情報処理学会論文誌 Vol. 52 No. 8 2422–2430 (Aug. 2011)

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

橋本祐介[†1,†2] 中島震[†3,†1]

ソフトウェアモデル検査はプログラムの信頼性向上の有望な手段である．本研究では，C プログラムを対象とした SAT に基づく有界モデル検査法に Design by Contract（DbC）の考え方を明示的に取り入れる．DbC に基づくモジュラー検証を実現することにより，モデル検査が直面するスケーラビリティの問題を解消する．さらに，関数ポインタを用いた C プログラムへの DbC の導入手法を提案し，MINIX ソースコードを対象とした実験により誤検出の削減や不具合の検出の観点から効果を示した．

Modular Verification of C Programs with Bounded Model Checking Method

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

Software model checking is a promising technique to improve the reliability of programs. This paper introduces a notion of Design by Contract (DbC) into SAT-based bounded model checking of C programs. It achieves the modular verification and resolves the scalability problem that model-checkers usually have. Further, we propose a new DbC notation for C programs using pointers to functions and a two-step method to check the programs. We also demonstrate the reduction of the number of spurious alarms and the detection of potential bugs in the experiment using source codes of MINIX.

†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
Service Platforms Research Laboratories, NEC Corporation
†3 国立情報学研究所
National Institute of Informatics

1. はじめに

ソフトウェアが重要な役割を果たすようになり，その品質に大きな関心が寄せられている．品質向上の一手段であるソフトウェアモデル検査は，有限状態空間上の網羅的な探索を実現するモデル検査法[4] のプログラム検証への適用である．

モデル検査法には，状態爆発と呼ばれるスケラービリティの問題があり，状態遷移の組合せが増えると計算量が指数関数的に増える．SAT に基づく有界モデル検査法（BMC）は，探索範囲を限定することによりスケーラビリティの問題を緩和するが[3]，探索範囲外の不具合を見過ごすという問題を持つ．BMC を用いたプログラム検査ツール[5),14)] は不具合を効率的に検出するが，検査対象プログラムが大きくなると不具合を見過ごしやすくなる．

モジュラー検証は，検査対象プログラムを手続きごとに検査することにより，スケーラビリティの問題を解消する考え方であり，Design by Contract（DbC）[17] を取り入れた拡張静的チェッカ[9),11)] として実現されている．DbC では，プログラムの外部から見た機能的な仕様をプログラムに明示する．手続き呼び出しにおける呼ばれる側の機能仕様を用いて，プログラム全体の検査を呼び出し元と呼ばれる側の 2 つの検査に分離する．しかし，モジュラー検証には，手続きごとの情報しか調べないのでコールシーケンスに関わる大域的な情報が不足し，誤警告（spurious alarm）が発生するという問題がある．さらに，産業界で普及している C プログラムは，関数間での大域変数を用いたデータの受け渡しや，関数ポインタを介した間接的な関数呼び出しを多用する．関数ポインタを用いたコールバック型のプログラムでは誤警告や不具合の見過ごしが起こりやすい[8),18)]．

本稿では，F-Soft [14] を拡張した新しい有界モデル検査ツール VARVEL [22] を用いて，モジュラー検証における誤警告の改善を報告する．VARVEL は DbC の導入により逐次的な C プログラムのモジュラー検証を可能とする．また，関数ポインタについて機能仕様を記述する DbC の記法と振舞いサブタイピング[16] の考え方に則った関数ポインタの検査手法を提案する．さらに，教育用の OS である MINIX [21] の公開ソースコードを対象とした実験により，誤警告の削減や不具合の検出の観点から提案手法の効果を示す．

本稿の構成は次のとおりである．2 章ではモデル検査法とモジュラー検証の背景を概説する．3 章では有界モデル検査ツール VARVEL の概要と DbC をどのようにサポートするかを述べる．4 章では関数ポインタのための DbC 記法の拡張と検査手法を議論する．5 章では実験の方法および結果を述べる．6 章では関連研究と比較し，続いて結論を 7 章に述べる．本稿のコード断片は実験に用いたプログラムから得たものである．

2. 背　　景

ソフトウェアモデル検査は，モデル検査法[4] のプログラム検証への適用である．モデル検査法では対象を有限状態遷移システムとする．調べたい性質（プロパティ）を時相論理で表現し，状態空間を網羅的に探索してプロパティの成立を判定する．モデル検査法には状態爆発と呼ばれる本質的な問題があり，状態遷移の組合せが増えると計算量が指数関数的に増える．SAT に基づく有界モデル検査法（BMC）は，探索の深さを制限し，その範囲内を網羅探索することにより，スケーラビリティの問題を軽減する．BMC は，探索範囲外に不具合が存在するかどうかは分からないが，不具合の検出には有効である[3]．

一般に，BMC を用いたプログラム検査ツールは呼ばれる側の手続きをインライン展開する[5),14]．これは状態爆発の問題を悪化させ，スケーラビリティを損ねる結果となる．さらにフローセンシティブな解析が必要な場合，呼び出し元の手続きもすべて考慮する必要があり，1 度の解析に必要なプログラムの規模は巨大になる．プログラムが巨大になり BMC の探索の深さを超えると，不具合の見過ごしが発生しうる．

モジュラー検証は，DbC の考え方を採用することにより，拡張静的チェッカ[9),11] として実現されている．DbC は元々は堅牢なソフトウェアを構築するアプローチとして提唱された[17]．DbC では手続きの機能仕様を事前および事後条件からなるコントラクトとして明示する．事前・事後条件はそれぞれ手続きの開始・終了時点で成立しなければならない．呼ばれる側の手続き $Proc$ が事前条件 Pre と事後条件 $Post$ を持つとき，呼び出し元の手続きにおいては，$Proc$ を呼び出す直前の状態 S で Pre が成り立ち，呼び出し復帰後の状態 R では $Proc$ 本体がどのように実行されるかに関知せずに $Post$ が成り立つことを保証すればよい．この保証すべき条件は次のように表せる．

$$S \Rightarrow Pre,\quad Pre \wedge Post \Rightarrow R \quad (\Rightarrow \text{は論理の含意}) \tag{1}$$

呼ばれる側の手続き $Proc$ においては，Pre が成立すれば手続きの本体 $Body$ を実行し，実行後の状態が $Post$ を満たすことを保証しなければならない．この保証すべき条件は，Dijkstra の最弱事前条件[6] の考え方を用いて，プログラム S の事後状態が条件 Q を満たすような事前状態が満たすべき最も弱い条件を $wp.S.Q$ とすると，次のように表せる[11]．

$$Pre \Rightarrow wp.Body.Post \tag{2}$$

呼び出し元と呼ばれる側の手続きを，それぞれ式 (1) と (2) に従って独立に検査するので，モジュラー検証が実現できる．しかし，モジュラー検証では，手続きごとの情報しか調べないのでコールシーケンスに関わる大域的な情報が不足し，誤警告（spurious alarm）が発生することが知られている[8),18]．

産業界で普及している C プログラムでは，大域変数を用いた関数間でのデータの受け渡しが広く行われている．また，デバイスドライバや GUI フレームワークといったコールバック型プログラムでは，関数ポインタを介した関数呼び出しを多用する．モジュラー検証を行ううえで，大域変数や関数ポインタによる誤警告を減らす工夫が必要である．

大域変数によるデータの受け渡しでは，様々な関数が大域変数をどう更新するかを把握することが難しく，誤警告が起こりやすい．この誤警告は，大域変数の値を不変条件によって制限することにより軽減できる．プログラムの実行前後の不変条件を Inv，Inv' とした場合，式 (1)，(2) は次のようになる．

$$S \Rightarrow Pre \wedge Inv,\quad Pre \wedge Inv \wedge Post \wedge Inv' \Rightarrow R \tag{3}$$

$$Pre \wedge Inv \Rightarrow wp.Body.(Post \wedge Inv') \tag{4}$$

他方，関数ポインタを介した関数呼び出しでは，呼ばれる関数は実行時に決まり，そのコントラクトを静的に特定することが難しく，誤警告や不具合の見過ごしが起こりやすい[8),18]．さらに，C 言語の既存の DbC 記法[2] は，関数ポインタをコントラクトの記述対象としておらず，関数ポインタを介した関数呼び出しに対してコントラクトを用いた検証を行うことができない．本稿では関数ポインタにコントラクトを付与する DbC 記法と，関数ポインタを用いたプログラムの検査方法を提案する．さらに，教育用の OS である MINIX[21] の公開ソースコードを対象とした実験により，誤警告の削減，コントラクトに違反する不具合の検出といった観点で提案手法の効果を示す．

3. VARVEL の概要

VARVEL は，有界モデル検査法[3] を用いて逐次的な C プログラムを検証するツールである[22]．VARVEL は，**表 1** に示した F-Soft の典型的なプログラミングエラーの検出機能[14] に加えて，C 標準ライブラリ関数のスタブを提供し，さらに DbC によるモジュラー検証を

表 1　F-Soft/VARVEL が検出できる典型的なプログラミングエラー
Table 1　Typical programming errors detectable by F-Soft/VARVEL.

検査の種類	検出できるエラー
ポインタ有効性	NULL/未初期化/解放済みポインタによる参照
配列境界	上限/下限違反（添字とポインタによるアクセス）
文字列操作	バッファオーバフロー/アンダフロー
メモリ管理	メモリリーク

サポートする．

3.1　VARVEL の動作

VARVEL は F-Soft の機能により，C プログラムを論理式に変換して有界モデル検査を行う．まず，F-Soft は入力として与えられた ANSI-C（C90）準拠のソースコードから，制御フローグラフ（CFG）を構築し，CFG を表す論理式 C を生成する．論理式 C は変数ごとの値の遷移を表す式の論理積であり，各遷移式はプログラムのある位置から次の位置に制御が移る際に変数の値がどう変わるか示す．次に，F-Soft はプロパティを表す論理式 P を生成する．たとえば，NULL ポインタ参照が起きないというプロパティは，プログラム上の位置を指す組み込み変数を pc，ポインタ ptr の参照外しの記述位置を l とすると，$(pc = l) \Rightarrow (ptr \neq NULL)$ となる．最後に，F-Soft は，条件がいつかは成り立つことを示す時相論理の演算子を $\Diamond$ として，式 $C \wedge \Diamond \neg P$ を満たす変数値の割当てがあるかどうかを有界モデル検査により調べる．変数値の割当てがあれば，P への反例として出力する[12]．

関数呼び出しについて，CBMC[5] と同様に，F-Soft は呼ばれる関数の本体のソースコードを呼び出し元のソースコード内にインライン展開する⋆1．関数ポインタを介した関数呼び出しについてはポインタ解析[20] によって関数ポインタにアドレスが代入される関数を特定し，インライン展開を行う．インライン展開によってスケーラビリティは低下する．ただし，VARVEL は 3.2 節の DbC の基本機能によるモジュラー検証をサポートしており，スケーラビリティの問題を緩和する．

与えられたソースコードから変数の値を決められない場合に，F-Soft は近似を導入する．たとえば，大域変数はプログラム開始時点ですべての値をとりうるとし，ソースコードを与えていない関数は，副作用を持たず，戻り値はすべての値をとりうることとする．近似により誤警告が生じることがあるが，この問題は，DbC 記法を用いて値を制限する記述を行うことにより，緩和できる．

3.2　DbC の基本機能

F-Soft は，CBMC[5] や Bogor[19] と同様に，変数の値を制限する手段として `__assert` と `__assume` という 2 つのプリミティブを関数の形式で提供する．プリミティブに与える条件を C，プリミティブを評価する前後の状態を表す式をそれぞれ S，S' とすると，`__assert(C)` は $S \Rightarrow C$ が成り立つかどうかを検査すべきことを示す．`__assume(C)` は $S' = S \wedge C$ の状態遷移があることを仮定する．検証者がこれらのプリミティブを用いてソースコードを補う情報をツールに与えることにより，近似による誤警告を削減できる．しかし，プリミティブの使い方は明確には決まっておらず，実際の作業では検査効率を向上させる系統的な使い方を示す必要がある．

⋆1 コールツリーを調べて再帰呼び出しを検知した場合は，一定の繰返し回数だけインライン展開を行う．

表 2　VARVEL の DbC 記法
Table 2　DbC notation of VARVEL.

仕様の種別	説明
@invariant C	論理式 C は大域データの不変条件である
@pre C	論理式 C は関数の事前条件である
@post C	論理式 C は関数の事後条件である
@param[out] v	変数 v を関数が更新する

組み込み変数・関数	説明
__return	関数の戻り値
__length(a)	配列 a の要素数を返す
__strlen(s)	文字列 s の文字数を返す
__old(v)	関数開始時の変数 v の値を返す

VARVEL は DbC を明示的にサポートするために DbC の記法（**表 2**）を提供する．構造体メンバのアクセス，ポインタの参照外しなど，C 言語の式で用いる表現を表せる．

VARVEL は，モジュラー検証を実現するために，C プログラムのコメントに記述された事前・事後条件を先のプリミティブを用いたプログラムコードに変換して，有界モデル検査を行う．たとえば，検証対象の関数については，その事前条件 Pre を `__assume(Pre)` に，事後条件 $Post$ を `__assert(Post)` に変換して，それぞれ関数本体の前と後に挿入する[13]．この変換により，VARVEL は $Pre \wedge C \wedge \Diamond \neg Post$ を満たす変数値の割当てがあるかどうかを有界モデル検査により調べることができる．変数値の割当てがあれば，それは事後条件への反例である．検証者が DbC 記法に則って関数の機能仕様を記述することで，VARVEL ではプリミティブを系統的に使うことができる．

4.　関数ポインタの DbC

DbC の基本的な機能（3.2 節）によるモジュラー検証だけでは，関数ポインタを用いたプログラムを検証することは難しい．本稿では，振舞いサブタイピング[16] の考え方に則って関数ポインタを検査する方法を提案する．

4.1　関数ポインタの問題

関数ポインタを介して呼ばれる関数は，アドレスを関数ポインタに代入することにより実行時に決まり，そのコントラクトを静的に特定することは難しい．また，呼び出し元と呼ばれる側の関数が分業して開発される場合，一方の関数の開発者は，他方の関数の宣言とコン

ファイル A
```
10 : void (*task)( int );
11 : void callTask( int n ){
12 :   if ( 0 <= n )
13 :     (*task)( n );
14 : }
```

ファイル B
```
20 : void execTask(){
21 :   task = doTask;
22 :   callTask( 0 );
23 : }
24 : /** @pre  0<x   */
25 : void doTask( int x ){
26 :   /* x が 0 より大きいことを想定した処理 */
27 : }
```

図 1　関数ポインタを用いるプログラムの例
Fig. 1　Example with a function pointer.

トラクトしか参照できないことがある．

たとえば，**図 1** で，ファイル A の関数 callTask は，その引数 n が 0 以上の場合に，n を引数として大域変数である関数ポインタ task を介した関数呼び出しを行う（行 12–13）．ファイル B の関数 execTask は，task に関数 doTask のアドレスを設定し，0 を引数として callTask を呼び出す（行 21–22）．ファイル A と B を与えると，VARVEL はポインタ解析により task を介して呼ばれる doTask を特定し，execTask のコードに callTask と doTask のコードをインライン展開する．VARVEL は doTask の事前条件 0<x（行 24）をプリミティブに変換して，task を介した関数呼び出し（行 13）の前に挿入し，有界モデル検査により事前条件への反例を出力する．しかし，ファイル B だけを与えると，VARVEL は execTask と doTask を独立に検査することになり，前述の事前条件への反例を出力できない．

この問題に対して，本稿では，関数ポインタ自体に機能仕様（仮コントラクト）を定義する．検査対象のプログラムに関する次の 2 つの条件を考える．

(1) 関数ポインタを介した関数呼び出しの位置で仮コントラクトが満たされるとの仮定の下で，検査対象のプログラムはプロパティ P を満たす．

(2) 関数ポインタを介して実際に呼ばれる可能性のあるすべての関数の機能仕様（実コントラクト）と仮コントラクトとは一貫性を持つ（実コントラクトが成り立てば仮コントラクトも成り立つ）．

この 2 つの条件がともに成り立てば，「関数群が実コントラクトを満たすならば，検証対象のプログラムはプロパティ P を満たす」が成り立つ．この考え方に従って，仮コントラクトの DbC 記法と，仮コントラクトを用いたモジュラー検証および実コントラクトと仮コントラクトの一貫性検査からなる関数ポインタの検査方法を提案する．この検査方法には，実際に呼ばれる関数本体のインライン展開が不要になるという長所がある．

表 3　仮コントラクトの DbC 記法
Table 3　DbC notation for formal contracts

仕様の種別	説明
@variable_contract (f)	以降の事前・事後条件は関数ポインタ f に関する
@array_contract (f)	以降の事前・事後条件は関数ポインタ配列 f の各要素に関する
@array_elem_contract (f, i)	以降の事前・事後条件は関数ポインタ配列 f の i 番目の要素に関する
@type_contract (T)	以降の事前・事後条件は T 型の関数ポインタに関する
@field_contract (Tf)	以降の事前・事後条件は T 型構造体の関数ポインタフィールド f に関する
@field_array_contract (T, f)	以降の事前・事後条件は T 型構造体の関数ポインタ配列フィールド f の各要素に関する
@field_array_elem_contract (T, f, i)	以降の事前・事後条件は T 型構造体の関数ポインタ配列フィールド f の i 番目の要素に関する

組み込み変数	説明
__aN (N=1..)	関数ポインタの N 番目の引数

ファイル A（仮コントラクトの記述後）
```
10a: /** @variable_contract( task )
10b:       @pre 0<=__a1   */
10 : void (*task)( int );
11 : void callTask( int n ){
12 :   if ( 0<=n )
13 :     (*task)( n );
14 : }
```

図 2　仮コントラクトの記述例
Fig. 2　Example with a formal contract.

ファイル A（プリミティブへの変換後）
```
10 : void (*task)( int );
11 : void callTask( int n ){
12 :   if ( 0<=n )
13a:     __assert( 0<=n );
13b:     __assume( 0<=n );
13 :     (*task)( n );
14 : }
```

図 3　プリミティブへの変換例
Fig. 3　Example with primitives for Fig. 2.

4.2　仮コントラクトの記法とモジュラー検証

仮コントラクトの DbC 記法を**表 3** に示す．産業界の C プログラムでは保守の観点から複雑な表現を用いないことが多く，表 3 の記法は，変数，1 次元配列，入れ子になっていない構造体といった単純な表現を対象とした．関数ポインタへのポインタや関数ポインタを引数に持つ関数ポインタなど，より複雑な表現には対応していない．仮コントラクトは，関数ポインタ型の変数を指定し，その後に事前・事後条件を列挙する．**図 2** の例では，関数ポインタ task について，事前条件を定義している（行 10a–10b）．

VARVEL は，関数ポインタを介した関数呼び出しを含むプログラムを，**図 4** のアルゴリズム check1 によって検証する．check1 には，対象プログラム prog と，prog で間接的

```
check1( prog, SRCs ) : Result
1.   FCs ← findFormalContracts( SRCs );
2.   foreach fc in FCs
3.     while ( loc ← findCall( fc, prog ) )
4.       prog ← insertPrimitives( loc, fc );
5.   return doSWMC( prog )
```

図 4　関数ポインタを介した関数呼び出しを行うプログラムを検査するアルゴリズム
Fig. 4　Algorithm to model-check a program with calls through function pointers.

表 4　仮コントラクトのソースコードへの変換ルール
Table 4　Rules to convert formal contracts to source codes.

	仕様の形式	ソースコード
a	`@pre P`	`__assert( P'); __assume( P');`
a'		`__assert( !n==i || P'); __assume( !n==i || P');`
b	`__old( v )`	`__old_v=v;`
c	`@param[out] v`	`v=__NONDET__();`
d	`@post Q`	`__assume( Q');`
d'		`__assume( !n==i || Q');`

ルール a' と d' は，仮コントラクトが配列要素に関する場合に用いる．
P'：P に現れる仮引数を実引数で置換した式．n：プログラムコードにおける配列要素の添え字．
i：仮コントラクトにおける配列要素の添え字．`__NONDET__()`：非決定的に任意の値を返す関数
Q'：Q に現れる仮引数を実引数で，式`__old( v )`を変数`__old_v`で置換した式．

な関数呼び出しに用いられる関数ポインタの仮コントラクトを含むような関連プログラム SRCs を与える．check1 は，まず，SRCs から仮コントラクトの集まり FCs を抽出し（行 1），各仮コントラクトごとに，仮コントラクトで指定された関数ポインタを介した関数呼び出しの位置 loc を特定する（行 2–3）．次に，loc ごとに，**表 4** の変換ルールに従って仮コントラクトをプリミティブを用いたプログラムコードに変換し，prog に挿入する（行 4）．表 4 のルール a と b については仮コントラクトから変換されたプリミティブを loc の前に挿入し，ルール c と d については loc の後に挿入する．最後に，check1 は，プリミティブが挿入されたプログラムに対してモデル検査を行い，その結果を返す（行 5）．結果には，プロパティとそのプログラム上の位置，不具合の有無を示す情報，不具合がある場合は反例が含まれる．

たとえば，図 2 で，対象プログラムとして関数 callTask を，関連プログラムとしてファイル A を与えると，check1 は，ファイル A から関数ポインタ task についての仮コントラクト（行 10a–10b）を抽出し，task を介した関数呼び出し (*task)(n) の位置（行 13）を特定する．

```
check2( prog, SRCs ) : Result
1.   Result ← φ
2.   FCs ← findFormalContracts( SRCs )
3.   foreach fc in FCs
4.     while ( loc ← findAssignment( fc, prog ) )
5.       ap ← findAssignedProg( loc, prog )
6.       ac ← findActualContract( ap, SRCs )
7.       Result ← Result ∪ checkConsistency( ac, fc, loc )
8.   return Result
```

図 5　実・仮コントラクトの一貫性を検査するアルゴリズム
Fig. 5　Algorithm to check the consistency between actual and formal contracts.

check1 は変換ルール a に従い，仮コントラクトの事前条件 `0<=__a1` を，仮引数 `__a1` を実引数 n に置き換えて，プリミティブを用いたプログラムコード `__assert(0<=n); __assume(0<=n)` に変換し，task を介した関数呼び出しの前に挿入する（**図 3** 行 13a–13b）．プリミティブ挿入後の callTask をモデル検査により調べると，`__assert`（図 3 行 13a）への反例は出力されない，すなわち，対象プログラムは仮コントラクトを守って関数ポインタを介した関数呼び出しを行っている．

4.3　実コントラクトと仮コントラクトの一貫性検査

振舞いサブタイピングでは，オブジェクト指向言語において，スーパタイプの事前条件がサブタイプの事前条件を満たし，かつ，サブタイプの事後条件がスーパタイプの事後条件を満たす場合に，スーパタイプをサブタイプに置換できる[16]．本稿では，手続き型言語 C における関数ポインタと関数ポインタを介して呼ばれる関数の一貫性を，振舞いサブタイピングにおけるスーパタイプとサブタイプの置換可能性と考える．すなわち，仮コントラクトの事前条件は実コントラクトの事前条件を満たし，実コントラクトの事後条件は仮コントラクトの事後条件を満たさなければならない．

$$Pre_{formal} \Rightarrow Pre_{actual} \tag{5}$$

$$Post_{actual} \Rightarrow Post_{formal} \tag{6}$$

実コントラクトと仮コントラクトの一貫性 (5)，(6) は**図 5** に示すアルゴリズム check2 によって検査する．check2 には，対象プログラムとして関数ポインタを介した関数呼び出しを行うプログラム prog と，prog が関数呼び出しに用いる関数ポインタの仮コントラクトと呼ばれる関数の実コントラクトを含むような関連プログラム SRCs を与える．まず，check2 は検査結果を格納する Result を空にする（行 1）．次に，check2 は SRCs から仮コントラ

```
1 :logic QF_LIA            ; 限量子なしの線形演算
2 :extrafuns (( a1 Int )) ; 第 1 引数
3 :assumption              ; (; から行末まではコメント)
4 (not (implies ; 含意の否定
5   (<= 0 a1)   ; 含意の前件 : 仮事前条件 0<=__a1
6   (<  0 a1))) ; 含意の後件 : 実事前条件 0<x
```

図 6　実・仮コントラクトの一貫性検査スクリプトの例
Fig. 6　Example of a script to check the consistency between actual and formal contracts.

クトの集まり FCs を抽出する（行 2）．FCs 中の各仮コントラクト fc ごとに関数ポインタに関数のアドレスを設定する位置 loc を探し，loc ごとに関数ポインタにアドレスが設定される関数 ap の実コントラクト ac を探す（行 3–6）．そして，ac と fc が一貫しているというプロパティを制約解消問題として検査し，結果を Result に追加する（行 7）．一貫性の検査では，式 (5)，(6) を否定した式をそれぞれ制約ソルバに与え，これらの式を満たすような変数への値の割当てがあるかどうかを調べる．割当てがあれば，それは一貫性への反例である．check2 は一連の繰返しを終えると，Result を返す（行 8）．Result は，プロパティ，関数ポインタを介した関数呼び出しの位置，不具合の有無を示す情報，不具合がある場合は反例を含む．

たとえば，対象プログラムとして図 2 の関数 execTask を，関連プログラムとして図 2 のファイル A と図 1 のファイル B を与えると，check2 は，ファイル A から関数ポインタ task の仮コントラクト@pre 0<=__a1（図 2 行 10a–10b）を抽出する．check2 は，execTask のコードから task に関数アドレスが設定される位置（図 1 行 21）を見つけ，アドレスが設定される関数 doTask の実コントラクト@pre 0<x（図 1 行 24）を見つける．check2 が見つけた事前条件に関する実コントラクト 0<x（図 1 行 24）と仮コントラクト 0<=__a1（図 2 行 10b）の一貫性を検査するためのスクリプト（SMT-LIB 形式）を**図 6** に示す．このスクリプトは，関数 doTask の引数 x と関数ポインタ task の引数__a1 を同じ名前の引数 a1 とした実コントラクト 0<a1 と仮コントラクト 0<=a1 との間で，式 (5) が成立しないような a1 が存在するかを調べる内容となっている．図 6 のスクリプトを Yices[7] に与えて，論理演算と線形演算の範囲で制約を解くと，変数値の割当て (= a1 0) を得る，すなわち，第 1 引数が 0 の場合に，仮および実コントラクトの事前条件の間に一貫性がないことが分かる．

5. 実　　験

4 章で述べた，関数ポインタを持つプログラムを検証する提案手法を，ソースコードが公開されている教育用 OS である MINIX[21]（version 3.1.1）に適用した．

表 5　実験対象としたソースコード
Table 5　Target source codes for the experiment.

層	機能	サイズ	関数	関数ポインタの形態
サーバ	ファイルシステム	718	29	構造体フィールド
ドライバ	フロッピーディスク	763	37	構造体フィールド
カーネル	システムコール	1,107	38	配列

MINIX はマイクロカーネルの考え方で実装されており，プロセス間メッセージ通信を多用する．信頼性を要求されるシステムソフトウェアであること，メッセージの受信において関数ポインタを利用していることから，実験対象とした．OS 本体のプロセスは，サーバ，デバイスドライバ，カーネルの 3 層に分けられる．本実験では，**表 5** に示す各層の機能から関数ポインタを介した関数呼び出しを行うソースコードを含むファイルを選んだ．表 5 の列において，サイズと関数はそれぞれ VARVEL ツールに与えたソースコードの行数と関数の個数を表し，関数ポインタの形態は関数ポインタが C 言語のどの言語要素として宣言されたかを示す．なお，MINIX のヘッダファイル中の関数ポインタを含む宣言 75 個を調べたところ，これらの宣言は表 3 の記法で仕様を記述しうるものであった．また，表 5 で選んだファイルに含まれる関数はすべて，表 2 と 3 の記法により仕様を記述できた．

本実験では，まず，コントラクトとして事前・事後条件のみを記述し，VARVEL を用いたモジュラー検証を行って，誤警告の発生を調べた．次に，不変条件と関数ポインタの仮コントラクトを追加し，モジュラー検証と実・仮コントラクトの一貫性検査を行った．

5.1　事前・事後条件のみを用いたモジュラー検証

実験対象の各関数には，文献 21) とソースコードを参考に，引数と戻り値についてコントラクトを記述した．VARVEL を用いたモジュラー検証の結果を**表 6** に示す．表 6 の列において，Pre，Post，Alarm はそれぞれ事前条件，事後条件，警告の個数であり，Bug，GV，FP，Others はそれぞれ不具合と考えられる警告，大域変数がすべての値をとりうるという近似による誤警告，関数ポインタを介した関数呼び出しに副作用はなく，かつ戻り値はすべての値をとりうるという近似による誤警告，その他の近似による誤警告の個数である．共通ヘッダ層は他層から参照されるヘッダファイルの集まりであり，関数宣言とコントラクトを含む．

大域変数による誤警告はサーバ層で多く発生した．たとえば，MINIX においてプロセス番号は −4（NR_TASKS）以上であるが，VARVEL はプロセス番号を表す大域変数が −4

表 6　事前・事後条件のみを用いたモジュラー検証の結果
Table 6　Result of modular verification using pre-/post-conditions.

層	Pre	Post	Alarm	Bug	GV	FP	Others
サーバ	46	37	20	2	11	6	1
ドライバ	41	18	2	1	0	1	0
カーネル	77	39	3	2	0	1	0
共通ヘッダ	37	71					

表 7　不変条件と仮コントラクトを追加したモジュラー検証の結果
Table 7　Result of modular verification using invariants and formal contracts.

層	Inv	F.C.	F.A.	Alarm	Bug	Others
サーバ	11	0	69	11	6	5
ドライバ	0	29	38	1	1	0
カーネル	0	22	22	20	20	0
共通ヘッダ	0	17				

未満をとりうるとし，プロセス番号を引数とする関数呼び出しで事前条件違反を警告した．また，関数ポインタを介した関数呼び出しは，各層において誤警告の原因となった．たとえば，応答メッセージを要求メッセージと同じ構造体変数で受け取る場合には構造体フィールドの値が変更されることがあるが，関数ポインタを介した関数呼び出しには副作用がない，すなわち構造体フィールドの値は変わらないとするために，後続のソースコードを正しく解析できなかった．この問題の解決方法を次節に提案する．

5.2　不変条件・仮コントラクトを用いたモジュラー検証

5.1 節の誤警告を削減するために，大域変数の値を制限する不変条件と，関数ポインタの仮コントラクトを追記した．関数ポインタに設定された関数アドレスを調べ，これらの関数のコントラクトと仮コントラクトが式 (5)，(6) を満たすようにした．また，関数ポインタを介した関数呼び出しの位置には，仮コントラクトに対応したプリミティブを挿入した．

VARVEL を用いたモジュラー検証の結果を**表 7** に示す．表 7 の列において，Inv，F.C.，F.A.，Alarm はそれぞれ不変条件，仮コントラクト，仮コントラクトから変換されたプリミティブ，警告の個数であり，Bug，Others はそれぞれ，不具合と考えられる警告，大域変数および関数ポインタ以外の近似による誤警告の個数である．

不変条件と仮コントラクトに対応したプリミティブにより，警告は増えたが，誤警告は減っている．警告の正しさを「不具合数/警告数」とすると，表 6 での 20%（5/25）に対して表 7 では 84%（27/32）と改善されている．大域変数によるデータの受け渡しを行うサーバ層においては不変条件によって，関数ポインタを介した関数呼び出しを行う各層においては仮コントラクトによって，誤警告を削減できたといえる．

```
  /**
 1: @pre  CANCEL <= op && op <= DEV_STATUS
 2: @pre  -NR_TASKS <= proc && proc < NR_PROCS
 3: @post !( op == CANCEL ) || ( __return <= OK )
  */
  int dev_io( int op, dev_t dev, int proc, ···· ){
 4:  dp = &dmap[(dev >> MAJOR) & BYTE];
       :
 5:  dev_mess.m_type   = op;
 6:  dev_mess.PROC_NR  = proc;
       :
 7:    __assert( 仮コントラクトの事前条件 ); __assume( 仮コントラクトの事前条件 );
 8:    (*dp->dmap_io)(dp->dmap_driver, &dev_mess);
 9:    __assume( 仮コントラクトの事後条件 );
       :
10:  return(dev_mess.REP_STATUS);
  }
```

図 7　不具合を含むソースコードの例
Fig. 7　Example of source codes with a detected failure.

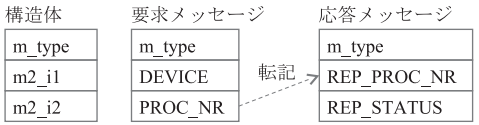

図 8　MINIX のデバイスドライバの要求応答メッセージ
Fig. 8　Request and reply messages in MINIX device drivers.

図 7 は，仮コントラクトを用いたモジュラー検証で見つかった不具合を含むソースコードの例である．関数ポインタ `dp->dmap_io` を介した関数呼び出し（行 8）は，要求メッセージの送信と応答メッセージの受信を行う．MINIX のメッセージ通信の様子を**図 8** に示す．要求と応答メッセージは同じ構造体変数に格納される．構造体を要求メッセージとして利用する場合には，フィールド `m2_i1` と `m2_i2` にそれぞれ別名 `DEVICE` と `PROC_NR` でアクセスする．`DEVICE` には処理の依頼先であるデバイス番号が，`PROC_NR` には要求元のプロセス番号が設定される．応答メッセージとしての利用では，フィールド `m2_i1` と `m2_i2` にそれぞれ別名 `REP_PROC_NR` と `REP_STATUS` でアクセスする．`REP_PROC_NR` には要求元のプロセ

表 8　実コントラクトと仮コントラクトの一貫性検査の結果
Table 8　Result of consistency checks between actual and formal contracts.

層	設定位置	Pre 反例	Post 反例
サーバ	7	0	0
ドライバ	12	0	0
カーネル	29	0	0

ス番号が転記され，REP_STATUS にはデバイスによる処理結果が設定される．メッセージの送受信に失敗した場合には，構造体の内容は変わらない．すなわち，dp->dmap_io を介した関数呼び出し（行 8）では，要求メッセージが設定された構造体 dev_mess を用いてメッセージ通信を行い，通信に成功した場合は dev_mess の内容は応答メッセージとなり，通信に失敗した場合は要求メッセージのままとなる．関数ポインタ dp->dmap_io の仮コントラクト（事後条件）に対応したプリミティブ（行 9）は，通信成功の場合は dev_mess の内容が図 8 に示すように変わり，失敗の場合は dev_mess の内容が変わらないことを表す．

検出された不具合は，次のようなものである．関数 dev_io の引数 op は事前条件（行 1）を満たす値 CANCEL であり，引数 proc は事前条件（行 2）を満たす 0 より大きい値とする．proc の値は要求メッセージのフィールド m2_i2（別名 PROC_NR）に設定される（行 6）．関数ポインタ dp->dmap_io を介した関数呼び出しで，通信に失敗した場合，m2_i2 の値は変わらず 0 以上である．m2_i2（別名 REP_STATUS）を戻り値として返すと（行 10），引数 op が CANCEL であれば戻り値は OK（値は 0）以下であるという事後条件（行 3）に違反する．

5.3　実コントラクトと仮コントラクトの一貫性の検査

関数ポインタの値を初期化する変数宣言と関数ポインタに値を代入する文ごとに 4.3 節に示した実コントラクトと仮コントラクトとの一貫性を検査するための制約ソルバ用のスクリプトを記述する．本実験では，制約ソルバとして Yices[7] を用いて検査を行った．検査結果を**表 8** に示す．

表 8 の列において，設定位置，Pre 反例，Post 反例はそれぞれ関数ポインタに値を設定する位置，式 (5) が成立しない場合，式 (6) が成立しない場合の個数である．設定位置ごとに実コントラクトと仮コントラクトの一貫性を検査したところ，一貫性への反例は見つからなかった．一貫性 (5)，(6) はコントラクトの設計にも注意を向ける効果があったといえる．

6. 関 連 研 究

ソフトウェアモデル検査ツールの実用化では，スケーラビリティの問題を解決しなければならない．手続き呼び出しは，制御を複雑にし，スケーラビリティを悪化させる一因である．

一般に，手続き呼び出しと戻りを表現して解析するには，プッシュダウンオートマトン（PDA）の表現力が必要である．C プログラムの検証ツール SLAM[1] は PDA に基づくモデル検査を行う．PDA 用のモデル検査の計算コストは大きく，SLAM は述語抽象により状態空間の規模を縮小する．

CBMC[5] や F-Soft[14] のような SAT に基づく有界モデル検査ツールは，手続き呼び出しを明示的には表現せず，呼ばれる側の手続きをインライン展開する．CBMC[5] は関数呼び出しをインライン展開して有限状態空間を作り，有界モデル検査法により探索の深さを限定する．ループや再帰呼び出しは n 回だけ展開することにより，無限の繰返しを避ける．F-Soft[14] も CBMC と同様のアプローチをとる．本稿の VARVEL は F-Soft の上に DbC に基づくモジュラー検証を導入し，スケーラビリティの問題を軽減する．

DbC に基づくモジュラー検証では，プログラム全体の検査を手続きごとの検査に分けることによりスケーラビリティを得る．モジュラー検証は拡張静的チェッカ ESC/Java[11] や Caduceus[9] として実現されており，DbC の記法として JML[15] や ACSL[2] が提案されている．ただし，関数ポインタの仮コントラクトは扱っておらず，本稿で提案した記法を追加することが望ましい．

C 言語の関数ポインタを引数とする関数に相当する高階関数用の DbC が Scheme[10] に導入されているが，コントラクトは Eiffel[17] 同様に実行時に検査される．本提案では，仮コントラクトを用いたモジュラー検証および実コントラクトと仮コントラクトの一貫性検査をいずれも静的に行う点が異なる．

7. お わ り に

本稿では，DbC[17] に基づくモジュラー検証において，C プログラムの関数ポインタに仮コントラクトを付与する記法を提案した．また，関数ポインタを介した関数呼び出しが仮コントラクトを守っていることをモジュラー検証で検査し，振舞いサブタイピング[16] の考え方に則って，仮コントラクトと実際に呼び出される関数のコントラクトが一貫していることを制約ソルバにより検査する手法を提案した．さらに，MINIX[21] を対象として，提案した仮コントラクトの記法および検査手法により，誤警告を削減し，不具合を検出した．

今後，DbC を用いたモジュラー検証のより大規模な実験が必要である．MINIX の他の部分についても，モジュラー検証によって不具合を検出できることを確認する．また，ソフトウェアモデル検査ツールが導入する近似により誤警告（spurious error）が発生するとい

う問題については，警告が誤警告であるか否かを自動判定する方法の研究が必要である．

参 考 文 献

1) Ball, T. and Rajamani, S.K.: The SLAM Project: Debugging System Software via Static Analysis, *Proc. POPL 2002*, pp.1–3 (2002).
2) Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y. and Prevosto, V.: ACSL: ANSI/ISO C Specification Language (2008).
3) Biere, A., Cimatti, A., Clarke, E. and Zhu, Y.: Symbolic Model Checking without BDDs, *Proc. TACAS 1999*, pp.193–207 (1999).
4) Clarke, E.M., Grumberg, O. and Peled, D.A.: *Model Checking*, The MIT Press (1999).
5) Clarke, E.M., Kroening, D. and Lerda, F.: A Tool for Checking ANSI-C Programs, *Proc. TACAS'04*, pp.168–176 (2004).
6) Dijkstra, E.W.: *A Discipline of Programming*, Series in Automatic Computation, Prentice Hall (1976).
7) Dutertre, B. and Moura, L.D.: The YICES SMT Solver, Technical Report, Computer Science Laboratory, SRI International (2006).
8) Fähndrich, M., Garbervetsky, D. and Schulte, W.: A Re-Entrancy Analysis for Object Oriented Programs, *9th FTfJP at ECOOP* (2007).
9) Filliâtre, J.-C. and Marché, C.: Multi-prover Verification of C Programs, *Proc. ICFEM'04*, pp.15–29 (2004).
10) Findler, R.B. and Felleisen, M.: Contracts for Higher-Order Functions, *Proc. ICFP'02*, pp.48–59 (2002).
11) Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B. and Stata, R.: Extended Static Checking for Java, *Proc. PLDI'02*, pp.234–245 (2002).
12) Hashimoto, Y. and Nakajima, S.: Modular Checking of C Programs Using SAT-Based Bounded Model Checker, *Proc. APSEC 2009*, pp.515–522 (2009).
13) Hashimoto, Y. and Nakajima, S.: Modular Checking with Model Checking, *Proc. SSV'09*, pp.105–122 (2009).
14) Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M.K., Kahlon, V., Wang, C. and Yang, Z.: Model Checking C Programs Using F-Soft, *Proc. ICCD'05*, pp.297–308 (2005).
15) Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P. and Zimmerman, D.M.: JML Reference Manual (2008).
16) Liskov, B.H. and Wing, J.M.: A Behavioral Notion of Subtyping, *ACM TOPLAS*, Vol.16, No.6, pp.1811–1841 (1994).
17) Meyer, B.: Applying Design by Contract, *IEEE Computer*, Vol.25, No.10, pp.40–51 (1992).
18) Meyer, B.: *The Dependent Delegate Dilemma*, *Engineering Theories of Software Intensive Systems*, Springer-Verlag (2005).
19) Robby, Rodríguez, E., Dwyer, M.B. and Hatcliff, J.: Checking Strong Specifications Using an Extensible Software Model Checking Framework, *Proc. TACAS'04*, pp.404–420 (2004).
20) Steensgaard, B.: Points-to Analysis in Almost Linear Time, *Proc. POPL 1996*, pp.32–41 (1996).
21) 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).
22) 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).

(平成 22 年 11 月 11 日受付)
(平成 23 年 5 月 14 日採録)

橋本　祐介（学生会員）

平成元年東京大学理工系大学院相関理化学専攻修士課程修了．同年日本電気株式会社入社（現在に至る）．平成 20 年総合研究大学院大学複合科学研究科情報学専攻博士後期課程．

中島　　震（正会員）

1981 年東京大学大学院理学系研究科修士課程修了．NEC，法政大学を経て，2004 年情報・システム研究機構国立情報学研究所教授．2005 年総合研究大学院大学教授（併任）．この間，1988〜1989 年米国オレゴン大学客員研究員，2001〜2007 年科学技術振興機構 PRESTO および SORST 研究員（兼務），2004〜2007 年北陸先端科学技術大学院大学 JJREX 客員教授．学術博士（東京大学）．2002 年度山下記念研究賞，2003 年度日本ソフトウェア科学会論文賞受賞．ディペンダブルソフトウェア工学，形式手法，モデル検査，モデリングに関する研究に従事．日本ソフトウェア科学会，ACM 各会員．