線形時相論理による Continuation based C プログラムの検証

下 地 篤 樹[†] 河 野 真 治[††]

継続を持つ言語 Continuation based C(CbC) で記述されたプログラムの検証について考察する。
本稿では、プログラムの状態と共に線形時相論理式をタブロー展開することで検証を行った。

Verification of Continuation based C Program using Linear-time Temporal Logic

Atsuki SHIMOJI[†] and Shinji KONO[††]

Verification of a program is described by Continuation based C(CbC) which is a programing language with continuation is considered. In this paper, verified a CbC program by a method that states of a program expanded with Linear-time Temporal Logical formula.

1. はじめに

近年、計算機科学の進歩によりソフトウェアは大規模かつ複雑なものになっている。そのため、設計段階において誤りが生じる可能性が高くなってきており、設計されたシステムに誤りがないことを保証するための論理設計や検証手法およびデバッグ手法の確立が重要な課題となっている。検証において、時相論理はシステムの要求仕様を記述する方法として形式的検証で利用される。

本研究室では、Continuation based C(CbC) 言語を提案している。この言語は、C 言語より下位でアセンブラより上位のプログラミング言語である。そのため、C 言語よりも細かく、アセンブラよりも高度な記述が可能であるという利点がある。また、CbC で記述されたプログラムは状態遷移記述と近い構造になるという性質がある。

本論文では、CbC が状態遷移記述と相性の良い言語であることに着目し、状態遷移記述に対して有効である、タブロー法による状態の展開を行った。状態を展開する際に、線形時相論理も同時に展開することにより検証を行った。^

2. ソフトウェア検証

ソフトウェアが大規模かつ複雑になるにつれてバグは発生しやすくなる。バグとは、ソフトウェアが、期待された動きと別な動きをすることである。また、その「期待された動き」を規定したものが仕様と呼ばれ、自然言語または論理で記述される。検証とは、ソフトウェアが仕様を満たすことを数学的に厳密に確かめることである。

ソフトウェア検証には、大きく分類して、モデル検査と定理証明がある。モデル検査では、有限状態モデルを網羅的に探索して、デッドロックや飢餓状態などの望ましくない状態を自動的に検出することができる。しかし、無限の状態を持つものや、有限でも多くの状態を持つものは取り扱うことが困難である。一方、定理証明では、定理や推論規則を用いて検証を行うため、そのような状態を持つものでも取り扱うことが可能であるが、対話的な推論が必要になる。

3. CbC の概要

CbC は、C からループ制御構造と、サブルーチン・コールを取り除き、継続を導入した言語である[3)]。この言語は、C に継続専用のコード単位 (code) と、継続 (goto) を導入した構成となっている。CbC のプログラムは、code segment と呼ばれるプログラム単位を引数付き goto によって接続することで構成されている (図 1)。CbC の `code segment` と引数付き `goto`、`if` は、それぞれオートマトンの状態と状態遷移および遷移条件に対応しており、CbC は状態遷移記述に適していることがわかる。

CbC の concurrency は code segment 単位であるため、実装の方法により任意に細分化することができる。

† 琉球大学理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
†† 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

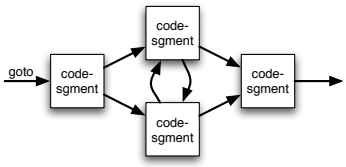

図 1 CbC プログラムの構成

4. SPIN の概要

有限状態遷移系に対する形式的検査手法としてモデル検査手法があり、その代表的なツールに SPIN[6] がある。

SPIN は、プログラム変換的な手法で検証するツールで、検証対象を PROMELA(PROcess MEta LAnguage) という言語で記述し、それを基に C 言語で記述された検証器を生成するものである。channel を使っての通信や並列動作する有限オートマトンのモデル検査が可能である。

SPIN は、PROMELA による記述を入力として網羅的に状態を探索し、その性質を検査する。また、SPIN は、PROMELA による記述をシミュレーション実行することもできる。SPIN によるモデル検査は、PROMELA による記述より、PAN(Protocol ANalyzer) という状態を網羅的に探索する実行形式を自動生成し、それにより様々な性質の検査を行う。

SPIN では以下の性質を検査することができる。

- アサーション
- 到達性
- 進行性
- LTL 式

SPIN はオートマトンの並列実行が可能であるが、これは厳密には実行する statement をランダムに選択し、実行している。

- PROMELA の記述例

  ";"で区切られた statement は並列に実行される。

```
byte state = 1;

proctype A()
{
  byte tmp;
  (state == 1) -> tmp=state; tmp=tmp+1; state=tmp
}

proctype B()
{
  byte tmp;
  (state == 1) -> tmp=state; tmp=tmp-1; state=tmp
}

init
{
  run A(); run B()
}
```

また、SPIN の concurrency は statement 単位となっている。この SPIN をモデルとして CbC プログラムの検証ツールを作成することを目標としている。

5. CbC プログラムの検証手順

プログラムにおいて非決定性な要素として入力と並列実行があげられる。プログラム自体は仕様が決まっており、決定性であるといえる。しかし、複数のプログラムを並列に実行する場合、その全体の動作は非決定性である (図 2)。

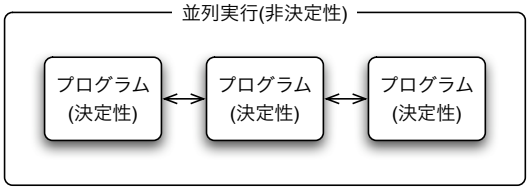

図 2 並列実行の非決定性

検証の手順として、並列実行するプログラムの可能な実行すべてをデータ構造として構築し、そのデータ構造に対して、仕様を検証する。

例えば、C、VHDL、JAVA で記述されたプログラムを並列実行する場合を考える。まず、それぞれのプログラムを CbC で書き換える。そして、それらの CbC プログラムを並列実行するために scheduler を用意する。それによってできた並列実行可能なプログラム全体は一つの CbC プログラムとみなすことができる。その並列実行を検証するために、プログラムの状態をデータ構造として構築する必要がある。それを行うために、CbC プログラムに対してタブロー展開を行う。そして、タブロー展開によってできたデータ構造に対して仕様を検証する (図 3)。

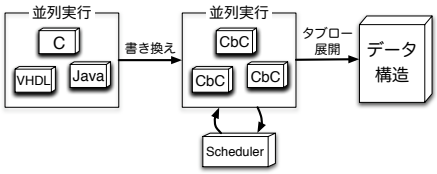

図 3 検証手順

6. 線形時相論理

時相論理は時間を様相として持つ、状態の遷移や時間の経過の観点よりシステムの性質を記述するための論理体系である。その派生である線形時相論理は、未来を記述することを可能にする時相論理であり、状態遷移システムの単一の経路に関する性質を記述するこ

とができる。

線形時相論理で表現できる重要な特性として、安全性特性と活性特性の２つがある。安全性特性とは、有限な期間での反例を無限の時系列に拡張しても反例であるような状態である。活性特性は、有限な期間での反例を無限の時系列に拡張したとき、それが反例でなくなる状態である。

線形時相論理の単項演算は以下の 3 つである。

- □ p
  Always：p は常に真である。
- ◇ p
  Sometime：p はいずれかの時点で真となる。
- ○ p
  Next：p は次の時点で真である。

□ p により安全性特性を表現し、◇ p により活性特性を表現することができる。

7. サンプルプログラム

検証用のサンプルプログラムとして Dining Philosophers Problem を用いる。これは資源共有問題の一つで、次のような内容である。

５人の哲学者が円卓についている。各々の哲学者にはスパゲッティーを盛った皿が出されている。スパゲッティーはとても絡まっているので、２本のフォークを使わないと食べられない。お皿の間に１本のフォークが置いてあるので、５人の哲学者に対して５本のフォークが用意されていることになる。哲学者は思索と食事を交互に繰り返している。空腹を覚えると、左右のフォークを手に取ろうと試み、首尾よく２本のフォークを手にできればしばし食事をし、しばらくするとフォークを置いて思索に戻る。隣の哲学者が食事中でフォークが手に取れない場合は、そのままフォークが空くのを待って飢えてしまう (図 4)。

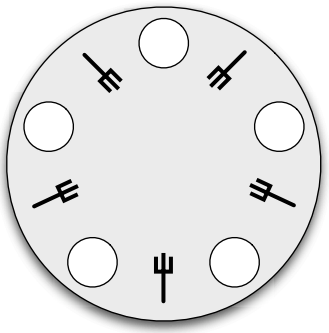

図 4 Dining Philosophers Problem のイメージ

各哲学者を一つのプロセスとみなし、そのプロセスを scheduler を用いて制御する。実際には、各 code segment の実行後、scheduler に遷移するようにしている。

これを CbC で実装するにあたり、次のような 8 個の code segment を定義した。

- 思索
- 両手にフォークを持っていない場合の飢餓
- 左手にフォークを持つ
- 左手にフォークを持っている場合の飢餓
- 右手にフォークを持つ
- 食事
- 右手のフォークを置く
- 左手のフォークを置く

各哲学者のフォークの有無と実行中の code segment、フォークの所在を状態として扱う。以下に code segment の例を示す。

左手でフォークを持つ code segment

```
code pickup_lfork(PhilsPtr self,
                  TaskPtr current_task)
{
    if (self->left_fork->owner == NULL) {
        printf("%d: pickup_lfork:%d\n",
                self->id, self->left_fork->id);
        self->left_fork->owner = self;
        self->next = pickup_rfork;
        goto scheduler(self, current_task);
    } else {
        self->next = hungry1;
        goto scheduler(self, current_task);
    }
}
```

この code segment は、左手側のフォークがあればそのフォークを取り、哲学者の id とフォークの id を表示する。そして、次の遷移先として右手にフォークを持つ (pickup_rfork)code segment をセットする。もし、フォークがなければ次の遷移先として両手にフォークを持っていない場合の飢餓 (hungry1)code segment をセットする。

8. CbC プログラムのタブロー展開

CbC プログラムのタブロー展開について説明する。タブロー展開の概要を以下に示す。

- プログラムの可能な実行の組合せ全てを調べることができる。
- 状態の探索は Depth First search で行う。
- プログラムの実行によって生成されるメモリパターンは Binary Tree で記録する。
- 同じメモリパターンは全て共有される。
- メモリパターンの集合である状態も Binary Tree で記録する。

8.1 状態の登録

状態として扱う変数は全て Binary Tree 構造で記録する。この Binary Tree をメモリパターンと呼ぶ。新たに追加されようとしている状態がメモリパターンに既に登録されている場合、その状態は登録されない。

- 状態を登録する関数
  状態の情報を取得し memory_range_lookup 関数により状態の登録を行う。実際に登録を行うのは memory_range_lookup 関数である。

```
MemoryPtr
add_memory_range(void *ptr,int length,
    MemoryPtr *parent)
{
    Memory m, *out;
    m.adr = ptr;
    m.length = length;
    m.left = m.right = 0;

    memory_range_lookup(&m, parent,&out);
    return out;
}
```

- memory_range_lookup 関数
  状態をメモリパターンから検索し、無ければメモリパターンに登録する。

```
int
memory_range_lookup(MemoryPtr m,
  MemoryPtr *parent, MemoryPtr *out)
{
  MemoryPtr db;
  int r;

  while(1) {
    db = *parent;
    if (!db) {
    /* not found */
      if (out) {
        db=create_memory(m->adr,m->length);
        *out = *parent = db;
      }
#if MEMORY_REPORT
      range_count++;
      range_size+=m->length;
#endif
      return 0;
    }
    if(!(r = cmp_range(m,db))) {
    /* bingo (actually an error) */
      if (out) {
        *out = db;
      }
      return 1;
    } else if (r>0) {
      parent = &db->left;
    } else if (r<0) {
      parent = &db->right;
    }
  }
  /* !NOT REACHED */
}
```

状態を全て登録した後、メモリパターンを状態データベースに登録する。状態データベースもメモリパターン同様 Binary Tree 構造で記録している。新たに追加されようとしているメモリパターンが既に登録されている場合は、既に登録されている方のメモリパターンをコピーし、状態データベースに追加する。

- メモリパターンを状態データベースに登録する関数
  メモリパターンを状態データベースからハッシュ値をもとに検索し、無ければ登録する。

```
int
lookup_StateDB(StateDB s, StateDB *parent,
  StateDB *out)
{
  StateDB db;
  int r;

  while(1) {
    db = *parent;
    if (!db) {
      /* not found */
      if (out) {
        db = create_stateDB();
        db->left = db->right = 0;
        db->memory = copy_memory(s->memory,&mem_db);
        db->hash = s->hash;
        state_count0 ++;
        *parent = db;
        *out = db;
      }
      return 0;
    }
    if (s->hash == db->hash) {
      r =  cmp_memory(s->memory,db->memory);
    } else
      r =  (s->hash > db->hash)? 1 : -1;
    if(!r) {
      /* bingo */
      if (out) *out = db;
      return 1;
    } else if (r>0) {
      parent = &db->left;
    } else if (r<0) {
      parent = &db->right;
    }
  }
}
```

8.2 状態の探索

状態の探索は Depth First Search で行う。最初のメモリパターンを登録後、code segment を実行する。その実行によって変化したメモリパターンを状態データベースからハッシュ値をもとに検索し登録を行う。同じメモリパターンが無かった場合、task iterator を作成し状態データベースを登録する。その後、1 段深く探索を行う。同じメモリパターンが状態データベースにあった場合、その経路の探索を終了し次の経路を探索する。探索している深さに探索可能な経路が無くなった場合、task iterator を 1 つ戻す。そこで探索可能な経路があればメモリパターンを restore し探索を行う。なければさらに 1 つ戻る。task iterator を戻していき root に到達した場合、全状態の探索が終了したことになる。

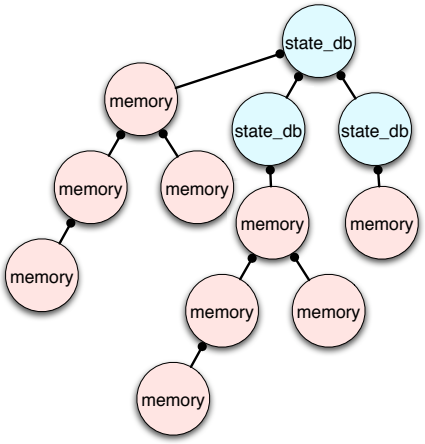

図 5 メモリパターンと状態データベース

以下に実際のソースコードを示す。

- タブロー展開を行う code segment

```
code tableau(TaskPtr list)
{
  StateDB out;

  st.hash = get_memory_hash(mem,0);
  if (lookup_StateDB(&st, &state_db, &out)) {
    // found in the state database
    printf("found %d\n",count);
    while(!(list = next_task_iterator(task_iter))){
      // no more branch,
      // go back to the previous one
      TaskIteratorPtr prev_iter = task_iter->prev;
      if (!prev_iter) {
        printf("All done count %d\n",count);
        memory_usage();
        goto ret(0),env;
      }
      printf("no more branch %d\n",count);
      depth--;
      free_task_iterator(task_iter);
      task_iter = prev_iter;
    }
    // return to previous state
    //  here we assume task list is fixed,
    //  we don't have to recover task list itself
    restore_memory(task_iter->state->memory);
    printf("restore list %x next %x\n",
      (int)list,(int)(list->next));
  } else {
    // one step further
    depth++;
    task_iter
      = create_task_iterator(list,out,task_iter);
  }
  printf("depth %d count %d\n", depth, count++);
  goto list->pkt->next(list->pkt, list);
}
```

`st.hash = get_memory_hash(mem,0);` でメモリパターンのハッシュ値を取得し、これをもとに状態データベースの検索を行っている。

`restore_memory` 関数によりメモリを restore する。これにより任意の時点でのメモリパターンを使用することができる。

`task iterator` は以下のような構造体である。

```
typedef struct task_iterator {
    struct task_iterator *prev;
    StateDB state;
    TaskPtr list;
    TaskPtr last;
} TaskIterator, *TaskIteratorPtr;
```

`task iterator` は状態データベースとタスクのリストを持っている。

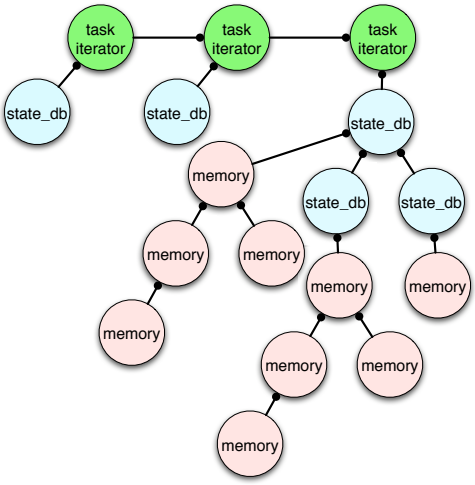

図 6 状態探索時のデータ構造

9. 線形時相論理による検証

タブロー展開を行う際に、LTL 式より生成された code segment を実行することで検証を行う。

Dining Philosophers Problem について検証を行う前に、まず、簡単な例題を用意しそれに対して検証を行った。

9.1 例　　題

2 つの code segment で構成されている。

- 1 つの変数の値をインクリメントとする code segment

```
code increment(PktPtr pkt, TaskPtr current_task)
{
    pkt->val++;
    printf("inc: %d\n", pkt->val);
    pkt->next = modulo;
    goto scheduler(pkt, current_task);
}
```

- 剰余を計算、代入する code segment

```
code modulo(PktPtr pkt, TaskPtr current_task)
{
```

```
    pkt->val %= 10;
    printf("mod: %d\n", pkt->val);
    pkt->next = increment;
    goto scheduler(pkt, current_task);
}
```

9.2 検証の実装

scheduler に遷移した後、tableau code segment に遷移する前に検証用 code segment を実行する。

- 検証用 code segment

```
code
check(int *flag,int *always_flag,PktPtr pkt,
    TaskPtr list)
{
    if (pkt->val <= 10) {
*flag = 1;
    } else {
*flag = 0;
*always_flag = 0;
    }

    goto tableau(list);
}
```

今回は p を (pkt->val <= 10) として□ p、すなわち pkt->val が常に 10 以下であることを判定した。

以下が実行の結果である。

```
All done count 19
        memory_header 107
        memcmp_count 81
        memory_body 176
        restore_count 0
        restore_size 0
        range_count 4
        range_size 24
[]p: valid.
```

状態数が全部で 19 個であった。□ p が成り立つことが判定できた。

10. ま と め

CbC プログラムの検証を行うため、並列動作させるサンプルプログラムとして Dining Philosophers Problem を作成した。その後、タブロー展開を行うプログラムを作成し適用することによりプログラムの全状態の展開を行った。

また、線形時相論理を適用させるために簡単な例題を新たに作成し、その例題に対して線形時相論理式を同時にタブロー展開することで検証を行った。

11. 今後の課題

Dining Philosphers Problem プログラムに対して線形時相論理による検証を行う。また、SPIN をモデルとして LTL 式から自動的に検証器を生成するツールの作成があげられる。

参 考 文 献

1) 下地 篤樹, 河野 真治. “タブロー法を用いた 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), May,2006.
2) 下地 篤樹, 河野 真治. “タブロー法を用いた Continuation based C プログラムの検証”. 日本ソフトウェア科学会第 23 回大会, 2006.
3) 河野 真治. “継続を持つ C の下位言語によるシステム記述”. 日本ソフトウェア科学会第 17 回大会, 2000.
4) 島袋 仁, 河野 真治. “C with Continuation と、その PlayStation への応用”. 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), May,2000.
5) 比嘉 薫, 河野 真治. “タブロー法の負荷分散について”. 日本ソフトウェア科学会第 18 回大会論文集, Sep, 2001.
6) http://spinroot.com/spin/whatispin.html Spin - Formal Verification