特別研究報告

題目

構文情報を利用して開発履歴から
コーディングパターンを抽出する手法の提案

指導教官
井上 克郎 教授

報告者
宮本 和輝

平成 19 年 2 月 20 日

大阪大学 基礎工学部 情報科学科

平成 18 年度 特別研究報告

構文情報を利用して開発履歴からコーディングパターンを抽出する手法の提案

宮本 和輝

内容梗概

ソフトウェア部品を再利用することは，ソフトウェアの生産性や品質の向上に役立つため非常に重要である．再利用を行う際はまず，再利用するソフトウェア部品の利用法を理解する必要がある．その際，一般的には部品に附属するドキュメントやサンプルプログラムなどが用いられるが，その種の文書が附属しないソフトウェア部品は多く，その結果，部品の再利用が行いにくかった．

この問題を解決する手法の一つとして，コーディングパターンの抽出がある．コーディングパターンとは，ソースコードに頻出する構造のよく似た記述であり，これにより再利用対象のソフトウェア部品の利用法を学ぶことができる．

このコーディングパターンを抽出する手法の一つに，版管理システムに蓄積されているソースコードの差分からコーディングパターンを抽出するというものがある．しかし，既存の差分を用いた抽出手法ではソースコードの中の一度も変更が加えられていない箇所に存在するコーディングパターンは抽出されず，また，抽出されたコーディングパターンにおいても利用法の理解に用いることのできない制御文が同時に抽出されてしまうという問題点があった．

そこで，本研究では，上記の問題を解決し，抽出されるコーディングパターンの有用性を向上させることを目的としたコーディングパターン抽出手法を提案する．本手法では，一度も変更が加えられていない箇所も差分とし，また，構文情報を用いてコーディングパターンの抽出を行う．提案手法を実装したシステムを作成し，実際のオープンソースプロジェクトに対して適用することで，本手法が上記の問題を解決していることを確認した．

主な用語

ソフトウェア部品
ソフトウェア再利用
コーディングパターン

版管理システム

Sequential pattern mining

目 次

1 まえがき

近年のインターネットの普及により，WWW を通じて大量のソフトウェア資産が容易に入手可能となった．また，オープンソースソフトウェアは，そのソフトウェアの規模が増加するにつれて，単にソフトウェア自身を利用することによる利便性の向上だけでなく，ソフトウェアを構成する様々な部品を発見して他のソフトウェア開発に再利用するといった，再利用可能なソフトウェア資産としての価値も高まっている．SourceForge [26] では，ソースコードのストレージやコミュニケーションに必要なメーリングリストなどソフトウェア開発に必要とされる開発資源を提供しており，2007 年 2 月の時点で，14 万以上ものプロジェクトを保持し，また，開発者として登録されているユーザ数は 150 万を超える．

一方，ソフトウェアの再利用 [2] [3] に関する研究はこれまで実際に既存のソースコードを流用することによって再利用を行う際には，再利用対象となるソースコードの断片 (ソフトウェア部品) の利用法を理解しなければならない．ここでいうソフトウェア部品の利用法とは，ソフトウェア部品の初期化の仕方や，処理を行うために必要な関数の呼出し順などである．一般に，ソフトウェア部品の利用法を理解する場合，そのソフトウェア部品に附属するドキュメントやサンプルプログラムを用いる．例えば，ソフトウェア部品のドキュメントを閲覧することでその仕様を学ぶことが出来，サンプルプログラムを閲覧することで詳細なコーディングの仕方を学ぶことが出来る．しかし，小規模なソフトウェアや過去に作成されたソフトウェアなどの場合，ドキュメントやサンプルプログラムが附属していない場合や存在していても実際のソースコードとは食い違う内容であることが多く，その結果ソフトウェア部品の利用法を学習するのは困難である．

この問題を解決するために，コーディングパターンを用いてソフトウェア部品の利用法を理解する手法がある．コーディングパターンとは，ソースコード中に頻繁に現れる構造のよく似たコード記述である．開発者はコーディングパターンを閲覧することで，再利用対象のソフトウェア部品を利用するためにはどのような関数をどのように呼び出せばよいか学ぶことができる．

このコーディングパターンをソースコードから抽出する手法の一つに，版管理システムに蓄積されているソースコードの差分からコーディングパターンを抽出するというものがある [1]．この手法は，版管理システムに蓄積されているソースコードの差分が個別の機能に関わるものである，ということに着目しコーディングパターンの抽出を行う．しかし，この抽出手法では，ソースコードの中の一度も変更が加えられていない箇所に存在するコーディングパターンを抽出することが出来ず，また，構文情報を用いてずにコーディングパターンの抽出を行っているため，抽出されたコーディングパターンにおいて，抽出された関数呼出しを含んでいない制御文が同時に抽出される，という問題が存在していた．

そこで本研究では，上記の問題を解決し，抽出されるコーディングパターンの有用性を向上させることを目的としたコーディングパターン抽出手法を提案する．具体的には，ソースコードの中に存在する一度も変更が加えられていない箇所も差分として扱い，コーディングパターン抽出時に構文情報を利用し，関係のない制御文の抽出を行わないようにしている．

以降，2 節では本研究の背景となるソフトウェア部品の再利用と版管理システムについて述べる．3 節では既存の差分を用いたコーディングパターン抽出手法とその問題点について説明し，4 節でその問題点に対して本研究で提案するコーディングパターン抽出手法について述べる．5 節ではその実装について述べる．6 節では提案手法の評価を行う．最後に，7 節で本研究のまとめと今後の課題について述べる．

2　ソフトウェア再利用と版管理システム

2.1　ソフトウェア再利用

本節では，まず，ソフトウェア部品とその再利用法について説明した後，コーディングパターンについて説明を行う．

2.1.1　ソフトウェア部品と再利用

一般にソフトウェア部品 *(Software Component)* とは，再利用 *(reuse)* できるように設計されたソフトウェアの実体とされている [3]．過去のソフトウェア開発における成果物などからなるソフトウェア部品の集合をライブラリ *(library)* という．以下，ソフトウェア部品を単に部品と呼ぶ．ライブラリ中の部品を同一システム内や他のシステムで用いることを再利用といい，ソフトウェアの生産性と品質を改善し，結果としてコストを削減するという報告が多く出されている [2] [4]．部品という概念はプログラムソースコードやバイナリ実行ファイル，その他に設計仕様や構造，テスト項目，そしてそれらの文書であったりと，ソフトウェア開発過程で生成されたあらゆる成果物に適用され，その種類は様々である．部品の再利用可能な度合いを再利用性 *(reusability)* といい，特定の設計やコーディングの標準に従うことで，部品の再利用性を向上させることができる．ソフトウェアの再利用は，部品の形式や再利用の目的をもとに，ホワイトボックス再利用 *(white-box reuse)* とブラックボックス再利用 *(black-box reuse)* という分類をすることがある．

ホワイトボックス再利用 :

仕様や文書，プログラムソースコードの再利用を指す．部品の内部構造に基礎を置くため，部品の詳細を把握することが可能であり，用途に応じて修正可能で，プラットフォームに非依存である．ホワイトボックス再利用における再利用者は主にソフトウェア開発者である．特にソースコード再利用に関しては，ライブラリ内の部品を利用者の再利用環境に応じて洗練する手法に関する研究などが存在する [5].

ブラックボックス再利用 :

主にバイナリ実行ファイルの再利用を指す．具体的には，*JavaBeans* や *ActiveX/DCOM*, *CORBA* などがあり，部品の詳細を知らずにその機能だけを再利用したいときに有効である．プログラムに詳しくないエンドユーザのソフトウェア開発で行われるもので，開発形態としてはビジュアルプログラミングなどがある．コンパイル不要で迅速に再利用が可能な反面，プラットフォームに依存しており，詳細な解析は困難である．

本研究で扱う再利用とは，ホワイトボックス再利用，特にソースコードに対象を絞った再利用のことである．ソースコードは，それが実現する個別の機能ごとにクラスや関数と呼ばれる単位に分けられる．開発者はこれらの個別の機能を実現する部品群を組み合わせることで大きな機能を実現するプログラムを書いていく．また，これらのクラスや関数には，それ単体では意味をなさず，複数のものをある一定の手順で組み合わせることで初めて望んだ機能を実現するものも存在する．

2.1.2 ソフトウェア部品の利用法

ソフトウェア部品を再利用する際には，まず，再利用対象となる部品の利用法を学習しなければならない．例えば，再利用対象となる部品が関数である場合は，以下のようなことを学習しなければ望んだ機能を実現できない．

- 実現したい処理に必要な関数群
- 必要な関数群の呼出し順
- 各関数の引数や返り値の扱い方

一般に，部品の利用法の学習はその部品に附属するドキュメントやサンプルプログラムをもとにして行われる．ドキュメントには，主に部品が実現する機能や仕様などが詳細に書かれている．また，サンプルプログラムには，部品を利用して目的の機能を実現するためのソースコードの例が書かれている．開発者は部品のドキュメントを見ることで，その部品についての理解を深め，それからサンプルプログラムを読んだり模倣したりすることで，基本的な部品の利用法を学習していく．さらに高度な利用法を学習するには，それまでに得られた基本的な知識とドキュメントのさらなる熟読が必要になる．

商用のソフトウェア部品や大規模なオープンソースのソフトウェア部品などにはこのようなドキュメントやサンプルプログラムが附属しているので，それを用いて利用法の学習を行うことができる．しかし，個人で開発した部品や規模の小さい部品ではこれらが附属していないことが多い．そのような部品では利用法を学ぶことが困難であるため再利用が難しくなる．

2.1.3 コーディングパターン

ドキュメントやサンプルプログラムが附属していない部品の利用法の理解を支援するために，再利用対象の部品を利用しているソースコードからその部品の利用例を取り出し，それを開発者に提示することで利用法の理解に役立てるという研究が過去になされている [6] [7] [8] [9].

その中の一つとして，再利用対象の部品を利用しているソースコードからコーディングパターンと呼ばれるものを抽出し，それを開発者に提示することで利用法の理解に役立てるという研究が存在する [10]．コーディングパターンとはソースコード内に頻繁に出現する構造のよく似たコード記述のことである．

ソフトウェア部品を再利用するにはある一定の手順を踏まなければならない．例えば，部品に対して初期化作業を行い，それから主要な処理を実行し，最後に終端処理を施して初めて機能を果たす部品も存在する．そのため，ある部品をソースコード内のいくつかの箇所で利用しているということは，ソースコード内によく似たコード記述がいくつか現れることになる．そこで，ソースコード内に頻繁に出現するコード記述であるコーディングパターンを抽出することで，部品の利用例を取り出し，それを利用法の理解に役立てるのである．

コーディングパターンを閲覧することで，開発者は以下のことを学習することができる．

実現したい処理に必要な関数群：

一般のソフトウェア開発においては，一つの機能は複数の関数の組み合わせからなることが多い．そこで，部品を再利用するにはまず，利用したい機能を実現している関数はどれであるかということを理解しなければならない．コーディングパターンには一連の処理に必要な関数呼出しが含まれているため，これを閲覧することで処理に必要な関数群を理解することができる．

関連のある関数の呼出し順：

一つの機能を実現する関数群の呼出し順序にも一定の決まりがある．初期化作業を行う関数は一番初めに呼び出さなければならず，ある関数によって得られる値を参照する関数はその関数より後に呼び出さなければならない．コーディングパターンは単なる関数の列挙ではなく，ソースコードの構造を持ったものであるため，そこに現れる関数呼出しの順序を閲覧することで，具体的な呼出し順を学習することができる．

関数の引数や返り値の扱い方：

コーディングパターンは部品の具体的な利用法を抽象化して取り出したものであるため，それだけでは具体的なソースコードの書き方までは理解できない．そこで，対象のコーディングパターンを実際に利用しているソースコードを閲覧することで，引数の指定の仕方や返り値の扱い方といった詳細なコーディングの仕方を学習することができる．

2.2 版管理システムとその性質

本節では，まず，版管理システムとその役割について説明した後，版管理システムへのソースコードの変更の保存が個別の機能ごとに行われるということについて述べる．

2.2.1 版管理システム

版管理とは，主として以下の 3 つの役割を提供する機構である．

- プロダクトに対して施された追加・削除・変更などの作業を履歴として蓄積する
- 蓄積した履歴を開発者に提供する
- 蓄積したデータを編集する

ソースコードやリソースといった各プロダクトの履歴データは，リポジトリ *(Repository)* と呼ばれるデータ格納庫に蓄積される．その内部では，プロダクトのある時点における状態であるリビジョン *(Revision)* を単位として管理する．1 つのリビジョンには，ソースコードやリソースなどの実データと，作成日時やログメッセージなどの属性データが格納されている．

また，リポジトリとのデータ授受をするために，開発者はシステムに依存したオペレーションを利用する必要がある．

版管理手法を述べるにあたり，その基礎となるモデルが幾つか存在する [11] [12]．本節では，多くの版管理システムが採用している Checkout/Checkin モデルについて概要を述べる．なお，以降本文において，プロダクトのある時点における状態のことをリビジョンと呼ぶことにする．

The Checkout/Checkin Model

このモデルは，ファイルを単位としたリビジョン制御に関して定義されている (図 1 参照)．

リビジョン管理下にあるコンポーネントは，システムに依存したフォーマット形式のファイルとしてリポジトリに格納されている．開発者はそれらのファイルを直接操作するのではなく，各システムに実装されているオペレーションを介してリポジトリとのデータ授受を行う．リポジトリから特定のリビジョンのコンポーネントを取得する操作をチェックアウト *(Checkout)* という．逆に，データをリビジョンに格納し，新たなリビジョンを作成する操作をチェックイン *(Checkin)* という．

単純にリビジョンを作成するのみでは，シーケンシャルなリビジョン列を生成することになる．しかし，過去のリビジョンに遡り，別の工程 (デバッグ等) で開発を行う場合のため

に，リビジョン列をブランチ *(Branch)* という操作によりブランチを生成し，その上にリビジョンを作成するという手法を採る．このブランチに対して，元のリビジョン列のことをトランク *(Trunk)* という．また，ブランチ上での作業内容 (デバッグ修正部分等) を，別のリビジョンに統合する作業をマージ *(Merge)* という．このように，リビジョン列は木構造になることからリビジョンツリー *(Revision Tree)* と呼ばれる．

版管理システムと呼ばれるものは多数存在する．UNIX 系 OS では多くの場合，RCS [13] や CVS [14] [15] といったシステムが標準で利用可能となっている．ClearCase [16] のように商用のものも存在する．また，UNIX 系 OS だけではなく，Windows 系 OS においても，VisualSourceSafe [17] や PVCS [18] をはじめ，数多く存在する．さらに，ローカルネットワーク内のみではなく，よりグローバルなネットワークを介したシステム [19] も存在する．

ここでは，版管理システムの内のいくつかを紹介する．

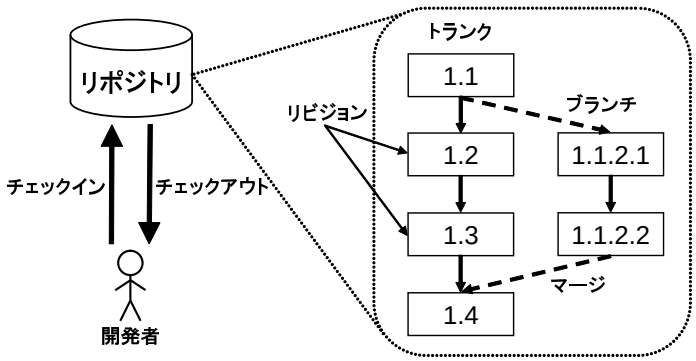

図 1: Checkout/Checkin Model

RCS :

RCS [13] は UNIX 上で動作するツールとして作成された版管理システムであり，現在でもよく使用されているシステムである．単体で使用される他，システム内部に組み込み，版管理機構を持たせるなどの用途もある．RCS ではプロダクトをそれぞれ UNIX 上のファイルとして扱い，1 ファイルに対する記録は 1 つのファイルで行われる．RCS におけるリビジョンは，管理対象となるファイルの中身がそれ自身によって定義され，リビジョン間の差分は diff コマンドの出力として定義される．各リビジョンに対する識別子は数字の組で表記され，数え上げ可能な識別子である．新規リビジョンの登録や，任意のリビジョンの取り出しは，RCS の持つツールを使用する．

CVS :

CVS [14] [15] は RCS 同様，UNIX 上で動作するシステムとして構築された版管理システムであり，近年最もよく使われるシステムの 1 つである．RCS と大きく異なるのは，複数のファイルを処理する点である．また，リポジトリを複数の開発者で利用することも考慮し，開発者間の競合にも対処可能となっている．さらに，ネットワーク環境 (ssh，rsh 等) を利用することも可能であるため，オープンソースによるソフトウェア開発の場面で活躍の場が多い．その最たる例が FreeBSD [20] や OpenBSD [21] 等のオペレーティングシステムの開発である．

2.2.2 ソースコード更新の性質

開発者はソースコードに対して変更を加える都度，その変更内容を版管理システムにチェックインしていくことで開発作業を進めていく．このとき，一度のチェックインで変更されるソースコードは，ある一つの機能についてのソースコードであることが一般的となっている．以下に，版管理システムへのチェックインが機能ごとに行われることの根拠について述べる．

不完全なソースコードをチェックインすることはないため：

版管理システムに変更内容を保存しながら開発を行うのは，保存された開発状態を後で復元して利用するからである．例えば，開発途上のソースコードの既存機能の部分にバグが存在することが分かったとする．ここで開発中のソースコードに対してバグ修正を行うと，不完全なソースコードを扱うことになるため，新たなバグを埋め込む可能性が高くなる．そこで，過去の安定版のソースコードからブランチを作成し，そこでバグ修正をした結果をトランクへマージするということを行う．しかし，この過去の状態のソースコードが不完全な状態だと，このような作業を行うことも困難になる．そのため，後でソースコードの状態が復元されることを考えて，一つ以上の機能が正しく更新された状態でソースコードをチェックインすることが慣習となっている．

開発状態の復元を柔軟に行うため：

チェックインした機能に問題が発生した場合，一旦そのチェックインの前の状態に戻るということがある．このとき，複数の機能を同時にチェックインしていると，その内の一つだけに問題があった場合でも，それら全ての機能への変更が無かった状態へと戻らなければならない．細かくチェックインを繰り返すことで，このような開発状態の復元作業に柔軟性を持たせることができる．

大規模なチェックインには困難が伴うため：

現在広く使われている版管理システムには複数の開発者が同時に開発作業を行うことを支援する機能を持つものが多い．それらの機能の一つに，複数の開発者間でのソー

スコード変更の競合に対処するための機能が挙げられる．しかし，競合による問題は全て版管理システムが解決するわけではなく，最終的には人間が解決することになる．そこで誤りが起きてしまう可能性もあるため，開発においてはできるだけ変更の競合は起こさないほうがよい．もし，一度に大量のソースコードをチェックインしていると，競合が起きたときに影響を受ける範囲が広くなってしまう．そのため，チェックインを行うときは出来るだけ細かく行う方がよい．

また，既存の研究からも，このチェックインの性質が一般的であることが伺える．Gall らは版管理システム内の開発履歴から，ソースコードの構造からは分からない Logical Couplings と呼ばれる依存関係を抽出し，ソフトウェアの理解や保守に役立てる研究を行った [22]．他にも，Zimmermann らは開発履歴からソースコードの変更ルールの抽出を行い，変更点の予測や不完全な変更による不具合を防ぐ為のシステムを開発している [23]．これらの研究に共通しているのは，一度のチェックインで扱うソースコードは共通の機能を実装しているということを仮定している点である．これらの研究が良い結果を残していることを考えると，実際にソフトウェア開発においても一度のチェックインで扱うソースコードは個別の機能に関わるものであるという仮定は問題無く受け入れられるものであることが分かる．

3　既存のコーディングパターン抽出手法と問題点

本節では，まず既存のコーディングパターン抽出手法について説明した後，既存の抽出手法に存在する問題点について述べる．

3.1　既存のコーディングパターン抽出手法

既存のコーディングパターン抽出手法は大きく分けて 3 つのステップからなっている．まず始めに行われるのが，ソースコードの差分からソースコードの特徴と呼んでいるもの取得することである．次に，取り出されたソースコードの特徴から特徴シーケンスと呼んでいるものを生成し，最後に，生成された全ての特徴シーケンスに対して sequential pattern mining と呼ばれる手法を適用することでコーディングパターンの抽出を行っている．

図 2 は，コーディングパターン抽出までの流れを表したものである．

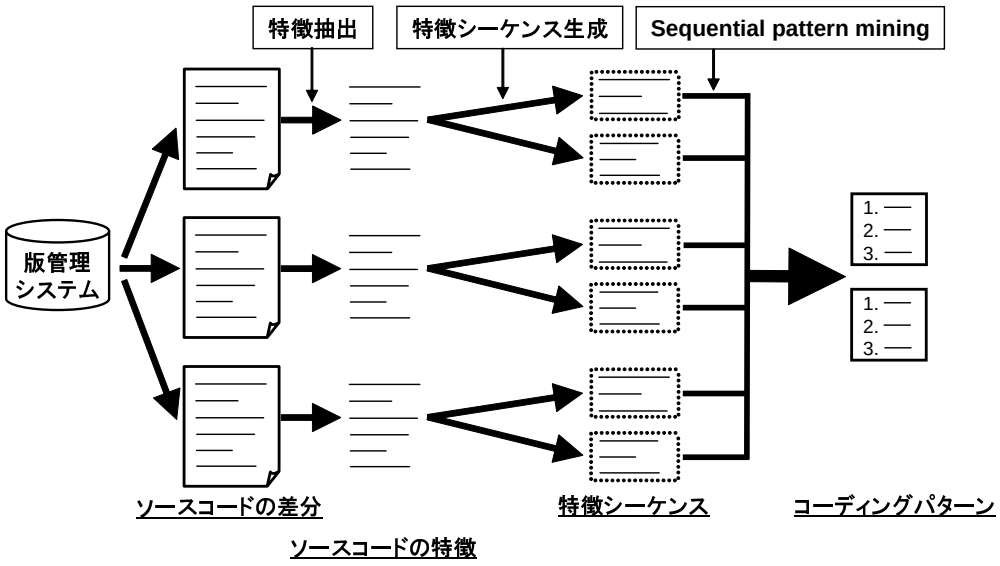

図 2: コーディングパターン抽出の流れ

以下で，これら 3 つのステップについての説明を行う．

3.1.1　ソースコードの特徴の取得

まず始めに，ソースコードの差分からソースコードの特徴と呼ばれるものを取得する．本手法では，以下に挙げる要素をソースコードの特徴としている．

- 関数呼出し
- 条件文の開始位置・終了位置
- 繰返し文の開始位置・終了位置

開発者は，ある機能を実装した関数を複数組み合わせることで，より大きな機能を実現していく．したがって，関数呼出しはそのプログラムがどういったことを実現しようとしているかを知るのに良い指標となることからソースコードの特徴として選ばれている．また，関数呼出しだけではなく，条件文の開始位置・終了位置や繰り返し文の開始位置・終了位置といったような，プログラムの制御構造を決定するような要素もソースコードの特徴として選ばれている．これは，関数呼出しを組み合わせてある機能を実現する際に，単純に関数呼出しを並べるだけで実現できるということは少なく，ある関数呼出しの返り値の結果によって処理を変えたり，処理が終るまで同じ関数呼出しを繰り返すといったことがよく行われるためである．そのため，最終的に抽出されるコーディングパターンに制御構造の情報も持たせている．以下では，条件文の開始位置・終了位置と繰返し文の開始位置・終了位置といったソースコードの特徴を制御文要素と呼ぶことにする．

ソースコードの特徴の取得は全ての隣接するリビジョン間のソースコードの差分に対して行われる．版管理システムではソースコードの差分を diff プログラムの出力で表しており，それらは新しいリビジョンにおいて追加された行，削除された行，そして編集された行の 3 種類に分類できる．特に，diff プログラムでは編集された行は，行の削除と追加を組み合わせて実現されているので，実質的にはソースコードの差分は追加された行と削除された行の 2 種類で構成されているといえる．本手法では，それらの内の新しいリビジョンにおいて追加された行をソースコードの特徴の取得対象としている．

ソースコードの特徴を取り出す対象を差分のみに限定している理由は，抽出するコーディングパターンが個別の機能に限定されたものにするためである．2.2.2 節で述べたように，版管理システムへのチェックインは個別の機能についてのソースコードの変更を基本として行われるため，隣接するリビジョン間の差分に注目することで，全体のソースコードからある一つの機能に関わるソースコードを抜き出すことができる．そのため，ソースコードの差分から抽出したコーディングパターンは個別の機能に限定されたものとなる．

次に，取得したソースコードの特徴から不要な要素を取り除く．本手法では不要な要素を標準関数呼出しと，同じ構造の制御文要素のみの繰り返しとしている．標準関数はドキュメントやサンプルプログラムも充実しているため，抽出しなくても十分に利用法を学習できると判断できるからである．同じ構造の制御文要素のみの繰り返しとは，例えば「*if* → *end if* → *if* → *end if* → *if* → *end if*」といったもので，このようなものはコーディングパターンに寄与しているとは考えにくいため除去の対象とされている．

3.1.2 特徴シーケンスの生成

次に，取得したソースコードの特徴を特徴シーケンスという単位にまとめる．特徴シーケンスとは関数の利用例を抽象化したものであり，具体的には，一つの関数定義内におけるソースコードの特徴のリストである．ただし，本手法ではソースコードの差分からコーディングパターンの抽出を行うため，特徴シーケンスを構成するソースコードの特徴は新しいリビジョンにおいて追加された行に存在していたもののみとなる．

図 3 は，ソースコードの差分から特徴シーケンスを抽出するまでの流れを表したものである．図の左側はソースコードの差分を表しており，先頭に'+' がついている行は新しいリビジョンにおいて追加された行を表している．そこからソースコードの特徴を取得した後，1 つの関数定義に対して 1 つの特徴シーケンスを生成している．この操作を全てのソースコードの全てのリビジョンに対して行い，全ての特徴シーケンスを取得する．

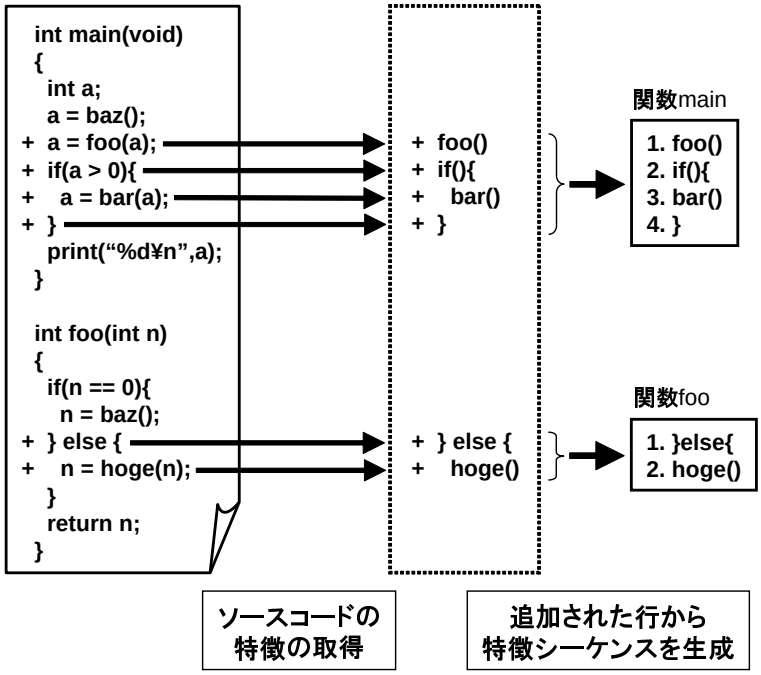

図 3: ソースコードの差分からの特徴シーケンス抽出の流れ

次に，生成された特徴シーケンスから不要なシーケンスを除去する．関数呼出し要素を一つも含んでいない特徴シーケンスは，コーディングパターンの抽出には必要ないと判断できるため，除去の対象となる．

3.1.3 Sequential pattern mining によるパターン抽出

最後に，生成した全ての特徴シーケンスを対象にして，sequential pattern mining と呼ばれる手法を適用することでコーディングパターンの抽出を行う．Sequential pattern mining とは，与えられた複数のリストから，ユーザが指定した閾値以上の頻度で共通して現れる部分リスト (シーケンシャルパターン) を求める手法である．本手法では Sequential pattern mining を行うアルゴリズムとして，一般的に良く用いられている PrefixSpan [25] を採用している．PrefixSpan は射影と呼ばれる操作を繰り返すことでシーケンシャルパターンを抽出するアルゴリズムである．射影とは，全てのシーケンスから特定の要素からの接尾辞を取り出す操作である．図 4 は要素 a についての射影を行う様子を表している．

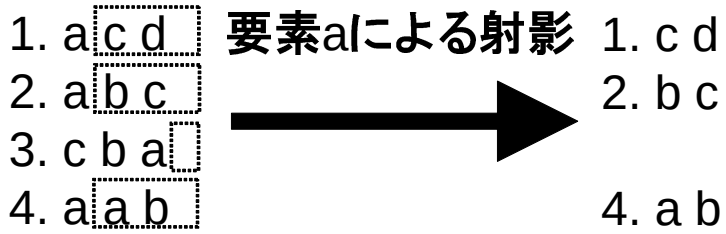

図 4: 要素 a についての射影

PrefixSpan アルゴリズムによる具体的な sequential pattern mining の流れを図 5 を用いて説明する．ここでは，パターン抽出の閾値である最小サポート値を 2 としてマイニングを行っている．

まず，各シーケンスを構成している全ての要素について，そのサポート値の計算を行う．サポート値とは要素が出現しているシーケンスの数のことである．次に，サポート値が最小サポート値以上の要素をシーケンシャルパターンとして出力する．図 5 の例では，a，b，c が長さ 1 のシーケンシャルパターンとして出力される．それから最小サポート値を越える各要素について射影を行う．そして，射影を行ったシーケンスに対してまた各要素のサポート値を計算を行い，サポート値が最小サポート値以上ならば，その要素を，1 つ前に出力したパターンの末尾につけたものをシーケンシャルパターンとして出力する．図 5 の例では，要素 a についての射影を行ったシーケンスにおいて，最小サポート値以上のサポート値を持つ b と c を，要素 a の後につけた ab，ac をシーケンシャルパターンとして出力する．この，射影，サポート値計算，シーケンシャルパターン出力，という操作を最小サポート値を満たす要素が無くなるまで繰り返すことで全てのシーケンシャルパターンを抽出する．

3.1.2 節のステップで生成した特徴シーケンスは部品の利用実績を抽象化したものと言えるので，この特徴シーケンス群に共通して存在するソースコードの特徴列を PrefixSpan アル

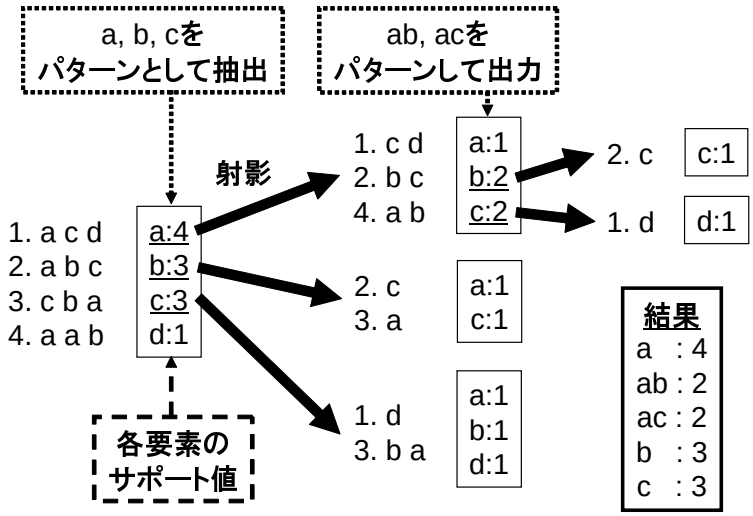

図 5: 最小サポート値 2 における PrefixSpan による sequential pattern mining

ゴリズムで抽出することによって，頻繁に用いられる部品の利用手順を抽出する．

このとき，単純に特徴シーケンスに対して sequential pattern mining を適用すると，sequential pattern mining の性質により，有用なパターンと共に非常に多くの部品利用法の理解に用いることのできないパターンが抽出されてしまう．部品利用法の理解に用いることのできないパターンには以下のようなものが存在する．

制御文要素の対応が取れていないパターン：

パターンの要素を前から順に見ていったときに，if 文の開始位置が出現する前に if 文の終了位置や else 文が出現するようなパターンのことである．このようなパターンは再利用することができない．

制御文要素の数がパターン全体の要素数の 3 分の 2 を越えるパターン：

関数呼出し要素と比べて制御文要素の数が多いパターンでは，大半の制御文要素は関数呼出しの利用パターンとは関係が無く，パターン全体としての利用価値も低い．

関数呼出し要素が 1 個以下しか無いパターン：

本手法では部品の利用パターンとして関数同士の相互呼出し関係を対象としている．パターンの中に関数呼出し要素が 0 個，ないしは 1 個しかないパターンはそのような相互呼出し関係を表しているとは考えられないため，利用法理解に用いることができないと考えられる．

本手法では，これらの不要なパターンを以下のようにして除去する．

対応する要素がない制御文要素を持つパターンを抽出しない :

PrefixSpan の各ステップでは，最小サポート値を持つ要素をその前のパターンの末尾に付加した上でパターンとして出力している．このとき図 6 のように，付加する制御文要素に対応するものが存在しない場合にそこで探索を中止することで制御文要素の対応が取れていないパターンの抽出を抑制する．

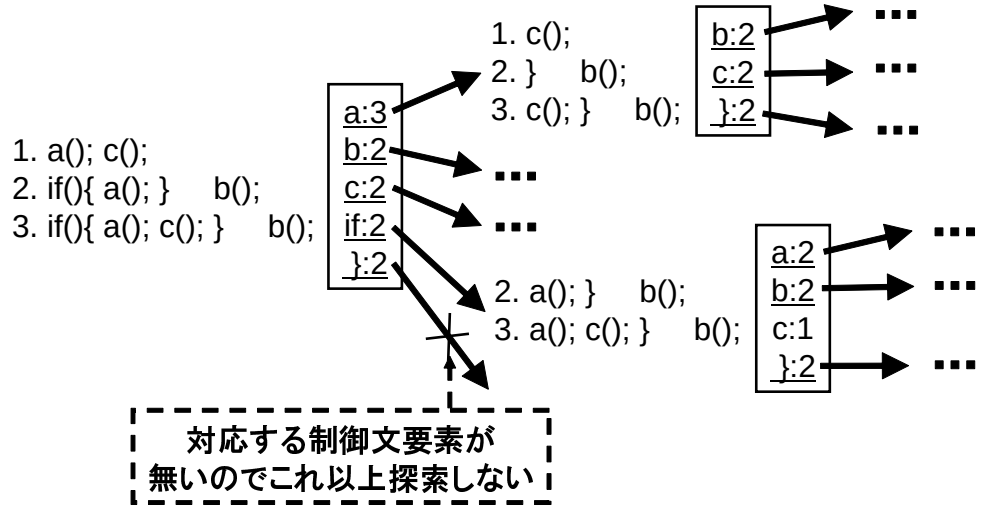

図 6: sequential pattern mining 時における不要なパターンの除去

同じ種類の制御文要素を持つパターンを抽出しない :

一般に，1 つのパターン中に同じ制御文要素がいくつも出現することはない．そこで，同じ種類の制御文要素を持つパターンの抽出を制限することで，不要なパターンの除去を行う．この制限により，制御文要素が大半を占めるパターンの抽出が抑制される．

制御文要素が連続するパターンを抽出しない :

間に関数呼出し要素をはさまない制御文要素の並びにはあまり意味が無い．そこで，パターン探索時に制御文要素が連続するような探索を抑えることで，制御文要素が連続するパターンが抽出されないようにする．この制限により，制御文要素が大半を占めるパターンの抽出が抑制される．

同じ関数呼出し要素が連続するパターンを抽出しない :

一般に，同じ関数呼出し要素が連続する場合，その関数呼出し要素はそれ単体で役割をなしていることが多い．これらの要素の除去を行わない場合，抽出されるパターンが必要以上に長くなって利用法の理解の妨げとなるため，パターン探索時に同じ関数呼出し要素が連続するような探索を抑えることで，同じ関数呼出し要素が連続するパ

ターンが抽出されないようにする．

制御文要素の数がパターン全体の 3 分の 2 を越えるパターンを除去する：

全てのパターンの抽出が終った後に，各パターンに対して検査を行い，制御文要素の数がそのパターン全体の要素数の 3 分の 2 を越えているものを除去する．

関数呼出し要素の数が 1 個以下であるパターンを除去する：

全てのパターンの抽出が終った後に，各パターンに対して検査を行い，関数呼出し要素の数が 1 個以下であるパターンを除去する．

3.2 既存手法の問題点

3.1 節で説明した既存手法に存在する問題点として以下のものが挙げられる．

一度も変更が加えられていない箇所に存在するコーディングパターンは抽出されない

ソースコード中の一度も変更されていない箇所に存在するコーディングパターンはソースコードの差分に出現することがないため，隣接するリビジョン間の差分のみを使用している既存の手法ではこのような箇所に存在するコーディングパターンを抽出することができない．

構文を考慮せずにコーディングパターンの抽出を行っている

既存の手法は，ソースコードの差分から生成した特徴シーケンスに共通して現れる部分リストを求めているだけなので，コーディングパターン中に含まれている関数呼出しが実際には含まれていない制御文も同時に抽出されることがある．このような制御文は利用法を理解するのに役立つとは言えない．

4　提案手法

本節では，3.2 節で述べた問題点に対して本研究で提案するコーディングパターン抽出手法について述べる．

本手法も，既存の手法と同じステップを経ることでコーディングパターンの抽出を行う．すなわち，ソースコードの特徴の取得，特徴シーケンスの生成，パターン抽出の 3 ステップである．

以下で，これら 3 つのステップについての説明を行う．

4.1　ソースコードの特徴の取得

本手法では，以下に挙げる要素をソースコードの特徴として取得する．

- 関数呼出し
- 条件文の開始位置
- 繰返し文の開始位置

このとき，構文情報として，各要素が存在した行番号と制御文要素についてはさらにその開始位置・終了位置の行番号を取得しておく．

図 7 に，実際にソースコードから特徴を取得する様子を示す．

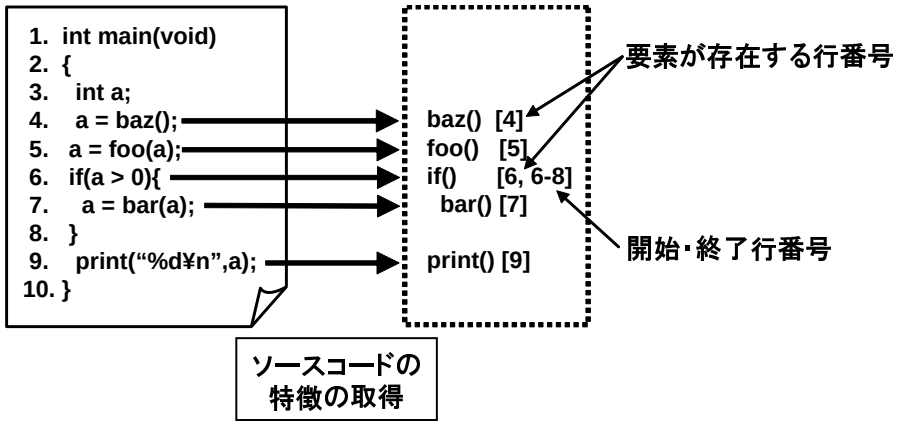

図 7: ソースコードから特徴を取得

本手法では，ソースコードの特徴の取得を，隣接するリビジョン間のソースコードの差分からだけではなく，最新リビジョンにおいて初期リビジョンから一度も変更が加えられていない箇所からも行う．初期リビジョンから一度も変更が加えられていないソースコードは，

それ自身である機能を実現していると考えられるため，個別の機能に限定したコーディングパターンを抽出するという既存手法の特徴を失うことなく，差分からは抽出することができないコーディングパターンの抽出を行うことが出来る．

次に，取得したソースコードの特徴から不要な要素を取り除く．本手法では不要な要素を標準関数呼出しと関数呼出し要素を一つも含んでいない制御文要素としている．標準関数呼出しはドキュメントやサンプルプログラムが充実しているため除去の対象とした．関数呼出し要素を一つも含んでいない制御文要素とは，開始位置から終了位置の間に関数呼出し要素が一つも存在していない制御文要素のことで，このような制御文要素はパターンを抽出する際に不要であるので除去の対象とした．

4.2 特徴シーケンスの生成

既存手法と同様の方法により，取得したソースコードの特徴を特徴シーケンスにまとめていく．具体的には，差分あるいは一度も変更が加えられていない箇所に存在しているソースコードの特徴を存在している関数定義ごとに分けて特徴シーケンスとする．

4.3 パターン抽出

本手法では，特徴シーケンスを構成するソースコードの特徴のリストに対してリストの後方から前方に向かって PrefixSpan [25] を適用することでコーディングパターンの抽出を行っていく．ソースコードの特徴のリストにおいて，関数呼出し要素を含む制御文要素は，もし存在するとしたら必ずその関数呼出し要素よりも前方に存在している．そのため，ソースコードの特徴のリストに対して後方から前方に向かって探索を行っていくことで，制御文要素の射影を行うときに，今までに抽出された関数呼出し要素を含んでいる制御文要素に対して射影を行うことができる．

具体的には，PrefixSpan の射影の操作を行うときに，射影する要素の種類に応じて以下の操作を行うことでコーディングパターンの抽出を行う．

関数呼出し要素に対して射影を行う場合

射影を行った関数呼出し要素が存在していた行番号の情報を，射影を行った後の特徴シーケンスに加える．ただし，同じ関数呼出し要素が連続するような射影は行わない．

図 8 にその様子を示す．

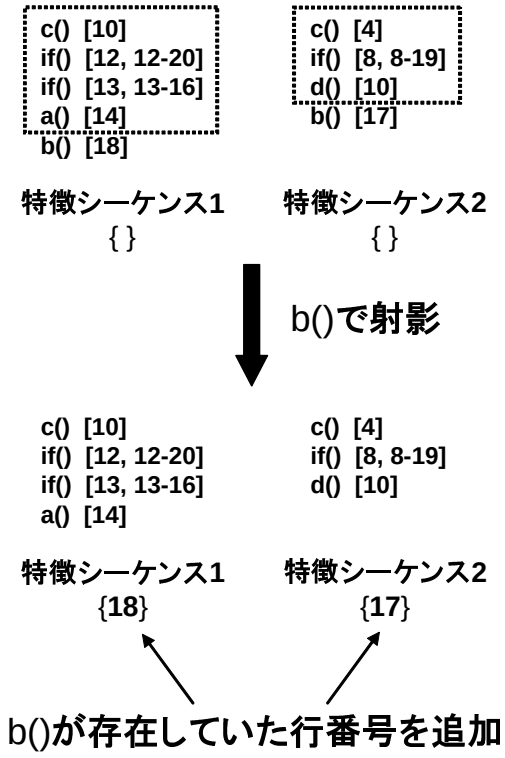

図 8: 関数呼出し要素に対しての射影

制御文要素に対して射影を行う場合

特徴シーケンスに加えられている行番号を含んでいる制御文要素に対してのみ射影を行う．射影を行った制御文要素が存在していた行番号の情報を，射影を行った後の特徴シーケンスに加える．

図 9 にその様子を示す.

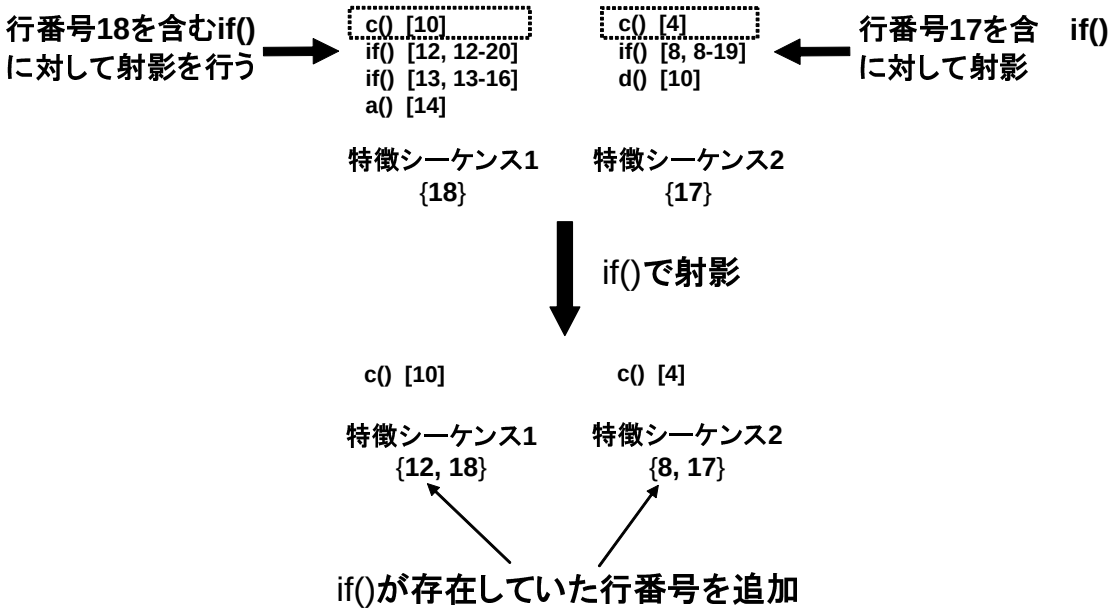

図 9: 制御文要素に対しての射影

全てのパターン抽出が終わった後に，各パターンに対して検査を行い，関数呼出し要素が 1 個以下であるパターンを除去する．

5　実装

本節では，4 節で述べた提案手法を実装したシステムの詳細について述べる．

本システムは大きく 3 つのサブシステムに分けることができる．1 つ目は，4.1 節と 4.2 節で説明した特徴シーケンスの生成を行う部分である．2 つ目は，4.3 節で説明したコーディングパターンの抽出を行う部分である．3 つ目は，抽出したコーディングパターンと，それを実現しているソースコードを閲覧するためのブラウザ部である．システム全体の概要を図 10 に示す．

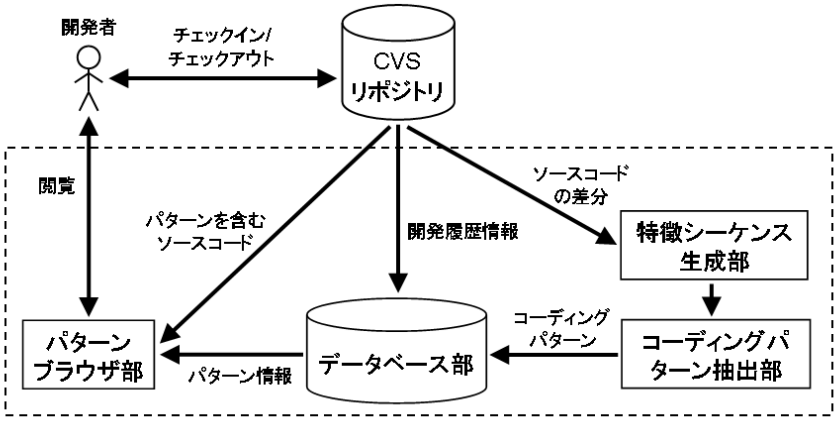

図 10: システムの概要

以下では，本システムの 3 つのサブシステムと，それらが用いるデータの保存と取得を行うデータベース部について説明していく．

なお，本システムの実装に用いた環境は以下の通りである．

- CPU:Pentium4 1.50GHz
- RAM:512MB
- OS:FreeBSD 6.2-RELEASE
- データベース:Postgresql 7.3.15
- Web サーバ:Jakarta Tomcat 5.5.17
- 版管理システム:CVS
- JDK 1.5.0

5.1 データベース部

データベース部は表 1 のように，6 つのデータベースからなる．以下では，各データベースの役割とそれがどのような情報を保持しているかを説明する．

リビジョン DB	チェックイントランザクション DB	変更箇所 DB
ID ファイルパス リビジョン番号 更新日時 更新作業者 ログメッセージ	ID 更新日時 更新作業者 ログメッセージ 所属リビジョンの ID のリスト	ID 変更箇所の行番号のリスト
特徴シーケンス DB	コーディングパターン DB	パターン検索 DB
ID 取得元リビジョンの ID 取得元変更箇所の ID 所属関数定義名 特徴シーケンス	ID パターンが存在した特徴シーケンスの ID パターンが存在した行番号のリスト カテゴリ番号 コーディングパターン	関数名 その関数を含むパターンの ID

表 1: システムが用いるデータベース

リビジョンデータベース :

リビジョンデータベースは，解析対象となるソフトウェアプロジェクトの全てのチェックイン履歴情報を保持するデータベースである．各チェックイン履歴情報には，チェックイン対象のファイルパス，リビジョン番号，更新日時，更新作業者，ログメッセージが含まれている．また，各リビジョン情報はそれらを特定するための ID を持つ．

チェックイントランザクションデータベース :

チェックイントランザクションデータベースは，解析対象となるソフトウェアプロジェクトの全てのチェックイントランザクション情報を保持するデータベースである．チェックイントランザクションとは，一度の更新作業で扱った全てのファイルのチェックイン作業をまとめて表現したものである．各チェックイントランザクション情報には，更新日時，更新作業者，ログメッセージ，更新されたリビジョンの ID のリストが含まれている．また，各チェックイントランザクション情報はそれらを特定するための ID を持つ．

変更箇所データベース：

変更箇所データベースは，特徴シーケンスを生成するときにソースコードの特徴の取得対象となった行の情報を保持するデータベースである．各変更箇所情報には，ソースコードの特徴が取得された行番号のリストが含まれている．また，各変更箇所情報はそれらを特定するための ID を持つ．

特徴シーケンスデータベース：

特徴シーケンスデータベースは，解析対象となるソフトウェアプロジェクトから取得した全ての特徴シーケンスの情報を保持するデータベースである．各特徴シーケンス情報には，取得元となったリビジョンの ID，取得元となった変更箇所の ID，取得元となった関数定義名，そして特徴シーケンス自身が含まれている．また，各特徴シーケンス情報はそれらを特定するための ID を持つ．

特徴シーケンスはソースコードの特徴のリストとなっており，各ソースコードの特徴には，ソースコードの特徴のタイプ，特徴の名前，それが存在した行番号，さらに制御文要素の場合はその開始行番号，終了行番号が含まれている．ソースコードの特徴のタイプには，関数呼出し型と制御文要素型が存在する．ソースコードの特徴の名前は，関数呼出し型の場合はその関数名に，制御文要素型の場合はその制御文要素の名前に相当する．

コーディングパターンデータベース：

コーディングパターンデータベースは，抽出されたコーディングパターンに関する情報を保持するデータベースである．コーディングパターンは，ソースコードの特徴の名前のリストとなっている．各コーディングパターン情報には，パターンが存在した特徴シーケンスの ID，パターンが存在した行番号のリスト，パターンが所属するカテゴリ番号，そしてコーディングパターン自身が含まれている．また，各コーディングパターン情報はそれらを特定するための ID を持つ．

パターン検索データベース：

パターン検索データベースは，コーディングパターンを関数名から検索するために必要な情報を保持するデータベースである．各パターン検索情報には，関数名とその関数呼出しを含んでいるコーディングパターン情報の ID が含まれている．

5.2 特徴シーケンス生成部

特徴シーケンス生成部は，4.1 節と 4.2 節で述べた処理を実現している．

まず始めに，全てのチェックイン情報を取得するために CVS からログを取得し解析する．解析して得られたチェックイン情報は全てリビジョンデータベースに保存される．

そして，取得したチェックイン情報からチェックイントランザクション情報を復元する．チェックイントランザクションの復元には，[24] で述べられている手法を用いている．復元された情報は全てチェックイントランザクションデータベースに保存される．

次に，各ソースコードの差分から特徴シーケンスを生成していく．まずは全てのリビジョン情報について，対象ファイルのリビジョンとその直前のリビジョンとの diff の出力を解析し，新しいリビジョンのどの行が特徴シーケンスの取得対象となるかを計算する．次に，新しいリビジョンのソースコードを解析し，追加された行か否かに関わらず全てのソースコードの特徴を取得する．そして，これらを先に計算しておいた取得対象となる行の情報とすりあわせることで，新しいリビジョンにおいて追加された行に存在するソースコードの特徴のリストを得る．さらに，各関数定義が何行目から何行目までに渡っているのかを計算し，先ほど求めた特徴のリストと合わせることで，各関数定義内に存在するソースコードの特徴のリスト，すなわち，特徴シーケンスを得ることができる．取得した特徴シーケンスの情報は特徴シーケンスデータベースに，特徴シーケンスの取得元となった追加された行についての情報は変更箇所データベースにそれぞれ保存される．このとき，対象となるリビジョンが最新リビジョンだった場合は，初期リビジョンとの annotate の出力を解析することにより，初期リビジョンから変更されていない行を計算し，その行に存在する特徴シーケンスを得る．

この特徴シーケンス生成処理は基本的に全てのリビジョンに対して行うが，マージと考えられるチェックイントランザクションに属するリビジョンに対しては行わない．マージとはブランチに蓄積されたソースコードの変更をトランクへと反映する作業であり，この場合，多くの機能についての変更が一度にチェックインされることが多い．そのため，このチェックインから得られる特徴シーケンスは，個別の機能に関する関数呼出しパターンを抽出するという目的にそぐわない．したがって，マージと考えられるチェックインからは特徴シーケンスを取得しないようにしている．

CVS はどのチェックイントランザクションがマージであるかを保持していないため，本システムではユーザが指定した閾値以上のファイルを更新しているチェックイントランザクションをマージであると判定している．

5.3 コーディングパターン抽出部

コーディングパターン抽出部では，4.3 節で述べた処理を実現している．

PrefixSpan [25] を Java 言語で実装したものを用いて，特徴シーケンスデータベース中の全特徴シーケンスと，ユーザが指定する閾値からコーディングパターンを抽出する．

抽出したパターンは，同じ種類の関数呼出し要素を持つパターン同士を同じカテゴリとしてカテゴリ分けを行う．これにより，同じ関数群が少し違う呼び出し方をされていることを把握しやすくする．パターンを抽出する過程において，そのパターンを抽出するために使用した特徴シーケンスとそのパターンが抽出された行番号が求まっているので，この情報を抽出されたパターンと共にコーディングパターンデータベースへ保存する．また，各関数呼出し要素がどのパターンに存在していたかという情報も計算してパターン検索データベースへと保存する．これらの情報は 5.4 節で述べるブラウザ部によって用いられる．

5.4 パターンブラウザ部

パターンブラウザ部は，抽出されたコーディングパターンとそれを実現しているソースコードを提示することで開発者の部品利用法の理解を支援する．図 11 にパターンブラウザのスクリーンショットを示す．

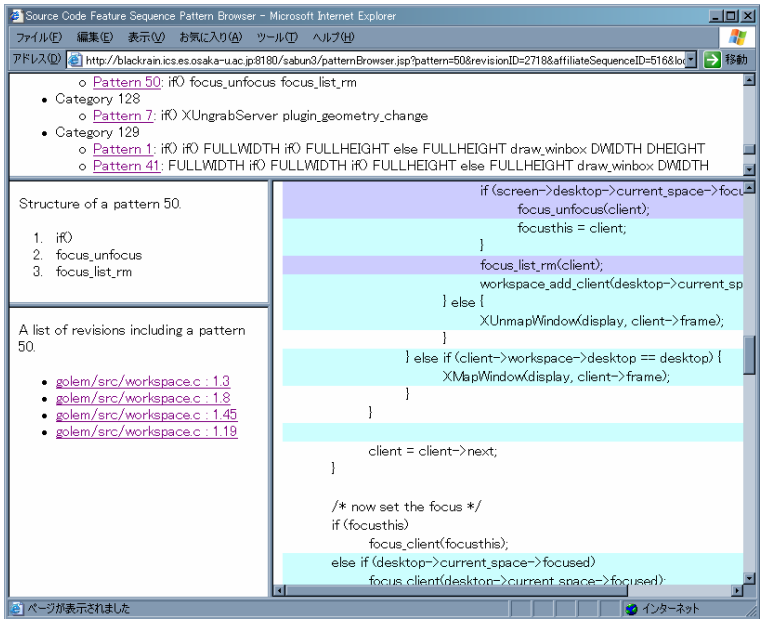

図 11: パターンブラウザ

画面の上部には抽出されたコーディングパターンがカテゴリ別にリストされている．この中から一つのパターンを選択してクリックすると，左下部にコーディングパターンとそれを実現しているリビジョンのリストが表示される．リビジョンのリスト内の一つを選択してクリックすることで，右下部にコーディングパターンを実現しているソースコードが表示される．このソースコードのうち，薄い青色で色付けされている行が新たに追加されたあるいは一度も変更がなされていない行を表しており，濃い青色で色づけされた行は該当するパター

ンが存在する行を表している．

また，パターンリストの上部にあるリンクをクリックすることでパターン検索画面へ移ることが出来る．パターン検索画面では，開発者が利用パターンを学びたい関数名を入力することで，その関数呼出しを含むコーディングパターンを検索することが出来る．検索された各コーディングパターンに張られているリンクをクリックすることで，そのパターンの内容とリビジョンのリストを閲覧することが出来る．図 12 に検索画面のスクリーンショットを示す．

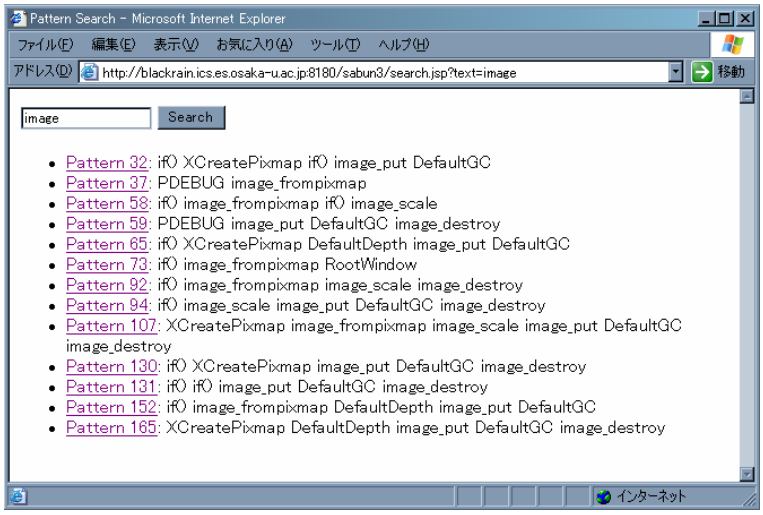

図 12: パターン検索画面

6 評価

本節では，本研究で提案したコーディングパターン抽出手法が既存の差分を用いたコーディングパターン抽出手法の問題点を解決できているかどうかについて評価していく．

評価に際して，コーディングパターンを抽出する対象のソフトウェアプロジェクトとして The Golem X11 Window Manager [27](以下，Golem と表記) を選択した．Golem の概要は表 2 の通りである．

開発期間	2001/3/27 〜 2006/9/14
開発者数	3
総リビジョン数	2812
総チェックイントランザクション数	650
ソースファイル数 (最新版)	196
総行数 (最新版のソースファイルのみ)	71128

表 2: The Golem X11 Window Manager の概要

Golem の CVS リポジトリに対して本手法を用いてコーディングパターンを抽出したところ，抽出されたコーディングパターンの数は 168 個となった．168 個のうち，一度も変更が加えられていない箇所から抽出されたコーディングパターンは 35 個であった．このことから，本手法を用いることで，一度も変更が加えられていない箇所に存在していたコーディングパターンの抽出が出来るようになったといえる．

次に，本手法により，既存の差分を用いたコーディングパターン抽出手法に存在していた問題点が改善されているかどうかについて評価する．この問題点とは，コーディングパターン中に含まれている関数呼出しを実際には含んでいない制御文も同時に抽出されることがあるという点である．

本手法により抽出されたコーディングパターンを全て調べた結果，いずれのパターンにおいても関数呼出しを含んでいない制御文は抽出されていなかった．このことから，本手法が，既存の差分を用いたコーディングパターン抽出手法に存在していた関数呼出しを実際には含んでいない制御文も同時に抽出されることがあるという問題点を改善しているといえる．

しかし，既存の差分を用いたコーディングパターン抽出手法により抽出されたパターンとの比較において本手法の問題点が明らかになった．既存の手法を用いて抽出されたものには存在するが，本手法を用いて抽出されたものには存在しないパターンがいくつか見られた．これは，本手法では構文情報を用いてコーディングパターンの抽出を行っているため，ソースコードの特徴のリストに同じ関数呼出し要素が複数存在した場合，どの関数呼出し要素に

着目するかによって，制御文が抽出されるかどうかが変わってしまうためである．

この問題を解決するためには，コーディングパターンを抽出する際に，ソースコードの特徴のリストの中のどの関数呼出し要素を抽出するかということについても考慮する必要がある．

7　まとめと今後の課題

本研究では，ソフトウェア部品の利用法の理解を支援することを目的にした，版管理システムに蓄積された情報からコーディングパターンを抽出する手法の提案を行った．本研究では，既存の差分を用いたコーディングパターン抽出手法に存在していた一度も変更されていない箇所に存在するコーディングパターンが抽出されないという問題点と，コーディングパターン抽出時に構文を考慮していないという問題点を解決するために，版管理システムに登録されてから一度も変更がなされていない箇所も差分として扱い，さらに，構文情報を用いてのコーディングパターンの抽出を行った．また，提案手法を実装し，実際のオープンソースプロジェクトに対して適用を行い，既存手法の二つの問題点を改善していることを確認した．

今後の課題としては，抽出されたコーディングパターンの定量的な評価，パターン抽出手法のさらなる改良などが挙げられる．

謝辞

本研究において，常に適切な御指導および御助言を賜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 教授に心より深く感謝いたします.

本研究において，常に適切な御指導および御助言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 助教授に心から感謝いたします.

本論文を作成するにあたり，様々な御指導および御助言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 中塚 剛 氏に深く感謝します.

本論文を作成するにあたり，様々な御指導および御助言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 檜皮 祐希 氏に深く感謝します.

最後に，その他様々な御指導，御助言等を頂い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研究室の皆様に深く感謝いたします.

参考文献

[1] 中山 崇. ソースコードの差分を用いた関数呼びだしパターンの抽出手法の提案と実装. 修士論文, 大阪大学 情報科学研究科, 2006.

[2] Victor Basili, Gianluigi Caldiera, Frank McGarry, Rose Pajerski, Gerald Page, and Sharon Waligora. The Software Engineering Laboratory - an Operational Software Experience Factory. In *Proceedings of the Fourteenth International Conference on Software Enigineering*, pp. 370-381, New York, NY, USA, May 1992. ACM Press.

[3] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software reuse: architecture, process and organization for buisiness success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[4] Sadahiro Isoda. Experience Report on Software Reuse Project: Its Structure, Activities, and Stastical Results. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pp. 320-326, May 1992.

[5] Katsuhisa Maruyama and Ken ichi Shima. A Mechanism for Automatically and Dynamically Changing Software Components. In *Proceedings of the 1997 symposium on Software reusability*, pp. 169-180, New York, NY, USA, May 1997. ACM Press.

[6] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in System Code. In *SOSP '01: Proceedings of the 18th ACM symposium on Operating Systems Principles*, pp. 57-72, New York, NY, USA, October 2001. ACM Press.

[7] Reid Holmes and Gail C. Murphy. Using Structural Context to Recommend Source Code Examples. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pp. 117-125, New York, NY, USA, May 2005. ACM Press.

[8] Amir Michail. Data Mining Library Reuse Patterns Using Generalized Association Rules. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pp. 167-176, New York, NY, USA, June 2000. ACM Press.

[9] Chadd C. Williams and Jeffrey K. Hollingsworth. Recovering System Speci.c Rules from Software Repositories. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pp. 7-11, New York, NY, USA, May 2005. ACM Press.

[10] 渥美紀寿, 山本晋一郎, 結縁祥治, 阿草清滋. FCDG に基づいたコーディングパターン. 日本ソフトウェア科学会 コンピューターソフトウェア, Vol. 21, No. 4, pp. 27-36, July 2004.

[11] Ulf Asklund, Lars Bendix, Henrik Bæbak Christensen, and Boris Magnusson. The Unified Extensional Versioning Model. In *SCM-9: Proceedings of the 9th International Symposium on System Configuration Management*, pp. 100-122, September 1999.

[12] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, Vol. 30, No. 2, pp. 232-282, 1998.

[13] Walter F. Tichy. RCS - A System for Version Control. *Software - Practice and Experience*, Vol. 15, No. 7, pp. 637-654, 1985.

[14] Brian Berliner. CVS II: Parallelizing Software Development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pp. 341-352, Berkeley, CA, January 1990. USENIX Association.

[15] Karl Franz Fogel. *Open Source Development with CVS*. Coriolis Group Books, Scottsdale, AZ, USA, 1999.

[16] Rational Software Corporation. Rational clearcase. `http://www.rational.com/products/clearcase/`.

[17] Microsoft Inc. Visual sourcesafe. `http://msdn.microsoft.com/ssafe/`.

[18] Serena Software Inc. Pvcs version control software. `http://www.serena.com/pvcs_version_control_software.html`.

[19] Peter Fröhlich and Wolfgang Nejdl. WebRC: Configuration Management for a Cooperation Tool. In *ICSE '97: Proceedings of the SCM-7 Workshop on System Configuration Management*, pp. 175-185, London, UK, May 1997. Springer-Verlag.

[20] The FreeBSD Project. `http://www.freebsd.org/`.

[21] OpenBSD. `http://www.openbsd.org/`.

[22] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. CVS Release History Data for Detecting Logical Couplings. In *IWPSE: '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, pp. 13-23, Washington, DC, USA, September 2003. IEEE Computer Society.

[23] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pp. 563-572, Washington, DC, USA, May 2004. IEEE Computer Society.

[24] Thomas Zimmermann and Peter Weisgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *MSR 2004: Proceedings of International Workshop on Mining Software Repositories*, pp. 2-6, May 2004.

[25] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, pp. 215-224, Washington, DC, USA, April 2001. IEEE Computer Society.

[26] SourceForge. `http://sourceforge.net/`.

[27] The Golem X11 Window Manager. `http://golem.sourceforge.net/`.