2011 年度 修士論文

並列モデル検査器 SLIM の
状態空間削減手法および
最適化問題向けの状態空間構築手法

提出日　：　2012 年 1 月 27 日
指導　　：　上田　和紀 教授

早稲田大学大学院基幹理工学研究科
情報理工学専攻

学籍番号：5110B037-0
川端　聡基

概要

社会基盤を構成するシステムにソフトウェア技術が浸透したことで, ソフトウェアの信頼性への要求が高まっている. また, システムは大規模複雑化しており, 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漏れのない検証することは容易ではない. そこで近年では, 形式手法の一つであるモデル検査が注目を集めている. モデル検査はシステムを状態遷移系で記述し, 網羅的探索を行う検証手法である. しかし, モデル検査は状態空間が巨大になりやすく, 探索にかかる時間と領域がモデルのサイズに対し指数的に増加する状態爆発問題を抱えている.

階層グラフ書換えに基づく並行言語 LMNtal を状態遷移系記述言語に持つモデル検査器 SLIM がある. LMNtal はプロセス, データ, メッセージを統一的に扱えることや, 並行処理を容易に記述可能であり, 様々なシステムや計算モデルが扱える. 近年, LMNtal の実行時処理系が拡張され, 状態空間構築機能, モデル検査機能を持った SLIM が開発された. これにより, LMNtal で記述したモデルの状態空間を構築, 検証することが可能になった. SLIM は並列化をはじめとし, 抽象化や状態圧縮など様々な状態爆発問題への対策がなされている. 特に 1 状態の表現の圧縮は重要で, SLIM は各状態を表す階層グラフをバイト列へ圧縮しているが, それでもなお使用する記憶領域の大半はバイト列が占めている.

本研究ではこの問題へ対処として, 状態空間圧縮手法である Δ-marking の SLIM への導入を行った. Δ-marking は状態を近い過去の状態からの差分で表現する手法であり, 記憶領域面でボトルネックとなっている階層グラフをさらにコンパクトに表現することが可能である. Δ-marking は元々 Colored Petri Nets で実現された手法であるので, 新たに LMNtal 上での Δ-marking を考案した. その結果, Δ-marking を使わない実行に比べ平均で実行時間が約 1.76 倍になり, 約 58% の記憶領域で状態空間を表現することが可能になった.

また, SLIM の状態空間構築機能のさらなる応用として最適化問題への適用を行った. まず, LMNtal で記述した状態遷移系の枝に非負の重みを導入した. 次に実行時処理系 SLIM を拡張し, 初期状態から目標状態への至るパスの中で重みが最小 (最大) となるパスを発見する最適パス検出機能を設計, 実装した. また, 最小パス検出時に, 既に発見された最小値より重みが大きくなったパスの探索を打ち切る枝刈り処理を導入した. 様々な最適化問題を LMNtal で記述できることを確認し, 記述した問題の一部を使い評価実験を行った. 今まで実行時間がかかり扱えなかった問題でも, 探索の枝刈りが発生すれば, 短い時間で実行が終了することを確認した.

Compaction of state space and state space search for optimization problems in the SLIM parallel model checker

Toshiki KAWABATA

Dependable software are needed, because the software technique struct a system constituting a social infrastructure. The system becomes complex nad large scale, and many system use parallel processing. It is difficult to verify the system without the leak by a testing and a simulation. In late years the model checking that is one of the formal method gathers attention. the model checking is the methods that is exhaustive search a state transition diagram that express the system behavior. However, Model checking is prone to state explosion. Time and memory that is required is easy to become huge.

There is SLIM model checker that has concurrent language LMNtal based on the hierarchy graph rewriting for a description the state transition diagram. LMNtal can express a lot of computation model, because LMNtal can handdle a process, data, and a message samely, and can describe concurrent processing easily. Late years, we expanded LMNtal runtime system, and developed SLIM that can construct state space and do model checking . SLIM has many technique for the state explosion problem that is parallelization, abstraction, state compression, etc. Minimize a state is important, so SLIM compress each state into byte sequence. However, byte sequence take the greater part of the memory.

Therefore, I introduced a state space compression techeique named ?-marking to SLIM. ?-marking is technique to express a state in the difference from a near past state. It is possible to express a hierarchy graph becoming the bottleneck in memory more compactly. Because originally ?-marking is made on Colored Petri Nets, I devised ?-marking on LMNtal newly. As a result, SLIM become able to express the state space by 58% memory exchanging for 1.76 times Execute time.

In addition, I made a new state space construction for the optimization problem as further application of SLIM. I introduced non-negative weight to state trantition diagram that was written by LMNtal. And I expanded SLIM to treat the weighted graph. SLIM became able to search smallest (maximum) path from initial state to an aim state with pruning. I confirmed that LMNtal express a lot of optimization problem. By experimenting some problem, I confirmed new SLIM can find the smallest path of the problem that cannot search by existing algorithm.

目次

第 1 章

はじめに

社会基盤を構成するシステムにソフトウェア技術が浸透し, 大規模複雑化している. また, 並行処理や並列処理が組み込まれたシステムが増加しており, その検証の重要性は高まっている. このようなシステムは, 非同期実行しているプロセス間のタイミングによるバグなど, 再現性の低い不具合が発生する可能性があり, テストやシミュレーションによって網羅的に検証することは容易ではない. そこで近年では, 形式手法の一つであるモデル検査 [6] が注目を集めている.

モデル検査はシステムの振る舞いを示す状態遷移系と, 時相論理などで記述された満たすべき性質を入力することで, 網羅的探索を行う検証手法である. 元々, システムの振る舞いの記述には専用の言語を使用していたが, 近年では Java や C++ で記述されたソフトウェアを直接的に検証するソフトウェアモデル検査 [20][26] の研究も取り組まれている. このようなモデル検査を行うツールの一つに, 並行言語 LMNtal[21] で記述したプログラムの振舞いを検査する並列モデル検査器 SLIM[34][30] が存在する.

LMNtal は, 階層グラフの多重集合書換えに基づく宣言型プログラミング言語である. LMNtal の言語モデル [25] は, プロセス, データ, メッセージを統一的に扱い可能であることや, 並行処理を容易に記述可能であることが特徴である. 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としており, これまで様々な計算モデルを表現することで LMNtal の高い記述力が示されている [22][23][32]. このような表現力の高さを持つ LMNtal に対してモデル検査を行うことができれば, 記述可能な様々な計算モデルの検証を容易に実現できる. このような背景から, LMNtal 実行時処理系を拡張することで, モデル検査器 SLIM が開発された.

一方, 状態空間の網羅的探索に基づくモデル検査法は, 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る. 近年の計算機の CPU

コア数が増加傾向にあることから, 大規模高速な検証を目指し並列モデル検査 [1][15] の研究が進んでおり, SLIM も共有メモリ環境での並列化がなされている. 他の高速化の改良としては, SLIM は差分書換えに基づく状態生成技術 [35] を導入している.

領域面の状態爆発問題対策としては, 抽象化が重要な研究分野であり, 等価な経路集団から代表経路のみを選択する Partial Order Reduction(POR)[19][4] を SLIM は導入している [31]. また, 対称性によって複数の状態を同一とみなす Symmetric Reduction も重要である. LMNtal は階層グラフ構造を状態, グラフ書換えを遷移とみなし状態遷移系を記述することにより, グラフの対称性に基づいた抽象化を自然に行っている. 反面, 1 状態のサイズが大きくなり, 領域面のボトルネックとなっている.

その他の領域面の対策としては, 状態のデータ圧縮アルゴリズム [14] や, 状態空間に保持する状態数を制限する State Caching 手法 [10] があり, SLIM にも状態のバイト列への可逆列圧縮技術が導入されている. それでもなお, SLIM が使用する記憶領域の大半は圧縮されたバイト列が占めている.

そこで, 本研究では, 状態空間圧縮手法 Δ-marking[7] の SLIM への導入を行った. Δ-marking は状態を近い過去の状態からの差分で表現する手法であり, 領域面でボトルネックとなっている階層グラフをさらにコンパクトに表現することが可能である. Δ-marking は元々 Colored Petri Nets[18] で実現された手法であるので, 新たに LMNtal 上での Δ-marking を考案した

また, ここまで述べてきたように SLIM には状態空間構築に関して様々な高速化, 大規模化がなされている. この SLIM の状態空間構築機能や, LMNtal の記述力のさらなる応用として, SLIM の最適化問題向けの拡張を行った. LMNtal で記述された状態遷移系を, 最適化問題向けの重み付き状態遷移系に拡張するとともに, SLIM の状態空間構築機能も重み付き状態遷移系を扱えるように拡張した. 初期状態から目標状態への至るパスの中で重みが最小 (最大) となるパスを発見する最適パス検出機能と, 最小パス検出時には, 既に発見された最小値より重みが大きくなったパスの探索を打ち切る処理を設計した.

本論文では, 要素技術である並列モデル検査器 SLIM, および状態遷移系記述言語 LMNtal の概要 (第 2 章) から, SLIM 上での Δ-marking 手法の設計と実装 (第 3 章), 最適化問題向けの状態空間構築手法 (第 4 章), 評価実験 (第 5 章) の順で述べてゆく.

第 2 章

並列モデル検査器 SLIM

本章では, 本研究の対象である並列モデル検査器 SLIM の概要として, モデル記述言語 LMNtal の言語解説, 実行時処理系からモデル検査器への拡張, SLIM に導入されている既存の状態爆発問題対策技術について述べる.

2.1 LMNtal 言語解説

SLIM がモデル記述言語としている LMNtal について述べる.

LMNtal は階層グラフ書換えに基づくプログラミング言語である. LMNtal の基本データ構造である階層グラフは, アトムを基本構成要素として, それをリンクと膜の二つの手段で構造化したものである. リンクはアトム同士を一対一に無向接続し, 膜はアトムの多重集合を構成する. 多重集合は入れ子にすることができ. どちらの構造も動的再構成が可能である. また, 膜は書き換え規則 (ルール) を持ち, これらによって構成される階層グラフ構造をプロセスと呼ぶ.

LMNtal は, このプロセスをルールに従い, 書き換えることによって計算を進めていく. 書き換え処理は, マッチングという書き換え可能なプロセスを探す処理と, プロセスを実際に書き換える処理に分けられる, 1 つのルールで書き換え可能な箇所が複数存在する場合や, 書き換え可能なルールが複数あった場合に, 計算に非決定性が生じる.

図 2.1 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)

$$
\begin{array}{lll}
P ::= \mathbf{0} & & (\text{空}) \\
\quad | \quad p(X_1, \ldots, X_m)\ (m \geq 0) & & (\text{アトム}) \\
\quad | \quad P, P & & (\text{分子}) \\
\quad | \quad \{P\} & & (\text{セル}) \\
\quad | \quad T :\!- T & & (\text{ルール}) \\
\\
T ::= \mathbf{0} & & (\text{空}) \\
\quad | \quad p(X_1, \ldots, X_m)\ (m \geq 0) & & (\text{アトム}) \\
\quad | \quad T, T & & (\text{分子}) \\
\quad | \quad \{T\} & & (\text{セル}) \\
\quad | \quad T :\!- T & & (\text{ルール})
\end{array}
$$

図 2.1 LMNtal の基本構文

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．LMNtal には略記法や拡張構文が数多く存在する [36] が, 詳しい説明は省く．

2.2 LMNtal 処理系と SLIM のモデル検査器への拡張

LMNtal には Java によって記述されたコンパイラと実行時処理系 LMNtal-Java [36], および C 言語によって記述された実行時処理系 SLIM が存在する．LMNtal プログラムの実行と検証の概要を図 2.2 に示す．

コンパイラはソースコードを LMNtal 抽象機械の中間命令列に変換する．どちらの実行時処理系もこの中間命令列を解釈実行することができるが, 後者にはすべての書換え方法を試みて全状態空間を構築する非決定実行機能と, 本機能を拡張した LTL モデル検査機能が備わっている．SLIM の非決定実行機能と LTL モデル検査器機能は共有メモリ計算機向けの並列化が施されており, 良好な並列効果が達成されている [30].

また, LMNtal プログラムの開発, 実行, 検証をサポートする統合開発環境 LaViT[27] の開発も行われており, UNYO-UNYO3G による LMNtal プログラムの実行状態の可視

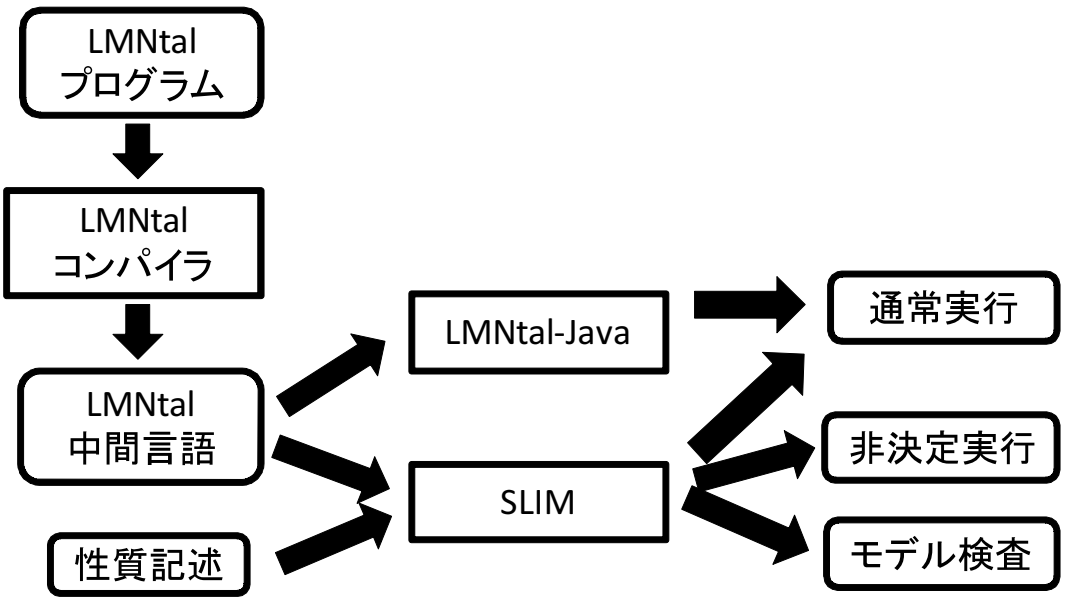

図 2.2 LMNtal プログラムの実行と検証の概要

化や, StateViewer による状態空間や検証結果の可視化が可能である. LaViT を利用することで, LMNtal プログラムの開発と検証をインタラクティブに実現可能であり, これまで, LMNtal をモデル記述言語としたモデル検査の有用性が示されてきた [24][28]. この場合, SLIM はバックエンドとして動作しており, 探索した状態空間や検証結果を LaViT 用の形式に合わせることで連携をとっている. SLIM の出力形式は LaViT 用だけでなく, グラフ可視化ツール Graphviz[11] 用の DOT 言語形式をサポートしている.

統合開発環境を含めたシステム全体の規模は 10 万行を超え, 一般公開されている[*1].

2.2.1 LTL モデル検査

モデル検査手法の一つに SPIN [12], 分散検証環境 DiVinE[1] などでも使用されているオートマトンベースの LTL モデル検査がある. これは要求される性質を線形時相論理 LTL (linear temporal logic) で記述し, Büchi オートマトンを用いて受理サイクル探索問題に帰着させる手法である. システムの各状態をノード, 状態変化をエッジとした状態遷移系を考える. システムの全実行を表すシステムオートマトンと, 検証する LTL 式の否定を表す性質オートマトンの同期積をとった積オートマトンを生成する. LTL 式から Büchi

[*1] http://www.ueda.info.waseda.ac.jp/lmntal/

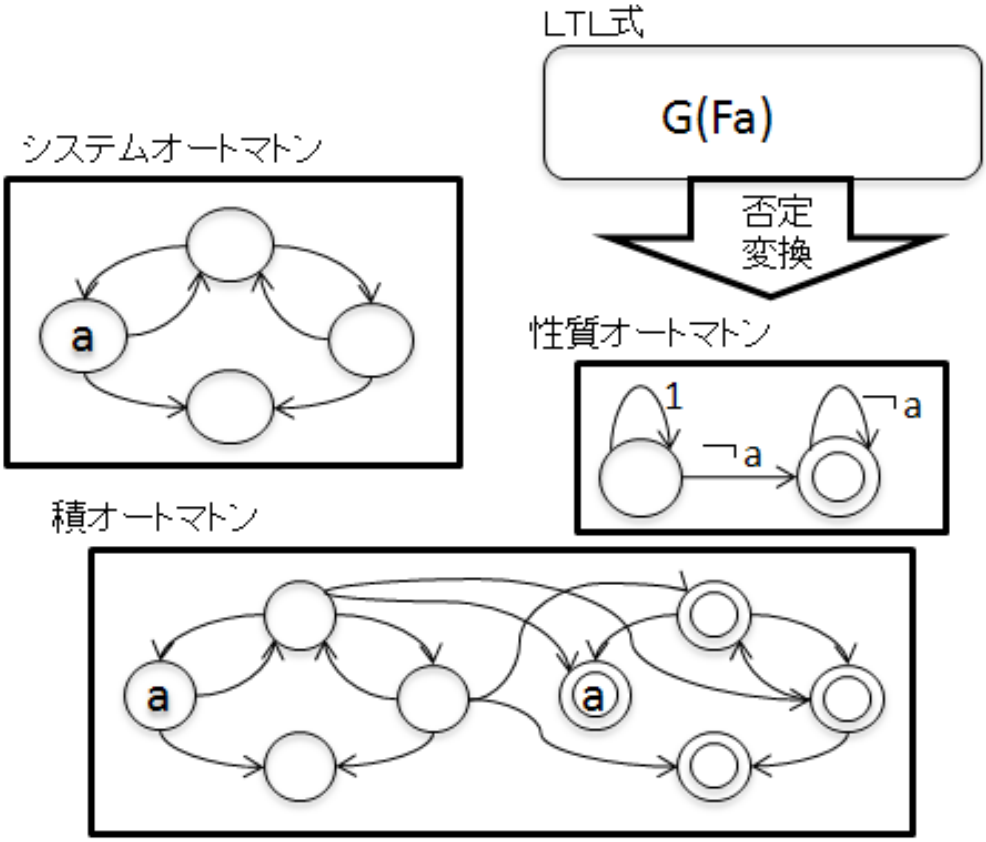

図 2.3　積オートマトン生成の概要

オートマトンへの変換は LTL2BA[9] などが使用される．そして, 受理状態への到達性や, 受理状態を含む閉路 (受理サイクル) への到達性を検査する．

同期積を取り, 積オートマトンを生成する概要図 2.3 に示す．積オートマトン上では, 性質オートマトン上で可能な遷移の数だけシステムオートマトン上で可能な遷移を展開していく．そのため, 性質オートマトン上で遷移に必要となる命題が成立しない場合は, 積オートマトン上の遷移は生成されない．一方, システムオートマトン上では可能な遷移が存在せず, 性質オートマトン上で遷移に必要な命題が成り立つ場合は, 同期積オートマトン上で ϵ 遷移 (stutter extensi on rule) を実行することで性質オートマトン上の実行状態を遷移させる．ϵ 遷移により, 正しく受理サイクルを形成できることが保障される．

一般的に状態管理には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

となる状態空間を構築する.

逐次の受理サイクル探索アルゴリズムとしては, 状態空間を構築しながら on-the-fly に二段階の深さ優先探索を行う Nested-DFS アルゴリズム [16] が最適であると知られている. Nested-DFS アルゴリズムは, まず一段階目の DFS が状態遷移系の探索を行い状態空間を構築する. 一段階目の DFS がバックトラックする際に二段階目の DFS が閉路の探索が始まる. 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, 既に探索済みの状態に対しては探索が重複しない. そのため, 状態空間のサイズに対して線形の計算量で探索が完了する.

並列の受理サイクル探索アルゴリズムとしては, One-Way-Catch-Them-Young(OWCTY) アルゴリズムや Maximal accepting predecessors(MAP) アルゴリズム [5] などが提案されている [2]. これらのアルゴリズムは検証するモデルによって得手不得手があり, 改良が研究されている. 近年, Nested-DFS の並列化も研究されている [8].

また, 性質オートマトンの強連結成分の情報を考慮することによって, 積オートマトンの受理サイクル探索アルゴリズムを効率化する手法も研究されており, 後方版 OWCTY を改良し, 新たに SCC-OWCTY アルゴリズムを考案したところ, 大幅な性能改善が得られた [29].

2.2.2 SLIM のモデル検査器への拡張

SLIM には, 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, 書換えを遷移とした状態遷移系を探索する非決定実行機能がある. 状態遷移系を探索し, 状態空間を構築する処理の概要を 2.4 に示す. まず, 未展開の状態に対し, 1 回のルール適用を網羅的に行うことによって次状態を生成する (状態展開). 次に, ハッシュ値の衝突判定とグラフの同型性判定によって, 生成した状態と全ての既出状態との間で等価性判定を行う (遷移先グラフの新規性判定). そして, 新規状態ならば, 状態空間に追加し, 新たな未展開状態として扱う. 既出状態であったならば, その状態への遷移を生成し処理を終了する. 状態空間は, チェイン法に基づくハッシュ表で表現しており, キーには階層グラフから求めたハッシュ値を使用している.

このように状態空間を生成し, さらに LTL によって性質記述を与えれば, SLIM は LTL モデル検査を行うことができる. SLIM の閉路探索アルゴリズムは, 逐次実行では Nested-DFS[16] を, 並列実行は簡易的な MAP アルゴリズムと OWCTY (One-Way-Catch-Them-Young) アルゴリズムの併用アルゴリズム [3] を採用している.

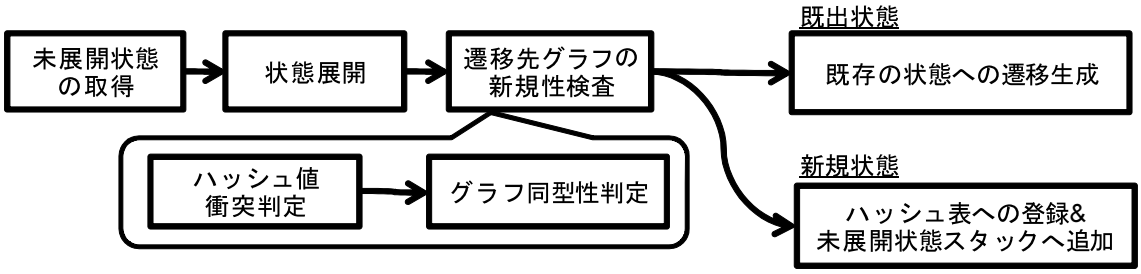

図 2.4 SLIM の状態空間構築処理概要

2.2.3 階層グラフのハッシュ値の計算

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は同じハッシュ値を生成する
- 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し, 実行時間が大幅に増加することがある. 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 [33] で提案され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, ベンチマークテストによって確かめられている. SLIM では D の値を 2 に設定し, 本手法を採用している.

2.2.4 階層グラフの同型性判定

同一ハッシュ値を持つ二つ状態の等価性判定はグラフ同型性判定 [34] によって行う.

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アトム, リンク, 膜, ルールが, 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. 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.

グラフ同型性判定は一般には NP 完全であるか否かも判明していない処理である. 階層グラフ構造の比較においては, 膜が形成する階層構造の同じ場所にリンクを持つ同名のアトムが多数存在する場合, 二つの階層グラフ間でそれらの対応関係のすべての可能性を探索する必要が出る場合があり, 高価な処理となる. 一方, 膜をもたない連結 LMNtal グラフ同士の比較は, 各アトムがもつリンクが順序づけられているため, もし二つのグラフがそれぞれユニークな（1 個しか出現しない）アトムを持つならば, それらを根とする走査を行えば, グラフの大きさに比例する手間で済む. 実際の LMNtal プログラムへのモデル検査においては, 高い対称性を持ち比較に手間のかかるグラフもあるが, 適切な根を選ぶことで効率よく比較できるグラフも多い. 更に, アトム, 膜, ルールの個数の一致のような定数時間で検査できる条件も利用することで, ハッシュ値が一致した場合に行われるグラフ同型性判定の高速化が図られている.

2.3 既存の状態爆発問題対策技術

SLIM は状態空間の構築処理や表現を様々な手法で改善している. 既存の手法の中から重要なものをここで述べる. なお, 各手法の詳しい説明は文献 [30][35] を参照して欲しい.

2.3.1 並列化

SLIM の状態展開は大規模化, 高速化を目指して共有メモリ環境向けの並列化がなされている. 並列化アルゴリズムとしては stack slicing[13] を採用している. stack slicing は実行に参加する PE (Processing Element) が通信方向を持った輪を作り, DFS をパイプライン分割するアルゴリズムである. 個々の PE は, 与えられた未展開状態から, ある定められた Cutoff Depth だけ探索し, それ以上の深さについては隣接 PE に探索を任せるアルゴリズムである. 展開する階層グラフ構造は PE 固有のデータとして扱い, 排他制御が必要なのはハッシュ表への状態の追加, およびハッシュ表の拡張時のみとなる.

ハッシュ表は全 PE で共有して利用されるため, 競合が発生しやすい. そのため, SLIM ではモデル検査に必要な同一状態の探索, 状態登録の 2 機能が高速に並列実行できるようにハッシュ表を設計している. 状態数が多い場合ハッシュ表の拡張が行われるが, これは稀なので逐次的に処理を行っている.

また, stack-slicing は静的な分割であり, 負荷が偏ることがある. そのため, SLIM では動的負荷分散を導入している. 隣接 PE が処理を行っていない場合, 探索している深さが Cutoff Depth 以下であっても未展開状態を送信する処理 (Work Sharing) と, 処理する未展開状態がない PE は他 PE から未展開状態を取得する処理 (Work Stealing) によって, 動的負荷分散を行っている.

図 2.5 に, 動的な負荷分散処理を導入した 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, 自身の Work Queue (my_work_queue) から状態を取得 (*dequeue*) し, DFS による状態展開処理 *dfs_parallel* を繰り返し行う手続きである. *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, PE 自身の DFS スタックへ追加 (*stack_push*) するか隣接する PE へ送信 (*handoff*) 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 PE 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, 自身の Work Queue が空である (*empty*) 場合に, 他の PE 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 PE 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.

2.3.2 階層グラフのバイト列表現

SLIM 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領域面のコストが高く, 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.

そこで SLIM では, LMNtal の階層グラフを再構築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実装している. バイト列にエンコードした状態は, 必要に応じて階層グラフの再構築処理を行っている. エンコードされたバイト列のサイズは元の階層グラフ

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

に依存するが, 公開版 SLIM[*2]に含まれているモデルでは 1 状態で数百バイト程度となっている.

同一なグラフからバイト列を生成しても, 処理系内部におけるアトムや膜の集合の具体表現 (内部表現) が異なれば, 生成されるバイト列は異なる. よって, 状態の等価性判定は,

[*2] http://code.google.com/p/slim-runtime/

バイト列からデコードされた階層グラフを同型性判定することによって行う.

階層グラフをバイト列にエンコードをするタイミングは, 展開された状態が等価性判定によって新規状態と分かり, 状態空間に登録するときである. バイト列をデコードするのは, ハッシュ値が等しい状態に対してグラフ同型性判定を行うときと, 状態展開時に階層グラフが既に破棄されているときである.

2.3.3 一意なバイト列表現

同一のグラフであれば, 内部表現が異なっていても同じバイト列を生成する手法がある. バイト列生成時に探索を行い, アトムや膜に一定の並び順を与えることによって, バイト列の一意性を保証する.

この手法を使用した場合, グラフ同型性判定は行われず, バイト列の比較によって状態の等価性判定を行う. ハッシュ値に関しても, 膜のハッシュ値の計算は行わず, 一意なバイト列から計算する.

また, 膜から計算したハッシュ値を使用しているときに, ハッシュ値の衝突が多く起こった場合, 同型性判定の回数が増え実行時間に影響を与えることがある. そのような場合, SLIM は一意なバイト列にエンコードし, ハッシュ値の再計算を行う. 再計算が起きた状態に関しては, 専用のハッシュ表に登録することで一貫性を損なわないようにしている.

2.3.4 差分情報を用いた状態生成

次状態の階層グラフは, 前状態の階層グラフを複製し書き換えることによって生成している. しかし, 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いため, 全階層グラフを複製するのは効率的でない. そこで, 書き換わるグラフの差分情報を利用することによって状態生成を高速化する手法がある.

まず, 元の階層グラフと次状態の階層グラフの差分情報を生成する. 差分情報としては, 各階層ごとに生成と消去が発生したアトム, 子膜 (その膜に所属する膜), ルールとリンクの変化を記録する. 生成された差分情報を元のグラフに適用することで次状態のグラフを生成する. 次状態について状態の等価性判定やバイト列エンコードを行う. その処理が終了したら, 差分の逆適用を行い, 元の階層グラフを得る. そして, 次の遷移についても同じ処理を行うことで, 網羅的に次状態を生成する.

第 3 章

SLIM 上での △-marking 手法の設計と実装

Δ-marking[7] は状態の差分表現を用いた状態空間圧縮手法である．まず, 全状態を状態遷移系の深さによって明示状態と差分状態に分ける．初期状態を深さ 0 としたとき, 深さがある定数の倍数である状態を明示状態, それ以外の状態を差分状態とする．また, この定数を Delta Depth と呼ぶ. 明示状態は通常通り保存するのに対し, 差分状態は明示状態からの差分で表現する．明示状態から差分状態への差分の取り方はモデル記述言語に依存する.

Δ-marking は元々 Colored Petri Nets 上で考案されたので, 今回新たに LMNtal での Δ-marking を設計した.

3.1 LMNtal での △-marking

LMNtal 上での Δ-marking について 2 つの手法を考案した．1 つ目は再展開による Δ-marking, 2 つ目は文字列差分による Δ-marking である.

再展開による Δ-marking とは過去に行った書き換え処理の情報を一部保存しておき, それに基づいてもう一度 LMNtal ルールの適用を行い, 状態を再構築する手法である．この手法では, どれだけ以前の書き換え情報を記録するかで差分情報のサイズやデコードの処理が変わる．例えば, 適用場所と書き換えルールを記録すれば, デコードの処理が一番少なくなる．しかし, LMNtal ルールは一般に階層グラフの任意の部分グラフを書き換える可能性があるので, これを記録するのは難しく, また差分情報が小さくなるとは限らない. 逆に, デコードで再計算する部分を多くすれば, 差分状態のサイズを非常に小さくするこ

とも可能である.

文字列差分による Δ-marking とは, エンコードされたバイト列の文字列としての差分を保存する手法である. この場合, 既存の Shortest Edit Script の計算手法などを使用することができ, 再展開による Δ-marking より高速に動作すると考えられるが, 再計算部分を大きくした再展開手法より, 差分状態のサイズは大きくなる.

3.2　再展開による Δ-marking

本論文では, 空間圧縮に主眼に置き, 再展開による Δ-marking を採用した. また差分状態を可能な限り小さくすることを目標にした.

ある状態 s を 1 ステップ展開して得られる状態を, 得られた順に s_1, s_2, ... としたとき, その番号を successor number とする. 今回実装した手法では, この successor number と状態 s への参照を差分情報として差分状態に記憶する. デコード時には, 状態 s の階層グラフを再展開し, successor number の番号に従い目標の階層グラフを取得する. Δ-marking を使用したときの状態空間のイメージを図 3.1 に示す.

この手法は以前一度試験的に実装されたことがあるが, その後導入された状態のバイト列圧縮技法と併用できないため, 現在の SLIM には残っていない. 併用できない理由は, 一度階層グラフをバイト列にエンコードした後, デコードを行うと, 階層グラフの内部表現が変化するからである. 内部表現とは, 処理系内部におけるアトムや膜の集合の具体表現を意味し, これが変化すると状態展開を行い全ての次状態を求めたときに, 次状態の生

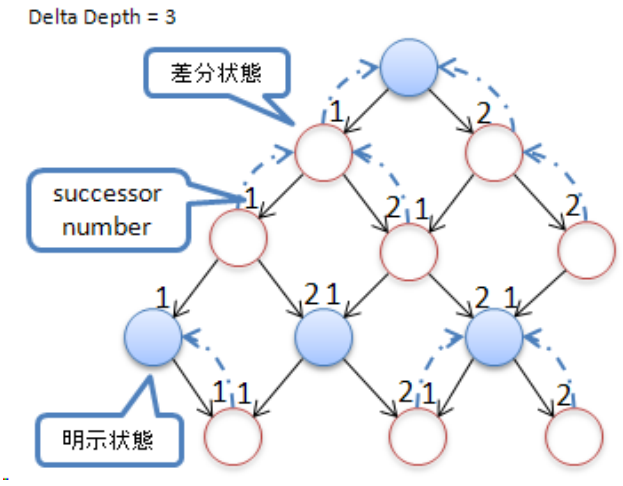

図 3.1　Δ-marking 使用時の状態空間

成順序が変化する．つまり, successor number が変化してしまうため, 上記の手法を使うならば, 階層グラフの内部表現を考慮しなければならない.

3.3 階層グラフの内部表現の一意性の確保

再展開による Δ-marking を現在の SLIM で実現するためには, 状態展開する階層グラフの内部表現を一意にする必要がある．明示状態の場合と, 差分状態の場合それぞれについて, 一意性の確保方法を述べる.

まず明示状態の場合は, バイト列をデコードして得られた階層グラフのみを展開に使用する．既存の処理では, 状態展開する際に階層グラフが破棄されていないと, その階層グラフを展開する．そこで, 状態展開には必ずバイト列をデコードした階層グラフを使用するように変更した．また, 同じ階層グラフのバイト列を複数生成しないようにした．つまり, ある階層グラフを一度エンコードしバイト列を生成したら, そのバイト列を実行終了まで保存する．バイト列デコードは, 実行状況によらず同じバイト列からは同じ内部構造をもった階層グラフを生成するので, 同一のバイト列を使用している限り, 明示状態の階層グラフの内部表現の一意性が保たれる.

次に差分状態の場合は, 前状態を書き換えた階層グラフをそのまま使用する．状態展開で生成される状態の内部表現も, バイト列のデコードと同じく, 実行状況などによらず展開する状態の内部表現に依存する．差分状態のデコードは状態展開によって行うので, 前状態の内部表現が一意であれば, 差分状態の階層グラフの内部表現は一意に保たれる．差分状態のデコードは, 明示状態の展開, あるいはそれによって生成された差分状態の階層グラフを再帰的に展開することによって行われる．つまり明示状態の内部表現が一意であれば, 差分状態の内部表現も一意となる.

3.4 差分状態のエンコードとデコード

再展開による Δ-marking での状態のエンコードとデコードについて述べる．なお, 明示状態のエンコード, デコード処理は既存の処理と同じものを使用している.

差分状態は前状態への参照と successor number を持つ必要がある．前者は SLIM では反例生成のために保存している．そこで, 差分状態は successor number と明示状態からの深さを記録するようにした. 明示状態からの深さは, 次状態を明示状態と見なすか, 差分状態と見なすか判定するのに使用する．1 状態からの出次数と Delta Depth が 256 以下ならば, 差分状態は 2 バイトで表現できる.

差分状態をデコードするには, 前状態からの再展開を行う. まず, 前状態の階層グラフの復元を行う. 前状態が明示状態であるならばバイト列デコードを行い, 差分状態ならば再帰的に差分状態のデコード処理を行う. 前状態の階層グラフ構造が得られたら再展開処理を行う. 復元する状態より successor number が小さい各次状態に関する処理は, 書き換え可能場所を探すマッチング処理のみ行い, successor number に対応する階層グラフを生成したら, 再展開処理を終了する. successor number が大きくなると, 再展開にかかる処理は多くなっていくが, 既存の状態展開よりは処理は少ない.

SLIM は並列化に際して, ハッシュ表を共有データとして扱い, 個々の階層グラフや状態展開などのための作業領域は PE 固有データとして扱っている. そのため, Δ-marking のデコードも PE ごとにローカルに計算することで, 並列動作が可能である.

3.5　状態空間圧縮率と実行時間の計算値

いくつかの仮定をもとに, 今回実装した Δ-marking の状態空間圧縮率とデコードの処理量を概算する. エンコードの処理は少ないので, デコードの処理量が実行時間に大きく影響する. まず, 状態空間圧縮率の計算値を求める.

SLIM での状態空間は以下の四つで表現している.

- StateDescriptor

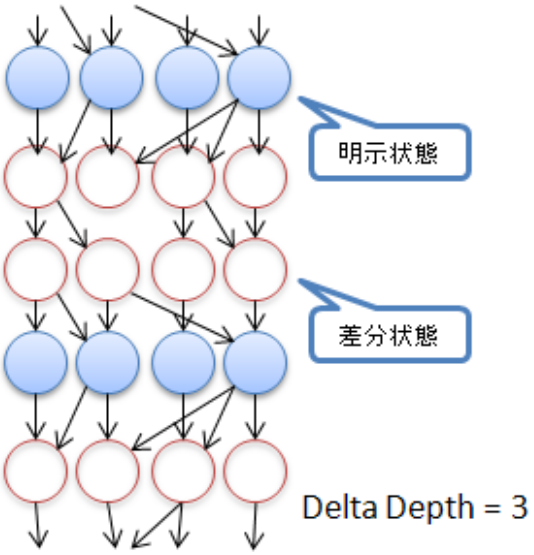

図 3.2　全ての深さで状態数が等しい状態空間

- BinaryString
- Transition
- HashTable

StateDescriptor は状態空間のノードを示し, 状態の ID やフラグ, 性質オートマトンのラベルや階層グラフへの参照を持つ. StateDescriptor は 1 状態につき 72Byte かかる. BinaryString は階層グラフをバイト列にエンコードしたもので, 各バイト列の大きさは階層グラフの大きさに比例する. Transition は遷移関係を表現するのに使用する領域である. HashTable はハッシュ表自体の占める領域である. これはハッシュ表のサイズに依存し, 拡張が起こらなければ一定である. モデルが大きくなると HashTable の占める領域は相対的に小さくなる.

本論文で提案する Δ-marking は差分状態のサイズが明示状態に比べ非常に小さい. よって, 差分状態のサイズを 0 とみなし, さらに二つの仮定を置くと状態空間圧縮率が概算できる. 一つ目は, 全ての階層グラフの大きさが等しいという仮定である. 二つ目は. 全ての深さで状態数が等しいという仮定である. つまり全状態のうちの $1/DeltaDepth$ が明示状態であるとする. この場合の状態空間のイメージ図を図 3.2 に示す.

Δ-marking を使わなかった場合に, 状態空間の占める全領域に対する階層グラフの表現の占める領域の割合を BSR (BinaryStringRatio) とする. 明示状態は状態数が $1/DeltaDepth$ になり, 差分状態はサイズ 0 とみなすので, バイト列全体の占める領域も $1/DeltaDepth$ となる. それ以外の占める領域は変化ないので, 状態空間圧縮率が式 3.1 で概算できる.

$$(1 - BSR) + BSR \times \frac{1}{DeltaDepth} \tag{3.1}$$

次にデコードの処理量の概算を行う. 上と同じように, 全ての深さで状態数が等しいと仮定を置く. また, 深さ 1 の再展開にかかる処理量がバイト列のデコードにかかる処理量の r 倍だとする. 明示状態からの深さが k の差分状態は, 全状態の $1/DeltaDepth$ だけあり, そのデコードの処理量はバイト列デコードの $r \times k/DeltaDepth$ 倍となる. よって, 式 (3.2) で概算できる. r が大きいと考えると, デコードの処理量は Delta Depth に比例して増加する.

$$\begin{aligned}
&\frac{1}{DeltaDepth} + \sum_{k=1}^{DeltaDepth-1} \frac{r \times k}{DeltaDepth} \\
&= \frac{1}{DeltaDepth} + \frac{r \times (DeltaDepth - 1)}{2}
\end{aligned} \tag{3.2}$$

第 4 章

最適化問題向けの状態空間構築手法

SLIM は元々 LMNtal の実行時処理系として開発された．その後, 実行方式を拡張し, 非決定実行やモデル検査機能という, 状態空間を扱う機能が導入された．また, これらの処理は大規模高速化を目指して並列化されている．今回, 最適化問題向けの状態空間構築, および最適化を行う値の伝播処理を並列動作可能に設計, 導入した.

本節では, SLIM に導入した最適化問題向けの状態空間構築処理と, その並列化手法について述べる.

4.1 最適化問題

まず, 今回扱う最適化問題の定義について述べる．状態遷移系 $G(S, T, s, A, c)$ が与えられる．S は状態集合, $T \subseteq S \times S$ は遷移関係の集合, $s \in S$ は初期状態, $A \subseteq S$ は受理状態の集合, c は T から非負整数の重みへの関数を表す．このとき, 状態と遷移関係の交互系列 $P = s_0, t_0, s_1, t_1, s_2, ..., s_{k-1}, t_{k-1}, s_k$ が $s_k \in S$ であり, かつ全ての $i = 0, ..., k-1$ において以下を満たすとき, P を s_0 から s_k へのパスと呼ぶ．また, t_{i+1} から見て t_i を前状態, t_i から見て t_{i+1} を次状態と呼ぶ.

- $s_i \in S$
- $t_i = (s_i, s_{i+1})$

パス P の重みを $\sum_{i=0}^{k-1} c(t_i)$ としたとき, 各状態の最適な重みを以下のように定義する.

定義 4.1.1 (状態の最適な重みの定義) s の重みを 0 とする．$u \in S/s$ の重みを, 初期状態 s から u に至る全てのパスの中で, 最適 (最小, または最大) なパスの重みとする.

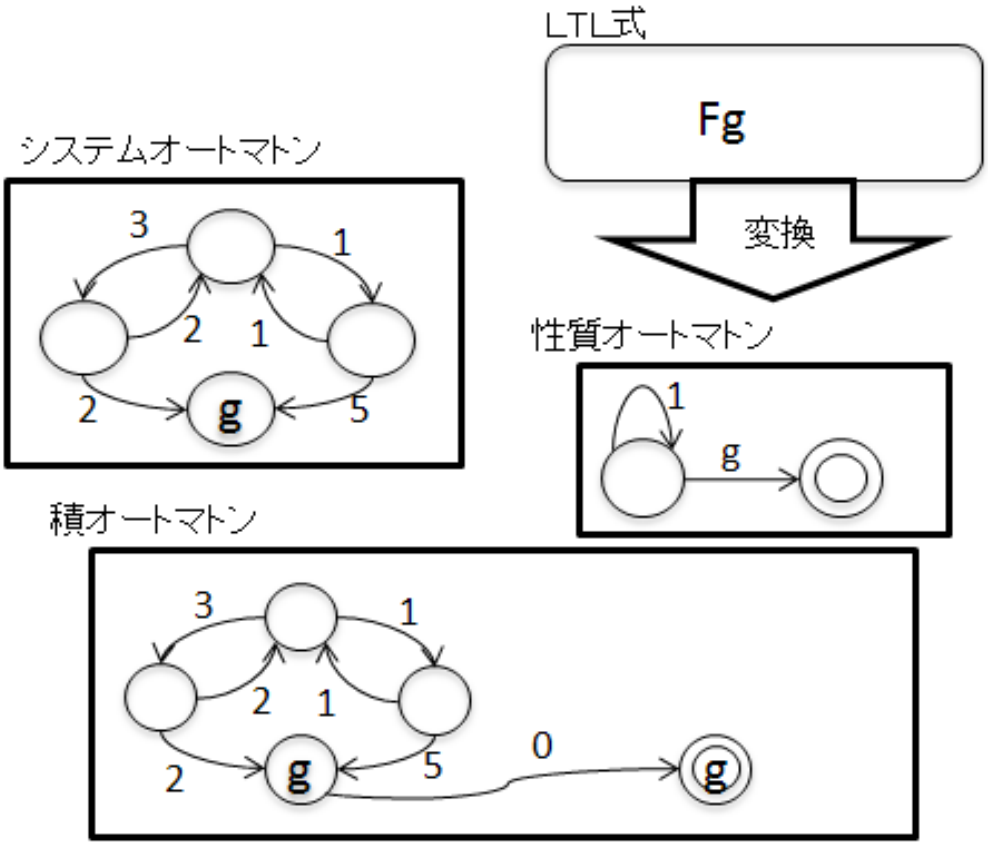

図 4.1 最適化問題の状態空間の概要

今回扱う最適化問題を以下のように定義する.

定義 4.1.2 (最適化問題の定義) 全ての $g \in A$ の重みの中で最適なものを発見する. 或いは s から g へのパスが存在しないことを決定する.

4.2 LMNtal での最適化問題の記述

LMNtal で最適化問題を記述することを考える.

まず, 状態 $u \in S$ と遷移 $t \in T$ はモデル検査と同様に階層グラフとその書き換えで実現できる. 初期状態 s は初期の階層グラフであり, 目標状態 A は受理状態集合で実現できる. 目標状態を入力するために, 問題の性質を LTL で記述する. モデルの状態遷移系を表すオートマトンと LTL 式を変換したオートマトンの同期積をとった積オートマトンを生成し, 探索はこの積オートマトンに対し行う. 同期積の計算は, 生成した状態がそれぞれ目

標状態であるかを検査する処理に対応する．遷移の重み関数 c は既存の SLIM の実行上では定義されていない．

探索する状態空間の概要を図 4.1 に示す．システムオートマトンと積オートマトンの全ての辺に重みが付き, システムオートマトンを探索する場合は, 全状態の重みを計算して終了する．積オートマトンを探索すれば, 受理状態の中で重みが最適なものを探索する．ϵ 遷移の重みは 0 とするので, 図 4.1 の受理状態へのパスの最小の重みは 6, 最大の重みは閉路があるので無限大となる．

既存の LMNtal でもプログラム内部で表現することは可能である．LMNtal モデル内に最適化する値 (最適化値) を記述し, 階層グラフの書き換え時にその値を必要な分だけ加算することで．記述できる．このように LMNtal 内部で c を記述した場合, 既存の SLIM で最適化問題を扱うには二種類の方式がある．1 つ目の方式としては, モデル内に最適化する機構も記述する方式である．つまり, LMNtal プログラムで扱いたい問題の最適化手法を記述する方式である．2 つ目は, 目標値を決め, その目標を達成するパスがあるか否かという決定問題として扱う方式である．この場合は SLIM のモデル検査機能を使用する．順次目標値の制約を緩和しながら行い, 初めてモデルが性質を満たしたとき, そのときの目標値が最適値である．

どちらの方式にもデメリットが存在する．1 つ目の方式は, ユーザーが各問題の最適化手法を自ら考え記述しなければならない．2 つ目の方式では何度も同じ状態空間を構築する必要がある．また, どちらの方式であっても最適化する値がモデルの内部に存在する問題がある．SLIM は最適化値のみが異なる二つ状態を, 異なる状態として状態空間に登録するので, その状態へ至るパスが多数存在し, その最適化値が異なれば, 状態数は爆発的に増加する．例として, ナップザック問題について, 得られる利得 (重み) をモデル内に記述した場合としていない場合の状態空間を図 4.2, 4.3 に示す．

上記の問題を解決するには処理系に 2 つの拡張を加えなければならない．1 つ目は最適化値をグラフ構造とは異なる場所に保存すること, 2 つ目はその値を実際に最適化する処理である．これからこの拡張について述べる．

なお, LMNtal プログラム内部で遷移の重みをどう記述するかは今回定めていない．コンパイラから生成された中間命令列に直接重みを記述し, 処理系に読み込んでいる．また, 今回は書き換え規則 (ルール) 毎に固定した重みを採用しており, どんなグラフを書き換えても同一ルールで書き換えたならば, 遷移の重みは等しくなる．LMNtal プログラムの仕様と, 中間命令列の仕様は, 今後最適なものを考えなければならない．

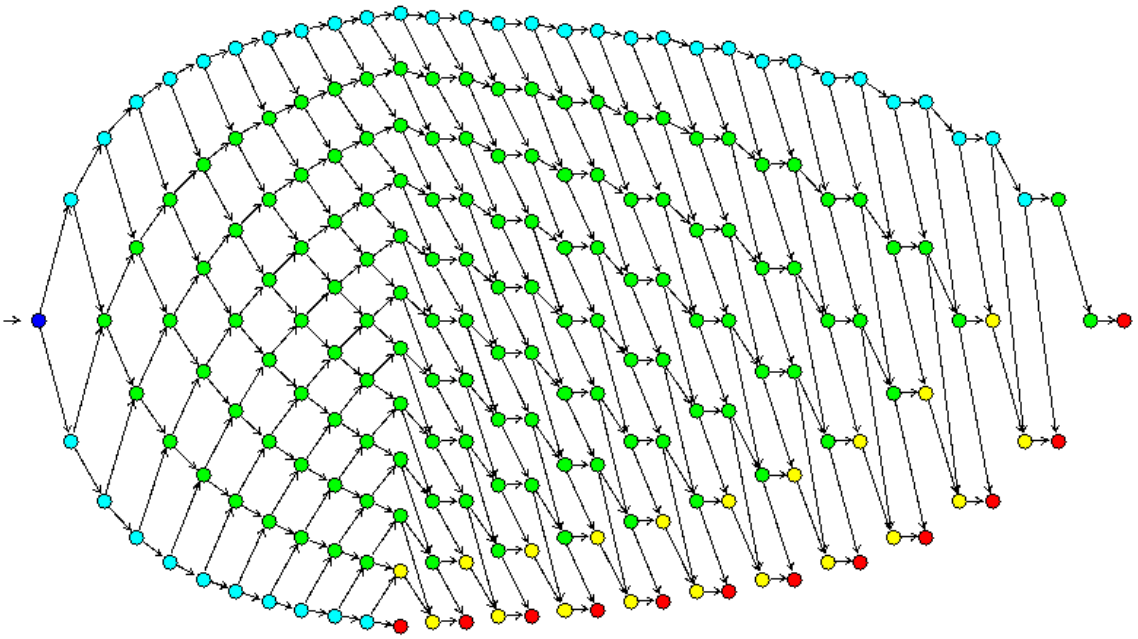

図 4.2　利得をモデル内部に記述したナップザック問題の状態空間

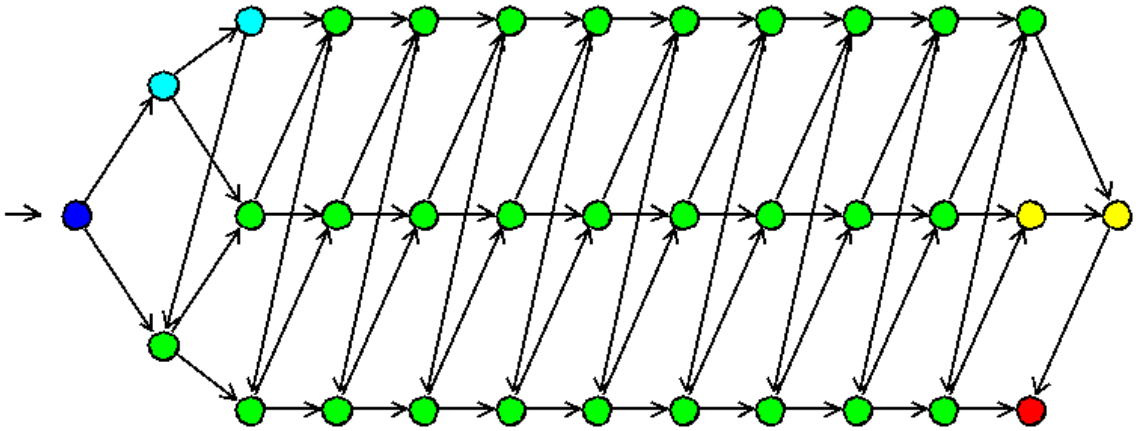

図 4.3　利得をモデル内部に記述していないナップザック問題の状態空間

4.3　逐次処理の拡張

まず, 新たに実行中に記憶する情報を述べる. 新たな情報は, 各状態の重みの情報とモデル全体で最適な重みの情報である. 各状態は, 既に発見されているその状態に至るパスの

中で最適なものを保持する. 具体的には, そのパス上の前状態への参照と, そのパスの自状態までの重みを記憶する. モデル全体で最適な重みの情報としては, 既に発見されている受理状態の中で最適な重みを持つ状態への参照である.

SLIM の逐次処理の中で拡張した部分について述べる. 変更点は 3 つあり, 次状態の重み更新処理, 更新伝播処理, 最適な重みの更新処理である.

まず 1 つ目の変更点は次状態の重み更新処理である. 状態展開を行い次状態を生成した後, 適用したルールから遷移の重みを計算する. 遷移の重みと展開した状態の重みの合計値が次状態の重みの候補となる. 次状態が新規状態であるならそのまま候補の値を書き込み, 新たな未展開状態として登録する. 次状態が既出状態であるならば, 既存の最適パスとの比較を行い, 候補値の方が適していれば更新をする. 更新した次状態が展開済みであれば, 伝播を行う状態として登録する.

2 つ目の変更点は更新伝播処理である. 重みを更新した展開済み状態は, その状態の次状態全てに対し, 上記の重みの更新処理を行う. 更新した次状態は重みの更新処理と同じく, 展開済みの場合のみ, 伝播を行う状態として登録する.

3 つ目の変更点はモデル全体での最適な重みの更新処理である. 展開した状態や, 更新伝播した状態が受理状態であった場合, 現在最適な状態の重みと比較し, より適していればモデル全体での最適な重みを更新する.

今回, 重みが最大のパスを発見する際に, 閉路があることを考慮していない. 初期状態から目標状態へのパス上に閉路があり, 閉路上の全ての遷移の重みが 0 でない限り, パスの重みは無限大となる. 今回はこのような状態遷移図がこないことを想定しているが, この問題に対処するならば, 閉路を検知する必要がある.

4.4 拡張した逐次処理の改良

節 4.3 により SLIM で最適化問題を解くことは可能となった. しかし, 性能面の改善が必要と考え, 2 つの処理を改良した.

1 つ目は, 探索の枝刈りである. 重みが最小なパスを探す際に, すでに見つかっている中で最小の重みより, 重みが大きい状態については探索する必要はない. なぜなら, 重みは非負であるので, そのパスの重みが最小になり得ないからである. よって, そのような状態については状態空間に登録だけしておき, 展開や次状態の重み更新を行わないように変更した. その状態の重みが更新され, 最小の重みより小さくなった場合は, 展開や更新伝播の処理が行われる.

2 つ目は, 更新伝播処理と状態展開処理の順番である. 今まで未展開状態の集合をス

タックで扱ってきた. だが, 最適化向けの状態空間構築では更新伝播処理より展開処理を優先した方が, 全体の処理が少なくなると考えられる. なぜなら, 更新伝播処理を先に行うと, ある状態の重みを何度も更新場合, 更新伝播処理を何度も行い, 最後の一回以外は無駄な処理になるからである. この問題に対応するために, 両端キューの実装を行った. 状態の取得は一方向のみにし, 状態の追加する方向を, 未展開状態と更新伝搬する状態で場合分けすることによって, 状態展開より更新処理を優先することが可能となった.

4.5 並列化の概要

SLIM は状態爆発問題対策として, 並列化がなされている. 最適化問題についても状態爆発問題が懸念されるので, 大規模高速化を目指し, 状態空間構築を並列動作可能に設計した. 並列動作をする上で問題になったのは, 各状態の重みや最適な重みの更新処理と更新伝播処理である.

重みの更新処理はロックを用いた排他制御を行った. 領域面のコストを考え定数個のロックを使用する. 一つのロックを複数の排他制御に用いる手法 [17] は, SLIM の並列ハッシュ表でも用いられている [30]. 各状態のハッシュ値から使うロックを決定し, 更新作業を排他的に行う. 最適な重みに関しては, 各状態の重み更新用のロックとは別に, ロックを 1 つ用意し排他制御を行った.

更新伝播処理は, 状態を更新した後に即時に行うわけではない. よって, 重みを更新した状態を全て, 更新伝搬する状態として両端キューに追加すると, 同じ状態に対し何度も更新伝搬処理を行うことになる. よって, 更新伝搬を行うか否かはは *update* フラグによって管理する. 重みを更新した状態は更新伝播処理を行う *update* フラグが真となり, *update* フラグが真の状態のみ更新伝播処理が行われる. 更新伝播を行ったら *update* フラグを偽にする.

しかし, 並列動作時は状態の重みを更新してから更新伝播処理の間に他の PE が *update* フラグを偽にすると, 最適値が伝播しない場合がある. 上記の更新処理で用いたロックを伝播時にも使用するとデッドロックが発生する可能性がある. そこで, ロックフリーに更新伝搬処理を設計した. *update* フラグを真にするタイミングを重みの更新の後, *update* フラグを偽にするタイミングを重みの伝播前にすることで, 更新された重みが伝搬されることなく *update* フラグが偽になることはなくなり, 更新漏れは起きなくなった.

図 4.4 に, 最適化問題向けに拡張した状態空間構築の並列アルゴリズムの疑似コードを示す. これは図 2.5 の $dfs\_parallel$ を拡張したものである.

$costed\_dfs\_parallel$ は最適化問題向け状態空間構築を並列に行う手続きを表しており,

```
procedure costed_dfs_parallel()
  s ← deque_pop()
  if compare_between_optimized_state(s) then
    return
  else if not_expanded(s) then
    expand_with_cost(s)
    successors_cost_update(s)
  else if is_update(s) then
    successors_cost_update(s)
  else
    return
  end if
  if is_accepted(s) then
    optimized_state_update(s)
  end if
  for succ ∈ updated_successors(s) do
    if loadbalancing() then
      handoff(succ)
    else
      deque_push(succ)
      dfs_parallel()
    end if
  end for
end procedure
```

図 4.4 最適化問題向けに拡張した状態空間構築アルゴリズム

まず両端キューから取得 (*deque_pop*) した状態 s に対して, 現在最適な重みを持っている状態との比較 (*compare_between_optimized_state*) を行う. 重みが最小なパスを探す際に, すでに見つかっている最小の重みより, 重みが大きい状態の処理はここで終了する. 次に未展開状態か検査し (*is_expanded*), 未展開状態は遷移先の計算 (*expand_with_cost*) を行う. このとき, 全ての遷移の重みを計算する. 遷移先の状態について, 可能ならば重みの更新を行う (*successors_cost_update*). 展開済み状態の場合, 次状態の重みの

更新が必要か検査し (is_update), 必要ならば状態展開後と同様に重みの更新を行う. $successors\_cost\_update$ は処理する次状態ごとに排他制御を行う. これらの処理を行った状態に対しては, 受理状態であるか検査をし ($is\_accepted$), 受理状態ならば最適な重みとの比較と更新を排他制御を用いて行う ($optimized\_state\_update$). 求めた s の遷移先状態の中から未展開状態や重みを更新した状態を求める ($updated\_successors$), 求まった状態を自 PE の両端キューに追加 ($deque\_push$) するか PE が行うか, 隣接 PE に送信する ($handoff$) か検査する ($loadbalancing$). $deque\_push$ は未展開状態が先に処理されるように, 両端キューのしかるべき方向から追加する.

第 5 章

評価実験

本章では, 大きく分けて二つの評価実験について述べる．一つ目は SLIM 上に実装した Δ-marking の性能評価実験．二つ目は拡張した最適化問題向けの状態空間構築処理の性能評価実験である．どちらも既存の SLIM の状態空間構築処理との比較を行った．また前者の比較実験については LTL モデル検査についてもその結果を述べる．

なお, 並列アルゴリズム stack slicing の cut off depth は既定値の 7 を使用している．

5.1 Δ-marking の評価実験

5.1.1 実験概要

Δ-marking の評価実験を行った実験環境を表 5.1 に示す. 共有メモリ計算機を使用しており，モデルは SLIM に含まれているベンチマークセットを使用した．Δ-marking 使用時に Delta Depth について言及がなければ既定値の 3 を使用している．

まず，使用領域の比較を行った．状態空間の表現とバイト列の表現に使用した領域につ

表 5.1　実験環境

CPU	AMD Opteron(tm) 2431
CPU 周波数	2.4GHz
PE 数	12 (6 × 2 Processors)
Memory	32 GBytes
GCC	Version 4.1.2 (-O2)

表 5.2 実行時間測定モデル

model	状態数	平均出次数
abp 04 0	1120000	1.68
abp 04 1	9600001	2.40
mutex 16 0	589841	9.33
mutex 16 1	901137	11.64
mutex 16 2	1802546	11.63
mutex 17 0	1245202	9.84
mutex 17 1	1900562	12.31
mutex 17 2	3801430	12.31
bakery 35 0	50270	2.60
bakery 35 1	72259	2.94
bakery 37 0	91348	2.62
bakery 37 1	131565	2.96

いて，Δ-marking 未使用の場合と，Δ-marking を使用し Delta Depth を 2, 3, 4, 6, 8, 10 と変化させた場合を比較する．使用モデルは状態数が数千から数十万までの 43 モデルである．1 状態からの出次数の平均値 が 1 程度のモデルから 11 程度のモデルがあった．扱ったモデル全体での出次数の平均値は 4.42，BSR の平均値は 0.65 であった．

次に，非決定実行とモデル検査についての実行時間の比較を行った．使用したモデルを表 5.2 に示す．Δ-marking 未使用の場合，1PE での実行時間が数百秒程度までであった．実行時間の測定は 2 回行い，平均値を用いる．

5.1.2 状態空間圧縮率

Δ-marking を使った場合の空間圧縮率と，BSR の関係を調べた．Delta Depth を 2, 3, 4, 6, 8, 10 と変更させたときの結果を図 5.1 に示す．

図 5.1 をみると，BSR が大きいほど，Delta Depth を大きくしたときの圧縮率が良くなる．いくつかのモデルで Delta Depth が小さい方が圧縮率が良い例があるが，これは次で述べる．

図中の直線は Delta Depth が 3 のときの圧縮率の計算値を示したものである．この線は式 (3.1) から計算したもので，実験値がこの線を下回ることもある．Delta Depth が 3

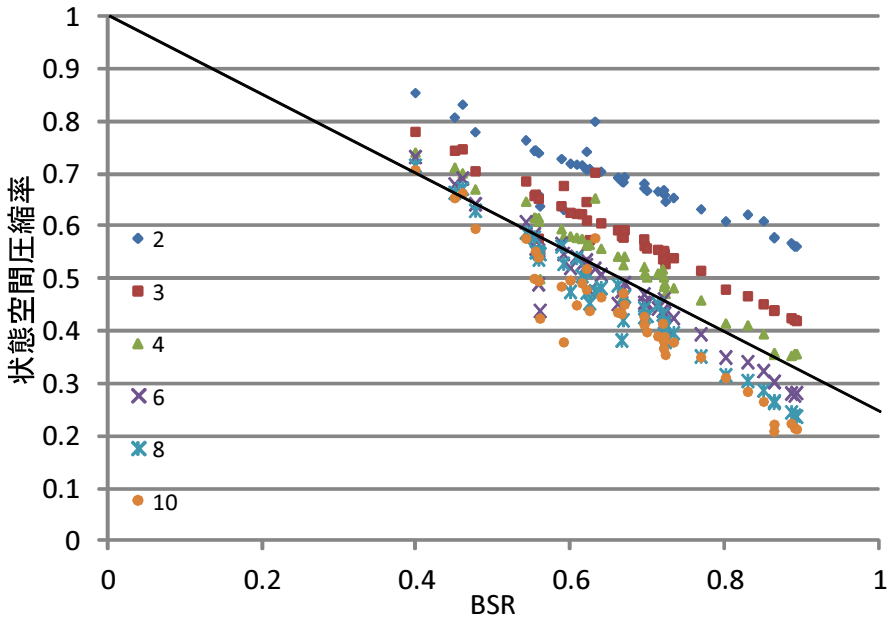

図 5.1　状態空間圧縮率と BSR の関係

表 5.3　状態空間圧縮率の幾何平均

Delta Depth	2	3	4	6	8	10
平均値	0.68	0.58	0.52	0.46	0.44	0.41

である点を見ると，ほぼ計算値に近い圧縮ができている．

状態空間圧縮率の各 Delta Depth ごとの幾何平均を表 5.3 に示す．

5.1.3　バイト列圧縮率

図 5.2 に状態空間の表現の中でバイト列が占める領域の圧縮率を示す．この図を見ると，多少のばらつきはあるものの，Delta Depth ごとに一定の圧縮率が得られるのが分かる．いくつかのモデルで Delta Depth が 8 の点が図中で一番下にプロットされている．これは，Delta Depth が 8 の方が 10 のときより圧縮率が高いことを意味する．これは深さごとにバイト列のサイズや状態数にばらつきがあるからだと推測できる．

圧縮率の平均値を表 5.4 に示す．バイト列の圧縮率は式 (3.1) と同じ仮定を置くと，$1/DeltaDepth$ で計算できる．表 5.4 の値を見ると，計算値と 0.05 ほどの差があるが，

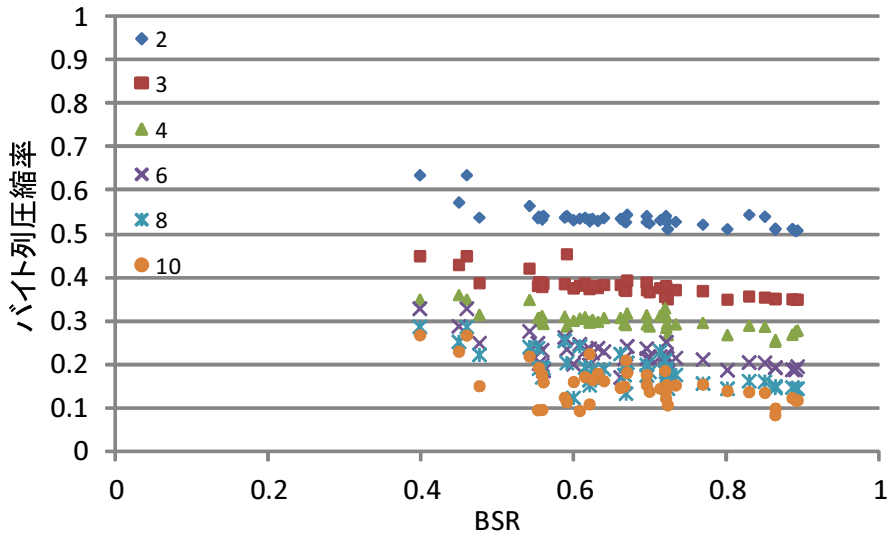

図 5.2 バイト列の圧縮率

表 5.4 バイト列圧縮率の幾何平均

Delta Depth	2	3	4	6	8	10
平均値	0.54	0.38	0.30	0.22	0.19	0.15

Delta Depth の増加とともに計算値に近い圧縮率の改善されているのが分かる．

5.1.4 非決定実行の実行時間

非決定実行における SLIM の並列効果を図 5.3 に示す．なお，normal とは Δ-marking を使わない場合の結果である．この結果をみると，SLIM は abp モデルを除き台数に近い並列効果が出ており，それは Δ-marking を使っても変わらないことが分かる．表 5.2 の平均出次数を見ると，abp は非決定性が少なく，元々並列効果が出ないモデルであることが分かる．図 5.3 の結果の幾何平均を計算したものを表 5.5 に示す．

非決定実行の実行時間比を図 5.4 に示す．図の 1PE の線は逐次実行で Δ-marking を未使用のときの実行時間に対する Δ-marking を使ったときの実行時間の比を表す．2PE, 6PE, 12PE の線は並列実行したときの同様の値を意味する．この図をみると，PE 数が変わっても実行時間比は大きく変化しない事が分かる．

逐次実行，並列実行 (12PE) 時に Delta Depth を変化させた場合の，Δ-marking を未

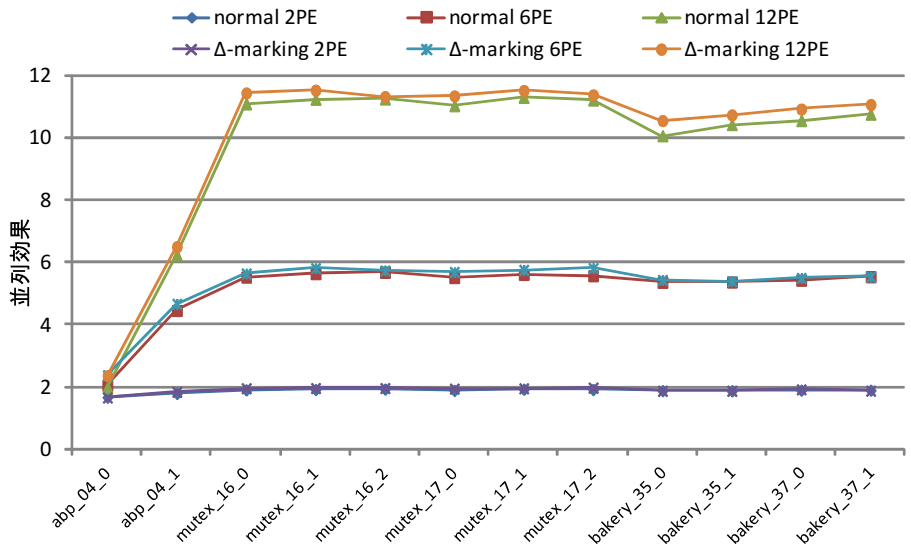

図 5.3 SLIM の並列効果

使用の場合に対する実行時間比を図 5.5, 5.6 に示す．この図を見ると，Delta Depth に比例して実行時間が増加するのが分かる．この結果は式 (3.2) から考えて適当である．また，逐次実行でも並列実行でも結果が大きく変わらなかった．図 5.5, 5.6 の結果の幾何平均を表 5.6 にまとめた．

図 5.3, 5.6 の結果を見ると，今回実装した Δ-marking は Delta Depth によらず高い並列効果を示している．

表 5.5 平均並列効果

	normal			Δ-marking		
PE 数	2	6	12	2	6	12
平均値	1.88	5.00	9.01	1.99	5.16	9.38

表 5.6 Delta Depth を変化させた場合の平均実行時間比

Delta Depth	2	3	4	6	8	10
1PE	1.45	1.76	2.02	2.52	3.04	3.53
12PE	1.42	1.69	1.94	2.42	2.90	3.49

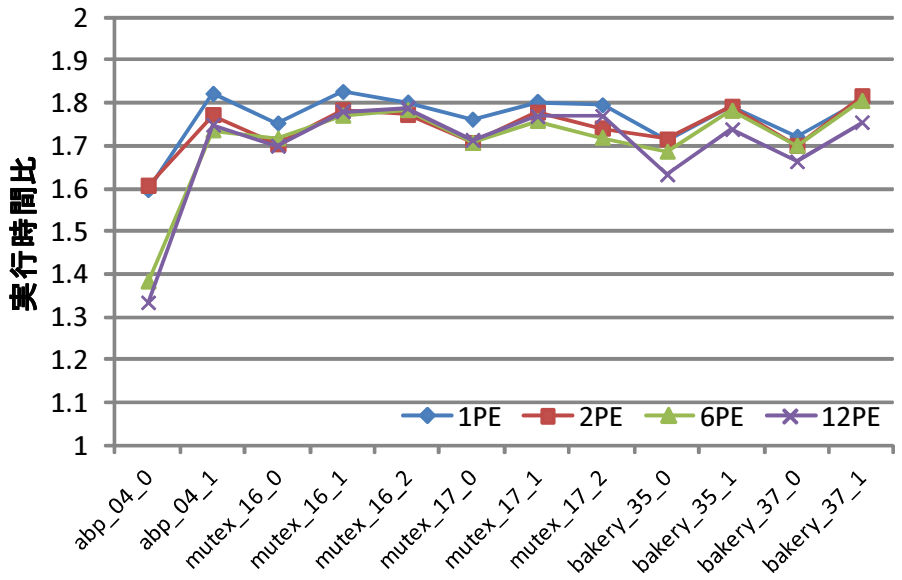

図 5.4　実行時間比

5.1.5　LTL モデル検査の実行時間

LTL モデル検査を行った場合の結果を図 5.7 に示す．左の 7 モデルはエラーが存在しないモデルであり，右の 5 モデルはエラーが存在するモデルである．

LTL モデル検査を行った場合は，モデルにエラーがあるかどうかで結果が大きく変化する．LTL モデル検査を行う場合，エラーが見つかれば反例を出力し実行を終了する．Δ-marking を使うと状態の生成順序が変わるので，エラーの見つかるタイミングが変化する可能性がある．もし，エラーの発見タイミングが変われば，実行時間も大きく変化する．SLIM は状態空間構築を行いながら，最終状態への到達性の検査と，簡単な閉路検査を行っているので，状態空間構築の途中でエラーを発見することがある．その場合，実行時間が大きく変化する．つまり，エラーのあるモデルの検査は偶然性による部分が高くなる．

エラーがない場合は，その後閉路探索をするか否かで変わってくる．閉路探索を行わないのは，受理状態への到達性のみを検査したり，受理状態が見つからない場合である．エラーがないモデルで閉路探索を行うと，実行時間比は下がると考えられる．エラーがない場合は全状態空間を探索することになる．全状態空間を探索する場合は Δ-marking を使用しているか否かに関わらず閉路探索の処理量は同じなので，そこにかかる時間は同じに

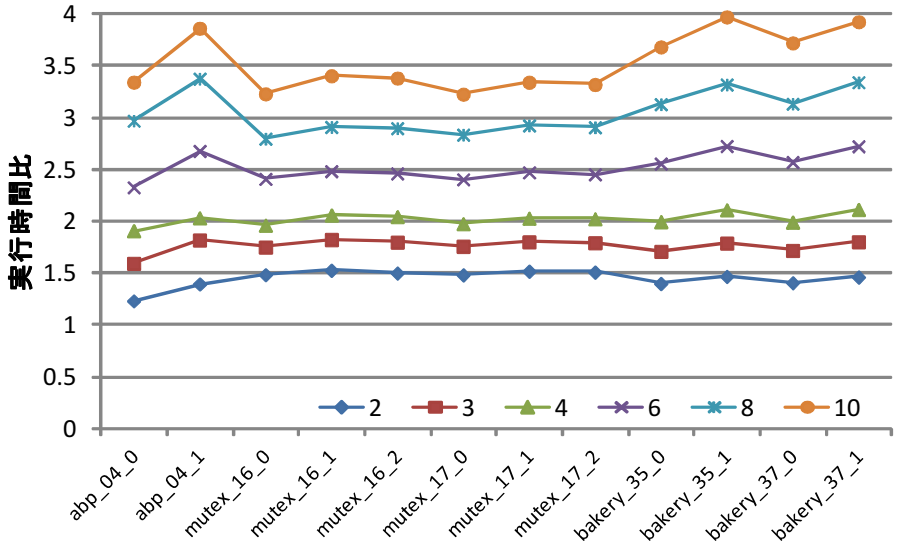

図 5.5　逐次実行時の Delta Depth と実行時間比の関係

表 5.7　実験環境

CPU	AMD Opteron(tm) Processor 6176SE
CPU 周波数	2.3GHz
コア数	48 (12 × 4 Processors)
Memory	256 GBytes
Cache Size	L1: 128KBytes (コア占有) L2: 512KBytes (コア占有) L3: 12MBytes (12 コア共有)
GCC	Version 4.1.2 (最適化オプション -O2)

なると考えられる．今回はエラーのないモデルは全て，閉路探索を行っていないので非決定実行時と同じような結果になっている．

5.2　最適化問題向け状態空間構築処理の評価実験

5.2.1　実験概要

最適化問題向けの状態空間構築処理の評価実験は，表 5.7 に示す共有メモリ計算機を使用した．実行時間の測定は 2 回行い，平均値を用いる．本研究では，N パズル問題 (8, 11, 15)，最短経路問題，最大充足化問題，ナップザック問題，巡回セールスマン問題，点彩色問題，辺彩色問題を LMNtal で記述した．その中で，最短経路問題，N パズル問題，ナップ

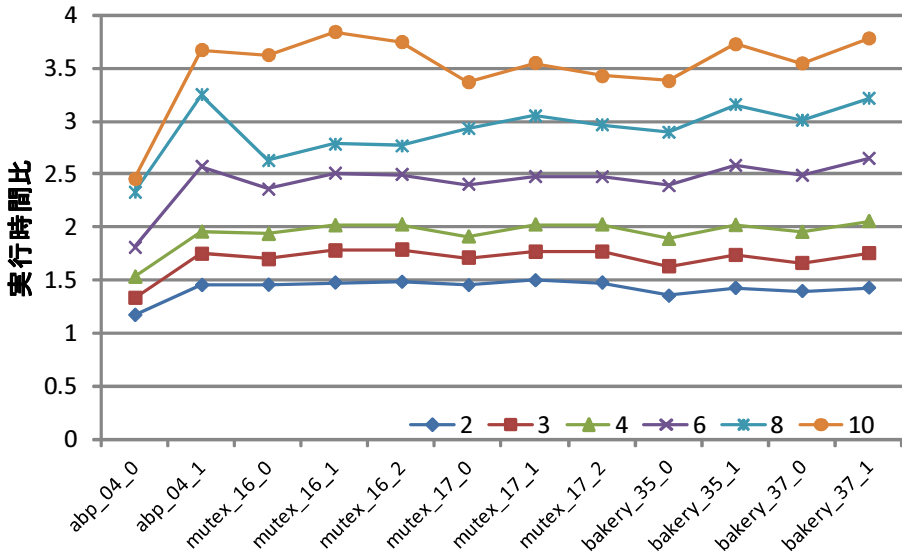

図 5.6　並列実行時の Delta Depth と実行時間比の関係

ザック問題の実験結果について述べる.

なお, 表中の No はランダム生成した問題番号を示し, 1_1 と 1_2 は同じパラメータで生成したことを示す. states は非決定実行した際に生成された状態数であり, cost はモデル内で最適のパスの重みを示す. nd は非決定実行, all は最適化実行で全ての状態に最適な重みを付けて終了する実行 (重み計算実行), min や max はモデル内で最適なパスの重みを計算する実行 (最小化実行, 最大化実行) にかかった時間 (秒) を示す. SpeedUp は 1PE に対する 48PE の並列効果を示す.

5.2.2　N パズル問題

N パズルとはスライドパズルの一種で, $m \times n$ のボードの上で $N = m \times n - 1$ 枚の駒を空いたマス目を利用して動かし目的の形にするパズルである. 初期状態として, 任意の初期位置から, 目的の形に至るまでの最小の手数を求める. もしくは, 目的の形に至ることができない場合は全状態を探索して終了する.

代表的な問題として 3×3 の 8 パズル, 4×4 の 15 パズルがある. まず, 8 パズル問題について実験を 行った. その実験結果を表 5.8 に示す. どの初期配置でも非決定実行した場合の状態数は 181440 である. Created States は最小化実行で生成した状態数を示す. No 1 の問題は目標状態へ至るパスのない問題である. 重みが最小なパスが早く見つかると,

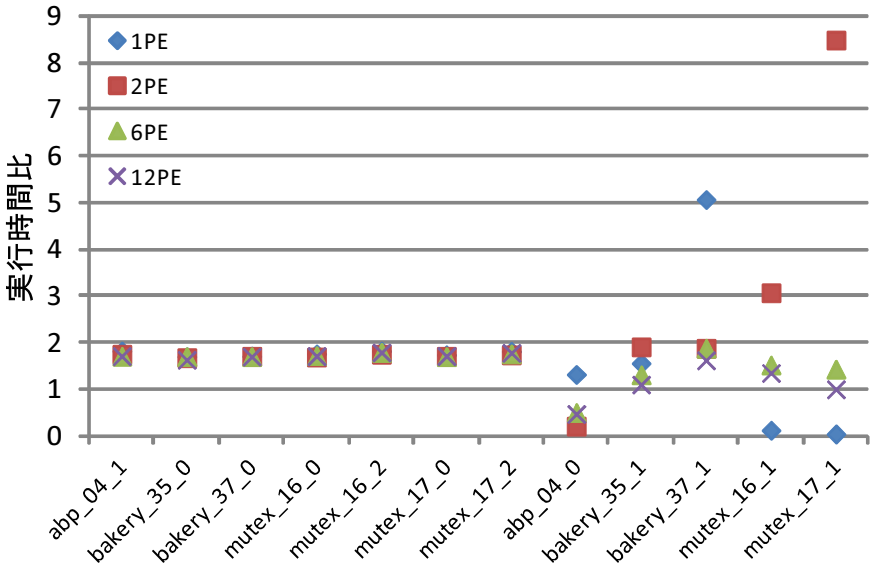

図 5.7　LTL モデル検査時の実行時間比

表 5.8　8 パズルの実験結果

No	Created States	cost	1 PE			48 PE			SpeedUp		
			nd	all	min	nd	all	min	nd	all	min
1	181440	–	7.89	8.42	10.87	0.23	0.26	2.43	33.60	31.78	4.48
2	181117	25	7.87	8.41	10.78	0.23	0.26	2.68	33.52	32.37	4.02
3	170047	17	7.84	8.37	10.73	0.24	0.27	2.12	32.03	31.00	5.06

探索の枝刈りが発生するので生成する状態数が少なくなる．しかし, 性質オートマトンとの同期積を取りながら実行をしているので, 実行時間が増加しており, 並列効果も下がっている．最小化実行の並列効果は, 性質オートマトンを入力し, 積オートマトンを探索した場合の非決定実行の並列効果に近い値となっている．

11 パズル問題, 15 パズル問題を 48PE 使用して非決定実行を行ったが 1 時間以内に, 終了しなかった．そこで, 最短手数が 7 以下になる初期配置について, 最小化実行を行ったところ最小値が求まった．最小値が求まった問題は全て 1 秒以内に実行が終了した．これは探索の枝刈りの効果によって探索する状態数が大幅に削減されたからである．また同じ初期配置であっても, 1PE では解けずに, 48PE でのみ解ける場合があった．これは使用する PE 数によって探索順序が変化し, 目的状態がすぐに見つかるかどうかが変化するからである．現在, 探索順序に関してはヒューリスティクスを導入していないので, 最適値が見つかり実行が終了するかどうかは偶然性が高い．

表 5.9　最短経路問題の実験結果

No	Created States	cost	1 PE			48 PE			SpeedUp		
			nd	all	min	nd	all	min	nd	all	min
1_1	10002	26	103.49	103.69	103.90	17.83	17.69	16.39	5.80	5.86	6.34
1_2	10000	20	104.32	104.37	104.08	17.74	17.76	16.14	5.88	5.88	6.45
1_1	19999	25	420.94	419.31	420.33	77.19	77.92	69.46	5.45	5.38	6.05
1_2	19960	25	420.39	419.33	419.99	77.81	77.62	49.25	5.40	5.40	8.53

表 5.10　制限付き最短経路問題の実験結果

No	States	cost	1 PE			48 PE			SpeedUp		
			nd	all	min	nd	all	min	nd	all	min
1_1	2001	205	122.01	122.21	121.80	57.27	57.29	56.81	2.13	2.13	2.14
1_1	2001	206	122.38	122.29	122.83	59.02	57.50	58.95	2.07	2.13	2.08
2_1	2001	109	60.79	60.93	61.03	18.89	18.26	18.89	3.22	3.34	3.23
2_2	2001	109	60.65	60.85	61.10	18.18	17.04	19.80	3.34	3.57	3.09
3_1	4001	207	487.24	489.59	490.08	139.66	145.13	142.72	3.49	3.37	3.43
3_2	4001	208	480.75	479.91	480.82	143.00	139.92	143.02	3.36	3.43	3.36

5.2.3　最短経路問題

最短経路問題とは重み付き有向グラフが与えられ, 2 つの頂点間の最小の経路を求める問題である. 今回は, 初期頂点から目標頂点への最小パスを求める問題を扱う.

実験結果を表 5.9 に示す. 非決定実行で生成された状態数と最小化実行で生成された状態数の差は非常に小さかった. つまり枝刈りが発生し探索しなかった状態は非常に少なかった. また, この問題は非決定実行でも並列効果が低い問題である. このような問題に関しては非決定実行, 重み計算実行, 最小化実行の実行時間の差が小さくなった.

有向グラフに制限を付けた最短経路問題についても実験を行った. グラフ内の全てのノードにある順番を付け, 番号の差が一定値以下のノードへのみ遷移するように, 扱う有向グラフを変更した. この場合の実験結果を図 5.10 に示す. この問題は枝刈りが発生しなかったので, 受理状態を生成した以外は状態数の違いはでなかった. この問題は実行フェーズごとに扱う状態が異なる, つまり実行中のある時刻に扱う状態数が少ない問題なので, さらに負荷分散が難しい問題である. この問題も実行方式の違いによる実行時間の差は小さくなった.

表 5.11 ナップザック問題の実験結果

No	States	cost	48 PE		
			nd	all	max
1_1	998344	1466841	6.83	12.60	16.06
1_2	997087	1392539	6.92	12.82	14.37
2_1	500002	740701	6.18	19.20	18.54
2_2	499326	744543	6.10	13.58	14.28
3_1	499142	734453	3.51	9.08	10.11
3_2	499141	746779	3.51	9.22	7.40

5.2.4 ナップザック問題

ナップザック問題とは, ある容量のナップザックに価値, 容積が定まっている n 個の品物を詰めていき, ナップザック内の品物の価値の合計を最大化する問題である. 今回は一種類の品物の個数を定めておらず, いくつでも詰めてよいとする.

48PE で実験した結果を表 5.11 に示す. 問題のサイズを大きくしたので, 1PE ではこの問題を扱うことはできなかった. ナップザック問題は最大化問題なので, 枝刈りは発生せず, 非決定実行より重み計算実行や最大化実行が早く終了することはない. また, 2 つの品物の容積の公約数が多ければ多いほど, 重みの更新回数が増加し, それに伴い実行時間が増加する. モデル内で最適な値を計算するために, 多くの場合で最大化実行の方が実行時間が長い. しかし, 並列で動作させている場合は, 重みの更新回数がタイミングによって変化するので, 重み計算実行より最大化実行の方が速い場合がある.

第 6 章

まとめと今後の課題

本論文では, SLIM に状態空間圧縮手法として Δ-marking を導入した. また, 新たに LMNtal で最適化問題を扱う手法を考案し, その実現として SLIM の状態空間構築手法を拡張した. 状態空間圧縮の結果として, 実行時間の約 1.76 倍の増加と引き換えに, 使用記憶領域の約 58% の圧縮に成功した. 最適化向け状態空間構築手法の導入によって, プログラム内部に最適化機構の記述が不必要になり, 重みのみ異なるグラフによる状態空間爆発を防ぐことに成功した. また, 最小化実行時には探索の枝刈りを行うので, 非決定実行では時間がかかる問題でも, 短い時間で最適値が求まることを確認した.

LMNtal 上での Δ-marking は, 本研究で詳しく述べた再展開による実現以外にも, 文字列差分による実現も存在する. 後者も SLIM には導入されているので比較評価が必要である. また, 文字列差分は特定の場合に差分が小さくならないことが考えられるので, 2 つの手法を自動的に使い分けることが必要である.

本研究では SLIM のみ最適化問題向けに拡張を行ったので, LMNtal の言語モデルやコンパイラは対応していない. 重み関数の記述方法とそのコンパイル方法を確立する必要がある. また現在, 最適値の探索は深さ優先に基づく stack-slicing と動的負荷分散によって行っているので, 最適化実行向けの探索ヒューリスティクスの導入と並列処理の最適化が課題である.

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. また, モデル検査関係で助言を下さった田辺 良則教授に感謝いたします.

HPV 班の方々には学部四年生, 修士一年生のころ非常にお世話になりました. 言語班の方々には修士二年生からお邪魔させていただきました. 特に清水涼子氏 中川遼平氏には様々な質問に答えていただき, お世話になりました. LaVit, UNYO-UNYO は LMNtal でモデルを作る際に, LMNtal に不慣れな私にとってとても助けになりました. また, 研究対象である LMNtal, SLIM の開発に携わった全ての方に深く感謝の意を示します.

高田 賢士郎氏には, 引き出しのお菓子を頻繁に更新や, 誕生日に好物を贈ってくれるなど, 研究室生活を豊かにして頂きました.

最後に研究室での生活を非常に楽しいものとしてくださった上田研の皆様に感謝します. ありがとうございました.

2012 年 2 月 川端聡基

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 Cerna, I.: Cluster-Based LTL Model Checking of Large Systems, In Formal Methods for Components and Objects, pp.259-279, 2005.

[3] 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.

[4] Barnat, J., Brim, L. and Ročkai, P.: Parallel Partial Order Reduction with Topological Sort Proviso, To appear in sl Proc. SEFM 2010, IEEE Computer Society Press, 2010.

[5] 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.

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

[7] Evangelista, S. 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.

[8] Evangelista, S., Petrucci, L. and Youcef, S.: Parallel Nested Depth-First Searches for LTL Model Checking, ATVA 2011, LNCS 6996, pp. 381-396, 2011.

[9] Gastin, P.: LTL 2 BA : fast translation from LTL formulae to Büchi automata, `http://www.lsv.ens-cachan.fr/ gastin/ltl2ba/`.

[10] Godefroid, P., Holzman, G. and Pirottin, D.: State-Space Caching Revisited, in *Proc. 4th Workshop on Computer Aided Verification*, Vol. 663 of LNCS, pp. 178–

191, 1992.

[11] Graphviz – Graph Visualization Software, `http://www.graphviz.org/`.

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

[13] 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.

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

[15] 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.

[16] 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.

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

[18] Kurt Jensen: Coloured Petri nets, Lecture Notes in Computer Science, Volume 254/1987, 248-299, 1987.

[19] Peled, D.: All from One, One for All: on Model Checking Using Representatives, in *Proc. 5th Int. Conf. on Computer Aided Verification*, Vol. 697 of LNCS, pp. 409–423, Springer, 1993.

[20] 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.

[21] Ueda, K.: LMNtal as a Hierarchical Logic Programming Language, *Theoretical Computer Science*, Vol. 410, No. 46, pp. 4784–4800, 2009.

[22] Ueda, K.: Encoding Distributed Process Calculi into LMNtal, Electronic Notes in Theoretical Computer Science, Vol. 209, pp. 187–200, 2008.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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, コンピュータソフトウェア, Vol. 27, No. 4, pp. 197–214, 2010.

[28] 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.

[29] 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.

[30] 後町将人，堀泰祐，上田和紀：LMNtal 実行時処理系の並列モデル検査器への発展，コンピュータソフトウェア，Vol. 28, No. 4, pp. 135-157, 2011.

[31] 後町将人: LMNtal プログラム検証における効率的な状態空間探索の並列化および Partial Order Reduction 手法早稲田大学大学院基幹理工学研究科, 修士論文, 2011.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

発表論文

[1] 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, 人工知能学会論文誌, Vol.26, No.2, pp.341-346, 2010.

[2] 川端聡基, 小林史佳, 上田和紀: 強連結成分の特性を用いた並列モデル検査アルゴリズム SCC-OWCTY の設計と評価, 2010 年並列／分散／協調処理に関する「金沢」サマー・ワークショップ, 電子情報通信学会技術報告, Vol. 110, No. 168, DC2010-17, pp. 13–18, 2010.

[3] 清水涼子, 川端聡基, 上田和紀: Explicit-time method によるモデル検査器 SLIM におけるリアルタイムモデル検査, 日本ソフトウェア科学会第 28 回大会, 日本ソフトウェア科学会, 2010.

[4] 川端聡基, 上田和紀: 並列モデル検査器 SLIM 上での Δ-marking 手法の実装と評価, ディペンダブルシステムワークショップ&シンポジウム (DSW & DSS 2011), 日本ソフトウェア科学会ディペンダブルシステム研究会, 2011.