Continuation based C を用いたプログラムの検証手法

比嘉 健太[1,a)]　河野 真治[2,b)]

概要：ソフトウェアが期待される仕様を満たしているか調べることは重要である。特に実際に動作しているソフトウェア自体を検証できると良い。従来は assert などを用いて検証しているが、モデル検査のような網羅的な検査を行うことはできない。そこでソフトウェアの実行自体を網羅的に行うように変更する。これは計算自体を変更するのでメタ計算となる。本論文ではプログラムを Continuation based C (CbC) で記述する。CbC では Code Segment という単位で処理を記述する。CbC を用いることによりメタ計算は CbC の Code Segment 間に Meta Code Segment を挟むという単純な方法で実現できる。そして、通常計算との切り替えも簡単に行うことができる。ここでは、赤黒木を実際の例題として検証した。比較対象として ANSI-C の記号実行を行う CBMC でも検証する。CBMC よりも広い範囲の検査が行えることを確認した。

キーワード：プログラミング言語, 検証, 赤黒木

Verification method of programs using Continuation based C

Yasutaka HIGA[1,a)]　Shinji KONO[2,b)]

Abstract: Checking desirable specifications of software is important. If it checks actual implementations, much better. Currently, assertions in the implementations is used, but it does not verify all possible executions like a model checker. We propose a modification of program executions to do the verification. This kind of modification is called a meta computation. In this paper, we describe programs in Continuation based C(CbC). CbC programs is composed of Code Segments. Using CbC, meta computations is easily implemented as an insertion of meta code segment between normal level code segments. Red-black Tree is verified by with our method. We also check it with CBMC which execute ANSI-C programs symbolically. Our method covers wider range of execution of the program.

Keywords: Programming Language, Verification, Red-Black Tree

1. ソフトウェアの検証

ソフトウェアの規模が大きくなるにつれ、バグは発生しやすくなる。バグとはソフトウェアが期待される動作とは異なる動作を行なうことである。ソフトウェアの期待される動作は仕様と呼ばれ、自然言語や論理で記述される。検証とは、ソフトウェアが定められた環境下において仕様を満たすことを保証するものである。ソフトウェアの検証手法には、定理証明とモデル検査がある。定理証明はソフトウェアが満たすべき仕様を論理式で記述し、その式が常に真であることを証明する。定理証明を行なう言語には依存型で証明を行なう Agda[7] や Coq[10] などが存在する。モデル検査はソフトウェアの全ての状態において仕様が満たされるかを確認するものである。モデル検査器には、Promela と呼ばれる言語でモデルを記述する Spin[9] やモデルを状態遷移系で記述する NuSMV[8] 、ANSI-C を記号実行することができる CBMC[6] などが存在する。証明やモデル検査器で検証を行なう際、実際に動作するコードでなく検証用にコードを書き直す必要があるなど、実際の実装との乖離が存在する。よって、書き直した記述では仕様

1　琉球大学 大学院 理工学研究科 情報工学専攻
Infomation Engineering Course Graduate School of Engineering and Science University of the Ryukyus
2　琉球大学
University of the Ryukyus
a)　atton@cr.ie.u-ryukyu.ac.jp
b)　kono@ie.u-ryukyu.ac.jp

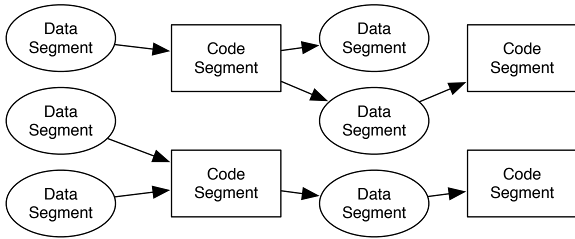


図 1: **Code Segment** どうしの組み合わせ

が満たされていても、実装にバグが入り込み信頼性を保証できない可能性がある。本研究は実際に動作するプログラムの信頼性を保証することを目的とする。

2. Code Segment と Data Segment

動作するコードを検証しやすい単位で記述するために本研究室では Code Segment と Data Segment を用いるプログラミングスタイルを提案している。Code Segment とは処理の単位であり、ループを含まないような単純な処理のみを行なう。プログラムは Code Segment どうしを組み合わせることで構築される (図 1)。Code Segment 間における値の受け渡しには、Data Segment というデータの単位で行なう。なお、接続された Code Segment には依存関係が発生するが、依存関係が無い Code Segment は並列に実行することが可能である。

ここで、Code Segment どうしの接続処理について考える。処理を表す Code Segment どうしの接続も処理である。このような、計算を実現するための計算をメタ計算と呼ぶ。メタ計算は Code Segment で記述することが可能であり、メタ計算を記述した Code Segment を Meta Code Segment と呼ぶ。Meta Code Segment は Code Segment 間に存在する上位の処理と考えることもできる (図 2)。

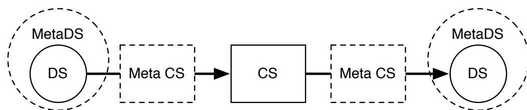


図 2: **Meta Code Segment** と **Meta Data Segment**

また、メタ計算に必要なデータは Meta Data Segment に格納し、通常の処理に必要な Data Segment も内包する。プログラムの性質を検証する機能をメタ計算として提供することで、ユーザが書いた Code Segment を変更することなく、メタ計算を追加するだけでプログラムの信頼性を上げる。

3. Continuation based C

Code Segment と Data Segment を用いたプログラミングスタイルで記述する言語として Continuation based C[3] を作成した。Continuation based C (以下 CbC) は OS や組込みシステムなどの記述を行なうことを目標に作られた、アセンブラと C 言語の中間のような言語である。CbC における Code Segment は C 言語における関数に、関数呼び出しが末尾のみである制約を加えたものである。Code Segment どうしの接続は goto による軽量継続で表される (図 3)。軽量継続とは呼び出し元の環境を持たずに次の処理へと移動することであり、呼び出し元のスタックフレームを破棄しながら関数呼び出しを行なうようなものである。なお、C 言語の資産を利用するために関数呼び出しを行なうことも可能である。また、Code Segment のみが導入された CbC で検証を行った例として Dining Philosophers Problem のデッドロック検知 [4] がある。

```
__code addTen(int a) {
    int b = a+10;
    goto twice(b);
}

__code twice(int x) {
    int y = 2*x;
    goto showValue(y);
}
```

図 3: **Code Segment** の接続 **(10** 加算して **2** 倍する**)**

CbC における Code Segment は、C 言語の関数宣言の返り値の型の部分に `__code` を記述して定義する。Code Segment 内部には変数の宣言や if 文による分岐といった C 言語の文法を用いて処理を記述する。図 3 の例では、`addTen` と `twice` という 2 つの Code Segment を定義している。`addTen` では int の値を受けとり、10 加算して `twice` を実行する。`twice` では受けとった int の値を 2 倍して `showValue` を実行する。

また、CbC における Data Segment は C 言語における構造体と共用体を用いたデータ構造である。各 Code Segment で必要な値を構造体で定義し、それらの共用体として Data Segment を定義する (図 4)。

```
union Data {
    struct Count {
        int x;
    } count;
    struct Port {
        int port;
    } port;
};
```

図 4: **Data Segment** の例

図 4 では Data Segment として int を持つ count と unsigned int を持つ port の 2 つを定義している。

Code Segment 内部ではポインタ演算は行なわず、メタ計算でポインタへの演算を行なう。これにはメモリ管理をメタ計算部分に分離することで、プログラムを検証しやすくするねらいがある。Code Segment がどの Data Segment にアクセスするかといった指定も Meta Code Segment で行なう (図 5 における `twice_stub`)。CbC における Meta Code Segment は Code Segment と Code Segment 間に存在する Code Segment である。

```
// Code Segment
__code addTen(union Data* ds, int a) {
    int b = a+10;
    goto twice_stub(ds);
}

// Meta Code Segment
__code twice_stub(union Data* ds) {
    goto twice(ds->count.x);
}

// Code Segment
__code twice(int x) {
    int y = 2*x;
    goto showValue(y);
}
```

図 5: Data Segment を指定する Meta Code Segment

CbC における Meta Data Segment は Data Segment を内包する構造体として定義できる (図 6 の Context)。また、goto する際に必ず通る Meta Code Segment を定義することで、 Code Segment どうしの接続もメタ計算として定義できる (図 6 の `meta`)。`meta` を切り替えることで Code Segment を変更することなくメタ計算のみを変更することができる。

メタ計算の例として、メモリ管理の他にも例外処理や並列実行、ネットワークアクセスなどがある。通常の計算を保存するようメタ計算を定義することで、例外処理などを含む計算に拡張することができる [5]。

```
// Meta Data Segment
struct Context {
    union Data *data; // Data Segment
    unsigned int gotoCount; // メタ計算に必要なデータ
    unsigned int stepOfAddTen;
};

// Meta Code Segment
__code meta(struct Context* context,
            enum Code next) {
    countext.gotoCount++;

    /* 接続時に行なうメタ計算を記述*/
    switch (next) {
        case AddTen:
            // 特定の Code へのメタ計算 Segment
            context.stepOfAddTen++;
            goto addTen_stub(context);
        case Twice:
            goto twice_stub(context);
    }
}

// Code Segment
__code addTen(struct Context* context, int a) {
    x = x+10;
    goto meta(context, Twice);
}

// Code Segment
__code twice(struct Context* context, int x) {
    x = x*2;
    goto meta(context, ShowValue);
}
```

図 6: メタ計算を用いて Code Segment を接続する例

4. CbC で記述した非破壊赤黒木

CbC で記述されたデータ構造に赤黒木がある。赤黒木とは木構造のデータ構造であり、各ノードに赤と黒の色を割り当て、その色を用いて木の高さのバランスを取る。赤黒木は通常の二分探索木としての条件の他に以下の条件を持つ。

- 各ノードは赤または黒の色を持つ。
- ルートの色は黒である。
- 赤ノードは 2 つの黒ノードを子として持つ (よって赤ノードが続くことはない)。
- ルートから最下位ノードへの経路に含まれる黒ノードの数はどの最下位ノードでも一定である。

これらの条件により、木をルートから辿った際に最も長い経路は最も短い経路の高々 2 倍に収まる。

赤黒木の実装として、当研究室で CbC を用いて開発している Gears OS [1] における非破壊赤黒木 [2] がある。非破壊赤黒木とは木への挿入や削除を行なった際、一度構築した木構造を破壊せず新しく木構造を生成する赤黒木である。非破壊赤黒木の実装の基本的な戦略は、変更したいノードへのルートノードからの経路を全て複製し、上に存在しないノードは変更前元の木構造と共有することである (図 7)。

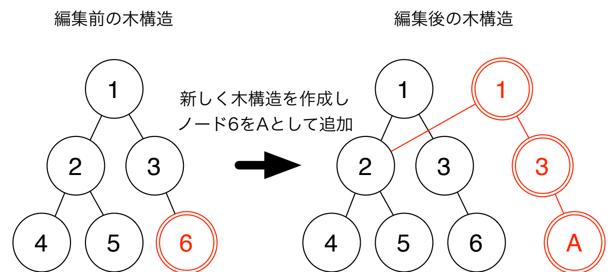


図 7: 非破壊赤黒木の編集

関数呼び出しが可能な言語では再起呼び出しで木を辿ることか可能だが、CbC は `goto` による軽量継続のみで記述

する必要があるため、経路を辿るにはノードに親への参照を持たせるか挿入・削除時に辿った経路を記憶するしかない。ノードが親への参照を持つ非破壊木構造は構築出来ないため、辿った経路を記憶する方法を用いる。辿った経路を記憶するため Meta Data Segment にスタックを持たせる (図 8 における Context 内部の `node_stack`)。赤黒木で利用する Data Segment を図 8 に表す。Data Segment は各ノードの情報を持つ `Node` と赤黒木を格納する `Tree` と、挿入などの操作中の一時的な木を持つ `Traverse` の共用体で表される。

```
// Data Segment for Red-Black Tree
union Data {
    struct Tree {
        struct Node* root;
    } tree;
    struct Traverse {
        struct Node* current;
        int result;
    } traverse;
    struct Node {
        int key;
        union Data* value;
        struct Node* left;
        struct Node* right;
        enum Color {
            Red,
            Black,
        } color;
    } node;
};

// Meta Data Segment
struct Context {
    stack_ptr node_stack;
    union Data **data;
};
```

図 8: 赤黒木の **Data Segment**

CbC による赤黒木実装の例として、赤が続かないという制約をチェックする Code Segment を図 9 に示す。まず、赤が連続するか判定するために親の情報が必須なため、経路を記憶しているスタックから親の情報を取得する。親の色が黒である場合は木が平衡であるために処理を終了し、ユーザ側の Code Segment へと `goto` する。親の色が赤である場合は次の条件判定へと `goto` するが、経路情報を再現するためにスタックへと親を再度挿入しつつ `goto` する。

ここで、赤黒木に対する処理を Code Segment とした場合、赤黒木のメモリ管理などは Meta Code Segment である。赤黒木を利用する Code Segment からは赤黒木の処理は意識する必要がないため、赤黒木の処理は Meta Code Segment のように見え、赤黒木のメモリ管理は Meta Meta Code Segment と考えられる。このようにメタ計算は階層構造を持つため、任意の Code Segment に対してメタ計算を適用することが可能である。

```
// Meta Code Segment
__code insertCase2(struct Context* context, struct Node
      * current) {
    struct Node* parent;
    stack_pop(context->node_stack, &parent);
    if (parent->color == Black) {
        stack_pop(context->code_stack, &context->next
            );
        goto meta(context, context->next);
    }
    stack_push(context->node_stack, &parent);
    goto meta(context, InsertCase3);
}

// Meta Meta Code Segment
__code insert2_stub(struct Context* context) {
    goto insertCase2(context, context->data[Traverse
          ]->traverse.current);
}
```

図 9: 赤黒木で赤のノードが続かないという制約の判定

5. メタ計算を用いた赤黒木の検証

CbC で記述された赤黒木と、それに対する処理の性質を実際に検証していく。赤黒木に求められる性質には以下のようなものがある

- 挿入したデータを参照できること
- 削除したデータは参照できないこと
- 値を更新した後は更新された値が参照されること
- 操作を行なった後の木はバランスしていること

本論文では操作を挿入に限定し、任意の順で木に要素を挿入しても木がバランスすることを検証する。検証には当研究室で開発している検証用メタ計算ライブラリ akasha を用いる。akasha では仕様を Meta Code Segment に記述するため、CbC で常に真となる式として定義する赤黒木の性質である、 木をルートから辿った際に最も長い経路は最も短い経路の高々 2 倍に収まることを CbC のコードで記述する (図 10)。

```
if (akashaInfo.maxHeight >
    2*akashaInfo.minHeight) {
   goto meta(context, ShowTrace);
}
```

図 10: 木の高さの仕様記述

図 10 で定義した仕様が常に成り立つか、全ての挿入順の組み合わせを列挙しながら確認していく。無限回の挿入順番を数え上げることは状態の抽象化無しには不可能なので、限られた要素数の挿入を検証する。検証の対象である赤黒木を内包する Meta Data Segment を定義し、挿入順の列挙に必要なデータを追加で定義する (図 11)。検証に必要なデータには挿入順の数え上げに使う環状リスト `Iterator` とその要素 `IterElem`、検証に使う情報を保持する `AkashaInfo`、木をなぞる際に使う `AkashaNode` がある。

```
// Data Segment
union Data {
    struct Tree { /* ... */ } tree;
    struct Node { /* ... */ } node;

    /* for verification */
    struct IterElem {
        unsigned int val;
        struct IterElem* next;
    } iterElem;
    struct Iterator {
        struct Tree* tree;
        struct Iterator* previousDepth;
        struct IterElem* head;
        struct IterElem* last;
        unsigned int iteratedValue;
        unsigned long iteratedPointDataNum;
        void* iteratedPointHeap;
    } iterator;
    struct AkashaInfo {
        unsigned int minHeight;
        unsigned int maxHeight;
        struct AkashaNode* akashaNode;
    } akashaInfo;
    struct AkashaNode {
        unsigned int height;
        struct Node* node;
        struct AkashaNode* nextAkashaNode;
    } akashaNode;
};
```

図 11: akasha の Data Segment

挿入順の数え上げには深さ優先探索を用いるが、CbC スタックフレームが無いために再帰では記述できず、データ構造に今の状態を保持させながら探索する。検証する要素を全て持つ環状リストを作成し、木に一度挿入する度に挿入した要素を排除した環状リストを複製していく。環状リストに全ての要素が無くなった時、1 つの組み合わせを列挙したことになる。1 つの挿入順を列挙後、前の深さの環状リストを再現し、環状リストの先頭を進めることで組み合わせの列挙を続ける。

挿入順を列挙しつつ、木に要素を挿入する度に図 10 の仕様が成り立つか Code Segment を実行して確認する。仕様には木の高さが含まれるため、その値を取得する Meta Code Segment が必要である (図 12)。

木をなぞるためのスタックに相当する `AkashaNode` を用いてノードを保持しつつ、高さを確認している。スタックが空であれば全てのノードをなぞったので、次の処理へと `goto` する。空でなければ今なぞっているノードが葉であるか確認する。葉ならば高さを更新し、スタックからノードを 1 つ破棄して自身に `goto` する。葉でなければ高さを 1 つ増やしながらスタックに左右の子供を積み、自身に `goto` する。

このように、Code Segment の検証に必要な仕様や処理は Meta Code Segment で記述することができる。akasha におけるメタ計算を用いて、要素数 13 個まで任意の順で挿入を行なっても仕様が満たされることを検証した。また、

```
__code getMinHeight(struct Context* context,
                    struct AkashaNode* left,
                    struct AkashaNode* right,
                    struct AkashaInfo* ai) {
    const struct AkashaNode* an = ai->akashaNode;

    if (an == NULL) {
        ai->akashaNode->height = 1;
        ai->akashaNode->node = context->data[Tree
            ]->tree.root;

        goto getMaxHeight_stub(context);
    }

    const struct Node* n = ai->akashaNode->node;
    if (n->left == NULL && n->right == NULL) {
        if (ai->minHeight > an->height) {
            ai->minHeight = an->height;
            ai->akashaNode = an->nextAkashaNode;
            goto getMinHeight_stub(context);
        }
    }

    ai->akashaNode = ai->akashaNode->
        nextAkashaNode;

    if (n->left != NULL) {
        left->height = an->height+1;
        left->node = node->left;
        left->nextAkashaNode = ai->akashaNode;
        ai->akashaNode = left;
    }

    if (n->right != NULL) {
        right->height = an->height+1;
        right->node = node->right;
        right->nextAkashaNode = ai->akashaNode;
        ai->akashaNode = right;
    }

    goto getMinHeight_stub(context);
}
```

図 12: 木の最も低い高さを取得する Code Segment

赤黒木の処理内部にバグを追加した際には akasha は反例を返した。

6. CBMC を用いた赤黒木の検証

同様に赤黒木の仕様を C 言語の有限モデルチェッカ CBMC[6] を用いて検証した。CBMC は ANSI-C を記号実行し、仕様の否定となるような実行パターンが無いかを検証するツールである。

比較のために全く同じ赤黒木のソースコードを用いたいが、CbC の構文は C とは異なるために変換が必要である。CbC は C と似た構文を持つため、`__code` を `void` に、`goto` を `return` に置換することで機械的に C 言語に変換できる。

CBMC における仕様は bool を返す式として記述するため、akasha と同様の仕様定義が利用できる (図 13。C では再帰呼び出しができるため、木の高さは再帰関数で確認している。assert が true になるような実行パターンを

CBMC が見付けると、その実行パターンが反例として出力される。

```
void verifySpecification(struct Context* context,
                         struct Tree* tree) {
    assert(!(maxHeight(tree->root, 1) >
           2*minHeight(tree->root, 1)));
    return meta(context, EnumerateInputs);
}
```

図 13: CBMC における仕様記述

挿入順の数え上げには CBMC の機能に存在する非決定的な値 `nondet_int()` を用いた (図 14)。検証する有限の要素数回分、ランダムな値を入力させることで挿入順の数え上げとする。

```
void enumerateInputs(struct Context* context,
                     struct Node* node) {
    if (context->loopCount >
        LIMIT_OF_VERIFICATION_SIZE) {
        return meta(context, Exit);
    }

    node->key = nondet_int();
    node->value = node->key;
    context->next = VerifySpecification;
    context->loopCount++;

    return meta(context, Put);
}
```

図 14: CBMC による挿入順の数え上げ

CBMC では有限のステップ数だけ C 言語を記号実行し、その範囲内で仕様が満たされるかを確認する。条件分岐や繰り返しなどは展開されて実行される。赤黒木検証時に CBMC が扱えた上限である 411 回まで展開を行なっても仕様は満たされていた。しかし、赤黒木にバグを追加した際にも仕様の反例を得られず、411 回の展開では赤黒木を正しく検証できないことも分かった。

よって、CBMC では検証できない範囲の検証を akasha で行なえることが確認できた。

7. 考察と今後の課題

CbC で記述したプログラムに対する仕様の検証を行なうことができた。Code Segment と Data Segment を用いてプログラミングすることで、検証用にコードを変更することなく検証を行なうことができた。

今回の検証では赤黒木が任意の順で有限の回数挿入されても木がバランスしていることを確認した。挿入順は深さ優先探索で総あたりで探索しているため、赤黒木の操作に削除が含まれると上手く数え上げられない。これは過去に探索した木の形状を記憶することで解決できる。また、挿入を行なった後の木の高さを調べるために毎回木をなぞるため、木の要素数に比例した計算時間がかかっている。各ノードに高さ情報を持たせ、木の操作時にノードの高さも更新することで高速に仕様をチェックすることができる。ノードの高さ情報そのものの処理が正しいかといった、メタ計算の仕様も検証したい。Meta Code Segment は任意の Code Segment に適用できるため、 akasha そのものの性質も akasha で検証できると考えている。加えて、今回の検証に用いたメタ計算は赤黒木へのデータ構造に依存している部分が多い。ポインタへの不正アクセス検出といった汎用的な検証機能を CBMC のように akasha にも追加していきたい。

参考文献

[1] 伊波立樹, 東恩納琢偉, 河野真治: Code Gear、 Data Gear に基づく OS のプロトタイプ, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2016)

[2] 小久保 翔平, 河野真治: Code Segment と Data Segment を持つ Gears OS の 設計 (2016)

[3] 徳森海斗, 河野真治: Continuation based C の LLVM/clang 3.5 上の実装について, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2014)

[4] 下地 篤樹, 河野 真治: 線形時相論理による Continuation based C プログラムの検証, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), (2007)

[5] Moggi, Eugenio: Notions of Computation and Monads, Inf. Comput (1991).

[6] Clarke, Edmund and Kroening, Daniel and Lerda, Flavio: A Tool for Checking ANSI-C Programs, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)

[7] `The Agda Wiki - Agda` 入手先 ⟨http://wiki.portal.chalmers.se/agda/pmwiki.php⟩ （参照 2016-07-05).

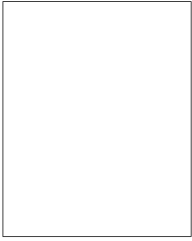
[8] `NuSMV home page` 入手先 ⟨http://nusmv.fbk.eu/⟩ （参照 2016-07-06）

[9] `Spin - Formal Verification` 入手先 ⟨http://spinroot.com/spin/whatispin.html⟩ （参照 2016-07-06）

[10] `Welcome! | The Coq Proof Assistant` 入手先 ⟨https://coq.inria.fr/⟩ （参照 2016-07-06）

比嘉健太

1992 年生。2015 年琉球大学工学部情報工学科卒業。2015 年琉球大学大学院理工学研究科情報工学専攻入学。



河野真治

1959 年生。1984 年 3 月東京工業大学理学部化学科卒業。1986 年 3 月東京大学大学院情報工学課程修了。1989 年 5 月工学博士。 1989-1996 年 Sony Computer Science Laboratory, Inc. 1996 年 琉球大学准教授。情報処理学会, ACM , 日本ソフトウェア科学会各会員。