2007－EMB－6 (7)
2007／12／4

Cプログラム・モデル検査におけるDbC

中島 震　池田健次郎　橋本祐介　岩崎新一
国立情報学研究所 / 総合研究大学院大学
NEC ソフトウェアエンジニアリング本部

あらまし　形式検証の技術を利用したCプログラムの自動検査ツールが実用的な段階に達している。有界モデル検査法に基づくツールのスケーラビリティに関する問題を解決するために、Design by Contract (DbC) の考え方を導入してモジュラー検査を可能にする。C言語では関数ポインタを利用したプログラミング方法が多用されるが、DbCを採用した現状の自動解析ツールは関数ポインタの取り扱いが不十分である。本稿では関数ポインタを考慮したDbCを提案し、振舞いサブタイピングの規則を検査法として採用する。これにより、不具合検出箇所の精度を向上させることができる。

キーワード　プログラムの形式解析、関数ポインタ、モジュラー検査

DbC in Model-Checking of C Programs

Shin NAKAJIMA　Kenjiroh IKEDA　Yusuke HASHIMOTO　Shin'ichi IWASAKI
NII / SOKENDAI
Software Engineering Division, NEC

Abstract　Automatic analysis tools for C programs, based on the formal verification technology, see much progress now to become industrial strength. Unfortunately, those tools to adapt the bounded model-checking methods have drawbacks in the scalability. Design by Contract (DbC) might be an approach to remedy the situation. DbC tools for C programs, however, do not consider function pointers in enough details although such progarms are many. This paper proposes a new style of DbC to consider the function pointers, and adapts the behavioral subtyping rules for the check. The proposed method can detect potential errors, due to function pointers, precisely.

Keywords　Formal Analysis of Programs, Function Pointers, Modular Checking

1 はじめに

高信頼ソフトウェア開発のための基礎技術として形式手法が注目を集めている [14]。元来、「開発過程で形式検証を行うことで高い信頼性を持つソフトウェアを構築する (Correctness by Construction)」ための技術確立を目的とした研究分野であった。特に、長い研究の発展の中で形式検証の技術が進展した。最近では、これを要素技術として利用するプログラムの自動検査法が実用的な段階に達している。実用的なプログラミング言語である C 言語や Java を対象とする自動検査ツールの開発が活発である。先の Correctness by Construction に対して、「開発済のプログラムに内在する不具合を発見すること」が目的であり、a posteriori falsification と呼ぶ。Correctness by Construction と並んで形式手法の代表的な使い方になっている。

実用的なプログラム自動検査ツールとしては、SAT ベースの有界モデル検査法 [15] に基づくツール [3][9][10] と定理証明系や決定手続きを利用した拡張静的検査ツール [5][6][13] がある。万能の自動検査は不可能であり、用いる技法やツールによって検査可能な項目が限定される。しかし、自動的に不具合が発見できるという点が実用上大きなメリットになる。従来からのテスト技法等と組み合わせて相補う形で利用されている。

上記 2 つの手法を比較する場合、有界モデル検査法によるツールはスケーラビリティの面で弱い。一般にプログラムを作成する場合、ライブラリを利用したり、あるいは、適切な大きさの手続きや関数に分割する。しかし、モデル検査法 [2] では手続き呼出しを明示的に取り扱うことが難しく、ソースプログラムをインライン展開する方法をとる。すなわち、規模に対して弱い。

拡張静的検査では、Design by Contract (DbC) の考え方 [12] によってモジュラー検証を実現する。DbC では利用側プログラムと呼び出される側の間で、守るべき条件を決めて、事前・事後条件とする。これによって利用側と呼び出される側を分離することができるため、検査対象を限定することが可能になる。検査のスケーラビリティ面で効果が大きい。

ところで、DbC はモジュール分割を中心とする開発法であって、特定の形式検査法と密着した考え方ではない。いくつかのプリテミティブ (assume、assert 等) を組み合わせることで、DbC スタイルの仕様表現をモデル検査法と統合することが可能である [3][16]。

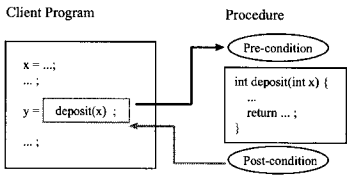

図 1. **Design by Contract** の考え方

```
int bal; /* invariant (bal >= 0)
                  && (bal < MAX); */

/*
  requires (x > 0);
  ensures  (bal == \old(bal) + x)
        && (\result == bal) ;
  modifies (bal);
*/
int deposit (x) int x;
{
  if((x > 0) && (bal + x < MAX)) {
    bal =+ x; return bal;
  } else { return ERR; }
}
```

図 2. **DbC** の記述例

本稿では、有界モデル検査法による C 言語プログラムの自動検査ツール F-Soft[10] に DbC を導入する方法を考察する。特に、既存の C 言語向け DbC 検査ツールで取り扱っていない関数ポインタを中心に議論する。

2 DbC とモジュラー検証

図 1 に示したように Design by Contact (DbC) では、呼ばれる側のプログラム (図では `Procedure`) が事前条件 (Pre_P) と事後条件 ($Post_P$) を持つとする。具体的な記述は、たとえば、図 2 のようなものが考えられる。関数 `deposit` は事前条件 (`requires`)、事後条件 (`ensures`)、副作用対象 (`modifies`) を DbC の仕様として持つ。また、大域変数 `bal` には不変条件 (`invariant`) を追加した。

DbC を用いる場合の検証は以下のように行う。呼出し側プログラム (`Client Program`) は、当該手続きを呼び出す直前に、Pre_P を満たす実行状態 S を作り出す。手続きがどのように実行されるかの

詳細は不問であり、実行完了後の状態 R は $Post_P$ が成り立つと仮定して以降の処理を行えば良い。すなわち、

$$S \Rightarrow Pre_P、\quad Pre_P \wedge Post_P \Rightarrow R \quad \ldots(1)$$

である。一方、Procedure では、Pre_P の仮定下で本体 $Body_P$ を実行した結果、$Post_P$ を成り立たせると考える。E.Dijkstra の最弱事前条件に基づく方法で書き表すと次のようになるだろう。

$$Pre_P \Rightarrow \mathrm{wp}.Body_P.Post_P \quad \ldots(2)$$

不変条件 Inv を考慮する場合は

$$Pre_P \wedge Inv \Rightarrow \mathrm{wp}.Body_P.(Post_P \wedge Inv') \quad \ldots(2')$$

となる。

(1) と (2) からわかるように、呼出し側と呼ばれた側の検査を独立に行うことが可能になり、モジュラー検証を実現することができる。

さらに、DbC のもうひとつの長所に、誤り箇所が詳細に判明するという点がある。たとえば、S が上式 (1) を満たさない場合、誤りは呼出し側プログラムにあることがわかる。

C 言語を対象とする有界モデル検査ツールでは、検証性質の表現手段として時相論理ならびに表明による方法を用いる [3][10]。検査表明 (assert) に加えて、検査対象の実行経路を限定するための仮定 (assume) をプリミティブとして提供する。これらのプリミティブを適宜、組み合わせれば、DbC の考え方を導入することは容易である。たとえば、上の (2) は、

```
assume(Pre); ...; assert(Post);
```

のような形式と考えれば良い。

検証問題としての定式化は上に述べたように明確であるが、実際の適用に際しては多くの難しい課題がある。事前・事後条件は一種の仕様であるため、実現方式の詳細から独立な宣言的な記述が好ましい。しかし、適切な詳細レベルでの表現方法を決めることは容易ではない。文献 [1] は抽象的なアルゴリズムまで表現する場合を扱っている。一方、対象性質を NULL ポインタ参照やバッファ・オーバフロー等の単機能に限定して実用的な自動検査を目指すツールが増えている [8]。

```
/*
  requires (valid_ptr(comp) && (n >= -10));
  ensures  (\result > 0);
*/
int delegate(comp, n)
int (* comp)(int a); int n;
{
  int x;
  if(n > 0) { x = (*comp)(n); }
  else { x = (*comp)(0); }
  return x;
}
```

図 3. 関数ポインタの DbC (1)

```
/*
  requires (n > 0);
  ensures  (\result > 0);
*/
int calc(n) int n; { ... }

int user(m) int m;
{
  return delegate(calc,-1);
}
```

図 4. 関数ポインタの利用例

一般に、事前・事後条件の表現力を豊かにすると自動検査が難しくなる。また、事前・事後条件を作成する手間の問題も発生する。本稿では、検出可能な不具合の種類を限定するが、誤り箇所の同定精度を高める工夫を考えたい。

3 関数ポインタの DbC

3.1 関数ポインタの問題

C 言語によるプログラミングの特徴のひとつはポインタを多用することである。本節では、関数ポインタの取り扱い方法を考察する。C 言語を対象とする DbC 自動検査ツールは、データに対するポインタは近似的に扱うが、関数ポインタを対象外とすることが多い [5][13]。

図 3 に関数ポインタ comp を引数としてとる関数 delegate の例を示した。ポインタ値がメモリ内を参照していることを示す組み込み述語 valid_ptr を用いて事前条件を書く。しかし、図 4 の場合、user が delegate 呼び出し時に与えた 2 つの実引数が整合していないにも関わらず、この呼出し箇所の検査は正常に終了する。不具合が判明するのは、delegate 中での (*comp)(0) の呼出し時点

```
/* for comp
  resuire  (a >= 0);
  ensures  (\result > 0);
*/
int delegate(comp, n)
int (* comp)(int a); int n; { ... }
```

図 5. 関数ポインタの **DbC (2)**

であり、不具合の検出箇所がずれる。

この問題を解決するためには、図 5 のように、仮引数 comp の事前・事後条件を delegate に明示すればよい。delegate(calc,-1); が不具合であることを検知できる。一方、これを修正して、delegate(calc,0); とすると、この検査は通るが、実際の calc の事前条件に違反する。図 5 の方法だけでは不十分であることがわかる。

ここでは、関数ポインタを引数とする場合について説明したが、同様に返却値になる場合も要注意である。このような問題は、関数をファーストクラスのデータとして取り扱う関数プログラミング言語の DbC では必須である。実際、Scheme に対して DbC を導入する研究がある [7]。しかし、この DbC 検査は Eiffel と同様にプログラム実行時に行う [12]。本稿で取り扱う方法は静的な検査である点が異なる。

3.2 置き換え可能性

図 5 に示した事前・事後条件を明示する方法だけで扱えない実引数の検査のために、振舞いサブタイピング [11] を用いる方法を提案する。

振舞いサブタイピングの考え方は、オブジェクト指向言語におけるクラス継承の意味を表現するために導入された。メソッド M をサブクラスでオーバーライドする場合、サブクラスのメソッド M_{sub} はスーパークラスの同名メソッド M_{super} に対して、事前条件を弱め (weaker precondition)、事後条件を強める (stronger postcondition) とする。

$$PreM_{super} \Rightarrow PreM_{sub}$$
$$PostM_{super} \Leftarrow PostM_{sub}$$

本稿の提案は、振舞いサブタイピングを、関数ポインタの仮引数 ($comp_f$) と実引数 ($comp_a$) の間で用いるというものである。

$$PreComp_f \Rightarrow PreComp_a$$
$$PostComp_f \Leftarrow PostComp_a$$

```
/*
  requires (n > 0);
  ensures  (\result > 0);
*/
int foo(n) int n;
{
  int (*comp)(int a);
  /*
    for comp
      requires (a > 0);
      ensures (\result > 0);
  */

  int x;
  void *h = dlopen('x.so', RTLD_LAZY);
  comp = (int (*) int) dlsym(h,'f1');
  x = (*comp)(n);
  dlclose(h);
  return (x*x + 1);
}
```

図 6. 動的リンク関数の **DbC**

先の図 4 と図 5 の事前条件に適用すると以下のようになって成り立つべき関係が満たされない。したがって、関数 calc を実引数として渡せないことがわかる。

```
¬ ((n >= 0) ⇒ (n > 0))
```

なお、本稿で用いている F-Soft [10] は、静的フロー解析を行うことで calc が実引数として渡されることを求めることができる。

4 関連する話題

4.1 動的リンク・ライブラリ

関数内の局所的な変数に対する条件明示が有用なこともある。図 6 は動的リンクによって読み込んだライブラリ関数を用いるプログラム例である。関数 dlsym が返したアドレスを変数 comp に代入し、その後、関数呼出しを行う。関数 foo の本体で、呼出しの形式 (*comp)(n) を検査しようとすると、変数 comp が参照するであろう関数の事前・事後条件を予め明示的に表現しておく必要がある。局所変数宣言の次行に追加した事前・事後条件を用いて検査すれば良い。

4.2 外部仕様と内部実現

図 7 の例は、関数 sub の事前・事後条件をもとに、関数 caller を作成したものである。呼び出

```
/*
  requires (n >= 0);
  ensures  (\result > 0);
*/
int sub(n) int n; { ... }

int caller(n) int n;
{
  if (n >= 0) { return sub(n); }
  else { return sub(-n); }
}
```

図 **7. DbC** に基づく利用

```
/*
  requires (n >= 0);
  ensures  (\result > 0);
and further
  requires (n < 0);
  ensures  (\result < 0);
*/
int sub(n) int n;
{
  if (n >= 0) { return 1; }
  else { return -1; }
}
```

図 **8.** ユニットテスト風の **DbC**

し時に実引数の値を確認して使うことによって事前条件を満たすことを保証する。DbC が宣言的な設計の外部仕様を表す場合、事前条件を満たさない状態で当該関数が使用されると実行結果の事後条件は保証されない。第 2 節の式 (1) による検査という観点から図 7 は事前条件の正しい使い方である。

一方、DbC をプログラム実現に関わる内部仕様と考える場合、図 8 のような事前・事後条件になる。入力引数 n に対して、設計時の事前条件に対応する場合 (n >= 0) とこれを満たさない場合の両方をプログラムとして作成する。この事前条件を満たさない場合を明示したプログラム・コードについても第 2 節の式 (2) にしたがって検査を行う場合、分岐 (n < 0) についての事前・事後条件を明示しておく必要がある。「外部との約束事」という DbC 本来の目的からは外れるが、事前・事後条件を書き表す際に気になる問題のひとつである。

本稿では、この問題も置き換え可能性として取り扱う。すなわち、図 7 相当を外部設計仕様 (External)、図 8 相当を内部実現仕様 (Internal) と呼び、次の関係が成り立つとする。

$$Pre_{extern} \Rightarrow Pre_{intern}$$
$$Post_{extern} \Leftarrow Post_{intern}$$

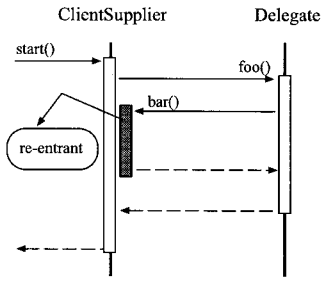

図 **9. Re-entrant** 性と不変条件

図 8 の事前・事後条件は次のような形として解釈する。内部実現仕様は外部設計仕様を置き換える事前・事後条件である。

$$Pre_{intern} = (\texttt{n >= 0}) \vee (\texttt{n < 0}) = true$$
$$Post_{intern} = ((\texttt{n >= 0}) \Rightarrow (\texttt{\result > 0})) \wedge ((\texttt{n < 0}) \Rightarrow (\texttt{\result < 0}))$$

図 8 のように外部設計仕様と内部実現仕様を明示的に区別して表現することによって、不具合箇所の同定精度を向上させることができる。両方の分岐を検査するという目的で事前条件を `true` とすると、図 7 での検査が意味をなさないのである。

4.3 再入性の問題

DbC に基づくモジュラー検証は設計情報の利用ならびにスケーラビリティの観点から良いアプローチである。しかし、DbC が本質的に 1 回の関数呼出しにおける関係を表現しているため、大域的な情報が欠如するという問題点がある。第 2 節の (2') に示したように不変条件が関係する「再入性の問題 (Re-entrancy)」である [4]。オブジェクト指向プログラムを対象とした問題点として議論されている。C 言語の場合でも同様のことは起こり得る。

図 9 に問題点を説明する状況を示した。第 2 節の (2') からわかるように、事前・事後条件の評価箇所は関数の呼出しと戻りの実行時点である事を思い出して欲しい。図 9 は関数 `start` から関数 `foo` を実行し、さらに関数 `foo` が `bar` を起動していることを示す。ここで、関数 `start` と `bar` とが同じ大域変数を参照 (`modifies`) する場合、関数 `bar` の起動時点では、当該変数が関わる不変条件が成り立つ保証がない。関数 `start` は不変条件が壊れて

いる実行状態で関数 foo を起動、さらには、関数 bar を起動している可能性を排除できない。現実に不変条件が破れている場合であっても、関数 bar は不変条件が成り立つとの仮定下で検査されることになる。

ところが、多くの場合、上記で心配したような不変条件の破れが起こらないことが経験的にわかっている。しかし、検査を安全に行うためには、本当に不変条件が破れていないことを確認しなければならない。文献 [4] は Java 系のオブジェクト指向プログラミング言語を対象として、メソッドの大域的な呼出し関係を追跡する方法を提案している。

本稿の自動検査では有界モデル検査法に基づく振舞い解析の機能を前提としている。特に、ベースとしている F-Soft [10] はプログラムの静的解析に基づく抽象化あるいは近似法を採用している。関数ポインタを含めて呼出し関係のグラフを作成し、振舞い検査のための情報を抽出する。したがって、再入性の問題で必須となる呼出し関係の追跡は既に行っていると言ってよい。すなわち、DbC に基づく局所的な解析と F-Soft が持つ振舞い解析の機能を適宜、組み合わせることで、再入性に起因する解析の不確かさを軽減することが可能と考えられる。今後、実際の適用事例での検討が必要である。

5 おわりに

本稿では、C 言語プログラムの DbC に関連して、特に、関数ポインタの取り扱い法について考察した。C 言語の自動検査を考える場合、関数ポインタを取り扱うことの重要性は十分に知られているが、実際に、これを扱うツールはほんとんどない。興味ある例が振舞いサブタイピングの考え方 [11] で説明できることを述べた。

なお、本稿の内容は、有界モデル検査法による C 言語プログラムの自動検査ツール [10] の実用版 VARVEL[9] に DbC を導入するための基本検討として行ったものである。本稿で述べなかった既存の機能やツールの実現方式については文献 [9][10] を参照して欲しい。実際に VARVEL に取り入れる際には、さらに実用上の観点からの検討が必要であると考えている。

参考文献

[1] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Softw. Pract. Exper.*, pp.583–599, 2005.

[2] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press 1999.

[3] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proc. TACAS'04*, pages 168–176, 2004.

[4] M. Fahndrich, D. Garbervetsky, and W. Schulte. A Re-Entrancy Analysis for Object Orietend Programs. *9th FTfJP* at ECOOP 2007.

[5] J.-C. Filliatre and C. Marche. Multi-prover Verification of C Programs. In *Proc. ICFEM'04*, pages 15–29, 2004.

[6] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended Static Checking for Java. In *Proc. PLDI'02*, page 234–245, June 2002.

[7] R. Funder and M. Felleisen. Contracts for Higher-Order Functions. In *Proc. ICFP'02*, pages 48–59, 2002.

[8] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular Checking for Buffer Overflows in the Large.

[9] 橋本祐介, 徳岡宏樹, 宮崎義昭. 形式手法による C 言語検証ツール「VARVEL」. NEC 技法, Vol. 60, No.2, pages 47–49, 2007.

[10] F. Ivancic, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model Checking C Programs Using F-Soft. In *Proc. ICCD'05*, 2005.

[11] B. Liskov and J. Wing. A Behavioral Notion of Subtyping. In *ACM TOPLAS*, 16(6), pages 1811–1841, 1994.

[12] B. Meyer. Applying "Design by Contract". *IEEE Computer*, Vol.25, No.10, pages 40–51, 1992.

[13] 南出 靖彦. C プログラムの検証ツール Caduceus. コンピュータ・ソフトウェア, 2007.

[14] 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, pages 27–48, 2007. (http://research.nii.ac.jp/TechReports/07-007J-j.html から入手可能).

[15] M.R. Prasad, A. Biere, and A. Gupta. A Survey of Recent Advances in SAT-Based Formal Verification.

[16] Robby, E. Rodriguez, M. Dwyer, and J. Hatcliff. Checking Strong Specifications Using An Extensible Software Model Checking Framework. In *Proc. TACAS'04*, pages 404–420, 2004.