履歴情報を用いたソースコードの変更危険度計測手法の提案

村尾 憲治[†]　松下 誠[†]　井上 克郎[†]

[†] 大阪大学 大学院情報科学研究科
〒 560-8531 大阪府豊中市待兼山町 1-3

近年のソフトウェアの大規模化，複雑化により，ソフトウェアの全体像を把握することは困難になってきている．そのため，ソフトウェアに対して不具合の修正や新機能の追加等の変更を行う際，その変更により新たな不具合が発生することがある．しかし，このような不具合の発生する可能性を事前に知るにはソフトウェアに対する十分な理解が必要であり容易ではない．本稿ではソフトウェアに対して変更を行う際に注意を必要とする箇所を知るため，変更危険度という「変更によって問題が発生する可能性の大きさ」を示すメトリクスを定義し，ソフトウェアのソースコードに対して変更危険度を計測する手法の提案を行う．

Change Risk Measurement Using Revision History for Source Code

Murao Kenji[†]　Makoto Matsushita[†]　Katsuro Inoue[†]

[†] Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan

As a software system evolves, it becomes difficult to understand. Therefore, programmers sometimes make changes which lead to problem. It is very difficult to know the possibility that such problem occurs unless they understand the software system well enough. In this study, we define the metrics ”change risk” - the possibility that some problems occur because programmers change the source code - and propose the method of measuring the ”change risk” in order to know the location we need to be careful when we change the source code.

1 まえがき

オープンソースソフトウェアの開発規模の増大に伴い，その開発形態は多人数化，分散化している．大規模なオープンソースソフトウェア開発では，複数の開発者が互いにソースコードを共有しながら同時に一つの開発作業に携わることが一般的になりつつある．開発過程で新規の開発者が参加することも珍しくないが，特にこのような新規の開発者にとってソフトウェアの全体像を把握することは困難になってきている．

また，開発過程で機能の追加，拡張あるいは不具合の修正等の必要から，開発者はしばしばソースコードに何らかの変更を施す．その際，本来変更すべきでない機能を変更したり，変更によって新たな不具合を引き起こしてしまうことがある．ソースコードに対してこのような問題を引き起こす変更を行ってしまう可能性を事前に知ることはソフトウェアに対する十分な理解を必要とする．ソフトウェアの全体像の把握が困難な新規開発者にとって，これは大きな問題となる．

一方，ソフトウェアを効率よく管理する為に，近年の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ことが多くなっている．版管理システムではソフトウェアに対する変更の履歴が全て個別のアーカイブに保存されている．このアーカイブを閲覧することによって，ソフトウェアの開発過程についてより深い理解が得られることが知られている [1]．従って，この版管理システムに蓄積された変更の履歴を解析することで，ソフトウェアに対して変更を行う際に注意を必要とする箇所の特定が可能であると期待できる．

本研究では版管理システムを利用し，変更によって問題が発生する可能性を事前に把握する為の手法を提案する．具体的には，リポジトリに蓄積されたソフトウェアの開発履歴を解析し，ソースコードの行毎の「変更危険度」を計測する．変更危険度とは「変更によって問題が発生する可能性の大きさ」を表すメトリクスである．変更危険度を示すことにより，ソースコードの変更を行う際，開発者の注意が喚起できると期待される．

以降，2 節ではオープンソースソフトウェア開発とその開発環境及び問題点について述べ，3 節では本稿で提案する変更危険度の計測手法について説明する．4 節では手法の実装について述べ，5 節では評価実験について述べる．最後に 6 節で本研究のまとめと今後の課題について述べる．

2 オープンソースソフトウェア開発

本節では，オープンソースソフトウェアの開発環境，特に版管理システム CVS について述べ，オープンソースソフトウェアの持つ問題点を説明する．

2.1 オープンソースソフトウェア開発環境

オープンソースソフトウェア開発では，各開発者がそれぞれ分散して並列的に開発作業を行うことが一般的である．一方，開発中のソースコードやドキュメント等のプロダクトを広く公開するため，それらの管理を行う必要がある．そこで，開発者はオープンソースソフトウェア開発環境と呼ばれる環境を用いてプロダクトの管理を行う．

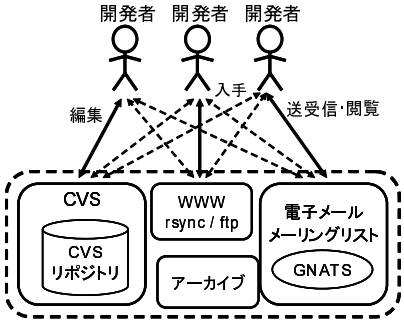

図 1: オープンソースソフトウェア開発環境の例

オープンソースソフトウェア開発環境の構成例を図 1 に示す．オープンソースソフトウェア開発環境は，一般に複数のシステムから構成される．図 1 の構成例の場合，ソースコードやドキュメント等のプロダクトは，版管理システム [2] の一つである Concurrent Versions System(CVS)[3] [4] を用いて管理される．それらのプロダクトは，rsync や ftp を利用して，各開発者に複製，配布される．また，開発者間で相互に行われる意志疎通の手段として，電子メールやメーリングリストが用いられる．その内容はアーカイブとして保存され，World Wide Web(WWW) を用いて自由に検索や閲覧が可能である．開発者からのバグ報告等フィードバックは，GNU Problem Report Management System(GNATS) を用いたバグデータベースによって管理される．

以下では，これらのシステムの中から，オープンソースソフトウェア開発で用いられる版管理システム CVS について説明する．

2.2 版管理システム CVS

CVS は 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，近年最も良く使われるシステムの１つである．

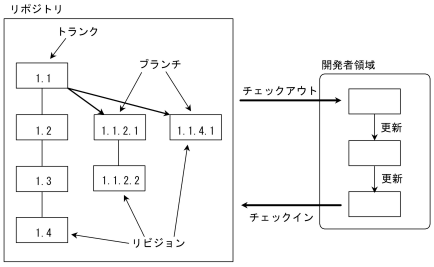

図 2: CVS を用いた作業の例

CVS では，プロジェクトの開発履歴をリポジトリというデータベースにリビジョンの列として格納する．このリビジョンとはプロジェクトのある時点でのスナップショットを表すものである．図 2 に示すように，開発者はチェックイン，チェックアウトという作業を行って，リポジトリから特定のリビジョンを取得したり，更新内容をリポジトリに反映したりする．また，あるリビジョンからバグ修正などの為に新たにリビジョン列を派生させることもでき，この派生したリビジョン列はブランチと呼ばれる．ブランチに対し，元のリビジョン列をトランクという．

2.3 問題点

オープンソースソフトウェア開発に携わる開発者はソフトウェアの理解に費やす時間が限られている．さらにソフトウェアが複雑化するにつれて理解に必要な情報量は増大するため，新規に参加した開発者であるほど開発の全体像を把握しにくくなる傾向がある．

また，開発過程で機能の追加・変更あるいは不具合の修正等の必要から，開発者はしばしばソースコードに何らかの変更を施す．その際，変更によって新たな問題が引き起こされてしまうことがある．例えば，仕様で定められた変数の初期値を変更してしまったり，他の開発者が新しい機能を実装するため頻繁に更新を行っている箇所に別の意図で変更を行ってしまう等の事態が起こり得る．

このような変更の難しさは箇所によって異なるが，それ事前に知ることはソフトウェアに対する十分な理解を必要とする．ソフトウェアの全体像の把握が困難な新規開発者にとって，このことは大きな問題となる．

3 変更危険度の計測手法

本節では，本稿で提案する手法について述べる．

3.1 用語

本手法における主な用語についての説明を以下に記す．

- 更新履歴情報

  更新履歴情報とは，現行のソースコードに至るまでの全ての更新についての情報である．この情報は，版管理システムのリポジトリに蓄積された開発履歴情報から得られる．
- 更新パターン

  更新パターンとは現行のソースコードに至るまで「どのような更新を行ってきたか」を示す更新履歴を示す．
- 変更危険度

  変更危険度とは「変更によって問題が発生する可能性の大きさ」を表す．

  本稿で提案する手法では更新パターンを分析し，それぞれの更新パターンについてトランク上の最新リビジョンにおけるソースコードの各行に対する変更危険度を計測する．

3.2 手法の概要

本節では，版管理システムの更新履歴情報から更新パターン毎の変更危険度を計測する手法について述べる．

変更危険度の値が大きい箇所に見られると考える更新パターンとして，以下の 3 つのパターンを用いることとした．

- 一度変更した内容が後になって戻されているパターン (Undoing pattern，以降**更新パターン U**)

  このパターンが見られるとき，過去に問題のある変更を行ってしまったと考えられる．そのため，将来変更を行った場合，同様の問題を引き起こす可能性がある．
- 変更されている量が多いパターン (large change Amount pattern，以降**更新パターン A**)

  このパターンが見られるとき，重大な不具合が発生してしまったか，あるいは大きな機能の追加や修正などのためにソースコードを大きく変更する必要があった可能性がある．一般にこのような箇所に対する変更は注意を要すと考えられる．

● 頻繁に更新されているパターン (high change Frequency pattern，以降**更新パターン F**)

このパターンが見られるとき，何らかの機能が開発途中である可能性があり，安易に変更を行うことは危険である．

本手法ではこの 3 つの更新パターンを分析し，変更危険度を計測する．以降では更新パターン U, A, F から計測される変更危険度をそれぞれ **CR(U)**, **CR(A)**，**CR(F)** とする．

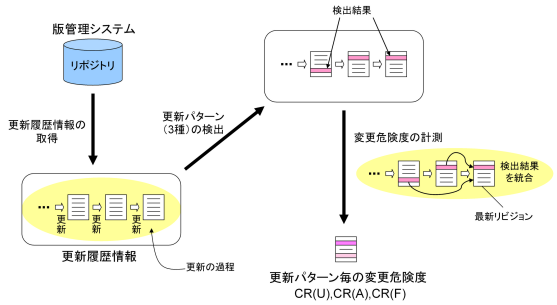

図 3: 提案手法の概要

本手法は図 3 に示されるように，3 つの手順に分けられる．まず，ソースコードの更新履歴情報を取得し，それを基に過去の更新の過程を導出する．次に，過去の更新の過程から各更新パターンの検出を行う．最後に，更新パターンの検出結果をそれぞれ統合し，変更危険度 CR(U),CR(A),CR(F) を計測する．なお，CR(U),CR(A),CR(F) はトランク上の最新リビジョンにおけるソースコードの各行に対して計測する．以下では，これら 3 つの手順について説明する．

3.3 更新履歴情報の取得

リポジトリを解析し，分析対象のソースコードファイルに対応した更新履歴情報を取得する．

また，取得した更新履歴情報の差分情報から更新が行われた前後のリビジョン間における行の対応付けとその行の内容を保存する．行の対応付けは，行だけでなく行と行間の対応を考える．

例えば，図 4 のような更新がなされたとする．i 行目と $(i+1)$ 行目の行間を $i+$ と表記すると，リビジョン 1.5 の行間 7+ に対してリビジョン 1.6 の 8 行目〜11 行目が追加されたことは更新履歴情報から知ることができる．しかし，この 2 つが対応関係にあると定義すると，リビジョン 1.6 の行間 7+,11+ に対応するリビジョン 1.5 の箇所が存在しため，本手法ではリビジョン 1.5 の行間 7+ とリビジョン 1.6 の行間 7+〜行間 11+ が対応関係にあると定義する．同様に，図 4 を例にとると，リビジョン 1.5 の行間 8+〜行間 10+ とリビジョン 1.6 の行間 12 + 〜行間 13+，リビジョン 1.5 の行間 16+〜行間 19+ とリビジョン 1.6 の行間 19+ がそれぞれ対応関係にある．

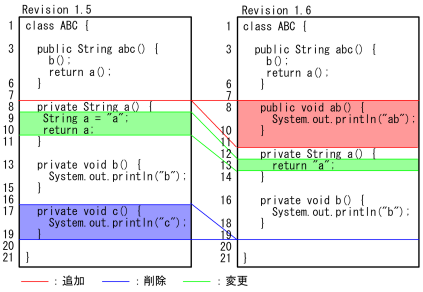

図 4: 行（及び行間）の対応関係の例

3.4 更新パターンの検出

更新履歴情報から得られた過去の更新過程を追跡することにより，更新パターン U, A, F の検出を行う．また，更新パターン U, A についてはそれぞれ値 u,a を検出箇所の各行に記録する．値 u,a は変更危険度を計測する際に用いる．以下では，それぞれの更新パターンついて説明する．

● 更新パターン U

図 5 及び図 6 は更新パターン U の例である．なお，図 5 及び図 6 のリビジョンは右に行くほど新しいものを表している．

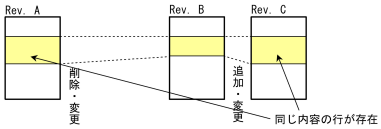

図 5: 更新パターン U の例 1

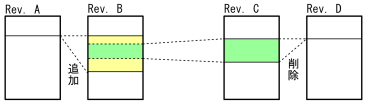

図 6: 更新パターン U の例 2

図 5 及び図 6 のような更新過程が見られるとき，一度変更した内容が後になって戻されていると判断する．図 5 の場合は，対応関係にある範囲内で削除または変更されたリビジョン A の行と同じ内容を持つリビジョン C の行それぞれに値を $u = 1$ として記録する．図 6 の場合は，リビジョン C から削除されてできたリビジョン D の行間に値を $u = \frac{k}{j}$ として記録する．j はリビジョン A に対して追加されたリビジョン B の

表 1: 変更量と値 a の記録

更新作業	更新前	更新後	総変更量	値 a の記録
追加	行間	m 行	m	更新後のリビジョンの m 行 それぞれに $a=1$
削除	n 行	行間	n	更新後のリビジョンの行間に $a=n$
変更	n 行	m 行	$n+m$	更新後のリビジョンの m 行 それぞれに値 $a=\frac{n+m}{m}$

※ n,m は正数

行数，k はリビジョン C から削除された行に対応するリビジョン B の行数である．更新パターン U が見られない箇所では $a=0$ と記録する．

- 更新パターン A

  更新パターン A はその存在の有無を明確に決定できない．従って本手法では，過去の更新過程から全ての変更においてその変更量の多さを求め，更新パターン A による変更危険度を計測する．変更量の定義及び値 a の記録については表 1 に示す通りである．a の値が記録されない箇所は $a=0$ と記録する．

- 更新パターン F

  更新パターン F はその存在の有無を明確に決定できない．従って本手法では，過去の更新過程において，ある箇所の更新が行われたかどうかを判定し，更新パターン F による変更危険度を計測する．計測には更新パターン A の検出した値を利用するため，更新パターン F としての値は記録しない．

3.5 変更危険度の計測

変更危険度はトランク上の最新リビジョンの各行について，CR(U), CR(A), CR(F) が計測される．これらの変更危険度の計測において，本手法では時間的な重みを考慮する．時間的な重みを用いるのは以下のような理由による．

- 最近のリビジョンで更新パターンが検出された場合のほうが最新のソースコードに与えている影響が大きいと考える．
- 過去のリファクタリング等による変更危険度の低下を表現する．

本手法では時間的な重みを以下のように設定した．更新時間については更新履歴情報から得る．

$$\text{時間的な重み} = (1-d)^{\frac{x}{t}}$$

$$d = 0.2$$

$$x = (\text{最新リビジョンの更新時間 [秒]}) - (\text{対象のリビジョンの更新時間 [秒]})$$

$$t = \frac{60\times 60\times 24\times 365}{10}[\text{秒}]$$

以下では，変更危険度 CR(U), CR(A), CR(F) の計測方法を説明する．基本となる計測方法を図 7 に示す．なお，トランク上の最新リビジョンを $i=0$ とし，トランク上の過去のリビジョンを新しいものから順に $1,2,3,4,\cdots$ とする．変更危険度 CR(U), CR(A), CR(F) では，それぞれ図 7 の検出した値 v_i が異なるのみで，検出した値 v_i の基本的な統合手法は同じである．

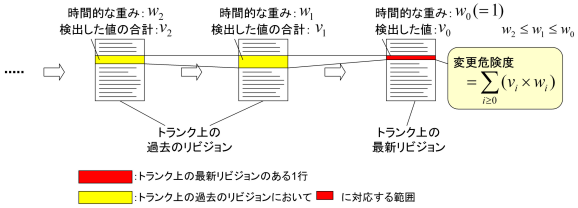

図 7: 基本となる変更危険度の計測方法

計測の具体的な手順を以下に示す．変更危険度を計測する最新リビジョンのある 1 行を $target$ と表記する．

(1) トランク上の各リビジョン i について時間的な重み w_i を設定する．最新リビジョンの時間的な重み w_0 は 1 となる．

(2) $target$ に記録された値 u,a から，更新パターン U,A の値をそれぞれ $v(U)_0=u$，$v(A)_0=a$, とする．更新パターン F の値は $v(A)_0>0$ ならば $v(F)_0=1$，$v(A)_0=0$ ならば $v(F)_0=0$ とする．

(3) トランク上の過去のリビジョンにおいて，$target$ に対応する範囲を定める．対応する範囲は，各リビジョン間の行と行間の対応を追跡することにより求める．

(4) 過去のリビジョン i の対応する範囲 R_i 内の行と行間に記録された更新パターン U,A の値 u,a をそれぞれ合計し，$v(U)_i=\sum_{R_i} u$, $v(A)_i=\sum_{R_i} a$ とする．また $v(A)_i>0$ ならば $v(F)_i=1$，$v(A)_i=0$ ならば $v(F)_i=0$ とする．

(5) 各リビジョンの値 $v(U)_i, v(A)_i, v(F)_i$ と時間的な重み w_i から，$target$ の変更危険度を以下のように計測する．

$$CR(U) = \sum_{i \geq 0} (v(U)_i \times w_i)$$
$$CR(A) = \sum_{i \geq 0} (v(A)_i \times w_i)$$
$$CR(F) = \sum_{i \geq 0} (v(F)_i \times w_i)$$

このようにして計測された CR(U) は，ある行の内容がそれ以前の内容に戻されたことがある行数に，それぞれの時間的な重みを掛け合わせた値を意味している．また CR(A) は，最新のソースコードの行が現在の内容に至るまでに変更された行数に，それぞれの時間的な重みを掛け合わせた値を意味し，CR(F) は，最新のソースコードの行が現在の内容に至るまでに変更された回数にそれぞれの時間的な重みを考慮した値を意味している．

4 実装

本節では，3 節で述べた提案手法を実装したシステムについて述べる．

本システムは，提案手法に従って変更危険度を計測し，HTML 形式で結果の出力を行う．出力例を図 8 に示す．

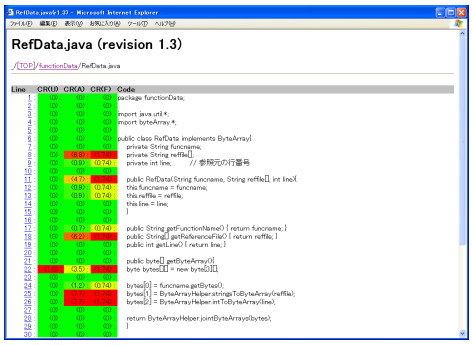

図 8: 分析対象ファイルの各行に対する変更危険度の出力の例

各変更危険度はその値を出力するだけでなく，パターン毎に値の大きさに応じた色を付けている．この色により，変更危険度の値の大きい箇所が視覚的に分かりやすいように配慮されている．彩色の基準は図 9 のように，緑から黄を経て赤に至るグラデーションになっている．この基準は 3 種の更新パターン毎に個別に設けられる．

図 9: 変更危険度の彩色基準

また，行番号にはリンクが貼られている．リンク先の HTML ファイルでは，図 10 のような分析対象ファイルの各行に対する変更危険度の詳細情報が記載されている．

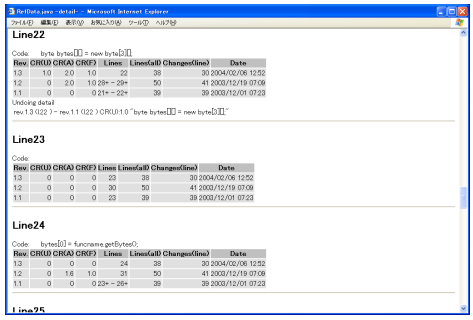

図 10: 分析対象ファイルの各行に対する変更危険度の詳細情報の出力の例

5 評価実験

本節では 4 節で述べたシステムを用いて実験を行い，得られた結果について述べ，考察を行う．

5.1 実験目的

本実験の目的は，変更危険度計測システムが出力する変更危険度の値を参考にすることにより，変更することによって問題が発生する危険性のある箇所を知ることが可能であるかを調べることである．

5.2 実験対象

実験対象には開発履歴理解支援システム CREBASS[5] を用いる．本実験では CREBASS のソースコードの内，Java で記述されたものに対してシステムを適用した．

5.3 実験方法

本実験では，データベース作成の機能を実装する 8 つのソースコードファイルについて評価を行う．

まず本システムの適用結果から CR(U), CR(A), CR(F) のいずれかの値が大きい箇所を抽出した．抽出の際には，コメント部などソースコードの危険性には関係がないと判断される箇所を除いた．その結果，値の大きな箇所は合計で 26 箇所となった．

次に抽出を行った 26 箇所について，CREBASS を熟知する開発者 1 人にアンケート調査を行った．アンケートの内容は以下の通りである．

- 過去にバグが発生したことがあった
- アルゴリズムが複雑である
- 他のモジュールに影響を及ぼしやすい
- 実装方法を何度も変更した

表 2: アンケートの結果

	箇所数	内訳						
		U	A	F	U,F	U,A	A,F	U,A,F
過去にバグが発生したことがあった	6	0	1	5	0	0	0	0
アルゴリズムが複雑である	0	0	0	0	0	0	0	0
他のモジュールに影響を及ぼしやすい	0	0	0	0	0	0	0	0
実装方法を何度も変更した	12	0	1	3	1	0	6	1
上記以外の問題が過去に存在した	1	0	0	0	0	0	1	0
小計	19	0	2	8	1	0	7	1
該当無し	7	0	0	7	0	0	0	0
合計	26	0	2	15	1	0	7	1

上記以外の問題が過去に存在した…仕様の変更

U…CR(U) の値のみが大きい箇所
A…CR(A) の値のみが大きい箇所
F…CR(F) の値のみが大きい箇所
U,F…CR(U) と CR(F) の 2 つの値のみが大きい箇所
U,A…CR(U) と CR(A) の 2 つの値のみが大きい箇所
A,F…CR(A) と CR(F) の 2 つの値のみが大きい箇所
U,A,F…CR(U),CR(A),CR(F) の 3 つの値全てが大きい箇所

- 上記以外の問題が過去に存在した

開発者は，このアンケートの項目に当てはまるものがあれば，そのうち最も適切であるものを選ぶ．アンケートのいずれかの項目に該当する箇所は，変更することによって問題が発生する危険性があると判断した．

5.4 実験結果

アンケートの結果を表 2 に示す．

全 26 箇所の内，19 箇所がアンケートの項目に該当している．これより適合率は $\frac{19}{26}$ となり，およそ 0.73 となった．

5.5 考察

実験結果を見ると，適合率が 0.73 と比較的高いことが分かる．このことより，本手法が概ね有効であることが示された．

以下では各更新パターンの変更危険度 CR(U),CR(A),CR(F) について，実験結果から分かる事柄について記載する．

- CR(U)

今回の実験では，CR(U) の値のみが大きい箇所，または CR(U) と CR(A) の値のみが大きい箇所はなかった．すなわち，CR(U) の値のみが大きい箇所では必ず CR(F) の値も大きかった．このことは，更新パターン U が見られる可能性が低いことを表している．すなわち，更新を行った後に更新前の状態に戻すような事態になることは珍しく，またそのような事態が発生した箇所は頻繁に更新されている傾向にあると考えられる．本手法ではそのような珍しい出来事が発生した箇所を CR(U) の値として検出できていることが分かる．

また，本実験において CR(U) の値が大きい箇所は，実装を何度も変更した箇所であることから，ソースコードの振る舞いに問題があったのではなく，仕様や設計上の問題により更新を以前の状態に置き換えた箇所であると考えられる．

本実験では，CR(U) の値は過去にバグを発生した箇所の抽出に役立ってはいない．これが偶然の結果なのかどうかは更なる実験によって確認する必要がある．しかし上記の事柄より，CR(U) の値は仕様や設計上の問題による危険性を把握するのに有用であることが示されている．

- CR(A)

本実験において CR(A) の値が大きい箇所は全部で 10 箇所あるが，その内 8 箇所では CR(A) だけでなく CF(F) の値も大きい．このことから，更新パターン A の見られる箇所では，同時に更新パターン F が見られやすいことが確認できる．本実験において，このような箇所は，いずれも危険性のある箇所に該当している．

また，CR(A) の値のみが大きい箇所については，本実験結果を見ると過去にバグが発生していたり，実装方法を何度も変更していることが分かる．しかし，全体的に CR(A) の値は CF(F) の値が大きい箇所と一致する傾向が強く，CR(A) 単体の特徴を把握するには更なる実験が必要である．

- CR(F)

本実験において CR(F) の値が大きい箇所は全部で 24 箇所と最も多い．また，CR(F) の値のみが大きい箇所は 15 箇所であるが，この内 7 箇所がアンケートのどの項目にも該当しておらず，しかもアンケートのどの項目にも該当しなかったのはこの 7 箇所だけである．

これら 7 箇所について開発者に意見を求めたところ，これらの箇所は設計やアルゴリズムの変更に伴い，最近更新を行った箇所であることが分かった．アンケート調査を行った開発者は，実質 1 人で CREBASS の開発を行っているため，CREBASS の開発過程を十分に理解しており，これらの 7 箇所には危険性がないと判断した．しかし，最近更新を行った箇所は，開発過程を十分に理解していない新規の開発者にとっても危険性が無いとは言えない．更新が行われた意図を理解せずにこれらの箇所に変更を加えると，開発を混乱させる可能性は十分にあると考えられる．

以上のことから，CR(F) の値が大きい箇所は全ての開発者にとって変更すると問題が発生する危険性のある箇所を示している訳ではないが，CR(F) から得られる情報は十分に有用であると言える．

また，アンケート調査を行った際に「データベースを実装したソースコードファイルには，今回抽出した 26 箇所以外にも存在する」という意見を開発者から得ることができた．これより，本手法で抽出できていない箇所が存在することが分かる．この問題を解決するには，更新パターン U, 更新パターン A, 更新パターン F 以外に新しい更新パターンを考案する必要がある．

6 まとめ

本稿では版管理システムのリポジトリを解析し，更新パターンを分析することで計測する変更危険度をユーザに提示することにより，更新作業において特に注意を要する箇所を特定し，ソフトウェア開発を支援する手法を提案した．

さらに，手法を実装したシステムを用いて評価実験を行った．本実験において，高い適合率で変更に注意を要する箇所を特定することができた．これにより，本手法はユーザのソフトウェア開発において有益な情報を提供できることが確認された．

今後の課題としては，以下のものが挙げられる．

- 構文解析の導入

今回の評価実験においては，変更危険度の高い箇所を選択する際，コメント部などのソースコードの危険性に関係のない箇所は除いた．プログラミング言語に応じた構文解析を実行できるように変更危険度計測システムを拡張すれば，コメント部などのソースコードの危険性に関係のない箇所を判別可能になる．これにより，変更危険度計測の精度向上が期待できる．

- 新たな更新パターンの考案

今回の評価実験時に得られた開発者の意見から，本来は変更危険度が大きいと判定されるべき箇所が本手法では完全には抽出できていないことが分かっている．このような箇所を抽出するため，新たな更新パターンの考案が望まれる．

- 多人数で開発されているソフトウェアに対する評価実験

今回の評価実験において実験対象とした CREBASS は実質 1 人で開発が行われていたため，今後は多人数で開発されているソフトウェアに対する評価実験も行う必要がある．

参考文献

[1] Peter H. Feiler, “Configuration Management Models in Commercial Environments”, CMU/SEI-91-TR-7 ESD-9-TR-7, March, 1991.

[2] Jacky Estublier, “Software Configuration Management: A Roadmap”. The Future of Software Engineering in 22nd ICSE, pp.281–289, 2000.

[3] Brian Berliner, “CVS II:Parallelizing Software Development”, In USENIX Association, editor, Proceedings of the Winter 1990 USENIX Conference, pages 341–352, Berkeley, CA, USA, 1990.

[4] Karl Fogel, “Open Source Development with CVS”, The Coriolis Group, 2000.

[5] 中山崇, 松下誠, 井上克郎, “関数の変更履歴と呼び出し関係に基づいた開発履歴理解支援システム”, 電子情報通信学会技術研究報告, SS2004-2, Vol.104, No.47, pp.7-12, 2004.