シンボリック実行を活用した網羅的テストケース生成

Exhaustive Test-case Generation using Symbolic Execution

● 上原忠弘

あらまし

現在の企業システムのソフトウェア開発において，ソフトウェアテストは3割から5割に及ぶ工数がかかっており，大きな課題である。また，テスト駆動開発や継続的インテグレーションなどの，テストの自動化を前提とした開発パラダイムが主流になりつつあるなど，その重要性はより一層増している。富士通研究所は，ソフトウェアテストの中でも，品質の確保や作業工数の削減において最も大きなウェートを占めるテストケース生成をメインターゲットに研究開発を進めてきた。その研究成果は，富士通の業務アプリケーション開発環境であるFUJITSU Software Interdevelop Designerのテスト機能として実用化されている。

本稿では，筆者らが早くからその可能性に着目し，現在学術界でも研究トレンドとなっているシンボリック実行を活用した網羅的テストケース生成を紹介する。そして，プログラムの単体テストと改版時のリグレッションテストの効率化をターゲットに，実用化に向けた三つの課題と，それらを解決するために採用したアプローチについて述べる。更に，実用的なソフトウェア資産に対する適用評価の事例も併せて紹介する。

Abstract

Software testing has been one of the major challenges in the development of software for enterprise systems because it takes between 30% and 50% of the total development cost required. Meanwhile, such testing has increasingly become important, because the upcoming paradigm for ICT system development, such as test-driven development and continuous integration, is directed at automated testing. Over a period, Fujitsu Laboratories has pursued R&D of software testing, mainly focusing on the test-case generation, which has a major impact on ensuring software quality and minimizing the development cost required. Such test-case generation has been successfully realized in the test function of the FUJITSU Software Interdevelop Designer—Fujitsu's business application development platform. We have recognized great potential in symbolic execution since the early days, a method which has become a popular academic research topic today. This paper presents an exhaustive test-case generation technology that utilizes symbolic execution. We describe three challenges in its practical application, aiming to improve the efficiency of program unit testing and regression tests for version upgrades. That is followed by accounts of the approaches adopted to overcome these challenges. This paper also describes some cases in which we evaluated its application to working software assets.

ま え が き

ソフトウェアテストは，ソフトウェア工学における重要なテーマの一つとして古くから認識され，数多くの技術革新が産学双方からなされてきた。しかし，現在の企業システムのソフトウェア開発においても，3割から5割に及ぶ工数がかかっており，依然として大きな課題である。また，プログラム開発に先んじてテスト作成を行うテスト駆動開発や，自動化されたテストを日々継続的に実施して問題の早期検出・改善を図る継続的インテグレーションなど，テストの自動化を前提としたソフトウェア開発パラダイムが主流になりつつあり，ソフトウェアテストはますます重要になってきている。

業務アプリケーション開発のテスト工程において，現在実用化されているテスト自動化技術を**表-1**に示す。テスト工程を縦軸，各工程における作業を横軸として分類している。[1] 自動化が進んでいる作業は，テスト実行，テスト管理の領域である。テスト実行に関しては，JUnitに代表される単体テストフレームワークが開発言語ごとに整備されている。結合テストやシステムテストにおいても，一旦テスト手順を記録したあとでそれを自動再生する，キャプチャー＆リプレイ型のツールが商用ツールやオープンソースソフトウェアとして提供されるなど，モダンな開発手法には必須のツールになっている。

一方で，テスト分析，テスト設計，およびテスト実装の領域，すなわちテストケースやテストデータの生成は，いまだ自動化が進んでいない領域である。直交表[2]やAll-Pair法[2]をベースとした組合せ生成ツールなどがあるが，どのようなテスト条件を抽出する必要があるかといったテストの品質に直結する部分は，人の役割となっている。富士通研究所は，このテスト条件をいかに抽出し，テスト品質を向上・均質化するかについて研究開発に取り組んできた。

本稿では，筆者らがこれまで研究を行ってきたテストケース生成に関する技術を紹介する。まず，テスト技術のコアとなるシンボリック実行について紹介する。次に，プログラム単体に対するテストケースを生成する技術を述べ，更にプログラム改版時のリグレッションテストへの応用技術を紹介する。最後に，今後の取組みと展望を述べる。

シンボリック実行によるテストケース生成

シンボリック実行は，プログラムが実行し得るパスを網羅的に抽出する技術である。1970年代から研究されている技術であり，[3] 学術的な分野では多くの成果があったが，その計算コストの高さにより実用化が進んでいなかった。しかし2000年代になり，制約充足問題（与えられた方程式を満たす解を見つける問題）を高速に解くSAT（satisfiability problem）/SMT（satisfiability modulo theories problem）技術[4]の急速な進歩をきっかけに，実用的な規模のプログラムに対して，シンボリック実行による解析が現実的な時間でできるようになった。現在では，ソフトウェア工学におけるテスト技術の研究において，シンボリック実行を用いたテストケース生成は，主要な技術領域として多くの研究者が取り組んでいる。[5] 富士通研究所は，早くからシンボリック実行のテストへの応用の可能性に着目し，研究を進めてきた。[6] その中の一つで

表-1　テスト自動化技術と研究ターゲット

	テストケース生成		テストデータ生成		
テスト工程	テスト分析	テスト設計	テスト実装	テスト実行	テスト管理
単体テスト		カバレッジ測定ツール		ユニットテストツール	ベンダーツール，Jenkinsプラグインなど
結合テスト		状態遷移テストツール，組合テストツール		キャプチャー＆リプレイツール	
システムテスト	研究ターゲット			性能テストツール セキュリティテストツール	
受入テスト					

あるプログラム単体に対するテストケース生成については実用レベルまで進み，富士通の業務アプリケーション開発環境であるFUJITSU Software Interdevelop Designerのテスト機能として提供している。(7)

ここでは，テストとシンボリック実行の原理について説明する。シンボリック実行は，入力をシンボルという具体値を伴わない値として扱う実行方式である。プログラム中の代入や演算により伝搬したシンボルに対して条件分岐を判定する際，判定が真，あるいは偽になる場合の2ケースに場合分けしてシンボルに対する条件式として記録し，それぞれ実行していく。その際，それまでに記録していたシンボルの条件式（パス条件）を考慮して，それぞれの場合分けがあり得るかについてSMTソルバを用いて判定し，あり得る場合のみその分岐の処理を継続する。この手順を繰り返すことで，そのプログラムが実行し得るパスを全て抽出できる。

シンボリック実行のテストケース生成への応用は，抽出したパスをテストケースとして扱うことである。SMTソルバは，与えられた条件式があり得る（制約充足可能な）場合，そのシンボルに対する具体値の一例を出力する。この値をシンボリック実行の入力データとして活用することで，実行可能なテストケースを得られる。

図-1を例に説明する。引数*x*をシンボルSym*x*とした場合，テスト対象プログラムの2行目と4行目にシンボル変数を評価する分岐があるため，パス条件は以下になる。

（1）（Sym*x*＝123）＆（Sym*x*＋1＜0）
（2）（Sym*x*＝123）＆ not（Sym*x*＋1＜0）
（3）not（Sym*x*＝123）＆（Sym*x*＜0）
（4）not（Sym*x*＝123）＆ not（Sym*x*＜0）

しかし，（1）についてはこれを満たすSym*x*が存在しないため，実行できないパスと判断し，テストケースは出力されない。一方，（2），（3），（4）のパス条件については，SMTソルバが解の一例として123，−2，3を生成し，これをシンボリック実行の入力データとする。つまり，図-1のように三つのパスが抽出され，変数*x*への入力データとして実行可能なテストケースが生成される。これにより，図-1中のプログラムが実行し得るパスの全てを，テストケースとして抽出できる。ただし，今日のシンボリック実行であっても，全てのプログラムに対してテストケースが抽出できるわけではない。その課題としては，主に以下の3点が挙げられる。

（1）課題1：データ型の制限

シンボルとして扱えるデータ型は数値型と文字列型に限られ，配列やリストなど可変長のデータ構造は扱えない。このため，固定長のデータ構造

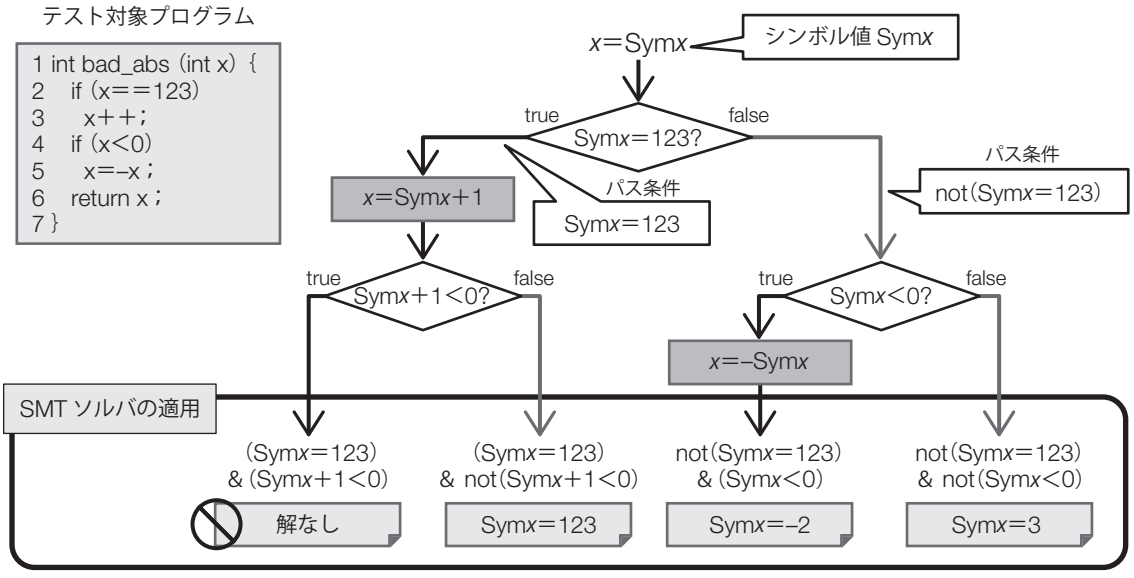

図-1　シンボリック実行の仕組み

を複数用意するなど，シンボリック実行とは別の枠組みでバリエーションを確保する必要がある。オブジェクト型のデータについても同様の工夫が必要である。

（2）課題2：解析時間の増大

プログラムの規模が大きくなると，解析時間が爆発的に増大する。条件分岐の数をnとしたときに，プログラムの実行し得るパス数は最大で2のn乗となり，実用規模のプログラムの解析は事実上不可能である。この問題に対し，DART（Directed automated random testing）[8]など少ないパスで多くの分岐条件をカバーすることを指向したアルゴリズムを用いることで，実用的な時間での解析を可能にしている。しかし，実際はアルゴリズムごとにカバーしづらいプログラム構造が存在し，探索回数の制限やタイムアウトなど解析を途中で打ち切る手法を併用することが多い。その場合は，テストケースとしてカバーできていない条件分岐が残ることとなる。

（3）課題3：外界の振舞いのモデル化

テスト対象プログラムが依存する外界の振舞いをどのようにモデル化するかが課題である。例えば，シンボリック実行を行う多くの解析エンジンは，特定の開発言語で記述されたプログラムのみを解析できる。一方で，そのプログラムが呼び出すライブラリや，その先のネットワーク，OS，DBMS（Database Management System）などの振舞いまでは解析できない。今日のソフトウェアは，一つのプログラムで処理が完結することはほとんどなく，プログラムが依存する外界の振舞いを考慮して解析を進めることが必須となる。

上記の課題をどのように解決していくかが，実用化に向けた取組みとなる。

単体テストの効率化

シンボリック実行によるテストケース生成の応用として，最初に考えられるのは単体テストへの応用である。単体テストは，業務アプリケーション開発において行われる細粒度のテストである。そのテストの網羅性の尺度としては，プログラム全体に対してテストケースが実行する部分の割合を表すコードカバレッジを用いることが多い。シンボリック実行によるテストケース生成は，コードカバレッジの向上を指向するため，単体テストとの親和性が高い。単体テストの作業のうち，テスト分析・テスト設計（テストケースの洗い出し），テスト実装（テストデータの準備），およびテスト実行が自動化され，実行結果の確認のみに開発者が注力できるため，効率化が図れる。

単体テストへの応用に向けて残された課題は，前述の三つの課題をどう解決するかにある。筆者らは，これらの課題に対するアプローチとして，テスト対象プログラム自身の型決めを行い，その外側をシンボリック実行解析のためのダミープログラム（スタブ）としてモデル化する方針を採った。今日の業務アプリケーション開発では，システムの制御部分を担当するフレームワークと，個別の業務ロジックを実装するプログラムとを明確に分けて開発する。開発量の比率としては業務ロジック部分の開発が格段に大きく，テストにかかる工数もそれに合わせて大きくなっている。筆者らはこの点に着目し，業務ロジックプログラムにフォーカスしてテストケースを生成することとした。フレームワークを解析せずに，業務ロジックを呼び出すドライバと，業務ロジックから呼び出されるフレームワークや共通部品のAPI（Application Programming Interface）のスタブを準備する。それらと業務ロジックプログラムを合わせたシンボリック実行によって解析することで，課題の解決・軽減を図る（**図-2**）。

（1）課題1に対する解決

可変長のデータ構造はシンボルとして扱えないため，シンボリック実行とは別の仕組みでテストのバリエーションを用意する必要がある。筆者らはこの課題に対応するために，配列やリストの長さ，オブジェクトの有無に関するバリエーションをあらかじめドライバやスタブの内部に用意する方式を採った。例えば，**図-3**に示すテスト対象プログラムmethodAが与えられると，生成されるドライバメソッドtestFuncはlist変数の要素数が0と1の場合の二つのバリエーションを持つ。これらはboolean型の変数cの値によって要素数を切り替える構造で生成される。実際に，シンボリック実行エンジンがカバーすべき分岐として認識するのは，変数cに対する分岐である。その分岐条件の真偽をカバーする2ケースを生成することで，結

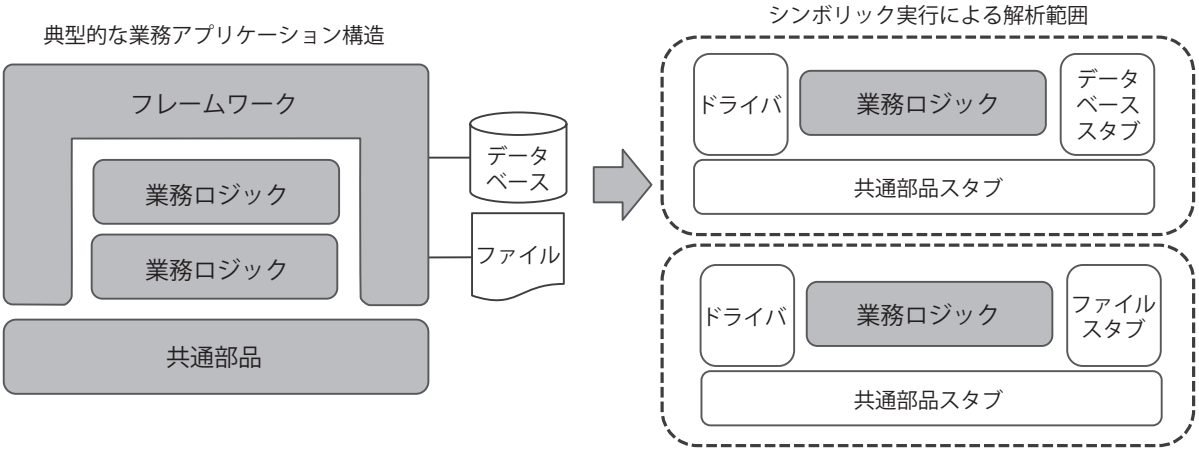

図-2　フレームワークによるドライバ・スタブ型決め

テスト対象プログラム

```
static void methodA(OrderBean order){
  List items = order.getItems();

  for (Item item: items)
...
}
```

ドライバスタブ生成

ドライバ

```
Public class SymbolicDriver{
  static boolean c;
  public void testFunc(){
    OrderBean p = new OrderBean();
    ArrayList l = new ArrayList();
    if (c){
      l.add(new Item());//要素数が1
    } else {
      //要素数が0
    }
    p.setItems(l);
    Target.methodA(p);
}
```

図-3　生成されるドライバの例

果的にlistオブジェクトの要素数が0と1の場合の2ケースが生成される形となる。

このように，あらかじめ決められたバリエーションのみをドライバやスタブとして生成するため，例えば，要素数が5の場合のみ特別な処理をするようなプログラムに対しては，十分なテストケースを生成できない。しかし，要素数に応じて処理を繰り返すなどの多くの業務ロジックに対しては，決められたバリエーションのみでカバーできる。カバーできなかった部分については，人手でテストケースを追加する方法で対応する。

（2）課題2に対する解決

筆者らは，解析時間の爆発的な増大を防ぐために，DARTなどのコードカバレッジ向上を指向した探索アルゴリズムを採用している。しかし，forなどのループを含む構造に対して，分岐を網羅するテストケースを発見できない場合がある。そのような状況を極力排除するために，ループ構造で処理を行うロジックを部品化し，フレームワークや共通部品のAPIとして実装する。一方で，シンボリック実行時には，それらのAPIはループ構造を持たないスタブで置き換えて解析することで，ループ構造の解析を極力減らすことができる。

（3）課題3に対する解決

業務アプリケーション開発におけるフレームワークの役割は，開発者を業務ロジックの実装に注力させることである。そのため，ネットワークやデータベースなどの外界へのアクセスは，フレームワークがAPIとして提供している。外界をモデル化することはフレームワークAPIをスタブ化することにほかならず，フレームワークAPIの振る舞いを再現するスタブをあらかじめ用意することで実現できる。

筆者らのアプローチの効果を検証するために，実プロジェクト資産に対し，研究員による検証実験を行った。[9] 実験では，以下の手順で作業を行い，生成テストケースの網羅性，および各作業の工数を評価した。

表-2　テストケース生成による工数削減

	テスト対象コード行数	テスト時間	1000行あたりのテスト時間	テスト工数の実績値に対する比率
プロジェクトでの実績値	569行	13.0 h	22.8 h	―
1. フレームワークスタブの作成 2. 業務Aに対するテスト生成・実行	189行	11.8 h	62.4 h	173％増 （62.4/22.8＝273%）
3. 業務Bに対するテスト生成・実行	57行	0.8 h	14.0 h	39％減 （14.0/22.8＝61%）

1. フレームワークスタブの作成
2. 業務Aに対するテストケース生成・実行
3. 業務Bに対するテストケース生成・実行

（1）網羅性の評価

業務Aおよび業務Bの26機能中，23機能についてコードカバレッジ100％のテストケースが生成された。残りの3機能について，カバーできていない部分を分析したところ，どのようなテストケースでも通過しないデッドコードであることが分かった。

（2）作業工数の評価

各作業の工数を**表-2**に示す。実験では，手順1, 2の作業が連続して行われたため，それらを併せた工数となっている。また，同プロジェクト資産を開発したときの実績値と比較している。なお，この開発プロジェクトでは，テストケースは人手で抽出している。

フレームワーク側のスタブを作成する部分で工数がかかっているために，業務Aのテスト工数は実績値に対して173％増（2.73倍）となっている。しかし，作成したフレームワークスタブを流用して行った業務Bのテスト工数は，実績値に比べて39％の工数削減となっている。この結果から，フレームワークスタブをフレームワーク開発担当者が一度作成しておけば，業務ロジックプログラムを開発する担当者のテスト工数を3 ～ 4割削減できると予想している。

リグレッションテストへの適用

シンボリック実行によるテストケース生成のもう一つの応用として有望と考えているのが，リグレッションテストへの応用である。リグレッションテストとはプログラムの改版時に行うものであり，改版前のプログラムと同じ動作をするかを確認するテストである。具体的には，あるテスト入力に対して，改版前のプログラムが返していた出力と同じものが改版後のプログラムでも返されるかを確認する作業である。リグレッションテストにおいても，網羅的なテストケースを抽出することは重要な課題である。特にプログラムの改版時には，そのプログラムの開発者がいない場合も多く，よりテストケースの洗い出しが困難になる。

筆者らが提案する自動化手法では，シンボリック実行によって改版前のプログラムに対して網羅的なテストケースを生成するとともに，そのテストケースを実行したときの出力をテストケースの期待値として記録する。このテストケースを改版後のプログラムで実行することで，手作業を行うことなくリグレッションテストを実施できる。単体テストでは，テストの実行結果の確認は自動化できずに開発者の作業として残っていた。しかし，リグレッションテストでは結果確認も自動化でき，劇的な効率化が期待できる。

有効性を確認するため，ある製品機能（C言語，約19 000ステップ）の再構築において，上記手法の検証実験を実施した。[10] この実験では，開発者によるテストを実施した後に，シンボリック実行によるテストケース生成・実行を行った。開発者によるテストで27件のバグを検出・修正した後に本手法でテストしたところ，更に5件のバグを検出できた。この結果は，シンボリック実行による網羅的な探索によって，人手のテストでは見つけることができなかったレアケースのバグを自動的に検出できたことを示すものである。

む　す　び

本稿では，富士通研究所が取り組んできたシンボリック実行による網羅的テストケース生成について，実用化に向けた課題とその解決，および

適用効果について述べた。現在，Java，C/C++，COBOL[11]の各環境での動作について，更なる効率化に向けた研究を進めるとともに，近年のモバイル化に対応すべく，JavaScriptとHTMLを組み合わせたアプリケーションに対するテストケース生成の研究開発を進めている。[12]更に，単体テストにとどまらず，モジュールを組み合わせて提供されるシステム機能を検証する結合テストの領域にも展開していく予定である。

参考文献

(1) ソフトウェアテスト技術振興協会ASTERテストツールWG:テストツールまるわかりガイド（入門編）. http://aster.or.jp/business/testtool_wg/pdf/Testtool_beginningGuide_Version1.0.0.pdf

(2) 秋山浩一：特集 ソフトウェアテストの最新動向 3. 組合せテストの設計. 情報処理, Vol.49, No.2, p.140-146 (2008).

(3) 玉井哲雄ほか：記号実行システム．情報処理, Vol.23，No.1，p.18-28（1982）.

(4) 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, p.24-35 (2010). https://www.jstage.jst.go.jp/article/jssst/27/3/27_3_3_24/_pdf

(5) A. Orso et al. ：Software Testing：A Research Travelogue（2000-2014）．36th International Conference on Software Engineering（2014）.

(6) 富士通：Webアプリケーション向け品質保証の基礎技術を開発. http://pr.fujitsu.com/jp/news/2008/04/4-1.html

(7) 富士通：業務プログラム開発支援ツール「Interdevelop Designer」販売開始. http://pr.fujitsu.com/jp/news/2014/08/28-1.html

(8) P. Godefroid et al. ：DART：Directed Automated Random Testing．Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation（PLDI '05），p.213-223.

(9) 片山朝子ほか：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．ソフトウェアエンジニアリングシンポジウム2013（SES2013）.

(10) S. Tokumoto et al. ：Enhancing Symbolic Execution to Test the Compatibility of Re-engineered Industrial Software. The 19th Asia-Pacific Software Engineering Conference（APSEC 2012）.

(11) 前田芳晴ほか：COBOLシンボリック実行によるテストケース生成．第19回ソフトウェア工学の基礎ワークショップ（FOSE 2012）.

(12) H. Tanida et al. ：Automatic Unit Test Generation and Execution for JavaScript Program through Symbolic Execution. The 9th International Conference on Software Engineering Advances（ICSEA2014）.

著者紹介

上原忠弘（うえはら　ただひろ）

システム技術研究所
サービス指向型ソフトウェア開発技術プロジェクト 所属
現在，ソフトウェアテスト，WebAPI開発技術関連の研究に従事。