卒 業 研 究 報 告 書

題　目　　複数の版管理システムを対象とした
　　不具合混入モジュール特定アルゴリズムの実装

指導教員　　水野　修　准教授

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　07122502

氏　　名　　川本　公章

平成23年2月15日提出

複数の版管理システムを対象とした
不具合混入モジュール特定アルゴリズムの実装

平成 23 年 2 月 15 日　　07122502　川本　公章

概　　要

ソフトウェア開発の過程で不具合は必ずと言って良いほど含まれてしまい，不具合によって，ソフトウェア開発スケジュールの遅れ，致命的な欠陥や不安定な動作をもつソフトウェアの開発が起きてしまう．したがって，ソフトウェア開発において不具合の予測は，効率的なソフトウェアの開発やソフトウェアの安定した動作のための重要な技術となる．不具合を予測する技術の開発には不具合の情報が必要であり，情報の取得のために不具合を含むモジュールと不具合を含まないモジュールが必要となる．しかし，現在ソフトウェア開発で用いられている開発履歴の管理を行うバージョン管理システムと不具合の管理を行うバグ管理システムには，不具合を含むモジュールを特定する機能が無いため，不具合混入モジュールを特定する技術が必要となる．また，バージョン管理システムには，いくつか種類があるため解析は困難となっている．

本研究では，複数のバージョン管理システムに対応した不具合混入モジュール特定アルゴリズムを開発し，解析結果を統一された形式にすることで，今後の不具合予測の研究に必要となる情報の提供を行う．

不具合混入モジュール特定には，過去に発表された手法を用いる．複数のバージョン管理システムの特徴や使用方法を調べることで，不具合混入モジュール特定の手法へ対応を行う．また，不具合予測の研究に必要となる情報を定め，不具合混入モジュールの特定結果を統一された形式へと変換する．

本研究により，複数のバージョン管理システムを対象とした不具合混入モジュールの特定が可能となり，解析結果を統一された形式で提供することができた．

目　次

1. 緒言

ソフトウェア開発の不具合予測の研究のために，不具合を含むモジュールと不具合を含まないモジュールの情報が必要である．本研究では，バージョン管理システムとバグ管理システムを用いて不具合混入モジュールの特定を行うアルゴリズムに，Śliwerski らが発表した論文 [1][2] に書かれている手法を用いる．この手法は論文の著者の頭文字を並べて SZZ アルゴリズムと呼ばれている．

しかし，バージョン管理システムには複数の種類が存在し，解析が困難となり，解析結果に違いが現れる心配がある．したがって，複数の種類が存在するバージョン管理システムについて，解析に必要となるコマンドを調べ，バージョン管理システムによって違いのある情報の形式を明らかにし，異なるバージョン管理システムの解析を可能にしている．

不具合混入モジュール特定の解析結果は，異なるバージョン管理システムを用いた解析結果でも同じ形式へと変換している．解析結果を SQL を用いたデータベースに格納することで，異なるソフトウェアによる解析結果の利用を容易にしている．

第 2 章では，解析の対象とするバージョン管理システムの特徴や使用方法について説明する．第 3 章では，不具合混入モジュール特定アルゴリズムとして利用する SZZ アルゴリズムについて説明する．第 4 章では，SZZ アルゴリズムを複数のバージョン管理システムに対応させる方法と解析結果について説明する．第 5 章では，オープンソースプロジェクトのソフトウェアに不具合混入モジュール特定の解析を行う．第 6 章では，実験に対する考察を述べる．第 7 章では，本研究のまとめと今後の課題を述べる．

2. バージョン管理システム

バージョン管理システムは，ファイルの変更履歴を管理するシステムで，ファイルの変更ごとに，編集者名，変更内容，変更日時や変更理由など取得しデータベース化している．そのデータベースをリポジトリと呼び，一般的にソフトウェア開発のプロジェクトごとに 1 つのリポジトリを作成しソフトウェア開発の管理を行う．

バージョン管理システムでは，複数のユーザが同一ファイルの変更を行うことも可能となっており，リポジトリはネットワーク上で管理される．このリポジトリの管理方法に集中型と分散型の 2 種類が存在する．

本研究は，集中型の CVS と Subversion，分散型の Git と Mercurial の 4 つのバージョン管理システムを研究の対象としている．

2.1 用語とコマンド

バージョン管理システムで使われている用語とコマンドの一部を説明する．

- チェックアウト（checkout）
  リポジトリからファイルを取り出すことをチェックアウトと呼ぶ．リポジトリに直接編集は行わず，チェックアウトを行ってファイルを作業ディレクトリに複製し，複製したファイルに対して編集を行う．
- アップデート（update）
  リポジトリにチェックアウトしたファイルより新しいファイルがあるか調べ，新しいファイルがある場合は，作業ディレクトリのファイルを新しいファイルに置き換える．
- コミット（commit）
  作業ディレクトリでのファイルの編集をリポジトリに反映させることをコミットと呼ぶ．コミットを行うときにファイルの編集した理由などをコメント文章として記述することができる．
- リビジョン番号（revision）
  ファイルの変更や，コミットを区別するための値のことをリビジョン番号と呼ぶ．リビジョン番号にはコミットごとに値を 1 増やしていく数値やコミットに

対するハッシュ値などが使われる．

- タグ（tag）

  コミットに対してリビジョン番号とは別に付ける名前をタグと呼ぶ．タグを付けることで，リリース用のコミットなどを区別しておくことができる．

- ブランチ（branch）

  作業ディレクトリを複製して，複数の開発を並行して行うことを可能とする機能．

- log

  コミットの履歴やファイルの変更履歴などを出力するコマンド．コミットの時刻やコメント文章などを見る事ができる．

- diff

  ファイルを比較して変更された場所（差分）を出力するコマンド．リビジョンを指定して差分を表示することもできる．

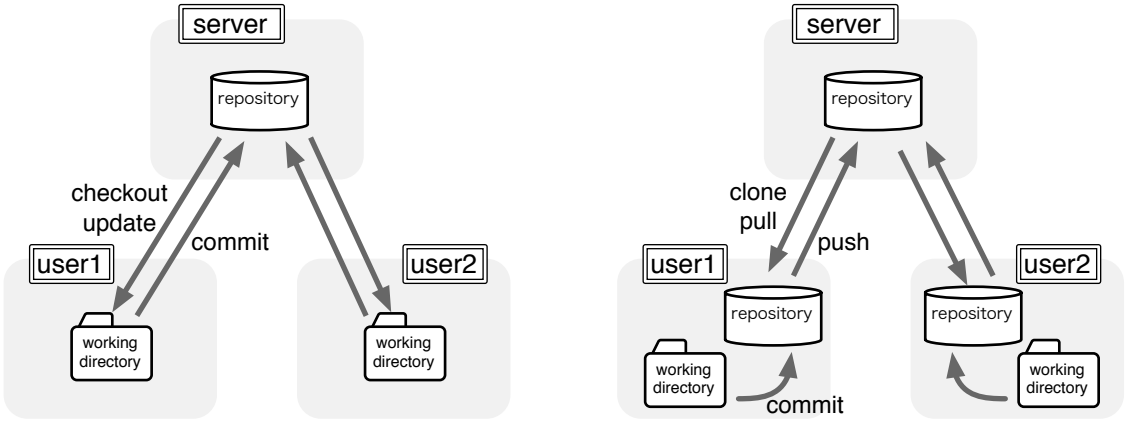
- annotate

  ファイルの各行に対して，変更・追加されたリビジョン番号，編集者名や日時が出力されるコマンド．

2.2 バージョン管理システムの種類

リポジトリの管理方法に違いを持つ，集中型と分散型の動作や特徴を以下に示す．また，図 2.1 に，集中型と分散型のサーバとユーザの関係図を示す．

- 集中型
  - 代表的なツールとして，CVS や Subversion がある．
  - リポジトリはサーバで管理される．
  - ユーザはサーバから必要なファイルをチェックアウトし，ローカル環境でファイルへの編集を行い，編集結果をサーバのリポジトリへコミットする．
  - リポジトリを使った作業を行う場合はサーバと通信できる環境にいる必要がある．
  - クライアント・サーバ型とも呼ばれる．



（a）集中型（CVS）　　（b）分散型 （Git）

図 2.1　集中型・分散型のサーバとユーザの関係

- 分散型
  - 代表的なツールとして，Git，Mercurial，Bazaar がある．
  - リポジトリはサーバにも保存されるが，チェックアウトによってリポジトリがユーザの作業ディレクトリに複製される．この作業を分散型ではクローン（clone）と呼ぶ．
  - ユーザは作業ディレクトリでファイルを編集し，作業ディレクトリにあるリポジトリにコミットを行う．サーバのリポジトリを変更する場合は作業ディレクトリのリポジトリをサーバへ複製する．この作業をプッシュ（push）と呼ぶ．
  - リポジトリが作業ディレクトリにあるので，サーバと通信できない環境でも，リポジトリを使った作業ができる．

集中型に対してリポジトリの解析を行う場合は，速度やネットワーク使用量の観点からサーバのリポジトリをローカル環境へ複製してから解析を行う方が良いと考えられる．

分散型では，チェックアウト（クローン）によってリポジトリがローカル環境に複製されるので，リポジトリの解析をそのまま行う事ができる．

2.3 対象とするバージョン管理システム

研究の対象とする，CVS，Sbuversion，Git，Mercurial のバージョン管理システムについてそれぞれの特徴を説明する．

- CVS
  - CVS（Concurrent Versions System）は 1990 年代から使われている，集中型のバージョン管理システムである．
  - バージョン管理はファイルごとに行われ，リビジョン番号もファイルごとに独立して割り当てられる．“(ファイル名), v” のファイルに差分を保存しファイルを管理している．
  - リビジョン番号はファイルごとに，1.1, 1.2, 2.1 などピリオドで分けられた十進整数を用いる．リビジョンが新しくなると一番右の番号が 1 加算され

る．ブランチのリビジョン番号は 1.2.2.1 などピリオドの数を増やして管理している．

- リポジトリ内でファイルの圧縮を行わないため，リポジトリの容量が大きくなるが，リポジトリの解析を行う場合は高速なアクセスが可能となる．
- ディレクトリの削除ができないことやコミットごとの履歴を見ることができないなど，いくつかの欠点がある．

- Subversion
  - Subversion（SVN）は CVS にある欠点を改善して設計された，集中型のバージョン管理システムである．
  - コミットごとにリビジョン番号が割り当てられ，リビジョンごとに 1 つのファイルに変更情報がまとめられて管理される．
  - リビジョン番号はコミットごとに，r1, r2, r13 など ‘r’ と十進整数を用いる．リビジョンが新しくなると整数値に 1 加算される．

- Git
  - Git は Linus Torvalds によって開発された，分散型のバージョン管理システムである．Linux カーネルの管理にも使われている．
  - コミットごとにリビジョン番号が作られ，commit，tree，blob，tag の 4 つのオブジェクトファイルを使ってファイルを管理している．
  - リビジョン番号はコミットごとに，コミット情報を SHA-1 ハッシュ値に計算してその値を用いる．リビジョン番号は十進整数では無いので，リビジョン番号を見ただけではコミットの順番を知ることができない．
  - commit，tree，blob，tag の 4 つのオブジェクトファイルのファイル名にもそのオブジェクトの値から計算されたハッシュ値が用いられ，ファイルの内容は全て圧縮されている．
  - コマンドの出力フォーマットをコマンドラインオプションで指定することができる．
  - ファイルの移動やファイル名の変更を自動で検出することができる．
  - 管理するファイルが増えると，古くなったいくつかのファイルを 1 つのファイルにパッケージ化して管理する事で，リポジトリの容量を小さくす

ることができる．しかし，リポジトリの解析を行う場合はアクセス速度が遅くなる．

- Mercurial
  - Mercurial は Python で開発された，分散型のバージョン管理システムである．コマンドはすべて hg で始まる．
  - コミットごとにリビジョン番号が割り当てられ，リビジョン番号にはハッシュ値か，ローカルでのみ有効な十進整数が用いられる．
  - コマンドの出力フォーマットをコマンドラインオプションで指定することができる．

3. SZZ アルゴリズム

SZZ アルゴリズムは，バージョン管理システムとバグ管理システムの情報を相互に結びつけることで自動的に不具合混入モジュールを識別するアルゴリズムである．

ソフトウェア開発において，バージョン管理システムとバグ管理システムは重要なツールとなっているが，今のところ，2 つのシステムの情報を相互に結びつけてバージョン管理されているファイルの不具合を見つける処理は 2 つのシステムに含まれていない．SZZ アルゴリズムではその処理を提案している．

基本となるアイデアを示す．

- バグ管理システムから不具合の修正を行っているログを抜き出す．
- 不具合のログとバージョン管理システムを結びつけて，不具合の修正を行ったソースコードの変更を抜き出す．
- 不具合の報告日以前のソースコードの変更を不具合の原因となった変更として決定する．

不具合の報告日以前のソースコードの変更が，その後の不具合の修正の原因の一つと考えられる．したがって，そのソースコードの変更を識別することが不具合混入モジュールを識別することとなる．ここで，そのソースコードの変更を fix-inducing changes と呼ぶこととする．

3.1 必要となるデータ

SZZ アルゴリズムを使って不具合混入モジュールを識別するためには，解析の対象とするソフトウェア開発プロジェクトのすべての変更とすべての修正の情報が必要となる．それらの情報はバージョン管理システムとバグ管理システムから調べる事ができる．

バージョン管理システムは，ファイルの変更ごとに，編集者名，変更内容，変更日時や変更理由などを管理することができる．変更理由はファイルの編集者が変更について記したコメント文章である．バージョン管理システムでは，ファイルの変更に対して revision と呼ばれる変更を識別するための値を与える．バージョン管理システムの代表的なツールとして，CVS や Subversion などがある．

バージョン管理システムで管理されるファイルの変更には，不具合の修正だけではなく新しい機能の追加など，変更にも種類があることがわかる．この変更の種類を判別できるのは，変更理由を記した編集者のコメント文章であるが，この情報だけでは正確な判断が難しいと考えられる．したがって，バグ管理システムの情報が必要となる．

バグ管理システムは，不具合の報告をまとめるシステムで，不具合の報告日，不具合の状況をまとめた文章，不具合の修正日，不具合の修正を行った編集者，不具合の状況や不具合の深刻度などの情報を管理することができる．バグ管理システムでは，不具合に対して他の不具合と識別するために番号を与えている，ここで，その番号を BugID と呼ぶこととする．バグ管理システムの代表的なツールとして，Bugzilla などがある．

3.2 アルゴリズムの流れ

SZZ アルゴリズムの手順を以下に示す．

I. バグ管理システムで使われている BugID に対して，その BugID が記述されているバージョン管理システムの管理しているファイル変更のコメント文章を見つけ出し，そのファイル変更を不具合の修正のための変更とする．

II. 不具合の修正のための変更が，ファイル変更の情報と不具合の情報を使って，構文的と意味的な観点から本当に不具合の修正のための変更なのか判定する．

III. 不具合の修正のための変更を fix change と呼ぶこととし，その revision を r_i とする．ここで fix change の一つ前の revision r_{i-1} と r_i の差分を，バージョン管理システムの diff コマンドを使って取得する．

IV. 差分の情報から r_{i-1} と r_i の間で変更・削除された行の行番号をすべて取得し，それらの行番号を L と置く．

V. バージョン管理システムの annotate コマンドを r_i に対して実行し，行 L が追加，変更された revision をすべて調べ，それらの revision を R とする．

VI. revision R の中から不具合の報告日より前に変更された revision を見つけ出し，その revision を fix-inducing changes とする．

手順 II の構文的と意味的な観点から本当に不具合の修正のための変更なのか判定するために，syntactic level syn と semantic level sem をバージョン管理システムとバグ管理システムの情報から求める．

構文的な判定を行うために，バージョン管理システムのコメント文章から以下のトークンを抜き出す．

- BugID：次の正規表現に当てはまるもの（FLEX の構文で示す）
  - bug[# \t]*[0-9]+
  - pr[# \t]*[0-9]+
  - show_bug\.cgi\?id=[0-9]+
  - \[[0-9]+\]
- 数字：[0-9]+
- キーワード：次の正規表現に当てはまるもの

  fix(e[ds])?|bugs?|defects?|patch
- 単語：文字列 [a-zA-Z]+

コメント文章のトークンが次に示す条件に当てはまった数を syntactic level syn とする．($0 \leq syn \leq 2$)

- BugID を含んでいる
- キーワードを含んでいる，または，BugID か数字だけ

意味的な判定を行うために，バージョン管理システムとバグ管理システムの情報を使う．次に示す条件に当てはまった数を semantic level sem とする．($0 \leq sem \leq 4$)

- 不具合が解決された状態である
- 不具合の状況をまとめた文章にバージョン管理システムのコメント文章が含まれている
- バグ管理システムの不具合の修正を行った編集者とバージョン管理システムのファイル変更の編集者が同じ

- 変更されたファイル名が不具合の状況をまとめた文章に含まれている

求めることができた syn と sem の値が式 (3.1) に当てはまるとき，構文的と意味的な観点から本当に不具合の修正のための変更とする．

$$sem > 1 \vee (sem = 1 \wedge syn > 0) \tag{3.1}$$

3.3 アルゴリズムの動作例

図 3.1 にバージョン管理システムとバグ管理システムの情報の例を示す．バージョン管理システムには CVS を，バグ管理システムには Bugzilla の情報を使っている．

実線で下線の引かれた 123 の数字が BugID となっており，ファイル foo.java の revision 1.9 のコメントに数字 123 が書かれているので，ファイル foo.java の revision 1.9 を不具合の修正のための変更とする．

図 3.2 に，今回示したの例の流れを示している．

最初に，syntactic level と semantic level を使った判定を行う．syntactic level の判定では図 3.1 の破線の引かれた部分の情報を使い，図 3.1 では，BugID とキーワードが含まれているので $syn = 2$ となる．semantic level の判定では図 3.1 の点線の引かれた部分の情報を使い，不具合が解決された状態で，編集者が同じで，ファイル名が文章に含まれているので $sem = 3$ となる．これは式 (3.1) に当てはまるので，構文的と意味的な観点から本当に不具合の修正のための変更となり，ファイル foo.java の revision 1.9 は fix change となる．

次に，fix change の一つ前の revision と fix change の差分を取得する．CVS では次のコマンドで取得できる．

```
$ cvs diff -r 1.18 -r 1.19 foo.java
```

コマンドの出力結果から，変更・削除された行の行番号をすべて取得する．ここでは，10,11,15,17,20 行目が変更・削除された行とする．$L = \{10, 11, 15, 17, 20\}$

次に，fix change の一つ前の revision に対して annotate を行い，各行がどの revision で追加・変更されたか調べる．CVS では次のコマンドで取得できる．

```
$ cvs annotate -r 1.18 foo.java
```

コマンドの出力結果から，行番号 L の追加・変更された revision をすべて取得する．ここでは，$(10, 1.14), (11, 1.14), (15, 1.17), (17, 1.16), (20, 1.11)$ の revision が取得できたとする．$(10, 1.14)$ は 10 行目が revision 1.14 で追加・変更されたことを表す．同じ revision をまとめると，$R = \{1.11, 1.14, 1.16, 1.17\}$ となり，revision R が，ファイル foo.java の revision 1.18 と 1.19 の間で変更・削除された行を追加・変更した revision となる．

最後に，revision R の変更日をすべて調べ，そのなかで不具合の報告日より前の revision を fix-inducing changes とする．ここでは，revision R の変更日が，1.11:2010-12-28，1.14:2011-01-13，1.16:2011-01-18，1.17:2011-01-19 とすると，不具合の報告日は 2011-01-15 なので，revision 1.11 と 1.14 が fix-inducing changes となる．

バージョン管理システム（CVS）

file: foo.java
revision: 1.9 date: 2011-01-23 author: kimiaki
message: bug #123 fixed.

バグ管理システム（Bugzilla）

bug_id: 123
resolution: FIXED opendate: 2011-01-15 assigned_to: kimiaki
short_desc: foo.java function Bar() error

図 3.1　バージョン管理システムとバグ管理システムの例

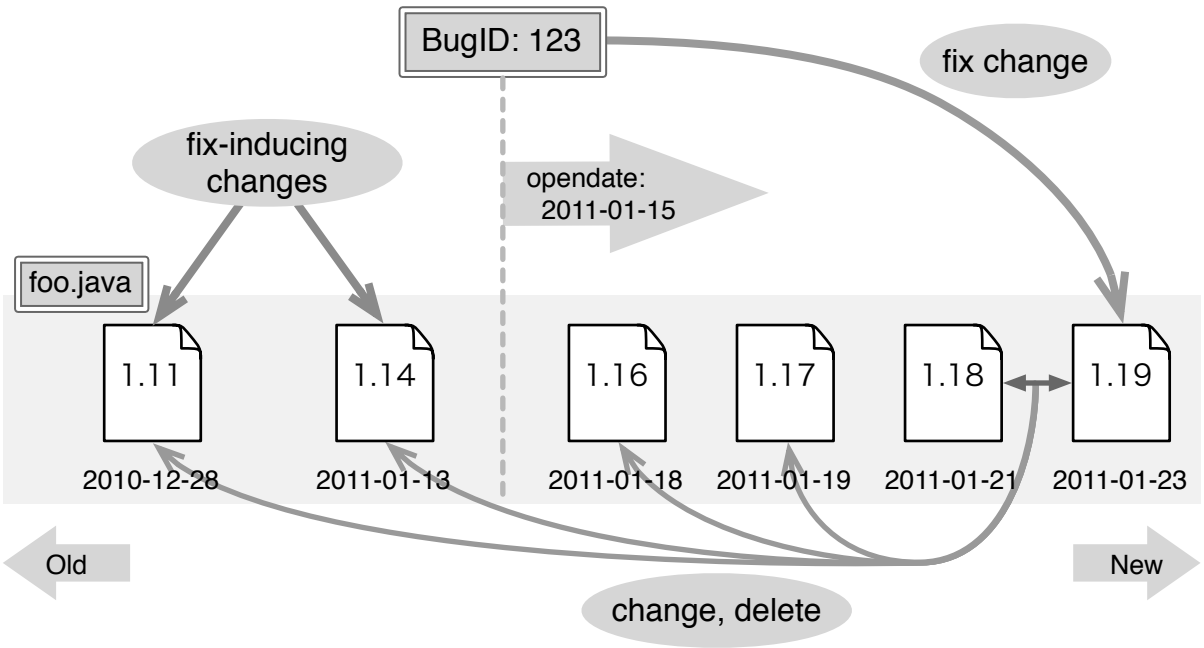


図 3.2　fix-inducing changes の例

4. アルゴリズムの実装

本アルゴリズムは，SZZ アルゴリズムを用いてリポジトリから fix-inducing changes を調べ，fix-inducing change から fix change の間にある不具合を含むモジュールを特定することである．複数のバージョン管理システムに対応したアルゴリズムを実装するためのアイデアを示す．

- アルゴリズムの中で，バージョン管理システムの情報が必要になる箇所を明確にし，バージョン管理システムの違いによるアルゴリズムの変更箇所を最小限にする．
- バージョン管理システムの出力をできるだけ統一された形式へ変換し，その後の解析処理の差を無くす．

この 2 つを徹底することで，新たなバージョン管理システムへの対応を簡単にすることができた．

4.1 アルゴリズムの流れ

実装したアルゴリズムの手順を以下に示す．

i. リポジトリのチェックアウト

ii. バージョン管理システムのファイル変更履歴の取得

iii. バグ管理システムのデータベースの読み込み

iv. fix change の探索

v. fix change の検証

vi. fix-inducing changes の探索と faulty modules の特定

vii. fix-inducing changes と faulty modules の整理

viii. 解析結果の保存

ix. ソースコードの取得と保存

手順 vi の fix-inducing changes の探索と faulty modules の特定では，fix change ごとに次に示す手順で探索を行っている．

iv-i. fix change のリビジョン番号 r_i のひとつ前のリビジョン番号 r_{i-1} を調べる．

iv-ii. diff コマンドを用いて，r_i から r_{i-1} の間で変更・削除された行番号 L を調べる．

iv-iii. annotate コマンドを用いて，r_{i-1} の行 L が追加・変更されたリビジョン番号 R を調べる．

iv-iv. 不具合の報告日より新しいリビジョンのリビジョン番号と行番号を R と L の中から削除する．

iv-v. リビジョン番号 R の中で最も編集日時の新しいリビジョンを fix-inducing change とし，リビジョン番号 r_{fixin} とする．

iv-vi. r_{fixin} から r_{i-1} までのリビジョンを faulty modules とし，r_{faulty} とする．

iv-vii. r_{faulty} と r_{fixin} と L をファイル名と共に保存する．

図 4.1 に，本アルゴリズムを 3.2 で示した例に対して適用した場合を示す．

本アルゴリズムでは，fix-induing changes の中で一番新しいリビジョンが，最も不具合に関連していると考え，fix-inducing change は fix change に対して 1 つだけにしている．

手順 vii の fix-inducing changes と faulty modules の整理では，fix change ごとに調べた r_{faulty} と r_{fixin} と L を，ファイルごとに集めてソートなどの整理を行う．

3.2 で示した SZZ アルゴリズムの手順との対応は次のようになる．

手順 iv の fix change の探索は，SZZ アルゴリズムの手順 I に対応する．

手順 v の fix change の検証は，SZZ アルゴリズムの手順 II に対応する．

手順 vi の fix-inducing changes の探索と faulty modules の特定は，SZZ アルゴリズムの手順 III,IV,V,VI に対応する．

4.2 バージョン管理システムへの対応

不具合モジュールの解析には，バージョン管理システムのリポジトリの情報がいくつか必要となる．

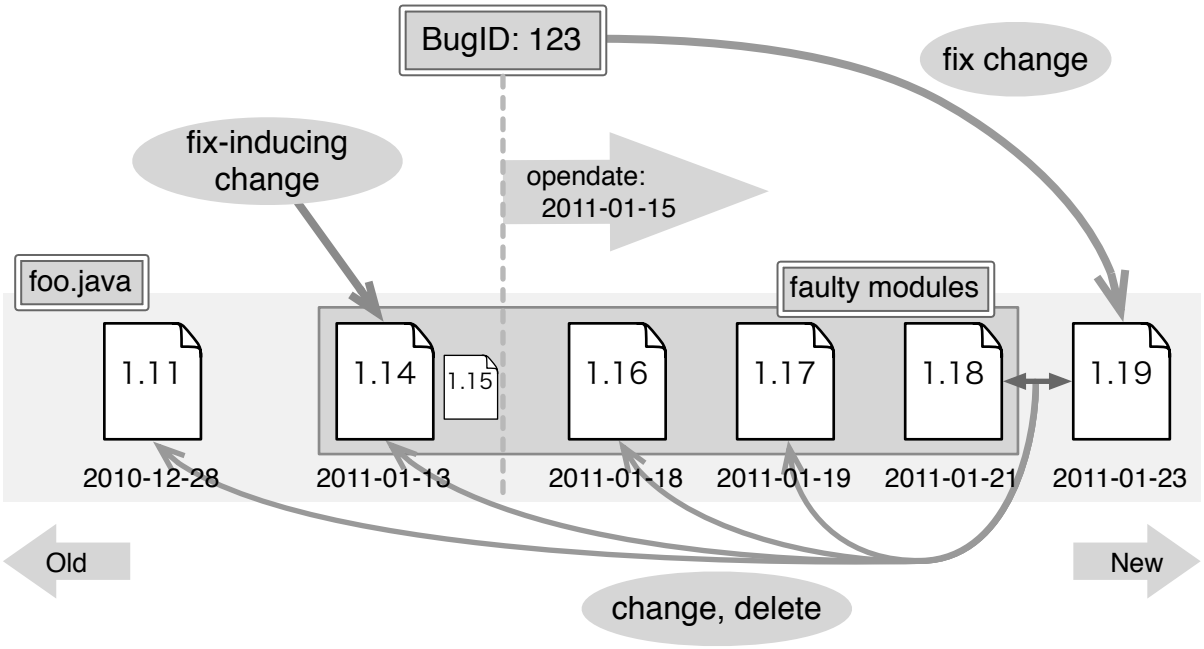


図 4.1　不具合混入モジュールの特定

4.1 で示したアルゴリズムの手順ごとに必要となるバージョン管理システムのコマンドを以下に示す.

i. リポジトリのチェックアウト

(CVS) checkout コマンドを使用. モジュールの指定が必要.

(Subversion) checkout コマンドを使用.

(Git) クローンしたリポジトリの解析なので必要ない.

(Mercurial) クローンしたリポジトリの解析なので必要ない.

ii. バージョン管理システムのファイル変更履歴の取得

(CVS) log コマンドを使用.

(Subversion) log コマンドを使用.

(Git) log コマンドを使用. 出力形式の指定.

(Mercurial) log コマンドを使用. 出力形式の指定.

iii. バグ管理システムのデータベースの読み込み

バージョン管理システムを使わない.

iv. fix change の探索

バージョン管理システムを使わない.

v. fix change の検証

バージョン管理システムを使わない.

vi. fix-inducing changes の探索

(CVS) diff コマンドと annotate コマンドを使用.

(Subversion) diff コマンドと annotate コマンドを使用.

(Git) diff コマンドと annotate コマンドを使用.

(Mercurial) diff コマンドと annotate コマンドを使用.

vii. fix-inducing changes の整理

バージョン管理システムを使わない.

viii. 解析結果の保存

バージョン管理システムを使わない．

ix. ソースコードの取得と保存

(CVS) update コマンドを使用．

(Subversion) cat コマンドを使用．

(Git) ls-tree コマンドと show コマンドを使用．

(Mercurial) cat コマンドを使用．

集中型のバージョン管理システムである CVS と Subversion では，リポジトリの解析を行うためにはチェックアウトを行い，作業ディレクトリを作る必要がある．分散型の Git と Mercurial では，リポジトリ全体を clone コマンドで取得でき，その場合は作業ディレクトリが作られるので，集中型とちがいチェックアウトの作業を行う必要がない．

ファイル変更履歴の取得では，4 つのシステムとも同じ log というコマンドで履歴の出力を行う事ができる．履歴の取得は，ファイル名を指定して，ファイルごとに編集履歴を取得し，リビジョンごとに分割している．この時取得する情報は，リビジョン番号，ファイル編集日時，ファイル編集者名，コミットメッセージの 4 つである．出力される形式は，Git や Mercurial のように出力形式を指定できる場合もあるが，システムごとに異なる形式となるので，それぞれに適したテキスト処理が必要となる．

リビジョン番号は，システムごとに異なる形式となっているが，これらを出力された形式のまま文字列として扱うことで，形式の差を気にすることなくその後の解析を行うことができる．diff コマンドや annotate コマンドなどでリビジョンを指定する場合に，リビジョン番号を文字列として扱っていても問題はない．

ソースコードの取得では，システムごとに異なったコマンドが必要となり，Git の場合は，ファイル名とリビジョン番号を指定してファイル内容を出力するコマンドが無く，ls-tree コマンドを用いてオブジェクトのファイル名（ハッシュ値）を調べてから，ファイル内容を取得する必要がある．

4.3 解析結果の統一

解析結果を SQL を使ったデータベースに格納することで，異なるアプリケーションからの利用を容易にできるようにしている．

また，解析に用いた情報や解析対象の内容（ソースコード）をデータベースに格納しておくことで，今後の研究において，解析結果のデータベースを参照するだけで様々な研究が可能になると考えられる．

データベース管理システムには SQLite3 を使用している．データ型には整数型（INTEGER），文字列型（TEXT），入力をそのまま格納する型（BLOB）があり，整数値は INTEGER，長さの短い文字列は TEXT，ソースコードには BLOB を用いている．日付は ISO-8601 形式（YYYY-MM-DDTHH:MM）の日付表現を用いている．

データベースは以下に示す 5 つのテーブルを持っている．また，図 4.2 にデータベースのスキーマを示す．

- TABLE log

  アルゴリズムの手順 ii で取得したバージョン管理システムのファイル編集履歴を，各ファイルのリビジョンごとに格納する．以下に示す 6 つのフィールドを持っている．

  - id INTEGER：リビジョンの識別番号
  - file TEXT：リポジトリ内のファイルパス
  - revision TEXT：リビジョン番号
  - date TEXT：ファイル編集日時
  - author TEXT：ファイル編集者名
  - message TEXT：コミットメッセージ

- TABLE bugdb

  アルゴリズムの手順 iii で読み込んだバグ管理システムのデータベースの情報を格納する．以下に示す 6 つのフィールドを持っている．

  - bug_id INTEGER：バグ管理システムで使用している BugID
  - opendate TEXT：不具合の報告日
  - assigned_to TEXT：不具合の報告者名

TABLE log

id INTEGER	file TEXT	revision TEXT	author TEXT	date TEXT	message TEXT

TABLE bugdb

bug_id INTEGER	opendate TEXT	assigned_to TEXT	resolution TEXT	short_desc TEXT	bug_severity TEXT

TABLE buglink

file TEXT	revision TEXT	date TEXT	id INTEGER	bug_id INTEGER

TABLE faulty

id INTEGER	file TEXT	revision TEXT	date TEXT	faulty INTEGER	fixin INTEGER	line TEXT

TABLE code

id INTEGER	code BLOB

図 4.2　解析結果データベースのスキーマ

- resolution TEXT：不具合の状況
- short_desc TEXT：不具合の状況説明文
- bug_severity TEXT：不具合の深刻度

- TABLE buglink

  アルゴリズムの手順 v で検証した fix change の情報を格納している．以下に示す 5 つのフィールドを持っている．

  - file TEXT：リポジトリ内のファイルパス
  - revision TEXT：リビジョン番号
  - opendate TEXT：不具合の報告日
  - id INTEGER：リビジョンの識別番号
  - bug_id INTEGER：不具合の BugID

- TABLE faulty

  アルゴリズムの手順 vii で整理した fix-inducing changes の情報を格納している．以下に示す 7 つのフィールドを持っている．

  - id INTEGER：リビジョンの識別番号
  - file TEXT：リポジトリ内のファイルパス
  - revision TEXT：リビジョン番号
  - date TEXT：ファイル編集日時
  - faulty INTEGER：faulty の場合 1，そうでない場合 0
  - fixin INTEGER：fix-inducing change の場合 1，そうでない場合 0
  - line TEXT：不具合の原因となった行番号

- TABLE code

  アルゴリズムの手順 ix で取得したソースコードを格納している．以下に示す 2 つのフィールドを持っている．

  - id INTEGER：リビジョンの識別番号
  - code BLOB：ソースコード

リビジョンの識別番号とは，ファイル編集履歴を取得するときに，リビジョンごとに取得した順につける 1 から始まる十進整数である．テーブル code にはファイル名

などのフィールドが無いが，リビジョンの識別番号 id を用いて，他のテーブルから取得することができる．

ファイルの識別には，リポジトリ内のファイルパスを用いるため，同名のファイルが存在しても問題にならない．

異なるバージョン管理システムによる解析結果の違いは，リビジョン番号の形式にしか現れず，他の情報は全て統一化されている．

5. 適用実験

本研究で実装したアルゴリズムを用いて，オープンソースプロジェクトのソフトウェアの不具合混入モジュールの解析を行う．

1 つ目の実験として，大規模なソフトウェアに対して解析を行い，不具合混入モジュールの特定ができることを確認し，不具合の報告数に対する不具合混入モジュールの特定できた数の比較を行う．

2 つ目の実験は，1 つのソフトウェアを異なるバージョン管理システムに変換し，同じソフトウェアを異なるバージョン管理システムを用いて解析した場合の解析結果の比較を行う．

5.1 実験対象

実験 1：不具合報告に対して不具合混入モジュールの特定できた数の比較

解析の対象とするオープンソースプロジェクトのソフトウェアは，統合開発環境の Eclipse と Netbeans，ウェブブラウザの Chrome と Firefox の 4 つのソフトウェアである．

Eclipse のバージョン管理システムには CVS，Netbeans には Mercurial，Firefox には CVS，Chrome には Subversion を用いて開発が行われている．

統合開発環境の Eclipse と Netbeans は，ともに Java 言語を用いて開発されており，不具合混入モジュールの解析対象とするファイルは java のソースコードとする．

ウェブブラウザの Chrome と Firefox は，複数の開発言語を用いて開発されているが，Chrome の解析対象とするファイルは c,cc,cpp,h,m,mm,py のソースコードとし，Firefox の解析対象とするファイルは c,cpp,h,java のソースコードとする．

実験 2：バージョン管理システムの違いによる解析結果の比較

解析の対象とするオープンソースプロジェクトのソフトウェアは，Eclipse のプラグインの Eclipse BIRT というビジネス文書作成プラグインである．

Eclipse BIRT のバージョン管理システムには CVS が用いられており，CVS リポジトリを他のバージョン管理システムに変換し解析を行う．

Subversion リポジトリへの変換には cvs2svn というツールを，Git リポジトリへの

変換には git cvsimport コマンドを，Mercurial への変換には hg convert コマンドを用いた．

Eclipse BIRT は，Java 言語を用いて開発されており，不具合混入モジュールの解析対象とするファイルは java のソースコードとする．

5.2 実験方法

実験 1：不具合報告に対して不具合混入モジュールの特定できた数の比較

解析に用いた不具合の BugID の中で，4.1 の手順 v において fix change の特定に使われた BugID の割合を BugID の使用率とする．

$$BugID\text{の使用率} = \frac{fix\ change\text{の特定に使われた}\ BugID\text{の数}}{\text{解析に用いた不具合の}\ BugID\text{の数}} \tag{5.1}$$

BugID の使用率をそれぞれのソフトウェアの解析結果について求め比較を行う．

実験 2：バージョン管理システムの違いによる解析結果の比較

解析結果から，ファイル数，リビジョン数，BugID 数，fix change を見つけた数（検証前 BugLink，検証後 BugLink），fix change の特定に使われた BugID の割合（linkBugID），fix-inducing change と判定したモジュールの数（fixin），faulty module と判定した数を求め比較を行う．

5.3 実験結果

実験 1：不具合報告に対して不具合混入モジュールの特定できた数の比較

Eclipse，Netbeans，Chrome，Firefox の解析結果を表 5.1 に示す．

BugID の使用率は，最も高くて Netbeans の 18.6%で，不具合報告全体の 2 割も使用出来ていない．Firefox では 0.3%という非常に低い値になっている．

実験 2：バージョン管理システムの違いによる解析結果の比較

Eclipse BIRT を各バージョン管理システムに変換し，解析した結果を表 5.2 に示す．

同じソフトウェアの解析であるが，不具合混入モジュールの特定できた数に違いあるなど，バージョン管理システムの違いが解析結果へ影響を与えていることがわかる．

表 5.1　実験 1 の解析結果（BugID の使用率）

	Eclipse(cvs)	Netbeans(hg)	Chrome(svn)	Firefox(cvs)
file	24426	53359	13113	3988
revision	308966	393531	131059	47376
BugID	254068	184767	36892	457277
BugLink	171637	115472	51491	3246
linkBugID	25098	34286	5593	1328
BugID 使用率	9.9%	18.6%	15.2%	0.3%
fixin	36357	57337	14020	1795
faulty	159818	165560	45490	17902

表 5.2　実験 2 の解析結果

	CVS	Subversion	Git	Mercurial
file	8973	8973	8974	8973
revision	70142	69513	69762	75534
BugID	21372	21372	21372	21372
検証前 BugLink	30510	30410	30507	32954
検証後 BugLink	30048	29950	30045	32487
linkBugID	6267	6260	6267	6357
fixin	10178	10176	10057	12655
faulty	32508	32474	32478	34947

6. 考察

実験結果の妥当性について以下に示す.

プログラムのバグ：不具合混入モジュール特定のために実装したプログラム自体に不具合が含まれている可能性もあり，特定した不具合混入モジュールが，本当は不具合を含んでいないことが考えられる.

アルゴリズム自体の不備：バージョン管理システムとバグ管理システムの情報を利用した不具合混入モジュール特定のアルゴリズムの手順に不備がある可能性もあり，特定した不具合混入モジュールが，本当は不具合を含んでいないことが考えられる.

バージョン管理システムの本質的問題：バージョン管理システムの履歴管理に不備がある可能性もあり，不具合混入モジュール特定が正しく行えないことが考えられる.

開発者に起因する問題：バージョン管理システムとバグ管理システムへの情報の入力は開発者が行っているので，間違った情報を入力する可能性もあり，不具合混入モジュール特定が正しく行えないことが考えられる.

以上の実験結果の妥当性について注意し，実験結果に対する考察を示す.

実験 1 の結果である表 5.1 に示した BugID の使用率は，アルゴリズムが不具合報告にたいして不具合混入モジュールを特定できる割合となっており，この値が大きいほど不具合混入モジュール特定が上手くできていると考えられる.

BugID の使用率が低い原因として，BugID を使ったバージョン管理システムとバグ管理システムのリンクでは，バージョン管理システムのコメント文に記述されている BugID のみを頼りとしているため，コメント文に正しく BugID が記述されなかったり，BugID の記述が無い場合は，BugID のリンクを出来なくなり，BugID の使用率が低くなっていると考えられる.

また，本実験では，解析の対象とするファイルをソフトウェアごとに制限しており，解析の対象外となっているファイルに対する不具合報告では，BugID を使う事ができないので，BugID の使用率が低くなる原因の一つと考えられる．特に，Firefox の解析では，対象とするファイル数に対する BugID の数がとても多く，解析対象外のファイルに対する BugID が多かったと考えられる.

実験 2 の解析結果をグラフ化したものを図 6.1 に示す.

図 6.1 では，Mercurial に変換した後の解析が他のバージョン管理システムを使った解析に比べて，不具合混入モジュールを多く特定できていることがわかり，バージョン管理システムの違いが解析結果に大きく影響を与えることがわかる．

バージョン管理システムが解析結果に影響を与える原因としては，バージョン管理システムの履歴管理方法に違いがあり，解析途中で取得したファイルの編集履歴に違いが表れて，最終的に不具合混入モジュールの特定に違いが表れたと考えられる．

また，実験 2 では，CVS リポジトリを Subversion，Git，Mercurial にツールを用いて変換しているため，他のバージョン管理システムに変換する過程で正しく変換が行われず，他のバージョン管理システムで取得できるファイルの編集履歴などに違いが生まれたのが原因にあると考えられる．

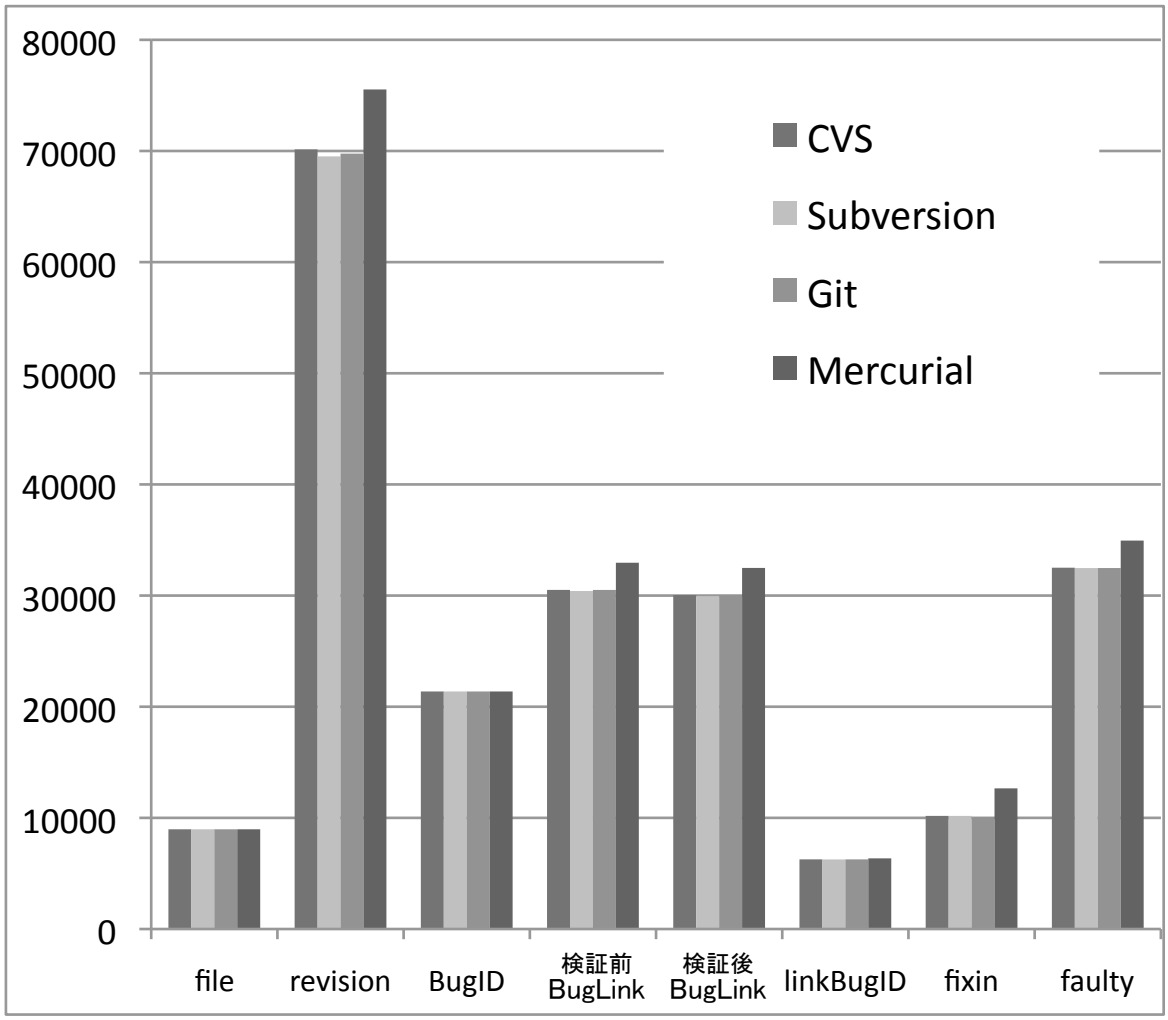


図 6.1　実験 2 の解析結果

7. 結言

本研究では，不具合混入モジュール特定アルゴリズムの SZZ アルゴリズムを複数のバージョン管理システムに対応する手法を提案し，解析結果を統一した形式に変換し提供することを可能にした．これにより，今後のソフトウェア開発の不具合予測の研究に役立てる事が出きると考えられる．

今後の課題として，BugID の使用率を高めるために，不具合混入モジュール特定アルゴリズムの改善が必要と考えられる．SZZ アルゴリズムの改善手法 [3] も提案されているのでそれも参考にしたい．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学部門水野修准教授に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻平田幸直先輩をはじめとする，ソフトウェア工学研究室の皆さん，学生生活を通じて筆者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?,” Proceedings of the 2005 international workshop on Mining software repositories, pp.1–5, ACM, New York, NY, USA, 0 2005. St. Louis, Missouri.

[2] T. Zimmermann and P. Weißgerber, “Preprocessing cvs data for fine-grained analysis,” Proceedings of the First International Workshop on Mining Software Repositories, pp.2–6, May 2004. Edinburgh, United Kingdom.

[3] S. Kim, T. Zimmermann, K. Pan, and E.J.J. Whitehead, “Automatic identification of bug-introducing changes,” Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, pp.81–90, IEEE Computer Society, Washington, DC, USA, 0 2006.