1B5-6

LMNtal 並列モデル検査における状態生成数削減及び高速化

Reduction of the Number of States and the Acceleration of LMNtal Parallel Model Checking

安田 竜[*1] 吉田 健人[*1] 上田 和紀[*2]
Ryo YASUDA Taketo YOSHIDA Kazunori UEDA

[*1]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
[*2]早稲田大学理工学術院 Faculty of Science and Engineering, Waseda University

SLIM is an LMNtal runtime. LMNtal is a programming and modeling language based on hierarchical graph rewriting. SLIM features automata-based LTL model checking that is one of the methods to solve accepting cycle search problems. Parallel search algorithms OWCTY and MAP used by SLIM generate a large number of states for problems having and accepting cycles. Moreover, they have a problem that performance seriously falls for particular problems. We propose a new algorithm that combines MAP and Nested DFS to remove states for problems including accepting cycles. We experimented the algorithm and confirmed improvements both in performance and scalability.

1. はじめに

モデル検査とは，対象となるシステムを状態遷移系としてモデル化し，そのモデルを網羅的に探索することによって，システムに要求された性質が満たされているかどうかを判定する検証技術である．モデル検査は網羅的かつ機械的な検証を行うことができるため，並列システムにおけるデッドロックなど動作テスト等では発見が難しいエラーを確実に見つけることが可能であり，システムの信頼性を高める方法として非常に有効である．しかし，モデルの規模が大きくなると爆発的に状態数が増加してしまい，膨大な処理時間がかかるという問題がある．

これらの問題を解決するため，階層グラフ書き換え言語 LMNtal[11] をモデリング言語とする SLIM モデル検査器では，並列化等による高速化が行われてきた．LTL モデル検査で逐次最適な探索アルゴリズムとされている Nested DFS は P-完全であり並列化は困難であるため，SLIM モデル検査器には並列探索アルゴリズムである OWCTY や MAP が実装されている．しかし，OWCTY，MAP はエラーを含む問題に対して状態生成数が多くなり易い．調査の結果，階層グラフ構造という強力なデータ構造で状態を扱う SLIM モデル検査器は，状態展開に高コストな計算を含み状態空間の構築が処理時間の 99%以上を占めており，状態数が増大すると処理時間に大きな影響を与えることが分かった．また，OWCTY，MAP は特定の問題に対して極端に性能が落ちてしまうという問題点がある．

このような背景から，本研究では状態展開に高コストな処理を含むモデル検査器において，エラーを含む問題に関して状態生成数を削減し，高速かつ安定した性能を出すことを目的として，Nested DFS と MAP を組み合わせた新たなアルゴリズムの設計と実装を SLIM 上で行い，その性能を評価した．

2. モデル検査

2.1 オートマトンを用いたモデル検査

オートマトンベースのモデル検査とは，要求された性質を LTL で記述し，オートマトンを用いて受理サイクル (受理頂点を含む閉路) 探索問題に帰着させる手法である．

対象となるシステムを A，性質を S とする．このとき，システムが取り得る全実行の集合を $L(A)$，性質によって許される実行の集合を $L(S)$ とした場合，$L(A)$，$L(S)$ が以下の関係にあるとき，システム A は性質 S を満たす．

$$L(A) \subseteq L(S) \tag{1}$$

ここで，(1) は $\overline{L(S)}$ を用いて次のように書くことができる．

$$L(A) \cap \overline{L(S)} = \emptyset \tag{2}$$

これは，性質 S に反した A の実行が存在しないことを表しており，もし仮に $L(A) \cap \overline{L(S)}$ が空でないならば，それは不正な実行 (エラー) となる．したがって，$L(A) \cap \overline{L(S)}$ をオートマトン (同期積オートマトン) として表現し，このオートマトンによって受理される実行が空かどうかを判定することで，システム A が性質 S を満たすかどうか検査することができる．同期積オートマトンを作成することを状態空間の構築，受理サイクルの探索を行うことを状態空間の探索という．

2.2 On-the-fly 実行

状態空間探索アルゴリズムの重要な性質の一つとして，On-the-fly level [1] がある．状態空間探索における On-the-fly とは，状態空間構築を行いながら探索を行うことで，探索中に受理サイクルを発見した場合は探索及び構築を打ち切り，全状態空間が構築される前に状態空間探索と構築を終了することができる．On-the-fly の性能は level 0〜2 までの 3 段階に分かれており，各 level は以下のように定義される．

- level 0 : On-the-fly 動作をしない．受理サイクルが存在する場合でも全状態空間を構築する．
- level 1 : On-the-fly 動作をする場合がある．受理サイクルが存在する場合，全状態空間を構築する前に探索が終了することがある．
- level 2 : On-the-fly 動作をする．受理サイクルが存在する場合，全状態空間を構築する前に探索が終了する．

モデル検査では，システム変数等の増加により生成される状態数の組み合わせが膨大な数になることがあり，状態空間が爆発しがちであるので，全状態空間を構築する前に探索を打ち切ることは実行時間やメモリ使用量に関して非常に重要であると言える．

連絡先: 安田 竜，早稲田大学 基幹理工学部 情報理工学科，〒 169-8555 新宿区大久保 3-4-1，03-5286-3340，yasuda(at)ueda.info.waseda.ac.jp

3. モデル検査アルゴリズム

主な状態空間探索アルゴリズムとしては SPIN モデル検査器 [9] で用いられている Nested DFS，分散検証環境 DiVinE[2] で用いられている OWCTY，MAP，及び OWCTY と MAP を組み合わせた OWCTY(+MAP) アルゴリズムなどがある．状態空間構築アルゴリズムとしては SPIN モデル検査器で用いられている Stack-Slicing アルゴリズム等がある．以下で各アルゴリズムの概要について述べる．

3.1 Stack-Slicing

Stack-Slicing アルゴリズム [8] は，全スレッド (Worker) で状態空間を共有し，深さ優先探索 (DFS) をパイプライン分割するアルゴリズムである．全スレッドは，状態を受け取る Work Queue を固有に持ち，通信方向を持った論理的な輪を構成する．DFS のスタックの深さが一定の閾値を超えた場合，隣接スレッドの Work Queue へ未展開の新規状態を送信する．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．しかし，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ないという問題がある．

3.2 Nested DFS (NDFS)

Nested DFS[5] は，On-the-fly level 2 の逐次探索アルゴリズムで，LTL モデル検査において逐次最適な探索アルゴリズムである．Nested DFS は, 二段階の DFS (Blue DFS，Red DFS) により構成される．Blue DFS が状態空間の構築を行い，Red DFS が状態空間の探索を行う．Red DFS は, ある受理状態から自分自身へ到達できるかということを DFS により判定する．Blue DFS は， DFS による状態展開を行いながら Red DFS の起点となる受理状態を探索する．Red DFD の起点となる受理頂点は postorder 順に選択される．これにより，同一の枝に対する traverse が発生しなくなる．しかし，受理頂点を postorder 順に選択しなければならないという制約があるため，状態の展開や探索を複数のスレッドで分担することができず，並列化が困難である．SPIN モデル検査器では，Red DFS と Blue DFS を並列に行わせる Dual DFS が並列アルゴリズムとして用いられているが，使用できる最大スレッド数は 2 つである．

3.3 Maximal Accepting Predecessors (MAP)

MAP[3] は，On-the-fly level 1 の並列探索アルゴリズムである．MAP は受理サイクルを作る受理状態は自分自身に到達可能であるという考えに基づき，状態遷移時にその状態遷移可能な受理状態の情報を伝達させていくアルゴリズムである．

3.4 One Way Catch Then Young (OWCTY)

OWCTY[4] は，On-the-fly level 0 の並列探索アルゴリズムである．グラフの構築と，受理サイクルを作らないと判断できる状態の削除を繰り返す手法である．OWCTY は，On-the-fly level が 0 であるため，全状態空間を構築してから探索を行う必要があるが，一部 MAP の手法を導入することにより On-the-fly level 1 で動作するようになる．受理サイクル発見のタイミングは MAP に依存しているため，状態生成数は MAP と同等になる．

4. SLIM におけるモデル検査アルゴリズム

SLIM モデル検査器では，状態空間構築アルゴリズムとして動的負荷分散を行うように拡張した Stack-Slicing アルゴリズムを，状態空間探索アルゴリズムとして Nested DFS，MAP，OWCTY(+MAP) を用いており，状態空間構築・探索処理の並列化がなされている [12]．状態空間探索アルゴリズムに MAP を用いた場合の処理を図 1 に示す．SLIM における Stack-Slicing アルゴリズムは，3.1 で述べた問題点を解決するため，隣接スレッドがアイドル状態を主張した場合に，閾値を待たずに状態の送信を行う処理と，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を追加することで負荷分散を行っている．*StackSlicing* は，スレッドが自身の Work Queue(*myWorkQueue*) から取得 (*dequeue(myWorkQueue)*) した状態から DFS による状態展開処理 (*dfsParallel*) を繰り返し行う手続きである．*dfsParallel* 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 (*stackPop*) した状態 *s* に対して遷移先の計算 (*expand*) を行う．その後，MAP の伝搬や受理サイクルの探索 (*mapStart*) 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 (*stackPush(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) を超えている場合と，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，*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(myWorkQueue)*) 場合に，他のスレッドの Work Queue から状態の取得を試みる (*workStealing*)．*terminationDetection* は終了検知を行う手続きを表し，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．

これらの処理において，状態空間構築処理では高い並列効果が出ているが，状態空間探索処理においては On-the-fly level が 0〜1 の並列アルゴリズムしか実装されていないほか，性能の評価も十分に行われていなかった．探索アルゴリズムの性能評価を行った結果，SLIM のように状態展開に高コストな処理を含んでいる場合，これらの探索アルゴリズムを用いた場合の受理サイクルを含む問題に対しての状態生成数増大は影響が大きく，十分な性能が出ていないことが分かった．

```
procedure StackSlicing
    while terminationDetection() do
       if empty(myWorkQueue) then
          workStealing()
       else
          stackPush(dequeue(myWorkQueue))
          dfsParallel()
       end if
    end while
end procedure

procedure dfsParallel
    STATE s := stackPop()
    expand(s)
    mapStart(s)
    for succ ∈ newSuccessors(s) do
       if loadBalancing then
          handoff(succ)
       else
          stackPush(succ)
          dfsParallel()
       endif
    end for
end procedure
```

図 1: Stack-Slicing Algorithm

5. 新たな並列モデル検査アルゴリズム

5.1 概要

On-the-fly level が 0〜1 の並列モデル検査アルゴリズムである MAP や OWCTY では，状態生成数が On-the-fly level 2 の Nested DFS に比べて多く，SLIM モデル検査器のように状態の展開に高コストな計算を含む場合に十分な性能がでない．そこで，Nested DFS と MAP を組み合わせた On-the-fly level が 2 の並列モデル検査アルゴリズムを作成した，

このアルゴリズムは MAP ではなく OWCTY(+MAP) を用いても実現できる．しかし，OWCTY(+MAP) の状態生成数は 3.4 節で述べた通り MAP の状態生成数と等しい．状態空間探索時間が無視できるほど状態展開に高コストな処理を抱えている場合，状態数が等しければ全体の実行時間もほぼ等しくなるため性能は MAP とほぼ同じとなる．よって，今回は OWCTY ではなく実装上扱い易い MAP を利用した．

また，On-the-fly level 2 の並列アルゴリズムとしては，複数スレッドで Nested DFS を複数スレッドで同時に動かすことによって並列化を行う Multicore Nested DFS[7] も存在する．しかし，各スレッドで Nested DFS を実行する関係上，各スレッドでは Nested DFS と同様の制約が生じ，状態の受け渡しを行って状態展開の並列化を行う Stack-Slicing とは併用ができず，負荷分散も行えなくなる．このため，状態生成数を削減することができても，Stack-Slicing を用いた場合と比較して状態の展開自体が遅くなってしまう可能性があることから，今回は独自のアルゴリズムを導入することとした．

5.2 方針

既存の SLIM 並列モデル検査アルゴリズムでは図 1 のように Stack-Slicing による状態空間構築と MAP による状態空間探索を行っていた．既存のアルゴリズムでは N スレッドが MAP を実行していたが，1 スレッド (*ndfsWorker*) で Nested DFS を実行し，残りの N-1 スレッド (*mapWorker*) で MAP を実行するように変更する．こうすることにより，N スレッドで並列で状態空間の構築を行いながら，Nested DFS による早期の受理サイクル検出が可能となる．受理サイクルを検出するタイミングは Nested DFS と同じになるため，状態の生成数は Nested DFS と同じ On-the-fly level 2 のアルゴリズムと同等になる．それに加え，同時に MAP を動作させることにより，MAP の方が Nested DFS よりも受理サイクルの検出が早いような問題や，エラーを含まない問題にも対応できる．ただし，1 スレッドを Nested DFS に使うことにより，既存のアルゴリズムと比較して探索速度は劣る．しかし，状態の展開に高コストな計算を含むモデル検査器の場合，効率的な探索を行うことよりも状態生成数を削減することによる状態展開処理の削減を行った方が効率的な場合がある．SLIM モデル検査器の場合は，状態空間構築に 99%以上の時間がかかっており，状態生成数削減により高速化が期待できるはずである．

新たなアルゴリズムでは，*ndfsWorker* が Nested DFS による探索と postorder 順の計算 (*calcPostorder*) を行い，*mapWorker* は既存のアルゴリズムと同様に MAP による探索を行う．postorder 順の計算は，状態 *s* の successor の中で未到達の状態を DFS スタックに積み直すことで行われる．*ndfsWorker* は postorder 順の計算を行わなければならないことから，*workStealing* による状態の取得や，*handoff* によって隣接 Worker から渡された状態を処理するといった処理を行うことができない．そこで，状態の $handoff$ は *ndfsWorker* から *mapWorker* への *handoff* または *mapWorker* から *mapWorker* への *handoff* のみ可能にするように処理を変更する (*newHandoff*)．*workStealing* に関しても，*ndfsWorker* は $workStealing$ を行わず，*mapWorker* のみが行うように変更を加える．これらの変更を加えたアルゴリズムを図 2 に示す．

```
procedure newStackSlicing
    while terminationDetection() do
       if empty(myWorkQueue) ∧ worker is mapWorker then
          workStealing()
       else
          stackPush(dequeue(myWorkQueue))
          newDfsParallel()
       end if
    end while
end procedure

procedure newDfsParallel
    STATE s := stackPop()
    expand(s)
    if worker is mapWorker then mapStart(s)
    if worker is ndfsWorker then
       calcPostorder()
       ndfsStart(s)
    endif

    for succ ∈ newSuccessors(s) do
       if loadBalancing then
          newHandoff(succ)
       else
          stackPush(succ)
          newDfsParallel()
       endif
    end for
end procedure
```

図 2: newStack-Slicing Algorithm

6. 評価実験の概要

実験環境は表 1 に示す 48 コアの共有メモリマシンを使用した．評価実験は，受理サイクルを含む問題 60 個と，受理サイクルを含まない問題 22 個を対象に，48 スレッドを用いた場合の全実行時間及び状態生成数を MAP と新たなアルゴリズムで比較する．Stack-Slicing の閾値は 5 とする．計測時間が 2000[sec] を超えるものはタイムアウトとみなし，計測結果は 2000[sec] とする．タイムアウトした問題の状態生成数は計測できないため，グラフ中には表記しない．また，受理サイクルを含まない問題の場合，どのアルゴリズムを用いても全状態空間を構築することから，状態生成数は全アルゴリズムで等しくなるので，結果は省略する．

使用した問題は，BEEM Database (`http://anna.fi.muni.cz/models/`) 上で公開されている，SPIN モデル検査器用のベンチマーク問題などを中心に，Promela で記述されたモデルを LMNtal でエンコードして利用している．問題には，食事をする哲学者問題や Bakery Mutex Algorithm 等の排他制御モデルや，Sliding Window Protocol などの通信プロトコル問題，N-Queen 問題などのパズル問題，その他システムモデルや計算モデル等の問題が含まれている．問題の規模としては，MAP により生成される状態数の上限が 700 万状態程度の問題を主な対象とした．受理サイクルを含む問題に関して，評価実験を行った結果を図 3，4 に示す．次に，受理サイクルを含まない問題に関して評価実験を行った結果を図 5 に示す．

7. 新たなアルゴリズムに対する考察

7.1 状態生成数

図 3 は，新たなアルゴリズムと MAP をそれぞれ 48 コアで動作させた場合の状態生成数を表している．約 60%の問題で状態生成数は 0.5 倍以下に，さらに約 30%の問題で状態生成数が 0.125 倍以下になっている．一方で，MAP よりも状態生成数が多くなってしまった問題は 7 つのみであり，いずれの問題に関しても MAP と比較して大きく超過していないことがグラフから分かる．これらの問題は，NDFS による受理サイクルの発見タイミングよりも MAP による受理サイクルの発見のタイミングの方が早いような問題だと思われる．この場合，状態生

表 1: 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)

成数は MAP と同等になるはずだが，若干超過してしまっている．これは，逐次実行の場合と違い，並列実行の場合は同一のアルゴリズムを実行しても状態生成数に若干の差異が生じるため，これらの超過はその影響であると考えられる．また，状態生成数がどの程度削減できるかは問題に依存しているが，この計測結果から多くの問題で状態生成数を削減することが可能であることが確認できる．一部の問題では，MAP では 6970337 状態であったものが新たなアルゴリズムでは 5346 状態にまで削減されるなど，非常に良く効果が出ているのが分かる．

7.2 実行時間

図 4 は，新たなアルゴリズムを用いた場合と MAP アルゴリズムを用いた場合の実行時間を表している．数十 ms で終了するような比較的小規模の問題も含まれているため，それらの問題では MAP よりも実行時間が長くなっている問題があるが，これは誤差の範囲内であると考えられる．数秒〜数十秒程度かかる問題に関して見てみると，状態数削減に伴い多くの問題で実行時間が改善している．さらに，状態数が爆発しやすく既存のアルゴリズムでは解くのが困難であった問題も，高速に解けるようになっていることが分かる．受理サイクルを含まない問題に関しては，状態生成数が変わらないため多くの場合で MAP と同等の性能となっていることが図 5 から確認できる．また，MAP や OWCTY は反復探索処理を行うため受理状態が一列に繋がっているような問題では極端に性能が低下してしまうが [13]，これらの問題も今回導入したアルゴリズムでは高速に解けている．

8. まとめと今後の課題

8.1 まとめ

Nested DFS と MAP を組み合わせた新たな並列モデル検査アルゴリズムの設計・実装を行った．新たな並列モデル検査アルゴリズムと SLIM に実装されている MAP を比較したところ，受理サイクルを含む多くの場合で状態生成数が削減され，それに伴い高速化が行われた．さらに，MAP では解けなかった一部の大規模問題にも対応できるようになった．受理サイクルを含まない問題でも MAP と相性の悪い問題に関しては高速化を行うことができた．これにより，状態展開に高コストな計算を含むモデル検査器において，状態空間探索アルゴリズムの On-the-fly level を上昇させることによって状態生成数を削減することにより，全実行時間を短縮できることが確認できた．

8.2 今後の課題

今回導入したアルゴリズムでは問題の性質を考慮していないが，例えば強連結成分 (SCC) による分割を利用し，問題の性質を考慮するように改良された Nested DFS [6] の導入などにより，さらなる状態数削減効果が得られるのではないかと考えられる．また，SLIM モデル検査器のように状態展開に高コストな処理を抱えるモデル検査器では状態空間構築処理に多くの処理時間が割かれるため状態空間探索処理の高速化というのは効果が薄い．よって，さらに高速化していくためには状態空間構築の改善が必要であると考えられる．状態空間構築の高速化を行うには，ある状態を起点にした経路集団から代表経路を決定することで実際に展開する状態数を削減する Partial Order Reduction [10] の応用などが考えられる．

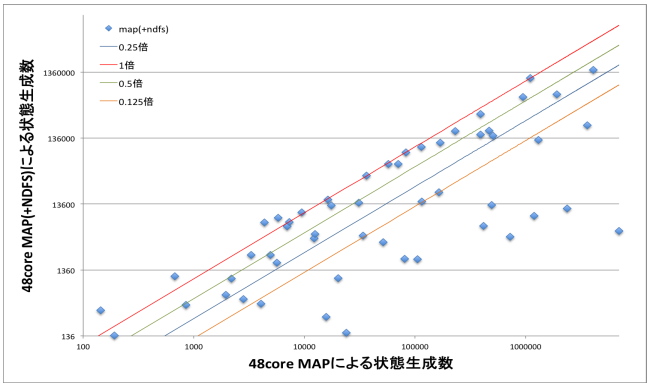

図 3: 状態生成数比較

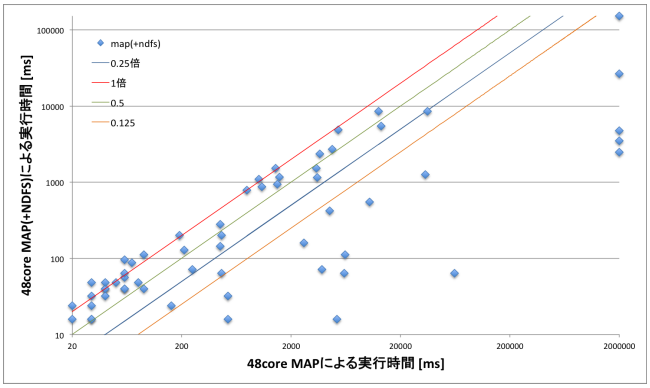

図 4: 実行時間比較

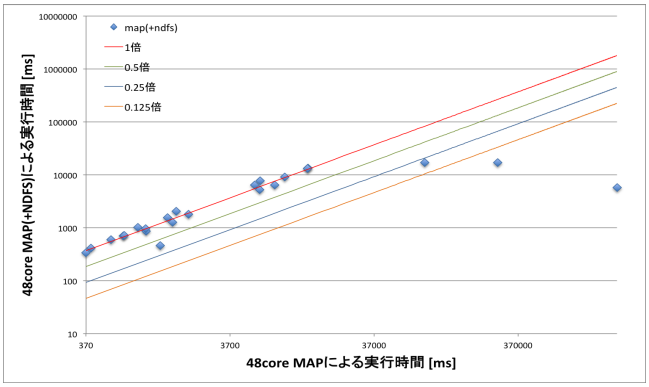

図 5: 実行時間比較 (受理サイクル無)

本研究の一部は，科学研究費（基盤研究 (B)23300011）の補助を得て行った．

参考文献

[1] Jiří Barnat, Luboš Brim, Petr Ročkai : On-the-fly Parallel Model Checking Algorithm that is Optimal for Verification of Weak LTL Properties, Science of Computer Programming, Vol.77, No.12, pp. 1272-1288, 2012.

[2] J. Barnat, L. Brim, I. Cerna, P. Moravec, P. Rockai, and P. Simecek : DiVinE - A Tool for Distributed Verification, in Proc. CAV 2006, LNCS 4144, pp. 278-281, 2006.

[3] Brim, L., Cerna, I., Moravec, P. and Simsa, J. : Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking, in Proc. FMCAD 2004, LNCS 3312, pp. 352-366, 2004.

[4] I. Černă, R. Pelánek : Distributed Explicit Fair Cycle Detection, in Proc. SPIN 2003, LNCS 2648, pp. 49-73, 2003.

[5] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis : Memory Efficiend Algorithm for the Verification of Temporal Properties. in CAV 1990, LNCS 531, pp. 233-242, 1990.

[6] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue : Directed Explicit Model Checking with HSF-SPIN, in Proc. SPIN 2001, LNCS 2057, pp. 57-79, 2001.

[7] Sami Evangelista, Alfons Laarman, Laure Petrucci, and Jaco van de Pol : Imporoved Multi-Core Nested Depth-First Search, in Proc. ATVA'2012, LNCS 7561, pp. 269-283, 2012.

[8] Holzmann, G. : A Stack-Slicing Algorithm for Multi-Core Model Checking, Electronic Notes in Theoretical Computer Science (ENTCS), Vol.198, No.1, pp. 3-16, 2008.

[9] Holzmann, G.J. : The Model Checker SPIN, IEEE Transactions on Software Engineering, Vol.23, pp. 279-295, 1997.

[10] Peled, D. : All from One, One for All : on Model Checking Using Representatives, in Proc. CAV 1993, LNCS 697, pp. 409-423, 1993.

[11] 乾敦行, 工藤晋太郎, 原耕司, 水野謙, 加藤紀夫, 上田和紀 : 階層グラフ書換えモデルに基づく統合プログラミング言語 LMNtal, コンピュータソフトウェア, Vol.25, No.1, pp. 124-150, 2008.

[12] 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. 137-157, 2011.

[13] 小林史佳 : LMNtal 実行時処理系 SLIM の LTL モデル検査機能の並列化, 早稲田大学大学院 理工学研究科, 修士論文, 2009.