リポジトリマイニング向けドメイン専用言語 ArgyleJの開発と実証的評価

山下 一寛[1,a)]　亀井 靖高[1,b)]　久住 憲嗣[1,c)]　鵜林 尚靖[1,d)]

概要：本稿では，リポジトリマイニングにおけるデータ取得・加工の効率化を目指して，リポジトリマイニング向けのドメイン専用言語・ArgyleJ を設計・実装する．ドメイン専用言語の設計・実装には，リポジトリマイニングに対するドメインの理解が必要である．そこで，設計・実装の前段階として，リポジトリマイニングに対するフィーチャ分析を行い，そのフィーチャモデルに基づき言語の設計・実装を行う．ArgyleJ の有用性を評価するために，ケーススタディとして具体的なリポジトリマイニングにおけるデータ取得・加工のシナリオを設定し，ArgyleJ と Java 言語それぞれで例題を実装した．オープンソースプロジェクトである Eclipse と Mozilla を題材としてケーススタディを行った結果，ArgyleJ は，Java で実装する場合と比べて，高い記述力を持つことが示された．また，編集距離という観点から評価した場合，ArgyleJ の変更容易性は Java と比べて高いことが示された．

An Empirical Study of ArgyleJ: Domain-Specific Language for Mining Software Repositories

Kazuhiro Yamashita[1,a)]　Yasutaka Kamei[1,b)]　Kenji Hisazumi[1,c)]　Naoyasu Ubayashi[1,d)]

Abstract: To improve the effectiveness of the process for collecting and pre-processing data from software repositories, we design and implement a domain specific language (DSL), named ArgyleJ (A repository mining query language for Java), for mining software repositories. To design and implement the DSL, domain knowledge for mining software repositories is required. Therefore, first, we perform a feature analysis to the domain of mining software repositories, then we design and implement the DSL based on the feature model. To evaluate the effectiveness of ArgyleJ, we implement one of the representative case studies of an MSR analysis by using both of ArgyleJ and Java. The case study using Eclipse and Mozilla projects shows that ArgyleJ outperforms Java to collect and pre-process data from software repositories.

1. まえがき

現在，ソフトウェア開発では企業での開発，オープンソースでの開発に関わらず，そのプロジェクトで計測可能な多くの情報 (ソースコード，バージョン管理情報，開発者間でやり取りされたメール等) がソフトウェアリポジトリ[*1]に保管されている．リポジトリに対してデータマイニングの技術を用いて分析すること（以降，リポジトリマイニング）で，ソフトウェア開発における有用な知見を得ることができる [1]．例えば，Shihab ら [2] は，変更されて間のない規模の大きいファイルには，不具合が含まれている可能性が高い，という結果をリポジトリマイニングを通して明らかにしている．

リポジトリマイニングは，1) リポジトリからのデータの取り出し，2) データの加工／データ形式の変換（標準化），3) データの分析の手順で行われる [3]．つまり，知見の獲得のために行いたい分析工程に対する準備として，データの取り出し，および，データ形式の変換を行う必要がある．それら準備工程において大半の場合は，リポジトリマイニングを行っている実務者／研究者おのおのがデータの取り出しやデータ加工のプログラムを汎用プログラミング言語で実装しており，実務者／研究者の多くの手間が費やされ

1 九州大学
Kyushu University
a) yamashita@posl.ait.kyushu-u.ac.jp
b) kamei@ait.kyushu-u.ac.jp
c) nel@slrc.kyushu-u.ac.jp
d) ubayashi@ait.kyushu-u.ac.jp

*1 情報が保管されているハードディスク等のストレージをリポジトリと呼ぶ

© 2012 Information Processing Society of Japan

ている．この現状の課題を起こす原因の 1 つは，準備工程の作業で用いられる汎用プログラミング言語には，リポジトリマイニングで繰り返し利用される処理を簡単に扱う仕組みがないことであると考える．

そこで本論文では，リポジトリマイニングにおけるデータ取得・加工の効率化を目指して，リポジトリマイニング向けのドメイン専用言語: ArgyleJ(A repository mining query language for Java) を提案し，その設計と実装を行う．ドメイン専用言語を用いることにより，スクリプトの編集距離が削減され，可読性が上昇することで，問題となっている準備工程の時間削減につながると期待される．まず，ドメイン専用言語の構築にはドメイン知識の獲得が必要であるため，リポジトリマイニングに対するフィーチャ分析を行う．フィーチャ分析では，リポジトリマイニングの先行研究 [3][4][5][6] に対して，Kang らの分析方法 [7] を適用した．次に，フィーチャモデルに基づき，ドメイン専用言語の設計，および，実装を行う．本論文では，任意の N 項演算子を組み合わせることで簡単に言語を実装できることや，型チェックに利用できる等の理由から LasticJ[8] を用いて言語機能を実装した．本言語は，Java 言語を拡張し内部ドメイン専用言語として実装することで，Java 言語の機能を利用することもできる．

そして，ArgyleJ がリポジトリマイニングにおけるデータ取得・加工を効率化しているかを確認するために実験を行った．実験では，具体的な例題を設定し，オープンソースプロジェクトである Eclipse プロジェクト，Mozilla プロジェクトのソースコードリポジトリに対して解析を行った．

以降，2 章でリポジトリマイニングとフィーチャ分析の概要を述べ，3 章でリポジトリマイニングに関するフィーチャモデルと，リポジトリマイニング向けドメイン専用言語の設計・実装について述べる．4 章でケーススタディについて報告し，5 章で考察を述べる．6 章で関連研究をまとめ，最後に 7 章で本論文のまとめと今後の課題を述べる．

2. リポジトリマイニング

2.1 リポジトリマイニング概要

リポジトリマイニングとはソフトウェア開発のプロジェクトでの様々な計測可能な情報を格納したリポジトリに対して，データマイニングの技術を用いて解析し，その後のソフトウェア開発に役立つ種々の有用な知見を得る技術である [9][10]．一般的に，リポジトリマイニングの工程は，1. リポジトリからのデータの取出し，2. データの加工・データ形式の変換（標準化），3. データの分析，の 3 つに分類される [3]．これらの工程のうち，知見を獲得する工程は 3 番目の “データの分析” であるが，分析を行うためには準備として，1，2 番目の工程のデータの取得，加工，変換という操作をする必要があり，この準備工程に多くの時間が費やされているのが現状である [11]．

データの取得，加工，変換という工程の多くの部分では，リポジトリマイニングを行う実務者／研究者それぞれが汎用プログラミング言語を用いてスクリプトを実装している．そのため，分析の準備段階でしかないデータ取得・加工の工程に多くの時間を取られているという問題が存在する．この問題の原因は，リポジトリからのデータの取得，リポジトリマイニングで行う解析に合わせたデータ加工といった作業を簡単に行う枠組みや言語が存在しないためである．そこで本論文ではこれらの作業を効率化するためのドメイン専用言語を提案する．

2.2 リポジトリマイニングを対象としたフィーチャ分析

ドメイン専用言語の構築には，対象ドメインの深い知識が求められる．そこで，本論文では ArgyleJ を構築する準備として，リポジトリマイニングに対するフィーチャ分析を行い，リポジトリマイニングのフィーチャモデルを作成する．フィーチャモデルとは，各機能の可変性を示すことで，各機能が再利用できるか否かを示すものである [7][12]．

フィーチャ分析は，主に (1) ドキュメントの収集，(2) フィーチャの選別と階層分け，(3) フィーチャの抽象化，(4) フィーチャ間の関係の定義の計 4 つのプロセスで行われる．図 1 に 4 つのプロセスによって得られたモデルを示す．各プロセスについて本論文で行った作業を説明する．

1. ドキュメントの収集. ドキュメントの収集では，リポジトリマイニングに関する論文の一例として文献 [3][4][5][6] を選択し，リポジトリマイニングの作業について調べた．

リポジトリマイニングの実施／研究目的は，大きく 8 つ（静的解析，コードクローン検出やソフトウェアメトリクス）に分類される [13] が，本論文では，メトリクスの計測を主なリポジトリマイニングの作業分野として選んだ．これは，筆者らがリポジトリマイニング分野の研究でも，特にメトリクスの研究に従事しており [2][5]，対象ドメインの知識の獲得を他の作業分野より容易にできると考えたためである．

2. フィーチャの選別と階層分け. 本プロセスでは，ドキュメントから候補となるフィーチャを選別し，それらフィーチャを 4 つのレイヤー (Capability，DSL，Domain Technology，Implementation Technique) に振り分け，階層化する．図 1 に示された各フィーチャについては本論文で提案する言語要素と深く関連するため，3 章で詳細を述べる．

3. 選別されたフィーチャの抽象化・具象化. フィーチャの抽象化・分類では，フィーチャ間の関係や構造を mandatory（必須），optional（選択），or（１つ以上），alternative（どれか１つ）の 4 つに仕分け，各フィーチャの抽象化，および，具象化を行った．例えば，Capability レイヤーの `ProductMetrics` は，`getSloc` や `getCycComplexity` と or の関係で定義される．

4. フィーチャ間の関係の定義. フィーチャ間の関係の定義では，モデルで表現できない各フィーチャ間の関係を図 1 の

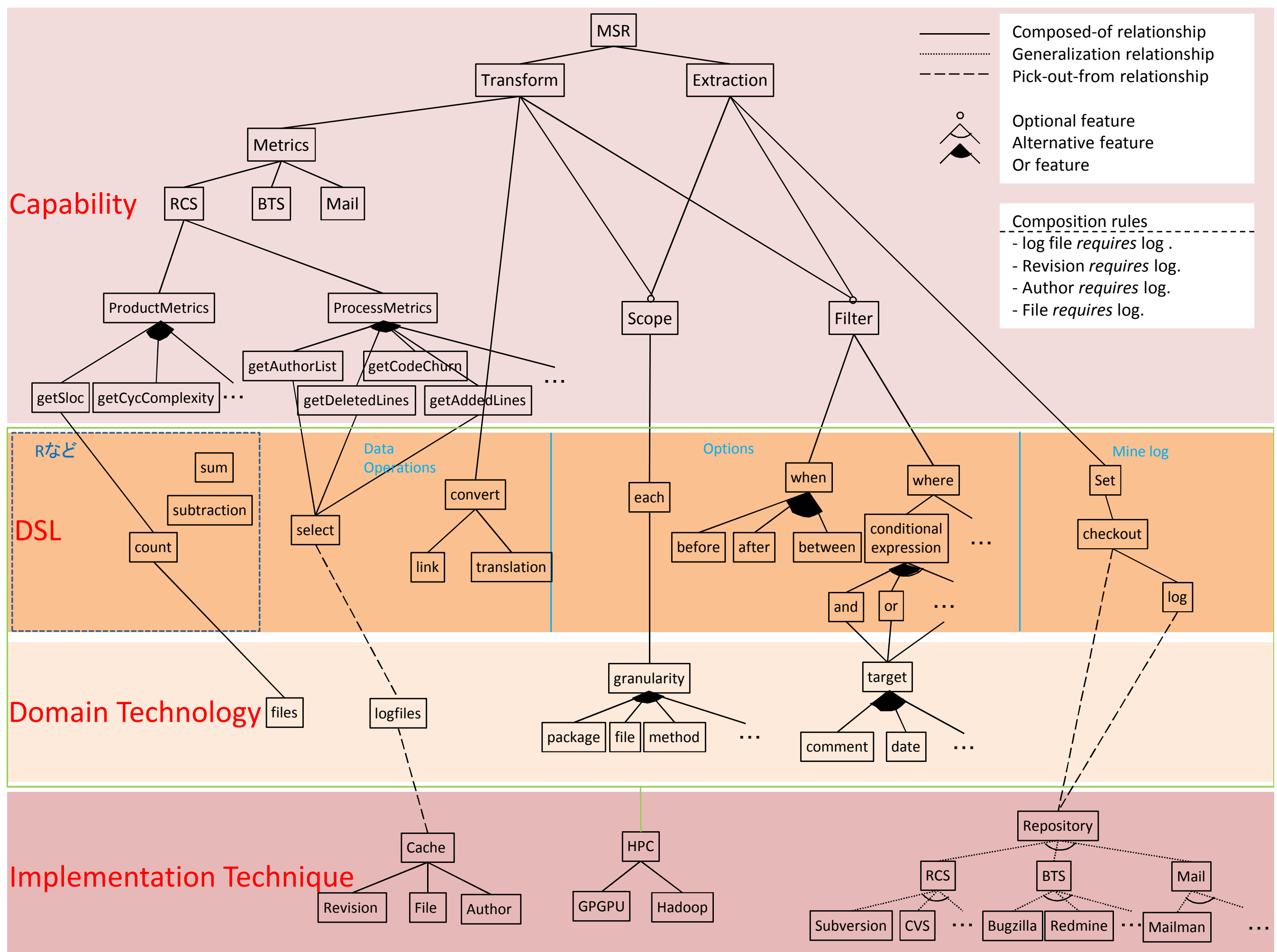

図 1 リポジトリマイニングのフィーチャモデル

Fig. 1 Feature Model of Mining Software Repositories

右上に Composition rules として示した．この Composition rules によって，DSL を定義する際の事前条件について調べることができる．

3. ArgyleJ の言語設計

ArgyleJ はリポジトリマイニングにおけるデータ取得・加工の効率化を目指す．その上で，スクリプトの記述量の削減や可読性の向上はもちろん，種々のリポジトリを簡単に扱えるということも考え設計を行った．ArgyleJ の基本的な構文は，前章で扱ったフィーチャ分析の成果物であるフィーチャモデル (図 1) に基づいて構築する．

3.1 リポジトリマイニングを対象としたフィーチャモデル

本節では，言語の構築の際に用いるフィーチャモデル (図 1) の詳細について説明する．得られたフィーチャは，4 つのレイヤー (Capability，DSL，Domain Technology，Implementation Technique) に振り分けられ，階層化されている．Capability レイヤーは，エンドユーザから見えるアプリケーションのフィーチャを定義するレイヤーである．本論文では，リポジトリからデータを取り出す機能として `Extraction`，取り出したデータを分析が容易な形に変える機能 `Transform` [3] を定義した．また，`Transform` を具体化したフィーチャとしてメトリクスの収集に関する機能 `Metrics` を定義し，`RCS`，`BTS`，`Mail` という具体的なリポジトリマイニングの分析対象ごとにフィーチャを定義した．そして，`Extraction`，`Transform` のオプションとして，収集するデータの粒度（パッケージ単位やファイル単位）を選択する機能 `Scope`，および，任意の条件で収集するデータを選択する機能 `Filter` を定義した．

Kang ら [7][12] の定義では，Operating Environment レイヤーは，Capability レイヤーのフィーチャを実行するための環境である．リポジトリマイニングでは，リポジトリを操作するユーザに対して，その操作結果（メトリクスの結果など）をフィードバックすることに相当する．つまり，ドメイン専用言語の各オペレーションに相当すると考え，DSL レイヤーとした．例えば，`Transform` を実現するために，`select` や `convert` などのフィーチャを定義した．また，任意の条件で収集するデータを選択する `Filter` のオペレータとして，`where`，`when` を定義した．

Domain Technology レイヤーは，オペレーションを実現す

© 2012 Information Processing Society of Japan

表 1　ArgyleJ の演算子
Table 1　Operators in ArgyleJ

分類	フィーチャ	演算子	演算子の型	引数の型	用途
Mine Log	set	**set** *project_name*	Project 型	String 型	対象を設定
	checkout	**checkout** *project*	CO 型	Project 型	ファイル取り出し
	log	**log** *co*	Log 型	CO 型	ログ取り出し
Data Operations	select	**select** *metrics* **from** *log*	ASQL 型	String 型，Log 型	メトリクスの取得
Options	each	...**each** *granularity*	Option	Granularity	粒度を指定
	where	...**where** *condition*	Option	Condition	条件を指定
	before	...**before** *date*	Option	String 型	ある日付以前を指定
	after	...**after** *date*	Option	String 型	ある日付以降を指定
	between	...**between** *date* **and** *date*	Option	String 型，String 型	ある期間を指定

る方法に関するものとして，本論文では DSL の機能を実現するためのメタモデルに相当すると考えた．例えば，リポジトリに格納されているファイルを表す `files`，各ファイルの変更履歴を表す `logfiles` などがある．Implementation Technique レイヤーは，Operating Environment レイヤー（本論文では DSL レイヤー）の機能を実現する際に使用される一般的な機能とされるので，各種リポジトリが該当すると考えた．例えば，版管理システムとして CVS や SVN，障害管理システムとして Bugzilla や JIRA などがあげられる．

また，ドメイン専用言語と連携する技術として `GPGPU` や `Hadoop` などのハイパフォーマンスコンピューティングの技術を Implementation Technique レイヤーに，また，リポジトリから抽出したメトリクスに対して外部の統計解析ツール（R や weka 等）を適用するためのフィーチャとして，`sum` や `subtraction`，`count` を DSL レイヤーに定義した．

本節で述べた，フィーチャモデルはまだ十分にリポジトリマイニングについて表現しているとはいえず，網羅性や十分性の点で問題があると考えられる．そのため，7 章でも述べる通り，今後モデルについても改善していく必要があると考える．

3.2　構文要素

ArgyleJ の言語要素は，図 1 の DSL レイヤー中の機能に該当する．図 1 の DSL レイヤーのフィーチャは以下のように，大きく 3 つに分類される．

(1) リポジトリを操作する機能 (Mine log) — 工程 1：抽出に該当
(2) データを操作する機能 (Data Operations) — 工程 2：加工・変換に該当
(3) 粒度や条件を指定する機能 (Options)

表 1 にフィーチャとそれに対応する実装した演算子，演算子の型，引数の型を示す．演算子の太字のフレーズがキーワードで，イタリック体のフレーズが引数である．演算子の型は，演算子が処理後に返す返り値の型を示し，引数の型は引数がとる型を示す．

リポジトリを操作する機能に分類される演算子には，対象となるプロジェクトを設定する `set` や，対象プロジェクトのファイル群を取り出す `checkout`，対象プロジェクトの変更履歴（ログ）ファイル群を取り出す `log` などが該当する．これらの演算子は対象となるリポジトリにアクセスをして，ファイル本体やログ情報といったリポジトリに格納されている情報を取り出す．

データを操作する機能に分類される演算子には，`select` がある．`select` は対象のプロジェクトのデータを操作し，例えば，変更履歴データ（ログデータ）から任意の条件のデータを取り出す．また，図 1 の DSL レイヤー中にある `convert` は ArgyleJ と他システムを連携させたり，複数リポジトリを扱うことを想定したフィーチャである．具体的には，`checkout` で取り出してきたデータを外部の統計解析ツール（R や weka など）で受け取れる CSV や XML といった形式に成形する `translating` や，`RCS` の中でも git や mercurial といった分散型のソフトウェアリポジトリや，Bugzilla や JIRA などの `BTS`，Mailman などのメールリポジトリと連携することを想定した `link` を定義した．これらのフィーチャに対する構文要素は今後定義する予定である．

最後に，粒度や条件を指定する機能に分類される演算子には，`each` や `where`，`when` といったものがある．これらの演算子はリポジトリを操作する機能やメトリクスを取り出す機能のオプションとしての役割を果たす．これらを用いることによって，ファイルの粒度や対象となるリポジトリ，高速化の技術，期間，条件を指定することができる．

3.3　ArgyleJ の記述例

List 1 に ArgyleJ の簡単な記述例を示す（具体的な記述例はケーススタディの章で示す）．List 1 では，1 行目から順に 1：対象プロジェクトの設定，2：ファイルを抽出，3：ログデータを抽出，4：求めたいメトリクス（表 2 では addedline）を取り出す．また，この例ではオプションとして 2000 年 1 月 1 日以前のデータを抽出するというオプションが付いている．

3.4　ArgyleJ の実装

ArgyleJ は，LasticJ [8] と JavaDB を使って実装されており，現段階では ArgyleJ を利用することで以下のことが簡単な記述で行える．

- リポジトリからファイル群を取り出す．
- ファイル群に関するログデータを取り出す．

© 2012 Information Processing Society of Japan

```
[List 1]
01: Project ex = set "example";
02: CO exco = checkout ex;
03: Log exlog = log exco;
04: ASQL rs = select addedline from exlog before
    "2000/01/01";
```

```
[List 2]
01: operators DSLOperators{
02:   Project ``set'' project_name (String project_name) :
      priority = 100{
03:       ・・・
04:       return new Project(project_name);
05:   }
06:
07:   CO ``checkout'' project (Project project) :
      priority = 100 {
08:       ・・・
09:       return new CO(project);
10:   }
11: }
```

表 2 各テーブルのスキーマ
Table 2 Table Scheme

テーブル名	テーブルの要素	要素の説明
Revision Table	revision_id	各ファイル，リビジョンの ID
	revision_num	リビジョンの番号
	file_id	ファイルの ID
	author_id	開発者の ID
	date	コミットされた日付
	addedline	追加行数
	deletedline	削除行数
	comment	コミット時のコメント
File Table	file_id	ファイルの ID
	file_name	ファイル名
	directory	ディレクトリ名
	full_path	ファイルのフルパス
	extension	拡張子
Author Table	author_id	開発者の ID
	author_name	開発者名
	email	開発者のメールアドレス

- ログデータからメトリクスを取り出す．

LasticJ は，内部 DSL の作成を支援するシステムである．LasticJ では，型によって字句・構文ルールを切り替える手法をとり，N 項演算子によって字句・構文ルールを記述する．実装に LasticJ を用いた理由は，LasticJ は型チェックを行っているので，コード生成の高速化や型推論による記述量の削減などに寄与すると考えたためである．

List 2 は LasticJ を用いた ArgyleJ の実装の一部であり，下線部が表 1 と対応している．List 2 において太線で書かれている語句は，LasticJ によって提供されている．1 行目の operators 宣言は，N 項演算子の定義であるという宣言である．priority は，その N 項演算子の優先度を示す値で数値が大きいほど優先度が高い．また，優先度は省略することも可能である．省略された場合は優先度は 0 になる．4 行目の return，new は Java での意味と同様である．

2 行目からが N 項演算子の定義文となり，返り値の型，演算子のパターン，オペランドの型，演算子の優先順位，N 項演算子の処理内容 (Java で記述) の順に記述する．

また，高速化実現のため，初回起動時に 3 つのテーブル（表 2）を作成し，解析したデータをキャッシュとして保存する．2 回目以降は，リポジトリから取り出さず，キャッシュされたテーブルから各データを取り出す．このように実装することで，求めたいメトリクスを追加する場合などのインクリメンタルに実験を行う際，キャッシュされているテーブルからメトリクスを取り出すため，実行時間が短縮される．

変更履歴を解析して得られたデータはそれぞれ “Revision”，“File”，“Author” に記録され，“Revision” テーブルの各行は，各ファイル・リビジョンごとに作られ，“File” テーブルの各行はファイル名ごとに作られ，“Author” テーブルの各行は author 名ごとに作られる．

4. ケーススタディ

4.1 概要

本論文では，リポジトリマイニングにおけるデータの取得・加工の工程の効率化を目指し，リポジトリマイニング向けドメイン専用言語 ArgyleJ を提案した．本章では，オープンソースプロジェクトである Eclipse，Mozilla のソースコードリポジトリに対して，3 つタスクを設定し，ドメイン専用言語としての ArgyleJ の有用性を評価する．

本論文における各タスクでは，それぞれ Java によるプログラムと ArgyleJ を用いた場合のプログラムに対して，スクリプトの記述量，変更容易性，プログラムの実行時間の観点から評価を行った．ここで，Java によるプログラムとは，何もない状態から記述した場合のことを指す．

4.2 例題設定

ArgyleJ を評価する例題として，不具合モジュール予測[*2][5][14][15] に用いるためのデータの作成を取り上げる．その理由は，不具合モジュール予測はリポジトリマイニング研究の 1 つとして広く取り組まれているテーマであるからである．今回の例題では，追加行数，削除行数，および，リリース前に行った不具合の修正回数を計測対象とした．具体的な計測手順は以下の通りである．

(1) CVS からファイルの一覧を取り出す．
(2) ファイルごとの変更履歴を取り出す．
(3) 変更履歴に記述されている追加／削除行数を取り出す．
(4) 変更履歴のコメント部分から不具合修正を行ったか否かを表すキーワードを調べる．本論文では，キーワードとして，“bug(s)”，“bugfix(es)”,“defect(s)” を選んだ．
(5) ファイルごとの追加行数，削除行数，および，リリース前に行った不具合の修正回数を求める．

タスク 1. 本タスクでは，ArgyleJ を用いることで，従来

[*2] モジュールの特性値（ソースコード行数や分岐の数，サイクロマティック数など）を説明変数とし，モジュールの不具合の有無を目的変数とする数学的モデル

表 3　各言語での記述量と編集距離
Table 3　LOC and Levenshtein Distance for Java and ArgyleJ

言語	記述量 (行数)	編集距離
Java	213	71
ArgyleJ	24	18

の場合（Java で実装する場合）と比べてどの程度記述量を削減できるかを調べるために，例題の工程を Java，および，ArgyleJ でそれぞれ実装する．評価には，実装に要した行数を用い，空行や括弧のみの行や import 文などの宣言文は行数に加えないものとする．

タスク 2.　タスク 2 では，タスク 1 のスクリプトを拡張して，メトリクスを算出する期間を指定する機能を実装する．今回は変更容易性の観点から評価するためタスク 1 と同じメトリクスを取り出す．対象とする期間を指定するためのスクリプトを実装し，タスク 1 のスクリプト間の編集距離を Java，ArgyleJ それぞれで比較し，評価する．

編集距離として本実験では，レーベンシュタイン距離 [16] を用いる．レーベンシュタイン距離とは文字列の類似度を測るアルゴリズムであり，挿入・削除・置換が一度行われるごとに，距離が 1 増えるといった距離である．例えば，“disk” と “risk” のレーベンシュタイン距離は 1 となり（“disk” の d を r と置換），“apple” と “play” のレーベンシュタイン距離は 4 となる（1．“apple” の a を削除，2．2 文字目の p を削除，3．e を a に置換，4．最後尾に y を追加）．

タスク 3.　ArgyleJ ではインクリメンタルな分析における高速化を支援する仕組みを実装した．本タスクでは，高速化がどの程度達成されているかを調べるために，実行時間に関して評価を行う．キャッシュが存在する場合としない場合，つまり初回起動時と 2 回目以降でどのくらい高速化されたかを比較評価する．また，データセットのサイズが実行時間にどの程度影響するかを調べるために，複数のサイズを用いて実験を行う．

実行時間として本実験では，工程 (1) のプログラム開始時から工程 (5) の各メトリクスをファイルに出力するまでの時間をミリ秒単位で計測する．

4.3　タスク 1 の結果と考察

タスク 1 のコード行数を表 3 の左に示す．表 3 に示すとおり，Java で記述する場合 200 行程度かかるプログラムが，ArgyleJ では 20 行程度で記述できることがわかる．ArgyleJ のコード（List 4）は，計測手順 (1)-(5) すべてに関するコードであり，一方，Java のコード（List 3）は，計測手順の (2)-(5) の一部を抜粋している．下線が引かれている部分はタスク 2 において拡張される部分を示す.

List 3 に示すとおり，Java では CVS リポジトリから取得した変更履歴を解析するなどの定型処理（例えば，13 行目から 15 行目では，変更履歴から日付に関する部分を取得する）を記述する必要があり，多くのソースコードの記述が求められる．その一方で，List 4 の ArgyleJ ではそれら定型処理が言語化されているため，少ないソースコードで記述することができる.

表 4　実行時間:Eclipse
Table 4　Execution time for Eclipse

モジュール名	データサイズ	変更回数	実行時間（ms）		
			Java	存在	存在しない
ui.cheatsheets[*3]	3.1MB	1,721	1,256	950	19,610
e4-incubator	46MB	7,366	15,274	990	86,368
swt.wpf.win32.x86[*3]	163MB	1,205	4,235	924	19,212
e4	214MB	18,249	96,691	1,057	107,004
releng.basebuilder[*3]	2.6GB	43,368	116,884	1,174	240,848

4.4　タスク 2 の結果と考察

タスク 2 の結果を表 3 の右を示す．また，期間を指定するための拡張部分を，List 3 と List 4 それぞれの下線部分として示す．表 3 の通り，ArygleJ の編集距離は，Java と比較するとソースコードの記述量と同様に，削減されていることがわかる.

Java のプログラム（List 3）では，対象期間を指定するスクリプトを実装するために，日付を表す文字列を文字列型から日付型に型変換し（List 3 の 34 行目から 36 行目），指定された期間内であるか否かを比較するための if 文を追加する必要がある（List 3 の 37 行目）．一方で，ArgyleJ を用いる場合，データを取得するコードの末尾にオプションとして，日付を指定する演算子を追加するのみで実装できる（List 4 の 13 行目）.

4.5　タスク 3 の結果と考察

タスク 3 では, Java のプログラムの場合, ArgyleJ でキャッシュが存在しない場合（初回起動時）とキャッシュが存在する場合（2 回目以降）においての実行時間の比較を行った．表 4 と 5 に示す通り，キャッシュがない場合はデータサイズや変更回数が増えるにしたがって，実行時間も増加している.

Java のプログラムでは，データベースにデータを挿入することなく，メトリクスを取り出しているので，キャッシュが存在しない場合では ArgyleJ を用いる場合のほうが実行時間は遅くなっている．しかしながら，キャッシュが存在する場合は，ある程度，変更回数の大きいモジュールに関しては，Java のプログラムよりも速くメトリクスを取り出せていることがわかった．例えば，Mozilla プロジェクトの atwork モジュール（156KB）を扱う場合，キャッシュが存在する場合よりも Java のプログラムのほうが速くメトリクスを取り出せている．しかしながら，buildtools モジュール（8.7MB）を扱う場合は，Java のプログラムよりもキャッシュがある場合の方が速く取得できることがわかった.

[*3]　モジュール名に本来 “org.eclipse” が付くが表のスペースの都合上

© 2012 Information Processing Society of Japan

```
[List 3:  Javaによる実装の一部]
01: while((line = br.readLine()) != null){
02:  // 各メトリクスを抽出する
              ・・・
13:  if(line.indexOf("date: ") > -1){
14:    date = line.substring(line.indexOf("date: ")+6,
       line.indexOf("date: ")+26);
15:  }
              ・・・
29:  if(flag2 && ((line.indexOf("------") >= 0) ||
       (line.indexOf("=========") >= 1))){
30:    // バグ修正か否かの正規表現（内容は割愛）
31:    Pattern p = Pattern.compile(・・・);
32:    Matcher m = p.matcher(comment);
33:    if(m.find()){
       // 指定された日付以前か否か
34:      DateFormat df = new SimpleDateFormat(・・・);
35:      Date formatDate = df.parse(date);
36:      Date compareDate = df.parse("2000/01/01 00:00:00");
37:      if(formatDate.compareTo(compareDate) <= 0){
38:        pw.println(・・・);
39:      }
40:    }
41:    if(line.indexOf("============") >= 0){
42:      filename = "";
43:    }
44:  }
45:  if(flag == true){
46:    comment += line;
47:  }
48: }
```

表 5　実行時間:Mozilla

Table 5　Execution time for Mozilla

モジュール名	データサイズ	変更回数	実行時間（ms）		
			Java	存在	存在しない
atwork	156KB	42	132	841	2,175
vtable_hide	180KB	42	353	817	2,170
l10nkits	1.1MB	150	469	899	4,063
buildtools	8.7MB	691	1628	806	2,172

5. 考察

本論文では，分析によって得られたフィーチャモデルから構文要素を洗い出し，それらを内部 DSL として設計，および実装した．その一方で，今回，内部 DSL としてまとめた部分を，再利用可能な部品群（ライブラリ）として実現する方法も存在する．ライブラリ化して本タスクを実行した場合，スクリプトの記述量は 22（ArgyleJ では 24），変更容易性は 48（ArgyleJ では 18），実行時間は再利用無しの Java とほぼ同じであった．

なお，ArygleJ では，LasticJ で拡張した内部 DSL から Java で実装されたライブラリを呼び出す形式を採用している．そのため，利用者は，ArygleJ を内部 DSL としてだけではなく，ライブラリとして利用することも可能である．内部 DSL とライブラリは，それぞれメリット／デメリットがある（例えば，内部 DSL を採用することで，エディタへの組み込みが可能となるが，従来のエディタでエラーが起きる可能性がある）ため，今後，どちらの方式を採用するかについては検討する必要がある．

割愛している．

```
[List 4:  ArgyleJ による実装]
01: public static void main(String[] args){
02:  //前処理
03:  Project project = set args[0];
04:  Connection con = getConnection;
05:
06:  // (1) CVS からファイルの一覧を取り出す（チェックアウト）
07:  CO exco = checkout project;
08:
09:  // (2) ファイルごとの変更履歴を取り出す
10:  Log logco = log exco;
11:
12:  // (3), (4) の実施
13:  ASQL rs = select "revision_id,addedline,deletedline,
       comment" from logco before "2000/01/01" ;
14:  ResultSetMetaData meta = rs.getMetaData();
15:  // 求めるメトリクスの数の設定
16:  int columnCount = meta.getcolumnCount();
17:  // 出力先の設定
18:  File result = new File(result.csv);
19:  PrintWriter pw = new PrintWriter(new BufferedWriter
       (new FileWriter(result)));
20:  // バグ修正か否かの正規表現（内容は割愛）
21:  Pattern p = Pattern.compile(...);
22:
23:  // 各メトリクスを参照
24:  while(rs.next()){
25:   Matcher m = p.matcher(rs.getString(5));
26:   if(m.find()){
27:    // (5) メトリクスを求める
28:    pw.println((rs.getString(1) + "," + rs.getString(2) +
        "," + rs.getString(3));
29:   }
30:  }
31:  Connection co = shutdown;
32: }
```

6. 関連研究

本研究と同様に，ドメイン専用言語を利用してソフトウェア開発の支援を目指した研究は今までに提案されている [17]．例えば，Wang らの研究 [17] では，the Dependency Query Language(DQL) というドメイン専用言語を作成し，開発者がメンテナンスや，拡張を加える際に，コードクローンやコードの類似した部分を発見する支援を行っている．DQL を用いることによって，ユーザはプログラムの System Dependence Graph(SDG) 上の依存状態やテキスト状態を含むクエリを作ることができ，コードの類似部分や依存度の高い場所を知ることができる．本研究では，Wang らの研究と同様にドメイン専用言語を用いることで，リポジトリマイニングを行う実務者，および，研究者を支援することを目的としている．

本研究と同様に，リポジトリマイニングの支援を目的とした研究も行われている [3]．例えば，Shang らの研究 [3] では，リポジトリマイニングの前段階のデータ準備という工程に着目し，クラウドベースの環境である Pig や Hadoop を用いて，巨大なデータに対していかに高速にマイニングを行うかについて評価している．本研究では，一般的な，リポジトリマイニングのデータの準備・加工を支援する．将来的には，Hadoop や GPGPU といった高速化技術も取り込み，ArgyleJ で実装したコードを自動的に Hadoop や GPGPU のコードに変換する仕組みを構築する予定である．

また，メトリクスを算出する仕組みとして，Matsumoto らの研究 [6] がある．Matsumoto らの研究では，Web サービスの形でメトリクスの算出を行う枠組みを提供してい

© 2012 Information Processing Society of Japan

る．一方，本研究では，リポジトリマイニング向けのドメイン専用言語を提供することで，データ加工における実装に要する時間の短縮を目指した．Matsumoto らの研究は，メトリクスの算出機能を Web サービスの形式で提供することで，利用者の導入にコストがかからないというメリットがある．ただし，サービスとして提供されていないメトリクスの算出は不可能であり，サービスを拡張することはできない．一方で，ArgyleJ では，条件等を組み合わせることで柔軟に算出するメトリクスの期間等を設定することができる．また，メトリクスの算出以外も支援している点が Matsumoto らの研究との違いである．

7. まとめと今後の課題

本論文では，リポジトリマイニングにおけるデータ取得・加工の効率化を目指して，リポジトリマイニング向けのドメイン専用言語 ArgyleJ を設計・実装した．ケーススタディとして具体的な例題を設定し，ArgyleJ と Java 言語のそれぞれで例題を実装し，ArgyleJ の有用性を評価した．得られた知見は，下記の通りである．

- スクリプトの行数という観点からは，ArgyleJ は，Java よりも高い記述力を持つことが示された．
- 編集距離という観点から評価した場合，ArgyleJ の変更容易性は Java と比較して，高いことが示された．
- ArgyleJ においてキャッシュが存在する場合の実行速度に関しては，分析対象のデータサイズに関わらず，安定した実行速度で結果が得られることがわかった．

今後の課題として，フィーチャ分析から得られたフィーチャモデルの足りない点を再検討しながら，言語仕様・文法の再検討を行う必要があると考える．また，ArgyleJ の言語仕様では，中央型の版管理システム（CVS, SVN など）にのみ対応しているので，分散型の版管理を扱うための言語仕様（例えば，clone など）を検討する必要がある．

謝辞　本研究の一部は, 科学技術振興事業団「JST」の戦略的基礎研究推進事業「CREST」における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」の研究課題「ポストペタスケール時代のスーパーコンピューティング向けソフトウェア開発環境」による助成を受けた．また、本研究の一部は，日本学術振興会 科学研究費補助金（若手 A：課題番号 24680003）による助成を受けた．

参考文献

[1] A. E. Hassan: “The road ahead for mining software repositories”, Proc. Frontiers of Software Maintenance (FoSM’08), pp. 48–57 (2008).

[2] E. Shihab, A. Mockus, Y. Kamei, B. Adams and A. E. Hassan: “High-impact defects: A study of breakage and surprise defects”, Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE’11), pp. 300–310 (2011).

[3] W. Shang, B. Adams and A. E. Hassan: “Using pig as a data preparation language for large-scale mining software repositories studies: An experience report”, Journal of Systems and Software (2011(Online)).

[4] A. Hindle and D. M. German: “SCQL: a formal model and a query language for source control repositories”, Proceedings of the 2005 international workshop on Mining software repositories, pp. 1–5 (2005).

[5] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams and A. E. Hassan: “Revisiting common bug prediction findings using effort aware models”, Proc. Int’l Conf. on Software Maintenance (ICSM’10), pp. 1–10 (2010).

[6] S. Matsumoto and M. Nakamura: “Service oriented framework for mining software repository”, Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement, pp. 13–19 (2011).

[7] K. C. Kang, J. Lee and P. Donohoe: “Feature-oriented product line engineering”, IEEE Software, **19**, pp. 58–65 (2002).

[8] 市川：“内部ドメイン専用言語支援のための型に連動した字句・構文ルールの変更機構” (2011). 東京工業大学卒業論文.

[9] 水野：“基調講演：リポジトリマイニングの研究動向”, ソフトウェア・メインテナンス・シンポジウム 2011 (2011).

[10] 小林, 林：“データマイニング技術を応用したソフトウェア構築・保守支援の研究動向”, コンピュータソフトウェア, **27**, 3, pp. 13–23 (2010).

[11] A. Mockus: “Tutorial: How to run empirical studies using project repositories”, Int’l Advanced School of Empirical Software Engineering (2006).

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson: “Feature-oriented domain analysis (foda) feasibility study”, Technical report, Carnegie-Mellon University Software Engineering Institute (1990).

[13] H. Kagdi, M. L. Collard and J. I. Maletic: “A survey and taxonomy of approaches for mining software repositories in the context of software evolution”, J. Softw. Maint. Evol., **19**, 2, pp. 77–131 (2007).

[14] R. Moser, W. Pedrycz and G. Succi: “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction”, Proc. Int’l Conf. on Softw. Eng. (ICSE’08), pp. 181–190 (2008).

[15] O. Mizuno and T. Kikuno: “Prediction of fault-prone software modules using a generic text discriminator”, IEICE Transactions on Information and Systems, **E91-D**, 4, pp. 888–896 (2008).

[16] V. I. Levenshtein: “Binary codes capable of correcting deletions, insertions, and reversals”, Soviet Physics Doklady, **10**, 8, pp. 707–710 (1966).

[17] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei and J. X. Yu: “Matching dependence-related queries in the system dependence graph”, Proc. Int’l Conf. on Automated software engineering (ASE), pp. 457–466 (2010).

© 2012 Information Processing Society of Japan