新 組み込みソフトへの数理的アプローチ
～形式仕様記述をどのように使うか～

第12回 テストとプログラム検証 ――TDD（テスト駆動開発）におけるテスト

藤倉 俊幸

前回（2010年4月号，pp.180-185）はソース・コードからプログラム検証ツールであるCBMCを使って事後条件を見つけ出し，さらにSMTソルバであるYicesを利用して事前条件を見つけ出す手法を紹介した．関数の仕様というのは，事前条件と事後条件であるから，ソース・コードから仕様をリバースしたといえる．

事前条件で重要なのは最弱事前条件といわれるもので，それを見つけ出す手順としてダイクストラ（Dijkstra）の方法とか，フロイド-ホーア論理（Floyd-Hoare Logic）とか，最近ではB-Methodなどがある．これらの手法を利用してソース・コードから仕様をリバースする作業に入る前に，今回はテストとプログラム検証やTDD（Test Driven Development；テスト駆動開発）について仕様記述という視点で類似性を調べてみる．

1 テストと検証の関係

● テスト・ケースは入力と出力の組

一般にテスト・ケースは，入力データと期待される出力の組によって定義される．入力データとは事前条件のインスタンスであり，期待される出力は事後条件のインスタンスなので，テスト・ケースを仕様と考えることは可能だろう．しかし，あくまでもインスタンスなので不安が残る．仕様というからにはどの程度をカバーしているのかが気になる．テストの目的をこのように仕様の記述の範囲に拡大すると，制約条件についても扱えるのではないかと思えてくる．つまり，制約条件を調べるためのプログラムを書いた場合は，それもテスト・プログラムの範疇（はんちゅう）に入りそうである．たとえば，実行環境のエンディアン（Endianness）を調べるプログラム（**リスト1**）や待ち行列がFIFOかLIFOか調べるプログラム[1]，OSのオーバヘッドを調べるプログラム[1][3]などである．

また一方で，このようなテスト・プログラムそのものに対してもテストが必要である．結果として，テストのテストが続くことになる．この連鎖を打ち切るために検証を利用する．CBMCを使用すると非常に手軽に検証できるので，リファクタリングなどを行う場合は変更のたびに検証することができる．**図1**に示したコマンドを打ち込むだけで，問題があればエラーの起こる条件とライン番号を教えてくれる．TDD（Test Driven Development；テスト駆動

リスト1　エンディアン判定プログラム

```
#define Const 1
union checker
{
int in;
unsigned char ch[4];
};

int endian_check()
{
    union checker En;
    En.in = Const;
    if (En.ch[0] == Const)
        printf("little\n");
    else
        printf("big\n");

    return 0;
}
```

```
>cbmc endian.c --function endian_check --bounds-check --div-by-zero-check --pointer-
check --overflow-check
file endian.c : Parsing
endian.c
Converting
Type-checking buffer
tmp.stdin2680.c
Generating GOTO Program
Function Pointer Removal
Pointer Analysis
Adding Pointer Checks
Starting Bounded Model Checking
size of program expression: 24 assignments
simple slicing removed 0 assignments
Generated 0 VCC(s), 0 remaining after simplification
VERIFICATION SUCCESSFUL
```

CBMCのバージョンが上がってチェック項目を個別に指定するように変更された

問題なし

図1　エンディアン・プログラムの検証