修 士 論 文

題　目　　組み合わせテスト最適化のためのベイズ推定
を用いた重み抽出に関する研究

主任指導教員　　水野　修　准教授

京都工芸繊維大学大学院 工芸科学研究科

情報工学専攻

学生番号　　14622011

氏　　名　　河端　駿也

平成28年2月10日提出

様式 2 号

学位論文の要旨（和文）

平成 28 年 2 月 10 日

京都工芸繊維大学大学院
工芸科学研究科長　殿

工芸科学研究科　　情報工学専攻
平成 26 年入学
学生番号　　14622011
氏　　名　　河端　駿也　㊞

（主任指導教員　　水野　修　㊞　）

本学学位規則第 4 条に基づき、下記のとおり学位論文内容の要旨を提出いたします。

1.　論文題目

組み合わせテスト最適化のためのベイズ推定を用いた重み抽出に関する研究

2.　論文内容の要旨（400 字程度）

理想的なテストの条件は，少ないテストケース数で多くのフォールトを検出できることである．テストケース数が少ないテストスイートを生成する手法の一つに，組み合わせテストというパラメータが取る値の組み合わせに注目したものがある．組み合わせテストで生成されたテストスイートは高い不具合検出性能を持つとされているが，さらにフォールトを検出する可能性の高いテストケースをテストスイートの最初に位置するよう並び替えることで，資源が少ない場合でもテスト実施が可能なように最適化することができる．本研究では，ベイズ推定を用いた組み合わせテスト最適化手法を提案する．手法は，1 テストケースを構成している各パラメータにフォールト検出した確率をマッピング，その確率から 1 テストケースの重みを計算，重みによりテストケースを降順に並び替えることで最適化を実現する．また，最適化を 3 つのプロジェクトから選択した 12 バージョンに適用し，提案手法と他の最適化手法との比較を行った．結果，提案手法は他メソッドと拮抗し，プロジェクトによっては最良なメソッドになるため，1 つの有効なメソッドといえる．

Priority Extraction Using Bayesian Inference for Prioritized Combinatorial Testing

2016 14622011 *Shunya KAWABATA*

Abstract

An ideal testing detects the large number of faults with the small number of test cases. One of techniques to produce the test suite with the small number of test cases is the combinatorial testing which focuses on a combination of parameter values. To prioritize combinatorial testing, test cases are sorted by priority of fault detection in descending order. Prioritized combinatorial testing can detect the large number of faults with a small testing resources. In this study, we propose a method to prioritize combinatorial testing using Bayesian inference. In this approach, we first map the probability of the faults detected to the parameters of each test cases. Next, we calculate the test case's weight from the mapped probabilities. We then sort the test cases by the calculated test case's weight in descending order. We compare our approach with the existing prioritization techniques applying twelve versions of three projects. From the results of experiments, we confirm that proposed method is comparable to others, and also confirm that it is the best method in some projects.

目　次

1. 緒言

ソフトウェアの不具合がリリース後に判明した場合，市場に普及したソフトウェアを修正するには多くの労力が必要であり，また普及したすべてのソフトウェアを修正できるとは限らない．これを防ぐためにもリリース前に十分なテストを実施し不具合を検出しておく必要がある．しかし，考えられるすべてのテストケースを実行するのは多くの時間と労力が必要となるため現実的でない．少ないテストケース数で多くのフォールトを検出できることが理想である．

テストケース数が少ないテストスイートを生成する手法の一つに組み合わせテストがある．組み合わせテストはパラメータが取る値の組み合わせに注目した手法であり，2 から 4 つの組み合わせパターンであればテストケース数を十分に少なくしフォールト検出が可能といわれている．組み合わせテストにより生成されたテストスイートは，フォールトを検出する可能性の高いテストケースをテストスイートの最初に位置するよう並び替えることで最適化ができる．優れた最適化を行うには，どのように並び替えるかが重要となる．

組み合わせテストの最適化手法はいくつか存在する．Qu ら [1] はコード網羅率を用いた手法を提案，実験対象の 1 つ前バージョンの重みを用いた回帰テストで評価している．Choi ら [2] は pricot という組み合わせテスト生成ツールに，順序重視，頻度重視，テストケース数重視の 3 つの戦略を考え，それらを組み合わせることができる最適化手法を提案している．

本研究では，仕様書とバグレポートからベイズ推定を用いて求めた値により並び替える最適化手法を提案する．方法としては，パラメータがある値を取ったときにフォールト検出した確率を各パラメータの各値にマッピング，次にその重みから 1 テストケースの重みを計算，計算した重みによりテストケースを降順に並び替えることで最適化を実現する．提案手法と他の手法とを比較，評価するために，Software artifact Infrastructure Repository（SIR）[3] から 3 つのオープン・ソース・プロジェクト，計 12 バージョンを用いて実験を行う．実験は，組み合わせテスト生成ツールである PICT により生成したテストスイートに対し提案手法を適用したものと，pricot により再生成，最適化を行ったものとの比較を行う．

本論文の以降の構成を示す．2 章では分析に必要な事前知識の説明を，3 章では提

案手法の詳細を，4 章では実験対象および提案手法との比較手法の詳細を，そして 5 章では事例研究を行い，6 章にて結言とする．

2. 分析準備

2.1 ベイズ推定

式 2.1 に事象 A が起こった条件にて事象 B が起こる確率を表す.

$$P(B|A) = \frac{P(A \cap B)}{P(A)} \tag{2.1}$$

ここで，P は確率，A, B は事象であり，$P(B|A)$ は条件付き確率，$P(A \cap B)$ は事象 A と事象 B が起こる共通事象の確率である.

2.2 組み合わせテスト

組み合わせテストは，パラメータが取る値の組み合わせに注目し，ある組み合わせ数のパラメータ間にて取る値のパターンすべてを網羅するテストのことであり，組み合わせ数を t としたとき t-way テストや t-wise テストと呼ぶ．以後，パラメータが取る値のことをパラメータ値と表記する.

例として表 2.1 のパラメータ仕様が与えられたときの 2-wise テストを生成する．まず総テストケースなら，3 パターンのパラメータ値を取るパラメータが 1 つ，2 パターンのパラメータ値を取るパラメータが 2 つなので，テストケース数は $3 \times 2 \times 2 = 12$ となる．対して 2-wise の組み合わせテストなら，表 2.2 のテストスイートになり，テストケース数は 6 となる．考えられるすべての 2 つの組み合わせ (X, Y), (Y, Z), (Z, X) のパラメータ値を確認すると，確かにすべてのパターンを網羅している.

組み合わせテスト生成ツールには，Microsoft 社の Czerwonka ら [4] が開発した PICT [2]，産業技術総合研究所の Choi ら [5] が開発した pricot，Bryce ら [6] が開発した DDA などがある.

2.3 組み合わせテストの最適化

組み合わせテスト生成ツールにより生成されたテストスイートは，テストケース数が少なくなるため，テスト実行に必要な資源を減らすことができる．しかし，このテストケースの並びを，更にフォールト検出する可能性が高いテストケースをテ

表 2.1　パラメータの仕様

Parameter	Parameter-value
$X =$	$1, 2, 3$
$Y =$	$1, 2$
$Z =$	$1, 2$

表 2.2　表 2.1 の 2-wise 組み合わせテスト

# of test cases	X	Y	Z
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1
5	3	1	1
6	3	2	2

ストスイートの初期に位置するような並びにすれば，実行するテストケース数が少ない場合でも，テストスイート初期の段階で多くのフォールト検出が可能になる．これを組み合わせテストの最適化と呼ぶ．

組み合わせテスト生成ツールには，pricot のように生成だけでなく最適化を行うものもある．

2.4 評価手法

2.4.1 Average Percentage of Faults Detected

良いテストスイートかを示す指標として，Elbaum ら [7] は Average Percentage of Faults Detected（以後 *APFD* と呼ぶ）を提案した．*APFD* の計算方法を式 (2.2) に示す．

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{m \times n} + \frac{1}{2n} \tag{2.2}$$

ここで，m は総テストケースにて検出できるフォールト数，n はテストケース数，TF_i はフォールト i を初めて検出したテストケース番号である．この式は x 軸をテストケース数，y 軸を検出したフォールトの累積数とした線グラフにて，縦が $y = 0$ からグラフの y 値まで，横が $x = 0$ からグラフの x 値までの面積を求めることを意味する．

例として，表 2.3 のテストスイートにおける *APFD* を計算すると式 (2.3) となり，値は 0.6 となる．

$$\begin{aligned} APFD &= 1 - \frac{1+0+4}{2 \cdot 5} + \frac{1}{2 \cdot 5} \\ &= 0.6 \end{aligned} \tag{2.3}$$

2.4.2 Normalized Average Percentage of Faults Detected

前章にて示した *APFD* の問題として，値を比較したい場合，異なるテストケース数のテストスイート同士は比較できない点がある．例として，総テストケースの表 2.3 とその組み合わせテストである表 2.4 を比較したい場合，テストケース数が 5 に対し 3 と異なるため，単純に *APFD* を計算し比較することができない．この点

表 2.3　*APFD* の計算例

# of test cases	F_1	F_2	F_3
1	✓		
2			
3			
4	✓		✓
5	✓		

表 2.4　*NAPFD* の計算例

# of test cases	F_1	F_2	F_3
1	✓		
2			
3	✓		

に対し，Qu ら [1] は新メソッドである Normalized Average Percentage of Faults Detected（以後 *NAPFD* と呼ぶ）を提案した．*NAPFD* の計算方法を式 (2.4) に示す．

$$NAPFD = p - \frac{TF_1 + TF_2 + \ldots + TF_m}{m \times n} + \frac{p}{2n} \tag{2.4}$$

ここで，p は与えられた組み合わせテストが検出できるフォールト数を総テストケースが検出できるフォールト数で除算したもの，その他の変数は *APFD* と同じである．変数 p の導入により *APFD* を正規化している．

上記の例の *NAPFD* を求める．表 2.3 は総テストケースであり $p = 1$ となるため *APFD* と変わらず値は 0.6，表 2.3 の組み合わせテストである表 2.4 の *NAPFD* は式 (2.5) にて計算でき，値は 0.25 となる．

$$\begin{aligned} NAPFD &= (1/2) - \frac{1+0+0}{2 \cdot 3} + \frac{(1/2)}{2 \cdot 3} \\ &= 0.25 \end{aligned} \tag{2.5}$$

実験結果の評価は *APFD* ではなく *NAPFD* で評価していく．

2.5 関連研究

組み合わせテストの有効性について述べる．Kuhn ら [8] はフォールト出現のきっかけとなるパラメータの組み合わせ数に注目し，アメリカ食品医薬品局の医療器具，Mozilla Web ブラウザ，Apache サーバ，ゴダード宇宙飛行センターの分散システムの 4 つのプロジェクトのバグレポートから，フォールト数とパラメータの組み合わせ数の関係を調査した．結果，プロジェクトの種類に関わらず，2 から 4 の組み合わせテストで十分にフォールトを検出できるとしている．

組み合わせテストの最適化とその有効性について述べる．Qu ら [1] は，flex と make にて，回帰テストでも最適化により効率の良いテストスイートを生成できるか調べるために，元の順序の組み合わせテスト，コード網羅率により最適化した組み合わせテスト，対象プロジェクトの前バージョンの組み合わせテストから計算したコード網羅率を用いて最適化と再生成した組み合わせテスト，そして TSL を用いて最適化と再生成した組み合わせテストにて比較を行った．結果，flex と make の一部にて最適化また再生成により確かに効率の良いテストスイートを生成できることを確認

した．また，最適化および再生成によるテストスイートの良さと組み合わせ数 t の影響を調べるために，$t = 2$ と $t = 3$ で比較した研究 [9] では，コード網羅率を用いた最適化は高い t であるとフォールトを早くに検出するが，再生成の場合や TSL を用いた最適化と再生成の場合は高い t の方が常に良くなるとは限らなかった．

3. 提案手法

3.1 研究設問

研究の目的を明確とするため，次の 2 つの研究設問を設ける．

RQ1 提案手法はテストを最適化できるか．

RQ2 提案手法は他の最適化手法と比べ優れているか．

RQ1 は，実際にフォールト検出の可能性が高いテストケースをテストスイートの初期に位置させることができるか調査する．RQ2 は，他の手法を同じ実験対象に適用し結果を比較することで，手法の優れている点を考察していく．

3.2 ベイズ推定を用いた最適化

本研究にて提案する最適化手法について，全体の流れを次に示す．

1. 仕様書とバグレポートからベイズ推定によりパラメータ値の重み w_v を定義．
2. 重み w_v を用いて，1 つのテストケースの重み w_t を定義．
3. 重み w_t によりテストケースを降順に並び替える．

以降で各ステップの詳細を説明する．

3.2.1 パラメータ値の重み定義

テストケースの重み定義に必要となるパラメータ値の重みを定義していく．実験対象には次のデータが含まれているとする．

- 各テストケースの入力引数データ
  テストにて使用する，パラメータ値の集合からなる入力データ．
- 各テストケースに関するバグレポート
  テストケースを実行した結果，各フォールトを検出した，または検出しなかったかを記したデータ．

- Test Specification Language (TSL) [10] で記述された実験対象に関する仕様書
  各パラメータが取ることのできるパラメータ値のリストや，取る値の組み合わせにより生じる制約条件を記したデータ．

これら 3 つのデータから条件確率を求め，その値を重みとする．用いる確率は次の 2 通りとする．

$$P(B \mid p_i = x) \tag{3.1}$$

$$P(B \mid (p_i = x) \wedge (p_j = y)) \tag{3.2}$$

ここで，P は確率，B はフォールト検出した事象，p_i, p_j はパラメータ，x, y はパラメータ値とする．式 (3.1) は 1 つのパラメータがあるパラメータ値を取る事象にてフォールト検出した確率を，式 (3.2) は 2 つのパラメータがそれぞれ特定のパラメータ値を取る共通事象にてフォールト検出した確率を表す．これらの確率をパラメータ値の重みと定義する．

3.2.2 テストケースの重み定義

定義したパラメータ値の重みにより，1 つのテストケースの重みを定義していく．定義方法に関して，本研究では次の 4 つのメソッドを提案する．

- 1 パラメータ-未指定値なし
  式 (3.1) の重みを各パラメータ値へマッピング，その平均をテストケースの重みとする．このとき，指定のないパラメータ値へはマッピングしない．
- 1 パラメータ-未指定値あり
  メソッド 1a を変更し，指定のないパラメータ値も 1 つの値として重みを計算しマッピングする．
- 2 パラメータ-未指定値なし
  式 (3.2) の重みを各パラメータ値の組み合わせへマッピング，その平均をテストケースの重みとする．このとき，どちらかのパラメータのパラメータ値が未指定ならば重みをマッピングしない．

- 2 パラメータ-未指定値あり

  メソッド 2a を変更したもの．指定のないパラメータ値を含む組み合わせでも重みを計算しマッピングする．

以後，それぞれのメソッドは上から m-1a, m-1b, m-2a, m-2b と表記する．テストケースの重みによりテストケースを降順に並び替えることで組み合わせテストの最適化が完了する．

4. 実験準備

4.1 対象プロジェクト

本研究では SIR [3] リポジトリを用いて実験を行った．SIR は研究者によってオープン・ソース・ソフトウェア（OSS）から収集され，フォールトを挿入されたプロジェクトデータである．

表 4.1 に対象プロジェクトの詳細を示す．本研究では Unix ユーティリティとして有名な flex, grep, make の 3 プロジェクトを実験対象とする．いずれも C 言語にて記述されている．各プロジェクトには複数のバージョンが存在するため，その中から実験に必要なデータが全て含まれており，かつフォールトを検出しているものを選択したところ，flex は 5 個，grep は 4 個，make は 3 個の計 12 バージョンとなった．コード行数 (Line of Code, LoC) は空行を除いた行数を grep コマンドにて集計した．

SIR の中には 3.2.1 節で示した，次のテストに関するデータが含まれている．

- Frame

  各テストケースにおいて各パラメータが取るパラメータ値（文字列）を格納したファイル．
- Universe

  各テストケースを実行したとき，埋め込まれた各フォールトを検出した，または検出しなかったかを格納したファイル．
- TSL

  各パラメータの仕様を TSL 形式にて格納したファイル．

これらのファイルより重みを求めテスト最適化を行う．最適化の流れを以下に示す．

1. BUG ファイルの生成

   テスト実行時のフォールトに関するファイル Universe から各フォールトを検出した，または検出しなかったを示す情報を抽出しファイル BUG へ出力する．
2. TS ファイルの生成

   各テストケースの各パラメータ値（文字列）を扱いやすくするために名義尺度の数値へマッピングしファイル TS へ出力する．このとき，パラメータ値の情報を仕様書ファイル TSL から確認する．

表 4.1　各プロジェクトの詳細

project	ver.	LoC	# of faults		# of all tests
			seeded	detected	
flex	v1	12,160	19	16	525
	v2	12,737	20	13	525
	v3	12,781	17	9	525
	v4	14,168	16	11	525
	v5	12,893	9	5	525
grep	v1	12,507	18	4	470
	v2	13,179	8	2	470
	v3	13,291	18	8	470
	v4	13,359	12	2	470
make	v1	18,460	19	4	793
	v2	19,149	6	1	793
	v3	20,340	5	1	793

3. 重みの計算

   ファイル BUG, TS からメソッドに必要なパラメータ値の重みを求める．

4. 重みによりテストスイートを降順に並び替える

   テストスイートを各メソッドにて並び替えることで最適化が完了する．

図 4.1 に組み合わせテスト最適化の流れを示す．

4.2 比較手法

提案手法により求めたテストスイートを評価するため，比較に用いる手法とそのテストスイートを説明する．

4.2.1 PICT に対する最適化

PICT が生成した組み合わせテストに提案手法と異なる手法で最適化した場合の比較を行う．最適化には次の 2 つの方法を用いる．1 つ目は，テストケースの順序が総テストケースの順序と同じになるよう並び替える．以後このテストスイートを org と表記する．2 つ目は，テストケースをシャッフル，つまりランダムに並び替える．シャッフルに用いる擬似乱数はメルセンヌ・ツイスタにより生成，シャッフルにより組み合わせテストを 100 通り生成する．以後，このテストスイートを rnd と表記する．

4.2.2 pricot による再生成と最適化

次に，PICT と異なる組み合わせテスト生成ツールを用いて，生成と最適化をした場合との比較を行う．生成ツールには pricot を用いる．pricot は産業技術総合研究所にて開発された組み合わせテスト生成ツールであり，特徴として戦略を調整することができる．戦略には，重み値が大きいパラメータ値をテストの初期に置くよう順序で最適化する戦略 co，重み値が大きいテストケースをテストに頻繁に出現させるよう頻度で最適化する戦略 cf，組み合わせテストをなるべく小さくなるようテストケース数を最適化する戦略 cs の 3 つがある．またこれらの戦略は優先度を付けて組み合わせることができる．例えば co.cf なら co > cf の優先度を持つ戦略となる．本研究では総テストケースと m-1b の重みを入力とし，2-wise の組み合わせテストを

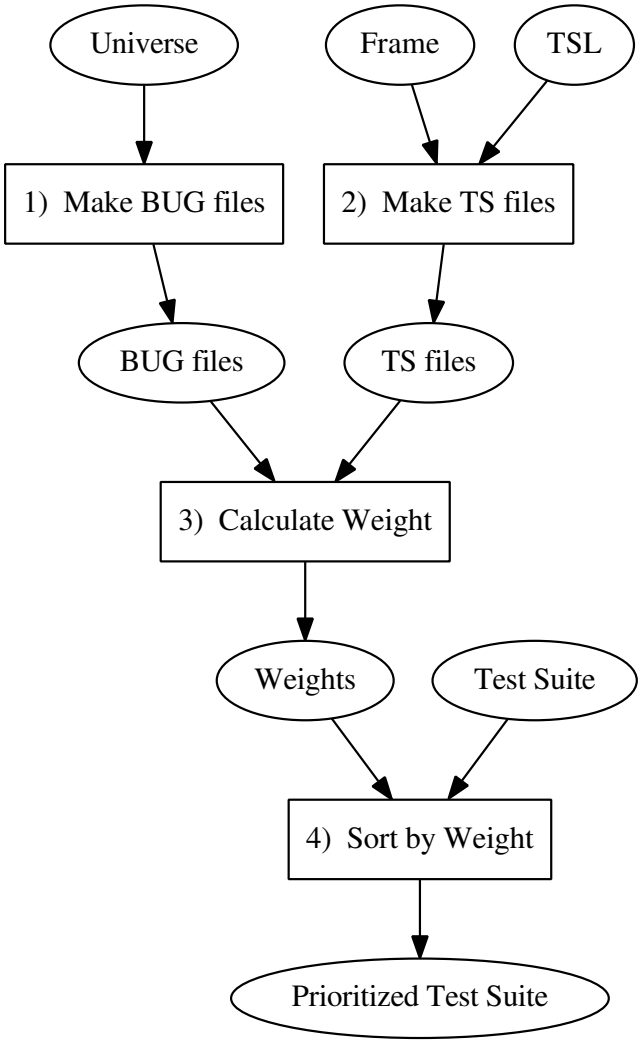

図 4.1　組み合わせテスト最適化の流れ

出力する．戦略には，1) cs, 2) co, 3) cf, 4) cs.co, 5) co.cs, 6) cs.cf, 7) co.cf, 8) cs.co.cf, 9) co.cs.cf を実行した内，NAPFD 値の全体平均が最も高かった co.cf を用いる．

5. 事例研究

5.1 実験結果

表 5.1 に各ツールで生成，最適化した 2-wise 組み合わせテストのテストケース数を示す．テストケース数は，PICT が生成した場合と pricot が生成した場合で多少の差はあるが，どちらも近い値となる．

表 5.2 に各組み合わせテストにより検出したフォールト数を示す．PICT と pricot で，どちらも検出するフォールト数は等しい．また表 4.1 より，検出したフォールト数は組み合わせテストの場合と総テストケースの場合とで等しい．

図 5.1 に，検出したフォールト累積数の推移を，各プロジェクト毎に示す．グラフは，x 軸が実行したテストケース数，y 軸がその時点で検出したフォールト数であり，org と 4 つの提案手法の結果をプロットしている．各プロジェクトにてフォールト検出の立ち上がりが早いメソッドを見ていくと，flex の v1, v2, v4 では m-2a, m-2b，v3 では m-1b，v5 ではすべての提案手法となっている．次に grep では，v1 では m-1b, m-2a, m-2b が早いがテストケース数 20 の時点では org と m-1a に検出数で抜かれている．v2 では org, m-1a, m-1b，v3 では m-2a，v4 では m-1a となっている．そして make では，v1 では m-1a, m-1b, m-2b，v2 では org, m-1a, m-1b, m-2b，v3 では m-1a, m-1b, m-2b となっている．

表 5.3 に各メソッドの *NAPFD* 値とその平均を示す．また図 5.2 に *NAPFD* 値のボックスプロットをプロジェクト毎に，図 5.3 に全体で示す．

5.2 考察

RQ に沿って考察していく．

5.2.1 RQ1: 提案手法はテストを最適化できるか

図 5.1 を見ると，それぞれのプロジェクトにおいて提案手法にて最適化した組み合わせテストは元の順序のテストスイートと同じか，または早い段階で，第一フォールトを検出している．またすべてのフォールト検出も同様の結果となっている．以

表 5.1　2-wise 組み合わせテストのテストケース数

project	ver.	PICT	co.cf
flex	v1	51	51
	v2	51	51
	v3	51	53
	v4	51	52
	v5	51	51
grep	v1	77	75
	v2	77	75
	v3	77	79
	v4	77	76
make	v1	34	32
	v2	34	33
	v3	34	33

表 5.2　2-wise 組み合わせテストの検出したフォールト数

project	ver.	PICT	co.cf
flex	v1	16	16
	v2	13	13
	v3	9	9
	v4	11	11
	v5	5	5
grep	v1	4	4
	v2	2	2
	v3	8	8
	v4	2	2
make	v1	4	4
	v2	1	1
	v3	1	1

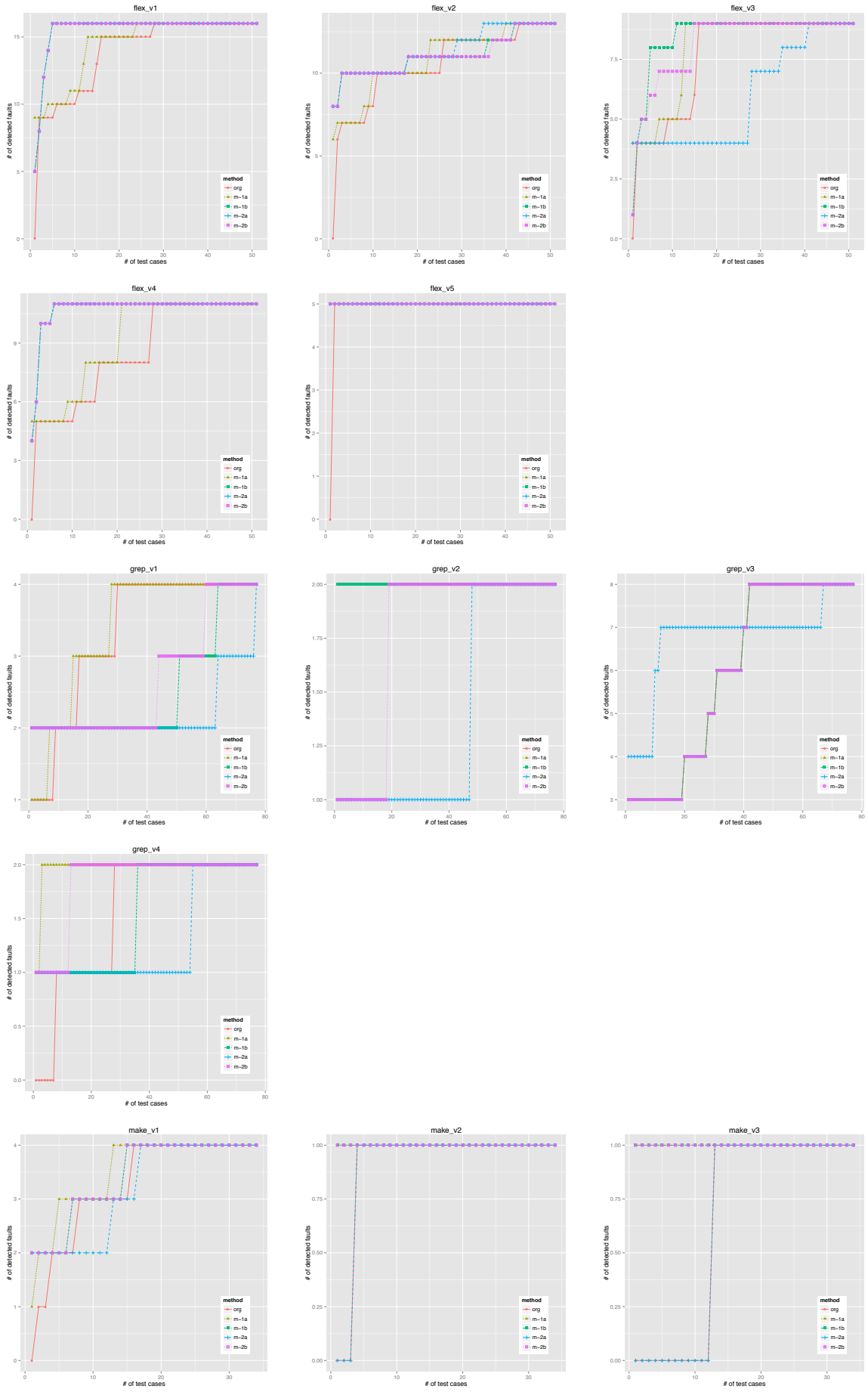

図 5.1　フォールト検出数の推移

表 5.3　各メソッドの *NAPFD* 値

project	ver.	org	rnd	m-1a	m-1b	m-2a	m-2b	co.cf
flex	v1	0.8566	0.9455	0.8922	0.9596	0.9596	0.9596	0.9767
	v2	0.7971	0.8659	0.8258	0.8439	0.8650	0.8424	0.8544
	v3	0.8355	0.7498	0.8747	0.9314	0.6525	0.8965	0.8606
	v4	0.7656	0.9079	0.8262	0.9635	0.9635	0.9635	0.9205
	v5	0.9706	0.9886	0.9902	0.9902	0.9902	0.9902	0.9902
	avg.	0.8451	0.8916	0.8818	0.9377	0.8862	0.9304	0.9205
grep	v1	0.8214	0.7985	0.8409	0.6266	0.5422	0.6623	0.6933
	v2	0.9935	0.7983	0.9935	0.9935	0.6883	0.8766	0.8733
	v3	0.7403	0.8742	0.7403	0.7403	0.8393	0.7403	0.8608
	v4	0.7727	0.7796	0.9805	0.7662	0.6429	0.9156	0.9276
	avg.	0.8320	0.8127	0.8888	0.7817	0.6782	0.7987	0.8387
make	v1	0.7941	0.8449	0.8603	0.8382	0.7794	0.8382	0.8047
	v2	0.8971	0.6353	0.9853	0.9853	0.8971	0.9853	0.9848
	v3	0.6324	0.4474	0.9853	0.9853	0.6324	0.9853	0.9848
	avg.	0.7745	0.6425	0.9436	0.9363	0.7696	0.9363	0.9248
all	avg.	0.8231	0.8030	0.8996	0.8853	0.7877	0.8880	0.8943

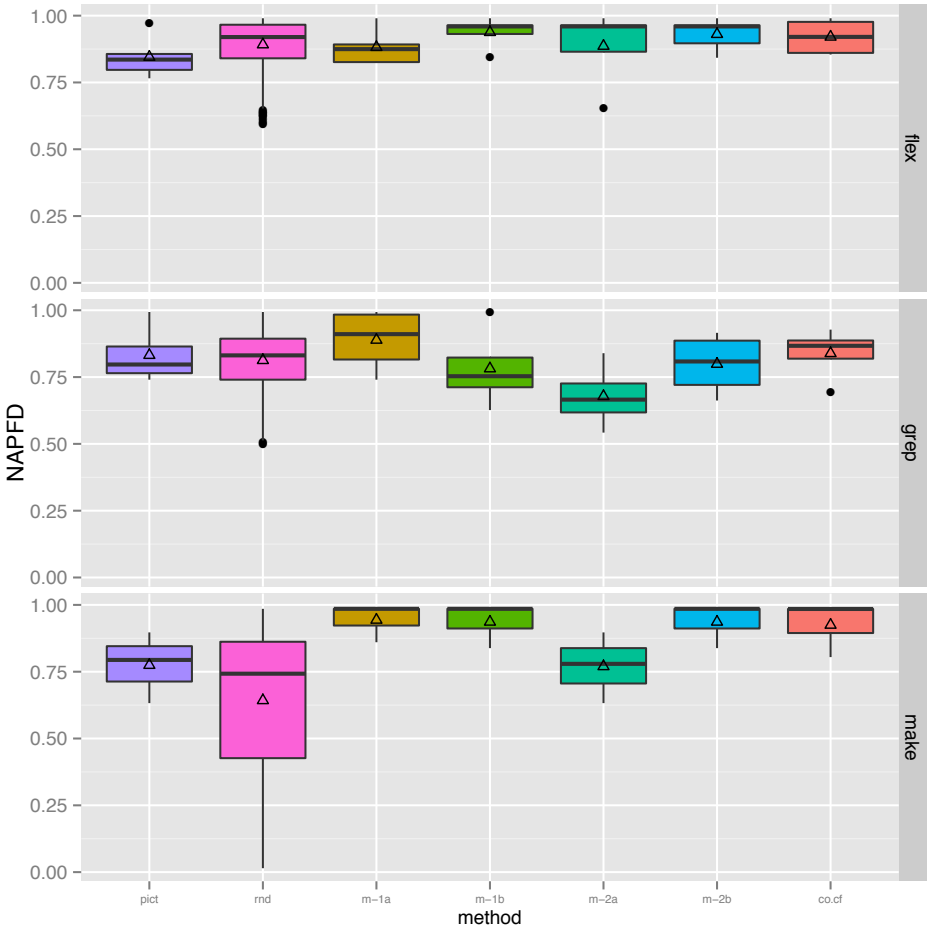

図 5.2　各プロジェクトにおけるメソッドの *NAPFD* 値

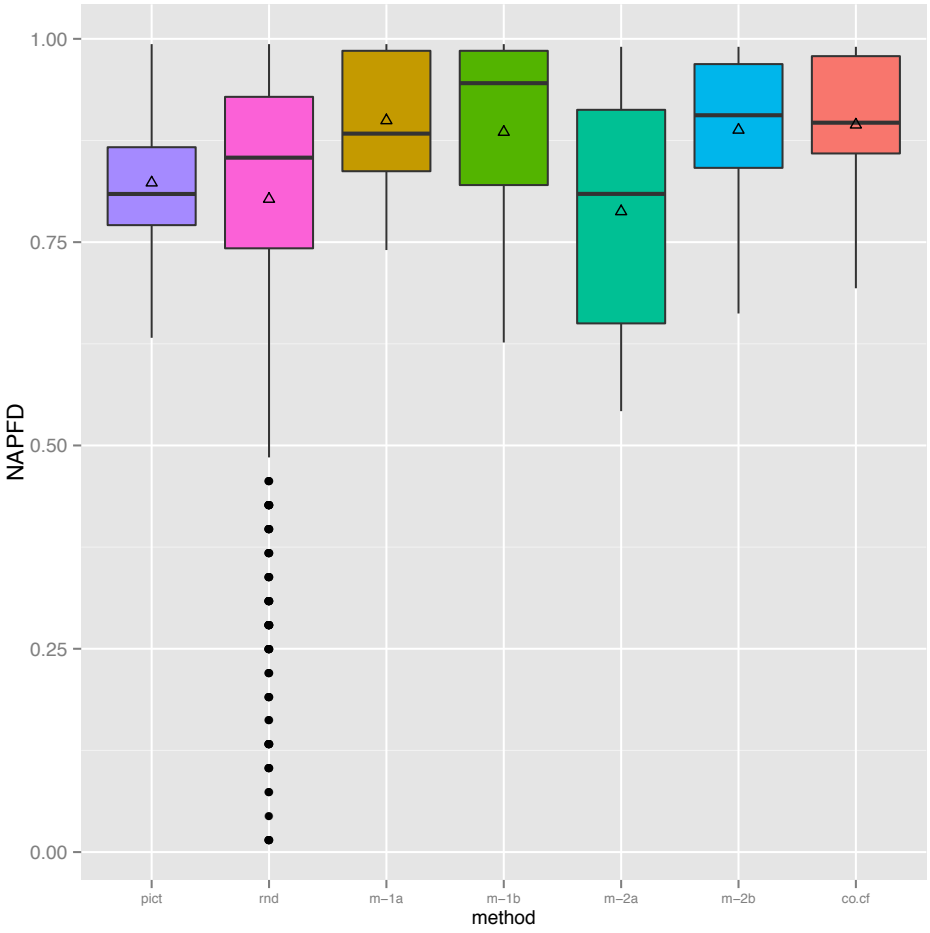

図 5.3　全プロジェクトにおけるメソッドの *NAPFD* 値

上の結果より，RQ1 への回答として，提案手法は確かに PICT が生成した組み合わせテストを最適化できるといえる．

5.2.2 RQ2: 提案手法は他の最適化手法と比べ優れているか

表 5.3 を見ていく．まず提案手法の 4 つのみで比較していく．プロジェクト毎の平均に注目すると，flex, grep, make の順番で最も優れたメソッドは m-1b, m-1a, m-1a となり，対して最も劣ったメソッドは m-1a, m-2a, m-2a となる．このとき両者の差は 0.0461, 0.2106, 0.1740 であり，grep, make の場合は大きな差となっている．そして 3 プロジェクトをまとめた平均では，最も優れたのは m-1a，最も劣っているのは m-2a であり，その差は 0.1119 となりメソッドで結果に差が生じている．

次に pricot の再生成と最適化である co.cf も含めて最適化手法を比較していく．プロジェクト毎の平均に注目すると，flex, grep, make の順番で最も優れたメソッドは m-1b, m-1a, m-1a となり，最も劣っているのは m-1a, m-2a, m-2a，両者の差は 0.0559, 0.2106, 0.1740 となる．そして 3 プロジェクトをまとめた平均では，最も優れたのは m-1a，最も劣っていたのは m-2a であり，その差は 0.1119 となる．

各メソッドにて *NAPFD* 値が最良となった回数を見ていく．表 5.4 にて，それぞれのメソッドが最良となった回数を集計した結果，m-1a が 7 回で最多となり，次が m-1b の 6 回となる．

図 5.2 の各プロジェクトのボックスプロットを比較していく．flex にて m-1b は平均値が大きく，第一から第三四分位数までの間隔が狭いため，優れたメソッドといえる．また m-2a, m-2b も中央値が 1.0 に近いので優れているといえる．grep では，m-1a は第一から第三四分位数の間隔が他メソッドと比べると大きめだが，中央値は大きく第三四分位数は 1.0 付近であるため，優れたメソッドといえる．make では org, rnd, m-2a を除いたメソッドは中央値，平均値とも 1.0 に近いため優れたメソッドといえる．次に図 5.3 のボックスプロットにて比較する．4 つの提案手法の内では m-1a, m-1b, m-2b が優れたメソッドであり，特に m-1a は平均値と第三四分位数が，m-1b は中央値と第三四分位数が全メソッドと比べ優れている．

以上の結果をまとめる．全体を通して優れている最適化手法は m-1a と m-1b であり，特に m-1a は平均値が，m-1b は中央値が優れている．プロジェクト毎には，flex では m-1b，grep では m-1a，make では m-1a, m-1b, m-2b と co.cf が優れたメソッド

表 5.4　メソッド毎に *NAPFD* 値が最良となった回数

	org	rnd	m-1a	m-1b	m-2a	m-2b	co.cf
count	1	2	7	6	2	4	2

となる．RQ2 への回答は，提案手法は他の最適化手法と比べ平均値と中央値が優れているメソッドがあるが，そのメソッドはプロジェクトによって変わる．

5.3 妥当性の検証

5.3.1 外的妥当性

今回，実験対象としたプログラムは 3 種類でありそのバージョンは計 12 個，すべて C 言語で記述された OSS である．他の OSS や商用ソフトウェア，他言語にて結果が一般的であるとはいえない．また実験対象とした SIR は，開発者ではなく第三者が作成したデータである．そのため，データセットに不備がないとはいえない．

5.3.2 内的妥当性

1 つ目に，今回テストスイートの最適化までに用いたプログラムは，組み合わせテスト生成ツールを除き著者の自作であった．プログラムは，十分にテストしているが，フォールトが存在しないとはいえない．2 つ目に，結果の比較にはフォールト検出数推移のグラフと *NAPFD* 値を用いた．*NAPFD* はテストスイートの良さを判断する指標の 1 つではあるが，テストの初期ですべてのフォールトを検出していても *NAPFD* 値が小さい場合がある．

5.4 今後の課題

結果を踏まえた今後の課題を以下に示す．

実験対象として，今回用いたプロジェクト数は 12 個であった．一般化のためにさらに多くの，使用言語や開発環境の異なるプロジェクトを対象にしていく必要がある．

提案手法の比較として取り上げたメソッドは，大きく分けると，元のテストケースの順序，ランダムな順序，他ツールにより生成と最適化をしたテストの 3 通りであった．今後は更に多くの他の最適化手法や生成ツールにて，また手法の組み合わせなどにより比較対象を増やしていく必要がある．

実験対象のバージョンに関して，最適化に用いる重みは最適化対象と同じバージョンのプログラムから導出した．しかし実際の開発では，対象の前バージョンのプロ

グラムから重みを導出し対象を最適化するのが自然な流れである．そのため，今後は回帰テスト時において手法を考察していく必要がある．

メソッドの評価方法について，検出できるフォールト数はどのメソッドも等しかったため，検出数で優劣を付けることができなかった．今後はミューテーションテストを行うことでフォールト数を増やし，フォールト検出数の違いでメソッドの優劣を判断していく必要がある．

6. 結言

本研究では，組み合わせテストの最適化手法として，ベイズ推定とバグレポートを利用した方法を提案した．提案手法を PICT ツールにより生成された組み合わせテストに適用することで最適化が行われていることを確認した．また pricot により再生成と最適化された組み合わせテストと比較することで，プロジェクトによっては最良のメソッドになりえることを確認した．

今後は比較対象の増強や検出するフォールト数の増強を行うことで，*NAPFD* 値以外でもメソッドの優劣を判定可能にする必要がある．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本論文の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学部門水野修准教授，加えて産業技術総合研究所情報技術研究部門ソフトウェアアナリティクス研究グループ崔銀惠研究員に厚く御礼申し上げます．また本論文執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻釆野友紀也君，山田晃久君，藤原剛史君，森啓太君，Anais Tournois 君，Fruy Nicolas 君，本学情報工学課程黒田翔太君，田中健太郎君，西浦生成君，原田禎之君に深く感謝致します．

参考文献

[1] X. Qu, M.B. Cohen, and K.M. Woolf, “Combinatorial interaction regression testing: A study of test case generation and prioritization,” Software Maintenance, 2007. ICSM 2007. IEEE International Conference on, pp.255–264, IEEE, 2007.

[2] Microsoft Corp., Pairwise Independent Combinatorial Testing, （オンライン）, 入手先〈https://github.com/Microsoft/pict〉（参照 2016-02-03）.

[3] H. Do, S. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact,” Empirical Software Engineering, vol.10, no.4, pp.405–435, 2005.

[4] J. Czerwonka, “Pairwise testing in the real world: Practical extensions to test case generators,” Microsoft Corporation, Software Testing Technical Articles, pp.●●–●●, 2008.

[5] E. Choi, T. Kitamura, C. Artho, A. Yamada, and Y. Oiwa, “Priority integration for weighted combinatorial testing,” Proc. of COMPSAC, pp.242–247, IEEE, 2015.

[6] R. Bryce and C. Colbourn, “Prioritized interaction testing for pair-wise coverage with seeding and constraints,” Information and Software Technology, vol.48, no.10, pp.960–970, 2006.

[7] S. Elbaum, A.G. Malishevsky, and G. Rothermel, “Test case prioritization: A family of empirical studies,” Software Engineering, IEEE Transactions on, vol.28, no.2, pp.159–182, 2002.

[8] D.R. Kuhn, D.R. Wallace, and A.M. Gallo Jr, “Software fault interactions and implications for software testing,” Software Engineering, IEEE Transactions on, vol.30, no.6, pp.418–421, 2004.

[9] X. Qu and M.B. Cohen, “A study in prioritization for higher strength combinatorial testing,” Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, pp.285–294, IEEE, 2013.

[10] T.J. Ostrand and M.J. Balcer, “The category-partition method for specifying and generating fuctional tests,” Communications of the ACM, vol.31, no.6, pp.676–686, 1988.