情報処理学会論文誌 Vol.53 No.2 578–589 (Feb. 2012)

Feature Locationを用いた ソフトウェア機能の対話的な実装理解支援

林 晋平[1,a) 関根 克幸[1] 佐伯 元司[1]

受付日 2011年5月27日, 採録日 2011年11月7日

概要：本論文では Feature Location（FL）を用いて対話的にソフトウェア機能の実装を理解する手法を提案する．既存の FL 手法は理解コストの削減に寄与するものの，機能に対応するコード片特定のための入力の構築は依然として難しい．提案手法では，FL の入力は利用者とシステムとの対話により段階的に改善されていく．利用者は，FL により発見したコード片を実際に読み，得た理解やコード片中に出現する識別子をもとに入力クエリを改善する．さらに，読んだコード片が理解に貢献したかの判断をシステムに与える適合フィードバックにより FL の評価関数を改善し，より適切な結果を得る．FL とコード片の読解，フィードバックを対話的に繰り返すことにより，利用者は効率的に機能の実装を理解する．提案手法の支援ツールを用いた事例においては，提案手法は非対話的手法に比べ理解の効率化に貢献することが分かった．

キーワード：Feature Location，プログラム理解，適合フィードバック

Interactive Support for Understanding Feature Implementation with Feature Location

Shinpei Hayashi[1,a) Katsuyuki Sekine[1] Motoshi Saeki[1]

Received: May 27, 2011, Accepted: November 7, 2011

Abstract: This paper proposes an interactive approach for efficiently understanding a feature implementation by applying feature location (FL). Although existing FL techniques can reduce the understanding cost, it is still an open issue to construct the appropriate inputs for the techniques. In our approach, the inputs of FL are incrementally improved by interactions between users and the FL system. By understanding a code fragment obtained using FL, users can find more appropriate queries from the identifiers in the fragment. Furthermore, the relevance feedback, obtained by partially judging whether or not a code fragment is required to understand, improves the evaluation score of FL. Users can then obtain more accurate results. We have implemented a supporting tool of our approach. Evaluation results using the tool show that our interactive approach is feasible and that it can reduce the understanding cost more effectively than the non-interactive approach.

Keywords: feature location, program understanding, relevance feedback

1. はじめに

ソフトウェア開発の保守プロセスでは，保守対象の機能を実装しているソースコードを理解する必要がある [1]．ソフトウェアは数多くの機能（feature）から構成されており [2]，保守工程で機能の変更が求められた際には，対象機能の実装を理解し，修正すべきコード片を特定する必要がある．しかし，ソフトウェアの大規模化と複雑化にともない，対象機能の理解に要する時間や労力は膨大となっている．このようなソフトウェア理解に関するコストは，ソフトウェア保守のうち最大を占める [3]．

理解の効率化のために，機能に対応するコード片を特定する feature location（以降，FL）手法 [2], [4], [5], [6], [7],

1 東京工業大学大学院情報理工学研究科計算工学専攻
Department of Computer Science, Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
a) hayashi@se.cs.titech.ac.jp

© 2012 Information Processing Society of Japan

[8], [9], [10] が有用である．FL では対象機能の特徴を入力とし，対応するコード片あるいはそれらの候補の順序付き一覧を出力する．変更要求に対応する機能を特定し，その特徴を入力として FL を行うことにより，理解に必要なコード片のみを抽出し，不要なコードを除外できる．特に，コード検索に基づく FL では，特徴として検索クエリ（以下，クエリ）が用いられる．

しかし，FL により理解に最適なコード片を自動的に，過不足なく抽出することは難しい．その理由として，FL 手法の入力の構築に必要な利用者の知識の欠如や理解に必要なコード片の個人差などがある（詳細は 2 章を参照）．FL による全自動での理解対象のコード抽出は困難であると我々は考えており，理解のためのより効率的な支援手法が求められる．

本論文では，FL を用いた対話的なコード理解手法を提案する[*1]．FL 結果の一部を読むことにより得た理解をもとにフィードバックを行う対話プロセスを追加し，FL を繰り返すことにより効率的なコード理解を行う．提案手法では，識別子の対応付けと動的依存解析とを組み合わせた FL によりコード片を抽出する．利用者は抽出したコード片の一部を理解し，それに応じてより適切なクエリを発行したり，コード片の要不要を判断して抽出結果の順位を再評価したりすることにより，結果を改善する．この手続きを対話的に繰り返すことで，抽出結果の過不足を修正し，適切なコード片の抽出を行う．

我々は，提案手法をツール iFL として実現し，その有用性を事例により評価した．評価では，提案手法により対話的に理解を行うことで，効率的に対象機能の実装箇所をすべて理解できることが分かった．

本論文の主な貢献は，(1) コード読解と FL を結合してコード理解を対話的に支援する手法を提案し，その実現可能性をツール iFL の実現により示したこと，(2) iFL の利用により理解が円滑となることを 2 つのプロジェクトへの適用事例により示したことの 2 点である．

本論文の構成を以下に示す．2 章では FL を用いた理解支援における問題点を取り上げ，我々の基本的なアイディアを述べる．3 章では提案する対話的手法，4 章ではその支援ツール iFL について述べる．5 章では iFL を評価し，その有用性を示す．6 章で関連研究を述べ，最後に 7 章で本論文をまとめる．

2. 問題点と基本的なアイディア

一般に FL では，抽出したい機能の特徴を入力として与える．特徴は，ソフトウェアの観察可能な振舞いや仕様書の記述などから利用者が想起する．特に，最も典型的なコード検索に基づく FL では，単語などのクエリを入力とする．たとえば，あるスケジュール管理ソフトウェアに対して，予定データの保存方法を変更することを考える．利用者はコード修正に先立ち，保存機能の現在の実装を理解する必要がある．利用者は “スケジュール” や “保存” などの単語をクエリとして FL を行い，得られたコード片を集中して理解することにより，理解に不要なコード片に関わる読解コストを下げる．

しかし，前述のとおり，FL により理解に最適なコード片を自動的に，過不足なく抽出することは難しいと考えている．我々はその難しさを，以下のように分析している．

利用者の知識の欠如 対象機能を実現するすべてのコード片を表現できるほど，対象機能の特徴を詳細化して入力することは難しい．たとえば，機能を実行するうえで重要な処理が外的振舞いとして現れない場合，適切な入力のためにはコード中の識別子に関する知識が必要となる．しかし，この知識が不十分な FL 利用者は多い．さらに，該当の識別子がコード中で別の意味で用いられている場合，偽陽性（false positives）が発生しうる．たとえば，スケジュール管理ソフトウェアに対しては，時刻に関する機能を実装するコードに用いる識別子には `date` や `time` など複数の候補があり，有益な FL 結果を得るためには適切なものを選ぶ必要がある．

利用者の求めるコードの個人差 コード理解には開発者の知識・スキルに基づく個人差が存在するため，理解に十分なコード片も利用者に依存する．たとえば，デザインパターンや理解対象プロジェクトのドメインに関する知識を持つ開発者は，メソッドや変数に使われる識別子名から，コード片の一部を読むのみで概要を把握できる場合があるものの，知識の少ない開発者は多くのコードを読む必要がある．このとき，知識を持つ開発者が特定の目的でコードを理解する場合においては，多くのコード片は不要なものとなる．

以上の理由により，現実的には開発者は FL 結果を精査し，依存関係などをもとに関連する足りないコード片を調査したり，他のクエリによる FL 結果との照合を行ったりするなど，より適切な結果を求めて試行錯誤する．たとえコード検索以外の FL 手法やそれらを組み合わせた高精度の FL 手法 [2], [9] を用いたとしても，FL 結果と理想とする結果との間には依然ギャップが残るため，試行錯誤の必要性は解消しない．

以上より，理解のためのより効率的な支援手法が求められる．FL を繰り返し行い，機能に対応するコードを理解する場合における，我々の観察は 2 点ある．

(1) **コードの知識のない間は曖昧なクエリが，重要な識別子を知ってからは具体的なクエリが適している．** 利用者は，はじめは実装に対する知識を持たないため，コード中の識別子に対応したクエリを想起しがたい．

[*1] 本論文の内容は，SES 2010 で発表した成果 [11] を発展させたものである．

© 2012 Information Processing Society of Japan

そのため，思いつくクエリを複数試し，その結果発見したいくつかのコード片から適切な識別子を発見し，新たなクエリとして用いる．

(2) **開発者は，FL により得られたコード片の一部の要不要を判断できる．**利用者は，FL により得られた複数のコード片の候補を選択し，実際にコードを読む．その結果，コード片の内容が目的に関係あるものか，ないものか，現時点では関係の有無を決定できない（理解できない）ものかを判断し，理解を深める．この判断を繰り返すことにより，理解すべきコード片の全体像を把握していき，特に重要なコード片に関連しているコードも重要であるとして，コードの読解を進める．

これらをふまえた我々のアイディアは，理解のためのコード読解と FL のプロセスとの緩やかな結合である．提案手法では，FL と FL により得たコード片の読解，読解により得た理解に基づくフィードバックのプロセスを対話的に繰り返すことにより，コード片の理解を進めていく．(1) に対応するため，前段の FL では類義語辞書を用いてクエリを拡張し，より多くのコードと対応付くようにする一方，後段では識別子から得た具体的な単語をクエリとし，誤検出を防ぐ．すなわち，初期のクエリは，`time` や `date` の複数の可能性を同時に検討できるよう，これらの集合に展開されるクエリを発行可能なようにする．また (2) に対しては，適合フィードバック [12]（詳細は 3.5 節）を行うことにより，重要なコード片に関連するコード片の FL での扱いを向上させる．

3. 提案手法

3.1 概要

本論文では，FL を用いたソースコードの対話的な理解支援手法を提案する．提案手法の概要を図 **1** に示す．提案手法では，情報検索に基づく，クエリを用いた FL を行い，FL 結果として順序付きのコード片群を得る．利用者は，高い評価値を持つコード片を選択し，対応するコードを実際に読む．その結果得た理解をもとに，クエリの更新およびヒントの追加を行い，再度 FL を行う．以降，FL と利用者のフィードバックを，必要なコードの理解を終えるまで対話的に繰り返す．図 1 において，FL を行う部分は自動化できるため，利用者が手動で行う部分は，クエリの改善，ヒントの追加，および FL 結果の吟味である．

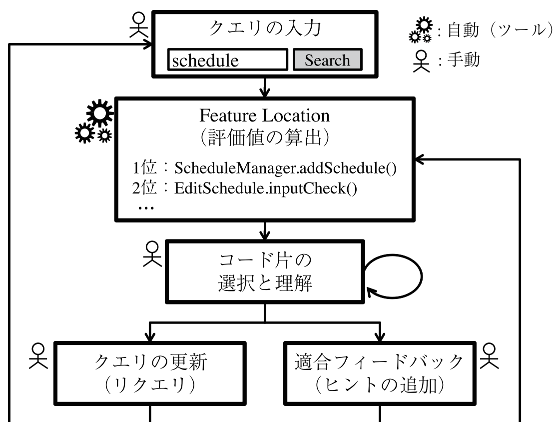

図 **1** 提案手法の概要
Fig. 1 Overview of the proposed approach.

提案手法はオブジェクト指向プログラムを対象とし，理解の単位をメソッドとする[*2]．特定の機能に関する処理はメソッドにまとめられることが多いため，理解単位として適していると考える．ソフトウェア理解を目的とするものを含む多くの FL 手法がメソッドを特定対象としていることもこのことを示唆している．たとえば，FL のサーベイ論文 [13] で紹介されている 58 手法のうち，50 手法はメソッドを特定対象としている．

さらに FL 精度の向上のため動的解析を用い，メソッドの呼び出しイベント（以降，イベント）を解析や抽出，評価の対象とする．すなわち，FL の結果はイベントの集合となり，利用者はイベントに対応するメソッドの実装を理解する．評価の対象をイベントとした理由の 1 つに，複数の機能で共有される汎用メソッドの依存関係を適切に扱えるようにすることがある．提案手法では，後述する適合フィードバックにより，重要だと考えられた理解単位と関連の深い（依存関係にある）理解単位が理解に有用であるとして利用者に推薦される．汎用メソッドは複数の機能で共有されるため，メソッドそのものを理解単位とすると，特定の機能の理解時に，理解対象機能ではない他の機能に関連するメソッドが多数推薦されてしまう可能性がある．着目する機能を実行する文脈で依存関係を扱うため，メソッドではなくイベントを扱っている．動的解析の利用にともない，対象機能を実行するテストケースを準備することは提案手法の前提となる．

提案手法で用いる FL では，情報検索と動的依存解析を組み合わせ，入力クエリとの類似性から実行系列中の各イベントを評価する．FL の概要を図 **2** に示す．FL の入力は理解対象のコード，動的解析のためのテストケース，利

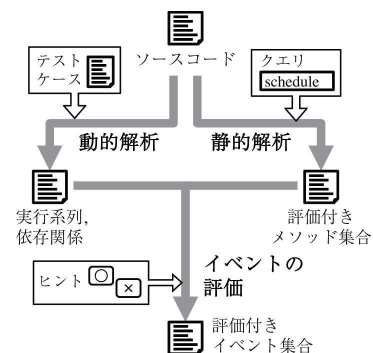

図 **2** FL 手法の概要
Fig. 2 Overview of our FL mechanism.

*2 本論文では，メソッドをコンストラクタを含む意味で用いる．

© 2012 Information Processing Society of Japan

用者の入力したクエリ，適合フィードバックにより得られたヒントであり，出力は評価により順序付けられたイベント集合である．まずテストケースを実行し，実行系列および動的依存関係を得る．また静的解析により，クエリと対応付いた度合いでコード中の各メソッドを評価する．最後に，実行系列中の各イベントを，依存関係とメソッドの評価値，ヒントにより評価し，順序付けて出力する．動的解析部はクエリ，ヒントに依存しないため，対話プロセスに先立ち1度のみ行う．

以降，FL 手法における動的解析，静的解析，イベントの評価の詳細を述べ，それらに基づく評価値の対話的な更新法を説明する．

3.2 動的解析：実行系列の生成

まず，与えられたテストケースとコードから実行系列と動的依存関係を生成する．コード中の全メソッドの集合を M で表す．テストケースを実行し，発生したイベントの集合 E を実行系列として得る．イベント間の動的依存関係（コスト付き）$\rightarrow \subseteq E \times E \times \mathbb{N}$ も同時に抽出し，有向の依存グラフ $G = \langle E, \rightarrow \rangle$ を得る．ここで，コストはイベント間の距離を表しており，距離の短いものほどイベント間の関連が強いものと考える．各 $e \in E$ はその呼び出し先メソッド $\mathit{callee}(e) \in M$ と関連付いている．すなわち，メソッド m を呼び出すすべてのイベント e において，$\mathit{callee}(e) = m$ である．動的解析を行うことにより，オーバライドなどによるメソッドの隠蔽，抽象メソッドの呼び出し時の実体の特定などが正確に行える．実行系列や動的依存関係の生成には，たとえば Reticella [14] などの既存の動的解析ツールを利用し，イベント間の距離には依存グラフにおける経由節の数（詳細は 4 章）などを用いる．

3.3 静的解析：識別子に基づくメソッドの評価

続いて，利用者の入力クエリに従い，コード中の各メソッドを評価する．クエリ Q は単語の集合とし，利用者が用意する．コード中の識別子には，その振舞いの目的を表すキーワードが用いられていることが多い [6] ことから，クエリと識別子が対応付く度合いによって，各メソッドの利用者の目的への相応しさを数値化する．対応付けは，Java で頻繁に用いられるラクダ型表記（camelCase）や単語の省略に対応するため，識別子のラクダ型表記は分解され，Q 中の単語とは先頭 3 文字以上の部分一致で対応付けが行われる．たとえば，単語 *schedule* や *vector* が識別子 `scheVec` と対応付く．

対応付け結果を用いて，各メソッド $m \in M$ を式

$$s(m) = \sum_{t \in T_Q(m)} w(t)$$

に基づき評価する．ここで，$T_Q(m)$ は m に対応する識別子*3のうち，クエリ Q に対応付くものの集合である．また，$w(t)$ は識別子の型に基づく重みであり，識別子の重要度に基づき事前に定めておく．たとえば，クラス名よりもメソッド名の方が対象の具体的な目的が表現される可能性が高いとし，相対的に高い値を与える．

類義語辞書によるクエリの拡張． 利用者が実装に関する知識を持たない場合，理解対象の概念に対応する識別子を想起しにくい．たとえば，表示機能に利用される識別子は `show` や `display` など複数存在する．この対応のため，クエリ中の単語を，その類義語群に展開してクエリに含め，語の揺れに対応させる．一方，類義語の利用は対応付けの精度を低下させるため，展開の利用の可否を単語ごとに利用者が選択可能とする．たとえば，表示には `display` ではなく `show` が使用されていることが分かれば，展開を行わず単に `show` を用いる．

3.4 依存関係を利用したイベントの評価

動的依存関係とメソッドの評価値を用いて，実行系列中のイベント $e \in E$ を評価する．評価値は，基本的には各イベントの素点 $s'(e)$ とイベント間の距離 d によって定義される：

$$f(e) = \max_{e' \in E} s'(e') \left(\alpha^{d(e,e')} + \beta \alpha^{d(e',e)} \right) + \operatorname*{avg}_{e' \in E} s'(e') \left(\alpha^{d(e,e')} + \beta \alpha^{d(e',e)} \right).$$

この式は，石尾らの関心事抽出手法 [2] におけるメソッドの評価式を拡張*4したものである．式において，$s'(e)$ はイベント e の素点であり，基本的には $s'(e) = s(\mathit{callee}(e))$ である（正確な定義は 3.5 節を参照）．また，$d(e, e')$ は依存グラフ G におけるイベント e，e' 間の最短コスト長であり，依存がない場合は ∞ である．$0 < \alpha < 1$ は減衰率であり，依存関係の距離に応じて減衰するため，より近い関係が強く考慮される．また，β は依存方向への重みであり，高素点のイベントが（被依存ではなく）依存しているイベントを強く考慮するため $\beta > 1$ と与える．第 2 項における avg は平均値を表しており，実行系列中の全イベントに対する平均を求める．以降，便宜上 $f(e)$ の第 1 項を最大部，第 2 項を平均部と呼ぶ．この式では，評価の高いメソッド自身の呼び出しや，評価の高いメソッドの呼び出しイベントと近い関係を多く持つイベントを高く評価する．

3.5 評価値の対話的更新

提案手法では，利用者から得たヒントを用いて FL 結果を改善する適合フィードバックを行っている．適合フィードバックの概要を図 **3** に示す．利用者は，FL 結果中の

*3 メソッド中の識別子以外にも，所属しているクラス内の識別子（クラス名，フィールド名など）も含まれる．

*4 元々の式は，以下に示すメソッドの評価式である：
$f(m) = \max_{m'} \alpha^{d(m,m')} + \operatorname{avg}_{m'} \alpha^{d(m,m')}$.

© 2012 Information Processing Society of Japan

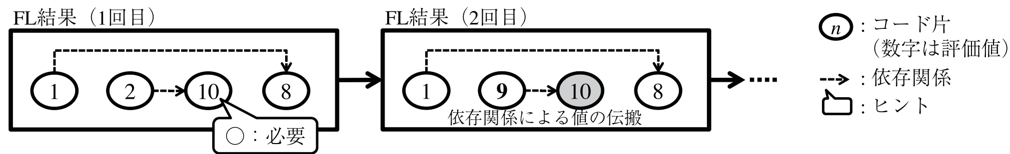

図 3 適合フィードバックの概要
Fig. 3 Overview of relevance feedback mechanism.

コード片を読解して，対象コード片の要不要を判断する．図 3 では，クエリに強く対応付いた（評価値 10）コード片が理解に必要だったことをヒントとして表明している．これにより，後段の FL では該当コード片に依存している他のコード片の評価値が 2 から 9 に向上している．このように，要不要の判断により，利用者により適するよう FL 結果を改善する．

利用者はイベントに付与された評価値に基づき，高評価のイベントを 1 つ以上選択する．利用者は選択したイベントに対応するメソッドを読み，可能ならシステムにフィードバックを行う．ここで，フィードバックとはヒント E_t, E_f およびクエリ Q の更新である．E_t は理解に必要，E_f は理解に不要だったイベントの集合であり，いずれも初期値は $\emptyset$ である．利用者にとって，e が理解に必要だったとき $E_t \leftarrow E_t \cup \{e\}$，$e$ が理解に不要だったとき $E_f \leftarrow E_f \cup \{e\}$ と更新する．また，重要な単語を発見したとき Q を修正する．

E_t，E_f，Q に変更があれば，それらに基づき FL を再実行する．3.4 節で導入した，イベント e の素点 $s'(e)$ の定義は，正確には以下のとおりである：

$$s'(e) = \begin{cases} \gamma \max_{m \in M} s(m) & \text{if } e \in E_t, \\ 0 & \text{if } e \in E_f, \\ s(\mathit{callee}(e)) & \text{otherwise.} \end{cases}$$

通常は e の呼び出し先メソッドの評価値がそのまま素点となる（otherwise 部）が，e の分類により，計算が異なる．e が理解に必要な場合（$e \in E_t$），素点が改善される．このときの値は，未評価の全イベントよりも重要とするため，既存メソッドの最大評価値に，さらに重み $\gamma > 1$ を乗じる．一方，e が理解に不要だった場合（$e \in E_f$），0 とする．ある e の素点 $s'(e)$ がヒントにより変化すれば，依存関係よりその周辺のイベントの評価値に影響が及ぶ．この作用により，クエリに含まれる識別子を含まないものの意味的に重要なメソッドの呼び出しイベントの評価を引き上げたり，不要な対応付けにより高評価となったイベントの順位を下げたりすることができる．

必要なイベントをすべて理解したと利用者が判断すれば対話プロセスを終え，理解を完了する．提案手法では，利用者による終了判断が可能なことを前提としている．このように，識別子の対応付けと動的依存解析を組み合わせて算出した評価値を，イベントの分類により対話的に更新することで，効率的に対象機能の実装を理解する．なお，プロセス終了時，必要と判断されたイベントの集合 E_t を E_t^* とし，正解イベント集合とする．

4. 支援ツールの実現

4.1 要件

提案手法の機械処理部分を自動化し，効率的なソフトウェア理解を支援するツール iFL を実現した．iFL の主な機能要件は，3 章で示した手法のうちクエリやフィードバックなどの対話プロセス以外を自動化することである．iFL はそれ以外にも，既存の開発環境上での稼働およびそれとの協調，利用者の入力の手間の軽減を実現するよう設計された．

4.2 構成

iFL のアーキテクチャを図 4 に示す．iFL は Java プログラムを対象としており，Eclipse プラグインとして構成されている．静的解析プロセスでは，Eclipse JDT が提供する構文解析器を用いてコードから識別子を抽出し，メソッドの評価を行う．また，動的解析には Reticella [14] を用いた．

Reticella はメソッド進入イベントやコンストラクタ進入イベント以外にも条件分岐や繰返し処理への進入など多数のイベントを処理しており，これらに基づく依存グラフを構築している．詳しくは文献 [14] で述べられているが，この依存グラフには，def-use を含むデータ依存関係，test-control を含む制御依存関係に加え，メソッド呼び出し依存関係や開始終了依存関係が関係として含まれている．我々が強く興味を持つものはメソッド・コンストラクタ進入イベントであるため，iFL では Reticella が扱う依存グラフ（G_0）中の節のうち，メソッド進入イベント（MethodEntry）・コンストラクタ進入イベント（ConstructorEntry）のみをイベントとして抽出し，これらを節とする小さな依存グラフとして G を構築している．イベント間の枝 $\rightarrow$ のコストには，G_0 における最短の経路長を用いる．経路が存在しない場合，枝は張られない．すなわち，3.4 節でのイベント間の距離 d の計算では，もとのグラフ G_0 での距離が反映されていることとなり，メソッド進入イベント間に他の種類のイベントを経由する依存があれば距離は増加する．

クエリ展開のための類義語辞書には WordNet [15] を利用した．iFL は，日本語版の WordNet [16] を用いて，日本語のクエリを英単語の集合に変換するクエリの翻訳機能も備

© 2012 Information Processing Society of Japan

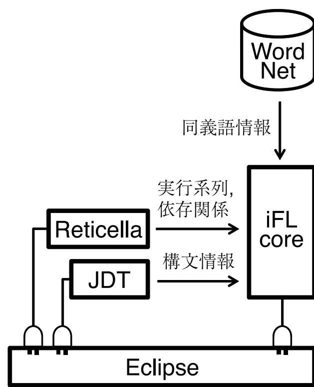

図 4　iFL のアーキテクチャ
Fig. 4　Architecture of iFL.

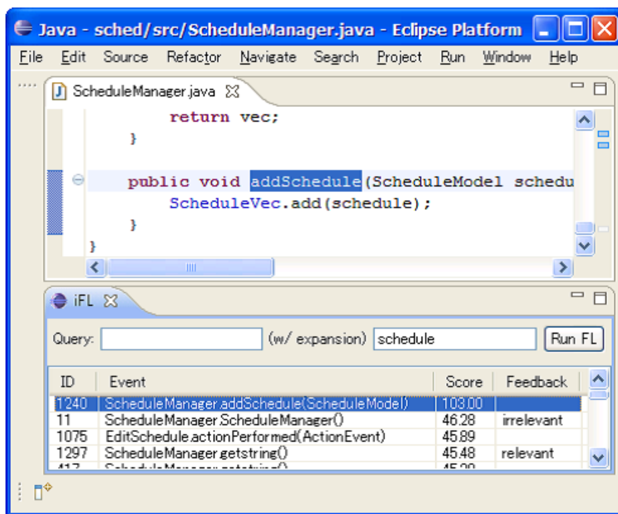

図 5　iFL の動作例
Fig. 5　Screenshot of iFL.

えており，次章での評価では，クエリの起点（初期クエリ）には日本語を用いた．

4.3　動作例

iFL の動作例を図 **5** に示す．この図は，動的解析を終え，FL を実行した状態を表している．クエリ入力のためのフィールドは 2 つあり，一方ではクエリが類義語の集合に展開される．図 5 では，これに単語“スケジュール”を入力しており，これは WordNet の辞書により“docket”，“agenda”，“schedule”，“timetable”に変換される．FL の結果は iFL ビューに出力される．ビュー内のリストの各列は，ID，対応するメソッド，評価値，ヒントを表す．ID はイベントごとに一意な整数であり，その大小は実行順序を表す．図 5 の第 1 行は，ID が 1240 のメソッド `ScheduleManager.addSchedule(ScheduleModel)` を表し，その評価値は 103.00，要不要の判断は行われていないことを表す．なお，変更のための理解に不要なライブラリの呼び出しイベントはリストから除外されているが，それ以外のイベントはすべてリストに含まれるため，同一のメソッドの呼び出しが複数回あった場合，メソッド名の表示が等しい行が複数出現する．利用者は，主に評価値，それ以外にも実行順序（ID），判断結果でリストをソートしながら次に読むべきメソッドを柔軟に決定する．利用者がイベントを選択すると，対応するメソッドのコードがエディタに表示される．動的解析の結果により，隠蔽されたメソッドの特定は正確に行われる．コード理解ののち，iFL ビュー上で要不要の判断を入力し，再評価のボタンを押すことにより，再度 FL を行い，新たな順序を得る．あるメソッドの呼び出しイベントはすべて対象機能の実行に含まれると見なした際の操作を簡単化するため，iFL ビューは同一のメソッドを呼び出すイベント群を一括して正解（不正解）イベント集合に追加するコマンドも備えている．Eclipse のコードナビゲーションによるコード読解と，その結果を考慮した FL に基づくイベントの順序付けビューにより，利用者は効率的にコードを理解する．

5.　評価

5.1　設定

適用事例の分析により手法の有用性を示す．本評価では，保守プロセスにおいて対象ソフトウェアに変更要求が生じた場合を想定し，提案手法を用いて対応する機能を特定する．

評価基準に用いる指標として，まず対話・非対話プロセスでの労力を定義する．提案手法を用いずに，すなわち対話的な順位の更新を行わない場合に理解する必要のあるイベント数を，対話プロセスにおける最後のクエリでの FL 結果のうち最も低位に現れた正解イベントの順位

$$C_N = \max_{e \in E_t^*} rank_N(e)$$

で見積もる．ここで $rank_i(e)$ は i 回目の FL におけるイベント e のイベント全体に対する順位（評価値の大小と逆順，同評価値はタイ，ヒントは未使用），N は対話プロセスにおける FL 回数である．すなわち，対話プロセスのうち最後の（最良の）FL 結果をもってしても，利用者は上位から順に C_N 個のイベントを選択する必要があったことに基づいている．これに対して対話的手法での労力は，プロセスを通して選択したイベントの総数

$$C_I = \sum_{i=1}^{N} \left| \{ e \mid e \in E \wedge rank'_i(e) \le r_i \} \right|$$

として見積もる．ここで，$rank'$ は要不要や理解保留の判断を行っていないイベントのみからなる順位を，r_i は i 回目の FL で検討した順位数を表す．すなわち，i 回目の FL 結果においては第 1 位から r_i 位までのイベントを順に選択し，その後次の FL を行う，といったステップを繰り返し，全体として C_I 個のイベントを選択して終了する場合を想定している．ただし，選択済みイベントと同名のメソッド

© 2012 Information Processing Society of Japan

表 1　適用対象
Table 1　Targets of case study.

		#class	$\|M\|$	$\|E\|$	$\|E'\|$
Sched	S1–5	7	24	179	38
JDraw	J1	173	1,726	5,898	1,761
	J2			5,880	1,725

を呼び出し，理解に貢献しないとしてスキップされたイベントは含まない．

これらを用いて定義される評価項目は以下の 3 つである．

オーバヘッド．選択するイベントがすべて対象機能と関連するとき，すなわち $C_I = |E_t^*|$ のとき，利用者の労力は最小になる．そうでない場合は，メソッドの実装を読んだにもかかわらず対象機能に含まないと判断したイベント数

$$O = C_I - |E_t^*|$$

分のオーバヘッドが生じる．このオーバヘッドの少なさを評価する．

利用者の不要な労力の削減．対話プロセスの導入による労力の削減率 ΔC は，対話・非対話プロセスでの不必要な労力の比を用いて

$$\Delta C = 1 - \frac{C_I - |E_t^*|}{C_N - |E_t^*|}$$

と表せる．分数部の分母・分子は非対話的・対話プロセスでの労力のオーバヘッドを表している．ΔC が大きかった場合，分子，すなわち対話プロセスでのオーバヘッドが分母，すなわち非対話プロセスでのものに比べ相対的に小さかったことを示しており，多くの労力が削減できたと見なす．途中計算においては，機能自体の大きさの影響を抑えるため，正解数 $|E_t^*|$ を適宜間引いている．

クエリ更新回数．クエリ更新回数 N' が 0 でなければ，該当機能の理解に更新が貢献していると考える．

5.2　適用対象と結果

適用対象は，Sched と JDraw 1.1.5 [*5]である．Sched は我々の研究室で開発した小規模のスケジュール管理ソフトウェアである．Swing で実装されており，予定の追加や削除，一覧の表示が行える．JDraw は画像の編集を行う中規模のオープンソースソフトウェアである．前者は提案手法の適用結果を詳細に分析できるよう，後者は現実的な大きさのソフトウェアへの適用が可能であることを確認できるよう選択された．

Sched に 5 つ，JDraw に 2 つの変更要求が生じた場合を考え，対応する機能を理解した．各ソフトウェアおよび動的解析による実行系列の規模を**表 1** に示す．表の各列はそれぞれ，各ソフトウェアのクラス数，メソッド数，実行系列中のイベント数，ライブラリの呼び出しを除いたイベント数を表す．ライブラリの呼び出しイベントは機能理解に不要として，非ライブラリの呼び出しイベントの評価値計算のための依存グラフ（G_0）には含めたが，評価値計算の対象（G）には含めなかった．Sched では，5 つの変更要求に対応する機能をすべて実行する，単一のテストケースを用意した．なお，JDraw の事例においては，Reticella の制限を回避するため，振舞いが変化しないよう注意しながら，元コードの一部書き換えや実行系列の結合を行っている．

理解は，筆者らのうちの 1 人（以下，被験者）により行われた．被験者は，コンピュータサイエンスの教育を受けている修士課程 2 年（当時）の学生であり，3 年の Java 利用経験があった．被験者は過去に Sched の開発経験を持っていたが，時間経過によりその詳細の理解は失われており，再度の理解を試みた．また，被験者は JDraw に関する知識を持っていなかった．

結果の再現性の向上のため，対話プロセスにおいて最大評価値を持つイベントが 2 つ以上存在した場合，FL を再度行う前にそれらすべてのイベントを順次選択した．また，次の対話プロセスでの作業は，(1) ヒントの更新，(2) クエリの更新，(3) 次順位のイベントの選択，の順に検討した．選択した順位数 r_i については，大きくならないよう努めながら，各々のランキング結果を確認しながら被験者がそのつど主観により定めた．

また，各対話プロセスは，被験者が理解に十分であると判断したタイミングで終了された．ただし，終了後には用意された変更要求を満たすようソースコードを実際に修正し，理解度を確かめた．すなわち，求める振舞いとなるよう修正できず，変更要求を満たせなかった場合には，理解が不十分であると見なし，対話プロセスに復帰して理解を継続した．正解イベント集合 E_t^* は，変更要求が正しく満たされたことが確認できた時点で確定させた．ここで，修正作業中には，正解と判断したイベントに対応するメソッド以外の参照を禁止し，対話プロセスでのコード読解以外でソースコードの理解が起こらないようにした．

適用結果を**表 2** に示す．表 2 の各列は，機能理解の理由となった変更要求，理解対象の機能，最終的に被験者が正解と判断したイベント集合の要素数，正解と判断したイベントに対応するメソッドの所属クラス数，FL 回数，対話・非対話的手法での労力，労力の削減率，オーバヘッド，クエリ更新回数を示している．FL における重みには，経験的に調整した値として $\alpha = 0.8$，$\beta = 2$，$\gamma = 1.5$ を用いた．また，識別子の重み $w(t)$ は，t がクラス名のとき 10，フィールド名のとき 3，メソッド名のとき 20，メソッド引数名および局所変数名のとき 1 とした．理解は Phenom 8400，RAM 2 GB を搭載する Windows PC 上に構築された Eclipse 3.4 を用いて行われた．初回 FL の計算時間には Sched で 20 秒，JDraw で 1 時間程度を要するも，値の対話的更新はそれぞれ 1 秒，15 秒以内（計算時間であり，対話

[*5] http://jdraw.sourceforge.net/index.php?page=6

© 2012 Information Processing Society of Japan

表 2 適用結果
Table 2 Application results.

変更要求	機能	$\lvert E_t^*\rvert$	$\lvert C\rvert$	N	C_I	C_N	ΔC	O	N'
S1 予定データの保存方法を XML に変更	予定データの読み込み	19	3	5	20	31	0.92	1	2
S2 時刻のデフォルト値を 18:00 に	文字列から時刻データへの変換	7	2	5	8	10	0.67	1	1
S3 ボタンの配置を中央下部から両端下部に	ボタンのレイアウト処理	1	1	2	2	2	0.00	1	0
S4 不正な入力時のエラー表示	入力から予定データへの変換	10	4	6	10	13	1.00	0	2
S5 予定一覧の項目に「詳細」の列を追加	予定一覧の表示	3	2	6	6	15	0.75	3	2
J1 タイトルバーの表示形式の変更	タイトルバーの文字列構築	10	3	4	20	156	0.93	10	2
J2 起動時のロゴの非表示	ロゴ表示処理	4	3	6	18	173	0.92	14	3

表 3 r_i の値
Table 3 Values of each r_i.

変更要求	r_1	r_2	r_3	r_4	r_5	r_6
S1	1	1	1	1	1	
S2	1	1	1	1	1	
S3	1	1				
S4	2	1	1	1	1	1
S5	1	1	1	1	1	1
J1	1	5	2	12		
J2	2	3	1	3	6	3

の時間を含まない）で実行された．計算時間を除いた作業にかかった時間（コード読解時間を除く）は 1 つのイベントにつきたかだか 20 秒程度だった．

S3 を除いて所属クラス数が 2 以上であることから分かるように，多くの機能は複数のクラスに分散したメソッドの集まりを読解することにより理解されている．また，1 クラスあたりの平均メソッド数は Sched では 3，JDraw では約 10 であり，多くの機能の理解にはクラス内の全メソッドを理解する必要がないことも分かる．対話プロセスにおけるイベントの選択数 r_i（$1 \leq i \leq N$）を**表 3** に示しており，総じて Sched の機能よりも JDraw の機能のほうが値が大きくなっている．

表 4 には，S5 における各 FL によりどういったメソッドの呼び出しイベントが順序付けられるかが表現されている．被験者はまず初期クエリとして “スケジュール”，“表示”，“リスト” を用い，類義語辞書の展開をともなって FL を実行している．下線の引かれたイベントは最終的に正解と判断されたものである．被験者は最上位のイベント 1240 に対応するメソッドを読み，スケジュールに相当する識別子 “schedule” を発見し，クエリを更新する．2 回目の FL では，イベント 1240 を不適として E_f に，イベント 1297 を適切として E_t に追加している．以上の結果，適合フィードバックにより，正解の 1 つであるイベント 380 の順位が 32 位から 12 位，12 位から 6 位に向上している．このように，対話的 FL により，下位にあった重要なイベントを上位に引き上げることができる．

5.3 評価と議論

表 2 より，7 例中 6 例でクエリ変更を必要とし，また ΔC は 6 割以上となった．特に 4 例では，9 割以上の労力を削減できた．S3 で ΔC が低かった原因は，初回の FL 結果が良好だったためであり，対話的手法の欠陥ではない．また，各事例の O は，Sched では平均 1.2，JDraw では 12 だった．プロジェクトの規模，変更要求によって大小があるものの，全体として小さな値となった．特に S4 では，FL 回数は 6 回と最多だったものの $O = 0$ であり，1 度も対象機能と関係のないイベントが最大評価値を持たなかった．また，iFL の時間効率は，値の更新が 15 秒以内に行われていることから，対話的環境として十分だったと考える．以上のことは，少なくとも本適用対象に限っては，提案手法の有用性を示唆している．

適合フィードバックの効果を確かめるため，Sched の機能について，対話プロセスの途中の状態における残コストを調べた．対話プロセスを，正解イベント集合のうち半数強 $\lceil |E_t^*|/2 \rceil$ のイベントにヒントを付与し終えたところまで進めた場合を考える．このときのヒントの集合および対話プロセスにおける最後のクエリを用いて行った FL 結果において，まだ残っている正解イベントのうち最も低位に現れた正解イベントの順位 C_H を対話プロセスの中間における残コストとし，非対話プロセスにおけるコスト C_N と比較した．その結果，正解イベントを 1 つしか持たなかった S3 を除いた全機能で，C_H は C_N より小さく，また $|E_t^*|$ と一致した．これは，残ったイベントを上から順に選択していけば不要なイベントを選択することなく正解を得られることを意味しており，適合フィードバックの効果が良好であることを示唆している．

クエリの更新はたびたび行われていたが，実際に行われた更新には 3 つのパターンが見られた．それぞれ，具体例を添えて示す．

- **新規クエリの追加**．たとえば S4 では，追加処理に関わる識別子 `add` が 3 回目の FL から追加された．
- **ノイズになったクエリの除去**．たとえば J2 では “ロゴ” が用いられていたが，ロギング処理に用いる識別子 `log` とマッチしてしまうため除去された．
- **クエリの具体化**．たとえば J2 では，前段では “画像”

© 2012 Information Processing Society of Japan

表 4 S4 における対話プロセスの例（抜粋）
Table 4 Example of interaction in S4 (excerpt).

		回目	1	2	3	ヒント
		類義語展開を行うクエリ	Q_1	Q_2	Q_2	
位	ID	類義語展開を行わないクエリ	$\emptyset$	Q_3	Q_3	
1	1240:	ScheduleManager.addSchedule(ScheduleModel)		1240	<u>1297</u>	○
2	11:	ScheduleManager.ScheduleManager()		<u>1297</u>	1294	
3	1075:	EditSchedule.actionPerformed(ActionEvent)		11	11	
4	<u>1297</u>:	ScheduleManager.getstring()		<u>417</u>	<u>417</u>	
5	<u>417</u>:	ScheduleManager.getstring()		1291	1291	
6	1217:	ScheduleModel.ScheduleModel(Time,String,String)		729	<u>380</u>	
...						
12	1115:	Time.Time(String,String,String,String)		<u>380</u>	1217	
...						
32	<u>380</u>:	MainViewer.MainViewer(JFrame.Drawer)		301	319	
33	713:	MainViewer.actionPerformed(ActionEvent)		319	1240	×
...						

$Q_1 = \{$“スケジュール”, “表示”, “リスト”$\}, Q_2 = \{$“表示”, “リスト”$\}, Q_3 = \{$“schedule”$\}$

が用いられ，類義語に展開されていたが，後段では “image” に置き換わった．

特に J2 では，クエリの展開および更新が有益に機能していることが見て取れた．被験者はまず，初期クエリとして “画像”，“ロゴ”，“表示” を用いて FL を行った．JDraw では，画像に関連する識別子として `picture` と `image` の双方が用いられており，またこれらは異なる意味を持っていた．初回の FL では `Picture.createDefaultPicture()` をはじめとする，picture に関連するメソッドの呼び出しイベントが最上位に，image に関連するものが中位にランクインしたが，前者は不適として E_f に追加された．被験者はその後の分析で image に関するメソッドの読解が目的に合致していると推察し，4 回目の FL でクエリを “image” に更新できた．

対話的手法が有効に機能しなかった場合として，各事例でのオーバヘッドの原因を考える．両ソフトウェアにおいて O は高くないものの，その原因を明らかにすることは重要である．特に Sched における 5 例を精査したところ，以下の 2 原因が抽出できた．

不適切なクエリ たとえば，S1 では初回クエリ “スケジュール” と対応付く識別子が多く，不要なイベントが上位となっている．しかし，これは以降のクエリ改善で回復できたため，$O = 1$ と大きな問題にならなかった．これは，クエリ改善の有用性をも示唆している．

対象機能を実装しているイベントとの関係 事例 S2 でのオーバヘッドは，理解対象の機能に関連するイベントが他の機能とも関連付いていたため，適合フィードバックにより不要なイベントの順位が向上したことによる．この場合，たとえばフィードバックの粒度をメソッド呼び出しイベントよりも小さくしたり，制御依存やデータ依存など，依存関係の種類を考慮して評価値を算出したりすることが対策として有効と考える．より小さな粒度としては，たとえばメソッド内の特定の制御の通過などがある．

また，適用結果の分析により，評価値の計算式 $f(e)$ の問題も明らかになった．関数 $f(e)$ は，最大部と平均部の和からなるが，多くのイベントでの評価値計算において最大部が支配的となった．たとえば，S5 の対話プロセスにおける 1 回目の FL での計算では，全 38 イベントにおける最大部の平均は約 30.4，分散は 259 であったが，平均部の平均は約 0.670，分散は 0.445 であり，最大部に比べ著しく均質に小さかった．これは，多くのイベントが対象イベントと依存を持たず，平均をとる部分式の値が 0 になるためであり，周辺に高素点のイベントを多く持つイベントの評価を上げるという狙いが失われていた．この問題を解決するための評価式の再構築は今後の課題の 1 つである．

本評価の妥当性への脅威として，以下をあげる．

- 本評価における比較に用いられた非対話的手法の労力は，対話的手法により得たデータから導出されたものであり，比較実験により得られたものではない．すなわち，対話的手法の導入による労力低下の評価は選択したイベントの個数の見積りで行われており，たとえば全体でかかった時間などの評価はできていない．対話的手法には，値の更新のための計算時間が必要であり，これを含めた作業時間に基づく理解の労力は本評価の結果と一致しない可能性がある．機能によって理解の難易度が異なること，同一機能に対して異なる理解手法を適用する場合の学習効果を排除しがたいこと，実験コストなどが比較実験を難しくしている．
- 本評価での理解対象はスケジュール管理および描画ソフトウェアであり，クエリの展開に用いられた同義語彙（たとえばスケジュールなど）は，WordNet に含ま

© 2012 Information Processing Society of Japan

れるものでカバーできていた．WordNet は一般的な同義語辞書であるため，理解対象ソフトウェアの属するドメインによっては，WordNet が十分な同義語彙を備えていない可能性がある．
- 本評価では，被験者がすべてのケースで対象機能を修正するに十分な理解を iFL ビューに表示されたメソッドを読むことにより得ている．これは，理解に必要なメソッドの呼び出しイベントが iFL ビュー上に網羅されており，用意したテストケースが必要なメソッドをすべて実行できていたことを示している．提案手法は動的解析に用いるテストケースの品質に依存しており，用意したテストケースが必要なメソッドを実行できなければ被験者の理解が十分に行えなかった可能性がある．
- 実験は筆者らの 1 人を被験者として行われた．我々は，対話的手法が有利な結果を得ないよう心掛けたが，それでもバイアスの存在の可能性は残っている．また，被験者が持っていた，過去での Sched における開発経験は，Sched の機能理解に影響を及ぼしている可能性がある．
- 理解結果のデータが 7 件，被験者が 1 名のみであるため，統計的検定を行っておらず，良好な結果を得られているものの，結論への脅威が残る．

ただし，これらをもってしても，限られたサンプルではあるが手法の有用性を示唆する重要なデータを得られたと考えている．より詳細な評価は今後の課題である．

手法の大規模実行履歴への適用には，動的解析結果のデータの肥大化に対する対処も必要と考える．対話的な FL 結果の更新には，JDraw で 15 秒程度の時間を要している．今回のテストケースはプログラム実行の一部のみを扱っており，たとえば実際にキャンバスに絵を描くなどにより得た履歴は容易に肥大するため，その対処が必要である．また同様に，依存グラフの肥大は，iFL ビューに並ぶイベント列の肥大を意味するため，適切な優先順位付けがされていたとしても，求める重要なイベントを選択する際に困難が生じる可能性がある．提案手法では，呼び出されたメソッドを正確に特定するため，またイベント間の距離を得るために動的解析を用いており，実際にメソッドがどういう順番で何回得られたかのデータを必要としない．実行系列から依存グラフをそのまま作成するのではなく，同一のメソッドを呼び出すイベントの節を単一化し，より小さいグラフを得ることにより，支援内容を悪化させずに依存グラフを縮退できる可能性がある．

6. 関連研究

6.1 Feature Location

FL は提案手法のベースになっている重要な技術である．FL は大きく 3 つのアプローチに分類され [6]，それらあるいはそれらの組合せが用いられる．

ソースコード検索．情報検索などの語彙検索に基づく手法は最も一般的な FL 手法である．コード中の識別子やコメントと入力クエリを対応付け，対応付いたコード片を抽出する．たとえば Shepherd らは，識別子中の機能についての関係性をモデル化し，コード検索を行っている [6]．また Liu らは，クエリと識別子の対応付けを用いたメソッドのランク付け手法を提案している [5]．コード中の識別子の情報だけでなく，オントロジなどの外部の知識資源の利用も有用である [17]．ただし，適切なクエリの構成には識別子の語彙に関する知識が求められる，クエリを拡張すると精度が低下する，などの問題点がある．

ソフトウェア構造ナビゲーション．ソフトウェアの構造として現れるコード片どうしの関係を解析し，強い関係を持つコード片をグループ化し，これと対象機能の特徴を比較して FL を行う試みもある．関係には，モジュールの含む識別子の類似性 [8] や依存関係 [18] などが用いられる．後者は，Dependency Structure Matrix（DSM）を用いて視覚化を行うことにより，高い理解容易性を得ている．また，独立成分分析を用い，同種の識別子によって構成されている関数群を機能の候補として全自動で発見する試みもある [19]．

動的解析．動的解析では，対象機能の起動を含んだテストケースを実行し，実行時のソフトウェアの内部動作の把握により機能の実装箇所を特定する．これは，実行されなかったコード片は理解に不要として除外できるため有用である．たとえば Edwards らは，対象機能の実行の有無の異なる 2 テストケースによる実行系列の差を分析して FL を行っている [7]．

複合的アプローチ．Poshyvanyk らはコード検索と動的解析の組み合わせ手法 [9]，石尾らはコード検索とプログラム構造ナビゲーションの組み合わせ手法 [2] をそれぞれ提案しており，それらは精度の向上に貢献することが分かっている．Zhao らは情報検索に基づく FL 手法を用い，コールグラフ上の探索により結果を改善している [20]．

しかし，提案されている FL 手法の多くは，コード理解プロセス全体にわたってどのように利用すべきかを定めておらず，そのままでは理解支援には不十分である．

6.2 適合フィードバック

提案手法で用いた，要不要の判断をシステムに伝えることによる評価の改善は，適合フィードバック [12] として情報検索分野でよく知られており，FL においても Gay らによる応用例がある [10]．Gay らの手法では，ベクトル空間モデルに基づく FL の結果を適合フィードバックにより更新しており，精度の向上があったことを報告している．我々の手法は依存関係に基づく評価を行っていること，FL とフィードバックの繰返しによりコードを理解する目的に

© 2012 Information Processing Society of Japan

特化しており，その支援ツールを構築していることが違いとしてあげられる．

6.3 対話的な理解支援

いくつかの手法は，対話的な理解支援を行っている．Chen らの手法 [21] は，継続的にコード片の選択・理解を繰り返すという点で我々の手法と類似している．FEAT [22] もプログラムの構造モデル上を対話的に探索し，機能に関連するコード片を発見する手法である．しかし，これらの手法は FL の途中でどのように利用者から現在の理解のフィードバックを得るかについては論じていない．

7. おわりに

本研究では FL を用いたコード理解支援手法を提案した．提案手法では，動的依存解析および識別子の対応付けに基づく FL に対して，適用結果のコード片の一部を読んで得た理解をもとに，クエリ改善や適合フィードバックを行い FL 結果を改善していく．FL と理解，フィードバックを対話的に繰り返すことで，利用者はコードを効率的に理解できる．支援ツール iFL を用いた評価では，7 例中 6 例で，非対話的手法に比べ効率的な理解が行えた．

今後はさらに以下の課題に取り組みたい．

さらなる評価 提案手法のスケーラビリティや iFL のユーザビリティの評価，類義語辞書の有用性の定量的評価も含め，より多くの事例研究を重ねる必要がある．その際には，同一のメソッドの呼び出しイベントを集約する動的解析データの間引きや効率の良い処理の検討を行う．

ドメイン知識の活用 提案手法ではクエリ展開に WordNet を用いており，理解対象のソフトウェアが属しているドメインを考慮していない．ドメインオントロジなどの知識資源を用いることにより，ドメインに特化した機能の理解を支援できる可能性がある．

イベント評価式の改善 評価式 $f(e)$ において，本論文での評価により発見された問題を解決し，素点の伝搬が効率良く行われるよう式を再構成する．

コードナビゲーションに基づくフィードバック 開発者は，開発環境の様々な機能を用いてコード上を巡回し，理解を行う．フィードバックにおいて，巡回の経路や履歴の利用，要不要決定の粒度の変更などにより，より効果的な評価改善が可能だと考えている．

他のソフトウェア理解手法との組合せ FL 以外のソフトウェア理解手法と組み合わせることで，より効率的な支援が行えると考える．たとえばコードや実行系列の可視化技術は，選択したイベントおよび対応するメソッドの理解の促進につながる．

謝辞 有用なツールをご提供いただいた名古屋大学の野田訓広氏および小林隆志准教授に感謝する．

参考文献

[1] Murphy, G.C., Kersten, M., Robillard, M.P. and Cubranic, D.: The Emergent Structure of Development Tasks, *Proc. 19th European Conference on Object-Oriented Programming*, pp.33–48 (2005).

[2] 石尾 隆，仁井谷竜介，井上克郎：プログラムスライシングを用いた機能的関心事の抽出，コンピュータソフトウェア，Vol.26, No.2, pp.127–146 (2009).

[3] Vestdam, T. and Nørmark, K.: Maintaining Program Understanding – Issues, Tools, and Future Directions, *Nordic Journal of Computing*, Vol.11, No.3, pp.303–320 (2004).

[4] Wilde, N. and Scully, M.C.: Software Reconnaissance: Mapping Program Features to Code, *Journal of Software Maintenance: Research and Practice*, Vol.7, No.1, pp.49–62 (1995).

[5] Liu, D., Marcus, A., Poshyvanyk, D. and Rajlich, V.: Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace, *Proc. 22nd International Conference on Automated Software Engineering*, pp.234–243 (2007).

[6] Shepherd, D., Fry, Z.P., Hill, E., Pollock, L. and Vijay-Shanker, K.: Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns, *Proc. 6th International Conference on Aspect-Oriented Software Development*, pp.212–224 (2007).

[7] Edwards, D., Simmons, S. and Wilde, N.: An approach to feature location in distributed systems, *Journal of Systems and Software*, Vol.79, No.1, pp.57–68 (2006).

[8] Shepherd, D., Pollock, L. and Tourwé, T.: Using Language Clues to Discover Crosscutting Concerns, *Proc. Workshop on Modeling and Analysis of Concerns in Software*, pp.1–6 (2005).

[9] Poshyvanyk, D., Marcus, A., Rajlich, V., Gueheneuc, Y.-G. and Antoniol, G.: Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification, *Proc. 14th International Conference on Program Comprehension*, pp.137–148 (2006).

[10] Gay, G., Haiduc, S., Marcus, A. and Menzies, T.: On the use of Relevance Feedback in IR-based Concept Location, *Proc. 25th International Conference on Software Maintenance*, pp.351–360 (2009).

[11] 関根克幸，林 晋平，佐伯元司：Feature Location を用いたソースコード理解の対話的支援，ソフトウェアエンジニアリング最前線 2010，pp.9–16, 近代科学社 (2010).

[12] Meadow, C.T.: *Text Information Retrieval Systems*, Academic Press (1992).

[13] Dit, B., Revelle, M., Gethers, M. and Poshyvanyk, D.: Feature Location in Source Code: A Taxonomy and Survey, *Journal of Software Maintenance and Evolution: Research and Practice*, DOI: 10.1002/smr.567 (2011).

[14] Noda, K., Kobayashi, T., Agusa, K. and Yamamoto, S.: Sequence Diagram Slicing, *Proc. 16th Asia-Pacific Software Engineering Conference*, pp.291–298 (2009).

[15] Miller, G.A.: WordNet: A Lexical Database for English, *Comm. ACM*, Vol.38, No.11, pp.39–41 (1995).

[16] Isahara, H., Bond, F., Uchimoto, K., Utiyama, M. and Kanzaki, K.: Development of Japanese WordNet, *Proc. 6th Language Resources and Evaluation Conference* (2008).

[17] Yoshikawa, T., Hayashi, S. and Saeki, M.: Recovering Traceability Links between a Simple Natural Language Sentence and Source Code Using Domain Ontologies, *Proc. 25th International Conference on Software Maintenance*, pp.551–554 (2009).

© 2012 Information Processing Society of Japan

[18] Sangal, N., Jordan, E., Sinha, V. and Jackson, D.: Using Dependency Models to Manage Complex Software Architecture, *Proc. 20th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.167–176 (2005).

[19] Grant, S., Cordy, J.R. and Skillicorn, D.: Automated Concept Location Using Independent Component Analysis, *Proc. 15th Working Conference on Reverse Engineering*, pp.138–142 (2008).

[20] Zhao, W., Zhang, L., Liu, Y., Sun, J. and Yang, F.: SNIAFL: Towards a Static Noninteractive Approach to Feature Location, *ACM Trans. Software Engineering and Methodology*, Vol.15, No.2, pp.195–226 (2006).

[21] Chen, K. and Rajlich, V.: Case Study of Feature Location Using Dependence Graph, *Proc. 8th International Workshop on Program Comprehension*, pp.241–247 (2000).

[22] Robillard, M.P. and Murphy, G.C.: Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies, *Proc. 24th International Conference on Software Engineering*, pp.406–416 (2002).

林 晋平 （正会員）

2004年北海道大学工学部情報工学科卒業．2006年東京工業大学大学院情報理工学研究科計算工学専攻修士課程修了．2008年同専攻博士後期課程修了．2009年より同専攻助教．博士（工学）．ソフトウェア変更やソフトウェア開発環境の研究に従事．IEEE-CS，ACM各会員．

関根 克幸

2008年東京工業大学工学部情報工学科卒業．2010年同大学大学院情報理工学研究科計算工学専攻修士課程修了．2010年より新日鉄ソリューションズ株式会社に勤務．修士（工学）．ソフトウェア開発環境やソフトウェア理解に興味を持つ．

佐伯 元司 （正会員）

1978年東京工業大学工学部電気電子工学科卒業．1980年同大学大学院工学研究科情報工学専攻修士課程修了．1983年同専攻博士後期課程修了．同大学助手，助教授を経て，2000年より同大学院情報理工学研究科計算工学専攻教授．工学博士．要求工学やソフトウェア開発技法等の研究に従事．IEEE-CS，ACM，ソフトウェア科学会，人工知能学会，電子情報通信学会各会員．