プログラミング 2−22
（1995. 8. 25）

マルチスレッド化目的コードを生成する LOTOS コンパイラの評価

安本 慶一[1] 東野 輝夫[2] 谷口 健一[2] 松浦 敏雄[3]

[1] 滋賀大学経済学部情報管理学科
[2] 大阪大学基礎工学部情報工学科
[3] 大阪市立大学生活科学部

あらまし マルチスレッド化された目的コードを生成する LOTOS コンパイラを作成した．得られる目的コードでは，LOTOS の動作式はランタイムプロセスと呼ばれる並列処理可能な部分動作式に分解され，それぞれがスレッドとして生成，実行される．ランタイムプロセス間でイベントの実行順序を保つための制御領域が目的コード内に生成される．各ランタイムプロセスは，制御領域を参照することによってイベントが実行可能かどうかの判定や全体としての実行順序を制御する．イベントが実行可能なら制御領域の内容を更新しそのイベントを実行する．実行不可能の場合そのランタイムプロセスを終了する．幾つかの実験結果から本手法により導出された目的コードが他の LOTOS コンパイラで生成された目的コードより効率が良いことが確かめられた．

Evaluation of LOTOS Compiler Generating Multi-threaded Object Codes

Keiichi Yasumoto[1], Teruo Higashino[2], Kenichi Taniguchi[2] and Toshio Matsuura[3]
[1] Dept. of Information Processing and Management, Shiga University
[2] Dept. of Information and Computer Sciences, Osaka University
[3] Faculty of Life Science, Osaka City University

Abstract We have developed a LOTOS compiler which generates multi-threaded object codes. In a derived object code, multiple runtime units in a behaviour expression are executed as threads where runtime units correspond to concurrently executable sub-expressions in the behaviour expression. To keep the temporal ordering of events among runtime units, a control area referred by all runtime units is created in the object code. Each unit refers the area to decide whether each event can be executed or not. If executable, the unit executes the event after modifying the control area. Otherwise, it kills itself. From some experimental results, derived object codes can run faster than the object codes derived by other LOTOS compilers.

1 はじめに

LOTOS [7] は通信システムやプロトコルの仕様を曖昧なく正確に記述するための形式記述言語の一つであり, ISO により国際標準として制定されている. これまでに, 多くのシステムの仕様が LOTOS で記述されており, それらの仕様から効率良い目的コード (実行可能プログラム) を自動生成するための方法が望まれている.

LOTOS の特徴として, (1) 並列, 選択, 割込などを含む強力な構文, (2) 複数の並列プロセス間でイベントを同期して実行するためのマルチランデブ機構 [7] などが挙げられる. 一般に, LOTOS 仕様における各プロセスに選択実行や, 割込などが指定されている場合, 各時点において同期可能なプロセスの組合せが複数発生し, それらの間での排他制御のため, プロセス間通信の頻度が高くなる. マルチランデブのためのプロセス間の折衝はイベント実行毎に行う必要があるため, プロセス間通信をいかに高速に行えるかが, 効率良い目的コードを生成するためのポイントとなる. また, LOTOS 仕様に含まれる各プロセスは, 基本的に並列実行されるため, 目的コード内で並列処理を効率的に行える仕組みも必要になる.

本論文では, マルチスレッド機構 (1 つのユーザプロセス内に複数の並列処理単位を生成し高速に実行する仕組み) を用いた効率の良い LOTOS 仕様の実装法を提案する. 本実装法に基づいて作成したコンパイラ [12] は, LOTOS 仕様の動作式を並列処理可能な単位動作式 (以後, ランタイムプロセスと呼ぶ) の集合に分解し, それぞれをスレッドとして生成・実行するような目的コードを生成する. 並列に実行されるランタイムプロセス間でイベントの実行順序を管理するため, 全てのランタイムプロセスから参照可能な制御領域を用いる. 制御領域は, ランタイムプロセス間の選択, 並列, 割込, 同期を実現するためのもので, 仕様の動作式に指定されているオペレータの接続関係に対応した構造になっている. 各ランタイムプロセス Rは自プロセス内の各イベントの実行前に制御領域を参照しそのイベントが実行可能かどうか調べ, 可能であれば Rを実行することを制御領域に書込み, イベントを実行する. またイベントの実行が不可能であればランタイムプロセス Rを終了する. 以上の動作により, ランタイムプロセス間の実行依存関係が実現される.

表 1: LOTOS のオペレータ

(1) アクションプレフィクス	$a;B$
(2) 選択実行	$B1 \mid\mid B2$
(3) 非同期並列実行	$B1 \mid\mid\mid B2$
(4) 同期並列実行	$B1 \mid[g_1,\cdots,g_n]\mid B2$ $B1 \mid\mid B2$
(5) 逐次実行	$B1 >> B2$
(6) 割り込み	$B1 [> B2$
(7) プロセス呼びだし	$P[g_1,\cdots,g_m](E_1,\cdots,E_l)$

(ただし, a はイベント, $B1, B2$ は上記を組み合わせてできる任意の動作式, $g_1,\cdots,g_n$は動作式 $B1, B2$ 間で同期させるゲートの並びである)

表 2: マルチランデブの生起条件

P_i	P_j	同期条件	作用
$a!E_i$	$a!E_j$	$val(E_i) = val(E_j)$	値の照合
$a!E_i$	$a?x:t$	$val(E_i) \in domain(t)$	値の代入
$a?x:t$	$a?y:u$	$t = u$	値の生成

同期の効果として, 代入 $x \leftarrow val(E_i)$, 値の生成 $x, y \leftarrow V, V \in domain(t)$ が行われる ($val(E)$ は式 Eの正規形, $domain(t)$ はソート t の値域).

表 3: 扱う動作式のクラス

$B0 =$	$hide\ HG\ in\ B1 \mid let\ EQS\ in\ B1 \mid B1$
$B1 =$	$B1 >> B2 \mid B2$
$B2 =$	$B2 [> B3 \mid B3$
$B3 =$	$B3 \mid\mid\mid B4 \mid B3 \mid\mid B4 \mid B3 \mid [SG] \mid B4 \mid B4$
$B4 =$	$B4 [] B5 \mid B5$
$B5 =$	$[EXP] -> B6 \mid B6$
$B6 =$	$a;B \mid stop \mid exit \mid (B0) \mid P[G](E)$

LOTOS 仕様では, データ型を ACT ONE [4] と呼ばれる抽象データ型記述言語により定義する. 一般に, LOTOS の抽象データ型を処理するには, 条件付き項書換え系の処理系が必要となり, クラスを制限しない限り効率の良い実装は難しい. 本コンパイラでは, 関数型プログラムとみなせる部分クラスに対して, 我々が設計・開発している関数型言語 ASL/F [6] のコンパイラを用いて, 効率の良い実装を行っている.

2 LOTOS 仕様の実装法の概略

2.1 LOTOS の概要

[オペレータ]

LOTOS では, システムの仕様を, いくつかのサブプロセスから成るプロセスとして記述する. プロセスには, 動作式として, システムの外部から観測可能な振る舞い, すなわち観測可能なイベントとそれらの時間的実行順序を指定する. イベントの実行順序の指定には, 表 1のオペレータが用いられる.

[マルチランデブの概要]

n 個のプロセス $(P_1,\cdots,P_n)$ がマルチランデブによりイベントを同期実行するには, n 個のうち任意の 2 つのプロセス $P_i, P_j (1 \le i,j \le n)$ が, 同一ゲートのイベントを実行可能でかつ, 表 2の条件を満たす必要がある [7].

本実装法で扱う LOTOS の動作式のクラスを表 3 に示す.

2.2 LOTOS 仕様の実装方針

マルチスレッド機構を用いて LOTOS 仕様を効率良く実行するには, 仕様に指定されている動作を選

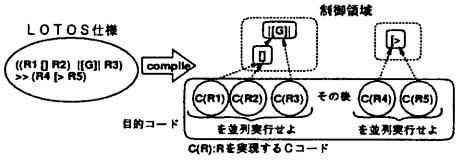

図 1: LOTOS 仕様から目的コードへの変換

表 4: 基本動作式への分解アルゴリズム

$$
\begin{array}{l}
F(B) \quad == \\
\quad if\ B = B_1\ op\ B_2, op \in \{|||, ||, |[G]|, [>\}\ then \\
\qquad F(B_1) \cup F(B_2) \\
\quad else\ if\ B = \sum_{i=1}^{n} B_i\ then \\
\qquad \{\sum_{i=1}^{k} a_i; B_i'\} \cup \bigcup_{B_i \neq a; B'} F(B_i) \\
\quad else\ \{B\}
\end{array}
$$

択的, 並列的に処理可能な部分動作に分割し, それぞれをスレッドとして処理するのが望ましい. 本稿では, 図 1に示すように, LOTOS 仕様を, ランタイムプロセスと呼ばれる並列処理可能な部分動作式群に分割し, それらの間に指定されているオペレータの階層関係を表す構文木を制御領域として生成し (生成法は 2.4にて詳述), 各ランタイムプロセスが制御領域内の適当なノードに参照・書込みを行うことで, ランタイムプロセス間の実行依存関係 (選択, 割込, 同期など) が実現され, それらのプロセス群が順次実行されていくような仕組みを考える.

2.2.1 動作式の基本動作式群への分解

各ランタイムプロセス Rをスレッドとして実行することを考慮すると, Rの動作式は並列処理を含まないイベントの逐次系列 $(a; B')$ であることが望ましい. 従って, 動作式 Bが $B_1\ op\ B_2$で表され, かつ op が並列, 同期, 割込オペレータのいずれかである場合には, Bを部分動作式 B_1, B_2に分割する. 分割された動作式 B_1, B_2についても同様に分割を行い, 分割が不可能になるまで繰り返す. 一方, 選択実行 $(B_1 [] B_2 [] ... [] B_k)$ においては, 全ての $B_i (1 \leq i \leq k)$ がイベントの逐次系列 $(a; B)$ である場合, その全体を 1 つのスレッド内で実現する. 従って, Bが選択実行 $(B_1 [] B_2 [] ... [] B_n$, 以後$\sum_{i=1}^{n} B_i$と記述する) である場合には, 上述のような Bを逐次系列の選択実行 $\sum_{i=1}^{k} B_i (B_i = a_i; B_i')$ の部分と逐次系列でない動作式の選択実行$\sum_{i=k+1}^{n} B_i (B_i \neq a; B_i')$ の部分に分割する. 以上を実現するアルゴリズム $F(B)$ を表 4に示す. $F(B)$ によって得られる集合の要素を基本動作式と定義する.

2.2.2 目的コードの内容

基本動作式は (1) イベント系列 $(a; B)$, (2) イベント系列の選択 $(\sum_{i=1}^{k} a_i; B_i)$, (3) 逐次実行 $(B_1 >> B_2)$, (4) プロセス呼出し $(P[g_1, \cdots, g_n](E_1, \cdots, E_m))$ のいずれかとなる. (1), (2) の形式の動作式をランタイムプロセスと呼ぶ.

各基本動作式 Rは C 言語の関数 $C(R)$ として実現される. $C(R)$ の内容は, (1), (2) では, 「各イベントの実行可能性を制御領域を参照して判定し, 実行可能なら, $C(R)$ が実行を行うことを制御領域に書込み, イベントを実行し, 不可能ならイベントを実行せず $C(R)$ を終了する」である.

(3) の逐次実行 $(B_1 >> B_2)$ では, 「$F(B_1)$ に含まれる各基本動作式を対応する C の関数をスレッドとして実行し, それらが全て終了するのを待った後, $F(B_2)$ に対応する C の関数群をスレッドとして実行する」のように実現する. (4) の場合は, プロセス Pの動作式 B_Pに対して「$F(B_P)$ に対応する C の関数群をスレッドとして実行する」となる. $F(B_1), F(B_2), F(B_P)$ に (3), (4) の形式の基本動作式が含まれる場合は再帰的に上記の処理を行う. また, 基本動作式の部分動作式でランタイムプロセスに分解可能なものについても, 分解後のランタイムプロセスを実現する C の関数を生成する.

```
process P[a,b,c,d]:noexit :=
    ((a;b;exit >> c;d;exit) []1 c;a;exit [] d;b;exit)
>> (a;c;exit |[a]| d;a;b;exit)
endproc
```

上記仕様から次のような動作を行う目的コードを生成する. 以下では, オペレータ $[]$, $|[a]|$ を実現するための制御領域, ランタイムプロセス群 $(a; b; exit\ (\equiv R_1)$, $c; d; exit\ (\equiv R_2)$, $c; a; exit[]d; b; exit\ (\equiv R_3)$, $a; c; exit\ (\equiv R_4)$, $d; a; b; exit\ (\equiv R_5))$, 基本動作式 $R_1 >> R_2$を実現する C の関数が既に生成されていると仮定している.

まず, プロセス Pの動作式 B_Pが $B_1 >> B_2$の形の基本動作式で, $F(B_1) = \{R_1 >> R_2, R_3\}$ なので, $C(R_1 >> R_2)$, $C(R_3)$ が制御領域のノード $[]^1$を参照点として並列に実行される. $C(R_1 >> R_2)$ では前述のようにまず $C(R_1)$ を実行し, $C(R_1)$ の終了後, $C(R_2)$ を実行する. $C(R_1)$, $C(R_3)$ が参照している制御領域のノードはともに $[]^1$なので, これらのうち最初にイベントを実行可能になったランタイムプロセスが実行権を獲得する ($C(R_1)$ の場合はイベント a, $C(R_3)$ の場合は, イベント c, d のどちらかが実行可能になれば, 実行権の獲得作業に移る). 実行権を獲得したランタイムプロセスは, ノード $[]^1$に $C(R_1)$(もしくは $C(R_3)$) が実行を行う事を書込み, イベントを実行する. 後で実行可能になったランタイムプロセスはノード $[]^1$に書込まれた内容を見て他のランタイムプロセスが既に実行されていることを検出しイベントの実行をやめプロセスを終了する. $C(R_1), C(R_3)$ が終了したら, $F(B_2) = \{R_4, R_5\}$ から, $C(R_4), C(R_5)$ が制御領域のノード $|[a]|$ を参照点として並列に実行される. R_4 はイベント a について同期が必要であることを知り, ノード $|[a]|$ に $C(R_4)$ が同期待ちであることを登録する. 一方, $C(R_5)$ は同期の必要がないイベント d の実行後, 次のイベント a の実行時にノード $|[a]|$ を見て同期相手を探す. この場合同期相手 $C(R_4)$ があるので, ノード $|[a]|$ を通じて同期可能であることを相手に通知し, a を実行する. 以上のように動作が進行して行く.

2.3 LOTOS のオペレータの実現

前述のように, 各ランタイムプロセスが選択オペレータ $[]$ に対応する制御領域のノードに参照・書込みを行うことによって, ランタイムプロセス間の選

択実行を実現する. 各ランタイムプロセスはノードを参照し, 既に他のプロセスが実行されていれば終了し, 実行されていなければ, 自分が実行することを書込み実行を継続する. 以上のことを行うには,

- 各選択オペレータ ('[]') において, どちら側 (左側あるいは右側) の動作式が選択されているかを記憶させる場所.
- 各ランタイムプロセスにおいて, 自分が選択オペレータ ('[]') のどちら側に接続されているのかの情報.

が必要である.

また, 割込 (B1 [> B2) の場合には, B2 に含まれるランタイムプロセスは, 実行の際に, 割込が起こったことを B1 に知らせる必要があり, B1 に含まれるランタイムプロセスは各イベントの実行前に割込の有無を調べて, 割込があれば実行されないようにしなければならない. したがって,

- 各割込オペレータ ('[>') において, 割込の有無を記憶する場所.
- 各ランタイムプロセスにおいて, 自分が割込を起こせるかどうかの情報 ('[>' の右側に接続されているか).

を必要とする.

また, これらのオペレータは階層的に指定されるため, 各オペレータに対応する制御領域のノードに記憶される情報 (各分岐点でどちら側が選択されたか等) は階層的に参照できるように生成されなければならない. また各ランタイムプロセスは, オペレータの階層構造中の各オペレータにおける情報 (左, 右のどちら側に属するか, 割込を起こせるか等) を持っている必要がある (以後, 制御用識別子 (制御 ID) と呼ぶ).

2.4 制御領域の生成

LOTOS 仕様 Sにおいて, メインプロセスの動作式を B_0, サブプロセスの動作式をそれぞれ $B_1, ..., B_n$ とする (ただし, $0 \leq n$). B_kを構文解析して得られる 2 分木 (ただし, 節ノードが LOTOS のオペレータで, 葉ノードがイベントまたはプロセス呼びだし) を $Tree(B_k)$ で表す.

(1) $0 \leq k \leq n$ である各 kに対して, $Tree(B_k)$ を作成する.

(2) $R \in F(B_k)$ である全ての Rについて, $Tree(B_k)$ から $Tree(R)$ を取り除いた木が B_kの制御領域となる. B_kの制御領域を $Area(B_k)$ と表す.

(3) $R \in F(B_k), R = B_1 >> B_2$の場合, B_1, B_2の制御領域 $Area(B_1), Area(B_2)$ を同様の方法で求める.

(4) $R \in F(B_k)$ の部分動作式 R'が複数のランタイムプロセスに分解可能である場合も, 同様に $Area(R')$ を求める.

上記の手順により, 必要な制御領域を仕様のコンパイル時に静的に生成できる. 例えば, プロセス

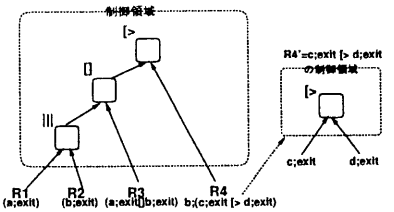

図 2: 制御領域

P:=((a;exit ||| b;exit) [] a;exit [] b;exit) [> b;(c;exit [> d;exit) では, 図 2のような制御領域が生成される.

また, $Tree(B_k)$ において, 各 $Tree(R), R \in F(B_k)$ の根ノードから, $Tree(B_k)$ の根ノードに向かって, 途中のオペレータを順に辿って行くことにより, Rの制御 ID を静的に生成する. また, Rの制御領域における参照点も同様に決定する. 制御領域の各ノードには, オペレータの種類の他に, 制御 ID の各項を保存するための領域を設定する.

2.5 目的コードの生成

LOTOS 仕様 Sの各サブプロセス B_kは目的コードの各関数 $C(B_k)$ として生成される. $C(B_k)$ の内容は制御領域 $Area(B_k)$ の宣言, $F(B_k)$ の各基本動作式 Rを実現する C の関数 $C(R)$ の並列実行である. 特に, B_0は $main()$ 関数として生成され, $Area(B_0)$ は初期の制御領域として利用される.

2.6 抽象データ型の実装

イベントの出力値 ($a!x+1$ 等), イベントの実行やある時点以降の動作を抑制するガード 式 [7]($a?x$: $int[x > 0], [f(x,y)] - > B$等), プロセス呼びだしの時に引き渡すパラメータ値 ($P[a,b,c](g(x)-1, x+10)$ 等) には, 抽象データ型で定義されている関数, データ値, 演算を用いた式が記述される.

本コンパイラでは高速化のため, これらの式が整数上の四則演算および大小関係の比較, 文字列の比較, 論理積, 論理和, 否定などのブール演算のみで記述されている場合には, その式を実現する C コードを直接生成する (この場合, これらのデータ型に対する ACT ONE による定義は必要ない). たとえば, LOTOS の動作式 $[x > 0] - > B$に対して, コンパイラは以下のコードを生成する.

```
if (x > 0){
  動作式 B に対応する C コード
}
```

抽象データ型部分に, 新たな関数が記述されている場合には, 我々が設計・開発している関数型言語 ASL/F [6] のプログラムに変換し, ASL/F のコンパイラ [6] を用いて対応する C コードを得る. ACT ONE から ASL/F への変換は, 構文的な置き換えのみで可能である (ただし, ASL/F で扱える関数型プログラムとみなせるクラスに限る).

3 ランタイムプロセスの動作アルゴリズム

2.2節で説明したように, 各基本動作式 Rは, (i) イベントの逐次系列 $(a;B)$, (ii) イベント系列間の選択実行 $(\sum_{i=1}^{k} a_i;B_i)$, (iii) 逐次実行 $(B_1 >> B_2)$, (iv) プロセス呼出し $(P[...](...))$ のいずれかである. (iii), (iv) の場合, Rは (i) または (ii) の形のランタイムプロセス群に展開される (3.3節参照). また, (i) $a;B$は (ii)$\sum_{i=1}^{k} a_i;B_i$において $k=1$ の場合と考えることができるので, 以下では Rが (ii) の形であると仮定して議論する.

各ランタイムプロセス $R=\sum_{i=1}^{k} a_i;B_i$において, Rはイベントの集合 $\{a_1,...,a_k\}(\equiv E)$ を発行可能であると定義する. 各 Rは, 次の繰り返しにより動作を行う.

1. 発行可能なイベントの集合 Eのうち実行可能なイベントの集合 $E^+(\subseteq E)$ を制御領域を参照することによって決定する.
2. E^+から 1 つのイベント a_iを選択する.
3. a_iを実行する ($E^+=\emptyset$ の時は実行終了).
4. Rの制御 ID を制御領域に書込む.
5. $B_i=\sum a;B'$の時は 1 からの処理を繰り返す.
6. $B_i \neq \sum a;B'$の時は $F(B_i)$ を生成し, $Area(B_i)$ を Rの接続している制御領域のノードに接続する (3.3節参照).

E^+の求め方は, 同期オペレータの有無によって異なるため, 以下それぞれの場合について説明する.

3.1 同期がない場合のアルゴリズム

各 '[]' オペレータにおいてどちら側が実行されたかを表すためそれぞれ l,r を, 割込 (B1 [> B2) では, B2 による割込, B1 の正常終了をそれぞれ他方に知らせるため I,Nを, それぞれ制御 ID の項目として使用する. それ以外の場合, nil (空白) を使用する.

ランタイムプロセス Rの接続している制御領域の葉ノードから根ノードに至るまでの全てのノードにおいて, 「ノードの値 = 制御 ID の対応する項目」または「ノードの値 $= nil$」が成立する時, $R(=\sum_{i=1}^{k} a_i;B_i)$ は $\{a_1,...,a_k\}$ のいずれかを実行可能である.

3.2 同期が必要な場合のアルゴリズム

同期実行 (B1 |[G]| B2) の場合, B1, B2 に含まれる各ランタイムプロセス $R(\equiv \sum_{i=1}^{k}(a_i;B_i))$ は, 以下のことを行う必要がある. R が発行可能なイベント群 ($\{a_1,...,a_k\} \equiv E$) について同期処理が必要かどうか調べる. Eのうち |[G]| の G に指定されているゲートを持つイベントの集合を E_sとする. 最初に実行を行うランタイムプロセスでは, E_sの各イベントについて, そのゲート名と入出力値を分岐点 ('|[G]|') に記憶しておき, 同期する相手を待つ. 2 番目以降に実行を行うランタイムプロセスでは, '|[G]|' に登録されている情報を参照して, E_sの各イベントと同期条件 (表 2) が成立するような同期相手があるか調べる. そのような同期相手が無くかつ他の同期候補がなければ同期は失敗となる. 他の同期候補があれば, E_sの各イベントの情報を新たに登録し, 同期相手を待つ.

同期オペレータ (|[G]|) が階層的に指定されている場合は, E_sのうち同期相手が見つかったイベント群 $E_s'(\subseteq E_s)$ について, 上部構造内のオペレータ (|[G]|) に対して再帰的に上記のことを行う (この際, E_s'の各イベントの入出力値は「値の代入」により 2 つのプロセス間で一致させた値を用いる). 最上位のオペレータ (|[G]|) について同期相手が見つかった時, 同期は成功する. ただし, ランタイムプロセス間で選択実行が指定されている場合には, 同時に複数の同期グループが存在する可能性があるため, それらの間の排他制御が必要になる.

以上のことを実現するには,

- 各同期オペレータ ('|[G]|') において, 同期するために必要な情報 (イベントのゲート名と入出力値, '|[G]|' のどちら側か) を記憶させる場所.
- 各ランタイムプロセスが '|[G]|' のどちら側に接続されているかの情報 (同期相手かどうかを判定するために必要).

が必要である.

以下では, 本論文における判定法を説明する. まず, 制御領域の各同期オペレータ ('|[G]|') に対応するノードに同期用テーブルを生成する. 同期用テーブルは, (1) ゲート, (2) 位置, (3) 状態, (4) 値リスト, の組を各行とする表として構成する (表 5参照). (1) にはゲート名, (2) にはランタイムプロセスのオペレータに対する位置 (lまたは r), (3) には同期の進行状況, (4) にはイベントの入出力値および型をそれぞれ保存する.

各ランタイムプロセス $R(\equiv \sum(a_k;B_k))$ は以下の 3 つのステップで, 実行可能性を判定する. Rが発行可能なイベントの集合 $\{a_1,...,a_k\}$ を Eとする.

ステップ 1: 同期要求

このステップでは, Eから, 実行可能なイベントの集合 E_sと, 同期相手を待っているイベントの集合 W を求める.

(1) 制御領域を葉から辿って, $a_k \in E, a_k \in G$ である '|[G]|' のノードを探す. 途中のノードの探索は同期がない場合のアルゴリズムに従う. |[G]| で同期が必要なイベントの集合を E_s^+, 同期が必要ないイベントの集合を E_s^-とする. E_s^-のイベントは, '|[G]|' に

ゲート	位置	状態	値 1		値 2		...
a	l	-	5	int	y	bool	
b	r	-	10	int	-	-	
...							

表 5: 同期用テーブル

おいて実行可能なので, 最初は $E_s = E_s^-$ とする.
以下, 各 $a_k \in E_s^+$ に対して,
(2) 同期用テーブルを調べて, イベント a_k と同期条件 (表 2参照) を満たす同期相手があるか調べる.
(3) 同期条件を満たす同期相手がなければ, 同期テーブルに, a_k の行を作成し, a_k のゲート名, 制御情報 (l 又は r), 入出力値および型を書込み, a_k を W に加える.
(4) 同期相手がある場合, a_k を E_s に加える.

制御領域のより上位の位置に, $|[G']|$ のノードがあれば, $E = E_s$ として, (1) からの処理を繰り返す (この際, 新しい E の各イベントの入出力値は, 表 2の「値の代入」により 2 つのプロセス間で一致させた値を用いる).

ステップ 2: 同期グループ間の排他制御

このステップでは, 同期グループが複数同時に存在する場合に排他制御により 1 つのグループを選択する.

(5-1) $E_s \neq \emptyset$ かつ $W = \emptyset$ の場合

E_s から 1 つのイベント a を無作為に選ぶ. a の全同期相手に $READY$ を書込み, 各同期相手の返事を待つ. 結果に応じて, ステップ 3 の処理を行う.

(5-2) $E_s = \emptyset$ かつ $W \neq \emptyset$ の場合

W のイベントのうち, いずれかの同期行の '状態' が $READY$ なったら, R と選択実行の関係にある他のランタイムプロセスが実行されていないかどうかを, 3.1節のアルゴリズムによって調べる. その結果に応じて OK または NG を書込み, '状態' が変わるのを待つ. 結果が $EXEC$ なら同期は成功となる. W 内の全てのイベントの '状態' が $RETRY$ になったら, ステップ 1 の (1) からの処理を繰り返す.

(5-3) $E_s \neq \emptyset$ かつ $W \neq \emptyset$ の場合

(7-1), (7-2) の処理のどちらを先に行うかを無作為に決め, 先に成功した方に従う.

ステップ 3: 結果の伝播

(6-1) 全ての返事が OK なら, 全同期相手の '状態' に結果 $EXEC$ を書込み, a_k を実行する. この際, 他のイベント a_k 以外のイベントの同期相手の '状態' に $RETRY$ を書込む.

(6-2) 1 つでも NG があれば, E の他のイベントを選び, (7-1) からの処理を継続する. 他のイベントが無ければ, E の各イベントの全同期相手に $RETRY$ を書込みステップ 1 からやり直す.

3.3 制御領域の結合

制御領域は, LOTOS のサブプロセス (P) ごとに生成されている (2.4 参照). ランタイムプロセス R がサブプロセス P あるいは逐次実行 $B_1 >> B_2$ の時は,

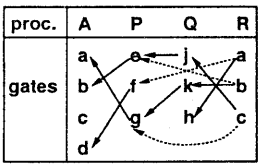

図 3: プロセス間におけるゲートの関係

サブプロセス P(逐次実行の場合は B_1) の制御領域を, より上位の制御領域に結合する必要がある. 従って, この場合 Rは以下の処理を行う.

- P (または B_1) の制御領域の根ノードを Rの接続されている (制御領域の) ノードに接続する.
- P (または B_1) に含まれる各ランタイムプロセスの制御 ID に Rの制御 ID を付加する (そのランタイムプロセスの現在の上位オペレータ構造をすべて知るため).
- P (または B_1) に含まれる全てのランタイムプロセスに対応する C の関数を並列に実行する.

$R = \sum_{i=1}^{k} a_i; B_i, B_i \neq \sum a; B'$ で a_i が選択, 実行された時も, 同様の処理を行う必要がある.

3.4 プロセス呼出しの実現

本コンパイラでは, LOTOS のプロセスは C の 1 関数として実装される. 従って, その関数を通常の手続きで呼出せば良い. ただし, LOTOS のプロセス呼出しでは, ゲートのリラベリング [7] を行う必要がある.

```
specification A[a,b,c,d]
behaviour
  P[b,d,a]
where
  P[e,f,g]:=Q[e,g] [] R[f,e,g]
  Q[j,k]:= hide h in R[h,k,j]
  R[a,b,c]:= a!1;b!2;c!3;exit
endspec
```

上のような LOTOS 仕様において, プロセス R のイベント a!1, b!2, c!3 がそれぞれどの実ゲートに対応するかを知るには, 図 3のようなプロセス間の呼出し関係を記憶しておく必要がある. 図 3において, 実線はプロセス P において左側のプロセス Q が選択された場合, 点線は右側のプロセス Rが選択された場合を表している.

本実装では, プロセス $P[a_1, ..., a_n]$ で宣言されている仮ゲート $a_1, ...a_n$ を $GTbl[0..n-1]$ という配列で表し, プロセスが呼出された時点で, この配列に呼出し元のゲートを割り当てる方法でリラベリングを実現した. 呼出し元におけるゲートのリストを配列 $PGtbl[]$ で表すとすると, 先程のプロセス $Q[j,k]$ に対して次のような C コードをコンパイラは生成する.

```
void proc_Q(GATE *PGTbl[], int G_j, int G_k)
{
```

```
  GATE *GTbl[3];

  GTbl[0] = PGTbl[G_j];
  GTbl[1] = PGTbl[G_k];
  GTbl[2] = hideによる新しいゲートの生成;
  R(GTbl, 2, 1, 0);
}
```

4 その他の工夫

4.1 目的コードの移植性

UNIX などの OS 上に, 移植性の良いマルチスレッド機構を実現するには, (1) プロセス切替えの方法, (2) スタックポインタの設定法などを, いかにアーキテクチャに依存せずに実現するかが問題となる. また, スレッド間の切り替えを円滑に行うためには, (3) 一部のスレッドの I/O 処理によって, 他のスレッドがブロックされてしまうのをうまく解決する必要がある. 我々が設計・開発した PTL (Portable Thread Library) [1] では, (1), (2) の問題について, BSD UNIX に共通なシステムコールを用いることによって, 解決を計った. ただし, システムコールでは実行効率が低くなる「スレッド生成時のスタックの初期化」については, 各アーキテクチャごとのアセンブリコードを用意することによって高速化を計っている. (3) については, 環境との通信をキャッシュを通じて行うことによって, 1 つのスレッドによるブロック期間をできるだけ短くするよう工夫している.

PTL を用いることにより, さまざまなアーキテクチャ上 (SunOS 4.1.x, Ultrix 4, DEC OSF/1, NEWS-OS 4.x, BSD/386 等) で動作し [1], かつアーキテクチャに依存する専用のマルチスレッド機構と同等以上の速度で実行可能な目的コードが生成可能になった.

4.2 ゲートと環境の対応付け

本コンパイラは, 標準では LOTOS 仕様における各ゲートを標準入出力に割り当てる. 各ゲートと環境との対応表を用意することによって, 目的コードにおいて, 特定のゲートを他のデバイスなどに対応させることも可能である.

現段階で, ゲートに割り当てることができるのは, 標準入力 (stdin), 標準出力 (stdout), 標準エラー出力 (stderr), 各デバイスファイル (/dev/*) である. 今後 TCP/IP によるソケットサービスが利用できるようコンパイラを拡張する予定である.

4.3 末端再帰のループ化

LOTOS では処理の繰り返しを, 再帰呼出しとして記述する. 本コンパイラでは, LOTOS のプロセスは C の関数に変換されるので, そのままの構造で目的コードに変換した場合, 目的コード実行時にスタックがあふれることになる. 本コンパイラでは, プロセスが次のように定義されている場合に限り, 再帰呼出しをループ化する. これ以外の形式では, コンパイラは仕様と同じ構造の目的コードを生成する.

$P[G][V] := B >> P[G'][E]$ または $a_1;...;a_k;P[G'][E]$

この場合, プロセスを呼び出すかわりに, 以下の処理を行うような目的コードを生成する.

1. ゲートリスト G を G'に置き換える.
2. パラメータ Vを Eにより上書きする.
3. プロセスの最初の処理へ戻る.

4.4 let およびパラメタ付き exit の実装

let およびパラメタ付き exit は, 仕様コンパイル時のプリプロセッサとして実現した. 例えば, let x=true in a!x; exit(3,x) ||| ...;exit(y,z-1) は仕様の構文解析の段階で hide δ in a!true;δ!3!true; exit ||[δ]|| ...; exit(y,z-1) に置き換えられる.

4.5 不要になった制御領域の削除

プロセス B_kの制御用データ領域 $Area(B_k)$ において, B_k内のランタイムプロセスが全て終了した時に, $Area(B_k)$ を開放する必要がある. そこで, 各制御領域のノードに, 接続しているランタイムプロセスの数を保存し, 次の順序で制御領域 $Area(B_k)$ の開放を行う.

- 各ランタイムプロセス Rの終了時には, Rが接続している制御領域のノードの値を 1 減らす.
- 0 になったら, 親のノードに対して同様の処理を行う. この際, $Area(B_k)$ の根ノードの値が 0 になった場合は, $Area(B_k)$ を開放する.

5 実行方式の評価

目的コードの実行効率について, 他の LOTOS コンパイラ COLOS および TOPO [8] と比較した. TOPO は, LOTOS 仕様をマシンアーキテクチャに依存しない仮想マシンに変換し, 仮想マシンから C 言語などのコードに変換することで, 移植性・汎用性を持たせている. COLOS は文献 [3] で提案されているアルゴリズムに基づいて作成されたコンパイラであり, SUN のライトウェイトプロセスと呼ばれるマルチスレッド機構を用いて動作する目的コードを生成する.

並列処理の数が増えた時のオーバーヘッドの増加度合の調査, およびコンパイラ間の比較のため, 数百のランタイムプロセスを並列実行が指定された LOTOS 仕様を実行した場合の, 単位時間あたりに実行されるイベントの数を測定した. ここで, 全てのランタイムプロセスは並列オペレータ (|||) で結合されており, 各ランタイムプロセスは, 10 個のイベントをアクションプレフィクスオペレータ (;) で結合したものである. Sun SparcStation IPX (24MB RAM) で測定した結果を表 6に示す. 並列実行においては, 本コンパイラの生成する目的コードが COLOS, TOPO に比べて, それぞれ 2.5〜3 倍, 30〜60 倍程度効率が良い. また, 1 つのスレッドに 8KB のスタック領域を割り当てた場合, 我々のコンパイラでは, 数千のランタイムプロセス群を含む LOTOS 仕様のコンパイル・実行が可能であった.

LOTOS では, システムの仕様を制約指向, 資源指向 [11] のスタイルで記述することが推奨されており, 既に多くの LOTOS 仕様がそれらのスタイル

表 6: 並列処理における単位時間あたりの実行イベント数

ランタイムプロセスの数	Ours	COLOS	TOPO
100	1724/s	700/s	57/s
200	1214/s	421/s	26/s
300	928/s	276/s	17/s
400	712/s	227/s	12/s
500	565/s	183/s	9/s

表 7: 同期処理における単位時間あたりの実行イベント数

動作式	Ours	COLOS	TOPO
B	1724/s	700/s	57/s
$B\|\|B$	244/s	509/s	12/s
$B\|\|B\|\|B$	195/s	365/s	0.9/s
$B\|\|B\|\|B\|\|B$	175/s	273/s	—
$B\|\|B\|\|B\|\|B\|\|B$	155/s	224/s	—

で記述されている. そこで, これらのスタイルで記述された仕様がどれくらい効率良く実行されるかを調べるため, 同期するプロセス (制約) の数を増やしていった時の単位時間あたりのイベント実行数の変化について調べた. 100 個のランタイムプロセスの並列実行からなる仕様 B に対して, $B||B$(制約が一つ), $B||B||B$(制約が 2 つ), $B||B||B||B$(制約が 3 つ), $B||B||B||B||B$(制約が 4 つ), を実験用の仕様として用いた. 結果を表 7に示す.

表 7によれば, 同期実行が多く含まれる場合には, COLOS の方が早い. しかし, COLOS では, 扱える LOTOS 仕様のクラスを制限している [3]. そのような制限された記述に対しては我々の実行方式をもっと精密化し, さらに効率的にすることも可能である. 我々のコンパイラは, 動作式の記述に LOTOS の全てのオペレータを使用でき, 動作式の形に制約を課していないなど, より広いクラスの LOTOS 仕様を対象にしている (表 3).

TOPO を除く既存の LOTOS コンパイラのほとんど [9, 3, 2] が抽象データ型の自動実装を行っていない. COLOS [3] では, 抽象データ型を扱える枠組みだけ備えるが, 抽象データ型で記述した関数の中身は, 設計者が手動で C コード内に記述する必要がある. 一方我々のコンパイラでは, いくつかの制限はあるが, 抽象データ型部分が関数型言語の枠内で記述されていれば, 我々の ASL/F コンパイラ [6] を用いて自動的に実装される.

文献 [3, 9] で提案されている LOTOS コンパイラは, 高速な目的コードを生成するが, ハードウェアアーキテクチャに依存したプロセススケジューラを用いるため, 移植性・汎用性に劣っていた. 一方, 我々のコンパイラが生成する目的コードは, アーキテクチャに依存しないマルチスレッド機構を利用するため, より多くのアーキテクチャ上で動作する. 以上により, 我々のコンパイラは実際のシステムやプロトコルの開発に, より広範囲に適用可能であると思われる.

6 おわりに

本論文では, マルチスレッド機構を利用した, 効率の良い LOTOS 仕様の実装法を提案した. 本実行方式は文献 [12] で我々が提案した手法に, (1) 選択実行を含むプロセス間のマルチランデブの効率化, (2) 再帰呼びだしのループ化などの局所的最適化, (3) hide, let などの未実装オペレータの実装, といった拡張を施したものである. 前節で得られた結果から, 本実行方式により, 他の方式に比べて, きわめて効率の良い目的コードを生成できることが分かった. 扱える LOTOS 仕様のクラスおよび生成される目的コードの汎用性から, 本手法は適用範囲が広いと思われる. 今後, 最適化による目的コードの性能向上と, 環境とのインターフェースの自動実装を実現し, 本コンパイラを用いたアプリケーションの開発を行う予定である.

参考文献

[1] 安倍広多, 松浦敏雄, 谷口健一: BSD UNIX 上での移植性に優れた軽量プロセス機構の実現, 情報処理, Vol. 36, No. 2, pp. 296-303 (1995).

[2] Cheng, Z., Takahashi, K., Shiratori, N. and Noguchi, S.: An Automatic Implementation Method of Protocol Specifications in LOTOS, IEICE Trans. Inf. & Syst., E75-D, 4(1992).

[3] Dubuis, E.: An Algorithm for Translating LOTOS Behavior Expressions into Automata and Ports, *Proc. of the 2nd Formal Description Techniques(FORTE'89)*, pp.163-177(1990).

[4] Ehrig, H. and Mahr, B. : Fundamentals of Algebraic Specification 1, *EATCS Monographs on Theoretical Computer Science*, Vol. 6, Springer-Verlag(1985).

[5] Gilbert, D.: Executable LOTOS: Using PARLOG to implement an FDT, *PSTV-VII*, pp. 281-294 (1987).

[6] 東野輝夫, 関浩之, 谷口健一: 代数的仕様から関数型プログラムの導出とその実行, 情報処理, Vol. 29, No. 8, pp. 881-896 (1988)

[7] ISO : LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour, *ISO 8807*(1989).

[8] Manas, J. A., Salvachia, J.: $\Lambda\beta$: a Virtual LOTOS Machine, *Proc. of the 4th Formal Description Techniques(FORTE'91)*, pp.445-460(1991).

[9] Nomura, S., Hasegawa, T. and Takizuka, T. : A LOTOS Compiler and Process Synchronization Manager, *PSTV-X*, pp.169-182(1990).

[10] Van Eijk, P., Kremer, H. and Van Sinderen, M.: On the use of specification styles for automated protocol implementation from LOTOS to C, *PSTV-X*, pp.157-168(1990).

[11] Vissers, C. A., Scollo, G. and Sinderen, M. v.: "Architecture and Specification Style in Formal Descriptions of Distributed Systems", *Proc. of the 8th Int. Symp. on Protocol Specification, Testing, and Verification(PSTV-VIII)*, pp. 189-204 (1988).

[12] 安本慶一, 由雄宏明, 東野輝夫, 谷口健一: マルチスレッド化されたオブジェクトコードを生成する LOTOS コンパイラの試作, 情処研報 PRO-94-18-15, pp. 113-120(1994).