動的記号実行と探索的テストによる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，動的記号実行 (Dynamic symbolic execution，以下 DSE) と探索的テスト (Search-based software testing，以下 SBST) を組み合わせることによって，結合テストにおける，現実的な時間内でのコードに対する網羅性（カバレッジ）の高い，テストデータの自動生成に取り組む．従来の DSE と SBST は，主に単体テスト向けに研究がされており，結合テストに適用した場合，テスト対象が単体テストとは異なり，複数のコンポーネントから構成されているため，構造が複雑になり，網羅性の高い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，初めに DSE でテストケースを生成し，DSE では通れなかったパスに対して SBST でテストケースを生成することで，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に対する網羅性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，回帰テスト，動的記号実行，探索的テスト

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，保守改造案件の工数の 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，単体テストの工数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常に乏しいにもかかわらず，保守改造案件の工数の多くを占めていることが大きな問題となっている．

[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

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工数を，自動化によって削減することを最終的な目的とする．結合テストは JSTQB [3] によると，「統合したコンポーネント（独立してテストできるソフトウェアの最小単位）間のインターフェースや相互作用の欠陥を検出するためのテスト」と定義される．本論文では，複数のコンポーネントから構成されたソフトウェアに対するテストを規模を問わず結合テストと定義する．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 [4] [5] [6] 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，抜本的な工数改善にはならない．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工数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，前バージョンのソフトウェア（以下，旧ソフトウェアと定義）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 [6] や画面結果のピクセル比較技術 [7] 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．そこで，本研究では，ソフトウェアから網羅的なテストケースを自動生成することを対象とする．テストケースの網羅性の観点は，テスト対象によって異なるため，本研究では検証対象として Web アプリケーションを用いる．Web アプリケーションは一般的に，プレゼンテーション層，アプリケーション層，データベース層の３層モデルで構成されているが，アプリケーション層内部の開発者記述箇所（フレームワークやライブラリ等以外）の網羅性（図 1 において，網羅的にテストしたい範囲の各コンポーネントを組み合わせた時に通りうるパスの網羅性）を向上させることを目標とする．アプリケーション層内部の網羅性は，コードに対するカバレッジ基準で評価をすることとし，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現実的な時間内で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. 結合テストにおけるテスト対象の特徴

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．しかし従来のテストデータ生成技術は主に単体テストにおいて成果が出ており，結合テストでは十分な成果は出ていない [8]．これは，単体テストにはない結合テストの特徴が，テストデータの自動生成において障壁になっているためと考えられる．本章では，単体テストにはない結合テストの特徴について述べ，次章でそれらの特徴がなぜ障壁になっているかについて述べる．

単体テストは JSTQB [3] によると，「個々のソフトウェアコンポーネント（独立してテストできるソフトウェアの最小単位）のテスト」と定義される．コンポーネントは，具体的にはメソッドや関数などが挙げられる．一方，結合テストは JSTQB [3] によると，「統合したコンポーネント間のインターフェースや相互作用の欠陥を検出するためのテスト」と定義される．したがって，結合テストでは，複数のコンポーネントで構成されたソフトウェアをテスト対象としていることになる．複数のコンポーネントで構成されていることにより，結合テストにおけるテスト対象は，以下の２つの特徴を持つ．

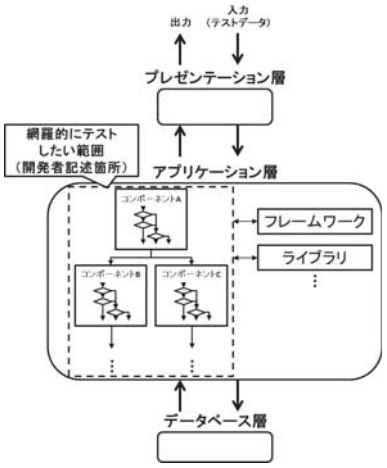

図 1　本研究で想定するテスト対象

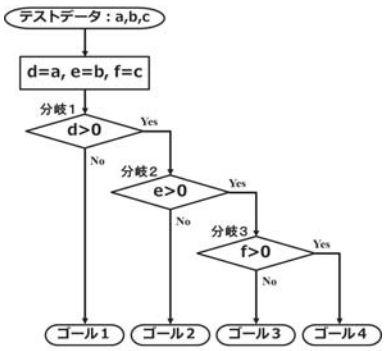

図 2　コードのフロー図（例１）

特徴１：単体テストと比較して，網羅的にテストしたい範囲を構成する分岐の数が増加

特徴２：単体テストと比較して，網羅的にテストしたい範囲以外のモジュールとのインターフェースや相互作用が増加

特徴２における網羅的にテストしたい範囲以外のモジュールは，図 1 において，フレームワーク，ライブラリ，データベースなどが挙げられる．

3. 従来手法

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に前章で紹介した結合テストにおけるテスト対象の２つの特徴が，どの

ように障壁になっているかについて説明する．

3.1 DSE

DSE [9] [10] 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．得られたパス式の一部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，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 2 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，DSE では特徴 1 は障壁にならない．DSE では 1 本のパスに対し，そのパスを通るための分岐の条件をすべて収集し，入力と収集した分岐の条件の関係性を厳密に表すパス式を生成し，そのパス式を満たす入力の値を厳密に求めることができるため，パスを通るために必要な分岐の条件の数に依存せず，テストデータが生成できる．しかし特徴 2 が障壁になる．上記の例のように，DSE は入力したテストデータが通るパスを，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等の網羅的にテストしたい範囲外に入ると，記号のトレースが途切れる．例えば図 3 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，網羅的にテストしたい範囲だけでなく，フレームワーク，ライブラリ，データベース等のすべての網羅的にテストしたい範囲以外のモジュールに，入力した値をトレースする機能を実装する必要があるが，そのような機能の実装は膨大なコストがかかるため現実的でない．

3.2 SBST

SBST（Search-Based Software Testing）[11] 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 [11]．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．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 3 において，SBST を用いて分岐 1 で「$a > 1$」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 $E = a - 1$ と設計することで，a の値が大きいほど評価関数の値も大きくなり，「$a > 0$」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，$E = -10$ 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，$E = -5$ となる．この場合，後者のテストデータの方が「$a > 0$」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 の値が大き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 の値が大きいテストデータが生成されていき，最終的に $a = 2$ 等のテストデータが取得できる．

上記の例より，SBST は特徴 2 は障壁とならないことがわかる．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，DSE 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，網羅的にテストしたい範囲以外のモジュールを複数通過しても，テストデータの生成は可能である．しかし特徴 1 が障壁となる．例えば図 2 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，DSE と異なり

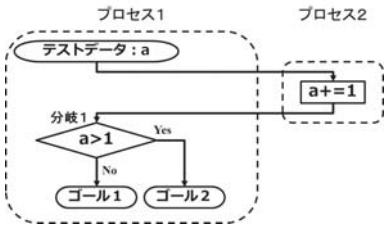

図 3　コードのフロー図（例２）

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を含む探索アルゴリズムでは，あるパスを通るための分岐の条件が多いほど，すべてを満たすテストデータを生成できる可能性は低くなるため，結合テストにおけるテスト対象のように，構成される分岐の数が多い場合，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 [11]．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 4 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．

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

$n = 10$，$Q_m = 1/3$，$R_m = 1/3$ 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ため，現実的な時間内では高い網羅性を持つテストデータは生成できない．

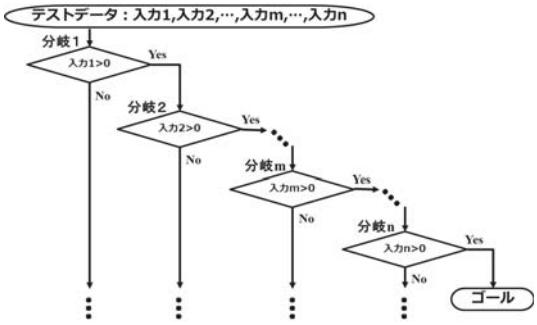

図 4　コードのフロー図（例３）

4. 提案手法

結合テストにおけるコードに対する網羅性の高いテストデータの生成を従来手法を用いて行った場合，DSE では，すべての外部プロセスに記号トレース機能を実装すると莫大な実装コストがかかるため，入力が外部プロセスに入り，その後，再び値としてテスト対象のプロセスに戻り，分岐の条件に影響を与えるようなパスに対してはテストデータが生成できず，SBST では，通過するために必要な分岐の条件の数が多いパスに対してテストデータを生成しようとすると，現実的な時間内では終わらないため，テストデータが生成できないという問題があった．そこで DSE は入力が外部プロセスに入り，その後，再び値としてテスト対象のプロセスに戻り，分岐の条件に影響を与えるようなパス以外に対してテストデータを生成し，SBST は，通過するために必要な分岐の条件の数が多いパス以外に対してテストデータを生成するように，DSE と SBST を組み合わせることで，結合テストにおける，現実的な時間内でコードに対する網羅性の高いテストデータを生成する手法を提案する．

本手法では，はじめに DSE を実行し，DSE で生成できるテストデータに関しては迅速に生成を行いつつ，DSE 実行中にテスト対象に実行したテストデータと，対応する評価関数の値を取得する．その後 DSE 中に取得した，テストデータと対応する評価関数の値を元に，DSE では通ることのできなかったパスに対するテストデータを SBST で生成する．本手法による利点は 2 つ存在する． 1 つ目は DSE，または SBST を単体で実行した場合と比較して，高い網羅性を持つテストデータを生成できることである．DSE と SBST を組み合わせることで，特徴 1 を持つパスに対しては DSE でテストデータを生成し，特徴 2 を持つパスに対しては SBST でテストデータを生成することで，高い網羅性を持つテストデータの生成が可能となる． 2 つ目の利点は，DSE と SBST をそれぞれ独立して実行し，最後に得られたテストデータをマージした場合と比べ，高い網羅性を持つテストデータの生成が現実的な時間内で可能なこと

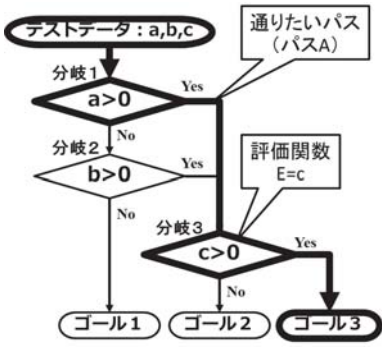

図 5 コードのフロー図（例４）

である．DSE を SBST それぞれ独立して実行した場合は，互いに生成したテストデータの共有ができないため，無駄が生じる．特に SBST 側では，テストデータ生成に莫大な時間がかかる，通るために必要な分岐の条件の数を多く持つパスに対して探索を行うため，現実的な時間内ではテストデータ生成が終わらない．一方提案手法では，はじめに DSE で，SBST で必要となる評価関数も取得しつつ，DSE で生成することのできるテストデータを生成し，続いて DSE 中に取得した，テストデータと対応する評価関数の値を元に SBST を実行することで，DSE で既に対象とし，テストデータを生成することのできたパス，特に SBST でテストデータに莫大な時間がかかる，通るために必要な分岐の条件の数を多く持つパスに対する探索を省略することができるため，SBST によるテストデータ生成を現実的な時間内で終わらせることができる．

提案手法について，以下に各手順を説明する．

(1) 事前準備

入力：旧ソフトウェア

出力：評価関数の値が取得できるようにコード変換された旧ソフトウェア

SBST で必要となる情報である評価関数の値を取得するために，プログラムの動きを変えないように注意しながら，網羅的にテストしたい範囲に対してコード変換を行う．コード変換のルールは，既存研究 [11] に従う．SBST で必要な情報を DSE 実行中に収集するため，DSE 実行前に，本工程を行っていることがポイントである．評価関数は分岐の判定条件の判定結果を定量的に表すための関数であり，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)DSE

入力：評価関数の値が取得できるようにコード変換された旧ソフトウェア

出力：テストデータと対応する評価関数の値

DSE を実行し，テストデータを生成する．DSE を行う過程で，(1) で行ったコード変換により，実行したテストデータに対応する評価関数の値が取得できる．

(3)SBST

入力：評価関数の値が取得できるようにコード変換された旧ソフトウェア，(2) で出力したテストデータと対応する評価関数の値

出力：(2) で出力したテストデータと本工程で生成したテストデータ

SBST を実行し，(2) で通過することのできなかったパスに対するテストデータを生成する．なお，探索アルゴリズムは SBST で最も多く用いられている [11]，GA を用いた．(2) で出力したテストデータと対応する評価関数の値を元に SBST を実行することで，既に (2) で通過したパスに対する探索を省略することができる．最終的に (2) で出力したテストデータと，本工程で生成したテストデータを合わせることで，コードに対する網羅性の高いテストデータの出力が可能である．

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

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，生成したテストケース数の比較を行った．

- 従来：DSE
- 従来：SBST
- 提案：DSE + SBST

SBST の探索である GA における各定数の値は，従来，提案問わずすべて表 1 の値で統一した．テスト対象は，アプリケーション層とデータ層で構成され，アプリケーション層が 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た．DSE は，既存の OSS である，CATG [10] を用いて実行し，SBST は新規に実装したツールを用いて実行した．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．

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]
従来：DSE	26.9	222	1047
従来：SBST	43.4	1045	1690
提案：DSE ＋ SBST	55.2	491	2149

5.2　結果

評価結果を表 2 に示す．提案手法が C2 カバレッジにおいて，DSE より約 28.3 ポイント，SBST より約 11.8 ポイント高い C2 カバレッジの値を記録した．4 章で述べた提案手法の 1 つ目の利点である，「DSE，または SBST を単体で実行した場合と比較して，高い網羅性を持つテスデータの生成」が確認できた．また，提案手法では SBST 単体よりも 554 分短い時間でのテストデータ生成を実現していることから，２つ目の利点である，「DSE と SBST をそれぞれ独立して実行し，最後に得られたテストデータをマージした場合と比べ，高い網羅性を持つテストデータの生成が現実的な時間内で可能」が確認できた．

5.3　考察

提案手法が DSE と SBST をそれぞれ単体で実行した場合と比較して，高い C2 カバレッジの値を記録することができたのは，結合テストにおけるテストデータ自動生成の課題について，課題 1 を DSE で解決し，課題 2 を SBST で解決したためである．また，テストデータ生成時間については，DSE を実行してから SBST を実行する提案手法の仕組み上，DSE を単体で実行した場合よりも約 269 分長くなったが，SBST を単体で実行した場合と比較して，約 554 分の短縮に成功した．これにより，提案手法における DSE 実行中に SBST で必要な評価関数の値を取得する方法によって，DSE と SBST をそれぞれ独立に実行し，得られたテストデータをマージする方法と比較して，テストデータ生成時間を約 554 分の短縮することができると推測できる．

一方，提案手法を用いても，100%の C2 カバレッジを達成することができなかった．入力が外部プロセスに入り，その後，再び値としてテスト対象のプロセスに戻り，多くの分岐の条件に影響を与える，といった処理が行われるパスについてテストデータが生成できなかったためである．このような処理が行われるパスに対しては，入力が外部プロセスを介して分岐の条件に影響を与えているため，DSE では解決することができず，またパスを通るために必要な分岐の条件の数が多いため，SBST でも解決することができなかったと考えられる．

6.　まとめ

本研究では，DSE と SBST を組み合わせることによって，結合テストにおける，現実的な時間内でのコードに対する網羅性（カバレッジ）の高い，テストデータの自動生成を実現した．従来の DSE と SBST は，主に単体テスト向けに研究がされており，結合テストに適用した場合，テスト対象が単体テストとは異なり，複数のコンポーネントから構成されているため，構造が複雑になり，網羅性の高い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，DSE と SBST を組み合わせることで，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し，評価によって DSE より約 28.3%，SBST より約 11.8%高い C2 カバレッジの値を記録することを確認した．今後は，提案手法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工数を，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] “JSTQB,” http://www.jstqb.jp/index.html.
[4] “Open2test,” http://www.open2test.org/.
[5] “Seleniumide,” http://www.seleniumhq.org/projects/ide/.
[6] “Seleniumwebdriver,” http://www.seleniumhq.org/projects/webdriver/.
[7] 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.
[8] A.M.B. Amit A Kadam, “A survey on generation of automated test data for coupling based integration testing,” European Journal of Advances in Engineering and Technology, 2014, pp.69–74, 2014.
[9] K. Sen, “Concolic testing,” ASE ’07 Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp.571–572, 2007.
[10] “CATG,” https://github.com/ksen007/janala2.
[11] 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.