アサーション動的生成のためのテストケース自動生成手法の生成アサーションの妥当性評価

Adequacy Evaluation of Assertions Generated by Dynamic Generation with Automated Generation of Test Suites

宮本 敬三*　堀 直哉†　岡野 浩三‡　楠本 真二§　西本 哲¶

Summary. Program assertion increases correctness, usability, and reusability of the software. Manual annotation of the assertions is, however, incredible due to the size of recent software; thus, recently dynamic generation of such assertions attracts attention. The dynamic generation of assertions uses test-cases to execute the target program, and generates assertions by observing the traces of values of program variables. Therefore, the quality of generated asserions depends on the quality of the test-cases. We have proposed a method to generate test-cases for the dynamic generation of assertions, which uses notion of invariant coverage and model checking techniques. This paper reports experimental results of the quality of generated assertions based on our technique. The results show improvement of the quality of the assertions.

1　まえがき

プログラムの表明 (assertion) は，プログラムがソースコード中のある特定の場所で満たすべき条件を表す．DbC(Design by Contract [1]) に基づいてプログラムの表明を記述することで，ソースコードの可読性の向上やプログラム検証が行える．しかし，近年のソースコードサイズの増加にともない手動による表明生成は困難になってきている．そのため，表明の自動生成手法や自動検査手法が注目されている．

表明の生成と検査の自動化手法には，大別して静的手法と動的手法の 2 種類がある [2]．静的手法 [3–9] はソースコードの状態を表すモデルを生成し，実行しうる全ての状態を求めるため，精度の高い表明の自動生成 [3–6] や自動検査 [7–9] が可能である．しかし，一般的にはモデルの状態数に対するスケーラビリティが課題である．一方，動的手法は比較的少ない時間，メモリで表明の生成が可能であるが，得られる実行データが少ない場合，自動検査には不向きである．そのため，動的手法は表明の自動生成に用いられることが多く，その代表的ツールに Daikon [10,11] がある．しかし，表明の動的生成手法の問題点として，生成表明が実行データを取得する際に用いるテストケースに依存するテストケース依存問題がある [12].

我々の研究グループは，表明の動的生成手法に着目し，そのテストケース依存問題を改善することを目的として，インバリアントカバレッジ [13]，モデル検査器 Java PathFinder [8,9]，記号実行 [14,15] の技術を活かしテストケースを生成する手法を提案してきている [16]．インバリアントカバレッジ [13] とは，動的生成ツールに用いるテストケースの有用性を測定するために提案されたカバレッジである．この値が高いとき，動的生成ツールは信頼性の高い表明を生成できる．文献 [13] で提案されているインバリアントカバレッジに基づくテストケースの生成を自動化する手法というのが，我々が提案した手法である [16].

*Keizo Miyamoto, 大阪大学大学院情報科学研究科

†Naoya Hori, 大阪大学大学院情報科学研究科，現在野村総合研究所

‡Kozo Okano, 大阪大学大学院情報科学研究科

§Shinji Kusumoto, 大阪大学大学院情報科学研究科

¶Satoru Nishimoto, 大阪大学大学院情報科学研究科

本稿では文献 [16] で提案した手法を実装したツールを作成し，複数の例題に対し適用実験を行った結果について報告する．適用実験では例題としてサイクロマティック複雑度 [17] の異なる複数の Java プログラムを用い，プログラムの複雑さと生成表明の精度の関係性を評価した．また，既存のカバレッジに基づくテストケースによる生成表明との比較も行った．比較対象のカバレッジとして，ステートメントカバレッジ，ブランチカバレッジ，マルチコンディションカバレッジを用いた．その結果，本手法により生成されたテストケースを用いた場合に表明の精度が向上したことを確認した．また，対象プログラムのサイクロマティック複雑度が高いほど本手法による表明の精度向上が大きかった．

以降，2 章では本研究に関連するツールや概念について述べ，3 章で提案手法について述べ，4 章で例題への適用実験について述べ，5 章で考察を行い，6 章でまとめる．

2 準備

この章では準備として本稿に関する諸概念について簡単に述べる．

Daikon [10,11] は入力であるソースコードとテストケースから，実行時にメソッドの入口，出口 (以降，プログラムポイントとする) にて参照可能な変数の値を観測する準備を行い，テストケースを用いてソースコードを実行する．そして，その実行から得られた変数の値の観測結果と Daikon が持つ表明パターンとを照合して各プログラムポイントにおける表明を自動生成し，出力する．また，このとき独自の表明推測アルゴリズムを適用し再現率，適合率を向上させている [11].

動的生成手法では，生成表明の精度が入力するテストケースの品質に依存する．これをテストケース依存問題 [12] と呼ぶ．動的生成手法で用いるテストケースは対象メソッドの引数生成部，テストケース制約判定部，対象メソッド呼び出し部の 3 つからなる手続きであるが，このテストケース制約が不十分な場合，得られる実行データが不十分になる．このような場合，生成表明の精度が低下する．

インバリアントカバレッジ (Invariant Coverage) [13] は，表明の動的生成ツールに用いるテストケースの品質測定を目的として提案されたカバレッジである．インバリアントカバレッジに基づいてテストケースを生成することで，テストケース依存問題を改善できる．テストケース依存問題を改善するためには，表明の動的生成に用いるテストケースは，対象ソースコードの全ての必要な実行パスを実行する必要がある．

インバリアントカバレッジ [13] は一般的な表明の生成に必要な全実行パスに対して，テストケースが実際に実行する実行パスの割合を示す．文献 [13] では，このような実行パスをプログラムポイントにて参照可能な変数の定義–使用連鎖を含む実行パスに近似している．

定義 1 定義–使用連鎖 *(Definition-Use Chain，以降 DUC)*

プログラムポイント S にて参照可能な変数 v の *DUC* は，次のように定義できる．

*Definiton-Use Pair(*以降，*DUP* とする*)* を変数 v，定義部 d，使用部 u の *3* 項組で定義し，$v(d,u)$ で表記する．ここで，d，u はプログラム記述中の位置を表す．*DUC* はこの *DUP* の *(*有限*)* 系列として定義できる．直前の d が次の *DUP* の u である．

この系列において，位置 d，u の複数回の出現は許すが，同一 *DUP* の複数回の出現は許さない．ただし，系列の最後において d，u が同一の位置のとき，この *DUP* の繰り返しを許す．

定義 2 インバリアントカバレッジ

全プログラムポイントにて参照可能な全変数の *DUC* の総数を DUC_{all}，テストケースによって実行された *DUC* の数を $DUC_{executed}$ とする．このとき，あるテストケースにおけるインバリアントカバレッジ C_{Inv} の値は式 *(1)* で定義される．

$$C_{Inv} = DUC_{executed}/DUC_{all} \quad (1)$$

Java PathFinder(以降，JPF とする) [8, 9] は Java ソースコードを対象としたモデル検査器である．JPF は対象ソースコードとそのソースコードが満たすべき条件を記述したプロパティファイルを入力とし，対象ソースコードがプロパティファイルに記述された条件を満たすかどうか判定し，満たさない場合に反例としてその実行パスを出力する．ここでメソッドの実行パスとは，メソッドの引数の取得から条件に違反するまでに実行するソースコード中の命令文の系列である．

記号実行 [14, 15] は，ソースコード中の各実行パスを実行可能な入力条件を記号操作的に論理式として求める手法である．記号実行では実行パスの入力条件を求める際に，メソッドの引数として X や Y などといった記号を代入することで，各実行パスを通る一般的な条件を求めることができる．

3 提案手法

3.1 提案手法の手順

提案手法では以下の手順でインバリアントカバレッジに基づいたテストケースを自動生成する．

(Step 1) 対象ソースコードから DUC 生成用プログラム依存グラフ（以降，DUC 生成用 PDG とする）を生成し，プログラムポイントにて参照可能な変数を取得する．(PDG Generator)；

(Step 2) DUC 生成用 PDG から対象メソッド内の全 DUC（定義部，使用部の位置はソースコード中の命令文の位置）を生成する．(DUC Generator)；

(Step 3) JPF を用いて (Step 2) で取得した全 DUC の実行パスを取得する．(Java PathFinder)；

(Step 4) 取得した全実行パスを記号実行し，テストケース制約を算出する．(Symbolic Executor)；

(Step 5) テストケース制約を満たすテストケースを生成する．(Test suites Generator)；

(Step 6) 生成したテストケースを用いて Daikon を実行する．

3.2 DUC の生成

まず，入力として与えられたソースコードから DUC 生成用 PDG を生成する．PDG は命令文に対応するノードと，制御依存辺，データ依存辺からなるグラフである．

本ツールではコストの問題から，対象ソースコード全体に対してではなく一部のみに対して PDG を生成している．具体的には，対象メソッド，対象メソッドが呼び出す自クラスのメソッドを対象に PDG を生成する．ここでは，この PDG を DUC 生成用 PDG と呼ぶ．DUC 生成用 PDG は，関数内 PDG(Interprocudual PDG) を拡張したものであり，拡張した点は以下のとおりである．

- 配列の要素は全て親変数とする
- オブジェクトの操作は全てそのオブジェクトの定義・参照とする
- return 文はフィールド変数，引数とデータ依存関係をもつ

最終的に DUC は，DUC 生成用 PDG の出口ノードからデータ依存辺を辿ることで得られる．

3.3 実行パスの検出

JPF を用いて実行パスを求める方法には以下の 3 種類がある．

1. 対象プログラムのソースコード中にアサート文を挿入する．
2. 対象プログラムが満たす条件をプロパティファイルに記述し JPF に入力する．
3. JPF のバーチャルマシンのイベントを検出するリスナを設定する．

2，3 の方法を用いることでデッドロックの検出など，より複雑な条件を検証することができる．しかし，本手法における JPF の使用目的は，DUC を通る実行パスの検出であるので，実装が容易である 1 のアサート文を挿入する方法を用いている．詳細は文献 [16] 参照.

3.4 記号実行

本研究では文献 [16] で用いられている記号実行を行っている．記号実行は JPF が生成した対象メソッドの DUC を通る実行パスと，対象メソッドのソースコードを入力とし，DUC を通る実行パスを実行するために必要なメソッドの引数の条件 (テストケース制約) を出力する．

本研究で行っている記号実行が適用可能なクラスには以下の制限がある．

1. 対象メソッドの仮引数の型はプリミティブ型，もしくは String 型である．
2. 対象メソッド中で呼び出されるメソッドに native メソッドが含まれていない．
3. 対象メソッドの条件式にフィールド変数が含まれていない．
4. 対象メソッド内に switch 文が存在しない．

3.5 テストケース生成

実際に対象メソッドを実行するためのテストケースについて述べる．テストケースは大きく以下の 3 つの部分により構成される．

1. 対象メソッドを持つクラスのオブジェクト生成部
   対象メソッドを持つクラスのインスタンスを生成する．
2. 対象メソッドへ入力する引数生成部
   記号実行により求められたテストケース制約を満たす引数を生成する．
3. 対象メソッド呼び出し部
   生成された引数を用いて対象メソッドを呼び出す.

4 実験

本実験では提案手法により生成されたテストケースと既存のカバレッジに基づいて生成されたテストケースを用いて Daikon による表明の生成を行う．本実験の目的は，各テストケースから生成された表明の精度の比較である．また，プログラムの複雑度により生成表明の精度に違いが生じるかを調べる．プログラムの複雑さの指標にはサイクロマティック複雑度 (以降，CC とする) [17] を用いた.

4.1 評価の基準

本実験では，生成表明の妥当性, 生成表明の妥当性とプログラムの複雑度の関連性，本ツールの効率性の三項目について評価を行った.

妥当性：入力された対象プログラムとテストケースより Daikon が生成した表明の適合率と再現率，F 値を求め，それぞれの値より生成表明の妥当性を評価する．

Daikon により生成された表明の集合を A，対象のソースコードから人が必要だと判断した表明の集合を B として，適合率 ($P = (|A \cap B|)/|A|$)，再現率 ($R = (|A \cap B|)/|B|$)，F 値 ($F = 2 \times P \times R/(P + R)$) を定義する．

また，提案手法の効率性を評価するために以下の評価基準を用いる．

- ツールの実行時間：本ツールがテストケース生成にかかる時間
- Daikon の実行時間：入力されたテストケースを用い Daikon がアサーション生成にかかる時間

4.2 実験環境

評価実験は，OS が Windows Vista (64bit)，CPU が Intel Xeon 2.00GHz 2.00GHz，Memory が 8.00GB の計算機で行った.

表 1　対象プログラムに対する生成表明の比較

		calc			compareHashes			BMmatch			equals		
M	表明	P	R	F	P	R	F	P	R	F	P	R	F
M0	pre	0.50	1.0	0.67	1.0	1.0	1.0	1.0	0.30	0.50	1.0	1.0	1.0
	post	0.14	0.17	0.18	0.23	0.60	0.33	0.75	0.75	0.75	0.80	0.57	0.67
M1	pre	0.50	1.0	0.67	1.0	1.0	1.0	1.0	0.30	0.50	1.0	1.0	1.0
	post	0.14	0.17	0.18	0.20	0.40	0.27	0.67	0.50	0.57	0.60	0.43	0.50
M2	pre	0.50	1.0	0.67	1.0	1.0	1.0	1.0	0.30	0.50	1.0	1.0	1.0
	post	0.14	0.17	0.18	0.20	0.40	0.27	0.67	0.50	0.57	0.60	0.43	0.50
M3	pre	0.50	1.0	0.67	1.0	1.0	1.0	1.0	0.30	0.50	1.0	1.0	1.0
	post	0.14	0.17	0.18	0.22	0.60	0.32	0.67	0.50	0.57	0.75	0.43	0.55

表 2　サイクロマティック複雑度と生成表明の F 値の関連性

		事前条件の F 値				事後条件の F 値			
対象メソッド	CC	M0	M1	M2	M3	M0	M1	M2	M3
ResultQE#calc	2	0.67	0.67	0.67	0.67	0.18	0.18	0.18	0.18
Hash#compareHashes	3	0.33	0.27	0.27	0.32	0.33	0.27	0.27	0.32
BoyerMoore#BMmatch	5	0.75	0.57	0.57	0.57	0.75	0.57	0.57	0.57
Arrays#equals	7	0.67	0.50	0.50	0.55	0.67	0.50	0.50	0.55

表 3　Daikon の実行時間の比較

対象メソッド	M	実行時間	対象メソッド	M	実行時間
ResultQE#calc	M0	4.11 秒	BoyerMoore#BMmatch	M0	16.39 秒
	M1	4.50 秒		M1	3.84 秒
	M2	4.50 秒		M2	3.84 秒
	M3	4.52 秒		M3	4.10 秒
Hash#compareHashes	M0	4.50 秒	Arrays#equals	M0	5.42 秒
	M1	5.04 秒		M1	6.06 秒
	M2	5.01 秒		M2	6.17 秒
	M3	5.04 秒		M3	6.43 秒

4.3　対象プログラム

本実験では以下の Java プログラムを用いた.

二次方程式の解を計算する ResultQE#calc (CC:2)，2 つの byte 型の配列変数に対してハッシュ値の比較を行う Hash#compareHashes (CC:3)，Boyer-Moore 文字列検索アルゴリズムを用いて文字列検索を行う BoyerMoore#BMmatch (CC:5)，2 つの int 型の配列変数のを比較を行う Arrays#equals (CC:7).

4.4　結果

4.3 節の各対象プログラムに対し提案手法を用いて生成された表明と，他のカバレッジを用いて生成された表明の適合率 (P)，再現率 (R)，F 値 (F) の結果を示す. 対象プログラムに対して生成された事前条件と事後条件における各値の比較結果を表 1 に示す．なお，表 1, 2, 3 では，テストケース生成手法を M とし，M0, M1, M2, M3 はそれぞれ提案手法，ステートメントカバレッジ，ブランチカバレッジ，マルチコンディションカバレッジに基づくテストケース生成手法を表す．また，pre, post はそれぞれ事前条件，事後条件を表す．次に，対象プログラムの CC と，各テストケースから生成された事前条件，事後条件から求めた F 値の比較結果を表 2 に示す. そして，各テストケースごとの Daikon の実行時間の比較結果を表 3 に示す.

最後に，各対象プログラムにおける本ツールの実行時間を，表 4 に示す．モジュール名の DUC，JPF，記号実行，TC は，それぞれ DUC 生成部，JPF 実行部，記号

表 4　本ツールの実行時間

	モジュール名				
対象メソッド	DUC	JPF	記号実行	TC	全実行時間
ResultQE#calc	6 秒	5 秒	1 秒	1 秒	47 秒
Hash#compareHashes	5 秒	29 秒	3 秒	1 秒	74 秒
BoyerMoore#BMmatch	83 秒	969 秒	197 秒	16 秒	1558 秒
Arrays#equals	4 秒	43 秒	2 秒	1 秒	76 秒

実行部，テストケース生成部を表す.

5　考察

4.4 節に示した評価実験の結果について考察を行う.

5.1　妥当性

対象プログラムに対して, 提案手法を用いて生成したテストケースによる表明と他のカバレッジに基づいたテストケースによる表明を比較することで, 生成表明の妥当性の評価，考察を行う.

表 1 を見ると, 事前条件は各テストケースで生成表明の適合率，再現率が同じであるが，事後条件は提案手法を用いた場合に適合率，再現率が向上していることがわかる．また，既存のカバレッジに基づくテストケースの中では，マルチコンディションカバレッジに基づくテストケースの適合率，再現率，F 値が他の 2 つのカバレッジに比べてよいことも分かる.

動的生成により精度の高い事後条件を生成するにはメソッド内部の実行パスを十分な回数実行する必要がある．ステートメントカバレッジは全ステートメント中一度でも実行されたステートメントの割合によりカバレッジの値が決まる，このため，カバレッジの値が高くても必要な実行パスが十分に実行されない場合がある．このときには生成表明の精度は低くなる．ブランチカバレッジは全条件分岐中一度でも実行された分岐の割合によりカバレッジが決まる．よって，ステートメントカバレッジと同様の理由により生成表明の精度が低くなると考えられる．マルチコンディションカバレッジは各条件分岐の組み合わせによりカバレッジの値が決まるために，このカバレッジの値が高い場合は実行されるパスの割合はステートメントカバレッジ，ブランチカバレッジに比べて高くなる．このため，生成表明の精度はステートメントカバレッジ，ブランチカバレッジに比べて高くなると考えられる.

事前条件の適合率，再現率，F 値は各テストケースによる違いが見られない．これは，Daikon は実行中にメソッドへ入力される値の観測結果から事前条件を生成するので，メソッド内部の条件分岐などがどの程度実行されているかは生成される事前条件の再現率などに影響を与えないためと考えられる.

5.2　妥当性とプログラムの複雑度の関連性

生成表明の妥当性と対象プログラムの複雑度の関連性について考察を行う.

表 2 を見ると，事前条件の妥当性とプログラムの複雑度には関連性がないと言える．これは，Daikon が事前条件を生成する際，対象プログラムの内部構造は考慮せず，入力された引数の値から生成するため，各テストケースごとの差が生じにくいためと考えられる.

次に事後条件の結果について考察を行う．CC が 2 である ResultQE#calc では，各テストケースにより生成される表明の妥当性を表す F 値に変化が見られない．次に，CC が 3 である Hash#compareHashes は，提案手法を用いた場合の方が妥当性は上がっているがその F 値の差は小さく，また全体的に F 値が低い．F 値の差が小さい理由としては，プログラムが単純なために各テストケースによって実行されるパスに差が出にくいためと考えられる．また，CC が低いプログラムの場合，テスト

ケースによるプログラム実行回数が減少する．その結果，適合率，再現率が低下し，全体的に F 値が低くなると考えられる．CC が高い BoyerMoore#BMmatch(CC:5) や Arrays#equals(CC:7) では，提案手法を用いた場合の F 値が他のテストケースに比べて F 値の向上が確認でき，また全体的に F 値が高い．提案手法の F 値が高くなるのは，他のテストケースでは実行されない DUC のパスを提案手法のテストケースでは実行するためと考えられる．全体的に F 値が高い理由は，プログラムが複雑になると実行されうるパス数が増加しテストケースによるプログラム実行回数が増加する．これにより多くの実行結果から表明が生成され精度が高くなると思われる．

以上より，対象プログラムの CC が高いほど本手法による生成表明の妥当性が上がり，提案手法の有用性が向上すると言える．

5.3 効率性

Daikon の実行時間と，本ツールの実行時間について考察を行う．

まず，Daikon の実行時間について述べる．表 3 の BoyerMoore#BMmatch に対する結果を見ると，提案手法を用いた場合に Daikon の実行時間が長くなっている．Daikon の実行時間は対象プログラムの実行回数に比例する．この対象プログラムはテストケースが実行すべき実行パスが多いために，Daikon の実行時間が長くなったと考えられる．また，テストケース制約の簡略化を行っていないため，テストケース制約が複雑になり，条件を満たす引数生成に時間がかかったというのも理由として考えられる．表 3 のその他のプログラムに対する結果を見てみると，総じて提案手法を用いた場合の時間が短い．この理由として，対象プログラム全体のパスから，必要なインバリアントカバレッジを満たすパスを見つけて，他のカバレッジのテストケースに比べて効率のいいテストケースを生成できたためと考えられる．

次に，本ツールの実行時間について述べる．表 4 を見ると，BoyerMoore#BMmatch に対する実行時間が他に比べて長いことがわかる．これは，プログラム中のループ内部の構造が複雑なため必要な DUC の数が多くなった，プログラムの引数が文字列であるため取りうる組合せが多い，テストケース制約が複雑ということが考えられる．表 4 を見ると，全体的に JPF の実行時間が大部分であることが確認できる．例外として，ResultQE#calc は JPF 実行時間の割合が小さい．これは ResultQE#calc の複雑度が低いため考慮すべき実行パスの数が少なかったためと考えられる．

5.4 今後の課題

現在，本手法が適用可能なクラスの制限と実行時間の長さが課題である．この課題の解決案として，静的解析器 ESC/Java2 [7] を用いて DUC を通る実行パスの取得，テストケース制約算出を行う方法を検討している．ESC/Java2 がアサーションが満たされない場合に出力する反例を利用する．

JPF は対象プログラムのバイトコードを独自の JVM 上で実行し解析を行う．このため DUC を含む実行パス取得には対象プログラムが入力値として取りうる値を網羅的に入力し，繰り返し実行を行う必要がある．このことが本手法において JPF 実行時間が長くなる原因となっている．ESC/Java2 は対象プログラムのソースコードより論理式を生成し静的検証を行う．このため，対象プログラムの繰り返し実行が不要になり JPF に比べ実行時間の短縮が可能であると考えている．また，ESC/Java2 が出力する反例にはプログラム中の変数やメソッドの返り値の値に関する条件が含まれている．この反例を利用することでより広いクラスに対する記号実行が実現できると考えている．

6 あとがき

本研究では，我々の研究グループによって提案された表明動的生成のためのテストケース自動生成ツールの実装とその評価実験を行った．提案手法は，表明の動的

生成法を基に，インバリアントカバレッジや記号実行，モデル検査技術を用いることでテストケース依存問題を改善している．また実装したツールの評価実験を行った結果，ツールが生成したテストケースから得られた表明は，既存のカバレッジに基づくテストケースから得られた表明よりも妥当性が高いことを確認した．また対象プログラムの複雑度が高いほど，妥当性の向上が大きいことを確認した．しかし，本ツールが適用可能なクラスの制限，実行時間の長さが課題である．今後はツールの適用可能クラスの拡張，実行時間の改善とより多くの評価実験を行い，より詳細に本手法の有用性を評価したい．

謝辞 本研究の一部は科学研究費補助金基盤 C(21500036) と文部科学省「次世代 IT 基盤構築のための研究開発」(研究開発領域名：ソフトウェア構築状況の可視化技術の普及) の助成による．

参考文献

[1] B. Meyer: “Applying Design by Contract,” IEEE Computer, vol.25, no.10, pp.40-51, 1992.
[2] J. W. Nimmer and M. D. Ernst: “Invariant Inference for Static Checking: An Empirical Evaulation,” in Proc. of SIGSOFT Symp. on Foundations of Software Engineering 2002, FSE 2002, pp.11-20, 2002.
[3] P. Cousot and R. Cousot: “Modular Static Program Analysis, ” in Proc. of Int. Conf. on Compiler Construction, LNCS, vol.2304, pp.159-178, 2002.
[4] C. Flanagan and K. R. Leino: “Houdini, an Annotation Assistant for ESC/Java,” in Proc. of Int. Symp. of Formal Methods Europe on Formal Methods for Increasing Software Productivity, FME 2001, pp.500-517, 2001.
[5] N. Tillmann, F. Chen and W. Schulte: “Discovering likely method specifications,” Int. Conf. on Formal Engineering Methods, ICFEM 2006, LNCS, vol.4260, pp.717-736, 2006.
[6] P. H. Schmitt and B. Weiss: “Invariants by Symbolic Execution,” in Proc. of the 4th International Verification Workshop, VERIFY 2007, pp.195-210, 2007.
[7] D. L. Detlefs, K. Rustan M. Leino, G. Nelson and J. B. Saxe: “Extended static Checking,” SRC Research Report 159, Compaq SRC, 1998.
[8] W. Visser, K. Havelund,k G. Brat, S. Park and F. Lerda: “Model Checking Programs,” Automated Software Engineering Journal 2003, vol.10, no.2, pp.203-232, 2003.
[9] F. Lerda and W. Visser: “Addressing Dynamic Issues of Program Model Checking,” in Proc. of Int. Workshop on SPIN Model Checking 2001, pp.88-102, 2001.
[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz and C. Xiao: “The Daikon System for Dynamic Detection of Likely Invariants,” Science of Computer Programming, vol.69, no.1-3, pp.35-45, 2007.
[11] J. W. Nimmer and M. D. Ernst: “Automatic Generation of Program Specification,” in Proc. of SIGSOFT Int. Symp. on Software Testing and Analysis 2002, pp.232-242, 2002.
[12] J. W. Nimmer and M. D. Ernst: “Static Verification of Dynamically Detected Program Invariants: Integrating Daikon and ESC/Java,” in Proc. of First Workshop on Runtime Verification, RV 2001, pp.152-171, 2001.
[13] N. Gupta and Z. V. Heidepriem: “A New Structural Coverage Criterion for Dynamic Detection of Program Invariants,” in Proc. of Int. Conf. on Automated Software Engineering, ASE 2003, pp.49-58, 2003.
[14] S. Khurshid, C. S. Pasareanu and W. Visser: “Generalized Symbolic Execution for Model Checking and Testing,” in Proc. of Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003, pp.553-568, 2003.
[15] J. C. King: “Symbolic Execution and Program Testing,” Communications of the ACM 1976, vol.19, no.7, pp.385-394, 1976.
[16] 堀 直哉, 岡野 浩三, 楠本 真二: “モデル検査技術を用いたインバリアント被覆テストケースの自動生成による Daikon 出力の改善”, ソフトウェア工学の基礎ワークショップ FOSE2008, ソフトウェア工学の基礎 XV, pp.41-50, 2008.
[17] T. J. McCabe: “A Complexity Measure,” IEEE Transactions on Software Engineering, vol.2, no.4, pp.308-320, 1976.