ソフトウェアモデル検査とテストケース生成の統合ツールチェイン

橋本祐介[†1,†2]　中島震[†3,†1]

有界モデル検査はプログラムの信頼性を向上する有力な手段である．しかし，プログラムを有限状態遷移システムへ変換する過程で，広い意味での近似を導入することから，誤警告や不具合の見過しを起こすという問題がある．本研究では，この問題に対して有界モデル検査をプログラムテスティングで補うツールチェインを提案する．特に，不具合の見過しの原因となる過小近似の場合について，有界モデル検査とテスティングに共通のカバレッジ基準の下で，近似導入の有無を自動検知する方式を提案する．MINIX のソースコードを対象とした実験により，有界モデル検査とテスティングとを合せて，カバレッジ基準を満たす検査が行えることを示した．

A Tool Chain to Combine Software Model Checking and Test Case Generation

Yuusuke Hashimoto [†1,†2] and Shin Nakajima [†3,†1]

Bounded Model Checking (BMC) is a promising appraoch to achieving high quality of programs. In BMC, programs are translated into finite-state transition systems to introduce some approximation. It may result in spurious alarms or missing of errors. We propose a set of tools to augment BMC with program testing. It includes an automated way of detecting under-approximation and a specification-based test generation method, both of which employs a common coverage criteria. We also demonstrate effectiveness of our approach by applying it to checking of MINIX source codes.

†1 総合研究大学院大学　The Graduate University for Advanced Studies
†2 日本電気株式会社 サービスプラットフォーム研究所　NEC Corporation
†3 国立情報学研究所　National Institute of Informatics

1. はじめに

ソフトウェアの信頼性向上に関心が寄せられている．従来からの信頼性向上手段であるテストは，プログラム実行に依るので，テストケース毎に１つのパスしか検査せず，網羅度が低い．ロジック・モデル検査は全てのパスを考慮した網羅探索を行う[8]．ソフトウェアモデル検査はモデル検査のプログラム自動検証への適用である．産業界では，信頼性の基準はテストで与えられており，自動検証ツールを使った場合であっても，テストによる検査との関係を論じる必要がある．

モデル検査には，状態爆発と呼ばれるスケーラビリティの問題があり，状態や状態遷移が大規模なプログラムについては，不具合の有無を判定できないことがある．状態などの数は，抽象化に基づく過大近似の導入により削減できる[10]．しかし，見かけのパスが増えることにより，実際には起こり得ない不具合を検出するという誤警告の問題を起こすことがある．有界モデル検査法[6]は，探索範囲を一定の深さまでに限定し，不具合を効率よく検出する手法である．プログラム検証に適用する場合，対象プログラムを探索範囲内に制限する過小近似が必要となる．範囲外の不具合を見過す問題があるので，過小近似の導入には注意が必要である．

本研究では，有界モデル検査法を用いたモジュラー検査において，モデル検査における近似導入による問題をテストによって補うツールチェインを提案する．提案方法では，検証ツールによる警告が誤っていないことを，モデル検査の反例から生成したテストケースを実行して判定する．また，不具合の見過しの可能性を，モデル検査の方法を応用したトラップ挿入によって自動検知する．トラップはモデル検査とテストに共通のカバレッジ基準に則って挿入する．モデル検査をテストで補うことにより，近似の影響がなくなり，網羅度の高い検査ができる．さらに，MINIX のソースコードを用いた実験により，モデル検査とテストとを合せて，網羅度の高い検査が実施できたことを報告する．

本論文は次の構成をとる．第 2 章では，本稿で用いる有界モデル検査ツールと，近似導入の問題について述べる．第 3 章では，モデル検査をテストケース生成で補うツールチェインと手法を提案し，提案手法を MINIX の一部に適用した結果を第 4 章に示す．第 5 章で関連研究に触れ，最後に，今後の課題を第 6 章に述べる．

2. 背　　景

2.1 有界モデル検査ツール VARVEL

C プログラムの有界モデル検査ツールとして，CBMC[9]，F-Soft[16]，SMT-CBMC[3]，ESBMC[11] などがある．対象プログラムの有限状態遷移システムとしての式を C，対象プログラムが満たすべき性質 (プロパティ) の式を P とすると，$C \Rightarrow P$ を示したい（$\Rightarrow$ は論理の含意）．有界モデル検査ツールは，前記の論理式の否定である $C \wedge \neg P$ を満たす変数値の割当を充足可能性判定器 (SAT/SMT ソルバ) により探す．見つかった変数値の割当は，対象プログラムがプロパティを満たさないことを示す反例，即ち，ある初期状態からプロパティを満たさない状態に至るパスの一例である．見つからなければ，対象プログラムはプロパティを満足する．状態爆発により，指定時間内に探索が終わらない場合やメモリが不足する場合には，対象プログラムがプロパティを満たすかどうかは判らない．

本稿で用いる VARVEL は，F-Soft を基盤とする有界モデル検査ツールである[23]．逐次処理の C/C++プログラムを対象として，Design by Contract (DbC) に基くモジュラー検査を行う．DbC は，事前・事後条件といった DbC 仕様をプログラムに明示して，堅牢なソフトウェアを構築する考え方である[20]．DbC 仕様をプログラムのコメントとして追記する記法が提案されている[5][17]．VARVEL は独自の DbC 記法を提供する．プログラム全体の検査は，次の (1)(2) により，関数単位のモジュラー検査に分けて行う[14]．(1) 検査の対象関数について，その事前条件を前提として，事後条件を満たすことを検査する．(2) 対象関数における関数呼出しについては，呼ばれる関数のボディの代わりに DbC 仕様を用いて，その事前条件を満たして呼出しが行われることを検査し，その事後条件を前提として呼び出し後の状態遷移を考える．

具体的には，VARVEL は，コメント中の DbC 仕様をプリミティブ関数 $assert$ と $assume$ を用いたプログラムに変換する．プリミティブが評価される前の状態を S とすると，$assert(C)$ は $S \Rightarrow C$ を検査すべきことを，$assume(C)$ は評価後の状態を $S \wedge C$ とすべきことを VARVEL に指示する．変換方法は関数の使い方の種類によって異なる．たとえば，コールツリーの起点となる関数であれば，事前条件 Pre は $assume(Pre)$ として関数の開始箇所に挿入する．事後条件 $Post$ は $assert(Post)$ として終了箇所に挿入する．ソースコードを検査ツールに与えていない関数などについての変換方法は他の文献[15] に詳しい．

さらに，VARVEL では関数ポインタを対象とした DbC 記法を提供する．また，関数ポインタを介した呼出しに関する検査を，関数ポインタの仮 DbC 仕様を用いたモジュラー検査と，実際に呼ばれる各関数の DbC 仕様と仮 DbC 仕様との一貫性検査に分けて行う．関数ポインタの DbC 記法と検査方法について，MINIX を対象とした実験を行い，関数ポインタに関係する不具合の検出に成功した[24]．

2.2 近似の問題

産業界では，信頼性の基準はテストで与えられており，たとえ自動検証ツールを使った場合であっても，テストによる検査との関係を論じる必要がある．

ソフトウェアモデル検査では，ロジック・モデル検査が適用できるように，対象プログラムを有限状態遷移システムに変換する．モデル検査において，状態爆発というスケーラビリティの問題が起きると，有限状態空間上の探索を終えられず，不具合の有無を示せない．変換の際に抽象化といった過大近似を導入することにより，状態や状態遷移の数を減らして状態爆発を緩和できる[10]．しかし，過大近似によって見かけの振舞いが増える．逆に，過大近似されたプログラムは元のプログラムには存在しない見かけのパスを含む．プログラムが含むパスとその集合を π と Π，変換の関係を α，変換後であることを $'$ とすると，過大近似は次のように表せる．

$\forall \pi \in \Pi \cdot \exists \pi' \in \Pi' \cdot \pi' = \alpha(\pi)$．　$\exists \pi' \in \Pi' \cdot \pi' \notin ran(\alpha)$　($ran(\alpha)$ は α の値域)

見かけのパスで検出される不具合は，元のプログラムでは起こり得ない見かけの不具合である．見かけの不具合は原因調査の必要がないので，誤警告として自動判定したい．モデル検査の反例から不具合を再現し得るテストケースを生成する研究[7] が判定に利用できる．

有界モデル検査法では，有限状態空間をある深さまでしか探索しないことにより，効率的に不具合を見つける．プログラムを探索範囲に制限するための工夫は，過小近似を導入することである．元のプログラムが変換後のプログラムの振舞いを全て含むような近似を過小近似と呼ぶ．逆に，過小近似されたプログラムは元のプログラムの一部のパスを含まない．

$\forall \pi' \in \Pi' \cdot \exists \pi \in \Pi \cdot \pi' = \alpha(\pi)$．　$\exists \pi \in \Pi \cdot \pi \notin dom(\alpha)$　($dom(\alpha)$ は α の定義域)

このようなパス π 上のみで起こる不具合は，変換後のプログラムを探索しても検出できない．不具合の見過しを防ぐためには，元のプログラムにおいてモデル検査では探索されないパスの有無を自動で検知したい．また，そのようなパスが検知されれば，モデル検査とは別の手段で検査したい．別手段としてはプログラムテストを考えることができる．

テストを行うには，テストケース（テスト入力と期待される結果）を生成するために，対象プログラムの仕様が必要となる．事前・事後条件といった仕様を同値分割などにより区分けし，区分ごとにテストケースを生成する研究がある[12][18]．

3. 統合ツールチェイン

3.1 近似の取り扱いの方針

ソフトウェアモデル検査による検査結果は，近似導入の影響を受けている可能性がある．本研究では，ソフトウェアモデル検査とテストケース生成を組み合せて，近似導入の自動判定やカバレッジ基準を満たす検査を行う方法を提案する．一般に，過大近似と過小近似は混在し得るため，あるパスにどの近似が導入されたかを特定することは難しい．そこで，パスがモデル検査により探索可能か，テストにより実行可能か，という観点から，近似導入の有無を調べる．

過大近似については，モデル検査で得られた反例からテストケースを生成し，テストによって不具合の再現を試みる．再現しなければ，不具合は過大近似の影響による見かけのものであり，反例は誤警告であると自動判定できる．

過小近似については，モデル検査で探索されれば必ず反例が得られるトラップを対象プログラムに埋め込む．反例が得られなければ，トラップを含むパスはある箇所から探索されていない，即ち，不具合の見過しの可能性を自動検知できる．検知した場合には，DbC 仕様からテストケースを生成し，テストを行う．テストと共通のカバレッジ基準に則って，検知用のトラップを埋め込むことにより，プログラムの大部分をモデル検査により網羅探索し，残りの部分を少ないテストケースで補助的に検査するといった，従来のテストに比べて網羅度の高い検査ができる．

3.2 ツールチェインの概要

本稿で提案する，有界モデル検査とテストケース生成を統合するツールチェインの概要を図 1 に示す．このツールチェインは，検査の対象プログラムについて有界モデル検査を行う A) ソフトウェアモデル検査，有界モデル検査で得られた反例からテストケースを生成する B) 反例に基くテストケース，対象プログラムに過小近似の検知用のトラップを挿入する C) カバレッジ用トラップ挿入，対象プログラムの DbC 仕様からテストケースを生成する D) 仕様に基くテストケース，の 4 つの機能から成る．本稿の時点では，機能 A と C を実装している．機能 B と D は既存研究[7)18)] をもとに実装可能である．

ツールチェインが行う処理のアルゴリズム `doSWMCandTest` を図 2 に示す．`doSWMCandTest` は，機能 A によりソフトウェアモデル検査を行う (行 1)．指定時間内に探索が終わらない場合には，機能 D により DbC 仕様からテストケースを生成し，終了する (行 2,3)．モデル検査の結果が不具合の反例を含む場合，機能 B により反例に基づいてテストケースを生成

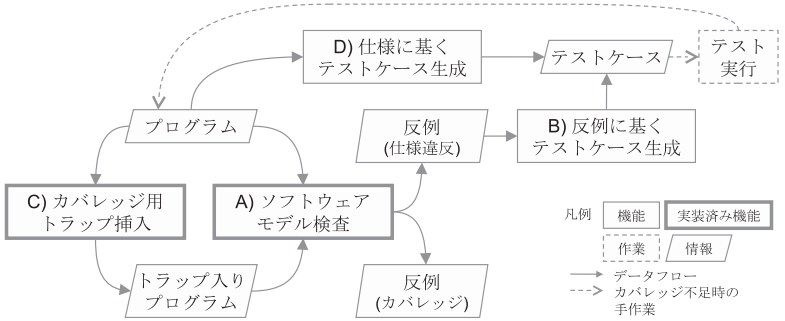

図 1　ツールチェインの概要
Fig. 1　Overview of tool chain

する (行 4–6)．生成されたテストケースを実行して，不具合が再現しなければ，反例は過大近似による誤警告であることがわかる．過小近似への対応としては，機能 C によりカバレッジ基準に則って対象プログラムの探索すべき箇所にトラップを挿入し，機能 A によりトラップ入りプログラムに対してモデル検査を行い反例を得る (行 7,8)．挿入した各トラップについて反例の有無を調べ，一つでも反例の無いトラップがあれば，未探索箇所があるものとして，機能 D により DbC 仕様からテストケースを生成し，終了する (行 9–14)．しかし，生成されたテストケースを用いて反例の無いトラップの箇所を実行できない場合には，DbC 仕様をさらに詳細に場合分けしてテストケースを生成し直す必要がある．

モデル検査の反例からテストケースを生成することはよく知られており，反例の初期状態において変数に割り当てられた値をテスト入力とする[2)]．以降の 3.3，3.4 節では，モデル検査では探索されない箇所の検知と DbC 仕様からのテストケース生成について述べる．

3.3 近似導入の検知方法

過小近似の導入による問題について，ループの繰返しを所与の回数に限定する近似を例として，図 3 を用いて説明する．プログラムの複雑さに応じて状態遷移は様々に変わるので，有界モデル検査の探索の深さとループの繰返し回数の上限との関係は一意には決まらない．ここでは，簡単のため所与の回数を 2 回とする．検査の対象関数 `foo` の DbC 仕様では，引数 `v` は，大域の配列 `a` のいずれかの要素と値が同じか，全ての要素と値が異なる (行 6)．`foo` の戻り値は `0` 以上か`-1` であり，前者の場合は戻り値を添え字とする `a` の要素が `v` と同じ値を持つ (行 7)．`foo` の関数ボディでは，他の関数 `bar` の戻り値と大域変数 `g`

```
    doSWMCandTest( program )
1     result1 := doSWMC( program );
2     if ( resourceOut( result1 ) )
3     then doTCGfromSpec( program );
4     else
        /* for Over-Approximation */
5       foreach counterexample in result1
6         doTCGfromCE( counterexample, program );
        endloop
        /* for Under-Approximation */
7       trapProgram := instrumentTrap( program );
8       result2 := doSWMC( trapProgram );
9       uncovered := false
10      foreach trap in trapProgram
11        if ( notFoundCounterExample( trap, result2 ) )
12        then uncovered := true;
          endif
        endloop
13      if ( uncovered )
14      then doTCGfromSpec( program );
        endif
      endif
```

図 2　有界モデル検査をテストケース生成で補うアルゴリズム
Fig. 2　Algorithm to enhance BMC with TCG

```
    #define  SIZ  1024
1   extern int a[SIZ];
2   extern int g;
3   /** @invariant g==-1 || g==0 */
4   /** @post  __return==0 || __return==1 */
5   int bar();
6   /** @pre __exist( k, 0, SIZ-1, a[k]==v ) ||
                 __foreach( k, 0, SIZ-1, a[k]!=v )
7       @post 0<=__return && a[ __return ]==v || __return==-1
     */
8   int foo( int v ){
9     int r=-1, i=0;
10    if ( bar()+g==1 ) {
11      r=-2; /* BUG */
12      while ( i<SIZ ) {
13        if ( v==a[i] ) { r=i; break; }
14        i=i+1;
        }
      }
15    return r;
    }
```

図 3　過小近似の影響を受けるプログラム例
Fig. 3　Example with a loop

の値の和が 1 の場合に (行 10)，while ループ内で，配列 a の各要素を調べ，引数 v と値の同じ要素があれば，その要素の添え字を返し (行 12–14,15)，それ以外の場合には，本来は-1 (行 9,15) を返す．図 3 では変数 r に-2 が代入されることがあり (行 11)，そのまま r が戻り値として使われると，事後条件への違反となる．過小近似されたプログラムは，ループボディ (行 12,13) の 3 回目の実行を行う前に処理を終了する．このプログラムをモデル検査で検証すると，事後条件違反の不具合は検出されない．しかし，ループボディの 3 回目以降については，モデル検査では調べておらず，事後条件が守られたのかどうかは判らない．

プログラムのパスは無数にあり得るので，探索されないパスを探し尽くすことは難しい．パスの代わりに，プログラムの特定の箇所が探索されないことを検知する．この検知は，特定の箇所でのみ真になるトラップ変数を埋め込み，対象プログラムの終了箇所でトラップ変数が偽であるというプロパティをモデル検査で調べればよい．反例がなければ，特定の箇所は探索されていない．探索されない箇所というプログラムの内部の性質について調べるので，便宜上，プログラム構造に着目したカバレッジ基準に則って，特定の箇所を決める．カバレッジ基準としては，CACC (Correlated Active Clause Coverage)[1] を用いる．分岐文やループ文における制御式の全体を述語と呼び，述語は論理演算子を含まない節を論理演算子で結合したものとすると，CACC では，着目した各節ごとに，その節の値によって述語の値が決まるような 2 通りのテストを行う．他の節の値は 2 通りのテストにおいて異なっていてもよい．CACC は産業界で実用的に使われている masking MCDC に相当する．

カバレッジ基準に則ったトラップを埋め込むために，表 1 の整形ルールを用いて，プログラムを整形し，トラップ入りプログラムを得る．表 1 にはないが，ループ文（for，do），分岐文（switch），論理演算子（!，||）といった他の言語要素についても同様のルールを決められる．

整形では，まず，ルール aN(N=1,…) によって，述語の値を保持する変数__d を導入し，元のループや分岐文を，その述語の値を__d に設定する文の集まりと，ループボディや副

表 1　トラップ挿入のための整形ルール (一部)
Table 1　Some transformation rules to insert traps

a1		`while (p) { S }`　　(p は定数を除く)
	→	`while (1) { if (p) { ; } else { break; } S }`
a2		`if (p) { S }`　　(p は__d を除く)
	→	`if (p) { __d=1; } else { __d=0; } if (__d) { S }`
b1		`if ( p && q ) { __d=b1; S } else { __d=b2; T }`
	→	`if ( p ) { if ( q ) { __d=b1; S } else { __d=b2; T } }` `else { __d=b2; T }`
c1		`if (p) { __d=b; }` → `if (p) { __d=b; __cov_i_j=1; }`
c2		`else { __d=b; }` → `else { __d=b; __cov_i_j=1; }`
d		`return <戻り値>;`
	→	`__assert(!__cov_i_j);`　　(トラップ変数分，繰り返す)
		`return <戻り値>;`

→:左辺と右辺はそれぞれ成形前と後．b,b1,b2:真偽値（b1≠b2）．&&:論理積．!:否定．p,q:述語．
S,T:空ではない文やブロックの並び．__d:述語の値を保持する変数．
__cov_i_j: トラップ変数，i は関数ごとの述語の連番，j は述語ごとの節の組合せの連番．

文に分ける．次に，ルール b1 によって，__d に述語の値を設定する分岐を，述語を節に分解した入れ子の分岐に変換する．入れ子の末端の全てが探索された場合には，各述語の真偽のいずれもが探索されたことになり，CACC の網羅性がある．探索されたことを判定するために，入れ子の末端の分岐にルール cN(N=1,...) を適用して，トラップ変数__cov_i_j (i=1,⋯．j=1,⋯) を導入し，その値として真（1）を設定する（トラップ変数は偽（0）で初期化する）．最後にルール d によって，return 文の直前に，各トラップ変数が偽であるという検査を指示するプリミティブ関数（__assert）を挿入する．モデル検査によって見つかった反例は，トラップ変数に真を設定する箇所が探索されたことを示す．反例が見つからなければ，探索されない箇所があることが検知されたと結論づけてよい．

図 4 は，図 3 のプログラムを，表 1 のルールを用いて整形した例である．

図 4 の行 10 と 13 の if 文および行 12 の while 文は，それぞれ図 4 の行 10a–c と 13a–c および 12a–d に整形される．トラップ入りプログラム（図 4）に対するモデル検査のカバレッジを表 2 に示す．見出し行の「i,j」はトラップ変数__cov_i_j に対応しており，SWMC 行の各セルの Y はトラップ変数に真 (1) を設定する箇所がモデル検査で探索されたことを示す．ループボディを 2 回のみ実行する近似の導入により，while 文の制御式 i<SIZ が偽となる場合が探索されていないことが分かる．

```
 8   int foo( int v ){
 9     int r=-1, i=0;
10a    if ( bar()+g==1 )  { __d=1; cov_1_1=1; }
10b    else               { __d=0; cov_1_2=1; }
10c    if ( __d ) {
11       r=-2; /* BUG */
12a      while ( 1 ) {
12b        if ( i<SIZ )   { __d=1; cov_2_1=1; }
12c        else           { __d=0; cov_2_2=1; }
12d        if ( __d ) { ; } else { break; }
13a        if ( v==a[i] ) { __d=1; cov_3_1=1; }
13b        else           { __d=0; cov_3_2=1; }
13c        if ( __d ) { r=i; break; }
14         i=i+1;
         }
       }
15a    __assert( !cov_1_1 );  __assert( !cov_1_2 );
15b    __assert( !cov_2_1 );  __assert( !cov_2_2 );
15c    __assert( !cov_3_1 );  __assert( !cov_3_2 );
15     return r;
     }
```

図 4　未探索箇所を検知するためのトラップ入りプログラムの例
Fig. 4　Example with inserted traps

3.4　DbC 仕様からのテストケース生成

ソフトウェアモデル検査で探索されない箇所が見つかった場合，DbC 仕様からテストケースを生成し，テストを行う．検査対象の関数について，その事前条件 Pre と事後条件 $Post$ は，次の DNF 形式であるとする．任意の論理式は DNF 形式に変換可能なので一般性を失わない．

$$Pre = \bigvee_i P_i\ , \qquad Post = \bigvee_j (G_j \wedge D_j)$$

事後条件はガード条件 G_j と定義条件 D_j から構成されており，ガード条件は関数によって値が変更される変数を含まず，定義条件はそれらを含む．関数全体の振舞いは機能シナリオフォーム FSF で表される．

$$FSF = \bigvee_{i,j} fs_{ij} = \bigvee_{i,j} (P_i \wedge G_j \wedge D_j)$$

FSF 中の各 fs_{ij} を機能シナリオと呼ぶ．上記 FSF は，劉が提案した FSF[18] に対して，事前条件も DNF 形式とし機能シナリオをより細分化してある．テストケースの生成では，

```
1  int bar() { return 1; }  /* Ql */
2  void test1(){
3    int result, defining;
4    a[0]=999, ···;          /* Random */
5    int v=999;              /* Pi ∧ Gj */
6    g=0;                    /* Ik */
7    result = foo(v);
8    defining = (0<=result && a[result]==v); /* Dj */
9    if (defining) printf("OK"); else printf("NG");
   }
```

図 5　テストプログラムの例
Fig. 5　Example test program

検査対象の関数の機能シナリオ以外に，対象関数に参照・更新される大域変数の値域や対象関数に呼ばれる関数の振舞いも考慮する必要がある．例えば，図 3 のプログラム例では，行 10 の `if` 文の制御式 `bar()+g==1` が真および偽となるような大域変数 `g` の値と関数 `bar` の戻り値の組合せをテストする必要がある．そこで，大域変数の不変条件 $Inv = \bigvee_k I_k$ と呼ばれる関数の事後条件 $Post^{called} = \bigvee_l Q_l$ も考慮して，$P_i \wedge G_j \wedge I_k \wedge Q_l$ を満足する変数値（大域変数と呼ばれる関数の副作用を含む）をテスト入力とし，D_j を期待結果とする．図 3 のプログラム例では，対象関数の事前条件は，`__exist(k, 0, SIZ-1, a[k]==v)` と `__foreach(k, 0, SIZ-1, a[k]!=v)` の 2 通り，大域変数の不変条件は，`g==-1` と `g==0` の 2 通り，呼ばれる関数の事後条件も `bar` の戻り値`==0` と `bar` の戻り値`==1` の 2 通りで，合計 8 通りの組合せがある．

図 3 のプログラム例に対して，DbC 仕様を元に生成したテストプログラムを図 5 に示す．関数 `bar` については事後条件`__return==1` を満たすスタブ関数を生成する (行 1)．機能シナリオに出現しない大域配列 `a` の各要素の値はランダムに生成する (行 4)．検査対象の引数と大域変数には，それぞれ $P_i \wedge G_j$ と I_k を満たす値を充足可能性判定ソルバ Yices[13] を用いて求める（行 5,6）．テストプログラムは，生成したテスト入力を与えて対象プログラムを実行し（行 7），実行結果について定義条件の値を調べ，ログを出力する（行 8,9）．元のプログラムの代わりに，トラップ入りプログラム（図 4）を用いて，トラップ変数の値を調べる`__assert` のログ出力に置き換えてカバレッジを調べた結果を表 2 の Testing 行に記す．ソフトウェアモデル検査をテストで補完することにより，両者を合せて少なくとも CACC 以上の網羅度での検査が行えている．

表 2　モデル検査とテストによるカバレッジの例
Table 2　Example of coverage by model checking and testing

	1,1	1,2	2,1	2,2	3,1	3,2
SWMC	Y	Y	Y		Y	Y
Testing	Y	Y	Y	Y	Y	Y

見出し i,j：i 番目の述語における節の値の j 番目の組合せ

4. 実験と考察

提案ツールチェイン (図 1) において，トラップ挿入ツールを作成し，既存のソフトウェアモデル検査ツール VARVEL と組み合せて，オープンソースの OS である MINIX[22] (Version 3.1.1) のソースコードの一部に適用した．MINIX はマイクロカーネルアーキテクチャを採用しており，サーバ，カーネル，ドライバ層の各プロセスがメッセージ通信によって協調的に動作する．ユーザアプリケーションからのドライバへの要求は，サーバ層のファイルシステム機能を介して行われる．とくにファイルシステム機能のファイル `device.c` にはドライバとのメッセージ通信に関する関数が集められている．MINIX の特徴を検査するのに適したファイルと考え，本稿で取り上げた．

表 3　MINIX への提案手法の適用結果
Table 3　Result of applying proposed approach to MINIX

Function	Size	#Trap	#BMC	#Test	#Total	#TC
`dev_open`	20	6	6	–	6	–
`dev_close`	9	2	2	–	2	–
`dev_status`	41	21	19	4	21	1
`dev_io`	48	8	8	–	8	–
`tty_opcl`	32	9	4	8	9	32
`ctty_opcl`	12	2	2	–	2	–
`do_setsid`	16	2	2	–	2	–
`do_ioctl`	39	5	5	–	5	–
`gen_io`	76	25	22	18	25	144
`ctty_io`	21	2	2	–	2	–
`clone_opcl`	52	7	7	–	7	–

Function：関数名．　Size：関数の空白行を除く行数．　#Trap：カバレッジ測定用のトラップ変数の個数．
#BMC：有界モデル検査 (BMC) で探索されたトラップの個数．　#Test：テストで実行されたトラップの個数．
#Total：BMC とテストのいずれかでカバーされたトラップの個数．　#TC：生成されたテストケースの個数．

表 4　MINIX を対象としたモデル検査とテストによるカバレッジ
Table 4　Coverage in MINIX by model checking and testing

	1,1	1,2	2,1	2,2	2,3	3,1	3,2	4,1	4,2	5,1	5,2
SWMC		Y	Y	Y	Y		Y	Y	Y	Y	Y
Test	Y	Y		Y		Y					

	6,1	6,2	7,1	7,2	8,1	8,2	8,3	8,4	9,1	9,2	
SWMC	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
Test											

見出しの i,j：i 番目の述語における節の値の j 番目の組合せ

device.c の主要な関数について，本稿の提案手法に則って，ソフトウェアモデル検査を行い，探索されない箇所があればテストケースを生成して，テストを行った．その結果を表 3 に示す．DbC 仕様としては，メッセージのタイプは特定の整数値であること，プロセス番号は-4 以上かつ 100 未満であること，デバイス番号は 0 であるか，16 ビット中の上位 8 ビットが 32 未満であること，処理の結果を表す値は成功が 0 であり，失敗が負値であること，の 4 点を DNF 形式で記述した．これらの DbC 仕様が検査したいプロパティである．テストケースは，DNF 形式の DbC 仕様の各節から引数や大域変数の代表値の組を求め，これらの代表値の組の直積として得た．各代表値は充足可能性判定ソルバを用いて線形演算の範囲で求めた．

表 3 の関数 dev_open では，6 個のトラップの全てをモデル検査で探索できたので，テストは実行していない．列#Test と#TC の-- は，テストケース生成およびテストを行わなかったことを示す．関数 dev_status では，21 個のトラップのうち 19 個をモデル検査によって探索した．また，残りの 2 個のトラップを含む 4 個のトラップを 1 個のテストケースによって実行した．dev_status の各トラップが有界モデルとテストにより，どのようにカバーされたかを表 4 に示す．見出し行の i,j はトラップ変数__cov_i_j に対応する．SWMC 列の Y はトラップがソフトウェアモデル検査によって探索されたことを，Testing 列の Y はテストによって実行されたことを示す．列 1,1 と 3,1 を見ると，ソフトウェアモデル検査をテストで補えたことがわかる．

テストケースが十分ではなく，実行できないトラップがある場合には，プログラムコードを調べて，トラップに関係する DbC 仕様をより詳しく記述し，テストケースを作り直した．例えば，関数 gen_io では，呼ばれる関数_sendrec の戻り値が 0 以下であるという事後条件（__return<=0）を，戻り値の値を明示した DNF の式（__return==0 || __return==-101 ||･･･）に修正した．この例では，戻り値の場合分けが多く，作り直しに約 1 時間かかったが，他は 10 分程度で済んだ．修正後の DbC 仕様を用いても，表 3 のモデル検査で探索したトラップの数は変わらなかった．

表 3 では，全トラップ 89 個のうち，89% に当たる 79 個のトラップが有界モデル検査によって探索された，即ち，これらのトラップを通り関数の終了箇所に至るパスが網羅探索された．対象プログラムの大部分を有界モデル検査によって網羅的に検査し，残りの一部についても分岐カバレッジ基準 CACC を満たすテストを行っており，従来のテストに比べてカバレッジの高い検査を実施できた．

引数や大域変数の値域の直積としてテストケースを生成する方法を用いて，テストのみで CACC 基準を満たそうとすると，関数 gen_io の場合，9418 個のテストケースが必要であった．変数値の組合せを特定するように DbC 仕様を記述し，テストケースを減らすことができるが，プログラムの内部構造に合せて DbC 仕様を場合分けする必要があり手間がかかる．できるだけ多くの範囲を有界モデル検査で網羅的に検査し，一部についてのみテストケースを考慮する本稿のアプローチはテストの省力化という点で有効な方法である．

5. 関 連 研 究

有界モデル検査法を用いてプログラムを検査するツール[3)9)11)16)23)] は不具合の検出に有効である．有界モデル検査を適用するために，プログラムを有限状態遷移システムに変換する際に，近似を導入することがある．ループを固定回数分のループボディの繰返しとするような変換を行うと[9)]，有界モデル検査では固定回数を超える部分を含むパスを探索しない．そのようなパス上の不具合は見過される．探索の深さに制限されないように有界モデル検査を拡張する手法[19)21)] が提案されているが，本論文ではテストとの役割分担という方法を提案した．産業界では，信頼性の基準はテストで与えられており，たとえ自動検証ツールを使った場合であっても，テストによる検査との関係を論じる必要があることが理由である．

単体テストが十分であるかはプログラム構造に関するカバレッジで測る．本稿では，分岐カバレッジ基準として CACC[1)] を採用し，モデル検査のカバレッジを測る．検査対象にトラップ変数を挿入して，意図したパスを示す反例をモデル検査によって得る考え方がある[7)]．本稿では，分岐の述語を節に分解し，節の真偽の組合せごとにトラップを挿入することにより，モデル検査が CACC を満たすパスを探索したことを調べる．

しかし，有限状態空間を網羅探索するモデル検査のカバレッジを，CACC のような分岐カバレッジ基準で測ることは適切とは言い難い．パスカバレッジはパスが無数あり得るた

め，カバレッジ基準としては不適切である．関連研究として，PCT(Predicate Complete Testing) カバレッジ[4] が提案されている．対象プログラムの文の個数 M と述語の個数 n に対して，各文における述語の取り得る値の組を 1 つの状態とする．対象プログラムをブーリアンプログラムに変換し，ブーリアンプログラム上のパスを辿って各状態への到達性を調べて，到達可能な状態の総数をカバレッジの分母とする．状態の総数は最大でも $M \times 2^n$ なので，分母を有限に押さえることができる．モデル検査のカバレッジは，PCT カバレッジのようにパスを考慮し，かつ母数が有限のカバレッジ基準を用いて測ることが望ましい．

6. おわりに

有界モデル検査法を用いたソフトウェアモデル検査において，過小近似によって探索されない箇所が生じる問題を明らかにし，探索されない箇所の検知と DbC 仕様からのテストケース生成とを合せて，ソフトウェアモデル検査を補完する手法を提案した．また，探索されない箇所の検知を自動化して，MIINX の一部のソースコードに提案手法を適用する実験を行い，提案手法の有効性を確認した．本稿では，VARVEL を具体的な有界モデル検査ツールとして，提案方法の考察・実験を行った．しかし，提案方法は，CBMC など C プログラムの有界モデル検査ツール一般に適用可能である．

今後の課題については，より大規模な実験が必要である．また，カバレッジ基準については PCT カバレッジのようなモデル検査の特性を考慮したものが好ましい．今後，さらに検討の余地がある．

参考文献

1) Ammann, P. and Offutt, J.: *Introduction to Software Testing*, Cambridge University Press (2008).
2) Ammann, P.E., Black, P.E. and Majurski, W.: Using Model Checking to Generate Tests from Specifications, *Proc. ICFEM*, IEEE Computer Society, pp.46–54 (1998).
3) Armando, A., Mantovani, J. and Platania, L.: Bounded Model Checking of Software using SMT Solvers instead of SAT Solvers, *Proc. SPIN*, pp.146–162 (2006).
4) Ball, T.: A Theory of Predicate-Complete Test Coverage and Generation, Technical report, Microsoft Research Technical Report, MSR-TR-2004-28 (2004).
5) Baudinl, P., Filliatrel, J., Marchel, C., Monatel, B., Moy, Y. and Prevosto, V.: ACSL: ANSI/ISO C Specification Language (2008).
6) Biere, A., Cimatti, A., Clarke, E.M. and Zhu, Y.: Symbolic Model Checking without BDDs, *Proc. TACAS*, pp.193–207 (1999).
7) Callahan, J., Schneider, F. and Easterbrook, S.: Automated Software Testing Using Model-Checking, *Proc. SPIN* (1996).
8) Clarke, E.M., Grumberg, O. and Peled, D.: *Model Checking*, The MIT Press (1999).
9) Clarke, E.M., Kroening, D. and Lerda, F.: A Tool for Checking ANSI-C Programs, *Proc. TACAS*, pp.168–176 (2004).
10) Clarke, E.M., Grumberg, O. and Long, D.E.: Model Checking and Abstraction, *ACM TOPLAS*, Vol.16, No.5, pp.1512–1542 (1994).
11) Cordeiro, L., Fischer, B. and Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software, *Proc. ASE*, pp.137–148 (2009).
12) Dick, J. and Faivre, A.: Automating the Generation and Sequencing of Test Cases from Model-Based Specifications, *Proc. FME*, pp.268–284 (1993).
13) Dutertre, B. and Moura, L.D.: A Fast Linear-Arithmetic Solver for DPLL(T), *Proc. CAV*, Springer, pp.81–94 (2006).
14) Hashimoto, Y. and Nakajima, S.: Modular Checking of C Programs Using SAT-Based Bounded Model Checker, *Proc. APSEC*, pp.515–522 (2009).
15) Hashimoto, Y. and Nakajima, S.: Modular Checking with Model Checking, *Proc. SSV*, pp.105–122 (2009).
16) Ivancic, F., Shlyakhter, I., Gupta, A., Ganai, M., Kahlon, V., Wang, C. and Yang, Z.: Model Checking C Programs Using F-Soft, *Proc. ICCD*, pp.297–308 (2005).
17) Leavens, G., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P. and Zimmerman, D.M.: JML Reference Manual (2008).
18) Liu, S.: Integrating Specification-Based Review and Testing for Detecting Errors in Programs, *Proc. ICFEM*, pp.136–150 (2007).
19) McMillan, K.L.: Interpolation and SAT-based Model Checking, *Proc. CAV*, pp. 1–13 (2003).
20) Meyer, B.: Applying Design by Contract, *IEEE Computer*, Vol.25, No.10, pp.40–51 (1992).
21) Moura, L.D., Ruess, H. and Sorea, M.: Bounded Model Checking and Induction: From Refutation to Verification, *Proc. CAV*, pp.14–26 (2003).
22) Tanenbaum, A.S.: オペレーティングシステム 第 3 版，ピアソンエデュケーション (2007).
23) 宮崎義昭，橋本祐介：C 言語へのフォーマルメソッドの適用，情報処理， Vol.49, No.5, pp.514–520 (2008).
24) 橋本祐介，中島　震：有界モデル検査法を用いた C プログラムのモジュラー検証，情報処理学会論文誌，Vol.52, No.8, pp.2422–2430 (2011).