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

谷田 英生[1,a)]　Guodong Li[2,b)]　Indradeep Ghosh[2,c)]　上原 忠弘[1,d)]

概要：JavaScript コードは，対話性の高い Web アプリケーションやモバイルアプリの実現のために，今後さらに多用される傾向にあり，そのテストが重要となる．一方，JavaScript に対するテスト効率化技術は十分に整備されていない．そこで本稿では，JavaScript プログラムに対して網羅性の高い単体テストデータを自動生成する手法を提案する．手法は，JavaScript 記号実行エンジンと，記号実行エンジンの解析対象外とする記述を自動でスタブ化するスタブ・ドライバ生成エンジンにより構成される．手法は，実ブラウザ上での対象プログラム実行に使用可能な，単体テストデータを生成可能である．手法を現場の資産へ適用することにより，高カバレッジを実現する単体テストデータを自動生成可能なことが確認された．

キーワード：JavaScript, テスト生成, 記号実行, シンボリック実行, スタブ生成

1. はじめに

信頼性の高いソフトウェアを実現するためには，充分なテストを行う必要がある．一方，従来の開発現場では，手動で作成したテストケースを用いてテストを行っており，テストケースの作成に工数が必要であった．また，作成したテストケースで実現可能なコードカバレッジも充分でなくソフトウェアの品質が担保出来ない問題が存在した．

これらの問題を解決する技術として注目されているのが，形式的検証技術やそれを応用したテスト技術である．そのうち，記号実行に基づくテスト入力自動生成技術では，ランダム値を入力として用いる技術に比較して，テスト対象プログラム実行時のコードカバレッジを高くするような入力が得られることが知られている [1], [2], [3], [4], [5].

あるプログラムを記号実行する際には，プログラムの入力変数を具体的な値が判らない記号変数として扱う．さらに，各実行パスを通過するために変数の値が満たすべき制約を求めながら，プログラムを実行する．そうして，プログラム内に存在する全ての実行パスについてパスを通過するための制約を求める．制約を SMT(Satisfiability Modulo Theories[6]) などのソルバに入力して充足解を得ることにより，各実行パスを通過するための入力変数の具体値を得ることが可能である．既存研究では，C/C++で記述されたプログラムの記号実行ツール [1], [3] や，Java で記述されたプログラムの記号実行ツール [2]，バイナリコードを対象とした記号実行ツール [4], [5] により，コーナーケースの不具合を発見したり，高いカバレッジを実現するテスト入力を生成できることが報告されている．

JavaScript で記述されたプログラムを対象とした記号実行処理系としては，Kudzu[7] や Jalangi[8] などが存在する．Kudzu は，対象プログラムのセキュリティ上の問題を Fuzzing によって発見することを目的として，検査対象の関数への入力データを生成する．Jalangi はプログラムの通常実行時とは異なる入力で対象プログラムを実行するためにパス条件の一部を改変する．

本稿で提案する手法では，JavaScript プログラムを記号実行可能な処理系を用いて，対象プログラムを単体テストする際の入力データを生成する．また，記号実行を行う際のスタブ・ドライバを自動生成することも可能なため，手動でスタブ・ドライバを作成する必要がなく，自動テストを行うことが可能である．生成されたテストプログラムは，テスト対象プログラム中の実行可能なパスを高いカバレッジで実行するようなものとなっている．

提案手法には，既存研究に比較して以下の新規点が存在する．まず，提案手法で使用する記号実行処理系では，既存研究 [7], [8] で必要となっていた，対象コードの改変・再実行が不要である．また，新たに開発した制約ソルバ (第

[1] 株式会社富士通研究所
[2] 米国富士通研究所
[a)] tanida.hideo@jp.fujitsu.com
[b)] gli@us.fujitsu.com
[c)] indradeep.ghosh@us.fujitsu.com
[d)] uehara.tadahiro@jp.fujitsu.com

2.2.3 節) により，さまざまな文字列操作を含んだプログラムに対して多数のテスト入力を生成可能である．さらに，JavaScript プログラム一般の単体テストデータ生成に適用可能な既存研究 [8] では手動でスタブ・ドライバを記述する必要があったが，本稿で提案するスタブ・ドライバ自動生成手法は，完全自動でのテスト生成を実現する．結果，提案手法はより多くの開発現場で適用可能である．

本稿の以降の構成は以下の通りである．まず，第 2 節において，テストの自動生成・実行が望ましい背景を例を用いて説明し，提案する手法の流れを説明する．そして，JavaScript コードの記号実行によるテストデータ自動生成手法を説明する．第 3 節では，テスト生成・実行の際に用いるスタブ・ドライバを自動生成する手法を説明する．本稿で提案する手法の評価は第 4 節において行なう．最後に第 5 節でまとめを行ない，今後の課題について議論する．

2. 背景と提案するテスト生成手法

2.1 テスト自動生成の必要性

ある特定のプログラムを実行した際に，プログラム内部に含まれる各パスが実行されるかは，プログラムへの入力に依存する．そのため，プログラムをテストで実行した際のコードカバレッジを高くためには，より多くのプログラム内のパスを実行するような入力を用いる必要がある．

例えば，図 1 に示した関数 `func0()` には，複数の実行パスが存在し，どのパスが実行されるかはテスト入力 (この例では引数 `s,a` の値と関数 `Lib.m2()` の戻り値) に依存する．図 1 に示したコードの実行パスを図示すると，図 2 のようになる．この例では，分岐結果により実行されるブロックが異なる部分が前半・後半に別れて存在する．そして，前半には block 0-3 の 4 個のブロック，後半には block A,B の 2 個のブロックが存在する．これらのブロックが実行されるための条件は図 2 の各ブロック最上部に示したようなものとなる．Block 2 については，実行されるための複数の条件 (`!"".equals(s)` と `"".equals(s)`) 間に矛盾があるため実行不可能であるが，それ以外のブロックについては実行可能である．そして，各テストでは，block 0,1,3 と blockA,B を組み合わせて実行することとなるため，実行されるブロックの組み合わせは，$3 \times 2 = 6$ 個存在する．

```
function func0(s,a) {
  if("".equals(s)) { // block 0
    s = null;
  } else {
    if(s.length <= 5) { // block 1
      a = a + status;
    } else {
      if("".equals(s)) { // block 2
        Lib.m0(); // Unreachable
      } else { // block 3
        Lib.m1();
      }
    }
  }
  if(a <= Lib.m2()) { // block A
    a = 0;
  } else { // block B
    a = a + s.length; // Error with null s
  }
}
```

図 1 手法の説明に用いるコード．func0() の引数 s,a は任意の値をとり，Lib.m2() は任意の値を返却する．

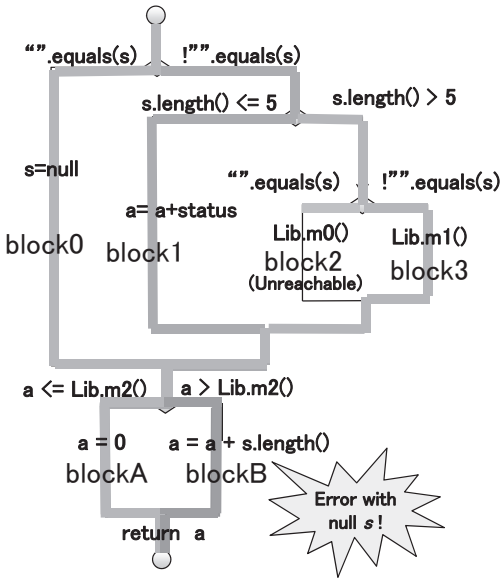

図 2 図 1 のコードに含まれる実行パス

表 1 には，実行されるブロックの組み合わせと，その際に引数 `s,a` の値・関数 `Lib.m2()` の戻り値が満たすべきパス条件が記されている．この例では，各パス条件を満たす値の組み合わせを求めることが可能で，それらをテストデータとして用いることが出来る．テストデータを全て用いて対象プログラムを実行した後には `func0()` 内の実行可能な文が全て実行された状態となる．以降ではこのようなテストデータを自動で生成する手法を説明する．

2.2 JavaScript コードの記号実行によるテストデータ自動生成

本稿では，著者らが提案・実装した JavaScript の記号実行処理系 SymJS を用いてテストデータを自動生成する手法を提案する．あるプログラムを記号実行する際には，プログラムに含まれる各パスを実行する際に各変数が充足する条件 (表 1 のパス条件) を求めつつ，プログラムを実行する．その結果，プログラム内の各パスに対応するパス条件が得られる．また，各パス条件を充足する変数の具体値は，SMT ソルバなどにより算出可能である．得られた具体値を用いてプログラムを実行すると，パス条件に対応す

表 1 図 2 のパスを実行するためにテスト入力が満たすべきパス条件と充足するテストデータ (変数 status の値は-1 とする)

Test No.	Blocks Executed	Path Conditions	Test Data
1	0,A	`"".equals(s) ∧ a<=Lib.m2()`	`s="", a=0 Lib.m2()=0`
2	0,B	`"".equals(s) ∧ a>Lib.m2()`	`s="", a=0 Lib.m2()=-1`
3	1,A	`!"".equals(s) ∧ s.length <= 5 ∧ a-1<=Lib.m2()`	`s="a", a=0 Lib.m2()=0`
4	1,B	`!"".equals(s) ∧ s.length <= 5 ∧ a-1>Lib.m2()`	`s="a", a=1 Lib.m2()=0`
5	3,A	`!"".equals(s) ∧ s.length > 5 ∧ a<=Lib.m2()`	`s="aaaaaa", a=0 Lib.m2()=0`
6	3,B	`!"".equals(s) ∧ s.length > 5 ∧ a>Lib.m2()`	`s="aaaaaa", a=0 Lib.m2()=-1`

るパスを実行することが可能なため，これらをテスト入力として用いることが出来る．

SymJS では，JavaScript で記述されたプログラムの記号実行が可能である．SymJS は，対象プログラムのバイトコードを解釈し，KLEE[1] や Symbolic JPF [2] と同様に記号実行する．SymJS では，JavaScript の中核を構成する ECMAScript[*1] の言語標準に定義された文法で記述されたプログラムを記号実行可能である．

2.2.1 JavaScript 記号実行インタプリタ SymJS

SymJS は対象プログラムをコンパイルしたバイトコード (Rhino [*2] の Icode) を解釈・実行するインタプリタとして実装されている．ある命令が実行されるとヒープ・スタックとパス条件で構成されるプログラムの状態が更新され，分岐が存在する場合にはプログラムの状態を複製して各分岐を実行する．

SymJS は，既存の Rhino の処理系に対して，算術論理演算・比較・分岐・関数呼び出し・オブジェクト操作・オブジェクト参照に使用する命令の解釈に対する改変を行って，対象とする JavaScript プログラムの記号実行を実現している．なお，スタック操作や例外処理・変数スコープの管理に関する命令については，改変を行っていない．

Rhino のバイトコードでは比較命令の後には分岐命令が配置されている．SymJS はこれらの比較・分岐命令の組み合わせに遭遇した場合，比較命令の実行結果を表現する真偽式を作成する．作成した式を c とした場合，c とその反転 $\neg c$ が命令実行前のパス条件 pc と合わせて充足可能かを判定する．$pc \wedge c$ と $pc \wedge \neg c$ のどちらも充足可能な場合，$pc \wedge c$ と $pc \wedge \neg c$ に対応する実行状態 s_1, s_2 を複製する．そして，状態 s_1, s_2 から各分岐の実行を継続する．片方の条件のみ充足可能な場合は，そちらに対応する分岐を選択して実行を継続する．

[*1] http://www.ecma-international.org/publications/standards/Ecma-262.htm
[*2] https://developer.mozilla.org/en-US/docs/Rhino

2.2.2 実行状態の管理手法

SymJS は対象プログラムを実行する際に，分岐などで複製された状態群を管理しているが，状態を複製する手法として二種類を使用可能である．第一の手法は，[1], [2] と同様に，ヒープ・スタックを含むプログラムの状態をメモリ上で記憶しておくものである．

もう一つの手法は，各分岐においてどちらの分岐を実行したかの情報のみを記憶しておくものである．こちらは，プログラムを途中から再実行する場合は初期状態からの実行をやり直さなければならないため，無駄な再演算が生じる．しかし，実装は単純で必要となるメモリ容量も少なくなる利点が存在する．こちらの手法は Fuzzing と呼ばれ，[3], [5] にて紹介されているものに類似する．Fuzzing を行って対象プログラムを記号実行する際には，各実行状態はプログラム内の各分岐で左右どちらの分岐を実行したかのみで記憶される．そして，その情報を元にプログラムを初期状態から必要に応じて再実行しつつ，対象プログラムの取り得る状態空間を探索する．

2.2.3 制約ソルバ PASS

前節に記したような手法で管理される各実行状態の情報に対して，表 1 に示したようなパス条件が得られる．パス条件を充足するテストデータは，数値についてはパス条件を SMT ソルバへ入力することにより得られるが，SMT ソルバは JavaScript コードで多用される文字列を扱うことができない．そこで，制約ソルバ PASS[9] を用いる．

PASS は整数・ビットベクタ・浮動小数点数や文字列に関する制約を解くことが可能であり，パラメタライズド・アレイを用いたモデル化を導入することにより，より多くの制約を効率的に解くことが可能となっている．

2.3 記号実行スタブ・ドライバ

SymJS では，自動生成したテスト入力により値を決定させたい記号変数を以下のような関数呼び出しで定義できる．

```
var s = symjs_mk_symbolic_string();
```

上記の例では文字列型の記号変数を定義しているが，関数 `symjs_mk_symbolic_int()`, `symjs_mk_symbolic_bool()`, `symjs_mk_symbolic_real()` の呼び出しによって，それぞれ整数値型・真偽値型・実数値型の変数も定義可能である．SymJS において記号変数として扱える型は，文字列型・整数値型・真偽値型・実数値型に限られるが，これらの型の変数のとる記号値がより複雑なオブジェクトのメンバとして代入・参照された場合にも記号値に対する制約は保持され，その具体値を変化させたテストの生成が可能である．

図 1 に示した関数 `func0()` のテスト入力を SymJS により決定するためには，以下に記すように記号実行ドライバ・スタブを追加する必要がある．まず，図 3 に示すような `func0()` の引数に対応する型の記号変数を新たに定義して関数を呼び出すような記号実行ドライバを追加する．さらに，関数 `Lib.m2()` の戻り値を SymJS により決定するためには，`Lib.m2()` の戻り値に対応する型の記号変数を新たに作成して返却する記号実行スタブ (図 4) を追加する．

SymJS はテストデータを JSON 形式の外部ファイルへ出力する機能を備える．ファイルのデータは，`symjs_mk_symbolic_*()` の呼出に応じて対応する変数のテストデータを返却するテストデータ再生ライブラリで読み込むことが可能で，実ブラウザ上でのテストデータを用いた対象プログラムの実行に用いることが可能である．

3. 記号実行スタブ・ドライバの自動生成

第 2.3 節にも記したとおり，テスト対象関数の記号実行を行ってテストデータを出力するためには，記号実行スタブ・ドライバが必要となる．記号実行スタブ・ドライバは，手動で作成したものを使用可能であるが，スタブ・ドライバの手動での作成には追加の工数が必要となる．特に，記号実行による単体テスト生成・実行を開発早期から適用するためには，スタブ・ドライバを自動生成することが望ましい．そこで，記号実行スタブ・ドライバを自動生成し，テスト生成・実行に用いることとした．

3.1 記号実行スタブ・ドライバの自動生成方針

3.1.1 記号実行スタブの生成方針

記号実行スタブの自動生成では，スタブ化対象に指定した関数・クラスのスタブを生成する．

提案する手法では，呼び出し元で期待する戻り値の型が判明している際に，その型に応じて以下のような値を返却する関数を自動生成し，スタブとして用いることとした．

- SymJS が記号変数として扱うことが可能な文字列型・整数値型・真偽値型・実数値型 (以下，SymJS 基本型)
  → 対応する型の記号変数を新たに定義して返却
- その他クラスのオブジェクト
  → 対応するクラスのインスタンスを新規に生成して返却．クラスがスタブ化対象の場合，スタブのインスタンスを生成して返却する．
- void 型 → 何も行わない

また，クラスをスタブ化する際には，コンストラクタのスタブも作成する必要があるが，コードを全く含まないコンストラクタを生成して用いる [*3]．結果として，自動生成するクラスのスタブは，状態を全く持たないものとなる．

なお，戻り値の型がその他クラスのオブジェクトである場合も，そのクラスがスタブ化対象で，クラス内で定義された関数の戻り値の型が SymJS 基本型であるならば，その関数の返却値は新たな記号変数となる．つまり，スタブ化されたクラスのインスタンスメソッドの呼出結果も，記号変数となり得る．そのため，インスタンスメソッドの呼出結果によって実行パスが異なるプログラムのテストデータも，自動生成したスタブにより生成可能である．

3.1.2 記号実行ドライバの生成方針

記号実行ドライバの自動生成では，対象とした関数を呼び出すコードを生成する．提案する手法では，以下のようなコードを生成して，記号実行ドライバとして用いることとした．

- テスト対象関数がインスタンスの存在を前提とするものであれば，関数が属するクラスのインスタンスを生成して関数を呼び出す
- テスト対象関数が静的でインスタンスの存在を前提としないものであれば，そのまま関数を呼び出す

また，テスト対象関数の引数の型が判明している際に，その型に応じて以下のように記号変数・オブジェクトを定義・生成し，引数として用いることとした．

- SymJS 基本型
  → 対応する型の記号変数を新たに定義
- その他クラスのオブジェクト
  → 対応するクラスのインスタンスを新規に生成．対応するクラスがスタブ化対象の場合，スタブのインスタンスを生成する．

引数の型を決定する方針は，自動生成された記号実行スタブの返却値の型を決定する方針に類似したものである．

3.2 アノテーションに基づくスタブ・ドライバの生成

第 3.1.1 節において提案した記号実行スタブ生成方針では，スタブ化対象関数の呼び出し元で期待する戻り値の型が判明していることを前提としていた．また，第 3.1.2 節において提案した記号実行ドライバの生成方針では，関数の引数の型が判明していることを前提としていた．

実際には，JavaScript は動的に型付けが行われる言語のため，実行時まで関数の戻り値や引数の型は判らない．しかし，多くの JavaScript プログラムにおいては，期待する関数の戻り値や引数の型が API などで定められている．さらに，期待する関数の戻り値や引数の型を表現する方法として，JSDoc 形式によるアノテーションが存在する．そこで，本稿で提案する手法では JSDoc3 [*4] 形式のアノテーションを含むソースコードから型情報を取得し，記号実行スタブ・ドライバの生成を行うこととした．

3.2.1 アノテーションに基づく記号実行スタブの生成

記号実行スタブの生成には，まずアノテーションを含む

[*3] なお，クラスのフィールドへの直接アクセスは想定しておらず，フィールドのスタブは生成しない

[*4] http://usejsdoc.org/index.html

```
var s = symjs_mk_symbolic_string();
var a = symjs_mk_symbolic_float();
func0(s,a);
```

図 3 図 1 の関数を呼び出す記号実行ドライバ

```
Lib.m2 = function() {
  return symjs_mk_symbolic_float();
};
```

図 4 図 1 で呼び出される Lib.m2() を定義する記号実行スタブ

```
/** @return {Number} m2 value */
Lib.m2 = function() { ... };
```

図 5 図 4 に示した記号実行スタブを
自動生成可能なアノテーションを含む関数定義

```
/** @param {String} s
  * @param {Number} a */
function func0(s,a) { ... }
```

図 6 図 3 に示した記号実行ドライバの自動生成を可能とする
テスト対象関数へのアノテーション

スタブ生成対象関数のソースコードが必要となる．さらに，記号実行スタブ生成対象となる関数のアノテーションに戻り値の型情報が含まれている場合に，関数の記号実行スタブを自動生成可能である．

JSDoc3 では，関数の戻り値の型情報は主に`@return` アノテーションを用いて定義する．図 1 の例で関数 `Lib.m2()` のスタブを生成することを考える．このとき，図 5 に示すようなアノテーションがスタブ生成対象関数の元の定義に付加されていると，関数の返却値の型が判り，図 4 に示したようなスタブの自動生成が可能である．

3.2.2 アノテーションに基づく記号実行ドライバの生成

記号実行ドライバの生成には，まずアノテーションを含むテスト対象関数のソースコードが必要となる．そして，記号実行ドライバ生成対象となる関数のアノテーションに引数の型情報が含まれているならば，テスト対象関数の記号実行ドライバを自動生成可能である．

JSDoc3 では，関数の引数の型情報は主に`@param` アノテーションを用いて定義する．図 1 の例で，関数 `func0()` のドライバを生成することを考える．このとき，図 6 に示すようなアノテーションがドライバ生成対象関数定義の前に付加されていると，関数の引数の型が判り，図 3 に示したようなドライバの自動生成が可能である．

3.2.3 アノテーションに基づく記号実行スタブ・ドライバ自動生成の実装

提案するアノテーションに基づく記号実行スタブ・ドライバの自動生成は，JSDoc3 のプラグインとして実装した．

JSDoc3 には，ドキュメント生成対象として入力されたプログラム内にクラス・関数などの定義を発見した際，プラグインのイベントハンドラを呼び出す機構が存在する．また，イベントハンドラ内では発見したクラス・関数などの戻り値や引数に関する情報を参照することが可能である．

そこで，JSDoc3 に入力されたプログラム内にクラス・関数などを発見した際にイベントハンドラに与えられる情報を元に，記号実行スタブ・ドライバを自動生成するプラグインを実装した．プラグインは，入力されたプログラム内の全クラス・関数のスタブ・ドライバを自動生成する．

表 2 実験に用いたアプリケーション (一部) の諸元

#Line	#Function	#File
431	23	1

表 3 スタブ生成に用いたフレームワーク・生成されたスタブの諸元

#Line(Orig.)	#Line(Stub)	#Function	#File
2843	1304	154	13

4. 評価実験

本稿で提案する手法によって高カバレッジを実現する単体テストの自動生成・実行が可能であることを確認するため，現場の開発資産を用いた実験を行った．資産は，社内で開発しているフレームワークを用いて動作する Web アプリケーションの一部を構成する JavaScript コードである．以下では，この JavaScript コードを「アプリケーション」と呼ぶ．アプリケーションは，HTML DOM へのアクセスやサーバとの通信など ECMAScript 標準に**含まれない**操作は，全てフレームワークが提供する API を介して行う．アプリケーションの諸元を表 2 に示す．

4.1 記号実行スタブ・ドライバの生成

第 2 節に提案した記号実行によるテストデータ生成を自動で行うため，第 3 節の手法で記号実行スタブ・ドライバを生成した．

記号実行スタブの生成を行うための入力として，アプリケーションの実装に用いられているフレームワークのソースコードを用いた．ソースコードには JSDoc3 形式のアノテーションが付与されており，スタブ生成に必要な関数の戻り値の型情報を取得可能である．スタブ生成を試みた結果，フレームワークに定義されている全てのクラス・関数について，スタブを自動生成することが出来た．フレームワーク・生成されたスタブのソースコードの諸元を表 3 に示す．アプリケーションはフレームワークが提供する API のみで動作するため，アプリケーションの記号実行に必要なスタブが全て生成できたことになる．

また，記号実行ドライバの生成を行うための入力としては，JSDoc3 形式によるアノテーションが付与された，アプリケーションのソースコードを用いた．ドライバ生成を試みた結果，ソースコードに含まれる全ての関数 23 件についてドライバを自動生成することが出来た．

4.2 テストデータの生成・データを用いたテスト実行

得られたスタブ・ドライバを用いて，アプリケーションを関数単位で記号実行し，テストデータを生成した．

全ての関数について記号実行は *1* **秒以内**に終了して，テストデータが生成された．各記号変数への値の割り当てを決定するテストデータは対象とした関数に依存して，最小 *0* **件** (分岐がない関数の場合)，最大 *27* **件**生成された．

得られたテストデータを用いて，各関数を実行した．実行には，Web ブラウザとテストデータ再生ライブラリを組み合わせた環境を用いた．また，JSCover [*5] により実行時のカバレッジを測定した．結果，テスト実行時のラインカバレッジは，アプリケーションのソースコード全体で *92%* となり，提案手法は高いカバレッジを実現する単体テストデータを自動生成可能であることが示された．

4.3 提案したテスト手法で実行されなかった処理

実験結果から，提案手法によって高いラインカバレッジを実現するテストを生成可能なことが確認されたが，ラインカバレッジは 100%ではなく，一部の文が実行されていない．以下に，どのような文が実行されなかったかを記す．

まず，想定外の型のオブジェクトに対する処理が実行されなかった．JavaScript は動的に型付けが行われる言語のため，想定した型と違う型のオブジェクトが関数の返却値となる状況が発生し得る．このような場合を想定して，アプリケーションのコードにも型のチェックとその結果に基づくエラー処理が含まれる．しかし，提案手法で生成するスタブは，常にアノテーションに記された型のオブジェクトを返却するため，そのような処理を実行させられない．

また，オブジェクトの型が予め判らない場合の処理の処理も，実行されなかった．アプリケーションのコードには実行時にオブジェクトの型を判定し，その結果に応じて処理を行うものが存在する．しかし，提案するスタブ生成手法では，ある関数の戻り値の型が不明の場合，既定の Object を返却する関数を生成してスタブとして用いている．そのため，既定の Object 以外が返却された際の処理を実行できない．

さらに，例外が発生した際の処理を記述する catch ブロックが，実行されなかった．アプリケーションのコードには，実行時エラーによりフレームワークから例外が発生した場合の処理も含まれる．しかし，自動生成されるフレームワークのスタブは例外を発生させず，そのような処理を実行させられない．

5. まとめ

5.1 結論

本稿では，JavaScript プログラムを記号実行することにより網羅性の高い単体テストデータを自動生成する手法を提案・評価した．手法は，記号実行によって対象プログラムを解析してテストデータを生成する技術と，記号実行に用いるスタブ・ドライバをソースコードのアノテーションから自動生成する技術により構成される．手法を現場の開発資産に適用した結果，92%という高いラインカバレッジを実現するテストを自動生成可能であることが確認され，手法の有効性が示された．

5.2 今後の課題

今後の課題としては，以下のようなものが挙げられる．

まず，さらに多くのプログラムを対象とした提案手法の評価を行う必要がある．また，今回は比較的小規模なプログラムを対象として評価を行ったが，さらに大規模なものを用いた評価も必要である．

また，今回の実験で実行されなかった処理を実行するには，用いる記号実行スタブの改善が必要と考えられる．想定外の型のオブジェクトに対する処理やオブジェクトの型が予め判らない場合の処理を実行するには，パラメータに応じて異なる型のオブジェクトを返却するスタブを用いることが有効であると考えられる．例外が発生した際の処理を実行するには，パラメータに応じて複数の箇所で例外を発生させるスタブを用いることが有効であると考えられる．これらのスタブ改善には，記号実行スタブ自動生成のアルゴリズムを改善する以外に，自動生成されたスタブを開発者が手動で修正する手法も有効であると考えられる．

[*5] http://tntim96.github.io/JSCover/

参考文献

[1] Cadar, C., Dunbar, D. and Engler, D. R.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, *OSDI* (2008).
[2] Păsăreanu, C. S. and Rungta, N.: Symbolic PathFinder: Symbolic Execution of Java Bytecode, *ASE* (2010).
[3] Sen, K., Marinov, D. and Agha, G.: CUTE: A Concolic Unit Testing Engine for C, *ESEC/FSE* (2005).
[4] Tillmann, N. and De Halleux, J.: Pex: White Box Test Generation for .NET, *International Conference on Tests and Proofs (TAP)* (2008).
[5] Godefroid, P., Levin, M. Y. and Molnar, D.: Sage: Whitebox Fuzzing for Security Testing, *Queue*, Vol. 10, No. 1, p. 20 (2012).
[6] Kroening, D. and Strichman, O.: *Decision Procedures: An Algorithmic Point of View*, Springer Publishing Company, Incorporated (2008).
[7] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S. and Song, D.: A Symbolic Execution Framework for JavaScript, *IEEE Symposium on Security and Privacy* (2010).
[8] Sen, K., Brutch, T., Gibbs, S. and Kalasapur, S.: Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript, *SIGSOFT FSE* (2013).
[9] Li, G. and Ghosh, I.: PASS: String Solving with Parameterized Array and Interval Automaton, *Haifa Verification Conference (HVC)* (2013).