Java メソッドの実行状況の自動抽出に向けて

石尾　隆[†1]　鹿島　悠[†1]

プログラムのあるメソッドの振舞いを理解するには，そのメソッドが，どのような状況で実行されるかを知る必要がある．本研究では，データフロー解析を用いて，Java メソッドの実行中に使用されるデータの集合を計算し，そのメソッドが依存するシステムの状態として示す手法を提案する．本稿では，研究の基本アイディアと，実現に向けた課題を解説する．

Towards Automatic Extraction of Execution Condition of Java Method

Takashi Ishio[†1] and Yu Kashima[†1]

To understand the behavior of a Java method, developers need to understand how the method interacts with the other methods in the system. In this research, we propose a data-flow analysis to automatically extract input data used by a Java method for understanding the behavior of the method. This position paper explains our basic idea and challenges.

1. はじめに

情報システムの仕組みを理解するには，その構造を分析し，構成要素それぞれの振舞いの概要を理解することが重要である．Java で記述されたソフトウェアの場合，構造の理解にはパッケージ構造やクラス階層の情報を用いる．また，振舞いを理解するために利用できる道具には，ソースコード上でのメソッド間の呼び出し関係を表したコールグラフ，設計時に作られるシーケンス図，そして実行時情報を分析するデバッガなどがある．

システム全体の振舞いを理解することが困難な理由の１つに，構成要素であるメソッドの役割を知る方法がないことが挙げられる．システムの主要な機能を実装するメソッドを特定したとしても，その機能がいつ，どのような状況で実行されるのかを知ることができない．コールグラフやシーケンス図を用いると，他のメソッド群との実行順序の関係を知ることはできるが，その時点でのプログラムの状態が直接示されることはない．デバッガを使えば実行時の具体的な変数の内容を知ることはできるが，システム全体の状態を，ある時刻の個々の変数の値のみから推測することは困難である．

そこで，本研究では，開発者が注目する１つの手続きが，プログラムの状態にどのように依存しているかを提示する手法を提案する．具体的には，１つのメソッドの実行開始から終了までに使用される（そのメソッドで使用される他のメソッドまで含めた）入力データを収集，整理することで，１つのメソッドとして表現された機能を実行する際の前提条件として抽出する．

†1 大阪大学
Osaka University

2. アプローチ

本研究では，Java プログラムを対象として，１つのメソッドが指定されたとき，その実行に関係する入力データを自動抽出する．

Java における手続き間のデータフローは，メソッド呼び出しの引数と戻り値，例外，そしてフィールドの 4 種類のみである．そのため，１つのメソッドの実行に必要なデータの集合は，そのメソッド自身の引数と，メソッドの実行中に参照するフィールドの集合によって表現できる．メソッドが内部で他のメソッドを呼ぶ場合は，呼び出し先のメソッドに関しても，参照するフィールドの集合を取得すればよい．そこで，メソッド m の実行に必要なデータの集合 $R(m)$ を次のように定義する．

$$R(m) = Params(m) \cup Fields(m) \cup \bigcup_{m' \in Calls(m)} R(m')$$

ここで，$Params(m)$ はメソッド m の引数の集合，

$Fields(m)$ はメソッド m が読み出すフィールドの集合，$Calls(m)$ はメソッド m からの呼び出しで実行される可能性があるメソッドの集合を指す．

この定義で求めた $R(m)$ には，m の実行中に作成されるデータも含まれている．そのため，メソッド m の実行中に作成されるデータを $W(m)$ として，m が参照する入力データ $I(m)$ を次のように定義する．

$$I(m) = R(m) - W(m)$$

$W(m)$ の要素は，$R(m)$ の部分集合である．$R(m)$ の要素となっている各フィールド，引数について，m の実行中に出現するすべての参照位置を取り出し，その参照される値を定義しているプログラム文を特定する．すべての定義文が m の実行に含まれていれば，その変数は $W(m)$ の要素となる．

図 1 のサンプルコードを用いて，計算の例を示す，`getGrade` メソッドは，一見すると `averageScore` フィールドに依存しているように見える．しかし，`averageScore` の値は，1 行目で呼び出す `qualified` メソッドの中で代入された値のみが使われるため，

$$R(getGrade) = \{averageScore, scores\}$$
$$W(getGrade) = \{averageScore\}$$
$$I(getGrade) = \{scores\}$$

となる．つまり，`getGrade` メソッドは，`scores` フィールドの値さえあれば動作することが分かる．

$I(m)$ は，m の実行よりも前に定義され，m の実行中に使用される引数，フィールドの集合となる．この情報を開発者に示すことで，メソッドが間接的に依存するシステムの状態を提示することができる．また，システムの主要な機能に対応する各メソッドについて，更新されるデータに関する情報を合わせて収集することで，機能間の依存関係を示すことも可能であると考えている．

3. 研究の位置付け

本研究は，データフロー解析の 1 つであり，プログラムスライシング[3] を応用した手法である．$W(m)$ を求める計算は，プログラムスライシングにおいて，フィールドに関するデータ依存関係を辿る操作に対応する．スライシングでは，m の実行に影響を与えるプログラム文の集合を求めるが，本研究では，m の実行よりも前に定義されていなければならないデータの集合を求めることを目指している．副作用解析[2] と比べた場合，メソッドが行う書き込み処理を調査するかわりに，メソッドが行う読み込み処理を調査する手法となっている．

```
class Student {
  double averageScore;
  List<Integer> scores = new ArrayList<>();

  public boolean qualified() {
    /* computeAverage の定義は省略 */
    averageScore = computeAverage(scores);
    return (averageScore >= 60);
  }
  public void addScore(int score) {
    scores.add(score);
  }
  public String getGrade() {
    if (!qualified()) return "F";
    else if (averageScore > 90) return "A";
    else if (averageScore > 70) return "B";
    else return "C";
  }
}
```

図 1 サンプルコード
Fig. 1 An Example Code

本研究はデータの値域についての情報を抽出するわけではないが，入力データが存在することは，メソッドの事前条件の一種と考えられる．メソッド m の事前条件は，通常，所属するオブジェクトの状態についてのみ言及する[1] が，本研究では，m の実行に関係するすべての状態を収集することを目指している．

4. まとめと今後の課題

本稿では，あるメソッドの実行状況を理解するため，その実行に必要となる入力データを抽出する手法を提案した．実現に向けた課題としては，解析結果として得られた $I(m)$ の情報を適切な形式で可視化する方法が挙げられる．たとえば 1 つのクラスのすべてのフィールドが入力データとして使用される場合には，クラス自体を入力データの代表として提示するといったように，データフローのまとまりを自動的に認識する技術を構築することを計画している．

謝辞 本研究は科研費（23680001）の助成を受けたものである．

参 考 文 献

1) Meyer, B.: *Object-Oriented Software Construction*, Prentice Hall, 2nd edition (2000).
2) Rountev, A.: Precise Identification of Side-Effect-Free Methods in Java, *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pp.82–91 (2004).
3) Weiser, M.: Program Slicing, *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, pp.352–357 (1984).