Vol.2020-EMB-55 No.1
2020/12/11

車載向け統合テストケースのコードクローン解析に基づくテストケース効率生成技術の開発

加藤 有真[1,a)]　溝口 将史[1,b)]　吉村 健太郎[1,c)]　石郷岡 祐[1,d)]　池田 暁[2,e)]　佐藤 陽春[2,f)]

概要：車載ソフトウェア開発工数の多くを占める Hardware-in-the-loop simulator(HiLS) を用いた統合テストの効率化が重要である．近年，テストケースを形式記述し統合テストの実行が自動化された．さらなるコスト削減のため，テストケース生成の効率化が必要である．テストケースが検証するテスト条件は一般に自然言語で記述されているため，自然言語解析を適用し過去に開発したテストケースに紐付けられたテスト条件の中から最も言語的意味の類似したテスト条件を探索することにより，テストケースを再利用する手法がある．しかしながら，テスト条件同士の言語的意味が類似していても全く異なるテストケースを生成する必要がある場合には適用できない．そこで本研究において，テストケースをコードクローン解析してテストケース間の類似度を算出し，テストケース間の類似度をテスト条件間の類似度として対応付ける手法を提案する．まず，共通するコードクローンを持つテストケースをクラスタに集約する．次に，同一クラスタに含まれる各テストケースに紐づくテスト条件は同一クラスタに属するとみなしてテスト条件をクラスタリングする．最後に，クラスタ内部を解析してクラスタから抽出すべきテストケースを決定する．提案手法を評価し，テストケース生成工数を 79%削減可能な見込みを得た．

キーワード：車載ソフトウェア，統合テスト，テスト自動化，テストケース生成

Efficient integration test case generation with code clone analysis of integration test cases for automotive software

Abstract: Due to the progress of electrification and autonomous driving, it is necessary to make automotive software development more efficient. Testing occupies 40% of total cost for the development, and most of them are on integration test with hardware-in-the-loop simulators (HiLSs). So, the efficiency of integration test with HiLSs should be improved. Recently, cost for the integration test has reduced with test cases written in a formal language in such a way that test cases are interpreted and executed automatically by the HiLSs. Towards further cost reduction, test case generation should be made more efficient. Usually, each test case is designed based on a test condition to be verified. In most cases, the test condition is written in a natural language. Then, a method was proposed to search a test condition that is used in a previous software development project and is linguistically closest to the test condition to be verified this time, where a new test case is generated by the modification of the test case to verify the previous test condition. However, linguistic closeness does not always imply the similarity between test cases. Then, in this paper, we propose a method to identify a test case to be reused from previous projects by mapping the similarity between test cases to psudo distance between test conditions. Code clone analysis is executed on test cases to evaluate the similarity, and test cases and test conditions are clusterized based on it. A test case is extracted from a cluster of test cases with graph theoretical approach. The proposed method is evaluated with integration test cases for automotive software, and we have a prospect of 79% cost reduction on the test case generation.

Keywords: Automotive software, integration test, test automation, test case generation

[1] (株) 日立製作所
Hitachi City, Ibaraki Pref., Japan
[2] 日立オートモーティブシステムズ (株)
Yokohama City, Kanagawa Pref., Japan
[a)] yuma.kato.vd@hitachi.com
[b)] masashi.mizoguchi.re@hitachi.com
[c)] kentaro.yoshimura.jr@hitachi.com
[d)] tasuku.ishigoka.kc@hitachi.com
[e)] akira.ikeda.eo@hitachi-automotive.co.jp
[f)] akiharu.sato.ek@hitachi-automotive.co.jp

1. はじめに

自動車に搭載されるソフトウェアの役割が重要性を増し，その規模が増大しつづけている．1960 年代から始まる排気ガス規制の導入・強化や燃費向上技術の発展に伴い，複雑なエンジン制御がソフトウェアによって実現されている [1][2]．また，1980 年代ころより導入されたアンチロックブレーキシステム [3][4] や，2000 年ころより主流となった電動パワーステアリング [5]，近年開発された先進運転支援システム [6][7] など，車両制御の高度化が進んでおり，その基幹となるのが電子制御装置 (Electronic Control Unit) に実装されたソフトウェアである．近年の電動化，自動運転の進展により車載ソフトウェアの規模はますます増大することが想定される [8]．したがって，車載ソフトウェア開発の効率化が不可欠となっている．

現在，ほとんどの車載ソフトウェアは V モデルに従い開発されている．一般に，開発工数のうち設計・実装が 6 割，テストが 4 割を占めるとされている [9][10]．テストは単体テスト・統合テスト・システムテストからなる．単体テストは各コンポーネントに関するテストであり，ツールを用いて自動化されている [11][12]．統合テストはコンポーネント間のインタフェースや相互作用に関するテストであり，車載ソフトウェアには多数のコンポーネントが搭載されることから，非常に多くのテストケースを生成し実行する必要がある．システムテストはシステムとして仕様を最終的に満たしているかを確認するためのものであり，ソフトウェアの不具合やバグはシステムテストより前に検出・修正されていなければならない．したがって，車載ソフトウェア開発効率化のため，特に統合テストの効率化が重要である．

統合テストにおいては，電子制御装置に搭載されたペリフェラルへのアクセスや他の電子制御装置との間の通信についても検証する．したがって，ソフトウェアを電子制御装置上に実装し，プラントモデルを搭載した Hardware-in-the-loop simulator (HiLS) と呼ばれる装置と電子制御装置を信号線等を用いて接続し，それらの間で信号を物理的に入出力することによりテストを行うのが一般的である．従来は HiLS の入出力操作を手動で実行する必要があったが，近年それらの操作パターンを形式記述し読み込ませることにより，HiLS を用いた統合テストの実行が自動化された [13][14][15]．しかしながら大量のテストケースの生成は必要であるため，テストケース生成工数の削減が求められる．

テストケース自動生成には多くの既存研究が存在する．例えば，プラントモデルを解析する手法 [16] や製品仕様を形式記述する手法 [17]，コンポーネント間の入出力組み合わせを考慮した手法 [18] がある．ただし，これらの手法を適用できる条件は限られているため一部のテストにしか適用できず，実際のテストケース生成においては，検証すべき振る舞いを自然言語で記述したテスト条件に基づき過去に開発したテストケースを抽出し，再利用することが一般的である．そこで，再利用するテストケースを特定するため，自然言語解析によりテスト条件間の類似度を評価し，言語的に最も類似したテスト条件を探索する手法が提案されている [19][20]．しかしながら，テスト条件が類似していてもテストケースは全く異なる場合には適用できない．

そこで本研究において，テストケースの類似度に基づきテスト条件の類似度を評価する手法を開発した．まず，テストケースが形式記述されていることに着目し，コードクローン解析を用いてテストケース間の類似度を評価する．次に，類似したテストケース同士をクラスタに集約する．さらに，各テストケースに紐づくテスト条件も同一のクラスタに集約し，テスト条件とクラスタの関係を機械学習する．最後に，クラスタ内部を解析しクラスタから抽出するテストケースを決定する．これにより，テスト条件が言語的に似ているが異なるテストケースを生成する必要がある場合においても，再利用すべきテストケースを抽出し効率的にテストケースを生成することが可能となる．

本論文の構成は以下のとおりである．2 章において，本研究の対象である HiLS を用いた統合テストを概説する．3 章において，関連研究である自然言語解析に基づくテスト条件の抽出について述べる．提案手法を 4 章で説明し，本手法を車載ソフトウェアのテストに適用した場合の評価結果を 5 章に示す．最後に結論を 6 章で述べる．

2. 研究対象

図 1 に本研究で対象とする Hardware-in-the-loop simulator (HiLS) を用いた車載ソフトウェアの統合テスト環境を示す．検討対象の車載ソフトウェアはマイコン (電子制御装置) 上に実装されており，HiLS とマイコンの外部端子が信号線と通信線を用いて接続されている．マイコンと HiLS の間で，デジタル・アナログ信号や通信メッセージを信号線や通信線を用いて入出力することによりテストが

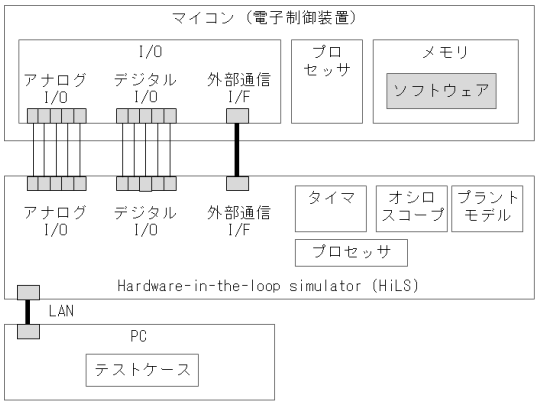

図 1 車載ソフトウェアの統合テスト環境.

実行される．これらの信号やメッセージの入出力に関する順序やタイミングを記述したものがテストケースとなる．

図 2 に形式記述したテストケースの例を示す．HiLS は

```
set(ch0, protocol=can);
set(ch1, type=output);]

wait(5sec);
set(ch1, DIGITAL_H);
prev = dummy;

for (i=0; i<20; i++)
  wait 5msec;
  new = read(rxBuf[ch0]);
  if (prev == new)
    return 1; // fail
  prev = new;

return 0; // pass
```

図 2 形式記述されたテストケースの例.

テストケースの記述に従い信号の入出力やメッセージ送受信を行うので，テストケースを自動的に実行することができる．

各テストケースは，そのテストケースにおいて検証しようとする振る舞いを記述したテスト条件に基づき開発される．テスト条件は，例えば*"Despite of short circuit, send messages at every 5ms."*のように，自然言語で記述される．これまでに開発したテストケースとテスト条件は過去テスト資産データベース内でトレーサビリティ管理されている．

図 3 に過去テスト資産データベースの概要を示す．テス

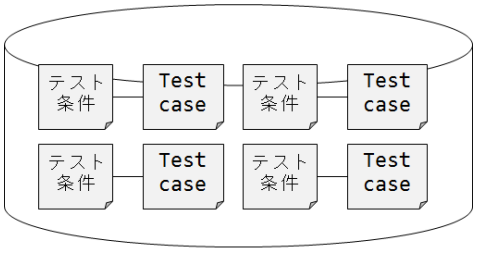

図 3 過去テスト資産データベースの概要.

トケースとテスト条件は互いに紐づけされ，各テスト条件を検証するために実行すべきテストケース，各テストケースが検証しようとするテスト条件がそれぞれ分かるように管理されている．

HiLS を用いた統合テストに要するコストを削減するためには，今回検証しようとするテスト条件に基づき，テストケースを効率的に生成することが必要である．

3. 関連研究

今回検証すべきテスト条件と全く同じものが過去テスト資産データベースに存在すれば，そのテスト条件に紐づけされたテストケースを取り出し再利用することにより，テストケースを効率よく生成することができる．しかしそのようなものが存在しない場合，過去テスト資産データベースの中から取り出すべきテストケースを何らかの手法により決定する必要がある．

過去研究において，自然言語間の類似度を評価するさまざまな手法が提案されている．これらの手法を用いて過去テスト資産データベースに保存されたテスト条件のうち今回検証すべきテスト条件に最も近いものを同定し，そのテスト条件に紐づけされたテストケースを取り出すことが考えられる．例えば，文献 [19] において，ユーザのさまざまな質問を分類し適切に応答するチャットボットの開発のための手法が提案されている．この手法によれば，構文解析と意味情報を用いて各文章をベクトル表示することにより，文章間の意味的類似性を数値的に評価することが可能となる．ただし，これまでに開発した膨大な量のテスト条件それぞれに対し意味情報を付加することはコストの点から現実的ではない．そこで文献 [20] において，意味情報の付加を不要とした手法が提案されている．これは，自然言語で記述された文章に対し，文章内に含まれる各単語の文法的役割をアノテーション形式でタグ付けすることにより，文章の意味を解釈可能とするものである．アノテーションを分析し互いの関連性をネットワークグラフに表現することで，ある文章に対して意味的に最も類似した文章を同定することが可能となる．

しかしながら，テスト条件同士を直接比較する手法はテストケースの効率生成に不十分である．なぜなら，２つのテスト条件が自然言語として意味的に類似していてもテストケースが類似しているとは限らないからである．テストケースが類似していなければ，過去テスト資産データベースから取り出したテストケースの多数の箇所を修正する必要が生じるため，多くのテストケース作成工数を要することとなる．すなわち，テスト条件同士の類似度の評価だけでは過去テスト資産データベースの中から適切なテストケースを取り出すことができず，テストケースの生成を効率的に行うことができない．

以上より，与えられたテスト条件に対し，過去テスト資産データベースの中から少ない修正行数で今回作成すべきテストケースを作成可能なテストケースを抽出することが課題である．

4. 提案手法

そこで本研究において，テストケースをコードクローン解析してテストケース間の類似度を算出し，テストケース間の類似度をテスト条件間の類似度として対応付ける手法を提案する．本手法において，下記に示す分類器と決定器を生成する．

分類器 過去テスト資産データベースに含まれるテストケースを共通するコードクローンを持つもの同士で集

約し，テストケースクラスタとして分類する．

決定器　クラスタに含まれるテストケース同士の類似度を詳細解析し，最も多くのテストケースと互いに類似したテストケースをひとつ決定する．

分類器と決定器を用いて，今回検証すべきテスト条件に対し，過去テスト資産データベース内から抽出するテストケースを決定するフローを図 4 に示す．分類器において今

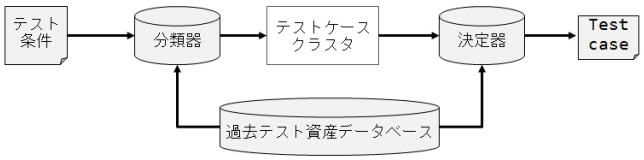

図 4　提案手法の概要．

回検証すべきテストケースがどのテストケースクラスタに該当するか判別し，決定器において該当するテストケースクラスタの中からひとつのテストケースを決定する．次節以降，分類器と決定器についてそれぞれ詳細を述べる．

4.1　分類器の生成

本節において，分類器の生成手法を説明する．分類器はテスト条件を入力とし 1 つのテストケースクラスタを出力する．図 5 に分類器の生成フローを示す．過去テスト資産

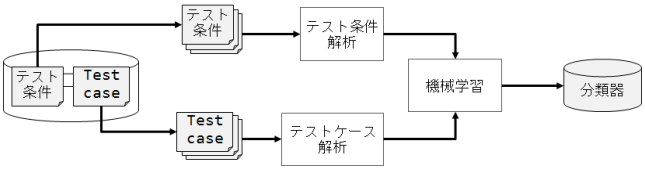

図 5　分類器の生成フロー．

データベースに含まれるテスト条件とテストケースをそれぞれ解析し，それらを機械学習することにより分類器を生成する．

4.1.1　テスト条件解析

テスト条件解析において，自然言語で記述されたテスト条件を機械学習可能な数値ベクトルデータに変換する．

まず，テスト条件に BOW(Bag-of-Words) と呼ばれる手法を適用する．これは，テスト条件に含まれる各単語の出現頻度をベクトルデータで表現するものである．例えば，表 1 に示すテスト要件が与えられた場合に BOW を適用した結果を表 2 に示す．テスト条件”rxBuf *is set to 1 and*

表 1　テスト条件の例．

条件 1	rxBuf is set to 1 and ANALOG_INPUT is 0V ···
条件 2	ANALOG_INPUT is 0V before sending a msg ···
⋮	⋮

ANALOG_INPUT *is 0V* ··· ”は [1 1 1 1 0 ···] なる数値

表 2　BOW の適用結果の例．

	rxBuf	set	ANALOG_INPUT	0V	before	···
条件 1	1	1	1	1	0	···
条件 2	0	0	1	1	1	···
⋮	⋮	⋮	⋮	⋮	⋮	⋱

ベクトルデータに変換される．過去テスト資産データベースに保存されているすべてのテスト条件に対して BOW を適用する．

すべてのテスト条件に BOW を適用すると，数値ベクトルデータに含まれる要素数は各テスト条件に含まれるすべての単語の種類数に一致することとなり，このまま機械学習を行うと計算コストが高い．このとき，各数値ベクトルデータは 0 を多く含む整数値からなるスパース (sparse) なデータとなっている．そこで，主成分分析による次元削減を行う．図 6 に主成分分析を行った例を示す．この場合，

図 6　主成分分析の例．

多数の整数値からなる数値ベクトルデータが 4 つの実数値からなる主成分を用いて表現される．合成する主成分の数は実験的に定める．

4.1.2　テストケース解析

テストケース解析において，テストケース間の類似度を解析し，互いに類似したテストケースを集約したテストケースクラスタを生成する．

図 2 に示したように，本研究で対象とするテストケースは HiLS を用いて自動実行可能とするため形式記述 (コード化) されている．したがって，コードクローン解析技術を用いることによりテストケース間の類似度を定量的に評価することができる．

コードクローン解析においては，コード中の類似または一致した部分（コードクローン）を検出する．コードをトークンと呼ばれる要素に分解し，2 つのコードを比較して一定数（ミニマムトークン数）以上連続して同様のトークン構成である場合に 2 つのコードの該当箇所が類似（一致）していると判定される．ゆえに，コードクローン解析においてはミニマムトークン数の設定が重要である．2 つのコードをトークンに分割した例を図 7，図 8 に示す．この 2 つのコードは共通して `int main ( void )` を持ち，

```
int main ( void ) {
a = 5 ;
}
```

図 7　トークンの例 (1)．

```
int main ( void ) {
print ( " Start Program " ) ;
}
```

図 8　トークンの例 (2)．

その直後の a と print は異なるトークンとなるため，連続したトークン構成は 6 である．したがって，ミニマムトークン数を 6 以下に設定すると 2 つのコード間にはコードが類似していると判定され，7 以上であれば類似していないと判定される．ミニマムトークン数は実験的に定める．

コードクローン解析により検出したテストケースの類似（一致）部分に基づき，テストケース間の類似度を算出する．テストケース間の類似度は以下の定義式を用いて算出する．

$$\text{類似度}_1(2) = \frac{\text{テストケース 1・2 の間で検出したコードクローンの行数}}{\text{テストケース 1 の総コード行数}}$$

ここで 類似度$_1(2)$ とは，テストケース 1 におけるテストケース 2 との類似度を表す．例えば，テストケース 1（総テストコード行数:40）とテストケース 2（総テストコード行数:30）において図 9 に示すコードクローンが検出されたとき，類似度は下記のように算出される．

- テストケース 1 に含まれる 30 行がテストケース 2 との間のコードクローンであることから

$$\text{類似度}_1(2) = \frac{30}{40} = 0.75,$$

- テストケース 2 に含まれる 20 行がテストケース 1 との間のコードクローンであることから

$$\text{類似度}_2(1) = \frac{20}{30} = 0.67.$$

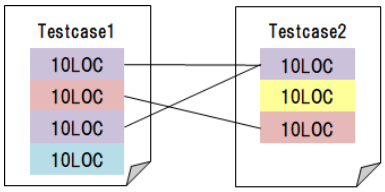

図 9 テストケース間で検出したコードクローンの例.

過去テスト資産データベースに含まれるすべてのテストケースに対して互いの類似度を計算し，表 3 に示すテストケース類似度テーブルを生成する．

表 3 テストケース類似度テーブル.

	Test case 1	Test case 2	Test case 3	···
Test case 1	1	類似度$_1(2)$	類似度$_1(3)$	···
Test case 2	類似度$_2(1)$	1	類似度$_2(3)$	···
Test case 3	類似度$_3(1)$	類似度$_3(2)$	1	···
⋮	⋮	⋮	⋮	⋱

テストケース類似度テーブルに含まれる各類似度をテストケース間の擬似的な距離とみなし，距離の近いテストケース同士をクラスタリングする．いくつかのクラスタリングアルゴリズムが存在するが，本手法においては VBGMM（Variational Bayesian Gaussian Mixture）を用いる．これは，クラスタ数を与えずにデータをクラスタリングする一般的な手法である．

以上より，各テストケースの類似度を定量化し，類似度の高いテストケース同士をクラスタに集約することができる．

4.1.3 機械学習

4.1.1 節で生成したテスト条件解析データと 4.1.2 節で生成したテストケースクラスタの関係性を機械学習し，分類器を生成する．

まず，クラスタリング結果に基づき，各テストケースがどのクラスタに属するかを示したテストケースクラスタテーブルを生成する．表 4 にテストケースクラスタテーブルの例を示す．

表 4 テストケースクラスタテーブルの例.

テストケース	所属クラスタ番号
Test case 1	クラスタ 0
Test case 2	クラスタ 1
Test case 3	クラスタ 1
Test case 4	クラスタ 2
⋮	⋮

次に，過去テスト資産データベースを参照し，テストケースとテスト条件の紐付け関係に基づき，テストケースクラスタテーブルにおけるテストケース列をテスト条件に置換したテスト条件クラスタテーブルを生成する．例えば，テストケース 1 がテスト条件 1，テストケース 2 がテスト条件 2，···，テストケース i がテスト条件 i，··· のように紐づけされており，テストケースクラスタテーブルが表 4 であるとき，テスト条件クラスタテーブルは表 5 のとおりとなる．テストケース 1 がテストケースクラスタ 0 に所属するのでテスト条件 1 はテスト条件クラスタ 0 に所属し，テストケース 2 がテストケースクラスタ 1 に所属するのでテスト条件 2 はテスト条件クラスタ 1 に所属する．

表 5 テスト条件クラスタテーブルの例.

テスト条件	所属クラスタ番号
条件 1	クラスタ 0
条件 2	クラスタ 1
条件 3	クラスタ 1
条件 4	クラスタ 2
⋮	⋮

続いて，テスト条件クラスタテーブルにおける各テスト条件をテスト条件解析（4.1.1 節）にて生成した各テスト

条件の数値ベクトルデータへの変換結果に置換した機械学習データテーブルを生成する．表 6 は，テスト条件解析において，テスト条件 1 が [1.34 1.62 5.73 2.65]，テスト条件 2 が [3.13 0.51 8.32 5.10]，··· に変換された場合における機械学習データテーブルを示している．

表 6 機械学習データテーブルの例.

テスト条件数値ベクトルデータ（入力ベクトルデータ）	所属クラスタ番号（正解ラベル）
[1.34 1.62 5.73 2.65]	クラスタ 0
[3.13 0.51 8.32 5.10]	クラスタ 1
[4.14 3.69 0.03 4.22]	クラスタ 1
[3.08 2.18 0.54 0.27]	クラスタ 2
⋮	⋮

機械学習データテーブルにおけるテスト条件の数値ベクトルデータを入力ベクトルデータとし，所属クラスタ番号を正解ラベルとした教師あり学習を行う．今回は，学習モデルとして SVM(Support Vector Machine) を用いた．

最後に，今回検証すべきテスト条件を分類器に入力する方法を説明する．過去テスト資産データベースに保存されたテスト条件と同様に，今回検証すべきテスト条件は自然言語で記述されているため数値ベクトルデータに変換してから入力する必要がある．したがって，今回検証すべきテスト条件に対し，テスト条件解析（4.1.1 節）における BOW および主成分分析を同様に適用する．例えば，今回検証すべきテスト条件が”txBuf *is set to 1 and* ANALOG_INPUT *is 0V* ···である場合，表 7 より，[0 1 1 1 0 ···] なる数値ベクトルデータに変換される．なお，表 7 でカウントしている単語およびその順序（すなわち表の列見出し）が表 2 に一致していることに注意されたい．この数値ベクトルデータを図 6 に示した主成分分析器に入力し，主成分分析器から出力された数値ベクトルデータを SVM に入力する．図 10 に，今回検証すべきテスト条件の入力からテストケースクラスタを決定するまでのフローの概要をまとめた．

表 7 今回検証すべきテスト条件に対する BOW の適用結果の例.

	rxBuf	set	ANALOG_INPUT	0V	before	···
今回	0	1	1	1	0	···

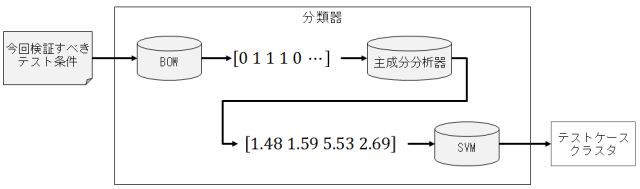

図 10 テストケースクラスタの決定フロー.

これまでに説明したとおり，分類器は，テスト条件の数値ベクトルデータを直接比較するのではなく，そのテスト条件に紐づくテストケースのクラスタリング結果に基づきテスト条件を同様にクラスタリングするものである．したがって分類器を用いることにより，テストケース間の類似度をテスト条件間の類似度に対応付けることができる．

4.2 決定器の生成

本節において，決定器の生成について説明する．決定器はテストケースクラスタの中から抽出するテストケースを１つ決定するものであり，以下の２つのステップで構成される．

(1) テストケースクラスタごとにテストケースグラフを生成する．
(2) テストケースグラフの次数中心性を解析し，もっとも次数の高いノードを抽出する．

図 11 に，テストケースグラフの例を示す．テストケースグラフは，テストケースクラスタに含まれる各テストケースをノード，テストケース間の類似関係をリンクで表現したグラフである．図 11 の場合，テストケース A とテストケース B の間はリンクで接続されているため互いに類似を表しており，テストケース A とテストケース D の間はリンクで接続されていないため互いに類似していないことを表している．２つのテストケース i とテストケース j の間の類似関係を定義するため，表 3 に示したテストケース類似度テーブルを用いる．ある閾値に対し，類似度$_i(j)$ または 類似度$_j(i)$ の少なくとも一方が閾値を超えていれば２つのテストケース i とテストケース j が互いに類似していると判定し，そうでない場合は互い類似していないと判定する．すべての類似関係の定義は実験的に定めた同一の閾値に基づき行う．

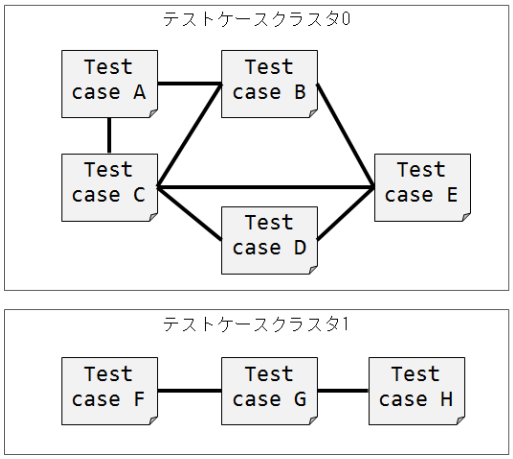

図 11 テストケースグラフの例.

次に，テストケースグラフにおける次数中心性を解析し，テストケースクラスタの中で最も次数の高いノードに該当するテストケースを抽出する．グラフにおける中心性と

は，各ノードの重要性を評価するための指標であり，本研究では次数中心性について解析する．次数とは各ノードに接続されているリンクの数であり，次数中心性とはより多くのリンクを持つノードを高く評価する指標である．テストケースグラフのリンクはノード間の類似度により接続されるため，次数の高いノードは多くのノードと類似していることを示す．例えば，分類器によりテストケースクラスタ 0 が選択され，テストケースグラフが図 11 のように生成されたとする．このとき，今回生成すべきテストケースはテストケース A〜E のいずれかのテストケースに類似していると推定される．ここで，次数の最も高いテストケース C を選択することにより，もし真に最も類似しているテストケースがテストケース E であったとしても，テストケース C と E は互いに類似しているので，少ない工数でテストケースを生成することができる．なお，真に最も類似しているテストケースはテストケースが生成された後はじめて判別可能であることに注意されたい．したがって，テストケースクラスタにおいて次数の最も高いテストケースを探索することにより，テストケースの中から抽出すべきテストケースをひとつ合理的に決定することができる．

5. 評価

本章において，提案手法を評価した結果を述べる．提案手法の評価は，分類器・決定器により抽出されたテストケースと実際に開発されたテストケースの間の類似度を評価することにより行った．図 12 に，評価実験のプロセスの概要を示す．実験手順は次のとおりである．

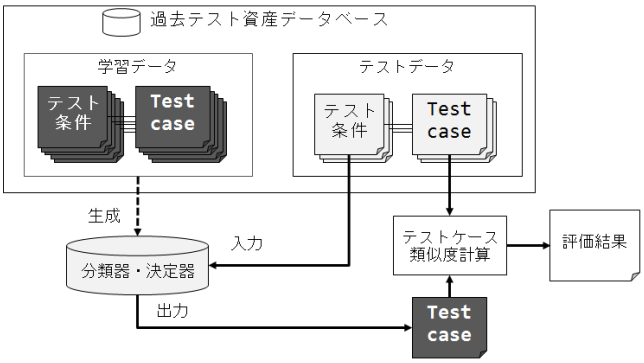

図 12 評価実験のプロセス概要.

(1) 過去テスト資産データベースに含まれるテスト条件とテストケースを学習データとテストデータに分割する．
(2) 学習データを用いて分類器・決定器を生成する．
(3) 分類器・決定器にテストデータに含まれる各テスト条件を入力する．
(4) 分類器・決定器から出力されたテストケース（以降，出力テストケースと呼ぶ）と，評価データのテスト条件に紐付いたテストケース（以降，正解テストケースと呼ぶ）との間の類似度を計算する．

本評価実験においては，分類器・決定器に入力されるテスト条件が「今回検証すべきテスト条件」に対応し，正解テストケースはそのテスト条件に対して「実際に開発されたテストケース」である．そのため，出力テストケースのコードの中で，正解テストケースに含まれているコードについては再利用することができる．また，出力テストケースと正解テストケースの間の類似度は以下の式に基づき算出する．

$$\text{各出力テストケースの類似度} = \frac{\text{出力・正解テストケースの間のコードクローン行数}}{\text{正解テストケースのコード行数}}$$

本評価実験においては，過去テスト資産データベースとして，通信関係の車載ソフトウェアモジュールを検証するテスト条件とテストケースを用いた．実験パラメータを表 8 に，実験結果を表 9 に示す．

表 8 実験パラメータ一覧.

テスト条件の合計数	117
テストケースの合計数	117
学習データのサイズ	107
テストデータのサイズ	10
主成分数	10
ミニマムトークン数	50
テストケース類似度閾値	90%

表 9 実験結果.

	出力・正解テストケース間のコードクローン行数	正解テストケースの総コード行数	類似度
1	1543	1544	100%
2	1546	1739	89%
3	48	268	18%
4	434	510	85%
5	411	522	79%
6	641	798	80%
7	904	1280	71%
8	558	654	85%
9	558	654	85%
10	186	657	28%
合計	6829	8626	79%

各出力テストケースの類似度の最小値は 18%，最高値は 100%となった．また，出力テストケース 10 件全体と正解テストケース 10 件全体で類似度を評価すると，出力テストケースにより合計 6829 行（79%）の修正が不要となり，残り 1797 行（21%）の修正が必要であることがわかる．テストケース生成工数はテストコード行数に比例するため，テストケース生成工数を 79%削減可能と見込まれる．

各出力テストケースの類似度を分析すると，テストデータ 3 と 10 で類似度が低い値となっているため，追加の解析を行った．図 13 に，テストデータと学習データの類似

度のヒストグラムを示す．

類似度の高いテストケースを抽出したテストデータ 1 および 2 では学習データの中に類似度 0.8 以上のテストケースが複数含まれているのに対し，類似度の低いテストケースを抽出したテストデータ 3 および 10 ではそのようなテストケースがほとんど含まれていないことが分かる．本提案手法は，過去に開発したテストケースから再利用するためのテストケースを決定するものである．したがって，本手法を適用するためには，類似したテストケースが過去に複数回開発されており，それらを学習しておく必要がある．

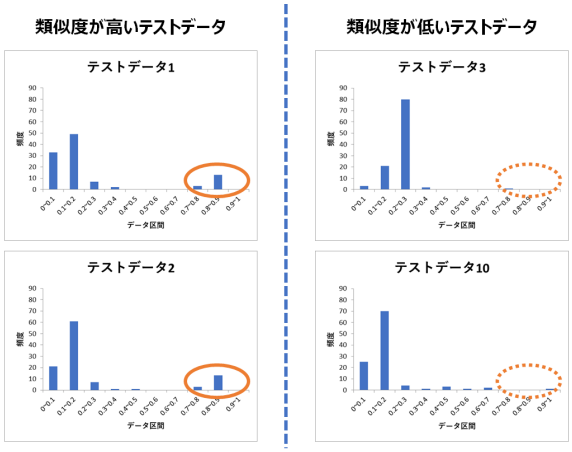

図 13 テストデータと学習データの類似度．

6. おわりに

車載ソフトウェア開発工数の多くを占める HiLS を用いた統合テストの効率化のため，過去に開発したテストケースを再利用しテストケースを効率的に生成することが求められている．したがって，与えられたテスト条件に対し，過去テスト資産データベースの中から少ない修正行数で今回作成すべきテストケースを作成可能なテストケースを抽出する必要がある．既存手法によればテスト条件間の自然言語解析により最も言語的意味の類似したテスト条件を探索することが可能であるが，テスト条件同士の言語的意味が類似していても必ずしもテストケースが類似しているとは限らない．そこで本研究において，テストケースをコードクローン解析してテストケース間の類似度を算出し，テストケース間の類似度をテスト条件間の類似度として対応付ける手法を提案した．まず，共通するコードクローンを持つテストケースをクラスタに集約する．次に，同一クラスタに含まれる各テストケースに紐づくテスト条件は同一クラスタに属するとみなしてテスト条件をクラスタリングする．最後に，クラスタ内部を解析してクラスタから抽出すべきテストケースを決定する．提案手法を車載ソフトウェア向けテストケースで評価し，テストケース生成工数を 79%削減可能な見込みを得た．パラメータ最適化による精度向上と BOW・主成分分析以外のテスト条件の解析手法の比較検討が今後の課題である．

参考文献

[1] Kiencke, U. and Nielsen, L.: Automotive control systems: for engine, driveline, and vehicle (2000).
[2] Luo, J., Pattipati, K. R., Qiao, L. and Chigusa, S.: An integrated diagnostic development process for automotive engine control systems, *IEEE Tran. on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, Vol. 37, No. 6, pp. 1163–1173 (2007).
[3] Lin, C.-M. and Hsu, C.-F.: Self-learning fuzzy sliding-mode control for antilock braking systems, *IEEE Tran. on Control Systems Technology*, Vol. 11, No. 2, pp. 273–278 (2003).
[4] Peng, D., Zhang, Y., Yin, C.-L. and Zhang, J.-W.: Combined control of a regenerative braking and antilock braking system for hybrid electric vehicles, *Int'l J. of Automotive Technology*, Vol. 9, No. 6, pp. 749–757 (2008).
[5] Kim, J.-H. and Song, J.-B.: Control logic for an electric power steering system using assist motor, *Mechatronics*, Vol. 12, No. 3, pp. 447–459 (2002).
[6] Geronimo, D., Lopez, A. M., Sappa, A. D. and Graf, T.: Survey of pedestrian detection for advanced driver assistance systems, *IEEE Tran. on pattern analysis and machine intelligence*, Vol. 32, No. 7, pp. 1239–1258 (2009).
[7] Paul, A., Chauhan, R., Srivastava, R. and Baruah, M.: Advanced driver assistance systems, Technical report, SAE Technical Paper (2016).
[8] Takei, C., Takada, H., Yamamoto, M. and Honda, S.: Integrated software platform for automotive systems, *Int'l SoC Design Conference*, IEEE, pp. 377–379 (2009).
[9] Harrold, M. J.: Testing: a roadmap, *Proc. of the Conf. on the Future of Software Engineering*, pp. 61–72 (2000).
[10] Boehm, B. W. and Papaccio, P. N.: Understanding and controlling software costs, *IEEE Tran. on software engineering*, Vol. 14, No. 10, pp. 1462–1477 (1988).
[11] Beck, K., Gamma, E., Staff, D. and Clark, M.: Junit 5.
[12] ETSI: TTCN-3.
[13] dSPACE: AutomationDesk.
[14] Vector: vTESTstudio.
[15] ETAS: LABCAR-AUTOMATION.
[16] Bringmann, E. and Krämer, A.: Model-based testing of automotive systems, *1st Int'l Conf. on Software Testing, Verification, and Validation*, IEEE, pp. 485–493 (2008).
[17] Galloway, A. J., Cockram, T. J. and McDermid, J. A.: Experiences with the application of discrete formal methods to the development of engine control software, *IFAC Proceedings*, Vol. 31, No. 32, pp. 49–56 (1998).
[18] Dhadyalla, G., Kumari, N. and Snell, T.: Combinatorial testing for an automotive hybrid electric vehicle control system: a case study, *IEEE 7th Int'l Conf. on Software Testing, Verification and Validation Workshops*, IEEE, pp. 51–57 (2014).
[19] Virkar, M., Honmane, V. and Rao, S. U.: Humanizing the Chatbot with Semantics based Natural Language Generation, *Int'l Conf. on Intelligent Computing and Control Systems*, pp. 891–894 (2019).
[20] Taimoor Hassan, Shoaib Hassan, Muhammad Asfand Yar, Waleed Younas: Semantic analysis of natural language software requirement, *2016 6th Int'l Conf. on Innovative Computing Technology*, IEEE, pp. 459–463 (2016).