特集
フォーマルメソッドの新潮流 4

Part II：産業界への応用

C言語へのフォーマルメソッドの適用

宮崎義昭　橋本祐介

● 日本電気(株) ソフトウェアエンジニアリング本部

ソフトウェアの信頼性を確保するための技術として形式手法が注目されている．我々は，C言語で記述されたソースコードから状態遷移モデルを自動的に作成し，有界モデル検査法によりソースコードに内在する不具合を引き起こすコードの検出や，表明によるプログラムの正しさを検証するツールであるVARVELの開発を行った．本稿では，本ツールが利用している要素技術や提供する機能，およびソフトウェア開発現場における適用事例について報告する．

背景

ソフトウェアは，社会基盤を支える情報システムや，日常生活に不可欠なさまざまな製品に組み込まれ，その役割は大きくなっている．ソフトウェアが大規模かつ複雑になるにつれて，その信頼性や安全性をいかに確保するかは，社会的に重要な課題の1つとなっている．このような状況において，ソフトウェアの品質を確保する技術として形式手法，中でも特にモデル検査が注目を集めており，産業界においてもさまざまな研究や応用が進められている[1]．

一般に，ソフトウェア設計の検証にモデル検査技術を適用するには，設計情報である状態遷移モデルを特別な仕様記述言語で定義し，検証ツールでその妥当性を検証する手法が使われており，すでに多くの適用事例が報告されている[2]．しかしながら，これらの手法は，上流工程における設計検証を主な目的としており，最終的な成果物であるプログラムを定義したソースコードを検証する目的には適していない．また一般の開発者にとっては仕様記述言語の習得やモデルの作成に多くの作業時間を要するという課題がある．このためソースコードを形式検証可能なモデルに自動変換し，モデル検査や定理証明といった技術により自動検証を行う研究も行われてきた．たとえば，CMUのCBMC（C Bounded Model Checker）[3]，NASAのJava PathFinder[4]などの検証ツールが知られている．

我々も，モデル検査の適用範囲をソースコードの正しさを検証する領域まで拡張し，またソフトウェア開発工程の中で無理なく利用できる環境を提供したいとの目的で，C言語で記述されたソースコードを対象とした検証ツールVARVELの開発を行った．C言語は数多いプログラミング言語の中でも，組込みソフトウェアの領域を中心に広く普及している．また，C言語はプログラマがメモリ管理を行う必要があるなど，メモリやポインタに関する実行時エラーを起こしやすい言語である．VARVELは，C言語プログラムで不具合の原因になりやすい，無効なポインタによる参照や配列の範囲外アクセスなど実行時エラーにつながる可能性のある個所を検出することができる．また，利用者の定義した関数インタフェース仕様をもとにソースコードの妥当性検査を行う表明検証機能を有している．

本稿では，まず，VARVELが利用している要素技術について概説し，次に，VARVELの構成や検出できる不具合の種類など機能について説明する．また，実際のソフトウェア開発現場において本ツールを適用した事例について報告する．

要素技術

●形式手法

高信頼性の実現手段の1つに形式手法がある．形式手法（Formal Methods）とは，論理学・集合論・代数学などを基盤としたシステムの記述手法・検証手法などの総称であり，システムを厳密正確に記す形式仕様記述と，不具合のないことを数学的に証明する形式検証とに大きく分けられる[1), 2)]．形式検証の1つであるモデル検査法は，システムの状態遷移モデルを定義し，その振舞いの正しさを網羅的にチェックする手法であり，航空宇宙・通信・半導体など高い信頼性や安全性が要求される分野における設計の検証に利用されてきた．特に，有限の状態空間を対象とし，さらに探索範囲に制限を置く有界モデル検査法は，不具合を早期に発見する手法として注目されており，ソースコード検証へ適用するための研究も近年進められている（図-1）．弊社は，長年にわたりLSIの設計検証を目的にVeriSolというモデル検査エン

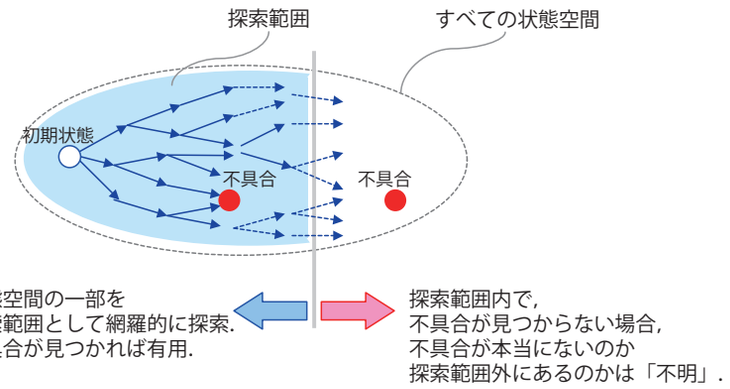

図 -1　有界モデル検査

ジンの開発を行っており[5]，VARVEL はこのエンジンをソースコードの検証に応用したツールである．

●ソースコード検証へのモデル検査の適用

ここでは，ソースコード検証に有界モデル検査法を適用する方法について説明する．VARVEL が利用している SAT（充足可能性判定）による有界モデル検査法では，有限な状態遷移モデルとプロパティ（検査したい性質）をそれぞれブール論理式として与え，これらの論理積を充足する変数値があるか否かを判定する．VARVEL の場合，処理の流れの概要は次のようになる（図 -2）．

①コントロールフローグラフ（CFG）への変換
② 状態遷移式への変換
③ Boolean Model への変換
④ 有界モデル検査の実行
⑤ 検証結果の表示

まず，複数のソースファイルをまとめて，制御の流れを表す CFG に変換する．この際に，エラーのチェックに必要な監視変数，監視変数をチェックするノード，エラー発生を表すノード，各ノードの識別に用いるプログラムカウンターという変数（図 -2 の pc）を追加する．次に，CFG を元にして，変数ごとに値の推移つまり状態遷移を表す式を作成する．これらの式の論理積が状態遷移モデルである．検査を行う上で，証明したいことは「常にエラーが発生しないこと」であるが，モデル検査エンジンには，その否定である「いつかエラーが発生する」＝「いつかプログラムカウンター変数の値がエラーノードの識別子となる」という時相論理式をプロパティとして与える（図 -2 のプロパティの式「F」は「いつか」を表す）．

有界モデル検査法により，エラー発生を示すプロパティが成り立つということは，状態遷移モデルとプロパティの双方を満たす変数の値が解けたということであり，解けた結果として分かる CFG 上の各ノードにおける変数の値を，ソースコード上の処理の開始からエラー発生個所までのトレースとしてユーザに提示することができる．なお，ソースコードを状態遷移モデルに変換する場合，ソースコード上のある位置において，各変数の値の 1 つの組合せを 1 つの状態と見なせばよいが，実際には次に述べる工夫が必要となる．

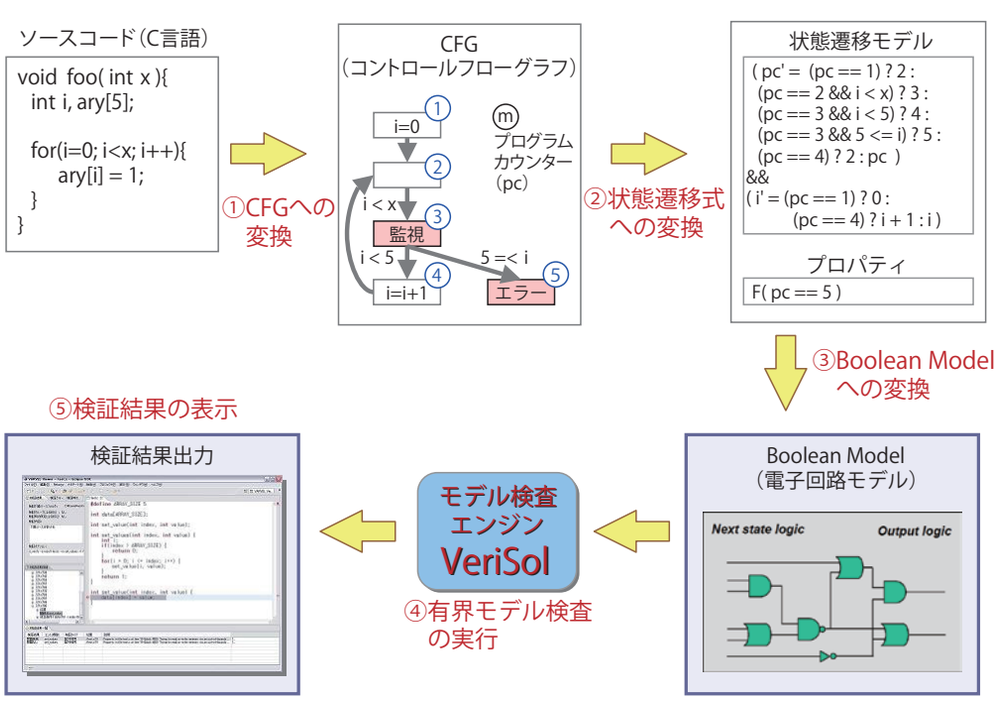

図 -2　配列上限チェックの場合の処理の流れ

■ 契約： 呼び出し側と呼ばれる側の関係を規定する厳密な条件群
- 事前条件：関数を呼び出す側が守るべき条件
- 事後条件：呼び出される関数が守るべき条件
- 不変条件：すべての関数呼び出しの前後で守られるべき条件

```
/**  総点数
  @invariant  0 <= totalScore ;
*/
int totalScore = 0 ;


/**  成績の問合せ
  @pre  0<=score && score <=100
  @post  A'<=__return && __return <= E'
*/
char queryGrade( int score )
{
    char  grade ;
    if( 90 < score ) grade =   'A'   ;
    else if( 75 < score ) grade =   'B'   ;
    else if( 60 < score ) grade =   'C'   ;
    else if( 45 < score ) grade =   'D'   ;
    else grade =   'E'   ;
    return  grade;
}
```

不変条件　@invariant
(静的な)初期化直後に満たされていて，
公開関数の実行（直前と）直後に
成立していなければならない条件．

事前条件　@pre
呼び出し側が成立を保証しなければならない条件．

事後条件　@post
(事前条件が保証された場合に)
呼ばれる側が成立を保証しなければならない条件．

図 -3　契約による設計

近似

ソースコードには再帰関数や動的にメモリ確保される配列などがあり，必ずしも状態空間を有限にできるとは限らない．また，モデル検査には，状態数に関して指数関数的に処理時間が長くなるという本質的な問題（状態爆発)がある．このため，再帰を固定回数までとする，あるいは配列要素は固定番目までとする，といった近似を行い，ソースコードを状態数の少ない有限状態空間に変換する必要がある．

仮定

検証対象である関数の引数や，この関数がアクセスする外部変数，この関数が呼び出す関数の戻り値や副作用については，与えたソースコードだけからは分からないことがある．このような場合には，引数などがとり得る値に仮定を設ける．任意の値をとると仮定した場合，問題は見逃さないが状態爆発が起きやすく，0 など特定の値を取ると仮定すると，状態爆発は抑えられるが，特定の問題のみを検出するようになる．

最適化

モデル検査を実用的な時間で行うためには，状態の数や遷移を少なくすることが必須である．そこで，コンパイラが行うような最適化を行い，不要な変数や処理を削除する．たとえば，検証したい性質に関して制御依存およびデータ依存のないコードを削除するスライシングや，定数の代入のみしかない変数やそのような変数の代入のみしかない変数を定数に置換する定数畳み込み等を行う．

契約による設計

プログラムが仕様として守るべき制約条件をソースコード中に表明(アサーション)として記述し，プログラムが意図したとおりに作られていることを確認する手法が利用されているが，特に，コンポーネント間の制約条件を互いが守るべき厳密な契約として捉え，契約を介してコンポーネントが強調するように設計する技法として「契約による設計」(Design by Contract ™) がある [6]．契約による設計では，呼び出し側と呼ばれる側の 2 つのコンポーネントに対し，コンポーネント間の責任分担を厳密に定義し，それを遵守させることで，ソフトウェア全体の品質，信頼性，および再利用性が確保できるとしている(図 -3)．

VARVEL では，この考え方を取り入れ，関数の事前条件・事後条件，外部変数の不変条件を表明として定義することで，関数インタフェースの正しさをモデル検査により検証する機能を提供している．

VARVEL の概要

●ツール構成

図 -4 が VARVEL の全体構成である．

(1) コマンド

C ソースコードを入力し検証を行う部分は Linux で動作するコマンドとして提供される．検証の結果は XML 形式のファイルとして出力され，ランチャやビューアに情報を提供する．

(2) ランチャ

ランチャはツール利用者向けの GUI である． 解析対象のソースコードの指定や解析コマンドに指示するオプ

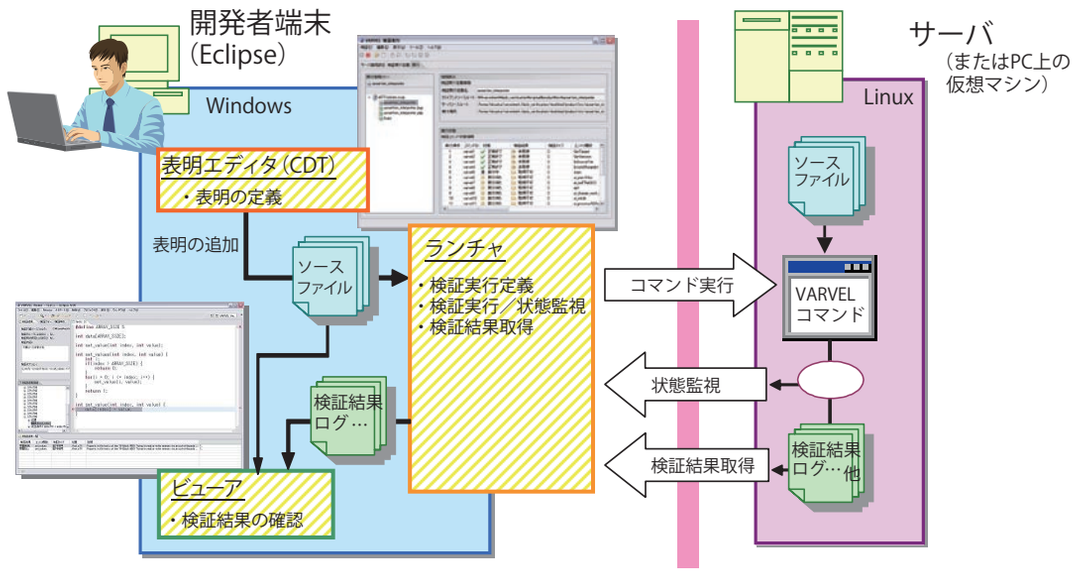

図 -4　VARVEL の全体構成

ションなどを設定し，コマンドの実行を制御する．動作 OS は Windows であり，Eclipse のプラグインとして実装されている．

(3) ビューア

ビューアは，コマンドが出力した検証結果を利用者が確認し，修正が必要な不具合かどうかを確認する機能である．検証結果の中には情報不足やモデル化の過程で行った仮定や近似に起因したいわゆる誤検出が含まれるため，それぞれの指摘事項に対して，ソースコードの該当個所を表示しながら，指摘内容とトレース情報を確認することができる．ランチャと同様に Eclipse プラグインとして提供されているため，ソースコードの編集環境と同様のビューで VARVEL による検証機能を利用することができる．

●検証タイプ

VARVEL が提供する検証の種類を検証タイプと呼んでおり，以下の 5 つがある．

- ポインタ有効性：0 番地の参照，開放済み領域の参照など，無効なポインタを利用したアクセスを検出する（図 -5）．
- 配列境界：配列境界の上限，下限アクセス違反を検出する（図 -6）．
- 文字列操作：文字列長と配列サイズの不整合によるバッファオーバーフロー／バッファアンダーフローなどを検出する．
- メモリ関連：malloc，calloc，free の呼び出し回数の不整合をチェックし，メモリリークなどの可能性を検出する．
- 表明検証：規定された文法で定義された表明をもとに，制約条件への適合性を検査し問題個所を検出する．

●表明検証

表明検証機能は，先に述べたように「契約による設計」の考え方に基づき，関数仕様を明確に定義することで，関数の動的な呼び出し関係においてインタフェース仕様に違反する事象が起きないことを検証する機能である．

表明検証を除く他の検証タイプが，ソースコードの処理ロジックを解析した情報だけで，実行時エラーの事象を検出するのに対して，表明検証は，検証する内容をツール利用者があらかじめ指定しておく必要があり，この点で他の検証タイプとは異なっている．

ソースコードから自動的にツールが解析し認識できる情報には限界があり，その情報に基づいた範囲では，実際のプログラム実行時には起こり得ない事象をエラーと検出するような誤検出がどうしても多くなる．そこで，VARVEL では，各関数が守らなければならないインタフェース仕様を定義するための表記法を規定し，この表記に沿ってツール利用者がソースコード中に（ソースコードとは別に定義することも可能）制約条件として記述しておく．これにより，的確な検査内容をツールに明示すると同時に，モデル構築に必要な情報を増やすことで，検証の精度を上げることができる（図 -7）．表明の文法は，ドキュメント化ツールとして広く利用されている Doxygen のコメント記法に準拠している．したがって，定義した関数インタフェース仕様はそのまま Doxygen により仕様書として出力し活用することができる．

表明として定義できる制約条件には以下の種類がある．

- 不変条件：スコープ内のすべての関数の開始時，終了時に満たすべき条件．

■ 無効なポインタによるメモリ参照の行を警告
- 「*ポインタ」の表現のある行で，ポインタが有効であることを検査
- 有効＝変数アドレス and 有効ポインタからポインタ演算で導出されるアドレス
- NULLを代入されたり，メモリ解放されたポインタは無効となる

問題なし

```
int array[]={1,2,3,4,5};
int* p=array; //変数アドレスの設定
p=p+4;        //ポインタ演算
*p=4;         //OK
```

```
p=malloc(sizeof(int)); //動的メモリ
if( p==null ) return;
*p=0;         //OK
```

問題発見

```
void setPointerNull( int** pp ){
  *pp = NULL;
}
void foo(){
  int* p;
  setPointerNull( &p );
  *p = 0; //NG; NULLポインタ参照
}
```

```
int* p=malloc(10);
if(p) free(p);
*p=0; //NG; 解放済みポインタによる参照
```

図 -5　ポインタ有効性

■ 配列の上限／下限を超えて配列要素へアクセスする行を警告．
- 添え字によるアクセス，ポインタによるアクセスのいずれも警告の対象とする．

問題なし

```
int array[]={1,2,3,4,5};
int i ;

for(i=0; i<sizeof(array), i++){
  array[i]=-1;//OK
              //添え字によるアクセス
}
```

```
#define SIZE 5
int* p=(int*)malloc(sizeof(int)*SIZE);
int i ;

for(i=0; i<SIZE; i++){
  *p =0;   //OK
           //ポインタによるアクセス
  p++;     //ポインタ演算
}
```

問題発見

```
#define SIZE 5
int i, array[SIZE];
int* p=(int*)malloc(sizeof(int)*SIZE);

for(i=0; i<SIZE; i++, p++)
{   array[i]=i;   *p = i;   }
array[i]=SIZE; //NG;上限違反( i==5 )
*p=SIZE;       //NG;上限違反( p==&array[5] )
```

```
#define SIZE 5
int i, array[SIZE];
int* p=(int*)malloc(sizeof(int)*SIZE);

for(i=SIZE-1; 0<=i; i--, p--)
{   array[i]=i;   *p = i;   }
array[i]=-1; //NG;下限違反 ( i==-1)
*p=-1;       //NG;下限違反 ( p==&array[-1])
```

図 -6　配列境界

- 事前条件／事後条件：特定の関数の開始時／終了時に満たすべき条件．
- 出力パラメータ：特定の関数の引数または大域変数のうち値が変更されるものを宣言する．
- assert/assume：ある特定の位置において，満たすべき条件(状態)を，検査条件あるいは前提条件として定義する．

●検証結果の出力

VARVELは，自動的に洗い出したすべての検証項目(プロパティ)に対して網羅的なチェックを行い，エラー事象の存在の有無により「問題発見」あるいは「問題なし」の判定結果をレポート出力する．ただし，モデル検査技術の特性として，検証項目によっては適切な時間内では検査が終了しない場合もあるため，あらかじめ指定された時間内に検査が終了しない場合は処理を打ち切り，当該の検証項目に対しては「不明」として結果を出力する．

また，VARVELによる解析結果にはエラー状態になる個所の行番号やエラーを引き起こした処理(変数への代入やポインタの参照など)の詳細が出力されると同時に，そのエラー発生個所へ到達するまでの実行パスがトレース情報として出力される．トレース情報には，エラー発生条件に関係する代入や分岐の状態が出力されているため，この情報を追いかけることで，デバッガを利用した動的テストを行う要領で，エラー発生の条件や原因を時系列に確認することができる(図 -8)．

適用事例

●社内プロジェクトへの適用

VARVELの有効性を確認するため，社内の複数のプロジェクトの協力を得て，開発中のソースコードを対象に，ツール試行適用を行った．すでに，開発の現場では，ソースコードの静的チェックツールを活用して，ソースコードから不具合の原因になりやすい記述や，コーディング規約に準拠していない個所の検出を行っている．しかし，これらのツールでは，変数の値や動的な実行パスを考慮したエラー検出はできないため，従来のレビューやテストでは摘出が難しかった不具合を検出する手段としてVARVELのような検証ツールには非常に関心が高い．

この試行で適用した検証タイプは，コーディング済みのソースコードに対してそのまま適用できる，実行時エラー系の検出機能である（表明検証は適用していない）．また，スクリーニングと呼ばれる，解析結果とソースコードを突き合わせて結果の妥当性を精査する作業に，ある程度のノウハウが必要であることから，専任の検証担当者が各プロジェクトからソースコードを入手し，スクリーニングした結果をプロジェクトに報告するという方式で行っている．もちろんプロジェクト外部の検証担当

- 表明を定義しておくことで，
  設計～コーディング～テストにおいて，関数呼び出しにおける契約違反を検証する
  - 呼ばれる側に関して，不変条件違反，事後条件違反を検出
  - 呼び出し側に関して，事前条件違反，不変条件違反を検出

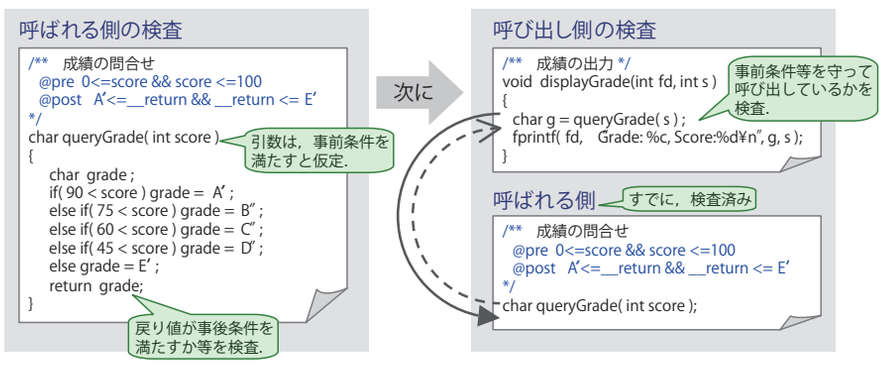

図 -7　表明検証

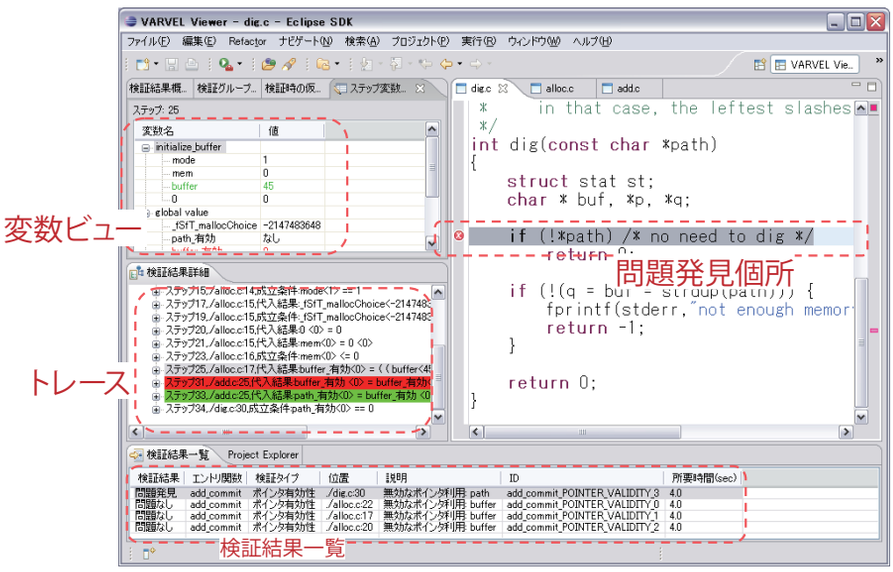

図 -8　検証結果の出力例（ビューア）

者は，システム構成やプログラムの詳細な仕様は把握していないため，ツールの指摘個所が本当のバグなのか判断がつかない場合も多い．したがって，明らかにバグではない指摘を除外したり，該当個所のソースコードを可能な範囲で確認した上で，バグの可能性をランク付けしたレポートを作成し，プロジェクト側に報告して最終確認を行ってもらう．

これまでに，全プロジェクト合計で約 3 百万行のソースコードに適用を行い，結果として数十件の不具合を検出することができた（プロジェクト側に報告した指摘事項の中から最終的にプログラムが修正された件数）．

●表明検証の適用

一方，表明検証については，あとから VARVEL に追加された機能であり，いくつかのプロジェクトで先行評価が始まった段階である．表明検証は，ソースコードに事前条件などの表明コメントを定義してもらう必要があるため，第三者がこの作業を代行するのが難しい．そこで，ツールの使い方と表明の書き方について 2 日程度の教育を行ったあとで，プログラマ自身に表明を記述して検証してもらう試行を実施した．

適用の結果，ほとんどの関数入出力において表明が遵守されていることが確認でき，さらに，図 -9 の例にあるような関数インタフェースに関連した数件の不具合も検出できることが確認できた．また，試行に参加したプログラマからは，表明を記述する作業が必要だが，テストケースを書いてテストを行うよりも，モデル検査による表明検証の方が楽だ，という評価も得られた．なお，この試行部門では，一度，表明コメントを記述しておけば，ソースコードに修正が行われた場合でも，比較的簡単に自動リグレッションテストとして検証ができる点を評価し，継続的にプロジェクトへの適用を行う予定である．

●適用効果について

ここで紹介した 2 つの事例を通して，VARVEL がソースコードの検証にある程度の効果があることを確認できたが，より具体的な定量効果の確認には，今後さらに，データ収集および分析が必要である．

一般的に，ソフトウェア開発においては，バグが摘出される工程が後になるほど，対処に必要なコストが大きくなることが知られており，より早い工程で問題点を摘出するフロントローディングが重要とされている．VARVEL は，この点において，従来は動的テストで摘出していた不具合の一部を，静的に検出することが可能で

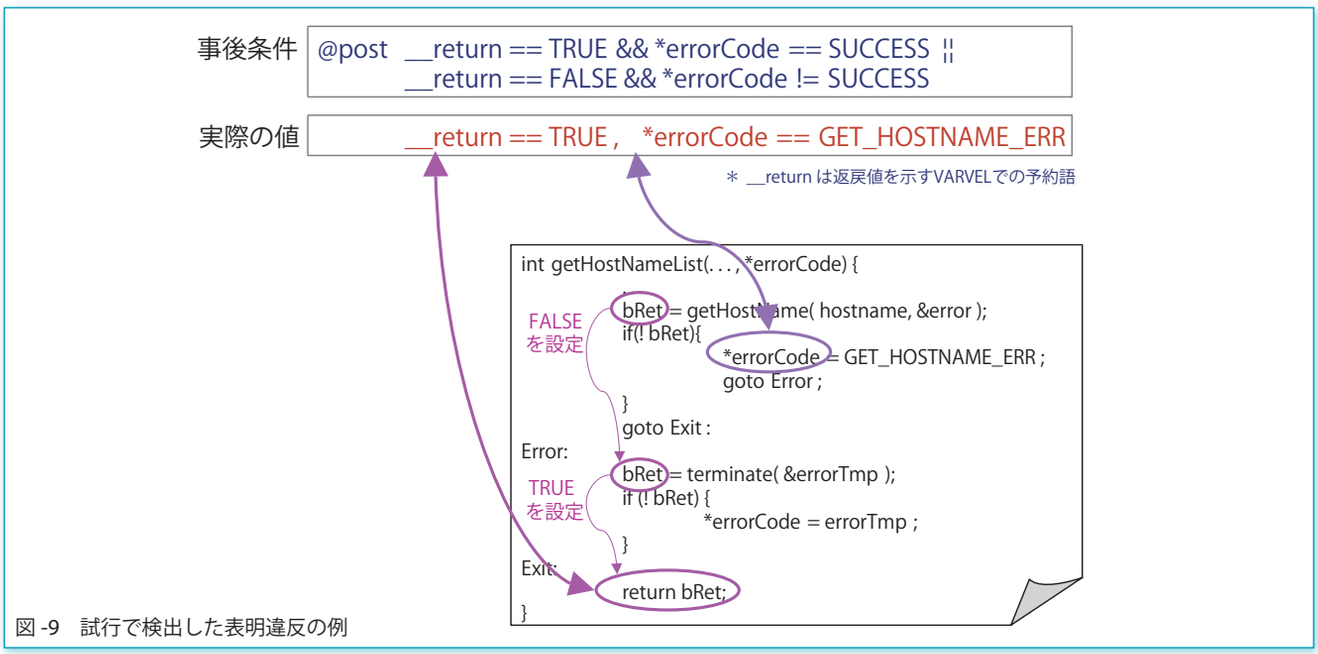

図 -9　試行で検出した表明違反の例

あり，より早い開発段階での品質向上に効果があると考えている．特に，表明検証の機能は，プログラム実装および単体テストの工程において，開発者自身に利用してもらうことを想定しており，フロントローディングの効果が高い．また，先ほどの事例の中でも紹介したように，表明を記述するための追加コストはそれほど大きくないと考えており，開発手法として定着すれば，かなりの費用対効果が期待できると考えている．

今後の取り組み

社内の複数の開発プロジェクトに適用し，ある程度の効果があることを確認できたが，「モデル検査技術をできるだけ簡単に利用し，生産性や品質向上に寄与したい」という目標に向けては，これから改善すべき余地も多い．

モデル検査を利用する上で一番の課題となるのが，検証時間の問題であり，一般的に利用されている静的チェックツールと比較すると，桁違いに長い時間が必要である．VARVEL においても，より短い時間で検証結果が得られるようにさまざまな最適化を行っているが，今後も改善が必要と考えている．

（検証時間の例）

ソースコードサイズ　約 5KL

検証項目数　約 4,200 項目

検証時間　約 2 〜 19 時間

※ CPU Pentium4 3GHz

※ 表明を除く 4 種類の検証タイプを指定

※ 検証タイプやタイムアウト時間設定により大きく異なるので注意

また，表明検証の機能は，設計仕様を明確に定義することで，意図した通りにソフトウェアが作られていることを実装・テスト工程の早い段階で確認できるという点で，品質向上の効果が大きいと期待しているが，検証によりバグのないことが確認できたコードの網羅性をチェックできる仕組みや，また表明を適切に定義してもらうための支援ツール環境の強化や，開発プロセスの改善などの取り組みが必要であると考えている．

参考文献

1) 中島　震：ソフトウェア工学の道具としての形式手法―彷徨える形式手法―. ソフトウェアエンジニアリング最前線 2007, pp.27-48 (2007).
2) 進藤智則：ソフトウェアは硬い，組み込みソフトウェア 2007 モデルに基づく開発方法論のすべて，pp.8-43 (2007).
3) Clarke, E., Kroening, D. and Lerda, F. : A Tool for Checking ANSI-C Programs, In Proc. TACAS'04, pp.168-176 (2004).
4) Brat, G., Havelund, K., Park, S. and Visser, W. : Java PathFinder – A Second Generation of a Java Model Checker, Workshop on Advances in Verification (2000).
5) Ganai, M. and Gupta, A. : SAT-Based Scalable Formal Verification Solutions, Springer (2007).
6) Meyer, B.（著），酒匂　寛（訳）：オブジェクト指向入門 第 2 版 原則・コンセプト，翔泳社 (2007).

（平成 20 年 3 月 24 日受付）

宮崎義昭
y-miyazaki@bq.jp.nec.com
1987 年日本電気（株）入社．コンパイラ，開発支援環境，インターネットシステム基盤に関連する製品開発に従事．現在は，ソフトウェアエンジニアリング手法の社内標準化や関連技術開発を担当．

橋本祐介
yu-hash@cb.jp.nec.com
1989 年日本電気（株）入社．CASE ツール・AP フレームワークなどの開発と適用支援を通じて，ソフトウェアエンジニアリング手法の社内普及を担当．現在は形式手法を担当．