B-04

Daikonを利用した表明動的生成改善手法の実プロジェクト教材への適用実験とテストデータ生成改善手法の検討

Application of Automatic Assertion Generation using Daikon to a Real Example and a study of test data generation

小林　和貴 †　　岡野　浩三 †　　楠本　真二 †
Kazuki Kobayashi　Kozo Okano　Shinji Kusumoto

1．　はじめに

Design by Contract に基づくアサーション記述（以下，表明という）は，ソースコードの仕様理解の補助やプログラム検証に役立つ．表明の自動生成手法の一つである表明の静的生成手法は，プログラムを静的に解析することで表明を生成する．Houdini [1] は静的解析器を繰り返し適用することにより，表明を生成する．この方法では，表明生成にかかる時間が長いことが課題である．一方，表明の動的生成手法を実装したツールの一つに Daikon [2] がある．対象プログラムに対し，テストケースを与え実行しメソッドの返り値を監視することで，表明を生成する．同様のツールとして DySy [3] がある．C ＃言語の対象プログラムを実行し，内部変数の更新を監視することで，表明を生成する．動的生成手法はテストケースを用い，対象プログラムを直接実行するため，静的手法と比較して短時間で表明が生成できる [4].

しかし，動的生成手法には，実行データを取得する際に用いるテストケースに生成される表明が依存するという，テストケース依存問題 [5] がある．そのため，インバリアントカバレッジ [6] が提案されている．このカバレッジの値が高いとき，動的生成手法は信頼性の高い表明を生成できる．著者が所属する研究グループでは，モデル検査技術を利用しインバリアントカバレッジの値が高いテストケースを自動生成する手法の提案を行ってきた [7–9].

本稿では，提案手法を実プロジェクト教材として利用されている大学教務システムに対して適用実験を行った結果について報告する．提案手法を実際のシステムに対する適用可能性や生成される表明の精度・有用性について評価した．また，適用対象のメソッドについて，テストデータとして要求されるクラスについて調査した．

その結果，全体の 4 パーセントのメソッドに対して手法を適用することができ，全てにおいて有用な表明を生成することができた．これらの表明を，対象システムの単体テスト時に利用されるテストケースを用いて生成した表明と比較した．その結果，人手によるテストケースを利用した場合に生成される，テストデータを表明記述に含むようなテストデータに依存した表明は，提案手法において生成されないことがわかった．さらに，対象システムに対して，別プロジェクトにおいて人手によって記述した表明と比較したところ，不要な表明も出力されているものの，ほぼ同様の表明を出力できていることがわかった．そして，今回表明生成改善手法の適用できなかったメソッドについて，テストデータを生成する手法について検討を行った．

† 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

2．　準備

2.1　表明

ソースコード中に表明と呼ばれる記述を行うことにより，Designed by Contract における契約を記述できる．この表明により，ESC/Java2 [10] などを用いた静的解析によりプログラムの妥当性を検証でき，開発者の意図しない不具合の混入を防ぐことができる．

2.2　表明の自動生成手法

近年のソースコードサイズの増加に伴い，手動による表明生成は困難になりつつある．そこで，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，静的手法と動的手法の 2 種類がある [11].

静的手法 [1,10] はソースコードの状態を表すモデルを生成し，実行し得る全ての状態とそのときの実行条件を求めることで，表明を生成する．そのため，精度の高い表明の自動生成 [1] や自動検査 [10] が可能である．しかし，一般的に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．

動的手法は対象ソースコードとテストケースを入力とし，テストケースを用いて対象ソースコードを実行し，得られたデータから表明を生成する．

テストケースの品質が低い場合，生成される表明の精

度が低下する問題が指摘されている [5] が，一般的に比較的少ない時間とメモリで表明の生成が可能である．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，その代表的ツールとして Daikon [2] がある．このツールを用いることで，手作業で表明を記述するより表明生成に必要な時間的，人的コストを軽減できる．また，実際にプログラムを実行した結果を用いるため，プログラマがソースコード記述時には気づかなかった表明を生成することもできる．これはプログラムの保守，デバッグにも有効である．

2.3 テストケース依存問題

動的生成手法により表明を自動生成する際，生成される表明の精度は入力であ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という．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とその引数が満たすべき条件，対象メソッド呼び出しの 3 つからなる手続きであるが，この引数の条件が十分でない場合，得られる実行データが少なくなる．このとき，限定的あるいは誤った表明が生成されてしまう場合がある．

2.4 インバリアントカバレッジ

テストケース依存問題を解決するため，インバリアントカバレッジが提案されている．インバリアントカバレッジは，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．この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．われわれの研究グループではインバリアントカバレッジに基づくテストケースを利用することにより，動的生成による表明の精度が向上することを文献 [8] で確認している．

3． テストケース自動生成問題

表明動的生成において，入力となるテストケースをどのように生成するかという問題がある．Daikon では人手により作成した単体テストを利用できるが，人手によるテストケースの作成はコストがかかることや，テスト漏れが生じるなどの課題がある．われわれの研究グループでは，インバリアントカバレッジの高いテストケースを自動生成する手法について研究してきた．特にモデル検査器や定理証明器を利用し対象プログラムを解析することで，インバリアントカバレッジの高いテストケースの条件を導出できる手法について提案してきた．

3.1 定理証明器を利用した手法

われわれの研究グループでは，既存手法 [7] を改善する手法として定理証明器を利用した手法について提案した [12]．改善手法は，ESC/Java2 の反例出力を利用したテストケース制約式の導出手法である．ESC/Java2 に『インバリアントカバレッジに影響を与えるパスを通らない』という仕様を与えることにより，インバリアントカバレッジに影響を与えるパスの情報を反例として取得することができる．

これは，ESC/Java2 が，Java1.4 以前の Java で記述されたプログラムと仕様記述言語 JML [13] で記述された仕様をそれぞれ述語論理に変換し，内部の定理証明器を用いて充足不能性を判定することで，プログラム仕様と実装の一致性の検証を行うツールであることを利用している．一般的な Java ライブラリに対する仕様を，あらかじめ準備しているため，より一般的な対象に対して検証を行える．

改善手法は，既存手法に比べて短時間でテストケース制約式を導出することができ，対象プログラムについても，既存のクラスでは対応できなかった参照型変数を含むメソッドに対しても適用できる．

3.1.1 インバリアントカバレッジに基づく実行パスの導出

表明動的生成に適したテストケースを生成するため，インバリアントカバレッジを満たす実行パスを導出する必要がある．対象メソッドについて，返り値の値が変化しうる実行パスをループを除きすべて求める．そのために，プログラム依存グラフを用いて変数の定義-使用関係を探索する．本実験の実装ではプログラム依存グラフの構築に MASU [14] を用いた．

3.1.2 ESC/Java2 の反例による対象プログラムの解析

ESC/Java2 は，入力したプログラムと仕様の間に不一致の可能性がある場合，反例を出力する．この反例出力には，変数間の関係や，型に関する情報が含まれている．改善手法では，この情報を構文解析器によって抽出し，テストケース制約式を導出する．

プログラム依存グラフから実行パスを取得した後，解析対象のプログラムに対し，求めた実行パスを通らないという仕様を与え，ESC/Java2 で検証を行う．検証により，実行パスを通る条件を反例として取得できる．

3.1.3 テストケース生成と表明の生成

得られたテストケース制約式を用いてテストケースを生成する．テストケースは，インバリアントカバレッジに影響を与える各実行パスごとに生成する．実行パスを通るために必要な入力データであるテストデータを，予め得られた型の情報から，ランダムなデータとして動的に生成し，テストケース制約式によって選択する．各実行パスに対し複数回テストデータを入力し実行することにより，実行パスにおける変数の情報を Daikon に取得させることができる．Daikon はこのテストケースを実行することで，対象プログラムの表明を生成する．

4． 実プロジェクト教材への適用実験

改善手法が実プロジェクトのプログラムにおいて適用可能な範囲や，人手と比べて表明の正確さに変化が見られるかなどを確認するため，実プロジェクト教材を対象に適用実験を行った．

4.1 適用対象

本稿における実プロジェクト教材は，ITSpiral で作成された実プロジェクトの教材データを用いた．大学教務システム開発プロジェクトによって作成されたシステムで，規模や JUnit テストスイートの数を表 1 に示す．

対象システムは実用的なサイズの Java プログラムであり，一般的な業務システムであるため，適用対象として十分なサイズである．また，文献 [15] において，人手で理想的な JML が記述されていることから，本実験で生成された JML と比較することにより，表明の精度を評価できる．

なお，対象システムは Java1.5 で記述されている．改善手法の現在の実装では，対象プログラムのテストケース制約解析に ESC/Java2 を利用しているため，表明を出力させたいメソッドやクラスは Java1.4 以前の文法で記述されている必要がある．本稿の実験においては，総称型による表現やイテレータによる配列や集合への操作を，実際の動作内容に変更がない範囲内で対象を Java1.4 に対応する文法にて一部書きなおしている．今回の対象特有ではあるが，大幅な変更が要求される列挙型に関しては，今回は改善手法の適用対象から除外した．

4.2 実験方法

人手により記述されたテストケースを用いて生成された表明と，改善手法で生成される表明および人手により記述した理想的な表明との間で表明の数や精度にどのような差があるかや，改善手法が実システムのプログラムに対してどの程度適用可能かを調べるために，以下の実験を行った．

4.2.1 適用可能なメソッド数の調査

対象システムに存在するメソッドのうち，改善手法が現在の実装において適用可能なメソッド数を調査した．今回の実験対象は Java1.5 で記述されており，そのままでは一部メソッドに対し改良手法を適用できないが，書き換え可能な範囲において Java1.4 に書き換えを行い，適用可能かどうかを調査した．この場合，適用できるかどうか判定する基準としては以下の基準が挙げられる．

1. 対象メソッドの引数が基本型およびその配列または java.lang.String 型であること
2. 対象メソッドを持つクラスが new キーワードによってインスタンス化可能であること
3. 対象メソッドを持つクラスのコンストラクタが private な場合は，getInstance など一般的なシングルトンパターンのメソッドを通じてインスタンスへの参照を得られること

これらの条件を満たすメソッドの数を対象システム内で計測した．

4.2.2 人手によるテストケースを利用した表明の取得

対象システムは開発時に JUnit によるテストを作成している．このテストスイートの JUnit 実行を Daikon で監視させ，Daikon から対象システム内のメソッドの事前条件・事後条件を表明として出力させた．

4.2.3 改善手法によるテストケースを利用した表明の取得

本実験では，人手によるテストケースと改善手法によるテストケースで生成される表明の数や精度の変化を調べるため，人手によるテストケースで表明が出力できた 376 メソッドのうち，データベースを直接操作したり単体テストで用いるためのデータベース復帰用のメソッドなど 71 メソッドを除き，改善手法におけるテストデータ生成部が対応しているデータ型を引数に取る 36 メソッドを対象に，改善手法により表明生成を試行した．

4.2.4 人手により記述した表明と動的生成した表明の比較

人手によるテストケースおよび改善手法によるテストケースを利用した表明と，人手で記述した理想的な表明を比較し調査した．調査では，対象メソッドの動作を表

表 1: 対象システムと人手で記述されたテストケースの規模

パッケージ数	15	（パッケージ）
クラス数	181	（クラス）
メソッド数	955	（メソッド）
コード行数	14521	（行）
テストメソッド数	467	（メソッド）

すのに必要な表明が生成されているかや，他の表明に包含され不要な表明が生成されているかを調査した．

5． 適用実験の評価

適用実験の結果について示し，考察を行う．改善手法の適用可能範囲について議論を行い，改善手法により生成された表明について人手によるテストケースで生成された表明および人手で記述した理想的な表明と，表明の生成数を計測した結果を示し，生成された表明について，有用性について考察を行う．

5.1 改善手法の適用可能範囲

実験で表明を生成できたメソッド数について，人手で記述したテストケース（単体テスト）と改善手法が対応しているメソッドとに分けて，表 2 に示す．本稿の実験では，改善手法は 4.2.1 において定義したメソッドに対して適用が可能である．対象システムが Java1.5 で記述されていたことに起因する適用不能なクラスが存在したものの，多くは対象メソッドの引数のデータ型に対応していなかったり，対象メソッドが引数を取らないメソッド（getter など）や，データベースの値に依存して返り値を返すメソッドであるためである．これらのメソッドは改善手法では表明を生成できないものとした．

メソッド引数にシステム固有のクラスのデータ型を用いている場合は，対応するデータを表すオブジェクトを生成するクラスを追加することで対応が可能である．モックオブジェクトを生成するライブラリ利用し，データを生成する [16] ことも可能であると考えられる．

引数を取らないメソッドに関しては，現在の実装においてはテスト対象のオブジェクトに対しフィールド値を設定していないため，実験を行っていない．対象メソッドに対する表明生成では，改善手法と人手によるテストケースとの間に生成される表明の精度に大きな差はないか，または人手によるテストケースで生成される表明が精度が良いことが考えられる．人手によるテストケースにおいて,『setter メソッドにて値を設定した後，getter メソッドなどを呼び出す』といったシナリオを複数回行っている場合は人手によるテストケースで精度の良い表明が生成されることが考えられる．しかし，テストデータの数が少ない場合は表明中に setter メソッドで設定した固有の値が現れるなど，表明の精度が低下するおそれもある．

データベースの値に依存して返り値を変えるメソッドは，スタブを利用することで対応可能であると考えられる．データベースと入出力を行っているクラスをスタブで置き換え，データを制御することにより，表明を生成できる可能性がある．

5.2 改善手法による表明の評価

実験において，改善手法により生成された表明の数と人手によるテストケースで生成された表明の数を比較し，テストケースの違いによって表明の生成数に差が現れるのかを調べた．また，人手で記述した表明と改善手法により生成された表明との比較を行い，改善手法により生成された表明の正確さについて評価を行った．

5.2.1 人手によるテストケースで生成した表明との比較

人手で記述した単体テストと改善手法によるテストケースで表明を生成した．表明の生成数は事前条件と事後条件を対象に表 3 に示す．表中『単体テスト（一部）』と表示した行は，改善手法を適用可能なメソッドに対して単体テストを適用した結果である．

人手で生成したテストケースを利用した場合，適用対象メソッド数と表明の生成数の割合に大きな変化がないことがわかる．個々の適用対象のメソッドによって生成される表明の数に差があったものの，改善手法が適用可能なメソッドにおいて特に表明の生成されやすさに変化は見られなかった．

一方，改善手法と人手によるテストケースを利用した表明生成を比較すると，改善手法において表明の生成数が明らかに減少していることがわかる．特に，事前条件について，事後条件に比べて減少した割合が高いことがわかる．

これは，改善手法では 1 つのメソッドに対するテスト

表 2: 表明を動的生成できたメソッドの数と割合

	単体テストによる生成	改善手法による生成	全体
メソッド数	376	36	955
割合	0.40	0.038	1

表 3: 動的生成手法による表明の生成数

生成方法	メソッド数	事前条件	事後条件	合計
単体テスト（全体）	376	6345	20052	26397
単体テスト（一部）	36	735	1753	2488
改善手法	36	59	278	337

データを投入する数が，人手によるテストケースより多いことが理由として考えられる．人手でテストケースを記述した場合最大で 1 メソッドあたり数個のテストデータを与えるのみであるが，改善手法では 1 つのメソッドにおいて最低 20 個のテストデータを与える．改善手法ではテストデータ生成はランダムであるため，同一の値を入力として与える可能性もあるが，より多くのテストデータを与えることによって表明の出力数を減少させることができると考えられる．特に，事前条件において表明生成数が減少したことは，この現象を表している理由と考えられる．

5.2.2 人手で記述した理想的な表明との比較

表明数の比較 人手で記述した表明数と改善手法で生成した表明数および人手で記述した単体テストを利用した表明の生成数を表 4 に示す．

人手による表明の数に比べ，改善手法や人手で記述したテストケースを利用した動的生成手法で生成した表明の数が多いことがわかる．人手による表明は，コードから人間が容易に分かる表明を中心に記述してあるため，実際に動作させて初めて分かる表明などは記述されていない．

一方，動的生成による表明は，コードを実際に実行させて表明を出力するため，テストデータに依存した表明など一般に成り立たない表明が出力されるものの，人間が想定していない表明が出力されるため，表明記述の見落としを発見する手がかりになる可能性がある．しかし，表明生成数が多くなるとそれぞれの表明が一般的に成り立つかどうか検証する手間が増えるため，必要な表明を残しつつもできるだけ表明の生成数は少ないほうが望ましいと考えられる．

表明精度の比較 表明の精度について人手による表明と改善手法による表明の比較（表 5）を中心に詳細に評価する．人手による表明について，事前条件は『引数が Null 参照でない』という条件であった．この条件は引数が Null 参照の際に処理ができないメソッドに対してのみ記述されていた．事後条件は,『引数が Null 参照でないときフィールドの値は引数になり，そうでなければ Null 参照になる』という setter メソッドの条件や，パスワードのハッシュ化を行うメソッドでは,『返り値の String のハッシュ後の長さが 32 である』という条件が記述されていた．

一方，改善手法によって生成された表明は，事前条件は『引数が Null 参照でない』という条件や『引数が正数である』といった一般に成立しうる条件の他に,『すでにフィールドに値が設定されている場合は引数とフィールドの値は一致しない』という条件や『引数は（対象フィールドとは異なる無関係な）フィールドの値より大きい（小さい）』といった，テストデータの入力に依存した表明や無意味な表明を生成した．また，事後条件においては,『引数で与えた値がフィールドの値になる』といった setter メソッドの表明が出力されたものの，多くは『他のフィールドの値同士が一致する』という一般的に成り立たない表明が生成された．これは，テストデータをテスト対象メソッドに投入するごとにテスト対象のオブジェクトのフィールドの値をランダムに設定していないため起こる現象であると考えられる．この問題については解決が可能と考えている．

また，人手によるテストケースによって生成された表明は，事前条件はテストデータに依存した表明である『引数は 2006,2007,2008 のいずれかである』のような値を含む表明や，改善手法においても生成された『引数は（対象フィールドとは異なる無関係な）フィールドの値

表 4: 動的生成手法による表明と理想的な表明の生成数

人手で記述した表明		改善手法による表明		単体テストによる表明	
事前条件	事後条件	事前条件	事後条件	事前条件	事後条件
2	36	59	278	735	1753

より大きい（小さい）』といった表明が生成された．事後条件においては，改善手法で生成された『引数で与えた値がフィールドの値になる』といった setter メソッドの表明が生成されたのに加え,『他のフィールドの値はメソッド呼び出し前のフィールドの値と同じ』という表明も生成された．これは，各フィールドの値が全体のテストケースを実行することを通じて変化し，対象となるオブジェクトの状態が多くなったために生成された有用な表明であると考えられる．

６． テストデータ生成改善手法の検討

6.1 テストデータ生成問題

表明の動的生成においては，その入力となるテストデータの生成方法は生成される表明の制度に影響を与える重要な要素である．本稿の実験では，人手で生成したテストケースと表明動的生成改善手法で生成したテストケースでは生成表明について，具体的な値を含む表明の生成割合が異なっていた．

人手でテストケースを生成する場合，境界値分析などを行い効率的にカバレッジの高いテストケースを生成することが一般的であると考えられる．また，テストにかかる工数や時間的・空間的な制約から，入力値としてとりうる値を全てをテストデータとして生成することは現実的ではない．参照型変数，特に複数フィールドを持つオブジェクトの場合は，各フィールドに対して入力値を設定する必要がある．

6.2 テストデータ生成改善手法

現在，われわれの研究グループでは，インバリアントカバレッジに基づくテストケースを生成することを提案しており，インバリアントカバレッジを満たすテストケースをプログラムの戻り値の変化しうるプログラムパスの集合に近似している．各プログラムパスを通るテストデータの条件をテストケース制約として求め，テストデータ生成に利用している．

しかし，現在の実装においては，対象となるテストデータの生成は乱数によるテストデータ生成を行っているため，テストケース制約を満たすテストデータの生成が効率的でない．文献 [16] は，テストデータに付加されている表明を利用し，モックオブジェクトを生成する手法について提案を行っている．モックオブジェクトを利用する場合，データベースやネットワークなど外部の環境をテスト実行と分離できる [16] ため，表明動的生成では有効である．参照型変数のランダム生成に比べて効率的にテストデータを生成することが可能であるが，対応する表明が部分的であることや，具体的なフィールド値固定値とすることが課題である．

また，文献 [17] では，記号実行と状態探索木を用いたテストデータ生成列の生成方法を提案している．これは，テストデータクラスのメソッドの呼び出しによるテストデータオブジェクトの状態遷移を木構造で保持し，各状態間を同値判定することにより，効率的なテストデータ生成を行う事が出来る手法である．テストデータクラスのメソッド呼び出しの最大長を限定する必要があるもの，各フィールドの値域を効果的にカバーできるため，精度の高い表明を生成するのに有効であると期待できる．

表明の動的生成のためのテストデータ生成を考えた場合，できるだけ多くのテストデータを生成することが望ましい．テストデータが少ない場合は，表明中に値を含む可能性があり，一般的な表明とならない可能性がある．いずれの既存研究においても，テストデータを最終的に一定の値としてテストデータを生成するため，ある一定の条件を満たした複数のテストデータを生成したいという要求に対しては改善する必要がある．

6.3 テストデータ生成対象の検討

また，改善手法の問題点として，テストデータ生成対象が限定されるという問題がある．表 6 は，対象としたシステム全体において，メソッド引数の有無及び引数の型の種類によってメソッド数を計測した結果である．

現在の改善手法の実装が生成可能なテストデータが基本型とその配列および java.lang.String 型のみである．本稿で利用した情報システムの場合，引数を取らないメソッド（主に getter メソッド）を多く定義するため，表明生成改善手法が有効に働かないメソッドが存在する原因となっている．これらのメソッドの表明を精度よく生

表 5: 人手による表明記述数と表明動的生成改善手法による生成表明数の評価

	人手による表明	表明動的生成改善手法による表明			
		正しい表明	有用な表明	不要な表明	誤った表明
事前条件	2	2	25	1	31
事後条件	36	36	64	73	105

表 6: メソッド引数のクラスによる分類

	対応	配列	Java ライブラリ	外部ライブラリ	システム固有	列挙型	引数なし
該当メソッド数	239	14	128	56	234	2	321

成するには，テスト対象のオブジェクトのフィールド値を適切に設定することが有効であると考えられる．特にインバリアントカバレッジが向上するようなパスの条件式中に現れるフィールド変数の制約を求め，制約を満たすフィールド変数の値を設定することで，精度の良い表明が生成されると考えている．

また，システム固有のクラスのテストデータを生成することも重要である．システム固有のクラスの場合，表明生成ツールとしてテストデータ生成クラスをあらかじめ準備することは難しいため，モックオブジェクトを利用したテストデータ生成が有効であると考えられる．テストケース制約中に現れるシステム固有クラスのオブジェクトをモックオブジェクト化し，テストケース制約を満たすように変更を加えた上でテストデータとして利用することが考えられる．

本稿の実験対象のメソッド数としては少ないものの，Java ライブラリのクラスや配列型の引数を取るメソッドは一般的であると考えられる．これらのクラスは仕様が定まっているため，テストデータとしてランダム生成がしやすいと考えられる．これらのクラスのテストデータを生成できるよう改善をすることは有効である．

7． おわりに

本稿では，表明動的生成手法における改善手法を実プロジェクトの教材として開発されたシステムに対して適用した．システム付属の人手によるテストケースを利用して生成した表明や文献 [15] において人手で記述した表明と，改善手法によって生成された表明とを比較した．その結果，動的生成手法によって表明を生成した場合，テストデータに依存した表明が多く生成され，メソッドの性質を表す有用な表明は少ないことがわかった．一方，改善手法などを用い，多くのテストデータを用意することや，テスト対象となるオブジェクトの状態を増やすことにより，不要な表明を減少させたり，有用な表明を出力させやすくなることがわかった．

今後の課題として，テストデータやテスト対象のオブジェクトの状態を変化させた場合に生成される表明の精度がどのように変化するかや，テスト対象の状態を分割する手法 [18] などを利用した場合の表明の精度について，定量的に調査する必要がある．

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] 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*, pages 500–5178, 2001.

[2] 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*, 69(1-3):35–45, 2007.

[3] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. *Proc. 30th ACM/IEEE Int. Conf. on Software Engineering (ICSE)*, pages 281–290, 2008.

[4] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 191–200, New York, NY, USA, 2011. ACM.

[5] 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*, pages 152–171, 2001.

[6] 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*, pages 49–58, 2003.

[7] 宮本敬三, 堀直哉, 岡野浩三, 楠本真二, and 西本哲. Java に対するループインバリアントを含む Daikon 生成アサーションの妥当性評価. **電子情報通信学会論文誌** *D*, J91-D(11):2721–2723, 2008.

[8] 堀直哉, 岡野浩三, and 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** *XV* **日本ソフトウェア科学会ソフトウェア工学の基礎ワークショップ** *FOSE2008*, pages 41–50, 2008.

[9] 宮本敬三, 岡野浩三, and 楠本真二. 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. **ソフトウェア工学の基礎** *XVI* **日本ソフトウェア科学会ソフトウェア工学の基礎ワークショップ** *FOSE2009*, pages 183–190, 2009.

[10] L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *In T. Arts and W. Fokkink, editors, Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS03), Electronic Notes in Theoretical Computer Science*, 80:73–89, 2003.

[11] 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*, pages 11–20, 2002.

[12] 小林和貴, 宮本敬三, 岡野浩三, and 楠本真二. 表明動的生成を目的としたテストケース制約の ESC/Java2 を利用した導出. **ソフトウェア工学の基礎** *XVII* **日本ソフトウェア科学会ソフトウェア工学の基礎ワークショップ** *FOSE2010*, pages 35–44, 2010.

[13] G. T. Leavens, Albert L. Baker, and C. Ruby. JML:A Notion for Detailed Design. *in Behavioral Specifications of Businesses and Systrems*, pages 175–188, 1999.

[14] 三宅達也, 肥後芳樹, 楠本真二, and 井上克郎. 多言語対応メトリクス計測プラグイン開発基盤 MASU の開発. **電子情報通信学会論文誌** *D*, J92-D(9):1518–1531, 2009.

[15] 宮澤清介, 花田健太郎, 岡野浩三, and 楠本真二. OCL から JML への変換ツールにおける対応クラスの拡張と教務システムに対する適用実験. In **信学技報**, volume 110 of *SS2010-72*, pages 115–120, 2011.

[16] Stefan J. Galler, Andreas Maller, and Franz Wotawa. Automatically extracting mock object behavior from design by contract specification for test data generation. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 43–50, New York, NY, USA, 2010. ACM.

[17] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *In TACAS*, pages 365–381, 2005.

[18] Nadya Kuzmina, John Paul, Ruben Gamboa, and James Caldwell. Extending dynamic constraint detection with disjunctive constraints. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08, pages 57–63, New York, NY, USA, 2008. ACM.