Vol.2016-SE-192 No.1
Vol.2016-EMB-41 No.1
2016/6/2

SBSTを用いた高カバレッジの結合テスト向けテストデータ自動生成手法

倉林 利行[1,a)]　張 暁晶[1,b)]　丹野 治門[1,c)]

概要：本研究では，テストデータを自然界のアルゴリズムを模倣し進化させていく手法である，遺伝的アルゴリズム（GA）を用いた Search-based software testing(SBST) によって，コードに対する網羅性（カバレッジ）の高い，結合テストにおけるテストデータの自動生成に取り組む．しかし従来の GA を用いた SBST は，主に単体テスト向けに研究がされており，結合テストの場合，現実的な時間内では十分なテストデータが生成できないという問題があった．本研究では，結合テストにおいても現実的な時間内で網羅性の高いテストデータの生成が可能な手法を提案する．また，同条件で生成したテストデータによるコードカバレッジを既存手法と比較した結果，提案手法の優位性を確認できた．

キーワード：ソフトウェアテスト，テストデータ自動生成，結合テスト，探索的テスト，遺伝的アルゴリズム

Code Coverage Oriented Test Data Generation for Integration Testing Using SBST

Toshiyuki Kurabayashi[1,a)]　Zhang Xiaojing[1,b)]　Haruto Tanno[1,c)]

Abstract: This research focuses on code coverage oriented test data generation for integration testing by using SBST with genetic algorithm(GA) which evolves test data by emulating natural world. However, conventional SBST with GA cannot generate enough test data in realistic time for integration testing because it is mainly researched for unit testing. In this research, high code coverage oriented test data generation method for integration testing in realistic time is proposed. We confirm the effectiveness of our approach by comparing it to the existing approach.

Keywords: Software Testing, Test Data Auto-Generation, Integration Testing, Search-Based Software Testing, SBST, Genetic Algorithm

1. 背景と目的

一般市場において，新サービスの提供が迅速化しており，市場投入と改善の繰り返しが短いサイクルで行われている．それに伴い，ソフトウェア開発案件の内訳にも大きな変動が見られる．2000 年から 2013 年にかけて，既存のソフトウェアに対する機能改善やバグ修正を行う保守改造案件の件数が，全案件数の 12%から 68%へと増加し，新しくソフトウェアを製造する新規開発案件の件数よりも上回っている [1]．そのような増加傾向にある保守改造案件において，回帰テストと呼ばれる工程が大きな負担となっている．回帰テストとはソフトウェアにおいて，デグレードと呼ばれる機能劣化（以前存在していた機能が失われていること）の有無を確認するためのテストであり，保守改造案件の稼働の 25%から 100%を占めている [2]．なお，これは結合テスト以降のテストが占める割合であり，単体テストの稼働については実装工程に含まれる．既存の機能に対する再テストである回帰テストは，価値創造の面において非

[1] NTT ソフトウェアイノベーションセンタ
東京都港区港南 2-13-34 NSS-II ビル 6F
[a)] kurabayashi.toshiyuki@lab.ntt.co.jp
[b)] zhang.xiaojing@lab.ntt.co.jp
[c)] tanno.haruto@lab.ntt.co.jp

Vol.2016-SE-192 No.1
Vol.2016-EMB-41 No.1
2016/6/2

常に乏しいにもかかわらず，保守改造案件の稼働の多くを占めていることが大きな問題となっている．

1.1 本研究の目的と技術的目標

本研究は，結合テストレベルの回帰テストにかかる人の稼働を，自動化によって削減することを目的とする．結合テストレベルの回帰テストを自動化する既存方法として，テスト自動実行ツールが入力とするスクリプト（以下テストスクリプト）を作成し，テスト実行を自動で行う方法 [3][4][5] があるが，テストスクリプトには，自動実施したい作業を逐一記述する必要があるため，テストスクリプトの作成に労力がかかってしまう．したがって，既存方法では，抜本的な稼働改善にはならない．そこで，本研究ではテスト対象からテストスクリプトを自動生成することで，従来の回帰テスト自動化の問題点であったテストスクリプトの作成に稼働がかかるという問題を解決し，回帰テストを全自動化することを目指す．具体的には，以下のステップで回帰テストを実施することで，回帰テストの全自動化が実現できると考えている．

Step1：結合テストレベルの網羅的なテストケースを旧ソフトウェアから自動生成する．

Step2：1 で生成したテストケースを新旧２つのソフトウェアに対して実行する．

Step3：2 で得られた結果を自動比較する．

本手法は，回帰テストの特性に着目している．回帰テストは，旧ソフトウェアが有していた機能に対し，デグレードの有無を確認するためのテストであるため，旧ソフトウェアの振る舞いを正解とする．したがって，旧ソフトウェアから網羅的なテストケースを生成することで，正常・準正常系における機能性テストにおいて，人手で作成したテストケースをカバーすることができると考えられる．Step2 と 3 に関しては，例えば web アプリケーションのドメインでは SeleniumWebDriver[5] や画面結果のピクセル比較技術 [6] 等の既存研究やツールが存在し一定の成果が出ているが [2]，Step1 の「結合テストレベルの網羅的なテストケースを旧ソフトウェアから自動生成する」に関しては，一定の成果が出ている既存研究は今回の調査では確認できなかった．結合テストにおけるテストケースの網羅性は，テストシナリオの網羅性と，テストデータの網羅性の２種類が存在する．テストシナリオは，JSTQB[7] によると，「テスト実行のために，一連の手順を定めたドキュメント」と定義される．例えば web アプリケーションの場合，「ID とパスワードを入力し，ログインボタンをクリックし，ログインが成功することを確認」といったテストシナリオが考えられる．テストデータは，JSTQB[7] によると，「テスト実行前に実在するデータであり，テスト対象のコンポーネントやシステムに影響を与えたり，影響を受けたりするもの」と定義される．例えば web アプリケーションの場合，「ID=0001，パスワード=aaaa」といったテストデータが考えられる．基本的には 1 つのテストシナリオに対して複数のテストデータが存在するため，網羅性を担保するためにはテストデータ生成の方が労力がかかる．そこで，本研究では，ソフトウェアから網羅的なテストデータを自動生成することを対象とする．テストデータの網羅性の観点は，テスト対象によって異なるが，本研究では様々なドメインで共通して用いることのできる，コードに対するカバレッジ基準を用いて網羅性を評価する．カバレッジ基準の中でも C2 カバレッジを高めるテストデータを自動生成することを技術的な目標とする．C2 カバレッジを高めるテストデータを自動生成することによって，より確実に人手で作成したテストケースをカバーできるだけではなく，ユーザの要望によっては，C0，C1 等のより緩いカバレッジ基準でのテストデータの出力も可能となる．

2. 結合テストのテストデータ自動生成に求められる要件

本研究では，結合テストレベルの網羅的なテストデータを自動生成することが目標である．本章では，結合テストの特徴を踏まえた，結合テストレベルのテストデータ自動生成に求められる２つの要件について説明する．

要件１：ネストが深い箇所に存在する分岐に対してテストデータが生成できること

結合テストでは，テスト対象が単体テストと比較して複雑になるため，コードのネスト（本論文では，条件分岐の入れ子構造のことを意味する）が深くなりやすい．コードのネストが深くなると満たさなくてはならない分岐の条件の数が増えるため，テストデータ生成が困難となる．例えば図 1 において，「ゴール 1 」に到達するための条件は，分岐 1 において「$d \leq 0$」を満たすことのみであり，「$a > 0$」を満足すればテストデータ b，c の値は任意である．しかし「ゴール２」に到達するための条件は，分岐２において「$e \leq 0$」を満たすことであるが，分岐２に到達するためには分岐 1 において「$d > 0$」を満たす必要があるため，結果として必要な条件は「$a > 0 \,\&\&\, b \leq 0$」の２つとなる．同様に「ゴール３」に到達するための条件は「$a > 0 \,\&\&\, b > 0 \,\&\&\, c \leq 0$」と必要な条件が増加する．結合テストの場合は，単体テストと比較してコードのネストが深くなりやすいため，ネストが深い箇所に存在する分岐に対してもテストデータが生成できることが求められる．

要件２：複数のプロセスによって構成されるシステムに対してテストデータが生成できること

結合テストでは，テスト対象は複数のプロセス（プログラムの実行単位）から構成されていることが多い．複数のプロセスで構成されている場合，プロセス間は互いにブラックボックスであり，入出力のやり取りのみが行われる．したがって，入力したテストデータが，どのような処理を

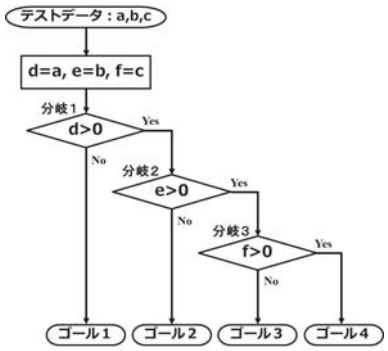

図 1 ネストを持つコードのフロー図（例 1）

され，最終的に分岐にどのような影響を与えるかというトレースを取れなくなるため，トレースが取れなくても，テストデータが生成できることが求められる．

3. 従来手法

従来の，テスト対象からテストデータを自動生成する技術は，主に単体テスト向けに研究がされており，結合テストに適用した研究例はほとんど存在しない．したがって本章では，単体テストにおける従来のテストデータ生成手法について紹介し，結合テストに用いた場合の，前章で紹介した 2 つの要件に対する適性について説明する．

3.1 Concolic Testing

Concolic Testing[8][9] では，入力値を記号と置き，プログラムを実行しながらパスを通るための制約情報（パス式）を入手する．得られたパス式を反転させて生成される新たなパス式を制約ソルバで解くということを繰り返すことで，パスカバレッジを高めるテストデータの集合を得る．例えば図 1 において，テストデータを「$a = 1, b = 2, c = 3$」と置いてプログラムを実行すると，「ゴール 4 」に到達する．この時，同時に「$a = A, b = B, c = C$」のようにテストデータを記号と置くと，「$A > 0 \,\&\&\, B > 0 \,\&\&\, C > 0$」といったパス式が取得できる．本パス式は，「ゴール 4 」に到達するために必要な条件を表す．したがって別のゴールに到達したい場合は，本パス式を反転させれば良い．例えば「$A > 0 \,\&\&\, B > 0 \,\&\&\, C \leq 0$」のように反転させた場合，「ゴール 3 」に到達することがわかる．この反転させたパス式を満たすような具体的なテストデータを，制約充足ソルバを用いて生成することで，「$a = 1, b = 2, c = -1$」等の，「ゴール 3 」に到達するためのテストデータが得られる．本工程を繰り返すことで，すべてのパスに対してテストデータを生成することが可能となるため，網羅性が向上する．

上記の例より，Concolic Testing の要件 1 への適性は高いと評価できる．Concolic Testing では 1 本のパスに対し，そのパスを通るための入力の条件を厳密に求めることができるため，ネストが深い箇所に存在する分岐に対してもテストデータが生成できる．しかし要件 2 への適性は低い．上記の例のように，Concolic Testing は入力したテストデータが分岐に到達するまで，厳密に記号でトレースする手法であるため，途中でテストデータが外部プロセスに入ると，記号のトレースが途切れる．例えば図 2 において，テストデータの入力と分岐 1 の間では，「$a+ = 1$」といった処理が行われているが，この処理が行われているのは別プロセスであるため，記号のトレースが途切れてしまう．したがって入力時は「$a = A$」とした記号が，分岐 1 に到達した時点ではトレースが途切れているため，パス式を生成することができない．本問題を解決するためには，全ての外部プロセスに対して，テストデータのトレースが行えるような機能を実装する必要があるが，これは現実的ではないため，Concolic Testing の結合テストへの適用は困難であると考えられる．

3.2 SBST

SBST（Search-Based Software Testing）[10] では，各分岐に対して分岐を満たす度合いを定量的に評価する評価関数を設計し，所望の評価関数の値を得るためのテストデータをヒューリスティックに生成していくことで，網羅性の高いテストデータを生成する．所望の評価関数の値を得るために適した入力の生成は，探索アルゴリズムを用いて行う．探索アルゴリズムは複数存在するが，SBST では遺伝的アルゴリズム（以下 GA）が最も多く用いられている [10]．GA は自然界の仕組みを模倣したアルゴリズムであり，データ（解の候補）を遺伝子で表現した「個体」を複数用意し，適応度の高い個体を優先的に選択して選択・交叉・突然変異などの操作を繰り返しながら解を探索する手法である．また個体はテストデータ，遺伝子はテストデータに含まれる変数の値と設定するのが一般的である．SBST で用いる探索アルゴリズムとして GA が最も多く用いられている理由は，GA のみが持つ，複数のテストデータの「遺伝子」を混ぜ合わせる交叉と呼ばれる工程が，ソフトウェアテストにおいて有効だからである．特にネストが深い箇所に存在する分岐に対しては，前章で説明したようにテストデータの各変数がそれぞれ適切な値を持つ必要がある．複数のテストデータがそれぞれ，いくつかの適切な値の変数を持つ場合，交叉の工程において，それらを組み合わせることですべての変数が適切な値を持つテストデータが生成できる可能性が高まる．したがって GA は，他の探索アルゴリズム（例：焼きなまし法，タブーサーチ等）よりも効率的に探索を進めることが可能である．図 2 において，SBST を用いて分岐 1 で「$c \leq 0$」を満たすようなテストデータを生成したい場合，評価関数 E を $E = c - 1$ と設計することで，c の値が小さいほど評価関数の値も小さくなり，「$c \leq 0$」を満たす度合いが大きくなると定量的に評価することができる．まず，$a = 10$ として実行すると，評価関数の値は，$E = 9$ となる．続いて，仮に $a = 5$ とし

て実行すると，評価関数の値は，$E = 4$ となる．この場合，後者のテストデータの方が「$c \leq 0$」を達成するためには優れていると評価できる．探索アルゴリズムによって，評価関数の値が優れているテストデータ，つまり a の値が小さいテストデータをベースに，新しいテストデータの生成が行われるため，徐々に a の値が小さいテストデータが生成されていき，最終的に $a = -1$ 等のテストデータが取得できる．

上記の例より，SBST の要件 2 への適性は高いと評価できる．SBST では，入力したテストデータの値と，分岐に対する評価関数の値のみを用いるため，Concolic Testing のように入力値のトレースを必要としない．したがって，入力と評価したい分岐までの間に，外部プロセスを複数通過しても，テストデータの生成は可能である．しかし要件 1 への適性は十分ではないと考えられる．例えば図 1 において，分岐 3 で「$f > 0$」を満たすようなテストデータを生成したい場合，評価関数 E は $E = f$ となり，評価関数の値が小さいほど，「$f > 0$」を満たすと考えることができる．前述した通り，SBST の場合，入力したテストデータと分岐の関連性は，あくまでも数回プログラムを実行して得られた評価関数の値を用いた推測であり，Concolic Testing と異なり厳密な繋がりはわからない．そのため，仮にテストデータ「$a = 1, b = 2, c = -1$」実行して評価関数の値 $E = -1$ を取得したとしても，評価関数 E の値はテストデータ c のみに依存するということはわからないため，次の探索においてテストデータ a と c を変更して「$a = -1, b = 2, c = 1$」とした場合，分岐 1 でパスが逸れてしまい分岐 3 に到達しなくなるため，評価関数の値が得られない．SBST は，評価関数の値を元にテストデータ生成を改善する手法であるため，評価関数の値が得られないと探索が進まなくなってしまう．従来のランダムな要素の強い探索アルゴリズムでは，ネストが深いほどテストデータをパスが逸れないように操作できる可能性は低くなるため，結合テストのようにネストが深い箇所に存在する分岐が多く存在する場合，現実的な時間内では探索が終わらないため，テストデータの生成ができない [10]．例えば，あるテスト対象において，テストデータの入力の数が n 個，各入力に対応する判定条件を持つ分岐が n 個存在し，図 3 に示すような入れ子構造になっているとする．このとき，ネストの最も深い箇所に存在するゴールに到達する（＝分岐 n を True する）ことが目標であるとする．今，分岐 n まで到達するが，分岐 n で False になるテストデータがあり，本テストデータに入力値の操作をして，目標を達成するテストデータを生成する．$1 \leq m \leq n$ の範囲における任意の自然数 m において，入力 m を操作する確率をそれぞれ Q_m，その操作によって，入力 m が対応する分岐 m の判定結果が変更される確率を R_m とした場合，分岐 n を True するためには，分岐 n に到達するまでの分岐の判定結果を

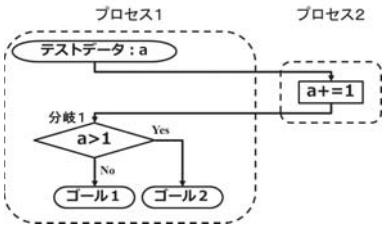

図 2 ネストを持つコードのフロー図（例 2）

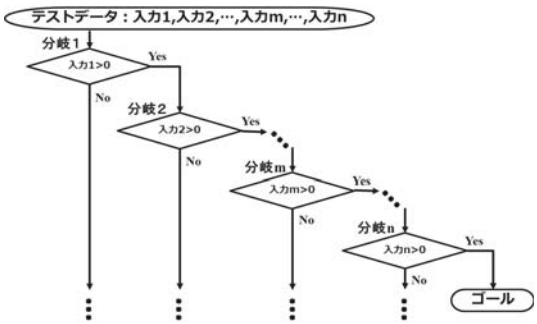

図 3 ネストを持つコードのフロー図（例 3）

変更することなく，分岐 n が True になるような入力 n を生成する必要あるため，目標を達成できる確率 P は式 (1) のように定式化できる．

$$P = \prod_{m=0}^{n-1} (1 - Q_m R_m) Q_n R_n \tag{1}$$

$n = 10$，$Q_m = 1/3$，$R_m = 1/3$ とした場合，式 (1) の結果は $P \approx 0.0385$ となるため，約 3.9%の確率でしか目標となるテストデータの生成ができないことがわかる．したがって多くの試行回数が必要となり，テストデータ生成に莫大な時間がかかる．

4. 提案手法

本研究では SBST において，「ネストが深い箇所に存在する分岐に対してテストデータが生成できない」という問題を解決し，要件 1 を満たすことで結合テストへの適用を目指す．探索アルゴリズムとして遺伝的アルゴリズム [11]（以下 GA）を選択し，テストデータの各変数を 1 つずつ変化させることで，目標となる分岐に至るまでのパスが逸れる確率を低減し，評価関数の値が取得できる確率を高め，探索を効率化するデルタ操作法と，目標となる分岐に至るまでのパスを通るテストデータと，目標となる分岐に対して優れた評価関数の値を記録するテストデータを交叉することで，目標とするパスを通過するテストデータを生成するキメラ交叉法の 2 つの手法を提案することで，ネストが深い箇所に存在する分岐に対しても現実的な時間内での網羅的なテストデータの生成を可能とする．共に，探索を効率化し，ネストが深い箇所に存在する分岐に対してもテストデータの生成を現実的な時間内で可能とする（要件 1 を満たす）手法であるが，デルタ操作法は選択・交叉・突然

変異すべてに適用されるため探索全般（C1・C2 カバレッジの向上）に，キメラ交叉法は交叉のみに適用されるため，複数のテストデータを組み合わせるという交叉の特性上，デルタ操作法で生成した，交叉に適したテストデータが出そろう探索中盤以降（C2 カバレッジの向上）に効果を発揮する．したがって要件 1 を満たすために必須となる手法はデルタ操作法，C2 カバレッジを向上させるための，中盤以降の探索をより効率化する手法がキメラ交叉法であるということが言える．

提案手法である，デルタ操作法とキメラ交叉法による GA を用いた SBST について，全体像を図 4 に示し，以下に各手順を説明する．デルタ操作法は (3-1)，(3-2)，(3-3) で，キメラ交叉法は (3-2) で用いられている．

(1) 事前準備

SBST で必要となる情報である「分岐の判定結果」,「分岐の判定順番」,「評価関数の値」をテスト対象プログラムの実行時に取得するために，プログラムの動きを変えないように注意しながら，コード挿入を行う．分岐の判定順番とは，どの順番で各分岐が実行されたかの結果のことであり，評価関数は分岐の判定条件の判定結果を定量的に表したものであり，判定結果が true の時は負の値，false の時は正の値を取る．例えば $if(x > 0)then\{hoge;\}$ という条件式において，$x = 1$ の時と $x = 100$ の時では後者の方が条件を満たしている度合いは大きと判定し，評価関数の絶対値は大きくなる．boolean 型を返すメソッドなどの何らかの評価文については定量的な測定が困難であるため，条件を満たしていれば −1，満たしていなければ 1 と評価関数を定義する．

(2) パス決定

通過するソースコード上のパスが互いに被らないテストデータと，そのテストデータを実行したときの各分岐判定結果，各分岐判定順番，評価結果の集合（以下集合 U）をベースにパス決定を行う．集合 U が存在しない場合は，任意の値のテストデータを実行し，集合 U を生成する．はじめに，集合 U の中から 1 つテストデータを選択し，そのテストデータが通った分岐条件を後ろから反転していき，集合 U が通過していないパスであればそのパスを今回通りたいパス（以下パス A），パス A の中で反転させた分岐を分岐 R とする．例えば選択されたテストデータが「分岐 1 : true，分岐 2 : false」というパスを通過していた場合，集合 U が「分岐 1 : true，分岐 2 : true」を通過していなければそれがパス A となり，分岐 2 が分岐 R となる．集合 U のすべてのテストデータで上記の操作を行っても新たなパス A が得られない場合，本アルゴリズムは打ち切りとする．

(3) テストデータ生成

集合 U を 3 つの集合に分ける． 1 つ目は分岐 R に到達するまで通過した分岐条件がパス A に等しいテストデータの集合 X，2 つ目は分岐 R に到達した集合 X 以外のテストデータの集合 Y，3 つ目は分岐 R に到達していないテストデータの集合 Z である．以上 3 つの集合を各集合内で分岐 R の評価関数の値でソート（分岐 R が true であれば負，false であれば正）をかけ，上から並べる．集合 X →集合 Y →集合 Z の順番で結合して新たに得られるテストデータの集合を集合 N とする．

(3-1) 選択（デルタ操作法）

集合 N の中から上から事前に決定した値だけテストデータを選択する．選択されたテストデータに対し，デルタ操作法を用いて変数の操作を行う．デルタ操作法は，選択されたテストデータの各変数を 1 つずつ変化させることで，目標となる分岐に至るまでのパスが逸れる確率を低減し，評価関数の値が取得できる確率を高め，探索を効率化する手法である．3.2 節で説明したように，ネストが深い箇所に存在する分岐に対して変数操作をランダムに行うと，目標の分岐（分岐 R）に到達する前にパスが逸れてしまう可能性が非常に高いため，評価関数の値が取得できなくなり，従来の選択では探索速度が著しく低下してしまう．一方デルタ操作法では，1 つのテストデータに対し，各変数を 1 つずつ操作したテストデータを生成（ 1 つのテストデータが 10 個の変数を持つ場合，変数操作された新しい 10 個のテストデータを生成）する．例えば図 6 の選択のように，優れたテストデータに対し，各変数を 1 つずつ操作した新しいテストデータを生成する．変数操作を 1 つに限定することで，目標の分岐（分岐 R）に到達するまでのパスが逸れにくくなり，評価関数の値の取得が可能となるため，探索を進めることができる．したがって，現実的な時間内でのテストデータの生成が可能となり，要件 1 を満足することができる．デルタ操作法を図 3 に適用すると，確率 P は式 (2) のように表すことができる．

$$P = \begin{cases} 0 & (m < n) \\ R_n & (m = n) \end{cases} \tag{2}$$

式 (2) より，デルタ操作法では，ネストの値 n に関わらず，$m = n$ のとき，必ず $P = R_n$ となるため，ネストの深さに依存しないことがわかる．$n = 10$，$R_m = 1/3$ とした場合，$m = 10$ のとき式 (2) の結果は $P = 1/3$ となるため，約 33.4%の確率で目標となるテストデータの生成ができる．したがってネストが深い箇所に存在する分岐に対しても，探索を効率的に進めることができるようになるため，従来より短い時間でテストデータが生成できる．

(3-2) 交叉（デルタ操作法＋キメラ交叉法）

集合 X と集合 Y からそれぞれ事前に決定した値だけ上から順番にテストデータを選択する．集合 X から選択されたテストデータと集合 Y から選択されたテストデータで交叉を行う．集合 X と集合 Y から 1 つずつ無作為にテストデータを選択し，集合 X から選択されたテストデータをベースに，各変数に対して 1 つずつ，集合 Y からから選択

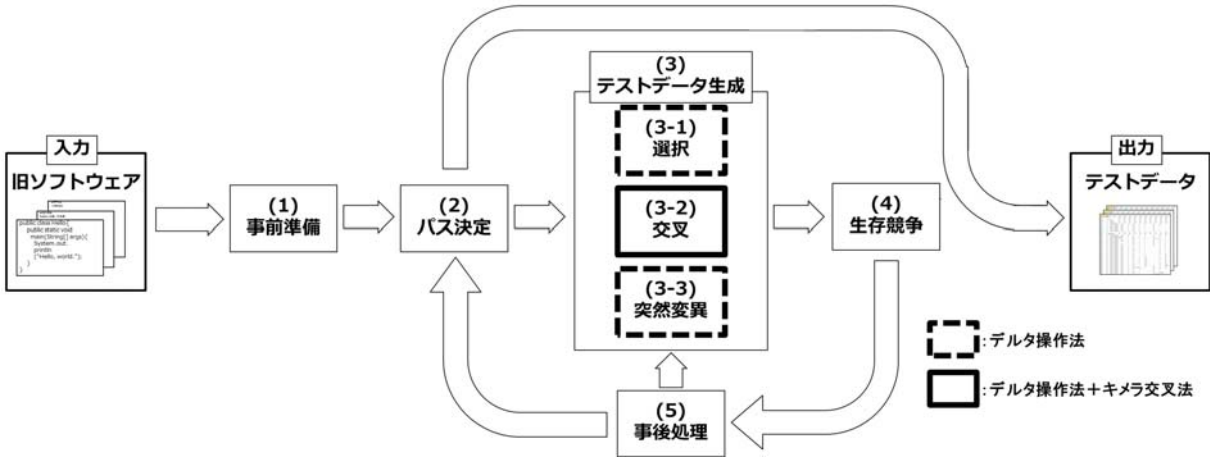

図 4　提案する GA を用いた SBST

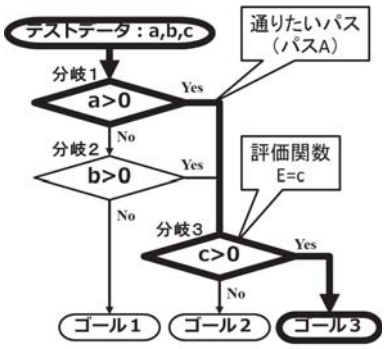

図 5　ネストを持つコードのフロー図（例４）

表 1　評価に用いた GA の各定数

定数名	値
選択で生成するテストデータ数［個］	12
交叉で生成するテストデータ数［個］	12
突然変異で生成するテストデータ数［個］	12
最大世代数［世代］	5

されたテストデータの変数と入れ替えて新しいテストデータの集合 C を生成する（デルタ操作法）．もし集合 Y の中に分岐 R を所望の判定結果で通過するテストデータがあれば，交叉により，集合 X の特徴である「分岐 R までに所望のパスを通過する性質」と混ざることでパス A を通るテストケースが生成される可能性が大幅に高まる．集合 X と集合 Y を混ぜ合わせる交叉を，キメラ交叉法という．キメラ交叉法は，目標となる分岐に至るまでのパスを満たすテストデータと，目標となる分岐に対して優れた評価関数を記録するテストデータを交叉することで，目標とするパスを通過するテストデータを生成する手法である．従来の交叉では，図 6(a) に示すように，評価関数の値が優れているテストデータ同士を組み合わせる．一方で，生成したいテストデータの要件は，「目標とする分岐まで通りたいパス（パス A）通りに通過すること」と，「目標の分岐（分岐 R）で達成したい判定条件を実現すること」である．評価関数の値が優れているということは，目標の分岐（分岐 R）で達成したい判定条件に対して優れているということしか保証しないため，後者の要件にのみ関連し，前者の要件には関連しない．そのため，従来の交叉では生成したいテストデータの要件が満たせない．キメラ交叉では，図 6 に示すように，生成したいテストデータの各要件を満たす条件のテストデータ同士を交叉させるため，通りたいパス（パス A）を通過するテストデータの生成が可能となる．

(3-3) 突然変異（デルタ操作法）

集合 N の中からランダムに事前に決定した値だけテストデータを選択する．選択されたテストデータに対し，(3-1) 選択と同様，デルタ操作法を用いて変数の操作を行い，新しいテストケースの集合 M を作成する．例えば図 6 の突然変異のように，ランダムに選択されたテストデータに対し，各変数を 1 つずつ操作した新しいテストデータを生成する．

(4) 生存競争

集合 S，集合 C，集合 M のテストケースを実行することで，(1) で行った事前準備によって分岐の判定結果と判定順番，そして評価関数の値を取得できる．

(5) 事後処理

得られた分岐の判定結果と判定順番を分析することで，集合 S，集合 C，集合 M の中にパス A のを通過するテストケースの有無を調べ，存在すればそのテストケースを集合 U に加え，世代数を 1 にリセットし (2) へ戻る．存在しなければ集合 S，集合 C，集合 M を集合 U とし，世代数に 1 を加えて (3) へ戻る．世代数が最大世代数に到達した場合，世代数を 1 にリセットし (2) へ戻る．

5. 評価

5.1 方法

提案手法の有効性を確認するために，以下の 3 つの手法における C2 カバレッジ，テストデータ生成に要した時間，

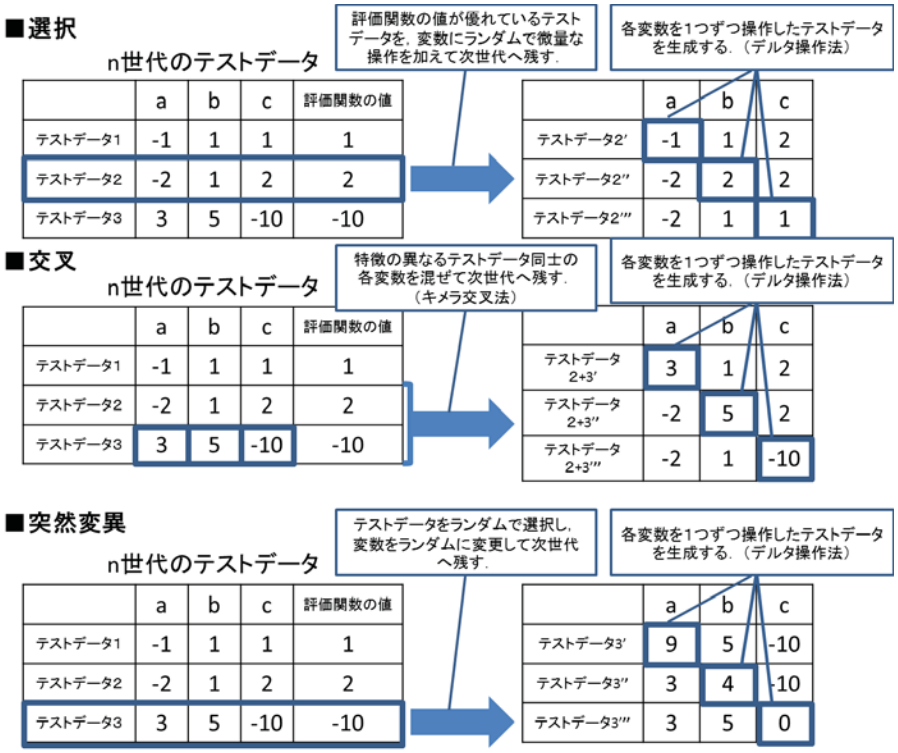

図 6　提案する遺伝的アルゴリズムを用いた SBST の選択・交叉・突然変異（例）

表 2　従来手法と提案手法の比較

	C2 カバレッジ [%]	実行時間 [分]	テストケース数 [個]
GA を用いた SBST（従来）	43.4	1045	1690
GA を用いた SBST（従来）＋デルタ操作法（提案）	88.9	580	3467
GA を用いた SBST（従来）＋デルタ操作法（提案）＋キメラ交叉法（提案）	88.9	198	3467

生成したテストケース数の比較を行った．

- GA を用いた SBST（従来）
- GA を用いた SBST（従来）＋デルタ操作法（提案）
- GA を用いた SBST（従来）＋デルタ操作法（提案）＋キメラ交叉法（提案）

GA における各定数の値は，すべて表 1 の値で統一した．テスト対象は，java で記述された約 0.4KLoc のプログラムを用いた．テスト対象を実行するために必要な入力は 12 個あり，これらの入力をテストデータとして，SBST を実行した．またテスト対象には 23 個数の分岐が存在し，ネストの深さの最大値は 11（分岐に到達する前に 11 個の他の分岐を通る必要がある）である．ネストが 11 の分岐は 3 個，10 の分岐が 10 個，9 以下の分岐が 10 個存在する．

5.2　結果

評価結果を表 2 に示す．デルタ操作法とキメラ交叉法による GA を用いた SBST が C2 カバレッジとテストデータ生成に要した実行時間において，最も優れた値を記録した．デルタ操作法とキメラ交叉法を用いた場合では，従来の GA を用いた SBST と比較して，約 2 倍の C2 カバレッジを記録し，また約 5.3 倍の速さでテストデータを生成することに成功した．デルタ操作法のみを用いた場合と比較しても，C2 カバレッジの値は変わらないものの，約 2.9 倍の速さでテストデータを生成していることも示せた．また，提案手法で C2 カバレッジを満たすように生成したテストデータの数は，3467 個であったが，C1 カバレッジを満たすだけであれば，21 個のテストデータで十分であることもわかった．ユーザに要望によっては，少ないテストデータ数で C1 カバレッジを達成するテストを実施することも可能である．

5.3　考察

提案手法が短い時間で高い C2 カバレッジを記録出来た理由は，デルタ操作法とキメラ交叉法により効率的にテストデータの生成ができたためと考えられる．実際にテストデータの生成において，提案手法では約 65%が最大世代数を超える前に目標とするパスに対してテストデータの生成ができており，これは式 (1) と式 (2) で示したように，提案手法の方がネストが深い場合でも高い確率でテストデータの生成ができることの裏付けであると考えられる．一方，従来の GA を用いた SBST では，約 8%しか最大世代数に到達する前でのテストデータ生成ができておらず，

目標とするパスを通るテストデータがほとんど生成できなかったと考えることができる．それでも C2 カバレッジが 43.4%となった理由は，探索の過程で生成されたテストデータが，偶然，目標とするパス以外の箇所を通過し，カバレッジを向上させるというケースが多く存在したためと考えられる．また，デルタ操作法のみを用いた場合でも高い C2 カバレッジを記録したことから，要件 1 を満たすために必須となる手法はデルタ操作法であることがわかり，キメラ交叉法と併用した場合の方がテストデータ生成時間が約 2.9 倍速かったことから，探索をより効率化する手法がキメラ交叉法であるということがわかった．

今回，C2 カバレッジが 100%にならなかった理由は，2 箇所，通過することのできなかった分岐が存在するためである．1 つ目は boolean 型を返すライブラリのメソッドを用いた分岐である．boolean 型を返すライブラリのメソッドを用いた条件は，評価関数で定量的な評価ができないため，SBST が苦手とするケースである．2 つ目は異常系の処理に関する分岐のため，どのような入力を与えても条件を満たすことができなかった．

6. まとめ

本研究では，テストデータを自然界のアルゴリズムを模倣し進化させていく手法である，GA を用いた SBST に対して，選択されたテストデータの各変数を 1 つずつ変化させることで，目標となる分岐に至るまでのパスが逸れる確率を低減し，評価関数の値が取得できる確率を高め，探索を効率化する手法であるデルタ操作法と，目標となる分岐に至るまでのパスを満たすテストデータと，目標となる分岐に対して優れた評価関数を記録するテストデータを交叉することで，目標とするパスを通過するテストデータを生成する手法であるキメラ交叉法を提案した．

本提案手法により，従来の現実的な時間内では十分なテストデータが生成できないという問題点を解決することで，コードに対する網羅性（カバレッジ）の高い，結合テストにおけるテストデータの自動生成を可能とし，評価によって，従来の GA を用いた SBST より約 5.3 倍の速さで，約 2 倍高い C2 カバレッジを達成できることを確認した．

今後は，今回 SBST でテストデータの生成ができなかったパスに対するテストデータの生成に取り組み，より高いカバレッジを実現するだけでなく，網羅的なテストシナリオの生成に取り組むことで，本研究の最終的な目的である，結合テストレベルの回帰テストにかかる人の稼働を，自動化によって削減することを目指す．

参考文献

[1] 独立行政法人情報処理推進機構，ソフトウェア開発データ白書 2014-2015，独立行政法人情報処理推進機構，2014.
[2] “ソフトウェア産業の実態把握に関する調査,” http://www.ipa.go.jp/files/000004628.pdf.
[3] “Open2test,” http://www.open2test.org/.
[4] “Seleniumide,” http://www.seleniumhq.org/projects/ide/.
[5] “Seleniumwebdriver,” http://www.seleniumhq.org/projects/webdriver/.
[6] W.G.H. Sonal Mahajan, “Finding html presentation failures using image comparison techniques,” ASE ’14 Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp.91–96, 2014.
[7] “JSTQB,” http://www.jstqb.jp/index.html.
[8] K. Sen, “Concolic testing,” ASE ’07 Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp.571–572, 2007.
[9] “CATG,” https://github.com/ksen007/janala2.
[10] P. McMinn, “Search-based software testing past, present and future,” ICSTW ’11 Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp.153–163, 2011.
[11] L.P. M Srinivas, “Genetic algorithms: a survey,” Computer,Volume:27,Issue: 6, pp.17–26, 1994.
[12] A.A. Gordon Fraser, “Evosuite: automatic test suite generation for object-oriented software,” ESEC/FSE ’11 Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp.416–419, 2011.