グラフ書き換えモデル検査器 SLIM への差分適用グラフ正規化手法の実装

恒川 雄太郎　上田 和紀

グラフ書き換え系はグラフ構造を用いてモデリングすることから高い表現力を持ち，モデリング対象の潜在的な対称性を自然に表現可能である点がモデル記述言語として優れている．グラフ書き換えに基づくモデル記述言語 LMNtal の処理系 SLIM は現在 C++で 3 万行規模で開発されており，並列モデル検査機能を備えている．SLIM の実装における最大の課題はグラフ同型性判定であり，これまでバイト列エンコーディングやハッシュ関数を用いた最適化が実装されている．本研究では同型性判定をさらに高速化するために，宮原・上田が提案した，グラフ書き換えのために最適化されたグラフ正規化手法をモデル検査器 SLIM に実装した．本発表ではアルゴリズムの概要，および実装の詳細について述べる．

Graph rewriting systems allow us to model various state transition systems with their highly expressive data structures and to express models in such a way that inherent symmetry manifests itself with graph isomorphism of states. SLIM is a publicly available model checker under active development with more than 30000 LOC in C++ and scales up to billions of states on shared-memory parallel computers. The biggest challenge in the implementation of SLIM is graph isomorhpism checking, for which optimization using byte string encoding and hash functions has been implemented so far. In our research, we implemented Miyahara-Ueda's incremental canonical labeling algorithm for graph rewriting systems in SLIM to make its graph isomorhpism checking more efficient. In this paper, we describe an overview of the algorithm and details of our implementation.

1　はじめに

グラフ書き換え系はグラフ構造を用いてモデリングすることから高い表現力を持ち、検査対象のシステムの潜在的な対称性を自然にモデリング可能である点がモデル記述言語として優れている。グラフ書き換え系をモデリングに用いるモデル検査器には SLIM [4] [7] や GROOVE [3] などが存在する。グラフ書き換えに基づくモデル記述言語 LMNtal の処理系 SLIM は現在 C++で 3 万行規模で開発されており，並列モデル検査機能を備えている．モデル検査においては状態数がシステムの複雑さに対して爆発的に増加する現象が大きな問題となるが，グラフ書き換えに基づくモデル検査器ではグラフの対称性を利用して状態空間の爆発を抑制することができる．しかし，状態空間の構築には同一視可能な状態を判定するためにグラフ同型性判定を頻繁に行う．図 1 は SLIM における，バイナリセマフォによる N プロセスの排他制御問題の状態数と実行時間全体に対するグラフ同型性判定に掛かる時間の割合の関係を表したグラフである．図から状態数の増加に従ってグラフ同型性判定に掛かる時間が実行時間全体の大部分を占めることが分かる．SLIM のグラフ同型性判定処理に対しては，これまでバイト列エンコーティングやハッシュ関数を用いたさまざまな最適化が提案および実装されている [7].

本研究では SLIM のグラフ同型性判定処理のさらなる高速化を目的として宮原・上田が提案したグラフ書き換えのために最適化されたグラフ正規化手法 [6]

Implementation of Incremental Canonical Labeling in the Graph-Based Model Checker SLIM

Yutaro Tsunekawa, Kazunori Ueda, 早稲田大学基幹理工学研究科情報理工・情報通信専攻, Dept. of Computer Science and Communications Engineering, Graduate School of Fundamental Science and Engineering, Waseda University.

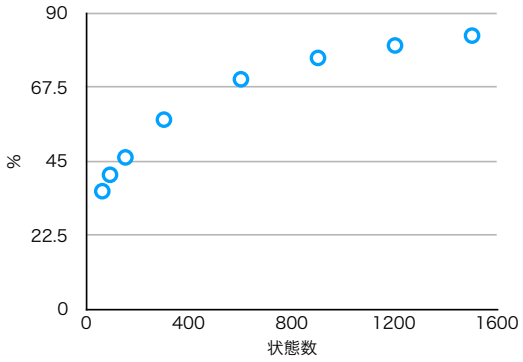

図 1　バイナリセマフォによる N プロセスの排他制御問題の状態数と実行時間全体に対する同型性判定に掛かる時間の割合の関係

を実装した．本論文ではアルゴリズムの概要および実装の詳細について述べる．

2　関連研究

本節では，関連研究について紹介する．

グラフ書き換え系を対象にしたモデル検査器は多くない．その中で GROOVE [3] と SLIM は現在も活発に開発が続けられているモデル検査器である．GROOVE はグラフ構造をモデルとして用いたモデル検査器である．GROOVE ではグラフに対するハッシュ関数を用いて，同型ではないグラフに対する同型性判定の回数を削減することで効率化を図っている．

SLIM では多数のグラフ間の同型性判定を効率良く行う手法として，グラフ構造の一意バイト列を生成し，それらの比較によって同型性を判定するオプションが備わっている．SLIM も GROOVE 同様にグラフ構造に対するハッシュ関数を用いた効率化を行っている．ハッシュ関数は複数種類を用意し，ハッシュ値が頻繁に衝突するようなグラフ構造に対しては状態空間探索中にハッシュ関数を切り替えてグラフのハッシュ計算を行う．SLIM では消費メモリの圧縮のために，ハッシュテーブルに格納するグラフ構造を一意でないバイト列にエンコードしている．SLIM においてはハッシュ値が衝突した場合に 2 つの状態を同型性判定によって真に同じ状態かどうか確認するが，その際には効率化のためにバイト列とグラフ構造とを同型性判定するなど工夫している．

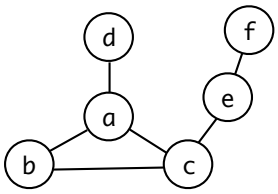

図 2　有限単純無向グラフ

3　グラフ正規化アルゴリズム

本節では，McKay のグラフ正規化アルゴリズム [2] と，それをベースに宮原・上田によって提案された差分適用グラフ正規化アルゴリズムについて概説する．詳細については [2], [6] を参照してほしい．

グラフ正規化とはグラフを入力として同型なグラフに一意な表現を生成することであり，一意表現とはグラフの各頂点に対する異なる自然数の割り当てである．正規化されたグラフ同士の同型性判定が効率的に行えることから，グラフ正規化はグラフ同型性判定において重要である．

3.1　McKay のグラフ正規化アルゴリズム

McKay のグラフ正規化アルゴリズムとは，入力としてグラフを受け取り，その正規形を出力するアルゴリズムである．グラフ正規化の過程では，細分化と呼ばれる各頂点をそれらの隣接情報を用いて分類する処理が主となる．細分化においては，まず各頂点を次数によって分類する．次に各分類クラス中の頂点を 1 近傍グラフの隣接情報を用いてさらに分類を行うという処理を繰り返し，分類クラスが増加しなくなった時点で細分化は停止する．分類クラスが増加しなくなった時点の分類を初回安定細分と呼ぶ．ここで，頂点 v を中心とする n 近傍グラフとは v から距離 n 以下で到達する部分グラフである．例として図 2 に示す単純無向グラフを考える．

次数による分類によって

df | be | ac

のように 3 つの分類クラスができる．次に各頂点を中心とする 1 近傍グラフの隣接情報を用いると，

d | f | b | e | a | c

のように全ての頂点を異なる分類クラスに振り分けることができる．この分類は初回安定細分であり，各分類クラスに自然数を割り当てることで正規形を得ることができる．

McKay の正規化アルゴリズムでは，初回安定細分が全ての頂点を異なるクラスに振り分けられない場合，複数の頂点が属するクラスから頂点を 1 つ選んで新たなクラスを追加する．新しくできた分類クラスに対してさらに細分化を行い，全ての頂点を異なるクラスに振り分けるられるまで繰り返し細分化を続ける．

3.2 差分適用グラフ正規化アルゴリズム

差分適用グラフ正規化アルゴリズムは McKay のグラフ正規化アルゴリズムをグラフ書き換え系のために最適化したアルゴリズムである．グラフ書き換えにおいては，グラフのサイズに対して書き換え差分が十分小さい場合が多い．そこで，書き換え前のグラフの正規化情報を書き換え後のグラフの正規化の際に再利用することで，正規化の効率化を図る．差分適用グラフ正規化アルゴリズムは，入力として書き換え前の初回安定細分を表すトライ木，書き換え前のグラフ，グラフ書き換えの差分，書き換え後のグラフを受け取り，書き換え後の初回安定細分を表すトライ木を出力する．

差分適用グラフ正規化アルゴリズムはグラフの各頂点を中心とする n 近傍グラフの隣接情報をトライ木で管理する点が特徴である．グラフ書き換えによってグラフの各頂点の n 近傍グラフの隣接情報は変化しうるため，書き換え後のグラフ正規化のために全ての隣接情報を保存しておく必要がある．頂点毎に共通な n 近傍グラフの隣接情報を効率良く管理するためのデータ構造としてトライ木を用いて頂点を管理する．トライ木のルート (深さ 0) から子ノードへの枝は 0 近傍グラフの隣接情報をラベルとして管理し，同様に深さ n のノードから深さ $n+1$ のノードへの枝は n 近傍グラフの隣接情報をラベルとして管理する．初回安定細分化終了時点のグラフ頂点はトライ木の葉に保持される．図 3 に図 2 に示したグラフの初回安定細分を保持するトライ木を示す．

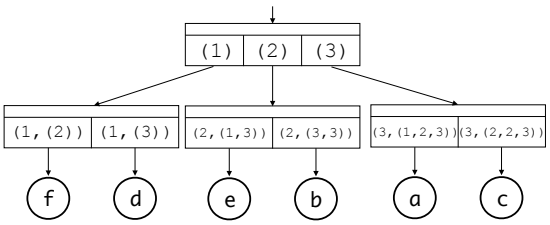

図 3　図 2 のグラフの初回安定細分化を保持するトライ木

4 グラフ書き換え系: LMNtal

本節ではモデル検査器 SLIM が対象とするグラフ書き換え系である LMNtal (の拡張である HyperLMNtal: 以下単に LMNtal と書く) について簡単に述べる．詳細は文献 [8]，[5] を参照してほしい．

LMNtal プログラムはアトム，リンク，膜，書き換え規則 (以下単に規則とも書く) の四つの基本要素から成る．アトム，リンク，膜は基本的なデータ構造である階層グラフを構成する．直感的には，アトムとリンクはグラフ理論におけるノードとエッジに相当する概念であり，膜によってグループ化されたグラフが階層構造を形成する．ただし，アトムは名前を持ち，接続するリンクには順序が存在する．また，アトムの持つリンクはグラフ理論における多重辺や自己ループを形成することができる．

規則は書き換え前のグラフパターンと書き換え後のグラフパターンから成り，書き換え前のグラフパターンにパターンマッチした階層グラフを書き換え後のグラフパターンへ書き換える．

図 4 に LMNtal の構文を示す．

P はプロセスと呼ばれ，階層グラフと規則の多重集合である．0 は中身のないプロセス，p はアトム名，$X_i (0 \leq i \leq m)$ はリンク名である．アトム名は大文字アルファベット以外の英数字で始まる識別子，リンク名は大文字アルファベットから始まる識別子であり，リンクの端点を表現している．アトムの持つリンクの端点には順序が存在する．例えば，

```
a(X, Y), x(X, Y, Z), y(Z, W, W)
```

というプロセスにおいて，a アトムの 1 つ目と 2 つ目のリンクの接続先に x アトムが接続し，x の 3 つ目のリンクの接続先に y アトムが接続している．さら

$$(Process)\ P ::= 0 \mid p(X_1, \ldots, X_m)\ (m \geq 0) \mid P, P \mid \{P\} \mid T\texttt{:-}T$$
$$(Template)\ T ::= 0 \mid p(X_1, \ldots, X_m)\ (m \geq 0) \mid T, T \mid \{T\} \mid T\texttt{:-}T \mid \texttt{@}p \mid \$p$$

図 4　LMNtal の構文

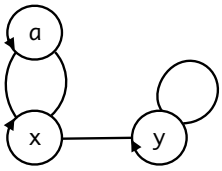

図 5　LMNtal グラフの例

に y アトムは自己ループを持つ．以降アトムの持つリンク端点の順序を図 5 のように矢印を使って表現する．！から始まるリンク名はハイパーグラフにおける多点間をつなぐハイパーリンクを表すが詳細は省く．$p(X_1, \ldots, X_m)$ のような m 本のリンクを持つアトムは m 価のアトムと呼ぶ．P, P は分子と呼ぶ．$\{P\}$ はセルと呼び，プロセス P を膜 {} によってグループ化して階層構造を形成する．以下，誤解のおそれがないときはプロセス P を膜で囲った $\{P\}$ を膜と呼ぶこともある．T :- T は規則であり，:- の左辺にパターンマッチしたグラフ構造を右辺のパターンに書き換える．

T はプロセステンプレートと呼ばれ，規則を構成する :- の左辺および右辺に出現するグラフパターンである．テンプレートはアトム，リンク，膜，規則の他に @p で表されるルール文脈と $\$p$ で表されるプロセス文脈を含む．ルール文脈は膜の中の全ての規則とパターンマッチし，プロセス文脈は膜の中の規則以外のプロセスのうち，明示的に指定されていないもの全体とパターンマッチする．

5　モデル検査器: SLIM

モデル検査器 SLIM はモデル記述としての LMNtal プログラムとモデルの性質記述としての LTL 式を入力として，モデルが性質を満たすかどうかの検証を行う．SLIM は検証の際に，LMNtal の階層グラフ構造を状態，規則によるグラフの書き換えを状態遷移とした状態遷移グラフを構築する．

SLIM の状態空間構築は深さ優先探索によって行われる．したがって，未展開状態のスタックの先頭の状態に対して遷移先の状態を求め，求めた状態が未展開の状態ならばスタックへ追加するという処理をスタックが空になるまで繰り返し行う．遷移先の状態とは，ある状態に対して規則の適用を行うことで得られる書き換え後の階層グラフである．LMNtal においてある規則で書き換え可能なグラフが他の規則でも書き換え可能な場合や，ある規則が複数の部分グラフを書き換え可能な場合に，どの規則がどの部分グラフを書き換えるかは非決定的である．したがって SLIM は，プログラム中の各規則ごとに可能な書き換えパターンを全て試すことによって遷移先の状態を計算している．

SLIM は階層グラフから計算したハッシュ値をキー，状態をエントリとしたハッシュ表を用いて状態の管理を行っている．状態空間構築の際に，遷移先の状態 s のハッシュ値から求めたエントリに既に状態が登録されている場合には，SLIM はグラフの同型性判定を行う．グラフの同型性判定によって，s が登録済みの状態と等しければ，s は既出の状態と判定できる．また，等しくなければハッシュ値に対応するエントリにリンクリンストを構成して s を新規の状態としてハッシュ表に登録する．

例えば，

```
a(1), a(2), a(3), (a(X) :- b(X)).
```

という LMNtal プログラムを SLIM に入力することを考える．書き換え規則 `a(X) :- b(X)` は一価の a アトムを一価の b アトムに書き換える．初期状態を表す LMNtal グラフ `a(1), a(2), a(3)` は上記の書き換え規則によって 3 通りに書き換えらる．書き換えられるたびに a アトムは減少し，最終状態は b アトムのみとなる．SLIM が構築する状態空間を図 6 に示す．

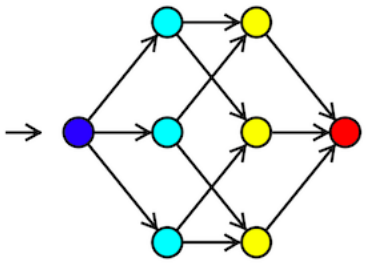

図 6　a(1), a(2), (a(3), a(X) :- b(X)). の状態空間

6　差分適用グラフ正規化アルゴリズムの実装

本節では，モデル検査器 SLIM への差分適用グラフ正規化アルゴリズムの実装について述べる．

差分適用グラフ正規化アルゴリズムでは入力として，書き換え前のグラフの初回安定細分化を表すトライ木，書き換え前のグラフ，書き換え差分，書き換え後のグラフを受け取り，出力として書き換え後のグラフの初回安定細分化を表すトライ木を返す．図 7 は b(1), a(2), a(3) から b(1), b(2), a(3) への書き換え時に発生する差分適用グラフ正規化によるトライ木と書き換え前後のグラフを含んだデータ構造の変化を示した図である．

図中のトライ木ノードにおける 16 進数のラベルは隣接情報のハッシュ値である．隣接情報を直接トライ木で管理するのは非効率であるため，隣接情報の代わりにハッシュ値のみを保持する．図下部に示したグラフ構造は 6.1 節で述べる LMNtal グラフから階層構造を除去したグラフである．図中の書き換え差分は書き換えの前後で削除されたアトムと追加されたアトムへのポインタである．

図中では四角で表されたトライ木の葉が保持するグラフノードはトライ木の更新処理に必要なメタ情報を持ったオブジェクトである．アルゴリズムの上ではグラフ中のアトムと同一視されるが，実装の上では必要となる．このオブジェクトはグラフ中のアトムと双方向ポインタによって接続されている．細分化の際にグラフ中でのアトム同士の接続情報が必要であるため，対応するアトムに接続するポインタが必要であり，また，書き換え差分が示すグラフ中のアトムからトライ木の中の対応するオブジェクトを参照するために逆向きのポインタが必要となる．

また，トライ木によって管理されるアトムは英小文字から始まる名前を持つアトムのみである．これをシンボルアトムと呼び，数字や ’ (クォート) で囲われた文字，文字列を表すアトムをデータアトムと呼ぶ．シンボルアトムは任意個のリンクを持つことが可能である一方で，データアトムの価数は 1，接続先はシンボルアトムに限定されるため，グラフの正規化においてデータアトムはシンボルアトムに付随するデータとして扱う．

6.1　階層構造の除去

差分適用グラフ正規化アルゴリズムは単純無向グラフのみならず，順序付き接続，自己ループ，多重辺などさまざまなグラフ構造に対して適用可能であるが，LMNtal の階層構造には対応していない．そのため，正規化を行う前に階層構造を持つ LMNtal グラフから階層構造を除去した LMNtal グラフに変換する．同時に，LMNtal グラフ中の連結でないグラフ同士を連結し，1 つの連結グラフに変換する．

LMNtal グラフの階層構造は膜によって生まれるため，まず，LMNtal グラフから膜を除去し代わりに’MEM’ という名前のアトムを生成する．次に膜によって同一階層にまとめられていた全てのグラフと生成された’MEM’ アトムをハイパーリンクによって接続する．ただし，一番外側の膜に覆われていない階層においては’MEM’ アトムは生成せず，代わりに同一階層のアトム全てに特別な属性で区別される仮想的なリンクを追加する．

例えば，

a(1), {b(X), c(X), {d(Y), e(Y)}}

のような LMNtal グラフの場合，’MEM’ アトムは 2 つ生成され，各階層に所属するアトムが全てハイパーリンクで接続される．結果として図 8 のように変換される．図 8 ではリンクは実線，ハイパーリンクは点線で表現されている．’GM’ と書かれたアトムは実際には存在しないが，一番外側の階層に所属する全て

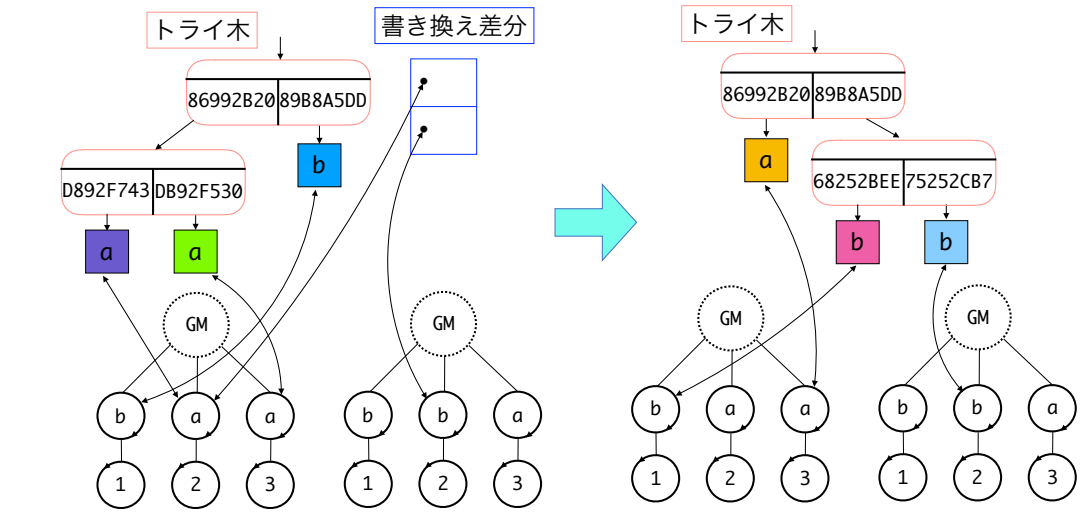

図 7　b(1)，a(2)，a(3) から b(1)，b(2)，a(3) への書き換え時に発生する差分適用グラフ正規化によるトライ木と書き換え前後のグラフを含んだデータ構造の変化

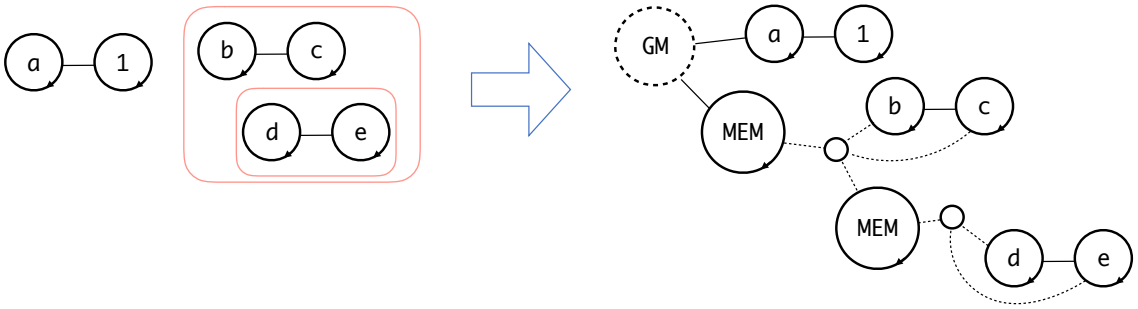

図 8　階層構造除去の例

のアトムは'GM' アトムに接続するリンクを持つ．

6.2　書き換え差分の生成

グラフ書き換え差分は以下の 3 つの集合から成る．

1. 追加されたアトムの集合
2. 削除されたアトムの集合
3. 接続先が変更されたアトムの集合

遷移先状態の正規形を得るために，書き換えによって新たに生成されたグラフとそれらの書き換え元のグラフとの書き換え差分を計算する．書き換え差分の計算には処理系内部でアトムに付けられる ID を用いる．書き換え後のグラフ中で，書き換え前に存在しなかった ID を持つアトムは追加されたアトムの集合へ追加され，書き換え前のグラフ中で，書き換え後に ID が存在しなくなったアトムは削除されたアトムの集合へ追加される．また，書き換えの前後で接続先の ID が変化しているアトムは，接続先が変更されたアトムの集合へ追加される．例えば，b(1)，a(2)，a(3) から b(1)，b(2)，a(3) への書き換えを考える．この書き換えによる書き換え差分の計算では，書き換え前のグラフで 2 に接続していた a アトムが削除されたアトムの集合へ追加され，書き換え後のグラフで 2 に接続した b アトムが追加されたアトムの集合へ追加される．

SLIM においてグラフの書き換えを行う際は，まずオリジナルの書き換え元グラフがエンコードされたバイト列からグラフ構造の再構築を行う．次に，再構築されたグラフに対して書き換え規則の適用する際に，

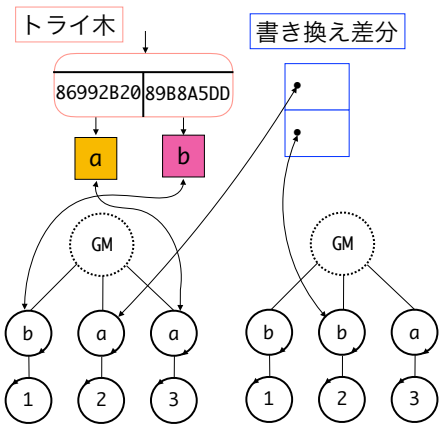

図 9　トライ木からアトムの削除

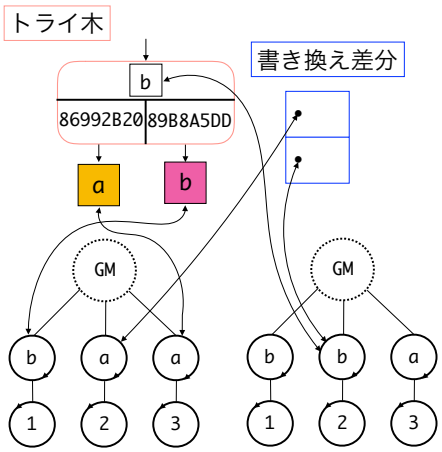

図 10　トライ木へアトムの追加

書き換え元のグラフをコピーしてから適用を行う．これは，書き換え可能な全てのパターンを試し，複数の書き換え結果を生成するために必要な処理である．最終的に書き換えられるグラフはオリジナルのグラフと同型であるが，グラフ中のアトムに付けられる ID は (i) バイト列から再構築した時，(ii) コピーした時の 2 回変更されている．本実装では書き換え元のグラフと書き換え後のグラフの ID の比較によって書き換え差分を計算するために，2 度の ID の変更の際にどの ID がどの ID に変更されたかを対応づけている．まず，バイト列エンコード時に，バイト列の先頭からの index と書き込まれるアトムの ID を組にして記憶しておき，グラフ構造を再構築する際に新しく付けられる ID とエンコード前の ID を対応づける．次にグラフのコピー時に，グラフのトラバース時にコピー前後の ID を対応づける．この対応関係によってオリジナルのグラフと書き換え結果のグラフの ID を比較することが可能となる．

6.3　差分適用によるトライ木の更新

書き換え差分を用いて，書き換え前のグラフに対応するトライ木の更新を行う．ここでは図 7 に示した，`b(1)`，`a(2)`，`a(3)` から `b(1)`，`b(2)`，`a(3)` への書き換えを例として用いる．まず，書き換えによって削除されたアトム全てをトライ木から削除する．この削除によって 3 に接続する `a` アトムが保持されている葉ノードの親ノードが持つ葉ノードは 1 つになる．このような葉ノードを 1 つだけ持つトライノードは冗長であるため，アトムの削除時に削除される．その際に持っていた葉ノードは自身の親ノードに接続する．図 9 で 2 に接続していた `a` アトムをトライ木から削除した直後のトライ木を示す．図では，書き換え前のグラフ中で 2 に接続していた `a` アトムがトライ木から削除され．3 に接続している `a` アトムを保持する葉ノードが移動している． 次に，書き換えによって追加されたアトムをトライ木のルートに挿入する．図 10 では，書き換え後のグラフ中で 2 に接続している `b` アトムがトライ木のルートに挿入されている．最後にトライ木内のアトムに対してルートノードに格納されているアトムから順に細分化を行う．ルートノードに格納されているアトムはそのアトムを中心とする 0 近傍グラフの隣接情報を用いて細分化される．LMNtal グラフにおける 0 近傍グラフの隣接情報とはアトム種別，アトム名，価数である．それらを用いて隣接情報を計算し，対応するトライ木ノードにアトムを移動させる．ルートノードに格納された `b` アトムは既に深さ 1 のトライ木ノードの葉ノードに保持されている `b` アトムと同じアトム種別，アトム名，価数であるため，同じトライ木ノードに移動される．

次に深さ 1 のトライ木ノードに格納されているアトムについて，1 近傍グラフの隣接情報を用いて細分化を行う．深さ 1 のトライ木ノードは `a` アトムを保持する葉ノードを持つノードと `b` アトムを保持する

葉ノードを持つノードが存在するが，a アトムを持つトライ木ノードにはアトムが 1 つしか存在しないので，これ以上細分化は行われない．b アトムが格納されているノードには 2 つのアトムが格納されているため，細分化を行う．これらの 2 つの b アトムはそれぞれ 1 と 2 に接続しているため，細分化によって新たなトライ木ノードが作成され，そのノードの葉ノードにそれぞれ移動する．以上の更新処理によって図 7 の右に示したトライ木が生成される．

6.4 差分逆適用

LMNtal プログラム `a(1), a(2), a(3), (a(X) :- b(X))`. の状態空間構築を行う場合，初期状態 `a(1), a(2), a(3)` に対する遷移先状態は `(b(1), a(2), a(3))`, `(a(1), b(2), a(3))`, `(a(1), a(2), b(3))` の 3 状態である．図 11 は構築中の状態空間を表しており，`b(1), a(2), a(3)` の正規化が終了した時点である．状態を表すノードのラベルは左から順に 1 に接続しているアトム，2 に接続しているアトム，3 に接続しているアトムを表す．また，図中の T ラベルは，現在のトライ木がラベルの付いた状態に対応していることを表す．次に `a(1), b(2), a(3)` の正規化を行うために，トライ木を親状態である `a(1), a(2), a(3)` に対応するように元に戻す必要がある．そのため，`a(1), a(2), a(3)` から `b(1), a(2), a(3)` への書き換え差分をトライ木に逆適用する．これによってトライ木は再び `a(1), a(2), a(3)` に対応するようになり，`a(1), b(2), a(3)` の正規化を行うことができる．

差分逆適用は上述のような複数の遷移先状態の正規化を行う場合だけでなく，下記の 2 つの場合にも行う必要がある．

- 遷移先状態が存在しない場合
- 子ノードが全て展開済みになった場合

これら 2 つの場合では，探索はバックトラックしてその時点でのトライ木が対応している状態の兄弟ノードが展開される可能性があるからである．図 12 の上に示すような `b(1), b(2), b(3)` の正規化が終了した時点について考える．`b(1), b(2), b(3)` の状態から遷移可能な状態は存在しなため，次に展開されるノードは bab ラベルが付いた状態である．この時，差分の逆適用を行い，図 12 の中心に示すようにトライ木を bba 状態に対応するように逆適用により更新する．bba 状態は全ての子ノードが展開済みであるので，さらに逆適用を行うことでトライ木を baa 状態に対応するように更新する．一連の逆適用によってトライ木は baa 状態まで戻り，bab 状態の正規化が可能となる．

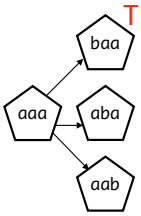

図 11　`b(1), a(2), a(3)` の正規化が終了した時点での状態空間グラフ

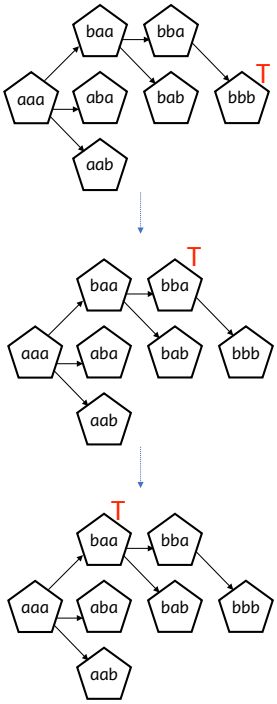

図 12　差分逆適用によってトライ木が戻っていく様子

6.5 一意表現の生成

グラフ同型性判定は正規化によって順序付けられたシンボルアトムの列から生成された一意表現を比較することで行う．差分適用正規化によって，グラフ中のシンボルアトムはハッシュ値の辞書式順で順序付けられる．本実装における一意表現は 2 重のリスト構造である．内側のリストの先頭は起点となるシンボルアトムであり，2 番目以降に引数の順序で接続先のアトムが並ぶ．内側のリストは先頭のシンボルアトムのハッシュ値の辞書式順序で並び，全体として 2 重のリスト構造を構成する．例として b(1)，b(2)，a(3) の一意表現を示す．

```
[[ATOM_a, 3][ATOM_b, 1][ATOM_b, 2]]
```

多重辺や自己ループのあるグラフ構造の一意表現も同様に構成される．図 5 で示した，a(X, Y)，x(X, Y, Z)，y(Z, W, W) の一意表現は下記である．

```
[[ATOM_a, ATOM_x, ATOM_x]
 [ATOM_y, ATOM_x, ATOM_y, ATOM_y]
 [ATOM_x, ATOM_a, ATOM_a, ATOM_y]]
```

多重辺を持つアトムでは接続先のシンボルアトムが隣接リストに 2 回出現し，自己ループを持つアトムでは自分自身が隣接リストに 2 回出現している．

次に，等価なアトムを含むグラフの例として，3 人の食事する哲学者問題を LMNtal でモデリングした際に現れるグラフ構造を以下に紹介する．

```
p_t(L0, R0), f_f(L0, R1), p_t(L1, R1),
f_f(L1, R2), p_t(L2, R2), f_f(L2, R0)
```

p_t アトムは哲学者，f_f アトムはフォークを表現しており，哲学者とフォークが交互に並んだ環状構造を成す．このグラフにおいては，p_t アトム同士はアトム名，価数，および他のアトムとの接続構造が同じであるため区別できない．f_f アトムについても同様である．そのため，初回安定細分化においては 3 つの p_t アトムが属する分類クラスと 3 つの f_f アトムが属する分類クラスの 2 つのクラスが生成される．初回安定細分化によって全てのシンボルアトムが異る分類クラスに振り分けられなかった場合には，書き換え後のグラフの初回安定細分化を表すトライ木の全ての分類クラスをハッシュ値の辞書式順序に並べたリストを構成し，McKay のグラフ正規化アルゴリズムに入力として与えて正規化を行う．McKay のアルゴリズムでは初回安定細分化によって全てのノードが異るクラスに分かれなかった場合，複数のノードが属するクラスから任意の 1 つのノードを選んで新たなクラスを作る．その後，複数のノードが所属する分類クラスについてさらに細分化を行う．ノードの選び方を全通り試し，計算された全ての正規形をハッシュ値の辞書式順序で比較することによって正規形を 1 つに定める．このグラフの一意表現は下記のようになる．

```
[[ATOM_f_f, ATOM_p_t, ATOM_p_t]
 [ATOM_f_f, ATOM_p_t, ATOM_p_t]
 [ATOM_f_f, ATOM_p_t, ATOM_p_t]
 [ATOM_p_t, ATOM_f_f, ATOM_f_f]
 [ATOM_p_t, ATOM_f_f, ATOM_f_f]
 [ATOM_p_t, ATOM_f_f, ATOM_f_f]]
```

7 まとめと今後の課題

本研究ではグラフ書き換えモデル検査器 SLIM におけるグラフ同型性判定処理の高速化を目的として，差分適用グラフ正規化アルゴリズムを SLIM に実装した．実装については多数の公開ベンチマークに対して正常に動作していることを確認し，実験と評価を順次行っている．本研究は差分適用正規化アルゴリズムの SLIM 実装の基礎を与えた．後述するように最適化の余地はいくつかあるものの，アルゴリズムのオーダーは変更していない．

今後の課題として，差分適用グラフ正規化アルゴリズム実装を LTL モデル検査へ適応させることが挙げられる．SLIM は LTL モデル検査アルゴリズムとして NestedDFS [1] と呼ばれる，2 段階の深さ優先探索を用いている．NestedDFS による状態空間の探索順は，DFS とは異なる場合があるため，差分逆適用によるトライ木の更新実装を NestedDFS に適応させる必要がある．

また，書き換え差分の計算においても最適化の余地がある．SLIM におけるグラフ書き換えは，中間表現にコンパイルされた LMNtal ルールの実行によって行われる．この中間表現の実行時に，追加するアトムや削除するアトムを書き換え差分として管理するこ

とが可能である．さらに，書き換えの前後でアトムの再利用を行う最適化手法[9] をより成熟させることによって，書き換え差分をより小さくすることができると考えている．そのほかにも，トライ木の管理方法やアトム ID の対応などについても最適化の余地があると考えている．

謝辞 本研究の一部は，科学研究費基盤研究 (B) 18H03223 の助成を受けて実施した．

参 考 文 献

[1] Holzmann, G., Peled, D., and Yannakakis, M.: On Nested Depth First Search (Extended Abstract), *In The Spin Verification System*, American Mathematical Society, 1996, pp. 23–32.

[2] McKay, B. D.: Practical graph isomorphism, *Congressus Numerantium*, Vol. 30(1981), pp. 45–87.

[3] Rensink, A.: The GROOVE simulator: A Tool for State Space Generation, *International Workshop on Applications of Graph Transformations with Industrial Relevance*, LNCS, Vol. 3062, Springer-Verlag, 2003, pp. 479–485.

[4] Ueda, K., Ayano, T., Hori, T., Iwasawa, H., and Ogawa, S.: Hierarchical Graph Rewriting as a Unifying Tool for Analyzing and Understanding Nondeterministic Systems, *Theoretical Aspects of Computing - ICTAC 2009*, Leucker, M. and Morgan, C.(eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, 2009, pp. 349–355.

[5] Ueda, K. and Ogawa, S.: HyperLMNtal: An Extension of a Hierarchical Graph Rewriting Model, *KI - Künstliche Intelligenz*, Vol. 26, No. 1(2012), pp. 27–36.

[6] 宮原和大, 上田和紀: グラフ書換え系のための効率的なグラフ正規化手法, コンピュータ ソフトウェア, Vol. 33, No. 1(2016), pp. 1_126–1_149.

[7] 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(2011), pp. 4_137–4_157.

[8] 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(2008), pp. 2_47–2_77.

[9] 信夫裕貴, 田辺良則, 上田和紀: LMNtal におけるグラフ書換え操作の Coq による形式化, 日本ソフトウェア科学会大会論文集, Vol. 30(2013), pp. 678–686.