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[††]

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、Alternating Bit Protocol 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。

Verification of Continuation based C Program using Tableau method

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, as a program for verification, Alternating Bit Protocol is adopted and a purpose of performing verification using tableau method to it is carried out.

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のに比例し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。

† 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.

(2) 検証対象プログラムとして Alternating Bit Protocol(ABP) を採用し、それを CbC で記述する。
(3) ABP には送信側と受信側の 2 つのプロセスがあり、それぞれの単体シミュレーションを行う。
(4) Scheduler を作成し、ABP に組み込むことで ABP の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 ABP 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。

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)。

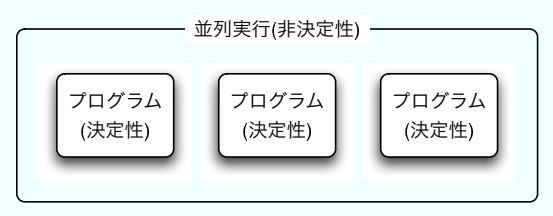

図 1 並列実行の非決定性

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 (プログラムのモデル) 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。

本稿では、scheduler を用意するまでを行った。

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)。

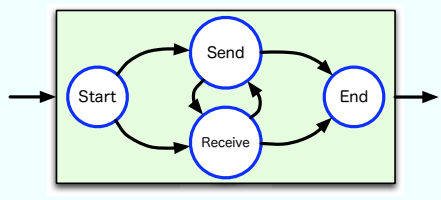

図 2 オートマトンの例

CbC の `code segment` と `interface`、引数付き `goto` は、それぞれ、オートマトンの状態遷移と入力および出力に対応しており (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。

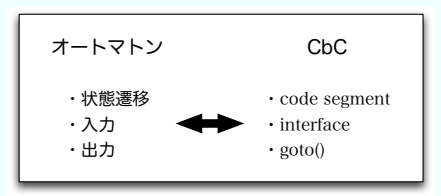

図 3 オートマトンと CbC の対応

6. Alternating Bit Protocol(ABP)

ここでは、検証用の例題プログラムとして Alternating Bit Protocol(以下 ABP) を用いる。ABP は、信頼できない通信路を挟んで、片方向のメッセージ転送を行うためのプロトコルである。送信側は、メッ

セージにヘッダーとして 1bit の Alternating Bit を付加して送信する。受信側は、メッセージに付加された Alternating Bit と、受信側が期待した bit との比較を行うことで、受信成功 (ACK) あるいは受信誤り (NACK) を送信側へ伝える。受信誤りが伝えられた場合、送信側は再送処理を行う。

ABP のプログラムは、以下のような仕様になる。

- Sender(送信側) と Receiver(受信側) の 2 つのプロセスからなる。
- Sender と Receiver はそれぞれ送信と受信の 2 つの状態を持っている。
- Sender と Receiver 間の通信は大域変数を用いる。
- Sender と Receiver はそれぞれ単体で実行可能である
- 両プロセスの制御は Scheduler が行う。

Sender と Receiver は、それぞれ 2 つの code segment を持ち、両プロセスは非同期に実行可能である (図 4)。

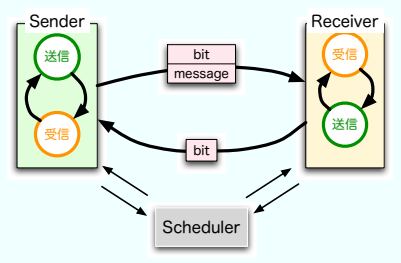

図 4 ABP の構成

7. Continuation based C(CbC)

当研究室では、Continuation based C(以下 CbC) 言語を提案している。この言語はアセンブラよりも上位で、C 言語よりも下位なプログラム言語である[1)]。

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。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 (図 5)。

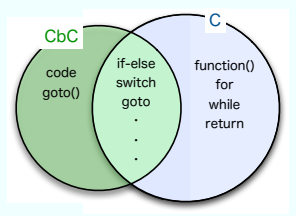

図 5 C と CbC の関係

7.1 CbC の要求仕様

CbC は以下のような要求仕様に従って設計されている[2)]。

- ハードウェアとスタックマシンの中間言語
  インタプリタ記述やコンパイラターゲットとして優れていること。アーキテクチャ依存性が少ないこと。また、アーキテクチャ依存性をモデル化できること。
- C 言語よりも下位の言語
  アセンブラよりも汎用性と記述生に優れ、C 言語と互換性があること。C を CbC にコンパイルでき、ハンドコンパイルの結果を同値なコードに変換できる。
- 明確な実行モデル
  C++や Prolog のような複雑な実行モデルは好ましくなく、ハードウェアに実行順序の変更を許す範囲を広くする。
- 状態遷移を直接記述できる
  Yacc のような表駆動や C のような巨大な switch 文ではなく、直接に状態遷移が実行できる。
- Thread を実行モデルに内蔵できる
  並列処理記述法ではなく、状態遷移として実現できる。
- クリティカルパスの最適化
  全体を散漫に最適化するコンパイルではなく、クリティカルパスを見つけ出して最適化できる。

これらの仕様は、ハードウェア記述とソフトウェア記述の両方を同時に行いつつ、C 言語よりも精密な実行記述を可能にするためのものである。また、CbC はプログラム変換やコンパイラターゲットとして使うことを意識している。状態遷移記述のみでは制御構造は静的なものになってしまう。CbC では、状態遷移記述に適した言語を作ることを考え、スタックマシンを避けて Continuation(継続) が導入されている。

7.2 継　　続

Scheme や C++、Java、あるいは、C も、大域脱出という形で継続を導入している。これらの言語では、継続は、必ず環境 (入れ子になった局所変数を格納するスタック) のセーブを伴う。CbC は関数呼び出しが存在しないために、この環境は存在しない。code 内部での局所変数は存在するが、それのネストは起こらない。そのため、環境抜きの継続を使用することができる。これは、基本的には、引数付きの (直接/間接)goto 命令である。これを軽量継続 (light-weight continuation) という。

CbC は code segment を引数付き goto で接続することで継続を実現している。code segment は、code という型で定義される。code segment への遷移は引数付き goto によってのみ行われる。goto 文の引数部分は interface と呼ぶ。code segment からの脱出は引数付き goto である。よって、CbC のプログラムは、複数の code segment が引数付き goto で接続されたものになる (図 6)。図 6 のように CbC プログラムの構

成は、オートマトンと似た構造になることがわかる。

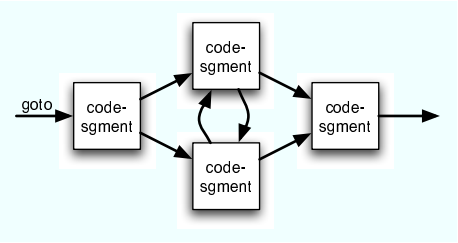

図 6 例：CbC プログラムの構成

このことより、CbC が状態遷移記述と相性が良い言語であるといえる。

8. CbC プログラムの作成と検証

検証用の例題プログラムとして Alternating Bit Protocol(ABP) を用いる。ABP には、Sender と Receiver の 2 つのプロセスがある。両プロセスは次のような構造体により内部状態を持つ。

```
struct packet {
    int        bit;  /* alternating bit */
    char     **msg;  /* messages */
    code (*next)();  /* next code segment */
};
```

8.1 Sender プログラム

Sender プログラムの仕様を以下にあげる。

- Sender の bit は 0 で初期化。
- 送信：bit を付加したメッセージを送信する。
- 受信:返ってきた bit' が `bit==bit'` なら `bit=!bit` とし、次のメッセージを用意する。
  `bit!=bit'` ならメッセージを再送信する。

このプログラムの送信 code segment を以下に示す。

```
code sendState(struct packet pkt, int i) {
    if (pkt.msg[i] == 0) goto ret(0), env;
    comm_bit = pkt.bit;
    comm_msg = pkt.msg[i];
    pkt.next = receiveState;
    goto schedule(pkt, i);
}
```

code segment の引数の部分を interface と呼び、これがオートマトンの入力に相当する。この code segment は、自身の内部状態である `pkt.bit` と `pkt.msg[i]` の送信処理を行う。

次に受信 code segment を示す。

```
code receiveState(struct packet pkt, int i) {
    if (pkt.bit == comm_bit) {
        pkt.bit = !pkt.bit;
        i++;
        pkt.next = sendState;
        goto schedule(pkt, i);
    } else {
        pkt.next = sendState;
        goto schedule(pkt, i);
    }
}
```

Receiver から返ってきた bit が ACK なら自身の bit を反転し、`int i` をインクリメントすることで次に送信するメッセージを用意する。また、返ってきた bit が NACK なら、再送信するために何も処理を行わず送信 code segment へ遷移する。

Scheduler を用いてプロセスの制御を行うため、code segment は全て Scheduler に遷移させている。

8.2 Receiver プログラム

Receiver プログラムの仕様を以下にあげる。

- Receiver の bit は 1 で初期化。
- 受信：受信した bit' が `bit==!bit'` ならメッセージを受理し、`bit = !bit` とする。
  受信した bit' が `bit!=!bit'` ならメッセージを受理しない。
- 送信：bit を送信する。これが ack(or nack) となる。

このプログラムの受信 code segment を以下に示す。

```
code receiveState(struct packet pkt, int i) {
    /* correct bit */
    if (pkt.bit != comm_bit) {
        printf("bit: %d\n", comm_bit);
        printf("message: %s\n", comm_msg);
        pkt.bit = !pkt.bit;
        pkt.next = sendState;
        goto schedule(pkt, i);
    /* wrong bit */
    } else {
        pkt.next = sendState;
        goto schedule(pkt, i);
    }
}
```

受信した bit が期待していた値なら、その bit とメッセージを表示し、自身の bit を反転する。受信した bit が期待していた値と違っていたら、自身の bit は反転せず、何も処理を行わず送信 code segment へ遷移する。

次に送信 code segment を示す。

```
code sendState(struct packet pkt, int i) {
    comm_bit = pkt.bit;  // ack or nack
    pkt.next = receiveState;
    goto schedule(pkt, i);
}
```

この code segment は、自身の bit をそのまま送信するだけである。この bit の値によって ACK か NACK になる。

8.3 Scheduler プログラム

Scheduler プログラムは CwC で記述している。状態の実行制御をキューを用いて行う。そのキューの操作を関数呼び出しとして実装しているため、CbC ではなく CwC で記述している。

まず、Scheduler は実行する順番に code segment をキューに登録する。そして、キューから code segment を一つずつ取り出し実行する。

Scheduler は、Sender と Receiver の各状態に遷移する code segment を持つ。また、実行する状態をキューに入れる code segment とそれを実行する code segment を持っている。

```
code task_entry(QueuePtr task_list,int i) {
    QueuePtr q;
    if (adrs[i] != 0) {
        q = new_queue(adrs[i], pkts[i]);
        if (!q) goto ret(0), env;
        task_list = enqueue(task_list, q);
        goto task_entry(task_list, ++i);
    } else {
        goto schedule(task_list, 0);
    }
}
```

この code segment は、関数 `new_queue()` によりキューを作成し、関数 `enqueue()` でキューに登録する。全てを登録し終わると `schedule` code segment へ遷移する。

```
code schedule(QueuePtr task_list, int i) {
    QueuePtr q;
    if (!task_list) goto ret(0), env;
    task_list = dequeue(task_list, &q);
    list = task_list;
    goto q->address(i);
}
```

この code segment は、キューから遷移する code segment の情報を取り出し、その code segment へ遷移する。Sender プログラムと Receiver プログラムは一つの処理が終わる毎に、この code segment へ遷移する。

9. シミュレーション

作成した ABP プログラムの単体シミュレーションおよび ABP 全体の動作のシミュレーションを行った。

9.1 単体シミュレーション

作成した Sender プログラムと Receiver プログラムの単体シミュレーションを行った。両プログラムとも、単体で実行するために、それぞれのプログラム中に dummy の処理を挿入している (図 7)。dummy の行う処理は、例えば、Sender プログラムでは、本来 Receiver プロセスが行うべき処理である。

Sender プログラムの単体シミュレーションの出力

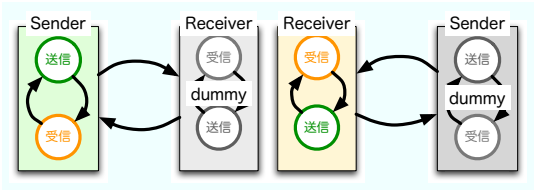

図 7 単体シミュレーション

は表 1 のようになった。Sender には本来、出力はないが、dummy 処理として Receiver の処理があるため、通信 bit とメッセージの出力がある。

表 1 単体シミュレーション：Sender 出力結果

code segment	送信	受信	送信	受信
bit	none	0	none	1
message	none	hoge1	none	hoge2
bit	none	0	none	1
message	none	hoge3	none	hoge4

この出力結果より単体シミュレーションが問題なく動作していることが確認できた。

9.2 ABP 全体の動作のシミュレーション

Sender プログラムと Receiver プログラムに Scheduler プログラムを加えて、ABP 全体の動作のシミュレーションを行った (図 8)。Sender プログラムと Receiver プログラムは Scheduler により制御する。

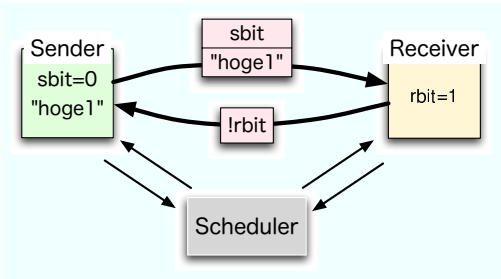

図 8 ABP 全体のシミュレーション

Scheduler を用いて、Sender プロセスと Receiver プロセスを交互に実行する。このシミュレーションの出力は表 2 のようになった。Sender および Scheduler には出力がなく、出力は Receiver のみである。

表 2 ABP シミュレーション：出力結果

Receiver	1	2	3	4
bit	0	1	0	1
message	hoge1	hoge2	hoge3	hoge4

このシミュレーションでは、通信路による誤り発生、消失などは入れていない。作成した ABP プログラムが正常に動作しているのが確認できた。

10. ま と め

並列動作のサンプルプログラムとして ABP を作成した。作成した Sender プログラムと Receiver プログラムの単体シミュレーションを行い、それぞれ正常に動作するのを確認した。

また、ABP 全体の動作のシミュレーションのために、CbC プログラムに Scheduler プログラムを作成し、挿入した。その後、Scheduler を用いた ABP のシミュレーションを行い、ABP が正常に動作するのを確認した。

11. 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。

11.1 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 9)。

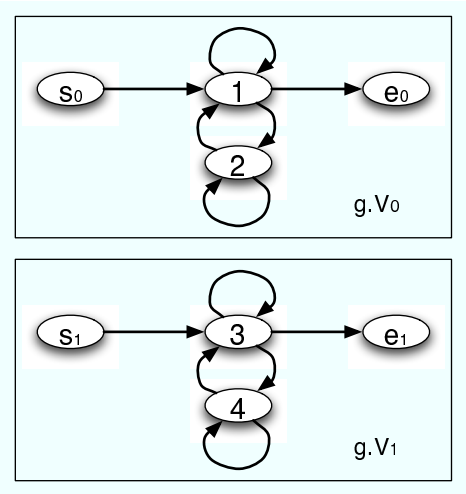

図 9 並列実行可能プロセス

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。

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 10 に示す。

この図 10 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

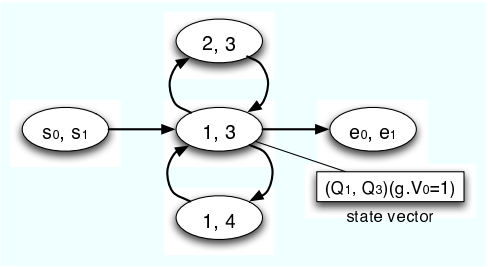

図 10 状態遷移の例

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 11)。

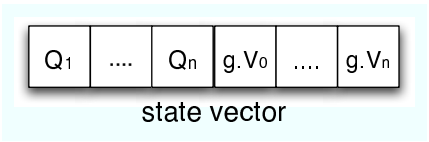

図 11 state vector の構成

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 12 に derivation tree の生成例を示す。

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

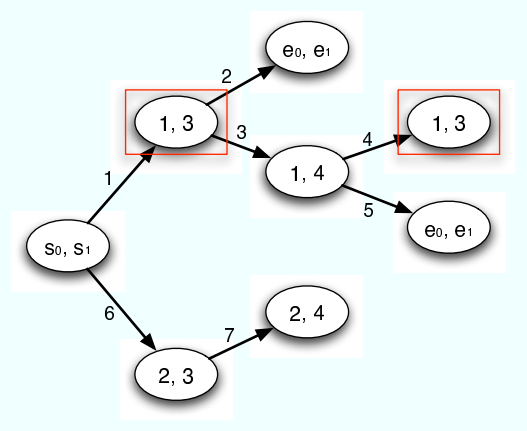

図 12 derivation tree の生成例

- Formal Verification