テストとホーア論理によるプログラムの形式検証手法の支援ツールの提案

著者	福岡 真吾
出版者	法政大学大学院情報科学研究科
雑誌名	法政大学大学院紀要．情報科学研究科編
巻	14
ページ	1-4
発行年	2019-03-31
URL	http://doi.org/10.15002/00021914

テストとホーア論理によるプログラムの形式検証手法の支援ツールの提案

Development of The Supporting Tool for Testing-Based Formal Verification

福岡　真吾
Shingo Fukuoka
法政大学情報科学研究科情報科学専攻
E-mail: shingo.fukuoka.8b@cis.k.hosei.ac.jp

Abstract

Software development is costly endeavors. In general, the development cost can be reduced by checking whether the program meets the specification. Mostly, Software is composed of several modules so that by checking the correctness of each module, developers can find the causes of errors efficiently. Formal verification and specification-based testing are widespread techniques to verify programs. Formal verification based on Hoare logic can establish the correctness of programs from the theoretical point of view. However, it is regarded as an impractical technique for realistic programs, due to some challenges. On the other hand, specification-based testing is able to detect errors, and it is quite easy to perform. Therefore, it is frequently used for realistic developments. However, in most cases, the testing cannot guarantee the correctness of programs. As we described above, both of these techniques cannot do satisfactory job alone. To solve this problem, a novel verification approach was suggested, which is called testing-based formal verification (TBFV). In this paper, we aim to reveal the feasibility of TBFV through developing a supporting tool for Java programs and conducting a case study. As a result, our supporting tool has achieved a semi-automatic application of TBFV, which can help reduce the cost of a verification process.

1. まえがき

形式仕様と実装に対し，プログラムが仕様に沿って正しく実装されているかどうかはソフトウェア信頼性の面から考慮すると，解決すべき非常に重要な課題の１つです．本論文では，劉が提案したソフトウェア検証手法である，Testing-Based Formal Verification (TBFV) [1]に着目し，その支援ツールを開発することで，TBFV の有効性を示すことを目的としている．同時に，提案している支援ツールでは境界値テスト[2]と組み合わせることで，TBFV の半自動的な適応による実用化の向上を目指している．提案している支援ツールを評価するため，小規模なプログラムに対し，支援ツールを用いて検証を行うことで有用性を示す．

2. Testing-Based Formal Verification

最初に，支援ツールの根底にある TBFV の仕組みを説明する．TBFV の機能は大きく次の 3 つの段階を経ることで実現されている．

2.1. Derivation of traversed paths

この部分では，Functional Scenario-Based Testing(FSBT)[3]に基づきテストケースを生成し，テストを実行する．プログラム実行中に，テストプログラム実行によって通ったステートメントを記録する．

FSBT はブラックボックステストの一種で，形式仕様の事前条件と事後条件を基にテストケース生成を行う．具体的には，事前条件と事後条件を functional scenario の積和形として扱いテストケースを生成する．functional scenario は以下のように定義される．

Definition.

形式仕様S の事後条件S_{post}を，$S_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \cdots \vee (C_n \wedge D_n)$としたとき，$C_i\,(i \in \{1, \ldots, n\})$を，"guard condition"とする．これらはいかなる出力変数も含まない．D_i を"defining condition"とし、少なくとも 1 つ以上の出力変数を含み，いかなる guard condition も含まない．このとき形式仕様S の functional scenario である f_s は，論理積$\sim S_{pre} \wedge C_i \wedge D_i$, であり，$(\sim S_{pre} \wedge C_1 \wedge D_1) \vee (\sim S_{pre} \wedge C_2 \wedge D_2) \vee \cdots \vee (\sim S_{pre} \wedge C_n \wedge D_n)$ は functional scenario form(FSF)となる．[3]

テストケース生成の際には出力の値は関係なく，入力変数のみを考慮する必要がある．functional scenario はそのままの形では出力変数を含んでしまっている．それゆえ FSBT では，functional scenario の各積の中から入力変数だけを抜き出す．この状態を test condition と呼び，以下のように定義される．

$(\sim S_{pre} \wedge C_1) \vee (\sim S_{pre} \wedge C_2) \vee \cdots \vee (\sim S_{pre} \wedge C_n)$,

これらの各積のことを test condition と呼び，FSBT において，各 test condition に対して少なくとも 1 つ以上のテストケースを生成する必要がある．

Supervisor: Prof. Shaoying Liu

TBFV がテストケース生成法として，FSBT を採用している理由は，形式仕様が well-formed である場合，複数の functional scenario を同時に考慮する必要がなく，各 functional scenario の検証を個別に行えるからである．形式仕様が well-formed である場合， functional scenario がそれぞれ独立して機能を担うことが保証されるため，事前条件を満たすいかなる入力に対しても，出力が 1 つの functional scenario によって定義される．well-formed は以下のように定義される．

Definition.

形式仕様S の FSF を， $(\sim S_{pre} \wedge C_1 \wedge D_1) \vee (\sim S_{pre} \wedge C_2 \wedge D_2) \vee \cdots \vee (\sim S_{pre} \wedge C_n \wedge D_n) S_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \cdots \vee (C_n \wedge D_n)$ *としたとき，* S *が条件* $\left(\forall_{i,j \in \{1,\ldots,n\}} \cdot \left(\mathrm{i} \neq \mathrm{j} \Rightarrow (C_i \wedge C_j \Leftrightarrow false)\right)\right) \wedge (\sim S_{pre} \Rightarrow (C_1 \vee C_2 \vee \cdots \vee C_n \Leftrightarrow true))$ *を満たしたとき，形式仕様* S *は well-formed である．[3]*

2.2. Application of Hoare logic

この部分では，得られたパスと形式仕様に対してホーア論理の公理[4]を適用することで，パスの形式的検証を行う．

pをテストの実行により通過したパスと仮定すると，パスpが functional scenario に沿って正しく実装されているかを検証するために，TBFV は path triple を形成する．path triple は以下のように定義される．

$$\{\sim S_{pre}\} p \{C_i \wedge D_i\}$$

path triple は Hoare triple と同じ構造をとっている，しかしプログラム全体より 1 つのパスに特化したものである．これは事前条件がパス p の実行前に満たされるとき，事後条件が必ず満たされるということである．ホーア論理の公理を繰り返し適用することで，pre-assertion を得ることが出来る．得られた pre-assertion を含んだ path triple は以下のように表現される．

$$\{\sim S_{pre}(\sim x/x)\}\{p_{pre}(\sim x/x)\} p \{C_i \wedge D_i(\sim x/x)\}$$

ここでの $\sim S_{pre}(\sim x/x)$, $p_{pre}(\sim x/x) and\ C_i \wedge D_i(\sim x/x)$は，ホーア論理に基づき全ての入力変数~$x$ が適切な入力変数 x にそれぞれ代入された結果である．入力変数と更新された変数を混合しないように，x と ~x を区別している．本論文では単純化のために，これらの述語論理 $\sim S_{pre}(\sim x/x)$, $p_{pre}(\sim x/x)\ and\ C_i \wedge D_i(\sim x/x)$の代わりに $\sim S_{pre}$, $p_{pre}\ and\ C_i \wedge D_i$と記述する．

2.3. Evaluation result

この部分では，得られた path triple から評価すべき含意を形成，評価を行い，検証中のパスの正当性を判定する．

$$\sim S_{pre} \Rightarrow p_{pre}$$

この含意が正しいなら，このパスは path triple を満たしており正しい．この時，含意が単純なものであれば SAT などの ATP(Automated Theorem Proving)を駆使することで，この述語論理を解くことができる．しかしそれはこの含意が単純なものである場合に限り，複雑すぎるものは ATP では証明することはできない．それゆえ TBFV では自動採用しておらず，開発者による導出に委ねている．

3. 設計

TBFV の支援ツールは，大きく 3つのモジュールから構成され，TBFV と同様の構造をとっている．このシステムを SOFL[5]に基づき設計した．

3.1. Informal Specification

Informal Specification は， Functions(機 能)， Data Resources(データリソース)，Constraints(制約)の 3 つを定義することで設計できる

3.1.1. Functions

本システムの機能は大きく分けると，「通過パスの導出」「ホーア論理の適用」「結果の評価」の 3 つである．

「通過パスの導出」は，検証したいプログラムを実行したステートメントを記録するように加工し，その後検証したいパスを通るテストプログラムを実行することで，実際にテストにより通過したステートメントを抽出する．

「ホーア論理の適用」は，テストにより通ったステートメントに，それぞれホーア論理の代入の公理を適用することで，パスの正当性を証明する path triple を形成する．

「結果の評価」は，「ホーア論理の適用」において得られた path triple に対して，自動的なテストを行う．

3.1.2. Data Resources

本システムが利用するデータリソースは，「形式仕様」「実装したプログラム」「テストプログラム」「パストリプル」「評価結果」の 5 つである．

「形式仕様」は，SOFL の文法に従い，pre，post，ext などによって記述できる．「実装したプログラム」の検証に利用する．

「実装したプログラム」は，検証を行うプログラムである．

「テストプログラム」は，「実装したプログラム」の中で検証を行いたい部分の呼び出しを行うプログラムである．「テストプログラム」を実行することで，「代表パスの導出」の機能を実現している．

「パストリプル」は，ホーア論理の適用を行った結果得られる，証明すべき述語論理である．

「評価結果」は，「形式仕様」と「パストリプル」を基に，境界値テストを実施したテストの結果である．

3.1.3. Constraints

本システムの制約は，4 つである．

・形式仕様で利用できる変数の型は，SOFL の基本型の int,double,bool のみに限定される．

・プログラムで利用できる変数の型は， Java の boolean,int,double のみに限定される．

・テスト実行後は class ファイルを削除するため，テストの再実行を行うには再度コンパイルを行う必要がある．

・結果を自動的に評価するにはテストケースを少なくとも 1 つ実施できる必要がある．

3.2. Formal Specification

Formal Specification では，Informal Specification で定義した 1 つの Functions を 1 つの Process，あるいは Function として定義する． 1 つの Process，Function は，事前条件, 事後条件と共に定義され，Data Resource へのアクセスは ext で記述する．SOFL では，Formal Specification は CDFD と組み合わせて定義する． 図 1 は Formal Specification 上の path triple の形成モジュール (generate_path_triple)の例である．加えて，図 2 はシステム全体の CDFD，図 3 は path triple の形成を含むホーア論理の適用(apply_Hoare_logic)の CDFD である．

```
process
generate_path_triple(proceed_generate_path_triple:
sign)proceed_evaluate_result: sign
ext rd formal_specification:Specification =
            composed of
            pre_condition:string
            post_condition:string
            end
    rd application_result:string
    wr path_triple:string
post path_triple = “{” +
formal_specification.pre_condition + “}”+ “{” +
application_result + “}” + “{”
formal_specification_post_condition + “}”
end_process;
```

図 1.　Formal Specification 上の generate_path_triple

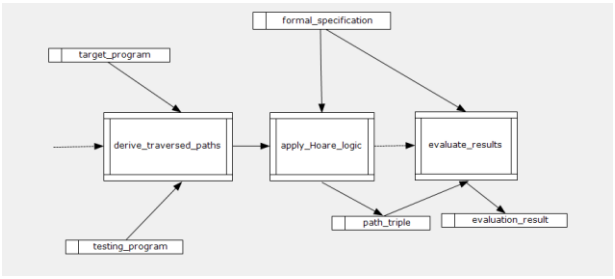

図 2.　システム全体の CDFD

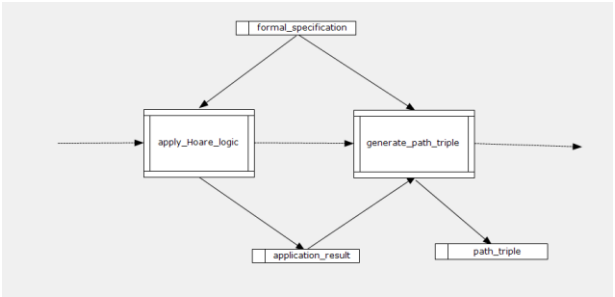

図 3.　apply_Hoare_logic の CDFD

4. 実装

支援ツールの開発言語は Java を使い，環境は eclipse で開発を行った．Java を使用している理由は，Java が最も成功し，頻繁に使われている言語の 1 つだからである．加えて，近年 Java の実行時の速度が C や C++などに比べても遜色ないほどに向上されていることもあり，スマートフォンや産業用ロボットなどの組み込みシステムに採用され始めている．それゆえ，Java を対象とした支援ツールを作ることが実践的で利益があると考えたためである．支援ツールの規模としては全体で 2700step ほどである．より実用的なものとするために GUI を実装した．ボタンクリックなどにより画面遷移できるよう実装した．図 4 の画面は，「ホーア論理の適用」の GUI である．

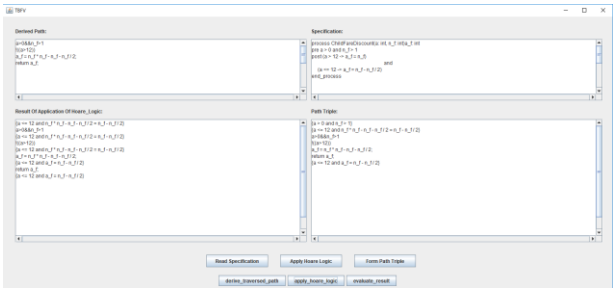

図 4.　「ホーア論理の適用」の GUI

5. 実験

本論文では，提案している支援ツールを評価するために具体的には 0 を含む自然数の偶奇を判定するプログラム(isEven)であり，その形式仕様を図 5 に示す．また，実装したプログラムを図 6 に示す．

```
process isEven(x : int) z : nat0
pre x >= 0
post x % 2 = 0 and z = 0 or x % 2 != 0 and z = 1
end process
```

図 5.　isEven の形式仕様

```
int isEven(int x) {
        if (x >= 0) {
                int z = 0;
                while (x > 1) {
                        x = x - 2;
                }
                if (x == 0) {
                        z = 0;
                        return z;
                } else {
                        z = 1;
                        return z;
                }
        } else {
                //"the pre condition is violated!");
                return 0;
        }
}
```

図 6.　isEven の実装プログラム

isEven の functional scenario は
(1) x >= 0 and x % 2 = 0 and z = 0
(2) x >= 0 and x % 2 = 1 and z = 1
(3) x < 0 and anything

の 3 つである．今回の実験では(1)の functional scenario を満たすように生成したテストケース{(x,8)}を使用する．このテストケースを実行するプログラムを図 7 に示す．

```
package TargetPrograms;

public class CaseStudyTestProgram{
    public static void main(String[] args){
        CaseStudyTargetProgramTest.isEven(8);
    }
}
```

図 7.　isEven のテストプログラム

6. 結果

実験の結果として，ホーア論理の公理の適用を行った結果を図 8 に示す．また，path triple の形成を行った結果を図 9 に示す．最後に，自動的な境界値テストを行った際の結果を図 10 に示す．

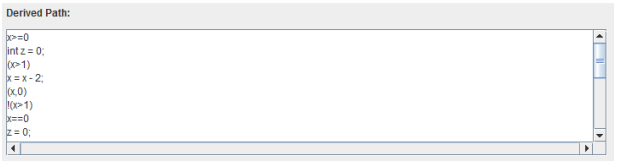

図 8.　ホーア論理の公理の適用結果

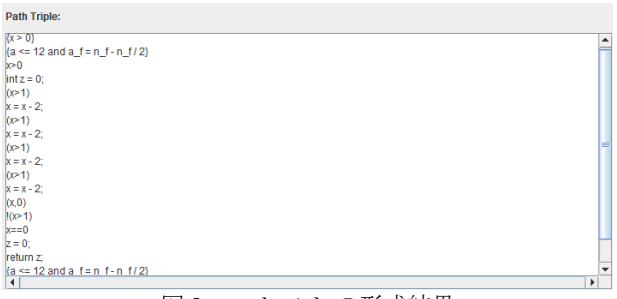

図 9.　path triple の形成結果

ここで，もし含意~$S_{pre} \Rightarrow p_{pre}$を証明できれば，このパス上にエラーがないことを証明できる． 具体的には，以下の含意を証明できればパスの正当性が得られる．

$$\{x \geq 0\} \Rightarrow \{(x-8)\,\%\,2 = 0\ and\ 0 = 0\},$$

実際にはこの含意を満たす必要のある入力 x の値は，このパスを可能性がある値のみに制限される．つまり今回の場合は，test condition である，x >= 0 and x % 2 = 0 and z = 0 を満たす値のみに制限される．

この条件のもと，境界線テストを行うため，自動的に作成したテストケースを図 10 に示し，それを実行した結果を図 11 に示す．

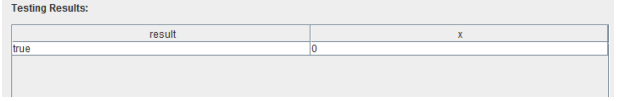

図 10.　境界値テストのテストケース

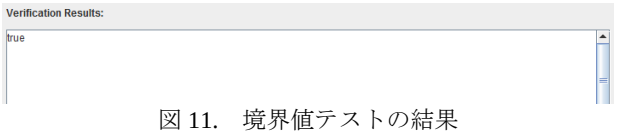

図 11.　境界値テストの結果

境界値テストは，検証を進める上での補助の役割を担っており，TBFV の検証は完了しない．path triple の導出やテスト結果を踏まえて，パスの正当性を証明する必要がある．今回は，以下の述語論理を証明する必要がある．

$\forall_{x<N\cup\{0\}}, x \geq 0\ and\ x\%2 = 0$，$(x-8)\%2 = 0$ and $0 = 0$

具体的には，

m を 0 を含む自然数とすると，任意の偶数は 2m と表現できる．このとき 2m - 8 は 2(m - 4)と表現できる．これは 2 で割り切ることが出来るため，上記の述語論理において，$(x-8)\%2 = 0$は常に真である．・・・(1)

0 = 0 は常に真である．・・・(2)

(1)，(2)より

$\forall_{x<N\cup\{0\}}, x \geq 0\ and\ x\%2 = 0$，$(x-8)\%2 = 0$ and $0 = 0$

は正しい述語論理であると結論付けられ，検証中のパスの正当性を得られる．

7. 考察

パスの正当性を得るためには，数学的な証明が必要であり，自動的な境界値テストだけでは弱すぎる検証である．しかしながら，結果に false が含まれていれば，プログラムの検証中のパス上に少なくとも 1 つ以上のエラーを含むことが分かる．それゆえ，この自動的なテストは検証を進めるにあたって有意義なものであると考える．

開発した支援ツールは現段階では Java における int, boolean，double の 3 つの基本的な方しか扱えず，ライブラリ対応が完了していないため複合型や列型などの複雑なデータ構造を取り扱うことが出来ない。他にもメソッド呼び出しでの返り値など，未だ 扱いきれていない部分も大きく，解決すべき課題が数多く残る．しかし現段階でも，path triple の導出における中間成果物生成やパスごとに分けた検証を行うことが出来るという点において，TBFV の実用性や有効性を示すことは出来たと考える．

文　献

[1] S. Liu, A Decompositional Approach to Automatic Test Case Generation Base on Formal Specification, SSIRI 2010, 147-155, Singapore, IEEE CS Press, 2010.

[2] S.C. Reid, An empirical analysis of equivalence partitioning, boundary value analysis and random testing, Proceedings Fourth International Software Metrics Symposium, IEEE, 1997.

[3] S. Liu, T. Hayashi, K. Takahashi, K. Kimura, T. Nakayama, and S. Nakajima, Automatic Transformation from Formal Specifications to Functional Scenario Forms for Automatic Test Case Generation, In 9th International Conference on Software Methodologies. Tools and Techniques, Japan, IOS International Publisher, 2010.

[4] C, A, R, Hoare. An Axiomatic Basis For Computer Programming. Communications of the ACM. Volume 12. Issue 10. 576-580，1969.

[5] S Liu, Formal Engineering for Industrial Software Development Using the SOFL Method, Springer-Verlag, 2004.