# タブロー法を用いたContinuation based C プログラムの検証

下地 篤樹 † 河野 真治 ††
Atsuki SHIMOJI Shinji KONO
† 琉球大学理工学研究科 †† 琉球大学工学部情報工学科
†Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
††Department of Information Engineering, University of the Ryukyus.

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、検証対象プログラムとして、Dining Philosophers Problem を採用し、それに対してタブロー法を用いた検証を行うことを目的する。

## 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。

本研究室では Continuation based C(CbC) 言語を提案している [1]。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。C に code segment と引数付き `goto` を導入した言語として、C with Continuation(以下 CwC) がある。CwC は C の上位言語である。CbC は、CwC の仕様の一部に制限されたものとみなすことができるので、CwC を CbC として使うことが可能である。

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、タブロー法による検証を行うことを目的とする。

## 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 を使ったプログラム検証

プログラムの検証としてモデル検査があり、その代表的なツールに SPIN[4] がある。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。

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。

作成する CbC プログラムの検証ツールは、検証対象の CbC プログラムを基に CbC で記述された検証器を生成することを目的とする。

以下の手順で研究を行う。

1. SPIN を調べる。
2. 検証対象プログラムとして Dining Philosophers Problem を採用し、それを CbC で記述する。
3. Dining Philosophers Problem の単体シミュレーションを行う。

4. Scheduler を作成し、それを組み込むことで全体のシミュレーションを行う。
5. タブロー展開器を作成し、Scheduler の代わりに組み込み、検証を行う。
6. 区間時相論理による検証を組み込む。

ここでは、タブロー法による検証までを行った。

### 3.1 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 1)。

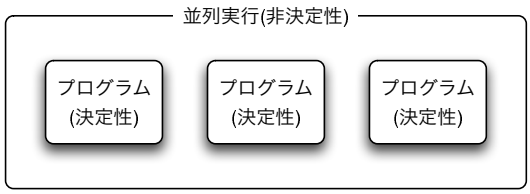


図 1: 並列実行の非決定性

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、プログラムのモデル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。

## 4 SPIN とは

SPIN は AT&T Bell 研が開発したオープンソースのモデル検査ツールである。チャネルを使って通信する、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ができる。

SPIN によるモデル検査は、pan(Protocol ANalyzer) という、状態を網羅的に探索してくれ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オートマトンをランダムに選択し、実行している。

## 5 オートマトン記述としての CbC

オートマトンとは、外からの入力に対して内部の状態に応じた処理を行い、その結果を出力するシステムである。オートマトンは、複数の状態で構成されており、それぞれの状態は入力に対してどうのような処理を行うかが定義されている (図 2)。

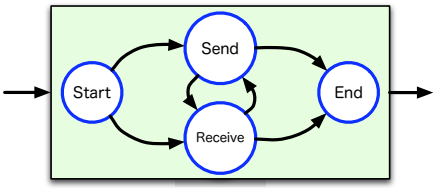


図 2: オートマトンの例

オートマトンの状態と状態遷移および遷移条件は、それぞれ、CbC の `code segment` と引数付き `goto`、`if` に対応しており (図 3)、CbC は、オートマトンを記述するのに適していると言える。

## 6 Dining Philosophers Problem

ここでは、検証用の例題プログラムとして 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

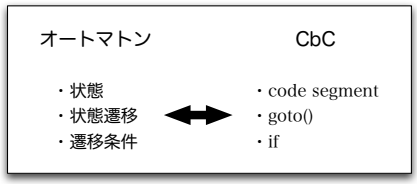


図 3: オートマトンと CbC の対応

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。

これを CbC で実装するにあたり、次のような code segment を提案した。

- 思索
- 両手にフォークを持っていない状態での飢餓
- 左手でフォークを持つ
- 左手にフォークを持った状態での飢餓
- 右手でフォークを持つ
- 食事
- 右手のフォークを置く
- 左手のフォークを置く

全部で 8 個の code segment により構成される。

## 7 CbC プログラムの作成

検証用の例題プログラムとして Dining Philosophers Problem を用いる。各哲学者を一つのプロセスとみなし、そのプロセスの実行順序を制御するために各 code segment は実行後、scheduler に遷移するようにしている。

### 7.1 構造体の作成

このプログラムで用いる構造体を示す。

```
typedef struct phils {
    int id;
    struct fork *right_fork;
    struct fork *left_fork;
    struct phils *right;
    struct phils *left;
    code (*next)(struct phils *,struct task *);
} Phils, *PhilsPtr;

typedef struct fork {
    int id;
    struct phils *owner;
} Fork, *ForkPtr;
```

これは哲学者とフォークの構造体である。哲学者構造体は id とフォークのポインタ、両隣の哲学者のポインタ、次に実行する code segment を持つ。フォーク構造体は id とオーナーである哲学者のポインタを持つ。

```
typedef struct task {
    struct task *next;
    struct phils *phils;
} Task, *TaskPtr;
```

これは実行するタスクの構造体である。内容は、次に実行するタスクへのポインタと哲学者の構造体である。この構造体から哲学者を取り出して実行する。

### 7.2 code segment の作成

各 code segment について説明する。

**・思索 (thinking) code segment**

```
code thinking(PhilsPtr self, TaskPtr current_task)
{
    printf("%d: thinking\n", self->id);
    self->next = hungry1;
    goto scheduler(self, current_task);
}
```

思索状態になった哲学者の id を表示し、次の遷移先である hungry1 をセットする。

**・両手にフォークを持っていない状態での飢餓 (hungry1) code segment**

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

飢餓状態になった哲学者の id を表示し、次の遷移先である pickup_lfork をセットする。

**・左手でフォークを持つ (pickup_lfork) 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);
    }
}
```

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 pickup_rfork 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 hungry1 をセットする。

**・右手でフォークを持つ (pickup_rfork) code segment**

```
code pickup_rfork(PhilsPtr self,
                  TaskPtr current_task)
{
    if (self->right_fork->owner == NULL) {
        printf("%d: pickup_rfork:%d\n",
                self->id, self->right_fork->id);
        self->right_fork->owner = self;
        self->next = eating;
        goto scheduler(self, current_task);
    } else {
        self->next = hungry2;
        goto scheduler(self, current_task);
    }
}
```

右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 pickup_rfork 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 hungry2 をセットする。

**・左手にフォークを持った状態での飢餓 (hungry2) code segment**

```
code hungry2(PhilsPtr self, TaskPtr current_task)
{
    printf("%d: hungry2\n", self->id);
    self->next = pickup_rfork;
    goto scheduler(self, current_task);
}
```

飢餓状態になった哲学者の id を表示し、次の遷移先である pickup_rfork をセットする。

**・食事 (eating) code segment**

```
code eating(PhilsPtr self, TaskPtr current_task)
{
    printf("%d: eating\n", self->id);
    self->next = putdown_rfork;
    goto scheduler(self, current_task);
}
```

食事状態になった哲学者の id を表示し、次の遷移先である putdown_rfork をセットする。

**・右手のフォークを置く (putdown_rfork) code segment**

```
code putdown_rfork(PhilsPtr self,
                   TaskPtr current_task)
{
    printf("%d: putdown_rfork:%d\n",
            self->id, self->right_fork->id);
    self->right_fork->owner = NULL;
    self->next = putdown_lfork;
    goto scheduler(self, current_task);
}
```

哲学者の id と右手のフォークの id を表示し、右手のフォークを置く。次の遷移先である putdown_lfork をセットする。

**・左手のフォークを置く (putdown_lfork) code segment**

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

哲学者の id と左手のフォークの id を表示し、左手のフォークを置く。次の遷移先である thinking をセットする。

### 7.3 scheduler の作成

並列実行をシミュレーションするために scheduler を作成した。scheduler プログラムについて説明する。
このプログラムは、まず、哲学者のリストとフォークのデータを生成・初期化し、それをタスクリストに登録する。登録が終了したら、scheduler code segment へ遷移する。

**• scheduler code segment** Dining Philosophers Problem の全 code segment はこの code segment に遷移する。

```
code scheduler(PhilsPtr phils, TaskPtr list)
{
    goto get_next_task(list);
}
```

get_next_task にはプログラム実行時のオプションにより、FIFO 実行かランダム実行どちらかの code segment が入る。これにより FIFO 実行とランダム実行を切替えることができる。

**・FIFO 実行 code segment**

```
code get_next_task_fifo(TaskPtr list)
{
    if (max_step--<0) {
        goto die("Simuration end.");
    }
    list = list->next;
    goto list->phils->next(list->phils,list);
}
```

タスクリストは環状になっている。この code segment はタスクを登録された順に実行する。max_step が 0 より小さくなると終了する。

**・ランダム実行 code segment**

```
code get_next_task_random(TaskPtr list)
{
    if (max_step--<0) {
        goto die("Simuration end.");
    }
    list = get_task(
        (random()%list_length(list)+1),list);
    goto list->phils->next(list->phils,list);
}
```

この code segment はタスクリストからタスクをランダムに選択し、実行するものである。list_length(TaskPtr list) 関数はタスクリストに登録されているタスクの個数を返す関数である。get_task(int num,TaskPtr list) 関数はタスクリストの num 番目のタスクを返す関数である。

## 8 CbC プログラムの検証

並列実行における状態変化をタブロー展開することで validity checking を行う。

## 9 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。このプログラムは 8 個の code segment からなり、それぞれが状態を表している。そして、この code segment を遷移しながら実行するプロセスが 5 個あり、その 5 個のプロセスを並列に実行させるために scheduler を作成した。

## 10 今後の課題

今後の課題として、本研究の目的であるタブロー法により検証の実装があげられる。タブロー法は、様相論理式の恒真性を検証する定理証明アルゴリズムで、木構造に基づく反駁手法である。

### 10.1 タブロー法の適用

例として 2 つの並列実行可能なプロセスを考える (図 4)。

これらのプロセスは、それぞれ $start(s_n)$ と $end(e_n)$、それとは別の状態を 2 つ持っている。そして、それぞれのプロセス内だけで有効ないくつかのグローバル変数 (g.Vn) を持っている。この 2 つのプロセスの並列実行における状態変化をタブロー展開する方法を述べる。

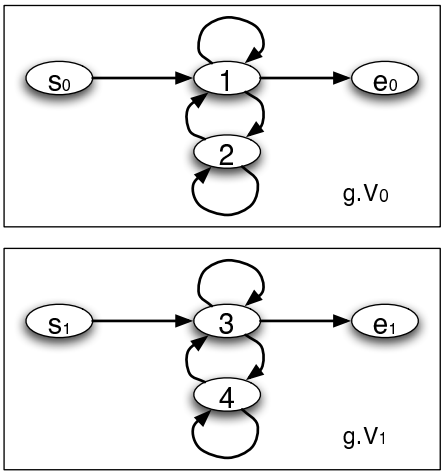


図 4: 並列実行可能プロセス

2 つのプロセスを並列に実行した場合の状態遷移の例を図 5 に示す。

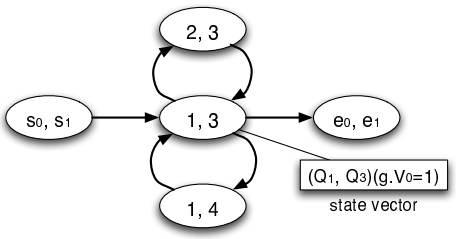


図 5: 状態遷移の例

この図 5 では、$(s_0, s_1)$ からスタートし、$(1, 3)$ に遷移する。そこから非決定的に $(2, 3)$ などに遷移する。

遷移していく途中の一部の状態を取り出したものを state vector と定義する。state vector は、いくつかの状態で構成され、可変長である (図 6)。

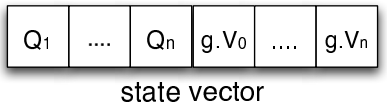


図 6: state vector の構成

次にタブロー法の適用について説明する。

状態遷移を様子を bynary tree を用いて表す。この tree を derivation tree(導出木) と定義する。derivation tree は以下のルールにより生成される。

- tree の探索は DFS で行う。
- state vector は、tree の末端に追加される。

- state vector を比較し、大きければ左、小さければ右に追加していく。
- 新たに追加する state vector が end state または、探索パス中の node と重複する場合は、tree に追加した後、node を一つ戻る。
- state vector が追加できなくなったら (つまり、全ての状態を尽くしたら) 終了する。

図 7 に derivation tree の生成例を示す。

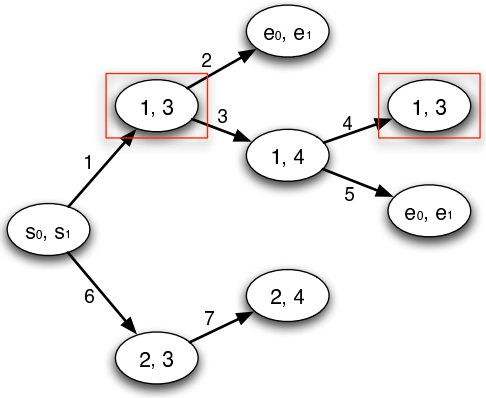


図 7: derivation tree の生成例

このように生成された derivation tree の branch をリスト化したものを history list として出力する。そして、この history list を基に検証を行う。

## 参考文献

[1] 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.

[2] 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.

[3] 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.

[4] http://spinroot.com/spin/whatispin.html Spin - Formal Verification