Concolic Testingを活用した実装ベースの回帰テスト ～人手によるテストケース設計の全廃～

松尾谷徹
デバッグ工学研究所
matsuodani@debugeng.com

増田聡
日本IBM
SMASUDA@jp.ibm.com

湯本剛
日本HP
tsuyoshi.yumoto@hp.com

植月啓次
フェリカネットワークス
Keiji.Uetsuki @ FeliCaNetworks.co.jp

津田 和彦
筑波大学大学院
tsuda@gssm.otsuka.tsukuba.ac.jp

要旨

ソフトウェアテストにおける回帰テストは，主に以前のリリースとの互換性を確認するために行われる．当該ソフトウェアが良く使われ利用価値が高まるほど，頻繁に行われる傾向がある．テストを行う側は，派生開発の頻度と規模が増えるほど回帰テストの困難さが増加し，それに比例してコストと期間が増加する問題を抱えている．

この問題の原因を既存の回帰テストが仕様ベースのテストを基にしていることにあると考え，ツールを用いた実装ベースのテスト方法を提案する．具体的な実現方法として *Concolic Testing* を利用し，回帰テストにおいて人手によるテストケース設計を全廃する実験例を示す．

1. はじめに

ソフトウェア開発コストの中で，テストに係るコストが大きな割合を占めている．その中でも，既存のソフトウェアに変更を加えた時，意図しない影響が無いことを確認する回帰テストは，我が国のソフトウェア開発の多くが派生開発であることから，テストの中で大きな割合を占めている [1].

回帰テストとは，テストの団体 International Software Testing Qualifications (ISTQB) による定義では「変更により，ソフトウェアの未変更部分に欠陥が新たに入り込んだり，発現したりしないことを確認するため，変更実施後，すでにテスト済みのプログラムに対して実行するテスト」とされている [2]．回帰テストは，変更行為に対して，想定される機能の変化を確認する「変更に伴う確認テスト」とは区別されている．

回帰テストの問題は，テスト対象範囲が未変更の部分であり，変更量とは無関係に広いため，テストの漏れを少なくするには膨大な量のテストが必要となり，人手によるテストでは対応できない．しかし，自動化の現実的な方法が知られていない課題である．自動化が難しい問題は，テストで発見すべき対象が，新たな欠陥が入っていないこと，潜在する欠陥が発現しないことであり，テスト結果の判定条件が曖昧なことにある．

我々の研究は，回帰テストの目的を未変更部分の互換性を確認するテストと捕え，その自動化を進めることにより，回帰テストの問題を解決する．提案する方式は，実装ベースのテストの考え方を用いて，人手によるテストケース作成を不要とし，それまでのテストケース蓄積にも依存しない，回帰テスト自動化の方式を提案する．この方式は，近年，研究が進んでいる静的実行 (Symbolic execution) や Concolic testing におけるツールを活用することにより実現している [3, 4, 5].

本稿では，回帰テストをプログラムの二つの世代（Rm と Rm+1) における互換性の確認と考える．互換性の確認とは，プログラムの挙動を入力と出力の関係で捕え，同じ入力に対して同じ結果を出力することを確認する．変更とは，バグの修正や機能の追加・変更・削除などを

意味し，その変更の想定される影響を確認するテストは，回帰テストとは別のテスト（変更に伴う確認テスト）とし，回帰テストには含めない．また，Rm に潜在する欠陥があったとして，回帰テストは，その潜在する欠陥を見つけるテストではなく，発現しないことを確認する．発現しないとは，二つの世代において同じ入力に対し同じ値を返すことと考える．

一般的に行われているテストは，仕様を基にプログラムの正しい動作に対応する，テスト入力値と期待結果のセットを人手によって求める．このセットをテストケースと呼んでいる [2]．回帰テストの自動化は，テスト入力値と対応する期待結果を人手に頼らずに得ることである．ここでは，プログラムが持つ制御パスを網羅することができる入力のセットを自動的に見つける手段と，そのテスト入力値に対応する版 Rm における出力を自動的に取得し，回帰テストの期待結果として用いる手段によって，回帰テストの自動化を行う．

本論文 2 章で仕様ベースの回帰テストが持つ原理的な問題を明らかにし，提案する方式と比較する．本論文 3 章で実装ベースの回帰テストを実現する方法について提案し，本論文 4 章で実験的に実装ベースの回帰テストを評価する．

2. 回帰テストの背景技術と課題

現在の主流である仕様ベースの回帰テストの問題について考える．次に，理想的なモデルベース開発によって解決できる部分を示し，提案する実装ベースによる方式を示す．

2.1. テストに関する定義と問題

回帰テストの課題について考える前に，ソフトウェアテストにおける定義と問題について示す．

テストの基本課題は，漏れの無い網羅性の高いテスト入力値の生成と，その入力に対する正しい期待結果の生成である．先ず，テスト入力値の網羅性について考える．

テストの性能の一つは，テストの網羅性である．回帰テストの場合，未変更部分に対する網羅が求められている．未変更部分とは，変更部分の補集合であるが，何が全体なのかを定義する必要がある．ここでは，全体を回帰テスト対象のプログラム（ソースコード）とする．

回帰テストの網羅性を対象とするプログラムに対するテストの網羅性と定義し，プログラムが持つ制御構造の網羅尺度を使う [2]．プログラムの網羅尺度研究は，ほぼ完成しており，網羅基準の順位についても定義がある．データフローを基準にするものと，制御フローを基準にするものがあり，制御フローの順位は，もっとも低い基準が実行文網羅 (SC: Statement coverage）や条件網羅 (BC: Branch condition coverage) であり，最も高い基準は分岐条件組合せ網羅（BCC: Branch condition combination coverage）であり，次が MC/DC(Modified condition/decision coverage) とされている [6, 7]．ここでは，テストで設定可能な変数群に着目した BCC を可達パス網羅と定義し回帰テストにおける最も高い網羅基準と考える．設定可能とは，プログラムでは関知出来ないマシン割込みなどの変数を除くことを意味する．

テストの性能を決めるもう一つは，期待結果の正しさである．膨大な量のテスト入力に対して期待結果を求める問題は，オラクル問題（Oracle Problem）と呼ばれ，テスト入力値の網羅性と共にテストの大きな課題である．関数レベルの単体テストなど人手により仕様から期待結果を求めることが容易な場合では問題にならないが，複雑なプログラムやシステムに対して，変数間の組合せを含む自動テストを行う場合に問題となる [8]．

テストの問題をまとめると，

1). テスト入力値の生成問題 : 網羅性の高いテスト入力値のセットを合理的に生成する方法を見つける．

2). 期待結果の生成問題 : オラクル問題と呼ばれ，変数間の組合せを含む膨大な期待結果を正しく生成する方法を見つける．

この二つの問題が，回帰テスト問題の背景にある．

2.2. 仕様ベースのテスト

仕様ベースのテストとは，仕様を基に実装されたプログラムをテスト方法である．仕様とは，テスト対象となるプログラムが達成すべき機能とその入力や事前条件について定義したものを意味する．仕様ベースのテストは，二つの活動，すなわち，テスト設計とテスト実行に分けて行われる．テスト設計とは，仕様に基づき，テストケース（テスト入力値と期待結果のペア）を設計とする活動である．テスト実行とは，テスト入力値を基にプログラムを動作させ，その出力結果を求め，期待結果と比較し，良否を判定する活動である．

次に，仕様ベースの回帰テストを考える．単純化してプログラムの二つの世代，版 Rm と版 $Rm+1$ だけで考える．版 Rm の仕様を S^{Rm}，版 $Rm+1$ を S^{Rm+1}，版 Rm から版 $Rm+1$ に加えた変更仕様を $S^{\triangle m}$ とし，実装されたプログラムを P^{Rm}，P^{Rm+1} とする．

回帰テストの前に行う，変更に伴う確認テストは，変更仕様 $S^{\triangle m}$ が P^{Rm+1} に正しく実装されたことをテストする．そのためのテストケース $TC^{S^{\triangle m}}$ を作成し，P^{Rm+1} をテストする．

その後，仕様 S^{Rm} において変更仕様 $S^{\triangle m}$ により影響を受けなかった部分，$S^{Rm} - S^{\triangle m}$ に対して回帰テスト行う．記号で影響を受けなかった部分，$S^{Rm} - S^{\triangle m}$ を表すのは簡単だが，自然言語で書かれた実際の仕様書において，この操作を行うことは非常に困難である．

仮に回帰テストの範囲を定義した仕様 $S^{Rm} - S^{\triangle m}$ が作れたとしても，この仕様から，テストケースを人手で作成することは，単体テストなどを除いて困難である．そうすると，S^{Rm} に対応するテストケース TC^{Rm} が既存であることが必要になる．

仕様ベースの回帰テストは，ベースとなる版の仕様とその仕様に対応するテストケースがしっかり維持されていることが必要である．実態として，これは難しい要件であり，変更仕様 $S^{\triangle m}$ から実装上の変更箇所を特定する活動ですら仕様をベースに行うことができず，ソースコードを調べたり，派生開発のための特別な技術が開発され活用されている [9].

2.3. モデルベースによる対策

仕様ベースのテストの問題は，自然言語による仕様記述では，厳密な仕様内容を変更とマージして保持できないことが原因と考えられる．前述の問題を解決する方法として，自然言語による仕様に代わり，定義した挙動をシミュレーションなどで返すモデルを作る方法が考えられる．モデルを作る方法は幾つかあり，システムの物理的な挙動を記述しシミュレーションを行う「MATLAB/Simulink」などを使ったモデルベース開発や，VDM++を使って仕様を詳細に記述し疑似実行する形式手法などである．究極のモデルとして，一部ではプロットタイプ用のソースコードまで生成することができる．

モデルベースのテストは，モデル自身の検証とは別に，プログラムの挙動がモデルの定義と合致していることを

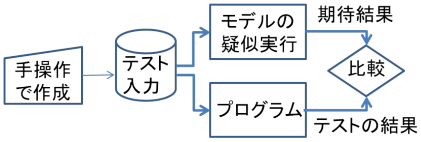

図 1. モデルを使ったテスト

確かめる．モデルが疑似実行できるなら，図 1 に示すように，プログラムとモデルに同じテスト入力を与え，結果を比較することと等価である．モデルベースによるテストの特徴は，期待結果を新たに設計する必要が無い．モデルを疑似実行することによって得られる．テストの問題の一つである，オラクル問題を解決できる．

もう一つの課題である入力値の生成問題については，幾つかの研究が行われているが，実用化には達していない [10, 11]．現時点においては，テスト入力値を作成し，網羅性を評価する必要がある．

回帰テストについて考える．版 Rm と版 $Rm+1$ に対応するモデルを M^{Rm}，M^{Rm+1} とし，プログラムは P^{Rm}，P^{Rm+1} とする．テストは，モデル M^{Rm+1} と P^{Rm+1} の挙動を比較する．モデルが持つ挙動のすべてを網羅する，テスト入力値については，モデルの記述から作成する必要がある．その方法やコストについては，ここでは検討しないが，容易では無い．

モデルベースの変更においては，変更に伴う確認テストと回帰テストを分離できない．版 Rm におけるモデル M^{Rm} に，変更を加えたモデル P^{Rm+1} が，モデル上，未改造部分に欠陥が入り込んだり発現しないないかは，モデル自身の回帰検証の問題であり，プログラムに対する回帰テストでは検出できない．

モデルベースにおける回帰テストをまとめると．モデルが正しく作られていれば，オラクル問題は解決できる．テスト入力の生成については，実用的なツールは出現していない．回帰テストの主役は，モデルの変更におけるモデル自身の検証に依存する．現実的な問題として，モデル化されていない既存のプログラムから，疑似実行可能なモデルを作るアプローチは非常に難しい．

2.4. 実装ベースのテスト

実装ベースのテストとは，実装物であるプログラムを静的，あるいは動的に解析し，テストを行う方法である．テストと言っても，本質的に仕様との合致を確認する期

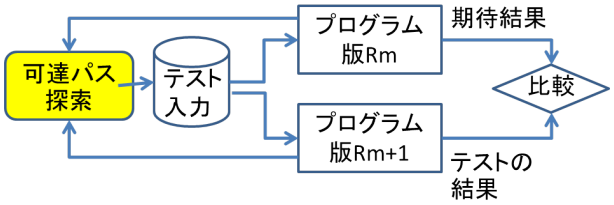

図 2. 実装ベースのテスト構成概要

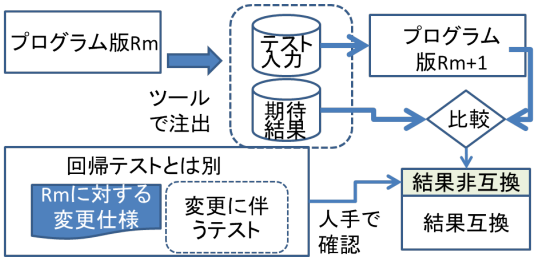

図 3. 実装ベースの回帰テスト構成概要

待結果を実装から作ることは不可能である．合否を判断するためには，何らかの期待結果を決める事後条件を指示する必要がある．例えば，メモリーリークやオーバーフローが生じない，セキュリティホールが無いことを確認するために活用されている．この種のテストは実装ベースのテストでは実現できない．

仕様との比較を行う先行研究の一つに，プログラムが持つ制御構造をツールを用いて探索し，プログラムに実装されている論理を決定表で表し，仕様から求めた決定表と比較する方式が提案されている [3, 4, 5]．この方法は，マシンを使ったテスト実行を省略できることが大きな特徴である．

プログラムが持つ制御構造を探索する方法は，対象とする変数を指定し，その変数が関与するすべての条件判断に対して，チェックポイントを埋め込み，すべてのチェックポイントを通過する変数値を探索する．チェックポイントを埋め込む前に，ソースコードは抽象構文木に展開している．探索の方法は，制約ソルバーを使って分岐条件を探り，解を求める．制約ソルバーで解けない場合は，乱数を使うなどヒューリスティックな手段を併用する．2008 年頃から，実用的なツールが Java,C,C++などの言語に対して公開されている [12, 13, 14, 15, 16, 17].

この方法によって可達パスが求まる．このツールをここでは，可達パスの探索ツールと呼ぶ．可達パスを求めることは，同時に可達パスを駆動する変数（複数）の入力値が決まるので，この値をテスト入力値として用いる．4 章の実験では，Concolice test の CREST を可達パスの探索ツールとして用いた．

実装ベースの回帰テストについて考える．テスト入力値については，プログラムから探索して求める．オラクル問題の解としては，前世代の版を利用することにより実現する．そのその構成概要を図 2 に示す．

実際の回帰テストでは，版 Rm と版 $Rm+1$ の挙動には差が有り，変更 $S^{\triangle m}$ が加えられているため，単純に図 2 に示すような比較では，回帰テストの目的を達成できない．変更が行われてい部分に対するテスト入力値に限定する必要があるが，これを求めることができない．

理由は，P^{Rm} は互換を確認するベースとして，正しいと仮定できる．しかし，変更が行われた結果である P^{Rm+1} には，変更を加えたため欠陥が含まれている可能性がある．その欠陥には，変更仕様を逸脱したものも考えられるため，仕様から範囲を決めることができない．そこで，図 3 に示すように，P^{Rm} から抽出した TC^{Rm} を用いて P^{Rm+1} をテストし，得られたテスト結果から，結果として非互換の部分（結果非互換）と互換の部分（結果互換）を識別する．

得られた結果非互換と，変更仕様 $S^{\triangle m}$ の内容，あるいは，それに基づく変更に伴う確認テストのテストケースから，結果非互換の項目について消込を行い，残った項目が有れば，デグレードの可能性がある．結果互換に欠陥があるとすれば，変更仕様 $S^{\triangle m}$ で定義された変更の漏れであるが，回帰テストの目的を超えるので議論しない．結果非互換の項目について消込は，人手による活動である．

仕様ベースのテストが，正しい結果を想定してテストケースを漏れなく設計するのに対し，実装ベースのテストは，プログラムが持つ非互換の部分を抽出し，それが仕様を満たしているか否かは，テスト結果を基に後から解析する．詳細については 3 章で示す．

2.5. 実装ベースを支える可達パスの探索ツール

実装ベースのテストでは，実装が持つすべての入力域を探索するツールが必要になる．プログラムが持つ可達パスを探索する研究は，モデル検査から発達した Symbolic Execution として活発に研究が進んでいる [18, 19]．近

年，抽象構文木で中間言語生成するコンパイラーが出現し，疑似実行の環境が開発されたこと，及び，制約ソルバーの発達から実用的な可達パスの探索ツールが開発され，普及が進んでいる．

ここでは，Symbolic Execution の応用である Concolic Testing を使用する．Concolic Testing とは，Sen, Koushik と Agha, Gu が 2006 年に発表した論文の中で定義した Concreat と Symbolic を合わせた造語である [16]．Concolic testing のためのオープン系可達パスの探索ツールとしては，CREST が公開されている [20]．ここでは，この可達パスの探索ツールを使た実装ベースの回帰テストについて示す．

CREST の探索能力については，多くの応用研究が行われ確認されている．代表的なものでは，Linux の core util を対象に行われた検証で，人手による十数年間のテストで発見できなかった欠陥を見つけている．grep など再帰呼び出しに対しても効率的にパスを探索することができる．規模としては 15 万ステップのプログラムに対する例が報告されている．

全ての場合において，制約ソルバーで解析的にパスを解けないため，ヒューリスティックな手段を併用する．よって，探索オプションなどを調整する必要がある．それでも完全に解けない場合もあるが，人手による探索と比べ高い網羅性を達成している．

3. 実装ベースの回帰テスト

実装ベースの回帰テストにおいて必要なものは，変更前と後のソースコードと，変更に関する仕様（変更仕様）である．ここでは変更前のプログラムを P^{Rm}，変更後を P^{Rm+1} と表示し，それがソースコードなのか実行形式なのかは文脈で表す．変更に関する仕様は，変更に伴う確認テストを設計できる詳しさで記述されているとする．変更に伴う確認テストとは，変更によって意図した変化が正しく動作していることを確認できることである．変更に伴う確認テストは，回帰テストの前に実行されてれいることを前提とする．

このような条件下で，実装ベースの回帰テストとして二つの方式「可達パス網羅の回帰テスト」と「同値組合せ網羅の回帰テスト」を提案する．

3.1. 可達パス網羅の回帰テスト

可達パス網羅の回帰テストは，回帰テストのテスト入力値の網羅基準として，可達パス網羅を用いる．

図 3 に示した実装ベースの回帰テスト構成概要を基に，その手順について以下に説明する．

(A1) 可達パスの探索 可達パスの探索ツールを用いて Rm のソースコードに対し，可達パスに関する情報を得る．

ここでの入力は，Rm のソースコード，出力は，テストケース TC^{Rm} である．

(A2) クロステストの実行 通常のテストハーネスを用い，Rm から得られた TC^{Rm} を用いて P^{Rm+1} をテストする．このクロステストは Rm から見た $Rm+1$ との互換/非互換を検出する．互換であるか非互換であるかは，Rm から得られた期待結果と比較することにより判断している．

ここでの入力は，テストケース TC^{Rm} と TC^{Rm+1}，出力は各テストケースの判定情報，すなわち合格＝互換，不合格＝非互換の一覧である．

(A3) 解析 この部分は，人手によって行う．ここでの入力は (A2) の出力であるテスト実行の結果と変更仕様である．出力は，非互換部分が変更仕様に基づくものか否かの判断である．

人手による判断の難易度は，主に変更の量に依存する．(A1)(A2) は，自動化出来ることから，変更を小さな単位で区切り，繰り返し確認することにより，判断の難易度を下げることができる．

以上に示した，可達パス網羅の回帰テストの特徴は，人手によるテストケース作成を一切行わない．可達パスの探索ツールの設定やテスト環境が出来上がれば，テストケースを人手により作成する作業から解放される．可達パスを探索するスピードと網羅性は，圧倒的に人手による作業能力を越えている．

解析者は，デバッグを除けば，プログラムのロジックを見ることなく，プログラムの入出力の関係から変更仕様に合致しているか否かを確認できる．但し，変更仕様により変更されたロジックについては解析し，非互換の精査を行う必要であるが，これは，仕様ベースのテストでも同様である．

この方式で回帰テストが出来るか否かは，入力変数の値に依存しない出力を生成するプログラムを除けば，用いる可達パスの探索ツールでテスト入力値を漏れなく検出できるか否かにかかっている．入力変数の値に依存しない出力とは，乱数生成に基づくロジックなどを意味する．また，用いる可達パスの探索ツールから，ソースコードを駆動するため，変数を指定して接続するためのインタフェースを作成する必要がある．

可達パスの探索ツールは，取り扱える変数（整数，ビット列，浮動小数点，など）と組込まれたソルバーによってその能力が決まる．解析的に解けない場合は，ヒューリスティックな探索を行うため，プログラムの規模が大きい場合や，再帰呼び出しが多いと，解析に時間がかかる．

ここでの提案は，実装ベースの回帰テストに関する方式であり，可達パスの探索ツールの能力そのものについては対象としていない．

3.2. 同値組合せ網羅の回帰テスト

同値とは，テスト技法で用いられる用語であり，テストの入力となる変数の変数域に着目し同値分割を行い得る．変数域をテストのために同値クラスに分割し，その中で選んだある値である．変数は複数の同値を持つ．テストの規格 (ISO/IEC 29119-4) では，変数と選んだテスト入力値を P-V pair(Parameter - Value) と呼び，テスト設計の基本的な成果物である．Value に同値を用いることにより，少ないテスト入力で効果的な網羅基準を達成することができる．

同値組合せ網羅の回帰テスト構成を図 4 に示す．可達パス網羅の回帰テストとの違いは，ツールが生成するテスト入力値ではなく，一度，変数の持つ同値を求め，その後，変数の同値を組合せてテスト入力値とする．組合せを指定することにより，テストの網羅性基準を調整することができるのが特徴である．その手順を次に示す．

(B1) パスの探索と変数の同値抽出 ツールを使う点は，可達パス網羅の回帰テストにおける (A1) と同じであるが，出力は変数に対する同値のリストである．

(B2) 同値の組合せからテスト入力値作成 変数間の組合せを指定してテスト入力値を作成する．変数は同値の数だけバリエーションを持つので，すべての組合せだと膨大な数になる．回帰テストのリソースや重要性から，組合せを指定する．例えば All-pair 法などを用いて網羅基準を緩和することが考えられるが，マシンパワーがあれば，その必要は無い．

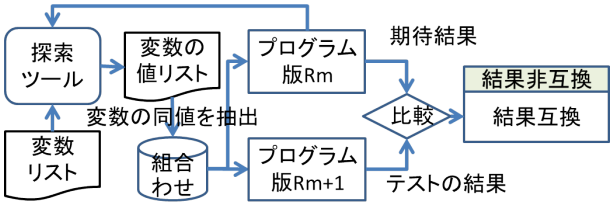

図 4. 同値組合せ網羅の回帰テスト構成

(B3) 期待結果の取得 (B2) で作成したテスト入力を使って期待結果を得る．

(B4) テストの実行 (B2) で作成したテスト入力を変更後のプログラムに与えテストを行い結果を得る．結果は (B3) で得た期待結果と比較し，非互換か互換に分割する．

(B5) 解析 この活動は，可達パス網羅の回帰テストと同じである．

同値組合せ網羅の回帰テストも，可達パス網羅の回帰テストと同様な特徴を持っている．異なるのは，網羅の基準を調整することができる点である．

4. 回帰テストの実験

先に示した実装ベースの回帰テストについて実験し評価を行う．実験に用いたプログラムは，ロジックの設計やテストを行うための演習問題を使った．簡単な問題ではなく，人手で解くには複雑な組合せが生ずる問題である．

4.1. 用いた例題：割引問題の仕様

このプログラムは，ある施設の入場割引に関するもで，入力として割引の条件が与えられており，それぞれの条件に対して，正規料金に対する割引後の値を%で返す．その仕様は次の通り．

1). 3 歳以下の場合には無料とする．(正規の 0%)
2). 水曜日なら，正規の 90%とする．
3). 60 歳以上なら，正規の 60%とする．
4). 女性の場合，50 歳以上なら正規の 65%とする．

5). 施設の記念日なら，正規の 80%とする．
6). 施設の地域住民なら，正規の 50%とする．
7). 15 時以降の入場なら正規の 70%とする．
8). 12 歳以下なら正規の 40%とする．
9). 但し，条件が重複する場合には，割引の大きい方を選択する．

この仕様を満たすプログラム waribiki_Rm.c があり，これを変更前のプログラムとする．このプログラムに次の仕様を追加したものを waribiki_Rm+1.c とする．
10). 冬季（1 月，2 月）なら，67%とする．

4.2. 回帰テストの条件

4.1 で定義した仕様から作られたプログラムに対して回帰テストを試行する．テスト側の条件としては以下を設定する．

- コード waribiki_Rm.c と同_Rm+1.c は与えられる．解析してもよい．
- コード waribiki_Rm.c に対するテストケースは引き継がれていない．
- 変更仕様として「冬季（1 月，2 月）なら，67%とする.」が waribiki_Rm+1.c に加えられたことを知っている．
- 割引全体に関する仕様（1. から 9.) は知らない．

4.3. 仕様ベースの回帰テスト

比較のために，仕様ベースの回帰テストについては机上考察を行う．回帰テストの条件で，元の仕様（1. から 9.) は解らないが，変更仕様は入手しているので，そこから変更に伴う確認テストを作る．そのテストケースとしては，次のものが考えられる．

1). 2 月なら 67%
2). 1 月なら 67%
3). 3 月から 12 月は 67%にならない，他は今までと同じ

他の影響範囲を推測するには，ソースコードを解析する必要があるが，ここでは考えない．もし，ソースコードを正確に理解すれば，この例題の仕様は，9 番目の仕様に「条件が重複する場合には，割引の大きい方を選択する」とあることから，10 番目に追加された仕様の意味は,「12 歳以下では無くて，地域住民では無くて，50 歳以上の女性ではなくて，60 歳以上でもない」場合において，冬季（1 月，2 月）なら，67%とする処理が追加されることである．同様な依存関係から，他の割引条件にも影響が生じる．仕様を正しく伝えても，この影響に気付くことは難しく，仕様を基に漏れの無い回帰テストを行うことは，誰にでも簡単にできることではない．

4.4. 可達パス網羅の回帰テスト

waribiki_Rm.c から waribiki_Rm+1.c へ変更が行われたとして,「可達パス網羅の回帰テスト」を実験する．ここでは次の手順で回帰テストを進める．

1). テストドライバの開発
2). Concolic Testing によるテストケース探索
3). クロステスト
4). 結果の差分から回帰テストの判定

(1) テストドライバの開発 二つのテストドライバを準備する．一つは Concolic Testing により可達パスを探索し，テスト入力値を求める．ここで用いたツールは，可達パスに対して実行結果も出力することができる．ここでは Concolic ドライバと呼ぶ．

もう一つは，テスト入力値をファイルから読込，結果を出力する普通のテストドライバである．Concolic ドライバは，簡単で次のようなコードである．

```
//Concolic のためのドライバTesting
#include <crest.h> /* Concolic header */
#include <stdio.h>
#include <stdlib.h>
int main(){
int sex,age,dayofweek,citizen,month,
    memorialday,intime;
  CREST_int(sex); /* variable to crest */
  CREST_int(age);
  CREST_int(dayofweek);
  CREST_int(citizen);
  CREST_int(month);
  CREST_int(memorialday);
  CREST_int(intime);
  int t_no=1;
/*（functionwaribiki)とテストケースの出力call*/
printf("%d %d %d %d %d %d %d \t\t %d\n",
sex,age,dayofweek,citizen,month,memorialday,
intime,waribiki(sex,age,dayofweek,
citizen,month,memorialday,intime));
```

(2)Concolic Testing によるテストケース探索
CREST を使って，P^{Rm} と P^{Rm+1} の探索を行う．P^{Rm+1} を探索するのは，変更が削除の場合を $P^{Rm+1} \rightarrow P^{Rm}$ として実験するためである．探索オプションは-dfs: BoundedDepthFirstSearch を用いた．ツール CREST は，整数の場合 long で膨大な範囲を探索するので，事前条件として範囲を限定する処理を Concolic ドライバに入れてある．探索されたテストケースを表 1 に示す．可達パス網羅の計測方法が無いので gcove を使った分岐網羅を参考に示している．

表 1. Concolic ドライバの結果

版	ケース数	実行時間	分岐網羅
Rm	48	0.274s	100%
$Rm+1$	72	0.522s	100%

表 2. Rm の結果から $Rm+1$ をテストした差異

性	歳	曜日	住居	月	記念	入場	改造後	改造前
1	13	1	0	1	0	10	67	100
1	13	1	0	1	0	16	67	70
1	13	1	0	1	1	10	67	80
1	13	1	0	1	1	16	67	70
1	13	1	0	2	0	10	67	100
1	13	1	0	2	0	16	67	70
1	13	1	0	2	1	10	67	80
1	13	1	0	2	1	16	67	70
1	13	6	1	1	0	10	67	70
1	13	6	1	1	0	16	67	70
1	13	6	1	2	0	10	67	70
1	13	6	1	2	0	16	67	70
2	13	1	0	1	0	10	67	100
2	13	1	0	1	0	16	67	70
2	13	1	0	1	1	10	67	80
2	13	1	0	1	1	16	67	70
2	13	1	0	2	0	10	67	100
2	13	1	0	2	0	16	67	70
2	13	1	0	2	1	10	67	80
2	13	1	0	2	1	16	67	70
2	13	6	1	1	0	10	67	70
2	13	6	1	1	0	16	67	70
2	13	6	1	2	0	10	67	70
2	13	6	1	2	0	16	67	70

(3) クロステスト P^{Rm} から得られたテストケース 48 項目を P^{Rm+1} に与え，Rm の結果と異なるものを比較したところ，48 ケース中 18 ケースで結果が一致しない非互換が見つかった．非互換である 18 のテストケース（入力と結果）を表 2 に示す．$Rm+1$ から得られたテストケース 72 項目を Rm に与え，$Rm+1$ の結果と異なるものを比較したところ，72 ケース中 24 ケースが異なっていた．

(4) 結果の差分から回帰テストの判定 クロステストの結果から，異なっていたものの共通点は，$Rm+1$ 側の結果である割引がすべて 67%であった．これは，変更仕様で追加した割引と一致している．他の割引で非互換のものは見つからなかった．可達パスによる回帰テストの網羅は，変数のすべての組合せを網羅しているので，挙動上の違いをすべて見つけている

この程度の差であれば，人がチェックすることも出来るが，差の数が増加すると比較が困難になる．対策としては，実装ベースのテストの先行研究で提案されている決定表で表現する．その例を表 3 に示す．テスト結果と比較する仕様側のロジックも決定表で表現できれば，比較が簡単である．表 4 にその例を示す．仕様で用いる決定表はビジネスルールとも呼ばれている．

なお，逆の変更として P^{Rm+1} のテストケースで PP^{Rm} をテストし他場合，18 件の非互換を検出し，変更仕様後と前の差を検出できた．

表 3. 決定表によるテストケースの表現

	テストケース	1	2	3	...	9	10
原因	性別 {1,2}	1	1	1		2	2
	齢 {0,4,13,50,60}	0	4	4		13	13
	曜日 {1,3,6,7}	1	1	1		3	1
	住居 {0,1}	0	1	0		0	0
	月 {1,2,3}	1	1	1		1	1
	記念日 {0,1}	0	1	0		0	0
	入場時 {10,16}	10	10	10		10	10
結果	0	X					
	30		X				
	40			X			
	80						
	90					X	
	100						X

表 4. 仕様の決定表（ビジネスルール）

	規則	1	2	3	4	...	9	10	11	12
条件	性別					省略				ELSE
	年齢	3 以下		12 以下						
	曜日				平日				水曜	
	住居		真		住民		真			
	月									
	記念		真					真		
	入場時間									
動作	割引後%	0	30	40	50		70	80	90	100

以上の実験の結果，提案した「可達パス網羅の回帰テスト」を使って，この例題について，人手を介さず互換/非互換となるプログラムに対する入力値と結果を洗い出せた．全てのプログラムに対して有効か否かは，この実験だけでは解らないが，この方式で回帰テストが実現できることは明らかになった．

4.5. 同値組合せ網羅の回帰テスト

次に，同値の組合せからテスト入力値を作成して回帰テストを行う「同値組合せ網羅の回帰テスト」について実験する．用いるのは，先の実験と同じ waribiki_Rm.c から waribiki_Rm+1.c である．手順的には，Concolic Testing で得られたテストケースから，(1) 同値を抽出し，(2) 同値組合せを作成し，(3) 期待結果の取得を行う．その後は，可達パス網羅と同様である．

(1) 同値の抽出 4.4 の (2) の結果から入力の同値を抽出する．その方法は，可達パスの探索ツールが生成した各入力変数のユニークな値である．項目別に sort し uniq コマンドで簡単に入手できる．抽出され各変数の同値の数は，Rm に対し性別:2, 年齢:5，曜日:4，住居:2，月:1，記念:2，入場時間:2 であった．変更後の $Rm+1$ については，変数月に対して同値が 1 から 3 に増加する．

(2) 同値の組合せ 同値から組合せを作成する．人手で組合せるのは大変なので all-pair を生成するツールを使って生成した．2 変数間の全同値の組合せだけでなく，全変数間の全同値の組合せまで生成できる．ここでは，全変数間の全同値の組合せを行い Rm において 480，変更後の $Rm+1$ において 1440 であった．

(3) 期待結果の取得 (2) で得られた同値組合せを入力として，Rm と $Rm+1$ を動作させて得る．実行時間は 0.003 秒と 0.005 秒であり，表 1 に示した concolic Testing と比べれば，テスト数が 10 倍であるが，実行時間は 100 倍ほど早い．

(4) クロステストの実行と解析 クロスでテストを実行した結果，Rm から得られたテストケース 480 項目を $Rm+1$ に与え，Rm の結果と異なるものを比較したところ，86 ケースが異なっていた．$Rm+1$ から得られたテストケース 1441 項目を Rm に与え，$Rm+1$ の結果と異なるものを比較したところ，165 ケースが異なっていた．可達パス網羅と同様，差が生じたテストケースの $Rm+1$ 側の出力結果は，67%であり変更仕様を満たしていた．

同値組合せ網羅の実験において，Rm と $Rm+1$ のクロステストにより見つかった非互換は，86 件と 165 件であった．この非互換としたテストケースで共通のものを除外した 42 件がユニークな非互換であった．この 42 件は，すべて変更仕様を満たしていた．

変更仕様から，机上でこの 42 件を予見するのは難しいと思われる．変更を論理的に考え，プログラマーが機能やロジックを変更することと，その結果として，利用者から観た入力と結果の挙動の違いを数え上げることとは，本質的に違う活動である．利用者には，変更が正しく行われたか否かより，生ずる変化に対して受け入れられるか否かが問題となる．実装ベースの回帰テストは，利用者の視点から，何が変化したのかを明らかにすることができる．

4.6. 実験のまとめ

この実験で用いたソースコードを示す．

```
// waribiki_Rm
int waribiki(int sex,int age,int dayofweek,int citizen,
int month,int memorialday,int intime) {
int  discount=100;

if ( age <= 3 ) discount=0;
else if (memorialday == 1 && citizen == 1) discount=30;
else if ( age <= 12 ) discount=40;
else if ( citizen == 1 && (dayofweek >=1 && dayofweek
        <6) ) discount=50;
else if ( age >= 60 ) discount=60;
else if ( sex == 2 && age >= 50 ) discount=65;
//else if ( month ==1 || month ==2 ) discount=67;
// Rm+1 では，この部分を追加
else if ( intime >= 15 ) discount=70;
else if ( memorialday == 1 && (dayofweek ==6 ||
      dayofweek ==7) ) discount=70;
else if ( citizen == 1 ) discount=70;
else if ( memorialday == 1 ) discount=80;
else if ( dayofweek == 3 ) discount=90;
else discount=100;

/* 後処理 */
  return discount;
}
```

仕様上は一つの条件と処理を追加するだけの変更であり，実際のコーディングも else if (month ==1 —— month ==2) discount=67; を一行追加するだけである．しかし，仕様ベースの回帰テストでは十分なテストが出来ないことは明らかである．

一方，実装ベースではテストケースを作成せずに，正確に互換/非互換を検出した．可達パス網羅では，プログラムの制御構造上の特性から，同値組合せ網羅より少ないテストケースとなった．

どちらも自動実行であり，その処理時間も小さいことから優劣は無いと考える．利用者に，挙動上の差を説明する．あるいは，挙動上の差から派生開発の良否を判断するには，同値組合せの方が漏れが少ないと思われる．

この実験では，特別な欠陥を挿入していないが，プログラムに存在するすべての条件分岐を検出するので，定義された変数に対して仕様外の条件分岐があれば検出できる．しかし，定義されない入力変数については検出できない．

5. おわりに

現状の回帰テストの課題に対して，その原因が仕様ベースのテストを使っていることから生じていることを示し，解決案として，実装ベースの回帰テストを提案した．実現する方法として，可達パス網羅と同値組合せ網羅を示し，その方法について実験を行い，実現できることを示した．

参考文献

[1] 深谷直彦，古川善吾，西康晴，他，“特集 ソフトウェアテストの最新動向,” 情報処理，vol.49，no.2，pp.126–173，2008.

[2] JSTQB 技術委員会，“ソフトウェアテスト標準用語集 日本語版,” http://www.jstqb.jp/dl/JSTQB-glossary.V2.3.J02.pdf.

[3] 植月啓次，松尾谷徹，津田和彦，“効率的なテスト設計を実現するデシジョンテーブルの拡張法,” ソフトウェア・シンポジウム 2013 論文・報告，2013.

[4] 植月啓次，“ソフトウェアの実装情報に基づく決定表を活用した論理検証手法,” ソフトウェア・シンポジウム 2013 論文・報告，2013.

[5] K. Uetsuki, T. Matsuodani, and K. Tsuda, “An efficient software testing method by decision table verification,” International Journal of Computer Applications in Technology, vol.46, no.1, pp.54–64, 2013.

[6] ボーリス・バイザー，ソフトウェアテスト技法，日経 BP 出版センター，1994.

[7] J. SC7/WG26, Software and systems engineering — Software testing Part: 4 Test techniques, ISO/IEC/IEEE，1994.

[8] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” IEEE SE, vol.41, no.5, pp.507–525, 2015.

[9] 清水吉男，派生開発を成功させるプロセス改善の技術と極意，技術評論社，2007.

[10] A.S. Santos, “Vdm++ test automation support,” Master’s thesis, Minho University with exchange to Engineering College of Arhus, 2008.

[11] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” Software Testing, Verification and Reliability, vol.22, no.5, pp.297–312, 2012.

[12] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, “Model checking programs,” Automated Software Engineering, vol.10, no.2, pp.203–232, 2003.

[13] G. Stergiopoulos, B. Tsoumas, and D. Gritzalis, “Hunting application-level logical errors,” Engineering Secure Software and Systems, pp.135–142, Springer, 2012.

[14] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” Code Generation and Optimization, 2004. CGO 2004. International Symposium onIEEE, pp.75–86 2004.

[15] C. Cadar, D. Dunbar, and D.R. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.,” OSDI, vol.8, pp.209–224, 2008.

[16] K. Sen and G. Agha, “Cute and jcute: Concolic unit testing and explicit path model-checking tools,” Computer Aided VerificationSpringer, pp.419–423 2006.

[17] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation,” Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineeringIEEE Computer Society, pp.443–446 2008.

[18] J.C. King, “Symbolic execution and program testing,” Communications of the ACM, vol.19, no.7, pp.385–394, 1976.

[19] C. Cadar, P. Godefroid, S. Khurshid, C.S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, “Symbolic execution for software testing in practice: preliminary assessment,” Proceedings of the 33rd International Conference on Software EngineeringACM, pp.1066–1071 2011.

[20] CREST, “Concolic test generation tool for c,” http://jburnim.github.io/crest/.