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

片山朝子[†1]　上原忠弘[†1]　藤原翔一朗[†1]　宗像一樹[†1]
M SUPASIT[†1]　徳本晋[†1]　前田芳晴[†1]

本論文は現在我々が開発中の Java 向けのシンボリック実行による検証システム（VaaS）を使って実際の業務システムを対象に検証を行った際の環境構築および実行結果をまとめたものである．業務システム中の特に業務ロジックを含むクラスを対象とし，ドライバ・スタブを作成して VaaS を適用し，網羅的なテストケース作成にかかる作業時間を測定した．その結果 VaaS による検証時と通常のテストを比較して，生産性は VaaS で行うと通常で行われるテストに比べて約 2/5 になるが，あらかじめスタブを用意しておけば通常テストに比べて 1．59 倍になる，という結果を得ることができた．また VaaS は（想定外と思われる）例外を 15 件発見し，例外発見にも有効であることが確認された．

Symbolic execution verification trial on business system

ASAKO KATAYAMA[†1]　TADAHIRO UEHARA[†1]
SHOICHIRO FUJIWARA[†1]　KAZUKI MUNAKATA[†1]
M SUPASIT[†1]　SUSUMU TOKUMOTO[†1]　YOSHIHARU MAEDA[†1]

This paper is trial report of the verification result and its environment on actual business system with our symbolic execution verification system (VaaS) which we produced now. We produced drivers and stubs to apply VaaS to the common component class of the system and set a goal to line coverage 100%.We got the results compare to the ordinary tests that the VaaS productivity was 1/3 but to make preparations of stubs in advance it became 1.59 times. Also VaaS detected 15 exceptions (seemed to be beyond the scope of the assumption) and confirmed the effectiveness of detection on unrecognized exceptions.

1. はじめに

一連のテストケースによるテスト実行の累積によってテストがどれほど達成されたかを測るテスト網羅度（test coverage）という指標がよく用いられる． テスト網羅度を高めるためにはその網羅度が高くなるようなテストデータを選んで実行することが必要である．

シンボリック実行とは，プログラムの入力として具体的値を与える代わりに，値を代表するシンボル（記号）を与えてプログラムを模擬的に実行し，その結果を評価する手法である．実行中にパスの条件を得ることができるので，その条件を分析することでテストデータを得ることができる．

この方法ではプログラムを網羅的に実行し，パスを抽出するため，人手で生成するデータより網羅性の高いテストデータが生成されることが期待でき，自動生成によりデータの質の均一化，作業効率に貢献することが期待されている．

我々のグループでは，このテストデータ生成機能を開発しこの技術を手軽に利用できるようにするために，本技術を Web サービス化し VaaS（Validation as a Service）と名付けた．

今回 VaaS を実際の業務システムに適用し，生産性およびテスト網羅度において人手によるテスト生成との比較を行い，業務開発プロセスに組み込む上での課題を明らかにした．

2. VaaS の概要

2.1 シンボリック実行とデータ生成

シンボリック実行[1][2]は，テスト対象プログラム中の変数を具体値にせずシンボルとして扱う実行方式である．プログラム中でシンボルを格納する変数をシンボル変数と呼ぶ．シンボルに関する条件分岐を解析しながらそれぞれのパスを実行し，すべての実行可能なプログラムパスを網羅するまで繰り返す．各パスにおけるシンボルの取り得る条件をパス条件と呼び， 充足性判定ツール（SMTソルバ）を用いてパス条件を満たす具体値を得ることで，そのパスを実行するテストデータを生成する．以下に検証対象プログラムの例とそのプログラムを表すフローチャート（図１）を示す．

†1 (株)富士通研究所
Fujitsu Lab Ltd.

```
package sample;

public class Test {

public static void runTest(int a, int b, int c) {
    if (b >= 0) {
        c = c + 10;
        if (a <= 0) {

            if (b > 15) {
                System.out.println("block5");
            } else {

                if (c == 0) {
                    System.out.println("block4");
                } else {
                    System.out.println("block3");
                }
            }
        } else {
            System.out.println("block2");
        }
    } else {
        System.out.println("block1");
    }
}
}
```

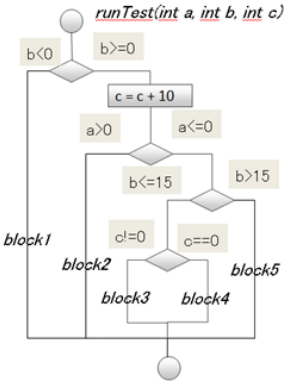

図 1　検証対象プログラムの例

Figure 1　The example of verification target program.

パラメータ a，b，c にシンボル sa，sb，sc を代入して解析すると，まずフローチャートの block1 を通るパスを実行するための条件として sb<0 を満たさなければならない．sb は代表値として適当な値-1 を割り当て，sa と sc は任意の値なのでデフォルト値 0 を割り当てれば block1 のパスを満たすデータになる．同様に block3 を通るパスを実行する条件は sb>=0∧sa<=0∧sb<=15∧(sc+10)!=0)というパス条件が得られる．この条件式を解いてデフォルト値と合わせると sa=0，sb=0，sc=0 がテストデータとして得られる． 同様にして以下の５つのテストケースとデータのセットを得られる（図２）．

テストケース	テストデータ	パスコンディション
1	a = 0 b = -1 c = 0	sb < 0
2	a = 1 b = 0 c = 0	(sa > 0) ∧ (sb >= 0)
3	a = 0 b = 0 c = 0	((sc + 10) != 0) ∧ (sb <= 15) ∧ (sb >= 0) ∧ (sa <= 0)
4	a = 0 b = 0 c = -10	((sc + 10) == 0) ∧ (sb <= 15) ∧ (sb >= 0)∧ (sa <= 0)
5	a = 0 b = 16 c = 0	(sb > 15) ∧ (sb >= 0) ∧ (sa <= 0)

図 2　生成されたテストケースとデータ

Figure 2　The created test cases and test data.

2.2　Java PathFinder

Java PathFinder（JPF）[3]は，米国National Aeronautics and Space Administration（NASA）が開発したJavaバイトコードプログラムを検証するためのモデル検査ツールである．JPFは独自のJVM上でプログラムを実行しながら，可能性のあるすべての実行経路を網羅的に探索する．Javaベースのシンボリック実行エンジンとして広く使われており，VaaSではこのJPFをシンボリックエンジンベースとして使い，独自にテストデータ生成機能を組み込んでデータ生成を行っている[a]．

2.3　VaaS の機能

VaaS（Validation as a Service）はJPFによるテストデータ生成を手軽に利用できるようにするために，SaaS（Software as a Service）型サービスとして開発した [4][5]（図３）．

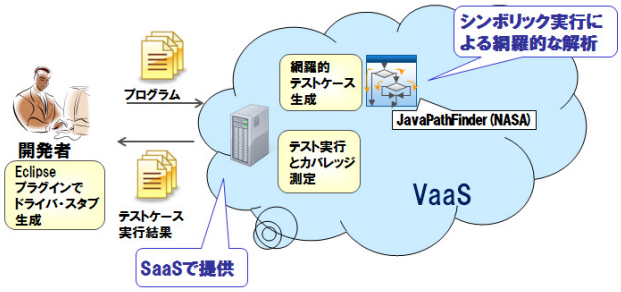

図 3　VaaS の概要

Figure 3　The overview of VaaS.

① VaaS クライアントプログラムにより指定したプログラムのドライバ・スタブ生成を行い，Web ブラウザ経由で対象プログラムと共に VaaS に入力する．

② VaaS 上で JPF シンボリック実行により網羅的な解析を行い，テストデータおよびテストケースを生成する．

③ 得られたテストデータでテストケースを実行(JUnit を利用)し，そのテスト網羅度も併せて測定する．得られたテストケースとその実行結果およびテスト網羅度をブラウザ経由で表示する．

VaaS 実行の流れは以下の通りである（図４）．

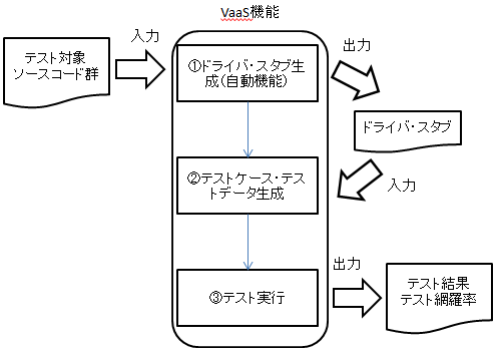

図 4　VaaS 実行の流れ

Figure 4　The flow of VaaS verification.

(1)　ドライバ・スタブ生成

JPF 上の実行には main 関数が必要でありそのためのドライバを生成する．大規模なテスト対象プログラムの場合には，人手によるドライバ作成の労力が増大するため VaaS はドライバ自動生成ツールを備えている．以下の項目を指定することによりテストドライバを自動生成することができる．

a) 以下，特にことわりのない限り，JPF とは我々が機能強化した版のものを指す．

- テスト対象メソッドの指定
- シンボル化する変数の指定
- スタブ化クラスの指定

以下にテストドライバ生成例を示す（図５）．テスト対象クラスのメンバおよび対象メソッドの引数をシンボル宣言し，メイン関数でドライバメソッド testFunc を呼ぶ．testFunc 内では以下を行う．

- スタブ化クラス指定があればスタブクラス初期化
- パラメータバリエーションのためのパラメータ初期化．
- 対象クラスのインスタンスを生成
- インスタンスメンバおよびパラメータに宣言したシンボルを格納
- 対象メソッドを起動

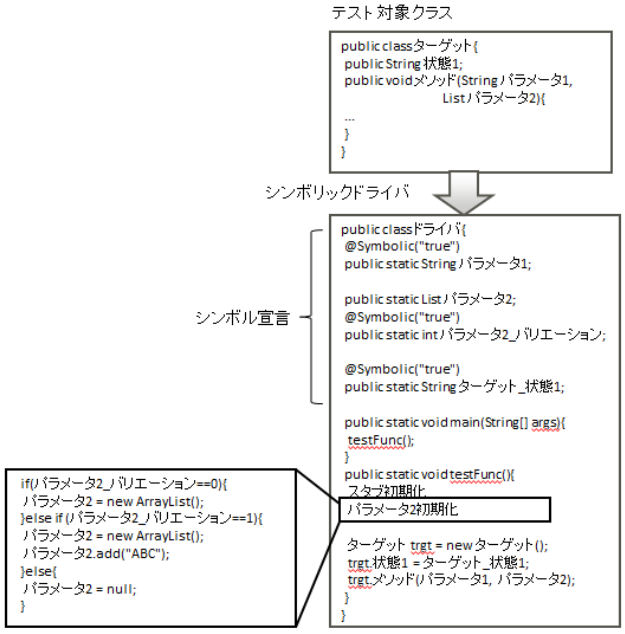

図 5　検証対象プログラムのテストドライバ生成

Figure 5　The test driver creation of verification target program.

テスト対象のメソッドのパラメータがオブジェクト型の場合，オブジェクトの生成をしてそのオブジェクトのフィールド変数にシンボルを設定するコードが生成される．しかし現状の実装ではプリミティブ型と文字列型のみシンボル化することが可能であり，オブジェクトの構造やリスト・配列の数などのバリエーションは扱えない．そこで、構造のバリエーションを表現するシンボル（図５の例ではパラメータ 2_バリエーション)を導入し，その値によって異なる構造のオブジェクトを生成するロジックをドライバに埋め込むことで，シンボリック実行時にバリエーションを生成できるようにした（図 5 ではパラメータ 2 が要素 0 個の ArrayList、要素 1 個の ArrayList null の 3 パターンが生成）．

また呼び出す部品が未実装である場合や外部機能（データベース等）であるなどの理由で利用できない場合スタブを用いる．こちらも大規模プロジェクトに対応した省力化のため，スタブ自動生成を備えている．以下に VaaS におけるスタブ自動生成の例を示す（図６）． 対象メソッドの入出力に注目し， その型に合った返り値を返すようにする．以下では返り値の int 型のシンボルを返す．

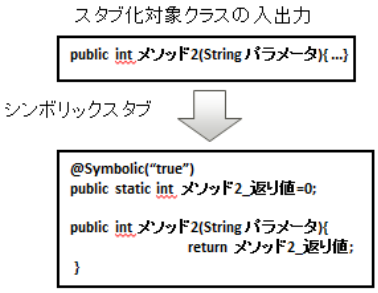

図 6　スタブ生成の例

Figure 6　The example of stub generation.

しかし java の Generic 型のテスト対象の引数やスタブ返却値に含まれるとインスタンス化するクラスの型が特定できないなど自動生成で全てを対応することはできない．そのためスタブやドライバの手動による修正は適宜必要となる．

(2)　テストケース・テストデータ生成

生成されたドライバ・スタブを用いJPFは網羅的実行を繰り返し，その実行パスをテストケースとして出力する．VaaSは各実行パスの持つパス条件をSMTソルバと文字列に関する制約ソルバを組み合わせて解くことで数値だけでなく文字列に対してもデータ生成を行うことができるようにした[6].

(3)　テスト実行

生成されたテストケースをJUnit（Javaプログラムの単体テストフレームワーク[7]）を利用してテスト実行を行う．VaaSでは生成されたテストデータを実行するためのJUnitドライバを生成し，生成されたデータを入力としてJUnitを実行することでテストを自動的に実行する．JUnitテスト実行時にラインカバレッジ/ブランチカバレッジでテスト実行の網羅率を測定する．VaaSでは期待値を設定しているわけではないため，テストが仕様に対して正しく実行されたかどうか判断することはできないが，ユーザがJUnitのテストケースアサーションを追記し実行することによって仕様との一致を確認することができる．

本サービスを利用することにより，テストデータ生成が自動化され，テストデータの網羅率の向上やテスト工程の省力を効果として期待している．

3.　業務システムへの VaaS 試行

3.1　業務システム概要

今回試行対象とした業務システムは開発過程にある 1 プロジェクトの資産であり，業務クラスとそのクラスが利

用するアプリケーション基盤ライブラリから成る（図 7）．今回の業務システムを対象とする試行にあたり VaaS が業務ロジックを満たすテストデータ，テストケースを生成できるかどうかを見極めるのが重要と考えた．そのため業務ロジックが集中する業務クラスの全 26 メソッドおよび各メソッドの呼び出しを行うテストハンドラクラスを検証対象とし，各メソッドのテスト実行のラインカバレッジを 100%にすることを目指した．

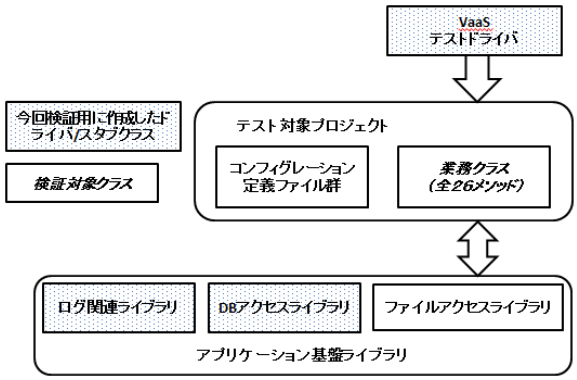

図 7　試行対象の業務プロジェクト概要

Figure 7　The structure of the trial business project.

3.2 ドライバの概要

自動生成されたドライバを手動にて対象メソッドに合うように随時修正を加え最終的に 320 行になった．そのうち自動生成された部分は 45 行程度あった．以下の図 8 が今回使用したテストドライバクラス概要である．

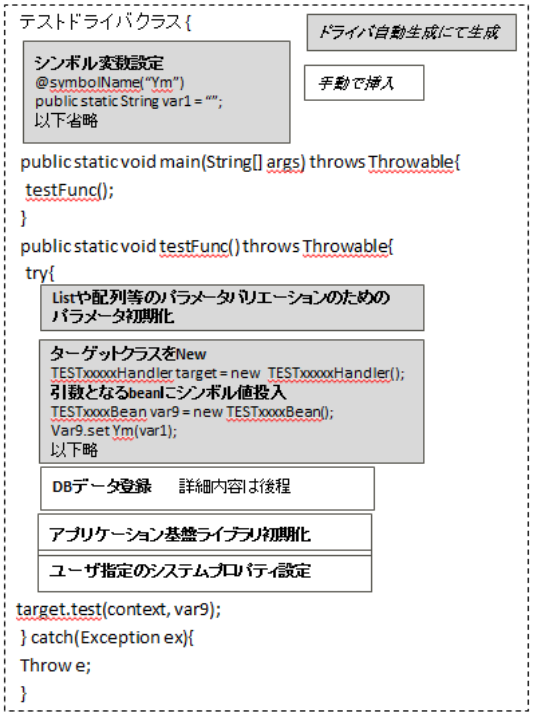

図 8　ドライバ概要

Figure 8　The driver overview.

以下に手動追加修正を加えた部分を説明する．

(1) シンボル値設定

自動作成+手動修正．作成された全てのシンボル変数のうち，対象メソッド内でプログラム分岐に関わるものだけを残すようにした．

(2) パラメータ初期化

自動生成+手動修正．2.3 節のドライバ・スタブ生成で説明した List や配列のパラメータバリエーション展開であるが，(1)と同様に対象メソッド内でプログラム分岐に関わるものだけを残すようにした．

(3) DB データ登録

手動挿入．DB スタブ部分．3．3 のスタブで説明する．

(4) DB データ登録

手動挿入．DB スタブ部分．3．3 のスタブで説明する．

(5) アプリケーション基盤ライブラリ初期化

手動挿入．フレームワークのライブラリを使うためのファイルアクセスライブラリ，ログライブラリ等の初期化を挿入．

(6) ユーザ指定のシステムプロパティ設定

手動挿入．VaaS には対象プログラムのシステムプロパティを挿入する手段がないため，ドライバで対象プログラムに埋め込む必要があった．

3.3 スタブ化したクラス

業務クラスを VaaS 試行するために作成したスタブクラスを以下に示す．

- 文字列チェッククラス
- DB アクセスクラス
- ログ関連クラス

この他に部分的にビジネスクラスの 1 つとして部分的に使用されているビジネス日時に関わるクラスについてもスタブ化した．スタブ化したクラス数は 11 クラス，1667 行になった．スタブ化したポイントは次の通りである．

(1) 文字列チェックロジック回避

文字列型のシンボルに対し文字列チェックロジックをシンボリック実行するとバリエーションが爆発してしまうためスタブ化を行った．文字種チェックだけではなく日時関連のチェックも同様にした．

(2) DB アクセスクラス回避

試行では DB を用意できなかったため，DB アクセス回避のために DB アクセスクラスモジュールのスタブ化を行った．スタブ化の仕組みは以下のように作った（図 9）．

- DB データ登録テーブルを作成する
- スタブ化した DB データ登録クラスを使ってテストドライバにてテストに必要な DB データをあらかじめデータ bean として登録しておく．
- テスト対象メソッド内で該当するデータ bean が呼ばれると DB データ登録クラスを経由してあらかじめ登録しておいたデータ bean が返る．

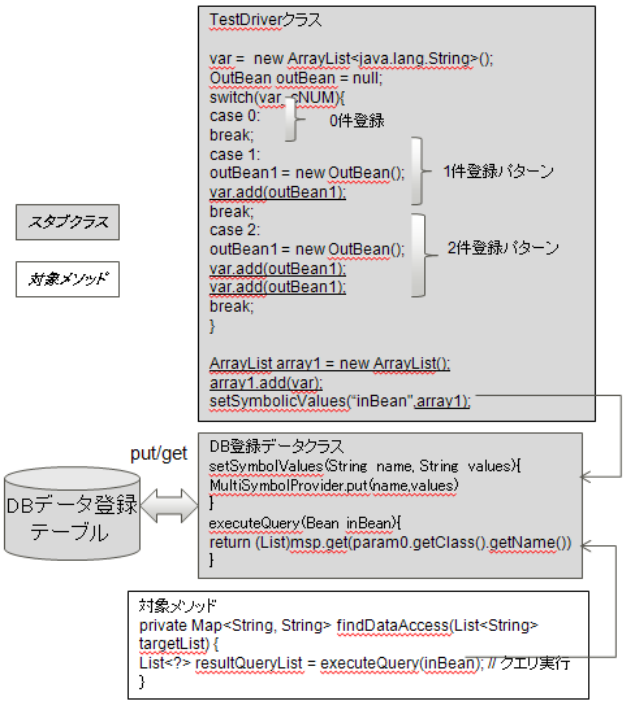

図 9　DB スタブの仕組み

Figure 9　DB stub structure.

(3)　ログ関連クラス回避

ログ関連モジュール自体が独立した別のフレームワークになっており，対象メソッド内の検証に絞り込むためにもスタブ化を行った．

(4)　仕様不明クラスを回避

ビジネス日時などアプリケーション特有の仕様に基づくもので仕様が不明なものに関してはスタブ化を行った．

4.　試行適用結果

試行対象の業務クラスの全 26 メソッドに対して VaaS にて試行を行い，23 メソッドについてテストケースおよびテストデータを得た．残り 3 メソッドについてはメモリの限界，実行が収束しない，JPF 未サポートメソッドの存在などによりテストケースを作成できなかった．従来テストとの比較には開発プロジェクトで開発時に作成したテストケースおよび工数を用いた．

4.1　生産性比較

以下は業務クラスのうちの 9 メソッドにあたるサブシステム A のテストの VaaS 試行の結果，および従来のテスト方法で行った時の結果の比較である（表 1）．

試行方法	行数	作業時間	1時間あたりのテスト実施行数	テストケース数
VaaS（サブシステムA）	189行	11時間8分(VaaS実行時間を除く)	189(行)/11.13(H) ≒ **17行/H**	6073(パス) 78(分岐)
従来のテスト法	561行	13時間	561(行)/13(H) ≒ **43.2行/H**	38件

表 1　VaaS 試行工数と従来テスト工数との比較表

Table 10　The table of comparing the test costs between VaaS trial and original test.

項目「テストケース数」でパス網羅するテストケース数を（パス）と表現し，分岐網羅するテストケース数を（分岐）と表現して区別している．VaaS 試行と従来のテスト法でテスト対象行数が異なるのは，VaaS で結果を出せず途中で断念したものを除いたためである．また項目「作業時間」も同様で VaaS 試行時に結果を出せず断念したテストメソッドに費やした時間は含めなかった．

サブシステム A のテストで従来の工数とで生産性の比較をすると，17（行）/ 43．2（行）＝0．39 となり，VaaS の生産性は 2/5 ということができる．

最初に実施したサブシステム A ではテスト実施のための準備（スタブ作成、入力ファイルの作成等）で時間がかかった．そこでスタブを再利用できたサブシステム B の値と比べてみることにした（表２）．

試行方法(試行対象)	行数	作業時間	1時間あたりのテスト実施行数	テストケース数
VaaS（サブシステムA）	189行	11時間8分(VaaS実行時間を除く)	189(行)/11.13(H) ≒ **17行/H**	6073(パス) 78(分岐)
VaaS（サブシステムB）	57行	ドライバ作成＋結果確認時間＝**50分(0.83H)**	57(行)/0.83(H) ≒ **68.7行/H**	144(パス) 27(分岐)

表２　モジュールごとの VaaS 試行工数の比較表

Table 2 The table of comparing the test costs between each module.

時間的に前で試行し，ドライバ，スタブを作成しながら試行したサブシステム A と，時間的に後で試行し，以前作成した環境を再利用しながら試行したサブシステム B とでは生産性に差があることがわかる．そこでそれまで作成したスタブ，ドライバをそのまま横展開して使うことができたとしてサブシステム B の工数で従来テスト法と比較すると VaaS の生産性は 68．7（行）/43．2（H）＝1．59 倍となる．この値は試行対象システムの開発関係者の一部には妥当との評価を得ている．

4.2　テスト実行の網羅率

上記モジュールのテスト実行の網羅率について調査した結果を以下に示す（表３）．従来テストの中にはテスト内容が不明のものもあり，比較可能なメソッドに限定して行った．

試行方法	テストカバレッジ（テストが存在するものに限定して比較）
VaaS（サブシステムA）	98%（デッドコードを含む）
従来のテスト法	96%

表 10　VaaS 試行と従来テストとのテスト網羅率比較

Table 11　The table of comparing the test coverage between VaaS and original test.

VaaS でも従来テストでも対象モジュールにデッドコードを含むため 100%にはならなかった．従来テストと VaaS の網羅率で差が出ている原因は異常終了系テストが従来テストで 1 か所抜けていたため．上記の結果からテスト実行

の網羅率は従来のテストに比べて同等またはそれ以上ということができる． このことから VaaS は業務ロジックを満たすテストデータ，テストケースの生成が可能であると結論づけることができる．

4.3 テスト工数以外における VaaS の利点

人が作成する従来のテスト作成では業務ロジックに基づいて作成されるので自分が知っている例外以外にはテスト作成がなされない．今回の VaaS による自動生成されたテストケースにおいては VaaS 試行中に NullPointerException，IndexOutOfBound 等，想定外の例外を 15 件（(試行全体では 20 件）発見することができた．このことから VaaS テストケース自動生成が想定外の例外発見に貢献することが分かった．

5. VaaS の課題

5.1 ドライバ・スタブ作成の課題

ドライバ･スタブは自動生成だけでは業務ロジックを網羅するようなものは作成できず手動で修正が必要となる．2.2 で指摘した Java の Generic 型のように自動生成が難しく未対応のものもある．また工数の問題として 1 つのフレームワークを使っている限りドライバ・スタブ作成はいくつか作成すればそれ以後はほぼ同等のものを使いまわすことで途中からは生産性が上がることはわかった．しかし作成に関しては

- プロジェクトのフレームワークの知識
- ドライバ・スタブ自動生成後の修正ノウハウ

が必要となるため VaaS によるテスト実施者にそれらの知識が前提となり，またその工数を組んでおくことが必要である． その負担軽減のためには VaaS による検証を行うプロジェクトのフレームワーク開発者が再利用可能なスタブおよびドライバを事前に開発し，テスト実施者に提供することが考えられる．

5.2 シンボル実行の潜在的限界

今回の試行の中でVaaSが実行時間内で終了しなかったり，メモリの限界により試行を断念せざるをえなかったりしたメソッドがいくつかあった.これらはシンボリック実行の潜在的な課題でもある.これらに対しては具体値による concolic 実行やその応用である DART(Dynamic automated random testing[8])などの探索手法を使うことが考えられる．

6. まとめ

我々が開発中の Java 向けのシンボリック実行による検証システム（VaaS）を使って実際の業務システムを対象に検証を行った際の環境構築および実行結果をまとめたものである．業務システム中の特に業務ロジックを含む業務クラスを対象とし，ドライバ・スタブを作成して VaaS を適用し，網羅的なテストケース作成にかかる作業時間を測定した．その結果 VaaS による検証時と通常のテストを比較してテスト実行の網羅率がほぼ同等である時，生産性は VaaS で行うと通常で行われるテストに比べて約 2/5 になるが，あらかじめスタブを用意しておければ生産性は通常テストに比べて 1．59 倍になるという結果を得ることができた．また VaaS は想定外の例外を 15 件発見し，例外発見にも有効であることが確認された．

謝辞 論文作成にご協力頂いた皆様に，謹んで感謝の意を表する．

参考文献

1) J.C. King.: Symbolic Execution and Program Testing, Communications of the ACM, Vol.19(7), pp.385–394, July 1976.
2) C. Cadar et al.: Symbolic Execution for Software Testing in Practice: Preliminary Assessment, *Proc. 33rd Int'l Conf. Software Eng.* (ICSE11), ACM Press, 2011, pp. 1066-1071.
3) Java PathFinder, http://babelfish.arc.nasa.gov/trac/jpf
4) J. Ginbayashi, T. Uehara, K. Munakata, K. Yabuta.: New Approach to Application Software Quality Verification, *Fujitsu Sci. Tech. J.*, Vol.46(2), pp.158–167, April 2010.
5) Fujitsu Labs. of Europe Ltd.: Validation-as-a-Service – Advancing Java Testing, *Brochure of 2011 Fujitsu Tech. Forum (Autumn)*, October 2011.
6) I. Ghosh, N. Shafiei, G. Li, and WF. Chiang. JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings, *Proc. Int'l Conf. Software Eng.* (ICSE13), ACM Press, 2013, ,pages 992-1001.
7) JUnit: Programmer-Oriented Testing Framework for Java, http://junit.org/
8) P.Godefroid, N. Klarlund, and K. Sen.: Dart: Directed automated random testing, PLDI '05, pages 213-223, New York, NY, USA, 2005. ACM.