修 士 論 文

題　目　　組み合わせテストにおける
　　コード網羅率の測定に関する研究

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　　13622502

氏　　名　　胡　軼凡

平成27年7月22日提出

様式 2 号

学位論文の要旨（和文）

平成 27 年 7 月 22 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 25 年入学
学生番号　　13622502
氏　　名　　胡　軼凡　㊞

（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

組み合わせテストにおけるコード網羅率の測定に関する研究

2.　論文内容の要旨（400 字程度）

ソフトウェアの品質を保証するためには，対象となるソフトウェアに対して，十分なテストを行うことが求められる．仕様において出現しうる全ての組を実施するテストを「全組み合わせテスト」と呼ぶ．全組み合わせテストは網羅的であるが、システムのサイズが大きくなると現実的には設計・実行が難しいという欠点を持つ．そのため、考慮する組み合わせの数を減らすことで、テストケースの数を減らす手法として，t-way テスト手法が考案された．t-way テスト手法においては，t 個のパラメータの値の組に着目すれば，全ての組がテストスイート内に存在する．文献によれば，t-way テストを実施することで t が 2〜6 の場合でも全組み合わせとほぼ同等の網羅率を持ったテストケースが作成でき，実用的なバグ検出が可能であると言われている．しかし，t-way テストのコード網羅率を調査した研究は少なく，t-way テストとホワイトボックステストにおける網羅率の関係は未だ不明な点が多い．本研究では実際のソフトウェアに対して全組み合わせテストと t-way テストのコード網羅率を調べることで，t-way テストの性質を実証的に示すことを目的とする．実験の結果，t-way テストによるテストスイートは全組み合わせテストに対して非常に少ないコストで，同等のコード網羅率を達成できることが確認された．

Study on measuring code coverages in combinatorial testing

2015 13622502 *Yifan HU*

Abstract

To Assure the software quality, enough testing is required for the target software. To do so, combinations of parameters and values in specification should be tested, and thus such testing is called the combinatorial testing. However, covering all the combinations of parameters and values is neither possible nor practical if the system becomes large. For reducing the number of combinations, t-way testing has been proposed in which combinations of up to t parameters are covered. Since t-way testing is a kind of blackbox testing, it is known that the combination of parameter can be covered. However, it is not known that the t-way tesing can cover code lines and branches as a whitebox testing. In this thesis, we make a tool to analyse code coverages from a given test suite. By using this tool, we analyse code coverages of t-way test suites in three open source software. From the result of experiment, we found that the code coverages of t-way testing become the same as that of all-way testing in many cases. This fact implies that t-way testing can achieve almost the same code coverage with lower cost than the all-way testing.

目　次

1. 緒言

ソフトウェアの品質を保証するためには，対象となるソフトウェアに対して，十分なテストを行うことが求められる．一般的に，ソフトウェアテストはブラックボックステスト (black box test) とホワイトボックステスト (white box test) に分類される．ブラックボックステストは，ソフトウェアの仕様に着目したテストであり，仕様上の入力 (パラメータ) とその値をテスト対象に対して与える．また，ホワイトボックステストは，プログラムの構造に着目したテストのことである．着目する構造には命令や分岐などがあり，注目した構造に対してどれだけの割合の部分を実行できたかを網羅率で表す．

一般的に，ブラックボックステストとホワイトボックステストは補完的な関係にあると言われる．ブラックボックステストは仕様の検査を行うという目的でソフトウェアの意味的なテストを実施することができ，また，そのテストによるバグ検出能力は大きい．一方で，ホワイトボックステストは仕様に依らないテストとなるため，バグ検出能力は低いものの，行・分岐網羅率を 100%にすることで，一度もテストを実施しなかった行や分岐が残るのを防ぐことができる．

ブラックボックステストは仕様におけるパラメータとその値の組を網羅させることで実施できる．出現しうる全ての組を実施するテストを「全組み合わせテスト」と呼ぶ．全組み合わせテストは網羅的であるが，システムのサイズが大きくなると現実的には設計・実行が難しいという欠点を持つ．そのため，考慮する組み合わせの数 (combinatorial strength) を減らすことで，テストケースの数を減らす手法として，t-way テスト手法が考案された [1]．t-way テスト手法においては，t 個のパラメータの値の組に着目すれば，全ての組がテストスイート内に存在する．しかし，全てのパラメータの全ての値を網羅する必要はない．そのため，テストケースの数を劇的に減らすことができる．文献 [1] によれば，t-way テストを実施することで $2 \leq t \leq 6$ の場合でも全組み合わせとほぼ同等の網羅率を持ったテストケースが作成でき，実用的なバグ検出が可能であると言われている．一方で，文献 [2] を除いては，t-way テストのコード網羅率を調査した研究は見られず，t-way テストとホワイトボックステストにおける網羅率の関係は未だ不明な点が多い．

そのため，本研究では実際のソフトウェアに対して全組み合わせテストと t-way テ

ストのコード網羅率を調べることで，t-way テストの性質を実証的に示すことを目的とする．そのため，大量サンプルに適用できる自動的テストツールが必要となった．本研究では，そのような機能を持つ「コード網羅率計測ツール」というツールセットを作成し，様々なソフトウェアに対する t-way テストのコード網羅率を調査し，t-way テストがコード網羅率に対して持つ性質を実証的に示す．

本論文の以降の構成は以下の通りである．2 章では研究の基礎となる用語の整理，および，関連研究について述べる．3 章では本研究での研究設問を定義する．4 章では作成したコードカバレッジ集計計測ツールについて説明する．5 章ではオープンソースプロジェクトのテストスイートリポジトリから得られたデータを用いて，作成したツールからコードカバレッジが得られることを確認する．また，本ツールを用いることで可能になった t-way 組み合わせテストにおけるコードカバレッジの傾向についての考察を行う．6 章でまとめと今後の課題に付いて述べる．

2. 準備

2.1 ソフトウェアテスト

ソフトウェアの品質を保証するためには，対象となるソフトウェアに対して，十分なテストを行うことが求められる．一般的に，テストはブラックボックステスト (black box test) とホワイトボックステスト (white box test) に分類される．

テストを実行する際の用語として以下の 2 つを定義する．

テストケース 単一のテストを実施するための項目．通常，テスト対象ソフトウェアに対するパラメータと値の組を列挙したものとされる．

テストスイート ある目的を満たすためのテストケースの集合．

2.1.1 ブラックボックステスト

ブラックボックステストは，ソフトウェアの仕様に着目したテストであり，仕様上の入力 (パラメータ) とその値をテスト対象に対して与える．内部仕様やコードが手元に無い場合はブラックボックステストを行うしか無い．システムテストや非機能テストは基本的にこちらに属する．

代表的な技法には，同値分割，境界値分析，決定表，状態遷移モデルに基づく技法，プロトコルに基づく技法などがあり，仕様から得られる情報のみを用いたテストケースを作成する．

2.1.2 ホワイトボックステスト

ホワイトボックステストは，プログラムの構造に着目したテストのことである．着目する構造には命令や分岐などがあり，注目した構造に対してどれだけの割合の部分を実行できたかを網羅率で表す．

ホワイトボックステストにおける網羅率の指標として,「行 (命令) 網羅」「分岐網羅」「条件網羅」が存在する．行網羅はテストスイートにおいて，実際に実行された行をソースコードの全行数で割った値として定義される．分岐網羅は，分岐命令の各分岐を通った数を全分岐数で割った値として定義される．条件網羅は各分岐が取り得る値の組をどれぐらい網羅したかで定義される．行網羅と分岐網羅は比較的計

測しやすいが，条件網羅は組み合わせが膨大になるために計測されることは少ない．本研究では，網羅率に指標として行網羅率と分岐網羅率を対象とする．また，これら 2 つの網羅率を総称してコード網羅率と呼ぶ．

2.1.3 それぞれの特徴

一般的に，ブラックボックステストとホワイトボックステストは補完的な関係にあると言われる．ブラックボックステストは仕様の検査を行うという意味でソフトウェアの意味的なテストを実施することができ，また，そのテストによるバグ検出能力は大きい．一方で，ホワイトボックステストは仕様に依らないテストとなるため，バグ検出能力は低いものの，行・分岐網羅率を 100%にすることで，一度もテストを実施しなかった行や分岐が残るのを防ぐことができる．

2.2 組み合わせテスト

組み合わせテスト技法はブラックボックステストにおける入力空間を減らすための一手法である．ブラックボックステストは仕様におけるパラメータと値の組を網羅させることで実施できる．出現しうる全ての組を実施するテストを「全組み合わせテスト」と呼ぶ．全組み合わせテストは網羅的であるが，システムのサイズが大きくなると現実的には設計・実行が難しいという欠点を持つ．そのため，考慮する組み合わせの数 (combinatorial strength) を減らすことで，テストケースの数を減らす手法として，t-way テスト手法が考案された [1]．t-way テスト手法においては，t 個のパラメータの値の組に着目すれば，全ての組がテストスイート内に存在する．しかし，全てのパラメータの全ての値を網羅する必要はない．そのため，テストケースの数を劇的に減らすことができる．文献 [1] によれば，t-way テストを実施することで $2 \leq t \leq 6$ の場合でも全組み合わせとほぼ同等の網羅率を持ったテストスイートが作成でき，実用的なバグ検出が可能であると言われている．

文献 [3] に 2-way 組み合わせテストの例がある．以下にその一節を引用する．

> あるアプリケーションを様々なプラットフォームでテストしたい．そのプラットフォームの構成要素が 5 つある．構成要素は OS(Windows XP, Apple OS X, Red Hat Enterprise Linux)，ブラウザ (Internet Explorer, Firefox),

表 2.1　Pairwise test configurations [3]

テスト	OS	ブラウザ	プロトコル	プロセッサ	データベース
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHEL	IE	IPv6	AMD	MySQL
8	RHEL	Firefox	IPv4	Intel	Sybase
9	RHEL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

プロトコルスタック (IPv4, IPv6)，プロセッサ (Intel, AMD)，とデータベース (MySQL, Sybase, Oracle) である．ここで考えられるプラットフォームは合計 $3 \times 2 \times 2 \times 2 \times 3 = 72$ 個となる．これを 2-way テストスイートとして作成すると，表 2.1 のようになる．ここでは 10 個のテストケースで，1 つの要素毎に他の 1 つの要素との間での組み合わせが少なくとも 1 回出現する．こうすると，10 回で 2 要素の組み合わせが全てテストされることになる．

このように，t-way 組み合わせテストを構成することで，テストスイートに含まれるテストケースの数を大幅に減らすことができる．ただし，最適な t-way テストを発見することは大変難しい．

t-way テストケースを生成するツールとして Microsoft が開発した PICT (Pairwise Independent Combinatoriak Testing tool) [4] などが広く使われている．本研究で利用した組み合わせテストのテストケースは PICT を用いて生成されている．

```
# grep_2.3 (test Regular expression)  NOTE---- Reverse inherit from v1
# Usage: target [OPTION]... PATTERN [FILE]...
# Search for pattern in each FILE or standard input.
#************************************************************************
#Function unit 1: test all kinds of regular expression

Parameters:

        Regexp selection and interpretation:
                -E.                                     [property Extended, NonFixed]
                -F.                                     [property Fixed]
                -G, no option.                          [property Basic, NonFixed]

        Quoting:
                pattern is single quoted.               [property SingleQuoted, Quoted]
                pattern is double quoted.               [property DoubleQuoted, Quoted]
                pattern is not quoted.                  [property NotQuoted]
                #pattern is improperly quoted.
[error] #can't be successfully tested
 ...
```

図 2.1　TSL 形式の例

2.3　コードカバレッジツール

ホワイトボックステストにおける網羅率の指標の内，行網羅と分岐網羅を計測するツールは多く存在する．代表的なものとして，GNU のコンパイラに対するカバレッジ計算ツール gcov [5] がある．

gcov はコードの性能を測るために以下のような情報を提供する．

- どれぐらい頻繁に各行が実行されているか
- コードのどの行が実際に実行されているか
- コードのどの分岐が実際に実行されているか
- あるコードの部分を実行するのにどれぐらいの時間がかかるか

gcov を利用するには，gcc のオプションに以下の 2 つのオプションを追加すればよい．このオプションを追加して実行されたバイナリからは，コードの各行や分岐に対する実行情報が自動的に収集される．

```
$ gcc -fprofile-arcs -ftest-coverage tmp.c
```

```
-P[-n I ../inputs/grep1.dat]
-P[-n 1 ../inputs/grep1.dat]
-P[-E '\?' ../inputs/grep1.dat]
-P[-G '\?' ../inputs/grep1.dat]
-P[-G '?' ../inputs/grep1.dat]
-P[-F '?' ../inputs/grep1.dat]
-P[-F '/*#include' ../inputs/grep1.dat]
-P[-G '[1-5\]' ../inputs/grep1.dat]
-P[-E -c '[^a-z\]' ../inputs/grep1.dat]
-P[-G '[X^YZ\]' ../inputs/grep1.dat]
-P[-E '[1-\]' ../inputs/grep1.dat]
-P[-G '\w' ../inputs/grep1.dat]
-P[-G '[[:alpha:\]\]' ../inputs/grep1.dat]
-P[-E "[[:cntrl:\]\]" ../inputs/grep1.dat]
-P[-E "[[:digit:\]\]" ../inputs/grep1.dat]
-P[-E "[[:graph:\]\]" ../inputs/grep1.dat]
-P[-G "[[:lower:\]\]" ../inputs/grep1.dat]
-P[-G "[[:print:\]\]" ../inputs/grep1.dat]
-P[-G '[[:punct:\]\]' ../inputs/grep1.dat]
-P[-G '[[:space:\]\]' ../inputs/grep1.dat]
...
```

図 2.2　Universe 形式の例

2.4　SIR リポジトリ

Do らはソフトウェアテスト研究のために，ソフトウェアと対応するテストケースのリポジトリを作成し，University of Nebraska-Lincoln において公開している [6]．これを Software-artifact Infrastructure Repository (SIR) リポジトリと呼んでいる．

リポジトリに含まれる多くのテストプロジェクトには，テストケースとして汎用テスト記述言語 TSL で記述された TSL ファイルや，TSL ファイルを実行可能な形式

```
_,_,1,_,_,_,_,_,_,_,_,_,_
_,_,2,_,_,_,_,_,_,_,_,_,_
_,_,3,_,_,_,_,_,_,_,_,_,_
...
1,1,0,1,9,12,0,1,1,2,2,1,1,1
1,1,0,1,9,12,0,1,3,0,2,1,1,1
1,1,0,1,9,12,0,4,1,0,2,1,1,1
1,1,0,1,9,12,0,4,3,0,2,1,1,1
...
```

図 2.3　TS 形式の例

としたUniverseファイルなどが格納されている．

TSL形式 仕様におけるパラメータとその値の組を抽出し，どういう意味を持つパラメータで取り得る値の値域などを記述したものである．図2.1に一例を示す．

Universe形式 TSLファイルの内容を現実のソフトウェアテストで実行できるように変換したものを集めたものである．図2.2に一例を示す．例えば，GNU grepのようなコマンドラインツールの場合，入力ファイルやオプションの与え方を具体的に示し，そのままテストが実行できるような形式になっている．

TS形式 仕様におけるパラメータとその値の組を数値化し，各種のテストケース生成ツールにおいて扱い易くした形式である．図2.3に一例を示す．

2.5 関連研究

文献[1]ではKuhnらが4つの事例に対して，不具合を起こすparameter interaction数を調査している．その結果，ソフトウェアのほとんどのバグは，2〜6個の少ないparameter間のinteractionで発生し，pair-wiseから，3〜4 way testingでもかなりの不具合を検出できる，という根拠に使われている．この論文がt-wayテストに与えた影響は大きく，特に本論文のFig. 1は多くの文献に引用されている．また，文献[3]でも同様の事例が紹介されている．Zhangらは商用MP3プレイヤーに対して4-wayテストを実施し，ですべてのinteraction faultsをカバーできたことを報告している[7]．

GiannakopoulouらはNASAにおいて3-way testと他のテストスイートのコード網羅率を調査している[2]．我々の知る限り，組み合わせテスト環境下におけるコード網羅率はこの論文でしか調査されていない．本研究では，一般のt-wayテストに対してカバレッジを調査できる環境を整備し，3つのオープンソースプロジェクトに対する調査を行う．

組み合わせテストに関する研究はこれまでにも多く行われているが，コード網羅率を多くのソフトウェアに対して計測する試みは未だ行われていない．本研究の新規性はこの点にある．

3. 研究の目的

3.1 研究の背景

組み合わせテストは，既にソフトウェアテスト技術者にとっては欠かせないツールとなっている．その技術は，およそ 25 年前から知られているが [8]，幅広く使われるようになったのはごく最近のことである．現在のところ，組み合わせテストケース作成できるツールはおよそ 40 個公開されている [4].

組み合わせテストの有効性は Kuhn らによる研究 [1] が有名であり，2〜6 の組み合わせを網羅すれば，全組み合わせテストにほぼ匹敵する性能のテストスイートが得られるとされている．一方で，我々の知る限り，文献 [2] を除いては，t-way テストのコード網羅率を調査した研究は見られず，t-way テストとホワイトボックステストにおける網羅率の関係は未だ不明な点が多い．

そのため，本研究では実際のソフトウェアに対して全組み合わせテストと t-way テストのコード網羅率を調べることで，t-way テストの性質を実証的に示すことを目的とする．

3.2 研究設問

本研究の目的を明確にするために以下の研究設問を設定する．

RQ1 コードカバレッジツールを用いて，テストケース実行順の時系列における行・分岐網羅率の推移を可視化できるか．

RQ2 全組み合わせテストと t-way テストによって，網羅率の推移はどのように変化するか．

RQ3 t-way テストは全組み合わせテストに対して優れていると言えるか．

設問 RQ1 が可能となれば，コードカバレッジツールにより各テストスイートの実行を視覚的に評価できるようになる．また，詳細に分析することで，各テストスイートの特徴を分析できるようになる．その上で，設問 RQ2 では，t-way テストと全組み合わせテストの特徴を実際にテストケースを実行することで分析することを目的とする．また，設問 RQ3 では t-way テストによる少数のテストケースによって，

全組み合わせテストによる網羅率がどの程度保証できるかどうかを検証する．具体的には，削減できたテストケース数と到達できなかった網羅率の割合を評価する．

4. コード網羅率計測ツール

4.1 各ツールの仕様

4.1.1 網羅率計測ツール (coverage.pl)

目的 与えられた実行可能なテストスイートに基づき，テストを自動的に実行するツールである．

入力 実行可能なテストスイートファイル (.universe)，テスト対象ソフトウェア

出力 網羅率を記録したファイル (.tsv)

動作 テスト実行する際には，まず与えられたテストスイート (Universe 形式ファイル) から実行可能なテストケース (パラメータの組合わせ) を取得し，テスト対象ソフトウェアに実行させる．対象ソフトウェアはあらかじめカバレッジ計測用にビルドされており，網羅率集計用の情報が記録される．テストにおいて記録された情報を gcov が集計し，テストを実行した回数，実行されたパラメータ，実行したしたソースファイル，行網羅率，分岐網羅率を記録する．この作業をテストスイートに含まれるテストケースについて繰り返す．最終的に，各テストケースに対する網羅率が実行順に記録されたファイルが出力となる．

4.1.2 データ変換ツール (result_reform.pl)

目的 ソースファイル毎に別々になっている `coverage.pl` の出力を統計言語 R で扱いやすい形式に変換する．

入力 網羅率を記録したファイル (.tsv)

出力 統合された網羅率を記録したファイル (.tsv)

動作 `coverage.pl` が記録した網羅率は，ソースファイル毎に別々になっている状況なので，適宜必要な計算を行い，カテゴリカル変数を加えた上で R で扱い易い形式に変換する．全てのテストケースに対するソースファイルの行数や分岐数の総和に対する網羅率を集計する．

4.1.3 グラフ作成ツール (linegraph.R.pl)

目的 各テストスイートの実行を視覚的に評価するため，網羅率の推移の記録を R により線グラフとして出力する．

入力 統合された網羅率を記録したファイル (.tsv)

出力 各テストスイートの実行結果を表す線グラフ (.pdf)

4.1.4 実行可能テストスイート生成ツール (mkUniverseFromCT.pl)

目的 組み合わせテストの出力は，テストケースを数値化して表した TS 形式と呼ばれる形で提供される．この出力を実行可能なテストスイート (Universe 形式) に変換する．

入力 数値化されたテストスイートファイル (.ts)

出力 実行可能なテストスイートファイル (.universe)

4.2 ツール適用の流れ

ツール適用の流れを図 4.1 に示す．

1. テスト対象ソフトウェアにおいて，gcov を適用できるように，`-fprofile-arcs` および`-ftest-coverage` オプション追加して make したものを作成する．
2. テスト対象ソフトウェアに対する実行可能なテストケースを準備する．t-way テストを実施する場合は，ここで `mkUniverseFromCT.pl` により，実行可能なテストケースを生成する．
3. `coverage.pl` を実行し，テスト対象ソフトウェアにおける行網羅率と分岐網羅率を計測する．
4. `reslt_reform.pl` を用いて，テストを実施した全ソースファイルについての網羅率データを統一して，グラフ描画用に変換する．
5. `linegraph.R.sh` を用いて，テスト対象ソフトウェアの各バージョン毎に行網羅率と分岐網羅率の時系列変化を示したグラフを得る．

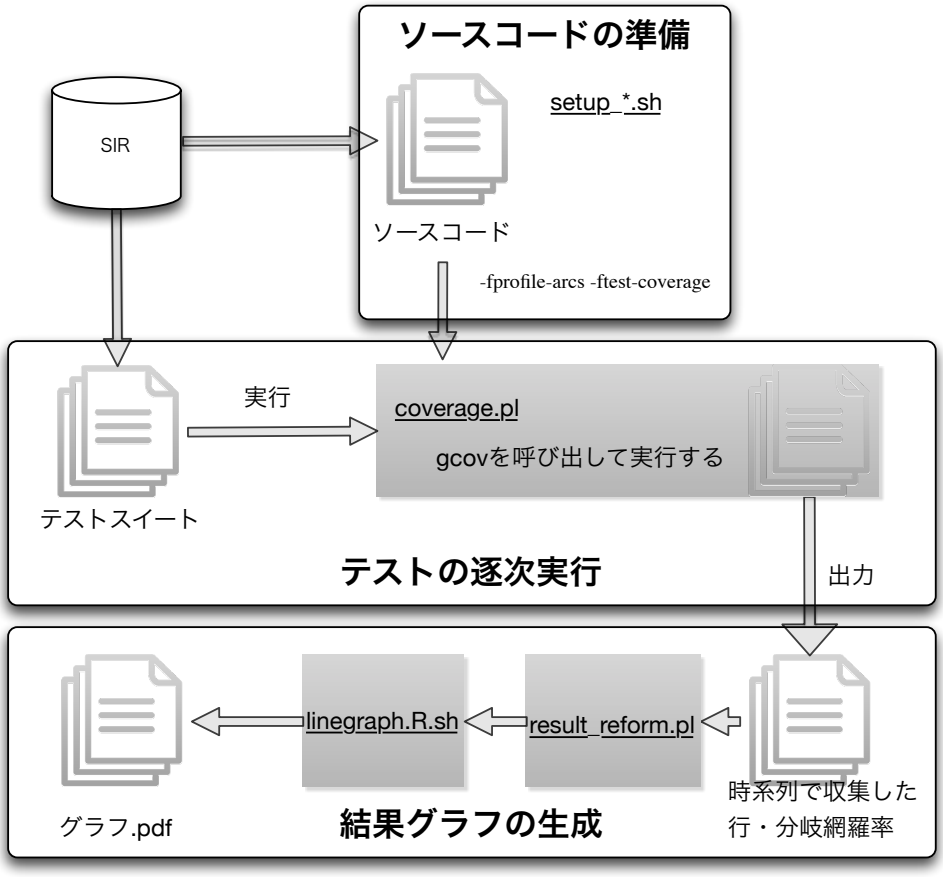

図 4.1　作成したコード網羅率計測ツールの流れ

対象とする全てのソフトウェアに対して，以上の適用を繰り返す．なお，対象とするソフトウェア毎に少しずつ適用方法が異なる．これはそもそも各ソフトウェアに対するテストの実行方法が多様であることに起因する．

5. コード網羅率計測実験

5.1 実験の目的

本実験では，GNU grep，GNU flex，GNU make の各バージョンを実験対象として，Code coverage tools を適用する．適用実験では，各バージョンに対して「全組み合わせテスト」と「t-way テスト」を行い，それぞれについて，行・分岐網羅率の時系列データを収集する．そのデータを分析することで，全組み合わせテストと t-way テストの特徴を明らかにする．

5.2 利用したデータ

本実験における実験データの取得先について述べる．まず，テストスイートのデータは SIR リポジトリより取得した．SIR リポジトリ内で C 言語で記述され，また，実行可能なテストケースを持ったものとして取得したもので，GNU grep [9], GNU flex [10], GNU make [11] が利用可能であった．さらに，これらのソフトウェアに対応するソースコードを取得した．今回利用したものは全て GNU のプロジェクトであったため，GNU の Web サイト [12] より対応するソースコードを取得した．

利用可能であった実験サンプルは GNU grep の 6 バージョン（v0: grep-2.0, v1: grep-2.2, v2: grep-2.3, v3: grep-2.4, v4: grep-2.4.1, v5: grep-2.4.2），GNU flex の 6 バージョン（v0: flex-2.4.3, v1: flex-2.4.7, v2: flex-2.5.1, v3: flex-2.5.2, v4: flex-2.5.3, v5: flex-2.5.4）と GNU make の 4 バージョン（v1: make-3.76.1, v2: make-3.77, v3: make-3.78.1, v4: make-3.79.1）の合わせて 16 種類であった．

なお，今回の実験では SIR リポジトリにおいて作成されていたテストスイートを「全組み合わせテスト」として採用している．以降では “all” と表記されるものがこの全組み合わせテストを意味する．

5.3 実験環境

本実験は MacBook Air (CPU 2 GHz Intel Core i7, 8GB 1600MHz DDR3, APPLE SSD SM256E Media) OS X 10.9.5 Mavericks の下で実施した．

5.4 t-way 組み合わせテストの作成

共同研究を実施している (独) 産業技術総合研究所において開発された t-way テスト生成ツールを利用する．テスト生成ツールの出力はそのままでは実行可能なテストケースとならないため，本研究で作成した `mkUniverseFromCT.pl` を利用して実行可能な Universe 形式に変換する．

今回の組み合わせテストは，必ず全組み合わせテストのサブセットとして生成されることになっている．そのため，t-way テストにおける行網羅率・分岐網羅率の最大値が全組み合わせテストの値を上回ることはない．

5.5 実験の実施

実験は前章で解説したツールを適用する順に応じて，以下の手順で行う．なお，以下の手順は GNU grep に対する実施手順であるが，他のプロジェクトでも基本的に同様である．

1. テストスイートのバージョンに対応したソースコードを取得し「実験対象/originals/」ディレクトリに配置し，テストツールを「実験対象/source/」に配置する．
2. `setup_grep.sh` を使い，「実験対象/originals」にあるアーカイブを展開し，Makefile に -fprofile-arcs -ftest-coverage を追加し各バージョンのバイナリを make する．
3. `mkUniverseFromCT.pl` を利用して t-way テスト用のテストスイート (.universe ファイル) を作成する．
4. `coverage.pl` を呼び出して，実験対象に対して全組み合わせテスト，および，t-way テストを実行する．
5. `result_reform.pl` を使い，得られたテスト実行結果をグラフ生成に適した形式へと変換する．
6. `linegraph.R.sh` を使い，全ての結果に対するグラフを PDF で生成する

この手順に基づき，各プロジェクトのバージョン全てに対してコードの行網羅率と分岐網羅率を共に計測し，合計 32 個の実験結果を記録した．

表 5.1　Line coverage for each version (GNU grep)

テスト	テストケース数	v0	v1	v2	v3	v4	v5
1-way	40	0.58	0.47	0.45	0.45	0.45	0.45
2-way	77	0.58	0.47	0.46	0.45	0.45	0.45
3-way	180	0.59	0.47	0.46	0.45	0.45	0.45
4-way	326	0.59	0.47	0.46	0.45	0.45	0.45
all	470	0.59	0.47	0.46	0.45	0.45	0.45

表 5.2　Branch coverage for each version (GNU grep)

テスト	テストケース数	v0	v1	v2	v3	v4	v5
1-way	40	0.56	0.42	0.42	0.42	0.42	0.42
2-way	77	0.56	0.43	0.43	0.43	0.43	0.43
3-way	180	0.57	0.44	0.43	0.43	0.43	0.43
4-way	326	0.57	0.44	0.44	0.44	0.44	0.44
all	470	0.57	0.44	0.44	0.44	0.44	0.44

図 A.1 から図 A.6 に GNU grep に対するテスト実行における行網羅率，分岐網羅率の推移を示す．また，図 A.7 から図 A.12 には GNU flex に対する結果を，図 A.13 から図 A.16 には GNU make に対する結果をそれぞれ示す．

5.6　実験結果の分析

GNU grep において，t-way テストと全組み合わせテストの全テストケースを実行し終わった時点での行網羅率と分岐網羅率を表 5.1 と表 5.2 にそれぞれ示す．同様に，GNU flex に対して行網羅率と分岐網羅率を示したものを表 5.3 と表 5.4 に，GNU make に対するものを表 5.5 と表 5.6 に示す．

以降では 3.2 節における研究設問に沿って分析を行う．

表 5.3　Line coverage for each version (GNU flex)

テスト	テストケース数	v0	v1	v2	v3	v4	v5
1-way	30	0.82	0.80	0.76	0.76	0.76	0.76
2-way	51	0.83	0.83	0.79	0.78	0.78	0.78
3-way	90	0.83	0.83	0.79	0.78	0.78	0.78
4-way	154	0.83	0.83	0.79	0.78	0.78	0.78
all	525	0.83	0.83	0.79	0.78	0.79	0.79

表 5.4　Branch coverage for each version (GNU flex)

テスト	テストケース数	v0	v1	v2	v3	v4	v5
1-way	30	0.92	0.91	0.80	0.80	0.80	0.80
2-way	51	0.93	0.93	0.82	0.81	0.81	0.82
3-way	90	0.93	0.93	0.82	0.81	0.81	0.82
4-way	154	0.93	0.93	0.82	0.81	0.81	0.82
all	525	0.93	0.93	0.82	0.82	0.82	0.82

5.6.1　RQ1: コードカバレッジツールを用いて，テストケース実行順の時系列における行・分岐網羅率の推移を可視化できるか

開発したツール群を用いて，3 種類の OSS ソフトウェアの 4〜6 バージョンに対して，Universe 形式のテストケースを準備し，テストを実行することができた．テストを実行した結果が図 A.1 に代表されるグラフである．この結果から，今回対象とした全てのソフトウェアに対して，テスト実行時の行・分岐網羅率の推移を可視化することができたと考える．

5.6.2　RQ2: 全組み合わせテストと t-way テストによって，網羅率の推移はどのように変化するか

Combinatorial strength (t) が 1 の場合，網羅率の上昇速度がもっとも早くなることが確認された．これは，$t = 1$ の場合に，少ないテストケースである程度の網羅率を得ることが可能であることが示されている．一方で，combinatorial strength が増える

表 5.5　Line coverage for each version (GNU make)

テスト	テストケース数	v1	v2	v3	v4
1-way	27	0.46	0.46	0.46	0.43
2-way	34	0.46	0.46	0.46	0.43
3-way	45	0.46	0.46	0.46	0.43
4-way	69	0.46	0.46	0.46	0.43
all	793	0.46	0.46	0.46	0.43

表 5.6　Branch coverage for each version (GNU make)

テスト	テストケース数	v1	v2	v3	v4
1-way	27	0.54	0.54	0.55	0.51
2-way	34	0.54	0.54	0.55	0.51
3-way	45	0.54	0.54	0.55	0.51
4-way	69	0.54	0.54	0.55	0.51
all	793	0.54	0.54	0.55	0.51

と共に，上昇速度は緩やかになるが，最終的な網羅率の高さは増加してくることがわかる．$t = 4$ の場合，最終的な網羅率が全組み合わせテストと同じになる割合は 87.5% (28/32) であった．

以下に，各プロジェクトで観察された特徴を述べる．

GNU grep どのバージョンでも行網羅率のほうが分岐網羅率よりも高い．全結果の中で t-way テストが全組み合わせテストと同じ行網羅率を達成した回数は 21/24（87.5%）で，分岐網羅率の回数は 8/24(33.3%) である．t-way テストが提供した最終の網羅率が t の変化により著しく違う数値となっている．4-way テストでは全組み合わせテストと同じ網羅率が提供できている．全組み合わせテストの行・分岐網羅率の上昇率を基準にすると，t-way テストの行・分岐網羅率の上昇率のほうが急であることが確認できる．行・分岐網羅率の上昇が速い場合，ソフトウェアにとって重要なテストケースが先に実行されたことを示している．これにより，不具合の探知も速くできて，それに対する修復作業も早めに開始

できると期待できる．

GNU flex どのバージョンでも分岐網羅率のほうが行網羅率よりも高くなる．t-way テストが全組み合わせテストと同じ行・分岐網羅率を達成した回数は両方同じ 12/24(50%) になっている．また，1-way テストを除いた t-way テストの行・分岐網羅率は全て同値となる．その結果，全組み合わせテストと同等の網羅率を実現した回数は 8/12(66.67%) であった．また，t-way テストの網羅率上昇はほとんどの場合，全組み合わせと同様の動きであるが，最後の数%の上昇は t-way テストのほうが早い．

GNU make t-way テストのテストケース数が非常に少なくなり，t の値によるテストケース数の違いが最も少ない．また，全バージョンにおいてすべての t-way テストが全組み合わせテストと同等の網羅率を実現できている．

これらの観察により，網羅率推移の特徴はテスト対象ソフトウェアの特徴も関係していると考えられる．

5.6.3 RQ3: t-way テストは全組み合わせテストに対して優れていると言えるか

本実験において，t-way テストが全組み合わせテストと同じ網羅率を達成した回数をまとめる．

- 1-way: 37.50% (12/32)
- 2-way: 65.63% (21/32)
- 3-way: 75.00% (24/32)
- 4-way: 87.50% (28/32)

一方で，実行したテストケースの数は次のようにまとめられる．

- 1-way: 5.43% (97/1788)
- 2-way: 9.06% (162/1788)
- 3-way: 17.62% (315/1788)
- 4-way: 30.70% (548/1788)

このことから，4-way テストを用いた場合，全組み合わせの 30%程度のテストケー

ス数を実行することで，ほとんどの場合，全組み合わせと同等のコード網羅率を達成できることが分かる．

また，全組み合わせと同じ網羅率を達成できないとはいえ，表 5.1〜表 5.6 からは，例えば 1-way テストであっても網羅率はほとんどの場合全組み合わせに対してわずかに低い程度である．

このことから，*t*-way テストは行網羅率・分岐網羅率に対しても全組み合わせテストに遜色ない性能を持つことが分かる．一方で，4-way テストであってもテストケース数を 70%削減でき，2-way テストであれば 90%近くのテストケースを削減できる．

以上より，*t*-way テストは全組み合わせテストに対して優れていると結論する．

6. 結言

6.1 まとめ

ソフトウェアの品質を保証するためには，対象となるソフトウェアに対して，十分なテストを行うことが求められる．仕様において出現しうる全ての組を実施するテストを「全組み合わせテスト」と呼ぶ．全組み合わせテストは網羅的であるが、システムのサイズが大きくなると現実的には設計・実行が難しいという欠点を持つ．そのため、考慮する組み合わせの数を減らすことで、テストケースの数を減らす手法として，t-way テスト手法が考案されている．

本研究では実際のソフトウェアに対して全組み合わせテストと t-way テストのコード網羅率を調べることで，t-way テストの性質を実証的に示すことを目的とする．そのために，SIR リポジトリから得られたテストスイートを用い，GNU grep, GNU flex, GNU make を用いた組み合わせテストを実施して，行・分岐網羅率の調査を行った．その結果，4-way テストの 87.5%が全組み合わせで得られる行・分岐網羅率を達成できることを確認した．また，4-way テストのテストケース数は全組み合わせテストの 30.7%程度であり，少ないコストで全組み合わせ同等の行・分岐網羅率が得られることが分かった．実用的に考えれば，t-way テストがほとんどの場合，全組み合わせより非常に少ないテストケースで同等の行・分岐網羅率ができることから，ソフトウェアテスト技術者にとって，t-way テストはかなり実用なツールになり得ると考える．

6.2 今後の課題

実験で用いた各プロジェクトにおいては，t-way テスト実施時の行・網羅率の推移の特徴が異なることが分かる．

この違いの原因を考えると，t-way テストで選択されたテストケースとソフトウェアの制御構造に関連があるのではないかと考える．その関連性が解明した時，ソフトウェアテスト技術者がソフトウェアの特徴を考え，適切な組み合わせ数を選び，最小のテストケース数で最大の網羅率を得られることも可能になると考えている．

今回は 3 つのプロジェクトでの適用実験を行ったが，一般性を高めるためにはよ

り多くのプロジェクトでの検証が必要になると考えられる．

また，ソフトウェアテストの目的は不具合を発見し，ソフトウェアの品質を保証することである．テスト網羅率が高ければ，不具合を発見率も高くなる．本研究では *t*-way テストの行・分岐網羅率の関係を研究した．今後，*t*-way テストと不具合検出の間の関連性も調べる必要があると考える．

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・人間科学系水野修准教授に厚く御礼申し上げます．研究課題の設定および *t*-way テストケースの生成にご協力頂きました (独) 産業技術総合研究所研究員 崔銀惠様に感謝します．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 河端駿也君，山田晃久君，釆野友紀也君，藤原剛史君，森啓太君，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，本学研究生 崔超君，および，学生生活を通じて筆者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] D.R. Kuhn, D.R. Wallace, and A.M. Gallo, Jr., “Software fault interactions and implications for software testing,” IEEE Trans. Softw. Eng., vol.30, no.6, pp.418–421, June 2004.

[2] D. Giannakopoulou, D.H. Bushnell, J. Schumann, H. Erzberger, and K. Heere, “Formal testing for separation assurance,” Annals of Mathematics and Artificial Intelligence, vol.63, no.1, pp.5–30, Sept. 2011.

[3] D.R. Kuhn, R.N. Kacker, and Y. Lei, “Practical combinatorial testing,” Technical Report NIST Special Publication 800-142, National Institute of Standards and Technology, 2010.

[4] J. Czerwonka, Pairwise Testing – Combinatorial Test Case Generation, （オンライン）, 入手先〈http://www.pairwise.org〉（参照 2015-7-11）.

[5] GNU, gcov – a Test Coverage Program, （オンライン）, 入手先〈https://gcc.gnu.org/onlinedocs/gcc/Gcov.html〉（参照 2015-7-11）.

[6] H. Do, S. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact,” Empirical Software Engineering, vol.10, pp.405–435, 2005.

[7] Z. Zhang, X. Liu, and J. Zhang, “Combinatorial testing on ID3v2 tags of mp3 files,” Proceedings of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp.587–590, 2012.

[8] K. Tatsumi, “Test case design support system,” Proceedings of the International Conference on Quality Control (ICQC), pp.615–620, 1987.

[9] GNU, GNU grep, （オンライン）, 入手先〈http://www.gnu.org/software/grep/〉（参照 2015-7-10）.

[10] GNU, GNU flex, （オンライン）, 入手先〈http://www.gnu.org/software/flex/〉（参照 2015-7-10）.

[11] GNU, GNU make, （オンライン）, 入手先〈http://www.gnu.org/software/make/〉（参照 2015-7-10）.

[12] GNU, GNU operating system, （オンライン）, 入手先〈http://www.gnu.org/〉（参照 2015-7-10）.

付録 A.　図表

5 章での実験結果グラフを以下に掲載する.

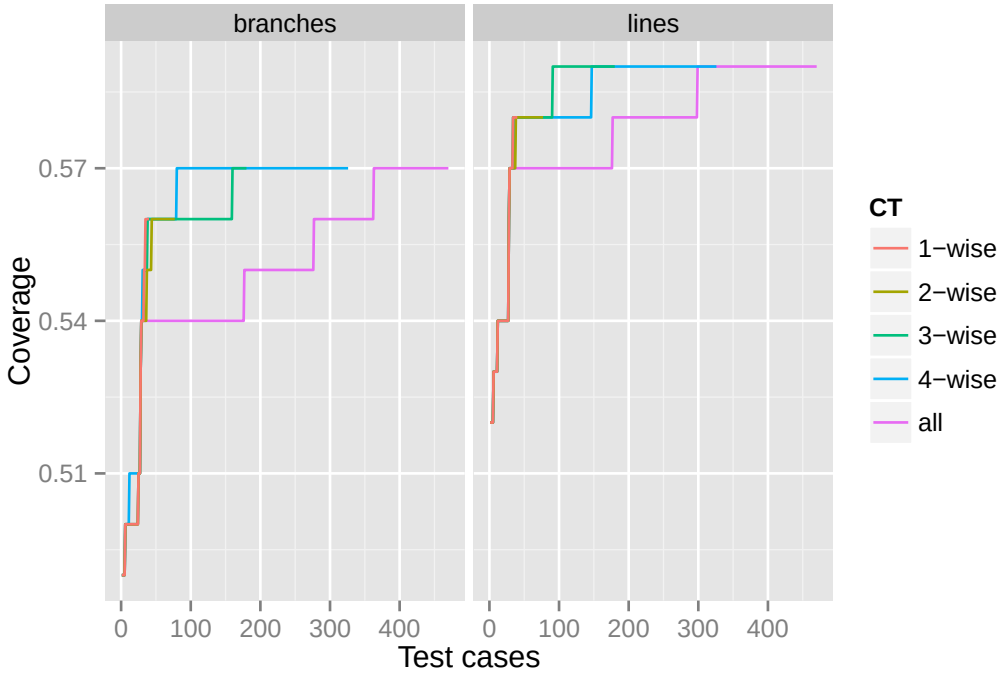

図 A.1　Lines and branches coverage. (GNU grep, v0.)

図 A.2　Lines and branches coverage. (GNU grep, v1.)

図 A.3　Lines and branches coverage. (GNU grep, v2.)

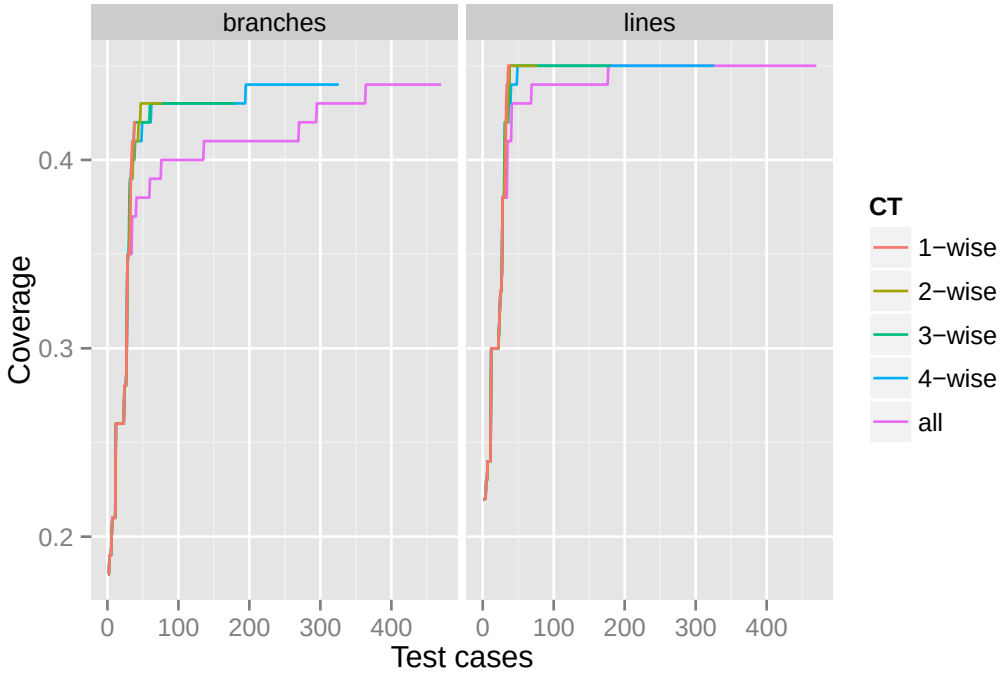

図 A.4　Lines and branches coverage. (GNU grep, v3.)

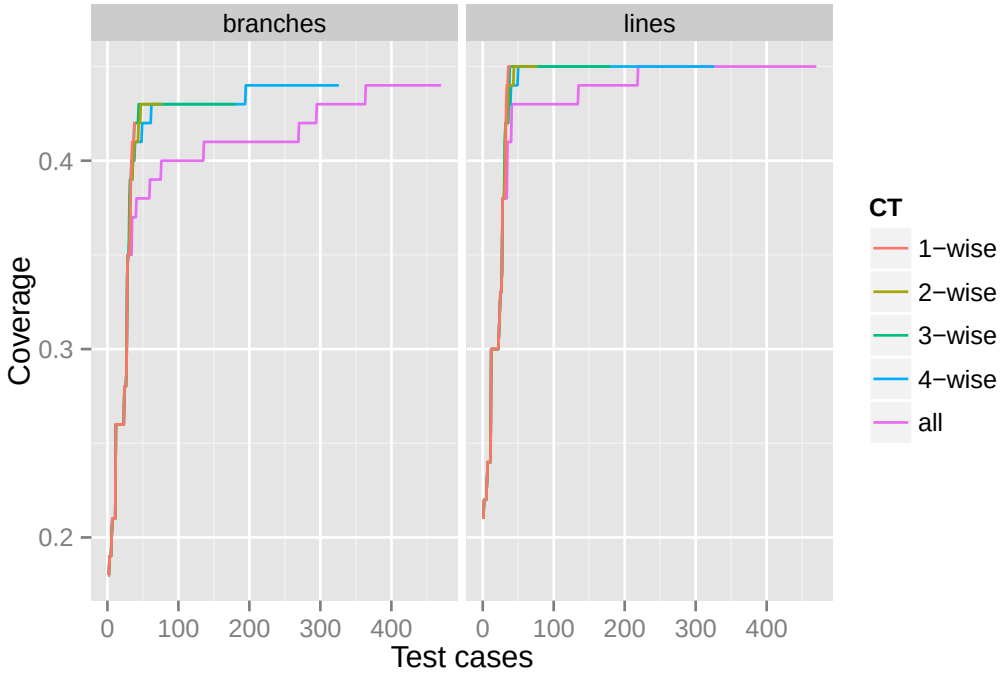

図 A.5　Lines and branches coverage. (GNU grep, v4.)

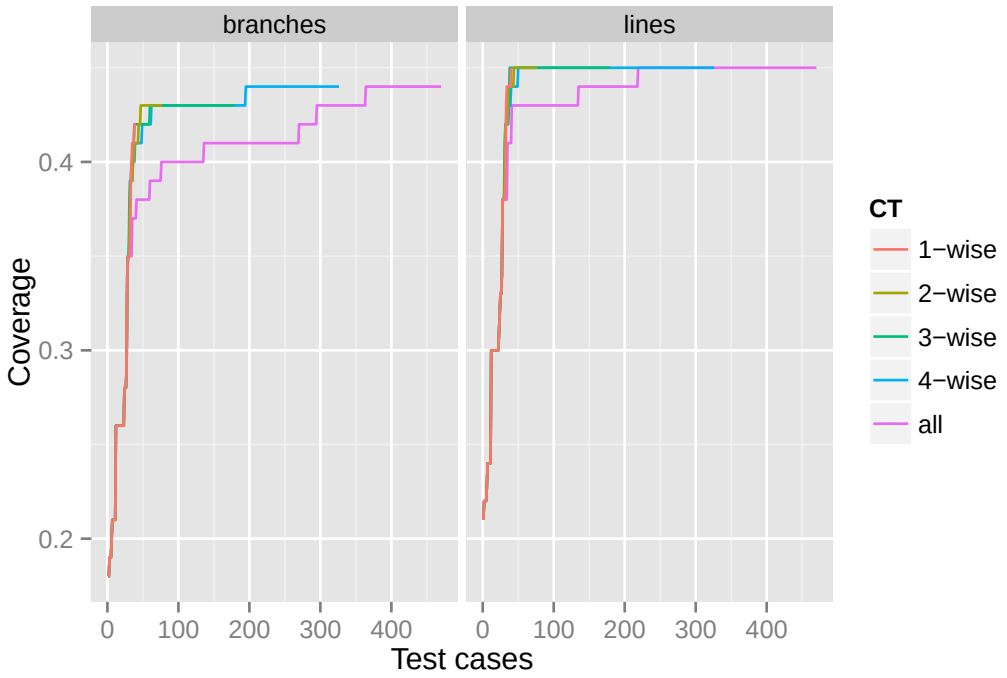

図 A.6　Lines and branches coverage. (GNU grep, v5.)

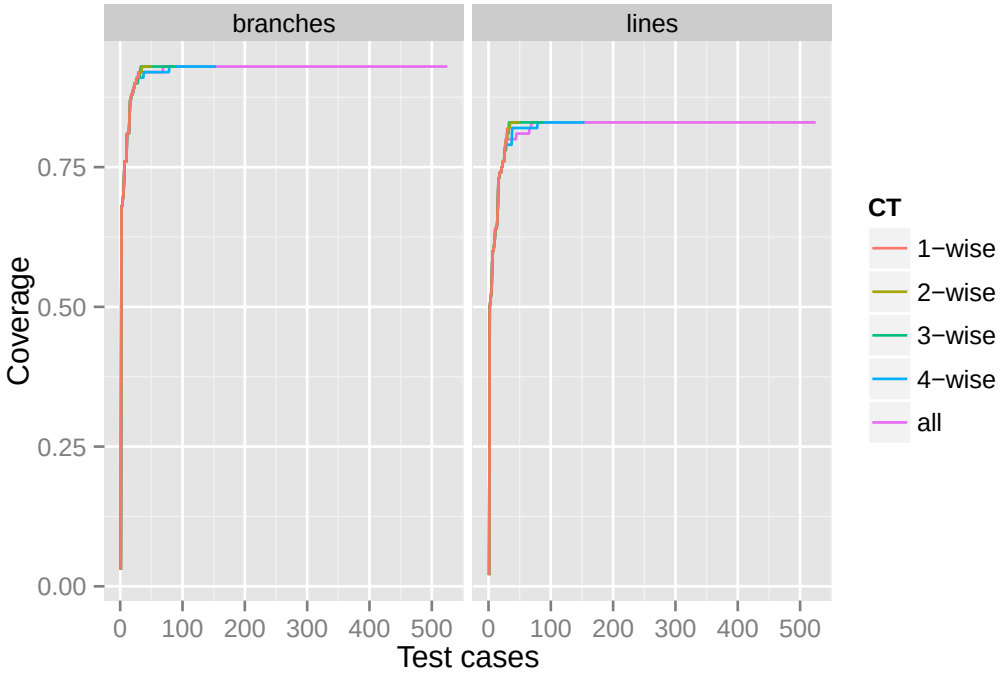

図 A.7　Lines and branches coverage. (GNU flex, v0.)

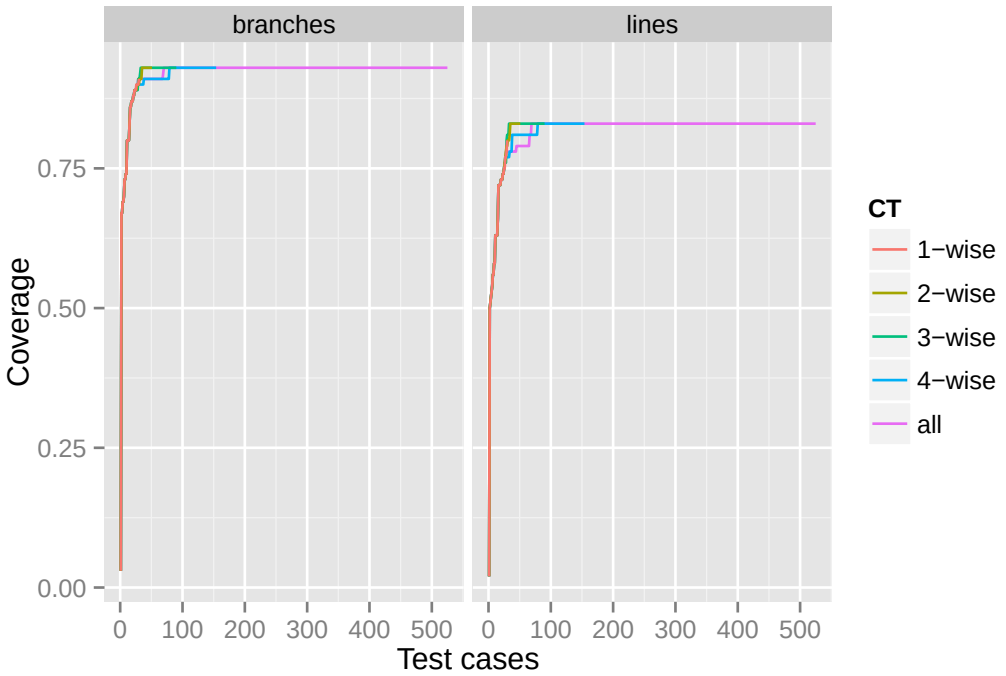

図 A.8　Lines and branches coverage. (GNU flex, v1.)

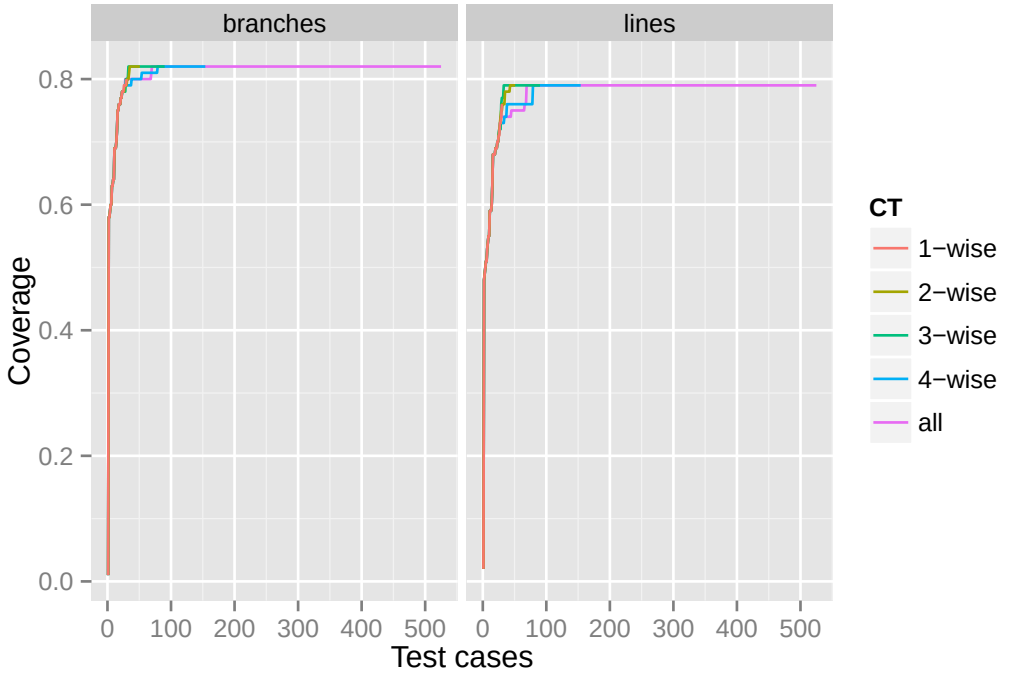

図 A.9　Lines and branches coverage. (GNU flex, v2.)

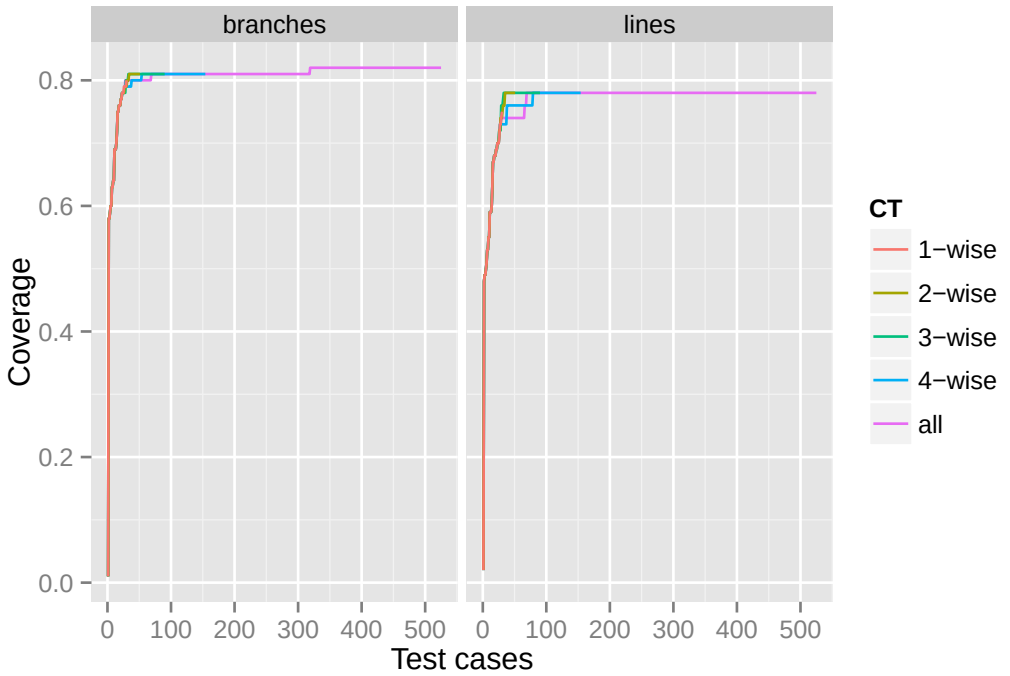

図 A.10　Lines and branches coverage. (GNU flex, v3.)

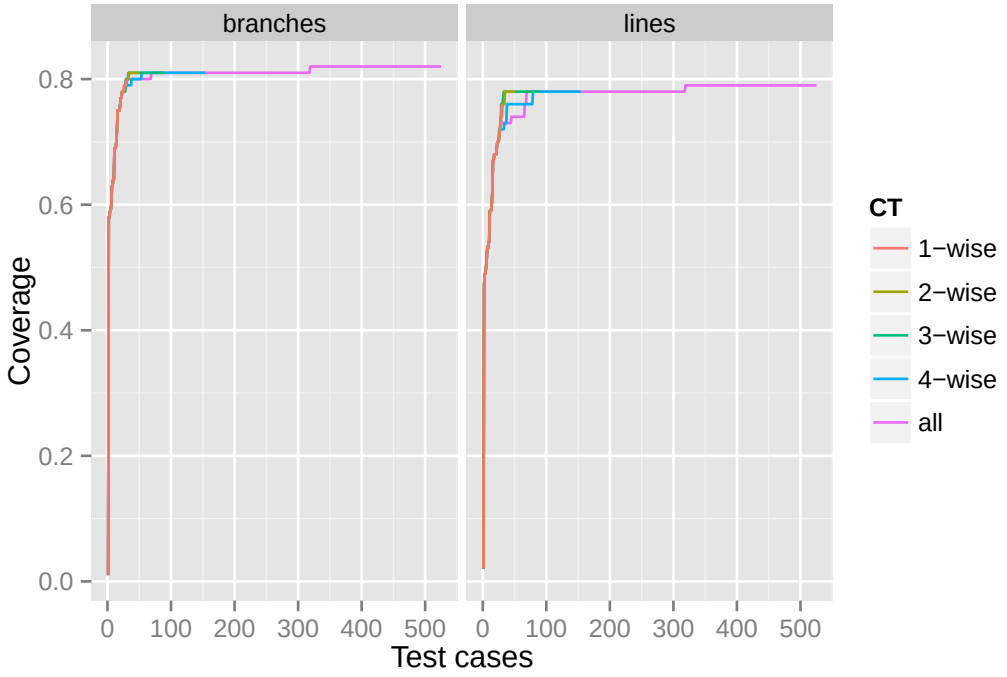

図 A.11　Lines and branches coverage. (GNU flex, v4.)

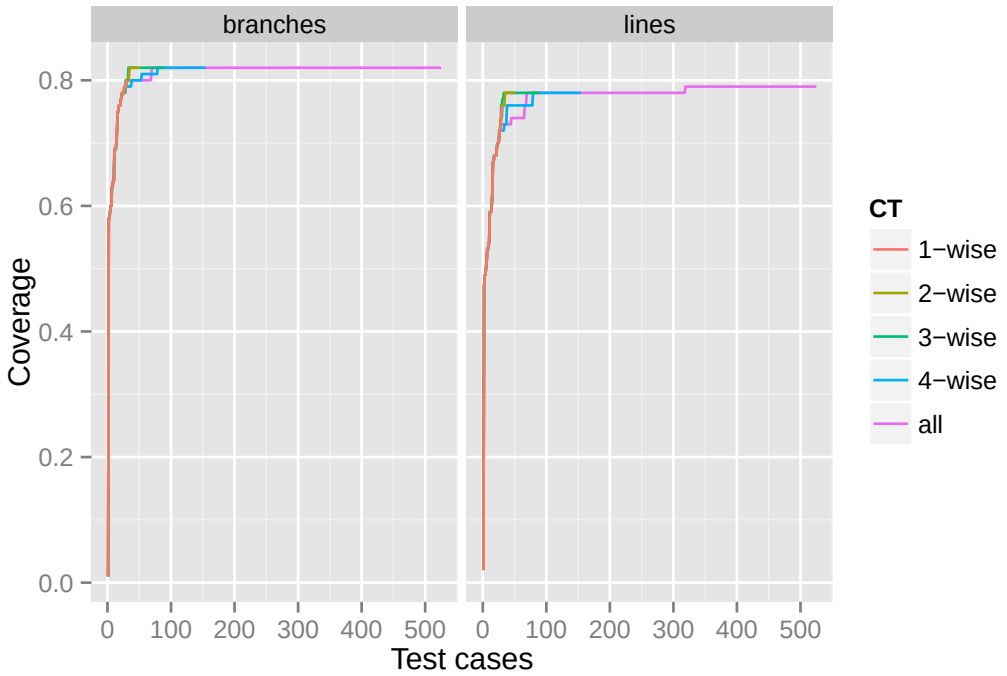

図 A.12　Lines and branches coverage. (GNU flex, v5.)

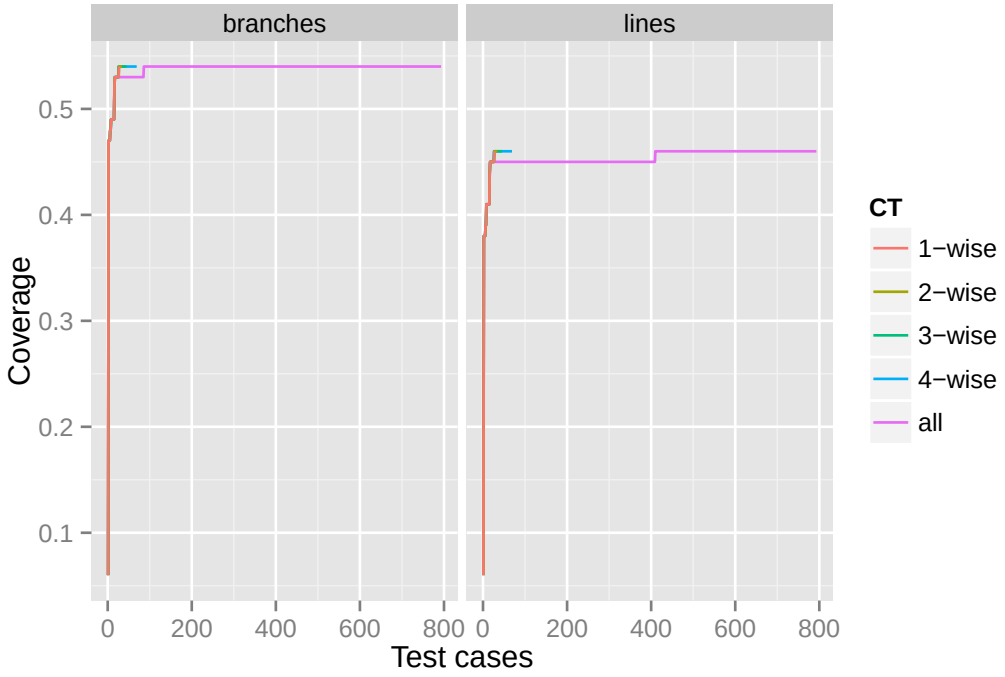

図 A.13　Lines and branches coverage. (GNU make, v1.)

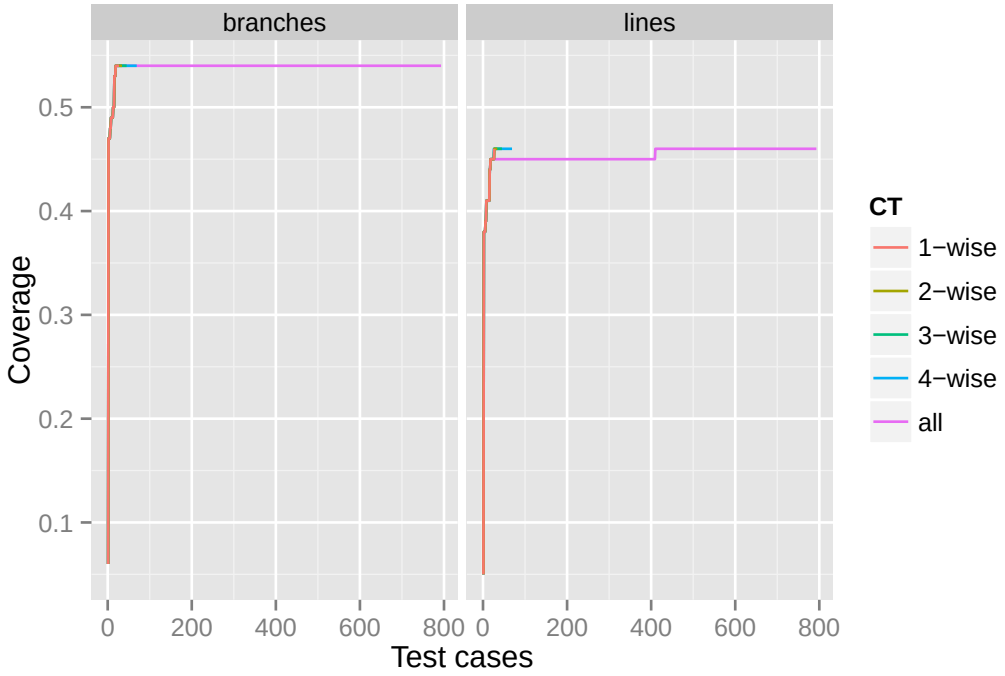

図 A.14　Lines and branches coverage. (GNU make, v2.)

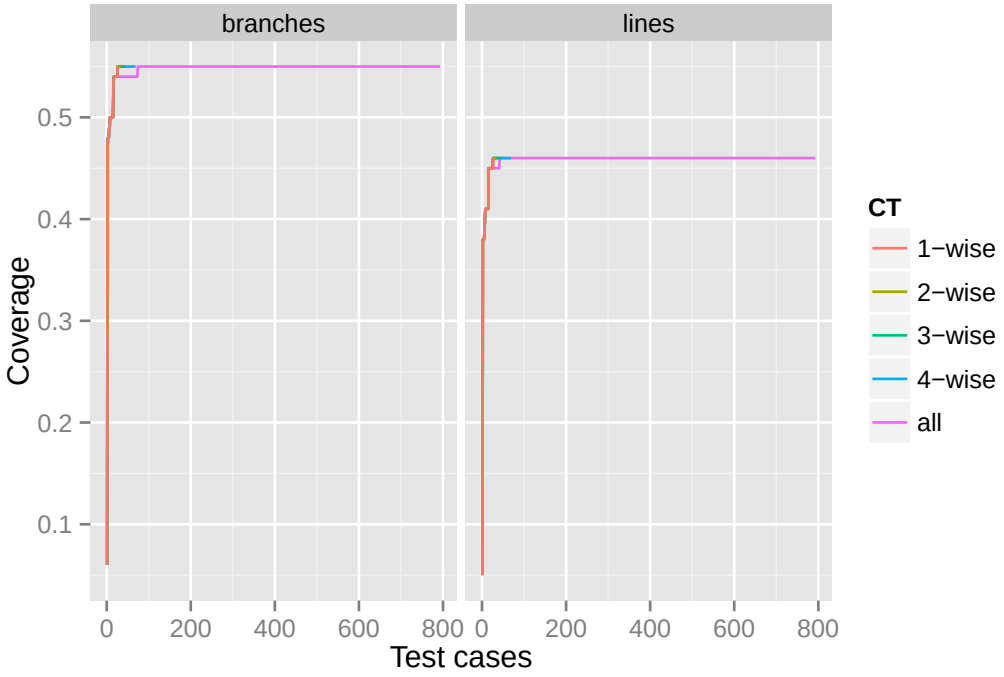

図 A.15　Lines and branches coverage. (GNU make, v3.)

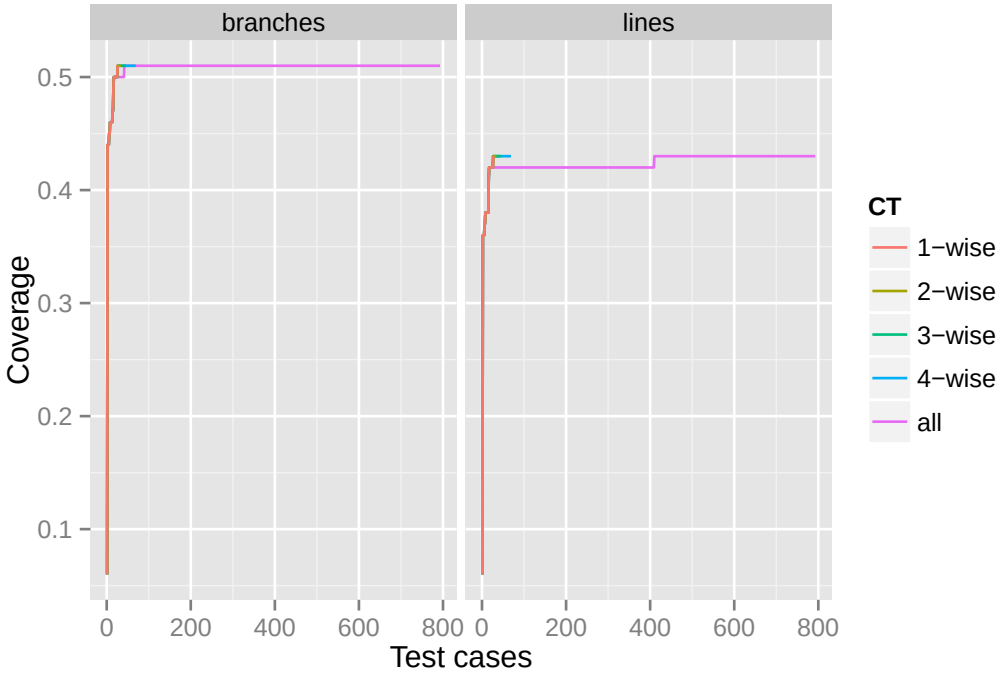

図 A.16　Lines and branches coverage. (GNU make, v4.)