6A-02

ソースコードの部分的な等価性検証による等価性判定精度向上手法

吉田 三喜也† 米持 一樹† 伊藤 益夫†
三菱電機株式会社 情報技術総合研究所†

1. はじめに

近年、情報技術の発展により、ソフトウェアで様々な機能が実現されている[1]。これに伴い、ソフトウェアの開発工数が増加している。一方で、ソフトウェアの論理的な演算内容を変更せずに、ソースコードを書き換えるリファクタリングを行い、ソースコードの再利用性を高めることでソフトウェアの開発工数を削減することがある。開発プロセスにおいて、ソースコード変更時の影響範囲を把握するために、等価性検証を実施する必要がある。等価性検証では、ソースコードの変更前後で論理的な演算内容が一致することを判定する。従来では、2 つのソースコードを指定し、変更などによって生じたテキストの差分を判別するツール [2]を利用する方法がある。しかし、テキストの差分では、論理的な演算内容を変更せずに修正したソースコードに対しても不等価と判定するため、不等価と判定する範囲が大きくなる問題がある。

そこで本稿では、モデル検査手法を利用して、検証対象のソースコードにおいて、関数単位で等価性検証を実施し、加えて検証対象の関数内で呼び出される関数の等価性を利用することで、等価と判定する範囲を拡大する等価性検証手法を提案する。ただし、提案する等価性検証は、機能変更による実装変更にも適用可能であるが、本稿ではリファクタリングを適用対象とする。

2. 関数ごとの等価性検証における問題設定

図 1 に等価性検証対象の関数（以下、被検証関数）を示す。図 1 において、左側の関数を変更前のソースコード（制御モデルx）、右側の関数を変更後のソースコード（制御モデルy）とする。また、図 1 に示した関数のコールグラフを作成すると図 2 となる。ソースコード内の論理的な演算内容が一致することを関数ごとに検証する方法[3]があり、この手法では、生成したコールグラフの末端関数から順番に等価性検証を実行している。そのため、図 1 の被検証関数内で呼び出す関数の等価性は既知である。ここで、被検証関数が呼び出す関数の等価性を表 1 とする。

```
void Func_B_x( int m_x ){
    int n_x;
    if( m_x > 0 )
      n_x = Func_D_x();
    else
      n_x = Func_E_x();
    return n_x;
}
```

```
void Func_B_y( int m_y ){
    int n_y;
    if( m_y > 0 )
      n_y = Func_D_y();
    else
      n_y = Func_E_y();
    return n_y;
}
```

図 1：被検証関数のソースコード

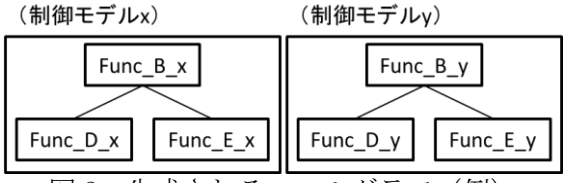

図 2：生成されるコールグラフ（例）

表 1：末端関数の等価性

制御モデルx	制御モデルy	等価性
Func_D_x	Func_D_y	等価
Func_E_x	Func_E_y	不等価

等価性判定基準が「同じ入力値に対して、常に同じ出力が得られること」であるため、表 1 の場合、図 1 の被検証関数は、不等価関数と判定される。図 1 と表 1 からわかるように、被検証関数への入力値によって、通過する分岐が異なり、等価関数を通る分岐では、出力値が一致する。一方で、不等価関数を通る分岐では、出力値が異なる。従来手法[3]では、被検証関数内部の分岐を考慮せず、全ての入力値に対して出力値が一致することを等価性の判定基準とするため、不等価関数と判定される。

この場合、例えば、意図せず論理的に演算内容が変更した場合の不具合解析において、解析対象範囲が広く、解析に費やす時間が増加する可能性がある。

「A Method to Improve the Accuracy of Equivalence Determination using Equivalence Checking for Each Section in Source Code」

†Mikiya Yoshida, Kazuki Yonemochi, Masuo Ito
Mitsubishi Electric Corporation, Information Technology R&D

3. 被検証関数内の分岐ごとに等価性検証を実施する提案手法

2 章で示した問題を解決するために、一度不等価と判定された関数に対して、分岐ごとに等価性検証を実施する方法を提案する。この提案手法の等価性検証には、有界モデル検査ツールの CBMC[3]を利用する。提案手法の等価性検証手順は次に示す(1)～(6)のとおり。

(1) 検証対象のソースコードのコールグラフを生成する。以降、生成したコールグラフにおいて、末端の関数から上位の関数に向かって、関数ごとに等価性検証を実施する。

(2) ソースコードを解析し、分岐箇所に識別子を付与する。この識別子と被検証関数内で呼び出す関数を関連付けする。
分岐箇所に付与する識別子は、その分岐箇所の通過の有無を判別できるように図 3 のように設定する。これにより、(4)で説明する CBMC での事前条件の追加を可能にする。一方で、表 2 は識別子と関数を関連付けした例を表す。

(3) 各被検証関数への入力値が一致することを事前条件、各被検証関数から得られる出力値が一致することを事後条件として、等価性検証プログラムに追加する。
図 4 は等価性検証プログラムであり、(3)は CBMC を用いた際の事前条件、事後条件の例である。図 4（(4)を除く）の等価性検証プログラムにおいて、全ての入力に対して事後条件を満たす場合、等価な関数、満たさない場合、不等価な関数と判定される。この検証結果が不等価な関数の場合、(4)に進み、それ以外の場合は等価性検証を終了する。

(4) 不等価な被検証関数内で呼び出される不等価な関数を通った場合に、その出力値を等価性検証に利用しない条件を事前条件に加える。
図 4 の(4)は CBMC で等価性検証をする際に、識別子を用いて事前条件を追加した例である。

(5) (4)で追加した事前条件を含み等価性検証プログラムで、再度等価性検証を実施する。

(6) (5)で実施した等価性検証結果が等価関数の場合、被検証関数を“部分等価関数”と判定する。この部分等価関数とは、被検証関数への入力値により被検証関数内の処理が分岐し、この分岐によって等価と不等価が存在する関数を指す。一方で、不等価関数と判定された場合、被検証関数は不等価関数とする。

上記に示した等価性検証を実施することで、被検証関数内の分岐ごとに等価性を判定でき、等価と判定する範囲を拡大できる。

```
void Func_B_x( int m_x ){
    int n_x;
    if( m_x > 0 ){
     //(2) 識別子付与
     count1_x++;
     n_x = Func_D_x();
    }else{
     //(2)識別子付与
     count2_x++;
     n_x = Func_E_x();
    }
    return n_x;
}
```

```
void Func_B_y( int m_y ){
    int n_y;
    if( m_y > 0 ){
     //(2) 識別子付与
     count1_y++;
     n_y = Func_D_y();
    }else{
     //(2) 識別子付与
     count2_y++;
     n_y = Func_E_y();
    }
    return n_y;
}
```

図 3：分岐箇所への識別子設定

表 2：識別子と関数の対応関係

制御モデルx	識別子	制御モデルy	識別子
Func_D_x	count1_x	Func_D_y	count1_y
Func_E_x	count2_x	Func_E_y	count2_y

```
void check_Func_B(int input_x, int input_y){
    int output_x, output_y;

    //(3) 入力一致条件
    __CPROVER_assume(input_x == input_y);

    output_x = Func_B_x(input_x);
    //(4) 等価性判定範囲制限
    __CPROVER_assume(count2_x == 0);

    output_y = Func_B_y(input_y);
    //(4) 等価性判定範囲制限
    __CPROVER_assume(count2_y == 0);

    //(3) 出力一致条件
    assert(output_x == output_y);
}
```

図 4：CBMC における等価性検証プログラム

4. おわりに

本稿では、被検証関数内の分岐ごとに等価性検証を実施する条件を加える、等価性検証方法を提案した。これにより、ソースコード変更時の影響範囲の粒度を細かく把握でき、リファクタリングにおいては、意図せず変更された論理的な演算箇所の特定が容易になり、不具合の解析にかかる時間削減が期待できる。

参考文献

[1]山口，他：“自動車制御ソフトウェア開発プロセスへのモデル検査の適用”，組み込みシステムシンポジウム 2012

[2] http://www.gnu.org/software/diffutils/

[3] Rupak Majumdar: “Compositional Equivalence Checking for Models and Code of Control Systems”, *52nd IEEE Conference on Decision and Control*, December 2013

[4] http://www.cprover.org/cbmc/