拡張有限状態機械を用いた運用プロファイルベースドテスト法のフレームワーク

高木 智彦[1,a)]　八重樫 理人[1]　古川 善吾[1]

概要：拡張有限状態機械に基づく運用プロファイルから，usage distribution coverage と N スイッチ網羅率ができるだけ大きいテストケースを生成するためのソフトウェアテストのフレームワークを提案する．

The Framework of Operational Profile-Based Testing Using Extended Finite State Machines

Tomohiko Takagi[1,a)]　Rihito Yaegashi[1]　Zengo Furukawa[1]

Abstract: This paper shows a software testing framework in which an operational profile that is based on an EFSM (extended finite state machine) automatically produces test cases to achieve high usage distribution coverage and N-switch coverage.

1. はじめに

MBT（model-based testing）はテスト対象ソフトウェアの期待される振舞いを表すモデルに基づいてテストケースを設計するソフトウェアテストである．モデルの構成要素を網羅するテストケースを設計することによって満遍なくフォールトを洗い出すことを目的とした手法が数多く提案されている [1], [2]．その一方で，ソフトウェア信頼性を評価したり，ソフトウェア信頼性に深刻な影響を与えるフォールトを重点的に発見することの重要性も認識されており，網羅性を指向するのではなくそのようなソフトウェア信頼性の観点でテストケースを設計する OPBT（operational profile-based testing）[3] が注目されている．

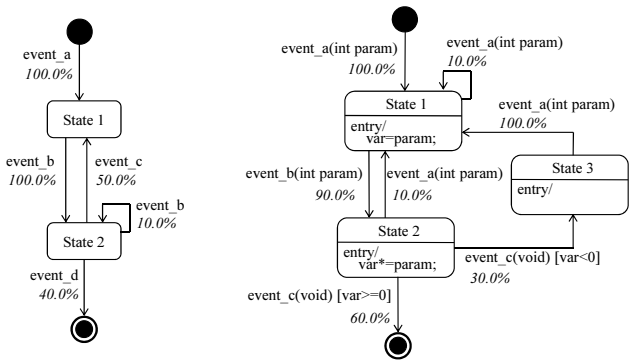
OPBT では，テスト対象ソフトウェアの振舞いを FSM（finite state machine）として定義し，これにユーザの利用特性を表す確率分布を付加したモデルを用いる．このモデルは運用プロファイル（operational profile）と呼ばれる．OPBT におけるテストケースは，運用プロファイル上の開始状態で始まり終了状態で終わる遷移列であり，確率分布に基づいてランダムに生成される．我々は，OPBT の有効性を高めるために，GA（genetic algorithm）を用いてテストケースを最適化する手法を提案した [4]．この手法では，テストに投入可能な労力をあらかじめ指定しておくと，その範囲内で UDC（usage distribution coverage）[5] と呼ばれるメトリクスの値ができるだけ大きくなるようにテストケースを選りすぐって生成するので，逼迫したテスト工程においても OPBT を効果的に導入することが可能である．

OPBT は実際のソフトウェア開発に適用され成果を挙げている [6] ものの，以下に示す 2 つの課題が存在する．

- FSM は単純なモデルであるため，テスト対象ソフトウェアの複雑な振舞いを運用プロファイルによって表現することが困難な場合がある．
- 使用される可能性がほとんどない機能はテストする必要がないという考え方に基づいているが，使用頻度が低くてもテストすべき重要な機能は存在する．

本稿ではこれらの問題を解決するために，文献 [4] の手法を発展させた新たな OPBT のフレームワーク（手法とその手法を実現するテスト環境）を示す．本フレームワークでは，EFSM（extended finite state machine）に基づく運用プロファイルである拡張運用プロファイル（extended operational profile）を作成する．FSM よりも表現力が高い EFSM を導入することによって，1 つ目の課題を解決できる．そして UDC だけでなく，長さ $(N+1)$ の遷移列を網羅することを目的とした N スイッチ網羅（$N \geq 0$）[7] をも考慮したテストケースを拡張運用プロファイルから生成

[1] 香川大学工学部
Faculty of Engineering, Kagawa University
[a)] takagi@eng.kagawa-u.ac.jp



(a) 従来の運用プロファイル　　(b) 本研究の拡張運用プロファイル

図 1　従来の運用プロファイルと拡張運用プロファイルの単純な例
Fig. 1　Example of conventional/extended operational profiles.

する．UDC によって頻度の高い使われ方を重点的にテストするだけでなく，N スイッチ網羅によって頻度の低い部分についてもテストするので，2 つ目の課題を解決できる．

本稿の構成は以下のとおりである．まず 2 章で従来研究について整理しつつ本フレームワークを構成する基本概念を述べる．次に 3 章で本研究が提案する OPBT の手順とテストケース生成アルゴリズムを示す．そして 4 章では本フレームワークを商用ソフトウェアに適用した事例に基づき有効性を議論する．最後に 5 章で本研究を総括する．

2. 基本概念

本章では，運用プロファイルやテストケース，メトリクスなど本フレームワークを構成する基本概念について従来研究をふまえて整理する．

2.1 運用プロファイル

従来の運用プロファイルは，テスト対象ソフトウェアの振舞いを表す FSM に，ユーザの利用特性を表す確率分布を付加したモデルである．ノードはテスト対象ソフトウェアの状態，アークはイベント（ユーザの操作や外部システムからの入力など）によるテスト対象ソフトウェアの状態の遷移を表す．FSM はテスト対象ソフトウェアの要求仕様に基づいて作成する．また，確率分布はテスト対象に類似したソフトウェアの運用データや技術者の予測などに基づいて決定する．運用プロファイルの単純な例を**図 1** (a) に示す．たとえば，State 2 では event_b，event_c，event_d がそれぞれ 10%，50%，40%の確率で生起することを示している．従来の OPBT では，この確率にしたがってランダムに遷移を選択していくことで，開始状態から終了状態に至る 1 本の遷移列を 1 つのテストケースとして生成する．テストケースの実行に要する労力を精密に考慮してテストケースを生成するために，各遷移のテスト実行に必要な労力を数値化して運用プロファイル中に記述することもできる [4]．この数値化された労力の記述を労力分布と呼ぶ．

FSM に基づく従来の運用プロファイルの問題は，テスト対象ソフトウェアの複雑な振舞いを記述するのが困難な場合があるという点である．たとえば，テスト対象ソフトウェアの状態を特徴付ける変数が 2 つあり，それぞれ 10 通りの値を取り得るとすると，それだけで 100 個の状態を定義することになる．大規模で複雑な FSM の作成には相応の労力が必要であり，近年の逼迫したテスト工程においてそのような労力をかけることは現実的ではない．そこで本稿では EFSM に基づく運用プロファイルである拡張運用プロファイル[*1]を提案する．EFSM は，以下の 2 点で FSM よりも優れた表現力を備えている．

- イベントパラメータ（イベントの引数）をもち，状態や遷移のアクションから参照することができる．イベントパラメータは，ユーザの操作や外部システムからの入力に伴うデータを表すもので，たとえば，インターネットショッピングシステムにおいてユーザが発注操作を行う際の商品 ID や発注数量などに相当する．
- テスト対象ソフトウェアの状態を特徴付ける変数をもち，状態や遷移のアクションにおいてその変数の値を参照したり更新したりすることができる．さらに，その変数を用いて，遷移の発火条件であるガードを記述できる．アクションやガードはプログラミング言語あるいは何らかの形式言語で記述される．

EFSM の有用性は開発現場で広く利用されていることからも明らかである．EFSM に確率分布や労力分布を付加したものが拡張運用プロファイルであるので，EFSM の表現力の高さはそのまま拡張運用プロファイルにも生かされる．ゆえに，拡張運用プロファイルによって従来では表現が困難であったテスト対象ソフトウェアの複雑な振舞いを表現し，そのような振舞いに関するテストケースを OPBT において効果的に生成できるようになる．拡張運用プロファイルの例を図 1 (b) に示す．この拡張運用プロファイルは変数 var をもち，State 1 と State 2 の entry アクションにおいては，当該状態への遷移のトリガとなったイベントのイベントパラメータを参照した上で変数 var の更新を行う．たとえば，State 2 の entry アクションは event_b の int 型イベントパラメータ param を参照し，その値を変数 var の値に乗算して変数 var に格納する．このようなイベントパラメータや変数などの概念が加わることによって表現力が強化される反面，拡張運用プロファイルには従来の運用プロファイルにはなかった実行可能性の問題が生じることになる．たとえば，図 1 (b) の State 2 においては，event_a と event_c がそれぞれ 10%と 90%の確率で生起し，event_c については，ガード var>=0 を満たして終了状態に遷移す

[*1] 文献 [4] では，労力分布を付加した運用プロファイルを拡張運用プロファイルと呼んでいる．本稿ではこれを EFSM の導入によってさらに拡張したものを拡張運用プロファイルと呼んでいる点に注意されたい．

る確率が 60%，ガード var<0 を満たして State 3 に遷移する確率が 30%であることが示されている．仮に「開始状態 → event_a(−10) → State 1 → event_b(1) → State 2 → event_c() → 終了状態」という遷移列が与えられた場合，これの最後の遷移がガード var>=0 を満たさないので実行不可能である．しかしながら，現在の状態とイベントの生起確率のみに基づいて遷移を選択する従来の OPBT のテストケース生成アルゴリズムでは，このような実行不可能な遷移列が生成される可能性がある．この問題を解決する新たなアルゴリズムを 3.2 節で提案する．

2.2 テストケースとそのメトリクス

本研究におけるテストケースは，拡張運用プロファイル上の開始状態から始まり終了状態で終わる遷移列である．各遷移は，(a) 遷移元状態，(b) 遷移元状態における変数の値，(c) イベント，(d) イベントパラメータの値，(e) 遷移先状態，(f) 遷移先状態における変数の値から構成される．(c)(d) をテストデータ，(a)(b)(e)(f) をテストオラクルと呼ぶ．(a)(b) はテストデータ適用の事前条件，また (e)(f) はテストデータ適用の事後条件とみなすことができる．ある遷移の事後条件は，その直後の遷移の事前条件と一致しなければならない．

一般的に，テストケースは何らかのメトリクスに基づいて設計される．本研究を含む FSM あるいは EFSM に基づくテスト技法におけるテストケースの設計とは，開始状態から終了状態に至るまでの間に実行する遷移を選択することを意味する．通常，単一のテストケースだけで充分なテストを行うことはできないので，メトリクスに基づいてテストケース集合（テストスイート）が構成される．また，メトリクスは，テストケースの設計やテストの質の評価の基準となるものであり，従来の FSM あるいは EFSM に基づく網羅性を指向したテスト技法では N スイッチ網羅が広く利用されている．N の値を大きくするにつれて，網羅すべき遷移列の数，ひいては実行すべきテストケースの数は急激に増加する．加えて，EFSM では前節で述べたように実行不可能な遷移列が存在する場合があるので，$N > 0$ ではすべての遷移列を網羅することは必ずしも求められないが，できるだけ高い網羅率を達成することが望ましい．一方，1 章ですでに述べたように，従来の OPBT はソフトウェア信頼性に注目した手法であるため，MTTF（mean time to failure）や Kullback 判別式などがメトリクスとして用いられてきた [3], [8]．本研究に導入する UDC [5] は，テストケース集合がユーザの利用方法をどれだけ網羅するかを表すものである．n 個のテストケースから構成されるテストケース集合を $ts = \{tc_1, tc_2, \cdots, tc_n\}$，テストケース tc の長さ（すなわち遷移数）を $\sharp tc$，tc の先頭から j 番目の遷移確率を $p(tc[j])$ と表すとき，ts の UDC は以下の式で求められる．

$$udc(ts) = \sum_{i=1}^{n} \prod_{j=1}^{\sharp tc_i} p(tc_i[j])$$

ただし，重複するテストケースについては重複して加算しない．

テストケースを構成する遷移の遷移確率が高いほど UDC は高くなる．OPBT では UDC が高いテストケース集合ほど質の高いテストができるといえるので，できるだけ UDC の高いテストケース集合を生成する必要がある．テストケース集合を構成するテストケースの量が多いほど UDC は大きくなるものの，テスト工程に割り当てることのできる労力には限りがある．この問題を解決するために文献 [4] では，UDC ができるだけ大きく，かつ実行に要する労力があらかじめ指定される上限を超えないテストケース集合を生成する手法が提案され，その有効性が示されている．なお，テストケース集合 ts の実行に要する労力 $effort(ts)$ は，テストケース tc の先頭から j 番目の遷移の労力を $e(tc[j])$ とすると以下の式によって定義される．

$$effort(ts) = \sum_{i=1}^{n} \sum_{j=1}^{\sharp tc_i} e(tc_i[j])$$

運用プロファイルにおいて労力分布が定義されていない場合は，各遷移の労力値を一律で 1 とみなすので以下の式が成り立つ．

$$effort(ts) = \sum_{i=1}^{n} \sharp tc_i$$

従来の OPBT では，N スイッチ網羅のような網羅性を指向したメトリクスは用いられていない．ゆえに，OPBT において生成されるテストケース集合では効果的に網羅率を高めることができず，しばしば OPBT の欠点として指摘される．そこで本研究では UDC だけでなく N スイッチ網羅も導入し，あらかじめ指定される労力の上限を超えない範囲でこれらのメトリクスの値ができるだけ大きいテストケース集合を生成する．なお，EFSM における UDC および N スイッチ網羅の評価に際しては，変数やイベントパラメータの値の違いによって状態や遷移を区別しないものとする．

3. 新たな OPBT フレームワークの提案

3.1 OPBT の概要

本稿において提案する OPBT フレームワークでは，ソフトウェア信頼性を評価したり，ソフトウェア信頼性に深刻な影響を与えるフォールトを重点的に発見したりするだけでなく，N スイッチ網羅によって EFSM の構成要素をできるだけ網羅することも目的とする．したがって，従来よりも幅広くフォールトを発見できる可能性があるし，フォールトを発見できなかったとしてもテスト対象ソフトウェアの品質についてより深い確信を得ることができる．

また，EFSM の導入によってより複雑な振舞いを扱うことができるようになる．

本研究における OPBT の手順は以下のとおりである．

ステップ 1. テスト対象ソフトウェアの要求仕様に基づいて EFSM を作成する．

ステップ 2. EFSM 上の確率分布（および労力分布）を導出し，拡張運用プロファイルを完成させる．

ステップ 3. 開発プロジェクトの状況に応じて，生成するテストケース集合の性質や量に関するパラメータ値を決定する．

ステップ 4. 拡張運用プロファイルと上記パラメータに基づき，UDC および N スイッチ網羅率ができるだけ大きく，かつ実行可能なテストケース集合を生成する．

ステップ 5. 生成されたテストケース集合を実行する．

文献 [4] との手順上の違いは，(1) FSM ではなく EFSM を作成すること，(2) N スイッチ網羅をも考慮したテストケース集合を生成するためのパラメータ設定を行うことである．また，文献 [4] 以前の OPBT との違いは，(3) GA を用いたテストケース生成アルゴリズムを使用すること，(4) 生成するテストケース集合の性質や量をコントロールするために各種パラメータ設定を行うことである．パラメータの詳細は次節で述べる．

3.2 テストケース生成アルゴリズム

ステップ 4 では，拡張運用プロファイルからテストケースを生成するアルゴリズムが必要である．そのアルゴリズムの提案に先立ち，まずは本フレームワークのテストケースが満足しなければならない制約を以下に整理する．

(A) すべてのテストケースが実行可能であること．

(B) テストケース集合の UDC をできるだけ大きくすること．

(C) テストケース集合の N スイッチ網羅率をできるだけ大きくすること．

(D) テストケース集合の実行に要する労力が指定された値を超えないこと．

EFSM では遷移間にデータ依存関係があるため，実行可能なテストケースを生成することが容易ではない．これは，ガードを満足するテストデータ集合を定式化によって求めることができない場合があることに起因している．この問題に対して，メタヒューリスティクスの一種である GA を応用することで解決を図ろうとする研究が近年行われており，その有効性が示されている [1], [2]．加えて，OPBT では多様なテストケース集合をランダムに繰り返し生成できることが重要である．様々な制約を考慮しつつ広大な空間をランダムに探索し，現実的な計算時間で最終的な解候補を導出できる GA は OPBT に適した技法の 1 つである [4]．そこで本研究では，上述の (A)〜(D) の制約を満足するテストケース集合を生成するために GA を応用したアルゴリズムを提案する．このアルゴリズムでは，**図 2** (a) に示すように，染色体（1 つの解候補を GA で扱える形式で表したもの）を 1 つのテストケース集合に，染色体を構成する各遺伝子を 1 つのテストケースに対応させる．そして，世代交代を繰り返し，自然淘汰を経て導出した最も優れた染色体を最終的な解候補として出力する．文献 [4] のアルゴリズムで (B)(D) は考慮されているので，本研究では (A)(C) についても対応できるようにアルゴリズムを拡張した．

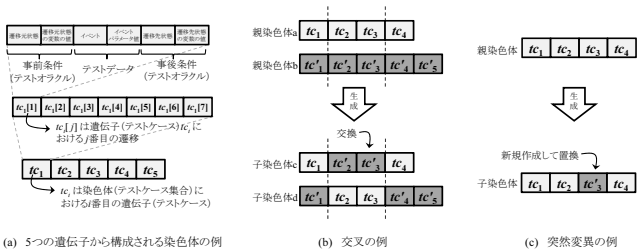


図 2 染色体，交叉，突然変異の概要

Fig. 2 Overview of chromosome, crossover, and mutation.

本アルゴリズムは以下の手順から構成される．

ステップ 4.1. 初期集団（最初の世代を構成する染色体の集合）を生成する．この染色体は，(A)(D) を満たすようにランダムに生成される．すなわち，拡張運用プロファイルの現在の状態において，状態のアクションの実行を終了した後，現在の状態を起点とする遷移に付随するガードをすべて評価することで次に発火可能な遷移を抽出する．そしてその中から遷移確率に比例する確率で 1 つの遷移をランダムに選択し実行する．イベントパラメータが存在する場合は，あらかじめ定義された値域に基づいてその値をランダムに生成する．また，遷移にアクションが付随する場合は遷移先の状態のアクションを実行する前に実行する．これらの操作を拡張運用プロファイルの開始状態から始め，終了状態に到達するまで行うことによって，1 つの遺伝子を生成する．このように本アルゴリズムでは実際に拡張運用プロファイルを動作させることによって (A) を満たすことを保証する．さらに，遺伝子を (D) を満たさなくなる直前まで生成することで，1 つの染色体を生成する．集団サイズ（1 つの世代を構成する染色体の数）としてあらかじめ指定された s 個（$s > 1$）の染色体を生成すれば初期集団が完成する．

ステップ 4.2. 交叉と突然変異によって新たな染色体を生成し，現在の世代に追加する．本アルゴリズムにおける交叉と突然変異の概要を図 2 (b)(c) に示す．突然変異では，ステップ 4.1 の方法によって別途新規作成した遺伝子を用いて置換する．交叉において親として選択される確率は交叉率，突然変異において交換するべき遺伝子として選択される確率は突然変異率といい，それぞれ c（$0.0 \leq c \leq 1.0$），m（$0.0 \leq m \leq 1.0$）とし

てあらかじめ指定される．

ステップ 4.3. 各染色体について適合度を求め，この適合度に基づいて次世代に残す染色体を選択する．適合度は各染色体がどれだけ優れているかを表す値であり，染色体 ts の適合度は以下の適合度関数 $ftn(ts)$ によって求められる．

$$ftn(ts) = f_1(ts) \cdot f_2(ts)$$
$$f_1(ts) = w_u \cdot udc(ts) + w_n \cdot \frac{1}{h+1} \sum_{i=0}^{h} nsc(ts, i)$$
$$f_2(ts) = 1 - pnl(ts)$$

ここで，$udc(ts)$ は ts の UDC を，$nsc(ts, i)$ は ts の N スイッチ網羅率（$N = i$）を意味する．h は，N スイッチ網羅をどこまで加味するかを表す値で，あらかじめ指定される．w_u と w_n は，それぞれ UDC と N スイッチ網羅にどの程度の重みを置くかというテスト戦術に基づいて決定される値で，$w_u + w_n = 1.0$, $0.0 \leq w_u \leq 1.0$, $0.0 \leq w_n \leq 1.0$ である．また，$pnl(ts)$ は，ts が (D) を満たさない場合に適合度から減じる値を求めるもので，これをペナルティと呼ぶ．(D) を満たさない場合とは，開発プロジェクトの状況に応じてあらかじめ指定される労力の上限 l を，ts の実行に要する労力 $effort(ts)$ が超える場合のことであり，$pnl(ts)$ は以下によって定義される．

$$pnl(ts) = \begin{cases} p, & effort(ts) > l \\ 0, & otherwise \end{cases}$$

p（$0.0 < p \leq 1.0$）はペナルティの大きさを決定する値であり，あらかじめ指定される．本アルゴリズムではエリート保存法とルーレット選択法を採用する．すなわち，適合度において上位 e 個の染色体を必ず選択し，残りの中から $(s - e)$ 個の染色体を適合度に比例する確率でランダムに選択する．

ステップ 4.4. 世代交代回数があらかじめ指定された回数 g に達するか，最良の染色体が変化しない期間があらかじめ指定された世代交代回数 t に達する場合，その時点における最良の染色体を最終的な解候補として出力し終了する．そうでなければステップ 4.2 に戻る．

ステップ 3 で決定するパラメータについて整理すると，集団サイズ s，交叉率 c，突然変異率 m，N スイッチの上限 h，UDC と N スイッチ網羅の重み w_u，w_n，ペナルティ p，労力の上限 l，保存エリート数 e，世代交代回数 g または t となる．これらに設定する値によって本アルゴリズムは様々なテストケース集合を生成することが可能であり，テスト技術者は開発プロジェクトの状況にふさわしいテストケース集合を得ることができる．

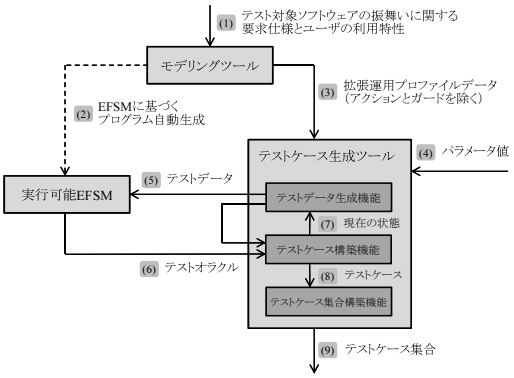


図 3 本フレームワークのテスト環境
Fig. 3 Test environment in the framework.

表 1 各メトリクスの改善の程度
Table 1 Improvement of each metric.

メトリクス	初期集団の平均	最終的な解候補	改善の幅
UDC	0.301	0.565	0.264
0 スイッチ	0.855	1.000	0.145
1 スイッチ	0.440	0.636	0.196
2 スイッチ	0.207	0.318	0.111
3 スイッチ	0.073	0.101	0.028
4 スイッチ	0.024	0.029	0.005

4. 適用例

本フレームワークの有効性を確認するために，**図 3** に示すテスト環境を構築し適用実験を行った．本適用実験におけるテスト対象ソフトウェアは，ある商用のテスト支援ツール中の機能の一部であり，そのツール全体の規模はおよそ 1000KLOC である．ソフトウェア開発会社でのテスト工程において 3 件のフォールトが発見されたことが分かっている．本適用実験の内容は次のとおりである．まず，ソフトウェア開発会社の技術者がテスト対象ソフトウェアの拡張運用プロファイルをおよそ 13 人日で作成した．そのうちのおよそ 10 人日はテスト環境の初期設定に要した労力であり，次回以降の適用時には不要となる．拡張運用プロファイルの規模は，状態数 10，遷移数 19 である．そして本研究において開発したテストケース生成ツールに対して $s = 5$，$c = 0.2$，$m = 0.1$，$h = 4$，$w_u = 0.5$，$w_n = 0.5$，$p = 0.5$，$e = 2$，$g = 1000$，l に 1 人日程度を想定した値を設定し，テストケース集合の生成を 10 回試行した．1 回の試行に要する時間はおよそ 15 分であった．最後に，得られた 10 個のテストケース集合のそれぞれについて，3 件のフォールトを発見できる可能性があるか否かを確認した．ここでは，あらかじめ分かっているフォールトの顕在化条件を EFSM 上で満たした場合に「テスト対象ソフトウェア上で当該フォールトを発見できる可能性がある」と判定する方法で行った．

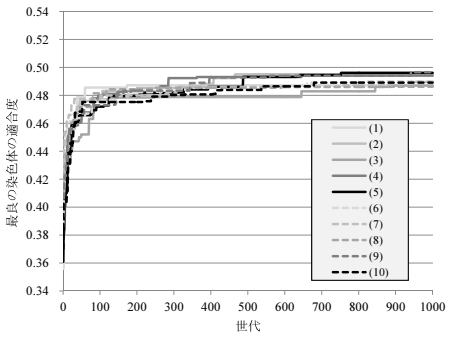


図 4 各試行における適合度の成長
Fig. 4 Growth of fitness in each experiment.

このテスト対象ソフトウェアにおいては，状態を特徴付ける変数の取り得る値の組合せが数百に及ぶと考えられるため，FSM ではなく EFSM を用いることで OPBT を適用することができたといえる．テストケース集合生成の 10 回の試行の結果については，まず生成過程において適合度が成長する様子を**図 4** の (1)〜(10) として示す．いずれも似た曲線を描いており，ばらつきも少なく安定した結果が得られているといえる．さらに，初期集団（第 0 世代）および最終的な解候補のテストケース集合における各メトリクスの平均値を**表 1** に示す．ここで初期集団を構成するテストケース集合は，UDC や N スイッチ網羅の改善を行わない旧来の方法で生成したものとみなすことができる．すべてのメトリクスにおいて改善がみられていることから，本フレームワークによってテストケース集合の質を高めることができたといえる．l による制限があるにも関わらず，UDC と N スイッチ網羅を同時に改善できる点は特筆すべきである．各テストケース集合の内容を確認した結果，10 個すべてのテストケース集合について先述の 3 件のフォールトを発見できる可能性があることが分かった．本フレームワークは従来のテストで発見されるフォールトを発見可能であり，従来のテストの一部を体系化するものといえる．

5. おわりに

本稿では，UDC と N スイッチ網羅率ができるだけ大きいテストケース集合を拡張運用プロファイルから生成するためのフレームワークを提案した．EFSM に基づいて作成される拡張運用プロファイルは従来の運用プロファイルよりも表現力が高いため OPBT の適用範囲を広げることができるが，遷移間のデータ依存性に起因する実行可能性の問題に対処する必要がある．さらに，現実のテスト工程を考慮すると，テストケース集合は限られた労力の範囲で実施できるものでなければならない．これらを解決するために本稿では GA を応用したテストケース生成アルゴリズムを示した．また，これを実現するテスト環境を構築し商用ソフトウェアに適用した結果，EFSM の使用によって複雑な振舞いをもつソフトウェアに対して OPBT が効果的に適用できること，UDC と N スイッチ網羅率を同時に改善可能でありテストケース集合の質を高めることができることなどが確認できた．したがって，本フレームワークは OPBT の有効性をより高めるものであると結論できる．

今後の研究では本フレームワークを拡張することでテスト戦術を体系化し，その可能性を調査する予定である．

謝辞 本研究は JSPS 科研費 23700038 の助成を受けた．また，ガイオ・テクノロジー株式会社 市川忠彦，大西建児，大城戸薫の諸氏には本研究に関して有益なご意見をいただいた．ここに深く感謝の意を表する．

参考文献

[1] Kalaji, A., Hierons, R. M. and Swift, S.: Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM), *Proc. International Conference on Software Testing Verification and Validation*, pp. 230–239 (2009).

[2] Doungsa-ard, C., Dahal, K., Hossain, A. and Suwannasart, T.: Test Data Generation from UML State Machine Diagrams using GAs, *Proc. International Conference on Software Engineering Advances*, p. 47 (2007).

[3] Whittaker, J. A. and Thomason, M. G.: A Markov Chain Model for Statistical Software Testing, *IEEE Transactions on Software Engineering*, Vol. 20, No. 10, pp. 812–824 (1994).

[4] 高木智彦，橋本慎一朗，八重樫理人，古川善吾：拡張運用プロファイルに基づく最適化されたテストスイートの生成手法，情報処理学会論文誌， Vol. 53, No. 2, pp. 557–565 (2012).

[5] Takagi, T., Nishimachi, K., Muragishi, M., Mitsuhashi, T. and Furukawa, Z.: Usage Distribution Coverage: What Percentage of Expected Use Has Been Executed in Software Testing?, *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Studies in Computational Intelligence, Vol. 209, Springer, pp. 57–67 (2009).

[6] Hartmann, H., Bokkerink, J. and Ronteltap, V.: How to reduce your test process with 30% – The application of Operational Profiles at Philips Medical Systems, *Supplementary Proc. 17th International Symposium on Software Reliability Engineering*, CD-ROM (2006).

[7] Chow, T. S.: Testing Software Design Modeled by Finite-State Machines, *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 3, pp. 178–187 (1978).

[8] Sayre, K. and Poore, J. H.: Stopping criteria for statistical testing, *Information and Software Technology*, Vol. 42, No. 12, pp. 851–857 (2000).