記号実行を用いたテストデータ自動生成の試行評価

Evaluation of Test Data Generation Using Symbolic Execution

株式会社デンソー 電子技術 3 部
DENSO CORPORATION　Electronics Eng. Div.3
榎本　秀美
Hidemi Enomoto

Abstract　This paper describes the evaluation of test data generation using symbolic execution. For decreasing omissions of test cases and software developing cost reduction, demand for automating in test increases. In order to generate test data automatically to satisfy MC/DC coverage to contribute to efficiency of the test, we evaluated the symbolic execution based tool and static analysis based tool using our software. We confirmed source code patterns that the symbolic execution based tool could generate test data to satisfy MC/DC coverage automatically.

1.　はじめに

自動車系の組込みソフトウェアは，燃費改善などの環境機能や，自動ブレーキなど走行安全に関する機能など，様々な開発が行われており，規模の増加と複雑化が急速に進んでいる．これに対して開発手法は，派生開発の充実だけでなく，新たな技法としてモデルベース開発の導入など積極的な改善が行われている．品質に対しても，システムレベルではシミュレーション技術の進歩により大きな改善が行われている．

一方で，メソッドや小さな関数群に対する単体テストに問題が生じている．現在では，一つのライブラリに数千の関数が存在することも珍しくない．さらに近年では機能安全に基づく詳細なテスト要件に対しエビデンスを残す作業が追加されている．数千に及ぶメソッドや関数に対し，これまでのように人手に頼ったテストを続けていては，テストケースの抜け漏れやコスト増加といった懸念が考えられることから，テストの自動化が求められている[1]．

我々のソフトウェア開発においても，機能安全のテスト要件を満たすテストの自動化を進めている．テストの自動化のためにテストツールベンダは様々な機能を持った製品を提供しており，その中でもテストデータの自動生成機能はテスト効率化に大きく貢献している．しかし市販ツールでは MC/DC カバレッジを満足しないケースが多く課題となっている．本稿では，この課題に対して分析を行い，対策の試行評価を行ったので報告する．

テストデータの自動生成は，ソースコードを基に解析を行うコードベーステスト又は実装ベースのテストと呼ばれ，古くから研究開発が行われている．その方式は大きく「静的解析」と「記号実行」の 2 種類がある．静的解析とは，コードを実行することなく制御パスを解析し，そのパスを通過するテストデータを生成する．記号実行は，静的解析の欠点である動的な変数を用いた判断文の解析が出来ないことを改善するために開発されたもので，コードを実行しながらテストデータを生成する[2]．市販されているツールは静的解析を用いているため，動的な変数を使ったコードに対して MC/DC カバレッジを満足できない．これは静的解析の限界であり，対策としては，記号実行に基づくツールの導入が考えられる．

記号実行は，まだ研究段階でありオープンソースのツールはあるが操作性の行き届いた製品は見当たらないため，ソフトウェア開発の現場で導入するための工夫や付随設定を行い評価が必要

株式会社デンソー　電子技術 3 部
Electronics Eng. Div.3, DENSO CORPORATION

愛知県刈谷市昭和町 1−1　Tel: 0566-61-7825　e-mail:hidemi_enomoto@denso.co.jp
1-1, Showa-cho, Kariya-shi, Aichi-ken, 448-8661, Japan

となる．本稿では，記号実行の一種である「Concolic testing」を用い試行評価を行った．他にも Concolic testing を用いたツールを使ったテスト効率化を行う事例研究が進んでいる[3,4,5]．

我々の試行評価は，実際の製品ソフトウェアからサンプリングした関数 40 個に対し，先ず，静的解析に基づくツール（市販ツール B）でテストデータの生成とカバレッジ評価を行った結果，MC/DC カバレッジを満足した関数は 9/40 個（23%）であった．次に，静的解析にモンテカルロ法による機能を加えたツール(市販ツールC)を用いたところ29/40個(73%)であった.Concolic testing を用いたツールでは36/40個(90%)となり,残り4個は解析の結果,冗長なコードが含まれており，静的な MC/DC カバレッジ対象外であった．つまり，Concolic testing により，可達パスに対するカバレッジは達成できる目処がついた．

2 章では，単体テストの現状と問題点を明確にし，その対策について説明する．3 章では，評価環境について述べ，4 章では評価結果を述べる．最後に 5 章では，考察と今後の改善点を述べる．

2. 組込みソフトウェアにおける単体テストの分析

2.1 単体テストの問題点

組込みソフトウェアでは，実際に製品が動作するターゲットマシンと，製品を開発する段階でコンパイルや単体テストを行う開発マシンの 2 系統を持つ特徴がある.一般的に，ターゲットマシンの環境は，ビット幅が少なく，メモリなど資源の制約があり，デバッグ機能が貧弱なことから，単体テストなど詳細なテストを行うのは困難である．そこで，開発環境のオブジェクトコードに変換してその環境でテストを行うか，あるいはターゲットマシンのシミュレータを用いてテストを行う方法（開発マシンでテストを行う方法）を併用している．本稿で扱う単体テストも開発マシンを利用したものである．

対象とする組込みソフトウェアの構造は多種多様であるが，ここでは，ヘッダーファイルと本体と関数群（ライブラリ）に分けて考える．単体テストは関数群に対して行うため，テストドライバとスタブが必要となる．関数群の参照関係はライブラリに閉じているが，関数の中にはレジスタの読み書きなどターゲットマシンとのインターフェースを持つものがある．よって開発環境において単体テストを行うには，ターゲットマシンをエミュレートするスタブを個別に設計する必要がある．単体テストの手順としては，次の手順で行っている．

①テストデータと期待結果の抽出
　仕様書を基に，テストデータと期待結果を設計し，テストケースとする．
②テストドライバとスタブの設計
③テストの実行と網羅度測定
　開発環境においてテストを実行し，その出力結果が期待結果と一致していることを確認する．当該単体テストによる網羅をツールによって確認する．網羅の基準としては，C0，C1，MC/DC カバレッジなどがある．
④未網羅箇所に対する追加テスト
　③の結果，未網羅であれば，追加テストにより確認を行う．

単体テストの問題として，網羅基準を MC/DC カバレッジとした場合，①で示した仕様書を基にしたテスト設計（一般的には仕様ベースのテストと呼ばれている）では，③において MC/DC カバレッジを達成できない場合が生ずる．そのため，④の追加テストの活動が増加する．追加テストの多くは，テストデータを仕様書から導くのが困難であり，関数のソースコードとヘッダーファイルの情報を熟知する必要がある．つまり，仕様ベースのテストでは対応できない．対策

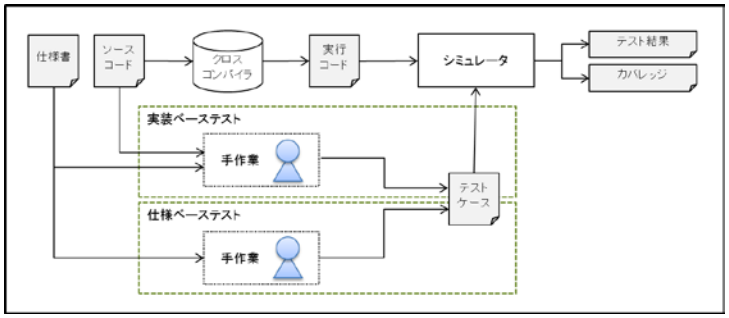

図 1　現行の単体テスト方法

としては，実装ベースのテスト（コードベーステストとも呼ばれる）を用いることになる（図 1）．実装ベースのテストは，ヘッダーファイルやソースコードの情報を利用して対象とする関数が持つ可達パスを洗い出し，テストデータを作成するテストである．期待結果は，洗い出した可達パスの仕様上の対応を調べて求める．

実装ベースのテストにおいてテストデータを自動生成するツールの導入が有効であり，既に導入を開始しているが，市販されているツールでは MC/DC カバレッジを達成できないケースが多く生じ課題となっている．カバレッジ未達の場合，不足するテストデータを人手により追加作成する必要があり，テストの効率化の妨げとなる．

2.2 対策案に向けて

近年，実装ベースのテストをサポートするツール，すなわち，テストデータの自動生成の研究が盛んになり，その研究成果を取り入れたツールが数多く存在する．その代表的な方法としては，以下の 3 つの方式がある[6,7,8]．(2)と(3)は原理的には同じであるが実装方式が異なる．

(1)静的解析（Static analysis）

古くから存在する方式で，多くのツールが存在する．ソースコードから制御フローを静的に解析し，可達パスを探索する．解析は，コンパイラの技術を用いている．静的解析では，配列やポインタなど動的なメモリを用いた制御を解析することが困難である．制御パスから，その制御パスを通過する入力データの組合せを得る方法は，Pathwise 法と呼ばれる手法が用いられている．制御パスの制限と同様，動的なメモリを使ったデータフローを解くことはできない．

(2)記号実行（Symbolic Execution）

(1)静的解析の制限を解決する方法として研究が行われている．ソースコードを中間言語に変換し，シミュレーションなどで順次実行しながら可達パスを探索するため，動的なメモリを使った場合についても制御フローを解析することができる．最新技術であり，多くのツールが公開されている．例えば，JAVA を対象とした NASA の JPF，.NETFramework を対象とした Microsoft の Pex，C 言語を対象とした KLEE などが知られている．記号実行は，古くから提唱されていたが，性能上の理由で実際には使われていなかった．近年，JAVA のように JAVA バイトコードを仮想マシン上で機械語と同程度のスピードで実行できる技術が完成したことから，実用化に至った．同様に，C 言語についても，LLVM プロジェクトによる C プログラムをビットコード（中間言語）に変換し，仮想マシンで実行できるようになったことから実現した．制御パスから，その制御パスを通過する入力データの組合せを得る方法は，制御プログラミング（Constraint Program）と呼ばれる変数間の関係を制約として記述し，制約ソルバ（SAT ソルバ）を用いて解を得る．制約ソルバは，一部ヒューリスティックであるが，他の方法では解けない解を得られる．ヒューリスティック（heuristic）とは，必ずしも正しい答えを導けるわけではないが，ある程度のレベルで正解に近い解を得ることができる方法である．

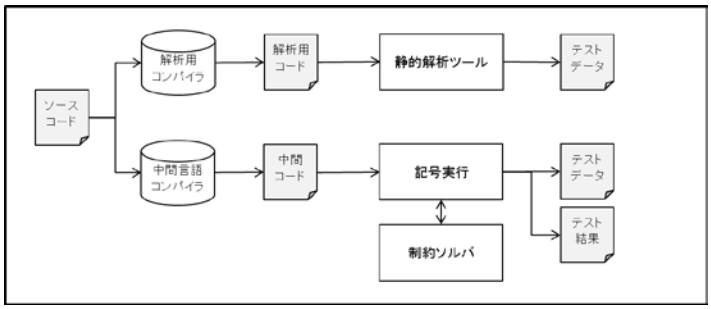

図 2　静的解析（上）と記号実行（下）の構成

(3)Concolic testing

(2)記号実行の一種であるが，記号実行の欠点である実行時間が改善される．ツールとしては，CREST が公開されている．記号実行を更に高速に実行するためのアイディアとして，判断文以外はシミュレーションを行わない方式である．用語「Concolic」は，concrete と symbolic を合成した用語である．concrete とは，concrete execution のことで，SAT ソルバによる制約ではなく，具体的な実テストデータを用いたテストを意味する．Concolic testing とは，

SAT ソルバが不得意とする制約に到達した場合，その箇所だけ実際に実行することで具体値を代入する．(2)記号実行との能力差は，ヒューリスティックな解を求める速度が早いことである．

調査の結果，実装ベースのテストデータ生成方式として幾つかあることが判ったので，我々の環境において複数のツールの評価を行い，どのようなツールをどのように用いるのか（単独か連携かを含めて）を評価し，適切な使い方を探る．

3. 対象としたツールと評価環境

3.1 対象ツール

実装物（ヘッダーファイルやソースコード）からテストデータを自動的に生成する方法は，前章で示した 3 つの方式がある．基本的な機能は，プログラムに存在する制御パスを見つけ出す能力と，見つけた制御パスを通過させる入力データのセットを求める能力である．

本稿では，3 つのツールを使って評価を行った．3 つのツールとは以下のものである．市販ツールであるツール B，ツール C は，我々のソフトウェア開発において使用実績があることから選定し比較を行う事とした．

＜ツール A＞

(2)記号実行は(3)Concolic testing に包含されるとことと，実行性能面で有利なことから(3)Concolic testing 方式ツールとした．その中で我々の環境で使い易いオープン系ツールである「CREST」を選択した[9]．CREST が使い易い理由は，一般的な C のライブラリを使えることである．CREST で扱えるテストデータ生成の対象となる変数は，符号付/なしの char，short，int である．テストの準備として，crest.h をインクルードし，テストデータを生成する変数を CREST に知らせるマクロで書く．テストデータを生成するための解析方法として，以下の実行時オプションがある．

- -dfs（Bounded Depth First Search）：主にソルバの性能を高めるオプション
- -random（Random Search）：原始的なヒューリスティック（モンテカルロ法）
- -random_input（Random Input Search）：入力に着目したヒューリスティック法
- -cfg（Cfg Heuristic Search）：最適化したヒューリスティック法
- -cfg_baseline （Cfg Baseline Search）：-cfg の改良版
- -hybrid（Hybrid Search）：ソルバとの組合せ
- -uniform_random（Uniform Random Search）：一様乱数によるヒューリスティック法

多様な解析方法が用意されているのは，SAT ソルバで解けない場合，ヒューリスティック探索を行うことになり，その探索アルゴリズムを指定できる．利用者は自分たちのプログラムの特性に合ったオプションを試すことができる．

＜ツール B＞

組込み系テストベッドと親和性の高い市販ツールである．原理は，(1)静的解析である．原理的に実行時に割り当てられる動的変数に対してデータを生成することはできない．関数単位でテストデータを生成する場合は，関数の引数や外部変数を自動的に調べ，静的解析を行い，テストデータを生成する．

＜ツール C＞

モデルベース開発と親和性の高い市販ツールである．モデルから自動生成されるソースコードを意識したテストデータの自動生成機能を有している．但し，今回の評価はモデルベース開発ではない．原理は(1)静的解析に基づき，テストデータ生成は，ランダム生成（モンテカルロ法）を中心に行い，実行の結果，カバレッジが達成されない場合には，独自の手法を使って探索を試みる．

3.2 評価環境

評価に当たり，テストデータ生成とテストによるカバレッジ測定を分離して実施した（図 3）．

カバレッジ測定は，ツール B と連携しているシミュレーション環境を用いた．評価にこの環境を用いたのは，我々のソフトウェア開発において多くのプロジェクトで使われ実績があることからである．

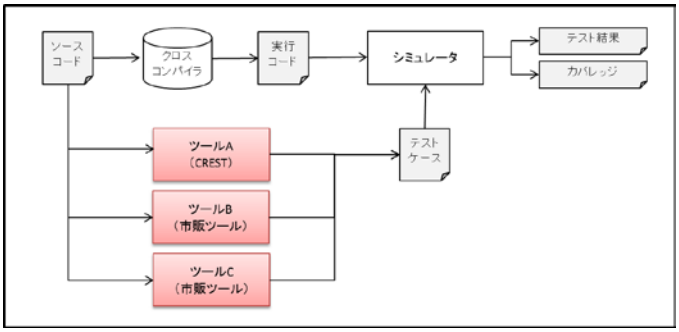

図 3　評価環境

```
#include <crest.h>
#include <stdio.h>
#include <stdlib.h>
unsigned char In1; /* 1 or 0 */
unsigned char In2;
unsigned char Out1;
int main ( void ) {
  CREST_unsigned_char( In1 );
  CREST_unsigned_char( In2 );
  if ( In1 <= 1 ) {
    printf( “%d %d ¥t¥t”, In1, In2 );
    TestFunc();
    printf( “%d ¥n”, Out1 );
  }
  return EXIT_SUCCESS;
}
```

図 4　ツール A のテストドライバ例

3.3　評価対象とした関数

ツールの原理上，ツール B，ツール C は動的なメモリを使った条件文を持つ関数に対してテストデータを生成することができない．よって，能力的にはツール A＞ツール B，ツール C であるが，ツール A は製品ではなく，オープン系の研究用ツールである．よって，専門的なスキルが必要となる．我々の評価の目的は，実務におけるテストデータ生成の合理的な方法を探ることにある．よって，手間を掛けずに自動生成できる部分は，ツール B かツール C で行い，原理的にツール B，ツール C では生成できない部分をツール A で解決できるかを評価する事とした．

評価対象の関数として，我々の製品に用いる 32bit 用ソフトウェアから 40 個サンプリングした．我々の製品に用いるソフトウェアの特徴は，グローバル変数を多く参照していることと，判断文のネストや構成する条件が多いことである．サンプリングは，関数の循環複雑度（以下，CCN）を基にすることとした．平均は CCN=6 で，約 99%が CCN=48 以下であったたため，CCN=3～50 の関数を評価対象とし，約 90%が CCN=11 以下であったことから，CCN=11 以下を中心にランダムに選定した．

評価は，この関数を対象にまずはツール B，ツール C によってテストデータ生成を行い，MC/DC カバレッジを達成できなかったものを対象に，ツール A を様々なオプションを駆使して実行し，達成できるか否かを調査した．

4.　評価結果

結果を表 1 に示す．ツール B を実行した結果，MC/DC カバレッジを満足した関数は 9/40 個で，ツール C を実行した結果，カバレッジを満足した関数は 29/40 個であった．カバレッジ未達であった関数の内，ツール B，ツール C で共通なものは，動的に算出される変数との比較の条件文を含む関数であった．尚，ツール C はテストデータをランダム生成しているため，動的に算出される変数を用いた条件文を含む関数においても，カバレッジを満たすテストデータを生成できている場合があった．また，ツール B では変数同士の比較の条件文を含む関数がカバレッジ未達であり，ツール C では構成する条件文の数が多い判断文を含む関数がカバレッジ未達であった．

ツール B，ツール C 共にカバレッジ未達であった関数 10 個に対し，ツール A を実行した．これ

らの関数には 1bit の変数など ON/OFF (1/0) を意味する変数を含むため，1 または 0 の値を取った時のみテストデータを生成するようにテストドライバを作成した（図 4)．まずは実行時オプションを-dfs で実行した結果，カバレッジを達成した関数は 5/10 個であった．カバレッジ未達での関数を確認したところ，4 個（表 1 の No.18，20，35，38）は冗長なロジックを含んでいたことが原因であった（図 5)．また， 3 個（表 1 の No.2，18，35）の関数はビット演算を含む条件に対するテストデータが生成できていなかった（図 6，7)．そこで，ツールの実行時オプションを変更して再実行した結果，-cfg，-cfg_baseline，-hybrid とした際に全ての関数において可達パスに対するテストデータを生成することができた．

```
if ( ( data1 != 1 )
  || ( ( data1 == 1 )
    && ( data2 < data3 ) ) ) {
  処理;
}
```

図 5　冗長なロジック例

```
data1 = (unsigned char)( ( data2 & 0x0000FF00 ) >> 8 );
if ( data1 == 0xE0 ) {
  処理;
}
```

図 6　実行時オプションにより MC/DC カバレッジが異なるロジック例 1

```
typedef struct{
  union{
    unsigned char data;
    struct {
      unsigned char b0 :1, b1 :1, b2 :1, b3 :1, b4 :1, b5 :1, b6 :1, b7 :1;
    }bit;
  }u1Byte;
}BitType;
BitType BitData;
#define data1 (BitData.u1Byte.bit.b0)
void  SampleTestFunc( void ) {
  if ( data1 == 1 ) {
    処理;
  }
}
```

図 7　実行時オプションにより MC/DC カバレッジが異なるロジック例 2

5.　まとめ

単体テストを効率化するために，現行の仕様ベーステストに加え実装ベーステストを用いたテストデータ自動生成ツールを導入するに際し，現在の市販ツールでは MC/DC カバレッジを満足できないケースが課題となっている．市販ツールのテストデータの自動生成方式は「静的解析」であり，解析の限界がカバレッジ未達の原因の一つと考えられる．そこで，静的解析の欠点を補うために開発された「記号実行」を用いたツールの導入を考え，我々の製品ソフトウェアを使用して記号実行の一種である「Concolic testing」を用いたツールを試行評価した．その結果，静的解析でカバレッジが達成できない関数に対して，Concolic testing を用いたツールでは可達パスに対するカバレッジが達成でき，課題を解決する目途がついた．Concolic testing を用いたツールに置換えることでテストデータ作成の自動化が可能となるため，コスト削減だけでなく，人手で完全には避けられないテストケースの抜け漏れを無くし開発工程の後戻り抑止も見込まれる．

現存する Concolic Testing を用いたツールはオープン系ツールであり，市販ツールに比べ操作性や作業性が落ちるため，ソフトウェア開発の現場で容易に使えるような環境の整備とツールの

保守方法の検討が今後の課題である．今回評価してきた中で，テストドライバやスタブの作成に多くの工数を要したが，定型作業のため自動化することも可能と考える．テストドライバの作成は，変数や型，データ範囲などの情報を抽出する必要があり，仕様書の記述方法の改善も含めて検討していく．また，派生開発においては，生成される全てのテストデータ，すなわち変更点以外のパスに対するテストデータについても仕様書との対応を確認する必要があり，効率化の障害となる．更なる単体テストの効率化のためには，確認するテストデータを仕様変更点に対するテストデータに絞り込むことが課題となる．

表１　生成したテストデータの MC/DC カバレッジ

○：達成　△：最良　－：省略

関数		MC/DC カバレッジ		
No	CCN	ツール B	ツール C	ツール A
1	3	100% ○	100% ○	－
2	4	0%	83%	100% ○
3	5	100% ○	100% ○	－
4	5	0%	100% ○	－
5	6	60%	100% ○	－
6	6	100% ○	100% ○	－
7	6	100% ○	100% ○	－
8	6	100% ○	100% ○	－
9	6	90%	100% ○	－
10	7	25%	100% ○	－
11	7	100% ○	100% ○	－
12	7	66%	100% ○	－
13	7	66%	100% ○	－
14	7	100% ○	100% ○	－
15	8	71%	100% ○	－
16	8	57%	100% ○	－
17	8	100% ○	86%	－
18	9	81%	81%	93% △ 注１
19	9	0%	100% ○	－
20	9	0%	62%	87% △ 注１

関数		MC/DC カバレッジ		
No	CCN	ツール B	ツール C	ツール A
21	10	66%	100% ○	－
22	10	33%	100% ○	－
23	10	0%	100% ○	－
24	10	44%	100% ○	－
25	11	100% ○	100% ○	－
26	11	0%	100% ○	－
27	11	80%	100% ○	－
28	12	9%	100% ○	－
29	13	16%	100% ○	－
30	14	53%	100% ○	－
31	14	53%	96%	100% ○
32	14	7%	100% ○	－
33	15	22%	96%	100% ○
34	17	10%	58%	100% ○
35	17	78%	93% △	93% △ 注１
36	19	27%	100% ○	－
37	20	50%	94%	100% ○
38	20	42%	75%	91% △ 注１
39	48	4%	23%	100% ○
40	50	30%	100% ○	－

注１　関数のコーディングが冗長であり網羅できないことが判明

参考文献

[1]大須賀竜治，ISO26262 の日本自動車業界での活動と今後について(〈 連載〉ISO 26262-自動車業界における機能安全に対する取り組み-)，品質，vol.43，no.2，pp.212-217，2013.

[2]高松宏樹，佐藤晴彦，小山聡，& 栗原正仁，(2014)．動的記号実行によるメソッドの複雑度を考慮したテストケース自動生成．情報処理学会研究報告．ソフトウェア工学研究会報告，2014(27)，1-7.

[3]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, 12, 2013
[4]岸本 渉, 安全系組込ソフトウェア開発におけるユニットテストの効率化,ソフトウェアシンポジウム 2015, C-14-2,2015
[5]松尾谷 徹, 増田 聡, 湯本 剛, 植月 啓次, 津田 和彦, 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,ソフトウェアシンポジウム 2015, C-14-2,2015
[6]K. Sen, D. Marinov, and G. Agha, CUTE: a concolic unit testing engine for C, vol.30, ACM, 2005.
[7]W. Visser, C.S. Psreanu, and R. Pel anek, Test input generation for java containers using state matching, Proceedings of the 2006 international symposium on Software testing and analysisACM, pp.37-48, 2006.
[8]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.
[9]CREST, Concolic test generation tool for c, http://jburnim.github.io/crest/.