How to use ECTEC

平成 26 年 5 月 29 日

目 次

1 まえがき

このドキュメントは，コードクローン追跡システム ECTEC (**E**nhanced **C**RD-based **T**racker for **E**volution of **C**lones) の使用方法並びに内部で行われている処理内容について述べているドキュメントです．このドキュメントは以下のように構成されています.

2 章 - 概要
: このツールが何をするツールなのかを簡単に説明すると共に，適用対象に関する制限について言及します.

3 章 - クイックスタート
: インストール方法や動作環境について説明し，実行方法について簡単に説明します.

2 概要

2.1 入出力

コードクローン追跡ツール ECTEC はソフトウェアの開発履歴情報を入力とし，そこからコード片やコードクローンの系譜を特定して出力するツールです.

開発履歴情報：
: ECTEC が入力として受け付ける “開発履歴情報” とは，Subversion や git 等のバージョン管理システムで管理されているソースコードリポジトリのことを指します．バージョン管理システムを用いたソフトウェア開発では，ソースコードリポジトリに保存されているソースコード (やその他関連ファイル) を開発者が取得し，そこに適宜変更を加えて，変更した内容をソースコードリポジトリに反映させる，という流れで開発が進んでいきます．変更した内容をソースコードリポジトリに反映させる操作のことを，“コミット” と呼びます．コミットが行われるたびに，ソースコードリポジトリにはコミット後のソースコードのスナップショットが保存されます*．ソースコードリポジトリに保存されているそれぞれのスナップショットのことを，“リビジョン” と呼びます．開発履歴情報を用いることで，任意のリビジョンのソースコードを取得することができますので，ソースコードがどのような進化・変遷を辿ってきたのかを分析することが可能になります.

コード片やコードクローンの系譜：
: ECTEC が出力する “コード片やコードクローンの系譜 (genealogies)” とは，開発履歴情報に蓄積された各リビジョンに存在するコード片やコードクローンを時系列順につなぎ合わせたものです．コード片やコードクローンの系譜図のようなものであり，系譜を分析することでそれぞれのコード片やコードクローンがいつ (どのコミットで) 生成され，どのような進化・変遷を辿り (どのコミットでどのように修正され)，いつ (どのコミットで) 消滅したのかを知ることができます.

ECTEC は与えられたソースコードリポジトリを分析し，特定した系譜に関する情報をデータベースに保存します．出力されたデータベースに保存されている情報を取得・加工することで，コード片やコードクローンの進化の様子を分析することができます.

2.2 制限

現在のところ，ECTEC を適用できるソフトウェアは以下のように制限されています.

言語： Java で記述されたソフトウェアのみ．複数の言語を併用して構成されているソフトウェアの場合，Java 以外の言語で記述されたソースコードはすべて無視されます.

バージョン管理システム： Subversion のみ．他のバージョン管理システム (git など) で管理されているソフトウェアの場合，リポジトリを Subversion リポジトリに変換する必要があります.

*厳密には必ずしもスナップショットが保存されるわけではありませんが，スナップショットを取得することが可能なので，ECTEC を使う際にはこのようなイメージを持っていただければよいと思います.

3 クイックスタート

3.1 インストール

ECTEC は Java で記述されており，Java Runtime Environment 1.7.0 以降のバージョンで動作します．内部で使用しているライブラリ等はすべてプロジェクト中にインポートされています．従って，Java が動作する環境が構築されていれば，ECTEC は動作します．

ECTEC が行う処理はいくつかのステップに分かれており，それぞれにメインクラスが定義されています．ECTEC を動作させる際は，それぞれのステップのメインクラスに必要な引数を与えて実行します．それぞれのステップの実行方法については後述します．

ECTEC のそれぞれのステップについて，実行可能 jar ファイルを用意しています．実行の際は，以下の URL より実行可能 jar ファイルを入手してください．

```
http://sdl.ist.osaka-u.ac.jp/~k-hotta/files/ECTEC-jars.zip
```

圧縮ファイルの中には，すべてのステップの実行可能 jar ファイルが含まれています．

また，ECTEC は GitHub プロジェクト[†] として公開されています．ソースコードをご覧になりたい場合は，こちらから入手してください．

3.2 実行の前に

また，ECTEC は各ステップの起動時に，設定情報が記述されたプロパティファイル (`*.properties`) を参照します．デフォルトでは，カレントディレクトリの `ectec.properties` というファイルを参照するように設定されています．プロジェクトのルートディレクトリ直下に `ectec-sample.properties` というファイルがありますので，ひとまずはそのファイルの名前を変更して使用してください．

3.3 実行手順

ここでは，小規模なリポジトリを対象として，実際に ECTEC を動かす手順について説明します．説明に用いるリポジトリは，このツールの開発者が以前に製作したソフトウェアの Subversion リポジトリです．このリポジトリは `http://sdl.ist.osaka-u.ac.jp/~k-hotta/files/repository-ScorpioTM.zip` よりダウンロードできます．本節で説明する手順を再現する場合は，ダウンロードした圧縮ファイルを解凍し，ローカルに配置してください．

以降の説明では，以下の環境を想定しています．

OS: Windows 7

Detector の実行可能 jar ファイル: detector.jar

リポジトリの場所: ローカルに配置．パスは G:\ECTEC\sample-repository

出力先: G:\ECTEC\sample.db

STEP1: データベースファイルの作成

ECTEC が行う処理は，事前分析とライブラリ呼び出しに大別される[‡]．先に事前分析を行いデータベースを構築した後，ライブラリ呼び出しを行うことでコードクローンの系譜に関する情報を得ることができる．

以降の小節でそれぞれの実行方法について簡単に述べる．なお詳細は 6 を参照されたい．

[†] `https://github.com/kusumotolab/ECTEC`

[‡] 詳細については 4 で述べる

3.4 事前分析

事前分析のメインクラスは `jp.ac.osaka_u.ist.sdl.ectec.detector.DetectorMain` である．以降，事前分析を行う機能のことを **Detector** と呼ぶ．メインメソッドに様々な引数を与えることで，Detector の挙動を細かく制御することができる．しかし，それらのすべてを逐一指定することは煩雑であるため，Detector はプロパティファイルから設定を読み込み，それをデフォルトの値として使用する．Detector は，カレントディレクトリの “ectec.properties” をデフォルトのプロパティファイルとするため，実行に際しカレントディレクトリにこの名前のプロパティファイルを配置する必要がある[§]．test-resources/properties/ectec-sample.properties にサンプルのプロパティファイルを配置しているので，さしあたりそのファイルをカレントディレクトリに配置し，名前を “ectec.properties” に変更されたい．

Detector を実行する際に，必ず指定しなければならない必須引数は以下の 2 つである．

-r: 分析対象リポジトリの URL

-d: 分析結果の出力先データベースファイルのパス

リポジトリの URL については，http，file，svn+ssh のそれぞれのプロトコルをサポートしている．出力先ファイルは “*.db” を拡張子とするファイルであることが望ましい．

例として，以下のような環境を想定する．

OS: Windows 7

Detector の実行可能 jar ファイル: detector.jar

リポジトリの場所: ローカルに配置．パスは G:\ECTEC\sample-repository

出力先: G:\ECTEC\sample.db

このとき，Detector を起動するためのコマンドは以下のようになる．

```
$ java -jar detector.jar -r file:///G:/ECTEC/sample-repository
-d G:\ECTEC\sample.db
```

この例の場合，リポジトリがマシンローカルに配置されているため，リポジトリにアクセスするためのプロトコルは file となる．リポジトリの URL 中のファイル区切り文字が Windows の \ ではなく / になっている点に注意されたい．

なお引数の指定は順不同である．従って，以下のように起動しても，先ほどと全く同じ結果が得られる．

```
$ java -jar detector.jar -d G:\ECTEC\sample.db
-r file:///G:/ECTEC/sample-repository
```

3.5 ライブラリ呼び出し

ライブラリ呼び出しでは，利用者が記述する Java のコードから ECTEC の解析結果にアクセスする機能を提供する．以降，ライブラリ呼び出しによって解析結果を提供する機能のことを **Analyzer** と呼ぶ．

以降の説明では，以下の環境を想定する．

OS: Windows 7

リポジトリの場所: ローカルに配置．パスは G:\ECTEC\sample-repository

出力先: G:\ECTEC\sample.db

Analyzer のメインとなるクラスは `jp.ac.osaka_u.ist.sdl.ectec.analyzer` パッケージに存在する `GenealogyAnalyzser` である．利用者は，はじめに `setup` メソッドを呼び出すことで Analyzer の初期化を行う必要がある．想定している環境の場合，メソッドの呼び出し方法は以下のようになる．

[§] どのプロパティファイルを読み込むかも引数で制御可能であるが，指定が無ければデフォルトのプロパティファイルを読み込む

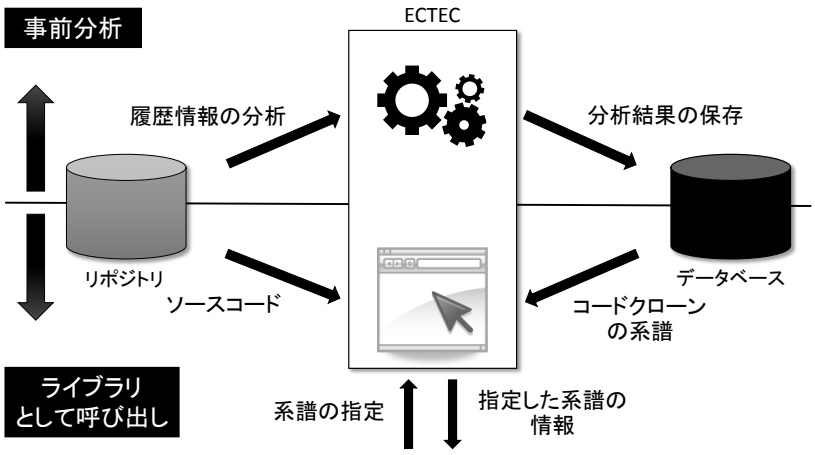


図 1: 処理の全体像

```
GenalogyAnalyzer analyzer =
      GenealogyAnalyzer.setup(G:\ECTEC\sample.db,
         file:///G:/ECTEC/sample-repository, VersionControlSystem.SVN);
```

次に，`selectAndConcretizeCloneGenealogies` メソッドを呼び出すことで，コードクローンの系譜に関する情報を取得することができる．コードクローンの系譜に関する情報は `CloneGenealogyInfo` という名前のクラスのインスタンスとして提供される．

このメソッドは `IConstraint` 型の引数を受け取る．この引数は取得する系譜に制約を設ける際に使用するものである．特に制約を設けない場合，`null` を指定することですべての系譜を取得することができる ¶.

具体的なメソッドの呼び出し方法は以下の通り．

```
Map<Long, CloneGenealogyInfo> genealogies =
          analyzer.selectAndConcretizeCloneGenealogies(null);
```

このようにして取得した `CloneGenealogyInfo` 型のインスタンスに対してメソッド呼び出しを行うことで，様々な情報を取得することができる．

最後に，プログラムを終了する前に，データベースとの接続を切る必要がある．そのために使用するメソッドは `close()` である．

```
analyzer.close();
```

4 処理

4.1 概要

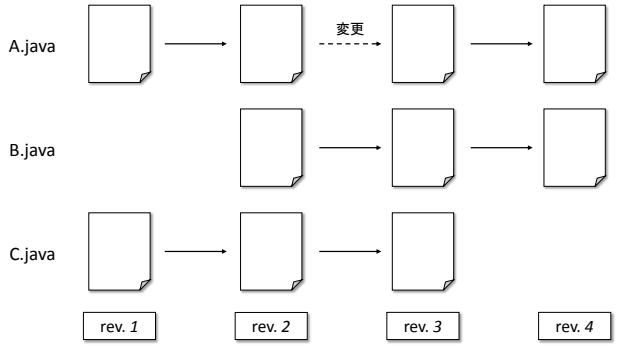
処理の全体像を図 1 に示す．処理は大きく以下の 2 つのステップから成る．

STEP1 事前分析: 開発履歴情報を分析してコードクローンの系譜を特定し，データベースに格納する．

STEP2 ライブラリ呼び出し: データベースに格納されている系譜のうち，指定されたものについて，系譜を構成するコードクローンのソースコードなどを取得する．

ライブラリ呼び出し機能を実行するためには，事前分析を行いデータベースを構築する必要がある．以降の小節でそれぞれの処理について述べる．

¶ただし，`CloneGenealogyInfo` は各系譜の各コードクローンのソースコードを保持しており，メモリ使用量が大きい．従って，すべての系譜を取得することが難しい場合は，制約を設けて取得する系譜を制限することを推奨する．



ID	Path	Start Rev.	End Rev.
1	A.java	1	2
2	A.java	3	4
3	B.java	2	4
4	C.java	1	3

図 2: 分析対象ソースファイル特定の例

4.2 STEP1: 事前分析

事前分析では，与えられたリポジトリを解析し，コードクローンの系譜並びにそれに関連する情報を取得する．その後，取得した情報をデータベースに保存する．

事前分析は以下のサブステップから構成される．

STEP1-1: 対象リポジトリから，分析対象コミット‖ 並びに分析対象リビジョン ** を特定する．

STEP1-2: 各分析対象リビジョンについて，そのリビジョンに存在するソースファイルを特定する．

STEP1-3: 分析対象ソースファイルを解析し，それぞれのソースファイル中に存在するブロックを特定する．

STEP1-4: STEP1-3 で特定したブロックについて，コミットの前後のリビジョンに存在するブロック間で CRD の類似度を用いた対応付け (以降，ブロックリンク) を特定する．

STEP1-5: STEP1-3 で特定したブロックについて，ブロックの文字列からハッシュ値を生成し，それを用いてブロック単位のコードクローン検出を行う．

STEP1-6: STEP1-4 で特定したブロックリンクと STEP1-5 で特定したコードクローンをもとに，コミット前後のリビジョンに存在するコードクローン間の対応付け (コードクローンリンク) を特定する．

STEP1-7: STEP1-6 で特定したコードクローンリンクをもとに，コードクローンの系譜を特定する．

4.2.1 STEP1-1: 分析対象コミットの特定

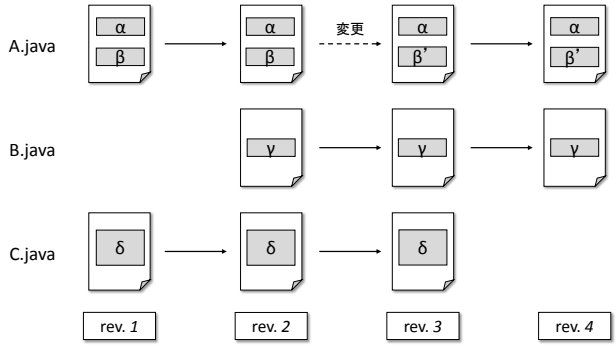
対象リポジトリに保持されている開発履歴中のすべてのコミットについて，ログを用いてそのコミットで追加・削除・変更されたファイルのリストを取得する．そして，リスト中に分析対象ソースファイルが含まれるか否かを判定し，含まれている場合はそのコミットを分析対象コミットとみなす．分析対象ソースファイルは対象とする言語により異なるが，Java の場合は “`.java`” を拡張子とするファイルである．最後に，特定した分析対象コミット，並びに各分析対象コミットの変更前後のリビジョンをデータベースに保存する．

4.2.2 STEP1-2: 分析対象ソースファイルの特定

STEP1-1 で特定した各分析対象リビジョンについて，そのリビジョンに存在する分析対象ソースファイルを特定し，その情報をデータベースに格納する．ここで特定したソースファイルがのちの解析処理の対象となる．

ただし，あるコミットで修正されなかったファイルはコミット前後のリビジョンにおいて同一のファイルとなるため，そのようなファイルに対してコミット前後それぞれのリビジョンで解析を行うことは冗長である．このような冗長な解析を避けるため，各ファイルについて，そのファイルが修正されずに存在していた

‖少なくとも一つ以上の分析対象ソースファイルが追加・削除・変更されたコミット
**分析対象コミットの前後のリビジョン



ID	Type	Start Rev.	End Rev.	Hash
1	α	1	2	10
2	α	3	4	10
3	β	1	2	20
4	β'	3	4	30
5	γ	2	4	40
6	δ	1	3	50

図 3: ブロック特定の例

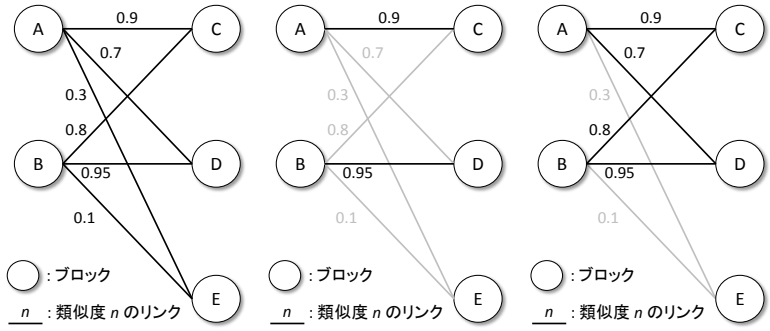
期間を付与し，解析結果を再利用する．図 2 に分析ソースファイルの特定例を示す．この例では，`A.java` がリビジョン 2 からリビジョン 3 へのコミットで修正されている．また，`B.java` がリビジョン 1 からリビジョン 2 へのコミットで追加されており，`C.java` がリビジョン 3 からリビジョン 4 へのコミットで削除されている．この例の場合，特定されるソースファイル情報は右表のようになる．`A.java` はリビジョン 1 からリビジョン 2 へのコミットで修正されていないため，リビジョン 1 で解析を行った結果をリビジョン 2 において再利用することができる．一方，`A.java` はリビジョン 2 からリビジョン 3 へのコミットで修正されているため，リビジョン 2 での解析結果をリビジョン 3 において再利用することはできない．従って，リビジョン 2 からリビジョン 3 へのコミットを区切りとして別々のソースファイル情報を作成している．最終的に，この例に対して解析する必要のあるファイルは 4 つとなる．

4.2.3 STEP1-3: ブロックの特定

STEP1-2 で特定したソースファイルを解析し，そのファイル中に含まれるブロックを特定する．

ここでブロックとは，クラスやメソッド，並びに各種ブロック文を指している．Java を対象とした場合，以下のものがブロックに該当する．

- クラス
- メソッド
- if 文
- else 節
- for 文
- 拡張 for 文 (for-each 文)
- while 文
- do-while 文
- switch 文
- synchronized 文
- try 文
- catch 節
- finally 節



(a) ブロック間の類似度 (b) Single (c) Multiple

図 4: ブロックリンク特定の例

ブロックについてもソースファイルと同様にその存在期間を付与することで，解析結果の再利用を行う．図 3 にブロック特定の例を示す．この例では，`A.java` がリビジョン 2 からリビジョン 3 へのコミットで変更されており，その変更によってブロック β の中身が変化し，ブロック β' となっている．よって，ブロック β とブロック β' の存在期間はそれぞれ，リビジョン 1 からリビジョン 2，リビジョン 3 からリビジョン 4 となっている．また，`A.java` には別のブロック α が存在している．このブロックはリビジョン 2 からリビジョン 3 へのコミットで変化していないが，ブロックを含むソースファイルが変化し再解析されているため，ブロック α についても β，β' と同様にリビジョン 2 とリビジョン 3 の間で別のブロックとして認識される．

4.2.4 STEP1-4: ブロックリンクの特定

このサブステップでは，2 つの連続するリビジョンに存在するブロックの間で対応関係を取る．対応関係の特定には，CRD に基づく手法 [1] を拡張した，CRD の類似度に基づく手法 [2] を用いる．ECTEC は以下の 2 つの対応付け方法を提供している．

Single: 前リビジョンに存在するあるブロックについて，後リビジョンに存在するブロックのうち最も確からしいブロック 1 つと対応付ける．

Multiple: 前リビジョンに存在するあるブロックについて，後リビジョンに存在するブロックのうち条件を満たすブロックすべてと対応付ける．

ここで，“最も確からしい” とは，条件 †† を満たすブロックの中で CRD の類似度が最も高いものを指す．

図 4 にブロックリンク特定の例を示す．図中の左側に並ぶブロックが変更前リビジョンに存在するブロック，右側が変更後のリビジョンに存在するブロックである．また，類似度の閾値は 0.5 としている．

Single の場合は，最も類似度が高くなる対応付け 1 つを求めるため，この例の場合，A と C，並びに B と D という 2 つの対応付けが得られる．一方 Multiple の場合は，条件 (この場合，類似度が 0.5 以上) を満たす対応付けすべてを許容するため，Single で特定された 2 つの対応付けに加え，A と D，B と C という対応付けが得られる．

4.2.5 STEP1-5: コードクローンの特定

このサブステップでは，STEP1-3 で特定したブロックからコードクローンの検出を行う．

コードクローンの検出には，各ブロックから生成されたハッシュ値を用いる．ハッシュ値の生成の際，各ブロックを文字列とみなし，その文字列にハッシュ関数を適用してハッシュ値を生成する．同じ文字列からは同じハッシュ値が生成されるため，同じハッシュ値を持つブロック同士はコードクローン関係にあるとみなすことができる．

ハッシュ値を生成する際に，ブロックの文字列に対して正規化を行うことが可能である．ECTEC は以下の 3 種類の文字列生成方法を提供している．

†† 文献 [2] に定義されている

```
public void method(int x, int y) {
    int z = x + y;
    int w = init(x, y);
    for (int i = 0; i < z; i++) {
        w += x * y;
    }
    return w;
}
```

(a) 元のソースコード

```
public void method(int x, int y) {
int z = x + y;
int w = init(x, y);
for (int i = 0; i < z; i++) {
w += x * y;
}
return w;
}
```

(b) 完全一致 (正規化なし)

```
public void method(int $, int $) {
int $ = $ + $;
int $ = init($, $);
for (int $ = $; $ < $; $++) {
$ += $ * $;
}
return $;
}
```

(c) 変数名，リテラルの正規化

```
public void method(int $, int $) {
int $ = $ + $;
int $ = init($, $);
FOR: i < z
return $;
}
```

(d) サブブロックの正規化

図 5: ソースコードの正規化

完全一致 (正規化なし): 空白並びにインデントの整形は行うが，それ以外の正規化は行わない．

変数名・リテラルの正規化: 空白並びにインデントの整形に加え，変数名並びにリテラルを特殊文字で置き換える．

サブブロックの正規化: 変数名・リテラルの正規化で行った正規化に加え，ブロック中に出現するサブブロックをそのサブブロックを表す特殊文字列で置き換える．

図 5 にソースコードの正規化の例を示す．完全一致ではインデントの整形のみが行われており，変数名・リテラルの正規化ではそれに加えて変数名・リテラルをすべて $ で置き換えられている．サブブロックの正規化では，さらにメソッド内部の for 文を “`FOR: i < z`” という特殊な文字列で置き換えている．この文字列は置き換えた文の種類，並びにその文を特徴付ける文字列 (条件式やシグネチャなど) を用いて生成されるものである．この正規化により，サブブロック内で行われている処理が異なるようなブロックをコードクローンとして検出することが可能となる．なお，この特殊文字列中に出現する変数名・リテラルについては正規化を行わない．

4.2.6 STEP1-6: コードクローンリンクの特定

STEP1-4 で特定したブロックリンク，並びに STEP1-5 で特定したコードクローンをもとに，コードクローンリンクを特定する．2 つのコードクローン C_1，C_2 が以下の条件を満たすとき，C_1，C_2 を対応付け，コードクローンリンクを形成する．

- C_1 が属するリビジョンと C_2 が属するリビジョンが連続している．
- C_1 中のあるブロックと C_2 中のあるブロックがブロックリンクを形成している．

なお，ブロックリンクの特定には一対一と多対多の 2 つの対応付け方法を提供しているが，コードクローンリンクには多対多の対応付けのみを提供している．

4.2.7 STEP1-7: コードクローンの系譜の特定

最後に，STEP1-6 で特定したコードクローンリンクをつなぎ合わせることで，コードクローンの系譜を特定する．

4.3 STEP2: ライブラリ呼び出し

ライブラリ呼び出しでは，事前分析で構築したデータベースからコードクローンの系譜を取得し，データベースへの格納によって失われた情報をリポジトリを用いて復元する．以降，この復元作業のことを**具体化**と呼ぶ．

ライブラリ呼び出しを行うために必要なステップは以下の通りである．

STEP2-1: 解析結果を格納しているデータベースと，解析に使用したリポジトリを指定し，初期化する．

STEP2-2: 具体化したいコードクローンの系譜を選択する．

STEP2-3: コードクローンの系譜を具体化する．

STEP2-4: データベース並びにリポジトリとの接続を閉じる．

これらのステップのうち，STEP2-2 と STEP2-3 は繰り返し行うことが可能である．具体的なメソッド呼び出しの方法等については，使用方法の説明を行う際に詳細に述べる．

5 データ構造

5.1 データベース

ECTEC は以下のデータベーステーブルを定義し，使用する．

- REVISION
- VCS_COMMIT
- FILE
- CODE_FRAGMENT
- CLONE_SET
- CODE_FRAGMENT_LINK
- CLONE_SET_LINK
- CLONE_GENEALOGY
- CRD

表 1〜表 9 に各テーブルのスキーマを示す．それぞれのテーブルは LONG 型の ID を主キーとしている．一部に他のテーブルのレコードを参照する列が存在するが，その参照の際にも各レコードの ID を使用する．

以降，それぞれのスキーマについて簡単に述べる．

REVISION

REVISION_IDENTIFIER は各リビジョンを表す識別子を表している．Subversion の場合，各リビジョンはリビジョン番号により識別されるため，識別子はリビジョン番号となる．分析対象とならないリビジョンが存在する影響で，リビジョンの ID とリビジョン番号は基本的には一致しない．

VCS_COMMIT

VCS_COMMIT はコミットをレコードとするテーブルであり，BEFORE* は変更前リビジョンを，AFTER* は変更後リビジョンを，それぞれ表している．

表 1: REVISION テーブルのスキーマ

カラム名	型	備考
REVISION_ID	LONG	primary key
REVISION_IDENTIFIRE	TEXT	nor null, unique

表 2: VCS_COMMIT テーブルのスキーマ

カラム名	型	備考
VCS_COMMIT_ID	LONG	primary key
BEFORE_REVISION_ID	LONG	external key
AFTER_REVISION_ID	LONG	external key
BEFORE_REVISION_IDENTIFIRE	TEXT	
AFTER_REVISION_IDENTIFIRE	TEXT	

表 3: FILE テーブルのスキーマ

カラム名	型	備考
FILE_ID	LONG	primary key
FILE_PATH	TEXT	
START_REVISION_ID	LONG	external key
END_REVISION_ID	LONG	external

表 4: CODE_FRAGMENT テーブルのスキーマ

カラム名	型	備考
CODE_FRAGMENT_ID	LONG	primary key
OWNER_FILE_ID	LONG	external key
CRD_ID	LONG	external key
START_REVISION_ID	LONG	external key
END_REVISION_ID	LONG	exteral key
HASH	LONG	
HASH_FOR_CLONE	LONG	
START_LINE	INTEGER	> 0
END_LINE	INTEGER	> 0
SIZE	INTEGER	> 0

表 5: CLONE_SET テーブルのスキーマ

カラム名	型	備考
CLONE_SET_ID	LONG	primary key
OWNER_REVISION_ID	LONG	external key
ELEMENTS	TEXT*	not null
NUMBER_OF_ELEMENTS	INTEGER	> 0

表 6: CODE_FRAGMENT_LINK テーブルのスキーマ

カラム名	型	備考
CODE_FRAGMENT_LINK_ID	LONG	primary key
BEFORE_ELEMENT_ID	LONG	external key
AFTER_ELEMENT_ID	LONG	external key
BEFORE_REVISION_ID	LONG	external key
AFTER_REVISION_ID	LONG	exteral key
CHANGED	INTEGER	1 or 0 (1 = true)

表 7: CLONE_SET_LINK テーブルのスキーマ

カラム名	型	備考
CLONE_SET_LINK_ID	LONG	primary key
BEFORE_ELEMENT_ID	LONG	external key
AFTER_ELEMENT_ID	LONG	external key
BEFORE_REVISION_ID	LONG	external key
AFTER_REVISION_ID	LONG	exteral key
CHANGED_ELEMENTS	INTEGER	> 0
ADDED_ELEMENTS	INTEGER	> 0
DELETED_ELEMENTS	INTEGER	> 0
CO_CHANGED_ELEMENTS	INTEGER	> 0
CODE_FRAGMENT_LINKS	TEXT*	not null

表 8: CLONE_GENEALOGY テーブルのスキーマ

カラム名	型	備考
CLONE_GENEALOGY_LINK_ID	LONG	primary key
START_REVISION_ID	LONG	external key
END_REVISION_ID	LONG	exteral key
CLONES	TEXT*	not null
CLONE_LINKS	TEXT*	not null
CHANGES	INTEGER	> 0
ADDITIONS	INTEGER	> 0
DELETIONS	INTEGER	> 0
DEAD	INTEGER	1 or 0 (1 = true)

表 9: CRD テーブルのスキーマ

カラム名	型	備考
CRD_ID	LONG	primary key
TYPE	TEXT	not null
HEAD	TEXT	not null
ANCHOR	TEXT	not null
NORMALIZED_ANCHOR	TEXT	not null
CM	INTEGER	>0
ANCESTORS	TEXT*	not null
FULL_TEXT	TEXT	not null

FILE

FILE はその存在期間を保持するレコードであるため，START_REVISION_ID 並びに END_REVISION_ID という列を持つ．これらはいずれも REVISION テーブルに格納されているレコードの REVISION_ID の値を参照している．

CODE_FRAGMENT

CODE_FRAGMENT は FILE と同様に存在期間を保持するレコードである．この存在期間は各コード片を保持するファイルの生存期間に準拠している．また，各コード片の CRD は別テーブルとして切り分けているため，対応する CRD の ID が保持されている．HASH 並びに HASH_FOR_CLONE は各コード片の文字列から算出されたハッシュ値であり，HASH は正規化を適用していない文字列 (図 5 の完全一致と同一) から算出されたハッシュ値であり，HASH_FOR_CLONE は正規化後の文字列から算出されたハッシュ値である．これらのうち，コードクローンの検出に用いられるハッシュ値は HASH_FOR_CLONE であり，HASH はコード片に何らかの修正が加えられたか否かを判別するために使用する．SIZE はこのコード片のサイズを表しており，現在のところそのコード片を構成する AST のノード数を表している．

CLONE_SET

CLONE_SET は互いにコードクローン関係にあるコード片の集合であるクローンセットをレコードとするテーブルである．ELEMENTS の型が TEXT* となっているが，これは要素の ID を “,” で連結した文字列であることを示している．例えばあるクローンセットが ID 1 のコード片と ID 2 のコード片から構成されている場合，ELEMENTS の値は “1,2” となる．

CODE_FRAGMENT_LINK

CODE_FRAGMENT_LINK は 2 つの連続するリビジョン間でのコード片のリンク (ブロックリンク) をレコードとするテーブルである．CHANGED はコード片がこのリンクよって結び付けられているコード片が，このリンクに関わるコミットで修正されたか否かを表しており，1 の場合は何らかの修正があったことを示している．修正されたか否かの判定には，CODE_FRAGMENT テーブルの HASH の値を使用しており，HASH が変化していれば修正されたものをみなしている．なお，コード片を含むファイルに変更が加えられなかった場合のブロックリンクは自明であるため，レコードとして保持されない．

CLONE_SET_LINK

CLONE_SET_LINK は 2 つの連続するリビジョン間でのクローンセットのリンク (コードクローンリンク) をレコードとするテーブルである．CHANGED/ADDED/DELETED_ELEMENTS はそれぞれ，変更，追加，削除されたコード片の数の合計を表している．

CLONE_GENEALOGY

CLONE_GENEALOGY はコードクローンの系譜を表すテーブルである．CLONES はこの系譜に含まれるすべてのクローンセットの ID を， CLONE_LINKS はこの系譜に含まれるすべてのクローンリンクの ID をそれぞれ表している．CHANGES/ADDITIONS/DELETIONS はそれぞれ，要素の変更，追加，削除が行われたリビジョンの合計を表している．また DEAD はその系譜が最終リビジョンにおいて生存しているかどうかを示しており，1 の場合は系譜が最終リビジョンにおいて生存していないことを示している．

CRD

CRD テーブルは CRD をレコードとするテーブルである．なお，1 つのブロックの CRD が 1 つのレコードとなる．TYPE はその CRD が表すブロックの型を，HEAD はその型のブロックから CRD を生成したときに先頭に配置される文字列をそれぞれ表す．ANCHOR はそのブロックの条件式などを表し，NORMALIZED_ANCHOR は正規化されたものを表している．CM はメトリクス値を示している．ANCESTORS はこのブロックの外側に位置するブロックに対する CRD の ID を連結したリストである．例えば，ID 1 の CRD を持つブロックの中に ID 2 の CRD を持つブロックがあり，ID 2 の CRD を持つブロックの中に ID 3 の CRD を持つブロックがあるケースにおける ID 3 の CRD を考えたとき，そのレコードの ANCESTORS の値は “1,2” となる．なおこのときの ID の順序は外側から順にブロックを辿ったときの順序となっている．FULL_TEXT はこの CRD を文字列表記したものであり，このブロックの外側に位置するブロックも考慮した文字列となる．

5.2 具体化後

ライブラリ呼び出しによって具体化された情報を保持するクラスは以下の通り．

- `CloneGenealogyInfo`
- `CloneSetLinkInfo`
- `CloneSetInfo`
- `CodeFragmentLinkInfo`
- `BlockInfo` extends `CodeFragmentInfo`
- `FileInfo`
- `RevisionInfo`
- `CRD`

以降，それぞれの構造について述べる．

5.2.1 `CloneGenealogyInfo`

フィールド名	型	意味
`startRevision`	`RevisionInfo`	系譜の開始リビジョン
`endRevision`	`RevisionInfo`	系譜の終了リビジョン
`clones`	`List<CloneSetInfo>`	系譜を構成するクローンセット
`links`	`List<CloneSetLinkInfo>`	系譜を構成するコードクローンリンク
`numberOfChanges`	`int`	要素が修正されたリビジョンの数
`numberOfAdditions`	`int`	要素が追加されたリビジョンの数
`numberOfDeletions`	`int`	要素が削除されたリビジョンの数
`dead`	`boolean`	true = 系譜が最終リビジョンで生存していない

5.2.2 CloneSetLinkInfo

フィールド名	型	意味
beforeRevision	RevisionInfo	前リビジョン
afterRevision	RevisionInfo	後リビジョン
beforeClone	CloneSetInfo	前クローンセット
afterClone	CloneSetInfo	後クローンセット
numberOfAddedElements	int	追加された要素の数
numberOfDeletedElements	int	削除された要素の数
numberOfChangedElements	int	変更された要素の数
fragmentLinks	List<CloneSetLinkInfo>	関連するブロックリンク

5.2.3 CloneSetInfo

フィールド名	型	意味
revision	RevisionInfo	リビジョン
elements	List<CodeFragmentInfo>	クローンセットを構成するコード片

5.2.4 CodeFragmentLinkInfo

フィールド名	型	意味
beforeRevision	RevisionInfo	前リビジョン
afterRevision	RevisionInfo	後リビジョン
beforeFragment	CodeFragmentInfo	前コード片
afterFragment	CodeFragmentInfo	後コード片
changed	boolean	true = コード片が修正された

5.2.5 BlockInfo

フィールド名	型	意味
ownerFile	FileInfo	所有ファイル
crd	CRD	CRD
startRevision	RevisionInfo	開始リビジョン
endRevision	RevisionInfo	終了リビジョン
startLine	int	開始行番号
endLine	int	終了行番号
size	int	サイズ
blockType	BlockType	ブロックの種類
node	? extends ASTNode	ブロックの AST ノード

5.2.6 FileInfo

フィールド名	型	意味
path	String	ファイルパス
startRevision	RevisionInfo	開始リビジョン
endRevision	RevisionInfo	終了リビジョン
node	CompilationUnit extends ASTNode	ファイルの AST ノード

5.2.7 RevisionInfo

フィールド名	型	意味
identifier	String	識別子

6 使用方法

ここでは，ECTEC の使用方法についてその詳細を述べる．なお，3 と同じく，事前分析機能のことを **Detector**，ライブラリ呼び出し機能のことを **Analyzer** と呼ぶ.

6.1 事前分析

3 で述べたように，Detector のメインクラスは jp.ac.osaka_u.ist.sdl.ectec.detector.DetectorMain である．

Detector には様々な引数を与えることができる．それらのうち，必須のもの 2 つを除いた引数については，プロパティファイルにデフォルト値が保持されているため，利用者が特に指定をしなければデフォルト値が読み込まれる．また，-p を用いることで読み込むプロパティファイルを指定することもできる．

なお，プロパティファイルはすべての任意引数に関する記述を持たなければならない．記述の無い引数が存在する場合は実行時にエラーとなる．

以降，Detector に与えることができる引数について述べる．

6.1.1 -r (必須)

引数名 (long version): –repository

意味: 分析対象リポジトリの URL

備考: サポートするプロトコルは，file，http，svn+ssh の 3 つ

使用例: `-r file:///G:/ECTEC/sample-repository`

6.1.2 -d (必須)

引数名 (long version): –db

意味: 分析結果保存先データベースファイルのパス

備考: 拡張子は “.db” を推奨

使用例: `-d G:\ECTEC\sample.db`

6.1.3 -a

引数名 (long version): –additional

プロパティファイルのキー: ectec.additional-path

意味: 分析対象の絞り込み

備考: 分析対象リポジトリの一部を分析したい時に使用する．例えば，file:///G:/ECTEC/sample-repository の /src のみを分析対象としたい場合，`-a /src` とすればよい.

使用例: `-a /src`

6.1.4 -l

引数名 (long version): –language

プロパティファイルのキー: ectec.language

意味: 分析対象言語

取り得る値: “java” (大文字小文字は区別しない)

備考: 現時点では Java にしか対応していない

使用例: `-l java`

6.1.5 -th

引数名 (long version): –threads

プロパティファイルのキー: ectec.threads

意味: スレッド数

取り得る値: 自然数

使用例: `-th 4`

6.1.6 -u

引数名 (long version): –user

プロパティファイルのキー: ectec.username

意味: ユーザ名

備考: ユーザ名・パスワードで制限がかけられているリポジトリにアクセスする際に使用

使用例: `-u hoge`

6.1.7 -pw

引数名 (long version): –password

プロパティファイルのキー: ectec.passwd

意味: パスワード

備考: ユーザ名・パスワードで制限がかけられているリポジトリにアクセスする際に使用．内部では平文でデータを保持するため注意．

使用例: `-pw hogehoge`

6.1.8 -s

引数名 (long version): –start

プロパティファイルのキー: ectec.start-revision-identifier

意味: 開始リビジョン

備考: 分析を開始するリビジョン．未指定の場合は一番初めのリビジョンから分析する．

使用例: `-s 100`

6.1.9 -e

引数名 (long version): –end

プロパティファイルのキー: ectec.end-revision-identifier

意味: 終了リビジョン

備考: 分析を終了するリビジョン．未指定の場合は一番最後のリビジョンまで分析する．

使用例: `-e 200`

6.1.10 -v

引数名 (long version): –verbose

プロパティファイルのキー: ectec.verbose-level

意味: 処理状況出力のレベル

取り得る値: (大文字小文字は区別しない)
- **強:** “strong”, “s”, “v”, “all” “a”.
- **弱:** “little”, “l”, “weak”, “w”, “default”, “d”, “yes”, “y”.
- **無:** “nothing”, “none”, “no”, “n”.

備考: 3 段階で指定可能．“強” は細かく経過を出力，“弱” は大まかな情報を出力，“無” は何も出力しない．各段階を指定するために複数の用語が使用できるが，同じ段階に属している用語であれば，どの用語を使っても動作に変化は無い．

使用例: `-v strong`

6.1.11 -vc

引数名 (long version): –version-control

プロパティファイルのキー: ectec.version-control-system

意味: 対象となるバージョン管理システム

取り得る値: “svn” (大文字小文字は区別しない)

備考: 現時点では Subversion にしか対応していない

使用例: `-vc svn`

6.1.12 -ow

引数名 (long version): –overwrite

プロパティファイルのキー: ectec.overwrite-db

意味: 出力先データベースがすでに存在する場合，上書きするかどうか

取り得る値: “yes” or “no” (大文字小文字は区別しない)

備考: yes を指定した場合，出力先データベースが存在していた場合はその内容を破棄し，新たに解析した結果を格納する．no の場合，出力先データベースが存在していれば実行を停止する．

使用例: `-ow yes`

6.1.13 -mb

引数名 (long version): –max-batch

プロパティファイルのキー: ectec.max-batch

意味: バッチ処理を行う際に一度に処理するステートメントの上限値

取り得る値: 自然数

使用例: `-mb 10000`

6.1.14 -ch

引数名 (long version): –clone-hash

プロパティファイルのキー: ectec.clone-hash

意味: クローン検出における正規化方法

取り得る値: (大文字小文字は区別しない)
完全一致 (正規化なし): “e”, “exact”
変数名・リテラルの正規化: “d”, “default”, “w”, “weak”
サブブロックの正規化: “s”, “strong”, “subtree”, “subblock”

使用例: `-ch exact`

6.1.15 -cs

引数名 (long version): –crd-similarity

プロパティファイルのキー: ectec.crd-similarity

意味: CRD の類似度算出方法

取り得る値: (大文字小文字は区別しない)
レーベンシュタイン距離ベース: “l”, “levenshtein”, “d”, “default”

備考: 現時点ではレーベンシュタイン距離ベースの方法のみ実装されている

使用例: `-cs l`

6.1.16 -fl

引数名 (long version): –fragment-link

プロパティファイルのキー: ectec.fragment-link

意味: コード片のリンク方法

取り得る値: (大文字小文字は区別しない)
Single: “s”, “single”
Multiple: “m”, “multiple”, “d”, “default”

使用例: `-fl multiple`

6.1.17 -st

引数名 (long version): –similarity-threshold

プロパティファイルのキー: ectec.similarity-threshold

意味: コード片をリンクする際の類似度の下限値

取り得る値: 0.0〜1.0 の実数値

使用例: `-st 0.7`

6.1.18 -g

引数名 (long version): –granularity

プロパティファイルのキー: ectec.granularity

意味: 分析対象とするブロックの粒度

取り得る値: (大文字小文字は区別しない)
Class: “c”, “class”, “f”, “file”
Method: “m”, “method”
Class-Method: “cm”, “class_method”, “larger_than_method”
All: “d”, “default”, “a”, “all”, “fine-grained”

備考: Class: クラスのみを対象とする．Method: メソッドのみを対象とする．Class-Method: クラスとメソッドを対象とする．All: すべてのブロックを対象とする．

使用例: `-g all`

6.1.19 -cst

引数名 (long version): –clone-size-threshold

プロパティファイルのキー: ectec.clone-size

意味: コードクローン検出を行う際の，コードクローンとして検出するブロックのサイズの下限値

取り得る値: 整数値

備考: 現時点ではサイズとして使用されるのは AST のノード数

使用例: `-cst 30`

6.1.20 -p

引数名 (long version): –properties

意味: 読み込むプロパティファイルへのパス

備考: デフォルト以外のプロパティファイルを読み込むときに使用

使用例: `-p G:\ECTEC\another.properties`

6.2 ライブラリ呼び出し

基本的な使用方法は 3 で述べた通りである．ここでは，`selectAndConcretizeCloneGenealogies` の返り値を用いる方法以外の具体化したデータへのアクセス方法と，系譜選択時の制約記述について述べる．

6.2.1 データへのアクセス

`selectAndConcretizeCloneGenealogies` を呼び出してデータを具体化した際，具体化されたすべてのデータは `DataManagerManager` に登録され，管理される．

`GenealogyAnalyzer` の `getDataManagerManager` メソッドを呼び出し，さらにそこから適切なメソッドを呼び出すことで，具体化されたデータへのアクセスを行うことができる．

また，`selectAndConcretizeCloneGenealogies` を用いる方法以外のデータの具体化方法として，具体化したい系譜の ID を直接指定する方法も提供されている．`concretizeCloneGenealogy` メソッドに ID を指定することで，指定した ID を持つ系譜を具体化することができる．

6.2.2 制約記述

制約記述には `IConstraint` 型のインスタンスを用いる．`Iconstraint` はインターフェースであり，`satisfy(DBCloneGenealogyInfo)` というメソッドを提供している．このメソッドは引数で受け取ったコードクローンの系譜が具体化の対象となる場合に true を返すメソッドである．

このインターフェースを実装したクラスは，現在のところ以下の 5 つである．

- `RevisionRangeConstraint`
- `NumberOfCloneSetsConstraint`
- `LaxConstraint`
- `AndConjunction`
- `OrConjunction`

`RevisionRangeConstraint` はリビジョン期間に関する制約を設けるためのクラスである．開始，終了リビジョンを指定することで，その期間に存在していた系譜を取得することができる．また，`mustBeComprised` を true とすれば，指定した期間内に生まれ，指定した期間内に消滅した系譜を取得することができる．

`NumberOfCloneSetsConstraint` は系譜を構成するクローンセットの数に関する制約を設けることができるクラスである．上限値と下限値をそれぞれ指定することができる．

`LaxConstraint` は常に true を返す `satisfy` メソッドを実装している．このクラスはすべての系譜を取得する際に使用される．`selectAndConcretizeCloneGenealogies` に null を指定した場合も，内部ではこのクラスのインスタンスが生成されている．

`AndConjunction` は 2 つの制約の and を取った制約であり，2 つの制約を共に満たす場合に true を返す `satisfy` メソッドが実装されている．同様に，`OrConjunction` は 2 つの制約の or を取った制約である．`AndConjunction`，`OrConjunction` はネストさせることが可能であるため，これらを組み合わせることで複雑な制約を記述することができる．

これらを使って `IConstraints` 型の実装クラスのインスタンスを作成し，それを用いることで，制約を満たす系譜のみを具体化することができる．

参考文献

[1] Ekwa Duala-Ekoko and Martin P. Robillard. Clone Region Descriptors: Representing and Tracking Duplication in Source Code. *ACM Transactions on Software Engineering and Methodology*, 20(1):3:1–3:31, June 2010.

[2] Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Enhancement of CRD-based Clone Tracking. In *Proceedings of the 13th International Workshop on Principles on Software Evolution (IWPSE 2013)*, pages 28–37, Aug. 2013.