レガシーシステム分析のための パターンマッチによるソースコードの自動抽象化

岡田 譲二[1,2,a)]　Abhay Parvati[1,b)]　石尾 隆[3,c)]　坂田 祐司[1,d)]　井上 克郎[2,e)]

概要：レガシーシステムのソースは SQL 相当の内容をループ文と条件文を駆使してプログラムとして実装している。本稿ではあらかじめ用意しておいたソースコードパターンをソースコードにマッチさせることで自動的にソースコードを抽象化する手法を提案する．

1. はじめに

業務上は重要だが保守の継続が困難なメインフレーム上の基幹システム (以降はレガシーシステムと呼ぶ) がいまだに多数存在している [1]．このようなレガシーシステムを新しいプログラミング言語や実行環境で動作するシステムへと再構築することが課題となっている．

一般的なソフトウェアシステムでは RDB を用いてデータを管理し， SQL を用いてデータの検索や更新を行うが，レガシーシステムでは VSAM や PS といったシーケンシャルなファイルを用いてデータを管理し，データの検索や更新を行うロジックをファイルの読み書きとループや条件文を用いてプログラムとして記述することが多い [2]．レガシーシステムにおいてデータの検索や更新を行う典型的なロジックとして，マッチング処理やコントロールブレイク処理などが広く知られている [3]．

これらの典型的なロジックは SQL で書き換えると 1 〜数行程度の内容にも関わらず，COBOL などで実装すると 10 行以上のボリュームとなり，プログラムの可読性を損なっている．例えば **図 1** に示す COBOL のコードは，FILE1 と FILE2 の I-KEY の値が一致するデータを取り出すマッチング処理であるが，同等の処理を SQL で記述すると **図 2** に示すように 1 行で記述できる．このサンプルコードは行数が短いため構造が把握しやすいが，実際のコードでは 5 行目に示す SELECT アイテムの設定に相当する記述が数十行続いているなど，この構造を把握するには労力が必要である．

```
1  read FILE1
2  read FILE2
3  perform until (I-KEY of FILE1 = high-value and
     I-KEY of FILE2 = high-value)
4    if (I-KEY of FILE1 = I-KEY of FILE2)
5      move I-KEY of FILE2 to O-KEY of FILE3
6      if (AGE of FILE2 > 20)
7        write FILE3
8      end-if
9      read FILE2
10   else if (I-KEY of FILE1 < I-KEY of FILE2)
11     read FILE1
12   else
13     read FILE2
14   end-if
15 end-perform
```

図 1　マッチング処理のサンプルコード

```
1  select I-KEY of FILE2 from FILE2 inner join FILE1
     on FILE1.I-KEY = FILE2.I-KEY where FILE2.AGE > 20
```

図 2　図 1 に対応する SQL

本研究では，典型的なロジックで書かれたレガシーシステムのソースコードを SQL やそれに類する表記方法に変換して提示することで，開発者が迅速にソースコードの概要を把握することを可能とする手法を提案する．具体的な方法としては，典型的なロジックが制御構造で判別することができることに着目し，制御構造に対応するパターンマッチによる変換を用いる．同様のアイディアに基づく既

[1] 株式会社 NTT データ 技術革新統括本部
NTT DATA Corporation, Koto, Tokyo 176–0024, Japan
[2] 大阪大学 大学院情報科学研究科
Osaka University, Suita, Osaka 565–0871, Japan
[3] 奈良先端科学技術大学院大学 情報科学研究科
Nara Institute of Science and Technology, Ikoma, Nara 630–0192, Japan
[a)] okadaju@nttdata.co.jp
[b)] abaip@pm.nttdata.co.jp
[c)] ishio@is.naist.jp
[d)] sakatayu@nttdata.co.jp
[e)] inoue@ist.osaka-u.ac.jp

```
read $x;
read $y;
loop (_) {
  branch subset {
    case ($a == $b) {
      repeat (_ of $z = _);
      write $z;
      read $y;
    }
    case ($a < $b) {
      read $x;
    }
    case ($a > $b) {
      read $y;
    }
  }
}
```

図 3 マッチング処理に対応するパターン

存研究として，Sridhara ら [4] は代入文・条件分岐文の連続や，ループ文内のパターンに着目してソースコードの概要を自然言語の説明文で生成する手法を提案している．しかしこの手法では，パターンにマッチしなかったソースコード行がどれだけあるのか分からない．このため，パターンにマッチした部分がソースコード中のごく一部だった場合，生成された説明文がソースコードのごく一部だけしか説明できず，マッチしていない部分に概要として重要な部分があったとしても見落としてしまう可能性がある．

2. 提案手法

本研究では，予め作成したプログラムの制御構造に着目したパターンをソースコードにマッチさせることでソースコードの抽象化を行うとともに，マッチしなかった部分をユーザに提示することで，重要な内容を見落とすことなくレガシーシステムの分析を実施できる手法を提案する．

本手法ではパターンマッチに利用するパターンに独自の表記法を用いる．紙面の都合上表記方法の詳細は割愛するが，本表記法はループ文にマッチする *loop* や，条件分岐文・多分岐文の条件順序に関わらずマッチする *branchsubset*，同様の記述の繰り返しにマッチする *repeat* などの構文を有する．本表記法で記述したマッチング処理のパターンを**図 3** に示す．

本手法は以下の 3 つのステップから構成される．

(1) 単純な文への変換
(2) パターンマッチによる概要の出力
(3) マッチしなかった文の出力

最初のステップでは，プログラミング言語による差異を無くすとともに，次のステップでのパターンマッチを行いやすくするため，各文を単純な文（ループ文，条件分岐文，外部入力文，外部出力文，代入文）に変換する．

次に，変換後の構文木に対して予め作成しておいたパターンをマッチさせ，マッチした場合はそのパターンに対応する概要を出力する．パターンは複数個存在するため，本ステップをパターンの数だけ繰り返し実施し，マッチしたパターンの分だけ概要を出力する．

最後に，全てのパターンがマッチしなかった文を抽出し，関係する文とともに出力する．関係する文とは，以下の条件に合致する文である．

- マッチしなかった文がいずれかの制御ブロックに含まれる場合は，その制御ブロックを持つ文
- マッチしなかった文が制御ブロックを持つ場合，そのブロック中に含まれる外部入力文と外部出力文

例えば，図 1 のソースコードに対して，図 3 のパターンマッチした場合，$\$x$ に $FILE1$，$\$y$ に $FILE2$，$\$a$ に $I-KEY of FILE1$，$\$b$ に $I-KEY of FILE2$ がそれぞれマッチし，*loop* として 3 行目の *perform* 文，*branch subset* 及びその配下の *case* として 4 行目，10 行目，12 行目の各 *if* 文がマッチする．このため，このソースコードは概要として「マッチング処理」が出力される．一方で，6 行目の *if* 文はこのパターンとはマッチしないため，「マッチしなかった文」としてこの 6 行目の *if* 文が出力され，「関係する文」として 3 行目の *perform* 文と 4 行目の *if* 文，7 行目の *write* 文が出力される．

このようにマッチしなかった文を出力することで概要として重要な文を見落とすことが無くなる．また，マッチしなかった文とともに関係する分も合わせて出力することで，マッチしなかった文の意味を把握しやすくなると考えている．

3. 現在の開発状況と今後の課題

現在我々は，パターンを解釈し構文木に対してパターンマッチングを行えるプロトタイプを開発したところである．今後は本プロトタイプを実システムに適用することで，どのくらいのソースコードが自動で抽象化できるのか，パターンに合致しなかった箇所から新しいパターンを定義することができるかなどを調査したいと考えている．

参考文献

[1] Khadka, R., Batlajery, B. V., Saeidi, A. M., Jansen, S. and Hage, J.: How do professionals perceive legacy systems and software modernization?, *Proceedings of the 36th International Conference on Software Engineering*, pp. 36–47 (2014).
[2] 神居俊哉，高尾　司：メインフレーム実践ハンドブック. z/OS (MVS), MSP, VOS3 のしくみと使い方，リックテレコム (2009).
[3] 穂積和子，藤山秋良：COBOL の総合研究，第 2 種情報処理試験合格ゼミ 6，技術評論社 (1995).
[4] Sridhara, G., Pollock, L. and Vijay-Shanker, K.: Automatically Detecting and Describing High Level Actions Within Methods, *Proceedings of the 33rd International Conference on Software Engineering*, pp. 101–110 (2011).