マルチランデブを用いたLOTOS仕様の実行の可視化

安本慶一[†] 東野輝夫[††]
松浦敏雄[†††] 谷口健一[††]

本稿では，仕様記述言語LOTOSで記述された仕様（プログラム）の実行を可視化するための一手法を提案する．本手法では，可視化を行うためLOTOSをアニメーションが記述できるよう拡張する．拡張したLOTOSを用いて，可視化を行いたいLOTOS仕様（元仕様）に対する可視化用シナリオを記述する．元仕様とその可視化用シナリオをLOTOSの同期メカニズムを用いて，あるイベントが実行されたときに対応するアニメーションが表示されるように指定する．元仕様と可視化用シナリオの組は，我々が開発しているLOTOSコンパイラによりマルチスレッド化プログラムに変換され高速に実行される．本手法により，元仕様を変更することなく可視化用シナリオを作成でき，実行状況をリアルタイムで視覚表示可能になる．いくつかのLOTOS仕様の可視化，作成したシステムの性能評価を行い，本手法の有効性を確認した．

Visualizing Dynamic Behavior of LOTOS Specifications Using Multi-rendezvous Mechanism

KEIICHI YASUMOTO,[†] TERUO HIGASHINO,[††] TOSHIO MATSUURA[†††] and KENICHI TANIGUCHI[††]

In this paper, we propose a method for visualizing LOTOS specifications using multi-rendezvous mechanism. For visualization, we have extended LOTOS by introducing some primitive animation events. Using the extended LOTOS, we describe a visualization scenario for events which we would like to visualize their execution. Then, we execute the original specification and its visualization scenario in parallel under LOTOS multi-rendezvous mechanism so that the corresponding animation is activated when each event is executed. The pair of the original specification and its visualization scenario is converted into the multi-threaded object code using our LOTOS compiler. In our visualization method, we can specify the visualization scenario without modifying the original specification, and we can derive an object code which animates the original specification in real time. We have tried to visualize some LOTOS specifications and evaluated the performance of the derived object codes. The results have shown the usefulness of the proposed visualization method.

1. はじめに

通信プロトコルの動作を把握するには可視化が有効である．通信システムの動作は，あるイベントが実行されたときに対応するアニメーションを表示することによって可視化することができる．このような可視化を行う際には，(1) 可視化を行う対象となるシステムの動作仕様（以後，元仕様と呼ぶ）をできるだけ変更することなく，可視化のために必要な追加部分（シナリオ）を独立に作成可能なこと，(2) 元仕様における各イベントの発生のタイミングあるいはその入出力値にしたがって，表示するアニメーションの変更が可能なこと，などを実現することが望ましい．並行システムの動作を可視化する場合には，可視化された後のプログラムをできるだけ高速に実行したい．複雑な動作の可視化には，シナリオ記述言語にいくつかのアニメーション間の選択動作や並列動作を記述できる機能が望まれる．

仕様の動作の把握や，仕様の設計支援を目的としてプロトコルの可視化を行う研究がいくつか提案されている[2),4),6)]．文献2)では，仕様記述言語Estelleによるシステムの仕様の可視化法が提案されている．また，文献4)，6)ではそれぞれ独自の図的言語およびG-LOTOS（LOTOSの図形表現）を用いた仕様の設

† 滋賀大学経済学部
Faculty of Economics, Shiga University
†† 大阪大学基礎工学部
Faculty of Engineering Science, Osaka University
††† 大阪市立大学生活科学部
Faculty of Human Life Science, Osaka City University

計支援を提案している．これらの研究では，プロトコルの可視化を仕様の動作の理解あるいは仕様の設計支援に役立てようとしているため，実行系としてインタプリタを使用しており，状態遷移図やメッセージ系列の表示等のあらかじめ定められた内容の可視化を行っている．しかし，プロトコルの動作をより正確に把握するには，実行効率の高いプログラムとして実装されたプロトコルの動的実行状況をその実行効率をあまり低下させることなく視覚化できることが望ましい．また，動的実行状況を様々な視点で観察するためには，可視化の内容を利用者が自由に指定できることが望まれる．

これらの目的のため，我々は通信システムなどの仕様記述言語LOTOS[3]を可視化のためのシナリオ（以下**可視化用シナリオ**）が記述できるよう拡張し，可視化用シナリオを含むLOTOS仕様をコンパイラを用いて実行効率の高い目的プログラムとして実装する方法を提案する．

提案する可視化方法では，まず元仕様中の特定のイベントが実行されたときに表示したいアニメーションを可視化用シナリオ中に記述する．拡張したLOTOSでは，アニメーションに用いる図形の表示，移動，削除といったアニメーション操作用プリミティブが使用可能になっている．各プリミティブはLOTOSの1つのイベントとして記述される．これらのプリミティブとLOTOSの選択，並列などのオペレータを組み合わせることにより様々な可視化用シナリオが記述可能である．次に，元仕様とその可視化用シナリオが並行に実行されるようLOTOSの同期オペレータで結合した仕様（**可視化LOTOS仕様**）を作成する．可視化LOTOS仕様を実行することにより，元仕様の各イベントと対応するアニメーションの表示が同期して実行される．

LOTOSの同期機構は複数の並列プロセス間での動的なデータ交換を可能にする[3]．これを用いることにより元仕様における各イベントの入出力値を可視化用シナリオで取得し，取得したデータ値に基づいて表示するアニメーションを変えることも可能になる．

本可視化法の有効性を評価するため，可視化LOTOS仕様の実行系（可視化システム）を作成した．可視化システムは我々が開発したLOTOSコンパイラ[7]をアニメーションが扱えるよう拡張することで実現している．可視化の様子をリアルタイムで表示するため，本コンパイラは可視化LOTOS仕様からマルチスレッドを用いて動作するオブジェクトコードを生成する．アニメーションの表示には，我々が開発しているアニメーション表示支援サーバ[5]を用いた．アニメーションサーバはアニメーションに用いる各図形の属性値（位置，色，ほかに対する重なりの優先度などの情報）を受け取り，それを一定の時間間隔で表示画面に反映する．いくつかの実験により，作成した可視化システムの性能を評価した結果，プロトコルや並行システムの実行効率をそれほど落とすことなく，その動作状況を視覚化できることが確かめられ，本手法が有効であることが分かった．

以下2章で，LOTOS仕様の可視化手法について提案し，3章で，可視化システムについて説明する．4，5章では，それぞれ並行システム，通信プロトコルの例について実際に可視化を行う．6章では，可視化システムの性能評価を行い，本手法の有効性を示す．

2. LOTOS仕様の可視化法

LOTOS仕様はいくつかのサブプロセスから成るプロセスとして記述される．プロセスの振舞いとして，いくつかのイベントから成る動作式が記述される．

イベントはシステムと外部環境との相互作用で，ゲートと呼ばれる相互作用点での値の入出力として記述される．たとえばゲート名をaとすると，“a?x:int” は「ゲートaからint型の値を読み込み，新たに作成した変数xに格納する動作」を表す．また，“a!x” は「変数xの値をゲートaから出力する動作」を表す．“a?x:int?y:bool!z” のように1つのイベントで複数の入出力を記述することもできる．また，値の入出力をともなわないイベントも記述できる（この場合，イベントは “a” のようにゲート名のみで記述される）．

動作式には，イベントとその間の順序関係を次のオペレータを用いて指定する．

オペレータの種類	表記
1. アクションプレフィクス	$\alpha; B$
2. 選択	$B_1 [] B_2$
3. 非同期並列	$B_1 \|\|\| B_2$
4. 同期並列	$B_1 \|[G]\| B_2, B_1 \|\| B_2$
5. 逐次	$B_1 >> B_2$
6. 割込	$B_1 [> B_2$

（ここで，α はイベント，B，B_1，B_2 は上記オペレータとイベントを結合した任意の系列である）

たとえば，複数のイベントをオペレータ “;” で結合することによってそれらの逐次的な実行を指定する．プロセス間の選択実行の指定には，選択オペレータ “[]” を用いる．いくつかのプロセスを並列オペレータ “|||” で結合することで，それらのプロセスをほかと依存せず独立に並列実行することを指定できる．また，割込オペレータ “[>” を用いることにより，オペレー

タの右側に接続されているプロセスのイベント実行時に，オペレータの左側に接続されているプロセスを強制終了することを指定できる．

マルチランデブ[3]とは，2つ以上の並列に動作しているプロセスが，あるゲートを介して同時に1つのイベントを実行する機構のことであり，同期オペレータ（$||[G]||$）を用いて指定される（G は同期を行うゲートの集合）．

n 個のプロセスによるマルチランデブ，

$$p_1||[G_1]||p_2||[G_2]||...||[G_{n-1}]||p_n$$

において，あるゲート $g \in G_1 \cap G_2 \cap \cdots \cap G_{n-1}$ のイベントを同期実行できるのは，すべてのプロセスが g のイベントを生起でき，かつ，どの2つのプロセス p_i, p_j においても，イベントの入出力値が次の同期条件を満たすときに限る（ただし，すべてのプロセスにおいて g のイベントが値の入出力をともなわない場合，同期条件は満たされるものとする）．

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

すべてのプロセスが値を出力するイベントの生起を要求した場合には，プロセス間で値の照合が行われ，すべての型および値が一致する場合に同期が可能になる．あるプロセス p_j が値を入力するイベントの生起を要求した場合には，他のプロセス p_i でのイベントの出力値（E_i）の型が p_j の入力変数（x）の型（t）と一致するか調べられ，一致した場合には，イベントの同期実行時に，p_j の変数 x に値 E_i が代入される．

2.1 可視化用シナリオの記述法

本稿では，イベント実行時に対応するアニメーションを表示するため，マルチランデブ機構を使用する．基本的なアイデアは次のとおりである．

- 可視化用シナリオに可視化したいイベントとそのイベントの実行時に表示したいアニメーションを指定する．
- LOTOSのマルチランデブ機構を用いて元仕様とその可視化用シナリオを同期並列実行する．

図1 に示すように，LOTOS仕様 S を可視化するためには，可視化用シナリオ V を記述し，S と V をLOTOSの同期オペレータを用いて次のように結合する（ここで，G は可視化したいイベントのゲートの集合）．

$$S|[G]|V$$

$S[E]$ を元仕様の動作式とする（ここで E は仕様で用いられているすべてのゲートの集合とする）．

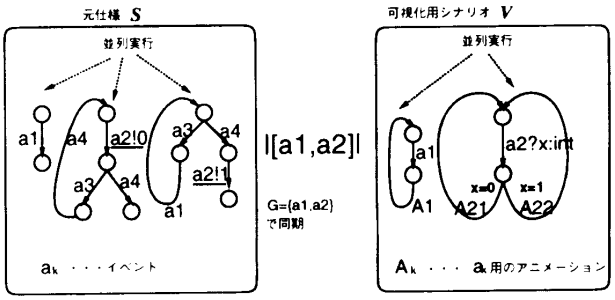

図1 マルチランデブを用いた可視化法
Fig. 1 Visualization method using multi-rendezvous.

今，仕様 $S[E]$ に含まれるイベントのうち，ゲートの集合 $G = \{a_1, ..., a_n\} \subseteq E$ に関するものを可視化したい場合，$S[E]$ の可視化用シナリオ $V[G]$ を次のように記述する（ここで，$\prod_{k=1}^{n} V_k$ は $V_1|||V_2|||...|||V_n$ を表すものとする）．

$$V[G] := \prod_{k=1}^{n} V_k[a_k]$$

$$V_k[a_k] := (a_k; A_k) >> V_k[a_k]$$

ここで，A_k はイベント a_k 実行時に表示したいアニメーションである．

あるイベントの集合 $G' \subset G$ に対して，G' の各イベントに対応するアニメーションが終了した後でのみ，次のイベント（およびアニメーション）が実行されるよう指定するには可視化用シナリオを次のように記述する（ここで $\sum_{k=1}^{n} V_k$ は $V_1[]...[]V_n$ を表す）．

$$V[G] := V'[G'] \;|||\; \prod_{a_k \notin G'} V_k[a_k]$$

$$V'[G'] := \left(\sum_{a_k \in G'} (a_k; A_k) \right) >> V'[G']$$

あるイベント a_i の生起により，現在実行中のシナリオ V_1 を強制終了し，他のシナリオ V_2 に切り替えたいときには，割込オペレータ（$[>$）を用いて次のように記述する．

$$V[G] := V_1[G - \{a_i\}] \; [> \; a_i; V_2[G]$$

この機構を用いて，同じインスタンスを持つイベントでも，そのイベントの実行タイミングに応じて異なるアニメーションを表示することができるようになり，エラー処理などについての動作状況の効果的な可視化が可能になる．

一般にイベントが実行されるときには対応するゲートを通じてデータ値が入出力される．LOTOSではマルチランデブ機構を用いることで複数の並列プロセス間でのデータ値の受け渡しが可能である[3]．イベントで入出力されるデータ値に応じてシナリオの進行を変

えたい場合には，この機構を用いて可視化用シナリオでデータ値を受け取り，異なるアニメーションを表示することが可能である．

元仕様 S に型が $sort$ の値 val を出力するイベント $a!val$ が指定されているとする．$a!val$ 実行時に val の値に応じて可視化を行う場合には，次のように記述する．

$$V[a] := a?x : sort; \left(\sum_{i=1}^{N} ([cond_i] - > A_i) \right) >> V[a]$$

ここで，N は値を区別するための条件の数である．また，A_i は条件 $cond_i$ が成立したときに表示したいアニメーションである．"$[cond_i] - > A_i$" はガード[3]と呼ばれ，条件 $cond_i$ が成立したときのみ動作式 A_i が実行可能であることを指定する構文である．

あるゲートに関するイベント群において異なるデータ型の値が入出力されることもある．たとえば，次のような動作式を可視化する場合を考える．

$S[a] := a!val_1; ... \ [] \ a!val_2; ...$

ここで，val_1, val_2 のデータ型がそれぞれ $string$, int であるとき，入出力されたデータ型に応じて異なるアニメーションを表示するには，たとえば次のように可視化用シナリオを記述する．

```
V[a] := ( (a?x : string; A11)
    []    (a?y : int; ( ([y < 0]- > A21)
                        [] ([0 ≤ y ≤ 10]- > A22)
                        [] ([10 < y]- > A23)
                        )
          )
    ) >> V[a]
```

以上のような方法で，各イベント実行時に入出力されるデータ値が可視化用シナリオに転送され，異なるアニメーションを表示可能になる．1つのイベントで複数の入出力が指定されている場合も同様である．

仕様のデバッグ時には，可視化用シナリオから元仕様において実行されるイベントの系列を制御したい場合がある．

たとえば，

```
a?data : int ;( [data eq 1]- > P
              [] [data eq 2]- > Q [] ...)
```

という元仕様に対して，V[a] := a!1; ... のように，イベントの具体的な入出力値を指定した可視化用シナリオを用意することで，元仕様においてつねにプロセスPが選択されるようにすることもできる．同様に，元仕様のあるゲートaにおいて，入力されたデータの型によって今後異なるプロセスが実行されるような場合でも，シナリオに，特定のデータ型の入力/出力イベントを記述することで，選択される系列を制御することができる．以上は，並列動作を含む仕様の実行の順序を再現する場合や，あるテスト系列に対する仕様の動作状況を表示するためなどに使用することができる．

表1 アニメーション操作用プリミティブ
Table 1 Primitives for animation operations.

プリミティブの名前	内容
CreateCast	ビットマップをキャストとして登録
CreateString	文字列をキャストとして登録
CopyCast	指定したキャストのコピーを作成
MoveCast	指定したキャストを指定位置まで指定時間で移動
DestroyCast	指定したキャストの削除
ChangeAttribute	指定したキャストの属性値の一部を変更
...	...

2.2 LOTOSによるアニメーションの記述

LOTOSでアニメーションを記述できるようにするため，最初にいくつかのアニメーション操作用プリミティブを用意する（**表 1**）．アニメーションに登場する図形を以後**キャスト**と呼ぶ．

各プリミティブをアニメーション用の特殊なゲートを介したLOTOSのイベントと見なして記述する．これらのイベントとLOTOSの並列，選択等のオペレータを組み合わせて記述することにより，様々なアニメーションを記述することができる．

ここでは，各アニメーション操作用のイベントを次のような形式で記述する（ただし，$cast\_t$ は各キャストの識別子を値域とするデータ型で，変数 id には登録したキャストの識別子が保存される）．

$$AE?id : cast\_t[id = Operation(parameters)]$$
または
$$AE!Operation(parameters)$$

たとえば，「ビットマップ "fork.xbm" をキャストとして登録する」といった操作は次のようなLOTOSのイベントとして記述する．

```
AE?fkid:cast_t[fkid=CreateCast("fork.xbm")]
```

ここで，アニメーションを表示するグラフィックウィンドウ（以後**ステージ**と呼ぶ）に対応する特殊なゲート AE を導入している．n 個のステージを使用するときには，ゲート $AE_1, AE_2, ..., AE_n$ を可視化用シナリオに宣言する．あるゲート上のイベントに対応するアニメーションをほかと区別したい場合には，$AE_{1,a}, AE_{2,b}$ のように区別を行いたいイベントのゲート名と関連づけた名前をアニメーション用ゲートの名前として用いる（2.3 節参照）．

2.3 構造的可視化

LOTOSではシステムの仕様を制約指向，資源指向

等の記述スタイルで記述することが推奨されている．大半のLOTOS仕様はモジュール化により階層的に記述されており，各資源の振舞いを指定したプロセスとそれらの間の制約を指定したプロセスが並行に動作するよう記述されている．そのようなLOTOS仕様を可視化する際には，全体の動作だけでなく一部の動作のみに注目してその動作を観察したい場合がある．

LOTOSは指定したゲートについてのイベントを外部から隠蔽するためのオペレータ **hide** を有する[3]．本可視化法では，指定したゲートに関するイベントと，そのゲートに関連づけられているアニメーション用ゲートに対するイベントを隠蔽するためのオペレータ **hideAE** を新たに導入する．

たとえば，元仕様 $S[a,b]$ は可視化用シナリオ $V[a,b]$ を用いて次のように可視化される．

VS[a, b] := S[a, b] |[a, b]| V[a, b]

ここで，ゲート a に関するイベントについてのみアニメーションを表示したい場合には，次のように **hideAE** 文を使用する．

hideAE b **in** VS[a, b]

元仕様のイベントと同時にそれに対応するアニメーション用イベントを隠蔽するためには，イベントa, bに関連づけられたアニメーション用ゲート名 AE_a, AE_b を隠蔽する必要がある（ステージが複数ある場合には，$AE_{j,a}$, $AE_{k,b}$ 等を隠蔽する）．“**hideAE** b **in** P” とすることで，プロセスP中のイベントbとイベント AE_b の両方が隠蔽される．**hideAE** オペレータは，**hide** のマクロとして実現できる．

【隠蔽されたイベントの可視化】

hide により隠蔽されたイベントの可視化は，以下のように，隠蔽されたゲート（たとえばa,b）を持つプロセス（P[a,b,c]）と，その可視化用シナリオ（V[a,b,AE]）を同じ階層で同期させることにより行う．

元仕様：hide a,b in P[a,b,c]

可視化LOTOS仕様：

hide a, b in (P[a, b, c] |[a, b]| V[a, b, AE_a, AE_b])

上記の可視化LOTOS仕様において，**hide** と **hideAE** オペレータを使い分けることによって，隠蔽されたイベントについてのアニメーション表示の有無を指定できる．

3. 可視化されたLOTOS仕様の実行

我々が開発しているLOTOSコンパイラ[7]を使用して，可視化LOTOS仕様を実行可能なオブジェクトコードに変換する．

3.1 LOTOSコンパイラ

可視化LOTOS仕様を高速に実行するためには，仕様に含まれる複数の並列プロセスをいかに高速に実行できるかが問題となる．1つの効果的な方法としてマルチスレッド機構を用いる方法がある．マルチスレッド機構では，1つのユーザプロセス内で複数の並列プロセスを効率良く取り扱うことができる．SunOS上のLight Weight Process（LWP），Mach OS上のC Thread等のマルチスレッド機構がすでに実装されているが，これらは特定のハードウェアあるいはオペレーティングシステムに依存しているため，これを用いたプログラムは移植性に問題があった．そこで，我々の研究グループでは以下の特徴を持つ移植性に優れたマルチスレッド機構PTL（Portable Thread Library）[1]を設計，開発している．

(1) 移植性のためすべてのコードをC言語で記述．
(2) 一部の処理についてはC言語によるコードに加えてアーキテクチャごとのアセンブリコードも用意．
(3) 汎用性のためPOSIX 1003.4aに基づいた標準的なインタフェースを提供．

我々はPTLを用いてLOTOS仕様を次のように実装している[7]．

- LOTOSの動作式をイベントの逐次実行系列（**ランタイムプロセス**と呼ぶ）の集合に分解．
- 各ランタイムプロセスを1つのスレッドに割当．
- ランタイムプロセス間でイベントの実行順序を実現するための共有データ領域（すべてのスレッドから参照，書き込み可能）を作成．

各スレッドではイベント実行前に次のことを行う．

- 共有データ領域を解析することにより自プロセス内のイベントを実行可能かどうか判定し，実行可能なら実行，不可能なら実行可能になるまで待機．
- 実行の必要がなくなったときはスレッド自体を終了．

複数のオペレータが階層的に指定されているような一般の動作式を実装するため，各オペレータが仕様に指定されているとおりに各ランタイムプロセスから階層的に参照されるように，共有データ領域を構成している．

コンパイラが生成する目的コードにおいて，各ランタイムプロセスがイベントを実行できるかどうかは，他のランタイムプロセスとの実行関係（制御領域の解析により実現）と外部環境の状況（具体的には通信ポートなどの各種デバイスが読み込み/書き込み可能かどうか）の両方により決定される．ただし，複数イベ

ントの選択実行では，環境に対して生起可能なイベントの実行が優先される．選択関係にある複数のイベントのうちいくつかが環境に対して生起可能である場合は，先に共有領域を参照したランタイムプロセスのイベントが実行される．たとえばa?x:int;exit [] b?y:int; exitという動作式に対して，ある時点で，ゲートa，bの両方の通信バッファにデータが格納されている場合には，実行されるイベント（a?xまたはb?y）はランタイムプロセスが共有領域を参照する順番により決定される．

アニメーションの表示には，我々の研究グループで設計，開発しているアニメーション表示支援サーバ[5]（以下アニメーションサーバ）を用いている．表1のアニメーション操作用プリミティブの大半は本アニメーションサーバの命令に1対1に対応しているため，LOTOS仕様中に現れるこれらのプリミティブをアニメーションサーバの命令に置き換えるようLOTOSコンパイラを拡張している．

3.2 リアルタイムアニメーションのメカニズム

アニメーションサーバを用いることで，キャストの登録，表示，移動といったアニメーション操作が容易に行える．本アニメーションサーバを用いてアニメーションを連続して表示するには，次の操作が必要になる．

- キャストの次のフレームにおける属性値（表示位置，色等）をアニメーションサーバ側に知らせる．
- 現在のキャストの属性値が反映されるよう表示画面（ステージ）を更新する．

本稿では，可視化用シナリオの記述を容易にするため，指定した時間に指定した位置までキャストを移動させるためのプリミティブを用意している．LOTOSは複数プロセスの並列実行の機能を有するため，複数のキャストを並行に移動させるための機構が必要になる．アニメーションを滑らかに表示するには，ステージをある一定の時間間隔で更新する必要がある．

本稿では，上記の機構を，ステージの更新を行うモジュールおよび各キャストの移動を行うモジュールの2つで，次のように構成する．簡単のため，ステージを更新する時間間隔を，たとえば，10フレーム/秒のように一定の値に固定している．

【キャストの移動を行うモジュール】

(1) ステージを更新する時間間隔，移動距離，移動時間からキャストの次のフレームにおける表示位置を計算し，アニメーションサーバに通知する．

(2) ステージが更新されるまで待ち，上の手順を移動先に到達するまで繰り返す．

【ステージの更新を行うモジュール】

(1) 一定の時間間隔でステージを更新するための命令を呼び出し，ステージが更新されたことをすべてのキャストの移動を行うモジュールに通知する．

上記のモジュールはすべてスレッドに割り当てられ，高速に実行される．マルチスレッドを用いることで，並列プロセス間の切り替えのオーバーヘッドが低くなるため，複数のキャストがリアルタイムに滑らかに移動することが可能になっている．

4. 並行システムの可視化例

本可視化手法が並行システムの動的振舞の把握に有効であるか調べるため，「哲学者の食事問題」のLOTOS仕様を本手法に基づき可視化してみた．

4.1 哲学者の食事問題の概要

「哲学者の食事問題」は並行システム上での様々な問題を議論するための典型的な例題である．本稿では「哲学者の食事問題」を次のように考えている．

- プロセスを表す5人の哲学者が「瞑想する」あるいは「食事を行う」のどちらかを繰り返しながら，並行に動作している．
- 5人の哲学者はテーブルのまわりに座っており，隣り合った任意の2人の哲学者の間には1本のフォークが配置されている．
- 各哲学者は食事を行う前には自分の左右に置かれている2本のフォークを取らなければならない．

すべての哲学者がデッドロックすることなく協調動作するためには，各フォークはそれを共有する2人の哲学者の間で公正に扱われる必要がある．

4.2 哲学者の食事問題のLOTOS仕様

哲学者およびフォークの動作を表すため以下のイベントを導入する．ここで，各フォークは取られた側から戻される必要があるため，どちら側かを表す記号l, rを導入する．

lfk!take!r	哲学者がl側のフォーク'lfk'を取る（'lfk'がr側の哲学者に取られる）
lfk!release!r	哲学者がl側のフォーク'lfk'を戻す（'lfk'がr側の哲学者から戻される）
rfk!take!l	哲学者がr側のフォーク'rfk'を取る（'rfk'がl側の哲学者に取られる）
rfk!release!l	哲学者がr側のフォーク'rfk'を戻す（'rfk'がl側の哲学者から戻される）
ph!eat	哲学者'ph'が食事を行う
ph!think	哲学者'ph'が瞑想を行う

上記のイベントを用いることによって，哲学者の動作を次のようなLOTOSのプロセスとして記述できる．

```
process Philosopher[ph,lfk,rfk]: noexit:=
( ph!think; exit
[]
  lfk!take!r;(
        rfk!take!l; ph!eat;
          rfk!release!l;
            lfk!release!r; exit
     [] lfk!release!r; exit )
) >> Philosopher[ph, lfk, rfk]
endproc
```

上記のプロセスにおいて，哲学者はまず瞑想を行うかあるいはフォーク lfk を取ろうと試みる．lfk を取ることができれば，次にフォーク rfk の取得を試みる．rfk も取ることができれば，食事を行う．フォーク rfk が取れなかったときはデッドロックを避けるため，フォーク lfk を元に戻すように記述している．

同様にフォークのプロセスは次のように記述できる．

```
process Fork[fk] : noexit :=
(  fk!take!l; fk!release!l; exit
[] fk!take!r; fk!release!r; exit
) >> Fork[fk]
endproc
```

「哲学者の食事問題」の全体は5個の哲学者のプロセスと5個のフォークのプロセスをLOTOSの並列，同期オペレータを用いて結合することで次のように記述することができる．5人の哲学者を区別するため，5個のゲート ph1,...,ph5 を，5個のフォークを区別するため，5個のゲート fk1,...,fk5 を使用する（ここで，fk1 は ph5 と ph1 に共有されているフォーク，fk2 を ph1 と ph2 の間で共有されているフォーク，... とする）☆．

```
process Philosophers[ph1,...,ph5, fk1,...,fk5]
   : noexit :=
  (Philosopher[ph1,fk1,fk2] ||| ...
||| Philosopher[ph5,fk5,fk1])
|[fk1,...,fk5]|
  (Fork[fk1] ||| ... ||| Fork[fk5])
endproc
```

4.3 可視化用シナリオ

「哲学者の食事問題」のLOTOS仕様の可視化においては，哲学者およびフォークに対応する図形（キャスト）をずっと表示しておき，元仕様Philosophersのイベントが実行されたときに対応するアニメーション（フォークの移動など）を再生することにより行うものとする．また，Philosophersにおける哲学者 ph1～ph5 は反時計まわりに配置されるように可視化を行う☆☆．

元仕様 Philosophers において，イベント「哲学者がフォークを取る」が実行されたとき，次のようなアニメーションを表示したい．

- フォークの形の図形を初期位置から哲学者の側に移動する（「フォークを戻す」の場合には，反対方向に移動する）．

同様に，イベント「哲学者が瞑想する（または，食事を行う）」が実行されたときには，次のようなアニメーションを表示したい．

- 哲学者の形をした図形を一定時間対応する形に変える（「食事を行っている図形」あるいは「瞑想を行っている図形」）．

可視化用シナリオを記述するためには，まずアニメーションに用いる図形（キャスト）の定義から始める．ここでは，5本のフォーク，5人の哲学者用のキャストを定義する．たとえば，フォーク1用のキャストは表1のプリミティブを用いて次のように定義できる．

AE?fkid1:cast_t[fkid1=CreateCast("fork1.xbm")]

次のステップで，アニメーションの内容をLOTOSで記述する．各フォークの移動のアニメーションを1つのプロセスで記述する．*home* をフォークの初期位置とする．*left (right)* を左側（右側）の哲学者がフォークを取ったときのフォークの移動先とする．フォークの移動先を選択するため，どちら側の哲学者がフォークを取ったかが分かるよう可視化用シナリオを記述する．フォークの可視化用シナリオの例を次に示す．ここで，各フォークのアニメーションの時間を t 秒と指定している．

```
process MvFork[fk,AE]
 (home,left,right:pos_t,fkid:cast_t):noexit:=
(  fk!take?dir:direction_t;
   ( [dir=l]->
        AE!MoveCast(fkid,home,left,t);exit
  [] [dir=r]->
        AE!MoveCast(fkid,home,right,t);exit
   )
[] fk!release?dir:direction_t;
   ([dir=l]->
        AE!MoveCast(fkid,left,home,t);exit
[]  [dir=r]->
        AE!MoveCast(fkid,right,home,t);exit
   )
) >> MvFork[fk,AE](home,left,right,fkid)
endproc
```

哲学者の他の動作「瞑想する」「食事を行う」に

☆ この記述では，哲学者 ph1～ph5 の配置される方向は決めていない．

☆☆ ここで，前述の l 側，r 側がそれぞれ左側，右側に決まる．

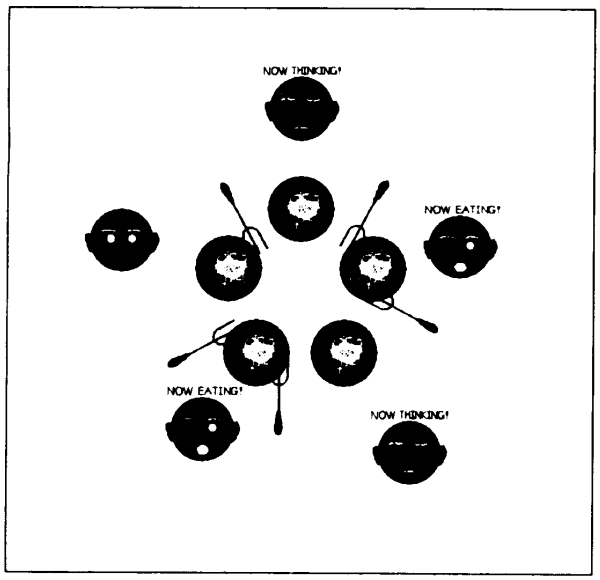

図 2 哲学者の食事問題の可視化
Fig. 2 Visualization of "dining philosophers".

ついても同様の方法で，それらの可視化用シナリオ MvPhilo[ph,AE](...) を記述することで可視化を行うことができる．

全体の可視化用シナリオ VS は次のように記述できる．

```
process VS[fk1,...,fk5,ph1,...,ph5,AE]
:noexit :=
( Definitions of all Casts ) >>
(   MvFork[fk1,AE](HOME1,LEFT1,RIGHT1,fkid1)
||| ...
||| MvFork[fk5,AE](HOME5,LEFT5,RIGHT5,fkid5)
||| MvPhilo[ph1,AE](...)
||| ...
||| MvPhilo[ph5,AE](...)
)
endproc
```

(ここで，定数 $HOME_n$, $LEFT_n$, $RIGHT_n$ はフォーク n のそれぞれ初期位置，左側/右側の移動先を表すものとする)

最後に元仕様 Philosophers とその可視化用シナリオ VS を同期オペレータで次のように結合する．

Philosophers |[fk1, ..., fk5, ph1, ..., ph5]| VS

可視化された仕様を 3 章で述べたシステムを用いて実行した例を図 **2** に示す．

5. プロトコルの可視化例

プロトコルの可視化例として，ここでは，ネットワーク上の複数の計算機から，それぞれの使用状況（ログインしているユーザ名や計算機にかかっている負荷など）を適当な時間間隔で収集する簡単なプロトコルを可視化する．このプロトコルにおけるクライアント，サーバの動作は以下のとおりである．

- 各クライアントは一定の時間間隔でその計算機を使用しているユーザ名および負荷を調べ，サーバに送信する．
- サーバは各クライアントからユーザ名，計算機の負荷の情報を受け取る．

本 LOTOS コンパイラは，特殊なゲート "system" を用いることにより，次のように UNIX 上の指定したプログラムを実行し，結果を適当なデータ型で受け取ることができるよう拡張されている．

system?data : data_t[data = program(arg_1, ..., arg_n)]

そこで，計算機にログインしているユーザの名前の集合，および計算機の負荷をそれぞれ適当なデータ型 user_t, load_t に整形して返す UNIX 上のプログラム Who(), Load() を用意し，LOTOS 仕様から呼び出すことにする．また，現在時刻を整数値に整形して返すプログラム Clock() も用意する．これらのプログラムは，csh, perl 等のスクリプトとして簡単に記述することができる．

ユーザ名，負荷をそれぞれ，T_1, T_2 秒おきに取得する場合，クライアントプロセスは，たとえば次のように LOTOS で記述できる．ここでは，サーバとの接続用に 1 つのゲート port を用いている．

```
process client[port] : noexit :=
  hide system in
   system?time : int[time = Clock()];
   ((([(time mod T1) eq 0]- >
     system?users : user_t[users = Who()]; port!users; exit)
   [] ([(time mod T1) gt 0]- > exit)
     )
    [] (([(time mod T2) eq 0]- >
     system?load : load_t[load = Load()]; port!load; exit)
   [] ([(time mod T2) gt 0]- > exit)
     )
   )>> client[port]
endproc
```

ネットワーク上の計算機の数が n のとき，サーバプロセスはそれらの計算機上のクライアントプロセスと通信するため n 個のゲート $from_1, ..., from_n$ を用いる．サーバプロセスの記述例を次に示す．

```
process server[from1, ..., fromn] : noexit :=
  hide system in
   (from1?users : user_t; exit
  [] from1?load : load_t; exit
   ...
  [] fromn?users : user_t; exit
  [] fromn?load : load_t; exit
   ) >> server[from1, ..., fromn]
endproc
```

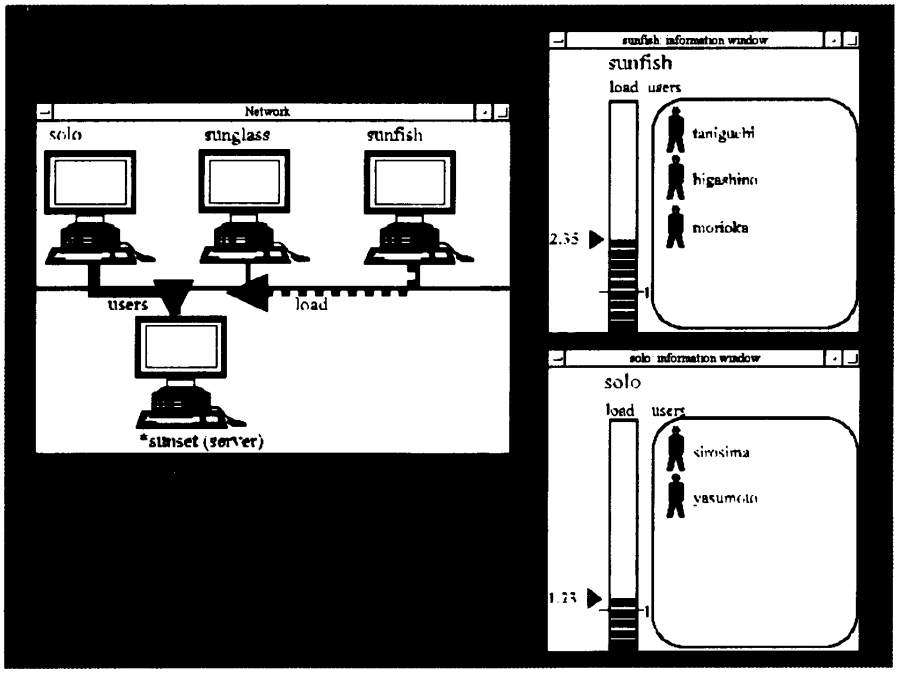

図3 プロトコルの可視化例
Fig. 3 Visualization of simple monitoring protocol.

本LOTOSコンパイラは，C言語のsocket()関数を用いてLOTOS仕様中に宣言されている各ゲートをTCP/IPのあるポートに対応させることが可能である．ここでは，n 個のクライアントプロセスのポート $port_1, ..., port_n$，およびサーバプロセスのポート $from_1, ..., from_n$ に対して，各 $port_k$ と $from_k$ をsocket()により連結することで，クライアント，サーバ間で通信させる．

次にサーバプロセスserverを $n+1$ 個のステージ $(AE_0, AE_1, ..., AE_n)$ を用いて可視化する．最初のステージ (AE_0) には，n 個の計算機を表す図形およびそれらの間の通信路を表示しておき，クライアントからの通信があったときには，矢印がクライアントの計算機からサーバの側に伸びていくアニメーション $(Anim_1, Anim_2)$ を表示するよう可視化用シナリオV_commを記述する（$Anim_1, Anim_2$ はそれぞれ，ユーザ情報通信時，負荷情報の通信時に表示するアニメーションで，それぞれ，実線，破線を用いている）．各計算機 k の使用状況はステージ k (AE_k) に表示する．ユーザ名を取得したときには，各ユーザを表す図形がステージに表示されるよう $(Anim_3)$ 可視化用シナリオV_usersを記述する．また，負荷の情報を取得したときには，棒グラフ上で現在の負荷を示すインジケータが前回の位置から新しい位置に動いて行くように $(Anim_4)$ 可視化用シナリオV_loadを記述する．

```
process V_comm[from_k, AE_k](...) : noexit :=
  (from_k?users : user_t; Anim_1[AE_k](...)
[] from_k?load : load_t; Anim_2[AE_k](...)
  ) >> V_comm[...](...)
endproc
process V_users[from_k, AE_k] : noexit :=
  (from_1?users : user_t; Anim_3[AE_k](...)) >> V_users[...]
endproc
process V_load[from_k, AE_k] : noexit :=
  (from_1?users : user_t; Anim_4[AE_k](...)) >> V_load[...]
endproc
```

以上のプロセスを用いて，全体の可視化用シナリオを以下のように構成できる．

```
process scenario[AE_0, AE_1, ..., AE_n, from_1, ...from_n]
  : noexit :=
  ( 各キャスト (静止図形も含む) の定義および表示 )
>>
  (V_comm[from_1, AE_0](...)|||...|||V_comm[from_n, AE_0]
  |||V_users[from_1, AE_1](...)|||...|||V_users[from_n, AE_n]
  |||V_load[from_1, AE_1](...)|||...|||V_load[from_n, AE_n]
)
endproc
```

n 台の計算機上でそれぞれclientを，1台の計算機上で可視化されたサーバプロセス，

server|[$from_1, ..., from_n$]|scenario

を実行することで，ネットワーク上の複数の計算機の使用状況が図的にかつリアルタイムで表示される．計算機が4台の場合の実行例を**図3**に示す．図では，2.3節で説明したhide文を用いて，2つの計算機の情報

のみ表示している．

6. 可視化システムの性能評価

本稿では，プロトコルの動的実行状況をその実行効率をあまり低下させることなく視覚化することを1つの目的としている．これを達成するには，LOTOSコンパイラが実行効率の良い目的コードを生成できること，アニメーションを高速に表示できることなどが望まれる．また，本可視化法では，別に作成した可視化用シナリオを元仕様と同期させることで可視化を行っているので，同期処理のためのオーバーヘッドが問題になりうる．

作成した可視化システムの有効性を確認するため，(i) LOTOSコンパイラ[7]が生成する目的コードの実行効率，(ii) アニメーション表示機構の実行効率，(iii) 同期処理のためのオーバーヘッドについて，実験・評価を行った．

まず，コンパイラが生成する目的コードについて，並列，選択，同期実行に関する基本性能を調べた．単純なI/Oイベントを順次実行するランタイムプロセスを100個並列に生成実行するLOTOS仕様から得られた目的コードでは毎秒1500個以上のイベントが実行された[7]．いくつかのイベント間の選択実行を行いそれを繰り返すLOTOS仕様から得られた目的コードにおいても，毎秒1000個以上のイベントが実行された[7]．同様に，ランタイムプロセスを100個並列に生成するプロセスPについて，2つのPを並行に実行し，その間ですべてのイベントについて同期実行させたところ（$P||P$ を実行），毎秒200個以上のイベントを実行できることが確認された[7]．

次に，いくつかのキャストが並行に移動するように記述されたLOTOS仕様から目的コードを生成・実行し，作成したアニメーション表示機構の実行効率を調べた．ステージを更新する時間間隔を1/10秒（10フレーム/秒）に固定した場合，3.2節で作成したリアルタイムアニメーション表示機構を用いて，100個以上のキャストが並行に移動するのを，指定したフレームレートで滑らかに表示できることを確認した(GATEWAY2000 P5-90, BSD/OS2.0.1)．

最後に，元仕様と可視化用シナリオの間の同期処理のオーバーヘッドを知るため，4章で記述した「哲学者の食事問題」について，元仕様のみを実行した場合，および元仕様と可視化用シナリオの組を実行した場合の実行時間の比を測定した．すべてのイベントについて，元仕様とシナリオの間で同期が行われる本例題においても，このオーバーヘッドのために要する時間は，各キャストについてのアニメーションの処理時間を考慮しない場合，元仕様Philosophersのみを実行したときの1/5程度の少なさであった．

以上の結果から，本可視化システムを用いて，プロトコルや並行システムの実行効率をそれほど落とすことなくその動作状況を視覚化できることが分かった．

7. おわりに

本稿では，LOTOSのマルチランデブ機構を用いたLOTOS仕様の可視化方法を提案した．

本手法の主な特徴は，(1) 可視化のためのシナリオをLOTOSで記述するため，与えられたLOTOS仕様の内容に応じた可視化が可能になっていること，(2) 可視化用シナリオを元の仕様を変更せず独立に記述できること，(3) 並行システムの動作をリアルタイムに可視化できること，(4) 可視化用シナリオから元仕様の実行制御が行えることなどである．

本可視化法は，コンパイラを用いているため，可視化された仕様を高速に実行することができ，4，5節で述べたような並行システムの可視化やプロトコルをネットワーク上で実際に運用しながら，その動作状況を画面上に表示することも可能となっている．

なお，本可視化法では，同期のための情報として元仕様における各イベントのゲート名と入出力値を用いている．同じ入出力値，ゲート名を持つイベントが元仕様の複数個所に含まれているときに，それらの位置（属しているプロセスなど）に応じて異なるアニメーションを表示したい場合には，それらが区別できるよう元仕様を若干変更する必要がある．本可視化法においては，各アニメーションは開始時のみ，元仕様のイベントの実行と同期する．アニメーションの終了時を元仕様のイベントの実行終了時と同期させるためには，そのための変更が必要である．また，本可視化法で元仕様中の内部イベントを可視化したい場合にも，その内部イベントが生起したことを可視化用シナリオで知ることができるよう元仕様を変更する．

現在，可視化用シナリオをもっと簡単に作成できるようにするためのツールを設計，開発中である．

参考文献

1) 安部広多，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，情報処理学会論文誌，Vol.36, No.2, pp.296–303 (1995).
2) Amer, P.D. and New, D.: Protocol Visualization in Estelle, *Computer Networks and ISDN Systems 25*, pp.741–760 (1993).
3) ISO: Information Processing System – Open

Systems Interconnection-LOTOS-A Formal Description Technique based on the Temporal Ordering of Observational Behaviour, *IS 8807* (1989).
4) Sarashina, K., Ando, T., Ohta, M., Tokita, Y. and Takahashi, K.: An Integrated Specification Support System for Communication Software Design Based on Stepwise Refinement and Graphical Representation, *Proc. 6th Formal Description Techniques (FORTE'93)*, pp.205-218 (1994).
5) 城島貴弘，松浦敏雄，谷口健一：対話型アニメーション表示支援機構の実現とその評価，情報処理学会コンピュータシステムシンポジウム論文集，Vol.95, No.7 (1995).
6) Turner, K.J. and McClenaghan, A.: Visual Animation of LOTOS using SOLVE, *Proc. 7th Formal Description Techniques (FORTE'94)*, pp.283-285 (1994).
7) Yasumoto, K., Higashino, T., Abe, K., Matsuura, T. and Taniguchi, K.: A LOTOS Compiler Generating Multi-threaded Object Codes, *Proc. 8th Formal Description Techniques (FORTE'95)*, pp.271-286 (1995).

(平成7年9月29日受付)
(平成8年3月12日採録)

安本 慶一（正会員）

平成3年大阪大学基礎工学部情報工学科卒業．平成5年同大学院博士前期課程修了．平成7年同大学院博士後期課程退学後，同年滋賀大経済助手．通信プロトコルや分散システムの形式仕様記述，言語処理系の実現に関する研究に従事．

東野 輝夫（正会員）

昭和31年生．昭和54年大阪大学基礎工学部情報工学科卒業．昭和59年同大学院博士課程修了．工学博士．同年同大助手．現在，同大基礎工学部情報工学科助教授．平成2年，6年モントリオール大学客員研究員．分散システム，通信プロトコル等の研究に従事．電子情報通信学会，IEEE_CS，ACM各会員．

松浦 敏雄（正会員）

昭和50年大阪大学基礎工学部情報工学科卒業．昭和54年同大学院基礎工学研究科（情報工専攻）博士後期課程退学後，同大基礎工学部情報工学科助手．平成4年同大情報処理教育センター助教授，平成7年大阪市立大学教授．工学博士．ユーザインタフェース，マルチメディア，情報教育等に興味を持つ．ACM，IEEE，電子情報通信学会等会員．

谷口 健一（正会員）

昭和40年大阪大学工学部電子工学科卒業．昭和45年同大学院博士課程修了．同年同大基礎工学部助手，現在，同情報工学科教授．工学博士．この間，計算理論，ソフトウェアやハードウェアの仕様記述・実現・検証の代数的手法および支援システム，関数型言語の処理系，分散システムや通信プロトコルの設計・検証法などに関する研究に従事．