プログラミング－言語・基礎・実践－ 18－15
（1994. 7. 21）

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

安本 慶一　由雄 宏明　東野 輝夫　谷口 健一

大阪大学 基礎工学部 情報工学科

あらまし　近年, 形式記述言語LOTOSで記述された通信システムなどの仕様から, 効率の良い目的プログラムを自動生成するための研究が進められている. LOTOSは, 多数の並行プロセス間の同期, 選択などの機能を有するため, 各並行プロセスをOSの管理するプロセスとして生成した場合, プロセス間通信およびプロセスの切り替えに時間を要し, 高速な動作が期待できない. また, 独自のスケジューラを作成して, それらを高速に行なう方法も考えられるが, 移植性に問題があった. そこで, 本研究では, 高速かつ移植性の良いLOTOSコンパイラを試作した. 本コンパイラが生成する目的プログラムは, 我々の研究グループで作成した移植性に優れた軽量プロセス機構を用いて動作する. 並行プロセス群の動作, それらの切り替え, 通信は, UNIXの1プロセス内で実現されるため, 高速動作が可能である. 出力プログラムおよび軽量プロセス機構は共にC言語のみで記述されているため, 移植性が高く, 多くのUNIXマシン上で動作する. なお本コンパイラはLOTOSの抽象データ型も扱う (ただし関数型プログラムとみなせる場合に限る).

An Implementation of LOTOS Compiler Generating Multi-threaded Object Codes

Keiichi Yasumoto, Hiroaki Yoshio, Teruo Higashino and Kenichi Taniguchi

Dept. of Information and Computer Sciences, Osaka University

Toyonaka, Osaka 560, Japan

In order to specify communication systems and distributed systems without ambiguity, LOTOS has been standardized within ISO. Recently, the researches for automatic generation of the object codes executing given LOTOS specifications have been focused. LOTOS can activate multiple parallel processes. If we assign each parallel process in a LOTOS specification to a UNIX process, the program does not run at high speed because of the overhead of process communications. To get efficient object codes, a high-speed scheduler for parallel processes is needed. We have developed a LOTOS compiler generating efficient object codes with portability. Each generated object code is executed under our *portable lightweight process mechanism*. Since the scheduling of lightweight processes and their communications are managed within one UNIX process, the program can run much faster than that using UNIX process management. The *abstract data types* in LOTOS can be also treated.

1 はじめに

近年, 通信システムやプロトコルの大規模化, 複雑化にともない, それらの仕様をあいまいなく正確に記述するための形式記述技法 (FDT: Formal Description Techniques) が重要になってきている. LOTOS[1] は FDT 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, 多くの通信システムやプロトコルの仕様が LOTOS で記述されている. その結果, LOTOS で記述された仕様を効率よく実行するためのコンパイラの開発が望まれてきた. LOTOS は, 多くの並行プロセス間の選択, 同期, 割り込みなどの機能を有するため, 仕様を実際のシステムとして実装するためには, それら並行プロセス群の取り扱い及びそれらの間の通信をいかに効率良く行なうかが問題となる.

その手段として, LOTOS 仕様から並列性を除去して *LTS*(ラベルつき遷移システム) で表し有限状態機械に変換する方式 [3] が提案されているが, 変換できる仕様に制限があり, また並列性を除去しているため, 仕様を忠実に実現できない. 並列性を実現して, 仕様における各並行動作部分を UNIX などの OS の管理する各プロセスに割り当てる方式 [4] では, 並列に動作するプロセスが多数存在する場合には, プロセスの切り替えおよびプロセス間の通信にかかるオーバヘッドが大きくなるため, 高速な動作を行なうことが出来ない. また, 独自のスケジューラを用いてプロセスの並列動作やプロセス間通信を行なう方式 [2] も実現されているが, アーキテクチャに依存した独自のスケジューラを用いているため, 移植性, 汎用性に劣る.

そこで本研究では, 高速性および移植性をともに実現する LOTOS コンパイラを試作した. 本研究のコンパイラが生成する目的プログラムは, 我々の研究グループで作成した軽量プロセス機構 [6] を用いて UNIX の単一プロセスとして実行される. そこでは, LOTOS 仕様の並列的, 選択的に実行される部分 (ランタイムプロセスと呼ぶ) は軽量プロセス群として生成され, 並行に動作する. ランタイムプロセス群の生成, 終了, 切り替え, メモリ管理などのスケジューリングは, 軽量プロセス機構が備えるスケジューラによって行なわれている. また, ランタイムプロセス間の通信は, 全ての軽量プロセスから参照できる共有データ領域を用いて実現している. 共有データ領域は, 様々なランタイムプロセス間における選択, 割り込み, 並列などの動作を実現するためのもので, 動作式で指定されているオペレータの接続関係に対応した構造となっている. 複数のランタイムプロセスがこの共有領域へ同時にアクセスすることを避けるため, 軽量プロセス機構の持つロック, アンロック機構を用いて排他制御を実現している. ランタイムプロセス群のスケジューリングおよびプロセス間通信は, 軽量プロセス機構が備えるスケジューラが行なうため, UNIX のプロセス管理による方法と比べ, きわめて高速な動作が可能である. また, 軽量プロセス機構及び出力プログラムは全て C 言語で記述されているため, 汎用性・移植性に優れている. また, 我々は関数型言語 ASL/F のコンパイラ [7] を用いて, 抽象データ型の自動実装を計った. ASL/F コンパイラでは, 関数型プログラムとみなせるクラスの抽象データ型に対して, 手続き型言語でそのプログラムを書いた場合に比べてそれ程遜色のない実行効率の目的プログラム (C プログラム) を生成する.

以下, 2で LOTOS コンパイラの実現方式を, 3で軽量プロセス機構の概要について述べる. また, 4でコンパイラの実現法について説明し, 5で簡単な性能評価を行なう.

2 コンパイラの実現方式

2.1 LOTOS の概要

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. LOTOS では, 次に示すオペレータを用いて, イベントの逐次的実行, 多数のサブプロセス間の選択実行, 並列実行, 同期, 割り込みなどの動作を指定する.

(1) アクション プレフィクス	$a;B$
(2) 選択実行	$B1[]B2$
(3) 非同期並列実行	$B1\|\|\|B2$
(4) 同期並列実行	$B1\|[G]\|B2$ $B1\|\|B2$
(5) 逐次実行	$B1 >> B2$
(6) 割り込み	$B1[> B2$

(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 は $B1, B2$ で同期させるゲートの並びとする)

2.2 実現方式

我々の研究グループでは, UNIX 上で動作する, 移植性に優れた軽量プロセス機構 (スレッドとも呼ぶ) を作成している [6]. 我々は, この軽量プロセス機構を利用して, LOTOS コンパイラを作成した [8]. 本コンパイラでは, 次の手順で LOTOS 仕様を C 言語のプログラムに変換する.

1. LOTOS 仕様を, 動作式宣言部と抽象データ型宣言部に分解する.
2. 動作式をランタイムプロセス群に分解する.
3. ランタイムプロセス群を, それぞれ軽量プロセスに割り当てる.
4. 選択, 同期, 割り込みなどの動作を実現するための共有データ領域を生成する.
5. 抽象データ型宣言部は ASL/F コンパイラ [7] を用いて, C 言語のプログラムに変換する.

目的プログラムを得るための具体的な方法, および, それらの動作原理については, 4で詳述する.

3 軽量プロセス機構

軽量プロセス機構とは, 一つのプロセスの中に, それぞれが独立した処理を実行できるような, 軽量プロセスを複数生成する機構のことを言う. 各軽量プロセスは, 一つの UNIX プロセス内で大域的なデータを共有でき, UNIX のプロセスに比べて生成, 消滅, プロセス切り替えに必要な時間が少ないという特徴を持つ. 一般に, UNIX などの OS のもとで, アーキテクチャに依存せずに, このような処理を行なわせるのは困難なため, 移植性の良い軽量プロセス機構は実現されていなかった. そこで, 我々の研究グループでは, 軽量プロセス機構を実現するための移植性の良いライブラリの実現法を検討し, 実際に作成した [6]. UNIX などの OS 上に, 移植性の良い軽量プロセス機構を実現するには (1) プロセス切替えの方法, (2) アーキテクチャに依存しないスタックポインタの設定, (3) 各軽量プロセスが用いるスタックの自動拡張, (4) 一部の軽量プロセスの入出力による全ての軽量プロセスのブロックの回避, などの問題が発生する. (1)-(3) については, 各 OS に共通なシステムコールを用いることによって, (4) については, キャッシュ機構を有するデーモンを用意することによって, 解決を計っている.

本軽量プロセス機構の特徴は以下の通りである.

- さまざまなアーキテクチャ上で動作する (SunOS 4, Ultrix 4, DEC OSF/1, NEWS-OS 4, BSD/386 等での動作を確認している).
- 特定のアーキテクチャに依存する専用の軽量プロセス機構と同等以上の速度で実行できる.
- OS の軽量プロセス拡張に対する IEEE の規格案である Pthread に準拠している.
- 軽量プロセス間で共有されるデータの相互排除機構が提供されている.

4 コンパイラの実現

4.1 諸定義

[ランタイムプロセス]

LOTOS の動作式 Sを並列的あるいは選択的に実行可能な部分動作式に分割していき, これ以上分割できなくなった時, その分割された部分動作式をランタイムプロセスと呼ぶ. Sにおいて, 最初に実行可能なイベントの集合を $\{a_1, a_2, ..., a_n\}$ とする時, 各ランタイムプロセス R_kは $a_k; B_k$ の形で表すことができる.

[例 4.1]

```
process P[a,b,c,q] : noexit :=
(((a?x:int; b!x; stop
   ||| a?y:bool; c!y;stop)
  [] b!10; P[a,b,c,q]
  [] b?x:int;(a!x;stop ||| c!(x+1);stop) )
 [> q?w:int;a!1;stop )
[> q!true;P[a,b,c,q]
endproc
```

上の仕様は R1:= a?x:int; b!x; stop, R2:= a?y:bool; c!y;stop, R3:= b!10; P[a,b,c,q], R4:= b?x:int; (a!x;stop ||| c!(x+1);stop), R5:= q?w:int; a!1; stop, R6:= q!true; P[a,b,c,q] の 6 つのランタイムプロセスを持つ.

4.2 実現の方針

本研究では次の方針で, LOTOS 仕様の階層的な実行制御を行なう.

1. 初期動作式に含まれる全てのランタイムプロセスを生成し, 並行動作させる.

2. 全てのランタイムプロセスから共通に参照できる共有メモリを用意する.

3. 各ランタイムプロセスは, 共有メモリを介して他プロセスと通信し, 自分自身の動作を決定する (結果として, 選択の場合なら片方のみ実行される).

いくつかのランタイムプロセスが並行動作するよう生成され, それらが選択的に実行されるには, 各ランタイムプロセスは自分が実行可能かどうか調べる必要がある. B1 [] B2 の場合, 最初に実行を行なうランタイムプロセスがオペレータ ('[]') にどちら側が選択されたかを記憶させ, その後実行を行なうランタイムプロセスは, '[]' に記憶されている情報を参照して, 相手側の動作式が選択されていれば, 実行を開始してはならない. 以上のことを行なうには,

- 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, 割り込み (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, 割り込みが起こったことを 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り込みの有無を調べて, 割り込み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,

- 各割り込みオペレータ ('[>') において, 割り込みの有無を記憶する場所.

- 各ランタイムプロセスにおいて, 自分が割り込み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も, 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, 割り込みを起こせるか等) を持っている必要がある. 前者のデータ構造を**共有データ領域**, 後者の情報を**制御情報**と呼ぶ.

本コンパイラでは次のように, 共有データ領域, 各ランタイムプロセスごとの制御情報を生成する.

[共有データ領域の生成方法]

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 $B_1, B_2, ..., B_n$ とする (ただし, $0 \leq n$). また, B_kに含まれるランタイムプロセスを $R_{k_i}(\equiv a_{k_i}; B_{k_i})$ とする ($1 \leq i \leq m_k$). 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 kに対して, $Tree(B_k)$ を作成する.

(2) $Tree(B_k)$ から $Tree(R_{k_i})(1 \leq i \leq m_k)$ を取り除いた木を T_kとする.

(3) T_kを B_kの共有データ領域として生成する.

(4) B_kに含まれる全ての $R_{k_i}(\equiv a_{k_i}; B_{k_i})$ の B_{k_i}に対して, 本処理を再帰的に適用する (4.4.1で使用).

上記のアルゴリズムにより, 共有データ領域を, 仕様のコンパイル時に静的に生成できる. 例えば, 例 4.1 の仕様では, 図 1のような共有データ領域が生成される.

また, $Tree(B_k)$ において, $Tree(R_{k_i})$ 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_{k_i}の制御情報 (各分岐点において左右のどちら側に属するか等の情報) を静的に生成する. また, 各 R_{k_i}が共有データ領域のどのノードを参照すべきかも同様に決定する.

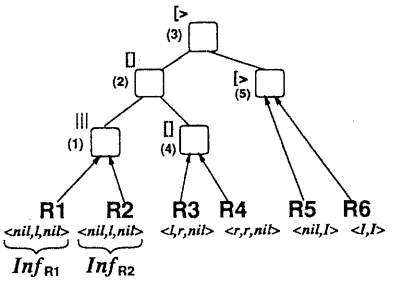

図 1: 共有データ領域

共有データ領域の各ノードには，オペレータの種類の他に，制御情報の各項を保存するための領域を設定し，内容を空白に初期化しておく．
[目的プログラムの生成]

LOTOS 仕様 Sは各サブプロセス B_kごとに，目的プログラムの各関数 C_kとして生成される．C_kの内容は共有データ領域 T_kの宣言，および B_kに含まれるランタイムプロセス $R_{k_i}(1 \leq i \leq m_k)$ の生成である．特に，B_0は $main()$ 関数として生成され，T_0は初期共有データ領域として利用される．次節に各ランタイムプロセスの動作ルーチンについて説明する．

4.3 ランタイムプロセスの動作

コンパイラは，次のように動作するランタイムプロセス $R(\equiv a; B)$ の動作ルーチンを生成する．
(1) Rは自分の制御情報と共有データ領域を比較し，実行可能性を判定する．
(2) 実行不可能なら実行をやめ，消滅する．
(3) 実行可能な時は，共有データ領域を更新し (空白のノードに制御情報を書き込み)，イベント a の実行を行なう．
(4) Bが $a'; B'$の形なら，Bに対して，1 からの処理を繰り返す．
(5) Bがそれ以外の形の時は，Bに含まれる各ランタイムプロセス R_iを生成し，R_iの共有データ領域を現在の共有データ領域に結合する (4.4.1参照)．

ランタイムプロセスの実行可能性の判定法は (i) 選択，割り込み，並列，(ii) 同期，(iii) 逐次実行，の場合で異なる．以下それぞれの場合について説明する．

4.3.1 選択，並列，割り込みの実現

各 '[]' オペレータにおいてどちら側が実行されたかを表す制御情報として，それぞれ l(左), r(右) を使用する．割り込み (B1 [> B2) では，B2 が B1 に割り込んだことを知らせるための制御情報として Iを使用する．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 の接続している共有データ領域の葉ノードにおいて制御情報を記憶する領域の内容を $Node_R[1]$ で表す．また，その親ノードの内容を $Node_R[2]$，根ノードの内容を $Node_R[n]$ で表す．例えば図 1において，$Node_{R3}[1], Node_{R3}[2], Node_{R3}[3]$ はそれぞれ，(4), (2), (3) のノードの内容を表す．また，R の制御情報の各項を $Inf_R[i]$ とする (R1 の制御情報 $\langle nil, l, nil\rangle$ に対して，$Inf_{R1}[1] = Inf_{R1}[3] = nil, Inf_{R1}[2] = l$)．
[実行可能性の判定]

$1 \leq i \leq n$ である各 i に対して，次のいずれかが成立する時，ランタイムプロセス R は実行可能である．
(1) $Inf_R[i] = Node_R[i]$
(2) $Node_R[i] = nil$
[例 4.1 の動作例]

図 1において，$R1 \rightarrow R3 \rightarrow R2 \rightarrow R5 \rightarrow R6$ の順に実行を開始することを想定する．オペレータの接続関係から，R1 実行後，R2 は実行できるが，R3, R4 は実行されてはいけない．また実行中のプロセスは，R5 あるいは R6 が実行されると中断されなければならない (R5 はさらに R6 により中断される)．

最初，共有データ領域の全てのノードの内容は空白 (nil) なので，R1 は実行可能となり，共有データ領域を更新 (図 1の (2) のノードに lを書き込む) した後，イベント a?x:int を実行する．次に，R3 が実行可能性を判定するが，$Inf_{R3}[2] \neq Node_{R3}[2](r \neq l)$ であるため，R3 は消滅する (R4 も同様に消滅する)．次に，R2 であるが，$Node_{R2}[1] = Node_{R2}[3] = nil, Inf_{R2}[2] = Node_{R2}[2]$ なので，実行可能となり，イベント a?y:bool が実行される．次に，R5 により割り込みが起こり，'[>' のノードに Iが書き込まれる ($Node_{R5}[2] = I$となる)．現在実行中の R1 は次のイベント実行前に共有データ領域を参照し，$Inf_{R1} \neq Node_{R1}[3](nil \neq I)$ であるため，消滅する (R2 も同様)．さらに R6 が割り込みを起こし，$Node_{R6}[1] = I$ となる．実行中の R5 は，次に共有データ領域参照した時，$Inf_{R5}[1] \neq Node_{R5}[1](nil \neq I)$ を検出し，消滅する．

4.3.2 同期の実現

同期実行 (B1 |[G]| B2) の場合，各ランタイムプロセスは，以下のことを行なう必要がある．まず，イベントを実行する前に，そのイベントについて同期処理が必要かどうか調べる．同期が必要なら，最初に実行を行なうランタイムプロセスのイベントにおける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．その後実行を行なうランタイムプロセスは，'|[G]|' に登録されている情報を参照して，自分の同期相手かどうか判定し，そうなら相手の入出力値と自分の入出力値が一致するか判定する．一致せず，かつ他の同期候補がなければ同期は失敗

し, 他の同期候補があれば, そのイベントの入出力値を新たに登録し, 同期相手を待つ. 値が一致した時, 同期は成功するが, 同期オペレータ ('|[G]|) が階層的に指定されている場合は, 一致させた値を確定値とみなして, 上部構造内のオペレータ ('|[G]|') について再帰的に上記のことを行なう.

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 (**同期制御領域**と呼ぶ).
- 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, 本コンパイラにおける, 同期実行を含む場合の各ランタイムプロセスの実行可能性の判定方法について説明する.

まず, 共有データ領域の各同期オペレータ ('|[G]|') に対応するノードに同期制御領域を生成する. 同期制御領域は, (1) ゲート, (2) 所属, (3) 状態, (4) 値, の組を各行とする表として構成する (表 1). (1) には, 同期を行なうイベントのゲート名, (2) には, 登録を行なったランタイムプロセスがオペレータのどちらに属しているか (l または r), (3) には, 同期が成功したかどうか (最初 nilで, 成功したら OK), (4) には, イベントの入出力値および型がそれぞれ保存される.

各ランタイムプロセス $R(\equiv a;B)$ は以下の手順で, 同期処理の実行可能性を判定する.

(1) a が G に含まれているような '|[G]|' のノードがあるかどうか調べる (なければ, 4.3.1の判定法に従う).

(2) 同期制御領域の 'ゲート', '所属' の各欄を調べて, a の同期相手を探す.

(3-1) 同期相手がいなければ, 同期制御領域に, a の行を作成し, a のゲート名, 制御情報, a の入出力値を書き込み, '状態' 欄が OK になるまで待つ.

(3-2) 同期相手があれば, 各入出力値が一致しているか調べる.

(3-3-1) 一致していれば, 確定した値をイベントの入出力値とし, さらに上位の '|[G']|'($a \in G'$) のノードがあるか調べる.

(3-3-2-1) '|[G']|'($a \in G'$) のノードがあれば (2) からの手順を繰り返す.

(3-3-2-2) なければ, 現在のノードから葉ノードまでの各 '|[G]|'($a \in G$) の同期制御領域の a 行の '状態' 欄に OK を書き込み, a が実行可能となる (同期成功).

(3-3-2) 一致していなければ, 同期制御領域に, 新たな a の行を作成し, a のゲート名, 制御情報, a の入出力値を書き込み, '状態' 欄が OK になるまで待つ.

[同期実行の動作例]

同期処理を含む次のような動作仕様を考える.

[例 4.2]

```
process Q[a,b,c] : noexit :=
 (a!5!true;stop [] a!4?x:bool;stop)
|[a]|
 (a?y:int!false;stop [] a?z:int!true;stop)
|[a]| a!5?w:bool;stop
endproc
```

上の例において, R1 := a!5!true; stop, R2 := a!4 ?x:bool; stop, R3 := a?y:int!false; stop, R4 := a ?z:int !true; stop, R5 := a!5:int?w:bool; stop とする.

$R1 \rightarrow R3 \rightarrow R2 \rightarrow R5 \rightarrow R4$ の順で, 各ランタイムプロセスが動作を開始していくものとする. 最初に R1 は, 共有データ領域の最初の '|[a]|' に同期制御領域を作成し, 入出力値, 5:int, true:bool を書き込み, 同期相手を待つ. R3 は '|[a]|' を見て, 同期の相手があるので, 自分の入出力値, y:int, false:bool と一致するか調べる. この場合一致しないので, 値を同期制御領域に登録して, 他の同期相手を待つ. R2 も同様に, '|[a]|' の同期制御領域を参照し, 同期相手 R3 との値の一致を調べる. この場合, x=false, y=4 と未定義変数の値を決めることで, 値は一致し, 上位に接続されている '|[a]|' を見に行く. ここでは同期相手がないので, 同期制御領域に確定値 4:int, false:bool を登録し, 同期相手を待つ. 次に R5 は, この値と自分の入出力値 5:int, w:bool と比較するが一致しないので, 値の登録後, 他の同期相手を待つ. 最後に, R4 は下位の '|[a]|' において, R1 の値と一致し, その時の確定値 5:int true は上位のオペレータに登録され

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

表 1: 同期制御領域の構造

ている R5 の値とも一致する. R4 は各 '|[a]|' の同期制御領域に OK を書き込むことによって, 同期が成功したことを R1, R5 に知らせる.

4.3.3 逐次実行の実現

逐次実行 (B1 >> B2) では, B1 実行後, B2 が実行されなければならない. この場合は, (1) B1 に含まれる全てのランタイムプロセスを生成し, (2) それらが全て終了したら, B2 に含まれるランタイムプロセスを生成して自分は消滅する, ような親プロセス P を生成して実現する.

プロセス Pは, 共有データ領域の '>>' のノードを通じて, B1 に含まれるランタイムプロセスが全て終了したかどうかを検出する.

4.4 その他

4.4.1 共有データ領域の結合

共有データ領域は, LOTOS のサブプロセス (P) ごとに生成されている (4.2参照). ランタイムプロセス R が $a;P$の形のときは, イベント a の実行後, サブプロセス P の共有データ領域を, 現在の共有データ領域に結合する必要がある ($R \equiv a;B$かつ R が複数のランタイムプロセスを含む時も同様). したがって, この場合, 各ランタイムプロセスは以下の処理を行なう.

- Pの共有データ領域の根ノードを Rの接続されている (共有データ領域の) ノードに接続する.
- Pに含まれる各ランタイムプロセスの制御情報に Rの制御情報を付加する (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に含まれる全てのランタイムプロセスを生成する.

4.4.2 ガーベージコレクション

選択実行 R1 [] R2 などで, R1 実行後, R2 が共有データ領域を参照しなければ, R2 は (実行は行なわないが) プロセスとして生成されたままになってしまう. そこで, 各ランタイムプロセスは, 一定時間動作を行なっていない時は, 共有データ領域を辿り, 実行不可能な場合には消滅する. また, 使用されなくなった共有データ領域のノードは, LOTOS のサブプロセス単位で実行終了時にまとめて消去を行なう.

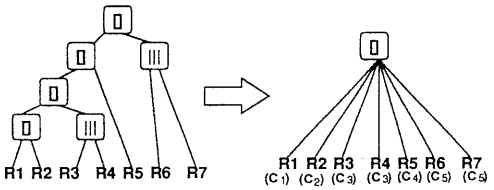

図 2: 共有データ領域の効率化

4.4.3 共有データ領域の効率化

これまで述べてきた実現法では, N個のランタイムプロセス間の選択実行に $N-1$ 個の共有領域が必要となる (共有データ領域が 2 分木であるため). そこで, N個の選択実行では, 各ランタイムプロセスに $C_1, C_2, ..., C_N$といった制御情報を持たせることによって, 実際には 1 つのデータ領域を用いた制御を行っている. また, 独立並列実行されるランタイムプロセスには, 互いに同じ制御情報を持たせることによって, '|||' に対応する共有データ領域のノードを削減している. 図 2では, R1, R2, R3, R5, R6 に異なる制御情報を, R4, R7 にそれぞれ, R3, R6 と同じ制御情報を割り当てることによって, 7 つの共有データ領域を 1 つに削減することが可能になっている (図 2).

4.4.4 抽象データ型の実現

従来のコンパイラには, LOTOS の抽象データ型の手動実装を前提としているものが多かった [2, 4]. 我々の研究グループでは, 代数的言語 ASL/F で書かれた関数型プログラムを C 言語のプログラムに変換するための ASL/F コンパイラを作成している [7]. 本 LOTOS コンパイラでは, ACT ONE で記述された LOTOS 仕様の抽象データ型の記述を, まず ASL/F プログラムに変換し, そのプログラムを ASL/F コンパイラを用いて C 言語のプログラムに変換する. 得られた C プログラムと, 動作式を変換して得られた C プログラムをリンクすることによって, 抽象データ型の自動実装を行なう. ただし, 対象とする抽象データ型は関数型プログラムとみなせるクラスに限定されている [7].

抽象データ型で記述された関数は, (1) イベントの出力値の計算 (`a!f(x)`), (2) ガード付きのイベントの

実行 (a!x[x>0]), (3) ガードの評価 ([g(x)]->a;B) を行なう場合に呼び出される. (1),(2) ではイベント実行時に, (3) では各ランタイムプロセスがイベントの実行可能性を判定する前に, これらの関数 (f(x), g(x), x>0) が呼び出される. なお, 同期実行されるイベントで入力される変数にガードによる制約が付けられている時 (例えば, a?x:int[x=2] |[a]| (a!1 [] a!2)) の処理は文献 [8] で説明している.

5 目的プログラムの評価について

コンパイラが, 効率の良い目的プログラムを生成しているかを評価する上で, 実際のプロトコルや通信システムの仕様を用いて, 実装実験を行なうことが望ましい. 本コンパイラは, 現在試作段階であるため, 今回は基本性能の調査のみを行なった. 仕様のコンパイルおよび, 目的プログラムの実行は, SONY NWS-5000 上で行ない, 時間は数回計測したうえでの平均値とした (計測時間は, プログラムを起動してから, 終了するまでの時間とした).

一般の LOTOS 仕様は, 数百の並行プロセスが高速に実行されることを前提として, 記述されている. したがって, 最大いくつのランタイムプロセスが同時に動作することができるか調べた. 各ランタイムプロセスに 6KB のスタック領域を割り当てた時, 約 3,000 個のプロセスまで並行に動作させることができた. また, 1 つのイベントを実行して終了する簡単なランタイムプロセス 600 個を並行に動作させたところ, 全てのプロセスが終了するまで, 約 4.5 秒を要した. プログラムを起動してから終了するまでに行なわれたプロセスの切り替え回数は約 3000 回, ランタイムプロセス 1 個あたりの生成時間は約 670 マイクロ秒であった.

次に, LOTOS において重要な機能であるマルチランデブ (複数プロセス間の同期処理) の処理効率を調べるため, 同期させるランタイムプロセスの数を増やしたとき, 処理時間がどのような増え方をするのか計測した. 結果を図 3に示す.

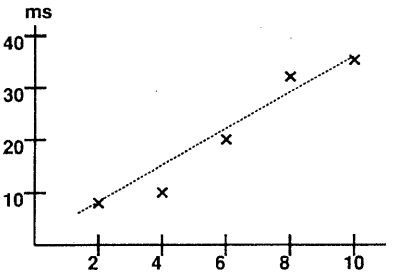

図 3: 同期するプロセス数と実行時間の関係

6 おわりに

本研究では, 汎用的な軽量プロセス機構を利用した, 移植性の良い LOTOS コンパイラの実現法を提案し実際に試作した. 本コンパイラでは, 複数プロセス間の選択, 並列, 同期, 割り込みなど LOTOS の基本的なオペレータを実現し, 抽象データ型の自動実装を行なった. 前節の結果より, 本コンパイラは実際のプロトコル仕様の実装にも適用可能であると考えられる. また, 本コンパイラの生成する目的プログラムは, 移植性が良いため, さまざまなアーキテクチャ上で動作させるために有効である. 現在未実装である LOTOS オペレータ (hide, let, par, choice, accept, パラメタ付き exit) の実装, コンパイラの最適化による速度向上, アブラカダブラプロトコルなどの実装による本コンパイラの有効性の評価などを, 今後行なっていく予定である.

参考文献

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

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

[3] 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).

[4] 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).

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

[6] 安倍広多, 松浦敏雄, 谷口健一: BSD UNIX の下でのポータブルマルチスレッドライブラリ PTL の実現, 情処研報 OS-94-62-10, pp.73-80(1994).

[7] 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)

[8] 由雄宏明: 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 コンパイラの作成, 大阪大学情報工学科特別研究報告 (1994).