プログラミングー言語・基礎・実践ー　6－1
（１９９２．　３．　１２）

仕様記述言語LOTOSにおけるリフレクション：RLOTOS

鵜飼 孝典　　広井 武　　佐伯 元司

東京工業大学 工学部

本論文ではプロセス代数をもとにした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RLOTOS (Reflective LOTOS) を提案する。リフレクションでは、メタレベルとオブジェクトレベルの2つのレベルの概念を導入し、メタレベルではオブジェクトレベルの記述をデータとして扱い、それを解釈実行する。ユーザがリフレクティブプロセスと呼ばれる特別なプロセスを記述することでメタレベルの動作を制御する。はじめにLOTOSのbehaviour expressionの抽象データ型による表現とインタープリタのLOTOS言語による記述に関して考察し、最後にRLOTOSを用いてアプリケーションを実際に記述する。

Reflective Computation in LOTOS : RLOTOS

Takanori Ugai　　Takeshi Hiroi　　Motoshi Saeki

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

This paper presents RLOTOS (Reflective LOTOS) which is extension of LOTOS (Language of Temporal Ordering Specification). We embedded *reflection* or *reflective computaion* facilities to behavior expression part of LOTOS. *Reflective System* has the concept of *meta level* and *object level*. Meta level system deal with object level description as data and execute object level description. Users can control meta level system by describing special process which is called *reflective process*. First, we represented behavior expression by abstract data type and described LOTOS interpreter with *LOTOS language*. Finally, we show some application written by RLOTOS.

1 まえがき

LOTOS [1] (Language of Temporal Ordering Specification) は、 OSI (Open System Interconnection) の通信プロトコルを記述する目的で作られた形式的仕様記述言語で、並列性や非決定性、同期、割り込みなどの記述を行なうことが可能である。また LOTOS は ISO (International Standard Organization) において標準化され、実際例の記述が行なわれ始めている。オブジェクト指向を採り入れるなどの拡張に関しても研究がなされている [2]。

一方、計算途中においてその計算の過程や計算状態へのアクセスを可能にするリフレクションの機構はさまざまなプログラミング言語へ導入され、 3-LISP[3] をはじめとし、並列オブジェクト指向型言語へリフレクションを導入した ABCL/R[4] 、並列論理型言語 GHC[5] などが提案されている。リフレクションの機構によって、実行時に動的にプログラムの意味論の変更を行なうことが可能となる。これによってプログラミング言語を容易に拡張することができ、強力な記述力を持つものになる。リフレクションでは、メタレベルとオブジェクトレベルの 2 つのレベルの概念を導入し、メタレベルではオブジェクトレベルの記述をデータとして扱い、その記述を解釈実行する記述が行なえる。

LOTOS による記述は、 システムの振舞いを観測可能なアクション列によって定義する behavior expression 部分と、静的なデータに関して定義する抽象データ型の部分からなる。本研究では LOTOS の behavior expression 部分にリフレクションの概念を導入した、 RLOTOS (Reflective LOTOS) を提案する。メタレベルでは、オブジェクトレベルのプロセスの記述をデータとして扱うために、通常の LOTOS の記述を LOTOS の扱うことのできる ACT-ONE の項に書き換える必要がある。

2 章では LOTOS の behaviour expression の抽象データ型の項への書き換えについて述べる。また LOTOS 言語自身を用いて記述した LOTOS のインタプリタについても述べる。 3 章で どのように LOTOS にリフレクションのメカニズムを導入すればよいかを考察する。本手法ではまずメタプロセスを定義し、それを用いて計算を制御する方法について述べる。メタプロセスは、 LOTOS によって記述された LOTOS インタプリタプロセスと同期をとりながら、インタプリタプロセスを制御する。この制御するプロセスはリフレクティブプロセスと呼ばれる。 4 章では RLOTOS による仕様の記述方法について述べ、 5 章では RLOTOS のインタプリタの実装法について述べる。最後に RLOTOS を用いて記述した実際例を示す。

表 1: 基本的な LOTOS オペレータ

オペレータ	ACT-ONE 表現	機能
a;B	pre(a, B^*)	a を実行して B へ
B1 [ ] B2	cho($B1^*$, $B2^*$)	B1 か B2 を選択
B1 || B2	sync($B1^*$, $B2^*$)	B1 と B2 が同期
B1 ||| B2	para($B1^*$, $B2^*$)	B1 と B2 が独立
B1 |[$g_1,\ldots,g_n$]| B2	gen($B1^*$, *gate-set*, $B2^*$)	ゲートについて同期
B1 \>\> B2	ena($B1^*$, $B2^*$)	B1 を実行後 B2 へ
B1 [\> B2	dis($B1^*$, $B2^*$)	B1 へ B2 が割り込み
hide $g_1,\ldots,g_n$ in B	hid(*gate-set*, B^*)	ゲートの内部化
stop	act_stop	inaction
exit	act_exit	termination

a: イベント
B,B1, B2: behaviour expression
$g_1,\ldots g_n$: ゲート
gate-set: ゲート集合の ACT-ONE 表現
a: イベントの ACT-ONE 表現
B^*,$B1^*$, $B2^*$: behaviour expression の ACT-ONE 表現

2 LOTOS と LOTOS インタプリタ

2.1 LOTOS

LOTOS による仕様は、動的な部分を記述する behaviour expression とデータを記述する抽象データ型 (以下 ADT と略記する) とからなる。前者はプロセス代数 (CCS[6])、後者は多ソート代数 (ACTONE[7]) によって意味づけされている。 behaviour expression のみ用いて、抽象データ型によるデータを扱わない LOTOS 記述は Basic LOTOS 、 behaviour expression で抽象データ型によるデータを扱うことのできる LOTOS 記述を Full LOTOS と区別する。

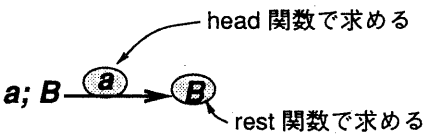
LOTOS では、システムをゲートを通して外部と通信し合う複数のプロセスとして仕様化する。それぞれのプロセスは、一般に複数のプロセスからなる階層構造になっている。プロセスの動作は、 behaviour expression によって規定される。 behaviour expression を構成するもっともアトミックなものがイベントである。

LOTOS では表 1 に挙げたオペレータを用いて behavior expression を記述する。

2.2 behaviour expression のデータ化とオペレータの遷移規則表現

LOTOS にリフレクションの概念を導入するために LOTOS の behaviour expression をデータとして表現する。 LOTOS で扱うことができるデータは抽象データ型言語 ACT-ONE の項なので、これを用いる。

LOTOS の behviour expression のそれぞれのオペレータについて、データとしての表現形を示す構成子を表 1 のように割り当てた。例えば、 (a;b[]c;b)|||(a;b[]c;d) という



head(pre(a, B)) = a;
rest(a, pre(a, B)) = B;

図 1: ラベルつき遷移システムと head と rest の関係

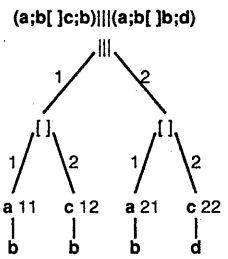


図 2: イベントの出現位置による番号つけ

behaviour expression を表 1 の規則にしたがってデータ化すると

para(cho(pre(a, b), pre(c, b)),cho(pre(a, b), pre(c, d)))

という ACT-ONE の Exp 型の項で表現される。

2.2.1 遷移規則の抽象データ型による表現

LOTOS の behaviour expression の操作的意味論として図 1 のようなラベル付き遷移システムが使用されている。状態遷移は推論規則によって定義されている [1]。LOTOS の操作的意味論を ACT-ONE の項上の操作で表現するために、現在の behaviour expression から次に起こるイベントを計算する "head" と現在の behaviour expression からあるイベントが起こった後の behaviour expression を計算する "rest" を "Exp 型" 上に定義する。B_1'、B_2' をそれぞれ B_1、B_2 の抽象データ型による表現であるとすると "$B_1 \xrightarrow{a} B_2$" という推論規則は、$\{\{a'\}\} \in head(B_1')$ であり、$rest(\{a'\}, B_1') = B_2'$ である。ただし、a' はイベントの名前 a と出現場所を表す id-number の対である。

図 2 は head と rest で同名のイベントを区別するためにイベントに付加する id-number の作成方法を示している。id-number は図 2 のように項の木構造にしたがって作成する。head と rest はこの方法にしたがって、イベントに num_atr という ACT-ONE の構成子を用いて NameId 型のイベント名に id-number を付加した Act 型を扱う。head が返すのは、同期して起こるイベントの対である ActSet 型の値の集合であり、rest の入力となるのは、イベントの対である ActSet 型の値で、これは head が返した集合の要素である。head と rest はそれぞれのオペレータの推論規則にもとづいて抽象データ型による表現が定義される。以下に、その定義の一部を示す。

```
type Exp is ...
sorts Exp
opns
  pre     : NameId,Exp -> Exp
  para    : Exp,Exp -> Exp
  cho     : Exp,Exp -> Exp
  (* constructors for behavor expression *)
  head    : Exp,Number -> ActSetSet
  rest    : ActSet,Exp -> Exp
eqns
forall E1,E2:Exp, A:Act, I:NameId,
       S1,S2:ActSet, N:Number
ofsorts ActSetSet
  head(pre(I,E1),N) =
       InsertSet(Insert(num_atr(I,N),{}),{}) ;
          (* rule for action prefix *)
  head(cho(E1,E2),N) =
       Union(head(E1,N+1),head(E2,N+2)) ;
  head(para(E1,E2),N) =
       Union(head(E1,N+1),head(E2,N+2)) ;
          (* rule for interleaving operator *)
  ....
ofsorts Exp
 rest(InsertSet(A,{}),pre(I,E1)) = E1 ;
        (* rule for action prefix *)
 IsIn(S1,head(E1)) =>
   rest(S1,cho(B1,E2)) = rest(S1,E1);
 IsIn(S1,head(E2)) =>
   rest(S1,cho(E1,E2)) = rest(S1,E2);
 IsIn(S1,head(E1)) =>
   rest(S1,para(E1,E2))=para(rest(S1,E1),E2);
 IsIn(S1,head(E2)) =>
   rest(S1,para(E1,E2))=para(E1,rest(S1,E2));
       (* rule for interleaving operator *)
  ....
endtype
```

例として、2.2 で示した (a;b[]c;b)|||(a;b[]c;d) という behaviour expression に関して、head と rest の返す値を示す。ここでは簡単のために、出現場所が *id-number* であるイベント A を示す num_atr(A, *id-number*) を "A*id-number*" と記す。また head の第 2 引数の Number 型の単位元を 0 とする。

```
head(para(cho(pre(a,b),pre(c,b)),cho(pre(a,b),pre(c,d))), 0) =
          {{a11},{a21},{c12},{c22}};
rest({a11}, para(cho(pre(a,b),pre(c,b)),
          cho(pre(a,b),pre(c,d)))) =
      para(b,cho(pre(a,b),pre(c,d)));
```

2.2.2 Basic LOTOS インタプリタの LOTOS による記述

Basic LOTOS インタプリタは図 3 のような仕組みになっている。図中のインタプリタは ACT-ONE の項表現を解釈

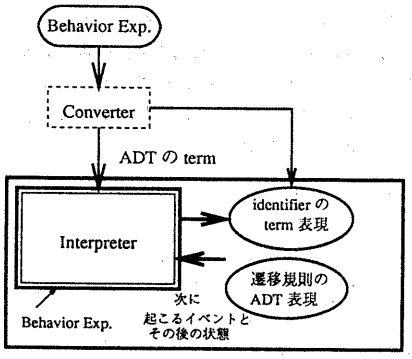


図 3: LOTOS インタプリタ

実行するためのもので LOTOS のプロセスとして記述されている。図中の "converter" は LOTOS 記述を抽象データ型の項へ書換えを行ない、LOTOS 記述中で使用されているさまざまな識別子を ACT-ONE の項として扱うための型定義部分をつくり出す。この抽象データ型の項が LOTOS で記述された "Interpreter" にわたされ、実行される。プロセス "Interpreter" は action prefix 、 choce、 general choice のオペレータを用いて次のように定義される。

```
process Interpreter[outg](exp) :=
   [eq(head(exp),{})] -> stop
   []
   [ne(head(exp),{})] ->
    (choice x: ActSet []
     ([IsIn(x,forall(head(exp))] ->
       (outg! x;
         Interpreter[outg](rest(x,exp)))
      []
      [NotIn(x,forall(head(exp)))] ->
         Interpreter[outg](exp)))
end proc
```

プロセス "Interpreter" は head と rest を用いて次に起こるイベントを順次計算しゲート outg からそのイベントに対応する項を出力する。

2.2.3 Full LOTOS インタプリタへの拡張

Full LOTOS ではデータ値を扱うことができる。イベントを通して値の受渡しを行なうことがでるので、以下に示す ACT-ONE の構成子を用いてイベントを表現する。

A !*value* ?*variable*:*type*...[*guard*]
⇓
atr(A, oup(*value*)+inp(*variable*, *type*)+..., *guard*)

atr: NameId, SymbolList, Symbol → ExNameId
oup: Symbol → Symbol
inp: NameId, SortId → Symbol
ExNameId 型： Full-LOTOS におけるイベントの型
SortId 型： 変数のタイプ宣言で用いるソートの名前を表す型

表 2: Basic LOTOS と Full LOTOS の対応

	Basic LOTOS	Full LOTOS
イベント	NameId	ExNameId
head の引数	Exp Number	Env Number
head の返値	ActSetSet	ActSetSet
rest の引数	ActSet Exp	ActSet Env
rest の返値	Exp	Env

Full LOTOS の各オペレータについても推論規則に基づいて head と rest を定義する。ただしイベントで扱う ACT-ONE の項には構成子の割当を行なっていないので、このインタプリタでは項上の操作で評価できない。そこで、ACT-ONE の項の評価は、ACT-ONE のインタプリタを作成し、それと通信することによって行なう。また、イベントで扱う ACT-ONE の項には ACT-ONE の構成子を割り当てないため、インタプリタ上では文字列として扱い、 Symbol 型の値とする。我々の作成した実行形のプロトタイプでは ACT-ONE を拡張して eval() と create() の 2 つの関数を作成することによって、Symbol 型の ACT-ONE の項を評価している。

Full LOTOS では、変数の値は behaviour expression 毎に管理される。変数とその値の対応関係を示す "Db 型" の項と、 behaviour expression の項を示す "Exp 型" から構成される項を "Env 型" とする。次に示すように、 head と rest はこの Env 型を扱う。

head : Env, Number → ActSetSet
rest : ActSet Env → Env

Basic LOTOS インタプリタと Full LOTOS インタプリタの対応を表 2 に示す。

3 LOTOS とリフレクション

本節では、 LOTOS にどのようにしてリフレクション機構を導入すればよいかについて述べる。

3.1 オブジェクトレベルとメタレベル

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という概念が存在する。メタレベル **(Meta-Lebel)** は、オブジェクトレベル (のプログラムコード) をその言語で扱うことのできるデータの形で表現されたものをデータとして持ち、オブジェクトレベルのプロセスを解釈し実行する。リフレクションではこのオブジェクトレベルとメタレベルの動作は互いに対応づけられている。

表 3: オブジェクトレベルとメタレベルの関係

オブジェクトレベル	メタレベル
ゲート a	項 (ゲートの識別子) a
プロセス p	項 (プロセスの識別子) p
変数 x	項 (変数の識別子) x
イベント a	outg !a

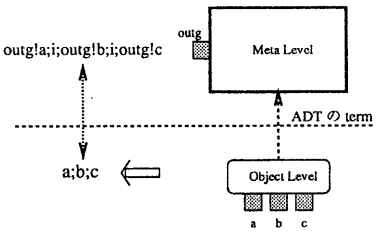


図 4: オブジェクトレベルとメタレベルでのイベントの関係

オブジェクトレベルにおけるイベントと項はメタレベルではどちらも項として扱われ、オブジェクトレベルでのイベントによる状態の遷移は、メタレベルでは項上の操作に対応している。オブジェクトレベルのプロセス "buffer" が "input" というゲートを持っているとする。メタレベルでは、"buffer" という識別子は、ProcessID 型という型の項である。同様に "input" という識別子は NameId 型という型の項である。オブジェクトレベルで、ゲート "input" でイベントが起こる時にメタレベルのプロセスで、ゲート "input" の識別子に該当する項をあるゲート (outg) から出力する。2.2.3 の Full LOTOS インタプリタと同様、オブジェクトレベルのイベントはメタレベル上のインタプリタによって特殊なゲート outg を通して渡される ACT-ONE の項に対応づけされている。例えば、オブジェクトレベルで a、b、c という順でイベントが起こる時はメタレベルでは、それに対応して図 4 のように *outg!a* 、 *outg!b* 、 *outg!c* という順で *outg* がおこる。

3.2 リフレクティブプロセスとリフレクティブゲート

2.2.3 で示した Full LOTOS のインタプリタをもとにして、インタプリタの解釈を制御する "リフレクティブプロセス (**reflective process**)" と "リフレクティブゲート" を備えた LOTOS のメタプロセスを定義する。メタプロセス中のリフレクティブプロセスはインタプリタプロセスとリフレクティブゲートを通して通信することでアクセスできる。リフレクティブゲートは currentg, nextg, restg の 3 種類であり、crrentg ゲートを用いて現在実行される

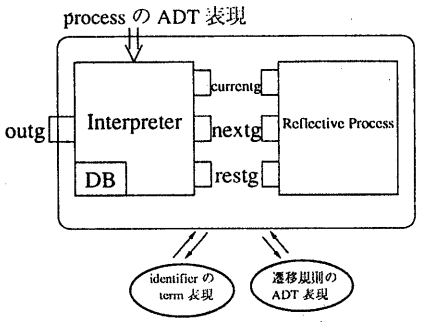


図 5: メタプロセスの内部構造

behaviour expression, nextg ゲートを用いて次に起こるイベント、 restg ゲートを用いてその次の状態が渡される。nextg と restg は次のイベントと、次のイベントが起こった後の behaviour expression を制御するために使われる。

リフレクティブプロセスはリフレクティブゲートを通してインタプリタプロセスの内部状態の情報を得ることができ、その情報を変更してリフレクティブゲートを通してインタプリタプロセスに渡すことでインタプリタプロセスの制御ができる。そしてインタプリタプロセスはリフレクティブゲートによって渡された情報にもとづいてオブジェクトレベルのプロセスを解釈する。つまり、リフレクティブプロセスがそのメタプロセスに対応するオブジェクトレベルのプロセスを制御する。メタプロセスの構造を図 5 に示す。このメタプロセスの構造を LOTOS で記述すると次のようになる。

```
process meta-process[outg](exp) :=
 hide currentg,nextg,restg in
  interpreter
          [currentg,nextg,restg,outg](exp)
 |[currentg,nextg,restg]|
  reflective_process[currentg,nextg,restg]

where
  process interpreter[currentg,nextg,
                      restg,outg](exp)
  :=  choice x:ActSet []
   [IsIn(x,head(exp))] -> currentg !exp;
     nextg !x; restg !rest(x,exp);
     nextg ?x:ActSet; outg !x;
     restg ?z:Exp;
     interpreter[currentg,nextg,restg,outg](z)
   [] [NotIn(x,head(exp))] ->
    interpreter[currentg,nextg,restg,outg](exp)
  endproc
endproc
```

したがってメタレベルの記述として、これらの 3 つのリフレクティブゲートを持ったリフレクティブプロセスが記述できるような言語構造を与えてやればよいことになる。

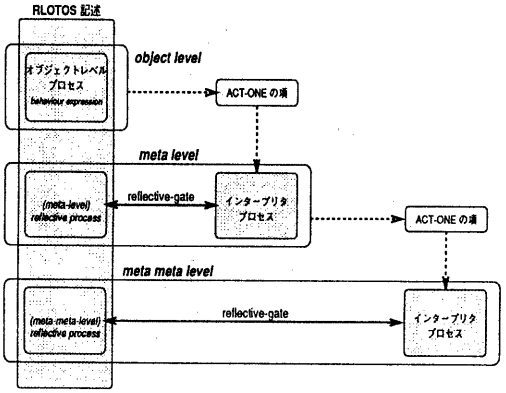


図 6: RLOTOS における各レベルの関係

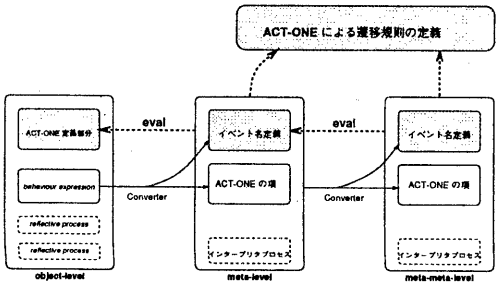


図 7: RLOTOS における各プロセスから参照できる ACT-ONE 定義

4 RLOTOS のリフレクティブ機構

4.1 RLOTOS の構造

図 6 は RLOTOS の各レベル上のプロセスの関係を示す。LOTOS にリフレクション機構を加えた RLOTOS ではインタプリタプロセスとして 2.2.3 節で述べた Full-LOTOS インタプリタとほぼ同様のものを使用する。このインタプリタプロセスへの入力には Converter を用いる。この Converter は、behaviour expression を 2.2.3 章で述べたデータ化の方法で ACT-ONE の項に変換し、オブジェクトレベルにおけるイベント名をメタレベルにおいて ACT-ONE の項として扱うためにイベント名の ACT-ONE による定義を構成する。また、RLOTOS では特別なプロセスであるリフレクティブプロセスとリフレクティブゲートを用いて各レベル（メタ、メタメタ…）のインタプリタにアクセスできる。

図 7 に示すように、メタプロセスはインタプリタプロセスがオブジェクトレベルプロセスを解釈するために、遷移規則の抽象データ型表現とオブジェクトレベルのイベント名

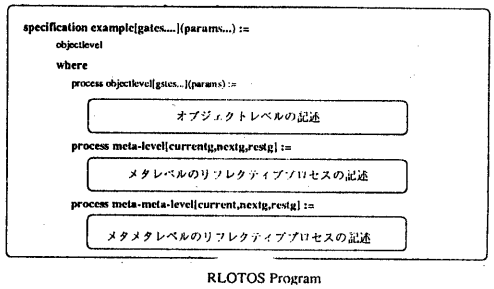


図 8: RLOTOS による記述

の ACT-ONE による定義を参照して評価する。またメタプロセスではオブジェクトレベルプロセスで使用されている抽象データ型の項を評価するために eval 関数を用いてオブジェクトレベルの抽象データ型の定義を参照評価できる。同様にしてメタメタプロセス、メタメタ…プロセスと必要に応じて存在が可能である。

4.2 RLOTOS の記述方法

RLOTOS では、ユーザの記述はオブジェクトレベルの記述と、メタレベルのリフレクティブプロセスに関する記述、メタメタレベル、あるいはそれ以上のレベルのリフレクティブプロセスに関する記述から構成され、一般には図 8 のように記述する。

メタレベルに対するリフレクティブプロセスを process meta-level、メタメタレベルに対するリフレクティブプロセスを process meta-meta-level として記述する。各レベルのインタプリタプロセスとリフレクティブゲートを通して内部状態に関する情報を通信するので、3.2 節で示したプロトコルにしたがってリフレクティブプロセスではゲート currentg, nextg, restg を使用しなくてはならない。他の記述部分に関しては LOTOS の構文にしたがう。

4.3 RLOTOS におけるリフレクティブ操作

RLOTOS では大きくわけて次の 2 つのリフレクティブ操作が可能である。

- behavior expression の順序に関する操作
- 変数、アクションに付加されている値に関する操作

RLOTOS におけるリフレクティブ操作は、リフレクティブプロセスによって行なわれる。behavior expression の順序に関する操作は、次に起こるイベントを監視し、そのイベントを別なイベントに変える、あるいは無視するなどの操作である。これらの操作に関してはメタメタレベルによっ

て、メタレベルの動作を操作することも可能である。例えば、次のようなプロセスをメタレベルに記述する。

```
process meta-level[currentg,nextg,restg]
 : noexit :=
  devil[nextg,restg]
  where
    process devil[nextg,restg] : noexit :=
      nextg? x :ActSet ; restg? y : Env ;
      ([eq(x,juice)] -> (nextg ! act_i ;exit)
      [][ne(x,juice)] -> (nextg ! x ;exit)) >>
          restg! y ; devil[nextg,restg]
    endproc
endproc
```

この process meta-level の記述は、次に起こるイベントが juice である場合、内部イベントである i に置き換える。メタレベルのプロセスでは次に起こるイベント対は ActSet 型の項として操作される。

一方、変数、アクションに付加されている値に関する操作は、主に変数やアクションの値を取得する操作と、セットする操作であると考えられる。これらの手続きは、 nextg や restg から送られる値に関する操作として定義する。

例えば、 nextg から送られるイベントやそれに付加されている値を得たいとする。その場合、抽象データ型 Action に ActSet の項からそのイベントの名前を取り出す関数 getname と、そのイベントを通して渡される値を得る関数 getvalue を以下のように定義すればよい。

```
type Action is ActSet,NameId
opns
     getname  : ActSet -> NameId
     getvalue : ActSet -> Symbol
eqns ofsort NameId
     forall name: NameId , syms : SymList ,
                 A : Act , idnum : NumList
     getname(Insert(num_atr(
          atr(name,syms),idnum),A)) = name ;
     getvalue(Insert(num_atr(
          atr(name,syms),idnum),A)) = syms ;
endtype
```

と定義される。

5 RLOTOS の実装

4 章で述べた RLOTOS のインタプリタを実現する。RLOTOS インタプリタは LOTOS のインタプリタを利用して実現している。

5.1 RLOTOS インタプリタ

図 9 は RLOTOS インタプリタの動作を示している。RLOTOS インタープリタは入力された RLOTOS 記述中のリフレクティブプロセスにもとづいて順番に各レベルを LOTOS 記述で構成する。そして構成されたレベルの中で最

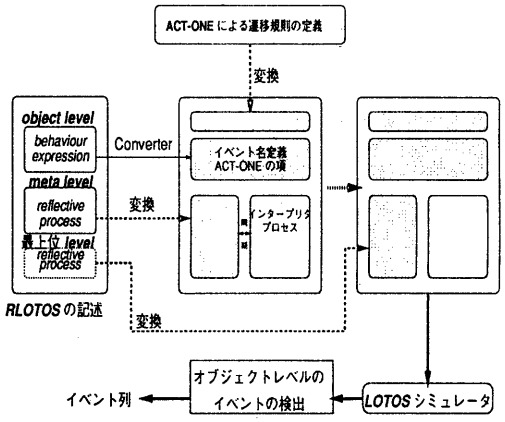


図 9: RLOTOS インタプリタの動作

上位のレベルを LOTOS シミュレータを用いて実行する。その振舞いからオブジェクトレベルのイベント列を検出して出力とする。この LOTOS シミュレータには sedos LOTOS hippo を使った。ここで使用する Converter は 2 章で述べた変換方法によって behaviour expression を ACT-ONE の項に変換するプログラムである。

5.2 メタレベルの構成方法

4 章で述べたように RLOTOS のインタプリタプロセスと、メタレベルに対応したオブジェクトレベルで定義されたリフレクティブプロセスがリフレクティブゲートで同期をとる LOTOS 記述を構成する。この LOTOS 記述をメタレベルの behaviour expression 部分とする。次に、 Converter を用いてオブジェクトレベルプロセスを ACT-ONE の項に変換して、イベント名の ACT-ONE による定義を構成する。また、図 7 にも示してある通り、遷移規則の ACT-ONE による定義は各レベルから参照できるので、遷移規則の ACT-ONE による定義と Converter で求めたイベント名の ACT-ONE 定義を結合したものをメタレベルの ACT-ONE 定義部分とする。

この behaviour expression 部分と ACT-ONE 部分の二つからなる LOTOS 記述は RLOTOS におけるメタレベルを実現するメタレベル記述である。この方法で、メタメタレベル、メタメタ... レベルを構成する。

5.3 オブジェクトレベルプロセスのイベント列検出方法

オブジェクトレベルで起こった 1 回の状態遷移が、それを解釈するためのメタレベルにおける数回の状態遷移に対応していて、異なったレベル間の対応関係は一対多である。

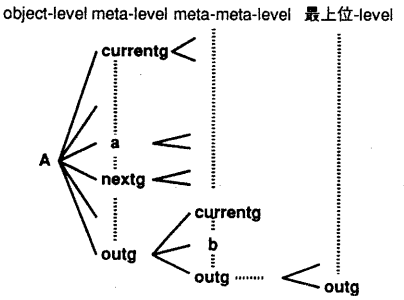


図 10: 異なったレベル間のイベントの対応関係

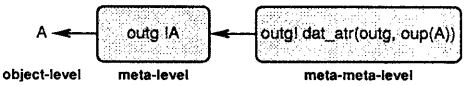


図 11: オブジェクトレベルイベント A の見え方

図 10 は異なったレベル間のイベントの対応関係を表している。オブジェクトレベルでイベント A が起こることは、メタレベルでは currentg, ... ,nextg, restg, outg が順に起こることに対応する。各レベルのインタプリタプロセスはゲート outg を通して、一つ下のレベルの実行イベントを出力する。図 11 は、オブジェクトレベルにおけるイベント A の出現がメタレベルとメタメタレベルではどのように扱われるかを示している。メタメタレベルの outg を通して渡されるデータはメタレベルの実行イベントを表している。RLOTOS インタプリタはイベント outg を結んだパスに対する最上位レベルプロセス上のゲート outg を通して渡されるデータを解釈すればオブジェクトレベルのイベントが得られる。このイベントと解釈は次のようになる。

outg! dat_atr(outg, oup(dat_atr(outg, oup(A))))
⇓
A

上段の outg はメタメタレベル上のイベントである。outg の扱うデータ dat_atr(outg, oup(dat_atr(outg, oup(A)))) はメタメタレベル上のイベント outg !dat_atr(outg, oup(A)) の抽象データ型表現である。dat_atr(outg, oup(A)) はメタレベル上のイベント outg !A の抽象データ型表現である。

6 RLOTOS の応用

6.1 悪魔の自動販売機

[1] の例題を改変して次のような悪魔の自動販売機の仕様を記述する。

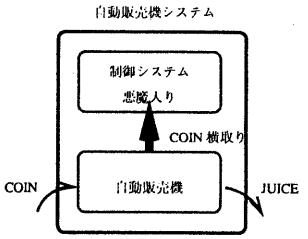


図 12: 悪魔入り自動販売機

これは coin を入れると juice がでる自動販売機を記述したものである。ただし、この自動販売機には coin を受け取っても悪魔が横取りをし juice を出さない場合がある。

この仕様を通常の LOTOS を用いて記述すると次のようになる。

```
process vender[coin,juice] : noexit :=
    coin; ((juice ;exit )
           []
            (i ; exit)) >> vender[coin,juice]
end proc
```

これを RLOTOS を用いて記述する。RLOTOS では自動販売機自体は通常の振舞いをするようにオブジェクトレベルに記述する。この例の場合、プロセス vender は coin を受けとって juice を出すように記述する。

図 12 のように悪魔はメタプロセスのリフレクティブプロセスに存在し、通常ではない動作は、環境によって引き起こされると考える。本来ならば次に juice を出す時に juice を出さずに内部イベント i に変換することによって盗む。このようにして記述した例を次に示す。これにより自動販売機本来の本質的な記述とそうでない事項が分離され見やすくなる。

```
specification system[coin,juice] : noexit
type Action is ActSet,NameId
opns
     getname  : ActSet -> NameId
eqns of sort nameid
     forall
        name: NameId , syms : SymList ,
          A : Act , idnum : NumList
      getname(Insert(num_atr(atr(name,syms),
                idnum),A)) = name ;
endtype
behavior
  vender[coin,juice]
  where
    process vender[coin,juice] : noexit :=
      coin;juice;vender[coin,juice]
    endproc

    process meta-level[currentg,nextg,restg] :
      noexit :=
      devil[nextg,restg]
      where
```

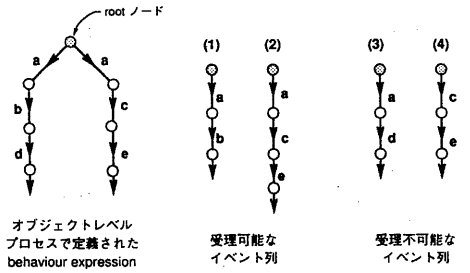


図 13: イベント列が受理可能な場合と受理不可能な場合の例

```
        process devil[nextg,restg] : noexit :=
          (nextg?x:Act;restg?y:Exp;exit(x,y))
        >> accept x :Act , y:Exp in
          ([eq(getname(x),juice)] ->
            ((nextg ! act_i ;exit(y))
            [] (nextg ! juice ;exit(y)))
          [] [ne(getname(x),juice)] ->
                          (nextg ! x ;exit(y)))
        >> accept y:Exp in
          restg! y ; devil[nextg,restg]
        endproc
      endproc
endspec
```

6.2 イベント列の受理の判定

ある behavior expression が別の有限のイベント列を受理できるかどうかを判定する。この記述は、有限のイベント列からなるオブジェクトレベルプロセスで定義した behaviour expression によって、 path で指定した有限長のイベント列が受理されるか否かを判定する。

図 13 に受理可能な場合と受理不可能な場合の例を示す。アクション木で示されたオブジェクトレベルプロセスで定義された behaviour expression に対して、 (1) と (2) のイベント列は受理可能な場合である。(3) と (4) のイベント列は受理不可能な場合である。(1) と (2) はアクション木の root ノードから始まって、同じアクション列をたどることができる枝が存在するので受理可能である。(3) はアクション木の rood ノードから 1 番目のアクションはたどることができるが 2 番目のアクションをたどるノードは存在しないので、受理不可能である。(4) はアクション木の rood ノードから 1 つめのアクションも同じアクションをたどることができないので受理不可能である。

このことを実現する RLOTOS の記述を次に示す。

```
specification  path_hantei[Success,
                 Failure, gate1,....]:noexit
(* type definition *)
type Ex_Exp is Exp, ActSetSet
opns  change : Exp,ActSetSet,Exp,Exp -> Exp;
      OneOf  : ActSetSet -> ActSet;
      Exc    : ActSetSet -> ActSetSet;
eqns    forall c_exp,exp,path:Exp,
            ASS:ActSetSet, AS:ActSet
   ofsort ActSet
   OneOf(Insert(AS,ASS)) = AS;
   ofsort ActSetSet
   Exc(Insert(AS,ASS)) = ASS;
   ofsort Exp
  ASS={} =>
   change(c_exp,ASS,exp,path) = exp;
  (ASS<>{}) and (OneOf(ASS)=head(path)) =>
   change(c_exp,ASS,exp,path) =
    change(c_exp,Exc(ASS),
       cho(rest(Exc(ASS),c_exp),exp),path);
  (ASS<>{}) and (OneOf(ASS)<>head(path)) =>
   change(c_exp,ASS,exp,path) =
    change(c_exp,Exc(ASS),exp,path);
endtype
behaviour
   (* arbitrary behaviour_expression *)
        input_behaviour_expression
where
   (* arbitrary object_level_process *)

process meta_level[currentg,nextg,restg]:noexit
  :=  check[currentg, nextg, restg](input_path)
where
  process check
          [currentg,nextg,restg](path):noexit
  := currentg ?c_exp:Exp; nextg ?x:ActSetSet;
     restg ?y:Exp;
     exit(change(c_exp,head(c_exp),
                 act_stop,path))
     >> accept ch_exp:Exp in
     owari_hantei[nextg,restg](path,ch_exp)
     >> check
          [currentg,nextg,restg](rest(path))
   where
     process owari_hantei(path, ch_exp):exit
     :=  [rest(path) eq act_stop] ->
           nextg !Success;
           restg !act_stop; exit
         [] [rest(path) nq act_stop] ->
             ([head(ch_exp) eq {}] ->
               nextg !Failure;
               rest !act_stop;exit
             [] [head(ch_exp) ne {}] ->
               nextg !x; rest !ch_exp; exit)
       endproc  (* owari_hantei *)
     endproc  (* check *)
  endproc  (* meta_level *)
endspec  (* path_hantei *)
```

この例は、リフレクティブプロセス中の変数 input_path で示したイベント列が、オブジェクトレベルプロセスで定義した behaviour expression で受理可能かどうかを判定する。受理可能な時は RLOTOS インタプリタが最後に Success を出力して終了し、受理不可能な時は Failure を出力して終了する。

図 14 はリフレクティブプロセス meta_level の状態遷移図を表している。メタレベルに対するリフレクティブプロセスは check, owari_hantei プロセスから構成され、オブジェクトレベルのプロセスを実行しながら、対象となる behavior expression と順次比較し、結果をオブジェクトレベルのイ

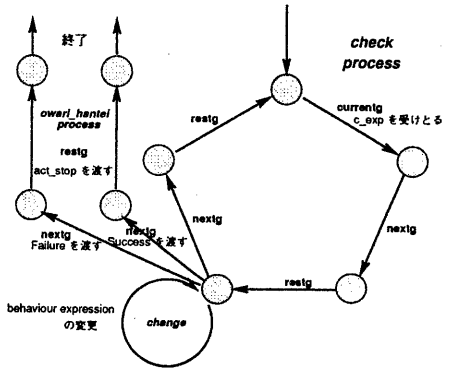


図 14: イベント列の受理可能性判定の状態遷移図

ベントとして起こす。

各プロセスの動作は次の通りである。

- input_path は対象となる有限イベント列の ACT-ONE の項による表現（behaviour expression と同様）であり、check プロセスの引数 path に入力される。

- check プロセスは、リフレクティブゲート currentg を通して渡される behaviour expression (c_exp) を change という関数で behaviour expression (ch_exp) に置き換え、これを owari_hantei プロセスで restg を通してインタプリタプロセスに渡す。変数 path で示されるイベント列の先頭イベントを除いた rest(path) と、インタプリタプロセスから currentg を通して渡された ch_exp を用いて、同様に check プロセスを再帰的に繰り返す。

- change() という関数は c_exp について head(c_exp) を計算する。head(c_exp) の各要素（次に実行可能なイベント）について hesd(path)（パスの先頭イベント）と同名なら、そのイベント実行後の behaviour expression を [ ] オペレータで結合して behaviour expression を作成する。ch_exp は、このようにして作成された behaviour expression の ACT-ONE 表現である。

- owari_hantei プロセスは終了を判定するプロセスである。path で示されるイベント列の長さが 0 になった時は、インタプリタプロセスに nextg を通して Success を渡し、restg を通して *act_stop* を渡す。そして、インタプリタプロセスは outg からイベント Success を出力して停止する。path で示されるイベント列の長さが 0 になる前に change プロセスで求めた ch_exp が act_stop なった時は、インタプリタプロセスに nextg を通して Failure を渡し restg を通して act_stop を渡す。そして、インタプリタプロセスは outg からイベント Failure を出力して停止する。この 2 つの場合以外の時は restg を通して ch_exp を渡し、check プロセスを繰り返す。

7 あとがき

本論文では LOTOS の behavior expression にリフレクションを導入することを考え、設計した RLOTOS について述べた。基本的には LOTOS 言語を用いて記述した LOTOS のインタープリタプロセスを利用する。そしてそのインタープリタプロセスを制御するリフレクティブゲートとリフレクティブプロセスを導入した。

LOTOS ではシステムをプロセスに分解して記述するので、プロセスの概念が非常に重要である。したがってプロセス毎にリフレクティブ動作が可能になることも必要であると考えられる。また RLOTOS の記述を支援するツール、設計法などを提案することが必要であると考えられる。

参考文献

[1] ISO 8807. *Information processing systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of obsevational behaviour*, 1989.

[2] Korean Experts. An object oriented extensition of lotos for distributed processing. Technical report, SG-ODP K116, 1989.

[3] B.C Smith. Reflection and semantics in Lisp. In *Proc. of 12th ACM Sympo. on POPL*, pp. 23–35, 1984.

[4] Takuo Watanabe and Akinori Yonezawa. Reflective Computation in Object-Oriented Concurrent System and Its Applications. In *Proc. of Fifth IWSSD*, pp. 56–58, 1989.

[5] Jiro Tanaka. An Experimental Reflective Programming System writtin in GHC. *Journal of Information Processing*, Vol. 14, No. 1, 1991.

[6] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[7] B. Mahr H.Ehrig. *Foundamentals of Algebraic Specification 1*. Springer-Verlag, 1985.

[8] 鵜飼孝典, 広井武, 佐伯元司. LOTOS におけるリフレクティブ計算について. 日本ソフトウェア科学会第 8 回大会論文集, pp. 537–540, Sep. 1991.