プログラミング言語横断類似コード断片検出ツールの試作

神谷 年洋
島根大学学術研究院理工学系
kamiya@cis.shimane-u.ac.jp

要旨

複数のプログラミング言語で記述されたプログラム，あるいは，異なるプログラミング言語へ書き換えている途中のプログラムに適用することを目的とした，類似コード断片を検出する動的解析手法を提案する．また，その実装により，*Java* により記述された *300* 行程度のプログラムとそれを *Python* に変換したものの間で類似コード断片を検出した適用例を紹介する．

1. はじめに

ひとつのソフトウェアの実装に複数のプログラミング言語が利用されることが一般的になっている．例えば，Web アプリケーションの場合は，フロントエンドが JavaScript で，バックエンドが Python，Ruby，Java 等で記述されることがある．同様に，ソフトウェアの記述におけるパラダイムシフト（手続き，オブジェクト指向，関数型）やいわゆるレガシーマイグレーションによる別のプログラミング言語（Cobol → Java 等）への書き換えといった要因により，システム全体として見たときに複数のプログラミング言語が混在した記述となる状況が生じている．

複数のプログラミング言語により記述されたソフトウェアの開発・保守の問題として，解析ツールが単一の言語にしか対応していない場合には，システム全体構造を分析したり，品質を評価したりする作業が困難になることが挙げられる．たとえば，コードクローンとは，ソフトウェアのソースコード内に存在する類似コードのことであり，典型的には，開発者がソースコードをコピー＆ペーストすることによって生じる．コードクローンが存在すると，ソースコードを修正する場合には，重複したコード断片のすべてについて修正する（判断を行う）必要が生じるため，ソフトウェア保守上の問題であるとされている．しかし，複数のプログラミング言語により記述されたソフトウェアに対応したコードクローン検出ツールはあまり存在しない（2 節で後述）．

本稿では，著者らが提案した手法 [2] を発展させた，動的解析により異なるプログラミング言語で記述されたコードから類似したコード断片（すなわち，コードクローン）を検出する手法を提案する．さらに，手法の初期的な実装を用いて，Java により記述さた 300 行程度のプログラムと，それを Python に変換したものの間から類似コード断片を検出した試みについても報告する．

2. 関連研究

異なるプログラミング言語で記述されたプログラムの間で類似コードを検出する手法としてこれまでに発表されているものは，[1][4] のような，それらの言語に共通の中間言語を利用して検出するものである．これらの手法を適用するためにはそのような中間言語が存在することが前提となる．これに対して，提案手法の前提は，プログラムを実行して実行トレースを取得できること，および，実行トレースに手続きの呼び出しや戻り，実引数の値や戻り値が記録されることである（3 節で後述）．

複数の言語で記述されたプログラム中の類似コードを求める手法の研究としては，HTML 中で JavaScript で記述された手続きを呼び出すようなものを対象として，呼び出される JavaScript の手続きの類似性を用いるもの [3] がある．

3. 実行トレース中に参照する値の類似性に基づく類似コード断片検出手法

提案する類似コード検出手法のステップを説明する．本節の説明では，プログラムの実行中に参照される値の類似性のみを用いていて，呼び出されているメソッドの名前の類似性等は用いていないことにも留意されたい．

ステップ 1 実行トレースの取得: 対象となる 2 つのプログラム（互いに異なる言語により記述された）を実行し，その際に双方の実行トレースを取得する．実行トレースに記録されるものは，手続き（関数やメソッド）が呼び出されるイベントと戻るイベントについて，それぞれの手続きの名前，ソースコード内での位置，実引数の値（あるいは戻り値）である．

また，補助的に，実行中にリテラル（定数）が参照されるイベントについて，その定数の値が取得可能であれば，それも記録する．

ステップ 2 手続き呼び出しと参照する値の特定: 実行トレースからコールツリーを構築し，コールツリーの節点である手続きの呼び出しのそれぞれ（c）について，その実行の間に参照する値の集合（V(c) とする）を特定する．ここでは，値とみなすことが可能な，真偽値，数値，文字列のみを残し，それ以外は除去した．また，文字列については，文字列を空白や記号で区切ったうえで空白は除去した．

コールツリーの葉に相当する節点の手続き呼び出し c については，その実行中に参照されるリテラルの集合が V(c) となる．それ以外の節点の手続き呼び出し d については，実行中に参照されるリテラル，d が直接あるいは間接的に呼び出す手続き e の実引数や戻り値，および，V(e) の要素を含む集合を作り V(d) とする．

ステップ 3 参照する値が類似した手続き呼び出しの特定: 双方の実行トレース中の手続き呼び出しのペア (c, d) から，V(c) と V(d) が類似しているものを選択する．

選択の基準には様々なものが考えられるが，4 で示した適用例では，ソースコード中のある手続き f について，p の呼び出し c を含むすべてのペア (c_i, d_i) $(i = 1, 2, ..., n)$ のうち，$V(c_i)$ と $V(d_i)$ のコサイン類似度が最大となるペアを選択した．ただし，共通の要素数が 3 に満たないものは取り除いた．

表 1. 類似コード検出例の検出結果集計

正しく特定できた	4 個
呼び出し階層が 1 つ深い（あるいは浅い）	4 個
全く異なる手続きが検出された	1 個
未検出	5 個

表 2. 類似コード検出例の検出結果の内容

Python の手続き (c)	Java の手続き	V(c)	コサイン類似度
main	–	(55)	–
startMatch	←	55	0.528
startGame	←	35	0.478
playerMove	←	20	0.496
put	startGame,-1	17	0.305
display	←	16	0.567
compMove	println*,+1	14	0.546
checkForWin	startGame,-1	10	0.199
move	playerMoved	8	0.208
playerMoved	–	(7)	–
init	–	(6)	–
mark	playerMoved,-1	6	0.178
fixWeights	–	(4)	–
__init__	–	(1)	–

「←」は Python のメソッドに対応する（書き直された元の）Java メソッドが類似コードとして検出されたもの.「–」は類似コードが検出されなかったもの.「+1」（あるいは「-1」）が付されたものは呼び出しの階層が 1 つ深い（浅い）手続きが類似コードとして検出されていたもの.「*」が付された手続きは標準 I/O ライブラリのメソッド.

4. 小規模なプログラムへの適用例

適用対象のプログラムは，ある三目並べのプログラムを Java で記述したもの[1]と，それを著者が Python に書き直したもの[2]である．書き換えはほぼ 1 行ずつ対応するように行われ，双方のプログラムとも，コンストラクタ 1 つを含めて 14 個のメソッド定義を含んでいる．

提案手法の初期的な実装により，これらプログラムの間から類似コード断片である手続きを特定した．表 1 に，検出結果の集計を示す．表 2 に各手続きについての検出

[1] http://www.rosettacode.org/wiki/Tic-tac-toe#Java
[2] https://gist.github.com/tos-kamiya/06e2f6298210001cc54b5b2439cde494

結果，すなわち，Python のプログラムの手続き 14 個のそれぞれについて，提案手法で最も類似度が高いと判定された Java プログラムの手続きを示す．全体的な傾向としては，手続きが参照する値の個数が多いほうが検出が正確になっていた．

4.1. 「未検出」になったメソッドの原因調査

Python のメソッドとその元となった Java のメソッドのペアのうち，表 1 で未検出だった（すなわち，類似コードであると全く判定できなかった）5 個について，`V(c)` およびソースコードにあたってその原因を調査した．

`main`, `playerMoved`: メソッドの処理内容が大きく異なっていた．`main` の場合には，プログラムのエントリポイントにあたるメソッドのため処理の内容はライブラリやクラスのロードであった．ロードすべきクラスの名前などが Java と Python とで大きく異なるために，類似しているとは判定されなかった．`playerMoved` の場合には，このメソッドの処理の内容に int 型の値を文字列に変換して文字列に連結することが含まれていた．プログラミング言語の違いにより，Java では暗黙の型変換，Python では明示的な変換と異なる記述となり，類似しているとは判定されなかった．

`init`, `fixWeights`, `__init__`: メソッドから参照される値の数が少なく類似性が判定できなかった．これらのメソッドの処理の内容の大部分は変数の参照や代入であったが，本実験で用いた実装では変数の参照や代入は実行トレースに記録されていなかった．

いずれも，本実験で用いた実装やアルゴリズムの制約に引っかかったものであり，これらを類似であると判定するためには，（実験に際して用いたパラメータを調整するなどでは対処できず）手法の実装やアルゴリズムの処理を改良すること必要であると考えられる．

5. まとめと展望

実行トレース中で参照する値の類似性により，異なるプログラミング言語で記述されたプログラムの間から類似コード断片を検出する動的解析手法を提案した．三目並べの Java プログラムとそれを Python に書き直したものに適用する例では，参照する値が 10 個以上ある手続きについては，呼び出し階層が 1 つずれたものまで含めれば大多数 (8 個中 `main` を除く 7 個) 特定が可能であった．

今後は，より大きな対象への適用実験による評価を行うとともに，提案手法のアルゴリズムや実装を改善し検出精度・性能の向上を図る．

参考文献

[1] A. Avetisyan, S. Kurmangaleev, S. Sargsyan, M. Arutunian, A. Belevantsev, “LLVM-Based Code Clone Detection Framework”, Proc. 10th International Conference on Computer Science and Information Technologies (CSIT), pp. 100–104, 2015.

[2] Toshihiro Kamiya, “An Execution-Semantic and Content-and-Context-Based Code-Clone Detection and Analysis”, Proc. IEEE 9th International Workshop on Software Clones (IWSC), pp. 1-7, 2015.

[3] 中村 勇太, 崔 恩瀞, 吉田 則裕, 春名 修介, 井上 克郎, “複数プログラミング言語で記述されたソフトウェアからのコードクローン検出”, 情報処理学会研究報告, Vol. 2016-SE-194, No. 7, pp.1–8, 2016.

[4] T. Vislavski, G. Rakić, N. Cardozo, Z. Budimac, “LICCA: A Tool for Cross-Language Clone Detection”, Proc. IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 512–516, 2018.