,nextbexp);meta-level[currentg,nextg,controlg]
        ]
        [(z='A1|A2')and(nextbexp='nextA1|A2')and(not(A1=nextA1))]->
          controlg'pair(y,'nextA1|Age(timepart(y),A2)');
          meta-level[currentg,nextg,controlg]
        ]
        [not(z='A1|[Glist]|A2')and(not(z='A1|A2'))]->
          controlg'pair(y,nextbexp);
          meta-level[currentg,nextg,controlg]
        ]
        [not(isin(pair(y,nextbexp),Select(x)))]->
          asap[currentg,nextg,controlg](y,x)
        ]
      endproc
    endproc

type ASAP is ActPairList, Time
opns
  Select : ActPairList -> ActPairList
  Select1 : ActPairList, ActSet -> ActPairList
  NoTime.Value : Act -> Bool
  NoSoonerEvent : Act, ActSet -> Bool
  Age : time, Bexp -> Bexp

eqns
  forall X,NextPairList:ActPairList,PossibleEventSet:ActSet,
    eventpair:ActPair,event,event2:Act,
    a:ActId,x:VarId,A1,A2,B:Bexp,Glist:GateIdList
  ofsort ActPairList
    Select(X) = Select1(X,Eventpart(X));
    NoTime.Value(eventpart(eventpair))=>
      Select1(Insert(eventpair,NextPairList),PossibleEventSet)
      =Insert(eventpair,Select1(NextPairList,PossibleEventSet));
    not(NoTime.Value(eventpart(eventpair)))
      and NoSoonerEvent(eventpart(eventpair),PossibleEventSet)=>
      Select1(Insert(eventpair,NextPairList),PossibleEventSet)
      =Insert(eventpair,Select1(NextPairList,PossibleEventSet));
    not(NoTime.Value(eventpart(eventpair)))
      and not(NoSoonerEvent(eventpart(eventpair),PossibleEventSet))=>
      Select1(Insert(eventpair,NextPairList),PossibleEventSet)
```

```
=Insert(eventpair,Select1(NextPairList,PossibleEventSet));
Select1({},PossibleEventSet) = {};
ofsort Bool
  NoSoonerEvent(event1,{}) = true;
  (timepart(event1) < timepart(event2))or(not InnerEvent(event2))=>
  NoSoonerEvent(event1,Insert(event2,PossibleEventSet1))
  =NoSoonerEvent(event1,PossibleEventSet1);
  (timepart(event1) < timepart(event2))and(InnerEvent(event2))=>
  NoSoonerEvent(event1,Insert(event2,PossibleEventSet1))= false;
  NoTime.Value = ...
ofsort Bexp
  Age(t,'exit?x:time...') = 'exit?x:time...';
  Age(t,'exit?tl...') = 'exit?(tl-t)...';
  Age(t,'a?x:time...;B') = 'a?x:time...;B';
  Age(t,'attl...;B') = 'a!(tl-t)...;B';
  Age(t,'A1|[Glist]|A2') = 'Age(A1|[Glist]|A2)';
  ...
endtype
endasap
```

構文上の拡張を避けるために、イベントに付加された時間パラメータをそのイベントの第1引数で表すことにする。つまり、 $a\{t\} \dots$ は、 $a!t \dots$ と表現することにする。時間パラメータなしのイベント $a \dots$ は、時間ソートの変数を第1引数につけ、 $a?x:time \dots$ とする。

メタレベルのプロセスは、nextg より先に起こる Event の集合を受けとり、その中から自分より先に起こる内部 Event のないような Event を求め、それを次に起こる Event として、controlg へ出力する。Select(EventPairList) は、イベントとそのイベントが起こったときの次状態を表す behavior expression の対の集合 EventPairList から、asap の条件を満足するものを求める。関数 Select は、抽象データ型 ASAP の中で定義される。Select の定義に使用されている関数 Select1(EventPairList,EventSet) は、イベントと次状態を表す behavior expression のリスト EventPairList から、述語 NoSoonerEvent で表される条件を満足する対を抜き出す関数である。NoSoonerEvent(event, EventSet) は、イベント集合 EventSet 中に event よりも早い時刻で起こるイベントがないことを判定する。述語 NoTime.Value は時間パラメータが変数となっているときに true を返す述語で、変数となっている場合は生起時刻の指定がないため次に起こるイベントの候補としても構わない。関数 Age は、behavior expression が Parallel Operator($[[\dots]]$)や Disabling Operator($[\dots]$)で構成されているときに、実行が選択されなかった部分 behavior expression の先頭のイベントの時刻を、生起されたイベントの時刻だけ進めるためのものである。例えば、 $a!2; B1 ||| b!3; B2$ で、 $a!2$ が先に起こったとすると、このイベントの生起により、時刻が2単位時間進められる。そのため、 a の次に b が起こるのであれば、それは3単位時間後ではなく、 a の生起から $3-2=1$ 単位時間後でなければならない。関数 Age を用いることにより、 a の生起後の behavior expression $B1 ||| b!3; B2$ は、

```
Age(timepart('a!2'), 'B1 ||| b!3; B2')
= 'Age(2, B1) ||| Age(2, 'b!3; B2')
= 'Age(2, B1) ||| b!1; B2'
```

に置き換えられ、controlg へ出力される。実際の RLOTOS 記述では、behavior expression も抽象データ型の項表現で記述しなければならないが、ここでは見やすさのために behavior expression の構造を直接記述した。記述中の $A1, A2, \text{nextA1}, \text{nextA2}$ は Bexp(behavior expression を表すソート)、Glist は GateList(ゲートのリストを表すソート)の変数である。

3.5 OS — MINIX

MINIX オペレーティングシステムは、Tanenbaum によって教育用に設計された UNIX like なオペレーティングシステムで、その内部構造だけでなく、ソースコードも公開されている [12]。MINIX はユーザプロセスを除くと3つの Layer から構成されており、最下層がプロセス管理、2層が I/O 装置のための Device Driver(DiskTask, TtyTask など)、3層が Memory Management プロセス (MM) と File System プ

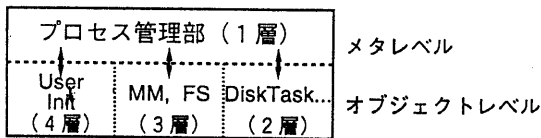


図 5: minix

プロセス (FS) の 2 つから構成される。ユーザプロセスがファイルアクセスなどの System Call を行なうと、第 3 層のプロセスにメッセージが送られ、さらに第 2 層の適当なプロセスが選択され、メッセージが送信される。このように、2 層、3 層では並列に動作する複数のプロセスがメッセージを交換し合うことによって処理が進められていく。1 層のプロセス管理部によって、2 層以上のプロセスが短時間に 1 つずつ切替えられながら実行され、疑似並列実行のメカニズムが実現される。MINIX 側のプロセス間のメッセージ送受信機構は、送信側受信側が両方準備ができなければ相手側を待つというランデブー方式である。ただし、メッセージ待ち状態にあるプロセスは Blocked と呼ばれる状態に入って待つ。実際に実行されているプロセスの内部状態は Running、実行可能ではあるが他の実行中のプロセスによって CPU を占有されているため、実行が待たされているプロセスの内部状態は Ready である。MINIX は優先順位付きのラウンドロビン方式で、2, 3, 4 層のプロセスの順に優先される。これらのプロセスで Ready 状態にあるものは優先順位別にキューにつながれ、1 層のプロセス管理部がキューの中からプロセスを優先順位に従って取りだし、実行させていく。あらかじめ決められた実行時間が来てしまったときはキューに戻す。また、メッセージ送受信待ちになった場合は、内部状態を Blocked にして、Blocked キューにつなぐ。

以上のような MINIX を RLOTOS で記述することを考える。2 層以上の MINIX のプロセスはそのまま LOTOS のプロセスに対応づける。プロセスの実行の切り替えなどのプロセスの制御を第 1 層のプロセス管理部が行うため、この部分の記述を RLOTOS のメタレベルで行う。LOTOS では、並列実行はインターリーブ方式で行われるため、メタレベルのプロセスでインターリーブの仕方を制御する。つまり、並列に動作しているプロセスのうち、Blocked 状態、Ready 状態にあるプロセスのイベント生起は、LOTOS の behavior expression として捉えたとき、可能であっても、メタレベルでは生起可能なイベントからはずす。これは、イベントがどのプロセスのものかを求める関数 processid の値が現在 running 状態にあるプロセス識別子 running_pid と等しいかどうかをチェックすることによって行なわれる。

プロセス間通信を行うためのメカニズムもメタレベルに記述する。メッセージを送ろうとするプロセスは、イベント send を起こす。このイベント生起がメッセージ送信のシステムコールを呼び出したことに対応する。メタレベルのプロセスは、メッセージの受信プロセスがメッセージの到着を待っている状態 (Blocked 状態にも入っている) かどうかをチェックする。到着待ちにあるプロセスは、receive_q キューにつながれている。もし

入っていれば、送信プロセスを引続きそのまま実行し (send イベントを起こし)、受信側プロセスの受信システムコールの実行を表すイベント receive を起こしてから、receive_q, blocked_q から受信プロセスを取り除き、Ready 状態のプロセスを保持しているキュー ready_q につなぐ (put_process(receiver, ready_q))。ready_q は、2, 3, 4 層のプロセスに対応して task_q, server_q, user_q の 3 つから構成され、この順に優先順位が低くなっている。もし、受信側プロセスが receive_q に入っていない (到着待ちでない) 場合は、送信プロセスを blocked_q、送信待ちを表す send_q につなぐ (enqueue(running, blocked_q) 及び enqueue(pair(sender, receiver), send_q))。Blocked 状態とし、Ready 状態にある他のプロセスのうち最優先のもの (first_priority_process(ready_q)) をとりだし (pick_process(ready_q))、実行を始める。プロセス切替えの時間が来たら (timeout イベントが起こると)、実行中のプロセス (running_pid) を ready_q につなぎ、同様に最優先プロセスを取りだし、それを新しい Running 状態のプロセス running_pid とする。

specification MINIX : noexit

(* 抽象データ型の定義 *)

behavior

DiskTask ||| TtyTask ||| ClockTask ||| ... ||| MM ||| FS ||| Init ||| UserProcess(0) ||| UserProcess(s(0)) ||| ...

where

(* 各プロセスの定義 *)

```
process meta_level[currenttg,nexttg,controlg]:noexit :=
  process_management[currenttg,nexttg,controlg]
  (z,x,(),(),InitRunningPid)
where
  process process_management[currenttg,nexttg,controlg]
    (ready_q:Ready_q,send_q,receiver_q:QueueOfProcessId,
     running_pid:ProcessId):noexit :=
      currenttg?z:Bexp;nexttg?x:ActPairList;
      process_management1[currenttg,nexttg,controlg]
        (z,x,ready_q,blocked_q,send_q,receiver_q,running_pid)
  where
    process process_management1[currenttg,nexttg,controlg]
      (z:Bexp,x:ActPairList,
       ready_q:Ready_q,send_q,receiver_q:QueueOfProcessId,
       running_pid:ProcessId):noexit :=
        hide timeout in
          (timeout;
           process_management[currenttg,nexttg,controlg]
             (pick_process(put_process(running_pid,ready_q)),
              blocked_q,send_q,receiver_q,
              first_priority_process(put_process(running_pid,ready_q)))
           (* --- timeout による Running プロセスの切替え *)
          )
        [choice y:ActSet,nextbexp:Bexp]
        [isIn(pair(y,nextbexp),x)]->
          [processid(y)=running_pid]->
            (* --- 次のイベントが Running プロセスのものであるとき *)
            ([y='send!sender!receiver!message']->
              (* send システムコールを次に実行しようとしているとき *)
              ([isIn(pair(sender,receiver),receiver_q)]->
                controlg!pair(y,'receiver!receiver!sender!message!nextbexp');
                process_management[currenttg,nexttg,controlg]
                  (put_process(receiver,ready_q),
                   dequeue(receiver,blocked_q),sender_q,
                   dequeue(pair(sender,receiver),receiver_q),
                   running_pid)
                )
              [not(isIn(pair(sender,receiver),receiver_q)]->
                process_management1[currenttg,nexttg,controlg]
                  (z,x,pick_process(ready_q),
                   enqueue(running_pid,blocked_q),
                   enqueue(pair(sender,receiver),send_q),
                   ready_q,first_priority_process(ready_q)))
              )
            )
          [y='receive!sender!receiver!pidfx:message']->
            (* receive システムコールを次に実行しようとしているとき *)
            [not(systemcall(y)='send' and not(systemcall(y)='receive'))]
            (* send, receive 以外を実行しようとしているとき *)
            -> controlg!pair(y,nextbexp);
            process_management1[currenttg,nexttg,controlg]
              (ready_q,send_q,receiver_q,running_pid)
          )
        [not(processid(y)=running_pid)]->
          process_management1[currenttg,nexttg,controlg]
            (z,x,ready_q,blocked_q,send_q,receiver_q,running_pid)
          (* --- 次のイベントが Running プロセス以外のは起こさない *)
        )
      )
    )
  endproc
endproc
endproc

Type Ready_queue is QueueOfProcessId
sorts
  ready_q
ops
  () : -> ready_q
  task_q, server_q, user_q : ready_q -> QueueOfProcessId
  pick_process : ready_q -> ready_q
  (* ready_q 中の最優先プロセスを取り出す *)
```

```

put_process : ProcessId, ready_q -> ready_q
(* ready_q にプロセス ProcessId を戻す *)
first_priority_process : ready_q -> ProcessId
(* ready_q 中の最優先プロセスを求める *)
eqns
...
endtype
endspec

```

4 考察

RLOTOS をはじめとしたリフレクションの機構を持った言語ではオブジェクトレベルとメタレベルの概念を用いることができる。これらの概念を用いることで、例えば、通信プロトコルでは割り込みやエラー処理を、fault-tolerant システムでは fault-tolerant 部分をそれぞれ分離することができる。これにより通常動作を分離して記述できる。MINIX をはじめとした OS ではプロセス管理部分を分離することで、また lift システムでは lift 個々の仕様とそれらを制御する部分とに分離することで管理、制御機能を明示的に仕様化できる。仕様記述では厳密性があることとともに了解性があることは非常に重要であり、RLOTOS で仕様記述を行なう際はこれらのレベルの概念を用いて記述の対象事象を、通常動作と例外動作、本質的な動作と補助的な動作、個々の動作とそれらを管理制御部分、とに分離することで了解性のある記述が行なえる。それゆえ、記述対象システムのレベルの分離が仕様化を進める上で重要である。本稿で扱ったシステムはいずれもオブジェクトレベル、メタレベルの分離が直観的に行ない易かったが、他の種類のシステム、例えばユーザインターフェース等にも適用し、分離基準を調べる必要がある。これまでの例からでは、対象分野ごとに分離に特徴があるため、パターン化することが可能であると思われる。

また、RLOTOS メタレベルを用いてオブジェクトレベルの動作を変更することが可能である。例えば、メタレベルでオブジェクトレベル上で起こるイベントの出現を規制する記述を行なうことで asap 機構を実現することが可能となり LOTOS-T を容易に実現することができる。RLOTOS ではメタレベルを記述することで拡張 LOTOS を容易に構成することができる。

3 節で扱った事象の他に RLOTOS はオブジェクトレベルとメタレベルの概念を用いることでソフトウェアプロセスプログラミングやデバッグなどの記述に適していると考えられる。

5 あとがき

本稿ではまず RLOTOS のリフレクション機構について述べた。そして RLOTOS を用いていくつかの事象について仕様の記述を行なった。これらの仕様はオブジェクトレベルとメタレベルの概念を用いて記述することで、了解性のある仕様記述を行なうことができた。また、拡張 LOTOS である T-LOTOS を RLOTOS を用いて構成した。そして、これらを評価した。

今後は、リフレクション言語を用いた際の仕様化技法の開発、支援ツールの作成が課題である。

謝辞

RLOTOS について助言を頂いた富士通国際研の鵜飼孝典氏、MINIX の仕様の記述するにあたり助言を頂いた大学の井口和久氏に感謝の意を示します。

参考文献

- [1] ISO 8807. *Information processing systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour*, 1989.
- [2] 大蒔他. 図書館の問題とエレベータの問題の LOTOS による仕様記述. 情報処理学会ソフトウェア工学研究会, Vol. 64, , 1989.
- [3] 大蒔, 二木. 形式的仕様記述言語 LOTOS の試用経験. 情報処理学会誌, Vol. 31, No. 10, pp. 1400-1413, 1990.
- [4] B.C Smith. Reflection and semantics in Lisp. In *Proc. of 12th ACM Sympo. on POPL*, pp. 23-35, 1984.
- [5] Takuo Watanabe and Akinori Yonezawa. Reflective Computation in Object-Oriented Concurrent System and Its Applications. In *Proc. of Fifth IWSSD*, pp. 56-58, 1989.
- [6] Jiro Tanaka. An Experimental Reflective Programming System writtin in GHC. *Journal of Information Processing*, Vol. 14, No. 1, 1991.
- [7] 鵜飼, 広井, 佐伯. 仕様記述言語 LOTOS におけるリフレクション: RLOTOS. 情報処理学会プログラミング言語, 基礎, 実践-, Vol. 92, No. 6, pp. 1-10, 1992.
- [8] *Problem Set for the 4th International Workshop on Software Specification and Design: Proc. of 4th International Workshop on Software Specification and Design*, 1987.
- [9] Nancy G. Leveson and Janice L. Stolzy. Safety analysis using petri nets. In *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-13, No. 3*, pp. 386-397, march 1987.
- [10] ISO/IEC JTC 1 N642: *Information Processing Systems—Open System Interconnection—Proposed Draft Technical Report on Guidelines for the Application of Estelle, LOTOS, and SDL*, janually 1990.
- [11] ISO/IEC JTC1/SC21/WG1 N1180. *Contribution on Enhancements to LOTOS*, 1992.
- [12] A.S. Tanenbaum. *Operating Systems — Design and Implementation*. Prentice Hall, 1987.