Gears OSでモデル検査を実現する手法について

東恩納　琢偉[1,a)]　奥田　光希[2,b)]　河野　真治[3,c)]

概要：GeasOS は CbC で記述されており処理単位である codeGear の間に自由にメタ計算をはさむことができる。ここに dataGear の状態を記録することにより、ユーザプロセスあるいはカーネルそのもののモデル検査が可能になる。一般的なモデル検査では状態数の爆発は避けられない。記録する状態を抽象化あるいは限定する手法について考察する。

1. Gears OS と CbC

Gears OS は軽量継続を基本とする言語 CbC を用いた OS の実装である。アプリケーションやサービスの信頼性を OS の機能として保証することを目指している。信頼性を保証する一つの方法は、プログラムの可能な実行を数え上げて要求仕様を満たしているかどうかを調べるモデル検査である。本論文では、Gears OS 上のアプリケーション、さらに Gears OS そのものを Gears OS 上でモデル検査することに関して考察する。

CbC (Continuation based C) は C の機能を持ち、goto 文で遷移する codeGear という単位を持っている。通常の関数呼び出しと異なり、stack や環境を隠して持つことがなく、計算の状態は codeGear の入力ですべて決まる。codeGear の入力は dataGear と呼ばれる構造体だが、これにはノーマルレベルの dataGear とメタレベルの dataGear の階層がある。メタレベルには計算を実行する CPU やメモリ、計算に関与するすべてのノーマルレベルの dataGear を格納する context などがある。context は通常の OS のプロセスに相当する。

メタレベルから見ると、codeGear の入力は context ただ一つであり、そこから必要なノーマルレベルの dataGear を取り出して、ノーマルレベルの codeGaer を呼び出す。この取り出しは stub と呼ばれる meta codeGear によって行われる。

Listing 1 codeGaer と goto

```
__code cg0(int a, int b) {
    goto cg1(a+b);
}

__code cg1(int c) {
    goto cg2(c);
 }
```

ノーマルレベルの codeGaer の goto

Listing 2 codeGear の stub と goto

```
 __code popSingleLinkedStack_stub(struct
 Context* context) {
   SingleLinkedStack* stack = (
   SingleLinkedStack*)context->data[D_Stack
     ]->Stack.stack->Stack.stack;
   enum Code next = context->data[D_Stack]->
   Stack.next;
   Data** O_data = &context->data[D_Stack]->
   Stack.data;
   goto popSingleLinkedStack(context, stack,
   next, O_data);
   }
```

1 琉球大学大学院理工学研究科情報工学専攻
2 琉球大学工学部工学科知能情報コース
3 琉球大学工学部
a) ikkun@cr.ie.u-ryukyu.ac.jp
b) Koki.okuda@cr.ie.u-ryukyu.ac.jp
c) kono@ie.u-ryukyu.ac.jp

メタレベルのでの codeGaer の stub と goto meta

ここで codeGear の実行は OS の中での基本単位である必要がある。つまり、codeGear は並行処理などにより割り込まれることなく、codeGear で記述された通りに実行される必要がある。これは一般的には保証されない。他の codeGear が共有された dataGear に競合的に書き込んだり、割り込みにより処理が中断したりする。しかし、Gears OS が codeGear が正しく実行さることを保証する。つまり、Gears OS はそのように実装されているとする。

プログラムの非決定的な実行は入力あるいは並列実行の非決定性から生じる。後者は並列実行される codeGear の順列並び替えになる。従って、これらの並び替えを生成し、その時に生じる context の状態をすべて数え上げれればモデル検査を実装できることになる。

ただし、context の状態は有限状態になるとは限らないし、有限になるとしても巨大になることがある。しかし、OS やアプリケーションのテストだと考えると、それらの動作の振る舞いを調べるのに十分な状態があれば良い。一つの方法は context の状態をなんらかの方法で抽象化することである。

本論文では context の中の状態を特定するのに必要なメモリ領域を登録する手法を用いる。メモリ領域は context の中のアドレスとサイズで指定される。その中に格納されるのはノーマルレベルの dataGear とメタレベルの dataGear になる。ノーマルレベル側にはアドレスは格納されないと仮定する。これは、ノーマルレベルの計算は末尾再帰な関数型プログラミングと仮定するのとだいたい同等である。現在は Agda (Haskell 上の定理証明系の関数型言語) で記述することを考えている。 メタレベル側にはメモリアドレスなどが直接入ることになる。

メモリ領域の集合で一つの状態が定義される。この状態をさらに格納するデータベースを用意する。codeGear のシャッフルの深さ優先探索を行ない、新しく生成された状態をデータベースで参照し、既にあれば、深さを一つ戻り、別な探索枝に移る。新しい状態が生成されてない、あるいは、バグを見つければモデル検査は終了と言うことになる。

ここでは例題として Dining Phisopher 問題の dead lock の検出を行う。研究の段階としては

(1) Dining Phisopher を Gears OS 上のアプリケーションとして実装する (DPP)。(2) DPP を codeGear のシャッフルの一つして実行する meta codeGear を作成する。(3) 可能な実行を生成する iterator を作成する。(4) 状態を記録する memory 木と状態 DB を作成する。

この段階で DPP のモデル検査が可能になるはずである。

一方で Gears OS そのものも codeGear で記述されている。CPU 毎の C.context、そして、それが共有する Kernel の K.context、それからユーザプログラムの U.context などの context からなる。これら全体は meta dataGear である K.context に含まれている。

U.context が DPP のように単純なものならば OS 全体の context も複雑にはならない。したがって、これ全体を Gears OS で実行することが可能である。

(5) Gears OS を含む codeGear のシャッフル実行を行ない、モデル検査を実現する

これにより、Gears OS の自分自身によるモデル検査が可能になる。この時に、検査する codeGear と検査される codeGear は同じ物であるが、実行する meta codeGar が異なっている。現状では、これは異なる meta codeGear を指定してコンパイルしなおすことにより実現する。

Gears OS の実装は Unix 上のアプリケーションとしての実装と、x.v6 の書き換えによる実装の二種類があるが、前者ではアプリケーションは OS に直接リンクされる。後者では x.v6 の exec 機構により実行される。実際に OS のモデル検査を実行するためには、必要な meta dataGear/meta codeGear の emulator を書く必要がある。しかし、検査する必要がない部分は無視できるようにしたい。

Gears OS は並列実行機構を持っているので、

(6) モデル検査を並行実行することができる

はずである。

2. 既存のモデル検査手法

モデル検査の方法としてよく利用される物として、SPIN と java path finder(以下 JVM) というツールがある。
SPIN は Promela という仕様記述言語で記述する事で C 言語の検証器を生成する事で、コンパイルまたは実行時に検証する事ができる。チャネル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 では以下の性質を検査する事ができる。

- アサーション
- デッドロック
- 到達生
- 進行性
- 線形時相論理で記述された仕様

Java Path Finder(JPF) は java プログラムに対するモデル検査ツールで、java バーチャルマシン (JVM) を直接シミュレーションして実行している。そのため、java のバイトコードを直接実行可能である。バイトコードを状態遷移モデルとして扱い、実行時に遷移し得る状態を網羅的に検査する。バイトコードの実行パターンを網羅的に調べるために、膨大な CPU 時間を必要とする。また JVM ヘースであるため、複数のプロセスの取り扱いが出来ず、状態空間が巨大になる場合は直接実行は出来ず、一部を抜き出してデバックをするのに使用される。
JPF では以下の事ができる。

- スレッドの可能な実行全てを調べる
- デッドロックの検出

図 1　CodeSegment と DataSegment

- アサーション
- Partial Order Reduction

3. Continuation based C

GearsOS は Continuation based C (以下 CbC) という言語を用いて拡張性と信頼性を両立させることを目的として本研究室で開発されている。CbC は C 言語と似た構文を持つ言語であるが、CodeSegment と DataSegment を用いるプログラミングスタイルを提案している。CodeSegment は処理の単位である。CodeSegument は値を入力として受け取り処理を行ったあと出力を行う、また他の CodeSegment を接続していくことによりプログラムを構築していく。DataSegment は CodeSegment が扱うデータの単位であり、処理に必要なデータが全て入っている。DataSegmen は Input DataSegment と呼ばれ、出力は Output DataSegment と呼ばれる。CodeSegment A と CodeSegment B を接続したとき、A の Output DataSegment は B の入力 Input DataSegment となる。

CodeSegment の接続処理はメタ計算として定義されており、実装や環境によって切り替えを行なうことができる。検証を行なうメタ計算を定義することにより、CodeSegment の定義を検証用に変更せずプログラムの検証を行なう。
CbC における接続は goto を用いて行われる。got は関数呼び出しのような環境変数を持たず goto の直後に遷移先を記述することで、遷移先に接続される。これを軽量継続と言い、遷移元の処理に囚われず、遷移先を自由に変更する事が可能でり 遷移元の code gear の goto 先以外に変更する事なく、処理の間にメタレベルの計算を挿入する事が可能である。CbC における遷移記述はそのまま状態遷移記述にすることができる。**??**

GearsOS では CodeSegment と DataSegment はそれぞれ CodeGear と DataGear と呼ばれている。マルチコア CPU 環境では CodeGaer と CodeSegment は同一だが、、GPU 環境では CodeGear には OpenCL/CUDA における kernel も含まれる。kernel とは GPU で実行される関数のことである。

4. DPP

検証用のサンプルプログラムとして Dining Philosohers Ploblem (以下 DPP) を用いる。これは資源共有問題の 1 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おり、各々スパゲッティーの皿

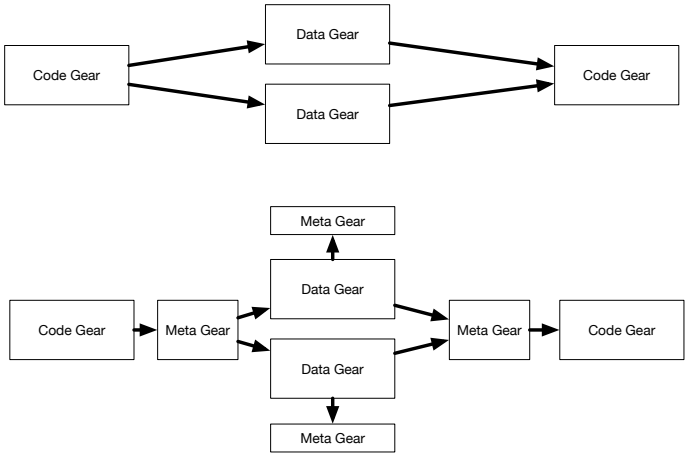

図 2　Gears OS のメタ計算

が目の前に用意されている。スパゲッィーはとても絡まっているので食べるには 2 本のフォークを使わないと食べれない。しかしフォークはお皿の間に一本ずつおいてあるので、円卓にフォークが 5 本しか用意されていない。**図 ??** 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、左右のオークを手に取ろうと試み、2 本のフォークを取ることに成功するとしばし食事をし、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置かれるのを待つ。
各哲学者を 1 つのプロセスとすると、この問題では 5 個のプロセスが並列に動くことになり、全員が 1 本ずつフォークを持って場合はデッドロックしていることになる。プロセスの並列実行はスケジューラによって制御することで実現する。

Listing 3
DPP

```
code pickup_lfork(PhilsPtr self,
TaskPtr current_task)
{
    if (self->left_fork->owner == NULL) {
        self->left_fork->owner = self;
        self->next = pickup_rfork;
        goto scheduler(self, current_task);
    } else {
        self->next = hungry1;
        goto scheduler(self, current_task);
}
```

5. タブロー展開と状態数の抽象化

GearsOS におけるモデル検査はタブロー展開を用い

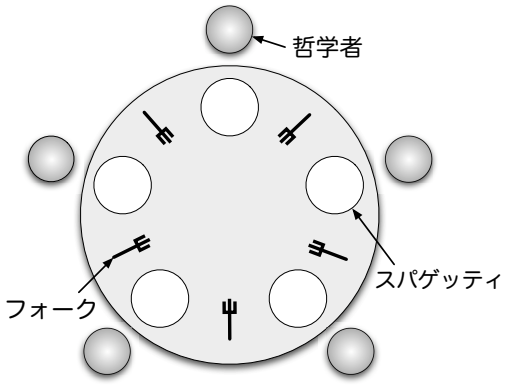

図 3　Dining Philosohers Ploblem

ることでデッドロックを調べる。タブロー法は生成可能な状態の全てを生成する手法である。反例を探す場合は反例が見つかった時点で状態の生成を停止してもよいが、証明を行う場合は全ての状態を生成する必要がある。状態の生成は初期状態から非決定的に生成される全ての次の状態を生成することにより行われ、これを状態の展開という。証明はプログラムの状態の数に比例し、またプログラムが含む変数の数の指数状の計算量がかかる。この展開の際に仕様うも同時に展開することでプログラムに対する仕様の検証を行う事が可能である。

タブロー法は実行可能な状態の組み合わせを深さ優先探索で調べ、木構造で保存する方法である。この時、同じ状態の組み合わせがあれば共有することで状態を抽象化する事で、状態数が増えすぎる事を抑える。

6. GearsOS を用いたモデル検査

DPP は哲学者 5 人が同時に行動するので、5 つのスレッドで同時に処理することで状態を生成する事ができる。まず Gears OS の並列構文の par goto が用いることでマルチスレッド処理の実装を行う。 par goto は引数として、data gaer と実行後に継続する`__exit` を渡す。par goto で生成された Task は`__exit` に継続する事で終了する。これにより Gears OS は複数スレッドでの実行を行う事が可能である。また Gears OS には　Synchronized Queue というマルチスレッドでのデータの一貫性を保証する事ができる Queue があり、これを使い 5 つのフォークの状態を管理する。Syncrhonized Queue は CAS(Check and Set) を用いて実装されており、値の比較、更新をアトミックに行う命令である。CAS を使う際は更新前の値と更新後の値を渡し、渡された更新前の値を実際に保存されているメモリ番地の値と比較し、同じデータ今日がないため、データの更新に成功する。異なる場合は他の書き込みがあったとみなされ、値の更新に失敗し、もう一度 CAS を行う。5 スレッドで行われる処理の状態は以下の 6 通りで、think のあと Pickup Right fork に戻ってくる。

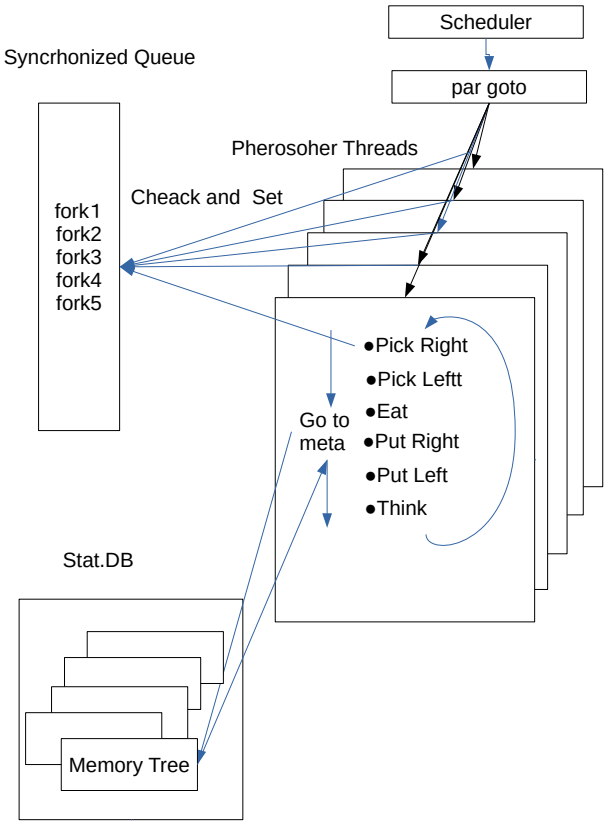

図 4　DPP chacking

- Pickup Right fork
- Pickup Left fork
- eating
- Put Right fork
- Put Left fork
- Thinking

この状態は goto next によって遷移する。またこの状態遷移は無限ループするので MemoryTree に保管し、保管されている状態とは stat DB によって保管される

参考文献