ソフトウェア工学 96－9
（1994．1．21）

リフレクティブ言語を用いた形式的仕様記述の支援

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 にリフレクションの概念を導入した言語 RLOTOS (Reflective LOTOS) 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。本稿では RLOTOS を用いて，メタレベル記述を記述スタイルに基づいて支援する方法を述べる。最初に通信プロトコルを例題として LOTOS の４つの記述スタイル (モノリシック，制約指向，状態指向，リソース指向) を用いて RLOTOS のメタレベル記述を行う．そしてこれに基づいてこれらの記述スタイルを RLOTOS のメタレベル記述に適用する際の手順を示す．

Supporting the Development of Formal Specification with Reflective Language

Takeshi Hiroi　　Motoshi Saeki

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

This paper discusses a specification technique by using a formal language with reflection(RLOTOS), which is an extension of LOTOS with reflective architecuture. RLOTOS has two level architecture — *object level* and *meta level*. In this paper, we propose a method to develop the meta level specification of RLOTOS based on the four specification styles(monolithic style, constraint oriented style, state oriented style, resource oriented style). Furthermore, we show the specificaion of alternating bit protocol as an example of our method.

1 はじめに

形式的仕様記述言語の一つに LOTOS(Language Of Temporal Ordering Specification)[1] がある．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目的で考案された言語でありプロセス代数と多ソート代数によって意味付けされている．また 1988 年に ISO で標準化されている．LOTOS は並列性や非決定性，同期などの豊富な記述力を持ち，様々なシステムへの適用も報告されている [2, 3].

また，リフレクションの概念を用いた形式的仕様記述に関する研究がなされており LOTOS にリフレクションの概念を導入した言語 RLOTOS(Reflective LOTOS) が提案されている [4]．RLOTOS は他のリフレクション言語と同様にオブジェクトレベルとメタレベルの概念を持っている．RLOTOS を用いた種々のシステムの記述も行なわれており，リフレクション言語による仕様記述では例外処理を扱うシステムや管理制御を行なうシステムをこれらのレベルの概念を用いることで分離して記述でき，了解性のある記述が行なえることが報告されている [5].

一方，リフレクション言語による記述は一般に柔軟な記述が可能なため，記述をするための指針を与えることは重要であるが，リフレクション言語における記述の支援に関する研究はほとんどなされていないのが現状である．

本稿ではリフレクション言語による仕様記述の支援方法について記述スタイルの点から支援する．記述スタイルとはある観点に基づいてシステムを仕様化する際に現れる記述の構文的な特徴を表したものである．現在，LOTOS の記述スタイルとして Vissers らによって提案されている記述スタイルがある [6]．外部から観測可能なイベントに着目するモノリシックスタイルと制約指向スタイル，内部処理や内部状態に着目する状態指向スタイルとリソーススタイルがある．本稿ではこれらの記述スタイルを RLOTOS 記述に適用しそれに対する考察を行なう．そして実際のアプリケーションを記述する際の記述スタイルの適用の方法について述べる．

2 RLOTOS

RLOTOS は LOTOS の動的式部分にリフレクション機構を導入した言語であり，メタレベルとオブジェクトレベルの概念を持っている．表 1 は LOTOS の基本的なオペレータを表している．RLOTOS の構造を図 1 に示す．メタレベル上にはオブジェクトレベルの LOTOS 記述を解釈実行する LOTOS のプロセスとして記述されたインタプリタが存在する．メタレベル上のインタプリタでは，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 の項表現に変換して項上の操作により解釈実行する．

RLOTOS では，リフレクティブプロセスを記述することでリフレクティブ計算を行なうことができる．リフレクティブプロセスではリフレクティブゲートという特別なゲートを用いて LOTOS の通信方式である同期通信によりメタレベル上のインタプリタにアクセスできる．

RLOTOS ではリフレクティブゲートとして図 1 に示した

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 [] B2	B1 か B2 を選択
B1 || B2	B1 と B2 が同期
B1 ||| B2	B1 と B2 が独立
B1 $\|[g_1,\ldots,g_n]\|$ B2	ゲートについて同期
stop	inaction

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

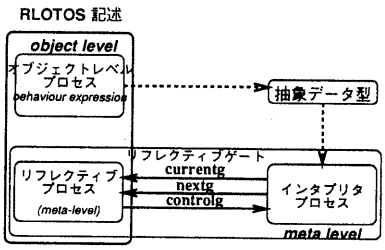

図 1: RLOTOS の構造

3 種類のゲートが提供されている．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．インタプリタから currentg, nextg を通してそれぞれ「現在の behavior expression の状態 (ソート Bexp)」と「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 (ソート ActPair) のリスト (ソート ActPairSet)」に関するデータを受けとれる．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 ActPairSet)」を渡しインタプリタを制御する．インタプリタが提供するこれらの情報をメタ情報と呼び，リフレクティブゲートを用いたこれらのメタ情報の取得変更に関する手続きのことをリフレクティブ手続きと呼ぶ．

また，RLOTOS 記述はオブジェクトレベルの動作式の記述と各レベルに対応したリフレクティブプロセスの記述からなり，オブジェクトレベルの動作式は **behaviour of objectlevel**，メタレベルのリフレクティブプロセスの記述は **behaviour of metalevel** で定義する．

3 オブジェクトレベルとメタレベルの分離について

RLOTOS ではオブジェクトベルとメタレベルの概念を用いた記述が可能である．RLOTOS を用いて種々のシステムの形式的な仕様記述が行なわれており，これらのレベルの概念を用いることによるシステムを分離して記述することの有効性が報告されている [5].

通信プロトコル等の例外処理を扱うシステムでは，通常の LOTOS を用いた場合，例外処理の記述が通常処理の記述の中に埋め込まれ，システム全体の記述が複雑になる．そのため，通常処理の把握が困難となり了解性が落ちる．

RLOTOS で例外処理を扱うシステムを記述する場合は，例外的な処理をメタレベルに抽出して記述でき，通常処理をオブジェクトレベルに記述できる．このような記述の分離により，通常処理の把握が容易になり了解性のある記述が可能となる．

3.1 alternating bit protocol

通信プロトコルの一つである alternating bit protocol [7] を例にレベルの概念を用いた分離について示す．図 2 は

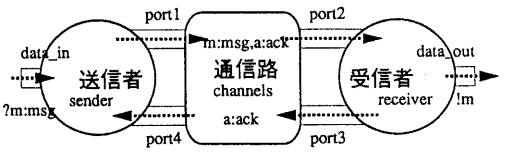

図 2: alternating bit protocol の構成図

alternating bit protocol のシステム構成を示している．alternating bit protocol は送信者と受信者，通信路から構成されている．図 3 はメッセージシーケンス図を表しており alternating bit protocol のエラーの起こらない通常時のメッセージの流れを表している．送信者 (sender) は data_in より受けたメッセージをエラー判定ビットを付加して port1 より通信路 (channels) へ送る．受信者 (receiver) は port2 より受けたメッセージが正常な場合は data_out より出力し，正常受信を示す acknowledgement を port3 より通信路へ送る．この時，通信路のエラーによりデータが壊れていた場合は受信者はメッセージを出力せず，異常受信を示す acknowledgement を port3 より送る．そして，送信者は port4 より通信路から acknowledgement を受けとり正常受信を確認できたら次のメッセージを送る．異常受信を確認した場合はもう一度同様の方法でメッセージを再送する．RLOTOS で通信プロトコルを記述する場合は，エラーの起こらない通常の処理に関する部分をオブジェクトレベルに，エラー処理に関する部分をメタレベルに分離して記述することが可能である．以下にオブジェクトレベル記述となるエラーの起こらない場合の alternating bit protocol の記述を示す．

```
behaviour of objectlevel
  sender[data_in,port1,port4](a0)
  |[port1,port4]|
  channels[port1,port2,port3,port4]
  |[port2, port3]|
  receiver[port2,port3,data_out](a1)
where
type  ACK is
sorts ack  (* エラー判定ビットを表すソート *)
opns  a0,a1 : -> ack
      add : ack -> ack
eqns  ofsort ack
      add(a0) = a1;
      add(a1) = a0;
endtype
type  MESSAGE is ACK
sorts msg  (* メッセージを表すソート *)
      ...
endtype
process sender[inp,port1,port4](a:ack):noexit :=
  data_inp?m:msg; port1!m!a; port4?a:ack;
  sender[data_in,port1,port4](add(a))
endproc
process channels[port1,port2,port3,port4]:noexit :=
  channel1[port1,port2] |||  channel2[port3,port4]
where
  process channel1[port1,port2]:noexit :=
    port1?m:msg?a:ack; port2!m!a; channel1[port1,port2]
  endproc
  process channel2[port3,port4]:noexit :=
    port3?a:ack; port4!a; channel2[port3,port4]
endproc
process receiver[port2,port3,data_out](a:ack):noexit :=
  port2?m:msg?a:ack; data_out!m; port3!add(a);
  receiver[port2,port3,data_out](add(a))
endproc
```

図 3 のように，送信者は入力されたメッセージとエラー判定ビット a0 を一緒に通信路に送る．受信者は通信路より受けとったメッセージを出力し acknowledgement(a0) を送信者に送り返す．そして次のメッセージ送信は a1 について同様に繰り返す．

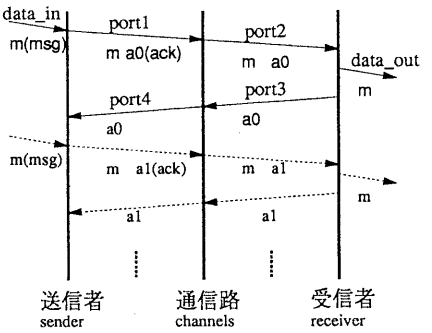

図 3: alternating bit protocol のメッセージシーケンス図

一方メタレベルには，通信路でエラーを起こしてメッセージを再送するといった例外処理の記述を行なう．図 4は通信路でエラーが起こりデータが壊れた場合のデータの再送を示したメッセージシーケンス図である．alternating bit protocol のエラー処理は (A)〜(D) の 4 通りある．例えば，図中の (A) では入力のあった送信者がメッセージとエラー判定ビットからなるデータを port1 に送り port2 でエラー（データが壊れる）が発生した場合の処理を示している．port2 でデータが壊れると受信者は図 4(A) のように port3 に異常データあることを知らせる acknowledgement(a1) を送り port4 より送信者に渡され，送信者はデータの再送をを行なう（太線部分）．また，(C) では port2 でエラーが起こり，そのエラー処理中に port4 でエラーが起こる場合を示している．この時は，acknowledgement(a1) が壊れる（点線）のみでメッセージの再送は行なわれない．図 4の data_crash, ack_crash は壊れたデータを表している．

RLOTOS でのメタレベル記述では図 4 に示したエラー処理の記述を行なう．メタレベル記述にはインタプリタより得たメタ情報に基づいてインタプリタの解釈を変更するように記述するので，これらのエラー処理に関する記述はメタレベル記述は条件分岐的な記述となる．

次節では，この alternating bit protocol について各記述スタイルを用いてメタレベル記述を行なう．

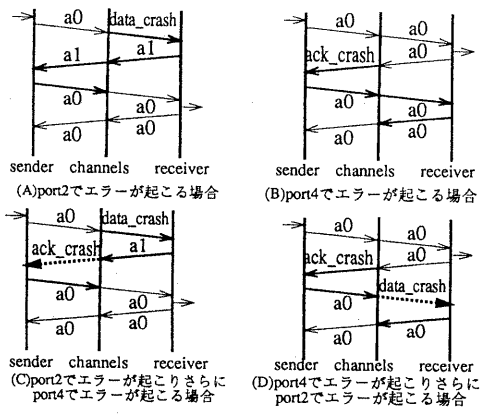

図 4: エラー発生時のシーケンス図

4 RLOTOS への適用

4.1 モノリシックスタイル

モノリシックスタイルでは外界から観測可能なイベントに着目し，内部状態等のシステムの内部情報を用いずに記述を行なう．このスタイルを用いた記述の構文的特徴はイベントが action prefix と条件分岐を用いて構成され，繰り返しは tail recursion の形で行なわれる．以下にこのスタイルを用いた alternating bit protocol の記述を示す．

```
behaviour of metalevel
  meta_input[currentg,nextg,controlg]
where
process meta_input[currentg,nextg,controlg]:noexit:=
    ...      (* for data_input *)
  meta_channelA[currentg,nextg,controlg]
endproc
process meta_channelA[currentg,nextg,controlg]:noexit:=
    ...      (* for port1 *)
  currentg?currentbexp:Bexp; nextg?y:ActPairSet;
  let next_pair:ActPair=choice(y) in
  let m:msg=getmsg(next_pair),a:ack=getack(next_pair) in
  controlg!next_pair;
   ... (* for port2 *)
  currentg?currentbexp:Bexp; nextg?y:ActPairSet;
  let next_pair:ActPair=choice(y) in
  (controlg!next_pair;
   meta_output[currentg,nextg,controlg](m,a)
   []
   controlg!<"port2!data_crash",  (*図4の(A)の場合*)
   "port3!add(a);port4!add(a);port1!dat(m,a);currentbexp">;
   meta_channelB_error[currentg,nextg,controlg](m,a))
endproc
process meta_output[currentg,nextg,controlg]
   ... (* for data_out *)              (m:msg,a:ack):noexit:=
process meta_channelB[currentg,nextg,controlg]
   ... (* for port3 *)                 (m:msg,a:ack):noexit:=
   ... (* for port4 *)
process meta_channelA_error[currentg,nextg,controlg]
   ... (* for port1 *)                 (m:msg,a:ack):noexit:=
   ... (* for port2 *)
process meta_channelB_error[currentg,nextg,controlg]
   ... (* for port3 *)                 (m:msg,a:ack):noexit:=
   ... (* for port4 *)
  currentg?currentbexp:Bexp; nextg?y:ActPairSet;
  let next_pair:ActPair=choice(y) in
  (controlg!next_pair;
   meta_channelA[currentg,nextg,controlg](m,a)
   []                                  (*図4の(A)の場合*)
   controlg!<"port4!ack_crash",bexp_part(next_pair)>;
   meta_channelA[currentg,nextg,controlg](m,a))
endproc
```

meta_input,meta_output,meta_channelA,meta_channelB, meta_channelA_error,meta_channelB_error の各プロセスではそれぞれ，data_in イベント，data_out イベント，port1,port2 イベント，port3,port4 イベントに対して行なわれるリフレクティブ計算を示している．port1 イベントに対しては port1 上で扱われるメッセージとエラー判定ビットを得ている．そして，port2, port4 イベントに対してはそれぞれ図 4 の (A),(B) に対応したエラー処理を行なう．また，meta_channelB_error プロセスでは図 4 の (C) に対するエラー処理を行なう．choice 関数は「次起こるイベントとその次状態の対の集合」より任意の一つの要素を取り出すための RLOTOS の ACT-ONE 上の組み込み関数である．また getmsg, getack 関数はそれぞれ，next_pair 上のイベント扱われているメッセージとエラー判定ビットを取得するための関数である．また，簡単のためにオブジェクトレベルのイベントや動作式はそれぞれ，"event"，" 動作式" で表し，イベントと動作式の対は<イベント, 動作式 > で表す．

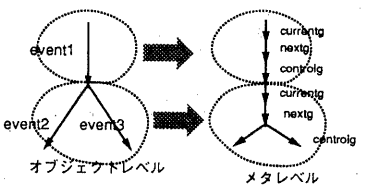

図 5: モノリシックスタイルによるリフレクティブ手続きの記述時の対応関係

このスタイルを用いたリフレクティブプロセスは，外界から観測可能なイベントとしてリフレクティブゲートを生起させそれぞれのゲートは action prefix を用いて構成する．このため，オブジェクトレベル記述のイベントを 1 つ又は 1 つ以上毎にまとめてそれに対応するリフレクティブ手続きを 1 つのプロセスとして tail recursion の形で他のプロセスを呼び出すことができる．また，モノリシックスタイルを用いて記述した場合のオブジェクトレベル記述の動作とメタレベル上のリフレクティブ手続きの対応関係は図 5 のように行なえる．オブジェクトレベルでの選択的な実行はメタレベル上では controlg に関しての条件分岐として図 5 のように記述することができる．リフレクティブ手続きはオブジェクトレベル記述でのイベントの生起を意識して記述するため，オブジェクトレベル記述が複雑な構造を持っていたり条件分岐が多い時は条件分岐が複雑になり見にくいものとなる．

4.2 状態指向スタイル

状態指向スタイルは記述全体を一つの状態遷移機械とみなしてシステムの状態が遷移する形で記述する．図 6 はリフレクティブプロセスと状態遷移機械の関係を示している．リフレクティブプロセスは currentg, nextg を入力イベン

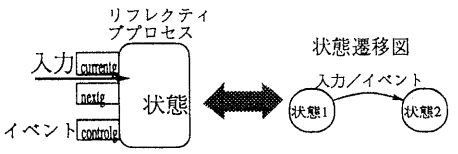

図 6: 状態遷移機械としてのリフレクティブプロセス

トとしてそこから得られる情報を状態遷移機械の入力に対応付けられ，controlg イベントを起こして次の状態に遷移する状態遷移機械に対応付けられる．このスタイルでリフレクティブ手続きを記述する場合は内部状態は状態変数として管理し，選択実行オペレータ [] と条件分岐を用いて各状態について動作を記述できる．以下に状態指向を用いた記述を示す．

```
behaviour of metalevel
  statetransiton[currentg,nextg,controlg](noerror,m0,a0)
where
process statetransition[currentg,nextg,controlg]
                    (s:state,m:msg,a:ack):noexit :=
  currentg?currentbexp:Bexp; nextg?y:ActPairSet;
  let next_pair:ActPair=choice(y) in
  [s==noerror]->
   ([event_name(next_pair)="port2"]->
     (controlg!next_pair;
      statetransition[currentg,nextg,controlg](noerror,m,a)
      []
      controlg!<"port2!data_crash",   (*図 4 の (A) の場合*)
        "port3!add(a);port4!add(a);port1!m!a;currentbexp">;
      statetransition[currentg,nextg,controlg](onerror,m,a))
    []
    [event_name(next_pair)="port4"]->
        ...
  []
  [s==onerror]->
   ([event_name(next_pair)=="port2"]->
     (controlg!next_pair;
      statetransition[currentg,nextg,controlg](noerror,m,a)
      []                          (*図 4 の (D) の場合*)
      controlg!<"port2!data_crask",bexp_part(next_pair)>;
      statetransition[currentg,nextg,controlg](noerror,m,a))
    []
    [event_name(next_pair)=="port4"]->
          ....
  []
  [event_name(next_pair)="port1"]->
        ....
  []
  [(event_name(next_pair)=="data_in") or
      (event_name(next_pair)=="port3") or
             (event_name(next_pair)="data_out")]->
        ....
endproc (* statetransition *)
```

この記述における状態遷移機械は error が起こったか起こっていないかを表す内部状態 noerror と onerror を持つ．noerror は図 4 の (A) と (B) の場合であり，onerror は (C) と (D) の場合である．最初，システムは noerror 状態にあり error が発生すると onerror 状態になる．また，onerror 状態から error が発生すると noerror 状態に戻る．

RLOTOS におけるリフレクティブプロセスの記述は図 6 に示したようにインタプリタから得られるメタ情報を得て（入力），それに応じてインタプリタを制御する（イベント）を起こすので状態遷移機械とみなして記述することが可能である．この時，状態に対する条件分岐を用いて各状態別に記述する．それぞれの状態は，オブジェクトレベルで生起するイベントに応じた処理を条件分岐の形で記述する．

つまり，リフレクティブプロセスは状態遷移機械的な振舞いをするので，リフレクティブ手続きを状態指向で記述することは適している．しかし状態指向は構造化できないため，大規模なシステムに対しては状態数や条件分岐が増大して複雑になってしまう．

4.3 制約指向

制約指向スタイルは LOTOS の動作式の記述を制約的に行なうものである．このスタイルでの各制約は外部から観測可能なイベントの生起順序のみを対象としており，それらの制約を和集合や積集合の形を用いて構成することでシステムの振舞いを規定するものである．リフレクティブプロセスの記述は図 6 に示したように currentg,nextg から得られるメタ情報に基づいて controlg を通してオブジェクトレベルの振舞いを制御する記述である．このため，「インタプリタより得られるメタ情報とオブジェクトレベルの振舞いの制御の関係」を記述することで制約的な記述が行なえる．

本節では sender プロセス側のゲート (data_in, port1, port4) に関して起こるイベントの生起順序に関する制約 (meta_send プロセス) と reciever プロセス側のゲート (data_out, port2,port3) に関して起こるイベントの生起順序に関する制約 (meta_receive プロセス) の和集合の形で構成している．

```
behaviour of metalevel
  meta_send[currentg,nextg,controlg](a0,m0,a0)
  |[currentg,nextg,controlg]|
  meta_receive[currentg,nextg,controlg](a0,m0,a0)
where
process meta_send[currentg,nextg,controlg]
                    (send_s:state,m:msg,a:ack):noexit :=
  currentg?currentbexp:Bexp; nextg?y:ActPairSet;
  let next_pair:ActPair=choice(y) in
  ([event_name(next_pair)="port4"]->
     (controlg!next_pair;
      meta_send [currentg,nextg,controlg](send_s,m,a)
      []
      ([eq(send_s,getack(next_pair))]->
          (*エラーがまだ起こっていない場合*)
        controlg!<"port4!ack_crash",  (*図 4 の (B) の場合*)
        "port1!m!a;port2!m!a;port3!add(a);currentbexp">;
        meta_send[currentg,nextg,controlg](send_s,m,a))
      []
      ([ne(send_s,getack(next_pair))]->
      (*既にエラーが起こっている場合--図 4 の (C) の場合*)
        controlg!<"port4!ack_crash",bexp_part(next_pair)>;
        meta_send[currentg,nextg,controlg](send_s,m,a))))
  []
  [event_name(next_pair)="data_in"]->
    (* メッセージとエラー判定ビットの取得 *)
    controlg!next_pair;
    meta_send[currentg,nextg,controlg]
          (add(send_s),getmsg(next_pair),add(a))
                  (* sender の状態 (send_s) をセット *)
  []
  [event_name(next_pair)="port1"]->
          ....
  [event_name(next_pair)<>"port1"
     and event_name(next_pair)<>"port4"
         and event_name(next_pair)<>"data_input"]->
    (* その他の場合 *)
    controlg?next_pair:ActPair;
    meta_send[currentg,nextg,controlg](send_s,m,a)
```

```
endproc (* meta_send *)
process meta_receive[currentg,nextg,controlg]
                    (receive_s:state,m:msg,a:ack):noexit :=
  currentg?currentbexp:Bexp; nextg?y:ActPairSet;
  let next_pair:ActPair=choice(y) in
  ([event_name(next_pair)="port2"]->
         ....
   [event_name(next_pair)="port3"]->
         ....
   [event_name(next_pair)="data_output"]->
         ....
   []
   [event_name(next_pair)<>"port1"
    and event_name(next_pair)<>"port4"
    and event_name(next_pair)<>"data_input"]->
         ....
endproc (* meta_receive *)
```

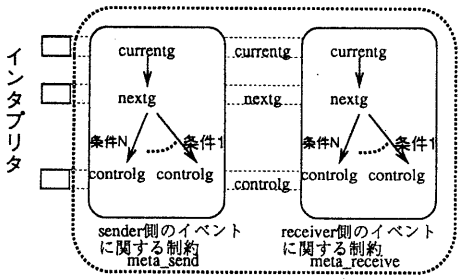

図 7: 制約指向による記述

図 7 は制約指向スタイルを用いた記述の構成を示している．制約プロセス meta_send と meta_receive は currentg,nextg,controlg で同期をとり，メタレベル記述におけるリフレクティブゲートの生起の順序と controlg での制御の方法を規定している（制約）．制約を表す各プロセスはそれぞれ，状態変数 send_s と receive_s でメッセージの入出力に関する状態を持っている．

4.4 リソース指向

RLOTOS ではメタレベル記述は，オブジェクトレベル記述を解釈する一つのインタプリタと通信するリフレクティブプロセスの記述である．メタレベル記述を複数のプロセスに構造化して記述する場合，それらのプロセス間のデータの値は整合性がとれている必要がある．RLOTOS では，このため複数のプロセスで構成されている場合は共有データを持つことは重要である．

リソース指向スタイルでは構成するリソースに着目して仕様を分割しそれぞれを LOTOS のプロセスに対応させて記述する．RLOTOS ではリフレクティブプロセスはリフレクティブゲートより得たメタ情報に基づいて計算を行ないインタプリタを制御する記述で構成する．リフレクティブプロセスをリソース指向スタイルを用いて記述する場合はインタプリタを制御する計算を行なうために用いるキューやデータベースなどをリソースに対応付けて記述することができる．

```
behaviour of metalevel
  hide msg_q,ack_q in
  (meta_send[currentg,nextg,controlg,msg_q,ack_q](a0)
   |[currentg,nextg,controlg]|
   meta_receive[currentg,nextg,controlg,msg_q,ack_q](a0))
   |[msg_q]| msg_queue[msg_q](m0)
   |[ack_q]| ack_queue[msg_q](a0)
where
process msg_queue[msg_q](m:msg):= (* データ管理用キュー *)
  msg_q!read!m; msg_queue[msg_q](m) (*データの読み出し*)
  []msg_q!write?m:msg;msg_queue[msg_q](m)
                                  (*データの書き込み*)
endproc (*msg_queue*)
process ack_queue[ack_q](a:ack):=
    ....
process meta_send[currentg,nextg,controlg]
                          (send_s:state):noexit :=
  currentg?currentbexp:Bexp; nextg?y:ActPairSet;
  let next_pair:ActPair=choice(y) in
  ([event_name(next_pair)="port4"]->
        ....
     []
     ([eq(send_s,getack(next_pair))]->
         (*エラーがまだ起こっていない場合*)
       msg_q!read?m:msg; ack_q!read?a:ack;
       controlg!<"port4!ack_crash",  (*図 4 の (B) の場合*)
         "port1!m!a;port2!m!a;port3!add(a);currentbexp">;
       meta_send[currentg,nextg,controlg](send_s))
       [] ....
     []
     [event_name(next_pair)="data_in"]->
       controlg!next_pair;
       msg_q!write!getmsg(next_pair);
       ack_q!write!getack(next_pair);
         ....
  endproc (* meta_send *)
  process meta_receive[currentg,nextg,controlg]
                          (receive_s:state):noexit :=
         ....
endproc
```

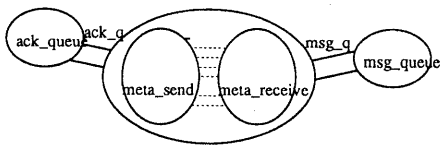

図 8: リソースへの対応

図 8 は上の記述のリフレクティブプロセスの構造を表している．この記述は 4.3 節で示した meta_send と meta_receive の 2 つのプロセスからなる記述に関して，変数データとして管理していた m:msg(メッセージ) と a:ack(エラー判定ビット) をそれぞれリソースプロセスとして管理している．キュー等のリソースを個々のプロセスとして各プロセスはゲートを通して通信する．メッセージを扱う msg_queue プロセスは msg_q ゲートで，例えば，"msg_q!write!x" で msg_queue に x を書き込むことができる．この時，通信に用いるゲートは内部化する．リソース間の通信は通常の LOTOS における記述と同様，LOTOS のプロセス間通信に基づいている．

5 4 つのスタイルの適用方法

4 節では各記述スタイルを RLOTOS 記述に適用した．本節ではこれらのケーススタディに基づいて RLOTOS でのリフレクティブプロセスの記述の際の 4 つのスタイルを適用するための手順を示す．

5.1 各スタイルの適用方法の概要

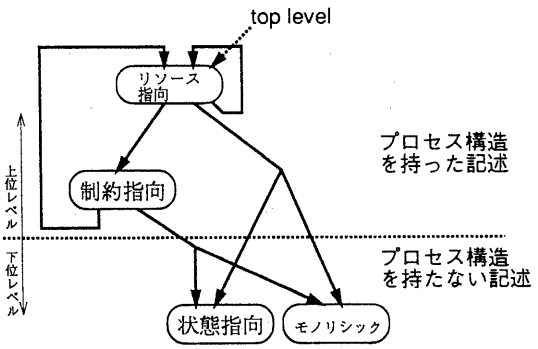

図 9: 4つのスタイルの階層関係

図 9 は RLOTOS のリフレクティブプロセスの記述に 4 つの記述スタイルを適用する際の階層関係を示している．4 つのスタイルに関して，複数のプロセスを用いた構造化の構文的な枠組みが提供されていないモノリシックスタイルと状態指向スタイルは下位レベルでの記述に利用される．構造化の枠組みが提供されているリソース指向スタイルと制約指向スタイルは上位レベルでの記述に利用される．

このためリフレクティブプロセスにこれらのスタイルを適用する手順は，リソース指向スタイルと制約指向スタイルに基づいた以下のステップからなる．

1. リソース指向スタイルを用いてリフレクティブ手続きから共有データなどのリソースを分割する
2. 制約指向スタイルを用いてリフレクティブ手続き部分について各制約に分割する

最上位のリフレクティブプロセスは，はじめに 1. を用いて，共有データを扱うリソースなどをリフレクティブプロセスから分割し，リフレクティブ手続きを行なう部分と分割する．このとき，さらに 1. を用いて分割可能なら行なう．そして，リフレクティブ手続きを行なう部分について 2. を用いて，種類別に各制約をプロセスに分割する．この分割された各々の制約を表すプロセスについては最上位の記述と同様に 1.,2. が繰り返される．もし，これ以上 1.,2. が適用できない場合は状態スタイルやモノリシックスタイルで記述する．

5.2 各スタイルの適用方法

5.2.1 リソース指向スタイル

RLOTOS では，リフレクティブプロセスでは，オブジェクトレベルを制御するためのリフレクティブ手続きの記述を行なう．4.4 節で述べたようにリフレクティブ手続きが複数のプロセスに分割されている場合は制御を行なうために他のプロセスの状態を把握する必要があり，共有データを持つこと重要である．共有データはリソースに対応付けることで実現できるので，リフレクティブプロセスの記述を行なう際は，はじめに共有データをリソースとして分割するとよい．

5.2.2 制約指向スタイル

リフレクティブ手続きの記述は「メタ情報を得てそれに基づいてオブジェクトレベルの動作を制御する」記述である．これは「メタ情報と制御の関係」という制約的な観点で捉えられるので，制約指向スタイルによる記述は適していると考えられる．しかし，RLOTOS における記述はリフレクティブ手続きにより規定を受けており，各制約は currentg, nextg, controlg のイベント系列から構成される必要がある．以下にリフレクティブプロセスにおいて制約の集合として記述するための指針を示す．

1. 排他的な集合の和集合
2. 和集合
3. 積集合

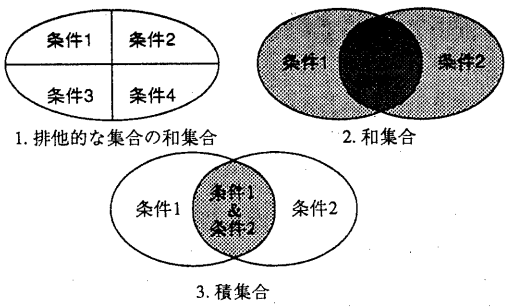

図 10: 各制約の集合

図 10 は 1 の排他的な集合の和集合，2の和集合，3 の積集合を表している．1. の排他的な集合の和集合では，各集合間には共通集合が存在しないような集合の集まりの場合であり，図 10 の各条件，条件 1,..,条件 N は互いに重複しない．以下のように記述することで各条件に対する制約をそれぞれプロセスに対応付けられる．

```
behaviour of metalevel
  const1[..] || const2[..] ||..|| const4[..]
where
process const1[..]:noexit :=
  currentg?x:Bexp; nextg?y:ActPairSet;
  ([条件 1]->
     (controlg!condition11_control(x,y,..); const1[..]
              :
      []
      controlg!condition1N_control(x,y,..); const1[..]))
   []
  ([not(条件 1)]->
      controlg?other_control:ActPair; const1[..])
endproc
    ....
```

例えば，条件 1 に関しては条件 1 に対する条件文の形で controlg での制御に関する記述で行なう．そして，条件 1 以外の場合は controlg での制御は行なわない．1. の場合，各制約条件はそれぞれ重なることはないので制約条件のうちのどれか一つ満たしていれば（例えば，図 10 では条件 1

～4 のどれか一つを満たしていれば）必ずその条件に基づいた controlg による制御が行なわれる．4.3 節で示した制約指向を用いた記述はこのタイプに属する．

2. の和集合では，1. と異なりこちらは各制約を表している集合間には共通部分が存在してもよい．以下のように記述することで，各条件に対する制約をそれぞれプロセスに対応付けられる．

```
behaviour of metalevel
  const1[..] || const2[..]
where
process const1[..]:noexit :=
  currentg?x:Bexp; nextg?y:ActPairSet;
  ([条件 1]->
      (choice z:cond_id []
          controlg!control(cond1,z); const1[..])
   [] [not(条件 1)]->
      (choice z:cond_id []
          controlg!control(notcond1,z); const1[..]))
endproc
process const2[..]:noexit :=
  currentg?x:Bexp; nextg?y:ActPairSet;
  ([条件 2]->
      (choice z:cond_id []
          controlg!control(z,cond2); const2[..])
   [] [not(条件 2)]->
      (choice z:cond_id []
          controlg!control(z,notcond2); const2[..]))
endproc
type CONTROL_FUNCTION is ActSet
  sorts cond_id
  opns cond1,cond2,notcond1,notcond2 : -> cond_id
       controlg : cond_id, cond_id -> ActPair
  eqns ofsort ActPair
       controlg(cond1,cond2)    = ...;
       controlg(cond1,notcond2) = ...;
        .....
endtype
```

2. の場合は，generalized choice オペレータ (choice z:cond_id [] ...) を用いて構成する．上述の例では，条件 1 を満たす場合は const1 プロセスは controlg!control(cond1,z) を起こせる．一方，条件 2 を満たす場合は const2 プロセスは controlg!control(z,cond2) を起こせる．const1 と const2 は controlg で同期がとれるので，control(cond1,cond2) が生起可能である．そして，control 関数の値を各条件に関して抽象データ型として定義することで各条件での controlg での制御を記述できる．1. は 2. の場合に含まれる．

1. の積集合では，以下のように記述することで各条件に対する制約をそれぞれプロセスに対応付けられる．

```
behaviour of metalevel
  const1[..] || const2[..]
where
process const1[..]:noexit :=
  currentg?x:Bexp; nextg?y:ActPairSet;
  ([条件 1]->
      (controlg!condition11_control(x,y,..); const1[..]
                :
       []
       controlg!condition1N_control(x,y,..); const1[..]))
endproc
    ....
```

各制約にはある条件で起こり得る controlg による制御を全て記述する．例えば，const1 プロセスは条件 1 に関して起こり得る制約を記述し，const2 プロセスは条件 2 に関して起こり得る制約を記述する．const1 と const2 は controlg で同期をとるので，条件 1 と条件 2 をともに満たす controlg による制御が起こり得る．

6 まとめ

本稿ではリフレクション言語による形式的仕様記述を支援するために，例外処理を扱った通信プロトコルを例題として，RLOTOS のメタレベルの記述に Vissers によって提案されている 4 つの記述スタイルを適用して記述を行なった．そしてそれに基づいて，RLOTOS におけるこれらの記述スタイルの適用方法に関して議論を行なった．

本稿で扱った 4 つの記述スタイルはシステムのモデル化については触れられていないので，今後の課題としては，一般的にいわれているモデル化手法との組合せがあげられる．

謝辞

RLOTOS について助言を頂いた富士通社会科学研究所の鵜飼孝典氏に感謝の意を示します．本研究の一部は，高度通信システム研究所の支援を受けました．

参考文献

[1] ISO 8807. *Information processing systems — Open Systems Interconnection — LOTOS — A formal description tecnique based on the temporal ordering of obsevational 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] 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.

[5] Motoshi Saeki, Takeshi Hiroi, and Takanori Ugai. Reflective Specification : Applying A Reflective Language To Formal Specification. In *Proc. of Seventh IWSSD*, pp. 204–213, 1993.

[6] C.A. Vissers, G. Scollo, and M. Sinderen. Architecture and specification style in formal descriptions of distributed systems. In *Protocol Specification, Testing and Verification VIII*, pp. 189–204, 1988.

[7] J.C.M.Beaten act W.P.Weijland. *Process Algebra*. Cambridge University Press, 1990.

[8] 更科, 安藤, 太田, 高橋. 形式的仕様記述言語 G-LOTOS の記述試験. 情報処理学会ソフトウェア工学, Vol. 92, No. 100, pp. 147–154, 1992.

[9] 内藤, 佐伯. 形式的仕様記述言語 LOTOS の記述スタイルについて. 情報処理学会ソフトウェア工学, Vol. 92, No. 100, pp. 131–138, 1992.