情報処理学会論文誌 Vol.62 No.4 1039–1050 (Apr. 2021)

記号実行のために前処理機能を導入した業務アプリケーション向けテスト入力値生成システム

大林 浩気[1,a)] 鹿糠 秀行[1]

受付日 2020年8月3日, 採録日 2021年1月12日

概要：記号実行技術を元にしてテスト入力値を自動生成する Java 向け記号実行ツール SPF（Symbolic Path Finder）は，Java EE（Java Platform, Enterprise Edition）などの業務アプリケーションフレームワークの一部機能を使用したプログラムが解析できず，業務アプリケーションへの適用が難しいという問題があった．本研究では，SPF を改修することなく業務アプリケーションに対して記号実行を適用可能にするため，業務アプリケーションのプログラムを前処理によって SPF が受理可能なプログラムに変換するアプローチを提案する．前処理として，データ型の書き換え，動的呼び出しの書き換え，スタブ化の機能を導入し，業務アプリケーション向けのテスト入力値生成システムを開発した．評価実験を行い，前処理機能によって SPF を改修することなく業務アプリケーションを実用時間内に解析でき，テスト入力値生成システムを利用することで網羅性の高いテストケースを手作業よりも少ない工数で作成できることを示す．

キーワード：ソフトウェアテスト，テスト入力値自動生成，記号実行

A Test Input Value Generation System for Enterprise Application to Introduce Preprocessing Functions for Symbolic Execution

Hiroki Ohbayashi[1,a)] Hideyuki Kanuka[1]

Received: August 3, 2020, Accepted: January 12, 2021

Abstract: An existing symbolic execution tool for Java, SPF (Symbolic Path Finder), which automatically generates test input values based on symbolic execution techniques, has a problem that it is difficult to apply to business applications because programs using some functions of business application frameworks such as Java EE cannot be analyzed. To apply symbolic execution to business applications without modifying the SPF, we propose an approach to convert a business application program into a program that can be accepted by the SPF by pre-processing. We studied three types of pre-processing: rewriting of data types, rewriting of dynamic calls, and stubbing. We developed a test input value generation system for business applications developed with Java that introduces the proposed pre-processing. The evaluation experiments show that the pre-processing function allows business applications to be analyzed in real time without modification of the SPF, and that the test input value generation system allows for the creation of highly comprehensive test cases with less effort than manual work.

Keywords: software testing, automatic test generation, symbolic execution

1. はじめに

ソフトウェアテストの効率化を目的に，テスト入力値を自動生成する手法が研究されている [1]．その 1 つとして記号実行技術 [2] をもとにしたテスト入力値生成手法は，プログラムの全実行パスを通過するための条件を静的

[1] 株式会社日立製作所研究開発グループシステムイノベーションセンタ
Center for Technology Innovation, Systems Engineering, Research & Development Group Hitachi, Ltd., Yokohama, Kanagawa 244–0817, Japan
[a)] hiroki.ohbayashi.eo@hitachi.com

© 2021 Information Processing Society of Japan

解析によって抽出し，条件を SMT（Satisfiability Modulo Theories）ソルバ [3] を用いて制約充足問題の解としてテスト入力値を得る．Java 向けの代表的な記号実行ツールとして SPF（Symbolic Path Finder）[4], [5] がある．

しかし，SPF は業務アプリケーションにおいて使用される一部のデータ型や提供される外部ライブラリの一部のメソッド，業務アプリケーションフレームワークの機能に対応していない．そのため，Java で開発された業務アプリケーションに対しては SPF プログラムを解析することができず，テスト入力値を自動生成することができないという問題があった．

そこで本論文[*1]では，問題点とその解決策を具体化するために，まず SPF を業務アプリケーションに適用した場合の問題点を明らかにする．

次に問題点を解決するために，SPF が受理可能なプログラムへと変換する前処理を施し，SPF の改修なしに業務アプリケーションに対して記号実行を適用可能にしてテスト入力値を得る方法を提案する．また，前処理のために必要となるスタブの出力値を記号実行で自動生成する方法を提案する．

提案方法をテスト入力値自動生成システムとして実装し，実際の業務アプリケーションを対象とした評価実験により提案手法の有効性を確認する．

本論文で提議するリサーチクエスチョンは以下である．

RQ1. 提案する前処理によって業務アプリケーションに対する SPF の解析の成功率が改善されるか．

RQ2. 提案する前処理を導入したテスト入力値生成システムは実用的な時間内でテスト入力値を生成可能か．

RQ3. 提案する前処理を導入したテスト入力値生成システムによって有用なテスト入力値を生成可能か．

RQ4. 提案する前処理を導入したテスト入力値生成システムによってテストケース作成工数を削減できるか．

本論文の貢献を以下に示す．

- 業務アプリケーションのプログラムを既存記号実行ツールが受理可能なプログラムへと変換するための前処理手法の提案
- 提案手法を実装したテスト入力値自動生成システムの開発
- 実際の業務アプリケーションプログラムを対象とした評価実験

以下，2 章で背景を述べた後，3 章でアプローチを説明し，4 章，5 章で提案手法の詳細について述べる．6 章では提案手法を実装したシステムについて述べる．7 章，8 章，9 章では提案手法の有効性を検証するために行った実験の結果とその考察について述べる．10 章で関連研究について述べる．最後に，11 章で本論文の結論と今後の展望について述べる．

2. 背景

2.1 対象とするアプリケーションとテスト

本研究は業務アプリケーションを対象とする．本論文において業務アプリケーションとは，以下のような特徴を持つアプリケーションを想定する．

- Java EE（Java Platform, Enterprise Edition）[6] や Spring Framework [7] などの業務アプリケーションフレームワークを利用して開発されている．
- プレゼンテーション層，ビジネスロジック層，データアクセス層の 3 層構造で構築されており，ビジネスロジック層，データアクセス層は Java で開発されている．

また，本研究が対象とするテストは，上記業務アプリケーションのビジネスロジック層に対する単体テストである．特に，プログラムの分岐構造に着目したホワイトボックステストを対象とし，分岐カバレッジ 100%のテスト入力値とその期待値を作成する作業の効率化を目的とする．

2.2 記号実行

テスト入力値を自動生成する手法の 1 つとして，記号実行技術が知られている．記号実行は静的解析によってプログラムのすべての実行経路と実行経路を通過するための入力値の条件を導出し，SMT ソルバを用いて導出した条件を解くことで具体的なテスト入力値を得る手法である．静的解析による条件の導出と SMT ソルバによる処理が成功すれば，プログラムのすべての実行経路を通るテスト入力値を生成可能である．Java を対象とした記号実行ツールとして，米国 National Aeronautics and Space Administration（NASA）により開発された Java 向けモデル検査ツール JPF（Java Path Finder）[9] に記号実行機能を拡張した SPF（Symbolic Path Finder）[4] がある．

2.3 業務アプリケーションに対する記号実行適用の問題

以下に示す問題点により，SPF では Java で開発された業務アプリケーションのプログラムを解析できず，テスト入力値が自動生成できない．

(1) データ型の問題

SPF は int 型などのプリミティブ型と String 型の変数を記号実行に必要な記号変数にして解析できるようにしている．

ユーザが定義するオブジェクト型変数は，そのまま記号変数として解析できないが，フィールド変数を再帰的に探

*1 なお本論文は，著者らの国際会議 APSEC2018 でのポスター発表 [8] を発展させたものである．本論文では新たに 5 章に述べるスタブ出力値を自動生成する方法を提案し，7 章に述べる評価実験を実施した．4 章に述べるプログラムの前処理方法は [8] で提案したものと同等である．

© 2021 Information Processing Society of Japan

索してオブジェクト型変数をプリミティブ型や String 型変数の組合せに帰着し，記号変数と見なして解析する．また，List 型などのコレクション型変数については，そのものを記号変数として解析できないが，要素数を固定してオブジェクト型変数と同様の方法で要素を探索してプリミティブ型や String 型変数の組合せに帰着し，解析できるようにしている．

しかし，BigDecimal 型，Date 型などの Java が提供するオブジェクト型変数は，原則的に内部構造が隠蔽されているためこの方法が使えず，解析することができない．また，コレクション型以外でのジェネリクス利用については，型変数で指定するクラスへのアクセス方法が統一されておらず様々であるため，上記の方法による解析ができない．

(2)　アノテーションの問題

SPF は Java のアノテーションを用いた業務アプリケーションフレームワークの機能を対象としていない．このため，たとえば Java EE 7 の CDI（Contexts and Dependency Injection）による動的なオブジェクト取得のためのアノテーションを解析できず，アノテーションによって生成されるべきオブジェクトを生成できず，プログラムの動作を正しく解析できない．

(3)　解析不可能なメソッドの問題

SPF は以下の種類のメソッドを解析できない．

- Java の実行環境外にあるデータベース，ファイルやシステム変数へのアクセスメソッド
- parseInt(String s) などのデータ型を変換するメソッド
- ソースコードがない外部ライブラリのメソッド

3.　アプローチ

2.3 節に示した 3 つの問題点に対する対応処理は，業務アプリケーションのプログラムを SPF が受理可能なプログラムに変換する前処理として実装するアプローチとする．

対応処理は SPF を改修し直接実装するアプローチもありうる．しかし，その場合は既存の SPF のプログラムの十分な理解が必要であり，SPF のプログラムは大規模かつ複雑であるため，次の実装上の懸念が生じた．

- 対応処理のそれぞれを SPF のプログラムのどの部分に実装すべきかの判断が困難．
- 改修した場合の影響調査・テストが難しい．

これらを回避するために，本研究では前処理によるアプローチを採用する．

4.　業務アプリケーションプログラムの前処理

ここでは 2.3 節に示した業務アプリケーションに対する記号実行適用の (1)～(3) の問題点を解消し，業務アプリケーションに SPF の適用を可能にするためのプログラムの前処理方法を示す．

提案手法では，SPF による記号実行を実施する前に入力 Java プログラムに対して前処理を行い，前処理済みプログラムに対して SPF でテスト入力値を自動生成する（**図 1**）．

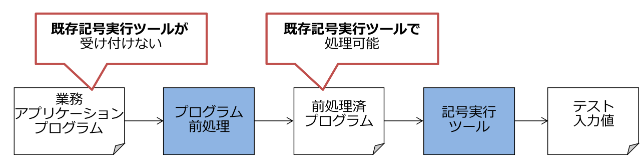


図 1　プログラム前処理を用いたテスト入力値生成の流れ
Fig. 1　Flow of test input value generation with preprocessing.

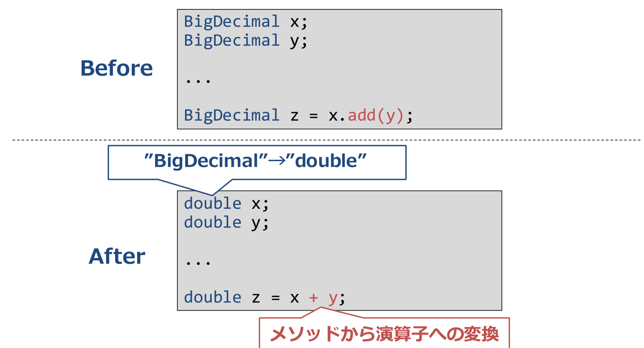


図 2　データ型の書き換えの例
Fig. 2　An example of rewriting data type.

提案手法の前処理は，(1) データ型の書き換え，(2) 動的呼び出しの書き換え，(3) スタブ化の 3 つの処理からなる．以下，各処理の詳細を説明する．

4.1　データ型の書き換え

プログラム内の既存記号実行ツールによる解析ができないデータ型を既存記号実行ツールで解析可能なプリミティブ型変数に書き換える．

たとえば，**図 2** に示すように，BigDecimal 型はプリミティブ型の double 型に書き換えるため，プログラム内のすべての BigDecimal 型の宣言を double 型の宣言に書き換える．同時に，該当するデータ型に関するメソッド呼び出しは，すべて同値なプリミティブ型変数の演算に書き換える．たとえば，BigDecimal 型変数 x，y に関して x と y の加算を行うメソッド呼び出し x.add(y) は double 型変数 x，y の + 演算子による加算 x+y に書き換える．

我々は，このようなデータ型の書き換えのルールを BigDecimal 型，Enum 型，Date 型，LocalDate 型，LocalTime 型，LocalDateTime 型の 6 種類のデータ型に対して定義した．

なお，コレクション型以外でのジェネリクス利用については，汎用的な書き換えルールの定義が困難であったため，本研究では対象外とした．

4.2　動的呼び出しの書き換え

プログラム内でアノテーションによって指定される動的呼び出しを静的呼び出しに書き換える．

© 2021 Information Processing Society of Japan

たとえば，Java EE 7 によるアプリケーション開発では，CDI による動的なオブジェクト取得が用いられる．これを，図 **3** に示すように，new 演算子による静的なオブジェクト生成命令に書き換える．プログラム内の CDI 適用箇所は@Inject などのアノテーションで示されているため，自動的に判定可能である．また，オブジェクトを生成する具体クラス名は CDI 機能の仕様に従い，アノテーションのパラメータやフレームワークの設定ファイルから特定できる．

そこで，このような書き換えのルールを Java EE 7 で利用される動的呼び出しのアノテーション@Inject，@EJB および Spring Framework2.5 以上で利用される動的呼び出しのアノテーション@Autowired に対して定義した．

4.3　スタブ化

プログラム内で呼び出される SPF で解析できないメソッドをスタブ化する．たとえばデータベースアクセスの場合は，図 **4** に示すように，データベースアクセスメソッドの呼び出しをスタブオブジェクトの生成命令に書き換え，さらにスタブオブジェクトに具体値を設定する命令を挿入する．これによりデータベースアクセスメソッドがスタブ化される．

同様の書き換え方法で，データアクセスメソッド以外にも，プログラム内のデータ型の変換メソッドとソースコードのない外部ライブラリのメソッドに対してもスタブ化をできるようにする．

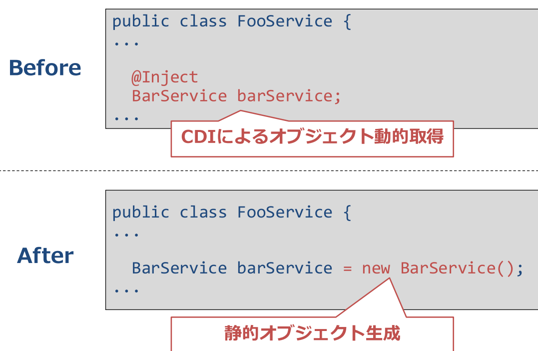


図 **3**　動的呼び出しの書き換えの例

Fig. 3　An example of rewriting dynamic invocation.

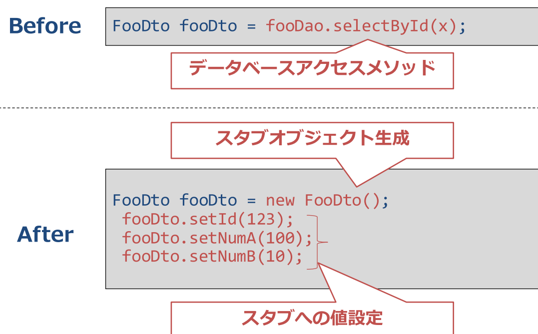


図 **4**　スタブ化の例

Fig. 4　An example of stubbing.

データベースアクセスメソッドの場合は，次に示す方法によってプログラム内でスタブ化のために書き換えが必要な個所を自動的に特定できるようにする．

業務アプリケーションの開発では一般に DAO（Data Access Object）という SQL によるデータベースへのテーブルアクセス処理を隠ぺいしたクラスとそのメソッドを作成しデータアクセス層に配置される．

DAO のクラス名は，たとえばテーブル “foo” に対する DAO のクラス名は “FooDao” とするなど，プロジェクトで規定した一定の命名規則に従って命名される．このような命名規則を正規表現で表すことにより，パターンマッチを用いて DAO を特定することができ，書き換え対象となるデータベースアクセスメソッドを自動的に判定する．

データ型の変換メソッドの場合は，Java の標準ライブラリのメソッドが用いられるため，スタブ化が必要なメソッドの名前が既知であり，書き換えが必要な個所を自動的に特定可能である．たとえば，標準ライブラリ Integer クラスにはデータ型の変換メソッド parseInt（String s）があるため，これらを事前に定義しておき特定できるようにする．

外部ライブラリのメソッドの場合は，スタブ化が必要なメソッドの名前が明らかでない．そこで，あらかじめスタブ化が必要なメソッドの情報を記載したスタブ設定ファイルを用意し，スタブ設定ファイルに記載されたメソッド名に従って，書き換えが必要な個所を自動的に特定するようにする．なお，スタブ設定ファイルにはメソッド名のほか，スタブ化の際の出力値も記載する．

5.　スタブ出力値の自動生成

5.1　スタブ出力値作成の課題

4.3 節で説明したスタブ化において，スタブに設定する出力値は，テスト実施時に想定されるデータベース初期状態や，スタブ化するメソッド呼び出しの後続の分岐を網羅できるように出力値のパターンを設定する必要がある．

たとえば，図 **5** のプログラムについて，データベースアクセスメソッドの呼び出し fooDao.selectById(x) をスタブ化することを考える．ここで，後続に fooDto.

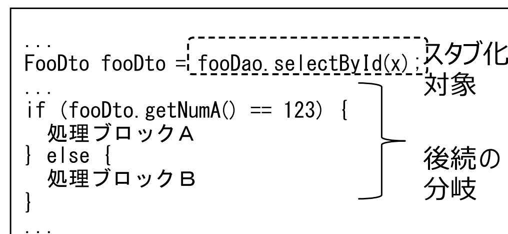


図 **5**　後続の分岐が問題となるスタブ化の例

Fig. 5　An example of stubbing where subsequent branches are the problem.

© 2021 Information Processing Society of Japan

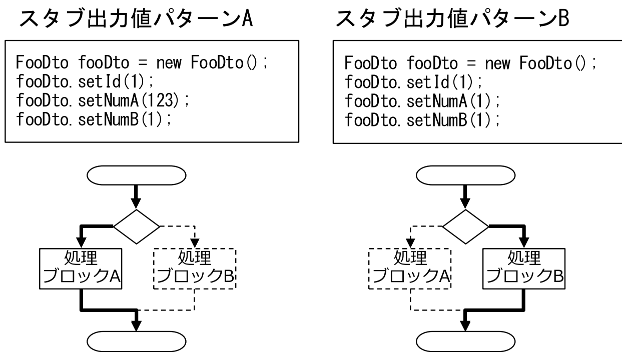


図 6　後続の分岐を考慮したスタブ出力値のパターン

Fig. 6　Pattern of stub output values considering subsequent branches.

getNumA()==123 というスタブ化対象メソッドの戻り値に関する条件の分岐が存在することに注意する．処理ブロック A，処理ブロック B の両方に対して記号実行による解析を行うには，図 **6** のようにスタブの出力値のパターンを分岐条件が真となる場合と偽となる場合を両方作成する必要がある．

すべてのスタブ化対象メソッドに対して後続の分岐を考慮した出力値を人手で作成するのは，人手によるテスト入力値の作成作業と同様に工数がかかる．

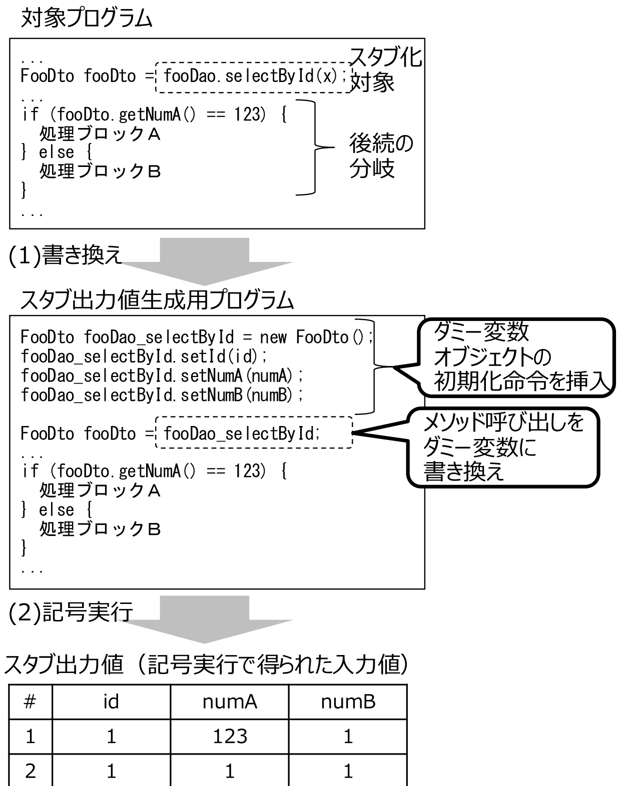
5.2　後続分岐を網羅するスタブ出力値の自動生成方法

前節で説明した課題を解決するために，記号実行を応用して後続の分岐を網羅するスタブ出力値を自動生成する方法を提案する．

提案手法では，スタブ化対象メソッド呼び出しのダミー変数への書き換えと，ダミー変数を入力値とした記号実行を行う．これにより，実行経路を網羅する入力値を生成できるという記号実行の特徴を利用して，後続の分岐を網羅するスタブ出力値を得ることができる．

以下に具体的な流れを示す．(1) プログラムのスタブ化対象メソッドの呼び出しをダミー変数に書き換えたプログラムを生成する（以降これをスタブ出力値生成用プログラムと呼ぶ）．(2) スタブ出力値生成用プログラムに対し，ダミー変数を入力値として記号実行による入力値生成を実行する．(3) 得られた入力値をスタブ出力値とする．

図 7 は図 5 のプログラムを対象プログラムとして上記の流れを実施した例を示す図である．(1) の書き換えでは，スタブ化対象メソッドの呼び出し fooDao.selectedById(x) をダミー変数 fooDao_selectById に書き換える．さらに，スタブ化対象メソッド fooDao.selectedById の戻り値のデータ型がオブジェクト型であるため，書き換え箇所の直前にダミー変数のオブジェクト fooDao_selectById の初期化命令（オブジェクトの生成命令，フィールドへの値設定命令）を挿入する．(2) では (1) で作られたスタブ出力値生成用プログラムに対してダミー変数を入力値として記号実行



#	id	numA	numB
1	1	123	1
2	1	1	1

図 7　提案手法の流れの具体例

Fig. 7　An example of the flow of the proposed method.

する．

この例ではダミー変数がオブジェクト型（FooDto）であるため，実際にはプリミティブ型フィールド変数（id，numA，numB）を入力値とする必要がある．SPF においては，記号実行時にどの変数を入力値とするかをメソッド名によって指定することができ，指定した任意のメソッドの引数を入力値とすることができる．この例においては FooDto のフィールドのセッターメソッドである setId，setNumA，setNumB を指定することにより，ダミー変数オブジェクトのフィールド変数 id，numA，numB を入力値として記号実行を行うことができる．記号実行の結果，たとえば図 7 下のような入力値の組合せが得られる．これはスタブ化箇所の後続の分岐を網羅するためのスタブ出力値となっている．

6.　テスト入力値生成システム

提案手法をテスト入力値生成システムとして実装した（図 **8**）．本システムはテスト対象の Java プログラムを入力とし，入力された Java プログラムに対して前処理を実施する．前処理済み Java プログラムに対して SPF による記号実行を適用し，テスト入力値を生成して出力する．

6.1　前処理の実現方法

プログラムの前処理は，Java プログラムを AST（Abstract Syntax Tree）へ変換し，AST に対して 3 章で示し

© 2021 Information Processing Society of Japan

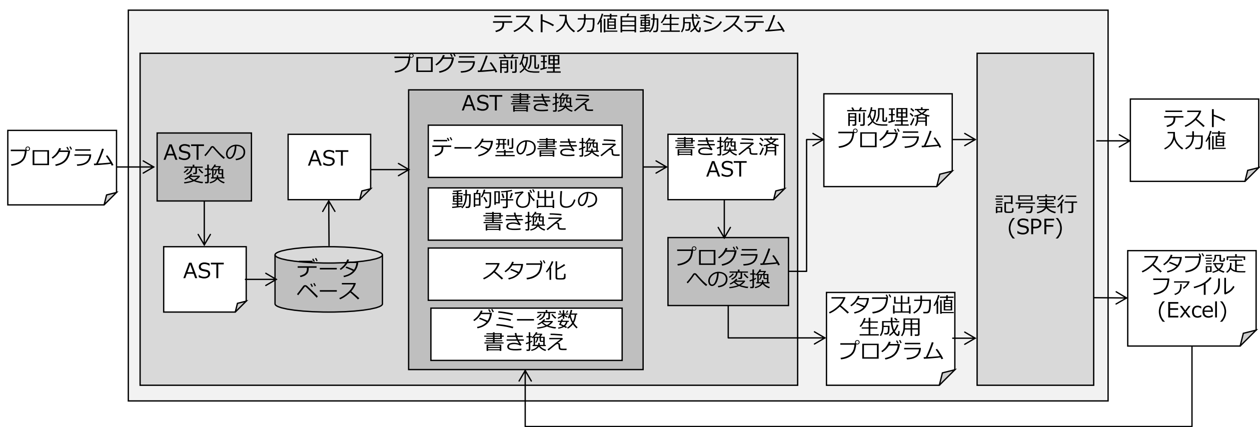


図 8　テスト入力値生成システムの処理フロー
Fig. 8　Processing flow of the test input value generation system.

```
SELECT * FROM TypeDeclaration,FieldDeclaration
  WHERE TypeDeclaration.Name = Foo
  AND FieldDeclaration.TypeDeclarationID = TypeDeclaration.TypeDeclarationID;
```

図 9　クエリの例
Fig. 9　An example of the query.

た書き換えを実施し，書き換えた AST を再度プログラムに変換することにより実現する．Java プログラムの AST への変換は，Eclipse JDT（Java Development Tools）[10] が提供する Java 用 AST パーサのライブラリ ASTParser を利用した．

あるプログラムを書き換えるために，書き換え対象プログラムが依存する別のプログラムの情報が必要になることがある．たとえば，メソッドをスタブ化する際の戻り値のクラスのフィールド情報などである．

このような情報を取得する方法として，たとえば依存するプログラムを逐次 AST に変換してメモリ上で AST を探索し必要な情報を取得するなど，様々な方法が考えられる．本システムでは，1 つの実装形態として，対象業務アプリケーションのすべてのプログラムを AST に変換してデータベースに保存した後に，AST の書き換え時に必要な情報をデータベースの検索機能を利用して取得する方法を採用した．

ASTParser では AST のノードの種類別にクラスが定義されている．これらのクラスと 1 対 1 となるようにデータベースのテーブルを定義し，ノードの親子関係をテーブル間の関係で表すことにより AST をデータベースに保存できるようにした．

これにより，たとえば “Foo” という名前のクラスのフィールドの情報が欲しい場合は**図 9** のクエリを実行すれば，“Foo” クラスの定義文に対応する TypeDeclaration ノードとそのフィールド定義文に対応する子ノードの FieldDeclaration ノードをデータベースから直接取得できる．

6.2　スタブ出力値の生成とスタブ化

スタブ化対象のメソッドの情報を記載した独自フォーマットで定義したスタブ設定ファイルを入力すると，本システムは 3 章と 4 章で示した方法に従い，前処理によってスタブ出力値生成用プログラムを生成し，それを記号実行してスタブの出力値を自動生成する．このスタブ出力値生成のための前処理と記号実行は，テスト入力値生成のための前処理と記号実行とは別に，事前に実施される．自動生成されたスタブ出力値はスタブ設定ファイルに自動的に補完され，テスト入力値生成のための前処理と記号実行を実施する際に，スタブ設定ファイルに記載されたスタブ出力値に基づいてスタブ化が行われる．

なお，スタブ設定ファイルはユーザが手作業により任意の出力値を追加することも可能である．

7.　評価実験

7.1　目的

提案手法の有効性を評価するため，以下のリサーチクエスチョンを提議し，実験により検証する．

(1)　RQ1：提案する前処理によって業務アプリケーションに対する SPF の解析の成功率が改善されるか

提案する前処理の有効性を示すため，SPF が解析できないプログラムを SPF が解析できるプログラムに変換できるかどうかを検証する必要がある．実際の業務アプリケーションに対して前処理なしのプログラムと前処理ありのプログラムで SPF を実行し，解析成功率を比較する．

ここでの解析成功率とは，SPF で解析した全数のうち，解析成功した割合を表す．解析成功は SPF がエラーなしに最後まで完了した場合と定義し，解析失敗は解析途中で

© 2021 Information Processing Society of Japan

何らかのエラーが発生した場合と定義する．

(2) RQ2：提案する前処理を導入したテスト入力値生成システムは実用的な時間内でテスト入力値を生成可能か

提案手法はプログラムの前処理を行うため，前処理のオーバヘッドによりシステムの処理時間が増加し，実用的な時間でテスト入力値の生成ができない可能性がある．開発したテスト入力値生成システムの処理時間を計測することにより，実際の業務アプリケーションに対して実用的な時間内でテスト入力値が生成できるかどうかを検証する．

(3) RQ3：提案する前処理を導入したテスト入力値生成システムによって有用なテスト入力値を生成可能か

提案する前処理ではプログラムを書き換えるため，その影響により，記号実行で生成したテスト入力値が元のプログラムには使えないものになってしまうことや，元のプログラムに対する網羅性が低下してしまうことが懸念される．生成したテスト入力値を元のプログラムに入力し，手作業で作成したテスト入力値とカバレッジを比較することにより，上記の問題がなく有用なテスト入力値が生成可能かどうかを検証する．

(4) RQ4：提案する前処理を導入したテスト入力値生成システムによってテストケース作成工数を削減できるか

本研究の目的はテストケースの作成作業を効率化することである．開発したテスト入力値生成システムによりテスト入力値の作成作業は自動化されるが，スタブ化対象の選定など，システム実行のために必要となる追加作業の工数が大きい場合，全体としては工数の削減につながらない可能性がある．また，テストケース作成のためにはテスト入力値に対する期待値を手作業で作成する必要があるが，テスト入力値をシステムで自動生成した場合，期待値の作成難易度が手作業でテスト入力値を作成した場合と比べて高くなる可能性があり，工数が増加してしまうことが懸念される．手作業の場合とシステムを利用した場合のテストケース作成の作業時間を比較し，システム利用によりテストケース作成工数が削減できるかどうかを検証する．

7.2 実験 1

7.2.1 実験内容

RQ1 を明らかにするため，Java で開発されたある実際の業務アプリケーション 3 つを対象として，提案手法の前処理を適用しない場合と適用する場合の 2 通りで SPF による解析を実施し，解析に成功したメソッド数を比較した．

対象とした 3 つの業務アプリケーションの概要を**表 1** に示す．3 つの業務アプリケーションのうちアプリ X とアプリ Z は Java EE 7 で開発されており，アプリ Y は Spring Framework で開発されている．

2.1 節で述べたように，本研究はビジネスロジック層の単体テストが対象であるため，実験の対象メソッドとして 3 つの業務アプリケーションのビジネスロジック層のクラスから 50 個ずつ，合計 150 個のメソッドを無作為に選出した．

表 1 実験対象の業務アプリケーション
Table 1 Target applications of experiment.

アプリケーション名	ステップ数	クラスの数	業務アプリケーションフレームワーク
アプリ X	約 14 万	1862	Java EE 7
アプリ Y	約 118 万	8173	Spring Framework
アプリ Z	約 46 万	1083	Java EE 7

表 2 SPF による解析の実験結果
Table 2 Experimental results of analysis by SPF.

アプリケーション名	前処理なし		前処理あり	
	成功数	成功率	成功数	成功率
アプリ X	21	42%	50	100%
アプリ Y	23	46%	50	100%
アプリ Z	18	36%	48	96%
全体	62	41%	148	99%

7.2.2 実験結果

実験の結果を**表 2** に示す．

前処理なしの場合は，3 つの業務アプリケーションから選出した全 150 個のメソッドのうち解析に成功したのは 62 個であった．解析に失敗した 88 個のメソッドの主な失敗の原因は以下のとおりであった．

- BigDecimal 型変数を記号変数として解析できない．
- Date 型変数を記号変数として解析できない．
- File 型変数を記号変数として解析できない．
- CDI 使用箇所でオブジェクトが生成できない．
- システムプロパティへのアクセスメソッドが解析できない．
- データベースアクセスメソッドが解析できない．
- 外部ライブラリのメソッドが解析できない．

前処理ありの場合は，3 つの業務アプリケーションのうち，アプリ X，アプリ Y について，選出した 50 個すべてのメソッドの解析に成功した．アプリ Z については，48 個のメソッドの解析に成功し，2 個のメソッドの解析に失敗した．解析に失敗した 2 個のメソッドの失敗の原因は，メソッドの入力値が SPF で解析できない File 型変数となっており，本システムで File 型変数の書き換え処理を実装していないためであった．

7.3 実験 2

7.3.1 実験内容

実験 1 で扱ったアプリ X のビジネスロジック層のクラスから 7 個のメソッドを選出し，選出したメソッドに対して被験者がシステムを利用する場合と利用しない場合の 2 通りでテストケースを作成した．ここで，テストケースと

© 2021 Information Processing Society of Japan

表 3 実験対象のメソッド
Table 3 Target methods of experiment.

メソッド名	ステップ数	IF 文の数	フィールド数	
			入力値	期待値
メソッド A	157	5	463	261
メソッド B	103	2	457	98
メソッド C	14	1	7	14
メソッド D	47	3	2	33
メソッド E	52	1	16	92
メソッド F	39	3	2	37
メソッド G	9	1	20	15

はテスト入力値とテスト入力値に対する期待値の組である．テストケースを作成する対象の 7 個のメソッドの概要を表 3 に示す．

表 3 中の「ステップ数」は各メソッド内の空行とコメント行を除いたソースコードの行数を，「IF 文の数」は各メソッド内にある IF 文の数を表す．

「フィールド数」の「入力値」はメソッドの引数となるプリミティブ型変数の数とオブジェクト型変数に含まれるフィールドの数の合計数を，「期待値」はメソッドの最後に戻り値として返されるオブジェクト型変数に含まれるフィールドの数の合計値を表す．

本実験における被験者は，数年間の Java による開発経験があり Java について習熟している．また，Java EE 7 の仕様についても理解しており，実験対象の業務アプリケーションのソースコードを問題なく読解できる．実験対象の業務アプリケーションについての事前知識は持っていない．

RQ2 を明らかにするため，システムを利用してテスト入力値を生成した際のシステムの処理時間を計測した．

RQ3，RQ4 を明らかにするため，システムを利用する場合と利用しない場合でテストケースの数，テストケースのカバレッジ，テストケース作成にかかる被験者の作業時間を計測して比較した．

システムの処理時間の計測では，以下の処理について実行時間を計測した．

- プログラムの AST への変換と DB への格納
- スタブ出力値の生成（前処理 + SPF 実行）
- テスト入力値の生成（前処理 + SPF 実行）

被検者の作業時間の計測では，以下の作業にかかった時間を計測した．

- システム利用なし
  - テスト入力値の作成
  - テスト入力値に対する期待値の作成
- システム利用あり
  - スタブ化対象の選定
  - システム入力
  - テスト入力値に対する期待値の作成

システム利用なしにおけるテスト入力値の作成は，被験者がソースコードをもとにして，分岐カバレッジが 100%となるようにテスト入力値を作成した．ここで，カバレッジは対象メソッドのみを範囲として測定する．対象メソッドは内部で他のメソッドを呼び出している場合もあるが，その場合は呼び出し先のメソッド内の分岐はカバレッジの測定範囲としない．

システム利用ありにおけるスタブ化対象の選定とは，テスト対象メソッドのプログラムを目視で確認し，テスト対象メソッドが呼び出しているメソッドからスタブ化が必要なものを選定し，スタブ設定ファイルに記入する作業を指す．システム利用ありにおけるシステム入力とは，テスト対象のプログラムやスタブ設定ファイルなど，必要な入力物をシステムに入力してシステムを実行する作業を指す．

表 4 システムの処理時間の計測結果
Table 4 Measurement result of system processing time.

システム処理	処理時間(分)						
	メソッド A	メソッド B	メソッド C	メソッド D	メソッド E	メソッド F	メソッド G
プログラムの AST への変換と DB への格納	5						
スタブ出力値生成（前処理 ＋SPF 実行）	4	3	0 (※)	3	3	3	4
テスト入力値生成（前処理 ＋SPF 実行）	12	16	12	18	12	16	12
合計	21	24	17	26	20	24	21

※スタブ化が必要なメソッドが無く処理を実施しなかった

7.3.2 実験結果

開発したシステムを利用してテスト入力値を生成した際のシステムの処理時間の計測結果は表 4 のとおりであった．

プログラムの AST への変換と DB への格納処理は，対象とした 7 個のメソッドの解析のために前処理が必要となる 99 個のクラスに対して実施した．この処理は 7 個のメソッドに共通の処理であるため，処理時間はすべて同じ 5 分となった．

システム利用なし/ありそれぞれの作成されたテストケース数，命令カバレッジ，分岐カバレッジは表 5 のとおりであった．

システム利用なし/ありそれぞれの各作業別の作業時間の計測結果は表 6 のとおりであった．

システム利用なし/ありそれぞれの作業時間の合計と，作業時間の削減率は表 7 のとおりであった．

© 2021 Information Processing Society of Japan

表 5 テストケース数とカバレッジ
Table 5 Number of test cases and coverage.

項目	システム利用	作業時間(分)						
		メソッドA	メソッドB	メソッドC	メソッドD	メソッドE	メソッドF	メソッドG
テストケース数	あり	5	3	2	12	4	12	3
	なし	9	5	2	9	2	9	2
命令カバレッジ	あり	100%	100%	100%	92%	100%	100%	100%
	なし	100%	100%	100%	100%	100%	100%	100%
分岐カバレッジ	あり	100%	100%	100%	100%	100%	100%	100%
	なし	100%	100%	100%	100%	100%	100%	100%

表 6 作業別の作業時間の計測結果
Table 6 Measurement result of work time according to work.

システム利用	作業	作業時間(分)						
		メソッドA	メソッドB	メソッドC	メソッドD	メソッドE	メソッドF	メソッドG
なし	入力値作成	89	49	50	51	26	43	18
	期待値作成	70	52	4	24	15	21	16
	合計	159	101	54	75	41	64	34
あり	スタブ化対象選定	10	5	3	8	6	8	5
	システム入力	2	2	2	2	2	2	2
	期待値作成	69	44	15	19	17	21	14
	合計	81	51	20	29	25	31	21

表 7 作業時間の合計と削減率
Table 7 Total work time and reduction rate.

システム利用	作業時間の合計 (分)							
	メソッドA	メソッドB	メソッドC	メソッドD	メソッドE	メソッドF	メソッドG	全体
なし	159	101	54	75	41	64	34	528
あり	81	51	20	29	25	31	21	258
削減率	-49%	-50%	-63%	-61%	-39%	-52%	-38%	-51%

8. 考察

8.1 RQ1：提案する前処理によって業務アプリケーションに対する SPF の解析の成功率が改善されるか

表 2 の結果から，全体の解析成功率が前処理なしの場合は 41%だったのに対し，前処理ありの場合は 99%に向上した．したがって，提案する前処理によって業務アプリケーションに対する SPF の解析の成功率が改善されると考える．

また，3 つの業務アプリケーションすべてについて，前処理実施後は解析成功率が 90%以上となった．このことから，SPF で業務アプリケーションのプログラムを解析できない原因の大部分を 2.3 節であげた問題で網羅できていると推測でき，提案手法の有用性は高いと考える．

実験によって前処理を実施しても解析できない例として File 型変数がメソッドの入力値になっているケースが見つかった．File 型変数は数値と対応付けることができないため，プリミティブ型に書き換えることができず，提案する前処理で対処することができない．このような Java が提供するオブジェクト型変数で数値と対応付けることができないものは提案手法では本質的に対応できない例であると考える．

上記の例や 4.1 節で述べたジェネリクス型変数は提案手法に限界があることを示している．これらへの対処は今後の課題とする．

8.2 RQ2：提案する前処理を導入したテスト入力値生成システムは実用的な時間内でテスト入力値を生成可能か

表 4 の結果から，前処理を含めた一連の処理の合計時間は 1 メソッドあたり最大で 26 分，平均 22 分であった．これは，テスト入力値の生成を夜間のバッチ処理で行うなどの運用を前提にすれば，十分に実用的な処理時間である．したがって，懸念していた前処理のオーバヘッドは許容範囲であり大きな問題にならないと考える．

ただし，対象メソッド内の分岐の数が多い場合，パス爆発により SPF の解析時間が増加し，実用的な時間内でテスト入力値を生成できない可能性がある．今回の実験においては対象としたメソッド内の IF 文が最大で 5 個と少ないためにこの問題は発生しなかった．提案する前処理はパス爆発の問題に対処できないため，今後の課題とする．

8.3 RQ3：提案する前処理を導入したテスト入力値生成システムによって有用なテスト入力値を生成可能か

システムにより生成されたテスト入力値はすべてテストにおいてそのまま利用でき，型や数値の範囲などの問題で実行できなくなるなどのケースはなかった．

表 5 の結果から，システムを利用して作成したテストケースのカバレッジについては，メソッド D 以外の 6 つのメソッドは命令カバレッジ，分岐カバレッジともに手作業と同等の 100%を達成できた．メソッド D のみ命令カバレッジが 92%となっており，100%を達成できなかった．これは，SPF は if 文の条件判定などにより明示的に例外を

© 2021 Information Processing Society of Japan

throw する以外の場合において例外処理を通過する入力値を生成できないために，例外処理の catch ブロックが網羅できていないことが原因であった．システムを利用して作成したテストケースの数については，手作業と必ずしも同じとはならなかったが，極端に多すぎる場合や少なすぎる場合などは見られなかったため，問題ないと考える．

以上より，例外処理への対応が今後の課題として残るものの，それ以外については高い網羅性を達成できており，提案する前処理を導入したテスト入力値生成システムにより十分に有用なテスト入力値が生成できると考える．

8.4 RQ4：提案する前処理を導入したテスト入力値生成システムによってテストケース作成工数を削減できるか

表 6 の結果から，システムを利用しない場合のテスト入力値作成の作業時間は最低で 18 分，最高で 89 分であり，平均 47 分であった．一方で，システムを利用する場合のスタブ化対象選定の作業時間はたかだか 10 分程度，システム入力の作業時間は 2 分程度であり，これはシステムを利用しない場合のテスト入力値作成の作業時間と比べて小さい．したがって，システム利用のために必要となる追加作業の工数は大きな問題にならないと考える．

また，テスト入力値に対する期待値の作成の作業時間についてはシステムを利用しない場合と利用する場合で大きな差はなかった．このことから，テスト入力値を自動生成することによって期待値の作成難易度が大きく難化するなどの問題はないと考えられる．

表 7 のとおり，テストケース作成にかかる一連の作業時間に関しては，7 個すべてのメソッドについて，システム利用によりシステムを利用しない場合と比べて作業時間が削減され，全体では 51%の工数削減効果を確認した．

以上より，提案する前処理を導入したテスト入力値生成システムによりテストケース作成工数を削減できると考える．

なお，実験において，対象メソッドの入力値フィールド数，ステップ数，IF 文の数とテスト入力値作成にかかる作業時間が相関しない場合が見られた．たとえば，メソッド C は入力値フィールド数，ステップ数，IF 文の数のすべてにおいてメソッド B よりも少ないが，入力値作成の時間はメソッド C の方が大きくなっている．

入力値フィールド数が入力値作成時間に相関しなかった理由は，分岐カバレッジを満たすように入力値を作成するにあたり，主に分岐に関係する部分の入力値フィールドのみを考慮すれば十分であったためであると考えられる．

ステップ数，IF 文の数が入力値作成時間に相関しなかった理由は，対象メソッドが内部で他のメソッドを呼び出しており，実行結果が対象メソッド内の分岐に関係する場合に，呼び出し先のメソッドの処理についても調査する必要があったためであると考える．実際，メソッド C の内部で他のメソッドを呼び出しており，その結果がメソッド C 内の分岐に関係していた．

9. 妥当性の脅威

実験 2 において，被験者 1 名が同じ業務アプリケーションのメソッドを対象に，システムの利用なし，システム利用ありの順番で実験を行った．そのため，期待値の作成において被験者の慣れや学習が影響し，システム利用ありの場合の期待値の作成時間が実際よりも短くなった可能性がある．これについては被験者を増やし，システム利用のなし/ありの順番を変えた実験を通じて確かめる必要があり，今後の課題としたい．

本実験で対象とした業務アプリケーションは特定の限られた分野のソフトウェアであるため，一般の業務アプリケーションのプログラムで使われるデータ型やメソッドを網羅していない．そのため，他の業務アプリケーションを対象とした場合，未知の問題が見つかり，実験結果が異なる可能性がある．本実験によって，前処理による SPF の解析成功率の改善を実証することができたが，この結果を一般化するためには，より幅広い分野の業務アプリケーションを対象とした実験が必要である．

10. 関連研究

業務アプリケーションのプログラムに対して記号実行を適用してテスト入力値を生成する取り組みが様々にされている [11], [12], [13], [14].

Monpratarnchai ら [11], [12] と谷田ら [13] は記号実行に必要となるスタブとドライバを自動生成する．これらの研究は記号実行を応用してスタブの値を自動生成する点で提案手法と共通する．しかし，本論文で提案している記号実行ツールで解析できない型の変数や動的呼び出しへの対応がない．

Ghosh らは SPF をベースとして業務アプリケーション向けに拡張したテストスイート生成ツール JST を提案している [14]．この研究では，既存の SPF が解析できないという問題に対し，独自のソルバや Java が提供するクラスのメソッドを処理するコンポーネントを SPF の拡張機能として実装することで対処している．提案手法は，既存記号実行ツールの拡張や独自解析機能の実装ではなく，解析対象プログラムに対する前処理により問題の解決を試みる点でこれらの研究と異なる．

記号実行で解析ができないデータ型や処理の問題を解消する別のアプローチとして，記号実行と他のテスト入力値生成手法を組み合わせる方法も研究されている．Baars ら [15]，Gotlieb [16]，Inkumsah ら [17] は記号実行とサーチベーステスト生成 [18], [19] を組み合わせている．記号実行による解析が可能な部分に対しては記号実行による解

情報処理学会論文誌 Vol.62 No.4 1039–1050 (Apr. 2021)

析を実施し，不可能な部分に対してはサーチベーステスト生成を用いることにより，網羅性の高いテスト入力値の生成を実現している．これらの方法は効果的であるが，ツールを実装するために記号実行とそれに組み合わせるテスト入力値生成手法の専門知識が必要となる．一方，提案手法は記号実行ツールで解析可能なプログラムの知識があればツールを実装できるというメリットがある．

11. おわりに

本論文では，既存記号実行ツール SPF を改修することなく業務アプリケーションに対して記号実行を適用可能にするため，業務アプリケーションのプログラムを SPF が受理可能なプログラムに変換するプログラム前処理を提案した．

そして，提案する前処理を提案する前処理を導入したテスト入力値生成システムを開発し，実際の業務アプリケーションを対象とした評価実験により，提案手法の有効性を示した．

既存の記号実行ツールとして，本論文では SPF を選定したが，提案した前処理は記号実行ツールに依存するものではなく，その他の記号実行ツールと連携する場合においても使用可能であると考える．

今後はテスト入力値生成の対象を単体テストから結合テストに広げていく．結合テストを対象にする場合，メソッドの多段呼び出しにより分岐が増えるため，パス爆発により記号実行の計算量が大きくなるという問題がある [20]．この問題の解決のため，著者らはプログラムの前処理による記号実行の計算量削減手法の開発に取り組んでいる [21]．この手法をテスト入力値生成システムの機能として実装し，業務アプリケーションの結合テストへの適用可能性を評価する予定である．

参考文献

[1] Orso, A. and Rothermel, G.: Software testing: A research travelogue (2000-2014), *Proc. Future of Software Engineering*, pp.117–132 (2014).

[2] King, J.C.: Symbolic execution and program testing, *Comm. ACM*, Vol.19, No.7, pp.385–394 (1976).

[3] de Moura, L., Dutertre, B. and Shankar, N.: A Tutorial on Satisfiability Modulo Theories, *Computer Aided Verification, CAV 2007*, Damm, W. and Hermanns, H. (Eds.), *Lecture Notes in Computer Science*, Vol.4590, Springer, Berlin, Heidelberg (2007).

[4] Pasareanu, C.S. and Rungta, N.: Symbolic PathFinder: Symbolic Execution of Java Bytecode, *Proc. IEEE/ACM International Conference on Automated Software Engineering*, pp.179–180 (2010).

[5] Symbolic PathFinder, available from ⟨https://github.com/SymbolicPathFinder/jpf-symbc⟩.

[6] Java EE, available from ⟨http://www.oracle.com/technetwork/java/javaee/⟩.

[7] Spring Framework, available from ⟨https://spring.io/projects/spring-framework⟩.

[8] Ohbayashi, H., Kanuka, H. and Okamoto, C.: A Preprocessing Method of Test Input Generation by Symbolic Execution for Enterprise Application, *2018 25th Asia-Pacific Software Engineering Conference* (*APSEC*), Nara, Japan, pp.717–718 (2018).

[9] Java PathFinder, available from ⟨https://ti.arc.nasa.gov/tech/rse/vandv/jpf/⟩.

[10] Eclipse JDT, available from ⟨https://www.eclipse.org/jdt/⟩.

[11] Monpratarnchai, S., Fujiwara, S., Katayama, A. and Uehara, T.: An Automated Testing Tool for Java Application Using Symbolic Execution Based Test Case Generation, *2013 20th Asia-Pacific Software Engineering Conference* (*APSEC*), Bangkok, pp.93–98 (2013).

[12] 片山朝子，上原忠弘，藤原翔一朗，宗像一樹，Supasit, T.，徳本 晋，前田芳晴：業務システムを対象としたシンボリック実行による検証試行，情報処理学会ソフトウェアエンジニアリングシンポジウム 2013 論文集，pp.1–6 (2013).

[13] 谷田英生，Li, G.，Ghosh, I.，上原忠弘：記号実行エンジンを用いた JavaScript プログラムの単体テスト自動生成実行，情報処理学会ソフトウェアエンジニアリングシンポジウム 2014 論文集，pp.158–163 (2014).

[14] Ghosh, I., Shafiei, N., Li, G. and Chiang, W.: JST: An automatic test generation tool for industrial Java applications with strings, *2013 35th International Conference on Software Engineering* (*ICSE*), San Francisco, CA, pp.992–1001 (2013).

[15] Baars, A. et al.: Symbolic search-based testing, *2011 26th IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2011*), Lawrence, KS, pp.53–62 (2011).

[16] Gotlieb, A.: Euclide: A Constraint-Based Testing Framework for Critical C Programs, *2009 International Conference on Software Testing Verification and Validation*, Denver, CO, pp.151–160 (2009).

[17] Inkumsah, K. and Xie, T.: Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution, *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, L'Aquila, pp.297–306 (2008).

[18] McMinn, P.: Search-Based Software Testing: Past, Present and Future, *Proc. 2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops*, pp.153–163 (2011).

[19] Fraser, G. and Arcuri, A.: EvoSuite: Automatic test suite generation for object-oriented software, *Proc. 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp.416–419 (2011).

[20] Cadar, C. and Sen, K.: Symbolic Execution for Software Testing: Three Decades Later, *Comm. ACM*, Vol.56, No.2, pp.82–90 (2013).

[21] 大林浩気，鹿糠秀行，鈴木哲也，岡本周之：分岐条件に基づく関数呼出し簡略化による記号実行のパス爆発抑制手法，日本ソフトウェア科学会 FOSE 2017 論文集，Vol.2017, pp.235–236 (2016).

Java は，Oracle Corporation およびその子会社，関連会社の米国およびその他の国における登録商標です．Spring Framework は，米国 Pivotal Software, Inc. の米国またはその他の国における商標または登録商標です．Eclipse は米国およびその他の国における Eclipse Foundation, Inc. の商標もしくは登録商標です．文中の社名，商品名などは各社の商標または登録商標である場合があります．なお，本文および図表中では，™，®マークは明記しておりません．

© 2021 Information Processing Society of Japan

大林 浩気

2005 年九州大学理学部数学科卒業．2011 年九州大学大学院数理学府数理学専攻博士課程修了．同年株式会社日立製作所入社．現在，研究開発グループ研究員．博士（機能数理学）．ソフトウェア工学に関する研究に従事．

鹿糠 秀行 （正会員）

2001 年武蔵工業大学工学部電子情報工学科卒業．2003 年東京工業大学大学院総合理工学研究科知能システム科学専攻修士課程修了．同年株式会社日立製作所入社．現在，研究開発グループ主任研究員．ソフトウェア工学に関する研究に従事．

© 2021 Information Processing Society of Japan