# 構文木の差分を用いた版管理システム向きマージ機能

早瀬康裕[†] 松下誠[†] 井上克郎[†]

オープンソースソフトウェア開発では，世界中に分散した開発者が，版管理システムを用いて並行して作業を行っている．このとき，互いの作業結果を 1 つにまとめるために，版管理システムが提供するマージ機能が用いられる．既存の版管理システムのマージ機能は，汎用性のため行単位によるマージを行うために，ソースコードを対象としたマージの場合，間違った出力をしたり，マージできるべき変更をマージできなかったりするという問題があった．本研究では，ソースコードを対象としたマージにおけるこれらの問題を解決するために，一般的なプログラミング言語が木構造を持つことに着目して，ソースコードを木構造を持つ中間言語に変換したうえで，中間言語に対するマージを行うアルゴリズムを提案する．また本アルゴリズムを，Java で書かれたソースコードを対象として，実際の版管理システム subversion 上に実装した．本システムによるマージと行単位のマージとの比較実験を行った結果，行単位のマージでは正しくマージすることができなかった問題を解決することができ，本システムの有効性が確認できた．

# Merging Algorithm for Revision Control System Using Syntax Tree Deltas

Yasuhiro Hayase,[†] Makoto Matsushita[†] and Katsuro Inoue[†]

In an opensource development process developers work together using a revision control system. While getting multi-developers working products together into a single form, merge feature of revision control systems is used. Nowadays, merge operations in existing systems are commonly implemented with a line-by-line approach that can fail if two changes to the same line of code happen at the same time. In this paper, we propose a merge algorithm for source code that exploit the tree structure of modern programming language grammar: the source code is transformed in an intermediate XML representation and the merge operation is conducted on the transformed version. We give an implementation of the algorithm for the Java language for the subversion revision control system. Experiments shown that the proposed algorithm gives more accurate merge result than the existing line-by-line algorithms.

## 1. はじめに

近年，インターネットの普及により，オープンソースソフトウェア開発[9] が広まっている．典型的なオープンソースソフトウェア開発では，世界中に分散した開発者により，協調かつ並行して開発作業が行われており，開発者は電子メールやメーリングリストによって互いに情報交換を行いながら，版管理システムを用いて生成するプロダクトを管理している[6]．

このような開発では，複数の開発者が並行して作業を行い，その結果を互いに独立して版管理システムへ登録することが頻繁に起こる．このとき，後から自分の作業結果を登録する開発者は，あらかじめ既存の修正内容と自らの修正内容を統合（マージ）した後に自らの修正を登録する場合がある．このマージ処理は，人手によって行われる場合と，版管理システムなどによって機械的に行わせる場合があるが，マージを行う回数が多いことや，作業の正確性，確実性を高めることを目的として，主に版管理システムが行うことが多い．

既存の版管理システムである CVS[1] や subversion[2] などのシステムが提供するマージ機能は，管理対象となるファイルの行を単位としている．しかし，この方法をソースコードに適用する場合，以下のような問題があった．

**間違った出力をしてしまう問題** ある変数を削除する変更と，その変数を利用する文を追加する変更とが，別々の開発者によって並行に行われたとする．この場合，これらの変更の組合せをマージすることはできない．しかし，既存のシステムでは，これらの変更された行が重なっていないときには，マージできないことを検出できず，間違った出力を行う．

† 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

**マージできるべき変更をマージできない問題** ある文を変更する作業と，その文と同じ行にコメントを追加する作業とが，別々の開発者によって並行に行われたとする．これらの変更の組合せはマージすることができる．しかし，既存のシステムでは，変更された行が重なっていることから，マージできない．

ソースコードのマージをより正確に行うための方法として，プログラミング言語の構文情報を用いたマージ処理がある[7]．しかし，プログラミング言語は数多く存在するため，各言語を対象として精度の高いマージ用アルゴリズムを定義するのは困難である．

そこで本研究では，プログラミング言語から独立したマージ用アルゴリズムを提案する．具体的には，一般的なプログラミング言語の文法が木構造を持つことに着目し，ソースコードを木構造を持つ中間言語に変換したうえで，中間言語に対するマージ用アルゴリズムを定義する．この中間言語に対して，Chawathe らの木構造差分計算アルゴリズム *FastMatch*/*EditScript* [5] を改変したアルゴリズムを用いて頂点のマッチングや差分計算を行い，その差分情報を木構造の中間言語を編集する操作の列として表現する．得られた操作の列を，中間言語に適用してマージ処理を行うことで，言語依存の部分とマージ処理の部分を分離する．

また本アルゴリズムを，版管理システムである subversion 上に実装した．本システムはプログラミング言語として Java を対象としており，変数の削除やソースコード断片の移動などを正しく扱うことができる．

さらに，作成したシステムを用いて既存システムとの比較実験を行う．具体的には，サンプルのソースコードおよび，実際のオープンソース開発履歴から抽出した情報を用いて，作成したシステムと行単位のシステムで実際にマージを行い，その結果を比較する．実験の結果，作成したシステムは行単位のシステムに比べてより多くのマージに成功し，提案するアルゴリズムの有効性を示すことができた．

以下，2 章では一般的な版管理について説明し，既存の版管理システムの問題を示す．3 章ではソースコードの構文を考慮したマージシステムを提案し，4 章ではそのシステムの実装について説明する．5 章では作成したシステムの適用実験と考察を行う．6 章では関連研究について説明する．最後に 7 章で本研究のまとめと今後の課題について述べる．

## 2. 版管理システム

本章ではまず，オープンソースソフトウェア開発で一般的に用いられている版管理システムについて述べる．次に，並行して行われた作業の結果をマージする際におきる問題について，例を用いて説明する．

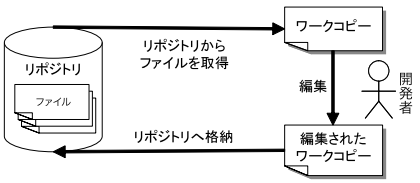


**図 1** 版管理システム
Fig. 1 Revision control system.

### 2.1 版管理システム

CVS に代表される版管理システムは，ソースコードやドキュメントなどのファイルの変更履歴を保存するシステムである（**図 1**）．

版管理システムにはリポジトリと呼ばれるデータベースがあり，開発対象のファイルはすべてリポジトリに保存される．開発者がファイルを編集する場合，リポジトリからファイルの複製（ワークコピー）を取得する．その後，開発者はワークコピーを編集し，リポジトリへ格納する．版管理システムを使ったソフトウェア開発は，この作業を繰り返して行われる．

また，版管理システムの多くは，複数人によるソフトウェア開発をサポートしている．開発者がファイルを取得する際には，他の開発者がそのファイルを取得していてもよく，それぞれの開発者は，取得したワークコピーを自由に編集することができる．複数人によって同時並行的にファイルの編集が行われた場合，それぞれの開発者が自分の編集結果だけを格納すると，最後に格納したファイル以外に行われた編集が失われてしまう．他人の格納した編集結果を失なわないようにするためには，何らかの方法により，自分の編集結果を他人の編集結果とともに取り込んだファイルを作成する必要がある．このファイルを得る処理は，マージと呼ばれる．マージは，版管理システムにより自動的に行われる（**図 2**）が，人間の手による解決が必要になる場合もある．

### 2.2 問 題 点

既存の版管理システムはマージを行単位で行っており，マージのときに同じ行が編集されている場合，かつその場合のみ，衝突として開発者に解決を求める．このことは，不要な衝突を引き起こしたり，不正なマージ結果を得たりするなどという問題がある．以下，それぞれの問題と理想的な解決を示す．

#### 2.2.1 不要な衝突

たとえば，開発者 A と B が同じファイルを取得し

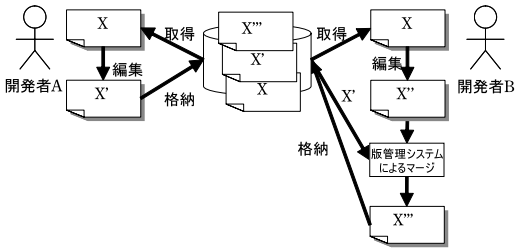


**図 2** マージした結果を格納
Fig. 2 Commit merged result.

て，編集していると仮定する．また，そのファイルに以下のような行が存在したとする．

```
int refs;
```

開発者 A は以下のように，初期値を設定する変更を加え，格納した．

```
int refs=0;
```

開発者 B は A の変更を知る前に，以下のように変数についてのコメントを追加し，格納をしようとしたとする．

```
int refs; /* reference count */
```

開発者 A と B は同じ行を編集しているため，版管理システムはこれを衝突と見なす．この場合，後から格納をした開発者 B が衝突を解決しなければならない．

もしシステムが変更点を正しく把握すれば，初期値とコメントの両方を含んだ，以下のようなコードが得られるはずである．

```
int refs=0; /* reference count */
```

#### 2.2.2 不正なマージ結果

開発者 A と B が同じファイルを取得して編集しており，そのファイルには，以下のように変数を宣言する行が含まれていたとする．

```
int num, sum, avg;
```

開発者 A は変数 avg は不要であると考え，以下のように削除して格納した．

```
int num, sum;
```

開発者 B は A が格納したこと知る前に，以下のように変数 avg を使う処理を追加した．

```
int num, sum, avg;
⋮
avg = sum/num;
```

B が編集結果を格納する前には，A と B の変更点をマージする必要がある．このとき，版管理システムは，両方の編集内容が行として重なりがないため，マージ結果として，以下のような出力を行う．

```
int num, sum;
⋮
avg = sum/num;
```

しかし，このマージ結果は，宣言されていない変数 avg を利用するソースコードとなり，開発者 B の編集意図とは異なるものとなってしまう．もし，外のスコープで，削除された変数 avg と同名の変数が宣言されていたとすると，コンパイルエラーとならず，開発者が問題に気付かない可能性がある．

もし，システムが両方の変更点を正しく把握すれば，マージを行った結果，衝突が起きていることを検出することができるはずである．

## 3. ソースコードの構文を考慮したマージアルゴリズム

本章では，2.2 節で述べた問題を解決するため，ソースコードの構文を考慮したマージアルゴリズムを提案する．

提案するアルゴリズムは，ソースコードを木構造の中間言語に変換したうえでマージを行う．以下，この中間言語をツリーと呼ぶことにする．

ツリーのマージ結果をソースコードに変換したものがアルゴリズムの出力となる．

マージアルゴリズム全体は，以下のようなアルゴリズムを組み合わせて用いることで作成する．

(1) ソースコードをツリーに変換する．
(2) ツリーの差分計算をする．
(3) ツリーの差分適用をする．
(4) ツリーをソースコードに変換する．

まず，図 1 に示したように版管理システムのリポジトリにソースコードをそのまま格納するのではなく，ソースコードを構文解析して作成したツリーを格納する（**図 3**）．

次に，リポジトリに格納された 2 つのツリーから計算した差分を，ワークコピーに適用することでマージを行う．マージ処理の際のデータの流れを表したのが，**図 4** である．マージの入力は，差分計算元のツリー A，差分計算先のツリー B，差分適用対象のツリー C の 3 つである．ツリー A と B はリポジトリから取得し，差分適用対象のツリー C はワークコピーであるソースコードから作成する．次に，ツリーの差分計算アルゴリズムで，差分計算元のツリーから差分計算先のツリーへの差分を計算する．差分は，ツリーを編集する操作の列として表され，これを編集スクリプトと

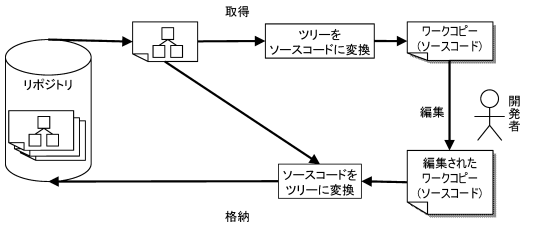


**図 3** リポジトリへのツリーの格納
Fig. 3 Commit a tree made from a source code.

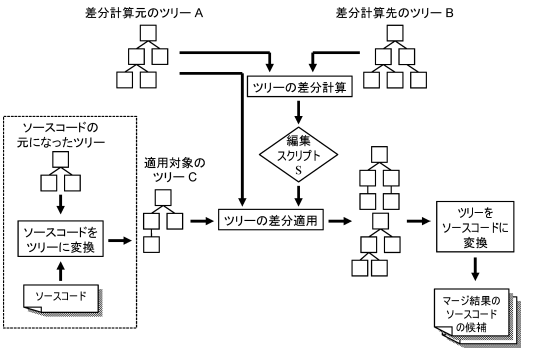


**図 4** マージを行うときのデータの流れ
Fig. 4 Data flow on merging.

呼ぶ．そして，得られた編集スクリプトを，適用対象のツリーに対して適用する．適用した結果として，複数のツリーが得られる．

以下，中間言語であるツリーのデータ構造を説明したあと，リポジトリへのツリーの格納，アルゴリズムの詳細について説明する．

### 3.1 ツリーのデータ構造

マージアルゴリズムの対象であるツリーが持つべき条件は，以下のとおりである．

(1) ソースコードからツリーを作ることができる．
(2) ツリーからソースコードを作ることができる．
(3) 行単位よりも細かい粒度の頂点からなる．
(4) 識別子の宣言と利用の関係を計算することができる．

このような条件を満たすツリーとして，具象構文木（Concrete Syntax Tree）に，空白文字やコメントの情報を加えたツリーを用いる．以下，このデータ構造を単にツリーと表記する．

具象構文木とは，文脈自由文法などで定義された構文から文字列を導出する過程を木構造にしたデータ構造である．具象構文木の頂点は，非終端記号と終端記号である．

具象構文木は空白やコメントなどの情報を持たないので，具象構文木から元のソースコードに戻すことはできない．そこで，終端記号の頂点と同様に，空白文字やコメントを表す頂点を具象構文木に加える．

```
start = normal-element
ChildrenOfNormalElement =
  (text | normal-element | identifier)*
normal-element = element * - identifier {
  attribute id { xsd:Name },
  ChildrenOfNormalElement
}
identifier = element identifier {
  attribute id { xsd:Name },
  attribute ref { xsd:Name }?,
  text
}
```

**図 5** ツリーの表現に用いる XML 文書のスキーマ
Fig. 5 The schema of XML documents for the tree representation.

頂点を識別するために，ツリーの中で重複しない ID を頂点に付ける．

また，ツリーは具象構文木の情報をすべて持っているため，意味解析を行うことで識別子の宣言と利用を計算できる．計算の結果を保存するために，識別子を宣言している頂点の ID を，識別子を利用している頂点に記録する．この情報をリンクと呼ぶ．

以上で述べたツリーのデータ構造は特定の言語に依存しないため，マージアルゴリズムに汎用性を持たせることができる．

提案するシステムでは，ツリーを表現する方法として XML 文書を用いる．以下，XML 文書について説明する．

#### 3.1.1 ツリーの表現に用いる XML 文書

ツリーを表現する XML 文書のスキーマを，**図 5** に RELAX NG Compact Syntax を用いて示す．すべての要素は，ツリーの中で重複しない値を持つ属性 `id` を持っている．

XML 文書のテキストはツリーの葉頂点を表しており，XML 文書のすべてのテキストを深さ優先探索順に出力すると，ツリーを作る元となったソースコードが得られる．

識別子を表す頂点には，`identifier` という名前の要素を用い，その他の頂点を表す要素には，`identifier` 以外の任意の名前を用いる．`identifier` 要素は子要素を持たず，識別子の名前を表すテキストだけを持つ．

識別子の宣言と利用の関係を表現するために，`identifier` 要素は，属性 `ref` を持つことができる．識別子を利用している `identifier` 要素は，その識別子を宣言している `identifier` 要素の属性 `id` の値を，属性 `ref` の値として持つ．

### 3.2 ソースコードをツリーに変換

提案するアルゴリズムでは，マージをツリーのうえで行うために，版管理システムのリポジトリにソース

コードをそのまま格納するのではなく，ソースコードを構文解析して作成したツリーを格納する．

新しいソースコードを作成し，ソースコードから作成したツリーをリポジトリに追加するときには，ツリーの頂点に新しい ID を付ける．

リポジトリに保存されているツリーを元に新しいツリーを作るとき（図 3）には，まず，リポジトリからツリーを取得し，ソースコードに変換する．開発者はソースコードを編集し，リポジトリへの格納を行う．

格納を行う際には，まず，ソースコードから，頂点の ID が付けられていないツリーを作る．そして，新しく作ったツリーと元となったツリーとの間で，類似した頂点の組を探す．この計算をマッチング計算と呼ぶ．

新しいツリーの頂点のうち，類似している頂点が見付かった頂点には，見付かった頂点と同じ ID を付ける．類似した頂点が見付からなかった頂点には，新しい ID を付ける．

マッチング計算には，*FastMatch* [5] にリネーム解析を加えることで，ソースコード向けに改変したものを用いる．

以下，リネーム解析について説明する．

#### 3.2.1 リネーム解析

*FastMatch* は，テキストの類似度に基づいた，任意の構造化テキストに対するマッチング計算アルゴリズムである．そのため，そのままソースコードに適用すると識別子のマッチングの際，意図しないマッチングを生じさせるという問題がおこる．

実際のプログラムでは，似た役割の変数を，その名前の末尾に付けた番号などで区別することは多い．このような変数は，文字列として見た場合には十分似ているため，間違った組合せでも一致していると判定されてしまう．また，テキストの類似度に基づいたマッチングでは，変数名が大きく変更された場合に，変更前後の変数の頂点が同じ頂点であることを検出できない．

このように，識別子をマッチングする際にテキストの類似度を用いるのは適切でない．そこで，識別子のマッチングには，*FastMatch* における他の頂点のマッチングとは異なる，利用関係を利用したアルゴリズムを用いる．

具体的には，識別子のマッチングは以下のような手順で行う．まず，識別子以外の頂点のマッチングを，*FastMatch* アルゴリズムを用いて計算しておく．次に，それぞれのツリーにある識別子の頂点を深さ優先探索順に並べた列を作り，*FastMatch* と同様に頂点のマッチングを計算する．ただし，*FastMatch* アルゴリズムで用いている頂点の一致を検査する関数 *equal* の代わりに，以下に示す条件のいずれか一方を満たす場合に一致すると判定する関数を用いる．

(1) 識別子の名前が完全に一致している．
(2) その識別子を利用している頂点の親が，一定以上の割合で共通している．

このような条件を用いることにより，類似した名前の識別子のマッチングを避けることができ，また，名前の変更された識別子であっても正しいマッチングを計算することができる．

### 3.3 ツリーの差分計算

2 つのツリーの間の差分は，編集スクリプトを用いて表す．編集スクリプトは編集操作の列であり，編集操作は以下の 4 種類である．

(1) `insert`：頂点の追加を表す．引数に，追加する頂点の ID，頂点が格納するデータ，親の頂点 ID，何番目の子供にするかを表す数値の 4 つをとる．
(2) `delete`：頂点の削除を表す．引数に，削除する頂点の ID をとる．
(3) `update`：頂点に格納されたデータの更新を表す．引数に，更新の対象となる頂点の ID と，その頂点に格納する新しいデータをとる．
(4) `move`：部分木の移動を表す．引数に，移動する部分木の根の ID，移動先の親の頂点 ID，何番目の子供にするかを表す数値の 4 つをとる．

対象となるツリーに編集スクリプトの先頭から順に編集操作を適用することにより，ツリーの編集を行う．

一般に，あるツリー A と B があるとき，A を B に変換するスクリプトは無数に存在する．

そこで，編集スクリプトにコストを定義し，コストが最も小さいスクリプトを，ツリー A から B への差分として採用することにする．編集スクリプトのコストは，含まれる編集操作のコストの総和であり，編集操作のコストは差分計算アルゴリズムによって定める．

ツリーの差分を計算するアルゴリズムとして，*EditScript* [5] を採用した．

### 3.4 ツリーの差分適用

ツリー A から B への差分を表す編集スクリプト S を，A 以外のツリー C に適用して新たなツリーを得るアルゴリズムについて説明する．アルゴリズムの入力は，編集スクリプト S と，編集スクリプトを適用する対象のツリー C，そして編集スクリプトを計算する元となったツリー A である（図 4）．

3.3 節で説明したように，編集スクリプトに含まれる編集操作は，対象頂点やツリー内での位置を表すた

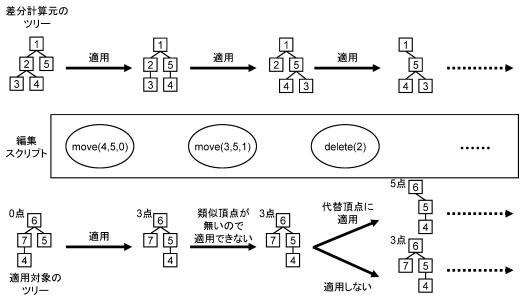


**図 6** 編集スクリプトの適用
Fig. 6 Applying editing script to source code tree.

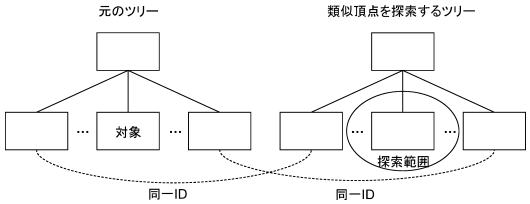


**図 7** 類似頂点の探索：兄弟頂点から探索
Fig. 7 Finding similar nodes from brother nodes.

めに，頂点の ID を引数にとる．しかし，C に S を適用する際に，編集操作の引数となる ID を持った頂点が C にあるとは限らない．そこで，操作する ID を持った頂点が C にない場合，操作対象に類似した頂点や位置を探索して代替する（**図 6**）．

類似した頂点を探すためには，編集元のツリー A に，同じ編集スクリプトを適用した状態のツリーと比較する必要がある．そこで，編集操作の適用を行うときには，差分計算元のツリーに対しても，同時に編集操作を適用する（図 6）．

また，`insert` 操作や `move` 操作は，頂点の追加する位置や，移動先の位置を引数としてとる．ここで，位置は親頂点の ID と，その何番目の子供かという情報で表されるため，直接 ID が付けられていない．そのため，ツリーに編集操作を適用する際には，必ず類似した位置を探索する必要がある．

以下，類似頂点と類似位置の探索方法について説明したあと，編集操作の適用について説明する．

#### 3.4.1 類似頂点と類似位置の探索

類似頂点や類似位置の検索は，以下の 2 つの方法で行い，それぞれの方法で得られた結果をすべて用いる．以下，類似頂点の探索の例を用いて説明するが，類似位置の探索と点数付けも同様の手順で行う．

1 つめの方法では，兄弟の頂点から探索する（**図 7**）．対象の左右にある頂点の組の集合を考える．そのうち，探索するツリー中に同じ ID の頂点があり，その親が共通している頂点の組だけの集合を作る．そこから，

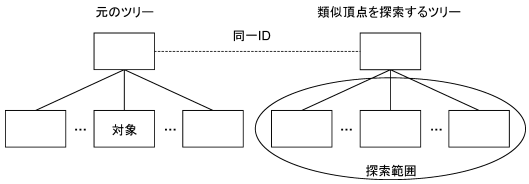


**図 8** 類似頂点の探索：親頂点から探索
Fig. 8 Finding similar nodes from parent node.

元のツリーで，頂点間の距離が最も短い頂点の組だけを取り出す．探索するツリーで，その組と同じ ID を持つ頂点に挟まれた頂点の中から，同じ ID を持つ頂点が元のツリーになく，テキストが類似している頂点を候補として取り出す．対象頂点が，元のツリーの兄弟頂点の中で左端か右端にある場合には，その反対側にある頂点だけを考えて，探索を行う．

2 つめの方法では，親の頂点から探索を行う．**図 8** のように，親の頂点と同一の ID を持つ頂点の子供を候補として，同じ ID を持つ頂点が元のツリーになく，テキストが類似している頂点を候補として取り出す．

2 つめの方法は，1 つめの方法で探した頂点の親の ID が，元のツリーの親の ID と異なる場合にのみ行う．

これらの方法で探索した頂点の候補には点数を付ける．ID の一致する頂点が見つかった場合には 3 点を与える．類似頂点を探した場合には，その頂点の親頂点，左の頂点，右の頂点のそれぞれが一致している場合に 1 点ずつ与える．

#### 3.4.2 編集操作の適用

3.4.1 項で説明したように，編集操作の適用をするときには，編集操作の対象頂点や位置と類似した頂点や位置を探す．

類似した頂点か位置のどちらか一方が見付からなかったときには，その編集操作を適用せず，編集スクリプト中の次の編集操作の適用に移る．

複数の類似頂点や位置が見付かったときには，すべての類似頂点と位置の組合せで代用して操作を適用したツリーを作成する．類似頂点が見付かった場合でも，よく似た頂点が含まれていない場合が考えられる．このような場合は，編集操作を適用しなかった場合のツリーも作成する．

以上に加えて，頂点の削除を行う `delete` 操作を適用するときに，操作の対象となる頂点に子頂点が存在する場合には，対象頂点以下の部分木を削除した場合のツリーと，操作を適用しない場合のツリーを作る．

編集スクリプト中の次の編集操作は，作成したすべてのツリーに対して適用する．

ツリーに編集操作を適用するときには，どれだけ正

確に編集操作を適用できたかの点数を記録する（図6）．1つも編集操作を適用していない状態のツリーの点数は0である．

類似頂点や位置で代用せずに編集操作を適用した場合，すなわち対象頂点のIDと同じIDを持つ頂点に対して編集操作を適用した場合には，新しく作成したツリーの点数は，前のツリーの点数に3を加えた値とする．

類似頂点や位置で代用した場合には，新しく作成したツリーの点数は，3.4.1項で説明した代用した頂点と位置の点数の平均値を編集前のツリーの得点に加えた値とする．たとえば，`move`操作を適用するときに，類似頂点の点数が2点，類似位置の点数が1点であった場合には，1.5点を加える．

### 3.5 ツリーをソースコードに変換

ツリーにはソースコードの情報が含まれているため，頂点に記録された文字列を深さ優先探索順で出力することで，ツリーをソースコードに変換できる．

## 4. システムの実装

3章で述べたアルゴリズムを，既存の版管理システムであるsubversionへの拡張として実装した．

以下，3.2節で述べたリポジトリからのソースコードの取得と格納の実装について4.1節で，3.4節で述べたマージの実装について4.2節でそれぞれ説明する．

### 4.1 取得と格納の実装

subversionシステムは，リポジトリを管理するサーバと，ワークコピーを管理するクライアントからなる．クライアントには，リポジトリからファイルを取得してワークコピーを作るときや，ワークコピーをリポジトリに格納するときに，改行文字を変換したり，特定のキーワードを置き換えたりする機能がある．

ここで，subversionクライアントに，以下の機能を追加する．

(1) ツリーからソースコードへの変換機能
(2) ソースコードからツリーへの変換機能

ツリーの表現には，XMLに基づく独自の記述言語を用いる．システムの拡張は，クライアントにのみ行い，サーバには手を加えない．これらの機能追加を施したsubversionシステムの概略図を**図9**に示す．

ファイルが特別なメタデータ`svn:conversion-handler`を持っていない場合は，既存のsubversionクライアントと同じ処理を行う．

開発者の編集しているソースコードが，メタデータ`svn:conversion-handler`を持っている場合には，ソースコードを格納する際に，クライアントがソースコードをXMLに変換する．クライアントがリポジトリからファイルを取得する際には，XMLをソースコードに変換し，ワークコピーを作る．

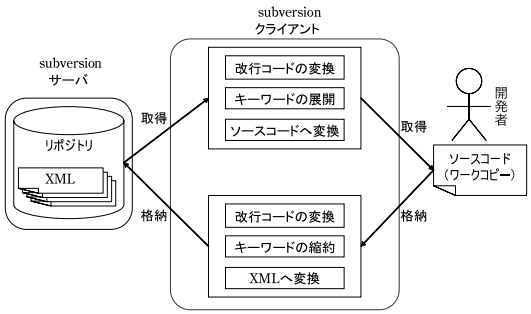


**図9** ソースコードの木構造を考慮した格納と取得
Fig. 9 Checkout and commit recognizing tree structure of source code.

### 4.2 マージの実装

既存のsubversionシステムでは，subversionクライアントが，行を単位としたマージ処理を行う．マージ処理の入力は，ワークコピーと，差分の計算を行うためのリポジトリに格納された2つのファイルである．まず，ワークコピーの改行文字の変換と，キーワードの縮約を行う．そして，リポジトリ内の2つのファイルから，行単位の差分を計算する．そして，得られた差分を，変換したワークコピーに適用する．最後に，適用結果の改行文字と，キーワードの展開を行い，ワークコピーを上書きする．

ここで，subversionクライアントに，以下の機能を追加する．

(1) ツリーからソースコードへの変換機能
(2) ソースコードからツリーへの変換機能
(3) ツリーの差分計算の機能
(4) ツリーの差分適用の機能

(1)と(2)の機能については，4.1節で説明したものである．これらの機能追加を施したsubversionのマージ機能の概略図を**図10**に示す．4.1節で説明したように，リポジトリにはXML文書が格納されており，ワークコピーはソースコードである．

まず，格納のときと同様に，開発者の編集したワークコピーをXMLに変換する，そして，2つのXMLファイルをリポジトリから取得し，差分計算を行う．この差分を，ワークコピーから作ったXMLファイルに適用することで，マージ結果のXMLファイルが得られる．

マージ結果が1つだけ得られた場合は，XMLファイルをソースコードに変換してワークコピーを上書きする．マージ結果が2つ以上得られた場合には，開発

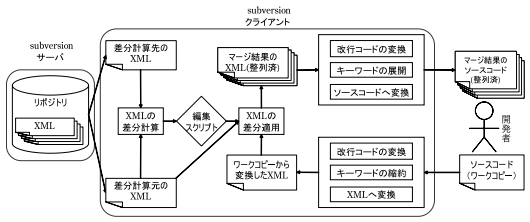


**図 10** 木構造に対応したマージ
Fig. 10 Merging recognizing tree structure of source code.

者が複数のマージ結果の候補から 1 つを選択し，ワークコピーを上書きする．この複数のマージ結果の候補は，3.4.2 項で説明した点数によって整列されている．

## 5. 実　　験

### 5.1 マージ結果の正確性

作成したシステムが正しいマージ結果を出力できるかを確かめるために，サンプルのソースコードを作成し，適用を行った．このソースコードを，オリジナルと呼ぶことにする．オリジナルに対して以下のような変更を加えたソースコード 1，2，3 を作成した．

**ソースコード 1** オリジナルで宣言されていた変数 x を削除した．

**ソースコード 2** 変数 x を利用する新しいメソッドを追加した．

**ソースコード 3** オリジナルで宣言されていた変数 x を改名した．

オリジナルからソースコード 1，2，3 への，ツリーの差分をそれぞれ計算し，差分 1，2，3 として作成した．この差分を，それぞれソースコード 1，2，3 へ適用し，その結果を観察した．また行単位のマージ処理でも，同様の適用実験を行い，木構造によるマージと比較した．実験結果を，**表 1** と**表 2** に示す．表の縦軸はソースコードを表し，横軸は適用した差分を表す．表の対角線は，ソースコードに加えられた変更と同じ差分を適用する意味のない作業となるため，省略した．

表 1 に示すように，行単位のマージは不正なソースコードをマージ結果として出力したり，マージできる変更を衝突と判定したりするなど，すべてのマージ結果で失敗した．

表 2 では，1 件を除いて，マージに成功するか，リンク先が存在しない不正なリンクを発見し，正しい結果を出力することができた．しかし，ソースコード 1 に対して差分 2 を適用したときには正しい結果を得られず，類似位置の探索に失敗して大量の候補を発生させてしまった．

**表 1** 行単位のマージを試みた結果
Table 1 Result of line-by-line merging.

| | 差分 1 | 差分 2 | 差分 3 |
|---|---|---|---|
| ソースコード 1 | | 不正な出力 | 衝突 |
| ソースコード 2 | 不正な出力 | | 不正な出力 |
| ソースコード 3 | 衝突 | 不正な出力 | |

**表 2** 提案手法によるマージを試みた結果
Table 2 Result of tree merging.

| | 差分 1 | 差分 2 | 差分 3 |
|---|---|---|---|
| ソースコード 1 | | 失敗 | 成功 |
| ソースコード 2 | 不正リンク検知 | | 成功 |
| ソースコード 3 | 成功 | 成功 | |

**表 3** 実験 2 の結果概要
Table 3 The result of experiment 2.

| 行単位のマージ | 件数 | 木構造によるマージ | 件数 |
|---|---|---|---|
| 成功 | 71 | 成功 | 71 |
| 失敗 | 13 | 成功 | 9 |
| | | 失敗 | 4 |

### 5.2 実際の開発履歴に対する擬似的な適用

#### 5.2.1 実 験 対 象

オープンソースソフトウェアのリポジトリから，マージが行われたと推測されるリビジョンを抽出し，マージを行う前の状態を擬似的に復元する実験を行った．実験に用いたのは，Eclipse プロジェクトのリポジトリ（22,606 ファイル，162,683 リビジョン）と Jakarta プロジェクト（19,420 ファイル，103,358 リビジョン）から，10 分以内に異なる開発者によって格納が行われた 84 件である．実験を行ったコンピュータは，Xeon CPU 2.80 GHz とメインメモリ 4 GB を搭載しており，OS として Linux 2.6.16 が動作している．

#### 5.2.2 結　　果

実験結果の概要を，**表 3** に示す．行単位のマージで成功した 71 件は，木構造によるマージでもすべて成功した．行単位のマージで失敗した 13 件のうち，木構造によるマージでは 9 件に成功している．

次に，行単位のマージで失敗した場合の詳細を，**表 4** に示す．木構造によるマージに失敗した 4 件のうち，2 件は意味的にマージできない 2 つの編集をマージしようとしたためであった．残りの 1 件は意味的にはマージできる編集であったが，マージすることができず，大量の候補が発生してしまった．

このように，提案手法がマージに失敗したのは 1 件のみであり，行単位のマージと比較して良い結果であった．

**表 5** に出力されたマージ結果の候補の数を示す．84 件のうち 2 件で，1,000 を超える候補が出力されたが，

**表 4** 実験 2 でテキストのマージに失敗した場合の詳細
Table 4 Details when line-by-line merging fails in experiment 2.

| 行単位のマージが失敗した原因 | 件数 | 木構造によるマージ | 件数 |
|---|---|---|---|
| 同じ行への空白文字の追加削除 | 4 | 成功 | 4 |
| 意味的な変更と整形 | 1 | 成功 | 1 |
| 改行コードの変更 | 1 | 成功 | 1 |
| 重なりあった編集 | 2 | 成功 | 2 |
| 後の変更が最初の変更を上書き | 2 | 成功 | 1 |
| | | 失敗 | 1 |
| 意味的な衝突 | 2 | 失敗 | 2 |
| 編集に失敗したファイルの格納 | 1 | 失敗 | 1 |

**表 5** 実験 2 のマージ結果の候補数
Table 5 Number of merge candidates in experiment 2.

| マージ結果の候補数 | 件数 |
|---|---|
| 1 | 57 |
| 2 | 9 |
| 3 | 3 |
| 4 | 3 |
| 8 | 3 |
| 12 | 1 |
| 16 | 2 |
| 18 | 1 |
| 24 | 1 |
| 48 | 1 |
| 1,250 | 1 |
| 1,312 | 1 |

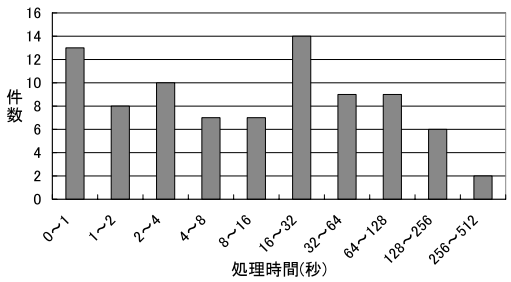


**図 11** 実験 2 のマージにかかった時間
Fig. 11 Execution time in experiment 2.

残りの 82 件では数件から数十件の範囲であった．

4.2 節で述べたように，複数の候補が出力された場合には，候補の持つ点数を参考としてマージ結果を選択する必要がある．当実験では，正しいマージ結果は点数の大きい候補であることが多かったため，多くの候補が出力された場合であっても選択は難しくなかった．

**図 11** にマージ処理にかかった時間を示す．この時間は，Java ソースコードのパース，頂点のマッチング，編集スクリプトの計算，編集スクリプトの適用にかかったすべての時間を足し合わせたものである．マージ時間の平均値は 37.8 秒であり，中央値は 11.8 秒であった．

## 6. 関連研究

Mens [13] は，ソフトウェアのマージ技術を分類した．この分類基準によると，我々の提案するアルゴリズムは，syntactic かつ operation-based に分類される．

Westfechtel [17] は，構文木を用いた 3-way マージアルゴリズムを提案した．このアルゴリズムは，プログラミング言語依存部と非依存部が分けられており，構文と識別子の情報を用いて衝突を検知する．しかし，構文木の差分の計算のために，頂点に付けられたタグを保ったまま編集する必要がある．我々のアルゴリズムは，タグの付いていないソースコードを，任意のエディタで編集することを許している．また，編集元のソースコードが共通でなくても使用できる点や，マージ結果の候補を複数出力する点でも異なる．

Binkley らは，手続きを持つ簡単な言語を対象とした，意味的なマージアルゴリズムを提案した[4]．

Shapiro ら[16]，Kermarrec ら[11] は，編集スクリプト内の操作間で依存関係を計算し，操作を並べ換えることで，複数の編集スクリプトを合成して適用する手法を提案した．

一般的な XML 文書の差分を計算するシステムとして，diffxml [15]，XmlDiff [3]，DeltaXML [12]，XML TreeDiff [8]，diffmk [14]，xydiff [10] などがあげられる．本研究は XML の要素に ID を付与することにより，差分の適用を簡略化している点が異なる．

## 7. まとめ

本研究では，版管理システムのマージ機能の問題を示し，問題への対処として，ソースコードの構文に従ったマージ処理を提案した．また，そのシステムの設計を示した．このシステムにより，マージを行う際の衝突を減らし，マージによっておこる問題を減らすことができる．

実際のソースコードを用いた実験の結果，行単位のマージでは誤って衝突と判断していた多くの場合に，本手法が正しいマージ結果を出力できることが分かった．しかし，いくつかのマージ結果で大量の候補が出力されたり，マージ処理に長い時間がかかったりする場合があることが分かった．

今後の課題は，マッチングの精度を向上させることや，マージ処理の効率を改善することによって，処理速度を改善することがあげられる．Java 以外の言語や，ソースコード以外の文書形式への対応も重要な課題である．また，マージ結果の候補が複数出力された

場合のシステム利用者の労力を，既存の行単位のシステムと比較して評価する必要があると考えられる．

なお，今回 subversion に対して行った拡張は，以下の URL でオープンソースライセンスに基づいて公開している．

http://sel.ist.osaka-u.ac.jp/˜y-hayase/svn-xml/

## 参 考 文 献


1) Concurrent version system. http://www.cvshome.org/
2) subversion. http://subversion.tigris.org/
3) XmlDiff. http://www.logilab.org/projects/xmldiff/
4) Binkley, D., Horwitz, S. and Reps, T.: Program integration for languages with procedure calls, *ACM Trans. Softw. Eng. Methodol.*, Vol.4, No.1, pp.3–35 (1995).
5) Chawathe, S.S., Rajaraman, A., Garcia-Molina, H. and Widom, J.: Change detection in hierarchically structured information, *SIGMOD Rec.*, Vol.25, No.2, pp.493–504 (1996).
6) Fogel, K.F.: *Open Source Development with CVS*, Coriolis Group Books, Scottsdale, AZ, USA (1999).
7) Horwitz, S., Prins, J. and Reps, T.: Integrating noninterfering versions of programs, *ACM Trans. Program. Lang. Syst.*, Vol.11, No.3, pp.345–387 (1989).
8) IBM: XML TreeDiff. http://alphaworks.ibm.com/tech/xmltreediff
9) Open Source Initiative: The open source definition. http://www.opensource.org/docs/definition.php
10) INRIA: XyDiff. http://www-rocq.inria.fr/gemo/XyDiff/
11) Kermarrec, A.-M., Rowstron, A., Shapiro, M. and Druschel, P.: The icecube approach to the reconciliation of divergent replicas, *PODC '01: Proc. 12th annual ACM symposium on Principles of distributed computing*, New York, NY, USA, pp.210–218, ACM Press (2001).
12) Monsell EDM Ltd: DeltaXML. http://www.deltaxml.com/
13) Mens, T.: A state-of-the-art survey on software merging, *IEEE Trans. Softw. Eng.*, Vol.28, No.5, pp.449–462 (2002).
14) Sun Microsystems: diffmk. http://www.sun.com/xml/developers/diffmk/
15) Mouat, A.: Xml diff and patch utilities (June 2002). http://diffxml.sourceforge.net/
16) Shapiro, M., Rowstron, A. and Kermarrec, A.-M.: Application-independent reconciliation for nomadic applications, *EW 9: Proc. 9th workshop on ACM SIGOPS European workshop*, New York, NY, USA, pp.1–6, ACM Press (2000).
17) Westfechtel, B.: Structure-oriented merging of revisions of software documents, *Proc. 3rd international workshop on Software configuration management*, New York, NY, USA, pp.68–79, ACM Press (1991).


(平成 18 年 3 月 2 日受付)
(平成 18 年 11 月 2 日採録)

**早瀬 康裕**（学生会員）

平成 14 年大阪大学基礎工学部情報工学科卒業．現在，同大学大学院情報科学研究科博士後期課程在学中．ソフトウェアプロセスの研究に従事．

**松下 誠**（正会員）

平成 5 年大阪大学基礎工学部情報工学科卒業．平成 10 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．平成 14 年大阪大学情報科学部コンピュータサイエンス学科助手．平成 17 年同専攻助教授．博士（工学）．ソフトウェアプロセス，オープンソースソフトウェア開発の研究に従事．

**井上 克郎**（正会員）

昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59〜61 年ハワイ大学マノア校情報工学科助教授．平成元年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．平成 14 年大阪大学情報科学部コンピュータサイエンス学科教授．工学博士．ソフトウェア工学の研究に従事．日本ソフトウェア科学会，IEEE，ACM 各会員．