Vol.2014-SE-183 No.17
2014/3/20

下位概念間の依存関係を用いた機能捜索手法

加藤 哲平[1]　林 晋平[1]　佐伯 元司[1]

概要：ソースコードの修正時には，各手順に対応するモジュールを特定する機能捜索が必要となる．しかし，既存の機能捜索手法は機能を構成する概念間の関係を考慮しておらず，高精度で各概念に対応するモジュールを特定することが難しい．本研究では，機能を構成する下位概念間の依存関係を用いて機能捜索を行う手法を提案する．各下位概念に対応する記述を入力として既存の概念捜索手法を適用し，対応するモジュールの一覧を得る．その後，下位概念間に存在する依存関係を満たさないモジュールを一覧から除外することで，得られた一覧の精度を向上させる．既存手法との比較実験により，提案手法の有用性の評価を行った．

キーワード：機能捜索，概念捜索，ユースケース記述，プログラム依存解析

A Feature Location Technique Using the Dependency between Concepts in a Feature

Abstract: Since existing feature location techniques lack the relationship between concepts in a feature, it is hard to locate the concepts in source code of high accuracy. This paper proposes a technique to locate the concepts in a feature using the dependency between the concepts. We apply an existing concept location technique to the description for each concept and obtain the list of modules. Some of modules which do not match the dependency between concepts are filtered out, and we can obtain more precise list of modules. The experiment shows the effectiveness of our technique.

Keywords: feature location, concept location, use case description, dependency analysis

1. はじめに

プログラムを保守する上で，保守対象となる機能 [1] を理解することは重要である [2]．プログラムの品質を維持，改善していく保守作業では，新たな機能の追加や既存の機能の修正を行う．開発者はこれらの作業の為に，保守対象となる機能が実装されている箇所をソースコードから発見する機能捜索（feature location）[3] を行い，得られたモジュールを理解する必要がある．実装箇所の捜索は，通常ソースコードを読むことで行われる．ソフトウェア理解のためのコストの増大 [4] を防ぐため，機能捜索の様々な支援手法が提案されている [3]．

既存の機能捜索手法の多くは，機能を表す単一のクエリを入力し，クエリに対して関連度の高いモジュールをソースコードから捜索する [5]．しかし，ユースケース記述のような手順が構造的に書かれた記述を入力とした場合，分析者は機能全体に対応するモジュール群ではなく，各手順に対応するモジュールをそれぞれ知りたいだろう．また，ユースケース記述に限定せずとも，ある機能が複数の小さな機能の集まりとして実装されていることを分析者が知っていたり想像していたりする場合もある．いずれの場合でも，機能を，それを構成する下位概念[*1] [6][7] の集まりとみなし，各概念に対応するモジュール群を得ることにより機能全体を理解することが有意義であると考える．ここで下位概念とは機能を構成する概念を指し，例えば前述するユースケースの手順がこれにあたる．

しかし，効率よく下位概念ごとの捜索を既存の概念捜索手法で行うことは難しい．既存の概念捜索手法では，単一の記述を入力として扱うため，機能記述全体を入力とする

[1] 東京工業大学
Tokyo Institute of Technology

[*1] 機能を構成するものがそれ単体で機能をなすとは限らないため，本稿では機能の構成要素を概念としている．また，その捜索も概念捜索として区別している．

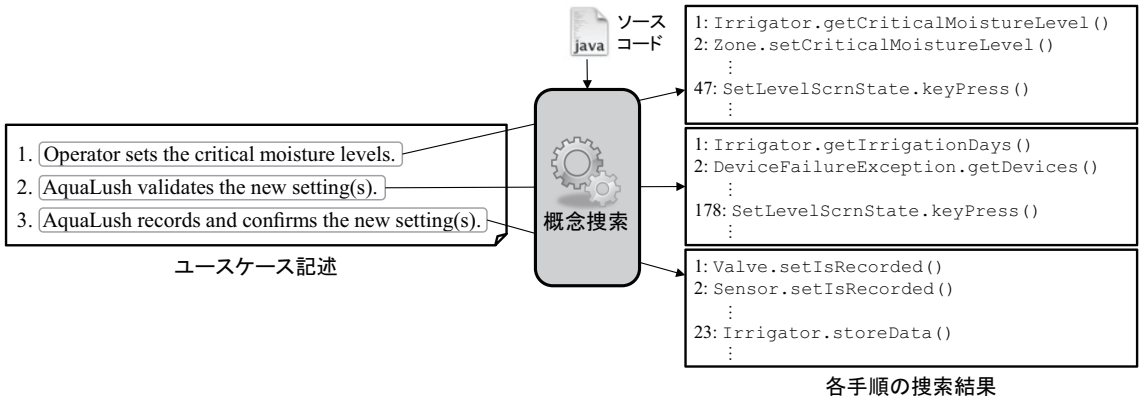


図 1　手順ごとに概念捜索を適用した例

と，出力されたモジュールがどの下位概念に関するものなのかが不明瞭である．また，各下位概念に対応する記述に対して概念捜索手法を適用することで，対応するモジュールを得られるものの，この方法では下位概念の間に考えられるデータの受け渡しや処理状態の遷移といった関係を考慮しないため，処理全体から考えると関連度の低いモジュールを出力してしまう．

本論文では，こういった下位概念の間の関係を考慮して，概念ごとの捜索結果を得る機能捜索手法を提案する．下位概念間には，ある概念が指す機能が実行されたのちに別の概念が指す機能が実行されるという時間順序の関係や，ある概念に関連するモジュールが生成したデータを別の概念に関するモジュールが使用するというデータの依存関係などがある．提案手法では，既存の概念捜索（concept location）[6] 手法が出力した各概念に関連するモジュールのランキングに対し，これらの依存関係を満たす可能性があるかを検査する．依存関係を満たす余地のないモジュールをランキングから除外することで，捜索作業の効率化を図る．

提案手法をツールとして実装し，既存の手法との比較による評価を行った．既存の機能捜索手法によって得たモジュールのランキングと，提案手法によって得たランキングを比較したところ，提案手法は既存の概念捜索手法が出力したランキングに対して，開発者の労力を削減しうる効率のよいランキングを出力できることがわかった．

本論文の主な貢献点を以下に挙げる．

(1) 下位概念間の依存関係に基づく機能捜索手法の提案
(2) 実験による評価の結果，既存の概念捜索手法より効率的にモジュールを特定できることを確認

本論文の構成を述べる．まず 2 章で提案手法の背景について述べ，3 章で提案手法について述べる．4 章で実装のために用いたツールについて述べたのち，5 章で提案手法の実用性の評価を行う．6 章で関連研究についてふれ，最後に 7 章でまとめと今後の課題について述べる．

2. 背景

機能捜索は，ソースコードの中から理解の対象である機能を実装している箇所を捜索する技術である [6]．プログラムの修正は機能単位で行われるため，機能捜索手法は単一の機能に関連するモジュールを出力することに重点を置いている．そのため，他の機能との依存関係や共通部分といった関係性を考慮した機能捜索手法は少ない．

既存の機能捜索手法の入力に，手順が書かれた仕様書を用いることがある．機能捜索手法で文書を入力する場合，その機能を十分に説明する文書を入力する．機能を説明する文書には，要求仕様書の機能要求や利用者が捜索したい機能を文章として記述したものがある．その中で捜索する機能の手順が書かれた仕様書は，その機能に対する情報量が多い為，機能捜索の入力として適している．手順の書かれた仕様書の例として，ユースケース記述を挙げる．

手順の書かれた仕様書を機能捜索の入力とした場合，各手順に対応した概念捜索結果が望まれる．機能捜索の入力となる仕様書は，対象の機能が行う処理を明確に記述している．そのため，開発者が機能の理解を行うとき，入力した仕様書の手順に沿った理解を行うことが考えられる．各手順に対応した結果を得ることができれば，開発者はその結果を用いることで，機能の手順を実装しているモジュールを容易に理解することができる．その結果，手順仕様書を用いない場合に比べ，機能の理解をより効率的に行うことができる．

手順ごとに概念捜索を行う場合，既存の手法では捜索結果の精度が悪い．既存の機能捜索手法では，入力した文書全体を 1 つの入力として扱う為，文書全体に関連度の高いモジュールのランキングを出力する．このとき，ランキングの上位に含まれるモジュールと，そのモジュールと関連性が高い処理との対応関係が不明瞭である．その為，開発者は対応関係を手動でさらに調査する必要がある．結果として，開発者の労力を増大させてしまう．また，各手順の結果を出力する方法として，手順の文をそれぞれ入力とし

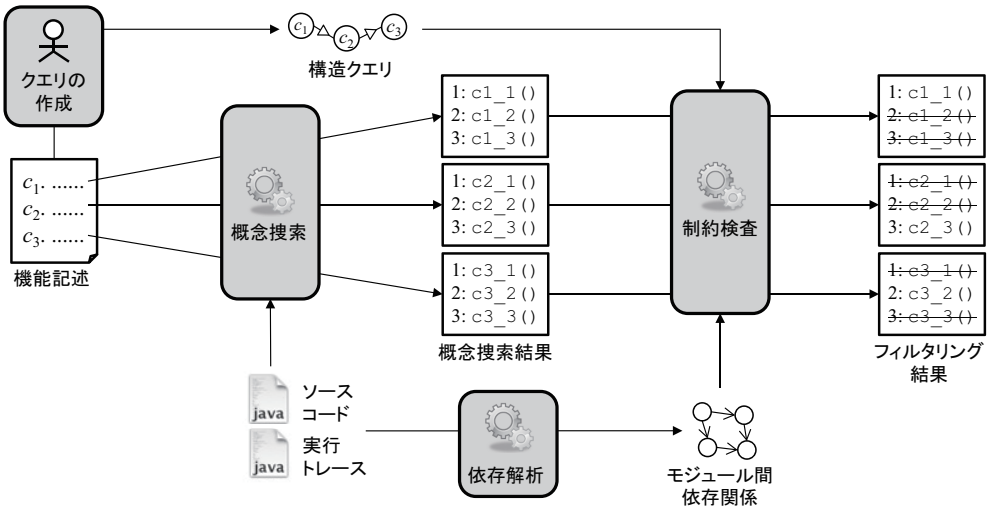


図 2　提案手法の概要

て概念搜索を行うことが考えられる．この方法では，入力となる手順の文のみを用いて搜索を行ってしまい，手順間で行われるデータやり取りや，代替系列の呼び出しと言った構造的な情報は失われてしまう．その為，各手順と関連度の低いモジュールがランキングの上位に出力されるようになり，開発者の理解の妨げとなる．

既存の手法を手順ごとに適用した例を図 **1** に示す．図中のユースケース記述を機能として見たとき，この機能は 3 つの手順で実装されていることがわかる．これらの手順を表す文書を，それぞれ既存の語彙的解析を用いた概念搜索手法の入力として適用し，各手順に関連度の高いモジュールのランキングを得る．手順 1, 2, 3 を実装しているモジュールのうち，ランキングのもっとも上位に現れるモジュールはそれぞれ，`SetLevelScrnState.keyPress()`，`SetLevelScrnState.keyPress()`，`Irrigator.storeData()` である．図中の各手順のランキングをみると，これらのモジュールはランキングの下位に現れており，開発者がこれらのモジュールにたどり着くまでに多くの関連性の低いモジュールを理解する必要がある．

この問題を解決するためには，手順間の依存関係を考慮した機能搜索手法が必要である．手順間にはデータのやり取りや，代替系列の呼び出しといった依存関係がある．この依存関係は，各手順を実装するモジュールによって実現されなければならない．すなわち，機能の手順を実装するモジュールは，手順が行うべき処理の他に，手順間に存在する依存関係を満たす必要がある．そのため，機能の各手順に対して機能搜索を行う場合には，手順との関連性の高いモジュールを出力できるだけでなく，そのモジュールが手順の間に存在する依存関係を満たすことのできるモジュールであるかを検査できる手法が望まれる．

3. 提案手法

3.1 概要

本論文では，下位概念間の依存関係を用いて，依存関係を満たす各下位概念に対応するモジュールを出力する機能搜索手法を提案する．提案手法の概要を図 **2** に示す．図中の灰色のノードは提案手法の主要なプロセスを指しており，棒人間の書かれたものは分析者が手動で行うものであり，それ以外は自動化されている．提案手法の入力は，特定する対象となる機能記述および対象ソフトウェアに関するデータ（ソースコードや実行トレース）である．まず，分析者が機能記述から下位概念を抽出し，構造クエリを作成する．クエリとは，対象機能がどういった下位概念から構成されており，またそれらにどういった関係があるかをグラフの形で構造的に記述したもので，これは後述する制約検査の際に用いる．次に，抽出した下位概念それぞれに対応する記述を概念搜索手法に適用し，対応するモジュールのランキングを搜索結果として得る．また，対象ソフトウェアのソースコードや実行トレースなどを解析し，モジュール間に張られる依存関係を抽出しておく．構造クエリを下位概念間の制約とみなし，モジュール間の依存関係情報を用いて制約を調べることにより，概念搜索の結果のランキングを，制約を満たす可能性のあるもののみにフィルタリングして出力する．

以降，これらのプロセスの詳細を説明していく．

3.2 クエリの作成

まず，分析者は機能記述から下位概念を抽出する．抽出は，処理の記述の主語や動作主，前後の処理との関連性に注意しながら行うとよい．本論文では，機能記述としてユースケース記述を用いる．ユースケース記述には，機能の行うべき処理が手順ごとにユースケースステップ（ステップ）

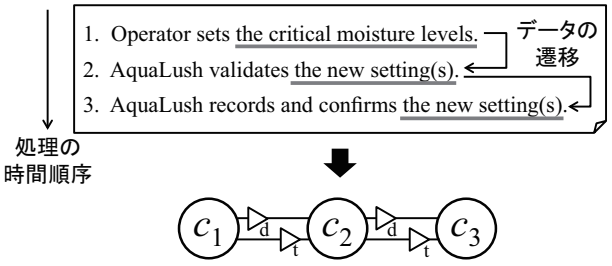


図 3　ユースケース記述からの構造クエリの抽出

として記述されており，このステップを 1 つの下位概念とみなす．こういった手順が書かれた機能記述が得られない場合には，分析者が考える機能の構成概念を想像して抽出すればよい．

次に分析者は本手法の入力となる下位概念間が満たすべき依存関係を制約として構造クエリに記述する．本手法で用いる構造クエリは，ラベル付き有向グラフ $Q = (C, \triangleright)$ の形で記述する．ここで C はグラフの節となる下位概念，グラフの辺 $\triangleright \subset C \times C \times L$ は下位概念間の制約を示す．ラベル L は制約 $\triangleright$ の種類を示す．下位概念 c_1 から，下位概念 c_2 へのラベル l の制約を，本論文では $c_1 \triangleright_l c_2$ のように記述する．

本論文では以下の 3 種類の制約を定義した．

時間制約 $\triangleright_{\mathrm{t}}$　c_1 が実行を開始した時間より後に c_2 が実行を開始すべきとしたとき，制約 $c_1 \triangleright_{\mathrm{t}} c_2$ を記述する．ユースケース記述では実行される処理が時系列順に並んでいるため，各ステップの前後関係から時間制約を抽出できる．

呼び出し制約 $\triangleright_{\mathrm{c}}$　c_1 を実行した後または実行途中で c_2 を呼び出すべきとしたとき，制約 $c_1 \triangleright_{\mathrm{c}} c_2$ を記述する．ユースケース記述では，主系列中のステップが代替系列を持つとき，それらに対応する下位概念の間に呼び出し制約を見いだせる．

データ制約 $\triangleright_{\mathrm{d}}$　c_1 を実行することで変更されたデータを c_2 の実行で読みとるべきとしたとき，制約 $c_1 \triangleright_{\mathrm{d}} c_2$ を記述する．ユースケース記述では，「ユーザーが x を入力する」という記述があるステップで変数への書き込みが，また x が記述されたステップで変数の読み込みが行われたと考え，それらの下位概念の間にデータ制約を見いだせる．

制約を表す構造クエリの例を図 **3** に示す．図上部は特定対象の機能を表したユースケース記述（図 1 で示したものと同じもの）である．このユースケース記述は 3 つのステップを持つため，それぞれに対応する 3 つの下位概念 c_1，c_2，c_3 を抽出する．これらの間の制約としては，まずステップが時系列順に並んでいることに注目し，時間制約 $c_1 \triangleright_{\mathrm{t}} c_2$ および $c_2 \triangleright_{\mathrm{t}} c_3$ を導出する．また，ステップ 1 でユーザーが入力したデータ “critical moisture levels” をステップ 2 で検査のために読み取るというデータのやり取りを見ることができるため，データ制約 $c_1 \triangleright_{\mathrm{d}} c_2$ を導出する．さらに，ステップ 2 で検査を行ったデータ “new settings” をステップ 3 で記録・適用するというデータのやり取りも伺えるため，同様に $c_2 \triangleright_{\mathrm{d}} c_3$ を導出する．このようにして，図 3 下部に示す構造クエリ $Q = (\{c_1, c_2, c_3\}, \{(c_1, c_2, \mathrm{t}), (c_2, c_3, \mathrm{t}), (c_1, c_2, \mathrm{d}), (c_2, c_3, \mathrm{d})\})$ を得る．分析者がこのクエリを作成したということは，対象機能が c_1（“Operator sets ...”）から c_3（“AquaLush records ...”）までの 3 概念によって構成されており，これらがこの順で実行され，またこの順にデータが流れていくということを期待していることを表す．

3.3　概念捜索

分解した各下位概念 $c \in C$ に対して概念捜索手法を適用し，関連度の強いモジュールのランキング $R_c \subseteq M$（順序付き集合）を取得する．ここで，M はソースコード中の全モジュールの集合を表す．以降，下位概念に関連度の強いモジュールのランキングを下位概念のランキングと呼ぶ．

3.4　依存解析

本手法では，下位概念間の制約のために，モジュール間の依存関係を抽出する．提案手法では時間依存関係，呼び出し依存関係，データ依存関係の 3 種類を扱う．これらは，それぞれ下位概念間の時間制約，呼び出し制約，データ制約に対応するものである．

時間依存関係． あるモジュール m_i が呼び出されたあとに別のモジュール m_j が呼び出されて処理を開始しうるとき，時間依存関係 $m_i \rightarrow_{\mathrm{t}} m_j$ を抽出する．本稿では，モジュールがこの順序で実行された例が存在することでもって，時間依存関係があるとみなす：

$$m_i \rightarrow_{\mathrm{t}} m_j \iff \exists t_i \leq t_j \bullet m_i @ t_i \wedge m_j @ t_j$$

ここで，$m@t$ は実行トレースにおいて時刻 t にモジュール m の実行が開始されたことを表す．

呼び出し依存関係． あるモジュール m_i が別のモジュール m_j を呼び出しているとき，モジュール間の呼び出し依存関係 $m_i \rightarrow_{\mathrm{c}} m_j$ を抽出する．呼び出し依存関係はプログラムがその機能を実行した際に呼び出したモジュール間のもののみを抽出し，条件分岐などによって呼び出されなかったモジュールへの関係は含めない．

データ依存関係． あるモジュール m_i が変数を操作し，その後別のモジュール m_j が変数を読み込んでいるとき，データ依存関係 $m_i \rightarrow_{\mathrm{d}} m_j$ を抽出する．本手法ではソースコードから得た静的な変数アクセスと実行トレースの動的情報を用いることで，擬似的に動的な変数アクセスを作成する．ソースコード上では変数名を参照したり，変数への代入文を記述したりすることで変数の読み書きを行うため，これらの記述をソースコードから発見することで，モ

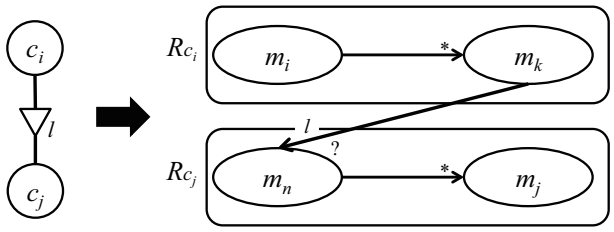


図 4　依存関係の変換方法

ジュール間の静的な変数アクセスを取得する．ただし，変数が指すオブジェクトに対する特殊なメソッド呼び出し（コレクション型やアクセサメソッドの操作など）も変数の読み書きとして扱う．本稿では，m_i が実行されてから m_j が実行されるまでの間に他のモジュールが変数に操作を行わない場合に限って依存関係を抽出する：

$$m_i \rightarrow_{\mathrm{d}} m_j \iff write_{m_i,v} \wedge read_{m_j,v} \wedge m_i \rightarrow_{\mathrm{t}} m_j \wedge \nexists m, t_i < t < t_j \bullet m_i@t_i \wedge m_j@t_j \wedge m@t \wedge write_{m,v}$$

ここで，$read_{m,v}$ および $write_{m,v}$ はモジュール m において変数 v への読み書きがあることを表す．

3.5　制約検査

ソースコードおよび実行トレースから得たモジュール間の依存関係を用いて，下位概念間の満たすべき制約を検査する．下位概念間の制約は，下位概念を実現するモジュール集合間に存在する制約と解釈できる．つまり，下位概念を実現しているモジュールが他の下位概念を実装しているモジュールとの依存関係をなすことにより，下位概念間の依存関係が成り立っていると考える．そこで，下位概念 c_i から c_j への制約を，下位概念のランキング R_{c_i}, R_{c_j} のモジュールの集合間に存在する制約とみることで，モジュール間の依存関係を解析し，依存関係を満たさないモジュールを除外することで，下位概念を実装しており，かつ他の下位概念とも制約を満たすモジュールを特定する．ただし，下位概念を実現しているモジュールは 1 つではなく，複数のモジュールの相互作用によっている場合もある．その場合，それらのモジュール間には，呼び出しや変数へのアクセスといったかかわりがあると考える．よって，他の下位概念を実現するモジュールと直接関わっていなくても，他のモジュールを経由して関わっているモジュールも制約を満たすと考えたい．さらに，制約の対象となっている下位概念のいずれのランキングにも存在しているモジュールは，それひとつで両下位概念を実現しうるため，これも制約を満たすと考えたい．

上記を満たすよう，概念間の制約に対応するモジュール間の依存関係の条件を導出する．図 **4** に示すような，下位概念 c_i, c_j の間の制約 $c_i \rhd_l c_j$ を考える．モジュールの対 m_i, m_j がこの制約を満たすことを $m_i \rhd_l m_j$ と書けば，これは以下のような依存関係の条件として分解できる：

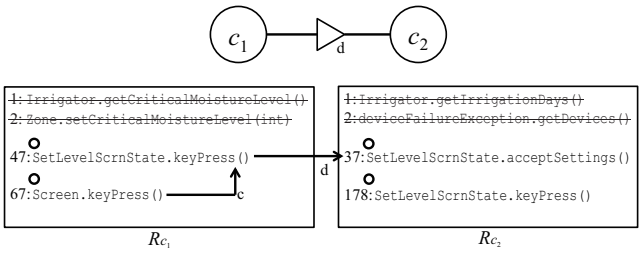


図 5　提案手法による検査

$$m_i \rhd_l m_j \triangleq \exists m_k \in R_{c_i}, m_l \in R_{c_j} \bullet m_i \rightarrow^* m_k \rightarrow_l^? m_l \rightarrow^* m_j$$

ここで，$\rightarrow^*$ は呼び出しおよびデータ依存関係の 0 回以上の繰り返し（ただし特定のランキング中のモジュール群に限る），$\rightarrow_l^?$ は種類 l の依存関係の 0 回以上 1 回以下の繰り返しを表す．

図 **5** に制約検査の例を示す．図上部の制約 $c_1 \rhd_{\mathrm{d}} c_2$ を用いて，下位概念 c_1 のランキング R_{c_1} と c_2 のランキング R_{c_2} をフィルタリングする．ここで，モジュール間の依存解析により，データ依存関係 `SetLevelScrnState.keyPress()` $\rightarrow_{\mathrm{d}}$ `SetLevelScrnState.acceptSettings()` を得られているとする．これらの依存関係は下位概念間の制約を満たすため，R_{c_1} の 47 位，R_{c_2} の 37 位は制約を満たす．また，呼び出し依存関係 `Screen.KeyPress()` $\rightarrow_{\mathrm{c}}$ `SetLevelScrnState.keyPress()` も得られたとすると，R_{c_1} の 67 位も候補となる．さらに，`SetLevelScrnState.keyPress()` は R_{c_1}, R_{c_2} の両方に含まれるため，R_{c_2} の 178 位も制約を満たす．このようにして制約を満たすモジュールを特定し，R_{c_1} の 1 位や 2 位，R_{c_2} の 1 位，2 位といった，余ったモジュールを制約を満たさないとしてランキングから削除する．

提案手法では，与えられた構造クエリ Q 中のすべての制約を検査し，制約を満足しないモジュールをランキングから除外していく．除外ができなくなったときのランキング，すなわち，制約をすべて満足する最大のランキング集合を，提案手法のフィルタリング結果として出力する．

4.　支援ツールの実装

提案手法を支援するツールの実装を行った．このツールではモジュール間の依存関係の抽出に，実行トレースとソースコードの静的解析情報を用いる．支援ツールの構成を図 **6** に示す．支援ツールは既存のツールである yebisu, inFamix，TraceLab，および，提案手法を実現するツール DependCalc によって構成される．図中の矢印は各構成要素の入出力関係を表し，ラベルは要素間で受け渡されるデータを表す．

支援ツールが持つ各構成要素について述べる．本ツールでは，時間依存関係及び呼び出し依存関係を，動的解析器

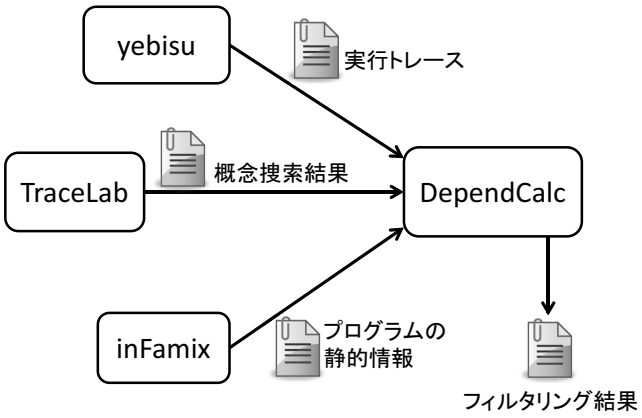


図 6　支援ツール構成

の yebisu [8] を用いて抽出する．yebisu はプログラムを実行した際に呼び出したモジュールの呼び出した順序と呼び出し関係を記録した実行トレースを出力する．この実行トレースを用いて 2 つの依存関係を抽出する．また，データ依存関係を inFamix [9] によって出力されたプログラムの静的情報を利用して抽出する．inFamix はソースコードを入力することで，ソースコード上のモジュールコールや変数への参照といったプログラムの静的情報を出力する．フィルタリング対象の概念捜索手法として Marcus らの語彙的解析手法 [10] を，TraceLab [11] を用いて適用した．TraceLab の入力はソースコードと捜索を行う概念に関する文書である．各下位概念に関する文書を入力することで各下位概念の概念捜索結果を出力する．最後に，3 つの既存ツールが出力するデータと，下位概念間の制約のクエリを入力とした，下位概念間の制約の検査を行う部分を新たに DependCalc として実装した．このツールは入力されたクエリをモジュール間の依存関係を用いて検査し，制約を満たさないモジュールを除外した下位概念のランキングのフィルタリング結果を出力する．

5. 評価実験

提案手法の実用性を評価する為，実験による評価を行った．本実験の目的は，既存の機能捜索手法と比べ，出力のランキングに含まれる正解の順位を上昇させることができるかどうかである．開発者にランキングを渡した際に，そのランキングを上から順に理解することが想定されるため，正解のモジュールを上位にすることで理解を効率的に行うことできると考える．

5.1 実験方法

実験対象としてオープンソースソフトウェアの AquaLush [12] を用いた．AquaLush では実装している各機能に対するユースケース記述を持ち，またユースケース記述，各仕様書，ソースコード間のトレーサビリティが明確化されている．そのため，下位概念への分解や対応するモジュールの発見が容易であることから実験対象として選択した．AquaLush の全 8 つのユースケース記述のうち，自動テストコードの存在する 6 つを機能記述とした．そのユースケースステップをそれぞれの機能が持つ下位概念とした．

本実験では，既存手法と提案手法によって得られた，それぞれの下位概念のランキングについて正解の順位を比較した．提案手法で抽出を行うモジュール間の依存関係は実行トレースを用いて動的に抽出したものと，動的な情報を用いずソースコードから静的にのみ抽出したものの 2 種類を用意し，それぞれの依存関係を用いて，下位概念間の制約検査を行った．動的なモジュール間の依存関係によるフィルタリングでは，既存手法による結果も実行トレースに記録されたモジュールに絞り込んでから比較した．

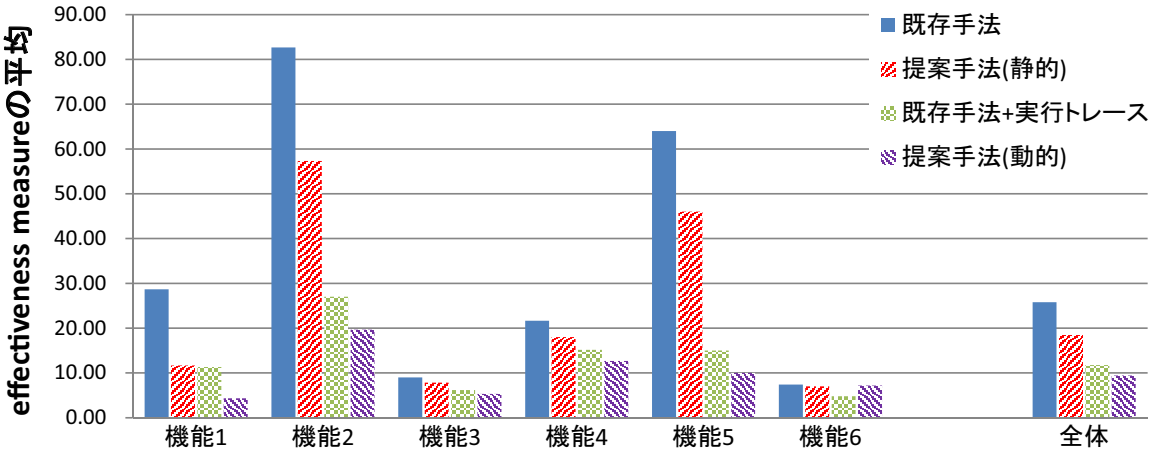
順位の比較には，effectiveness measure [13] を用いた．effectiveness measure は，ランキングに現れる正解のモジュールのうち，最上位にあるモジュールの順位である．6 つの機能が持つそれぞれの下位概念に対して，4 つの手法によって得られたランキングの effectiveness measure を計算し，機能毎の平均をとることで，各機能の正解のモジュールの順位の改善度を比較した．ただし，平均の計算には，ランキングに正解が残った下位概念のみを用いた．

本論文の著者らにより正解を作成した．AquaLush の対象となるユースケース記述から，追跡可能な設計情報，モジュール名，ファイル名といった情報を抽出した．次に，ユースケース記述に対応した自動テストケースを実行し，実行トレースを得た．これらの共通部分を取り出し，その中から各ステップの実装に強く関わるモジュールを手動で探索し，得られたモジュールと，そのモジュールから呼び出しのあるモジュールの集合を作成した．集合を著者らのうち 2 名によって協議を行い，機能と関連性の低いと判断したモジュールを取り除き，残りを正解として用いた．

入力する下位概念間の制約のクエリは，著者らによって作成を行った．

5.2 実験結果

本実験の結果を図 **7** に示す．図のグラフは，各手法によって得られた各機能が持つ下位概念の effectiveness measure の平均を表す．各機能の 4 本のグラフは左から順に，既存手法，静的な依存関係を用いた提案手法，動的に絞り込みをした既存手法，動的な依存関係を用いた提案手法による effectiveness measure の平均を表す．また，右端のグラフは，6 つの機能の下位概念すべての effectivenss measure の平均である．静的な依存関係を用いた提案手法は既存手法と比較し，動的な依存関係を用いた提案手法の結果を実行トレースを用いてしぼり込んだものと比較する．グラフを見ると，既存手法に対してはすべての機能で，動的に絞り込みをした既存手法との比較では 6 つの機能のうち，5



	機能1	機能2	機能3	機能4	機能5	機能6	全体
既存手法	28.67	82.67	9.00	21.67	64.00	7.40	25.79
提案手法(静的)	11.67	57.33	7.83	18.00	46.00	7.00	18.46
既存手法+実行トレース	11.33	27.00	6.17	15.17	15.00	4.80	11.75
提案手法(動的)	4.33	19.67	5.33	12.67	10.00	7.20	9.42

図 7　実験結果

つで effectiveness measure の値を削減できていることがわかる．また，機能全体についても，提案手法によるランキングの値が既存のものより低く，効率的にフィルタリングが行えたことがわかる．機能 6 については，動的に絞り込みをした既存手法に比べ，提案手法の結果が悪くなっている．これは，下位概念間の制約の検査によって，最上位の正解のモジュールが除外されてしまったために，次点のモジュールの effectiveness measure を用いた為である．effectiveness measure の削減量としてみると，機能全体では静的依存関係で 17.33 ポイント，動的依存関係で 2.33 ポイント削減することができた．また，最も効果の大きかった機能 1 については静的な提案手法で 17 ポイント，動的な提案手法で 7 ポイント削減することができた．

以上の結果から，提案手法は既存手法による下位概念の捜索結果をフィルタリングすることで，正解の順位を向上させることができたと考える．

5.3　考察

本実験で得られた各手法による下位概念のランキングを比較した．その結果，順位の低い正解のモジュールをフィルタリングによって大幅に順位を上昇させたことが分かった．順位の上昇量は最も大きかった下位概念で 17 ポイントであった．このことは，提案手法は effectiveness measure の高い，難度の高いタスクについて効率的にフィルタリングを行う事ができることを示唆している．

一方で，制約の検査によって正解としたモジュールがランキングから除外されてしまうということが起こった．このようなモジュールに対して分析を行った結果，本手法で想定していない依存関係の流れを持ち，制約を満たしていなかった．例えば機能 6 の下位概念 c_1, c_2 の間にはデータ

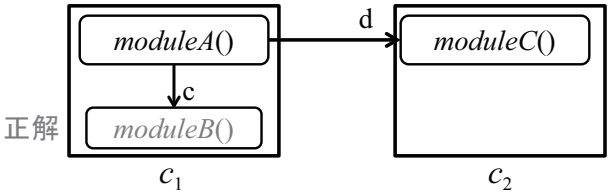


図 8　検出できなかった依存関係の例

制約があり，下位概念のランキングのモジュール間の依存関係は図 8 のようであった．このとき，c_1 の正解である *moduleB* で得たデータを *moduleA* を経由して *moduleC* へ流れていたが，提案手法の制約はこれを通すことができず，*moduleB* はランキングから除外された．この問題の解決のために，制約を満たすモジュール間の依存関係のパターンの精査を行う必要があると考える．

また，フィルタリング対象の既存手法による概念捜索結果に正解のモジュールが 1 つも現れないランキングが出力されるという問題も発生した．この問題は，提案手法の不備ではないが，このような問題を解決するためにはより検出能力の高い既存手法を用いることが望まれる．

5.4　妥当性への脅威

本実験で作成した正解が都合の良いように意図的に作成されていた可能性がある．本実験で用いた正解は，著者らのうち 2 名によって協議を行い作成したものである為，そのような意図は可能な限り排除を行うことができたと判断する．

本実験では検査を行う下位概念間の制約のクエリを入力する必要があるが，クエリに制約を正しく記述していないために，評価値を不当な値にしている可能性がある．本実験で用いたクエリは筆者が下位概念の記述を読み，張られ

ていると判断した制約である．その為，入力したクエリにバイアスがかかっていないとは言えないが，下位概念間の制約は，3.2 節で記述した判断材料を用いることで．恣意的な判断は極力排除した．

本実験の対象とした AquaLush は，ユースケース記述があり，かつ記述と，仕様書や，ソースコードでの実装箇所とのトレーサビリティが確保されているプログラムである．しかし，多くのプログラムではユースケース記述のような手順の書かれた仕様書を得ることができず，機能を下位概念へと分解することや下位概念間の依存関係の制約を記述することが難しいかもしれない．そのような場合には，開発者が考える機能の行うべき処理を下位概念とし，その間にあるべき依存関係をクエリとして記述することで，同様の実験を行うことができると考える．

6. 関連研究

既存の機能捜索手法のうち，語彙的解析と動的解析を組み合わせた機能捜索手法には例えば以下の 2 つがあるが，本論文とは，目的や手法が異なる．

それぞれのランキングの組み合わせ

語彙的解析によって得られたランキングの値と，動的解析によって得られたランキングの値とを組合わせることで，より実装している可能性の高いモジュールを出力する手法がある [13][14]．これらの手法は，2 つの手法をそれぞれ適用するが，提案手法は語彙的解析によって得たランキングに対するフィルタリングを行う為，本手法は異なる．

語彙的解析の探索範囲の削減

実行トレース内に記録されたモジュールのみに対して語彙的解析を行う事で，探索範囲を機能の実装に関わるモジュールに限定する手法がある [15]．この手法は，動的解析を行う回数を縮小させる事を目的としているが，提案手法は下位概念間の制約を検査する為に動的解析を利用している為，この手法とは目的が異なる．

7. おわりに

本論文では，既存の機能捜索手法の結果を下位概念間の依存関係による制約を検査することで，より良い結果を得る機能捜索手法を提案した．機能を下位概念に分解し，各下位概念の概念捜索結果が下位概念間の制約を満たすかを検査し，制約を満たさないモジュールを除外することで概念ごとの捜索を行う．提案手法を 3 つの下位概念間の依存関係を扱う手法として実現を行い，支援ツールとして実装した．支援ツールを用いて既存の概念捜索手法との比較による評価を行い，既存手法によって得られたランキングから，依存関係を用いた検査によって開発者の労力を削減できることを示した．

今後の課題として，下位概念間の制約を満たすモジュール間の依存関係パターンの精査や，フィルタリング対象の既存手法をより検出能力の高いものに変更するといったことが挙げられる．

謝辞　本研究の一部は文部科学省科学研究費補助金（課題番号：23700030）の助成を受けた．

参考文献

[1] The Institute of Electrical and Electronics Engineers (IEEE): *830-1998 - IEEE Recommended Practice for Software Requirements Specifications* (1998).
[2] Murphy, G., Kersten, M., Robillard, M. and Čubranič, D.: The Emergent Structure of Development Tasks, *Proc. ECOOP*, LNCS, Vol. 3586, pp. 33–48 (2005).
[3] Dit, B., Revelle, M., Gethers, M. and Poshyvanyk, D.: Feature Location in Source Code: A Taxonomy and Survey, *J. Softw.: Evol. and Proc.*, Vol. 25, No. 1, pp. 53–95 (2013).
[4] Thomas, V. and Kurt, N.: Maintaining Program Understanding - Issues, Tools, and Future Directions, *Nordic Journal of Computing*, Vol. 11 (2004).
[5] Gay, G., Haiduc, S., Marcus, A. and Menzies, T.: On the Use of Relevance Feedback in IR-based Concept Location, *Proc. ICSM*, pp. 351–360 (2009).
[6] Rajlich, V. and Wilde, N.: The Role of Concepts in Program Comprehension, *Proc. IWPC*, pp. 271–278 (2002).
[7] Rajlich, V.: *Software Engineering: The Current Practice*, CRC Press (2011).
[8] Kazato, H., Hayashi, S., Oshima, T., Miyata, S., Hoshino, T. and Saeki, M.: Extracting and Visualizing Implementation Structure of Features, *Proc. APSEC*, pp. 476–484 (2013).
[9] inFamix: http://www.intooitus.com/products/infamix.
[10] Marcus, A., Sergeyev, A., Rajlich, V. and Maletic, J.: An Information Retrieval Approach to Concept Location in Source Code, *Proc. WCRE*, pp. 214–223 (2004).
[11] Dit, B., Moritz, E. and Poshyvanyk, D.: A TraceLab-based Solution for Creating, Conducting, and Sharing Feature Location Experiments, *Proc. ICPC*, pp. 203–208 (2012).
[12] Ben Charrada, E., Caspar, D., Jeanneret, C. and Glinz, M.: Towards a Benchmark for Traceability, *Proc. IWPSE-EVOL*, pp. 21–30 (2011).
[13] Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G. and Rajlich, V.: Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval, *IEEE Trans. Softw. Eng.*, Vol. 33, No. 6, pp. 420–432 (2007).
[14] Revelle, M., Dit, B. and Poshyvanyk, D.: Using Data Fusion and Web Mining to Support Feature Location in Software, *Proc. ICPC*, pp. 14–23 (2010).
[15] Liu, D., Marcus, A., Poshyvanyk, D. and Rajlich, V.: Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace, *Proc. ASE*, pp. 234–243 (2007).