修 士 論 文 概 要 書
Summary of Master's Thesis

Date of submission: 1 / 31 / 2012 (MM/DD/YYYY)

専攻名（専攻分野） Department	情報理工学	氏名 Name	中川 遼平	指 導 教 員 Advisor	上田 和紀 印 Seal
研究指導名 Research guidance	並列知識情報処理	学籍番号 Student ID number	CD 5110B096 – 3		
研究題目 Title	LMNtal グラフの一意バイト列表現のインクリメンタルな生成手法				

1 はじめに

システムのテストやシミュレーションだけではすべての不具合を発見できない問題に対して，システムの振る舞いを漏れなく検証するモデル検査手法が有効である．LMNtal モデル検査器は並行言語 LMNtal[1] の実行時処理系 SLIM に組み込まれている．LMNtal グラフは階層グラフ構造を成すために，遷移先状態の構築と管理の最適化がなされてきた [2]．本研究では，現状の SLIM において対称性の高い LMNtal グラフで有効な最適化である，グラフの一意バイト列表現の生成を高速化するために Incremental Canonical Labeling を導入した．

2 LMNtal モデル検査器

並行言語 LMNtal は階層グラフ構造の書換えに基づく言語である．階層グラフはアトム・リンク・膜で構成されていて，アトムはグラフのノード，リンクはノードを 1 対 1 に接続する無向エッジに相当し，膜はアトムの多重集合を階層化する仕組みである．グラフ書換えにはルールが用いられ，書換え可能なグラフ構造が複数存在する場合には，書き換え順序に非決定性が生じる．LMNtal では状態を構成するこれら 4 つの要素をプロセスと呼ぶ．

SLIM は軽量かつ高速な LMNtal 実行時処理系であり，C 言語で実装されている．SLIM では通常実行だけでなく非決定的な実行が可能である．非決定実行では実行途中で生成した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取得することができ，検証したい条件を与えることでモデル検査を行うことができる．図 1 に LMNtal モデル検査器における状態構築と管理方法の概要図を示す．ある未展開の状態に対して，その状態が持つ各ルールによって書換え可能な部分グラフ構造を全探索し，それぞれの書換えに対して書換え後の LMNtal グラフを構築することで，次の遷移先を計算している．構築された全ての LMNtal グラフはハッシュ値が計算され，ハッシュ値をキーとして状態管理表に登録される．登録の際にハッシュ値が衝突した場合にはグラフ同型性判定が行われ，等価な状態が状態管理表に複数回登録されてしまうのを防いでいる．大規模なモデルの検証を高速化するために，SLIM にはさまざまな最適化技術が導入されている．

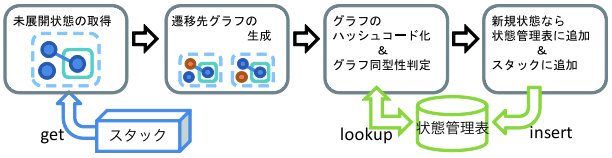

図 1　状態構築と管理方法の概要図

2.1 プロセスへの差分適用

未展開の状態を展開する際，状態が持つ LMNtal グラフを書き換えるたびに元のグラフをコピーする必要があった．しかしながら，ルール適用による LMNtal グラフの書換えは，階層グラフ構造全体のうち一部分だけが書き換わる場合が多い．そのため，ルール適用によって書き換わるプロセス（差分）を保持しておくことで，書換え後のグラフから元のグラフを高速に再構築することができる．

2.2 プロセスのエンコード

状態が保持す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非常に高い．ゆえに，大規模なモデルのメモリ使用量を抑えて扱うために状態をコンパクトに表現する必要がある．そのため，SLIM ではプロセスを管理する際にエンコード処理によって，階層グラフ構造をバイト列表現に変換している．バイト列表現に変換したプロセスをルール適用する際には，デコード処理によって階層グラフ構造へ変換を行う．

プロセスを構成するアトムや膜は多重集合であるため，バイト列が並ぶ順序は決定的ではない．それゆえ，エンコード順序により 1 つのプロセスに対して複数個のバイト列が生成される場合があるが，列の並びが辞書順で最小となるバイト列は一意に定まる．そのようなバイト列を特にプロセスの一意バイト列表現と呼ぶ．プロセスの一意バイト列表現はプロセスに対して 1 対 1 に対応するので，プロセスの一意バイト列表現同士の比較によりグラフ同形成判定が可能となる．

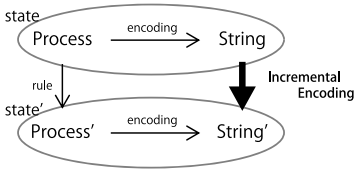

図 2　状態のプロセスと一意バイト列表現の関係図

3　状態遷移グラフ構築の高速化

状態遷移グラフの大きさによらず，性能の良いハッシュを高速に求める方法として Incremental Hashing[3] という手法がある．Incremental Hashing とは前状態のハッシュと状態間の差分から状態のハッシュを計算する手法である．一般的に状態遷移グラフの状態間の差分は個々の状態の大きさに比べて小さい．それゆえ，状態のハッシュは始めから求めるよりも，前状態のハッシュからインクリメンタルに求める方が高速である場合が多い．本研究では，SLIM に対してインクリメンタルな計算手法を導入するにあたり，ハッシュ関数を変更するのではなく，一意バイト列表現のインクリメンタルに計算する手法 (Incremental Canonical Labeling) を実装した．なぜなら，プロセスの一意バイト列表現を求める処理が，一連のハッシュ計算において，最大のボトルネックとなっているためである．

一意バイト列の生成手法について従来の手法と Incremental Canonical Labeling を比較しながら解説をする．図 2 に状態と遷移先状態の関係図を示す．図の *Process* は状態が持つプロセスを表し，*String* は状態が持つプロセスの一意バイト列表現を表している．また，*rule* はルール適用，*encoding* は一意バイト列表現を求める手続き *id* を適用する処理を表す．

従来手法では *Process* にルール適用をして *Process′* を生成する．そして，*Process′* を計算した後にプロセスの一意バイト列表現を求める手続き *id* を使って *String′* を算出している．一方，Incremental Canonical Labeling では状態間の差分 *delta* を用いて一意バイト列の計算を行う．状態間の差分 *delta* はプロセスの差分適用機能を利用するときに計算する差分と同等である．状態 *State* は状態管理表にすでに登録されているので，状態 *State* が持つプロセスの一意バイト列表現 *String* は計算済みである．*String* と *String′* は状態が持つプロセスを一意バイト列表現したものであり，*String* と *String′* のバイト列の差分は *delta* に相関している．よって，*Process* と *Process′* の差分 *delta* を使って *String* から *String′* を計算することで手続き *id* を使わずに済む．また，すでに一意バイト列表現されている全プロセスに対して，変化するプロセスの割合が小さければ *String′* の生成をさらに高速化することができる．

表 1　mutexN（左）と exchangeN（右）の実行時間 [sec]

手法	N=6	N=7	N=8	N=9	N=10	N=13	N=15	N=17	N=19
Inc	0.06	0.20	0.66	2.07	8.69	1.20	3.09	7.50	18.80
Nor	0.07	0.27	0.97	3.79	14.08	3.58	11.83	33.92	87.70

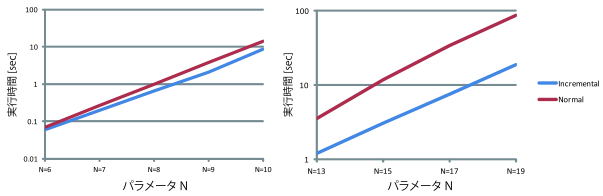

図 3　mutexN（左）と exchangeN（右）での実行時間比較結果

4　本手法の評価

Incremental Canonical Labeling を 2 種類のプログラムを使って従来手法と比較実験する．全体の実行時間に対して従来手法で一意バイト列表現を生成するのに要する時間の割合は mutexN は約 70%, exchangeN は約 80% である．実験には 4 コア，RAM 4GB のマシンを使用した．

実行結果を表 1，結果をまとめたグラフを図 3 に示す．mutexN では実行時間にそれほど差が見られなかったのに対し，図 3 の傾きから N を増やすことによって，実行時間の差を数倍から十数倍に広げられると考えられる．この計算量の違いはプロセスの変化量に起因するものである．exchangeN も同様に N が増えると従来手法より実行時間が短くなり，全体の実行時間を最大で約 80% 縮小することに成功した．

5　まとめと今後の課題

本研究では，LMNtal グラフの一意バイト列表現をインクリメンタルに求める手法を実装し，正しく動作することを確認し，実行時間の短縮を実現した．

今後の課題としては，Incremental Canonical Labeling の適用できるグラフの拡大や変化量 *delta* をさらに減らすような一意バイト列表現の設計が挙げられる．

参考文献

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

[2] 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展開，コンピュータソフトウェア，Vol. 28, No. 4, pp. 137–157, 2011.

[3] Stefan Edelkamp, Tilman Mehler: Dynamic Incremental Hashing in Program Model Checking, Proceedings of the Third Workshop on Model Checking and Artificial Intelligence, 2006.