階層グラフ書換え言語LMNtal処理系における非同期実行の実現

Asynchronous execution of LMNtal,
a language based on the rewriting of hierarchical graphs

水野 謙[†] 加藤 紀夫[*] 原 耕司[†] 上田 和紀[‡]
Ken MIZUNO Norio KATO Koji HARA Kazunori UEDA

[†] 早稲田大学大学院理工学研究科情報・ネットワーク専攻
[‡] 早稲田大学理工学部コンピュータ・ネットワーク工学科
[*] 産業技術総合研究所 システム検証研究センター
[†‡]Dept. of Information and Computer Science, Waseda University
[*] AIST Research Center for Verification and Semantics
{mizuno, n-kato, hara, ueda}@ueda.info.waseda.ac.jp

LMNtal は，階層的グラフの書換えに基づく並行言語である．本研究では，複数の膜にある書換え規則を非同期に実行できる処理系を設計・開発した．この処理系では，非同期実行を実現するために膜単位のロック機構を提供している．また，非同期に変更されたプロセスは書換え規則の適用検査を再度行う必要がある事を考慮した，完全かつ効率的な書換え処理を実現している．この非同期実行の仕組みを用いて，他言語インターフェースにおけるスレッド処理や分散拡張が容易に実現できた．

1 はじめに

LMNtal[8] は階層グラフの書換えに基づく並行言語モデルであり，プロセス・データ・メッセージの統一的な扱いを特徴としている．これらの実体を構造化する手段としてリンク（チャネル）と膜とが用意されており，リンクによって接続関係（グラフ構造）を，膜によって多重集合とその階層関係を表現することができる．計算はグラフ書換え規則の適用によって進行するが，膜は書換え規則を局所化する役割も担っている．計算の進行にしたがって，グラフ構造と階層構造の両方を動的に再構成することができるので，高い表現能力をもっている．

膜による階層構造を備えた関連計算モデルとしては Chemical Abstract Machine[2]，Mobile Ambients[3]，P-Systems[7]，Bigraphical Reactive System (BRS) [5], Seal Calculus [13], Kell Calculus [14] などがあるが，LMNtal は簡潔な計算モデルを提供すると同時に，広い適用範囲をもつ実用プログラミング言語の基盤となることを重要な目標としている．このため，研究の比較的初期から実装のプロジェクトを立ち上げ，処理方式の基本の確立を行う [9] [10] とともに，実用言語とするための諸機能の設計と実装も進めてきた [4].

現在稼働･公開している処理系 [1] は，Java と Ruby による二つの試作処理系の設計･実装の経験を踏まえて，Java を実装言語として作成したものである．その開発においてもっとも困難であった点は，並列分散実行のための拡張がスムーズに行えるような基本処理方式を確立することであった．LMNtal の実装は，逐次実装に限定すればある程度単純化できるが，リンクと膜との相互作用を正しく実装することは自明ではない．さらに，複数の処理実体が階層グラフ書換えを並行・分散・非同期的に行う技術を確立することは，LMNtal の応用分野拡大にとって非常に重要であると考えた．

階層グラフ書換えの非同期処理方式が確立すれば，LMNtal をグラフ書換えに基づく分散処理記述言語として利用できるようになる（その最初の試みは [6] 参照）ばかりでなく，上掲の各種計算モデルを LMNtal に埋め込むことによってそれらの分散処理系も実現される．また他言語インタフェースを用いてユーザやシステム開発者が Java スレッドを定義して，そこから LMNtal プロセスを安全に操作することも可能になる．

非同期実行の方法としては，膜を処理単位とする方法や，一つの膜の中のグラフの異なる部分を非同

期に書換える方法などが考えられるが，現処理系は膜を処理単位とする非同期実行をサポートしている．膜は階層構造を形成できるので，あるサブグラフは異なる膜に属する書換え規則によって書換えられるかもしれない．そこで非同期実行を正しく制御して，デッドロック発生させることなくルール適用検査の完全性を保証する方法が必要となる．

本論文では，LMNtal 処理系の全体概要を紹介したのち，我々が設計実装した非同期実行方式について，設計目標，方式の詳細，および実現した方式が満たす性質などについて論じる．

2 グラフ書換え言語 LMNtal

2.1 基本構文

LMNtal の構文は図 1 の通りである．P は LMNtal プログラムが扱う**プロセス**である．T はプロセスの書換え規則の表現に用いる**プロセステンプレート**であり，局所文脈（特定のセルの内部での文脈）を扱う機能をもつ．X_i は**リンク**，p は**名前**を表す．具体構文ではリンクは大文字から始まる識別子，名前はリンクと区別できる識別子で表記する．名前 = は，LMNtal 唯一の予約名である．

P ::=	$\mathbf{0}$	(空 / null)
	$p(X_1,\ldots,X_m)$	($m \geq 0$) (アトム / atom)
	P, P	(分子 / molecule)
	$\{P\}$	(セル / cell)
	T :- T	(ルール / rule)
T ::=	$\mathbf{0}$	(空)
	$p(X_1,\ldots,X_m)$	($m \geq 0$) (アトム)
	T, T	(分子)
	$\{T\}$	(セル)
	T :- T	(ルール)
	@p	(ルール文脈)
	\$$p$[$X_1,\ldots,X_m$ | A]	(プロセス文脈)
	$p(*X_1,\ldots,*X_m)$	($m > 0$) (アトム集団 / aggregate)
A ::=	[]	(空) †
	$*X$	(リンク束 / bundle)

図 1: LMNtal の構文

LMNtal プロセスは**プロセスのルール外には，同じリンクが 2 回を越えて出現してはならない**という**リンク条件**を満たさなければならない．

プロセス P のルール外に 1 回だけ出現するリンクを P の**自由リンク**と呼び，P に出現するそれ以外のリンクを P の**局所リンク**と呼ぶ．

直感的には，$\mathbf{0}$ は中身のないプロセス，$p(X_1,\ldots,X_m)$ は m 本のリンクをもつアトム，P, P はプロセスの並列合成，$\{P\}$ は**膜** { } によってグループ化されたプロセス，ルール T :- T は P のための書換え規則である．アトム X = Y は，リンク X の一端と Y の一端とを接続する機能をもつ．

リンクは，2 つのアトムを一対一で接続する．リンク名は，接続先を識別するためのもので，文字列自体に意味はない．そのため，リンク名の 2 つの出現を同時に別の新しい名前に置き換える事ができる．

ルールを膜に入れることができるので，膜は計算の局所化のために利用できる．ルールは，そのルールが所属する膜やその子孫膜の内容を書換えることができるが，親膜の内容を書換えることはできない．

ルール文脈は膜の中のすべてのルールの多重集合とマッチし，プロセス文脈は膜の中のルール以外のプロセスのうち，明示的に指定されていない全体とマッチする．個々のルール中のリンクや文脈の出現はいくつかの構文条件を満たさなければならない [8]．

プロセス文脈の引数は，自由リンクの出現に関する制約条件を指定するものである．直感的には，ルール左辺のプロセス文脈 \$$p$[$X_1,\ldots,X_m$ | A] の引数 $X_1,\ldots,X_m$ は，その文脈が持っていなければならない自由リンクを指定している．剰余引数 A が $*V$ の場合，$*V$ はその文脈が持つ $X_1,\ldots,X_m$ 以外の 0 本以上の自由リンクの束を表し，[] の場合は $X_1,\ldots,X_m$ 以外に自由リンクがないことを表す．

ルール左辺の膜の後に “/” と記述すると，それ以上簡約不能な膜にしかマッチしなくなる．これは，子孫膜における計算終了の検出に利用できる．

膜の階層構造に関する議論をするときには，対象となるプロセス全体を含む仮想的な膜を考え，その膜を**世界的ルート膜**と呼ぶ．

2.2 ガード

現在公開している処理系では**ガード**を含むルールをサポートしている．ガードを含むルールは以下のように表される．

左辺 :- ガード | 右辺

ガードには，左辺にマッチしたプロセスが満たすべき条件を記述することができる．具体的には，次の 2 つのルールは同一のものとして扱われる．

```
{$p} :- $p=(a,$q) | $q
{a,$q} :- a,$q
```

これは，検査に利用するが操作はしないプロセスを記述するために利用できる．

また，**型付きプロセス文脈**を用いて，型制約を記述することもできる．これについては [11] を参照して欲しい．

2.3 省略構文

グラフ構造が簡潔に記述できるように，いくつかの省略構文がサポートされている．

- アトム a の n 番目のリンクとして最終リンクを省略したアトム b を書くと，a の n 番目のリンクと b の最終リンクが繋がっているものとみなす．例えば `a(b(c))` は `a(A), b(C, A), c(C)` と等しい．
- Prolog リスト表記にならい，`[a, b|c]` と書くと `'.'(a, '.'(b, c))` の事であるとみなす．`'.'` は 3 価のアトムであり，リスト構造を作るためのドット対に対応する[1] ．

3 LMNtal 処理系

3.1 処理系概要

現在公開されている LMNtal 処理系は約 27,500 行の Java コードから成り，Eclipse と CVS を用いてチーム開発されている．

処理系は，コンパイラと，コンパイルしたプログラムを実行するときに利用する実行時処理系とからなる．

実行時処理系では，アトム，リンク，膜，および**ルールセット**はそれぞれオブジェクトとして表現されている．なお，ルールセットとは，膜の階層構造の同じ場所に所属する 1 つ以上のルールをまとめてコンパイルしたものである．ルールセットは，これらのオブジェクトを検査・操作する命令列として表現されている．

コンパイラは，LMNtal プログラムをルールセットにコンパイルし，ルールセット毎に 1 つの Java クラスを生成する．ルール以外の部分については，初期状態を生成するルール（これを**初期化ルール**と呼ぶ）が存在すると考え，そのルールをコンパイルする．初期化ルールは，実行開始時に 1 回だけ適用される，左辺が空の特別なルールである．

コンパイルされたプログラムを実行すると，ランタイムはまず初期化ルールを適用して初期状態を生成する．その後，ルールを順次選択して適用検査を行う．ルールの選択順序や，検査する LMNtal プロセスの選択順序の制御はランタイムが行っている．適用できるルールがなくなると，ランタイムは実行を終了する．

処理系のパッケージ構成は以下のようになっている．

- compile (6,186 行)
  意味解析・中間コード生成のためのクラス
- compile.parser (5,627 行)
  構文解析時のデータ構造
- compile.structure (631 行)
  意味解析・中間コード生成時のデータ構造
- runtime (10,840 行)
  ランタイムシステムおよびデータ構造，コマンドラインインターフェース
- java_cup (1,513 行)
  CUP，JFlex 付属のライブラリ
- daemon (1,873 行)
  分散処理系の通信部分
- util (831 行)
  各種ユーティリティクラス

3.2 膜をまたがるリンクの実装

次のようなプログラムについて考える．

```
(a(X) :- b(X)),
a(A), z(A)
```

LMNtal ではリンクが双方向なため，初期状態では a と z は，互いに自分の接続先へのポインタを保持している．したがって，このプログラムでは，ルールを適用する際，ルールに出てきていないアトム z の情報も書換える必要がある．すると，次のようなプログラムでは問題が発生する．

[1] LMNtal ではリンクは全て双方向なので，ドット対には 3 本の腕が必要になる．

```
{ (a(X) :- b(X)), a(A) },
{ (a(X) :- b(X)), a(A) }
```

この例では，リンク先のアトムは兄弟膜に所属している．したがって，このままでは一方のルールを適用する際には両方の膜をロックする必要が出てくる．これでは，ルール適用に直接関係のない膜をロックすることになってしまう上に，デッドロックが発生する原因となる．

そこで，本処理系では自由リンク管理アトムという 2 種類の特殊なアトムを用いることでこの問題を解決している [9]．自由リンク管理アトムは，リンクが膜をまたがっている箇所に，膜の内側と外側に挿入される，意味的には = と等価のアトムである．内側のものを in，外側のものを out と表記すると，上の例は内部的には次のように表現されている．

```
{ (a(X) :- b(X)),
  a(A), in(A1,A) },
out(A1,A2), out(A3,A2),
{ (a(X) :- b(X)),
  a(A4), in(A3,A4) }
```

これにより，また，通常のアトムのリンク先は全て同じ膜内にあることが保証されるので，上述の問題は解消できる．

3.3 モジュールシステム

実際にさまざまなアプリケーションを記述するために必要となるモジュールの宣言・読み込み機能がサポートされている [12]．

モジュール m を定義するには，(i) モジュール m に含めたいルール群および (ii) モジュール名を表す module("m") というプロセス からなるセルを定義し [2]，それを m.lmn ファイルに保存してライブラリパスが通ったディレクトリに置けばよい[3]．

モジュール m を使用するときは任意の膜の中に m.use と書けばモジュール m に含まれるルールがその膜に読み込まれる[4]．

例として bool モジュールの一部を紹介する．

[2] この記法はマクロ的なものなので，実行時にアトム名を module に変えるなどして新たなモジュールを定義したりモジュール名を変更したりすることはできない

[3] デフォルトでは ./lmntal_lib だが実行時にコマンドライン引数 -I により追加することもできる

[4] 実際には m. で始まるアトムがあるとモジュール m が読み込まれる．

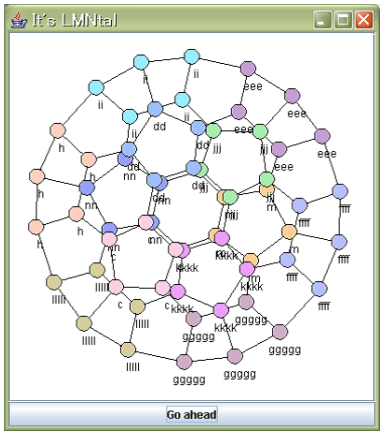

図 2: 可視化器

```
{ module(bool).
  H=not(true) :- H=false.
  H=not(false) :- H=true. }
```

このモジュールは真偽値演算に関するルールからなり，bool.use， r=not(true) と書くと bool モジュールが読み込まれ，r=not(true) が反応して r=false に置き換わる．

3.4 他言語インターフェース

アトムの一種であるインラインアトムの中に Java コードが書けるようになっていて，そのようなアトムが生成された直後にはそこに書かれたコードが実行される．この仕組みにより，入出力，ソケット通信，コマンドライン引数へのアクセスなど OS とのやりとりを担うライブラリが実現されている．また，インラインコードは誰でも書くことができるので LMNtal をグルー言語として使うこともできる．

3.5 可視化

内部状態を表すグラフ構造に力学モデルを適用しアトムの位置を反復計算することで処理系内部のデータ構造を可視化する機能がある（図 2）．

LMNtal で扱うデータは階層的なグラフ構造であるのでこれを可視化すれば反応経過が一目瞭然である．リンクをバネとみなし，さらにあるアトムが持つリンクの角度が等間隔になるように力が加わるというモデルを採用している．

処理系に -g オプションをつけて起動すると可視化

機能が有効になる．ルールが 1 回適用されるたびに実行が中断されグラフ構造が表示される．`Go ahead` ボタンを押すと実行が再開する．

3.6 シャッフルモード

LMNtal では，書換え可能なプロセスや適用可能なルールが複数存在する場合の適用順序について規定されておらず，処理系が自由な順序で適用することができる．本処理系では，標準の実行モードでは，LMNtal のソースコード中の順序に基づいて，選択するプロセスの順序が決まっている．

しかし，これではプロセスの選択が規則的になってしまうため，プログラムによっては望ましい動作とならない場合がある．そこで，本処理系ではシャッフルモードを用意している．シャッフルモードを利用すると，膜内のルール選択や，ルールにマッチするアトムや膜の選択をランダムにすることができる．

4 非同期実行の目的

本論文の目的は，前節に紹介した LMNtal 処理系に導入した非同期実行の仕組みについて論じることである．この非同期実行機能は，膜を処理単位とした複数スレッドによる非同期実行を実現するもので，これにより以下の機能が実現できる．

- 並列実行
  共有メモリ型の並列計算機において，膜単位の並列処理が実現できる．これは，タスク並列処理に相当する．
- 分散処理
  分散処理系を実装する場合，LMNtal のプロセスが他のマシンによって非同期に書換えられる事を考慮する必要がある．逐次処理系があらかじめ非同期実行をサポートしていれば，通信部分を実装するだけで分散処理系が実現できる．
- 他言語インターフェース
  本研究によって，ユーザーがインラインコードを用いて作成した Java スレッドが安全に LMNtal プロセスを書換える事ができるようになる．

5 非同期実行の設計方針

5.1 必要な性質

LMNtal 処理系に非同期実行機能を追加する際に必要な性質として，次のようなものがある．

- デッドロック防止
  複数のスレッドが非同期に LMNtal プロセスを操作するので，排他制御を行う必要がある．その際，処理系がデッドロックを起こさないようにする必要がある．
- 完全なルール適用検査
  LMNtal では，適用できるルールがなくなった時，計算終了と見なす．そのため，処理系は適用できるルールがなくなった事を検知する必要がある．
- 効率的な実行
  非同期実行の目的の一つに，共有メモリ型マシンにおける並列実行がある．並列効果の高めるために，必要以上に広い範囲をロックしないようにする必要がある．

このうち，デッドロックと完全なルール適用検査とは相互に関係しているため，両方を同時に満たす方法は自明ではない．この事が，本研究の難しさの原因となっている．

5.2 非同期実行の単位

本研究では，非同期実行の単位として膜を利用することにした．具体的には，異なる膜に所属するルールは同時に実行することができるが，同一の膜に所属するルールを同時に実行することはできない．

本研究の目的は，LMNtal 処理系でタスク並列処理を実現することである．LMNtal プログラムでは膜に計算の局所化の機能があるので，膜をタスクの単位として利用するのが自然である．そのため，膜を単位とすることにした．

5.3 ロックの単位

本研究では，ロックの単位として膜を利用することにした．従って，複数のスレッドが同時に同一の膜を操作することはできない．なお，膜の操作とは，その膜に所属するアトムの作成や除去やリンクのつなぎ替え，および子膜の作成や除去である．子膜の操作は含まない．また，膜の内容を他の膜に移動する場合，その子孫膜のロックを取得する必要はない．例として，次のようなプログラムを考える．

```
{                                  %M1
  { a, (a :- calc_something) }, %M2
  doing_something_here_too
},  ({$q, go_up} :- $q)
```

外側の膜を M1 とし，M1 の子膜を M2 とする．この例において，M1 の外側にあるルールによってプロセスの移動を行う場合，M1 はロックする必要があるが，M2 はロックしなくて良い．したがって，このプロセス移動ルールと M2 内にあるルールとは同時に実行できる．

非同期実行の単位として膜を利用しても，ロックの単位としてはより細かい単位を利用することも考えられる．その場合は，上の例において，M1 の外にあるプロセス移動ルールは M1 内にあるルールとも同時に実行することができるようになる．しかし，非同期実行の単位として膜を利用している以上，このような膜単位より細かい並列性は少ないと考えられる．そのため，ロック機構の複雑化によるオーバーヘッドの方が大きいと考え，膜単位のロックを採用した．

6 非同期実行を実現する構成要素の設計と実現

6.1 タスクとルールスレッド

本処理系では，膜単位の非同期実行を実現している．この実行主体を**タスク**と呼ぶ．タスクはいくつかの膜を管理し，管理する膜にあるルールの適用処理を行う．各タスクは異なるスレッドで実行される．このスレッドを，**ルールスレッド**という．

プログラマは，複数タスクの利用を指定するために，**ルート膜**を指定する．膜の後に `@"localhost"` と記述すると，その膜がルート膜になる．ルート膜を生成すると新しいタスクが生成され，ルート膜とその子孫膜（他のルート膜とその子孫膜をのぞく）はそのタスクによって管理されるようになる．

この記法は，分散処理の記法に由来している．`@` の後にマシン名や IP アドレスを記述すると，その膜は指定されたマシン上で生成・実行される．ホスト名部分に `localhost` と指定することで，同一マシン上で新たなタスクが生成され，そのタスクにおいて膜が生成・実行される．

例として，次のようなプログラムを考える．

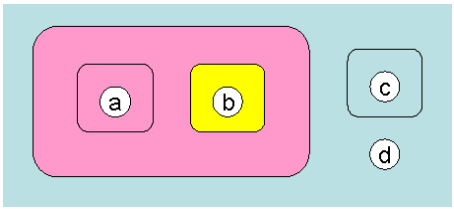

図 3: タスク階層

```
{ { a },
  { b }@"localhost"
}@"localhost",
{ c }, d
```

このプログラムには 4 つの膜があり，3 つのタスクを用いて実行される．このプログラムのグラフィカル表現を，同じタスクが管理する部分を同じ色で塗ると，図 3 のようになる．

6.2 非ルールスレッド

本処理系では，インラインコードを用いてユーザーが自由に Java プログラムを実行する事ができる．従って，処理系はインラインコードによって LMNtal プロセスが操作される事も考慮する必要がある．

インラインコードの実行は，ルールスレッド上で，ルール適用処理の一部として実行される．ルールに出現する膜はロックされており，自由に書換えることができるが，それ以外の膜を操作する必要がある場合はルールスレッドにおける規則に従ってロックする必要がある．

一方，インラインコード中でユーザーが Java スレッドを生成し，そのスレッドから LMNtal プロセスを操作することもできる．これは，インラインコードを用いて GUI アプリケーションを作成する場合などには必須の機能である．この場合は，タスクによるルール適用処理のプロセスとは独立して非同期に実行されるため，ユーザーがロック処理を行う必要がある．このようなスレッドを**非ルールスレッド**と呼び，ルールスレッドとは区別して扱う．

6.3 ロックによる排他制御

ロックに関して，次の性質が成り立つようにする必要がある．

- デッドロックしない．

- ロック要求は，短期間で受け入れられる．

これらの性質を満たすようにするために，次の規則を設けた．

1. 膜のロックを取得するスレッドは，どの膜のロックも取得していないか，またはその親膜のロックを取得していなければならない．
2. ルールスレッドが最初にロックを取得する膜は，自タスクが管理する膜でなければならない．
3. 非ルールスレッドが最初にロックを取得する膜は，ルート膜でなければならない．

1. により，複数のスレッドが互いに他方の保持するロックを取得しようとしてデッドロックする事がなくなる．

また，この 3 つの規則により，ある膜が，その膜を管理するタスクを実行しているルールスレッド以外でロックされる場合は，その膜からその膜を管理するタスクのルート膜までの膜は全て同一のスレッドによってロックされることになる．

なお，非ルールスレッドがこの規則を満たすようにすることは，プログラマの責任である．実際には，ルート膜から指定された膜までの間を親膜から順にロックするメソッドを用意し，プログラマの負担を軽減している．

また，ロック要求が短期間で受け入れられることを保証するために，次の規則を設けた．

- ルールスレッドは，**ロック取得要求**を受け取ったら速やかに全てのロックを解放しなければならない．
- 非ルールスレッドは，取得したロックを短期間で解放しなければならない．

ルールスレッドは，一般にはロックを保持したまま複数回のルール適用を行うことができる．しかし，ロック取得要求があるときには新たなルール適用処理を開始せずにロックを解放してブロックする．

非ルールスレッドは，その目的から長期間にわたってロックを保持し続ける必要はないと考えた．そのため，プログラマがロックを短期間で解放するように気をつける必要がある．

この規則から，ロック取得に失敗してブロックする場合，ロック取得要求を出すことで短期間にロックを取得できることが保証される．

6.4 タスク内の実行制御方式

6.4.1 実行膜スタック

LMNtal では，ルールを膜に入れることができる．この機能によって，計算の局所化を実現している．

ルールがグローバルではないので，本処理系ではルールの適用処理をルールが所属する膜毎に行うことにした．この処理を行うため，各タスクは**実行膜スタック**を持っている．実行膜スタックには，そのタスクが管理する膜のうち，適用可能なルールが存在する可能性があるものが積まれる．タスクは実行膜スタックの先頭の膜（これを**本膜**という）に対してルール適用検査を行い，本膜に適用できるルールが存在しない場合には実行膜スタックから除去する．

なお，本処理系では親膜にあるルールより子膜にあるルールを優先することにした．LMNtal では子孫膜の書換えを許しているため，ある膜の内容が変化すると，その先祖膜にあるルールの適用が可能になる可能性がある．そのため，親膜側のルール適用検査を先に行ってしまうと，子膜側のルールを適用した際に親膜側のルール適用をやり直す必要がでてくる．従って，子膜側のルール適用を優先的に検査することで，親膜側の適用検査のやり直しを最小限に抑えることができる．

そこで，不変条件として，親膜は子膜より底側に積まれていなければならないことにした．これにより，処理系の設計や，その性質に関する議論が容易になる．なお，不変条件の詳細については 6.4.3 節で述べる．

6.4.2 仮の実行膜スタック

ルールスレッドが自タスク管理の膜 M を操作する場合は，その膜の子孫膜が M と同じ実行膜スタックに積まれていることはない．したがって，操作した膜は単純に実行膜スタックに積むことができる．しかし，他タスク管理の膜を操作する場合，操作した膜の子孫膜は実行膜スタックに積まれているかもしれない．その場合は，操作した膜を単純に実行膜スタックに積んでしまうと，子膜が親膜より底に積まれてしまう可能性がある．

ロックに関する規則から，ルールスレッドが他タスク管理の膜 M のロックを取得している場合，M を管理するタスクのルート膜から M までの間の膜を全てロックしている必要がある．したがって，他タスク管理の膜を操作した場合は，その膜からルート膜

までの膜を，実行膜スタックの底側に置くようにすくことでこの問題を解消することができる．

そのために，各タスクは**仮の実行膜スタック**を持っている．実行膜スタックと同様，仮の実行膜スタックにはそのタスクが管理する膜のみが積まれる．仮の実行膜スタックの内容は，ルート膜のロックが解放されるときに，実行膜スタックの底に移動される．

6.4.3 実行膜スタックに関する条件

ある膜 M が実行膜スタックに積まれており，M がルート膜でない場合，M の親膜は次のいずれかの状態にある必要がある．

1. 実行膜スタックの，M より底の方に積まれている．
2. 仮の実行膜スタックに積まれている．
3. ロックされている．

また，ある膜 M が仮の実行膜スタックに積まれており，M がルート膜でない場合，M の親膜は仮の実行膜スタックの，M より底の方に積まれている必要がある．

6.4.4 実行膜スタックの操作

ある膜を実行膜スタックに積む際，親膜が実行膜スタック中にない場合は，親膜を先に積む必要がある．この一連の操作を膜の**活性化**という．具体的には，膜 M の活性化とは，以下の操作である．

- M が次のいずれかの状態にあるときは何もしない．
  - ロックされている．
  - 実行膜スタックに積まれている．
  - 仮の実行膜スタックに積まれている．
- それ以外の場合，以下の操作を行う．
  - M がルート膜の場合，M を仮の実行膜スタックに積む．
  - M がルート膜でない場合，まず親膜を再帰的に活性化し，次に M を親膜と同じスタックに積む．

これにより，6.4.3 の条件を満たすように，膜 M を実行膜スタックに追加することができる．

膜の活性化は，以下の場合に行われる．

1. ルール適用によって本膜以外の膜の内容が変化した時，その膜を活性化する．
2. ルール適用によって新たに膜を生成したとき，生成した膜を活性化する．
3. ルート膜の実行を終了した時，そのルート膜の親膜を活性化する．
4. 非ルールスレッドが膜のロックを解放する時，その膜を活性化する．

膜を活性化する際には，その膜のロックを取得していなければならない．活性化の結果仮の実行膜スタックに積まれた場合は，ルート膜のロックを解放時に実行膜スタックに移動される．

4. は，ロック取得失敗によって失敗した検査をやり直すためにある．ルールスレッドは，ルール左辺の膜のロック取得をノンブロッキングで行い，失敗した場合にはルール適用処理が失敗する．ロック解放時に膜を活性化することで，その膜の先祖膜のルール適用をやり直すことができる．

ルールスレッドが取得したロックを解放する際は，その内容を変更していなければ活性化する必要はない．なぜなら，明示的に活性化しなくても，本膜の先祖膜はいずれ活性化されるからである．

6.5 膜の静止判定

6.5.1 stable フラグ

膜の静止判定のために，膜は stable フラグを持つ．stable フラグがオンの場合，その膜とその子孫膜には適用可能なルールは存在しない．

世界的ルート膜の stable フラグがオンになった場合，適用可能なルールがなくなったことにを表すので，処理系は実行を終了する．

非同期実行を行わない場合は，stable フラグがなくても実行膜スタックに積まれているかどうかで判断できる．しかし，非同期実行を行う場合には実行膜スタックに積まれていなくても stable ではない可能性がある．

6.5.2 perpetual フラグ

LMNtal では，適用できるルールがなくなった場合，実行を終了する．しかし，非ルールスレッドが LMNtal プロセスを操作する場合は，適用できるルールがなくなっても，その後で非ルールスレッドによって書換えられるかもしれないので，実行を終了しないようにしなければならない．

そのため，膜に perpetual フラグを用意した．非ルールスレッドがある膜の内容を変更する可能性が

ある場合，その膜の perpetual フラグをあらかじめオンにしておく．これにより，この膜の stable フラグがオンになることはなくなり，処理系は終了しなくなる．

6.5.3 フラグの操作

新たな膜が生成されるときは，その膜の stable フラグはオフである．本膜に適用できるルールがなかったとき，本膜の perpetual フラグがオフで，かつ子膜の stable フラグが全てオンなら，本膜の stable フラグをオンにする．ある膜の内容（子孫膜の内容を含む）を変更した場合，その膜の stable フラグをオフにする．

ある膜 M の stable フラグがオフであるにもかかわらずその膜が実行膜スタックに積まれていないのは，以下のような場合である．

- M の子孫膜に，perpetual フラグがオンの膜がある．
- M の子孫膜に，実行膜スタックに積まれている膜（これを M2 とする）がある．

後者の場合，M が実行膜スタックに積まれていないので，M2 が M とは別のタスクに管理されている．この場合，M はいずれ活性化され，再度ルールの適用検査が行われる．

このことから，適用できるルールが存在するにもかかわらず処理系が終了したり，永久にブロックしたりしないことが保証される．

7 まとめと今後の課題

本論文では，グラフ書換え言語 LMNtal 処理系のための非同期実行方式について述べた．膜単位のロック機構と実行膜スタック，および膜の stable フラグを採用し，デッドロックが発生したり，適用できるルールが存在する状態で処理系が終了することはないことを保証した．これにより，膜を処理単位とする並列実行や，他言語インターフェースを用いたユーザー定義スレッドによる LMNtal プロセスの安全な操作が実現できた．

今後の課題として，データ並列処理や自動並列化，および優先度の実現がある．優先度のうち，異なる膜間のルール優先度に関しては，本研究の上にタスク間優先度を実装することで実現できるが，その際には正当性について再度検討する必要がある．

謝辞

本研究の一部は，科学研究費補助金（基盤 (B)(2)16300009，特定 (C)(2)13324050，特定 (B)(2) 14085205）の補助を得て行った．

参考文献

[1] LMNtal website: `http://www.ueda.info.waseda.ac.jp/lmntal/`

[2] Berry, G. and Boudol, G., The Chemical Abstract Machine. In *Proc. POPL'90*, ACM, pp. 81–94.

[3] Cardelli, L. and Gordon, A. D. : Mobile Ambients, in *Foundations of Software Science and Computational Structures*, Nivat, M. (ed.), LNCS 1378, Springer-Verlag, 1998, pp. 140–155.

[4] 原耕司, 水野謙, 矢島伸吾, 永田貴彦, 中島求, 加藤紀夫, 上田和紀: LMNtal 処理系および他言語インタフェースの設計と実装, 情報処理学会第 50 回プログラミング研究会 (SWoPP2004), 2004.

[5] Milner, R., Bigraphical Reactive Systems. In *Proc. CONCUR 2001*, LNCS 2154, Springer, 2001, pp. 16–35.

[6] 中島求，加藤紀夫，水野謙，上田和紀：LMNtal を用いた分散処理の実現．第 8 回プログラミングおよび応用のシステムに関するワークショップ（SPA2005), 2005.

[7] Păun, Gh., Computing with Membranes. *J. Comput. Syst. Sci.*, Vol. 61, No. 1 (2000), pp. 108–143.

[8] Ueda, K. and Kato, N., LMNtal: A Language Model with Links and Membranes. In *Proc. Fifth Int. Workshop on Membrane Computing (WMC 2004)*, LNCS 3365, Springer, 2005, pp. 110-125.

[9] 矢島伸吾，永田貴彦，加藤紀夫，上田和紀：LMNtal プロトタイプ処理系の設計と実装．日本ソフトウェア科学会第 20 回大会論文集，2003, pp.21-25.

[10] 水野謙，永田貴彦，加藤紀夫，上田和紀：LMNtal ルールコンパイラにおける内部命令の設計．情報処理学会第 66 回全国大会，5G-2, 2004.

[11] 工藤晋太郎，加藤紀夫，上田和紀：LMNtal 処理系におけるグラフ構造の操作機能の設計と実装．情報科学技術レターズ, Vol.4, 2005 (掲載予定).

[12] 原耕司：LMNtal 処理系のモジュール機能と他言語接続機能の設計と実装．早稲田大学卒業論文，2003.

[13] Castagna, G., Vitek, J. and Zappa Nardelli, F., The Seal Calculus. To appear in *Information and Computation*, 2005.

[14] Schmitt, A. and Stefani, J.-B., The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Proc. Int. Workshop on Global Computing.*, LNCS 3267, Springer, 2005, pp. 146–178.