社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

SMT ソルバーと PDG 作成ツールを用いた Java のテストケース自動導出手法の提案

佐々木幸広† 小林 和貴† 岡野 浩三† 楠本 真二†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †y-sasaki@ics.es.osaka-u.ac.jp, †{k-kobays,okano,kusumoto}@ist.osaka-u.ac.jp

あらまし Design by Contract に基づく表明の生成手法には静的生成手法と動的生成手法が存在する．動的生成手法では，テストケースを用いて引数やフィールド変数の値を変えながら対象メソッドを解析し，データを取得するため，比較的少ないコストで表明を生成できる．一方で，テストケースがどれだけの実行範囲をカバーしているかというテストケースの質が生成される表明の質に影響を与える．この問題に対し，インバリアントカバレッジという指標が存在する．本研究では対象メソッドの PDG からインバリアントカバレッジを満たす可能性のある実行パスを列挙し SMT ソルバを用いてそのパスの実行可能性と具体的な引数値の導出を行う方法を提案する．また，複数の例題に適用し有効であることを確認した．

キーワード SMT ソルバ, PDG, Daikon, テストケース, Java

Automatic derivation of test cases for Java using an SMT solver and a PDG generator

Yukihiro SASAKI†, Kazuki KOBAYASHI†, Kozo OKANO†, and Shinji KUSUMOTO†

† Osaka University
E-mail: †y-sasaki@ics.es.osaka-u.ac.jp, †{k-kobays,okano,kusumoto}@ist.osaka-u.ac.jp

Abstract Assertions and Design by Contract play important roles for program understanding and program verification. In order to generate assertions, there are two kinds of methods, static analysis and dynamic analysis. Dynamic approach needs less CPU times, however suffers the test case depending problem in which the quality of assertions depends on the quality of test cases. Invariant coverage is one of indicator to specify the quality of test cases. In other words, the coverage of test cases affects the quality of generated assertions. The proposed method uses PDG of a target method to generate a candiate path set and uses an SMT solver to decide whether a candidate path is executable or not, as well as a concrete set of real parameters. We have implemented a tool and evaluated the usefulness of the proposed method through several programs.

Key words SMT solver, PDG, Daikon, test case, Java

1. はじめに

オブジェクト指向ソフトウェアの設計における Design by Contract(DbC) [1] はプログラムにおける仕様をソースコード中に記述することにより，ソフトウェア開発における誤りの箇所を明確にする手法である．これにより，仕様理解の補助やプログラムの妥当性を確認できる．Java 言語においてはプログラムのソースコード中に事前条件・事後条件・不変条件などを表明という形で記述し，静的解析ツール ESC/Java2 [2,3] などを用いてプログラムを検証できる．これによりプログラムの保守性や信頼性が向上する．しかし，プログラムのソースコード量の増大により，人手で表明をソースコードに記述するのは困難となる．これを解決するために，表明を自動で導出するツールとして Daikon [4,5] がある．しかし，テストケースがどれだけの実行範囲をカバーしているかというテストケースの質が生成される表明の質に影響を与える [6]．そこで，著者らが所属する研究グループでは，表明導出に利用するために有効なテストケースを自動で生成する手法を研究している．

本報告の提案手法では，対象プログラムのメソッドに対して，プログラム依存グラフ (PDG) [7] からインバリアントカバレッジ [8,9] に影響を及ぼす実行パスを抽出する．そしてインバリアントカバレッジを満たすパスを SMT ソルバ [10,11] の文法 SMT-lib2.0 Standard [12] 文法に準拠した制約式に変換し，SMT ソルバの解を用いてテストケースを導出する手法である．

また，本稿において提案した手法を実装した．ツール全体での出力の妥当性を検証するため，Java プログラムのソースコードに対してツールを適用する実験を行った．その結果，いくつかのメソッドに関して本稿で示す既存手法 [13] の課題点を解消する結果が出力された．

以降，2 章で研究の背景について述べ，3 章で提案手法について述べ，4 章で評価について述べ，5 章でまとめる．

2. 準　　備

研究の背景となる諸技術と関連研究について簡単に触れる．

2.1 表　　明

Java プログラムにおいては，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，DbC における契約を記述できる．この表明により，ESC/Java2 などの静的解析器を用いて，プログラムの妥当性を検証することで，開発者の意図しない不具合の混入を防止できる．

2.2 表明の自動導出手法

ソースコードサイズの増加に伴い，人手による表明記述は困難となる．そこで，表明の自動生成手法が注目されている．表明の生成の自動化手法に，静的手法と動的手法の 2 種類がある [14].

2.2.1 表明の静的自動導出手法

表明の静的自動導出手法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，精度の高い表明を自動生成できる．Houdini [15] は静的解析器 ESC/Java2 を繰り返し適用することにより，表明を生成する．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．

2.2.2 表明の動的自動導出手法

表明の動的自動導出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．ここでテストケースとは，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．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が，一般的に比較的少ない時間とメモリで表明を生成できる．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 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，デバッグに有効である．

```
01: int example(int x, int y) {
02:    int ret=0;
03:    if (x > 0) {
04:      x=x+2*3;
05:      ret=y;
06:    }
07:
08:      return ret;
09: }
```

図 1　メソッド example のソースコード

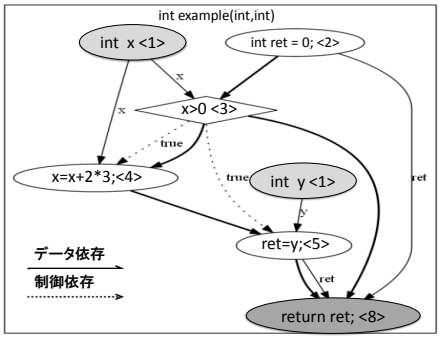

図 2　メソッド example の PDG

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とき，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，実行パスの網羅性が保証されず，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

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．インバリアントカバレッジ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 [8,18].

2.5 プログラム依存グラフ

プログラム依存グラフ (PDG) とは，プログラム中の命令をノード，ノード間の依存関係を有向辺で表したグラフである [7]. 主な依存関係には制御依存とデータ依存がある．制御依存辺は次に実行しうるノード間に構築され，データ依存辺はデータ依存があるノード間に構築される．図 1 で示すメソッドに対する PDG を図 2 に示す.

2.6 SMT ソルバ

SMT [10](Satifiablity Modulo Theories) は特定の分野の数学的な体系を表す理論である．例えば，算術の理論，集合の理論などがある [11]．幾つかの SMT ソルバでは一階述語論理の充足可能性判定問題を解くことができる．なお，充足の可否だけでなく，充足する場合の変数への割り当て方の情報も含む．

2.7 Z3

Z3 [16] は，Microsoft 社が開発した SMT ソルバである．自社で開発するプログラム解析，テスト，検証など様々なツール内で Z3 を利用している [11]．Z3 は本研究で用いる SMT ソルバであり，線形の実数や整数の算術や配列，レコードの表現など多様な機能を持ち，また解の導出速度が速く正確である．文法は SMT-lib2.0standard 言語 (以下 SMT 言語という) に従う．

2.8 既存手法

本節では著者が所属する研究グループが提案しているテストケース生成手法の概要と課題点について述べる．

2.8.1 既存手法概要

既存手法では以下の手順でテストケースを生成する．

（1） 対象メソッドの PDG を生成する

（2） 手順 1 で生成した PDG よりインバリアントカバレッジを満たすパス p を 1 つ取得する．

（3） パス p を通るトラップ変数を適切に埋め込み，ESC/Java2 [2] を用いてそのトラップ変数の組み合わせが真にならない（すなわちパス p を通らない）という性質 s_p を作る．

（4） ESC/Java2 を実行し，s_p の反例を得る．この反例はパス p を通る証拠となる．

（5） 手順 4 で取得した反例を解析し，Daikon を用いる際に使用するテストケース生成に必要な，パス p を通る十分条件を得る．

（6） (2)-(5) をインバリアントカバレッジを満たす全てのパス候補だけ繰り替えす.

（7） テストケースを実行するときは制約を満たす値を乱択により導出し必要回数繰り返す.

2.8.2 既存手法の課題点

既存研究には，反例から得られたテストケース制約が実際のパス実行条件より狭い．

この問題点は静的解析器である ESC/Java2 の反例出力を用いているために，実際のパス実行条件より狭いテストケース制約しか導出されないためおこる．

例えば， `x>0 || y>0` が実際のテストケース制約であったとしても，`x<=0 && y>0` が導出される場合がある．

このように実際より狭いテストケースしか生成されないため，テストケースの質が下がり，結果表明の質が下がるという問題がある．

3. 提案手法

2.8.2 小節であげた課題を解決するため，テストケース生成手法に SMT ソルバ，Z3 を用いる．SMT ソルバでは反例として制約を導出せず，パスの分岐条件そのものを用いてテストケースを求めるため，得られた制約が狭くなることはない．また，SMT ソルバの解は制約ではなくその制約を満たす割り当てを返すため，（7）のような乱択を再び繰り返すという無駄を省ける．SMT ソルバを用いたテストケースの導出のためには，入力として SMT 言語で書かれた制約を満たすための式を与えなければならない．そのために実行パス候補を SMT 言語で書かれたパス実行条件の式への変換について実装した．これにより，既存研究のテストケース生成手法を改善できる．

3.1 提案手法概要

入力:Java の 1 メソッド

出力:テストケース

（1） 対象メソッドの PDG を生成する．

（2） 手順 1 で生成した PDG よりインバリアントカバレッジに影響を及ぼす 1 実行パスを取得する．

（3） 手順 2 で得られたパス集合を実行するための条件式を SMT 言語の文法に変換する．

（4） SMT ソルバに入力し，パス実行条件を満たす変数割り当てを求める．

（5） (2)-(4) をインバリアントカバレッジに影響を及ぼす全ての実行パス候補分繰り返す.

（6） 変数割り当てからテストケースを求める．

3.2 PDG から 1 実行パスの抽出

まず，対象メソッドに対し，MASU [17] を用いて PDG を生成する．PDG を利用し，インバリアントカバレッジに影響を及ぼす，すべての実行パスを取得する．実行パスの取得法は次の通りである．まず PDG から，返り値にデータ依存をするノードを取得する．次に変数定義が制御依存する式を取得する．制御依存する式中で使用されている変数に関しても，データ依存・制御依存を取得する．取得した変数定義を組み合わせて選択することで実行パス候補を導出する．ただし，これらの実行パスが実際に実行可能かどうかは，パスを抽出した時点では不明である．

3.3 1 実行パスから SMT 言語への変換

入力を Java の 1 実行パスとし，出力を SMT 言語で表現されたその実行パスを通るための変数割当を求める SMT 言語の式とする．本節では，変換規則の一部について説明を行っていく．変換は抽出された実行パスの系列に対し行う．以下, 変換規則 μ を次のように与える μ: $EP \rightarrow ES$

ここで EP，ES はそれぞれ，PDG の 1 ノードに相当する Java の 文，SMT 言語への部分式である．

3.3.1 変換補助関数

変換に当たり，以下の補助関数を用いる．

（1） String name(String variable)

（2） String use(String variable)

（3） String type(String Java_type)

（4） int class(String className)

（5） int point(String Instance)

補助関数 *name* とは，変数名 *varibale* を引数にとり，変数名に ID をつけたものを返す．なお，ID は変数が更新された回数を表し，各変数ごとに ID を持つものとする．ID の初期値は 0 である．すなわち，i 回値が更新されている変数が，*name* 関数により呼び出されたときは，$variable\%i+1$ を返す．そしてその変数の ID をインクリメントする．

補助関数 *use* は，変数名 *variable* を引数にとり，変数名に ID をつけたものを返す．ID は変数が更新された回数を表す．すなわち，各変数の最新の値をかえす関数である．i 回値が更新されている変数が，*use* 関数により呼び出されたときは，$variable\%i$

表 1　Java 基本型と SMT 基本型の対応

Java の基本型	byte	short	int	char	long	float	double	boolean
SMT 基本型	Int	Int	Int	Int	Int	Real	Real	Bool

Java:	Java:
integerExpression	realExpression
SMT:	SMT:
(to_real integerExpression)	(to_int realExpression)

図 3　プリミティブ型のキャストの表現

を返す.

補助関数 *type* は，Java での変数の型 *Java_type* を引数にとり，それに対応する SMT 言語での型を返すものとする.

補助関数 *class* は，クラス名を表す変数 *className* を引数にとり，クラスごとに定義しておいた固有の整数値を返す.

補助関数 *point* は，インスタンスを表す変数 *Instance* が何番目にインスタンス化されたかを返す関数である.

インスタンスが参照している場所を区別するために用いる.

3.3.2　変数の変換

SMT 言語では代入という概念が存在せず，式は等式か不等式でしか表すことができない．よって SMT 言語では同一の変数名であっても，変数の後に連番をつけることで区別をする．そこで前述の補助関数 *name*, *use* を用いる.

3.3.3　プリミティブ型の対応

Java プログラムにおいて，定義される変数は SMT-LIB では表 1 のように表す．ここで *java_type* は Java プログラムでの型，*variable_name* は変数名，*smt_type* は SMT-LIB での型を表す.

3.3.4　式 の 変 換

本節では代入以外の式の変換方法について説明する．Java 演算子の優先順位に従って，前置記法で表現する．例えば $1+2*x$ のような変換は演算子の優先順位にしたがい，$(+\ 1\ (*\ 2\ use(x)))$ のように変換される.

3.3.5　プリミティブ型のキャスト変換

Java 言語では整数と実数の演算が可能であるが，SMT 言語では整数と実数の演算は不可能である．例えば Java では 1+1.0 などは自動でキャストされるが，SMT 言語では 1+1.0 のように型の違う演算を行うとエラーを出力する．そこで，整数を実数に変換する SMT 言語の関数 *to_real*，*to_int* を用いる．整数の部分式 *integerExpression*, 実数の部分式 *realExpression* はそれぞれ図 9 のように変換を行う．表 3 の左側を μ の引数，右側を μ の返り値とする.

3.3.6　配列の変換

図 4 に 1 次元配列の宣言と使用, 代入についての変換を示す.

3.3.7　文字の変換

Java 言語では文字は文字コードを用いて整数として表すことができる．よって文字を表す char 型に関しては整数型 Int を用いて表す．String 型も同様に整数型の配列として表現する.

Java:
array_Type array_Name;
a[i];
a[i]=expression;
SMT:
(declareconst name(array_Name) Array Int type(array_Type))
(select use(a) i)
(assert (= (store use(a) i expression) name(a)))

図 4　配列の変換例

表 2　実験対象となるソースコード

プログラム名	行数	実行パス候補数	平均パス長
Calc	21	2	3.5
Telephone	29	7	5.5
Flower	28	41	9.5
JanCord	26	3	13
Operator	21	6	3.7
RPS	19	5	5
Date	28	9	5.4

Calc:2 次方程式の求解
Telephone:電話番号がどの種別を表すかの判定
Flower:窓辺の花 [19] のプログラム
JanCord:8 桁のバーコードチェックディジットが正しいかを判定する
Operator:演算子を表す文字を与えたとき，対応する演算を行う
RPS:じゃんけんの判定
Date:二つの日付のうちどちらが新しいかの判定

className
ftype fieldName; …
rType methodName(pType parameterName); …

図 5　定義対象クラスの UML 表現

Java:
className oName = new className();
oName.ftype;
SMT:
(declare-datatypes () ((mk-obj (field (fieldName type(ftype))
(%pointer Int) (%class Int)))))
(declare-const name(oName) type(className))
(assert (= (%pointer use(a)) point()))
(assert (= (%class use(a)) class(className)))
(fieldName use(oName))

図 6　インスタンス化の変換

3.3.8　クラスのフィールドの変換

まずクラスのフィールドを表すために，そのフィールドを持つレコードを定義しておく．図 5 のクラスがあるとき，インスタンス化とフィールドの参照の変換方法は図 6 のようになる．ここで，%pointer はインスタンスが参照している場所，%class はクラスの種類を表すものとする．以下，変換を表す図では上段を変換規則 μ の引数とし，下段を変換規則 μ の返り値とする.

本変換により，既存の研究では表現できなかったクラスに対してもテストケースが導出できる.

Java:
method_Name(parameter_Name)

SMT:
(declare-const name(methodName) type(method_type))
(declare-const name(instance) type(className))

図 7　メソッド起動式の変換

Java:
variable_type variable;
variable = Expression;

SMT:
(declare-const name(variable) type(variable_type))
(declare-const name(variable) type(variable_type))
(assert (= use(variable) Expression))

図 8　代入文，宣言文の変換

3.3.9　メソッド起動式の変換

式中でメソッドを呼び出すときの変換方法について述べる．メソッドの返り値に関しては，厳密な値を求めようとすると，状態爆発を起こす場合がある．そのためメソッドの返り値は自由な値をとるものとする．void 型のメソッドに関しては，変換しない．図 5 のクラスのメソッド *methodName* の変換は，図 7 のようになる．

この変数 *name*(*methodName*) を式中に使うことで，メソッドの返り値を表現する．既存研究ではうまく定めることができなかったメソッドの返り値を求解できる．またこのメソッドを呼出したインスタンス *instance* はメソッドによりフィールド値が変更される場合があるため，インスタンスそのものも更新している．

3.3.10　条件式の変換

条件文を *condition* としたときに，条件が真となるパスを通るときは，

```
(assert  condition)
```

条件が偽となるパスを通るときは，

```
(assert (not condition))
```

と変換をおこなう．

3.3.11　変数宣言の変換

変数名を *variable*，変数の型を *variable_type* とすると，変数宣言は図 8 のように変換を行う．

3.3.12　代入文の変換

変数名を *variable*，変数の型を *variable_type* とし，代入する式を *Expression* とすると，代入文は図 8 のように変換する．

3.3.13　if,while,for 文の変換

if 文は PDG 上では分岐条件のみ与えられ，その分岐条件が真となるパス，偽となるパスをそれぞれ取得することで対応する．while 文と for 文は任意の整数回だけループするようにパスを展開し，そのパスそれぞれに対して変換を行うことで対応する．

3.3.14　複文の変換

PDG から得た実行パス系列に対し，本稿で示してきた変換規則 μ を各文ごとに順に適応していくことにより，複文全体を変換することが可能になる．

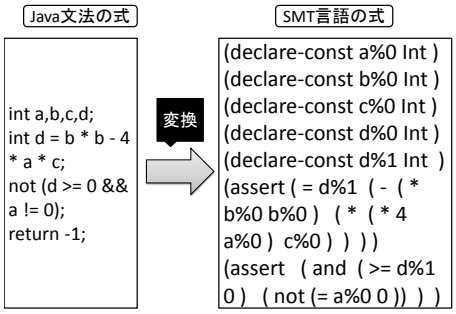

図 9　2 次方程式の解判定の変換

表 3　実 験 結 果

プログラム名	実行時間 (秒)	インバリアントカバレッジ	テストケース数
Calc	28.6	100	40
Telephone	23.4	92	26
Flower	631.5	100	29
JanCord	161.2	100	44
Operator	25.9	100	38
RPS	26.4	100	34
Date	29.8	100	160

3.3.15　変　換　例

変換例を図 9 に示す．この変換は 2 次方程式 $ax^2 + bx + c$ が実数解を持たないときの変数の割り当てを求める SMT 言語の制約式へと変換を行っている．なお，return 文は変換しない．

3.4　SMT ソルバを用いたテストケースの導出

制約を表す式が得られた後，満たす割り当てを SMT ソルバで求解する．この処理により，パスの実行可能性と，実行可能な場合の変数への割り当てを求めることができる．引数が以前求めた値と異なるという条件を加えて繰り返し実行することにより，同じ実行パスを通る複数の変数への割り当てを求めることができる．与えるテストケースの数が多いほど Daikon では質の高い表明を導出できる．

4.　評 価 実 験

本章では，評価実験の結果および考察について述べる．本研究では評価実験として，手法を実装したツールを本章にて示す Java プログラムに対して適用した．本ツールを実行時間及び出力結果の妥当性の観点から評価を行う．

4.1　評価実験の目的

本提案手法が実プログラムにおいて適用可能な範囲や，テストケースがインバリアントカバレッジを満たすための妥当な値であるかを様々なプログラムを対象に適用実験を行った．

4.2　実 行 環 境

実行環境には，`Intel(R) Core(TM)2 Duo CPU U9300 1.20GHz Memory:2.00GB,Windows Vista,JDK1.6_17` および `Z3-3.2` を使用した．

4.3　適 用 対 象

本稿における実験対象を表 2 に示す．これらのソースコード

の一部は本研究グループの行った既存研究 [18] でも実験対象としている．また，既存手法では本来より狭いテストケース制約を導出する実験対象となっている．

4.4 実験の手順と結果

提案手法から得られたテストケースのインバリアントカバレッジと実行時間，テストケース数を計測する．また得られたテストケースから既存手法で得ることのできないテストケースがあるかを調査する．実験結果として，表 3 の結果が得られた．なおテストケース数は，値の同じテストケースは一つとして数える．

4.5 考　　察

提案手法で生成したテストケースが既存手法で導出できるかについて考察を行う．例として，flower というプログラムでは既存手法では $0 \leqq num \leqq 3$ の否定から $num \geqq 4$ というテストケース制約を得られたが，これは $num < 0$ というテストケース制約が抜けているため，制約としては不十分であり，Daikon で正しく表明を生成できない．しかし，本提案手法では，$num = -1, a[0] = true$ など既存手法では生成できないテストケースを得られた．また，このテストケースは表明の生成をするうえで有用である．Telephone のインバリアントカバレッジが 100%でないのは，提案手法ではメソッドの返り値を任意の値を返すと定義していたり，配列の長さを SMT では求めることができないためである．実行時間に関しては現実的な実行時間でテストケースを生成することができたといえる．実行時間は PDG の生成と SMT ソルバの求解が大部分を占めている．本手法では実行時間はパス数や生成したテストケース数，平均のパス長に依存している．既存手法との比較として，テストケース制約の広さの点から提案手法がよりメソッドの状態を詳しく表す表明を生成するために必要なテストケースを生成可能であるといえる．

5. あ と が き

本研究では我々の研究グループが提案している，表明自動導出手法のためのテストケース生成手法の提案とその評価実験を行った．提案手法では，SMT ソルバや，PDG を利用してテストケース依存問題を解消している．その結果，既存手法で導出できなかったテストケースを導出できた．しかし，適用クラスの狭さが課題点である．今後は適用可能なクラスを拡張し，その評価実験を行うことで本手法の有用性を評価していきたい．

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．

文　　献

[1] B. Meyer, “Applying Design by Contract,” in Computer(IEEE), vol.25, no.10, pp.40–51, 1987.
[2] C. Flabagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata, “Extended static checking for Java,” Proc. of the ACM SIGPLAN 2002, pp.234–245, 2002.
[3] D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe, “Extended static checking,”, Compaq SRC Research Report, 1998.
[4] M.D. Ernst, J.H. Perkins, S.M. P. J. Guo, 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.
[5] M.D. Ernst, “Dynamically discovering likely program invariants,” Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.
[6] 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.
[7] J. Ferrante, K.J. Ottenstein, and J.D. Warren, “The program dependence graph and its use in optimization,” ACM Trasactions on Programming Languages and Systems, vol.9, no.3, pp.319–349, 1983.
[8] 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.
[9] N. Gupta, “Generating test data for dynamically discovering likely program invariants,” in Proc. of ICSE 2003 Workshop on Dynamic Analysis, WODA 2003, pp.21–24, 2003.
[10] L. De Moura, and N. Bjørner, “Satisfiability modulo theories: introduction and applications,” Commun. ACM, vol.54, pp.69–77, Sept. 2011.
[11] 梅村晃広，“SAT ソルバ・SMT ソルバの技術と応用”，コンピュータソフトウェア，vol.27，no.3，pp.24–35，2010.
[12] D.R. Cok, “The smt-lib v2 language and tools: A tutorial,” GrammaTech, 2011.
[13] 小林和貴，宮本敬三，岡野浩三，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，pp.35–44，2010.
[14] 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.
[15] 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.500–5178, 2001.
[16] L. De Moura, and N. Bjørner, “Z3: An Efficient SMT Solver Tools, and Algorithms for the Construction and Analysis of Systems,” vol.4963/2008, chapter 24, pp.337–340, Lecture Notes in Computer Science, Springer Berlin, Berlin, Heidelberg, April 2008.
[17] 三宅達也，肥後芳樹，楠本真二，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，vol.J92-D，no.9，pp.1518–1531，2009.
[18] 宮本敬三，堀直哉，岡野浩三，楠本真二，“Daikon 生成表明改善のためのテストケース自動生成手法とその評価実験”，日本ソフトウェア科学会誌コンピュータソフトウェア，vol.28，no.4，pp.306–317，2011.
[19] 産業技術総合研究所システム検証研究センター，“4 日で学ぶモデル検査 (初級編)(CVS 教程 (1))”，エヌティーエス，2006.