# ソフトウェア変更作業の
# 見積りと支援に関する研究

2007年4月

早瀬　康裕

# ソフトウェア変更作業の
# 見積りと支援に関する研究

提出先　　大阪大学大学院情報科学研究科

提出年月　2007年4月

早瀬　康裕

# 内容梗概

コンピュータシステムの発展に伴い，有用なソフトウェアが様々な形で蓄積されている．コンピュータシステムを取り巻く環境の変化にあわせて，ソフトウェアを変更しなければならない機会は多く，変更には多大なコストがかかっている．例えば，ある機能を持ったソフトウェアを作成する際，新規にソフトウェアを作成するよりも，既存のソフトウェアに変更を加えて機能を実現した方が低コストに作成できるため好ましい．しかし，既存のソフトウェアに対する変更は，新規にソフトウェアを開発する場合に比べて，既存のソフトウェアの理解や回帰テストが必要になるなど，多くの点で異なる．

既存のソフトウェアに対する変更の一例として，ソフトウェア保守が挙げられる．ソフトウェア保守とは，ソフトウェアのリリース後にそのソフトウェアに対して行われる変更全てであり，環境変化への適合や欠陥修正，機能追加などといった理由により行われる．長期間にわたって使用されるソフトウェアではソフトウェア保守にかかるコストは非常に大きいことが知られている．

もう一つの例として，近年急速な発展を遂げたオープンソースソフトウェア開発が挙げられる．オープンソースソフトウェアは世界の誰であってもソースコードを入手して変更することが出来るという特徴がある．オープンソースソフトウェア開発は，世界中の開発者らによる変更を共有することで行われるが，世界中に分散した開発者による変更を事前に知ることは困難であるため，同時かつ独立に行われた複数の変更を一つのソースコードに集約（マージ）しなければならない．

そこで本論文では既存のソフトウェアに対する変更に着目し，ソフトウェアの変更をより正確に行う手法の提案，実装を行う．具体的には特に以下の問題点に着目する．

1. 保守コストの見積り手法が充分でないこと

2. マージ処理が行単位で行われていること

まず， 1 で述べている問題について説明する．前述の通り，ソフトウェア保守にかかるコストは大きいため，その見積りは重要な問題である．現在，ソフトウェア保守コストの見積りは新規開発の見積り手法を流用して行われているか，熟練

者の経験に基いて行われていることが多い．しかし，新規開発の見積り手法は保守対象のソフトウェアが存在することを前提としておらず，保守作業にかかる労力のうち大きな割合を占めると言われる理解やテストの労力を考慮していないため，高い見積り精度を得ることはできない．また，熟練者による見積りにも定量的な根拠に欠け，見積り作業者によって見積り値が異なるという問題がある．そこで保守作業の変更による影響範囲の複雑さを表わしたメトリクスを提案する．提案するメトリクスは保守対象のソフトウェアとそれに対する変更要求から自動的に計算可能であるため，作業者に依存しない見積りが可能となる．実際に保守作業を行う実験によって作業時間とメトリクスの値を調べた結果，既存のメトリクスに比べて提案するメトリクスがより高い相関を持つことを確認した．

次に，2 で述べている問題について説明する．オープンソースソフトウェア開発におけるマージ処理は，版管理システムと呼ばれる，ソフトウェア成果物とその変更履歴を管理するソフトウェアによって行われるのが一般的である．しかし，既存の版管理システムのマージ機能はソースコードの構造を考慮せず，行単位で行われるため，間違った結果を出力してしまうことがあった．そこでソースコードの構文木を用いたマージ手法を提案する．本手法では，ソースコードに対する変更を構文木の部分木に対する操作と考え，その操作を別の構文木に適用することで複数の変更を一つにまとめる．既存の行単位の手法に比べて，本手法がより正確なマージ結果を得られることを適用実験によって確認した．

# 主要論文

[1] Yasuhiro Hayase, Makoto Matsushita, Katsuro Inoue: “Revision Control System Using Delta Script of Syntax Tree”, 12th International Workshop on Software Configuration Management (SCM 2005), pp. 133-149, Lisbon, Portugal, September 5-6, 2005 (国際会議)

[2] 早瀬 康裕, 松下 誠, 井上 克郎: “構文木の差分を用いた版管理システム向きマージ機能”, 情報処理学会論文誌, Vol.48, No.2, pp. 858-867, 2007 年 2 月. (学術論文)

[3] 早瀬 康裕, 松下 誠, 楠本 真二, 井上 克郎, 小林 健一, 吉野 利明: “影響波及解析を利用した保守作業の労力見積りに用いるメトリクスの提案”, 電子情報通信学会論文誌 D [採録決定] (学術論文)

# 謝辞

本研究の全般に関し，常日頃より適切なご指導を賜わりました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に，心から深く感謝申し上げます.

本論文を執筆するにあたり，適切なご助言とご指導を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 増澤 利光 教授，同 楠本 真二 教授に心から感謝致します.

本論文を執筆するにあたり，直接具体的なご指導を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授に心より感謝致します.

本研究を行うにあたり，ご助言やご指導を頂きました，富士通研究所 吉野 利明 氏，小林 健一 氏に感謝致します.

最後に，井上研究室の皆様のご助言, ご協力に御礼申し上げます.

# 目 次

# 図 目 次

# 表 目 次

# 第1章　まえがき

近年，ソフトウェアは社会的基盤の一部にも浸透しており，その重要性は日々増大している．それだけ作成されたソフトウェアは不具合のない頑健なソフトウェアであることが強く求められている．一方，ソフトウェアの担う責務の膨大さから，ソフトウェアの大規模化，複雑化は避けられず，ソフトウェア開発に要する費用は増加の一途を辿っている．このように高品質なソフトウェアを低コストで開発するという，相反する要望を満たすことを目的とする研究が活発に行われている．このような研究はソフトウェア工学と呼ばれている．

ソフトウェア工学において行われた研究にはさまざまなアプローチが存在するが，本研究では既存のソフトウェアに対する変更に着目する．コンピュータシステムの発展に伴い，様々な種類の大量のソフトウェアが資産として蓄積されている．ソフトウェアを開発する際，蓄積された既存のソフトウェアに変更を加えて実現することが出来れば，新たにソフトウェアを開発する場合に比べて小さなコストで要求を実現することが出来る．一方，環境の変化や不具合のために，コンピュータシステムを構成するソフトウェアに変更を加えなければならないことは多い．既存のソフトウェアに変更を加える機会は増大しているため，上記のような変更を効率的に行う必要がある．

既存のソフトウェアに対する変更の例として，ソフトウェア保守作業が挙げられる．ソフトウェア保守作業とは “納入後，ソフトウェアに対して加えられる，フォールト修正，性能または他の性質改善，変更された環境に対するプロダクトの適応のための改訂” と定義されている [36]．一般に，長期間にわたって使用されるソフトウェアでは，総所有コストのうち保守コストが大きな割合を占めることが知られている [44][51]．

また，近年，オープンソースソフトウェア開発 [23] が急速に普及している．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いる．オープンソースソフトウェアの利用者はソースコードを入手し，変更することができるため，ソフトウェアの開発に容易に参加できる．このため，オープンソースソフトウェアは常に変更を受けており，その作業の効率化は重要な課題であると言える．

このように，ソフトウェアに対する変更作業は，ソフトウェア開発の中で重要

性を増してきており，その作業を効率的かつ正確に行う必要があると言える．以下，コスト見積り手法と，オープンソースソフトウェア開発支援の研究について取りあげる．

## 1.1 ソフトウェアの変更作業

### 1.1.1 作業手順

Boehm[9] や Martin ら [37] によると，ソフトウェアの変更作業は以下のような手順で行われる (図 1.1)．

1. 利用者による欠陥の発見や，環境の変化，機能追加の需要などにより，変更要求が発生する．
2. 変更要求の内容を理解し，対象のソフトウェアのうち変更される部分と，変更によって影響を受ける範囲を特定し，見積りを行う．
3. 実際に変更を行う．
4. 変更が正しく実装されていることと欠陥が作りこまれていないことを確認するためにテストを行う．
5. 最後にソフトウェアをリリースする．

もし，このような作業が効率的かつ正確に行われなかった場合，見積もりと実際にかかるコストが大きく異なってしまう．

### 1.1.2 コスト見積りの既存手法

一般に，新規ソフトウェアの開発にかかるコスト見積りでは，要求仕様からファンクションポイント [25] 等の機能量などを計算することで作業量を予測することが多い．近年では，開発のより早い段階での見積り手法として，ユースケース図からの見積り手法が提案されている [27]．

一方，ソフトウェア保守コストの見積りは，新規開発の手法を用いて行うか，熟練者が経験に基いて行うことが多い．保守コストの見積りを改善するために，以下のような研究が行われている．Fioravanti ら [14] は，適応保守の見積りに有用なメトリクスを提案し，実際に見積りを行った．Sneed [46] は，保守による影響範囲の大きさを様々な要因で補正して，工数を見積る手法を提案した．Jørgensen [26] は，実際の保守作業で様々なメトリクス値を収集し，見積りに有効なメトリクスと見積り手法を調べた結果，変更行数と工数との相関が高いことを明らかにした．

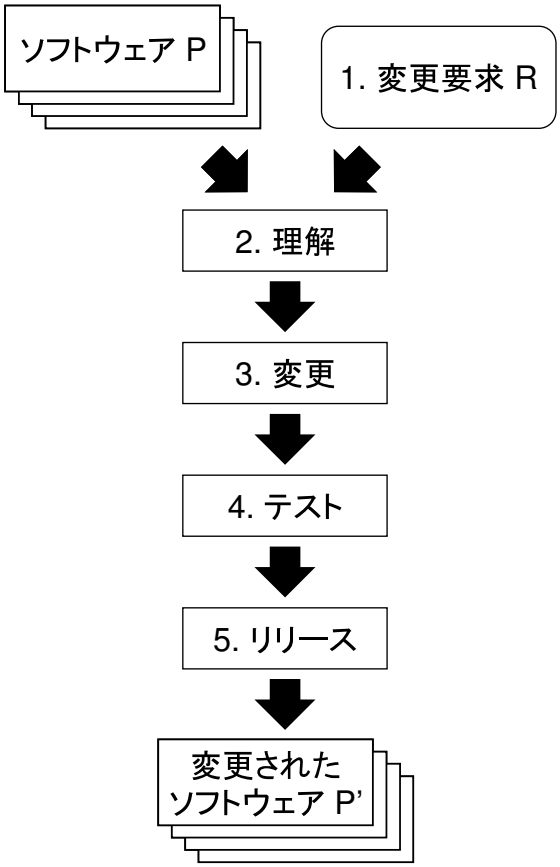


図 1.1: ソフトウェアの変更作業

## 1.2 オープンソースソフトウェア開発

### 1.2.1 オープンソースソフトウェア開発の概要

近年，多くのオープンソースソフトウェアが開発され，様々な環境で利用されるようになってきている．このため，オープンソースソフトウェアの社会的な重要性が高まっている．オープンソースソフトウェアのソースコードは誰でも入手し，変更することが出来るため，開発に参加することが容易であるという特長がある．

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協同して開発を行うために (1) メーリングリスト，(2) 版管理システム，(3) 欠陥追跡システム，の 3 つのシステムを使うのが一般的である (図 1.2)．以下，3 つのシステムについて説明する．

(1) メーリングリスト

オープンソースソフトウェア開発では，開発者が世界中に分散しているため，開発者間の意思の疎通はメールを用いて行うのが一般的である．このとき，メールを個々の開発者同士が直接送りあうと，メールの送り忘れが起きたり，メールの記録が公開される場所に残らないために，他の開発者と知識の共有ができないといった問題がある．

そこで，開発に関わる議論のやりとりには，メールの同報配信システムであるメーリングリストを用いる．メーリングリストに送られたメールはアーカイブと呼ばれるデータベースに保存されるため，後からメールの検索による知識の共有が可能となる．

(2) 版管理システム

ソフトウェアに対する変更を一元化して管理し，変更の記録を残すために，オープンソースソフトウェア開発では版管理システムを用いる．

版管理システムは，リポジトリと呼ばれるデータベースを持ち，成果物を保存する．リポジトリには過去の任意の時点での成果物が保存されている．このため，開発者はいつでも過去の成果物の状態や，変更の内容を参照することができる．

また，版管理システムは複数人が同じ成果物を同時に変更することを可能にする仕組みを持っている．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作業を行うため，成果物を同時に変更することは避け難く，このような仕組みは必須と言える．同時に変更された成果物は，マージと呼ばれる処理によって変更点をまとめ，新たな成果物としてリポジトリに保存される．

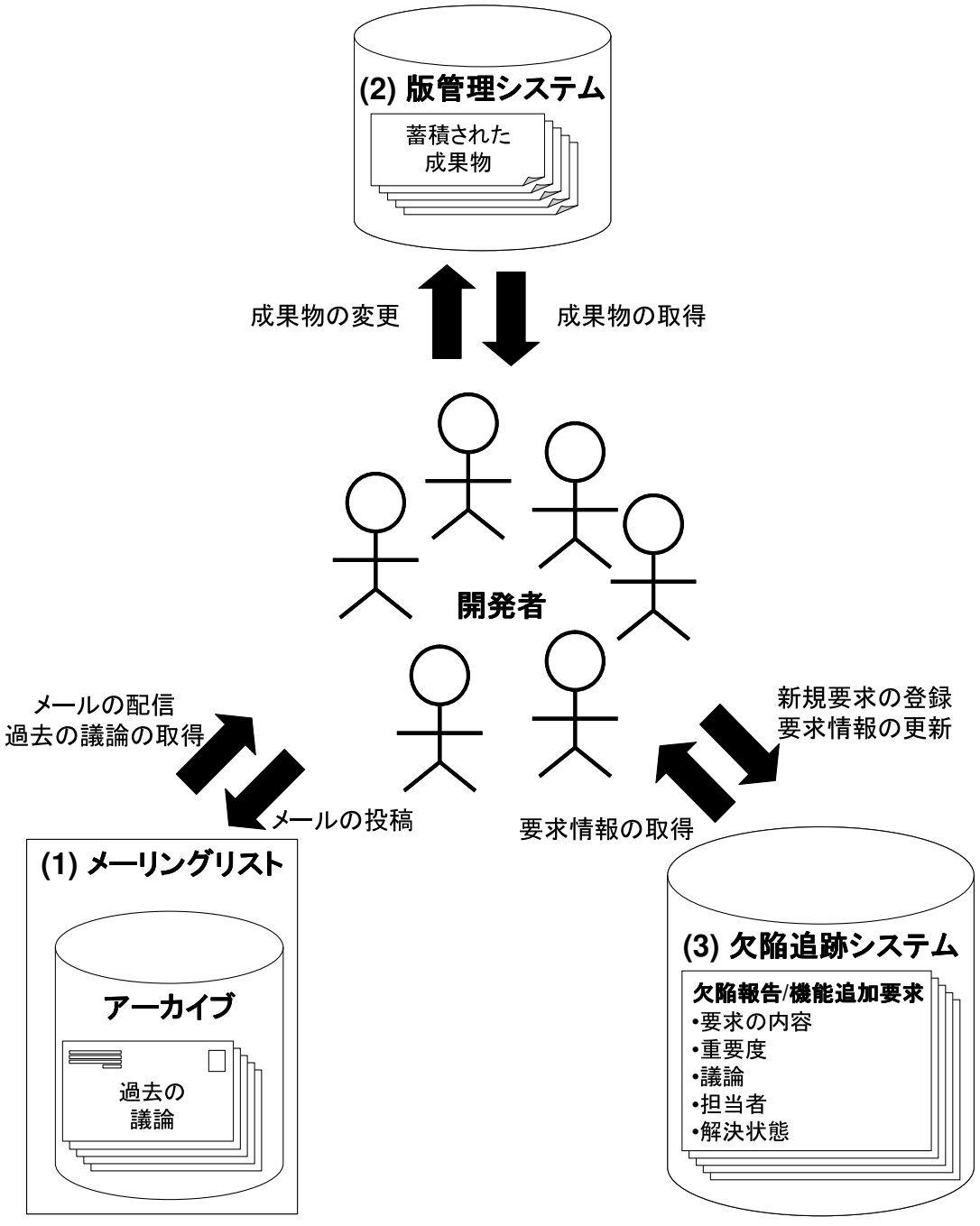


図 1.2: オープンソースソフトウェア開発

一般的な版管理システムでは，マージ処理はテキストファイルの行を単位として行われる．

(3) 欠陥追跡システム

利用者によって発見された欠陥や，ソフトウェアに対する機能追加の要望を管理するために，オープンソースソフトウェア開発では欠陥追跡システムが用いられる．

一般に，欠陥管理システムは以下のような機能を持っている．

- 欠陥修正や機能追加の要求を登録する
- 登録された情報を閲覧する
- 要求の内容について議論する
- 作業担当者を決定する
- 解決状態を管理する

オープンソースソフトウェア開発は上記のようなシステムにより行われているが，世界中に分散した開発者同士がコミュニケーションを取るためのコストは大きく，改善の余地もまた大きい．一方，オープンソースソフトウェアでは文書の整備がしばしば不十分であり，開発への参加が容易であるという特長を損っている．これらの問題を解決するために，上記システムを活用して，開発者を支援しようという試みがなされている．

### 1.2.2 オープンソースソフトウェア開発支援

1.2.1 節で説明した 3 つのシステムを利用して，オープンソースソフトウェア開発を支援する研究が行われている．

- **3 つのシステムに蓄積された情報を統合**

  1.2.1 節で説明したように，ソースコード等の成果物は版管理システムに，議論はメーリングリストアーカイブに，欠陥の情報は欠陥追跡システムに，別々に保存される．これらの情報は互いに関連しているため，ソフトウェアを正しく理解するためには，全ての情報を人手で集めなければならなかった．佐々木らは，3 つの情報の関連を取得・統合し，開発者が簡単に検索できるシステムを作成した [43].

- **リポジトリマイニング**

  版管理システムのリポジトリには，開発初期からのソフトウェアに対する変更が全て記録されている．この記録から，ソフトウェアを理解，変更するために役立つ情報を抽出する研究が行われている．

  Zimmermann ら [52] は，モジュールやディレクトリをまたいで更新が頻繁に行われた場合に，再構築を促す手法を提案した．Hassan ら [20] は，ソフトウェアの更新の履歴を用いて，ソースコード中のある部分に変更を加えた時，その変更が他のどの部分に影響を与えるかを予測する研究を行った．

  中山ら [42] は，ソースコードの変更履歴から関数の利用方法を抽出する手法を提案した．

  楠田 [32] は，メソッドの同時更新履歴から，クラスを，そのクラスが実装している機能毎に分類し，ソフトウェアの理解に役立てる手法を提案した．

  川口ら [28] は，ソースコードの変更履歴の中で，コードクローンがどのように変化したかを調査した．

- **ソースコードを対象としたマージ機能**

  一般的な版管理システムの持つマージ機能は，テキストファイルに対する変更を，行単位で統合する．しかし，行単位のマージは，ソースコードの構造を無視するため，正しくない結果を得てしまう場合がある．この問題を解決するため，ソースコードのマージ技術について様々な研究が行われてきた．

  Bernhard Westfechtel [49] は，構文上のマージアルゴリズムを提案した．この手法では，開発者はタグ付けされたソースコードを編集する．

  David 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 [8]．

## 1.3 既存手法の問題点

これまでにソフトウェアに対する変更についての研究について述べた．本研究では以下の 2 点に着目する．

- ソフトウェア保守のコスト見積り手法が適切でない

  新規ソフトウェアの開発コストの見積り手法は，既存のソフトウェアを考慮しないため，保守コストの見積りに用いるのは適切でない．また，熟練者の

経験に基づく方法は，定量的な方法でないために見積り作業者によって予測が異なるという問題がある．

過去の研究で提案された保守コストの見積り手法も，変更行数などの保守工程の後半でしか入手できない情報を使うため，早い段階での見積りに用いることができないという問題がある．これを解決するためには，保守工程の早い段階で入手可能な情報を用いた見積り手法が必要となる．

そこで本研究では，保守工程の早い段階で入手可能な情報である，変更対象のソフトウェアと，保守作業によって変更されるモジュールの情報から機械的に計算可能なメトリクスを提案する．

- 版管理システムのマージ機能の問題

  一般的な版管理システムに組込まれているのは行単位のマージであるが，これはソースコードの構造を考慮しないため，ソースコードのマージに用いるのは適切でない．

  また，過去の研究で提案されたマージシステムは，版管理システムに組込まれていなかったり，専用のエディタを必要とするか，ソースコードのまま編集することができないといった問題がある．

  これを解決するために，本研究では任意のエディタを用いてソースコードを編集できるマージシステムを，既存の版管理システムに組込む．

## 1.4 本論文の概要

本研究では既存のソフトウェアに対する変更を，正確に実行する手法を提案し，その評価を行う．本論文では，これまでに提案した二つの手法について述べる．

1. 影響波及解析を利用した保守作業の労力見積りに用いるメトリクスの提案

   ソフトウェア保守の労力を見積る際には，客観的な基準を用いずに，熟練者が経験に基づいて見積りを行うことが多かった．本研究では，ソースコードを対象とした影響波及解析手法を用いて，保守作業の労力を見積るメトリクスを提案する．メトリクスの値は，保守作業によって影響を受ける部分のソースコードの複雑度に重みをつけて足し合わせることで求める．本論文では，実際に保守作業を行う実験によって提案したメトリクスと既存のメトリクスを比較し，提案するメトリクスについて考察する．

2. 構文木の差分を用いた版管理システム向きマージ機能

   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．この時，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，マージ出来るべき変更をマージ出来なかったりする問題があった．

   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上で，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．本論文では，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通じて，本システムの有効性について議論する．

以下，第 2 章では保守作業の労力見積りに用いるメトリクスについて述べる．第 3 章では構文木の差分を用いたマージ機能について述べる．最後に第 4 章で本論文の研究についてまとめ，今後の研究方針を述べる．

# 第2章　影響波及解析に基いた保守作業の労力見積り向きメトリクス

## 2.1　導入

ソフトウェア保守とは，文献 [4] では，“納入後,（ソフトウェア）プロダクトに対して加えられる，欠陥修正，性能または他の性質改善，変更された環境に対するプロダクトの適応のための改訂”と定義されている．そして，保守作業を，修正保守（発見された欠陥を取り除く），適応保守（環境が変化した際にプロダクトを利用可能に保つ），完全化保守（性能または保守性を改善する），予防保守（潜在的な欠陥が顕在化する前に対処する）のいずれかに分類している．一般に，長期間にわたって使用されるソフトウェアでは，総所有コストのうち保守コストが大きな割合を占めることが知られている [44][51]．そのため，保守作業に必要な労力を早い段階で見積もることが求められている．

一般に，保守作業は 図 1.1 のような手順で行われる [47]．まず，既存のプロダクト $P$ に対し，環境の変化や機能追加の要求，障害の発生により，変更要求 $R$ が発生する．次に，その変更を行った場合の影響を調査し，実際に変更して，テストを行い，変更によって欠陥が作りこまれていないことを確認してプロダクト $P'$ をリリースする．

現在，保守作業の見積りは，熟練者の経験に基いて行われたり，新規開発の見積りと同様の手法が用いられたりすることが多い．しかし，熟練者による見積りは，見積りを行う人によって結果が異なるという問題がある．また，新規開発の見積りは要求仕様からファンクションポイント [25] などの機能量などを計算することで作業量を予測するが，保守作業は要求仕様から影響を受けるだけではなく，既存のプロダクトの品質や複雑さなどからも大きな影響を受けるため，新規開発向けの見積り手法をそのまま適用するのも適切でない．

見積りに用いるべきメトリクス値と，見積りに用いる計算手法について，Jørgensen [26] は，実際の保守作業で様々なメトリクス値を収集し，見積りに有効なメトリクスと見積り手法を調べた結果，変更行数と工数との相関が高いことを明らかに

した．Sneed [46] は，保守による影響範囲の大きさを様々な要因で補正して，工数を見積る手法を提案した．Fioravanti ら [14] は，適応保守の見積りに有用なメトリクスを提案し，実際に見積りを行った．これらの手法は，保守作業の早い段階で得ることができない変更の具体的な内容を用いたり，適用対象が適応保守に限られたりする問題があった．

そこで我々は，ソースコードと変更要求のみから計算できるメトリクスを提案する．ソースコードと変更要求は保守作業の早い段階で得られる情報であるため，様々な保守作業の見積りにメトリクスを使用することができる．提案するメトリクスは，プロダクトを構成する各モジュールの変更に必要な労力と，モジュール間の関係の複雑度を反映している．メトリクスを計算するために，プロダクトを有向グラフを用いてモデル化する．

本メトリクスを評価するために，ある一定の条件の下で実際に保守作業を行う実験を行い，提案するメトリクスや既存のメトリクスと労力との関係を調べた．その結果，提案するメトリクスの有効性を確認した．

以降， 2.2 節で，保守対象のプロダクトをモデル化し，メトリクスとその計算方法について説明する．2.3 節では提案したメトリクスの評価実験について述べ，2.4 節で関連研究について説明する．2.5 節で実験やメトリクスの計算方法について議論し，最後に， 2.6 節でまとめと今後の課題について説明する．

## 2.2 保守ポイント

本節では，プロダクトと変更要求をモデル化し，保守工数の見積りに用いることのできるメトリクス「保守ポイント」を定義する．

### 2.2.1 プロダクトモデル

一般に，保守対象のプロダクトは複数のモジュールから構成される．このプロダクトを，有向グラフを用いてモデル化したものをプロダクトモデル $G_P = (M, I)$ と呼ぶ．プロダクトモデルの例を 図 2.1 に示す． $M$ はモジュールの集合， $I$ は有向辺の集合で影響波及関係を表す．モジュール m1 から m2 への有向辺は，モジュール m1 に何らかの変更作業を行なった場合に，モジュール m2 にも作業が発生することを表す．

各頂点 $m$ もモジュール単体の保守作業の労力を表す値 $\mathrm{eff}(m)$ (0 以上) を持つ．また，各有向辺 $s$ は，影響の強さを表す重み $w(s)$ (0 以上 1 以下であり，大きな値ほど強い影響を表す) を持つ．この $w(s)$ の値は， s とその起点 m1 と終点 m2 だ

けがあると仮定した場合に， m1 に変更を加えるときに発生する m2 の理解・テストの労力の eff(m2) に対する割合を表している．

保守によって変更作業を行う必要のあるモジュールの集合を，変更要求 $R$ ($R \subseteq M$) いう．

### 2.2.2 保守ポイントの定義

プロダクトモデル $G_P$ と変更要求 $R$ から得た影響範囲の情報を用いて，保守作業の労力の指標「保守ポイント」を定義する．

プロダクトモデル $G_P$ 上の経路 $p$ を構成する辺の系列を $s_1, s_2, \ldots, s_i$ とする．今， $p$ の重み $v(p)$ を次のように定義する．

$$v(p) = \prod_{j=1}^{i} w(s_j)$$

次に， $m_1$ が変更されたときに， $m_2$ にどの程度の作業が発生するかを表す値である影響度 $\mathrm{imp}_m(m_1, m_2)$ を以下のように定義する．

$$\mathrm{imp}_m(m_1, m_2) = \begin{cases} 1 \quad (m_1 = m_2 \text{ のとき}) \\ 0 \quad (Q(m_1, m_2) = \phi \text{ とき}) \\ \max(\{v(p) | p \in Q(m_1, m_2)\}) \\ \quad (\text{それ以外のとき}) \end{cases}$$

(ただし，$Q(m_1, m_2)$ は $m_1$ から $m_2$ までの全経路の集合とする．)

影響度は，そのモジュールを理解・テストしなければならない割合の期待値を表す．影響度が 1 の場合は，保守作業でそのモジュールを完全に理解し，テストしなければならない．

また，変更要求 $R$ が与えられたとき，頂点 $m$ が受ける影響度 $\mathrm{imp}(R, m)$ を以下のように定める．

$$\mathrm{imp}(R, m) = 1 - \prod_{r \in R} \{1 - \mathrm{imp}_m(r, m)\}$$

$\{1 - \mathrm{imp}_m(r, m)\}$ は，モジュール $r$ を変更するとき，モジュール $m$ のうち理解・テストしなくてよい割合を表す．上記の式は，多くのモジュールから影響を受けるほど，理解・テストしなければならない割合が増えることを表している．

今，$G_P = (M, I)$, $M = \{m_1, m_2, \ldots, m_n\}$ に対し，保守ポイント $C(G_P, R)$ は次の式で与えられる．

$$C(G_P, R) = \sum_{i=1}^{n} \{\mathrm{eff}(m_i)\mathrm{imp}(R, m_i)\}$$

この保守ポイントは， $R$ に対して，各モジュールの変更に必要な労力の総和を示していると考えられる．

保守ポイントの値は， $G_P$ と $R$ が明らかになった時点，すなわち保守作業によってどのモジュールを変更するのかを決定した時点で計測可能となる．このため， $R$ の決定が比較的容易である修正保守や適応保守，完全化保守の見積りに用いることができよう．

保守ポイントの値は保守作業による影響範囲内の労力を反映した値であるため，実際に見積りを行う際には，保守作業者の生産性や設計文書の整備状態などを考慮する必要がある．また，保守作業によってモジュールと影響波及関係が追加されうるが，これにかかる労力は保守ポイントに反映されていない．このため，労力見積りの際には，新規モジュールの開発にかかる労力などを考慮しなければならない可能性がある．

実際のプロダクトとプロダクトモデルを対応づけるには，頂点（モジュール）と辺（影響波及関係）に何を用いるかを決定する必要がある．本研究の評価実験 (2.3 節) では比較的簡単に求められるものを頂点と辺にしたが， 2.5.2 節で述べるように，他の選択も多数ある．

### 2.2.3 計算例

図 2.1 のプロダクトモデル上での，保守ポイントの計算例を示す．

変更要求 $R = \{\mathrm{m1}, \mathrm{m4}\}$ の場合を考える．モジュール m4 の変更により影響を受けるのは，モジュール m3 とモジュール m2 である．モジュール m4 から m2 への経路は，経路 1: s4 と，経路 2: s3, s2 の 2 つが存在する．経路 1 の影響度は 0.1，経路 2 は $0.6 \times 0.5 = 0.3$ で，経路 2 の方が大きい．同様に，モジュール m1 から影響を受けるのは m2 であり，経路 s1 の影響度は 0.5 である．

よって，この場合の保守ポイントは，以下の計算で与えられる．

$$
\begin{aligned}
& C(G_P, \{\mathrm{m1}, \mathrm{m4}\}) \\
= \; & \mathrm{eff}(\mathrm{m1})\mathrm{imp}(\{\mathrm{m4}\}, \mathrm{m1}) \\
& + \mathrm{eff}(\mathrm{m2})\mathrm{imp}(\{\mathrm{m4}\}, \mathrm{m2}) \\
& + \mathrm{eff}(\mathrm{m3})\mathrm{imp}(\{\mathrm{m4}\}, \mathrm{m3}) \\
& + \mathrm{eff}(\mathrm{m4})\mathrm{imp}(\{\mathrm{m4}\}, \mathrm{m4}) \\
= \; & 10 \times \{1 - (1 - 0)(1 - 1)\} \\
& + 4 \times \{1 - (1 - 0.5)(1 - 0.3)\} \\
& + 5 \times \{1 - (1 - 0)(1 - 0.6)\} \\
& + 4 \times \{1 - (1 - 0)(1 - 1)\} \\
= \; & 19.6
\end{aligned}
$$

## 2.3 評価実験

保守ポイントと保守作業の労力の相関を調べるために実験を行った．実験では被験者があるソフトウェアに対して実際に保守作業を行い，その作業に要した労力と保守ポイントとの関係を分析した．本実験で実施した保守作業は，欠陥に対する修正保守と機能追加に係わる完全化保守である．

被験者は，大阪大学大学院情報科学研究科の博士前期課程に所属する 6 人である．保守の対象となるソフトウェアや，作業内容は以下のようになっている．

### 保守対象のソフトウェア

保守作業の対象として，酒屋問題 [50] の仕様に基いたソフトウェアを用いた．これは，酒屋の倉庫への商品の入庫や出庫を管理することを目的としたソフトウェアであり，コマンドラインインタフェースを持つ．実装には Java を用い， 8 クラス，25 メソッド， 309 行の規模である．

本実験の保守作業は修正保守を含んでいるため，保守作業で取り除くための潜在的な欠陥を 2 個埋め込んでおいた．

### 課題

被験者に与えた課題は，ソフトウェアに埋め込まれた欠陥修正の作業 D1，D2 の 2 つと， 機能追加を行う作業 E1 〜 E4 の 4 つの，計 6 つである．

このうち，課題 E1 は保守対象のソフトウェアに慣れさせるための練習として用いる．そのため，評価に用いる課題は D1, D2, E2, E3, E4 の 5 つである．

課題の概要は以下のようなものである．

- D1: ライブラリ使用法の誤りの修正
- D2: 関連するモジュールが仮定している条件が矛盾している誤りの修正
- E1: 酒屋の倉庫の中を表示するコマンドの追加
- E2: 倉庫を表すデータ構造の変更
- E3: 処理順序の変更
- E4: 既存のコマンドの拡張

被験者に対する課題の作業内容の指示は以下のように行った．欠陥修正の課題では，保守対象のソフトウェアへの入力と異常な出力を被験者に示し，正しい出力をするようにソフトウェアを変更するよう指示した．機能追加の課題では，新しい機能の要求仕様として，自然文による機能の説明と，その機能を実装したときの入出力の例とを与えた．

欠陥修正と機能追加のどちらの作業でも，どのモジュールを変更するかは指示せず，被験者の判断に委ねた．保守ポイントの計算モデルでは変更要求は外から与えられることとなっているが，本実験は変更するモジュールと労力との関連を調べることが目的であるため，変更するモジュールを指示しないことは問題とはならない．

## 作業順序

被験者の作業順序を図 2.3 に示す．

まず最初に，保守対象のソフトウェアに慣れるために，被験者全員が機能追加課題 E1 を行った．この作業に要した時間は実験の評価に用いない．

次に 2 件の欠陥修正の課題を行った．慣れの影響を除くため，半分の被験者は D1 を先に行い，残りの半分の被験者は D2 を先に行った．

最後に，E2 〜 E4 の 3 件の機能追加の課題を行った．欠陥修正と同様に，慣れの影響を除くため，被験者毎にすべて異なる順序で作業を行った．

課題の作業対象は，直前の課題の作業結果のソースコードである．

### メトリクス値の計測

プロダクトモデルとの対応づけは以下のように行った.

- 頂点（モジュール）: メソッド
- 頂点の労力: LOC と McCabe のサイクロマチック数の 2 通り
- 辺（影響波及関係）: メソッド間の呼出しと，フィールド変数を介したデータ依存関係
- 辺の重み: 各辺同じ 0 から 1 まで 0.05 刻みの 21 通りの値

労力の見積りに適した辺の重みの設定を見つけるために，21 通りの重みでメトリクス値を計算することとした.

図 2.2 に，保守対象ソフトウェアの課題 E1 を行う前の状態から作成したプロダクトモデルを示す．頂点の労力としては， LOC と McCabe のサイクロマチック数の 2 通りを計測するので，2 つを併記した．有向辺の重みは 21 通りの場合を用いるので，表記していない.

変更要求 $R$ は，被験者が課題の内容を理解した後に直接変更する必要があると判断したメソッドの集合を用いた.

以下，サイクロマチック数を用いた保守ポイントを，**保守ポイント** [**McCabe**] と呼び， LOC を用いた保守ポイントを，**保守ポイント** [**LOC**] と呼ぶことにする.

### 作業時間の計測

被験者が作業に要した時間として，被験者に課題を渡してから，プログラムが正しい入出力を行うことを確認するまでの時間を用いた.

## 2.3.1 実験結果

各被験者が 5 つの作業を終えるのに要した時間は，最短で 1 時間 43 分，最長で 5 時間 2 分となり，被験者毎に大きな差があった．このため，作業時間を正規化した値を労力を表す値として用いた．この値を正規化労力と呼ぶ．正規化労力は，各作業に要した時間を 5 つの作業全体に要した時間で割った値である.

また，30 件の作業のうち 9 件において，平均 24.2 行 (標準偏差 23.8) の新しいモジュールの導入がなされていた.

### 労力と保守ポイントの相関

保守ポイントの値と正規化労力との相関と，辺の重みとの関係を図 2.4 に示す．すべての辺の重さの場合において，保守ポイント [McCabe] は保守ポイント [LOC] よりも高い相関を示した．保守ポイント [McCabe] と保守ポイント [LOC] の両方で，辺の重みが 0.3 付近で相関が最大となり，辺の重みが 0.7 以上である場合の相関は，辺の重み 0 のときの相関よりも小さくなった．辺の重みを 1 としたときの相関はほぼ 0 であった．

相関が最大となる辺の重みが 0.3 の場合の，保守ポイント [McCabe] と正規化労力の散布図を 図 2.5 に示す．■は新規モジュールが追加されていた場合，△は新規モジュールの追加がなかった場合のデータである．新規モジュールが追加されるのは，正規化労力と保守ポイントの両方が大きい場合であることが分かる．直線は最小二乗法によって求めた一次関数による近似である．

### 既存のメトリクスとの比較

次に，保守ポイントを既存のメトリクスと比較する．比較に用いる保守ポイントの値は，すべて辺の重みが 0.3 のときのものである．

一つ目の比較対象として，保守により変更されるメソッドの集合 $R$ の要素の複雑度を，単純に足し合わせたものを用いた．複雑度には，保守ポイントと同じく，McCabe のサイクロマチック数と LOC の 2 種類を用いた．以下，サイクロマチック数を足し合わせたメトリクスを，単純和 [McCabe] と呼び， LOC を足し合わせたメトリクスを，単純和 [LOC] と呼ぶことにする．単純和は，すべての辺の重みを 0 としたときの保守ポイントと同じである．

二つ目の比較対象として，Jørgensen の提案した見積り手法 [26] で用いられている，作業によって書換えられた行数（変更行数）を用いた．変更行数は労力との相関が高いことが知られているが，保守作業の早い段階で計測することができないため，見積りに用いるのは困難である．

表 2.1 に取得したメトリクスと，正規化労力との相関を示す．正規化労力との相関が最も高かったのは変更行数であり，次に正規化労力との相関が高かったのは保守ポイント [McCabe] であった．また，モジュールの複雑度に用いたメトリクスが同じ場合，単純和よりも保守ポイントの方が，高い相関を示した．

次に，保守ポイントの正規化労力見積りの誤差が，単純和に比べて有意に小さいかを調査した．調査は，保守ポイント [McCabe] と単純和 [McCabe] の組 $T_M$ と，保守ポイント [LOC] と単純和 [LOC] の組 $T_L$ について行った．

まず，それぞれのメトリクスについて，一次式による労力予測式を最小二乗法

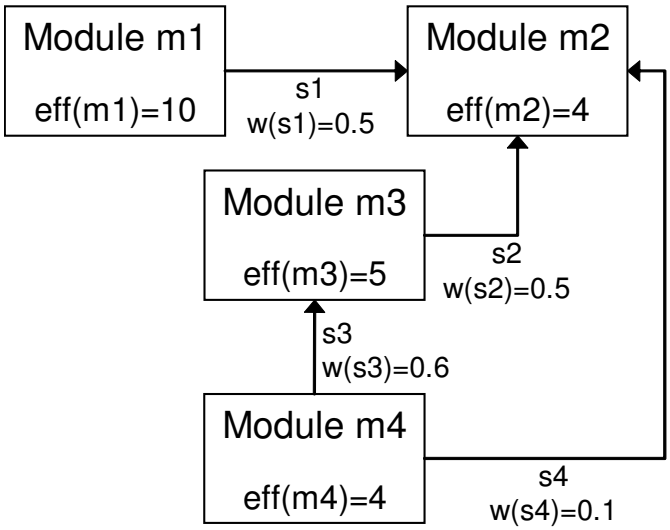


図 2.1: プロダクトモデルの例

表 2.1: 正規化労力と各メトリクスの相関

| | 保守ポイント [McCabe] | 単純和 [McCabe] | 保守ポイント [LOC] | 単純和 [LOC] | 変更行数 |
|---|---|---|---|---|---|
| 正規化労力相関との相関 (p 値) | 0.82 (<0.01) | 0.73 (<0.01) | 0.77 (<0.01) | 0.65 (<0.01) | 0.87 (<0.01) |
| 符号付順位和検定 二乗誤差 (p 値) | 101(<0.01) | | 95 (<0.01) | | |
| MRE (p 値) | 105(<0.01) | | 86 (<0.01) | | |

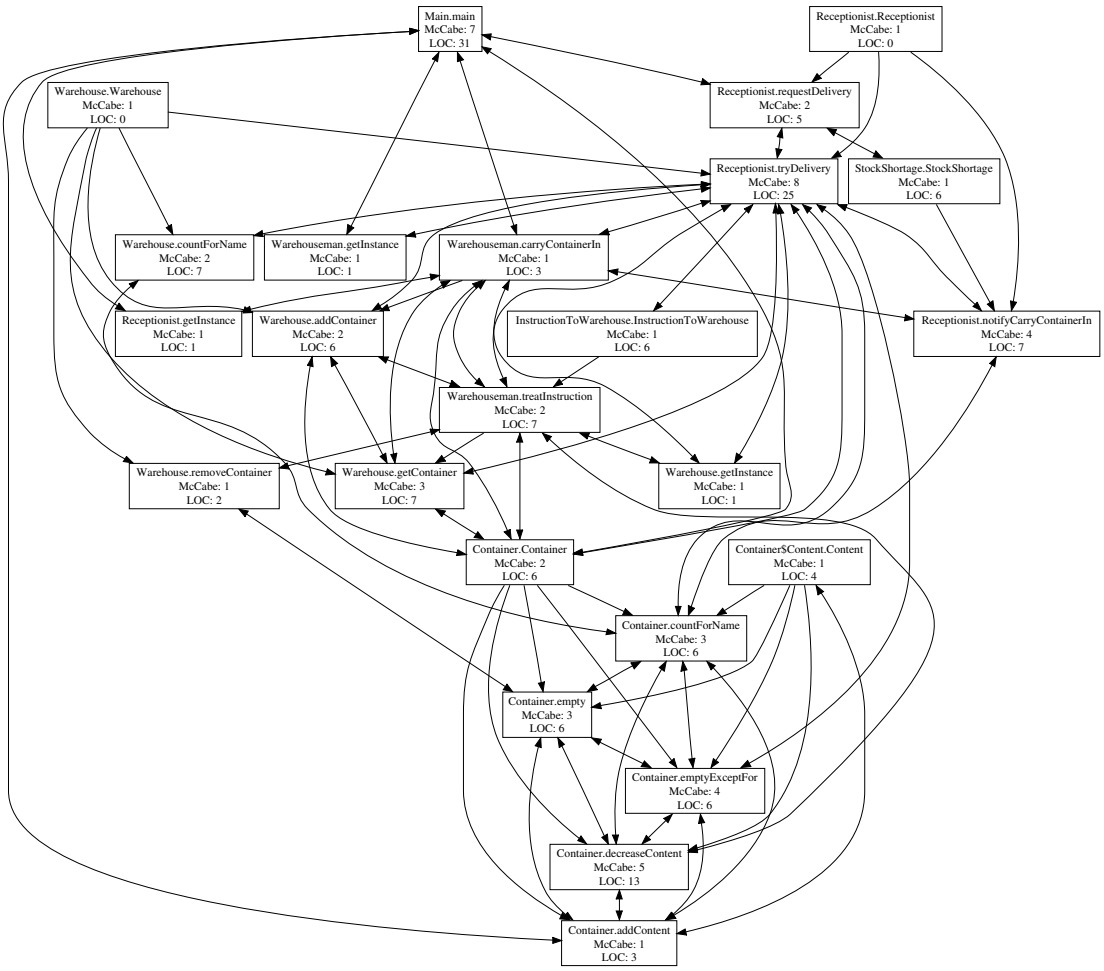


図 2.2: 作業対象のプロダクトモデル

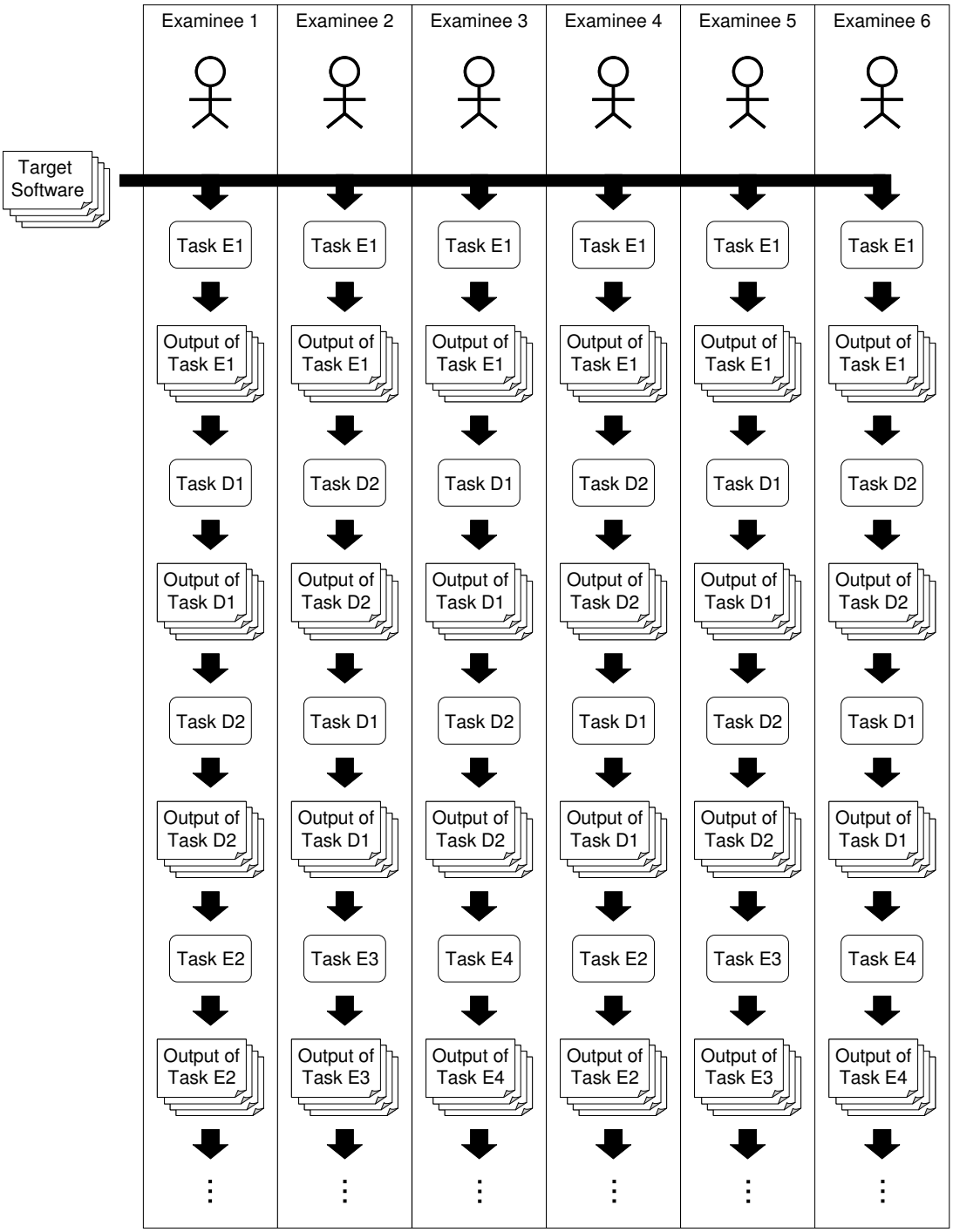
Examinee 1
Examinee 2
Examinee 3
Examinee 4
Examinee 5
Examinee 6
Target Software
Task E1
Output of Task E1
Task D1
Task D2
Output of Task D1
Output of Task D2
Task D2
Task D1
Output of Task D2
Output of Task D1
Task E2
Task E3
Task E4
Output of Task E2
Output of Task E3
Output of Task E4

図 2.3: 実験の作業順序

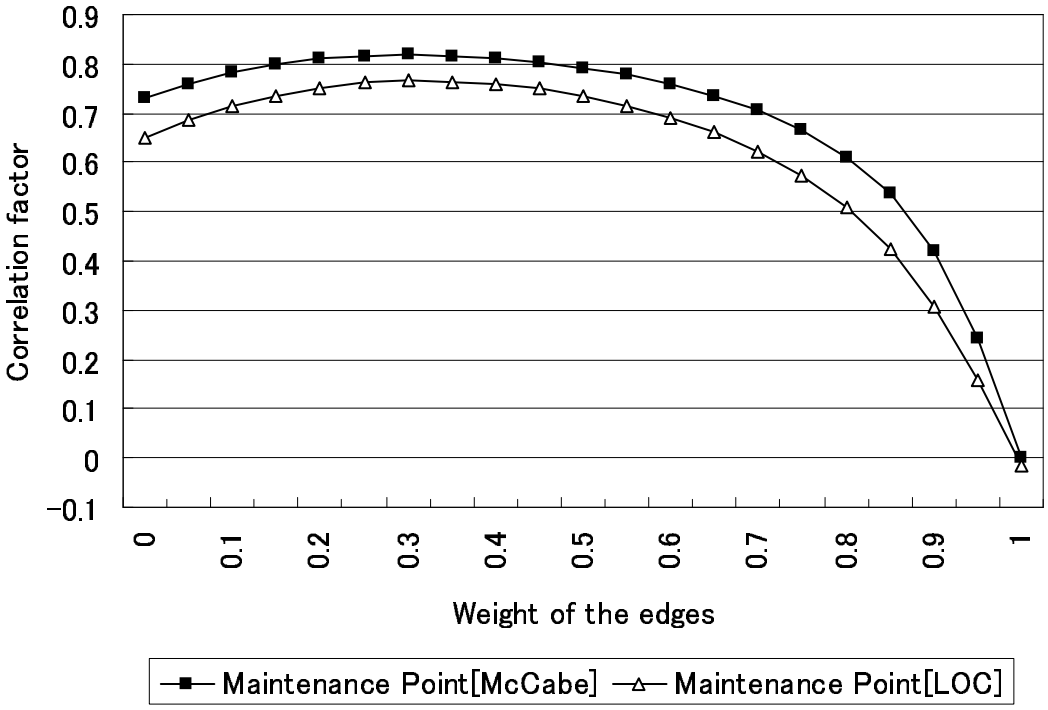


図 2.4: 辺の重みを変化させたときの保守ポイントと正規化労力との相関

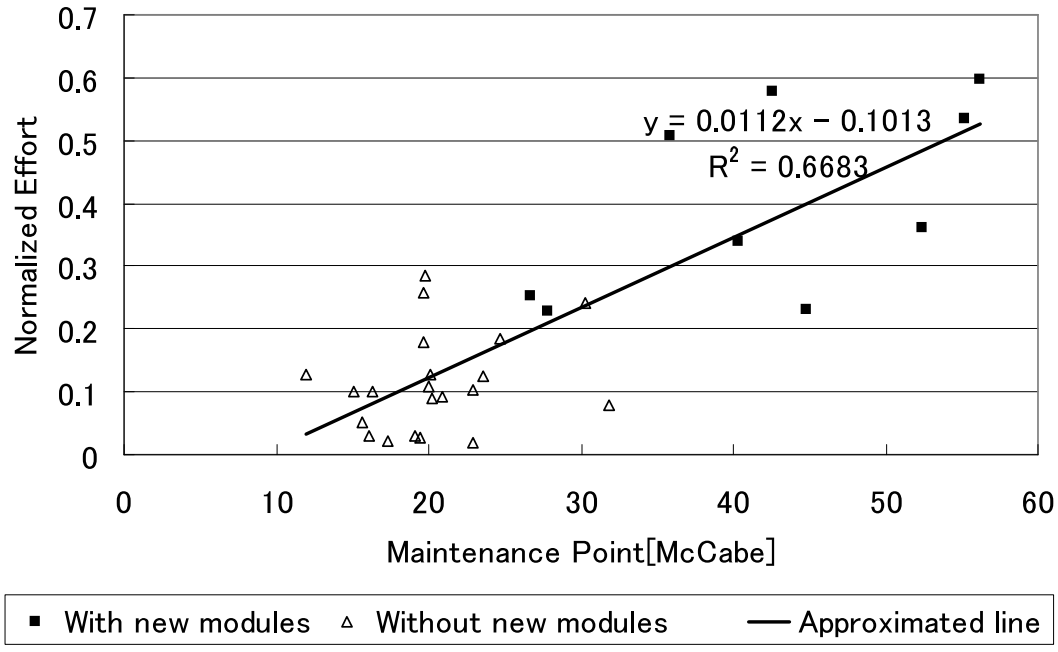


図 2.5: 辺の重みを 0.3 とした場合の保守ポイント [McCabe] と正規化労力の散布図

を用いて作成した．そして，それぞれの組について，予測式による二乗誤差と，メトリクスの評価で用いられる相対誤差 ($= \left|\frac{\text{見積り値}-\text{実測値}}{\text{実測値}}\right|$．以下 MRE と表記する) の値に有意な差があるかを調べた．二乗誤差や相対誤差の値は正規分布とならないため，平均値の検定ではなく，ウィルコクソンの符号付順位和検定を用いて片側検定を行った．

その結果，表 2.1 に示したように，$T_M$ と $T_L$ 両方の組に対して，二乗誤差と MRE のどちらでも有意水準 1% で，保守ポイントの誤差が小さいと言えることが分かった．

さらに，計測した 30 件のデータをランダムに 10 件ずつの 3 群に分割し，クロスバリデーションを行った．見積り式には一次式を用い，最小二乗法でパラメータを決定した．評価項目としては，メトリクスの評価として頻繁に用いられる以下の 4 つを用いた．

- **MAE**: 絶対誤差の平均
- **MMRE**: MRE (相対誤差) の平均
- **PRED(25)** 相対誤差が 25% 以下である割合
- **PRED(50)** 相対誤差が 50% 以下である割合

MAE と MMRE の値は小さいほど，PRED(25) と PRED(50) は大きいほど見積りが正確であることを示す．

クロスバリデーションの結果を 表 2.2 に示す．表の各行のうち，太字で書かれた列が最良の結果であったことを表す．MMRE では変更行数が最良であったが，次に良いのは保守ポイント [McCabe] であった．また， PRED(25) では保守ポイント [McCabe] と変更行数が，その他の評価項目では保守ポイント [McCabe] が最良となり，保守ポイント [McCabe] の見積り精度が高いことが分かった．

### 新規モジュールが正規化労力に与える影響

保守ポイントは新規に追加されるモジュールの量を考慮していない．そこで，新規モジュールを考慮することで，正規化労力の見積り精度がどの程度改善するかを調査した．

まず，辺の重みを 0.3 とした保守ポイント [McCabe] の影響を除いた，新規モジュールの行数と正規化労力との偏相関係数は 0.4903 であった．

2.3.1 節で行った保守ポイント [McCabe] による見積りの残差と新規モジュールの行数との散布図を，図 2.6 に，見積りの残差と新規モジュール数との散布図を， 図 2.6 に示す．一次式による近似では決定係数はそれぞれ，0.1397，0.1272 となった．

表 2.2: クロスバリデーションの結果

| | 保守ポイント [McCabe] | 単純和 [McCabe] | 保守ポイント [LOC] | 単純和 [LOC] | 変更行数 |
|---|---|---|---|---|---|
| MAE | **0.07840** | 0.09270 | 0.09405 | 0.1177 | 0.07868 |
| MMRE | 0.9616 | 1.427 | 1.263 | 1.766 | **0.8266** |
| PRED(25) | **0.3667** | 0.3000 | 0.3333 | 0.1667 | **0.3667** |
| PRED(50) | **0.6667** | 0.6000 | 0.6000 | 0.5000 | 0.6000 |

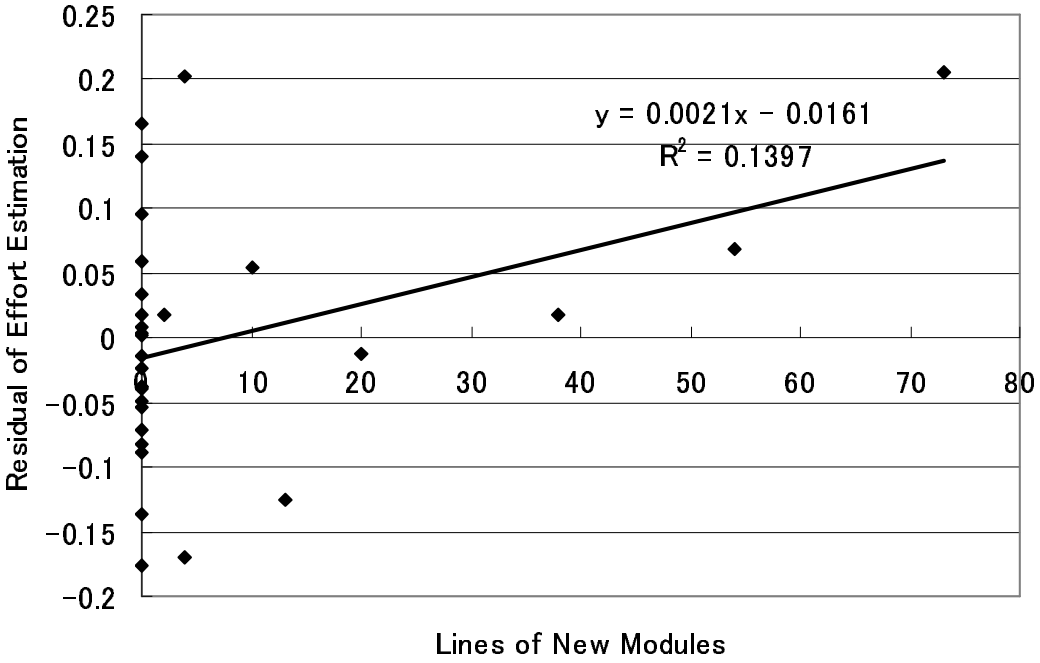


図 2.6: 保守ポイント [McCabe] による見積り残差と新モジュールの行数の散布図

以上のことから，今回の実験では新規モジュールの量が正規化労力に与える影響は大きくなかったと言える．

### 2.3.2 考察

実験結果が示すように，保守ポイントは既存のメトリクスである単純和よりも高い労力との相関を持っており，より高い精度で労力を予測することができると分かった．

Sneed の提案した保守コスト見積り手法 [46] は，保守工数見積りの主な基準として影響範囲の大きさを用いる．影響範囲の大きさは辺の重みを 1 としたときの保守ポイント [LOC] の値と等しいが，2.3.1 で示した通り，辺の重みが 1 のときの保守ポイントと労力との相関はほぼ 0 である．

## 2.4 関連研究

Leitch ら [33] は，リファクタリングによる依存関係の変化を，変更前後の依存関係と COCOMO II による作業量見積りを用いて評価する方法を提案した．

また，上述のように， Sneed [46] は，保守による影響範囲の大きさを様々な要因で補正して，工数を見積る手法を提案した．我々の手法は，影響範囲の大きさではなく，複雑さの和を用いる点が異なる．

Ko ら [30] は，保守に特化した統合開発環境を作るために，保守作業を観察した．その結果，保守作業では，最初に小さな作業範囲を特定し，その範囲内で作業を行っていることがわかった．我々のメトリクスは，機械的に計算可能な影響範囲を用いて作業範囲を近似することで，保守作業の工数を見積もっていると考えられる．

保守作業に関係する新しいメトリクスが，いくつか提案されている．Lindvall ら [34] は，モジュール結合度に基づいて，設計と実装との乖離を計測するメトリクスを提案し，実際の保守作業の前後に計測した．Chen ら [12] は，オブジェクト指向ソフトウェアの凝集度を，属性とメソッドの間の依存関係を用いて計測する方法を提案した．Tran-Cao ら [48] は，データ操作やアルゴリズムの複雑さを考慮した機能量を計算する方法を提案した．Arisholm ら [6] は，オブジェクト指向ソフトウェアの結合度を，実行時情報を用いて計測する方法を提案した．これらのメトリクスは，モジュール内部の性質や，直接利用しているモジュールとの関係を表すものである．我々の手法は影響範囲を推移的に辿る点で，実際の保守作業をより反映していると考えられる．

Bianchi ら [7] は，ソフトウェア開発の設計・実装など各段階を構成するコンポーネントの段階間の依存関係を用いて，ソフトウェアの劣化具合を測定するメトリクスを提案した．我々の手法とは，個々のモジュールの複雑さを考慮しない点や，値の利用方法が大きく異なる．

## 2.5 議論

### 2.5.1 モデルの妥当性

保守ポイントは，プロダクトモデルと変更要求を基に計算する．一般に，ソフトウェア保守では理解とテストの労力が大きいと言われており [13][15][19][38]，その工数が影響範囲に強い影響を受けることがわかっている [10][30]．そのため，保守作業の労力は影響範囲内の複雑さから大きな影響を受けると考えられる．

保守ポイントは，保守作業によって変更されるモジュールから計算されるので，保守作業によって新規に追加されるモジュールを考慮していない．そのため，新規モジュールの開発に大きな労力が割かれる保守案件の工数を保守ポイントだけで見積るのは適当でない．このような場合は，新規モジュールの開発に要する労力を機能量などで見積り，保守ポイントを用いて見積った既存モジュールの変更に要する労力を足し合わせることで，全体の労力を求めるのが適当であると考えられる．

しかし，2.3.1 節で行なった実験の評価では，新規モジュールのサイズを考慮しても見積り精度はあまり改善しないという結果になった．これは，新規モジュールを開発するためには，新規モジュールを利用する既存のモジュールを理解し，テストする必要があり，その理解およびテストの労力が新規モジュールの開発にかかる労力の大部分を占めるためであると考えられる．

### 2.5.2 プロダクトモデルの対応づけ

プロダクトとプロダクトモデルの対応づけには，2.3 節で示したもの以外にも，多くの方法が考えられる．

モジュールの候補としては，プログラミング言語により様々なものが考えられる．例えば Java の場合，メソッド，クラス，パッケージといった階層があり，モジュールの候補となりうる．モジュールの保守作業の労力には，複雑度メトリクスを用いる．

影響波及関係の計算方法として，いろいろなものが考えられる．例えば，モジュール $m_1$ から $m_2$ への呼び出しの数を $k$，外にデータフローを起こす変数の数を $l$

としたとき，$1-\alpha^k\beta^l$（$\alpha$ と $\beta$ は 0 より大きく 1 より小さい定数）を重みとする．この式は，変数 $k$ および $l$ の定義域を 0 以上の整数とした場合，値域は 0 以上 1 未満となり，$k$ と $l$ それぞれについて単調増加関数となる．$k$ と $l$ が無限に増加してゆけば，この式の値は限りなく 1 に近づく．$\alpha, \beta$ の値は，過去に計測したメトリクス値と労力との相関が最大となるように決定する．

また，CVS リポジトリ等から細粒度の変更履歴が入手可能な場合には，同時更新傾向に基づいた影響波及関係の推測手法である Logical Coupling [18] を影響波及関係として用いることもできる．

モジュールとその労力，影響波及関係の選択は，対象とするプロダクトの規模やドメインを主な基準として，メトリクスの計測者が行う．このとき，過去の類似プロダクトと同じ条件で計測することで，回帰分析などを用いた労力見積りが可能となる．

### 2.5.3 実験の設定

我々の行った実験は，実際のソフトウェア開発現場に比べて以下のような違いがある．

1. 被験者が少ない
2. ソフトウェアの規模が小さい
3. 被験者の能力が，実務者に比べて低い可能性がある
4. 被験者の能力にばらつきがある

(1) と (2) については，今後，より大規模な実験を行う必要があるだろう．

(3) については，実験問題は特殊な実務経験を必要としない一般的な問題であり，被験者は情報科学科の課程を修めているため，問題に対して十分な能力を有していると思われる．

(4) の能力のばらつきは，実際のソフトウェア開発現場でも存在するものである．また，労力を計算する際に個人の作業能力で正規化しているため，実験結果からは排除されている．

### 2.5.4 具体的な変更要求が得られないときのメトリクス

保守ポイントを求めるためには，プロダクトモデル $G_P$ と変更要求 $R$ が必要である．しかし，ソフトウェア保守を包括的に請負うときなどのように，どのよう

な作業が発生するかが分からないときには，具体的な $R$ を求めるのは非常に困難である．

そこで， $R$ の代わりに保守要求 $R$ の確率分布 $R_d$ を用いた，以下のような労力指標モデル $C_{\mathrm{gen}}(G_P, R_d)$ を考える．

$$C_{\mathrm{gen}}(G_P, R_d) = \sum_{i=1}^{n} \{q_i C(G_P, \{m_i\})\}$$

(ただし， $R_d = (q_1, q_2, \ldots, q_n)$（n は $G_P$ の頂点数），$0 \leq q_i \leq 1$ ， $q_1 + q_2 + q_3 + \cdots + q_n = 1$ である． $q_i$ は一つの保守作業においてモジュール $m_i$ が変更される確率を表す．)

$R_d$ の要素を均等に $1/n$ とすれば，簡単に試算することができる．このモデルを一般化保守ポイント $C_{\mathrm{ave}}(G_P)$ と呼ぶ．

$$\begin{aligned} C_{\mathrm{ave}}(G_P) &= C_{\mathrm{gen}}(G_P, (1/n, \ldots, 1/n)) \\ &= \frac{\sum_{i=1}^{n} C(G_P, \{m_i\})}{n} \end{aligned}$$

より正確に $R_d$ を計算する方法として， $q_i$ の値を $C(G_P, \{m_i\})$ から求める方法も提案されている [31].

これらの提案するメトリクスの妥当性は，今後，長期にわたる実験を行い検証する予定である．

## 2.6 まとめと今後の課題

この論文では，保守作業の労力見積りに用いるメトリクスを定義するために，プロダクトを有向辺と頂点から成るグラフとしてモデル化した．このモデル上で，保守作業の労力見積りに用いるメトリクスである保守ポイントを定義した．さらに，保守ポイントと労力との関係を調べる実験を行った結果，保守ポイントは，既存のメトリクスに比べて，より正確な労力を見積るうえで有用であることを確認した．

今後の課題としては，より大規模な実験を行ってメトリクスを評価すること，長期的な実験を行って一般化保守ポイントを評価すること，実際のソフトウェア保守現場での使用が挙げられる．

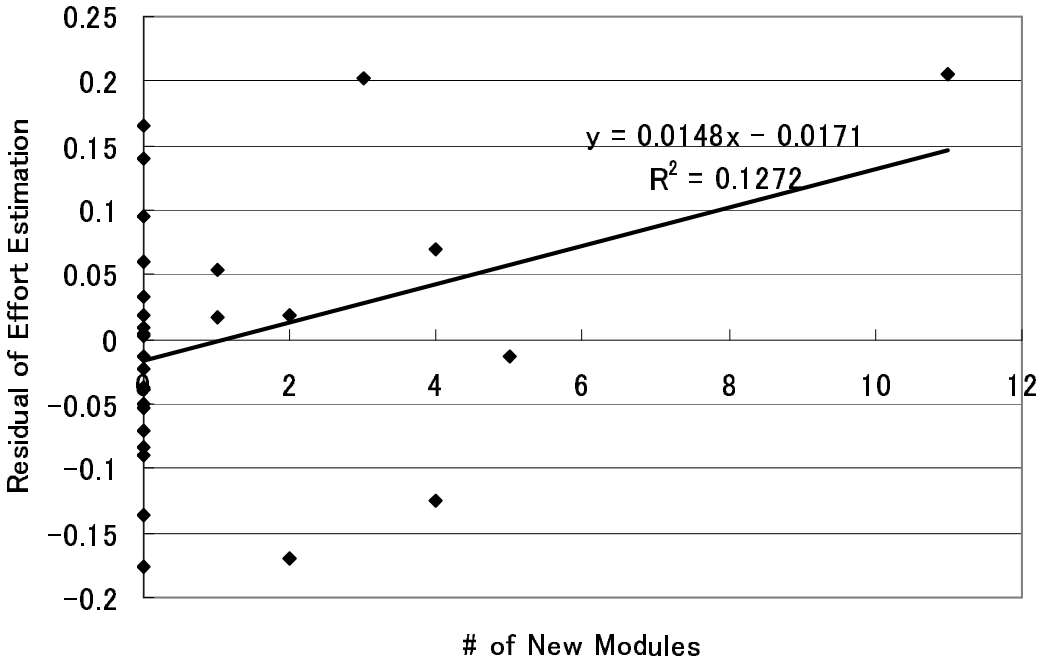


図 2.7: 保守ポイント [McCabe] による見積り残差と新モジュール数の散布図

# 第3章　構文木の差分を利用した版管理システム向きマージ手法

## 3.1　導入

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 [23] 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 [16].

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．この時，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] 等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，これらの変更の組み合わ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時には，マージできないことを検出できず，間違った出力を行う．

**マージ出来るべき変更をマージ出来ない問題** 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．これらの変更の組み合わ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 [21] ．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上で，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* [11] 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出来る．

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及び，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， 3.2 節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． 3.3 節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， 3.4 節ではそのシステムの実装について説明する． 3.5 節では作成したシステムの適用実験と考察を行う． 3.6 節では関連研究について説明する． 最後に 3.7 節で本研究のまとめと今後の課題について述べる．

## 3.2 版管理システム

本節ではまず，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．

### 3.2.1 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等のファイルの変更履歴を保存するシステムである． (図 3.1)

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全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．図 3.2 で示すように，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良く，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出来る．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（図 3.3）．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共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 3.4）が，人間の手による解決が必要になる場合もある．

### 3.2.2 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，マージの時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言う問題がある．以下，それぞれの問題と理想的な解決を示す．

#### 不要な衝突

例えば，開発者 A と B が同じファイルを取得して，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．

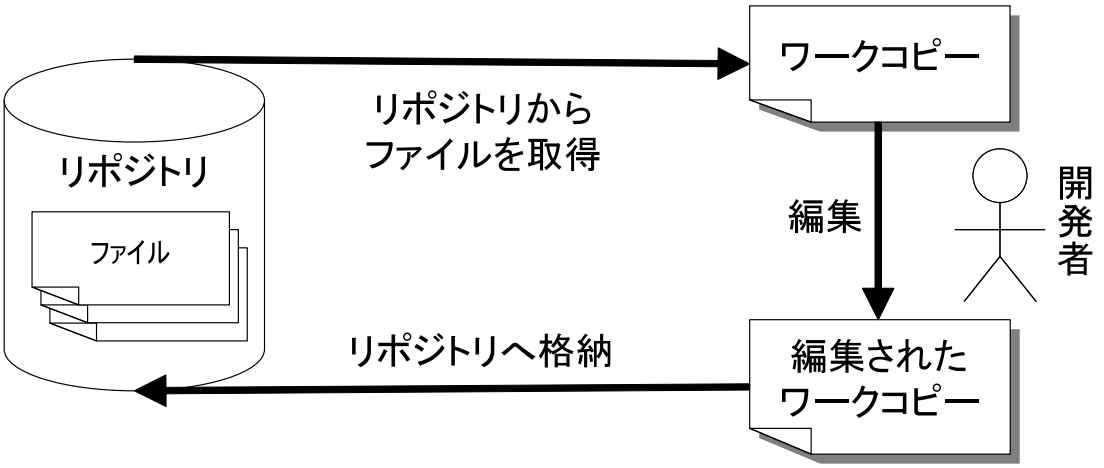


図 3.1: 版管理システム

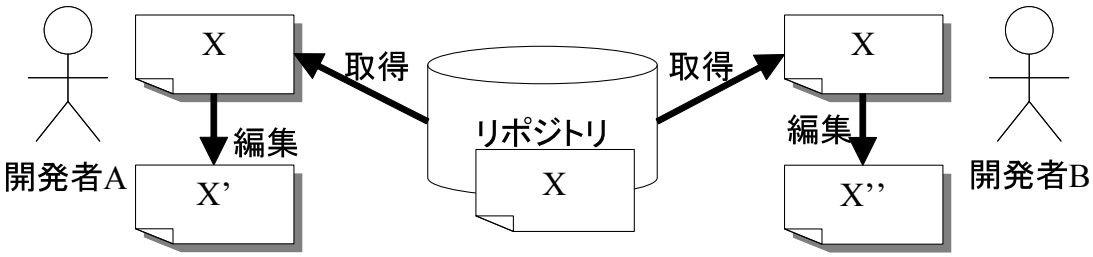


図 3.2: 複数の開発者による同時開発

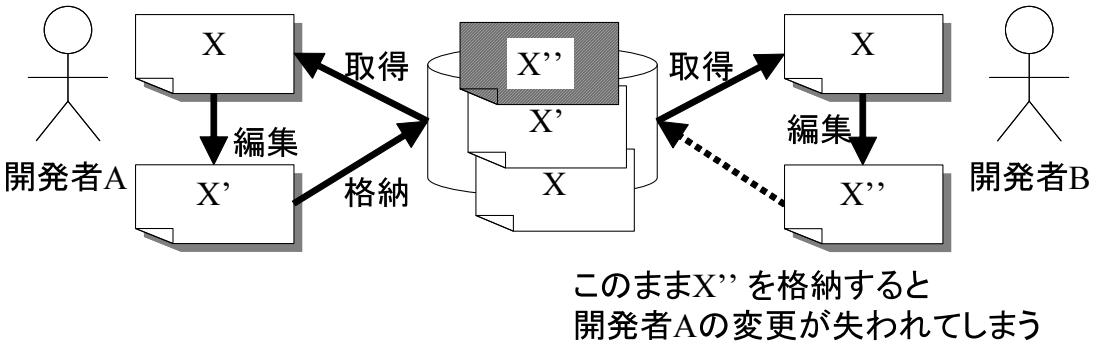


図 3.3: 編集内容の衝突

```
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做す．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 */
```

### 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．この時，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出来るはずである．

## 3.3 ソースコードの構文を考慮したマージアルゴリズム

本節では， 3.2.2 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上でマージを行う．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 3.1 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 3.5)．

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

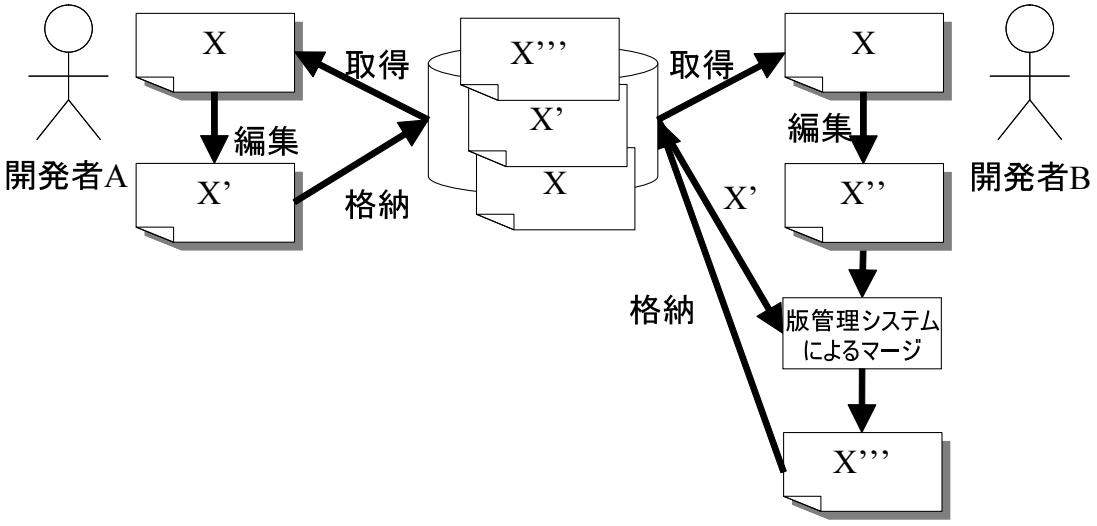


図 3.4: マージした結果を格納

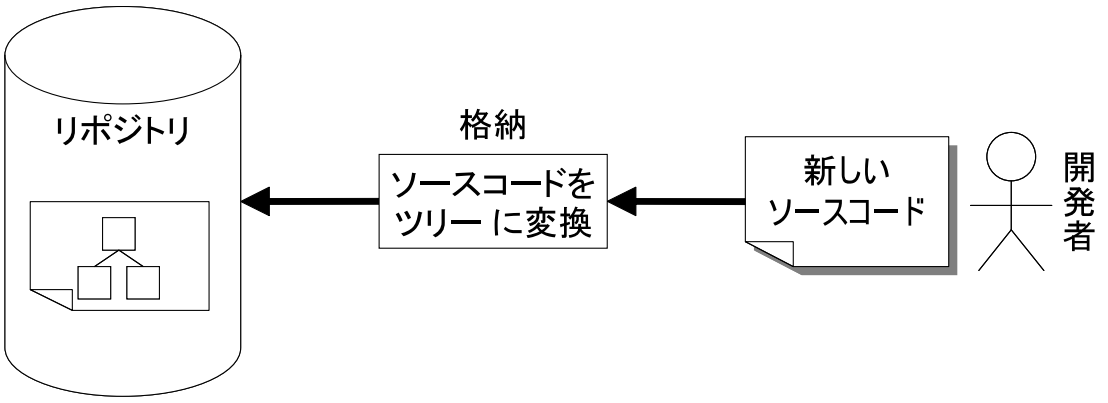


図 3.5: リポジトリに新しいソースコードを追加する時のデータの流れ

3.6 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．

### 3.3.1 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，以下の通りである．

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

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．

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全て持っているため，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．

#### 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 3.7 に RELAX NG Compact Syntax を用いて示す．全ての要素は，ツリーの中で重複しない値を持つ属性 `id` を持っている．`id` の値には， Universally Unique Identifier (UUID) を用いる．

XML 文書のテキストはツリーの葉頂点を表しており， XML 文書の全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．

### 3.3.2 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，マージをツリーの上で行うために，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時には，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時 (図 3.8) 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* [11] 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．

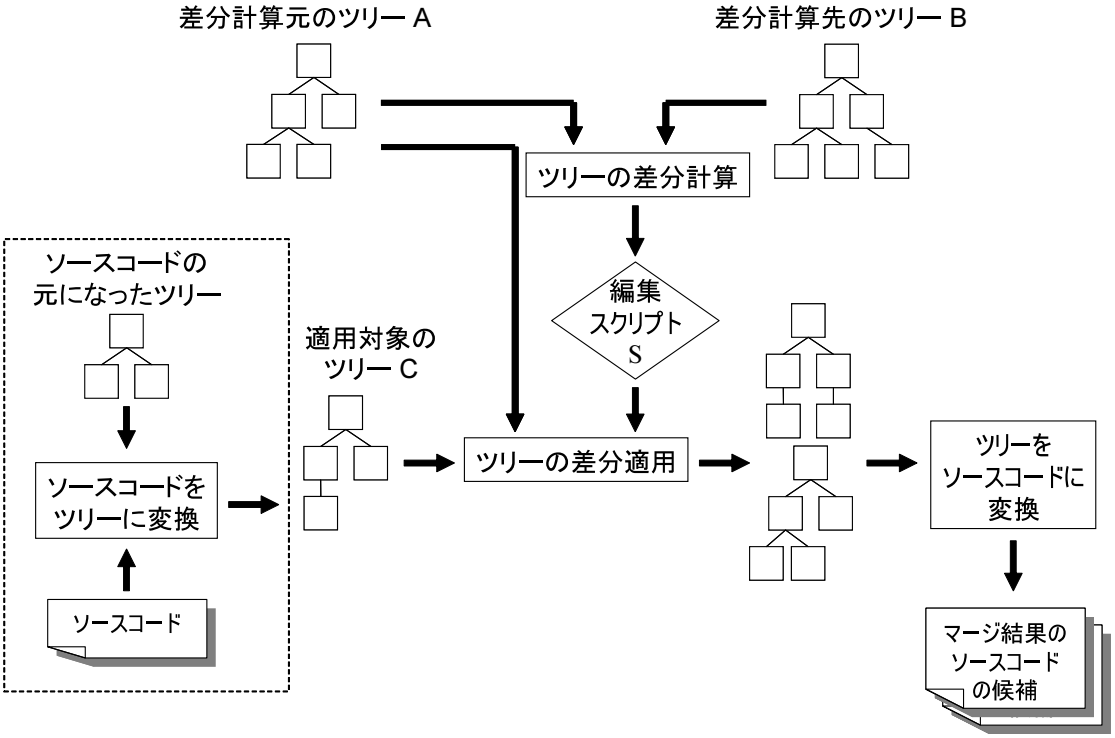


図 3.6: マージを行う時のデータの流れ

```
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 3.7: ツリーの表現に用いる XML 文書のスキーマ

以下，リネーム解析について説明する．

#### リネーム解析

*FastMatch* は，テキストの類似度に基いた，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，間違った組み合わ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出来ない．

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．

### 3.3.3 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 つを取る．

2. delete: 頂点の削除を表す．引数に，削除する頂点の ID を取る．

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取る．

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 つを取る．

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．ツリー A に編集スクリプト S を適用するとツリー B が得られる時，$A \xrightarrow{S} B$ と表す．

任意のツリー A を B に変換する編集スクリプトは， insert 操作と delete 操作の組み合わせで表現可能である．例えば A の頂点を全て delete した後に，B の頂点を insert する編集スクリプトは，A を B に変換する．しかし，人間がソースコードを編集する時には， ソースコードの一部分を移動したり，文字列の一部を書換えるといった操作を行う．これらの操作を自然に表現するために，move 操作や update 操作を用いる．

一般に，あるツリー A と B がある時， A を B に変換するスクリプトは無数に存在する．例えば， $A \xrightarrow{S} B$ である任意の $S$ の末尾に， A にも B にも含まれない頂点を追加する操作と，その頂点を削除する操作を加えた編集スクリプト $S'$ も， $A \xrightarrow{S'} B$ を満たす．ツリーの差分を表すスクリプトとしては，このような無駄な編集操作が含まれているものは望ましくない．

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* [11] を採用した．

### 3.3.4 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 3.6)

3.3.3 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めに，頂点の ID を引数に取る．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．

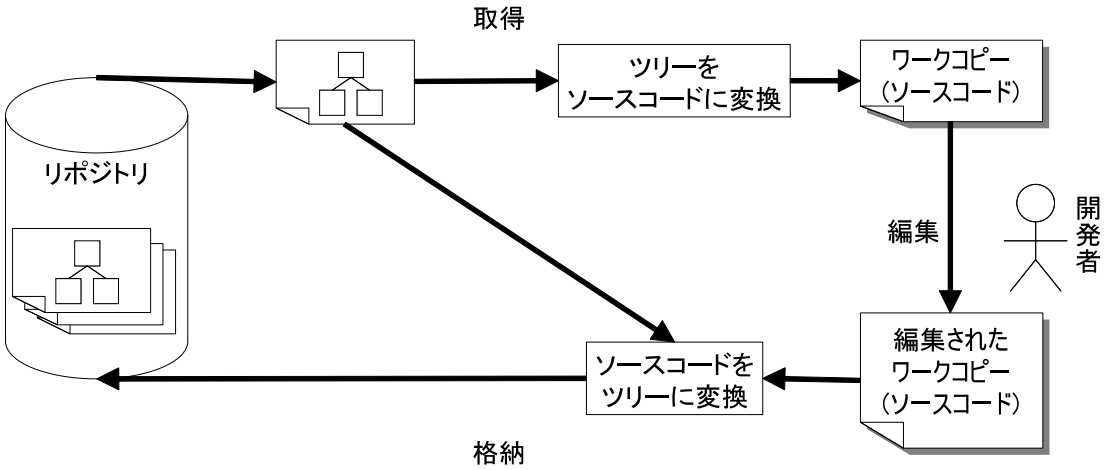


図 3.8: リポジトリへのツリーの格納

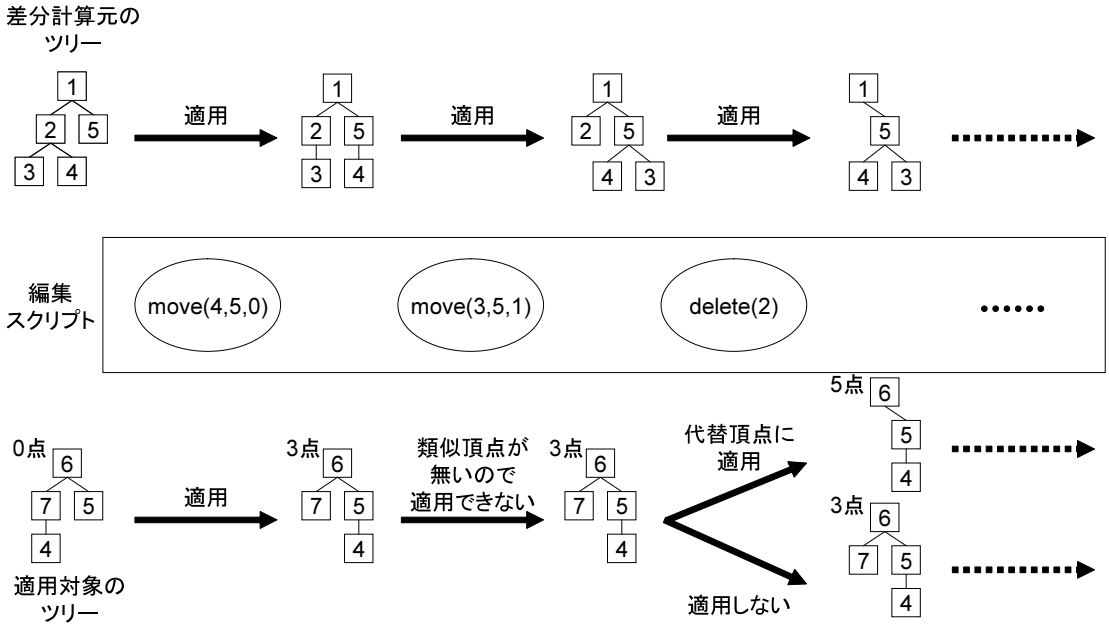


図 3.9: 編集スクリプトの適用

そこで，操作する ID を持った頂点が C に無い場合，操作対象に類似した頂点や位置を探索して代替する (図 3.9).

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時には，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 3.9)．このとき，差分計算元のツリーに対して差分を適用する際には，適用先の頂点や位置が必ず存在するため，類似頂点や類似位置を探索する必要はない．

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取る．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．

#### 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全て用いる．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 3.10）．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つくる．そこから，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無く，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（図 3.11）.

2 つめの方法では，親の頂点から探索を行う．図 3.12 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無く，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．

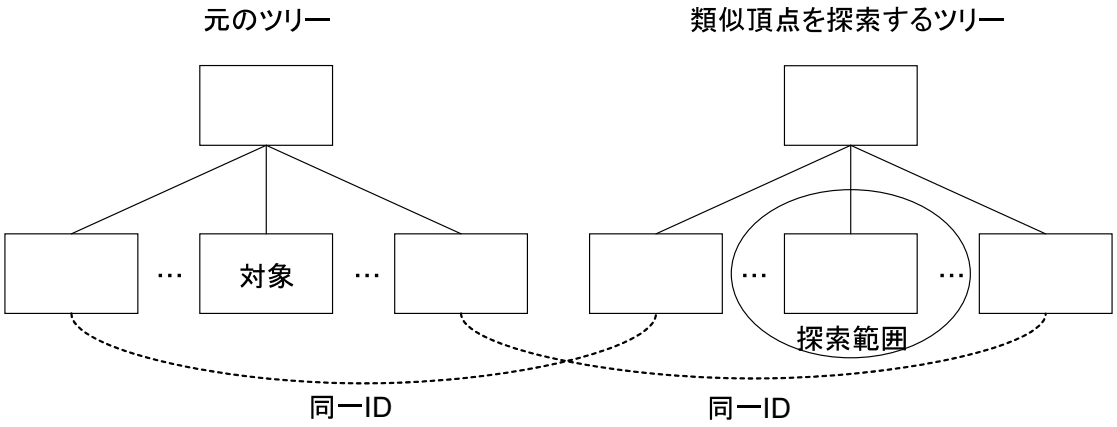


図 3.10: 類似頂点の探索: 兄弟頂点から探索

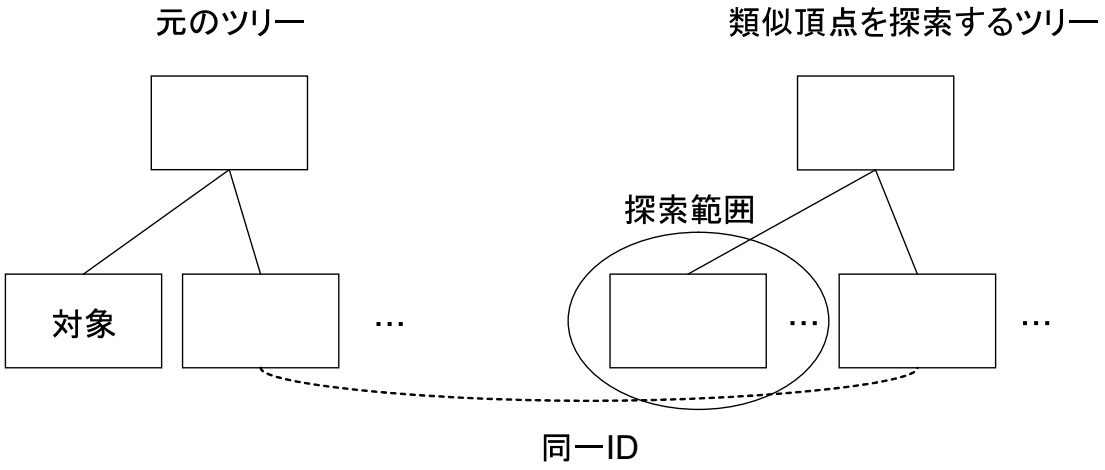


図 3.11: 類似頂点の探索: 兄弟頂点から探索（端にある場合）

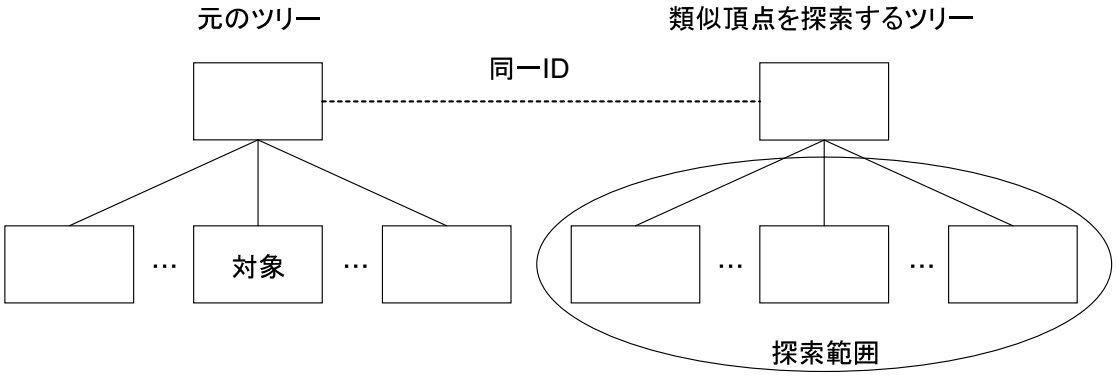


図 3.12: 類似頂点の探索: 親頂点から探索

#### 編集操作の適用

3.3.4 節で説明したように，編集操作の適用をする時には，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時には，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時には，全ての類似頂点と位置の組み合わ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，良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` 操作を適用する時に，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，作成した全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時には，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 3.9)．一つも編集操作を適用していな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，3.3.4 節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．例えば， `move` 操作を適用する時に，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．

### 3.3.5 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．

## 3.4 システムの実装

3.3 節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， 3.3.2 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 3.4.1 節で，3.3.4 節で述べたマージの実装について 3.4.2 節でそれぞれ説明する．

### 3.4.1 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成る．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時や，ワークコピーをリポジトリに格納する時に，改行文字を変換したり，特定のキーワードを置き換える機能がある（図 3.14）．

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 3.14 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．

### 3.4.2 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（図 3.15）．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

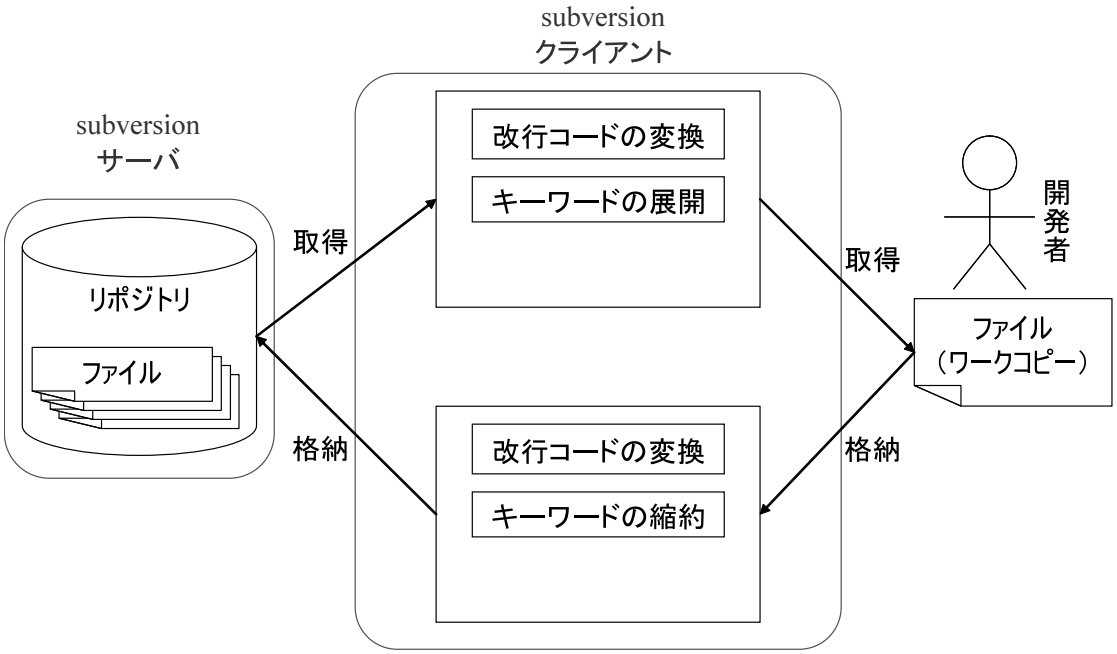


図 3.13: subversion におけるファイルの格納と取得

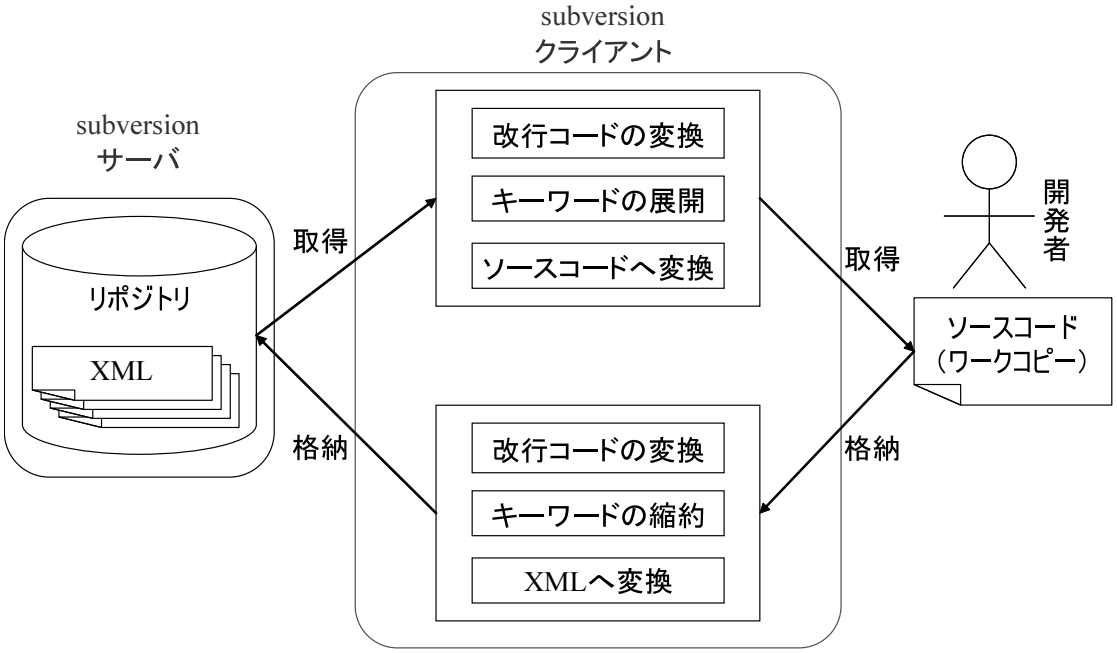


図 3.14: ソースコードの木構造を考慮した格納と取得

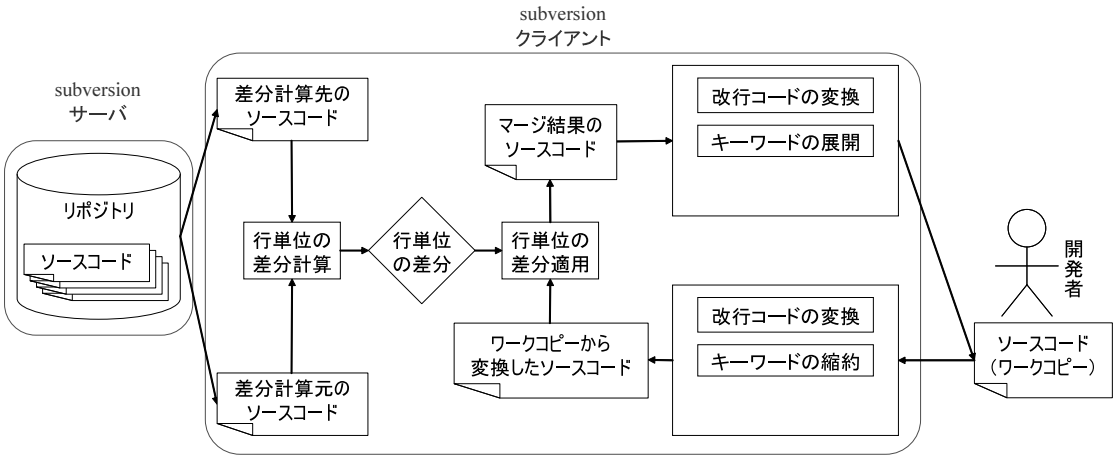


図 3.15: subversion のマージ

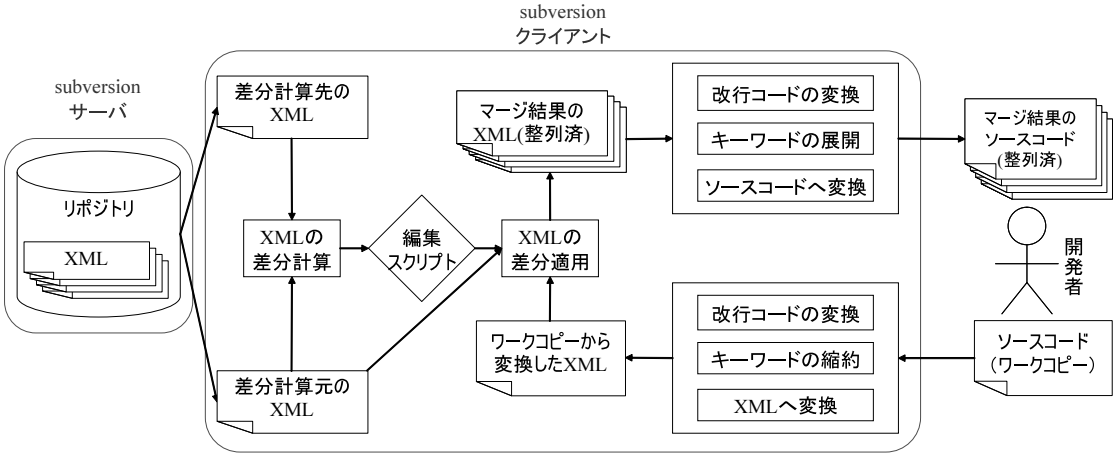


図 3.16: 木構造に対応したマージ

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，3.4.1 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 3.16 に示す．3.4.1 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，格納の時と同様に，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 2 つ以上得られた場合には，開発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， 3.3.4 節で説明した点数によって整列されている．

## 3.5 実験

### 3.5.1 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 3.1 と 表 3.2 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 3.1 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するなど，全てのマージ結果で失敗した．

表 3.2 では， 1 件を除いて，マージに成功して 1 つの出力をするか，リンク先が存在しない不正なリンクを発見することができた．しかし，ソースコード 1 に対して差分 2 を適用した時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．

### 3.5.2 実際の開発履歴に対する擬似的な適用

#### 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（22606 ファイル，162683 リビジョン）と Jakarta プロジェクト（19420 ファイル， 103358 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．実験を行なったコンピュータは， Xeon CPU 2.80GHz とメインメモリ 4GB を搭載しており，OS として Linux 2.6.16 が動作している．

#### 結果

実験結果の概要を，表 3.3 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全て成功した．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 3.4 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，マージすることが出来ず，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 3.5 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，1000 を越える候補が出力されたが，残りの 82 件では数件から数十件の範囲であった．

表 3.1: 行単位のマージを試みた結果

| | **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 3.2: 提案手法によるマージを試みた結果

| | **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 3.3: 実験 2 の結果概要

| **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 |

表 3.4: 実験 2 でテキストのマージに失敗した場合の詳細

| **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 3.5: 実験 2 のマージ結果の候補数

| 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 |
| 1250 | 1 |
| 1312 | 1 |

表 3.6: 実験 2 の提案手法の出力における正解の順位

| 順位 | 件数 |
|---:|---:|
| 1 | 78 |
| 5 | 1 |
| 16 | 1 |

表 3.6 に提案手法がマージに成功したときに，正しいマージ結果が何位に出現したかを示す．5 位と 16 位に 1 回ずつ正しい結果が出現したが，残りは全て 1 位の候補が正しいマージ結果であった．

以上のことから，開発者が正しいマージ結果を選択するのは容易であると言える．

図 3.17 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全ての時間を足し合わせてものである．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．

## 3.6 関連研究

Tom Mens [39] は，ソフトウェアのマージ技術を分類した．この分類基準に拠ると，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．

Bernhard Westfechtel [49] は， 構文上の 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，タグのつ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とも使用できる点や，マージ結果の候補を複数出力する点でも異なる．

David 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 [8].

Marc Shapiro [45] A. Kermarrec [29] らは，編集スクリプト内の操作間で依存関係を計算し，操作を並び換えることで，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 [41]，XmlDiff [3]， DeltaXML [35]， XML TreeDiff [22]， diffmk [40] ， xydiff [24] などが挙げられる．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．

## 3.7 結論と課題

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，マージによって起こる問題を減らすことが出来る．

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わかった．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る場合があることがわかった．

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挙げられる．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，システム利用者の作業量を評価する必要がある．

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

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

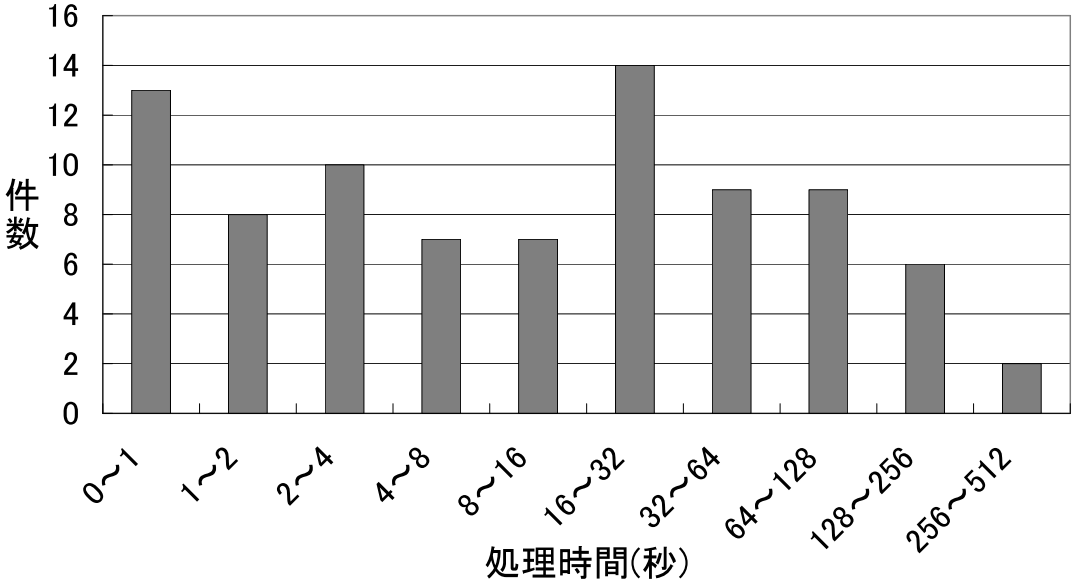


図 3.17: 実験 2 のマージにかかった時間

# 第4章　むすび

## 4.1　まとめ

本研究では既存のソフトウェアに対する変更に着目し，そこに存在する問題点2つの解決を試みた．

影響波及解析を利用した保守作業の労力見積りに用いるメトリクスの提案では，保守の見積りが客観的な基準を用いずに熟練者によって行われている問題に対し，ソースコードを対象とした影響波及解析手法を用いて，保守作業の労力を見積るメトリクスを提案した．実際に保守作業を行う実験によって提案したメトリクスを評価した結果，提案したメトリクスは，既存のメトリクスに比べて，労力と高い相関を持つことが確認できた．さらに，ソースコード以外の成果物を対象とした場合や，具体的な変更要求が得られないときの本手法の適用可能性について議論した．

次に，構文木の差分を用いた版管理システム向きマージ機能では，過去の研究で提案された構文レベルのマージシステムが版管理システムに組込まれていなかったり，ソースコードのまま任意のエディタで編集することが出来ないのに着目し，任意のエディタを用いてソースコードを編集できる構文レベルのマージシステムを提案し，既存の版管理システムである subversion に組込んだ．提案したシステムは，リポジトリに XML で表された構文木を格納することで構文レベルのマージを実現するが，開発者には構文木から復元したソースコードを提示することで，これまでの開発プロセスに与える影響を最小にすることに成功した．さらに，提案したシステムの評価を行った結果，これまで版管理システムで一般的であった行単位のマージ機能と比べてより正確なマージ結果を出力することが示された．

## 4.2　今後の研究方針

今後の研究方針としては，これまでに行った研究を応用することで，既存のソフトウェアに対する変更作業を，より正確かつ小さな労力で行えるよう支援することを考えている．

まず，保守ポイントは現在，個々の保守作業の労力を見積るためにのみ用いている．これは，保守ポイントの値が大きいモジュールは，そのモジュールに保守作業が発生した場合に大きな労力が必要であることを表している．別の観点として，大規模なソフトウェアの中から，保守ポイントが大きいモジュールを自動的に抽出することで，リファクタリング [17] の必要な個所を保守作業者に提示することが出来るのではないかと考えている．また，保守ポイントの定義のために作成したモデルを応用することで，保守作業を包括的に請け負う際のコスト見積りを行うことも大きな目標である．

また，版管理システムの研究についてであるが，現在の構文木の差分を用いた版管理システムでは，同一ソースコードの異なる版の間でのみ，頂点の対応を計算していた．これを，異なるソースコード間での頂点の対応を計算するようにすることで，より正確なマージ作業が可能になると期待できる．また，ソースコードと構造化された設計文書との間での頂点の対応を計算することで，文書とソースコードのどちらか一方だけを変更することによって起きる設計と実装との乖離を防止し，保守作業を効率化することが出来ると考えている．

# 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] *IEEE Standard for Software Maintenance (IEEE Standard 1219-1998)*, 1998.

[5] *18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada*. IEEE Computer Society, 2002.

[6] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Software Eng.*, 30(8):491–506, 2004.

[7] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio. Evaluating software degradation through entropy. In *METRICS '01: Proceedings of the 7th International Symposium on Software Metrics*, p. 210, Washington, DC, USA, 2001. IEEE Computer Society.

[8] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.

[9] B. W. Boehm. Software engineering. *IEEE Trans. On Computers*, 12(25):1226–1242, December 1976.

[10] L. C. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on uml designs. In *ICSM* [5], pp. 252–261.

[11] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, 1996.

[12] Z. Chen, Y. Zhou, B. Xu, J. Zhao, and H. Yang. A novel approach to measuring class cohesion based on dependence analysis. In *ICSM* [5], pp. 377–384.

[13] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Syst. J.*, 28(2):294–306, 1989.

[14] F. Fioravanti and P. Nesi. Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. *IEEE Trans. Software Eng.*, 27(12):1062–1084, 2001.

[15] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In G. Parikh and N. Zvegintzov eds., *GUIDE 48*, Philadelphia, PA, April 1983.

[16] K. F. Fogel. *Open Source Development with CVS*. Coriolis Group Books, Scottsdale, AZ, USA, 1999.

[17] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[18] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, p. 190, Washington, DC, USA, 1998. IEEE Computer Society.

[19] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pp. 312–326, New York, NY, USA, 2001. ACM Press.

[20] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pp. 284–293, Washington, DC, USA, 2004. IEEE Computer Society.

[21] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[22] IBM. XML TreeDiff. http://alphaworks.ibm.com/tech/xmltreediff.

[23] O. S. Initiative. The open source definition.
http://www.opensource.org/docs/definition.php.

[24] INRIA. XyDiff.
http://www-rocq.inria.fr/gemo/XyDiff/.

[25] International Function Point Users Group. *Function Point Counting Practices Manual: Release 4.2*, 2004.

[26] M. Jørgensen. Experience with the accuracy of software maintenance task effort prediction models. *IEEE Trans. Software Eng.*, 21(8):674–681, 1995.

[27] G. Karner. Use case points - resource estimation for objectory projects. Objective Systems SF AB (Rational software), 1993.

[28] 川口, 松下, 井上. 版管理システムを用いたコードクローン履歴分析手法の提案. 電子情報通信学会論文誌 D, J89-D(10):2279–2287, 2006.

[29] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pp. 210–218, New York, NY, USA, 2001. ACM Press.

[30] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pp. 126–135, New York, NY, USA, 2005. ACM Press.

[31] 小林, 吉野, 井上, 早瀬, 松尾, 上村. 保守の影響波及範囲に基づいたレガシーシステムの障害予測. 信学技報, 105(491):31–36, December 2005.

[32] 楠田. メソッドの同時更新履歴を用いたクラスの機能別分類法. 大阪大学大学院情報科学研究科 修士論文報告会, Feb 2006.

[33] R. Leitch and E. Stroulia. Assessing the maintainability benefits of design restructuring using dependency analysis. In *IEEE METRICS*, pp. 309–. IEEE Computer Society, 2003.

[34] M. Lindvall, R. T. Tvedt, and P. Costa. Avoiding architectural degeneration: An evaluation process for software architecture. In *IEEE METRICS*, pp. 77–86. IEEE Computer Society, 2002.

[35] M. E. Ltd. DeltaXML.
http://www.deltaxml.com/.

[36] S. Mamone. The ieee standard for software maintenance. *SIGSOFT Softw. Eng. Notes*, 19(1):75–76, 1994.

[37] J. Martin and C. L. McClure. *Software Maintenance: The Problems and Its Solutions*. Prentice Hall Professional Technical Reference, 1983.

[38] C. McClure. *The three Rs of software automation: re-engineering, repository, reusability*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[39] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[40] S. Microsystems. diffmk.
http://www.sun.com/xml/developers/diffmk/.

[41] A. Mouat. Xml diff and patch utilities. http://diffxml.sourceforge.net/, June 2002.

[42] 中山, 松下, 井上. ソースコードの差分を用いた関数呼び出しパターン抽出手法の提案. 情報処理学会研究報告, 2006(35):49–56, Mar 2006.

[43] 佐々木, 松下, 井上. 開発履歴情報に基づいたダイナミックコミュニティ選定支援手法. 信学技報, 104(571):1–6, January 2004.

[44] S. R. Schach. *Software Engineering*. Asken Associates, 1990.

[45] M. Shapiro, A. Rowstron, and A.-M. Kermarrec. Application-independent reconciliation for nomadic applications. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pp. 1–6, New York, NY, USA, 2000. ACM Press.

[46] H. M. Sneed. Estimating the costs of software maintenance tasks. In *ICSM*, pp. 168–181. IEEE Computer Society, 1995.

[47] I. Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1996.

[48] D. Tran-Cao, G. Lévesque, and A. Abran. Measuring software functional size: Towards an effective measurement of complexity. In *ICSM* [5], pp. 370–376.

[49] B. Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configur ation management*, pp. 68–79, New York, NY, USA, 1991. ACM Press.

[50] 山崎. 共通問題によるプログラム設計技法解説. 情報処理, 25(9):934–935, 1984.

[51] S. Yip and T. Lam. A software maintenance survey. In *First Asia-Pacific Conference on Software Engineering*, pp. 70–79, 1994.

[52] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, p. 73, Washington, DC, USA, 2003. IEEE Computer Society.