2017 年度 修士論文

グラフ書き換え言語 LMNtal による 容易に拡張可能なモデル検査器の実装

提出日： 2018 年 1 月 26 日
指導： 上田 和紀 教授

早稲田大学 基幹理工学研究科
情報理工・情報通信専攻

学籍番号：5116F063-8
恒川 雄太郎

概要

恒川 雄太郎

LMNtal とはグラフ書き換えに基づくモデリング言語であり，そのモデル検査器 SLIM は状態空間探索および LTL モデル検査を備えている．これまで SLIM に対して様々な拡張や変種が開発されてきたが，それらはすべて C 言語で実装された SLIM のソースコードを改変・修正することによって成されてきた．もしモデル検査器がモデリング言語自身で実装されれば，モデル検査器の基礎部分を変えることなく容易に様々な拡張が可能になるはずである．このようなメタプログラミングの手法は Lisp や Prolog では伝統的に行われてきた手法である．

本論文では容易に拡張可能なモデル検査器の実装のためのフレームワークについて論じる．まず，モデリング言語 LMNtal における第一級の書き換え規則を定義する．次にプログラムの状態を LMNtal 自身から操作するための API を設計する．これらの機能によってプログラマは LMNtal プログラムの状態遷移グラフを第一級のオブジェクトとして LMNtal プログラム内で扱えるようになり，SLIM を変更することなく様々な変種を実装することが可能になる．フレームワークを用いたモデル検査器の実装例として LTL モデル検査器，CTL モデル検査器，TCTL モデル検査器などを紹介する．さらに，実装した CTL モデル検査器に対して公平性を扱うための拡張を施した例を紹介し，フレームワークを用いて実装したモデル検査器に対する拡張がどれほど容易かを示す．また，フレームワークを用いて実装されたモデル検査器のオーバーヘッドは SLIM に比べて 10 倍以下であった．これらの結果は本研究で設計および実装したフレームワークはシステムの状態空間を探索するメタインタプリタを容易に実装するのに十分な能力を持っていることを示唆するものである．

Abstract

Yutaro Tsunekawa

LMNtal is a modeling language based on hierarchical graph rewriting, and its implementation SLIM features state space search and an LTL model checker. Several variations and extensions of the SLIM have been developed, and all of them achieve their functionalities by modifying SLIM written in C. If a model checker is implemented in the modeling language itself, it should be easy to develop prototypes of various model checkers without changing the base implementation of the model checker. This approach is called metaprogramming which has been taken extensively in Lisp and Prolog communities.

In this paper, we design a framework for implementing extendable model checkers. First, we define first-class rewrite rules to extend a modeling language. Second, we design an API to operate on the states of programs. These features enable programmers to handle state transition graphs as first-class objects and implement diverse variants of a model checker without changing SLIM. We demonstrate it by implementing an LTL model checker and its variant and a CTL model checker. Furthermore, we show how easy it is to extend these model checkers in our framework by extending the CTL model checker to handle fairness constraints. The overhead of metainterpretation is around an order of magnitude or less. All these results demonstrate the viability of the resulting framework based on meta-interpreters that handle explicit state space in a flexible manner.

目次

図目次

表目次

第 1 章

はじめに

1.1 研究の背景と目的

グラフ書き換え系とはグラフとグラフの書き換え規則から成る．グラフはリスト，木，多重集合を部分集合として含む高い表現視力を持つデータ構造であるため，グラフ書き換え系によって様々な状態遷移系をモデリングすることが可能である．例えば，動的再構成可能なネットワークで構成された並行システムをグラフ書き換え系によってモデリングすることができる．いくつかのグラフ書き換え系はモデル検査器を備えており，それらは書き換えの非決定性によって生まれる状態遷移グラフを探索する．例えば，Groove [14] や SLIM [11][17] などが在り，後者は LMNtal [16] をモデリング言語として持つ．

状態遷移システムと仕様記述に用いる様相論理に依って様々なモデル検査器が存在する．例えば，離散的遷移システムに対する LTL モデル検査，リアルタイムシステムに対する TCTL モデル検査，マルコフ決定過程に対する PCTL モデル検査などが在る．確かにこれまで様々な拡張や変種が SLIM に対して成されてきた．例えば，リアルタイムモデル検査を SLIM に導入しようとした研究が在った．それらの拡張は約 5 万行の C 言語で実装された SLIM のソースコードを修正したり拡張したりすることによって達成されてきた．しかしモデル検査器のように巨大で複雑なモデル検査器のソースコードを変更することは容易でないため，新しいモデル検査器の迅速なプロトタイピングは難しい．

Lisp や Prolog などの記号処理のためのプログラミング言語においては伝統的に自身の実装に手を加えることなく言語の構文や意味を変更するメタプログラミングの手法が知られていた [3]．そのようなメタプログラミングの手法はその言語のメタインタプリタを実装し，拡張・修正することによって達成される．メタインタプリタを用いたメタプログラミングによって実験的なプログラミング言語のプロトタイプを実装を行ったり，DSL を

作ったりすることが容易に可能に成る．実際にプログラミング言語 Erlang の初期の実装は Prolog のメタインタプリタを拡張することによって得られた [1].

もしプログラムが言語の第一級のデータ構造で表現され，プログラムからプログラム実行のための処理系の機能を使うことができれば，メタインタプリタの実装および拡張によってプログラマは容易に対象言語の変種を創ることができる．しかし一般にプログラムの非決定的な状態遷移を第一級のオブジェクトとして扱うことはできないため，メタインタプリタの自然な発展としてモデル検査器を実装できるかどうかは自明ではない．

本研究の目的は LMNtal にプログラムから状態遷移を扱う能力を追加して，様々なモデル検査アルゴリズムの状態空間探索および状態空間構築の戦略をプログラムから明示的に指定できるようにすることである．さらに，状態遷移を明示的に扱うことによってプログラマは個々の状態に付帯情報を追加して，付帯情報によって異なる状態として比較できるようにすることである．このような付帯情報の追加によって例えば状態空間のヒューリスティック探索が可能になると考えられる．本研究ではまず (1) LMNtal の第一級のデータ構造としての書き換え規則を設計および実装を行い，(2) SLIM のプログラム実行機能や状態管理機能に対する API の設計および実装を行った．これらの状態を扱う能力示すために LMNtal による LMNtal メタインタプリタのプロトタイピングを行った．また，実装した LMNtal メタインタプリタの自然な発展として LMNtal のためのモデル検査器をいくつか実装し，さらにそれらの拡張も行った．これらの結果は本研究で設計および実装したフレームワークはシステムの状態空間を探索するメタインタプリタを容易に実装するのに十分な能力を持っていることを示唆するものである．

1.2　本論文の構成

本論文の構成は以下の様になっている．第 2 章では，グラフ書き換え言語 LMNtal について説明する．LMNtal プログラムはグラフとグラフの書き換え規則によって構成されるが，グラフや書き換え規則の構成要素について詳しく解説を行う．また，LMNtal のためのモデル検査器 SLIM についても紹介する．第 3 章では，本研究で設計および実装を行ったプログラムの状態空間を構築するメタインタプリタ実装のためのフレームワークについて解説する．まず容易に拡張可能なメタインタプリタについて触れ，第一級書き換え規則と SLIM の内部機能への API の設計と実装について述べる．第 4 章では，前章で述べたフレームワークを用いて実装を行った LMNtal メタインタプリタについて述べる．LMNtal メタインタプリタのソースコードを参照しながら詳しく実装を解説する．また，SLIM との性能比較実験の結果とメタインタプリタの拡張例も紹介する．第 5 章では，前

章で紹介した LMNtal メタインタプリタを自然に発展させて実装した様々なモデル検査器を紹介する．LTL，CTL，TCTL モデル検査器の実装を解説するが，LTL と CTL については性能評価実験の結果とそれぞれの拡張例についても紹介する．第 7 章では，本研究のまとめと今後の課題について述べる．なお，本論文で紹介するメタインタプリタやモデル検査器の完全なソースコードは `https://github.com/lmntal/McLMNtal` で公開している．

第 2 章

LMNtal

この章では，まず階層グラフ書き換え言語 LMNtal について解説した後，モデル検査器 SLIM についても解説する．本章は [19] を基に再構成したものである．

2.1 構成要素の概要

LMNtal プログラムは階層グラフと書き換え規則の集合である．階層グラフはアトム，リンク，膜から構成される．直感的にはアトムはグラフのノードに，リンクはグラフのエッジに対応し，膜が階層構造を表現する．LMNtal グラフの一般的なグラフ理論のグラフとの大きな違いはエッジの端点に順序が付いていることである．また，膜はグラフをグループ化して階層構造を表現する役割の他に，書き換え規則の適用範囲を限定するためにも用いられる．膜の階層構造を議論する場合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．このように考えることによって，プロセス全体を膜による階層構造とみなすことができる．書き換え規則はルールと呼ばれる．ルールは書き換え前のグラフのパターンと書き換え後のグラフのパターンを記述したものであり，書き換え前のグラフパターンにパターンマッチして書き換え後のグラフパターンに基づいてグラフを書き換える．書き換え規則の適用によってグラフ構造が変化していく様子は様々な状態遷移システムをモデリングするのに適している．

2.2 構文

LMNtal の構文規則を図 2.1 に示す．

図 2.1 中の P はプロセスと呼ばれる．p はアトム名と呼ばれる小文字から始まるアル

$$
\begin{array}{lcll}
P & ::= & 0 & \text{(空)} \\
 & | & p(X_1,\ldots,X_m) \quad (m \geq 0) & \text{(アトム)} \\
 & | & P, P & \text{(分子)} \\
 & | & \{P\} & \text{(膜)} \\
 & | & T :- T & \text{(ルール)} \\
 & & & \\
T & ::= & 0 & \text{(空)} \\
 & | & p(X_1,\ldots,X_m) \quad (m \geq 0) & \text{(アトム)} \\
 & | & T, T & \text{(分子)} \\
 & | & \{T\} & \text{(膜)} \\
 & | & @p & \text{(ルール文脈)} \\
 & | & \$p[X_1,\ldots,X_m] \quad (m \geq 0) & \text{(プロセス文脈)} \\
 & & & \\
A & ::= & [] & \text{(空)} \\
 & | & *X & \text{(リンク束)}
\end{array}
$$

図 2.1　LMNtal の構文規則

ファベットの文字列である．同様に X はリンク名と呼ばれる大文字から始まるアルファベットの文字列である．0 は空のプロセス，$p(X_1,\ldots,X_m)$ は m 価のアトムと呼ばれる．P, P の分子はプロセスの並列合成であり，$\{P\}$ は膜 {} でグループ化されたプロセス (このプロセスをセルと呼ぶ) である．$T :- T$ はプロセスの書き換え規則である．書き換え規則を構成する要素 T はプロセスとは少し異なる．

図 2.1 中の T はテンプレートと呼ばれる．テンプレートはルールを構成する構文要素である．ルール文脈は膜内のルールの集合にパターンマッチし，プロセス文脈はルールを除く膜内のプロセスの集合にパターンマッチする．プロセス文脈の引数 $p[X_1,\ldots,X_m|A]$ の内 $X_1,\ldots,X_m$ はそれぞれが膜を貫くリンクのリンク名を表しており，A は $X_1,\ldots,X_m$ 以外の 0 本以上のリンクをまとめて表現している．

LMNtal の構文には “プロセスには同じリンク名が 2 回を越えて出現してはならない” というリンク条件が在る．また，あるプロセスに 1 回だけ出現するリンク名は自由リンクと呼ばれ，それ以外のリンク名は局所リンクと呼ばれる．

$(E1)$	$0, P$	$\equiv$	P
$(E2)$	P, Q	$\equiv$	Q, P
$(E3)$	$P, (Q, R)$	$\equiv$	$(P, Q), R$
$(E4)$	P	$\equiv$	$P[Y/X]$　(ただし X は P の局所リンク名)
$(E5)$	$P \equiv P'$	$\Rightarrow$	$P, Q \equiv P', Q$
$(E6)$	$P \equiv P'$	$\Rightarrow$	$\{P\} \equiv \{P'\}$
$(E7)$	$X = X$	$\equiv$	0
$(E8)$	$X = Y$	$\equiv$	$Y = X$
$(E9)$	$X = Y, P$	$\equiv$	$P[Y/X]$　(ただし P はアトムで X は P の自由リンク名)
$(E10)$	$\{X = Y, P\}$	$\equiv$	$X = Y, \{P\}$　(ただし X と Y のうちのちょうど一方が P の自由リンク名)

図 2.2　LMNtal の構造合同規則

$(R1)$	$\dfrac{P \to P'}{P, Q \to P', Q}$
$(R2)$	$\dfrac{P \to P'}{\{P\} \to \{P'\}}$
$(R3)$	$\dfrac{Q \equiv P \quad P \to P' \quad P' \equiv Q'}{Q \to Q'}$
$(R4)$	$\{X = Y, P\} \to X = Y, \{P\}$　ただし X と Y は $\{X = Y, P\}$ の自由リンク名
$(R5)$	$X = Y, \{P\} \to \{X = Y, P\}$　ただし X と Y は P の自由リンク名
$(R6)$	$T\theta, (T :- U) \to U\theta, (T :- U)$

図 2.3　LMNtal の遷移規則

2.3　操作的意味

LMNtal の操作的意味論は図 2.2 に示す構造合同規則と図 2.3 に示す遷移規則から成る．

構造合同規則を満たす最小の同値関係 $\equiv$ をプロセスの構造合同関係と呼ぶ．$\equiv$ の関係にあるプロセスは本質的に等価なプロセスと見なし，0 ステップで変換可能である．図 2.2 中の $[Y/X]$ はリンク X をリンク Y に置き換えるリンク代入を表す．(E1)∼(E3) はカンマによって構成される分子が多重集合を成すことの特徴付けである．(E4) はリンク名の α 変換を許す構造合同規則である．(E5) は $\equiv$ をプロセスの並列構造と協調させるための規則であり，(E6) は $\equiv$ をプロセス階層構造と協調させるための規則である．(E7) は自己ループが空と等価であることを表し，(E8) は $=$ アトムの対称性を表す．(E9) と (E10) はそれぞれアトムとセルによる $=$ アトムの吸収と放出を表す．

遷移規則を満たす最小の関係 $\to$ をプロセスの遷移関係と呼ぶ．LMNtal の計算はプロ

セスが遷移規則に従って書き換わってゆくことによって進む．(R1) は並列構造の部分プロセスの遷移によって全体のプロセスが遷移することを表し，(R2) は階層構造の部分プロセスの遷移によって全体のプロセスが遷移することを表している．(R3) は構造合同関係を遷移関係に取り込むための規則である．(R4) と (R5) は = アトムの移動規則である．(R6) は書き換え規則にプロセスの書き換えを表した規則で，この規則を中心にして LMNtal の計算は進行する．同じ階層にプロセスと書き換え規則が存在し，そのプロセスと書き換え規則の左辺が同じだった場合，パターンマッチしたプロセスは書き換え規則の右辺へと書き換えられる．θ は代入を表しており，プロセス文脈，ルール文脈，アトム集団を具体的なプロセス，ルール，アトムに対応付ける．

具体例として次のような LMNtal プロセスを考える．

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

書き換え規則の範囲を明確にするために括弧を付けている．このプロセスから `(a(A):-b(A)),b(X), x(X)` への遷移の導出木を図 2.4 に示す．ただし，導出木中では `a(A):-b(A)`$=R$ としている．このように構造合同規則と遷移規則によって LMNtal の計算は進行する．

$$\dfrac{R, a(X), x(X) \equiv R, a(A), x(A) \qquad \dfrac{\mathcal{C}}{R, a(A), x(A) \to R, b(A), x(A)} \qquad R, b(A), x(A) \equiv R, b(X), x(X)}{R, a(X), x(X) \to R, b(X), x(X)}$$

$\mathcal{C}$ の導出木は以下

$$\dfrac{R, a(A) \equiv a(A), R \qquad a(A), R \to b(A), R \qquad b(A), R \equiv R, b(A)}{R, a(A) \to R, b(A)}$$

図 2.4　遷移関係の導出木

2.4　拡張構文

LMNtal には基本型 (組込みの型) が存在する．それらの型の検査や演算をガード部で行い，それらの指定に型付きプロセス文脈という拡張構文を用いる．例として図 2.5 に示す．ガード条件に指定可能な基本型は int 型 (整数)，float 型 (浮動小数点)，unary 型 (1 価アトム)，string 型 (文字列)，ground(強連結なアトムの集合) などがある．また，これら型に対する演算子としては int 型に対して `=:=,=\=,<,<=,>=,>,+,-,*,/,mod`，float 型に対して `>.,<.,>=.,=<.,==,=:=.,=\=.,+.,-.,*.,/.`, などがある．

```
a(X),$n[X] :- int($n) | b(X),$n[X]
```

図 2.5　ガード条件と型付きプロセス文脈

$Head$ `:-` `new`$(X_1), \ldots,$ `new`$(X_i) \mid Atom_1(X_1), \ldots, Atom_i(X_i)$	(生成)
$Atom(H)$ `:-` `hlink`$(H) \mid Body$	(型制約)
$Atom(H_1), Atom(H_2)$ `:-` H_1`><`H_2	(併合)

図 2.6　ハイパーリンクに関する構文

2.4.1　省略構文

LMNtal にはプログラムが簡潔に記述できるように省略構文が存在する．

膜に接続するリンクの省略構文

P をプロセス，X をリンクとして `p(...,X,...),{+X,P}`というプログラムは `p(...,{P},...)` と省略できる．p はセルを参照するアトムとみなして，参照を表すリンク X に接続されるアトムの名前として + が用いられることが標準である．

プロセス文脈の引数の省略構文

プロセス文脈のリンクが空の時は引数を省略してよい．つまり，$p[] というプロセス文脈は$p と省略してよいということである．

2.5　HyperLMNtal

ハイパーリンク [18] とは LMNtal でハイパーグラフを自然にモデリングするための言語機能である．ハイパーリンクに関する構文の一部を図 2.6 に示す．また，ハイパーリンク型制約は unary 型に包含されるため，プロセス $p(H)$ における H がハイパーリンクの場合 p には unary 型の無名アトムが接続されているとみなすことができ，そのアトムをハイパーリンクアトムと呼ぶ．それぞれの機能の意味は以下の通りである．

2.5.1　生成

1 引数の new 制約によって，引数に与えたリンクと同じ名前のリンクを新規作成されたハイパーリンクアトムを接続する．

2.5.2　型制約

引数に与えられたリンクの先がハイパーリンクアトムへと接続されているかどうかを検査する．

2.5.3　併合

異なるハイパーリンク同士を接続する．演算子><の両辺にハイパーリンクアトムが接続されたとき，そのハイパーリンクアトムがそれぞれ属する 2 つのハイパーリンクを併合する．

2.6　SLIM

SLIM は LMNtal によるモデル記述と LTL による仕様記述を入力に取り，モデルが仕様を満たしているかどうかを検証するモデル検査器である．SLIM は階層グラフ構造 (以下誤解の恐れがない場合は単に LMNtal グラフまたはグラフと呼ぶ) を状態，書き換え規則の適用を状態遷移とした状態グラフを構築する．

状態空間グラフは標準的には未展開状態のスタックを用いて構築される．遷移先状態は遷移元状態に書き換え規則を適用することによって得られる．LMNtal においては複数の書き換え規則を同じグラフに適用できる場合や 1 つの書き換え規則を異なる部分グラフに適用できる場合に書き換え結果に非決定性が生じる．したがって，SLIM は可能なすべての書き換えを試すことによって遷移元状態から遷移可能なすべての状態を計算する．

SLIM は LMNtal グラフのハッシュ値をキーとしたハッシュテーブルによって状態を管理する [11]．状態遷移グラフの各状態はグラフ構造であるから，一般的には状態遷移グラフを構築するために (ハイパー) グラフの同型性検査を行う必要がある．しかしながら，SLIM のグラフ同型性検査アルゴリズムはハッシュ値が衝突した時のみ呼び出される．

SLIM は LMNtal のモデル記述のみを入力として使用することもでき，この場合 SLIM はモデルの全状態空間を探索して状態遷移グラフを構築する．

第 3 章

メタプログラミングのためのフレームワーク

この章では，拡張性の高い，プログラムの状態遷移グラフを構築するメタインタプリタを容易にプログラミングするための提案フレームワークについて述べる．本研究が提案するフレームワークは LMNtal の第一級の書き換え規則と SLIM の内部機能への API から成る．第一級の書き換え規則によって LMNtal プログラムは LMNtal の第一級のオブジェクトとなる．また，SLIM 内部機能への API によってプログラム実行や状態の管理が可能になる．本研究の目的はフレームワークによってメタインタプリタを実装し，その自然な発展として様々なモデル検査器を容易に実装することである．

まず，容易に拡張可能なメタインタプリタとはどのようなメタインタプリタかということを議論する．次に第一級の書き換え規則と API の設計と実装について詳細に述べる．

3.1　メタインタプリタとは

3.1.1　メタプログラミング

メタプログラミングとは文字通りプログラムを扱うプログラミングを指すが，具体的には用いられる文脈に依って様々である．本論文においてはメタプログラミングとはプログラミング言語の実装 (コンパイラやインタプリタ) を変更することなく言語の構文と意味を変更するプログラミングテクニックやそのためのツールと定義する．Lisp や Prolog のような記号計算のためのプログラミング言語では伝統的にこの種のメタプログラミングの技法が用いられてきた．3.1.2 節でより詳細に議論するが，このメタプログラミングは新

```
prove(true).
prove((Goal1, Goal2)) :- prove(Goal1), prove(Goal2).
prove(Goal) :- clause(Goal, Body), prove(Body).
```

図 3.1　大ステップ意味論に基づく Prolog メタインタプリタ

たな言語のプロトタイピングや特定の領域のプログラミングに特化した言語機能の実装を容易に行うために用いられてきた技術である．

メタプログラミングの具体例として図 3.1 に大ステップ意味論に基づく最も簡素な Prolog メタインタプリタを示す．このメタインタプリタはすべての実行経路を探索することによって到達可能な状態を列挙する．よく知られているように，メタインタプリタを変更および拡張することによって (メタインタプリタによって解釈実行される) 言語の意味論を変更することができる．例えば，このメタインタプリタを少し変更するだけで，証明木を構築して返すメタインタプリタやトレース機能付きの小ステップ意味論に基づくメタインタプリタを実装することができる．

次節では本節で紹介した Prolog メタインタプリタのように容易に実装およびその拡張が可能なメタインタプリタの特徴について議論する．

3.1.2　メタプログラミング可能なメタインタプリタ

メタプログラミング可能なメタインタプリタとは容易に実装および変更が可能なメタインタプリタを指す．メタインタプリタは原理的にはあらゆるプログラミング言語で実装することができるが，メタインタプリタの規模と抽象度が適切でなければそのインタプリタはメタプログラミングをする上では使い物にならない．複雑すぎるメタインタプリタは容易に変更することができないし，単純すぎるメタインタプリタは拡張することが難しい．例えば，C 言語のメタインタプリタは優に 1000 行を超えるであろうし，Python のメタインタプリタは `exec` を使えば 1 行で実装できる．一方で Prolog のメタインタプリタは約 10 行程度，Lisp のメタインタプリタは約 100 行である．このようなメタインタプリタはそれらの言語機能を柔軟に変更するのに適切な抽象度のインタプリタである．

Lisp と Prolog には共通する 2 つの特徴が在る．1 つ目の特徴はプログラムがその言語自身の第一級のデータ構造で表現されているという，同図像性 (Homoiconicity) と呼ばれる性質を持っている点である．2 つ目の特徴はプログラム実行のための処理系の基本的な

機能がプログラムから使用できる点である．Prolog においてはプログラムは項 (term) で表現され，`clause` や `call` といったプログラム実行のための述語が処理系から提供されている．また，Lisp においてはプログラムはリストで表現され，`eval` や `apply` などの関数が在る．

本研究では LMNtal においてメタプログラミング可能なメタインタプリタを実装するためのフレームワークを提案する．フレームワークは上述の Lisp と Prolog に共通する 2 つの特徴を LMNtal で実現するためのもので，以下の 2 点から成る．1 つ目は第一級の書き換え規則である．通常の書き換え規則の代わりに第一級の書き換え規則を用いた LMNtal プログラムは LMNtal の第一級のデータ構造となる．2 つ目は SLIM のプログラム実行のための内部機能を LMNtal プログラムから使用できるようにした API である．次章以降で第一級書き換え規則と API について詳細に述べる．

3.2　第一級書き換え規則

本章では第一級書き換え規則の設計についてその構文と意味にわけて詳しく述べる．また，第一級書き換え規則の実装についても述べる．

LMNtal プログラムは階層グラフと書き換え規則によって構成される．もし書き換え規則をグラフとして扱うことができれば，LMNtal プログラムは第一級の値となり，LMNtal プログラムによって自由に作ったり操作したりすることが可能となる．本研究ではグラフで表現された書き換え規則として第一級の書き換え規則を設計および実装を行った．第一級書き換え規則は 3 価の`':-'` を起点として構成される下記のようなグラフである．

$$\texttt{':-'(\{}\mathit{Head}\texttt{\},\ \{}\mathit{Guard}\texttt{\},\ \{}\mathit{Body}\texttt{\})},$$

ただし，*Head*，*Guard*，*Body* はプロセスを表している．このような第一級書き換え規則は書き換え規則 *Head* `:-` *Guard* `|` *Body* を表す．

3.2.1　構文

第一級書き換え規則の構文を図 3.2 に示す．

ヘッド部とボディ部においては，第一級書き換え規則内ではテンプレートの構文要素をプロセスで表現することになる．アトムはアトムによって，膜は膜によって表現する．ただし，`=` アトムは `==` アトムによって表現する．例として同じ意味のルールと並べて図 3.3 に示す．

$$
\begin{array}{lcl}
F & ::= & \texttt{:-}\ (\{B\},\{G\},\{B\}) \\
B & ::= & 0 \\
 & | & \texttt{==}(L_1, L_2) \\
 & | & p(L_1, \ldots, L_m) \\
 & | & B, B \\
 & | & \{B\} \\
 & | & '\$p'(L_1, \ldots, L_m) \\
G & ::= & Con('\$p') \\
 & | & Op(X_1, \ldots, X_m) \\
 & | & G, G \\
Con & ::= & int \mid float \mid ground \mid unary \mid hlink \mid new \\
Op & ::= & =:= \mid = \backslash\backslash = \mid > \mid < \mid =< \mid >= \mid := \mid == \mid \backslash\backslash = \mid >< \\
 & | & + \mid - \mid * \mid \backslash \\
L & ::= & X \\
 & | & !H
\end{array}
$$

図 3.2　第一級書き換え規則の構文

$$
\begin{array}{lcl}
a\ \texttt{:-}\ c & \Rightarrow & '\ \texttt{:-}\ '(\{a\},\{\},\{c\}) \\
\{a\}\ \texttt{:-}\ c & \Rightarrow & '\ \texttt{:-}\ '(\{\{a\}\},\{\},\{c\})
\end{array}
$$

図 3.3　アトム及び膜を含む第一級書き換え規則

プロセス文脈は '\$p' アトムによって表現する．クォートされているのは，LMNtal のアトム名として \$ から始まる文字列は許されていないからである．

リンクの表現について説明する．リンクには局所リンクと自由リンクという 2 種類のリンクが存在するが，局所リンクは局所リンクで表現し，自由リンクはハイパーリンクで表現する．リンクの表現方法については設計段階でいくつかの候補が存在したため，以下ではそれぞれの候補の特徴を説明する．

候補の 1 つ目は全てのリンクをリンクで表現する方法である．この設計ではルールの $Head$,$Body$ に出現する局所リンクは局所リンク，自由リンクも自由リンクで表現する．例を図 3.4 に示す．この仕様の長所は，仕様が直感的でわかりやすい点とプログラムが簡潔である点である．リンクはそのままリンクで表現するのが 1 番直感的であるし，ハイパーリンクでの表現に比べて，ルールを用いて生成することなくプロセスとして記述できるためプログラムが簡潔である．また，実装に関しても処理系に実装してあるプロセスを

```
 a(X),b(X) :- a(Y),c(Y)
⇒
 ':-'({a(X),b(X)},{},{a(Y),c(Y)})
 a(X) :- b(X)
⇒
 ':-'({a(X)},{},{b(X)})
```

図 3.4　全てのリンクをリンクで表現した第一級書き換え規則

文字列として出力する関数を使ってルールの文字列表現を生成することができる．短所は任意の第一級書き換え規則を消去したり，複製したりすることができない点である．リンク条件によって，第一級書き換え規則における $Head$ 部を表す膜の自由リンクは全て $Body$ 部を表す膜につながる．そのため，直感的には図 3.5 のルールで任意の第一級書き換え規則を消去できると考えるが，実際はリンク束に関する実装がなされていないことが原因でコンパイルできない．

```
':-'({$head[|*X]},{$guard[]},{$body[|*X]}) :-
```

図 3.5　任意の第一級書き換え規則を消去するルール

候補の 2 つ目は. 全てのリンクをハイパーリンクで表現するこの設計ではルールの $Head$,$Body$ に出現する局所リンクも自由リンクもハイパーリンクで表現する．例を図 3.6 に示す．この仕様の長所は，任意の第一級書き換え規則を消去したり，複製したりする

```
 a(X),b(X) :- a(Y),c(Y)
⇒
Head:- new($x),new($y) |':-'({a($x),b($x)},{},{a($y),c($y)})
⇒
 a(X) :- b(X)
⇒
Head:- new($x)|':-'({a($x)},{},{b($x)})
```

図 3.6　全てのリンクをハイパーリンクで表現

ことができる点である．ハイパーリンクは unarry 型に包含されるため，$Head$ 部，$Body$ 部を表す膜それぞれに自由リンクが存在しない．そのため図 3.7 に示すルールによって任意の第一級書き換え規則を消去することが可能である．同様に複製することも可能である．短所としてはプログラムが長くなる点がある．全てリンクで表現したプログラムに比

```
':-'({$head[]},{$guard[]},{$body[]}) :-
```

図 3.7　任意の第一級書き換え規則を消去するルール

べて相当長くなってしまう．さらに，図 3.8 のようにリンクを含むプロセスのパターンから書き換え規則を生成することができない．1 行目は初期プロセスで，2 行目のルールでは膜内のプロセスから第一級書き換え規則を生成しようとしているが，うまくいかない．なぜなら，$Head$ 部，$Body$ 部を表現する膜内に局所リンクが含まれているからである．

```
{head,a(b)},{body,c(c)}.
{head, $head[]}, {body, $body[]} :- ':-'({$head[]},{},{$body[]})
```

図 3.8　局所リンクを含むプロセスパターンから書き換え規則を生成する

最終的に採用した表現方法は局所リンクは局所リンクで表現，自由リンクはハイパーリンクで表現というものである．この設計の特徴は任意の第一級書き換え規則を消去，複製することができ，局所リンクを含むプロセスのパターンから書き換え規則を生成することも可能である点である．$Head$，$Body$ 部を表現する膜内に自由リンクが出現しないため，全てのリンクをハイパーリンクで表現したときと同様にルール (図 3.7) を用いて任意の書き換え規則を消去することが可能である．また，局所リンクはそのまま局所リンクで表現するため，図 3.8 のように局所リンクを含むプロセスのパターンから第一級書き換え規則を生成することが可能である．さらに，ハイパーリンクを用いるのは自由リンクを表現する場合のみなので，全てをハイパーリンクで表現したときに比べてプログラムの長さも短い．また，この設計では自由リンクを含むプロセスのパターンから第一級書き換え規則を生成できない．なぜなら，自由リンクはハイパーリンクで表現しなければならないからである．例を図 3.9 に示す．しかし，このように自由リンクを含むプロセスパターンから書き換え規則を生成するような操作はあまり重要ではないと考える．

```
{head, a(X)},{body, c(X)}.
{head, $head[X]},{body,$body[X]} :- ':-'({$head[X]},{},{$body[X]})
```

図 3.9　自由リンクを含むプロセスパターンから書き換え規則を生成する

ガード構文においては型名はその名前をアトムを用いる．ガードで用いる算術式はその構文木をグラフで表現する．例えば，

$$\$p{=<}\$q + 3$$

というガード構文は第一級書き換え規則内では

$$=<('\$p', +('\$q', 3))$$

のように表現される．また，=はイコールアトムとしての意味を持ってしまうため，ガード部内では代わりに:=を用いる．例えば，

$$\$p = \$q * 3 + \$r$$

というガード構文は第一級書き換え規則内では

$$:= ('\$p', +(*('\$q', 3), '\$r'))$$

のように表現される．

3.2.2　意味論

第一級書き換え規則のための遷移規則を図 3.15 に示す．規則中の P はプロセス，F は第一級書き換え規則，c は後述する第一級書き換え規則から対応する書き換え規則への変換関数である．$c(F)$ はこの関数に第一級書き換え規則を渡すことを表しており，同時にその結果得られる書き換え規則を表現している．新たに導入した遷移規則は，第一級書き換え規則によるプロセスの書き換えは，第一級書き換え規則を対応する書き換え規則に置き換えた場合に発生する書き換えであることを表している．

変換関数 c の定義を図 3.10 に示す．変換関数は引数として第一級書き換え規則を受け取り，第一級書き換え規則を構成する階層グラフ構造をトラバースすることによって対応する書き換え規則を返す．c 内で呼び出される関数 $links$ 及び関数 c' の定義をそれぞれ図 3.11 と図 3.12 に示す．これらの関数は第一級書き換え規則内のリンクの接続構造を

```
function c(': −'({B}, {G}, {B'})
 S := links(F_h, links(F_b, ∅))
 (FH, S') := c'(B, S)
 (FB, S'') := c'(B', S')
 FG := g(G)
 if S'' = ∅ then
  return FH :− FG | FB
 else
  return 0
 end
end
```

図 3.10　変換関数 c

書き換え規則内のリンクに変換するための関数である．$links$ によってリンクを集め，c' によって接続を行う．また関数 c 内ではガード部の処理のために関数 g が呼び出される．関数 g の定義を図 3.13 に示す．関数 g はガード内のグラフ構造をトラバースして，書き換え規則のガード部を生成する．関数 g_op はガード内に記述される算術式のトラバースを行う．算術式はその構文木をグラフで表現しているため，g_op は構文木の構文解析を行う．図 3.14 に関数 g_op の定義を示す．$links$ 内で呼び出されている関数 cnt はカウンター関数であり，g_op 内で呼び出される関数 $connected$ は引数にリンクを取り，接続先のアトムを返す関数である．

3.2.3　実装

本節では今回行った第一級書き換え規則の実装に関する説明を行う．LMNtal における第一級書き換え規則の実装は既存の LMNtal 処理系 SLIM[19] に実装するという形で導入を行ったため，SLIM に関する解説もあわせて行う．

3.2.4　SLIM の通常実行モード

LMNtal プログラムはまず JAVA で記述されたコンパイラによって中間命令列にコンパイルされ，C 言語で記述されたランタイム (SLIM) によって実行される．コンパイラはソースプログラムに記述されているルール一つに対して対応する中間命令列を一つ出力する．従って，コンパイル済みの LMNtal プログラムは対応するルール毎にグルーピングされた中間命令列の列となっている．SLIM の実行形態はプロセスの最終状態の内の一つ

```
function links(B, S)
 if B = p(L_1, ..., L_m) then
  for i = 1 to m do
   if (L_i, j) ∈ S then
    S := S ∪ {(L_i, j)}
   else
    S := S ∪ {(L_i, cnt()}
   end
  end
  return S
 elseif B = X==Y then
  if (X, j) ∈ S then
   S := S ∪ {(X, j)}
  else
   S := S ∪ {(X, cnt()}
  end
  if (Y, j) ∈ S then
   S := S ∪ {(Y, j)}
  else
   S := S ∪ {(Y, cnt()}
  end
  return S
 elseif B = B_1, B_2 then
  return links(B_1, links(B_2, S))
 end
 elseif B = {B'} then
  return links(B')
end
```

図 3.11　関数 *links*

のみを出力する通常実行と，可能なすべての書き換えパターンを試して実行途中のプロセスの状態とそれら状態の遷移を出力する非決定実行がある．今回実装を行ったのは通常実行モードである．SLIM の通常実行における実行経路を今回実装した部分に絞って解説する．概略を図 3.16 に示す．

SLIM による通常実行は初期プロセスの配置から始まる．初期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び，プロセス全体をネストされた膜の階層構造とみる．そして，世界的ルート膜から順にネストされた膜をスタック構造に積んでゆく．この時，コンパイルされたルールは中間命令列としてそのルールが所属していた膜に

```
function c'(B, S)
 if B = p(L_1, ..., L_m) then
  for i = 1 to m do
   S := S \ {(L_i, j)}
   replace(B, L_i, L_j)
  end
  return (B, S)
 elseif B = X==Y then
  S := (S \ {(X, i)}) \ {(Y, j)}
  return (X=Y, S)
 elseif B = B_1, B_2 then
  (F1, S') := c'(B_1, S)
  (F2, S'') := c'(B_2, S')
  return ((B_1, B_2), S'')
 end
end
```

図 3.12　関数 c'

```
function g(G)
 if G = Con('$p') then
  return Con($p)
 elseif G = Op(X_1, ... X_m) then
  return g_op(G)
 elseif G = G_1, G_2 then
  return g(G_1), g(G_2)
 end
end
```

図 3.13　関数 g

結びつけられている．次にスタックのトップにある膜から順にその膜に所属するプロセスに対して，その膜に結びつけられた中間命令列を実行してゆく．中間命令列の実行によってプロセスの書き換えが発生しなくなるまで繰り返し書き換えを行い，書き換えが発生しなくなると，スタックをポップして次の膜の処理に移る．これら一連の処理をスタックが空になるまで行う．このようにして LMNtal プログラムは実行される．

第一級書き換え規則の実装とは前節で述べた関数 c の実装である．実行対象になっている膜内に第一級書き換え規則が存在すれば，関数 c によって対応する書き換え規則の文字

```
function g_op(Op(X_1, ... X_m))
 if m = 1 then
  return Op
 else
  res := ""
  if connected(X_1) is data then
   res += connnected(X_1)
  else
   res += g_op(connnected(X_1))
  end
  if Op = ':=' then
   res += '='
  else
   res += Op
  end
  if connected(X_2) is data then
   res += connnected(X_2)
  else
   res += g_op(connnected(X_2))
  end
 end
 return res
end
```

図 3.14　関数 *g_op*

$$\frac{P, c(F) \to P', c(F)}{P, F \to P', F}$$

図 3.15　第一級書き換え規則の遷移規則

列表現を得る．その後コンパイラを呼び出して文字列をコンパイルして中間命令列を得る．得られた中間命令列は膜にロードされ，通常の書き換え規則同様に実行対象となる．

第一級書き換え規則に対する書き換えが発生した場合は，対応する中間命令列も書き換える必要がある．本研究では第一級書き換え規則と生成された中間命令列の対応関係を膜毎に記憶しておくことによって，第一級書き換え規則の書き換えが発生した場合には対応する中間命令列を書き換えられるようにしている．各膜毎に:-アトムのポインタをキーに，中間命令列に付けられるルール ID をバリューとしたテーブルを用意する．アトムを

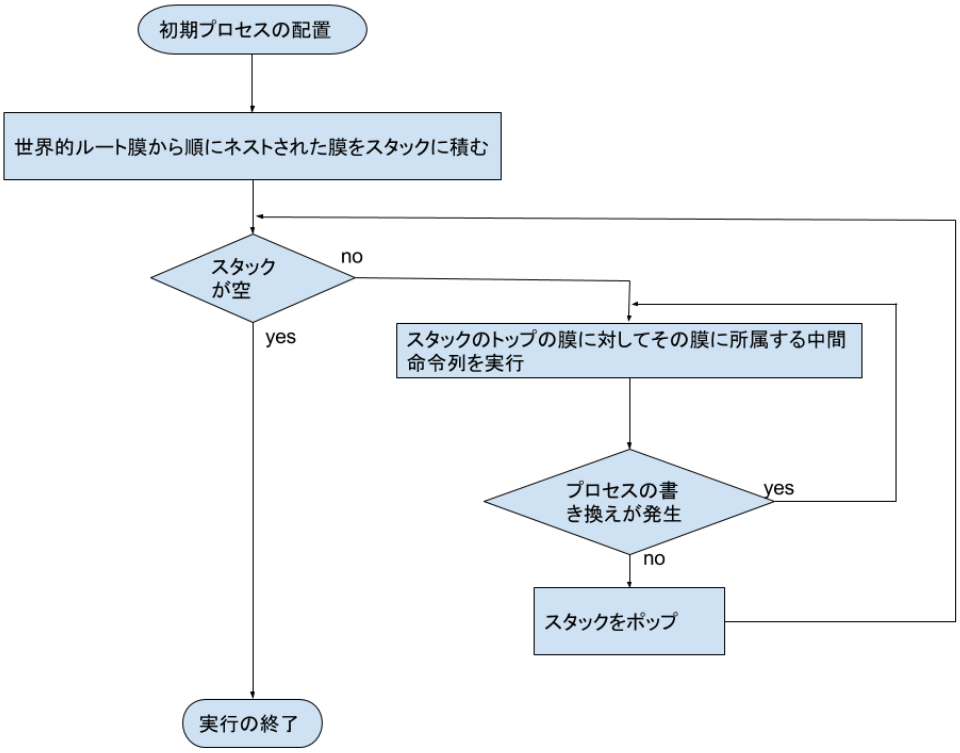

図 3.16　通常実行時における SLIM の実行経路概略図

追加したり，削除したりする中間命令の実装に，その命令対象となっているアトムが:-アトムかどうかを検査する処理を追加した．対象になっている場合は第一級書き換え規則の書き換えが発生したとして，結び付けられている中間命令列を一旦削除し，新たに生成された第一級書き換え規則に対応する中間命令列を生成する．このように第一級書き換え規則と中間命令列を結びつけることによって，第一級書き換え規則の書き換えが発生した時のみ新たな第一級書き換え規則のコンパイル処理を行うように実装することができた．

3.3　処理系内部機能への API

本研究では LMNtal プログラムがプログラムの状態遷移グラフを構築できるようにするための API を設計および実装した．API は図 3.17 に示すように 4 つのアトムで構成されており，それらは SLIM の内部機能にアクセスするための多言語インタフェースを用いて実装を行った．1 つ目の `state_space.react_nd_set` は非決定的な状態遷移のすべての遷移先状態を計算するための API であり，残りの API は状態管理のための

- state_space.react_nd_set(*RuleMem*, *GraphMem*, *RetRule*, *Ret*)
- state_space.state_map_init(*Ret*)
- state_space.state_map_find(*Map*, {$*key*[]}, *Res*, *Ret*)
- state_space.state_map_find(*Map*, $*key*, *Res*, *Ret*)

図 3.17　state space API の構文

state_map コレクションの操作関数である．ユーザは state_space ライブラリをインポートするだけでこれらの API をプログラムから自由に使用することができる．

図 3.17 のアトムは与えられた書き換え規則を返すために *RetRule* を持つ。LMNtal では API に入力されたデータは他のリンクを通して明示的に返さない限り，資源として消費されるため，このようなリンクは後にルールを使用するために必要である．

3.3.1　state_space.react_nd_set

state_space.react_nd_set は *RuleMem* に接続された膜内のルールを *GraphMem* に接続された膜内のプロセスに適用し，可能なすべての書き換え結果を返す．それぞれの書き換え結果はリストに格納されて *Ret* を通して返される．もし書き換え結果が存在しなければ，空リストが返される．

例えば，state_space.react_nd_set に以下のような入力があれば

```
state_space.react_nd_set({':-'({a(!X)}, {}, {b(!X)})},
                         {a(1),a(2),a(3)},
                         retrule,
                         ret).
```

下記のように書き換える．

```
retrule({':-'({a(!X)}, {}, {b(!X)})}),
ret([{b(1),a(2),a(3)}, {a(1),b(2),a(3)}, {a(1),a(2),b(3)}]).
```

3.3.2　state_map collection

state_map コレクションはそれぞれの状態を管理し，個々の状態にユニークな ID を割り振る．ユニーク ID は複雑な LMNtal グラフを扱いやすくし，状態の遷移関係を効率よく表現するのに適している．state_map_init と state_map_find はこのコレクションに対する操作関数である．state_map_init は *Ret* に空の state_map を接続して返す．state_map_find は第 2 引数の型によって返される結果が異なる．もし第 2 引数が状態としての膜なら，状態の一意な ID を *Res* を通して返す．もし状態の ID を表す int アトムならば，その ID を持つ状態としての膜を *Res* を通して返す．state_map コレクションは *Map* に入力として与え，*Ret* を通して返される．

例えば state_map_init と state_map_find は下の様に振る舞う

```
    state_map_find(state_map_init, {a(1),a(2),a(3)}, res, ret).
⟶ state_map_find(<state_map>, {a(1),a(2),a(3)}, res, ret).
⟶ res(13458), ret(<state_map>).
```

最初のステップでは state_map_init が空の state_map コレクションを返す．次のステップでは state_map_find が膜 {a(1),a(2),a(3)} を記録して，その一意な ID(13458) と新しい state_map を返す．第 2 章で述べたように，LMNtal グラフのユニーク ID の効率的な計算はグラフのハッシュ計算と同型性判定を含むため，自明ではない．state_map はこれらの同型性判定などの SLIM 内部の機能を LMNtal プログラマがそれらを再実装することなく使用することを可能にしている．

3.3.3　API の設計領域

本節では 4 つの状態空間のための API を導入した。state_map コレクションは状態管理を効率的に行うための API である．状態空間を構築するための API として，状態間の等価性を判定する API を state_map の代わりに用いるという選択肢もあった．[15] で発表した前バージョンでは state_map の代わりにグラフ同型性判定を行う membrane.eq を用いてメタインタプリタを構築した．この等価性判定の API を用いても十分強力メタインタプリタの実装も可能ではあったが，実装されたメタインタプリタはパフォーマンス上の問題を抱えていた．そこで，SLIM の状態空間を管理する機能を利用するためのインターフェースを実装し，SLIM で実装されている様々な最適化を利用することにした．これにより，状態の等価性判定と状態管理の機能を保ちつつメタインタプリタの性能が向上

した．実装したメタインタプリタの性能に関しては詳しく後述する．

3.3.4　実装

本研究では処理系 SLIM の多言語インタフェースを用いて API の実装を行った．SLIM の多言語インタフェースとは，グラフの書き換えを C 言語で実装した関数によって行う仕組みである．多言語インタフェースによってプログラマは，LMNtal プログラムから C 言語で実装したコールバック関数を呼び出して書き換えを行うことができ，LMNtal の書き換え規則ではできないような書き換えを行うことができる．’$callback("*function_name*",...)’ というアトム生成することによって LMNtal プログラムからコールバック関数 *function_name* を呼び出すことができる．’$callback’ アトムの第二引数以降はコールバック関数の第一引数から順次渡される．例えば，前節で紹介した以下のグラフ構造は

```
state_space.react_nd_set({':-'({a(!X)}, {}, {b(!X)})},
                         {a(1),a(2),a(3)},
                         retrule,
                         ret).
```

実際には state_space ライブラリ内の以下の書き換え規則によって’$callback’ アトムに書き換えられる．

```
Ret = state_space.react_nd_set(Rule, Graph, RetRule) :-
 '$callback'('cb_react_ruleset_nd', Rule, Graph, RetRule, Ret).
```

すなわち，state_space.react_nd_setAPI の実装は C 言語で実装されたコールバック関数 cb_react_ruleset_nd である．

cb_react_ruleset_nd

処理系 SLIM には書き換え規則とグラフ構造を引数にとり，適用した書き換えた結果を返す関数 react_rule が存在する．さらに，react_rule にはモードが存在し，通常実行モードの他にモデル検査時に使用する非決定モードが在る．非決定モードではパターンマッチ可能グラフ構造すべてに対して書き換えを行い，書き換え結果のグラフ構造はベクタで返す．

cb_react_ruleset_nd 関数内では引数を適切に整形する前処理を行い，react_rule

を非決定モードで呼び出している．その後，ベクタで返された書き換え結果をリスト構造に変換する処理を行っている．

state_map コレクション

state_map コレクションの実装では SLIM がモデル検査を行う時に用いる StateSpace 構造体とその操作関数を用いている．StateSpace 構造体の基本構造は状態のハッシュテーブルであるが，様々な最適化のためのフラグや補助のハッシュテーブルを内包しているためその構成は非常に複雑である．特に状態の等価性判定機能ための最適化は非常に精密に実装されている．

状態を受け取ってユニーク ID を返す state_map_find のコールバック関数では，受け取った状態をそのまま StateSpace 構造体の操作関数 statespace_insert へ渡している．この関数では，もし受け取った状態が StateSpace の状態のハッシュテーブルに登録済みであれば，既に登録されている状態へのポインタを返し，新規状態であれば受け取った状態を返す関数である．statespace_insert を呼び出し側では，引数で渡した状態のポインタと返ってきた状態のポインタを比較することで，新状態かどうかを判定している．state_map_find が返す状態のユニーク ID は内部的には状態のポインタである．

第 4 章

LMNtal メタインタプリタ

本章ではまず LMNtal メタインタプリタについて述べる．LMNtal メタインタプリタとは，LMNtal プログラムを入力としてプログラムの状態遷移グラフを出力する LMNtal プログラムである．このメタインタプリタは第一級の規則と処理系 SLIM の内部機能に対する API を用いて実装を行った．実装した LMNtal メタインタプリタの性能についても述べる．また，実装した LMNtal メタインタプリタと Prolog メタインタプリタとの違いについても述べる．

4.1 アルゴリズム

LMNtal メタインタプリタは入力されたプログラムから状態遷移グラフを構築する．状態遷移グラフ構築のアルゴリズムを示した擬似コードを図 1 に示す．状態遷移グラフは状態の集合 S と状態遷移の集合 T の組である．はじめは S は初期状態 s_0 のみを要素としており，T は空である．また，未展開状態のスタックには初期状態のみがプッシュされている状態である．アルゴリズムは全体で 2 重ループを構成し，深さ優先探索によって状態遷移グラフを構築する．while ループでは未展開状態のスタックの先頭状態が展開される．*expand* は渡された状態を遷移元として，遷移先の状態を全て返す手続きである．forall ループでは，展開によって得られた遷移先状態は新状態であるか，またその遷移関係が新規であるかを判定さる．新規であればそれぞれ集合に追加される．

Algorithm 1 状態遷移グラフの構築

```
S := {s_0}; T := ∅; Stack := ∅
push s_0 Stack
while Stack ≠ ∅ do
  s := pop Stack
  succ := expand(s)
  for all s' ∈ succ do
    if s' ∉ S then
      S := S ∪ {s'}
      T := T ∪ {(s, s')}
      push s' Stack
    else if (s, s') ∉ T then
      T := T ∪ {(s, s')}
    end if
  end for
end while
```

4.2　実装

本節では LMNtal メタインタプリタの実行の流れについて述べる．実装した LMNtal メタインタプリタは LMNtal プログラムを入力として，プログラムの状態遷移グラフを出力する．プログラムの非決定的な実行に伴って現れる全状態とその遷移関係を表したグラフ構造を状態遷移グラフと呼ぶ．LMNtal プログラムの状態とは階層グラフであり，状態遷移とは規則による階層グラフの書き換えである．

LMNtal メタインタプリタの入力は LMNtal プログラムである．入力とする LMNtal プログラムは次のような階層グラフ構造で表現される．

$$Ret = \texttt{run}(RuleSet,\ Init).$$

RuleSet には第一級書き換え規則を含む膜，*Init* には初期状態を表す階層グラフが接続される．*Ret* はこの階層グラフへの参照である．メタインタプリタの出力は状態遷移グラフであり，次のような階層グラフで表現される．

$$Ret = \texttt{state\_space}(Init,\ Map,\ States,\ Transition)$$

Ini は初期状態の uniq ID が接続される．Map は状態遷移グラフ中のすべての状態を ID へとマッピングする `state_map` コレクションが接続される．*States* は状態の集合としてのハッシュテーブル，*Transition* は状態遷移の集合としてのハッシュテーブルが接続される．*Ret* はこの状態遷移グラフへの参照である．

LMNtal プログラムの状態はアトム，リンク，膜から成る階層グラフ構造であるが，出力結果の状態遷移グラフではそれを一意な ID で抽象化している．抽象化によってメタインタプリタの実行時間を効率化し，メモリ使用量を削減することができる．グラフ構造から ID への抽象化，ID からグラフ構造への具体化は `state_map` コレクションと `state_map_find` によって可能である．状態遷移は次のような階層グラフで表現する．$Ret = \text{'.'}(From, To)$, where $From$ には遷移元の状態を表す階層グラフを含む膜，To には遷移先の状態を表す階層グラフを含む膜が接続される．遷移関係をハッシュテーブルで管理するのは，状態 A から状態 B への遷移が複数ある場合に重複する辺を除去するためである．重複する辺を除去することによって，冗長な状態遷移グラフの構築を避けることができる．

図 4.1 に実装した LMNtal メタインタプリタのプログラムを示す．各規則の先頭には識別子 *rulename* を “*rulename*`@@`” のように付加した．以下説明のために各規則を *rulename* で呼ぶ．run と exp0 はメタインタプリタの初期化を行う．これらのルールで空の集合や初期状態だけを push したスタックを生成する．exp と exp’ のルールは擬似コード中の whie ループの条件判定に相当する処理を行う．スタックが空でなければ，exp は `react.nd_set` よって未展開状態を展開する．スタックが空であれば，exp’ が `state_space` アトムを返してメタインタプリタは停止する．suc と suc’ のルールは擬似コード中の forall ループの条件判定に相当する処理を行う．展開元となった状態 s の遷移先状態が調べ終わっていなければ，suc はその内の一つ s’ を選んで uniq ID を `state_map_find` によって調べる．調べ終わっていれば，suc’ によって exp にもどる ns と ns’ は s’ が新状態かどうか判定する新状態であれば，ns は s’ を状態の集合に追加，s と s’ の遷移関係を遷移の集合に追加する．nt と nt’ は s と s’ の遷移関係が新規かどうか調べる．遷移関係が新規であれば，遷移の集合に遷移関係を追加する．我々は状態と状態遷移の管理に LMNtal の set library を用いた set.empty は空の集合を表現している set.find はハッシュテーブルの lookup, set.insert は要素をハッシュテーブルへ挿入する

例としてこのメタインタプリタに次のような階層グラフを入力して実行する．

```
ret = run({':-'({a(!X)}, {}, {b(!X)})}, {a(1), a(2), a(3)}).
```

実行の結果 4 価の `state_space` アトムとして状態空間が得られる．この状態空間は以下

表 4.1　実験環境

CPU	Intel Xeon E5-4620 v2
CPU frequency	2.6GHz
Memory	512 GiB

の 2 つのルールを用いて状態を簡単に LMNtal プロセスに変換することができる．

```
Ret = state_space(I, M, S, T) :-
    Ret = ss(I, M, set.to_list(S), set.to_list(T)).
Ret = ss(I, M, [$x|S], T) :- int($x) |
    Ret = ss(I, state_space.state_map_find(M, $x, Res), S, T),
    state($x, Res).
```

この 2 つのルールを state_space アトムに適用することによって以下を得る

```
ret(ss(5056,<state_map>,[],
      [[6336|6208],[5568|6208],[5440|6336],[5056|5312],[5056|4928],[5312|5696],
       [5440|5568],[4928|5696],[5312|6336],[4928|5568],[5696|6208],[5056|5440]])).
state(5056,{a(1). a(2). a(3). }). state(5696,{a(3). b(1). b(2). }).
state(5312,{a(1). a(3). b(2). }). state(6336,{a(1). b(2). b(3). }).
state(4928,{a(2). a(3). b(1). }). state(5568,{a(2). b(1). b(3). }).
state(6208,{b(1). b(2). b(3). }). state(5440,{a(1). a(2). b(3). }).
```

初期状態 a(1)，a(2)，a(3) からの遷移先はどの a アトムが書き換えられるかによって 3 通りに分岐する．また，最終状態は b(1)，b(2)，b(3) のみである実行結果出力される階層グラフが表現する状態遷移グラフを図 4.2 に示す．図 4.2 の円は状態を表し，矢印は状態遷移を表す．図の最も左に位置する状態は初期状態であり，最も右に位置する状態が最終状態である．

4.3　性能評価

本節では LMNtal メタインタプリタの実行時時間性能について述べる．実験環境は表 4.1 に示す計算機を使用した．評価実験では下記 4 題について LMNtal メタインタプリタと SLIM の実行時間を比較した．実験結果を図 4.3 に示す．

- 食事する哲学者
- ハノイの塔

- ユークリッドの互除法
- リスト構造のソート

食事する哲学者は哲学者全員が左のフォークを手にとって右のフォークが食卓に置かれるのを待つというデッドロック状態に至る可能性が存在するモデルである．リスト構造のソートのプログラムを下に示す．

```
L = [$x,$y|L2] :- $x > $y | L = [$y,$x|L2].
```

このプログラムは LMNtal 風に書いたリストのソーティングのプログラムである．ルールはリストの隣接した値が昇順になっていなければ，値の順序を交換する．どの隣接した値も昇順になっていれば，プログラムは停止してリストはソートされる．

上記 4 題以外の 20 題以上の例題についても同様の実験を行った．実験結果を表 4.2 に示す．実験では各例題について状態数毎に実行に要する時間を 3 回計測した平均値を評価した．実験結果から LMNtal メタインタプリタの性能は SLIM の状態遷移グラフの構築処理と比較して，オーダーに大きな差は見られず，1 桁以下の定数倍の性能低下にとどまっていることを確認した．図 4.4 にメタインタプリタの実行時間に対する API の実行時間の割合のグラフを示した．用いた例題は上で示したハノイの塔である．`interpret` は状態空間構築における状態の等価性判定と状態の展開処理以外の部分に相当する．各 API の実験結果からメタインタプリタが呼び出す API に目立ったボトルネックは無いことが確認できる．

4.4　メタインタプリタの拡張

4.4.1　ユーザ定義の構造合同規則

メタインタプリタを利用して，LMNtal の意味論を拡張した処理系を簡単に作ることができる．例として，LMNtal の構造合同規則をユーザ定義可能になるように拡張した処理系を作成する．このインタプリタは先ほどまでと異なり，SLIM のシミュレーションモードを実装する．つまり，適用可能な書換え規則の中から 1 つを選んで適用し，適用できなければ停止する．$2x + x^2 + 3x + 1$ のような多項式を計算する LMNtal プログラムを考える．項 aX^b を `term(`a`,`b`)` とし，加算演算子を `add` というアトムで表すと，上記の多項式は次のような LMNtal プログラムと対応する．

```
ans = add(add(add(term(2,1), term(1,2)), term(3,1)), term(1,0))
```

同じ次数の項を加算するルールは次のようにかける．

表 4.2　状態空間構築の性能

Instance Name	States	Time(s) (SLIM)	Time(s) (Meta)	Ratio
Knight_4	1657	0.06	0.28	4.66
Knight_5	508493	15.42	137.31	8.90
Peterson_2	115	0.03	0.06	2.00
Peterson_3	7779	0.70	5.17	7.38
PhiM_5	1370	0.22	1.68	7.68
PhiM_6	5785	1.12	9.21	8.22
PhiM_7	24484	5.96	52.57	8.82
PhiM_8	103691	31.25	308.90	9.88
Qlock_5	657	0.07	0.35	5.00
Qlock_6	3920	0.34	2.33	6.85
Qlock_7	27407	2.55	19.39	7.60
Qlock_8	219210	22.59	203.27	8.99
Queen_8	2057	0.09	0.46	5.11
Queen_9	8394	0.32	1.85	5.78
Queen_10	35539	1.37	7.98	5.82
Queen_11	166926	6.56	46.46	7.08
Rabbit_10	22052	1.20	4.75	3.95
Rabbit_12	92020	5.63	23.77	4.22
Rabbit_14	377234	25.36	159.66	6.29
Rabbit_16	1531664	110.04	1570.23	14.26

```
R = add(term($a,$x), term($b,$y)) :- $x=:=$y | R = term($a+$b,$x).
```

しかし，このルールによって上記の多項式を簡単にすることはできない．なぜなら，2x と 3x は + の作る木の中で兄弟ではないからであるそこで，結合法則と交換法則を構造合同規則として定義する．

$$(Assoc)\quad \texttt{R = add(add(A, B), C)} \equiv \texttt{R = add(A, add(B, C))}$$
$$(Comm)\quad \texttt{R = add(A, B)} \equiv \texttt{R = add(B, A)}$$

これを用いて，次のように計算できる．

```
     ans = add(add(add(term(2,1), term(1,2)), term(3,1)), term(1,0))
≡    ans = add(add(add(term(1,2), term(2,1)), term(3,1)), term(1,0))   (Comm)
≡    ans = add(add(term(1,2), add(term(2,1), term(3,1))), term(1,0))   (Assoc)
→    ans = add(add(term(1,2), term(5,1)), term(1,0))
```

これを実現する処理系の実装の概略を以下に示す．まず，この処理系では現在の状態に対して構造合同規則の左辺を右辺に，右辺を左辺に書き換えるようなルールを用いて状態空間を構築する．次に，構築された状態の中から通常のルールを適用できるものを探す．次に，通常のルールを適用してできた新たな状態を現在の状態とする．ルールが適用できなくなるまで上記 3 ステップを繰り返す．

4.4.2　状態遷移グラフの探索への A*探索の導入

我々は別のメタインタプリタの変種として A*探索を行うインタプリタを示す．ある目的状態があり，初期状態からの最短手順を求めたいとする．こういった問題をモデル検査器を用いて解く手法はプランニング問題などで散見される．初期状態からの最短路を見つけるために，A*探索をメタインタプリタに組み込むことにする．このインタプリタは LMNtal プログラムとヒューリスティクス関数を入力として，初期状態から目的状態までの最短経路を出力する．ヒューリスティクス関数は状態のスコアを計算するルールの集合である．インタプリタは書き換え結果からこの値を参照して状態遷移のコストを得る．状態の等価性判定は `state_map` API を用いて実装している．ヒューリスティクス値や初期状態からの最小コストなどの状態の付帯情報は状態の ID を Key とした HashMap で管理している．実装したプログラムの主部分は 20 本ほどのルールから成っている．

```
run @@  Ret = run(Rs, {$ini[]}):-
        Ret = exp0(Rs, s(ID,{$ini[]}),
          state_space.state_map_find(state_space.state_map_init, {$ini[]}, ID),
          set.empty, set.empty).

exp0@@  Ret = exp0(RS, S0, Map, Ss, Ts), S0 = s($id,{$ini[]}) :- int($id) |
        Ret = exp(RS, [s($id,{$ini[]})], Map, set.insert(Ss, $id), Ts), ini($id).

exp @@  Ret = exp(RS, S0, Map, Ss, Ts), S0 = [s($id,{$src[]})|Stk] :- int($id) |
        Ret = suc(R, Stk, Exp, p($id,{$src[]}), Map, Ss, Ts),
        Exp = state_space.react_nd_set(RS, {$src[]}, R).
exp'@@  Ret = exp({$rs[],@rs}, [], Map, Ss, Ts), ini(I) :-
        Ret = state_space(I, Map, Ss, Ts).

suc @@  Ret = suc(RS, Stk, [{$dst[]}|Suc], Src, Map, Ss, Ts) :-
        M = state_space.state_map_find(Map, {$dst[]}, ID),
        Ret = ns0(RS, Stk, Suc, Src, p(ID,{$dst[]}), M, Ss, Ts).
suc'@@  Ret = suc(RS, Stk, [], p($id,{$src[]}), Map, Ss, Ts) :- int($id) |
        Ret = exp(RS, Stk, Map, Ss, Ts).

ns0 @@  Ret = ns0(RS, Stk, Suc, Src, p($d,D), Map, Ss, Ts) :- int($d) |
        Ret = ns(RS, Stk, Suc, Res, Src, p($d,D), Map, S, Ts),
        S = set.find(Ss, $d, Res).
ns  @@  Ret = ns(RS, Stk, Suc, some, p($s,Src), p($d,Dst), Map, Ss, Ts) :-
        int($s), int($d) |
        Ret = nt(RS, Stk, Suc, Res, p($s,Src), p($d,Dst), Map, Ss, T),
        T = set.find(Ts, '.'($s, $d), Res).
ns' @@  Ret = ns(RS, Stk, Suc, none, p($s,Src), p($d,Dst), Map, Ss, Ts) :-
        int($s), int($d) |
        Ret = suc(RS, [s($d,Dst)|Stk], Suc, p($s,Src), Map, S, T),
        S = set.insert(Ss, $d), T = set.insert(Ts, '.'($s,$d)).

nt  @@  Ret = nt(RS, Stk, Suc, some, Src, p($d, {$dst[]}), Map, Ss, Ts) :-
        int($d) |
        Ret = suc(RS, Stk, Suc, Src, Map, Ss, Ts).
nt' @@  Ret = nt(RS, Stk, Suc, none, p($s,Src), p(D, {$dst[]}), Map, Ss, Ts) :-
        int($s) |
        Ret = suc(RS, Stk, Suc, p($s,Src), Map, Ss, set.insert(Ts, '.'($s,D))).
```

図 4.1　LMNtal meta-interpreter

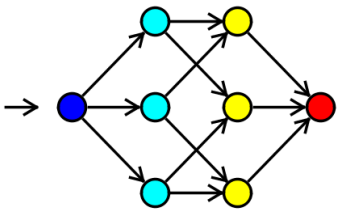

図 4.2　SLIM と LaViT によって描画される状態遷移グラフ

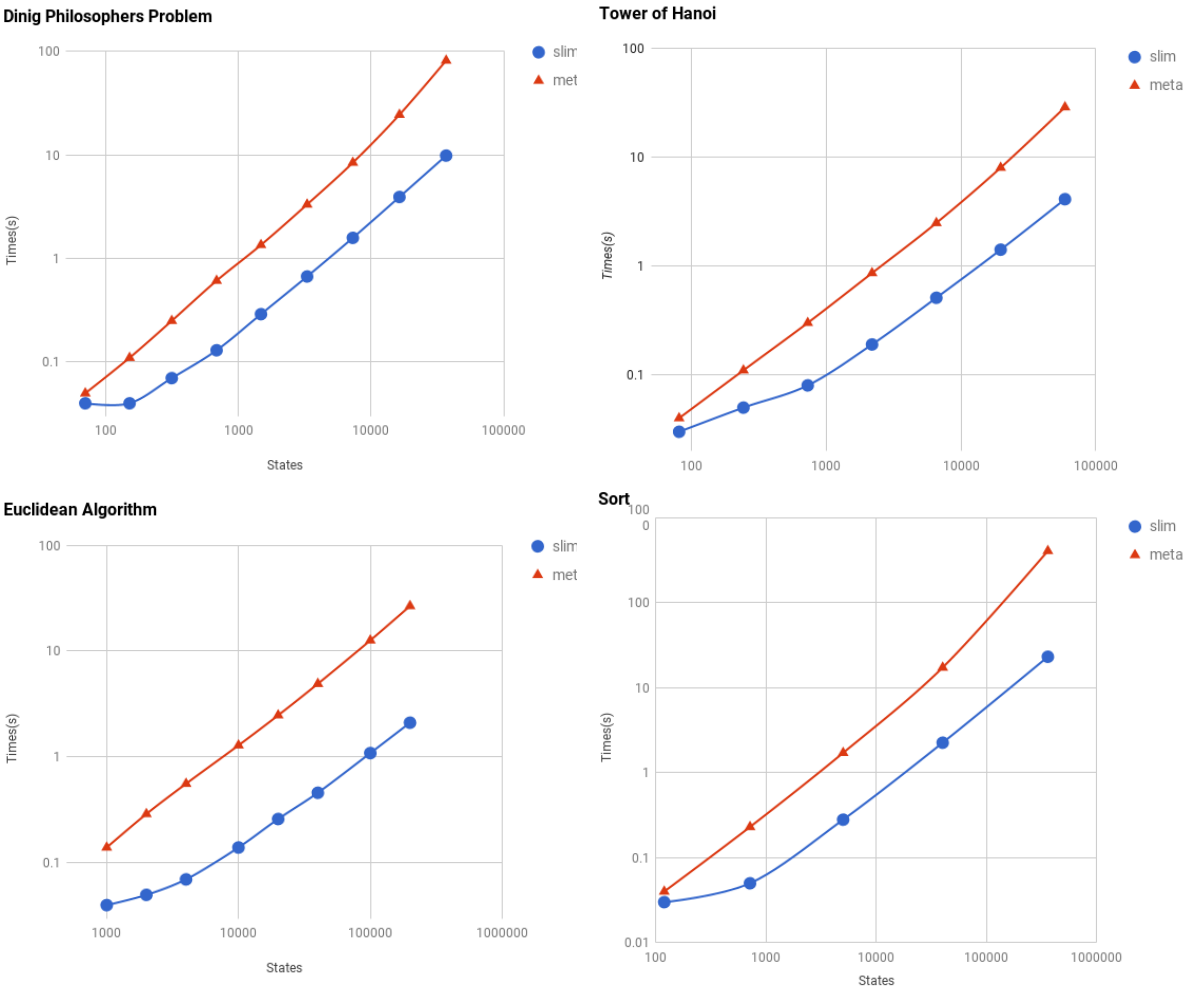

図 4.3　メタインタプリタと SLIM の実行時間の比較

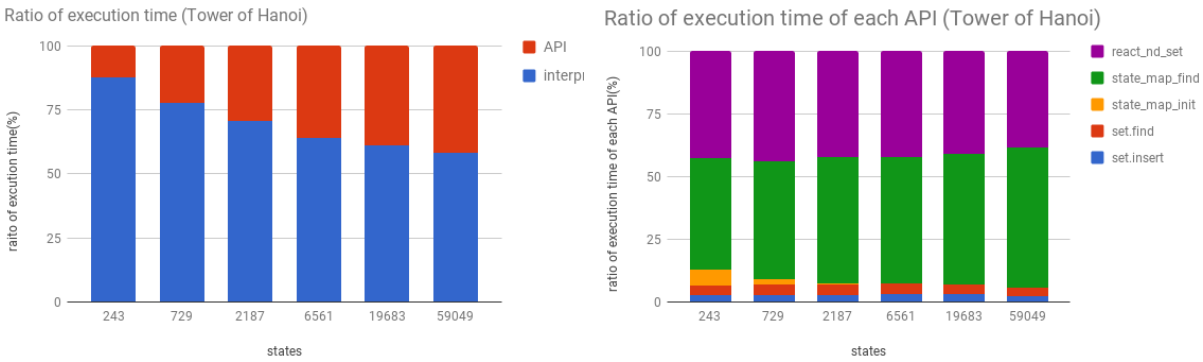
Ratio of execution time (Tower of Hanoi)
raito of excution time(%)
100
75
50
25
0
243
729
2187
6561
19683
59049
states
API
interpr
Ratio of execution time of each API (Tower of Hanoi)
ratio of execution time of each API(%)
100
75
50
25
0
243
729
2187
6561
19683
59049
states
react_nd_set
state_map_find
state_map_init
set.find
set.insert

図 4.4　実行時間の詳細

第 5 章

LMNtal モデル検査

本章では LMNtal メタインタプリタを用いて実装したモデル検査器について述べる．

5.1 LTL モデル検査

LTL モデル検査とは線形時相論理 (LTL) で記述された仕様を状態遷移システムが満たすかどうかを検証する手法である．LTL モデル検査器は LTL 式とモデル記述を入力して受け取り，モデルが LTL 式を満たしていれば真を返し，そうでなければ反例を返す．LTL モデル検査器がモデルが LTL 式 p の充足を検査する時，実際には $\neg p$ を満たす状態列を探索する．ある LTL 式 p は時相演算子を含まない命題 $p_0, p_1, \ldots$ の無限列の表現に対応する．状態列 $\pi = s_0, s_1, \ldots$ が LTL 式 p を充足することを $\pi \models p$ と記述し，任意の $k \geq 0$ について $s_k \models p_k$ となる場合に限り $\pi \models p$ となる．モデル検査器によって展開される状態遷移グラフはモデル記述と LTL 式から得られる Büchi オートマトンの同期積オートマトンから得られる．LTL 式 p から得られる Büchi オートマトンとは $\pi \models p$ となる状態列 π を受理するオートマトンである．

LTL 論理式は時間に関する性質を記述するのに使われる [5]．図 5.1 に LTL 論理式のグラフ表現を示す．LTL 論理式は時相演算子として $\Box$，$\Diamond$，U を持つ．図 5.1 ではこれらの演算子は `g`， `f`， `u` として定義されている．式 `f`(ϕ) は将来 ϕ を満たす状態が在ることを意味し，式 `g`(ϕ) は実行経路中の全状態が ϕ を満たすことを意味し，式 `u`(ϕ_1,ϕ_2) は ϕ_2 が満たされるまでずっと ϕ_1 が満たされることを意味し，式 `x`(ϕ) は次の状態で ϕ が満たされることを意味する．

実装した LTL モデル検査器は (1) モデル記述としての第一級書き換え規則と (2) 初期状態と (3)LTL 式の否定から得られる LMNtal グラフで表現された Büchi オー

(*LTL Formula*) ϕ ::=
`true` | `false` | *Predicate* | `not(`ϕ`)` | `or(`ϕ_1`,`ϕ_2`)`
| `and(`ϕ_1`,`ϕ_2`)` | `imply(`ϕ_1`,`ϕ_2`)` | `g(`ϕ`)` | `f(`ϕ`)` | `x(`ϕ`)` | `u(`ϕ_1`,`ϕ_2`)`

図 5.1　LTL 式の構文

(*Proposition*) P ::=
`true` | `false` | *Predicate* | `not(`P`)` | `and(`P_1`,`P_2`)` | `or(`P_1`,`P_2`)` | `imply(`P_1`,`P_2`)`

図 5.2　命題の構文

トマトンを入力として受け取る．モデルを LTL 式が充足した場合はモデル検査器は`no_acceptance_cycle_exists`アトムを返し，そうでない場合は反例として状態のリストを返す．

5.1.1　仕様記述

本節では LMNtal でどのように Büchi オートマトンを表現するかということについて述べる．Büchi オートマトンの各状態は`int`型の整数で表現される．Büchi オートマトンは次の 5 価のアトムで表現される．

$$Ret = \texttt{ba}(S, Delta, S_0, F)$$

S は状態の集合を表す`int`型アトムのリストが接続される．$Delta$ は次のようなアトムのリストが接続される．

$$Ret = \texttt{d}(From, Prop, To)$$

`d`アトムは状態の遷移関係を表すアトムである．S_0 は初期状態を表す`int`型のアトムが接続される．F は受理状態を表す状態のリストが接続される．

`d` アトム $Ret = \texttt{d}(From, Prop, To)$ は状態 $From$ から状態 To へ命題 $Prop$ が満たされている場合に遷移可能であることを表す．$From$ と To には状態表す`int`型アトムが接続され，$Prop$ には図 5.2 に示すような命題を表すグラフ構造が接続される．

図 5.2 中の *Predicate* は状態に関する述語を表すグラフ構造へのハイパーリングである．3 価の`pred`で表現される状態に関する述語とは次のようなものである．

$$Ret = \texttt{pred}(\{Head\}, \{Guard\}).$$

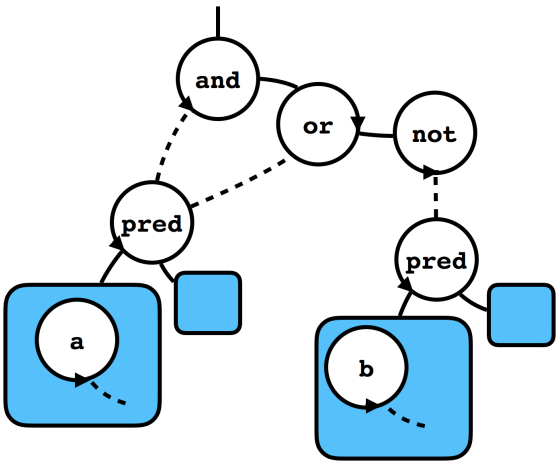

図 5.3　P を”a(X) が存在する”とし，Q を”b(Y) が存在する”とした時の命題 $P \wedge (\neg Q \vee P)$ を表す LMNtal グラフ

状態を表すグラフが $Head$ にマッチする部分グラフを持ち， $Guard$ の制約を充足する場合に，この述語は満たされる．述語のコピーが在る場合には，その述語はハイパーリンクによって共有される．3.3 章で述べた state_space.react_nd_set を用いて述語の充足を判定する．書き換え規則 $Head$:-$Guard$|$Head$ が状態に適用できる場合はその状態は述語 Ret = pred({$Head$},{$Guard$}) を充足する．

例として，述語 P を”a(X) が存在する”とし，述語 Q を”b(Y) が存在する”とする．命題 $P \wedge (\neg Q \vee P)$ を表すグラフ構造を図 5.3 に示す．

5.1.2　実装

本節では LMNtal で実装した LTL モデル検査器について述べる．モデル検査のアルゴリズムは *Nested Depth-First Search*[7] を用いた．

この LTL モデル検査器への入力は下のような mcアトムである．

$$Ret = \texttt{mc}(A, Rs, Init)$$

A には 5.1.1 で述べた baアトムが接続される． Rs には第一級書換規則を含む膜が接続される． $Init$Ini には初期状態を表すグラフが接続される．もし入力された baが表す仕様を Rsで表すモデルが満たさなかった場合は，このモデル検査器は次のような反例を出力する:counterexample($Path$) もしモデルが仕様を満たすならば，$Path$ には反例としての状態のリストが接続され，そうでなければ no_acceptance_cycle_existsアトムが接続

```
dfs1, stack1([['.'($s,$q)|T]|T0]), hash1(H0), on_stack(H1) :- int($s), int($q) |
    dfs1_foreach(succ($s, $q)), stack1([['.'($s, $q)|T]|T0]),
    hash1(set.insert(H0, '.'($s, $q))), on_stack(set.insert(H1,'.'($s,$q))), st_([]).

dfs1, stack1([[], ['.'($s,$q)|T]|T0]) :- int($s), int($q) |
    dfs1_acc($q, [], f), stack1([['.'($s,$q)|T]|T0]).

dfs1, stack1([[]]) :-
    no_acceptance_cycle_exists.

dfs1_acc(t), stack1([['.'($s,$q)|T]|T0]):- int($s), int($q) |
    dfs2, stack2([['.'($s,$q)]]), stack1([['.'($s,$q)|T]|T0]).

dfs1_acc(f) :- dfs1_pop.

dfs1_pop, stack1([['.'($s,$q)|T]|T0]), on_stack(H) :- int($s),int($q) |
    dfs1, stack1([T|T0]), on_stack(set.erase(H,'.'($s,$q))).

dfs1_foreach(['.'($s,$q)|T]), hash1(H0) :- int($s), int($q) |
    dfs1_foreach_inner(['.'($s,$q)|T],Res), hash1(set.find(H0, '.'($s,$q), Res)).

dfs1_foreach([]), st_(St_), stack1(St) :-
    dfs1, stack1([St_|St]).

dfs1_foreach_inner(['.'($s,$q)|T], none), st_(St) :- int($s), int($q) |
    dfs1_foreach(T), st_(['.'($s,$q)|St]).

dfs1_foreach_inner(['.'($s,$q)|T], some) :- int($s), int($q) |
    dfs1_foreach(T).
```

図 5.4　LTL モデル検査器の 1 つ目の DFS

される．LTL モデル検査器が探索する状態空間の状態は次のようなアトムで表される．A

$$Ret = \text{'.'}(Sm, Sa)$$

Smはモデルの状態を表す ID が接続され，Sa はオートマトンの状態を表す ID が接続される．

図 5.4 と図 5.5 に LMNtal で実装した LTL モデル検査器の中核部分のソースコードを示す．LTL モデル検査器全体は 72 本の書き換え規則によって実装されている．状態がモデルの状態とオートマトンの状態の組になっている点や探索アルゴリズムが複雑になって

```
dfs2, stack2([['.'($s,$q)|T]|T0]), hash2(H0) :- int($s), int($q) |
    dfs2_foreach(succ($s,$q)),stack2([['.'($s,$q)|T]|T0]),
    hash2(set.insert(H0, '.'($s,$q))), st_([]).

dfs2, stack2([[]]),  :- dfs1_pop.

dfs2, stack2([[], ['.'($s,$q)|T]|T0]) :- int($s), int($q) |
    dfs2, stack2([T|T0]).

dfs2_foreach(['.'($s,$q)|T]), on_stack(H) :- int($s), int($q) |
    dfs2_foreach_inner0(['.'($s,$q)|T],Res),on_stack(set.find(H,'.'($s,$q),Res)).

dfs2_foreach([]), st_(St_), stack2(St) :-
    dfs2, stack2([St_|St]).

dfs2_foreach_inner0(S, none) :- dfs2_foreach2(S).

dfs2_foreach_inner0(['.'($s,$q)|$g], some) :- int($s), int($q), ground($g) |
   terminate0([]).

dfs2_foreach2(['.'($s,$q)|T]), hash2(H0) :- int($s), int($q) |
    dfs2_foreach_inner(['.'($s,$q)|T],Res), hash2(set.find(H0, '.'($s,$q), Res)).

dfs2_foreach_inner(['.'($s,$q)|T], none), st_(St) :- int($s), int($q) |
    dfs2_foreach(T), st_(['.'($s,$q)|St]).

dfs2_foreach_inner(['.'($s,$q)|T], some) :- int($s), int($q) |
    dfs2_foreach(T).
```

図 5.5　LTL モデル検査器の 2 つ目の DFS

いる点を考慮すると，実装されたモデル検査器は十分小さいと言える．

Nested-DFS は帰りがけ順や反例など，探索順が重要なアルゴリズムである．LMNtal のような再帰関数のない言語で *Nested-DFS* を実装するためにデータ構造を工夫した．`stack1`や `stack2`には隣接頂点のリストを要素とするスタックが接続されており，未展開頂点や頂点の探索順を管理している．スタックの先頭のリストが空リストであれば，先頭の次のリストの先頭頂点に帰りがけ順で訪問したと判定できる．スタックの先頭が空リストでなければ，先頭のリストの先頭頂点を展開する．そのため，スタックに積まれたリストの先頭頂点の列は探索順を示す．`hash1`, `hash2`にはハッシュテーブルが接

続されており，それぞれ 1 つ目の DFS と 2 つ目の DFS で訪問済みの頂点を管理する．on_stackにもハッシュテーブルが接続されているおり，閉路探索を高速化するために 1 つ目の DFS の探索経路上にある頂点を管理している．閉路は 2 つ目の DFS が訪問する頂点が on_stack 上にあるとき発見される．LTL モデル検査器は dfs1アトムを処理する部分と dfs2アトムを処理する部分から構成される．前者は受理状態を訪問するまで状態遷移グラフを構築し，後者はそこから閉路を探索する．遷移先状態はモデルの状態とオートマトンの状態の両方の遷移先状態の組み合わせによって得られる．図 5.4 と図 5.5 には現れていないが succアトムを処理する書き換え規則が API を用いて遷移先の状態を計算する．これらの書き換え規則では state_mapと state_space.react_nd_setが遷移先状態を計算するために使われている．state_space.react_nd_setはモデルの遷移先状態を得るためと Büchi オートマトンの論理式が充足されているかの検査に用いられる．オートマトンの遷移先状態はオートマトンの遷移が存在し，かつ遷移元状態が遷移に付随する命題を充足している場合に，得られる．状態は 4.2 章で示したように state_mapによって管理される．

表 5.1 に SLIM と LMNtal で実装された LTL モデル検査器の実行時間を示す．本実験で用いた仕様記述は safety, recurrence, responseなどの性質を含んでいる．受理状態の無い例題では定数倍になっているが，他の例題では SLIM によりも LMNtal 実装の方が非常に実行時間が大きくなっている例題も在る．これはそれぞれのモデル検査器の Nested-DFSが異なる順序で状態空間を探索し，異なる反例を見つけていることが原因として考えられる．

5.1.3　例題:食事する哲学者

食事する哲学者の問題とはプロセスの同期に関する問題である．哲学者達は円卓に座って思考したり食事したりする．それぞれの哲学者の両脇にはフォークが置いてあり，哲学者は 2 つのフォークを手に取ることで食事することができる．ただし哲学者はまず左のフォークを取ってから右のフォークを取る．食事が終わると哲学者は元あった場所にフォークを戻す．この食事する哲学者のモデルに対して “”哲学者は食事を行う”ことが無限回発生する” ということを検証する．図 5.6 で第一級書き換え規則を用いたモデル記述と初期状態のグラフを示す．このグラフ中のアトムは哲学とフォークに対応する．これらすべてのアトムとリンクは環状のグラフを構成するこの方法では LMNtal は自然な形でモデルの持つ対称性を扱うことができている．LMNtal は対称なグラフ構造を同一視することによってモデルの状態空間を小さくすることができる．5 人の食事する哲学

表 5.1　LTL モデル検査器の性能

Instance Name	Results	States (SLIM)	Time(s) (SLIM)	States (Meta)	Time(s) (Meta)	Ratio
Byzantine_10	counterexample	557	0.11	1434	4.35	39.5
Byzantine_11	counterexample	688	0.12	1918	6.95	57.9
Byzantine_12	counterexample	833	0.14	2500	12.08	86.3
Mutex_10	no accepting cycle	6144	0.82	6144	9.81	11.96
Mutex_11	no accepting cycle	13312	1.83	13312	23.38	12.77
Mutex_12	no accepting cycle	28672	4.71	28672	56.67	12.03
PhiM_5	counterexample	332	0.07	65	0.03	0.43
PhiM_6	counterexample	665	0.10	93	0.04	0.40
PhiM_7	counterexample	2073	0.20	126	0.05	0.25
Rabbit_8	counterexample	1612	0.15	1457	1.71	11.4
Rabbit_9	counterexample	3268	0.25	2857	3.63	14.54
Rabbit_10	counterexample	6839	0.50	5831	8.77	17.54

者の問題ではそれぞれの哲学者を区別せずにモデリングした場合は 18 状態であるが，それぞれに固有の名前をつけてモデリングした場合は状態数 82 になる．命題 p はプロセス `p_eating(L,R)` が存在することを表すとする．LTL 式 $\Box\Diamond p$ はモデル中で “”哲学者が食事する”ことが無限回発生する” ことを表現する．$\Box\Diamond p$ の否定を受理する Büchi オートマトンのグラフ表現は以下のようになる．

```
ba([0, 1],
   [d(0,true,0), d(0,not(!P),1), d(1,not(!P),1)],
   0,
   [1]),
pred({p_eating(!X,!Y)}, {}, !P).
```

図 5.6 のモデルと LTL 式 $\Box\Diamond p$ のグラフ表現をモデル検査器に入力すると，反例としてすべての哲学者が左手にフォークを持つまでの状態列が返される．反例を図 5.6 に示す．

```
// system rules
':-'({p_thinking(!Lx0, !Rx0), fork_free(!Rx1, !Lx0)},
     {},
     {p_one_fork(!Lx0, !Rx0), fork_used(!Rx1, !Lx0)}),
':-'({p_one_fork(!Lxx0, !Rxx0), fork_used(!Rxx1, !Lxx0), fork_free(!Rxx0, !Lxx1)},
     {},
     {p_eating(!Lxx0, !Rxx0), fork_used(!Rxx1, !Lxx0), fork_used(!Rxx0, !Lxx1)}),
':-'({p_eating(!Lxxx0, !Rxxx0), fork_used(!Rxxx0, !Lxxx1), fork_used(!Rxxx1, !Lxxx0)},
     {},
     {p_thinking(!Lxxx0, !Rxxx0), fork_free(!Rxxx0, !Lxxx1), fork_free(!Rxxx1, !Lxxx0)}),
// init state
p_thinking(L0, R0), fork_free(R0, L1).
p_thinking(L1, R1), fork_free(R1, L2).
p_thinking(L2, R2), fork_free(R2, L0).
```

図 5.6　食事する哲学者の LMNtal によるモデリング

```
counterexample([
  '.'({p_thinking(fork_free(p_thinking(fork_free(p_thinking(fork_free(L8))))),L8)},  0),
  '.'({p_thinking(fork_free(p_one_fork(fork_used(p_thinking(fork_free(L9))))),L9)},  1),
  '.'({p_thinking(fork_free(p_one_fork(fork_used(p_one_fork(fork_used(L10))))),L10)},1),
  '.'({p_one_fork(fork_used(p_one_fork(fork_used(p_one_fork(fork_used(L11))))),L11)},1),
  '.'({p_one_fork(fork_used(p_one_fork(fork_used(p_one_fork(fork_used(L12))))),L12)},1)
              ])
```

図 5.7　食事する哲学者問題の反例

5.1.4　拡張:深さ制限付き探索

図 5.4 と図 5.5 に示した Nested-DFS を深さ制限が可能な Nested-DFS に拡張を行った．状態の深さを初期状態からその状態に至るまでの最小の遷移数として定義する．Nested-DFS に用いるスタックと訪問済みノードのためのハッシュテーブルの要素に深さのためのパラメータを追加した．深さの値は新状態を展開する時に増え，既に存在する状態へのより短い経路が発見される時に減る．新状態は与えられた深さの制限値よりもその状態の深さが小さい値だった場合に，展開されるが，そうでなければモデル検査器はその経路の探索をやめる．深さ制限付きの探索を基礎にして反復深化の深さ探索を行う LTL モデル検査器を実装することも可能である．

5.2　CTL モデル検査

CTL モデル検査とは計算木論理 (CTL) で記述された仕様を状態遷移システムが満たすかどうかを検証する手法である．CTL モデル検査器はシステムの初期状態が与えられた CTL 論理式を充足する状態の集合に含まれる場合に真を返し，そうでなければ偽を返す．

CTL 論理式のグラフ表現を図 5.8 に示す．CTL 式における様相演算子はパス量化子 E，A と時相演算子 F，G，U の組から成る．E は CTL 式を満たすパスが存在することを意味し，A はすべてのパスで CTL 式が満たされることを意味する．例えば，$EG\phi$ はすべての状態で ϕ が充足されるようなパスが存在することを意味する．

CTL モデル検査は入力された CTL 式を ϕ として，ϕ を充足する状態の集合 S_ϕ を求める問題に帰着される．S_ϕ は ϕ の構造について帰納的に計算することが可能である [5]．例えば，p を命題，ϕ を CTL 式 $EX\neg p$ とする．まず，モデルの全状態の集合 S から $S_p = \{s \in S \mid s \models p\}$ を求める．次に $S_{\neg p}$ を求め，$S_{\neg p}$ から次のように $S_{EX\neg p}$ を求める．

$$S_{EX\neg p} = \{s \in S \mid \text{there is a transition from } s \text{ to } s' \in S_{\neg p}\}.$$

5.2.1　実装

本研究では LMNtal で記述されたモデルのための CTL モデル検査器を LMNtal で実装した．実装したモデル検査器のアルゴリズムは [4] に基づいている．モデル検査器は第一級書き換え規則とモデルの初期状態，仕様を記述した CTL 論理式を入力として受け取る．CTL モデル検査器の入力は以下のアトムで表現される．

$$Ret = \texttt{mc(ctl(}Ctl\texttt{)}, Rs, Init\texttt{)}$$

Rs には第一級書換規則を含む膜が接続される．$Init$ には初期状態を表すグラフが接続される．Ctl には図 5.8 で示した CTL 式のグラフ表現が接続される．

LMNtal で実装した CTL モデル検査器の中核部を図 5.9 に示す．CTL モデル検査器はメタインタプリタの自然な発展として実装を行った．モデル検査器は 105 本の書き換え規則で構成され，それらは図 4.1 で示した 11 本のメタインタプリタと CTL の演算子それぞれについて充足する集合を計算するための 94 本である．

書き換え規則 `finish`は入力された CTL 式を満たす状態集合に初期状態が含まれているか確認する．もし，状態集合に初期状態が含まれていればモデル検査器は `true`アトム

(*CTL Formula*) ϕ ::=
`true` | `false` | `p(`*Predicate*`)` | `not(`ϕ`)` | `or(`ϕ_1`,`ϕ_2`)` | `and(`ϕ_1`,`ϕ_2`)` | `imply(`ϕ_1`,`ϕ_2`)`
| `ax(`ϕ`)` | `ex(`ϕ`)` | `ag(`ϕ`)` | `eg(`ϕ`)` | `af(`ϕ`)` | `ef(`ϕ`)` | `au(`ϕ_1`,`ϕ_2`)` | `eu(`ϕ_1`,`ϕ_2`)`

図 5.8　CTL 論理式のグラフ表現

を返し，そうでなければ `false`アトムを返す．書き換え規則 `pred`は入力された述語を充足する状態の集合を求める．`s_`から始まる名前のアトムは CTL の演算子それぞれについて，充足する状態集合の計算を意味する．それ以外の書き換え規則は論理演算子に対応する．例えば，入力された CTL 式が $\phi_1 \vee \phi_2$ だった場合は，書き換え規則 `or`は状態集合 S_{ϕ_1} と S_{ϕ_2} を受け取って $S_{\phi_1 \vee \phi_2} = S_{\phi_1} \cup S_{\phi_2}$ を計算する．$S_{EG\phi}$ の計算は書き換え規則 `eg`から始まるが，その計算過程は少々複雑である．有限の状態遷移グラフで ϕ を満たす状態が無限回現れるためには，その状態は状態遷移グラフ中の強連結成分に含まれなければならない．つまり，$s \models EG\phi$ は状態遷移グラフ中に強連結成分が存在し，状態 s から強連結成分中の状態への有限の実行パスが存在することを意味する．そのため，$S_{EG\phi}$ を求める処理では状態遷移グラフの強連結成分を計算し，強連結成分中の状態に到達可能な状態を求めている．

5.2.2　例題: 電子レンジ

電子レンジの制御モデルの例を図 5.10 に示す．各円に書かれたラベルは各状態で成り立つ命題を表している．ラベルの接頭辞 “∼” は否定を意味する．このモデルに対する CTL 式 `ag(ef(p(`*Init*`)))` の検証を考える．`ag(ef(p(`*Init*`)))` は “任意の状態から初期状態へ到達可能である” ということを表している．なおこのモデルは [5] の例題に対して，その全状態に *Init* のラベルを足して拡張したものである．

電子レンジのモデルの第一級書き換え規則としてモデル記述と初期状態を図 5.11 に示す．CTL 式 `ag(ef(p(`*Init*`)))` は式

`not(eu(true, not(eu(true, p(`*Init*`)))))`

と等価である．図 5.11 に示したモデルと式 `not(eu(true, not(eu(true,p(`*Init*`)))))` をモデル検査器に与えて実行すると，真を返す．図 5.10 から確かに *Init* を満たす状態 3は任意の状態から到達可能である．よって CTL モデル検査器はこの例題に対して正しく動作していることがわかる．

```
finish@@Ret = mc1(Ini, ctl(sat(S)), SS) :-
    Ret = result(set.find(S, Ini, Res), Res, SS).

true@@Ret = mc1(Ini, Ctl, state_space(M, S, T)), Ret_=true :-
    Ret = mc1(Ini, Ctl, state_space(M, set.copy(S, S_), T)), Ret_=sat(S_).

p@@Ret = mc1(Ini, Ctl, state_space(M, S, T)), R = p($x), pred({$h[]}, {$g[]}, $y):-
    hlink($x), hlink($y), $x==$y |
    Ret = mc_(Ini, Ctl),
    R = s_p(set.init, set.to_list(S_), {':-'({$h[]}, {$g[]}, {$h[]})},
        state_space(M, set.copy(S, S_), T)), pred({$h[]}, {$g[]}, $y).

not@@Ret = mc1(Ini, Ctl, state_space(M, S, T)), Ret_=not(sat(S_)) :-
    Ret = mc_(Ini, Ctl, state_space(M, set.copy(S, R), T)),
    Ret_=s_not(set.diff(R, R_), set.copy(S_, R_)).

or@@Ret = mc1(Ini, Ctl, SS), Ret_=or(sat(S0), sat(S1)) :-
    Ret = mc1(Ini, Ctl, SS), Ret_=sat(set.union(S0, S1)).

ex@@Ret = mc1(Ini, Ctl, state_space(M, S, T)), Ret_ = ex(sat(S_)) :-
    Ret = mc_(Ini, Ctl),
    Ret_ = s_ex(set.init, set.to_list(T_), S_, state_space(M, S, set.copy(T, T_))).

eu@@Ret = mc1(Ini, Ctl, state_space(M, S, T)), Ret_=eu(sat(S0), sat(S1)) :-
    Ret = mc_(Ini, Ctl),
    Ret_=s_eu(set.copy(S1, S1_), S0, set.to_list(S1_), set.to_list(T_),
              state_space(M, S, set.copy(T, T_))).

eg@@Ret=mc1(Ini, Ctl, state_space(M, S, T)), Ret_=eg(sat(S0)) :-
    Ret = mc_(Ini, Ctl, state_space(M, S, set.copy(T, T_))),
    Ret_=s_eg0(S0, set.to_list(T_), [], []).
```

図 5.9　CTL モデル検査器の中核部

表 5.2 に LMNtal で実装した CTL モデル検査器の実行時間を示す．`Mutex` の例題では LTL モデル検査器よりもよい性能が出ている．*set* ライブラリに加えて *hashmap* ライブラリを用いて状態遷移グラフを管理しているため，状態遷移の管理をより効率的に実装することが可能になっている．一方で他の例題では性能が悪い．これは述語の充足判定部分の実装に `react_nd_set` を用いていることに起因していると考えられる．

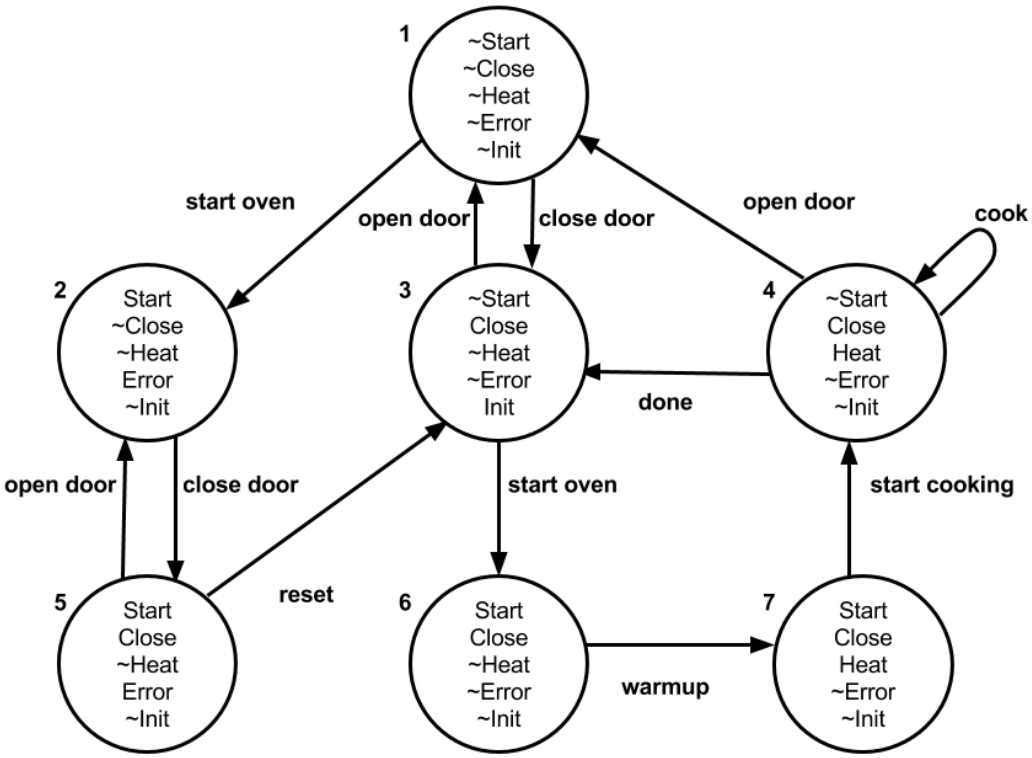

図 5.10　電子レンジの状態遷移グラフ

5.2.3　例題:公平性

実装した CTL モデル検査器を公平性を取り扱うように容易に拡張を行うことができる．公平性制約は “一人の哲学者が食事し続ける” というような非現実的なパスを制限することができる．つまり，拡張されたモデル検査器は公平性制約を満たすパスのみを検査対象とする CTL モデル検査器である．公平性制約 FC は次のようなリストで表現される

$$Ret = \texttt{[pred\_fair}(\{Head\}, \{Guard\}), \ldots\texttt{]}$$

LTL モデル検査器で用いた `pred`アトムと同様に，この `pred_fair`アトムも状態に対する述語を表す．公平性制約 FC を充足するパスとは FC のリストのすべての `pred_fair`を満たすような状態が無限回現れるようなパスである．

CTL モデル検査器から大きく拡張されたのは $S_{EG\phi}$ を計算する処理である．*fair scc*

```
//system rules
':-'({-start, -close, -heat, -error, -init}, {}, {start, -close, -heat, error, -init}),
':-'({-start, -close, -heat, -error, -init}, {}, {-start, close, -heat, -error, init}),
':-'({start, -close, -heat, error, -init}, {}, {start, close, -heat, error, -init}),
':-'({-start, close, -heat, -error, init}, {}, {-start, -close, -heat, -error, -init}),
':-'({-start, close, -heat, -error, init}, {}, {start, close, -heat, -error, -init}),
':-'({-start, close, heat, -error, -init}, {}, {-start, -close, -heat, -error, -init}),
':-'({-start, close, heat, -error, -init}, {}, {-start, close, heat, -error, -init}),
':-'({-start, close, heat, -error, -init}, {}, {-start, close, -heat, -error, init}),
':-'({start, close, -heat, error, -init}, {}, {start, -close, -heat, error, -init}),
':-'({start, close, -heat, error, -init}, {}, {-start, close, -heat, -error, init}),
':-'({start, close, -heat, -error, -init}, {}, {start, close, heat, -error, -init}),
':-'({start, close, heat, -error, -init}, {}, {-start, close, heat, -error, -init}),
//init state
-start, close, -heat, -error, init
```

図 5.11　LMNtal による電子レンジのモデル記述と初期状態

表 5.2　CTL モデル検査器の性能評価

Instance Name	States	Property	Results	Time(s)
Byzantine_3	420	recurrence	false	0.49
Mutex_10	6144	safety	true	6.60
Mutex_11	13312	safety	true	15.67
Mutex_12	28672	safety	true	37.90
PhiM_4	327	response	false	0.75
PhiM_5	1370	response	false	3.67
PhiM_6	5785	response	false	20.52
Rabbit_4	200	safety	false	1.35
Rabbit_5	482	safety	false	6.69
Rabbit_6	1096	safety	false	32.77

を，FC 中の各述語を充足する状態を含む強連結成分とする．$S_{EG\phi}$ の計算では *fair scc* でない強連結成分を除外する．exと euの書き換え規則も拡張を行った．基礎にした CTL モデル検査器では exは $S_{EX\phi}$ を計算するために S_ϕ を受け取る．一方で拡張した CTL モデル検査器では exに渡される集合に属する状態は状態遷移グラフの *fair scc* に含まれていなければならない．したがって exは状態集合 $S_\phi \cap \{s \mid s \in fair\ scc\}$ を受け取るよう

(*TCTL Formula*) ::=
　　CTL Formula | au(ϕ_1,ϕ_2,J) | eu(ϕ_1,ϕ_2,J)
ϕ ::= true | false | p(*Predicate*) | g
g ::= '<'(x,c) | '=<'(x,c) | '>'(x,c) | '>='(x,c) | '=:='(x,c) | and_g(g,g) | t_
J ::= '<'(c) | '=<'(c) | '>'(c) | '>='(c) | and_j(J,J) | t_

図 5.12　TCTL 論理式のグラフ表現

に拡張した．

5.3　時間オートマトンに対する TCTL モデル検査

時間オートマトンに対する TCTL モデル検査とは時間計算機論理 (TCTL) で記述された仕様を，時間オートマトンで記述されたモデルが満たすかどうかを検証する手法である．TCTL モデル検査器はモデルの初期状態が与えられた TCTL 論理式を充足する状態の集合に含まれる場合に真を返し，そうでなければ偽を返す．本研究では，記述に用いる時間オートマトンはクロック変数が 1 種類のみ使用可能かつ制限された TCTL 論理式を入力とした TCTL モデル検査器を LMNtal で実装した．以下で述べる TCTL モデル検査の手法は [2] で紹介されている手法を基礎にしている．

TCTL 論理式のグラフ表現を図 5.12 に示す．図 5.12 中の c は任意の整数アトムを表す．また，g および J の t_は制約が真であることを表すアトムであり，ϕ の true と区別している理由は単に実装上の理由である．TCTL 式は CTL 式の他に au(ϕ_1 , ϕ_2 , J) と eu(ϕ_1 , ϕ_2 , J) を使用することができる．au(ϕ_1 , ϕ_2 , J) はすべてのパスで J 以内 (以降) に ϕ_2 が成立し，それまでは ϕ_1 が成立することを表す．eu(ϕ_1 , ϕ_2 , J) は J 以内 (以降) に ϕ_2 が成立し，それまでは ϕ_1 が成立するパスが存在することを表す．TCTL 式はそれと等価な CTL 式に変換が可能である．変換では CTL に現れない au と eu について下記のような変換を行う．

- au(ϕ_1 , ϕ_2 , J) → au(or(ϕ_1 ,ϕ_2),and(ϕ_2 ,zcc(J)))
- eu(ϕ_1 , ϕ_2 , J) → eu(or(ϕ_1 ,ϕ_2),and(ϕ_2 ,zcc(J)))

変換の際に時間オートマトンにシステム全体の時間経過を観測するためのクロック変数 z

を導入する．変換後の CTL 式に現れる zcc は時間オートマトンに導入されたクロック変数 z に関する述語を表す．クロック変数 z の値が J で表現される時間領域内に入っていれば真，そうでなければ偽として評価される．

TCTL モデル検査は領域状態遷移システムと呼ばれる特殊な状態遷移グラフに対する CTL モデル検査を行うことで達成される．領域状態遷移システムの状態とは時間オートマトンの状態とクロック変数の領域との組で表現される．領域とはクロック変数に対する制約式である．領域はクロック変数の値が制約式を満たす範囲内であることを表し，一種の抽象化を提供する．クロック変数は実数をドメインとしているため，クロック変数の値を状態として含めた場合，状態数は簡単に無限になってしまい，検査が行えない．領域という概念を導入することによって，本来無限になってしまう状態空間でも有限の状態空間に抽象化することが可能となる．詳しくは [2] を参照されたい．領域状態遷移システムの状態は次のようなグラフ構造で表現する．s({St}, $region$)
St は時間オートマトンの状態を表すグラフ構造である．$region$ は以下に示す領域を表す 9 価の r アトムである．

$$r(lb_x, ub_x, op_x, lb_z, ub_z, op_z, i, j, op'_z)$$

ただし，$lb_x, ub_x, lb_z, ub_z, i, j$ は整数は整数アトムとし，op_x, op_z, op'_z は $'<'$ か $'=<'$ のどちらかのアトムとする．このように定義された $region$ は次のような制約式を表現する．

$$lb_x\ op_x\ x\ op_x\ ub_x\ \wedge\ lb_z\ op_z\ z\ op_z\ ub_z\ \wedge\ x+i\ op'_z\ z\ op'_z\ x+j$$

例えば r(0, 1, '<', 0, 1, '<', -1, 0, '<') は $0 < x < 1 \wedge 0 < z < 1 \wedge x - 1 < z < x$ という制約式を表す．なお，x は時間オートマトンのクロック変数，z は TCTL 式を CTL 式に変換した際に時間オートマトンに導入されるクロック変数を表す．領域状態遷移システムの状態遷移には 2 種類の遷移が存在する．1 つ目は時間オートマトンの状態遷移による状態遷移である．時間オートマトンの遷移規則に従った状態の遷移によって領域状態遷移システムの状態も遷移する．2 つ目は領域の遷移による状態遷移である．クロック変数は時間経過を表すため，その値は常に変化する．ある領域内にあったクロック変数の値が別の領域内に入った時，この領域の遷移が発生する．このような 2 種類の状態遷移によって非決定的に状態が遷移してゆくシステムが領域状態遷移システムである．

次に LMNtal による時間オートマトンの記述方法について述べる．時間オートマトンは次のようなグラフ構造で表現される．

```
ta({‘$Rule$\verb‘}, {‘$Inv$\verb‘}, {‘$Ini$\verb‘}, ‘$c_{max}$\verb‘)
```

Inv はインバリアント関数を表す次のようなグラフ構造が入った膜であるである．

```
inv({‘ $Graph$ \verb‘}, ‘ $Guard$\verb‘)
```

Graph はグラフ構造であり，*Graph* が表す状態で *Guard* が成立することを表す．*Guard* には図 5.12 中の *g* が接続される．*Rule* は時間オートマトンの遷移規則を表現する以下のようなグラフである．

```
’:-’({‘$Src$\verb‘},{‘$Reset$\verb‘},‘$Guard$\verb‘,{‘$Dst$\verb‘})
```

Src には遷移元状態のグラフ構造，*Dst* には遷移先状態のグラフ構造，*Guard* には図 5.12 中の *g* が接続され，*Reset* には `reset` アトムが入った膜か空の膜かが接続される．時間オートマトンの遷移規則は *Guard* を満たしかつ，*Dst* でインバリアントが真である時に *Src* から *Dst* へ状態遷移可能という意味である．なお，*Ini* は初期状態，c_{max} は TCTL 式および時間オートマトン中の *Guard* に現れる整数 *c* の最大値である．

5.3.1　実装

実装した TCTL モデル検査器は時間オートマトンによるモデル記述と TCTL 論理式を入力として受け取る．TCTL モデル検査器の入力は以下のアトムで表現される．

$$Ret = \texttt{mc(tctl(}Tctl\texttt{)},Ta\texttt{)}$$

Tctl には図 5.12 で示した TCTL 式のグラフ表現が接続される．*Ta* には LMNtal による時間オートマトンの表現である `ta` アトムが接続される．LMNtal で実装した TCTL モデル検査器の中核部を図 5.14 に示す．この部分は領域状態遷移システムの状態遷移グラフを深さ優先探索で構築する部分である．すなわち，基本構造はメタインタプリタと同じである．`run1` はスタックが空かどうかを確認して空なら状態遷移グラフを出力し，そうでなければ先頭の状態を pop する．`run2` では `succ_disc` が呼び出されている．`succ_disc` は時間オートマトンの状態遷移による遷移先状態を求める処理である．`succ_disc` では’:-’({*Src*},{*Reset*},*Guard*,{*Dst*}) から:-({*Src*},{},{*Dst*}) のように第一級書き換え規則を生成して遷移先状態を取得している．`run3` では領域の遷移による遷移先状態を求める `succ_cont` が呼び出されている．`succ_disc` と `succ_cont` によって求められたすべての遷移先状態について新状態かどうか，新遷移かどうかを判定するループが `run4` から始まる．新状態および新遷移の判定はメタインタプリタ同様に `state_map` コレクションとハッシュテーブルを用いて行っている．

5.3.2　例題:時間制限付きのスイッチの切り替え

例として，図 5.3.2 に示すような時間制限付きのスイッチの切り替えモデルを表す時間オートマトンを考える．オートマトンの状態は `off` と `on` があり，初期状態は `off` である．`x` はクロック変数であり，`x=1` を満たす時，状態から `off` から `on` へ遷移することが可能である．またこの時 `x` の値を 0 にリセットする．状態 `on` では `x`$\leq$ 1 を満たす限りは状態遷移をしなくても良い．`off` に遷移する際も `x=1` を満たしていなければならない．なおこの例題は [2] から引用したものである．このような時間オートマトンに対して `eu(true, p(!Q), '=<'(1)), pred({on}, {}, !Q)` という “1 秒以内に `on` になるようなパスが存在する” ことを意味する TCTL 式のモデル検査を行う．その場合時間オートマトンの LMNtal グラフによる表現は以下のようになる．

```
ta({
   ':-'({off}, {reset}, '=:='(x, 1), {on}),
   ':-'({on}, {}, '=:='(x, 1), {off})
   },
   {inv({on}, '=<'(x, 1))},
   {off},
   1))
```

実際にこれらの入力を与えてモデル検査を行うと真を返す．確かに初期状態から丁度 1 秒たった時に状態 `off` から状態 `on` へ遷移可能である．

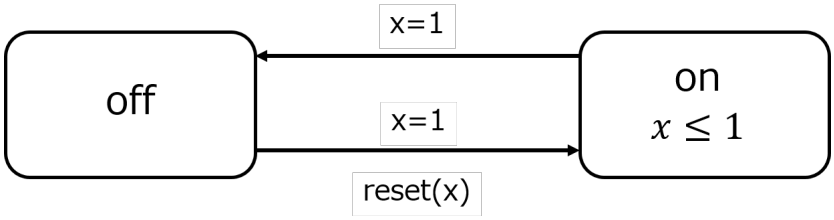

図 5.13　時間制限付きスイッチの切り替えモデル

```
Ret=run1(Ini,Rs,Inv,C,M,S,T,[St|Stk]):-
 Ret=run2(Ini,Rs,St,Inv,C,M,S,T,Stk).

Ret=run1(Ini,{$rs[]},{$inv[]},$c,M,S,T,[]):-int($c)|
 Ret=state_space(Ini,M,S,T).

Ret=run2(Ini,Rs,St,Inv,$c,M,S,T,Stk):-int($c)|
 Ret=run3(Ini,RetRs,RetSt,RetInv,$c,M,S,T,Stk,
  succ_disc(Rs,St,Inv,RetRs,RetSt,RetInv,$c)).

Ret=run3(Ini,Rs,St,Inv,$c,M,S,T,Stk,Succ):-int($c)|
 Ret=run4(Ini,Rs,RetSt,RetInv,$c,M,S,T,Stk,Succ,
  succ_cont(St,Inv,$c,RetSt,RetInv)).

Ret=run4(Ini,Rs,St,Inv,C,M,S,T,Stk,Succ,[]):-
 Ret=run5(Ini,Rs,St,Inv,C,M,S,T,Stk,Succ).

Ret=run4(Ini,Rs,St,Inv,C,M,S,T,Stk,Succ,[Succr]):-
 Ret=run5(Ini,Rs,St,Inv,C,M,S,T,Stk,[Succr|Succ]).

Ret=run5(Ini,Rs,s($id,{$s[]},r($lbx,$ubx,$opx,$lby,$uby,$opy,$i,$j,$opy_)),Inv,C,M,S,T,Stk,[]):-
 int($id),int($lbx),int($ubx),unary($opx),int($lby),int($uby),unary($opy),int($i),int($j),unary($opy_)|
 Ret=run1(Ini,Rs,Inv,C,M,S,T,Stk).

Ret=run5(Ini,Rs,Src,Inv,C,M,S,T,Stk,[ss({$s[]},r($lbx,$ubx,$opx,$lby,$uby,$opy,$i,$j,$opy_))|Succ]):-
 int($lbx),int($ubx),unary($opx),int($lby),int($uby),unary($opy),int($i),int($j),unary($opy_)|
 Ret=run6(Ini,Rs,Src,s(ID,{$s[]},r($lbx,$ubx,$opx,$lby,$uby,$opy,$i,$j,$opy_)),Inv,C,
  state_space.state_map_find(M,{ss({$s[]},r($lbx,$ubx,$opx,$lby,$uby,$opy,$i,$j,$opy_))},ID),S,T,Stk,Succ).

Ret=run6(Ini,Rs,Src,s($id,Dst,R),Inv,C,M,S,T,Stk,Succ):-int($id)|
 Ret=run7(Ini,Rs,Src,s($id,Dst,R),Res,Inv,C,M,set.find(S,$id,Res),T,Stk,Succ).

Ret=run7(Ini,Rs,s($id_s,Src,SR),s($id_d,Dst,DR),none,Inv,C,M,S,T,Stk,Succ):-
 int($id_s),int($id_d)|
 Ret=run5(Ini,Rs,s($id_s,Src,SR),Inv,C,M,
  set.insert(S,$id_d),set.insert(T,'.'($id_s,$id_d)),[s($id_d,Dst,DR)|Stk],Succ).

Ret=run7(Ini,Rs,s($id_s,Src,SR),s($id_d,Dst,DR),some,Inv,C,M,S,T,Stk,Succ):-
 int($id_s),int($id_d)|
 Ret=run8(Ini,Rs,s($id_s,Src,SR),s($id_d,Dst,DR),Res,Inv,C,M,S,set.find(T,'.'($id_s,$id_d),Res),Stk,Succ).

Ret=run8(Ini,Rs,s($id_s,Src,SR),
 s($id_d,{$dst[]},r($lbx,$ubx,$opx,$lby,$uby,$opy,$i,$j,$opy_)),none,Inv,C,M,S,T,Stk,Succ):-
 int($id_s),int($id_d),int($lbx),int($ubx),unary($opx),int($lby),int($uby),
  unary($opy),int($i),int($j),unary($opy_)|
 Ret=run5(Ini,Rs,s($id_s,Src,SR),Inv,C,M,S,set.insert(T,'.'($id_s,$id_d)),Stk,Succ).

Ret=run8(Ini,Rs,Src,s($id_d,{$dst[]},r($lbx,$ubx,$opx,$lby,$uby,$opy,$i,$j,$opy_)),
 some,Inv,C,M,S,T,Stk,Succ):-
 int($id_d),int($lbx),int($ubx),unary($opx),int($lby),int($uby),unary($opy),int($i),int($j),unary($opy_)|
 Ret=run5(Ini,Rs,Src,Inv,C,M,S,T,Stk,Succ).
```

図 5.14　TCTL モデル検査器の中核部

第 6 章

関連研究

モデリング言語と実装言語が同じであるようなモデル検査器は多く存在する．Java バイトコードのための Java Pathfinder や C、C++ のための CBMC などが在る．しかし，本研究と近い手法を用いているモデル検査器は以下で述べる宣言的言語のためのモデル検査器である．

XMC は tabled logic プログラミング言語 XSB で実装されたプロセス計算のためのモデル検査器である [8]．このモデル検査器はトップダウンインタプリタを基礎に XSB の tabling 機構を状態空間管理のために用いている．XMC はインタプリタを用いた手法のため完結で tabling のための最適化されたインデックス方法によって効率的である．そのようなインデックス手法を本研究で実装したグラフのための状態空間管理機能にどのように取り入れるかは自明ではない．モデル検査に tabled logic プログラミングを用いる手法の経験が [13] にまとめられており，そこではインタプリタを用いる手法と CTL モデル検査との親和性が指摘されている．検証のための宣言的言語の使用に関しては [12] に簡潔にまとめられている．

XSB で実装された CTL モデル検査器と比較して本研究で実装した CTL モデル検査器のソースコードは大きい．本研究の実装では明示的に状態空間を探索するのに対して，XSB では Prolog のバックトラックを用いているためである．筆者の知る限りでは tabling の手法は API を通してプログラマに第一級のオブジェクトとして提供されていない．本研究では状態空間を明示的に構築しているが，グラフ同型性判定のような複雑で効率的なアルゴリズムの実装をプログラマに公開していることはそれ自体有用であるからである．LMNtal と Prolog の他の違いとしては LMNtal は本質的に並行言語であるため様々なモデルが簡潔に記述可能であるが，Prolog では並行性のあるモデルの記述はインターリービングが必要である点が挙げられる．

McErlang[9] は Erlang で実装された Erlang のためのモデル検査器である．Erlang では関数は第一級のオブジェクトであるため，Erlang プログラムから関数の操作は柔軟に行える．しかし，Erlang の関数はデータ構造で表現されていないため，変更したり拡張したりすることが難しい．一方本研究で拡張した LMNtal では書き換え規則は第一級のデータ構造で表現されているため，動的に変更することが柔軟に行える．

Maude は [6] 本格的なメタプログラミングのための機構を備えた項書換え系であるが，LMNtal と異なり Maude は大量の記述が必要になる．Maude ではこの記述によって性能を向上させることができるがソースコードは肥大化してしまう．

OPEN/CAESAR[10] はモデル検査器の実装に必要な様々機能を共有するための検証とテストのプラットフォームを提供する．OPEN/CAESAR のコールバックを基礎にした設計は高いモジュール性と直交性を実現しているが，フレームワーク全体は C のようなメインストリームが中心であるため状態は固定長バイトストリングで表現されている．対照的に本研究の手法は，動的に変化するグラフという非常に複雑なデータ構造のサポートを提供している．

第 7 章

まとめと今後の課題

7.1　まとめ

本研究では様々なモデル検査器のプロトタイピングを可能にするためのメタプログラミングのフレームワークを提案した．フレームワークはメタプログラミングのために設計された第一級の書き換え規則と SLIM の内部機能をプログラムから利用するための API から成る．フレームワークを用いて実装した状態遷移グラフを出力するメタインタプリタは SLIM と比較して 10 倍以内のオーバーヘッドで動作することが確認できた．また，状態遷移グラフに対する柔軟な操作を用いて様々なモデル検査器を実装および拡張することに成功した．これにより，本研究で設計したフレームワークはモデル検査器を実装するための本質部分を抽出していると言えるだろう．

7.2　今後の課題

今後の課題としては，メタインタプリタを用いて実装したモデル検査器の並列化が挙げられる．本研究の基礎になったモデル検査器 SLIM はスケーラブルなマルチコアモデル検査器であるが，状態空間生成のための API は並行アクセスができない．API のスレッドセーフな実装が達成できたとしても，LMNtal で複雑な並列モデル検査器のアルゴリズムを簡潔に実装することは自明ではない．このようにモデル検査器の並列化はチャレンジングな課題であるものの，非常に有益であるため今後の課題として取り組みたい．

謝辞

本研究を進めるにあたり様々な方の指導，助言をいただきました． まず, ご指導を賜わった上田和紀教授に深く感謝致します．自分のやりたい研究を自由にさせて頂きましたし，沢山のチャンスを与えてくださいました．誠にありがとうございました．また，親身になって相談にのっていただいた先輩方や後輩に感謝致します．特に松本先輩には生活面や研究に向かう姿勢についても多くのアドバイスを頂きました．同期の松澤君には同期にしか言えない悩みも沢山聞いて頂きました．後輩の冨岡君は共著者として論文執筆や実装面で大変な助力をして頂きましたし，普段見つけた些細な疑問点についても沢山議論して頂きました．誠に感謝しております．ありがとうございました．最後に，入学から現在に至るまで，金銭面でも精神面でも私を支えてくれた家族 3 人に深く感謝致します．

2018 年 1 月　恒川 雄太郎

参考文献

[1] J. Armstrong. Erlang—a survey of the language and its industrial applications. In *Proc. INAP*, volume 96, 1996.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[3] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 2011.

[4] E. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1982.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[6] M. Clavel, F. Dura, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. *All About Maude—A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer-Verlag, 2007.

[7] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *Proc. CAV 1990*, volume 531 of *LNCS*, pages 233–242. Springer-Verlag, 1991.

[8] B. Cui, Y. Dong, X. Du, K. N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In *Proc. 10th International Symposium on Principles of Declarative Programming (PLILP/ALP'98)*, volume 1490 of *LNCS*, pages 1–20. Springer-Verlag, 1998.

[9] L.-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. *ACM SIGPLAN Notices*, 42(9):125–136, 2007.

[10] H. Garavel. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. In *Proc. TACAS 1998*, volume 1384 of *LNCS*, pages

68–84. Springer-Verlag, 1998.

[11] M. Gocho, T. Hori, and K. Ueda. Evolution of the LMNtal runtime to a parallel model checker. *Computer Software*, 28(4):137–157, 2011.

[12] M. Leuschel. Declarative programming for verification: Lessons and outlook. In *Proc. PPDP'08*, pages 1–7. ACM, 2008.

[13] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In *Proc. LOPSTR'99*, volume 1817 of *LNCS*, pages 62–81. Springer-Verlag, 2000.

[14] A. Rensink. The groove simulator: A tool for state space generation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 479–485. Springer-Verlag, 2003.

[15] Y. Tsunekawa, T. Tomioka, and K. Ueda. Implementation of LMNtal model checkers: a metaprogramming approach. In *Proc. Meta-Programming Techniques and Reflection,* *http://2016.splashcon.org/event/meta2016-implementation-of-lmntal-model-checkers-a-metaprogramming-approach*, 2016.

[16] K. Ueda. LMNtal as a hierarchical logic programming language. *Theoretical Computer Science*, 410(46):4784–4800, 2009.

[17] K. Ueda, T. Ayano, T. Hori, H. Iwasawa, and S. Ogawa. Hierarchical graph rewriting as a unifying tool for analyzing and understanding nondeterministic systems. In *Proc. ICTAC 2009*, volume 5684 of *LNCS*, pages 349–355. Springer-Verlag, 2009.

[18] K. Ueda and S. Ogawa. HyperLMNtal: An extension of a hierarchical graph rewriting model. *Künstliche Intelligenz*, 26(1):27–36, 2012.

[19] 村山 敬, 工藤 晋太郎, 櫻井 健, 水野 謙, 加藤 紀夫, and 上田和紀. 階層グラフ書換え言語 lmntal の処理系. **コンピュータ ソフトウェア**, 25(2):2_47–2_77, 2008.

発表論文

[1] Implementation of LMNtal Model Checkers: a Metaprogramming Approach, Yutaro Tsunekawa, Taichi Tomioka, Kazunori Ueda, Journal of Object Technology special issue for META'16.(28 pages) (掲載予定).

[2] Implementation of LMNtal Model Checkers: a Metaprogramming Approach, Yutaro Tsunekawa, Taichi Tomioka, Kazunori Ueda, Meta-Programming Techniques and Reflection(Meta'16) , Amsterdam, Oct.2016 (8 pages), http://2016.splashcon.org/event/meta2016-implementation-of-lmntal-model-checkers-a-metaprogramming-approach.

[3] グラフ書き換えに基づくモデル記述言語 LMNtal による LMNtal モデル検査器の実装，恒川雄太郎，冨岡太一，上田和紀，日本ソフトウェア科学会第 33 回大会，東北大学，2016. (14 pages).

[4] メタインタプリタを用いた容易に拡張可能なモデル検査器の実装，恒川雄太郎，上田和紀，第 19 回プログラミングおよびプログラミング言語ワークショップ (ポスター発表)，2017.

[5] グラフ書き換え言語 LMNtal における第一級書き換え規則の設計と実装，恒川雄太郎，上田和紀，第 18 回プログラミングおよびプログラミング言語ワークショップ (ポスター発表)，2016.

Appendix

A.1 第一級書き換え規則の実装部

以下に第 3.2.3 節で述べた変換関数 c の実装を示す.

```
 1  struct LinkConnection
 2  {
 3   LmnSAtom atom;
 4   HyperLink *hl;
 5   int link_pos, link_name;
 6  };
 7
 8  int linkconnection_push(Vector *link_connections, LmnSAtom satom, int link_p, HyperLink *
        hl)
 9  {
10    int link_name = vec_num(link_connections);
11    struct LinkConnection *c = LMN_MALLOC(struct LinkConnection);
12    c->atom = satom;
13    c->hl = hl;
14    c->link_pos = link_p;
15    c->link_name = link_name;
16    vec_push(link_connections, (vec_data_t)c);
17    return link_name;
18  }
19
20  int linkconnection_make_linkno(Vector *link_connections, LmnSAtom satom, int link_p)
21  {
22    if(LMN_IS_HL(LMN_SATOM(LMN_SATOM_GET_LINK(satom, link_p)))) {
23      HyperLink *hll = lmn_hyperlink_at_to_hl(LMN_SATOM(LMN_SATOM_GET_LINK(satom, link_p)));
24      HyperLink *p_hl = hll->parent;
25
26      for(int i = 0; i < vec_num(link_connections); i++) {
27        struct LinkConnection *c = (struct LinkConnection *)vec_get(link_connections, i);
28        if(c->hl && lmn_hyperlink_eq_hl(p_hl, c->hl)) {
29          return c->link_name;
30        }
31      }
32      return linkconnection_push(link_connections, NULL, -1, p_hl);
33    }
34
35    for(int i = 0; i < vec_num(link_connections); i++) {
36      struct LinkConnection *c = (struct LinkConnection *)vec_get(link_connections, i);
37      if(c->atom == satom && c->link_pos == link_p) {
38        return c->link_name;
39      }
40    }
41
42    LmnSAtom dst_atom = LMN_SATOM(LMN_SATOM_GET_LINK(satom, link_p));
43
```

```
44    if(LMN_SATOM_GET_FUNCTOR(dst_atom) == LMN_IN_PROXY_FUNCTOR) {
45      LmnSAtom out_proxy = LMN_SATOM(LMN_SATOM_GET_LINK(dst_atom, 0));
46      LmnSAtom atom = LMN_SATOM(LMN_SATOM_GET_LINK(out_proxy, 1));
47      int arity = LMN_FUNCTOR_GET_LINK_NUM(LMN_SATOM_GET_FUNCTOR(atom));
48      for(int i = 0; i < arity; i++) {
49        LmnSAtom linked_atom = LMN_SATOM(LMN_SATOM_GET_LINK(atom, i));
50        if(LMN_SATOM_GET_FUNCTOR(linked_atom) == LMN_OUT_PROXY_FUNCTOR) {
51          LmnSAtom in_proxy = LMN_SATOM(LMN_SATOM_GET_LINK(linked_atom, 0));
52          if(satom == LMN_SATOM(LMN_SATOM_GET_LINK(in_proxy, 1))) {
53            return linkconnection_push(link_connections, atom, i, NULL);
54          }
55        }
56      }
57    }
58    else if(LMN_SATOM_GET_FUNCTOR(dst_atom) == LMN_OUT_PROXY_FUNCTOR) {
59      LmnSAtom in_proxy = LMN_SATOM(LMN_SATOM_GET_LINK(dst_atom, 0));
60      LmnSAtom atom = LMN_SATOM(LMN_SATOM_GET_LINK(in_proxy, 1));
61      return linkconnection_push(link_connections, atom, 0, NULL);
62    }
63
64    int arity = LMN_FUNCTOR_GET_LINK_NUM(LMN_SATOM_GET_FUNCTOR(dst_atom));
65    for(int i = 0; i < arity; i++) {
66      if(satom == LMN_SATOM(LMN_SATOM_GET_LINK(dst_atom, i))) {
67        return linkconnection_push(link_connections, dst_atom, i, NULL);
68      }
69    }
70
71    LMN_ASSERT(false);
72    return -1;
73  }
74
75
76  LmnStringRef string_of_data_atom(LmnDataAtomRef data, LmnLinkAttr attr)
77  {
78    LmnStringRef result = lmn_string_make_empty();
79    if(attr == LMN_INT_ATTR){
80      char *s = int_to_str((long)data);
81      lmn_string_push_raw_s(result, s);
82    }
83    else if(attr == LMN_DBL_ATTR){
84      char buf[64];
85      sprintf(buf, "%#g", lmn_get_double(data));
86      lmn_string_push_raw_s(result, buf);
87    }
88
89    return result;
90  }
91
92
93  LmnStringRef string_of_template_membrane(Vector *link_connections, LmnMembraneRef mem,
        LmnSymbolAtomRef cm_atom)
94  {
95    LmnStringRef result = lmn_string_make_empty();
96    AtomListEntryRef ent;
97    LmnFunctor f;
98    char istr[(int)(8 * sizeof(int) * 0.3010) + 2]; /* 型の桁数 int + より長い 1 */
99
100   EACH_ATOMLIST_WITH_FUNC(mem, ent, f, ({
101     LmnSAtom satom;
102     if(LMN_IS_EX_FUNCTOR(f)) continue;
103     if(LMN_IS_PROXY_FUNCTOR(f)) continue;
104
105     EACH_ATOM(satom, ent, ({
106       int arity = LMN_FUNCTOR_GET_LINK_NUM(LMN_SATOM_GET_FUNCTOR(satom));
107       const char *atom_name = lmn_id_to_name(LMN_FUNCTOR_NAME_ID(LMN_SATOM_GET_FUNCTOR(
```

```
            satom)));
108
109       if (f == LMN_UNARY_PLUS_FUNCTOR) {
110         LmnSAtom in_proxy = LMN_SATOM(LMN_SATOM_GET_LINK(satom, 0));
111         LmnSAtom out_proxy = LMN_SATOM(LMN_SATOM_GET_LINK(in_proxy, 0));
112         if(cm_atom == LMN_SATOM(LMN_SATOM_GET_LINK(out_proxy, 1))) continue ;
113
114         sprintf(istr, "%d", linkconnection_make_linkno(link_connections, satom, 0));
115         lmn_string_push_raw_s(result, atom_name);
116         lmn_string_push_raw_s(result, LINK_PREFIX);
117         lmn_string_push_raw_s(result, istr);
118       }
119       else if(strcmp(atom_name, "==") == 0) {
120         lmn_string_push_raw_s(result, LINK_PREFIX);
121         sprintf(istr, "%d", linkconnection_make_linkno(link_connections, satom, 0));
122         lmn_string_push_raw_s(result, istr);
123         lmn_string_push_raw_s(result, "=");
124         lmn_string_push_raw_s(result, LINK_PREFIX);
125         sprintf(istr, "%d", linkconnection_make_linkno(link_connections, satom, 1));
126         lmn_string_push_raw_s(result, istr);
127       }
128       else if(atom_name[0] == '@') {
129         lmn_string_push_raw_s(result, atom_name);
130       }
131       else if(atom_name[0] == '$') {
132         lmn_string_push_raw_s(result, atom_name);
133         lmn_string_push_raw_c(result, '[');
134
135         for(int i = 0; i < arity; i++) {
136           if(i > 0) lmn_string_push_raw_s(result, ",");
137           lmn_string_push_raw_s(result, LINK_PREFIX);
138           sprintf(istr, "%d", linkconnection_make_linkno(link_connections, satom, i));
139           lmn_string_push_raw_s(result, istr);
140         }
141
142         lmn_string_push_raw_c(result, ']');
143       }
144       else {
145         if(strcmp(atom_name, ":-") == 0) {
146           lmn_string_push_raw_s(result, "':-'");
147         } else if(strcmp(atom_name, ".") == 0) {
148           lmn_string_push_raw_s(result, "'.'");
149         } else if(strcmp(atom_name, "[]") == 0) {
150           lmn_string_push_raw_s(result, "'[]'");
151         } else {
152           lmn_string_push_raw_s(result, atom_name);
153         }
154
155         if(arity > 0) {
156           lmn_string_push_raw_s(result, "(");
157           for(int i = 0; i < arity; i++) {
158             LmnLinkAttr attr = LMN_SATOM_GET_ATTR(satom, i);
159             if(i > 0) lmn_string_push_raw_s(result, ",");
160
161             if(LMN_ATTR_IS_DATA(attr) && LMN_HL_ATTR == attr)
162               {
163                 lmn_string_push_raw_s(result, LINK_PREFIX);
164                 sprintf(istr, "%d", linkconnection_make_linkno(link_connections, satom, i))
                        ;
165                 lmn_string_push_raw_s(result, istr);
166               }
167             else if(LMN_ATTR_IS_DATA(attr) && LMN_INT_ATTR == attr){
168               LmnAtomRef data = LMN_SATOM_GET_LINK(satom, i);
169               char *s = int_to_str((long)data);
170               lmn_string_push_raw_s(result, s);
```

```
171           }
172           else if(LMN_ATTR_IS_DATA(attr) && LMN_DBL_ATTR == attr){
173             LmnAtomRef data = LMN_SATOM_GET_LINK(satom, i);
174             char buf[64];
175             sprintf(buf, "%#g", lmn_get_double((LmnDataAtomRef)data));
176             lmn_string_push_raw_s(result, buf);
177           }
178           else{
179             lmn_string_push_raw_s(result, LINK_PREFIX);
180             sprintf(istr, "%d", linkconnection_make_linkno(link_connections, satom, i));
181             lmn_string_push_raw_s(result, istr);
182           }
183         }
184         lmn_string_push_raw_s(result, ")");
185       }
186     }
187     lmn_string_push_raw_s(result, ",");
188   }));
189  }));
190
191  for(LmnMembraneRef m = lmn_mem_child_head(mem); m; m = lmn_mem_next(m)) {
192    LmnStringRef s = string_of_template_membrane(link_connections, m, cm_atom);
193    if(lmn_string_last(s) == ',') lmn_string_pop(s);
194
195    lmn_string_push_raw_s(result, "{");
196    lmn_string_push(result, s);
197    lmn_string_push_raw_s(result, "},");
198
199    lmn_string_free(s);
200  }
201
202  return result;
203 }
204
205
206 LmnStringRef string_of_guard_op(LmnSymbolAtomRef satom)
207 {
208   LmnStringRef result = lmn_string_make_empty();
209   const char *atom_name = lmn_id_to_name(LMN_FUNCTOR_NAME_ID(LMN_SATOM_GET_FUNCTOR(satom))
        );
210   int arity = LMN_FUNCTOR_GET_LINK_NUM(LMN_SATOM_GET_FUNCTOR(satom));
211   LmnLinkAttr attr;
212   if(arity == 1)
213     lmn_string_push_raw_s(result, atom_name);
214   else{
215     attr = LMN_SATOM_GET_ATTR(satom, 0);
216     if(LMN_ATTR_IS_DATA(attr))
217       lmn_string_push(result, string_of_data_atom((LmnDataAtomRef)LMN_SATOM_GET_LINK(satom
            , 0), attr));
218     else
219       lmn_string_push(result, string_of_guard_op(LMN_SATOM_GET_LINK(satom, 0)));
220
221     if(strcmp(":=", atom_name) == 0)
222       lmn_string_push_raw_s(result, "=");
223     else
224       lmn_string_push_raw_s(result, atom_name);
225
226
227     attr = LMN_SATOM_GET_ATTR(satom, 1);
228     if(LMN_ATTR_IS_DATA(attr))
229       lmn_string_push(result, string_of_data_atom((LmnDataAtomRef)LMN_SATOM_GET_LINK(satom
            , 1), attr));
230     else
231       lmn_string_push(result, string_of_guard_op(LMN_SATOM(LMN_SATOM_GET_LINK(satom, 1))))
            ;
```

```
232   }
233
234   return result;
235 }
236
237
238 LmnStringRef string_of_guard_mem(LmnMembraneRef mem, LmnSymbolAtomRef cm_atom)
239 {
240   LmnStringRef result;
241   AtomListEntryRef ent;
242   LmnFunctor f;
243   const char* constraint_name[] = {"int", "float", "ground", "unary", "hlink", "new"};
244   const char* op_name[] = {"=:=", "=\\=", ">", "<", "=<", ">=", ":=", "==", "\\=", "><"};
245   result = lmn_string_make_empty();
246   EACH_ATOMLIST_WITH_FUNC(mem, ent, f, ({
247     if(LMN_IS_EX_FUNCTOR(f) || LMN_IS_PROXY_FUNCTOR(f)) continue;
248     LmnSymbolAtomRef satom;
249     EACH_ATOM(satom, ent, ({
250       const char *atom_name = lmn_id_to_name(LMN_FUNCTOR_NAME_ID(LMN_SATOM_GET_FUNCTOR(
              satom)));
251
252       if(f == LMN_UNARY_PLUS_FUNCTOR){
253         LmnSymbolAtomRef in_proxy = LMN_SATOM_GET_LINK(satom, 0);
254         LmnSymbolAtomRef out_proxy = LMN_SATOM_GET_LINK(in_proxy, 0);
255         if(cm_atom == LMN_SATOM_GET_LINK(out_proxy, 1)) continue;
256       }
257       else{
258         for(int i = 0; i < ARY_SIZEOF(constraint_name); i++) {
259           if(strcmp(constraint_name[i], atom_name) != 0) continue;
260           LmnSymbolAtomRef typed_pc_atom = LMN_SATOM(LMN_SATOM_GET_LINK(satom, 0));
261           const char *typed_pc_atom_name = lmn_id_to_name(LMN_FUNCTOR_NAME_ID(
                LMN_SATOM_GET_FUNCTOR(typed_pc_atom)));
262           lmn_string_push_raw_s(result, constraint_name[i]);
263           lmn_string_push_raw_s(result, "(");
264           lmn_string_push_raw_s(result, typed_pc_atom_name);
265           lmn_string_push_raw_s(result, "),");
266         }
267         for(int i = 0; i < ARY_SIZEOF(op_name); i++){
268           if(strcmp(op_name[i], atom_name) != 0) continue;
269           lmn_string_push(result, string_of_guard_op(satom));
270           lmn_string_push_raw_s(result, ",");
271         }
272       }
273     }));
274   }));
275
276   return result;
277 }
278
279 LmnStringRef string_of_firstclass_rule(LmnMembraneRef h_mem, LmnMembraneRef g_mem,
        LmnMembraneRef b_mem, LmnSAtom imply)
280 /* 引数の 3':-' のアトムで接続先が全て膜. 引数は第一引数から順につながってる膜
281    */
282 {
283   Vector *link_connections = vec_make(10);
284
285   LmnStringRef head = string_of_template_membrane(link_connections, h_mem, imply);
286   LmnStringRef guard = string_of_guard_mem(g_mem, imply);
287   LmnStringRef body = string_of_template_membrane(link_connections, b_mem, imply);
288
289   LmnStringRef result = lmn_string_make_empty();
290   lmn_string_push(result, head);
291   lmn_string_push_raw_s(result, ":-");
292   lmn_string_push(result, guard);
293   lmn_string_push_raw_s(result, "|");
```

```
294   lmn_string_push(result, body);
295   lmn_string_push_raw_s(result, ".");
296
297   lmn_string_free(head);
298   lmn_string_free(guard);
299   lmn_string_free(body);
300
301   for (int i = 0; i < vec_num(link_connections); i++) LMN_FREE(vec_get(link_connections, i
          ));
302   vec_free(link_connections);
303
304   return result;
305 }
306
307 LmnMembraneRef get_mem_linked_atom(LmnSymbolAtomRef target_atom, int link_n)
308 {
309   LmnAtomRef atom = LMN_SATOM_GET_LINK(target_atom, link_n);
310   return LMN_PROXY_GET_MEM(LMN_SATOM_GET_LINK(atom, 0));
311 }
312
313
314 void delete_ruleset(LmnMembraneRef mem, LmnRulesetId del_id)
315 {
316   Vector *mem_rulesets = lmn_mem_get_rulesets(mem);
317
318   for(int i = 0; i < vec_num(mem_rulesets); i++) {
319     LmnRuleSetRef rs = (LmnRuleSetRef)vec_get(mem_rulesets, i);
320     if (lmn_ruleset_get_id(rs) != del_id) continue;
321
322     /* move successors forward */
323     for(int j = i; j < vec_num(mem_rulesets) - 1; j++) {
324       LmnRuleSetRef next = (LmnRuleSetRef)vec_get(mem_rulesets, j + 1);
325       vec_set(mem_rulesets, j, (vec_data_t)next);
326     }
327
328     mem_rulesets->num--;
329     break;
330   }
331 }
332
333 st_table_t first_class_rule_tbl;
334
335 static int colon_minus_cmp(LmnSAtom x, LmnSAtom y)
336 {
337   return x != y;
338 }
339
340 static long colon_minus_hash(LmnSAtom x)
341 {
342   return (long)x;
343 }
344
345 static struct st_hash_type type_colon_minushash =
346 {
347   (st_cmp_func)colon_minus_cmp,
348   (st_hash_func)colon_minus_hash
349 };
350
351
352
353 void first_class_rule_tbl_init()
354 {
355   first_class_rule_tbl = st_init_table(&type_colon_minushash);
356 }
357
```

```
358
359 LmnRulesetId imply_to_rulesetid(LmnSAtom imply)
360 {
361   st_data_t entry;
362   if(st_lookup(first_class_rule_tbl, (st_data_t)imply, &entry)){
363     return (LmnRulesetId)entry;
364   }
365   return -1;
366 }
367
368
369 LmnRuleSetRef firstclass_ruleset_create(LmnSymbolAtomRef imply) {
370   /* ':-'アトムがプロキシにつながっていなければ中止_3 */
371   for(int j = 0; j < 3; j++){
372     LmnAtomRef pa = LMN_SATOM_GET_LINK(imply, j);
373     if(!LMN_SATOM_IS_PROXY(pa)) return NULL;
374   }
375
376   /* ':-'_3(head, guard, body)からルール文字列を生成してコンパイル */
377   LmnMembraneRef head = get_mem_linked_atom(imply, 0);
378   LmnMembraneRef guard = get_mem_linked_atom(imply, 1);
379   LmnMembraneRef body = get_mem_linked_atom(imply, 2);
380   LmnStringRef rule_str = string_of_firstclass_rule(head, guard, body, imply);
381   FILE *compiled_rulesets = lmntal_compile_rule_str((char *)lmn_string_c_str(rule_str));
382   lmn_string_free(rule_str);
383
384   /* コンパイルされたルールからルールセットを生成*/
385   RuleRef ruleAST;
386   il_parse_rule(compiled_rulesets, &ruleAST);
387   LmnRulesetId id = lmn_gen_ruleset_id();
388   LmnRuleSetRef ruleset = lmn_ruleset_make(id, 1);
389   lmn_ruleset_put(ruleset, load_rule(ruleAST));
390   lmn_set_ruleset(ruleset, id);
391
392   fclose(compiled_rulesets);
393
394   /* アトムとコンパイルされたルールセット:-を対応付けるハッシュテーブルへ追加 ID */
395   st_insert(first_class_rule_tbl, (st_data_t)imply, (st_data_t)id);
396
397   return ruleset;
398 }
399
400 void firstclass_ruleset_release(LmnSymbolAtomRef imply) {
401   LMN_ASSERT(st_contains(imply));
402   st_delete(first_class_rule_tbl, (st_data_t)imply, NULL);
403 }
404
405 LmnRuleSetRef firstclass_ruleset_lookup(LmnSymbolAtomRef imply) {
406   LmnRulesetId id = imply_to_rulesetid(imply);
407   return (id > 0) ? lmn_ruleset_from_id(id) : NULL;
408 }
```

A.2 処理系内部機能への API の実装部

以下に第 3.3.4 節で述べたコールバック関数の実装を示す.

A.2.1 cb_react_ruleset_nd

```
1  /**
2   * apply rules in rulesets by one-step.
3   *
4   * the reacted graphs are added to {\c pos} of the list {\c head}.
5   */
6  static void apply_rules_in_rulesets(LmnReactCxtRef rc, LmnMembraneRef mem,
7                                      LmnMembraneRef src_graph, Vector *rulesets,
8                                      LmnSymbolAtomRef *head, int *pos)
9  {
10   for (int i = 0; i < vec_num(rulesets); i++) {
11     LmnRuleSetRef rs = (LmnRuleSetRef)vec_get(rulesets, i);
12
13     for (int j = 0; j < lmn_ruleset_rule_num(rs); j++) {
14       LmnRuleRef r = lmn_ruleset_get_rule(rs, j);
15       mc_react_cxt_init(rc);
16       RC_SET_GROOT_MEM(rc, src_graph);
17       RC_ADD_MODE(rc, REACT_ND_MERGE_STS);
18       react_rule(rc, src_graph, r);
19       int n_of_results = vec_num(RC_EXPANDED(rc));
20
21       for (int k = n_of_results - 1; k >= 0; k--) {
22         LmnSymbolAtomRef cons = lmn_mem_newatom(mem, LMN_LIST_FUNCTOR);
23         LmnMembraneRef m = (LmnMembraneRef)vec_get(RC_EXPANDED(rc), k);
24         LmnSymbolAtomRef in = lmn_mem_newatom(m, LMN_IN_PROXY_FUNCTOR);
25         LmnSymbolAtomRef out = lmn_mem_newatom(mem, LMN_OUT_PROXY_FUNCTOR);
26         LmnSymbolAtomRef plus = lmn_mem_newatom(m, LMN_UNARY_PLUS_FUNCTOR);
27         lmn_mem_add_child_mem(mem, m);
28         lmn_newlink_in_symbols(in, 0, out, 0);
29         lmn_newlink_in_symbols(in, 1, plus, 0);
30         lmn_newlink_in_symbols(out, 1, cons, 0);
31         lmn_newlink_in_symbols(cons, 1, *head, *pos);
32         *head = cons;
33         *pos = 2;
34       }
35     }
36   }
37 }
38
39 void cb_react_ruleset_nd(LmnReactCxtRef rc,
40                          LmnMembraneRef mem,
41                          LmnAtomRef rule_mem_proxy, LmnLinkAttr rule_mem_proxy_link_attr,
42                          LmnAtomRef graph_mem_proxy, LmnLinkAttr graph_mem_proxy_link_attr,
43                          LmnAtomRef return_rule_mem_proxy, LmnLinkAttr
                                 return_rule_mem_proxy_link_attr,
44                          LmnAtomRef react_judge_atom, LmnLinkAttr react_judge_link_attr)
45 {
46   LmnMembraneRef rule_mem = LMN_PROXY_GET_MEM(LMN_SATOM_GET_LINK(rule_mem_proxy, 0));
47   LmnAtomRef in_mem = LMN_SATOM_GET_LINK(graph_mem_proxy, 0);
48   LmnMembraneRef graph_mem = LMN_PROXY_GET_MEM(in_mem);
49
50   lmn_mem_delete_atom(graph_mem, LMN_SATOM_GET_LINK(in_mem, 1), LMN_SATOM_GET_ATTR(in_mem,
         1));
51   lmn_mem_delete_atom(graph_mem, in_mem, LMN_SATOM_GET_ATTR(LMN_SATOM(graph_mem_proxy), 0)
       );
52
53   LmnReactCxtRef tmp_rc = react_context_alloc();
54   mc_react_cxt_init(tmp_rc);
55
56   LmnSymbolAtomRef head = lmn_mem_newatom(mem, LMN_NIL_FUNCTOR);
57   int pos = 0;
58
59   Vector *rulesets = lmn_mem_get_rulesets(rule_mem);
60   apply_rules_in_rulesets(tmp_rc, mem, graph_mem, rulesets, &head, &pos);
61
```

```
#ifdef USE_FIRSTCLASS_RULE
  Vector *fstclass_rules = lmn_mem_firstclass_rulesets(rule_mem);
  apply_rules_in_rulesets(tmp_rc, mem, graph_mem, fstclass_rules, &head, &pos);
#endif

  lmn_mem_newlink(mem, head, LMN_ATTR_MAKE_LINK(pos), pos,
                  react_judge_atom, react_judge_link_attr,
                  LMN_ATTR_GET_VALUE(react_judge_link_attr));

  lmn_mem_remove_mem(mem, graph_mem);

  lmn_mem_newlink(mem,
                  return_rule_mem_proxy, return_rule_mem_proxy_link_attr,
                  LMN_ATTR_GET_VALUE(return_rule_mem_proxy_link_attr),
                  rule_mem_proxy, rule_mem_proxy_link_attr,
                  LMN_ATTR_GET_VALUE(rule_mem_proxy_link_attr));

  mc_react_cxt_destroy(tmp_rc);
  react_context_dealloc(tmp_rc);
  lmn_mem_delete_atom(mem, graph_mem_proxy, graph_mem_proxy_link_attr);
}
```

A.2.2 state_map コレクション

```
struct LmnStateMap {
  LMN_SP_ATOM_HEADER;
  StateSpaceRef states;
};

static int state_map_atom_type;

static LmnStateMapRef lmn_make_state_map(LmnMembraneRef mem)
{
  LmnStateMapRef s = LMN_MALLOC(struct LmnStateMap);
  LMN_SP_ATOM_SET_TYPE(s, state_map_atom_type);
  s->states = statespace_make(NULL, NULL);
  return s;
}

void lmn_state_map_free(LmnStateMapRef state_map, LmnMembraneRef mem)
{
  statespace_free(((LmnStateMapRef)state_map)->states);
  LMN_FREE(state_map);
}

/*----------------------------------------------------------------------
 * Callbacks
 */

/*
 * 生成
 * -a0 Map
 */
void cb_state_map_init(LmnReactCxtRef rc,
                       LmnMembraneRef mem,
                       LmnAtomRef a0, LmnLinkAttr t0)
{
  LmnStateMapRef atom = lmn_make_state_map(mem);
  LmnLinkAttr attr = LMN_SP_ATOM_ATTR;
  LMN_SP_ATOM_SET_TYPE(atom, state_map_atom_type);
  lmn_mem_push_atom(mem, atom, attr);
```

```
38    lmn_mem_newlink(mem,
39                    a0, t0, LMN_ATTR_GET_VALUE(t0),
40                    atom, attr, 0);
41  }
42
43  /*
44   * 解放
45   * +a0 Map
46   */
47  void cb_state_map_free(LmnReactCxtRef rc,
48                         LmnMembraneRef mem,
49                         LmnAtomRef a0, LmnLinkAttr t0)
50  {
51    lmn_state_map_free((LmnStateMapRef)a0, mem);
52    lmn_mem_remove_data_atom(mem, (LmnDataAtomRef)a0, t0);
53  }
54
55  /*
56   * 状態->ID
57   * +a0 Map
58   * +a1 状態
59   * -a2 ID
60   * -a3 Map
61   */
62  void cb_state_map_id_find(LmnReactCxtRef rc,
63                            LmnMembraneRef mem,
64                            LmnAtomRef a0, LmnLinkAttr t0,
65                            LmnAtomRef a1, LmnLinkAttr t1,
66                            LmnAtomRef a2, LmnLinkAttr t2,
67                            LmnAtomRef a3, LmnLinkAttr t3)
68  {
69    LmnMembraneRef m = LMN_PROXY_GET_MEM(LMN_SATOM_GET_LINK(a1, 0));
70    StateSpaceRef ss = ((LmnStateMapRef)a0)->states;
71    LmnSymbolAtomRef out = a1;
72    LmnSymbolAtomRef in = LMN_SATOM_GET_LINK(a1, 0);
73    LmnLinkAttr in_attr = LMN_SATOM_GET_ATTR(a1, 0);
74
75    LmnSymbolAtomRef at = lmn_mem_newatom(m, lmn_functor_intern(ANONYMOUS, lmn_intern("@"),
          1));
76    LmnSymbolAtomRef plus = LMN_SATOM_GET_LINK(in, 1);
77    lmn_newlink_in_symbols(plus, 0, at, 0);
78
79    lmn_mem_delete_atom(m, in, in_attr);
80    lmn_memstack_delete(RC_MEMSTACK(rc), m);
81    lmn_mem_remove_mem(mem, m);
82
83    State *new_s = state_make(m, 0, TRUE);
84    State *succ = statespace_insert(ss, new_s);
85
86    if (succ == new_s) { /* new state */
87      state_id_issue(succ);
88    } else {
89      state_free(new_s);
90    }
91
92    lmn_mem_push_atom(mem, succ, LMN_INT_ATTR);
93    lmn_mem_newlink(mem,
94                    a2, t2, LMN_ATTR_GET_VALUE(t2),
95                    succ, LMN_INT_ATTR, 0);
96
97    lmn_mem_newlink(mem,
98                    a0, t0, LMN_ATTR_GET_VALUE(t0),
99                    a3, t3, LMN_ATTR_GET_VALUE(t3));
100
101   lmn_mem_delete_atom(mem, a1, t1);
```

```
102 }
103
104 /*
105  * ID状態->
106  * +a0 Map
107  * +a1 ID
108  * -a2 状態
109  * -a3 Map
110  */
111 void cb_state_map_state_find(LmnReactCxtRef rc,
112           LmnMembraneRef mem,
113           LmnAtomRef a0, LmnLinkAttr t0,
114           LmnAtomRef a1, LmnLinkAttr t1,
115           LmnAtomRef a2, LmnLinkAttr t2,
116           LmnAtomRef a3, LmnLinkAttr t3)
117 {
118   State *s = (State *)a1;
119   st_data_t entry;
120
121   LmnMembraneRef new_mem = state_mem_copy(s);
122   LmnFunctor at_functor = lmn_functor_intern(ANONYMOUS, lmn_intern("@"), 1);
123
124   AtomListEntryRef ent;
125   LmnFunctor f;
126   LmnSymbolAtomRef at_atom;
127   EACH_ATOMLIST_WITH_FUNC(new_mem, ent, f, {
128     if (f != at_functor) continue;
129
130     LMN_ASSERT(atomlist_ent_num(ent) == 1);
131     at_atom = atomlist_head(ent);
132   });
133
134   LmnSymbolAtomRef plus = LMN_SATOM_GET_LINK(at_atom, 0);
135   lmn_mem_delete_atom(new_mem, at_atom, 0);
136
137   LmnSymbolAtomRef in = lmn_mem_newatom(new_mem, LMN_IN_PROXY_FUNCTOR);
138   LmnSymbolAtomRef out = lmn_mem_newatom(mem, LMN_OUT_PROXY_FUNCTOR);
139
140   lmn_newlink_in_symbols(plus, 0, in, 1);
141   lmn_newlink_in_symbols(in, 0, out, 0);
142
143   lmn_mem_newlink(mem,
144       a2, t2, LMN_ATTR_GET_VALUE(t2),
145       out, LMN_ATTR_MAKE_LINK(1), 1);
146
147   lmn_mem_newlink(mem,
148       a0, t0, LMN_ATTR_GET_VALUE(t1),
149       a3, t3, LMN_ATTR_GET_VALUE(t3));
150
151   lmn_mem_add_child_mem(mem, new_mem);
152 }
```