Flash メモリ管理における Concolic Testing の活用 ～メモリパターンを含む自動回帰テスト～

西村 隆　　古畑 慶次　　松尾谷 徹
デンソークリエイト　デンソー技研センター　デバッグ工学研究所
nishimura_t@dcinc.co.jp　keiji_kobata@denso.co.jp　matsuodani@debugeng.com

要旨

Flashメモリを含むソフトウェアでは，メモリに書き込まれたパターンの影響を受けることから，網羅的なテストが難しい問題を持っている．ここでは，書き込まれたパターンがプログラムの動作に与える影響を Concolic Testing を用いて解析し，パターンの組合せを特定する．同様に，メモリ管理に与えられた引数（変数）についても Concolic Testing を用いて網羅性の高いテストケース作成することができる．実際のメモリ管理ソフトウェアを疑似したモデルを作成し，メモリパターンを含むテスト入力を自動生成した．このテストの能力を試すためバグを埋め込み，発見できることを確かめたので報告する．

1. はじめに

自動車に搭載されるソフトウェアは，その使われ方から高い信頼性が求められてきた．一方で，自動車に搭載されるソフトウェアは大規模化・複雑化が進んでいる．そのため，高い信頼性を保ちつつ，高い生産性を継続的に実現することが課題になっている[1].

その課題には，テストに関する問題が多く含まれている．テストの問題とは，ソフトウェアの動作に影響を与える因子の数と，因子間の関係によってテスト量が決まる．特に状態変数と呼ばれる順序論理が存在すると，膨大な組合せ数が必要となり，テストの効率を悪化させている．ここで対象とした Flash メモリを含むソフトウェアでは，メモリに書き込まれたパターンが，一種の状態変数として影響を与えることから，網羅的なテストが困難とされていた．

本稿では，Flash メモリに書き込まれたパターンが，どのようにソフトウェアの動作に影響を与えているのかについて，記号実行(Symbolic Execution) の技術を用いて探索し，その影響が生ずるパターンを特定する手法[2]を用いた．具体的には Concolic Testing のツール[3][4]を用いて評価プログラムを作成し，Flash メモリを実メモリで疑似することにより実現した．

実用化においては，テスト結果の期待値の問題，即ち「何が正解なのか？」を解決する必要がある．この問題については，回帰テストによる差分検出による方法[5]を用いて解決した．

2 章では，従来のテストにおける問題点を分析し，3 章で Concolic Testing を用いたメモリ管理ソフトウェア結合テストの方法を提案する．4 章では，単純なメモリ管理制御を行う評価用のサンプルプログラムでの実験と評価結果を示し，最後にまとめと今後の課題について述べる．

2. 現行の結合テスト取り組みと課題

2.1. メモリ管理ソフトウェアにおける結合テスト

メモリ管理ソフトウェアは，図1に示すように関数単体のユニット，複数のユニットの集まりであるコンポーネント，複数のコンポーネントの集まりである結合ソフトウェアで構成されている．テストレベルは，次の 3 段階で実施している．

(1) ユニットテスト
　関数単体に対して， MC/DC カバレッジ 100％の基準でテストケースを設計しテスト実施している．
(2) 単体コンポーネントの結合テスト
　複数のユニットを結合した単体のコンポーネントに対してテスト実施している．関連する外部コンポーネントは，テストドライバやスタブにより実現する．テストケースは，機能仕様と状態遷移に着目し，仕様を網羅する基準でテストケースを設計する．対象の規模が大きく，すぐに組み合わせ爆発を起こすため，テスト技法により合理的にテストケース数を減らす工夫を取り入れている[6].
(3) 複数コンポーネントの結合テスト
　複数のコンポーネントを結合したメモリ管理ソフトウ

ェア全体に対してテスト実施している. テストケース設計基準は(2)と同様である. 本テストは, ECU 実機によるテストと Flash メモリをシミュレートして実施するシミュレーションテストの 2 つの環境で実施している. ハードウェア依存部分の品質確保を ECU 実機によるテストで行い, ロジック部分の詳細をシミュレーションテストにより実施している. シミュレーションテストではテストの自動実行を取り入れ, 合理化を進めているが, 次に示す課題がある.

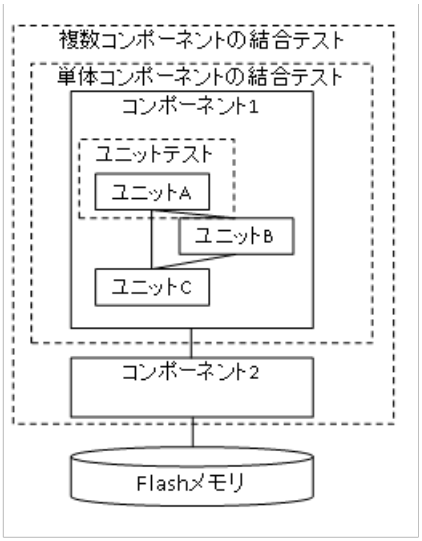

図 1 メモリ管理ソフトウェアのテスト

2.2. メモリ管理ソフトウェアの特徴と課題

状態遷移を含む仕様を基に行うテストケース設計において, 因子間の組合せ基準の問題がある. 組合せを仕様で定義された正しい組合せに限れば組合せ数の爆発は生じない. しかし, 現実問題として, バグは仕様で定義されていない組合せで生じることから, 組合せテストが必要である.

特にメモリ管理ソフトウェアのテスト入力となり得る因子は, プログラム入力であるユーザからの要求(Call 有無, 引数の値), 外部からの応答(応答有無, 応答の値)だけでなく, 図 2 に示すようにプログラム自体が持つ状態である制御の状態変数やタイマ, そして, Flash メモリの格納データもテスト入力となり得る.

メモリ管理ソフトウェアで問題となるのは. Flash メモリの格納データであり, 膨大なメモリ領域を持ち, データのアクセス順序によって取り得る状態は無数となる. 図 3 に示すように, 一旦書き込んだデータに対して上書きができず, 未書き込み領域へ追記する方式を取っているため順序が問題になる.

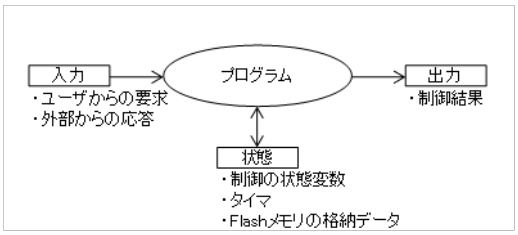

図 2 メモリ管理のテストケースとなり得る因子

ID1	Data	CS	無効
ID1	Data	CS	無効
ID2	Data	CS	有効
ID1	Data	CS	無効
ID1	Data	CS	有効
blank			
blank			
:			
blank			

処理順序
① ID1:書き込み
② ID1:書き込み
③ ID2:書き込み
④ ID1:書き込み
⑤ ID1:書き込み

※最後に書き込まれたデータを有効データとして扱い, それ以前に書き込まれたデータは無効データとする.

図 3 Flash メモリの格納データの例

今までの対策は, ユースケースにより想定しシステムに対して一連のフローを実行したときの様子を記述し, その結果できあがる Flash メモリの格納データパターンや Flash メモリの典型的な故障事象によるエラー推定によりテストケース設計している. 因子の網羅性の面では不十分である課題が残る. 実際の開発においても, Flash メモリの格納データが特殊な配置となってしまった場合において, 設計意図とは異なる動作をしてしまう問題が, 開発終盤の静的検証にて発覚した例もある.

課題としては, 起こり得る組合せを全て網羅する, 合理的で実現可能なテストとすることである.

3. 結合テスト方法の提案

3.1. 実装ベースの回帰テスト

メモリ管理を含む基盤ソフトウェアは，頭出しと呼ばれる新規開発完了後は，実装するマイコン種類の増加に伴う派生開発と，信頼性向上のためのリファクタリングが主な開発となっている．その開発のつど，回帰テストを実施する必要があり，大きな作業割合を占めている．本稿ではこの回帰テストを対象に検討する．回帰テストとは，「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，既にテスト済のプログラム対して実施するテスト」である．

回帰テストを実現するためには，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ことが必要である．また，膨大な入力に対するテスト期待値を求めること(オラクル問題と呼ばれる)も必要となる．これらの課題に対するアプローチとして仕様ベースのテストと実装ベースのテストが考えられる．

仕様ベースのテストとは，実装されたプログラムを，仕様を元にテストする手法である．我々が開発するメモリ管理ソフトウェアでは，頭出しの新規開発のときから本手法にてテストケース設計を進め，回帰テストを行ってきた．しかし，本手法には 2 つの課題がある．ひとつは，限られた人にしか対応できない点である．自然言語で書かれた仕様書から，変更の影響がある非互換部分と変更の影響がない非互換部分を正しく抽出するのは困難であり，変更対象プログラムへの深い理解と十分な影響解析が必要なためである．もうひとつは， 2.2 章で述べたように Flash メモリの格納データのような複雑な条件をもつ状態値をテスト入力およびテスト期待値として設定する必要があり，テストケース設計の難易度が高く膨大な工数も必要となる点である．この課題を解決するために，仕様ベースのテストに加えて，実装ベースのテストも実施することを提案する．

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，テストを行う手法である．先行研究の一つに，プログラムが持つ制御構造を探索し，到達可能パスを解析し自動でテスト入力を生成するツールがあり，この有効性が報告されている[7][8][9]．このツールを変更前プログラムと変更後プログラムの双方へ適用し，到達可能パスを網羅的に実行できるテスト入力が自動で生成できる．図 4 に示すように，このテスト入力に対する変更前プログラムのテスト出力と変更後プログラムのテスト出力を比較することで，変更の影響のない互換部分は一致し，変更の影響がある非互換部分は不一致となる．この不一致部分に対して，変更仕様に基づくものか否かの判断を行うことで，回帰テストが可能となる．

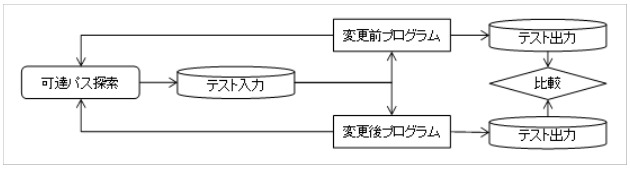

図 4 実装ベースの回帰テスト構成概要

3.2. ツールの選択

到達可能パスを解析し自動でテスト入力を生成するツールの選択を行う．ここでは，現場にて実用化するため，容易に試用できることが求められることから，研修用に提供されている環境の CREST を用いることとした．CREST とは，オープンソースとして公開されている C 言語向け Concolic Testing ツールである．

Concolic Testing は，Concrete Execution と Symbolic Execution を組み合わせた手法である．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，制約ソルバを使って，分岐条件を探り，解を求める．制約ソルバで解けない場合は，乱数を使うなどヒューリスティックな手段を併用している．

3.3. メモリ管理ソフトウェアへの適用

Concolic Testing ツールである CREST を用いた実装ベースの回帰テストを，メモリ管理ソフトウェアに適用する方法について説明する．

(1) 対象とする変数の決定

CREST は，テスト入力となり得る変数(以後，CREST 変数と呼ぶ)を指定し，到達可能パスの探索を行う必要がある．メモリ管理ソフトウェアのプログラム入力全てを CREST 変数にできれば，プログラム入力全てを変化させて到達可能パスの探索が可能であるが，実行時間が膨大となり現実的なオーダーではテスト実行できなくなる課題がある． Flash メモリの格納データ以外は，仕様ベースのテストケース設計により，網羅的なテスト入力値を既に有している．

Flash メモリの格納データは，データフォーマットは定義されているものの，膨大なメモリ領域を持ち，データのアクセス順序によって取り得る状態は無数にあるため，人手による仕様ベースのテストケース設計では網羅的な入力値を算出できていない．この課題を解決するために，強力な到達可能パス解析ツールである CREST を活用する．そのため，Flash メモリの格納データを CREST 変数とする．Flash メモリの格納データ以外のプログラム入力は，仕様ベースのテストケース設計で算出した値を予め固定値として与える．

(2) ハードウェア制御のシミュレーション

メモリ管理ソフトウェアは，ハードウェア制御基幹部のソフトウェアコンポーネントであり，ECU 実機において Flash メモリの格納データへアクセスするためには，ハードウェア制御が必要となる．CREST は，C 言語プログラムに対応したツールであるため，そのままでは Flash メモリの格納データを CREST 変数とすることはできない．これを解決するため，ハードウェア制御を実行しているドライバ部をシミュレートしたシミュレーション環境を準備する．シミュレーション環境では，Flash メモリの格納データを RAM 領域として準備し，これを CREST 変数とすることで，CREST の実行が可能となる．

(3) テストドライバの作成

図 5 に示すように，メモリ管理ソフトウェアは，ユーザプログラムからのインターフェース呼び出し（書き込みや読み出し）と周期起動スケジューラからの定期処理（ユーザプログラムからの要求に従い，Flash アクセス制御を順次実行する処理）呼び出しによって実行されるアーキテクチャである．仕様ベースのテストでは，ユースケース想定によりテストケース設計した処理シーケンスをテストドライバで模擬し，テスト実行している．図 6 に仕様ベースのテストドライバ処理具体例を示す．図 6 は，ユーザプログラムから 1 周期目に書き込み処理，2 周期目に読み出し処理したテストケースのテストドライバ処理である．このようなテストドライバは，CREST を用いた実装ベースの回帰テストにおいて 2 つの問題がある．

一つ目の問題は，CREST 変数とした変数に対して，プログラムで代入を行った場合，その時点で CREST による到達可能パスの探索が打ち切られることである．CREST を実行するためには，探索対象プログラムの先頭で CREST 変数を宣言することが必要であるため，図 6 の A 時点で Flash メモリの格納データを CREST 変数として宣言を行う．ところが，1 周期目の書き込み処理で Flash メモリの格納データへ代入してしまうと，B 時点以降の 2 周期目の読み出し処理は，CREST によって与えられた値ではなく，プログラムによって代入された値で動作することになり，CREST による探索ができない．

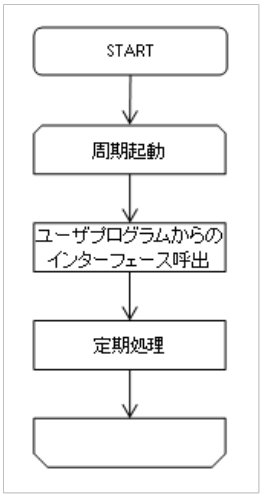

図 5 メモリ管理ソフトウェア動作

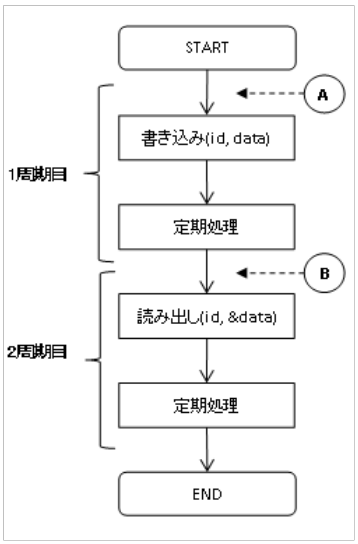

図 6 仕様ベーステストのドライバ処理

二つ目の問題は，メモリ管理ソフトウェアは順序論理を含むプログラムであり，起こり得る組み合わせを全て網羅するためには，爆発的な組み合わせのテストケースを必要とすることである．順序論理を含むプログラムとは，状態を持つプログラムのことである．プログラムの出力は，入力だけでなく以前の実行結果によって決定される状態の影響も受ける．よって，テストケースは，入力パターンに加え処理順序の影響

も受けるため，これを加味したテストケースは組み合わせ爆発を起こす．2.2 章で述べた通り Flash メモリの格納データは，データのアクセス順序によって状態が決定されるため，順序論理を含むプログラムといえる．

これら2つの問題に対して，処理をユーザプログラムの実行単位に分割したテストドライバを用意しCREST 実行することとする．図 6 のような順序制御も，1 周期目，2 周期目それぞれ単体で見ると順序論理を持たない（出力は，入力の組み合わせで一意に決まる＝組み合わせ論理）と言える．これが成立する前提として，分割した処理の途中で外因により状態が変化しないことが必要であるが，メモリ管理ソフトウェアはシングルタスクで駆動されておりこれを満たしている．2 周期目の出力は，1 周期目実行後の Flash メモリの格納データ（状態）によって決まるが，1 周期目と切り離して考えると，Flash メモリの格納データ（状態）も2周期目の入力の一つと考えることができる．よって，処理をユーザプログラムの実行単位に分割したテストドライバであれば，組み合わせ論理のプログラムとなるため，2 つの問題は解消される．一つ目の CREST 探索打ち切り問題は，組み合わせ論理であれば，解消される．プログラム中に代入があったとしても，シングルタスクで駆動されている中では，プログラム中の代入文以降は固定ロジックでありテスト不要なものといえる．二つ目の組み合わせ爆発も，順序論理から組み合わせ論理に置き換えることで解消される．組み合わせ論理となることで，処理順序の組み合わせが不要となり，組み合わせ爆発は起こらない．変更前プログラムと変更後プログラムにおいて，処理の実行単位毎に，全ての分岐条件を網羅した入力に対する出力が一致していれば同じプログラムと言えるため，結合されたソフトウェアに対する回帰テストができているといえる．

テストドライバ作成についてまとめると，図 7 のようになる．最初にテスト入力値設定処理を行う．テスト入力値としては，予め固定値として与えるものとCERST 変数宣言するものがある．次に，テスト対象プログラムを実行し，最後にテスト出力を行う．

4. 提案手法の実験

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．実験には，分析や評価の容易性のため，メモリ管理ソフトウェアを疑似した評価用のサンプルプログラムを準備し，これを使用した．サンプルプログラムは，3 つのコンポーネントで構成し，全体の規模は 615 ステップである．実行環境は，Windows7 上で Vagrant を用いて構築した Ubuntu15.04 の LINUX 環境とした．

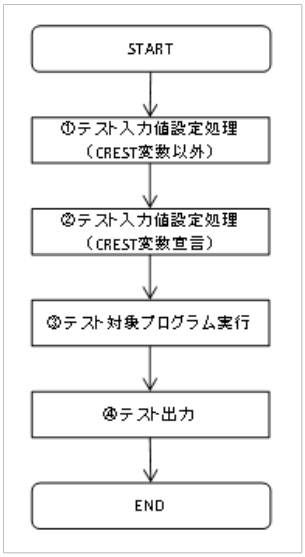

図 7 CREST テストドライバ概要

4.1. テストドライバの実装

3.3 章の方針に従い CREST のためのテストドライバを準備する．テストドライバは簡単で，図 8 のようである．2 行目は，仕様ベースのテストケース設計により，あらかじめ準備した Flash メモリの格納データ以外のテスト入力値設定処理（図 7-①に対応）である．4～15 行目は，Flash メモリの格納データを CREST 変数として宣言している（図 7-②に対応）．17～18 行目は，処理の実行単位に分割したテスト対象プログラムの実行（図 7-③に対応）である．20 行目は，テスト出力処理（図 7-④に対応）である．

```
1       /* set inputdata */
2       setTestData(i);
3       /* declare CREST value */
4       CREST_unsigned_char(flashdata[0]);
5       CREST_unsigned_char(flashdata[1]);
6       CREST_unsigned_char(flashdata[2]);
7       CREST_unsigned_char(flashdata[3]);
8       CREST_unsigned_char(flashdata[4]);
9       CREST_unsigned_char(flashdata[5]);
10      CREST_unsigned_char(flashdata[6]);
11      CREST_unsigned_char(flashdata[7]);
12      CREST_unsigned_char(flashdata[8]);
13      CREST_unsigned_char(flashdata[9]);
14      CREST_unsigned_char(flashdata[10]);
15      CREST_unsigned_char(flashdata[11]);
16      /* execute testcase */
17      ret = ms_write(id, data);
18      ms_periodic();
19      /* output */
20      printf("i = %d, ret = %d¥n", i, ret);
```

図 8　CREST のためのテストドライバ

4～15 行目の CREST 変数の宣言に工夫がある. Flash メモリの格納データは, シミュレーション環境において連続の RAM 領域に確保しているため, CREST 変数としては, unsigned char 型の変数として 1byte ずつ宣言している. Flash メモリは, 通常数十 Kbyte 程度と膨大であり, 全ての領域を CREST 変数として宣言し探索を行うことは現実的な実行時間ではできない. ユーザが取り扱う Flash メモリのデータは, 固定サイズで, Flash メモリ内のデータフォーマットも固定フォーマットであり, これを保持データ数分繰り返している（図 3 のイメージ）. この Flash メモリのデータフォーマット特徴に着目し, 同値分割の考え方より, 実際の Flash メモリが保持できるデータ数よりも小さいサイズを設定することとした. これにより, CREST 変数の個数を削減し, 実行時間の短縮を図っている. このとき, Flash メモリの格納データを何データ分確保すべきか（どこまでデータ数を削減できるか）を決定する必要がある. メモリ管理ソフトウェアには, 複数データを保持している場合のみ動作するロジックが実装されている. 実装ベースのテスト入力としては, 実装されたプログラムの分岐条件を網羅するデータが必要である. これの分析にも CREST が活用できる. Flash メモリの格納データ数を 1 個ずつ増やして, CREST による探索を行う. このときの分岐網羅率を計測し, この値が頭打ちとなったところで, データ数に依存するロジックが十分に実行できるデータ数であると判断し, テスト入力として採用する. 表 1 にサンプルプログラムの Flash メモリデータ数と分岐網羅率の計測結果を示す. 分岐網羅率が頭打ちとなったデータ数は 2 個であり, サンプルプログラムが持つ, 複数データを保持している場合のみ動作するロジック「書き込み済データの CheckSum が異常であった場合, 他の書き込み済データがないかを探索する処理」を持つ設計と一致しており妥当なデータサイズと言える. この考え方を抽象化すると, Flash メモリの格納データは, 一定の形式で定義されたデータの集合体といえる. このようなデータの集合体を入力とし, 分岐条件へ影響を与える実装を持つプログラムへ, 本手法は適用可能である. 具体的には, データベース管理や文字列探索プログラムへ応用できる.

表 1　Flash メモリデータ数と分岐網羅率

データ数	1 個	2 個	3 個	4 個
Flash メモリサイズ	4byte	8byte	12byte	16byte
Iteration 数	56 回	2112 回	57824 回	60000 回以上
実行時間	1.6s	12.2s	320.3s	打ち切り
分岐網羅率	93.55%	100.00%	100.00%	100.00%

4.2. 評価

CREST の実行結果を評価する.

これまでの仕様ベースのテストで抽出したテスト入力（Flash メモリ格納データは固定値採用）時の分岐網羅率と CREST により生成されたテスト入力（Flash メモリの格納データの到達可能パスを CREST により探索した入力値採用）時の分岐網羅率を表 2 に示す. 両者を比較したところ, CREST 変数採用時の分岐網羅率は 100%であり固定値採用時よりも分岐網羅率が高い. この結果より, Flash メモリの格納データのような複雑な条件をもつ状態値を人手によりテスト入力として設定するのが困難な場

表 2　CREST による分岐網羅率

Flash メモリの格納データ	固定値採用	CREST 変数採用
実行時間	1.2s	13.2s
分岐網羅率	70.19%	100.00%

合でも，到達可能パスを解析し自動でテスト入力を生成する CREST のようなツールを利用することで，分岐網羅率が高いテスト入力値を生成できることが確認できた．

さらに，実際の基盤ソフトウェア開発で，開発中に発生した「変更によるインターフェース不一致」が実装ベースの回帰テストにより発見可能かを検証した．今回のサンプルプログラムにおいて，下位コンポーネントに機能追加し，処理成功時の戻り値のパターンを増やす変更を行った．上位コンポーネントは，下位コンポーネントからの戻り値のパターンが増えているため，これに対応する必要があるが，この変更が漏れた状況で，実装ベースの回帰テストを行った．その結果，表 3 に示す変更前プログラムと変更後プログラムのテスト出力にて，互換部分であるはずのテストケース 5 において不一致を検出し，今回の変更漏れが検出できることが確認できた．

表 3 変更前と変更後プログラムの回帰テスト結果

テスト番号	変更前 テスト出力	変更後 テスト出力	比較結果
1	0	0	一致
2	0	0	一致
3	0	0	一致
4	1	1	一致
5	1	2	不一致
6	0	0	一致
7	0	0	一致
8	0	0	一致
9	0	0	一致
10	0	0	一致
11	1	1	一致
12	0	0	一致
13	0	0	一致
14	0	0	一致
15	0	0	一致
16	1	1	一致

5. おわりに

C 言語向け Concolic Testing ツールである CREST をメモリ管理ソフトウェアに適用し，実装ベースの回帰テストが実行可能であることを，サンプルプログラムを使って確認することができた．これにより，Flash メモリの格納データのように，非常に多数のパターンを取る状態変数を持つプログラムにおいて，分岐網羅率を向上したテスト入力が自動で生成でき，ユニットやコンポーネントを結合した状態での，結合時のインターフェース不一致や相互作用により発生する欠陥を摘出するためのテストが可能であることを確認できた．

今後，本提案を実際の開発で利用可能とするために，Flash メモリの格納データサイズや探索対象プログラムの規模増加時の実行可能な上限の調査，および，テストドライバ作成・回帰テスト実行の効率化を検討する予定である．

参考文献

[1] 菅沼賢治，村山浩之，“自動車組み込みソフトウェアにおける開発戦略”，情報処理，vol.44，no.4，pp.358-368，2003.

[2] C. Cadar, P. Godefroid, S. Khurshid, C.S. Pasareanu,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.

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

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

[5] 松尾谷徹，増田聡，湯本剛，植月啓次，津田和彦，“Concolictesting 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 2015in 和歌山 SEA:ソフトウェア技術者協会，pp.47-56,2015.

[6] 中野 隆司・小笠原 秀人・松本 智司，“組み合わせテスト技術の導入・定着への取り組み，および上流設計への適用検討の事例”，SQiP シンポジウム 2011，2011

[7] 丹野 治門・星野 隆・Koushik Sen・高橋 健司，「Concolic Testing を用いた結合テスト向けテストデータ生成手法の提案」，Vol.2013-SE-182 No.6，2013

[8] 岸本 渉，「安全系組み込み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 2015，2015

[9] 榎本 秀美，「記号実行を用いたテストデータ自動生成の試行評価」，ソフトウェア品質シンポジウム 2015，2015