Vol.2012-SE-178 No.8
2012/11/1

# 制御ソフトウェア向けテストケース生成方式の提案

磯田 誠[†]　　徳永雄一[†]

制御システムは，制御装置を中心にセンサ装置と駆動装置を備え，外部環境に対して適切な物理的制御をかけることを目的とする．制御装置の高機能化・付加価値向上のため，制御ソフトウェアの大規模化・複雑化が進んでいる．近年，設計工程で専用記法を用いて制御ロジックを作成し，実装・試験工程で活用するモデルベース開発の取り組みが盛んである．これに対して実開発では，単体・S/W 結合試験でのテストケースの作成工数が大きい，システム試験での機能動作の期待値の作成工数が大きいことが問題になっている．本稿では，制御ソフトウェアの開発効率化・高品質化の基盤となる単体・S/W 結合試験に取り組み，制御ソフトウェアに必須の状態依存処理を対象に，実装コード中の分岐箇所に到達するための時系列入力のテストケースを自動生成する方式を提案する．

# An Approach of Test Case Generation for Control Software

MAKOTO ISODA[†]　YUICHI TOKUNAGA[†]

Embedded control software has to fulfill high level requirements for functionality and extensibility, so that it becomes difficult to implement such complex software correctly. It is certainly needed to assure quality of software, moreover needed to improve productivity of software testing and system testing. In this paper, we propose a test case generation method using formal verification technology. This method allows us to achieve enough code coverage in software unit and component testing.

## 1. はじめに

制御システムは，制御装置を中心にセンサ装置と駆動装置を備え，外部環境に対して適切な物理的制御をかけることを目的とする．これまで制御装置の低コスト化・短納期化に応えるための電子化が進められ，制御機能をソフトウェアで実現する割合が増加してきた．さらに，制御装置の高機能化・付加価値向上のため，制御ソフトウェアの大規模化・複雑化が急速に進んでおり，品質確保のための開発工数増加が問題になっている．

近年，設計工程で専用記法（モデル）を用いて制御ロジックを作成し，さらに実装・試験工程で活用するモデルベース開発の実用化の取り組みが盛んである．作成したモデルを元にコード自動生成，テストケース自動生成，試験自動判定といった作業自動化を進め，制御ソフトウェアの品質確保と開発工数削減の両立を狙っている．実際の製品開発では，開発プロセスの中でも単体・S/W 結合試験でのテストケース（ソフトウェアへの入力値）の作成工数が大きい，システム試験での機能動作の期待値（ソフトウェアからの出力値）の作成工数が大きいことが問題になっている．

これに対して我々は，制御ソフトウェアの開発効率化・高品質化の基盤となる単体・S/W 結合試験に取り組む．本稿では,制御ソフトウェアに必須の状態依存処理を対象に,実装コード中の分岐箇所に到達するための時系列入力のテストケースを自動生成する方式を提案する．また，制御ソフトウェアの題材を用いて，本方式の適用例と評価結果を示す．

† 三菱電機(株) 情報技術総合研究所
Mitsubishi Electric Corporation Information Technology R&D Center

## 2. 関連技術

### 2.1 CBMC

CBMC（Bounded Model Checking for ANSI-C）は，ANSI-C 準拠の C 言語関数を解析し，演算結果のオーバーフローやアンダーフロー，0 による除算など，不具合となる可能性がある処理を検出する技法・ツールである[1][2].

ツール内部で形式検証の一種である SAT solver を用いており,変数や演算をすべて論理型に変換してから解析する.

ユーザが記述する assert 文のチェック機能を，1 ステップ分のテストケース生成に応用する研究がある．しかし，状態依存処理に対するテストケース生成は見当たらない.

### 2.2 ESBMC

ESBMC（Efficient SMT-Based Context-Bounded Model Checker）は，2.1 の CBMC と同様に，不具合となる可能性がある処理を検出する技法・ツールである[3].

ツール内部で形式検証の一種である SMT solver を用いており，整数型や浮動小数点型の変数や演算を直接扱うことで，CBMC より処理効率を向上していることが特徴である.

ユーザが記述する assert 文のチェック機能を，1 ステップ分のテストケース生成に応用する研究がある．しかし，状態依存処理に対するテストケース生成は見当たらない.

## 3. 制御ソフトウェアのモデルベース開発

### 3.1 開発プロセス概要と実開発の問題点

近年，製品開発の設計工程で専用記法（モデル）を用いて制御ロジックを作成し，さらに実装・試験工程で活用するモデルベース開発の実用化の取り組みが盛んである．作成したモデルを元にコード自動生成，テストケース自動生

成，試験自動判定といった作業自動化を進め，制御ソフトウェアの品質確保と開発工数削減の両立を狙っている．モデルベース開発を適用した開発プロセスを図 1 に示す．

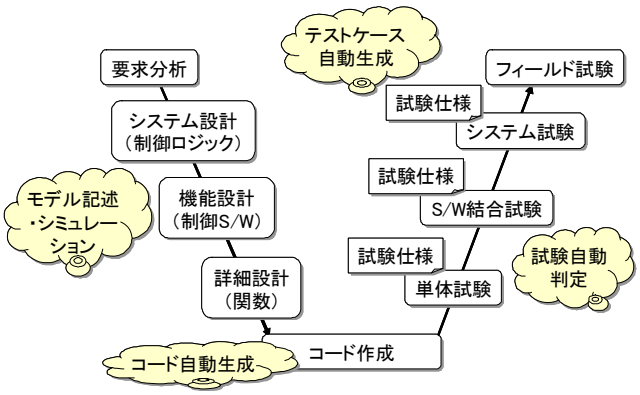


図 1　モデルベース開発を適用した開発プロセス

実際の製品開発では，既に以下の点が問題になっている．本稿では，制御ソフトウェアの開発効率化・高品質化の基盤となる(b)単体・S/W 結合試験に取り組む．

(a) 開発全体では，設計・コード作成工程よりも試験工程の工数割合が大きい．そのため，モデルベース開発で注目されるコード自動生成の効果が小さい．

(b) 単体・S/W 結合試験では，実装コードの構造の網羅が重要である．例えば，自動車では ISO26262，宇宙航空では DO-178B という標準規格でコードカバレッジを求められている．そのため，テストケース（ソフトウェアへの入力値）の作成工数が大きい．

(c) システム試験では，機能的に正しく動作するか確認することが重要である．そのため，機能動作の期待値（ソフトウェアからの出力値）の作成工数が大きい．

上記の問題点と設計／実装物の対応付けを図 2 に示す．

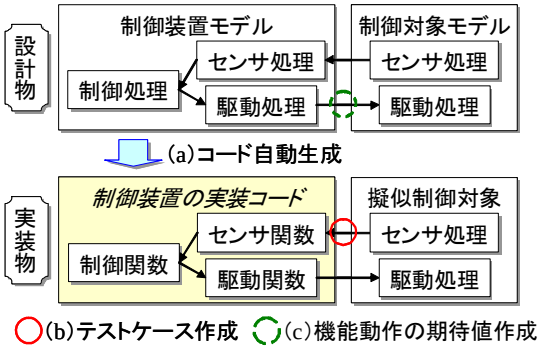


図 2　実開発の問題点と設計／実装物の対応付け

### 3.2 単体・S/W 結合試験の課題

制御ソフトウェアには，外部環境に適切な物理的制御をかけるため，過去の入力履歴や制御対象からのフィードバックに基づいて制御演算する状態依存処理が必須である．そのため，単体・S/W 結合試験では，状態依存処理がある場合でも実装コード中の分岐箇所をすべて実行するテストケースを自動生成する必要がある．具体的には，以下を実現することが求められる．

(1) 実装コードを複数ステップにわたって実行する時系列入力を生成すること．

(2) 実装コード中のループ箇所を解析可能であること．一般に，リアルタイム性を保証するためコーディング規約でループ回数が制限されることが多いので，回数上限つきでよい．

(3) 論理的に到達不可能な箇所（デッドコード）を特定可能であること．

従来のテストケース自動生成技術では，1 ステップ入力なら可能である．しかし，状態依存処理に対する時系列入力は手作業で作成する必要がある．

### 3.3 課題解決の方針

3.2 に示した課題に対して我々は，実装コード中の分岐箇所に到達するための入力変数値を解析し，時系列入力のテストケースを自動生成する方式を提案する．本方式の特徴を以下に示す．

(1) 実装コードを複数ステップ分連結した状態依存処理コードを形式検証で解析し，分岐箇所に到達するための入力変数値を求めて時系列に組み立てる．

(2) 実装コード中のループ処理（while 文，for 文，do 文）を有限回数のフラットな処理に展開し，ループのない場合と同様のアルゴリズムで解析可能とする．

(3) 形式検証で入力変数値が求まらない箇所を到達不能と判定する．実装コード中にループがある場合は，有限回数で到達不能と判定する．

本方式の適用対象コードの条件は以下のとおり．

(i) Embedded Coder[a]を用いて Simulink[a]モデルから自動生成した，シングルスレッドで動作する C 言語関数．

(ii) Simulink モデルの入出力信号と，入力履歴やフィードバックを保存する状態変数は，大域変数で定義される．

また，従来方式との対比を図 3 に示す．本方式では，実装コード中の到達可能箇所と到達不能箇所を合わせて，分岐カバレッジ 100%を達成するテストケースを生成する．

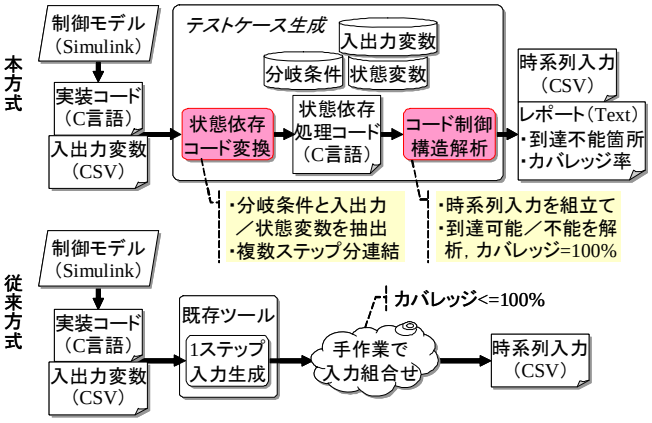


図 3　本方式の特徴および従来方式との対比

a Embedded Coder および Simulink は The MathWorks, Inc.の登録商標である．

## 4. 時系列入力テストケース生成方式

### 4.1 機能一覧

本方式で提供する機能の一覧を表 1 に示す．Simulink モデルから生成した C 言語関数を入力に，最終的に時系列入力とレポート（到達不能箇所，カバレッジ率）を出力するまでの処理を細分化し，各段階を機能として定義する．

表 1　機能一覧

| 分類 | 項目 | 説明 |
|---|---|---|
| 状態依存コード変換 | 分岐記録文付きコード出力 | 実装コード中のすべての分岐箇所に到達を記録する文を挿入したコードを出力. |
| | 状態依存処理コード出力 | 分岐記録文付きコードを複数ステップ分連結したコードを出力. |
| コード制御構造解析 | 分岐箇所到達解析 | 各ステップの分岐箇所の到達可能／到達不能を判定し, 到達可能の場合は入力変数値を計算. |
| | 時系列入力生成 | 各ステップの入力変数値を並べて, 時系列入力を生成. |
| | レポート出力 | 実装コード中の到達不能箇所, 実装コードのカバレッジ率を出力. |

### 4.2 データ構造一覧

4.1 に示した各機能で処理するデータ構造の一覧を表 2, 表 3 に示す.

表 2　データ構造一覧（1/2）

| 分類 | 項目 | 説明 |
|---|---|---|
| 試験対象 | 実装コード (C 言語) | Simulink モデルから生成した，シングルスレッドで動作する ANSI-C 準拠の C 言語関数. |
| | 入出力変数 (CSV) | Simulink モデルの入出力信号に相当する大域変数．コード中で，入力変数に対する操作は読出のみ，出力変数に対する操作は読出と設定. |
| | 初期化関数 (C 言語) | 実装コード（C 言語）中の大域変数を初期化するための関数．コードカバレッジの対象外. |
| 解析用コード | 状態依存処理コード (C 言語) | 実装コード（C 言語）中のすべての分岐箇所に到達を記録する文を挿入し, 複数ステップ分連結したコード. |
| | 分岐条件 | 実装コード（C 言語）中の分岐箇所を一意に特定する ID と, 到達を記録するフラグ. |
| | 状態変数 | 入出力変数（CSV）を除いたすべての大域変数．前値保持による処理切り替えなどの状態処理に使われる. |
| | 入出力変数 | 入出力変数（CSV）で指定された入出力変数. |

表 3　データ構造一覧（2/2）

| 分類 | 項目 | 説明 |
|---|---|---|
| テストケース | 時系列入力 (CSV) | 実装コード（C 言語）の入力変数に対する時系列入力のリスト. |
| | レポート (Text) | 実装コード（C 言語）の到達不能箇所およびカバレッジ率のレポート. |

### 4.3 アルゴリズムの方式

本方式のアルゴリズムは，実装コードを複数ステップ分連結した解析用コードを作成する状態依存コード変換，形式検証の一種である SMT solver（Satisfiability Modulo Theories solver）を用いて入力変数値を求めて時系列入力を生成するコード制御構造解析という，大きく 2 段階の処理で構成する.

#### 4.3.1 適用対象コードの構造

本方式の適用対象コードの典型的な構造を図 4 に示す. センサ装置で読み取った情報や外部装置から受信したメッセージの内容が入力変数（図 4 では i，j）に格納され, Simulink モデルから生成した C 言語関数（図 4 では step()）でこれを読み込んで制御処理を実行する．1 ステップ分の C 言語関数では，ループ処理や分岐処理の制御構造に従って，それまでの入力履歴や制御対象からのフィードバックを格納する状態変数（図 4 では x，y）を読み込んだり書き出したりしながら制御処理を進め, 処理結果を出力変数（図 4 では m，n）に書き出す．この C 言語関数を指定された周期で実行し, 入力変数を新たな値に書き換えていくことで, 外部からの連続的な入力と制御処理を実現する.

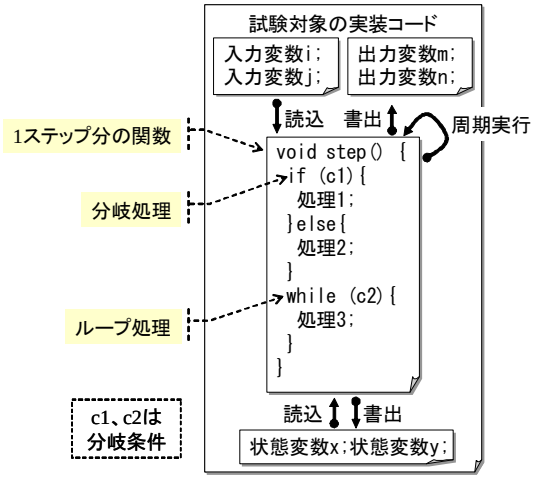


図 4　適用対象コードの典型的な構造

本方式でサポートする実装コードの文法を表 4, 表 5 に示す.

表 4　サポートする実装コードの文法（1/2）

| 項目 | サポート内容 |
|---|---|
| 分岐処理 | ・if 文，else if 文，else 文<br>・switch-case 文<br>・条件演算（a ? b : c の形式） |

表 5　サポートする実装コードの文法（2/2）

| 項目 | サポート内容 |
|---|---|
| ループ処理 | ・while 文，for 文，do 文<br>・無限ループは未サポート<br>・backward goto 文は未サポート |
| 関数 | ・非再帰呼び出し<br>・再帰呼び出し，循環呼び出し，マイコン依存処理は未サポート<br>・ポインタ処理は未サポート |

### 4.3.2 状態依存コード変換

アルゴリズムの 1 段目となる状態依存コード変換のフローチャートを図 5 に示す.

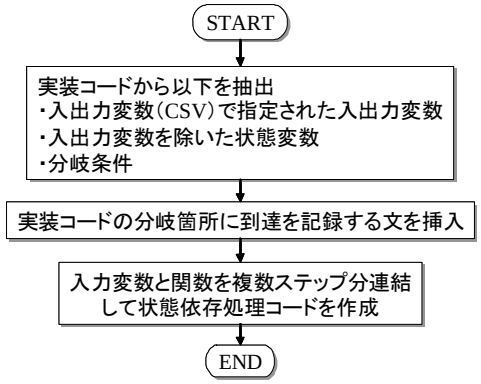


図 5　状態依存コード変換のフローチャート

状態依存コード変換のフローチャートについて詳細に説明する. まず図 6 では, 図 4 の実装コードのすべての分岐処理とループ処理を識別し，分岐箇所への到達を記録する文（図 6 では分岐記録文 A など）を挿入する.

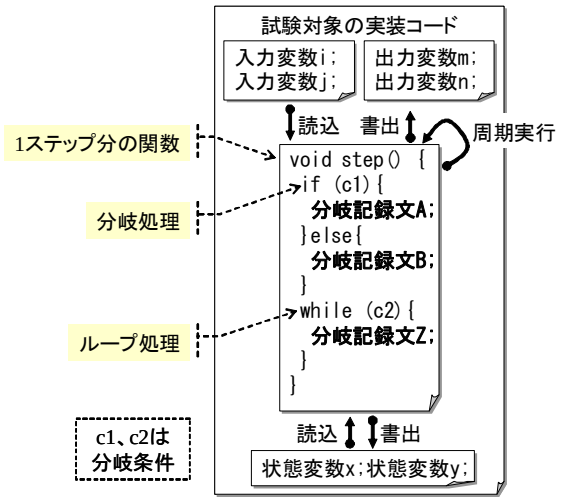


図 6　実装コードへの分岐記録文の挿入

次に図 7 では，C 言語関数を複数ステップ分複製して連結し，連結した関数を逐次的に実行することで周期実行を模擬する（図 7 では一点破線の矢印）. また，毎周期書き換えていく入力変数を，各ステップと 1 対 1 になるように新たに定義し（図 7 では i1，j1，i2，j2，in，jn），各ステップでは対応する入力変数を読み込むように処理を修正する. このとき，状態変数（図 7 では x，y）は各ステップ間で共有して値を受け渡す必要があるので，元の処理から変更しない. なお, 出力変数とループ処理の記載は省略した.

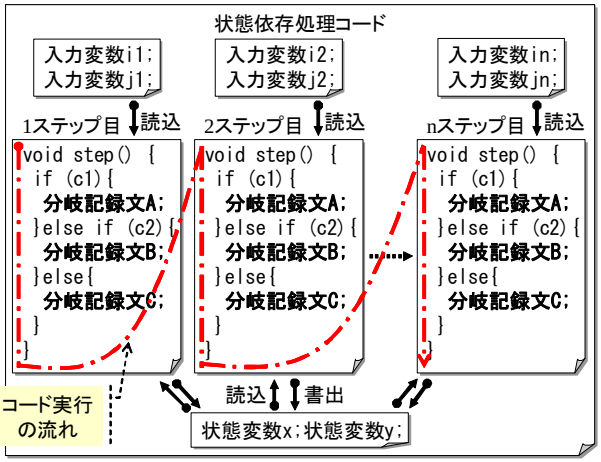


図 7　状態依存処理コードへの変換

### 4.3.3 コード制御構造解析

アルゴリズムの 2 段目となるコード制御構造解析のフローチャートを図 8 に示す.

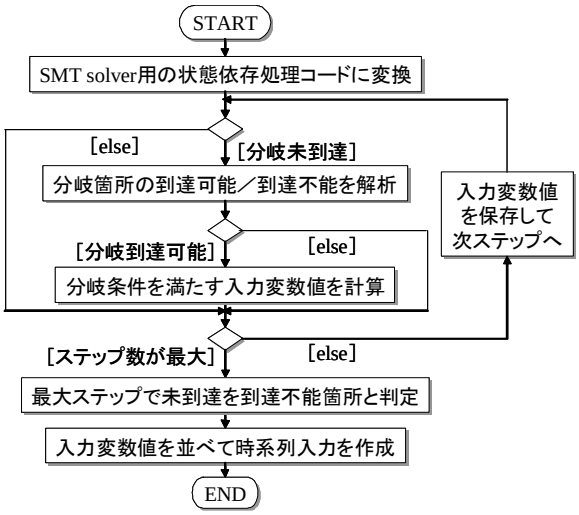


図 8　コード制御構造解析のフローチャート

コード制御構造解析について詳細に説明する. まず，以下の手順で状態依存処理コードを SMT solver 用コードに変換する.

- 状態依存処理コード中のループ処理を有限回のフラットな処理に展開，関数呼び出しをインライン化し，ただ一つの関数に変換[1][2].
- C 言語の手続き処理を SMT solver の数式処理に変換するため，中間形式として静的単一代入形式に変換[4][5]および Conditional Normal Form に変換[6][7].
- Conditional Normal Form を SMT solver の標準形式である SMT-LIBv2 形式（以降，SMT 形式）に変換[8][9].

ここで，中間形式である静的単一代入形式と Conditional Normal Form について補足説明する. C 言語の手続き処理では, 変数の値を何度でも書き換えることができる. 一方, SMT solver の数式処理では，変数の値は解析した結果求ま

るものであり，書き換えることはできない．そこで，両者の意味論を対応付けるため，実装コードを静的単一代入形式（Static Single Assignment Form，以降 SSA 形式）に変換する．SSA 形式とは，変数を世代管理することで各変数に一度だけしか値が代入されない，すなわち書き換えられないように変換した形式である．GNU gcc バージョン 4 以降を用いると，実装コードから SSA 形式を生成することができる．また，SMT solver は入れ子の分岐処理を扱えない．そこで，SSA 形式の入れ子の分岐処理を，単一の if 文のフラットな羅列に展開した Conditional Normal Form に変換する．以上のように，SSA 形式と Conditional Normal Form を経由することで，意味論を変えずに C 言語の手続き処理を SMT solver の数式処理に変換する．

次に図 9 では，SMT 形式に変換した状態依存処理コードの 1 ステップ目にある分岐箇所について SMT solver で解析し，到達可能（図 9 では分岐記録文 A）なら入力変数値（図 9 では i1，j1）を求める．到達可能ではない箇所は，2 ステップ目以降の分岐箇所について同様に解析していき，最大ステップ（図 9 では n ステップ目）でも到達可能とならなかった分岐箇所（図 9 では分岐記録文 C）を到達不能箇所と判定する．このとき，一度到達可能と判定した箇所（図 9 では分岐記録文 A）は以後解析しないようにして，解析処理を効率化する．

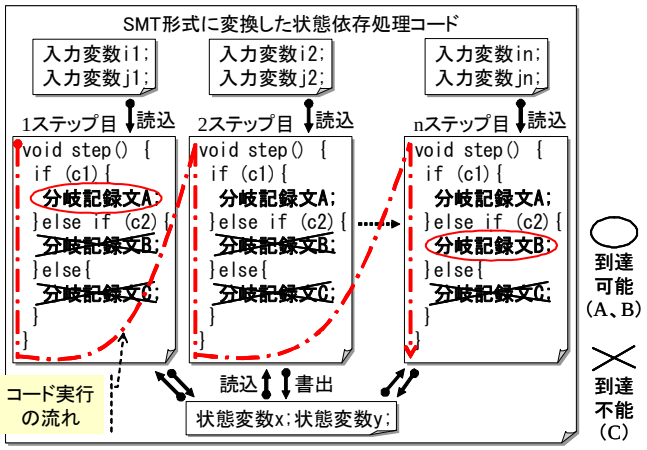


図 9　SMT solver を用いた入力変数値の解析

最後に，分岐記録文ごとに求まった入力変数値を並べて，時系列入力テストケースを生成する．分岐記録文 A と B がそれぞれ 1 ステップ目と n ステップ目で到達可能となって時系列入力を生成し，分岐記録文 C が到達不能となった例を図 10 に示す．

| 分岐記録文 | 到達可能／到達不能 | 入力変数i1<br>入力変数j1 | 入力変数i2<br>入力変数j2 | ・・・ | 入力変数in<br>入力変数jn |
|---|---|---|---|---|---|
| A | 可能 | i1の値<br>j1の値 | ― | ・・・ | ― |
| B | 可能 | i1の値<br>j1の値 | i2の値<br>j2の値 | ・・・ | inの値<br>jnの値 |
| C | 不能 | ― | ― | ― | ― |

図 10　到達可能／到達不能判定と時系列入力生成の例

## 5. 制御ソフトウェアへの適用例

### 5.1 適用対象の制御モデルおよびコード

制御ソフトウェアの一例であるクルーズコントロールに対する本方式の適用例を示す．題材として，Simulink に付属の制御モデルおよび Embedded Coder を用いてこのモデルから自動生成した実装コードを用いる．

クルーズコントロールとは，自動車に搭載される制御装置（コントローラ）が制御対象（スロットル）からのフィードバックを受けて，アクセルペダルを踏み続けることなく定速走行するように制御する機能である．クルーズコントロールモデルの入出力信号を表 6 に示す．

表 6　クルーズコントロールモデルの入出力信号

| 分類 | 信号名 | 説明 |
|---|---|---|
| 入力 | enable | クルーズコントロール機能の ON／OFF |
| | brake | ブレーキ踏み込みの有無 |
| | speed | 車速の現在値 |
| | set | 車速制御の停止／開始 |
| | inc | 車速の目標値のインクリメント |
| | dec | 車速の目標値のデクリメント |
| 出力 | throt | スロットルの開閉量 |
| | target | 車速の目標値 |

クルーズコントロールモデルの制御ロジックは以下のとおり．

- 機能 ON 指示の後，車速制御の開始指示により，車速の目標値での定速走行を制御．
- 制御中に，ブレーキの踏み込みを検知した場合は，制御を停止．
- スロットルの開閉量は PI 制御により演算．

上記の入出力信号および制御ロジックを実現した Simulink モデルを図 11，図 12 に示す．

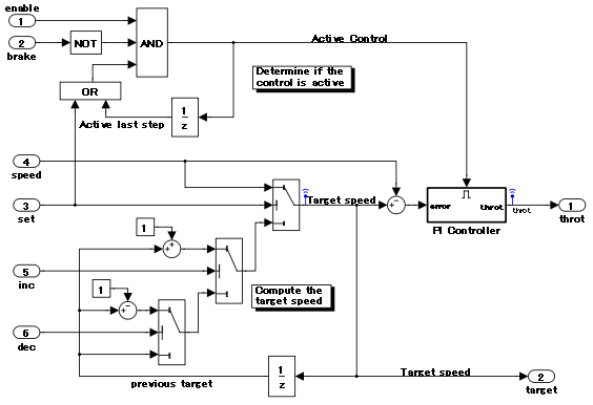


図 11　Simulink モデル一全体

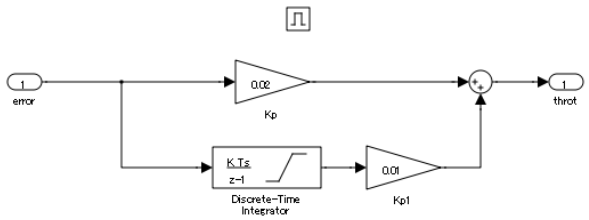


図 12　Simulink モデルーPI Controller サブシステム

図 11，図 12 の Simulink モデルから，Embedded Coder を用いて自動生成した実装コードを図 13，図 14 に示す．図 13，図 14 では入力変数（ex_cruise_controller_U），出力変数（ex_cruise_controller_Y），状態変数（ex_cruise_controller_DWork），C 言語関数（ex_cruise_controller_step）のみ掲載し，他の部分（コメント，include 文，他の関数など）は省略した．

```
/* Block states */
D_Work_ex_cruise_controller ex_cruise_controller_DWork;

/* External inputs */
ExternalInputs_ex_cruise_contro ex_cruise_controller_U;

/* External outputs */
ExternalOutputs_ex_cruise_contr ex_cruise_controller_Y;

/* Model step function */
void ex_cruise_controller_step(void)
{
 boolean_T rtb_ActiveControl;
 real_T rtb_Sum1;
 real_T rtb_Switch3;

 if (ex_cruise_controller_U.set) {
  rtb_Switch3 = ex_cruise_controller_U.speed;
 } else {
  if (ex_cruise_controller_U.inc) {
   rtb_Switch3 = 1.0 + ex_cruise_controller_DWork.UnitDelay_DSTATE;
  } else {
   if (ex_cruise_controller_U.dec) {
    rtb_Switch3 = ex_cruise_controller_DWork.UnitDelay_DSTATE -
1.0;
   } else {
    rtb_Switch3 = ex_cruise_controller_DWork.UnitDelay_DSTATE;
   }
  }
 }
 rtb_Sum1 = rtb_Switch3 - ex_cruise_controller_U.speed;
 rtb_ActiveControl = (ex_cruise_controller_U.enable &&
                      (!ex_cruise_controller_U.brake) &&
                      (ex_cruise_controller_U.set ||
                  ex_cruise_controller_DWork.UnitDelay1_DSTATE));
```

図 13　Simulink モデルから自動生成したコード（1/2）

```
 if (rtb_ActiveControl) {
  ex_cruise_controller_Y.throt = 0.02 * rtb_Sum1 + 0.01 *
   ex_cruise_controller_DWork.DiscreteTimeIntegrator_DSTATE;
  ex_cruise_controller_DWork.DiscreteTimeIntegrator_DSTATE = 0.01
* rtb_Sum1 +
   ex_cruise_controller_DWork.DiscreteTimeIntegrator_DSTATE;
  if (ex_cruise_controller_DWork.DiscreteTimeIntegrator_DSTATE >=
5.0) {
   ex_cruise_controller_DWork.DiscreteTimeIntegrator_DSTATE = 5.0;
  } else {
   if (ex_cruise_controller_DWork.DiscreteTimeIntegrator_DSTATE <=
-5.0) {
    ex_cruise_controller_DWork.DiscreteTimeIntegrator_DSTATE    =
-5.0;
   }
  }
 }
 ex_cruise_controller_Y.target = rtb_Switch3;
 ex_cruise_controller_DWork.UnitDelay_DSTATE = rtb_Switch3;
 ex_cruise_controller_DWork.UnitDelay1_DSTATE = rtb_ActiveControl;
}
```

図 14　Simulink モデルから自動生成したコード（2/2）

### 5.2 本方式で生成したテストケース

4 章で説明したアルゴリズムのフローチャートに沿って，中間形式も含めて図 13，図 14 の実装コードを変換した結果と，本方式で生成したテストケースについて説明する．

図 13，図 14 の実装コードから変換した状態依存処理コードを図 15，図 16 に示す．コードの先頭に，2 ステップ目以降の入出力変数（ex_cruise_controller_U02 など）を新たに定義しており，状態変数（ex_cruise_controller_DWork）の定義は変更していない．C 言語関数（ex_cruise_controller_step）の中の「BUNKI_0」「BUNKI_1」などが，実装コード中の分岐箇所を自動的に識別して挿入した分岐記録文である．C 言語関数は，1 ステップ目から順々に複数ステップ分複製して連結したものになっている．

```
D_Work_ex_cruise_controller ex_cruise_controller_DWork;


ExternalInputs_ex_cruise_contro ex_cruise_controller_U;
ExternalOutputs_ex_cruise_contr ex_cruise_controller_Y;


ExternalInputs_ex_cruise_contro ex_cruise_controller_U02;
ExternalOutputs_ex_cruise_contr ex_cruise_controller_Y02;
（中略）
```

図 15　状態依存処理コード（抜粋）（1/2）

```
void ex_cruise_controller_step(void)
{
 boolean_T rtb_ActiveControl;
 real_T rtb_Sum1;
 real_T rtb_Switch3;
 if (ex_cruise_controller_U.set) {
  BUNKI_0;
  rtb_Switch3 = ex_cruise_controller_U.speed;
 } else {
  BUNKI_1;
  if (ex_cruise_controller_U.inc) {
   BUNKI_2;
   rtb_Switch3 = 1.0 +
    ex_cruise_controller_DWork.UnitDelay_DSTATE;
  } else {
(以降省略)
```

図 16　状態依存処理コード（抜粋）(2/2)

図 15，図 16 の状態依存処理コードを変換した SMT 形式を図 17，図 18 に示す．先頭の「declare-fun」が変数宣言であり，SSA 形式での世代管理により，変数名に世代番号が付加されている(BUNKI_0_10, rtb_Switch3_11 など)．「assert」で囲まれている部分が制御演算であり，「=>」とこれに続く部分が if 文と条件文，その後の「and」で結合された部分が一連の命令文を表す．制御演算および条件文では，型は Int, Real, Bool, およびこれらの配列に変換し，演算子は算術演算子（+，-，*，/，-(負)），関係演算子（<，<=，>，>=，=，distinct），論理演算子（and，or，not），代入演算子（=），ビット演算子（&，|，^，<<，>>，~）に変換する．

```
(set-logic AUFNIRA)
(declare-fun D_3683_9 () Int)
(assert (<= 0 D_3683_9))
(declare-fun BUNKI_0_10 () Int)
(declare-fun ex_cruise_controller_U.speed () Real)
(declare-fun rtb_Switch3_11 () Real)
(declare-fun rtb_Switch3_2 () Real)
(中略)
(assert (
  = D_3683_9 ex_cruise_controller_U.set
))
(assert (
=>  (distinct D_3683_9 0)
   (and
        ( = BUNKI_0_10 0 )
        ( = rtb_Switch3_11 ex_cruise_controller_U.speed )
        ( = rtb_Switch3_2 rtb_Switch3_11 )
   )
))
```

図 17　SMT 形式（抜粋）(1/2)

```
(assert (
=>  (= D_3683_9 0)
   (and
        ( = BUNKI_1_12 0 )
        ( = D_3687_13 ex_cruise_controller_U.inc )
   )
))
(assert (
=>  (and (= D_3683_9 0) (distinct D_3687_13 0))
   (and
        ( = BUNKI_2_14 0 )
        ( = rtb_Switch3_15 ( + UnitDelay_DSTATE_6 1.0 ) )
        ( = rtb_Switch3_2 rtb_Switch3_15 )
   )
))
(以降省略)
```

図 18　SMT 形式（抜粋）(2/2)

図 17，図 18 の SMT 形式中の分岐記録文 (BUNKI_0_10 など)ごとに，分岐箇所への到達可能／到達不能を解析し，到達可能の場合は入力変数値を求める．SMT solver は，商用利用可能な CVC3[10]を使用する．SMT solver で解析した結果，到達不能箇所はなく，生成した時系列入力テストケースを図 19 に示す．各ステップの入力変数値の組は，(brake, dec, enable, inc, set, speed) を表す．テストケース数は 7，最長 2 ステップで分岐カバレッジが 100%という解析結果となった．

| No | 1 ステップ目 | 2 ステップ目 |
|---|---|---|
| 1 | (0, 0, 0, 0, 1, 0) | － |
| 2 | (0, 0, 0, 0, 0, 0) | － |
| 3 | (0, 0, 0, 1, 0, 0) | (0, 0, 0, 0, 1, 0) |
| 4 | (0, 0, 1, 0, 1, 0) | (0, 0, 0, 0, 1, 0) |
| 5 | (0, 0, 1, 0, 1, 0) | (0, 0, 1, 0, 0, -501) |
| 6 | (0, 0, 1, 0, 1, 0) | (0, 0, 1, 0, 0, 501) |
| 7 | (0, 1, 0, 0, 0, 0) | (0, 0, 0, 0, 1, 0) |

図 19　生成した時系列入力テストケース

本方式のアルゴリズムの正しさを確認するため，図 19 の時系列入力テストケースを図 14 の実装コードに実際に入力して動作させ，GNU gcov を用いて分岐カバレッジ率を測定した．その結果，分岐カバレッジ率は 100%となり，アルゴリズムの基本的な正しさを確認できた．

## 6. 提案方式の評価

### 6.1 状態依存コード変換の評価

Simulink モデルから自動生成した C 言語関数を対象に，複数ステップにわたる状態依存処理を模擬する一つの関数に変換するアルゴリズムを開発した．また，ループ処理については，一般にコーディング規約でループ回数が制限されることから，回数上限つきで変換可能とした．

これにより，過去の入力履歴や制御対象からのフィード

バックに基づいた制御演算が必須な制御ソフトウェアの解析が可能となった．また，状態依存コード変換はツール実装して自動化したので，開発担当者の作業負担の軽減が見込める．

一方，Simulink モデルからの自動生成に限定したとしても，ポインタ処理を含む C 言語関数が生成されることがあるので, これに対応したアルゴリズムの改良が必要である．

### 6.2 コード制御構造解析の評価

6.1 で変換した状態依存処理コードを，形式検証の一種である SMT solver 用コードにさらに変換し，分岐カバレッジ 100%を達成する時系列入力テストケースを生成するアルゴリズムを開発した．また，実装コード中の到達不能箇所(デッドコード)をもれなく特定することを可能とした．

これにより，制御ソフトウェアの開発効率化・高品質化の基盤となる単体・S/W 結合試験で重要な，実装コードの構造を網羅するテストケースの作成が可能となった. また，6.1 と同様にツール実装して自動化した．

一方，今回は題材を用いたアルゴリズムの基本的な正しさの確認に留まっている．生成したテストケースの必要十分性，および解析処理効率の評価が必要である．

### 6.3 今後の課題

6.1，6.2 の評価結果から以下の改善点を抽出した．

(a) 状態依存コード変換については，Simulink で自動生成されるポインタ処理を含む C 言語関数に対応し，実開発に適用する際の有効性を向上させる．

(b) コード制御構造解析については，テストケースの必要十分性と解析処理効率の評価を進め，テストケース作成工数の削減効果を明確にする．

## 7. おわりに

本稿では，制御ソフトウェアの開発効率化・高品質化の基盤となる単体・S/W 結合試験の取り組みとして，制御ソフトウェアに必須の状態依存処理を対象に，実装コード中の分岐箇所に到達するための入力変数値を解析して時系列入力のテストケースを自動生成する方式を提案した．

具体的には，Simulink モデルから自動生成した C 言語関数を対象に，複数ステップにわたる状態依存処理を模擬する一つの関数に変換するアルゴリズムを開発した．また，形式検証の一種である SMT solver を用いて，分岐カバレッジ 100%を達成する時系列入力テストケースを生成するアルゴリズムを開発した．

これにより，過去の入力履歴や制御対象からのフィードバックに基づいた制御演算が必須な制御ソフトウェアの解析が可能となった．また，単体・S/W 結合試験で重要な，実装コードの構造を網羅するテストケースの作成が可能となった．今回開発したアルゴリズムはツール実装して自動化したので，開発担当者の作業負担の軽減が見込める．

この評価結果を踏まえ，今後は Simulink で自動生成されるポインタ処理を含む C 言語関数への対応，テストケースの必要十分性と解析処理効率の評価に取り組む予定である．

## 参考文献

1) Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, Salvatore Sabina, “Automatic Test Generation for Coverage Analysis of ERTMS software,” *ICST 2009*, 2009.
2) Edmund Clarke, Daniel Kroening, Flavio Lerda, “A Tool for Checking ANSI-C Programs,” *TACAS 2004*, vol.2988 of LNCS, 2004.
3) Lucas Corderio, Bernd Fischer, Joao Marques-Silva, “SMT-Based Bounded Model Checking for Embedded ANSI-C Software,” 2011.
4) “GCC online documentation,” http://gcc.gnu.org/onlinedocs/, 3.9 Options for Debugging Your Program or GCC
5) 中田育男，*コンパイラの構成と最適化 第2 版*, 朝倉書店, 2009.
6) Alessandro Armando, “Building SMT-based Software Model Checkers: an Experience Report,” *FroCoS 2009*, 2009.
7) Alessandro Armando, Jacopo Mantovani, Lorenzo Platania, “Bounded Model Checking of Software using SMT Solvers instead of SAT Solvers,” *13th SPIN Workshop*, vol.3952 of LNCS, 2006.
8) Clark Barrett, Aaron Stump, Cesare Tinelli, *The SMT-LIB Standard Version 2.0*, http://www.smtlib.org/, 2010.
9) David R. Cok, *The SMT-LIBv2 Language and Tools: A Tutorial Version 1.1*, http://www.smtlib.org/, 2011.
10) http://cs.nyu.edu/acsys/cvc3/