日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

LMNtal実行時処理系の並列モデル検査器への展開

後町 将人　堀 泰祐　上田 和紀

システム検証技術であるモデル検査は, 状態遷移系として表現したシステムの振舞いから不具合の探索を網羅的に行う技術として注目を集めている. 階層グラフ書換えに基づく状態遷移系記述言語 LMNtal は, 実行時処理系 SLIM を拡張する形でモデル検査器へと発展してきた. 階層グラフ構造は, 状態空間探索において重要となる対称性吸収メカニズムを備えた強力なデータ構造であるが, それでもなおモデル検査は状態空間爆発によるメモリ使用量と検証時間の爆発的な増加を招きやすく, 時間と空間の両面で効率的な状態管理方式を必要としていた. この問題に対処して検証機能としての有用性を高めるべく, 我々は共有メモリ環境を対象にした並列モデル検査器への拡張と新たな状態管理手法の開発に取り組み, LMNtal をモデル記述言語としたモデル検査の検証規模拡大と高速化を実現した.

1　はじめに

プログラムが交通システム, 金融システム, 組込み機器など，社会のあらゆる場所で使われるようになった一方で，プログラムの誤動作は人命に関わる事故や経済的に大きな損失に繋がりかねず, ソフトウェアの信頼性への要求が高まっている. 特に並行処理を含むプログラムに対してテストやシミュレーションによって網羅的に不具合を検出することは容易ではない. 近年では, システムの信頼性向上のための検証技術として形式手法の一つであるモデル検査 [9] が注目を集めている.

モデル検査とは, システムの振舞いをモデル記述言語を用いて状態遷移系として記述し, 計算機上で網羅的に不具合の探索を行う自動検証技術である. 特に近年では Java や C++といったシステム開発に利用されるプログラミング言語の記述をそのまま検証可能なソフトウェアモデル検査ツール [24] [30] も登場し, モデル検査による検証は身近な技術となってきている. このようなモデル検査の有用性に注目し, 我々は並行言語 LMNtal [25] をモデル記述言語としたモデル検査器 [38] の開発, 公開に取り組んでいる.

LMNtal は階層グラフの多重集合書換えに基づく状態遷移系記述言語である．LMNtal の言語モデル [29] は従来の計算モデル，特に並行計算モデルを統合することを目標に 2002 年より設計開発が行われ, 現在も発展中のプロジェクトである. プロセス, データ, メッセージを統一的に扱い可能なことや, 並行処理を容易に記述可能であることが特徴であり, これまで様々な計算モデルを表現することで LMNtal の高い記述力を示してきた [26] [27] [36]．このような表現力と記述力の高さを持つ LMNtal をモデル記述言語に用いてモデル検査を行うことができれば様々な計算モデルの検証を容易に実施できる. そこで, LMNtal の有用性向上を図ることを視野に, LMNtal 実行時処理系 SLIM に対して通常のプログラム実行の手法を自然に拡張する形でモデル検査器を構築した.

一方で, モデル検査は探索する状態空間が組合せ爆発を招きやすく, メモリ不足や爆発的な検証時間の増加が問題となっている. 階層グラフ構造を状態, グラフ書換えを遷移とする LMNtal モデル検査器は状態の等価性をグラフ同型性判定により実現したことで,

---

Evolution of the LMNtal runtime to a parallel model checker

Masato Gocho, Taisuke Hori, 早稲田大学大学院基幹理工学研究科, Graduate School of Fundamental Science and Engineering, Waseda University.

Kazunori Ueda, 早稲田大学理工学術院情報理工学科, Faculty of Science and Engineering, Waseda University.

グラフの対称性に基づく状態数削減効果を備えており, 問題によっては状態爆発を大きく緩和することができる. しかしながら, それでもなお LMNtal の表現力, 記述力の高さをそのままモデル検査に用いたことで必要となった複雑な状態管理手法は状態爆発の影響が大きく, 時間と空間の両面で効率的な手法への改善が求められていた.

我々はこの問題に対処すべく, まずは状態である階層グラフ構造をバイト列へ可逆圧縮する新たな状態管理手法を開発することで検証可能な規模を拡大した. また, 遷移先状態と遷移元状態との差分情報を利用して状態生成を高速化すると共に, 共有メモリ計算機を利用した並列モデル検査器への発展に取り組んできた. 本論文では, 開発公開中の LMNtal 実行時処理系 SLIM の並列モデル検査器への展開について, 実行時処理系からモデル検査器への発展 (第 2 節), 状態管理手法の改善 (第 3 節), 状態遷移グラフ構築の高速化 (第 4 節), 並列化 (第 5 節), 評価実験 (第 6 節) の順で述べてゆく.

2 LMNtal モデル検査器

本節では, LMNtal の言語仕様や実行時処理系 SLIM の概要について述べると共に, モデル検査器として展開する上で採用した基本的な枠組と, 状態空間の構築や探索のために行う処理について説明する.

2.1 並行言語 LMNtal

LMNtal モデル検査器は, 階層グラフの多重集合書き換えに基づく状態遷移系記述言語 LMNtal でモデリング (プログラミング) した記述に対し, 階層グラフ構造を状態, ルールによる書換えを遷移とした状態遷移グラフを構築する. 各状態を表す階層グラフ構造はアトム, リンク, 膜, ルールの 4 つの構成要素を基本データ構造として持つ. アトムはグラフにおけるノードを表し, エッジを表すリンクはアトム同士を一対一に無向接続する. 膜は階層を表し, アトムの多重集合を局所化する. このような階層グラフ構造をプロセスと呼び, 膜はプロセスへの書換え規則 (ルール) を持つことで繰返しプロセスを書き換えて計算を進める. 書換え可能なプロセスや適用可能なルールが同時に複数存在する場合の書換え順序には非決定性が生ずる.

P ::=	$\mathbf{0}$	(空)
	$p(X_1, \ldots, X_m)$ $(m \geq 0)$	(アトム)
	P, P	(分子)
	$\{P\}$	(セル)
	T :- T	(ルール)
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)

図 1 LMNtal の基本構文

図 1 に LMNtal の基本構文を示す. X_i はリンク名, p はアトム名, P はプロセスであり, 具象構文ではリンクは大文字から始まる識別子, アトム名は小文字から始まる識別子で表現する. 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) を扱う機能を持つ. $\mathbf{0}$ は中身のないプロセス, $p(X_1, \ldots, X_m)$ は m 本のリンクを持つアトムであり, アトム名とリンク本数の組 (ファンクタ) でアトムの種類を区別する. 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であり, アトム同様に名前 (膜名) を持つことができる. また, リンクを辿ることで到達可能なアトムと膜の集合を分子と呼ぶ. T :- T はプロセスの書換え規則 (ルール) である. ルールは, 左辺に書換え対象となるプロセスのテンプレートを記述し, :- を挟んだ右辺に書換え後のプロセスのテンプレートを記述する. 更に, LMNtal には柔軟な記述を行うための略記法や拡張構文が数多く存在するが本論文では割愛し, 詳細は文献 [40] へ委ねる.

2.2 実行時処理系からモデル検査器への発展

公開[†1]中の LMNtal 言語処理系には, Java で実装したコンパイラと実行時処理系 [40], C で実装した実行時処理系 SLIM が存在する. LMNtal コンパイラ

†1 http://www.ueda.info.waseda.ac.jp/lmntal/

は, 初期状態となる階層グラフ構造の構築と書換え処理の指示をバックトラック機能を持った Static Single Assignment 形式の独自の中間語命令列によって行い, 実行時処理系は命令語を一語ずつ解釈実行することでプログラムを処理する.

LMNtal の応用分野として並列実行や検証への応用を見据えた場合, Java による実行時処理系が豊富な機能を提供することや外部タスクとの非同期実行を目的に設計されてきたことから, より高い性能を持つ新たな処理系が必要となった. このような背景から 2007 年より高速かつ軽量な新たな実行時処理系として SLIM の開発が始まった. 現在では修正 BSD ライセンスのオープンソースソフトウェアとして Google Code 上のプロジェクトとして公開[†2]を行っている. 従来の処理系と比較して高い実行性能を実現するため, SLIM には言語仕様に対してわずかな制約があるものの, ほとんど全ての LMNtal プログラムに対して網羅的な実行経路の取得や検証を行うことができる. 検証手法には, 検証を状態遷移グラフに対するグラフ探索として定式化したオートマトンベースの LTL モデル検査手法を採用し, LMNtal の通常のプログラム実行を状態遷移グラフ構築処理へ拡張することで実現した. SLIM 同様にオープンソースソフトウェアとして開発公開中[†3]の LMNtal 統合開発環境 LaViT [31] を通じて, LMNtal をモデル記述言語としたモデル検査の有用性を示してきた [28] [32].

2.2.1 オートマトンベースの LTL モデル検査

オートマトンベースの LTL モデル検査手法とは, 状態遷移系モデル (プログラム) の各状態を頂点, 状態変化を枝とした状態遷移グラフに対し, グラフ探索を行うことで満たすべき仕様に対する反例検出を行う検証手法である. システムが満たすべき性質の記述には線形時相論理 (LTL) を使用し, LTL 式で与えた性質をシステムが満たすか否かを検査する. 安全性 (safety) の検証では, 危険な状態を示す頂点への到達可能性を探索し, 活性 (liveness) の検証では, 目標となる頂点へ到達しない閉路の探索を行う. なお, 厳密な定義や詳細については文献 [9] [14] へ委ね, 本論文では割愛する.

一般的には状態管理にハッシュ表を使用し, 必要に応じて状態同士の等価性判定処理を行うことで, 新たに展開した遷移先の状態が既出か否かを検査する. 未展開の状態は, 深さ優先探索 (DFS) や幅優先探索 (BFS) といった訪問順序で展開していくことで, 探索対象となる状態遷移グラフを構築する.

本手法を採用したモデル検査器としては Spin [14] や DiVinE [3] が挙げられる. また, メンバシップ等式論理と書換え論理に基づく代数仕様言語 Maude [10] のモデル検査器への拡張 [11] も本手法を採用している.

2.3 SLIM におけるデータ管理

実行時処理系 SLIM における LMNtal プロセスの表現方法について説明する.

SLIM では, アトム名とリンクを持った通常のアトムをシンボルアトム, 他の特別な役割を持つアトムをデータアトムとして区別し, 異なるメモリ管理を行っている. データアトムには, 整数値を表すアトム, 浮動小数点数を表すアトム, ファイルポートや C によるインラインコードを実現するための特殊データを表すスペシャルアトムがある. シンボルアトムは種類数が 65536 個以下で, アトム一つあたりのリンク本数が 128 本未満, データアトム同士は接続ができないという制限を設けることで軽量化を図っている. シンボルアトムの種類を表すファンクタは 2 バイトの整数 ID で管理しており, ファンクタ ID からアトム名とリンク本数を取得するための辞書を備えている. リンクは接続先がシンボルアトムかデータアトムかを判別するリンク属性とリンクデータを組にした管理を行っており, リンク属性がシンボルアトムを示すならば接続先のシンボルアトムへの参照を持ち, 整数値のアトムを示すならば直接その値を埋め込んで保持する. 浮動小数点数やスペシャルアトムを示す場合は, そのメモリへの参照を持つ.

階層化を表現する膜は, 膜自身が所属する膜 (親膜) への参照を持つ. また, 膜に所属する膜 (子膜) の集合は一つの双方向リストによって管理しており, 膜は子膜リストの先頭への参照を持つ. 膜に所属するアトムはファンクタの種類ごとに双方向リストを構成してお

†2 http://code.google.com/p/slim-runtime/
†3 http://code.google.com/p/lavit/

り, ファンクタ ID をキーにアトムリストを取得するハッシュ表を備えている. この他, 膜名, 書換え処理が全て終了したことを示す stable フラグ, 階層に存在する各ルール (中間語命令列) への参照を配列として持っている. stable フラグは LMNtal プログラムの停止性判定に使用し, 最上位の階層の stable フラグが真になった場合, LMNtal プログラムが停止したことを判定する.

N 本のリンクを持つシンボルアトムを表現するために必要なメモリ量をワード (ポインタの表現に必要なデータサイズ) 単位で計算すると, 双方向リストを構成するポインタに 2 ワード, アトムの種類を表すファンクタの整数 ID に 2 バイト, 各リンクのリンク属性に 1 バイト, および各リンクデータに 1 ワードを要するため, 合計メモリ量は以下の式で表される. なお, 式中の W は 1 ワードを表す.

$$2 + \lceil (2 + N) \,/\, W \rceil + N \qquad (1)$$

また, 膜の表現に必要なメモリ量は最低でも 324 バイトを必要としている. 膜を単位にルール適用処理を行うため, 膜は多くの情報を管理しているためである.

状態はこのような階層グラフ構造やハッシュ値, 遷移先状態への参照を持ち, 全ての状態をハッシュ表で管理している.

2.4 状態遷移グラフ構築処理の概要

状態遷移グラフの構築は, 未展開状態の集合 (スタック) から深さ優先順に状態を選択し, 求めた遷移先状態のうちの未展開状態をスタックへ追加する処理を繰り返して行う.

遷移先の計算は, 未展開の階層グラフ構造に対して, 各ルールごとに, 書換え可能な部分グラフ構造を全探索し, それぞれに対して書換え後の階層グラフ構造を構築することで実現している. 状態管理表は, 階層グラフ構造から計算したハッシュ値をキーとしたチェイン法に基づくハッシュ表を使用しており, キーから求めたエントリ先に既に異なる状態が登録されている場合にはリンクリストを構成する. 等価な状態の検出は, ハッシュ値の同じ状態に対してグラフ同型性判定を行うことで実現している.

図 2 に, 遷移先を計算する処理 *expand* の疑似コードを示す. まずは DFS によって訪問した展開元となる状態 base_s の階層グラフ構造 base_g を *get_state_graph* の手続きで取得し, *gen_successor_graphs* の手続きによって遷移先の階層グラフ構造を計算する. 求めた遷移先の階層グラフ構造の集合は new_graphs として保持する. *state_make* は求めた遷移先の階層グラフ構造 succ_graph を持った状態の生成手続きを表し, succ_grpah のハッシュ値を計算して付加する. *state_space_insert* は生成した状態 new が状態管理表に既出か否かを検査する手続きであり, 状態 new と等価な状態が状態管理表に存在するならばその状態への参照を返し, 状態 new が新規状態として状態管理表へ追加されたならば状態 new 自身への参照を返す. *add_successor* は, この返り値となる状態への参照を展開元状態の遷移先として設定する手続きを示す. 以上の手続きを経て未展開状態リストを更新し, 繰り返し *expand* の手続きを行うことで状態遷移グラフを構築していく.

```
procedure expand(base_s)
  base_g ← get_state_graph(base_s)
  new_graphs ← gen_successor_graphs(base_g)
  for all succ_graph ∈ new_graphs do
    new ← state_make(succ_graph)
    ret ← state_space_insert(new)
    add_successor(base_s, ret)
  end for
end procedure
```

図 2 状態の遷移先計算処理

2.4.1 階層グラフ構造のハッシュ値の計算処理

階層グラフ構造のハッシュ関数には,

- 等価な階層グラフ構造に対して異なるハッシュ値を生成してはならず,
- 異なる階層グラフ構造が同じハッシュ値をできるだけ生成しない

といった性質が求めれられる. ハッシュ値の衝突が発生した場合, グラフ同型性判定の処理回数が増加するため実行性能に大きく影響してしまう. このような背景を踏まえ, 階層グラフ構造のハッシュ関数は文献

[37] で提案したアルゴリズムをベースに実装している.

ハッシュ値の計算は, 膜を単位に, より上位の階層の膜から分子, 子膜の順に計算していく. 分子のハッシュ値は, 各アトムを基点にアトムと膜を頂点とする深さ D までの木構造として読み出したグラフを計算単位とし, この木構造の集合から計算を行う. 子膜のハッシュ値の計算では, 膜のハッシュ値を再帰的に計算して重み値を付加していく. 最終的に, 全ての分子と子膜のハッシュ値の総和と総積の排他的論理和がハッシュ値となる.

D の値が大きい場合はハッシュ値は好ましく分散するが, 計算に要する時間面のコストが増加してしまう. 逆に D の値が小さい場合は計算に要する時間面のコストは小さく済むが, 異なる状態に対して同じハッシュ値が生成されやすくなってしまう. 本手法では, 求める木構造の判別がつかなくなるような, 例えば同一のアトムが数多く連結している場合にハッシュ値が衝突しやすいという問題もある. 問題を抱えてはいるものの, 多くの例題ではハッシュ値が散らばることがこれまで蓄積してきた例題によるベンチマークテストで確かめられている. 現状では本手法が最良であり, LMNtal モデル検査器では D の値を 2 に設定し, デフォルトのハッシュ関数として採用している.

2.4.2 階層グラフ構造同士の同型性判定

状態遷移グラフを構築する際には, 未展開の状態から計算した遷移先となる状態が出現済か否かを判定する状態の等価性検査をグラフ同型性判定によって実現している [38].

グラフ同型性判定処理は, 二つの階層グラフ構造に存在する全てのプロセスが, 過不足なく一対一に対応付け可能であることを探索によって判定する. プロセスの比較は膜を単位として行い, 膜内に存在するアトムの接続関係が互いに一致しているかどうかを DFS でグラフ走査することで判定する. グラフ同型性判定の計算コストは, ルール適用時の部分グラフ構造の探索に比べて探索するグラフ構造が巨大となるため計算コストが高い. そこで, 比較対象となった二つの膜に存在するアトム, 膜, ルールの種類と個数が互いに一致することをグラフ走査を行う前の段階で確認する処理を組み込み, 判定の高速化を図っている.

2.5 閉路探索アルゴリズム

LMNtal モデル検査器では, 状態空間を構築しながら探索を行う二段階の深さ優先探索アルゴリズム (on-the-fly Nested-DFS [19]) を導入している. Nested-DFS アルゴリズムでは, 状態遷移グラフの構築を一段階目の DFS が行い, 一段階目の DFS がバックトラックする際に二段階目の DFS が閉路の探索を行う. 二段階目の DFS は一段階目の DFS で訪問した状態の後行順に探索を始めるため, DFS の根となる状態から到達可能な状態は全て遷移先の計算が完了している. そのため, 最悪の場合でも全状態への訪問がグラフの構築時と探索時の合計 2 回で済む効率的な手法であり, Spin, DiVinE, Maude など多くのツールが実際に導入している.

3 コンパクトな状態管理手法の実現

LMNtal モデル検査器において, 状態である階層グラフ構造をそのままハッシュ表で管理することはメモリ面のコストが高く, 検証可能な規模を大きく制限する主な要因となっていた. 1 ワードが 8 バイトの環境では, アトムは式 (1) よりリンクが 0 本のシンボルアトムの表現に 24 バイト, リンクが 2 本のシンボルアトムの表現には 40 バイトを必要とし, 膜はハッシュ表などのデータ構造を持つことから, 網羅的な状態管理を行う上でメモリ面のコストが非常に高価であった. 同じメモリ容量でより多くの状態を表現可能にするためには状態の表現をコンパクトにする必要がある. 本研究では, LMNtal の階層グラフ構造を復元可能な形式で表現したバイト列へエンコードする手法を開発し, 検証可能な規模拡大を実現した. 本手法の導入により, 状態管理表に登録した状態をバイト列として管理するため, 必要に応じて階層グラフ構造の再構築を行う処理を追加している.

3.1 モデル検査における状態圧縮の研究

状態空間の各状態を明示的なデータとして管理するオートマトンベースのモデル検査手法では, 状態毎のサイズを圧縮する研究は広く取り組まれている. 一般的なデータ圧縮アルゴリズムによる可逆圧縮や損失圧縮を適用することでメモリ使用量の削減を実現

表 1　バイト列に記録するタグとデータ

タグの種類	タグが示す情報	記録データ	バイト数
TAG_ATOM_START	シンボルアトム	ファンクタ ID	2
TAG_ATOM_REF	記録済みのアトム	記録済みアトムへの訪問順序	2
		接続先アトムのリンク番号	1
TAG_FROM	DFS で辿ってきたリンクと接続している	—	0
TAG_INT_DATA	整数値アトム	整数値	1W
TAG_DBL_DATA	浮動小数点数アトム	浮動小数点数への参照	1W
TAG_MEM_START	膜の始まり	—	0
TAG_NAMED_MEM_START	名前を持つ膜の始まり	膜名の ID	4
TAG_MEM_END	膜の終わり	—	0
TAG_MEM_ESCAPE	辿ったリンク先が膜を抜ける	—	0
TAG_MEM_REF	記録済みの膜への訪問番号	記録済みの膜への訪問順序	2
TAG_RULESET1	1 種類のルール集合	集合 ID	2
TAG_RULESET	N 種類のルール集合	集合数	2
		N 個の集合 ID	2 * N

した事例や, モデル記述言語の特徴を利用した圧縮法の研究も報告されている [16]. 可逆圧縮法の適用は, 検証速度を多少犠牲にすることで表現可能な規模の拡大を見込むことができる. 一方で, 損失圧縮法の適用には可逆圧縮法より高い圧縮率を期待できるものの, モデル検査の利点である網羅性を損なう恐れがある. 文献 [23] では, 異なる状態から状態データを再構築する圧縮アルゴリズムが提案されている. しかしながら, 既存のデータ圧縮アルゴリズムを階層グラフ構造へ適用することは難しい. 開発した圧縮アルゴリズムは LMNtal の言語の特徴である階層グラフ構造に応じた手法であり, バイト列としてエンコードすることでデータ圧縮ライブラリ zlib [12] の適用も実現している.

3.2　圧縮アルゴリズム概要

階層グラフ構造のエンコード処理は膜を単位に, より上位の階層の膜から分子, 膜, ルールの順に一つのバイト列として記録していく. エンコードアルゴリズムはプロセス数に対して常に線形の計算量で動作するものの, 多重集合であるアトムや膜の並びが決定的ではないため, 等価な階層グラフ構造に対して常に等価なバイト列をエンコードするとは限らない. そのため, 階層グラフ構造同士を比較する場合はバイト列から階層グラフ構造を再構築する必要があり, エンコードするバイト列にはプロセスの記録に先立って 0.5 バイトのタグを記録する形式を採っている. 表 1 に, エンコードに用いるタグと, 各タグに対応した記録情報と使用メモリ量の一覧を示す. 例えば, タグ TAG_ATOM_START は記録したプロセスがアトムであることを示しており, バイト列中のアトムは 2 バイトのファンクタ ID と 0.5 バイトのタグの合計 2.5 バイトで表現している. 高価なデータ構造であった膜は, 始端と終端を表すタグのみの合計 1 バイト (膜名を持つ場合は 5 バイト) で表現しており, 大きくメモリ使用量を削減している.

図 3 に, 階層グラフ構造をエンコードする疑似コードを示す. エンコードは最上位の階層である膜 mem を *encode_membrane* の手続きに渡すことで始まる. 分子のエンコード *encode_molecules* は, バイト列として未記録の全てのアトムを基点に DFS でリンクを辿りながら訪問したプロセスを先行順にバイト列として記録する. リンクを辿る際には各プロセスに対する先行順の訪問順序を作業領域へ記録しておくことで, バイト列として記録済みのプロセスを重複してエンコードしないようにしており, 循環構造を持つグラフをエンコードする際の停止性を保証している. バイト列へ記録済みのアトムへ再訪問が発生した場合には, TAG_ATOM_REF と共にアトムへの訪問番号とリンク番号を記録する. また, リンクはアトムを一対一に接続しているため, DFS で辿ってきた訪問元のアトムのリンクと接続しているリンク

```
procedure encode_membrane(mem)
  set_visited(mem)
  write_mem_start()
  encode_molecules(mem)
  encode_child_membranes(mem)
  encode_rulesets(mem)
  write_mem_end()
end procedure
procedure encode_molecules(mem)
  for all atom ∈ mem_atomset(mem) do
    if not visited(atom) then
      encode_atom(atom)
    end if
  end for
end procedure
procedure encode_child_membranes(mem)
  for all child_mem ∈ children(mem) do
    if not visited(child_mem) then
      encode_membrane(child_mem)
    end if
  end for
end procedure
```

図 3　圧縮アルゴリズムの疑似コード

が訪問先のアトムには必ず存在する．この場合には TAG_FROM のみを記録し, TAG_ATOM_REF を用いた際の記録メモリ量より少なく済むよう工夫を行っている．全ての分子のエンコードが終了したならば, *encode_child_membranes* を呼び出し, 分子をエンコードした際に未訪問な子膜を *encode_membrane* で再帰的にエンコードしていく．最後に膜に存在するルールの集合を 2 バイトの ID で記録する．ルールの複製や移動が発生した場合には複数の ID が存在することになるため, ID が一つの場合と複数存在する場合でタグを切り分け, より軽量化を図っている．

3.3　バイト列圧縮と状態展開処理

状態遷移グラフ構築処理に対して状態をバイト列へエンコードする処理を導入したことで, 状態管理表に登録した状態は全てバイト列として管理するものとなった．そのため, 図 2 に示した遷移先の計算手続き *expand* において, まずは *state_get_graph* 処理の際にバイト列から階層グラフ構造への再構築を行う．再構築した階層グラフ構造は遷移先計算の手続きが終了する際に破棄する．計算した遷移先の状態が出現済みの状態であるか否かを判定する手続き *state_space_insert* では, 比較対象となる状態のバイト列と階層グラフ構造とで同型性判定を行う．

バイト列と階層グラフ構造との同型性判定では, 比較先のバイト列を階層グラフ構造として再構築する必要がある．しかしながら, 階層グラフ構造全体を再構築してから比較を行う手段は時間面のコストが高い．そこで, バイト列から階層グラフ構造を再構築せず, バイト列から読み出したデータを比較元の階層グラフ構造から探索する手法を実装した．比較先のバイト列を直接グラフ走査するため, 必要に応じてバイト列の読出し位置はバックトラックする．そのため, 作業領域としてバイト列の読出し位置にバックトラックポイントを設定しながら判定を行っていく．

4　状態遷移グラフ構築処理の高速化

LMNtal モデル検査器の主な処理では, 遷移先状態の生成, ハッシュ値の計算，状態の等価性判定, バイト列へのエンコードといったグラフ走査に基づく高コストな処理が頻繁に行われており, 速度性能の最適化が求められていた．特に遷移先状態を生成する処理のコストが高く, 全体の実行時間のおよそ 50%を占めていた．本節では, より高速な処理を実現するために, アトムと膜に対して一意な整数 ID を付加することによるグラフ走査の高速化と, 書換え元のグラフ構造から書換え後のグラフ構造への差分情報に基づく新たな状態生成手法を開発し, 高速化を図った．

4.1　アトムと膜に対する一意な整数 ID の付加

遷移先状態を生成する処理のコストが高い主な要因は, 書換え元の階層グラフ構造を複製し, 複製した階層グラフ構造を書き換えることで複数の新たな状態を生成していることにある．プロファイリングの結果, 複製を始めとしたグラフ走査を行う際に作業領域

として使用する辞書の操作がボトルネックとなっていることが分かった. 辞書は, アトムと膜へのアドレス値をキーにしたハッシュ表を用いており, 作業領域をハッシュ表から配列へ変更することが高速化に繋がる. アトムと膜に対して一意な整数値を ID として付加することで, ID を添え字とした配列を辞書として利用できる. これまでは, アトムや膜に対して新たなデータ領域を追加することはメモリ使用量の観点から好ましくなかった. しかし, バイト列へのエンコード手法を開発したことで, バイト列に記録する必要のない情報は全て削減できるため, メモリ使用量へ悪影響を及ぼすことなく高速化に必要なデータ領域の追加が可能になった.

アトムと膜に対する一意な整数 ID は, 階層グラフ構造を複製する場合とバイト列から再構築を行う際に割り当てる. ID は, 一つの状態に存在するアトムと膜が一意な整数値を持つことを保証するが, 等価な階層グラフ構造が常に等価な整数値の持ち方になるとは限らない.

4.2 差分情報に基づく状態展開処理

書換え処理による階層グラフ構造の変化は, 階層グラフ構造全体のうちの一部分だけが書き換わる場合が多く, 階層グラフ構造全体を複製して書き換えることによる状態の生成には改善の余地がある. そこで, ルールによって書き換わる差分のグラフ構造の情報を記録しておき, 遷移先が既存の状態か否かを判定する際に, 差分情報の適用と逆適用を行い, 一つの階層グラフ構造を遷移元の状態や遷移先の状態へと変化させる手法を開発した.

4.2.1 差分情報の概要

差分情報の記録は, ルール適用時に書換え対象となるプロセスの探索が成功したとき, 中間語命令列によるプロセスの書換えを, 差分情報の記録に置き換えて実施する. 記録する差分情報は, 生成と消去が発生したアトム, 膜, ルールを膜単位で集合として管理している. シンボルアトムのリンクデータとして管理しているリンクの書換えには, 特殊な差分情報の記録を行う. リンクの書換えにおいて, 新たに生成したアトムのリンクを書き換える場合は, リンクを直接書き換える. そうでない場合は, リンクを書き換えるアトムを複製し, 複製したアトムのリンクを書き換える. 複製したアトムは生成アトムとして差分情報に記録するため, リンクの書換えもアトムの生成と消去の枠組みに沿った記録方式となる. これにより, 差分情報の適用操作が階層グラフ構造に対するプロセスの追加と除去の操作のみとなるため, 差分情報の逆適用も実現可能になる.

```
procedure expand_delta(base_s, delta_info)
  base_g ← restore_state_graph(base_s)
  succ_deltas ← gen_successor_delta(base_g)
  for all succ_d ∈ succ_deltas do
    dmem_commit(base_g, succ_d)
    new ← state_make(base_g)
    ret ← state_space_insert(new)
    add_successor(base_s, ret)
    dmem_revert(base_g, succ_d)
  end for
end procedure
```

図 4 差分情報を用いた際の遷移先計算処理

4.2.2 差分情報と状態展開処理

差分情報による状態生成手法の導入により, 図 2 に示す遷移先計算の手続き *expand* では, 遷移先の階層グラフ構造の集合を求める手続きが, 遷移先への差分情報を求める手続きへと変更している. 遷移元の階層グラフ構造は, この差分情報を適用することで一時的に遷移先の階層グラフ構造に書き換える. 書き換えた階層グラフ構造に基づき, 遷移先の状態が新規状態であると判定した場合は, 一時的に構築した階層グラフ構造をバイト列へエンコードし, エンコードしたバイト列を状態管理表へ登録する. 一つの遷移先状態に対する処理を全て終えたら, 一時的に書き換えた階層グラフ構造に対して差分情報を逆に適用することで遷移元の階層グラフ構造への復元を行う. これにより, 全ての遷移先状態の生成と検査を階層グラフ構造を完全に複製することなく実施できるため, ボトルネックであった階層グラフ構造全体の複製処理の省略が実現した.

図 4 に差分情報に基づく状態の遷移先を計算する手続き *expand_delta* を示す. 状態管理表に存在する状態は全てバイト列として管理しているため, まずは *restore_state_graph* によって階層グラフ構造 base_g を再構築する. 次に, 再構築した階層グラフ構造の遷移先への差分情報 succ_deltas を *gen_successor_delta* によって取得する. *dmem_commit* は, 遷移先への差分 succ_d をそれぞれ遷移元の階層グラフ構造 base_g へ適用し, 遷移先となる階層グラフ構造へ書き換える手続きを示す. 状態の生成と, 新規性の検査は図 2 と同様であり, 遷移後の階層グラフ構造に対するハッシュ値やバイト列を計算する. 必要な処理を終え, 遷移先状態の登録まで終了したならば, *dmem_revert* の手続きにより base_g に対する差分 succ_d の適用を取り消す.

5 並列モデル検査器への展開

LMNtal モデル検査器の速度性能向上を考える上で, 近年の CPU 開発がコア数の増加方向にシフトしていることを考慮すると, CPU の性能を最大限に利用するためにも並列処理機能の導入が重要となる. LMNtal モデル検査器は, 階層グラフ構造をバイト列へエンコードして状態管理する技術を導入したことでより大規模な検証問題を扱うことを可能にしたが, 64bit 環境の計算機が主流となりアドレス空間が拡大したこととともあいまって, 検証の長時間化はますます深刻な課題となっている. マルチコアプロセッサを複数個搭載した共有メモリ計算機が普及していることから, これを対象とした並列化が高速化手段として重要になると考え, 我々は SLIM を並列モデル検査器へと発展させることで更なる速度向上を目指してきた [34]. 本節では, LMNtal モデル検査器に導入した状態遷移グラフ構築処理の並列化手法について述べる.

5.1 モデル検査における並列化

並列処理による状態爆発対策としては, 分散メモリ環境を対象に大規模化と高速化を目的とした並列モデル検査の研究が取り組まれている [1] [5] [21]. 近年では共有メモリ環境を対象とした並列モデル検査の研究も取り組まれつつあり, DiVinE は共有メモリ環境を対象とした並列モデル検査器 [3] を公開し, Spin は並列化への拡張 [17] をリリースした. オートマトンベースの LTL モデル検査手法を並列化する際には, まず状態遷移グラフの構築を分割する手法が求められる. その上で, より複雑な性質 (活性など) を検証するために閉路探索の分割手法が求められる. 逐次最適な Nested-DFS アルゴリズムは DFS の後行順に基づくため並列化が困難 [22] であり, DiVinE では状態のハッシュ値に基づき状態遷移グラフをランダムに静的分割する BFS を採用し, 閉路探索の分割手法を複数提案, 実装している [2] [6] [7]. 既存の閉路探索の分割手法には得手不得手とする検証問題の特徴があり, アルゴリズムの改善や評価も重要な研究課題として取り組まれている [4] [33].

一方, DFS に基づき状態遷移グラフの構築を分割する手法としては, Stack-Slicing アルゴリズム [15] が挙げられる. モデル検査器 Spin の並列処理への拡張は, 安全性を検証する並列アルゴリズムとして Stack-Slicing アルゴリズムを採用しているが, 活性の検証は並列化を 2 スレッドに限定している. Spin は状態爆発対策として逐次処理向けに導入してきた技術を生かすことを重視しており, マルチコアプロセッサを複数個搭載した計算機を対象とした設計は行われていない. 近年の Spin の研究は, 損失圧縮法を用いた際に低下する網羅性を向上させる手段として並列処理を行う方向へシフトしている [18].

5.2 並列化の概要

本研究における LMNtal モデル検査器の並列処理機能の実装には POSIX Thread を使用し, 逐次処理のコード約 25000 行にスレッドセーフ化のコードを追加することで並列アルゴリズムを動作可能にしている. 例えば, 状態空間構築処理では状態から任意のタイミングでグラフ構造が破棄 (バイト列へ圧縮) されるため, 図 4 の疑似コード中の *restore_state_graph* の手続きでは再構築した階層グラフ構造をスレッド固有データとして扱うことで遷移先の計算を安全に並列処理可能にしている. 他にも, 遷移先の計算に必要となる作業領域や動的なメモリ確保と開放 (malloc / free) 処理のコスト軽減に必要なメモリプールをス

レッド固有データにするなどの対応を行っている．また, マルチスレッドによる並列処理を行う上で, 状態管理表であるハッシュ表と, メモリアロケータの処理が並行性の低下に繋がることを考慮する必要がある．モデル検査器 DiVinE ではメモリアロケータとしてマルチスレッド用メモリアロケータ Hoard[8] を採用しており, 現在 LMNtal モデル検査器はライセンスが同一のオープンソースソフトウェアであるマルチスレッド向けメモリアロケータ tcmalloc[13] を導入している．

LMNtal モデル検査器に対して並列アルゴリズムの導入を検討するにあたり, 状態管理に重要な役割を果たすハッシュ値の計算処理を考慮する必要がある．LMNtal における状態のハッシュ値は階層グラフ構造のハッシュ値であるが, 現在実装しているハッシュ関数は不得手とする特徴のグラフ構造があることが判明している．ハッシュ値に基づく分割法による並列効果はハッシュ関数の性能に依存することから, LMNtal モデル検査器の並列アルゴリズムとしては採用できない．そこで, 並列アルゴリズムとして Stack-Slicing アルゴリズムを採用し, 現在は閉路探索の分割手法の導入, 開発を目指している．

5.2.1 Stack-Slicing アルゴリズムの導入と拡張

Stack-Slicing アルゴリズムは, 全スレッドで状態管理表を共有し, 図 5 で示すように DFS をパイプライン分割するアルゴリズムである．実行に参加する全スレッドが状態を受信する Work Queue を固有に持ち, 図 6 のように通信方向を持った輪を論理的に構成する．DFS スタックの深さが一定の閾値 (Cutoff Depth) を超えた場合, 隣接スレッドの Work Queue へ未展開の新規状態を送信 (handoff) し, 各スレッドは Work Queue から取り出した状態を根にした DFS を繰り返す．状態を送受信する際に競合が発生しない利点がある一方で, DFS スタックの深さが状態遷移グラフに依存した静的分割法であることと, 状態管理表を共有して扱うことによる競合が問題となるため, 高い並列効果を出すためには工夫を施す必要がある．

LMNtal モデル検査器の並列処理では, Stack-Slicing アルゴリズムに対して, 動的負荷分散処理を拡張している．Stack-Slicing アルゴリズムは, 1 状態あたりの遷移数の割合が小さい場合や, 合流する遷移数, 遷移先を持たない状態数が多い場合に, 実行に参加するスレッド数によっては全てのスレッドに処理すべき状態がうまく行き渡らず, 扱う検証問題によってはコア数に応じた並列効果が得られない．そこで, 隣接スレッドがアイドル状態を主張した場合に Cutoff Depth を待たずに状態の送信を行う処理 (Work Sharing) と, アイドル状態になったスレッドが他のスレッドの Work Queue から状態を取得する処理 (Work Stealing) の追加によってこの問題の改善を図った．

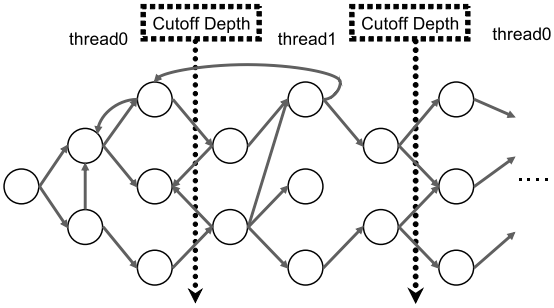

図 5 Stack-Slicing アルゴリズムによる状態展開のパイプライン分割

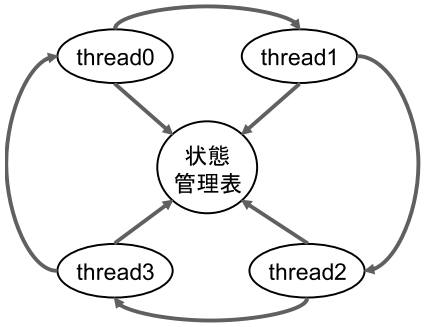

図 6 Stack-Slicing アルゴリズムにおけるスレッド間の関係

図 7 に, 動的な負荷分散処理を拡張した Stack-Slicing アルゴリズムによる状態空間構築処理の疑似コードを示す．*start_stackslicing* は, スレッドが自身の Work Queue (my_work_queue) から取得 (*dequeue*(my_work_queue)) した状態から DFS による状態展開処理 *dfs_parallel* を繰り返し行う手続き

```
procedure start_stackslicing()
  loop
    if termination_detection() then
      return
    else if empty(my_work_queue) then
      work_stealing()
    else
      stack_push(dequeue(my_work_queue))
      dfs_parallel()
    end if
  end loop
end procedure
procedure dfs_parallel()
  s ← stack_pop()
  expand(s)
  for succ ∈ new_successors(s) do
    if loadbalancing() then
      handoff(succ)
    else
      stack_push(succ)
      dfs_parallel()
    end if
  end for
end procedure
```

図 7 動的負荷分散処理を拡張した Stack-Slicing アルゴリズム

である. *dfs_parallel* は状態空間構築を行う手続きを表しており, DFS スタックから取得 (*stack_pop*) した状態 s に対して遷移先の計算 (*expand*) を行う. 求めた s の遷移先状態の中から新規状態を, スレッド自身の DFS スタックへ追加 (*stack_push*(succ)) するか隣接するスレッドへ送信 (*handoff*(succ)) するかを選択して処理を進める. このとき, 選択の条件式である *loadbalancing* は, DFS スタックの深さが閾値 (Cutoff Depth) を越えている場合と, 隣接するスレッドがアイドル状態を主張している場合に真を返す手続きである. また, *start_stackslicing* の手続きにおいて, スレッドは自身の Work Queue が空である (*empty*(*my_work_queue*)) 場合に, 他のスレッドの Work Queue から状態の取得を試みる (*work_stealing*()). *termination_detection* は終了検知を行う手続きを表し, Work Queue が空の場合に実施する. 全てのスレッドの Work Queue が空でありアイドル状態を主張している場合, 実行を終了する.

5.2.2 状態管理表の並列化

LMNtal モデル検査器において状態管理に使用するハッシュ表は, 同一の key に対する異なる複数の value(状態) をエントリのリストとして管理し, 登録した状態数と容量数の割合に応じて動的にテーブルサイズを拡張する.

状態管理表に対する状態の追加操作である手続き *state_space_insert* は, ハッシュ値の比較とグラフ同型性判定により等価な状態が存在するか否かの検査を行い, 存在しない場合はバイト列へのエンコードを経て状態を追加するため, ハッシュ表としては探索と追加にかかる実行時間のコストが高い. ハッシュ表の排他制御にはこれらの処理がなるべく互いに干渉しないような手法が求められる. モデル検査における状態管理表はハッシュ表ではあるものの, ハッシュ表に対する汎用的な操作であるエントリの探索, 追加, 削除といった単体処理は全て不要である. モデル検査では同じ状態が存在しない場合は追加を必ず行うという特徴がある. 本手法では, 頻繁に発生しないハッシュ表の拡張処理は逐次的に行うものとし, 探索処理を完全に非同期に動作させ, 追加操作時の同期のコストも小さく済むよう設計を行った.

図 8 に, 並列実行時の状態追加操作の概要を示す. 排他制御には, 2 種類のロック (E_Lock, W_Lock) を使用している. E_Lock はスレッドの数だけ用意したロックであり, 状態管理表を拡張する処理を逐次的に行うために使用する. また, W_Lock は状態管理表に対して状態を追加する際にエントリリスト単位で同期を取るために用意した定数個のロックである. 定数個であるため, 一つの W_Lock が複数のエントリリストの同期に使用される. 一つのロックを複数のエントリリストの同期に用いる手法は文献 [20] で提案されており, Java のライブラリとしても提供されている. 既存手法ではエントリリスト毎のロックを必要としな

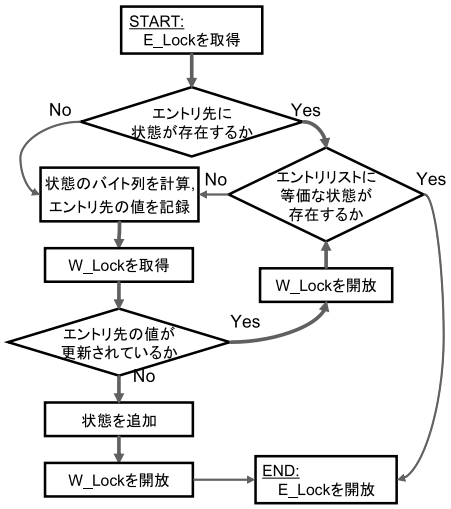

図 8　状態管理表に対する排他制御の流れ

いため, メモリ面のコストを抑えられることが利点がある. しかしながら, 状態爆発を起こした状態遷移グラフは遷移が合流することが多く, ハッシュ値の衝突が著しい場合にも並列性の低下が問題となる. 本手法では, 既存手法に対してロックの数を増やすことなく同期の粒度を細かくすることでこの改善を図った.

各スレッドは状態管理表に対する操作を開始する際に, スレッド固有のロック E_Lock を取得する. 状態管理表の拡張操作を行う際は, この E_Lock を全て確保することで同期を取っている. ハッシュ値に基づき算出したエントリリストに対して探索を行い, 等価な状態が存在しないことを確認した際にはまずバイト列へのエンコードを行う. エンコード終了後, リストを構成する末尾エントリの位置をエントリ追加予定先として記録しておき, 書込み用のロック W_Lock を取得する. W_Lock の取得後に記録しておいたエントリ追加予定先がリストの末尾であることに変わりがなければエントリリストの末尾に新たなエントリとして状態を追加する. 記録したエントリがリストの末尾でなくなっていた場合は, 他のスレッドと競合し, 先に更新が行われたことを表す. この場合は, W_Lock を解除した後にエントリリストに対する探索処理を再開する. このような処理を, 状態の追加に成功するか等価な状態を検出するまで繰り返していく.

設計した並列ハッシュ表は, 汎用的な機能を持たないものの,

- 探索が完全に非同期に動作可能
- 書込み用ロックの取得待ち時間が非常に小さい
- ハッシュ値の衝突により並列性が低下しない

ことが大きな利点となっている.

表 2　実験環境

CPU	AMD Opteron(tm) Processor2431
CPU 周波数	2.4GHz
コア数	12 (Six-core × 2 Processors)
Memory	32 GBytes
Cache Size	L1: 128KBytes (コア占有) L2: 512KBytes (コア占有) L3: 6144KBytes (6 コア共有)
GCC	Version 4.1.2 (最適化オプション -O2)

6　評価実験

本節では, LMNtal 実行時処理系 SLIM を拡張して構築したモデル検査器に対して第 3 節〜第 5 節に述べた大規模化, 高速化, 並列化を施した結果の評価を示す.

6.1　評価実験と結果の概要

実験環境は表 2 に示す 12 コアの共有メモリ計算機を使用した. 評価実験は状態遷移グラフの構築処理のみに要する実時間とメモリ使用量を対象に, 逐次の DFS を用いた際の

- メモリ使用量の削減率
- 逐次の速度向上比
- 並列化による速度向上比

の三項目を調査する. 並列化の評価実験では, 2, 4, 6, 8, 10, 12 コアを使用した並列実行による実時間を三度計測した平均値を基に, DFS による逐次実行の実時間からの速度向上比について評価する. 実験に使用する並列アルゴリズムは, Stack-Slicing アルゴリズムに対して動的負荷分散を行う場合 (O-SSA) と行わない場合 (SSA) の二種類である. なお, 分割の閾値 (Cutoff Depth) は 9 に固定して評価を行った.

実験に使用した問題と結果の一覧を表 3 に示す. 実験に使用した問題は全て公開中の SLIM に含まれており, メモリ使用量 1〜20 ギガバイト程度, 状態数が 100 万〜1 億 5000 万程度, DFS による逐次の実行時

表 3　実験に使用したインスタンス毎のデータ

Instance Name	# of States ($\times 10^6$)		Time (sec)		Memory (MB)		Ratio	
	Stored	Created	Use Enc.	Full Opt.	Orginal	Use Enc.	Speed Up	Compress
Mutex_18_p1	4.00	51.90	2765.02	110.65	7979	1587	24.99	20%
Mutex_19_p1	8.39	114.56	6408.83	268.25	17503	3500	23.89	20%
Mutex_18_p2	8.00	103.81	6322.12	220.49	16035	3190	28.67	20%
Mutex_19_p2	16.78	229.11	12849.19	533.07	*Unk.*	6969	24.10	—
Peterson_4_p0	0.90	3.25	247.79	6.13	2878	384	40.41	13%
Peterson_4_p1	1.75	6.67	491.03	12.10	5612	744	40.59	13%
Bakery_43_p0	0.87	2.96	225.89	6.84	2785	372	33.04	13%
Bakery_45_p0	3.36	11.47	934.37	26.32	10710	1409	35.50	13%
Bakery_43_p1	1.19	4.57	343.15	10.58	3813	510	32.44	13%
Bakery_45_p1	4.73	18.48	1343.80	41.64	*Unk.*	2000	32.27	—
PhiM_10_p0	3.21	38.79	3719.79	77.23	*Unk.*	1535	48.16	—
PhiM_11_p0	13.58	180.81	18827.29	400.12	*Unk.*	7067	47.05	—
Qlock_9_p0	1.97	3.95	492.17	11.05	13193	623	44.53	5%
Qlock_10_p0	19.73	39.46	5082.52	122.11	*Unk.*	6711	41.62	—
Qlock_9_p1	3.95	11.40	1128.35	28.91	*Unk.*	1265	39.03	—
Qlock_10_p1	39.46	114.42	13033.50	320.54	*Unk.*	13757	40.66	—
Lbully_11_p0	2.79	13.15	1405.37	32.43	14993	1353	43.34	9%
Lbully_12_p0	11.46	63.96	7316.60	173.16	*Unk.*	6014	42.25	—
Firewire_4_p2	2.00	21.30	7549.79	150.73	*Unk.*	1674	50.09	—
Byzantine_5a_p0	0.99	11.49	893.20	18.50	4414	411	48.28	9%
Byzantine_5b_p0	15.34	197.37	15659.62	319.79	*Unk.*	6416	48.97	—
Abp_64_p0	17.92	30.08	407.61	163.98	*Unk.*	2591	6.93	—
Abp_64_p1	153.60	368.64	2506.54	721.63	*Unk.*	23360	15.12	—
Queen_13_p0	4.75	6.54	407.61	15.65	11039	2977	26.04	25%
Queen_14_p0	27.72	38.12	2506.54	105.42	*Unk.*	18599	23.78	—
Jsp_3_p0	2.46	4.79	594.47	15.97	14727	1423	37.22	9%
Elevator_54_p0	3.23	11.95	1250.46	32.28	*Unk.*	1112	38.74	—
Elevator_63_p0	5.05	15.45	1678.85	47.95	*Unk.*	1846	35.02	—
Elevator_54_p1	10.37	44.23	4213.69	121.80	*Unk.*	3624	34.60	—
Elevator_63_p1	16.04	56.68	5629.35	174.97	*Unk.*	5926	32.17	—
Sstd_06_p0	2.35	15.63	1274.03	30.44	9681	1178	41.85	10%
Sstd_07_p0	15.96	123.36	10805.82	273.14	*Unk.*	9021	39.56	—

間が 250〜20000 秒程度の問題を 32 題選択した. 表中の列 Instance Name は問題名, # of States は状態数 (Stored が保存数, Created が生成数), Time は実行時間, Memory はメモリ使用量, Ratio は性能向上率に関する列を示す. Time の項目には, バイト列へのエンコードを使用した際の逐次の実行時間と, 逐次処理の高速化を行った上で 12 コアの並列処理による実行実行の二項目のみを示し, 逐次処理の高速化と並列化のそれぞれの速度向上比についてはグラフを用いて後述するため, 表 3 では割愛している.

表 3 の Ratio における Speed Up は逐次処理の性能最適化に加えて 12 コアを使用した際の速度向上比, Compress は使用メモリの圧縮率を示している. 結果として, メモリ使用量は少なくとも平均 13%へ削減, 逐次処理の速度向上比は平均 3.28 倍, 12 コアの並列実行による速度向上比は 10.21 倍となり, 従来の LMNtal モデル検査器の性能と比較して約 10 倍の軽量化と約 35 倍の高速化の実現を確認した.

6.2　メモリ使用量の削減率

各問題に対するバイト列への圧縮処理導入以前の使用メモリ量と導入後の使用メモリ量は, それぞれ表 3 中の Memory の項目における Orginal と Use Enc. の列で示す. 表中の *Unk.* はメモリ不足によって使用メモリ量が不明であったことを意味する. 圧縮前のメモリ量が *Unk.* であった問題を対象外とすると, 平

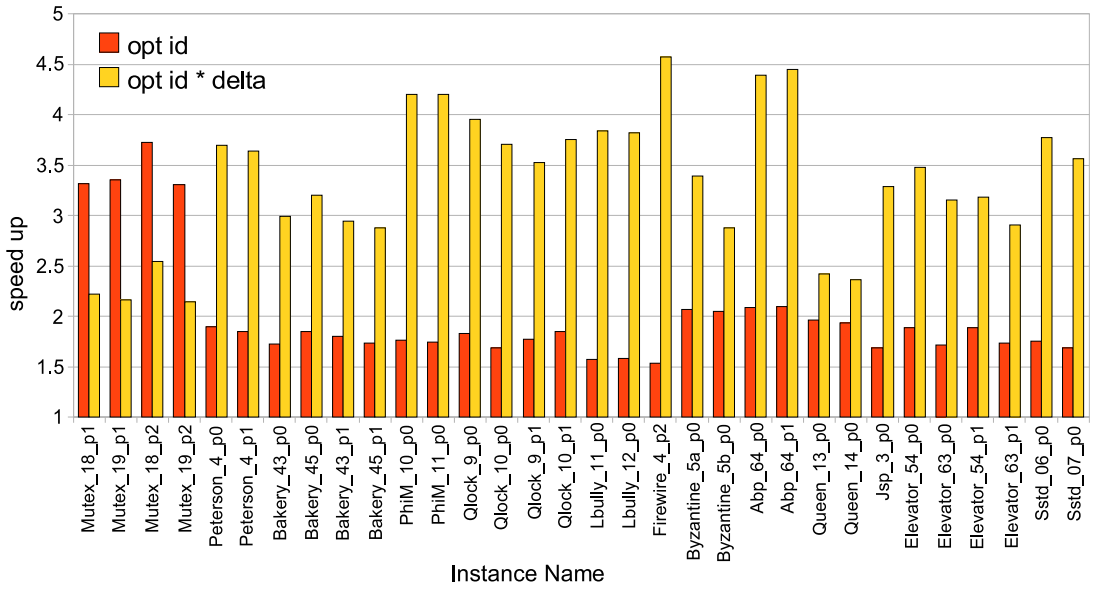

図 9　プロセス ID と差分生成による速度向上比

均の圧縮率は 13%となった．バイト列へのエンコードでは, 膜が多用されているほど高い圧縮率で圧縮していることが分かる．例えば, 問題 Firewire_4_p2 や Qlock_9_p1 はバイト列圧縮を利用した際のメモリ使用量が 1.5 ギガバイト程度だが, バイト列圧縮を利用しなかった場合は 32 ギガバイトで表現することができなかった．これらの問題は膜を多用してモデル記述した例題である．膜を多用することが, 検証可能な規模へ与える影響が小さくなったと言える．ユーザは膜を多用したことでメモリ不足に陥る問題を気にする必要がなくなるため, 記述性の向上にも繋がったと言える．また, バイト列への圧縮と階層グラフ構造への再構築に要する実行時間は, 全体の 5%未満であった．しかしながら, 逐次の速度性能を最適化によって, これらの処理が要する実行時間は, 全体の 10〜20%程度と無視できない計算コストになりつつあり, 現在我々は計算コストの削減を目指している．

6.3　逐次処理の速度向上比

各問題に対するバイト列への圧縮処理導入後の逐次実行時間は, 表 3 中の Time の項目における Use Enc. の列で示す．Full Opt. は速度性能の最適化に加えて 12 コアを使用した際の実行時間を示している．逐次処理の速度性能を最適化した結果を図 9 に示す．グラフでは横軸が問題, 縦軸は速度向上比を示す．各問題に示す 2 種類の凡例 opt id, opt id * delta はそれぞれ, アトムと膜に対して一意な ID を付加する手法を用いた際の速度向上比と, ID の付加に加えて差分情報に基づく状態生成手法を用いた際の速度向上比を示す．差分情報を使用した場合は平均 3.28 倍, 使用しなかった場合は平均 1.96 倍の速度向上が得られた．

多くの問題は期待通りの性能向上を得ることに成功した．特に Firewire_4_p2 は, 状態あたりの階層グラフ構造が巨大なこともあり, 差分情報による状態生成によって 4.57 倍と高い速度向上を得られた．しかしながら, Mutex の問題に関しては差分情報を用いることで逆に速度性能が低下していることが分かる．原因として, 状態の等価性判定が探索に基づくグラフ同型性判定であることが考えられた．グラフ同型性判定は, 探索によってグラフの対応付けを確認していくため, 等価な階層グラフ構造であっても, 管理されているアトムや膜の並び順序が計算量に大きく影響する．差分情報に基づき生成した状態と, 従来の複製した階層グラフ構造の書換えによって生成した状態で, アト

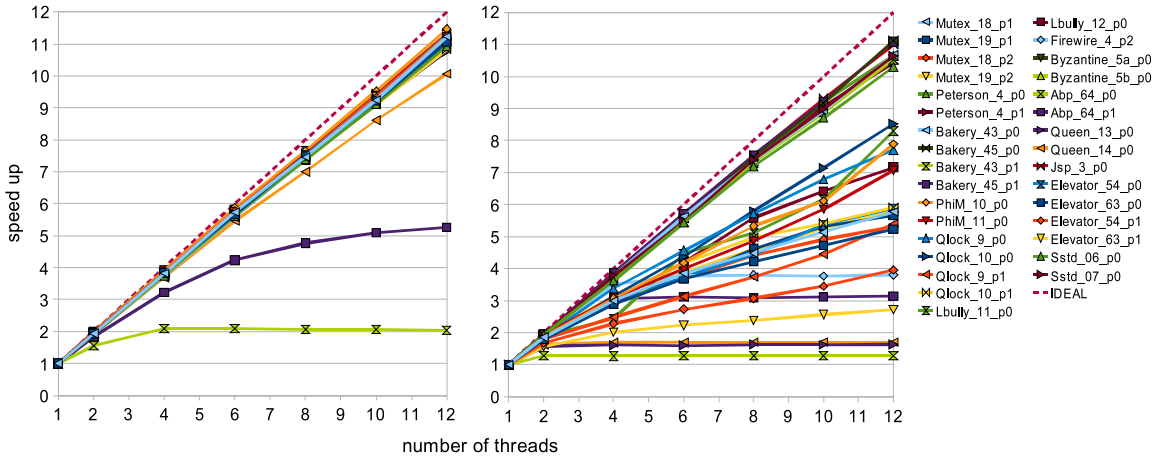

図 10　逐次 DFS からの N コア使用時の O-SSA(左) と SSA(右) の速度向上比

ムと膜の並びが異なっていたことが, 結果として実行速度に悪く影響したと考えられる. このような問題への対策としては, 予めアトムと膜を一意な並びで管理することが考えられる. 我々は, 既にアトムと膜を一意な順序に並べ直すことでエンコードしたバイト列を正規化し, 状態の等価性判定をバイト列同士の比較による線形の計算量で実現している [39]. しかしながら, 一意な順序に並べ直す計算コストが非常に高いため, 差分情報に基づくインクリメンタルなエンコード処理の実現による問題の解決を図っている.

6.4　並列化による速度向上比

図 10 に O-SSA と SSA による並列実行時の速度向上比を示す. 左のグラフが O-SSA, 右のグラフが SSA の結果であり, グラフでは横軸が使用したスレッド数, 縦軸が DFS による逐次実行からの速度向上比を示す. 凡例の IDEAL はリニアな並列効果の指標であり, グラフ中では破線で示されている.

SSA による速度向上比は, 12 コアで平均 5.96 倍という結果が得られた. 動的負荷分散処理を使用しない場合も, 問題によっては高い並列効果を得られているが, 一部の問題や 4 コア以上を使用した場合に並列効果が得られない問題が確認できる. これに対し, O-SSA による速度向上比は 12 コアで平均 10.21 倍と, 安定して高い並列効果を獲得することに成功した. 多くの問題は, より多くのコア数を使用した場合でも並列効果の獲得を期待できる. 一方, 一部の問題はコア数に応じた並列効果が頭打ちになっている. Abp_64_p0 と Abp_64_p1 は, 非決定性が少なく組合せ爆発を起こさない問題で, 状態遷移グラフの構築における並列性が低かったことが要因と考えられる. このような問題に対して, 状態単位で分割処理する既存の並列モデル検査手法では並列効果を得ることは難しく, 並列高速化を狙うためには新たな手法の開発が必要となる.

7　まとめと今後の課題

本論文では, LMNtal 実行時処理系 SLIM に対して, モデル検査器への拡張を経て, 大規模化と高速化を目指した並列モデル検査器への展開を行った過程と成果について述べた. 結果として, 検証可能な規模を約 10 倍拡大すると共に約 35 倍 (12 コア使用時) の高速化を実現し, LMNtal をモデル記述言語とした検証の有用性を大きく向上させることに成功した. 並列化手法として導入した Stack-Slicing アルゴリズムに対して, 並列処理可能な探索アルゴリズムを開発することは今後の課題である. また, 逐次性能の更なる最適化も課題としている. 現在我々は状態変化の差分情報を拡大利用することで軽量化と高速化を目指しており, グラフ同型性判定を始めとした処理性能を最適化で

きる見込みを持っている. 効果的な状態数削減手法である Partial Order Reduction の導入 [35] も検討しており, このような既存の状態爆発対策技術の導入や並列処理との効率的な連携も今後の大きな課題である.

謝辞 本研究の対象である LMNtal と LMNtal モデル検査器の研究開発に携わってきた早稲田大学上田研究室の OB の皆様, 並びに, 本研究で利用した実験マシンの管理者である同研究室 現 HPV 班の山根裕二氏に感謝の意を表す. また, 本研究における差分アルゴリズムの実装には同研究室 現言語班の中川遼平氏に助けを頂いた. 本研究の一部は, 科学研究費補助金 (特定 18049015) の補助を得て行った.

参 考 文 献

[1] Barnat, J., Brim, L,, Černá, I. and Šimeček, P.: DiVinE – The Distributed Verification Environment, in *Proc. 4th International Workshop on Parallel and Distributed Methods in verifiCation*, pp. 89–94, 2005.

[2] Barnat, J., Brim, L. and Chaloupka, J.: Parallel Breadth-First Search LTL Model-Checking, *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pp. 106–115, 2003.

[3] Barnat, J., Brim, L. and Ročkai, P.: DiVinE Multi-Core – A Parallel LTL Model-Checker, *Automated Technology for Verification and Analysis (ATVA 2008)*, Vol. 5311 of LNCS, pp. 234–239, Springer, 2008.

[4] Barnat, J., Brim, L. and Ročkai, P.: A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties, *Formal Methods and Software Engineering (ICFEM 2009)*, Vol. 5885 of LNCS, pp. 407–425, Springer, 2009.

[5] Barnat, J., Brim, L. and Stříbrná, J.: Distributed LTL Model-Checking in SPIN, in *Proc. SPIN Workshop on Model Checking of Software*, Vol. 2057 of LNCS, pp. 200–216, Springer, 2001.

[6] Brim, L. Černá, I., Krčál, P. and Pel'anek, R.: Distributed LTL Model Checking Based on Negative Cycle Detection, in *Proc. Foundations of Software Technology and Theoretical Computer Science 2001*, Vol. 2245 of LNCS, pp. 96–107, Springer, 2001.

[7] Brim, L., Cerna, I., Moravec, P. and Simsa, J.: Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking, Formal Methods in Computer-Aided Design, pp. 352–366, 2004.

[8] Bergerm, E., Mckinleym, K., Blumofe, R. and Wilson, P.: Hoard:A Scalable memory allocator for multithreaded applications, in *International Conference on Architectural Support for Programming Language and Operating System*, pp. 117–128, 2000.

[9] Clarke, E., Grumberg, O. and Long, D.: *Model Checking*, The MIT Press, 2000.

[10] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. and Talcott, C.: *All About Maude - A High-Performance Logical Framework*, Vol. 4350 of LNCS, Springer, 2007.

[11] Eker, S., Meseguer, J. and Sridharanarayanan, A.: The Maude LTL Model Checker, in *Proc. 4th International Workshop on Rewriting Logic and Its Applications (WRLA 2002)*, Electronic Notes in Theoretical Computer Science, Vol. 71, pp. 162–187, 2004.

[12] Gailly, J. and Adler, M.: zlib, `http://www.zlib.net/`.

[13] Google: google-perftools, `http://code.google.com/p/google-perftools/`.

[14] Holzmann, G.: *The SPIN Model Checker - Primer and Reference Manual*, Addison-Wesley, 2004.

[15] Holzmann, G.: A Stack-Slicing Algorithm for Multi-Core Model Checking, in *Proc. 6th Int. Workshop on Parallel and Distributed Methods in Verification (PDMC 2007)*, pp. 1–15, 2007.

[16] Holzmann, G.: State compression in Spin: recursive indexing and compression training runs, in *Proc. 3rd SPIN Workshop*, 1997.

[17] Holzmann, G. and Bosnacki, D.: The Design of a Multi-Core Extension of the SPIN Model Checker, *IEEE Transactions on Software Engineering*, IEEE Computer Society, pp. 659–674, 2007.

[18] Holzmann, G., Joshi, R. and Groce, A.: Swarm Verification, in *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, pp. 1–6, 2008.

[19] Holzmann, G., Peled, D. and Yannakakis, M.: On Nested Depth First Search, in *Proc. The Spin Verification System*, Vol. 32 of Dimacs Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, pp. 81–89, 1996.

[20] Herlihy, M. and Shavit, N.: *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

[21] Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R. and Gopalakrishnan, G.: Parallel and distributed model checking in eddy, in *SPIN'2006*, Vol. 3925 of LNCS, pp. 108–125, Springer, 2006.

[22] Reif, J.: Depth-first search is inherently sequential, *Information Processing Letters*, Vol. 20, No. 5, pp. 229–234, 1985.

[23] Sami, E. and Jean-françois, P.: Memory Efficient State Space Storage in Explicit Software Model Checking, in *SPIN'2005*, Vol. 3639 of LNCS, pp. 43–57. Springer, 2005.

[24] Thompson, S., Brat, G. and Venet, A.: Software Model Checking of ARINC-653 Flight Code with MCP, in *Proc. Second NASA Formal Methods Symposium*, 2010.

[25] Ueda, K.: LMNtal as a Hierarchical Logic Programming Language, *Theoretical Computer Science*, Vol. 410, No. 46, pp. 4784–4800, 2009.
[26] Ueda, K.: Encoding Distributed Process Calculi into LMNtal, Electronic Notes in Theoretical Computer Science, Vol. 209, pp. 187–200, 2008.
[27] Ueda, K: Encoding the Pure Lambda Calculus into Hierarchical Graph Rewriting, in *Proc. 19th International Conference on Rewriting Techniques and Applications (RTA 2008)*, Vol. 5117 of LNCS, pp. 392–408, Springer, 2008.
[28] Ueda, K., Ayano, T., Hori, T., Iwasawa, H. and Ogawa, S.: Hierarchical Graph Rewriting as a Unifying Tool for Analyzing and Understanding Nondeterministic Systems, in *Proc. Sixth International Colloquium on Theoretical Aspects of Computing (ICTAC 2009)*, Vol. 5684 of LNCS, pp. 349–355, Springer, 2009.
[29] Ueda, K. and Kato, N.: LMNtal: A Language Model with Links and Membranes, in *Proc. Fifth Int. Workshop on Membrane Computing (WMC 2004)*, Vol. 3365 of LNCS, Springer, pp. 110–125, 2005.
[30] Visser, W., Havelund, K., Brat, G. Park, S. and Lerda, F.: Model checking programs, in *Proc. 15th IEEE International Conference on Automated Software Engineering*, Grenoble, 2000.
[31] 綾野貴之, 堀泰祐, 岩澤宏希, 小川誠司, 上田和紀: 統合開発環境による LMNtal モデル検査, 第 11 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論文集, pp. 1–15, 2009.
[32] 小川誠司, 綾野貴之, 上田和紀: LMNtal を用いた状態空間探索, 第 23 回人工知能学会全国大会, 2H2-3, 2009.
[33] 川端聡基, 小林史佳, 上田和紀: 強連結成分の性質を用いた OWCTY モデル検査アルゴリズムの高速化, 第 24 回人工知能学会全国大会, 2E1-3, 2010.
[34] 後町将人, 上田和紀: LMNtal モデル検査器における状態展開処理の並列化, 第 6 回ディペンダブルシステムシンポジウム論文集, pp. 169–178, 2009.
[35] 佐々木隆之: LMNtal 状態遷移グラフの Partial Order Reduction, 早稲田大学大学院理工学研究科, 修士論文, 2009.
[36] 中島求, 加藤紀夫, 水野謙, 上田和紀: LMNtal 分散処理系の設計と実装, 日本ソフトウェア科学会第 21 回大会論文集, 2004.
[37] 広戸康平: LMNtal データ構造のハッシュコード化と同型性判定, 早稲田大学理工学部, 卒業論文, 2006.
[38] 堀泰祐, 佐々木隆之, 岡部亮, 村山敬, 上田和紀: LMNtal に基づくモデル検査器, 日本ソフトウェア科学会第 25 回大会論文集, 2008.
[39] 堀泰祐: LMNtal 実行時処理系 SLIM における検証機能の性能最適化, 早稲田大学大学院基幹理工学研究科, 修士論文, 2010.
[40] 村山敬, 工藤晋太郎, 櫻井健, 水野謙, 加藤紀夫, 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, Vol. 25, No. 2, pp. 47–77, 2008.