Gears OS のモデル検査の実装

河野真治
琉球大学工学部
Shinji KONO
Faculty of Engineering, University of the Ryukyus

Abstract

Gears OS は継続を基本とする Kernel と User Program の記述を採用している。メタプログラミングにより、元のプログラムを変更することなくモデル検査を行うことができる。ここでは codeGear 単位での可能な並列実行の列挙を登録した dataGear に対して行うことにより CTL で記述された仕様を検証することができた。検証のメモリと CPU の使用量、技術手法についての考察を行う。

1 OS の信頼性

OS とは一般的にハードウェアへのアクセス、資源管理を行っているソフトウェアである。つまりコンピュータに接続されている全てのメモリやハードディスクといった記憶装置、また CPU や GPU といった計算処理装置には通常 OS の機能を利用する事でしかアクセスすることは出来なくなっている。

OS はアプリケーションやサービスの信頼性を保証する基礎となる部分だが、同時に拡張性も必要となる。しかし、OS は非決定的な実行を持つため、その信頼性を保証するには従来のテストとデバックでは不十分と考えられている。

テストとはソフトウェアやアプリケーションの検証は用いられる手法で、ソフトウェアあるいはアプリケーションに対して、決まったの入力を与えた場合に設計で予想された出力が返ってくる事を確かめる事によって信頼性を保証する手法である。このテストによる手法は、検証側が定めた範囲での入力による検証であるためテストしきれない部分が残ってしまう可能性がある。

テスト以外の信頼性を保証する検証としては形式検証があり、形式検証には定理証明とモデル検査 [?] の２つの手法がある。定理証明は数学的な証明を用いて信頼性を保証する手法である。証明を用いるため、入力や、状態数に比較的影響をうけずに検証を行う事が出来るが、場合分けの複雑化や、場合分け自体が多くなる事がある。agda[?] といった定理証明支援器と言われるものがある。しかし証明には数学的な知識が不可欠となるモデル検査は抽象化されたソフトウェアの実装である。プログラムの仕様である logic を満たすかどうかをモデル検査器を用いて調べる事で信頼性を保証する。モデル検査の場合、自動で全ての状態を網羅的に出力し調べるため検証時間を工夫して短くすることが出来る。

本研究室で開発している Gears OS[?] は アプリケーションやソフトウェアの信頼性を OS の機能として保証することを目指しており、信頼性を保証する手法としてモデル検査や hoare logic を用いた定理証明 [?] を用いて信頼性へのアプローチを行っている。本論文では、この Gears OS におけるモデル検査を実現する手法について考察する。

2 モデル検査とは

モデル検査は、検証したい内容の時相論理式 p をつくり、対象のシステムの初期状態 s のモデル M があるとき、M,s が p を満たすか (M,s $\models$ p と表記される) モデル検査器を用いて調べることによって信頼性を保証する手法である。

様相論理式には CTL(Computational Tree Logic) を用いる。CTL の様相演算子は、枝を表すパス量化子と、いつ成り立つかを表す時相演算子がある。

パス量化子

- ある枝で存在する E

- 全ての枝で存在する A

時相演算子

- 枝の次の状態で成り立つ X
- この先いつか成り立つ F
- このあとずっと成り立つ G
- この先いつか状態 a になる、そのときまでは状態 b が成り立つ U
- 状態 b がなりたつまで状態 a 成り立つ R

この様相演算子を用いて表した CTL の導出木が図 2.1 から 2.4 のようになる。

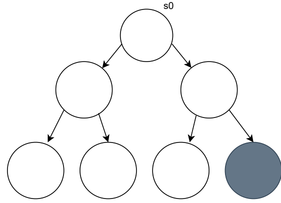

Figure 1: ある枝でいつか真になる EF

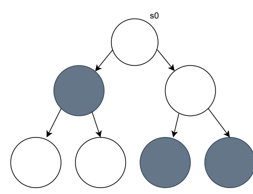

Figure 2: すべての枝でいつか真になる AF

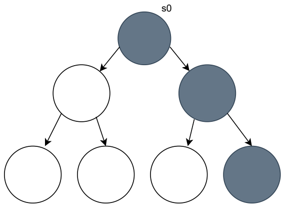

Figure 3: ある枝でいつも真になる EG

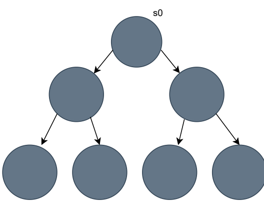

Figure 4: すべての枝でいつも真になる AG

参考:蓮尾 一郎, モデル検査入門 (2009 年)p12[?]

3 モデル検査の実装例

モデル検査の方法として 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 ではオートマトンの並列実行処理の検証が可能であるが、これは厳密には実行するステートをランダムに選択し、実行することで実現している。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
- 到達性
- 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。しかし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 ベースであるため、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 は Java の reflection の機能に依存しており、この部分は Java9 において大幅に変更されてしまったので、Java9 以降では動作しない。

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
- アサーション
- Partial Order Reduction

4 Continuation based C

Gears OS は軽量継続を基本とする言語 Continuation based C (以下 CbC)[?] を用いた OS の実装である。CbC は Code Gear という単位を用いて記述する C の機能を持つプログラミング言語である。コンパイルには llvm-cbc[?] を用いて行う。Code Gear は一般的な処理記述に

あたり関数に比べて細かく分割されている。Code Gear 間の遷移は軽量継続である goto 文によって行われる。軽量継続である goto は継続前の Code Gear に戻ることはないため、プログラムの記述をそのまま状態遷移に落とし込むことが出来る。C の関数の型が**__code** であるような構文で定義することができる。つまり、codeGear は dataGear を通して、次の codeGear に goto で接続される (図 5)。

例えば、ソースコード 3.1 は DiningPhilosohersPloblem(以下 DPP) の例題で右の fork を取るという処理を行っているは以下のように書ける。ここでは **cas** (check and set) と busy wait で書いてある。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。

Listing 1: pickUrforkp

```
__code putdown_rfork(struct PhilsImpl* phils, __code next(...)) {
    struct AtomicT_int* right_fork = phils->Rightfork;
    goto right_fork->set(-1, putdown_lfork);
}
```

メタ計算 と stub 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。

遷移は次の図 (6) の上のように codeGear から Code Gear に移動するだけだが、その前に出力する dataGear を context に書き出す必要がある。これは通常の関数呼び出しの return の処理に相当する。

図 3.2 の下はメタレベルから見た codeGear である。goto 先は meta という meta codeGaer であり、そこで必要なメタ計算が行われる。ここに任意のメタ計算を置くことができる。この場合の引数は context と行き先を表す番号だけである。ソースコード 3.2 は DPP における 右のフォークを持ち上げる例題の stub[?] である pickup_rfork_stub と、その stub meta に goto するノーマルレベルのものになる。このようにノーマルレベルの CodeGear からメタレベルに遷移する際には goto meta で引数を渡すだけで、メタレベルの計算は隠されている。

Listing 2: pickuprfork

```
__code pickup_rfork_stub(ContextPtr cbc_context)
{
    PhisPtr self = GearRef(cbc_context,D_Phisoloper);
    PhisPtr next = GearRef(cbc_context,D_next);
    goto pickup_rfork(ContextPtr cbc_context, self,next);
}

__code pickup_rfork(PhilsPtr self,
    __code __next(ContextPtr cbc_context));
{
    if (cas(self->right_fork->owner, self) == self) {
        goto meta(context,C_phil_think);
    } else {
        goto meta(context,C_pickup_rfork);
    }
}
```

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。(図 7) これは通常の関数呼び出しの ABI(引数とレジスタやスタック上の関係を決めたバイナリインターフェース) に相当する。

Figure 5: codeGear と DataGear

stub codeGear は codeGear の接続の間に挟まれる Meta Code Gear である。ノーマルレベルの codeGear から MetadataGear である Context を直接参照してしまうと、ユーザーがメタ計算をノーマルレベルで自由に記述できてしまい、メタ計算を分離した意味がなくなってしまう。stub Code Gear はこの問題を防ぐため、Context から必要な dataGaer のみをノーマルレベルに和刺す処理を行なっている。

このようにノーマルレベルの Code Gear からは context を見ることはできず、メタ計算レベルではノーマルレベルの引数の詳細を気にしないで処理を行うことができるようになっている。ノーマルレベルとメタレベルは、必要ならば CPU の system mode と user mode の状態を変えても良い。

5 Gears OS における平行実行の検証

codeGear の実行は OS の中での基本単位として実行される。つまり動作として codeGear は並行処理などにより割り込まれることなく記述された通りに実行される。これは OS によって相対的に保証される。例えば機械故障のような場合まで保証するものとは異なる。他の codeGear によって共有されている 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ても、Gears OS は codeGear が正しく実行さることを保証する。

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発生する。後者は並列実行される codeGear の順列並び替えとなる。よ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ず、また有限になるとしても巨大になる場合が考えられる。この場合は OS やアプリケーションのテストとして動作する十分な状態にまで、context の状態を抽象化することができればモデル検査を行える。

context はプロセス全体に相当するがモデル検査では変更された部分のみを考慮すれば良い。そこで、メモリ領域の集合から状態を作り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なくなる、もしくは、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。

(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。

(5) 状態 DB が CTL の仕様を満たしているかをフラグを使って調べる。

(2)-(4) は Gears OS のメタ計算として実装される。この段階で DPP のモデル検査が可能になるはずである。

6 OS そのもののモデル検査

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 からなると考えれ、これら全体は meta dataGear である K.context に含まれていると考える。

これらのうちのいくつかはハードウェアに関連した動作を持っているが、それをエミュレーションに置き換えることが常に可能である。Gears OS を構成している dataGear / codeGear は特殊なものではなく、普通にモデル検査の対象となる context に登録して実行して良い。つまりモデル検査を Gears OS 全体に対して実行可能であると考えられる。この時、外側では普通の

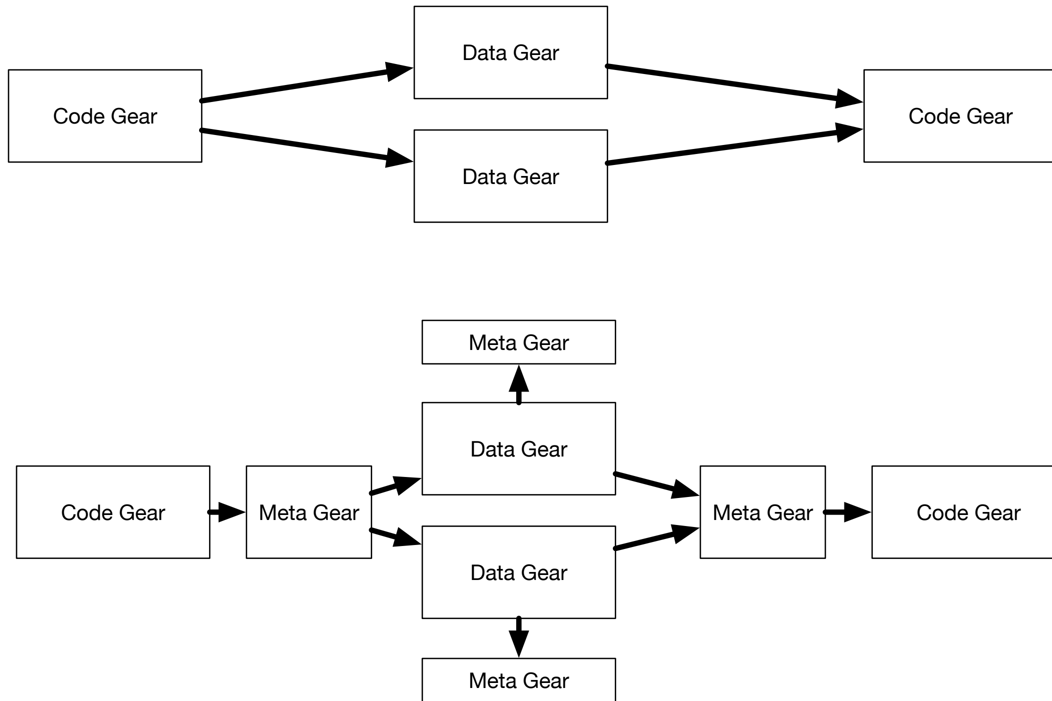

Figure 6: Gears OS のメタ計算

Gears OS が動作し、モデル検査の対象となる context 内ではエミュレートされた dataGear / codeGear が実行される。

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

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と考えている。

Gears OS は並列実行機構を持っているので、

(6) モデル検査そのものを並行実行 [?] することができる

と考えられる。

7 Dining Philosophers

モデル検査の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。??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

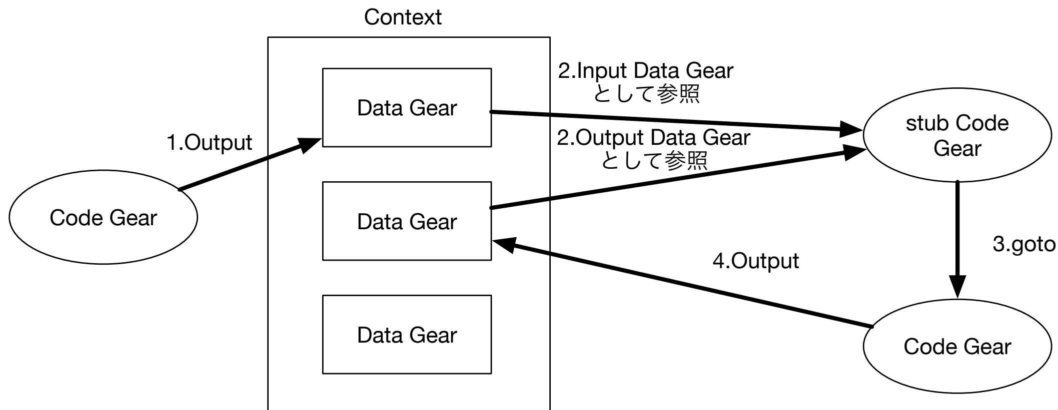

Figure 7: Contex が持つ DataGaer へのアクセス

となる。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。

以下は DPP における右側のフォークを取るプログラムである。最初に右のフォークを持たれているかどうかを確認し、いなければ自分を持ち主に登録する。その後 next に次に遷移する自分の状態を入れ scheduler に遷移することによって scheduler を通して次の状態に遷移する。このとき scheduler からメタ計算を挟むことによって状態を MemoryTree に入れる事ができる。左のフォークの持ち主がいた場合は飢餓状態を登録し scheduler に遷移する事で待機状態を維持する。

Listing 3: pickuplfork

```
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);
}
```

8 Gears OS を用いた DPP

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。5 つのフォークの状態は CAS で管理する。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。

- 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。状態を遷移する際、MemoryTree によって状態を保存する。またこの状態遷移は無限ループするので MemoryTree に保管される。またこの MemoryTree はスレッドの数だけあり、sutats DB によってまとめられている。

DPP の状態遷移は 6 つの状態を繰り返すため、同じ状態が出た場合には終了させなければならない。そこで state DB を用いて同じ状態を検索することで終了判定をだす。

この実行は Single thread に行われるが、iterator を使って並行実行しても良い。

必要な時相論理的な仕様は codeGear にコンパイルすることができるので、それを同時に走らせることによりチェックできる。これは SPIN の LTL の仕様記述と同じことになる。このようにモデル検査を codeGear と dataGear 上に実現することができる。

メモリ領域の登録には **add_mamory_area(ContextPtr cbc_contex ,void *addrss,long length)** のような API を用いる。状態の格納は、mera codeGear で行われるので、customize が可能である。この段階で対称性の利用や状態の抽象化を行うこと可能となる。

9 Gears OS でのモデル検査実装

モデル検査を行うのに、次のものを用意した。

- MCTaskManagerImpl.cbc (導出木を作るために dataGear に process (context) を登録する。)
- MCWorker.cbc (導出木を作る meta codeGear)
- TaskIterator.c

  次の実行を選択する **interator**

- memory.c (memory range の扱い (2 分木))
- crc32.c (memory 状態の hash)
- state_db.c (状態の data base (2 分木))

CbC で実装された DPP は ソースコード 4.2 の Perl script meta.pm によって正規表現を用いて goto 時のメタ計算実行を実現する。

mcMeta はモデル検査を行う場合で random は並行実行のシミュレーションを行う場合となっている。7 行目は多次元リストの PhilsImpls を regular expression に渡している。これは goto の遷移先である。その後 11 行目で受け取った文字列を goto 先にランダムに、配置しており。16 行目では文字列を受け取った文字列に遷移する前に mcMeta を挟むようにしている。

Listing 4: meta.pm

```
package meta;
use strict;
use warnings;

sub replaceMeta {
  return (
    [qr/PhilsImpl/ => \&generateMcMeta],
  );
}

sub generateRandomMeta {
  my ($context, $next) = @_;
  return "goto random($context, $next);";
}

sub generateMcMeta {
  my ($context, $next) = @_;
  return "goto mcMeta($context, $next);";
}

1;
```

10 導出木の作り方

Listing 5: putdown_lfork

```
__code putdown_lfork(struct PhilsImpl* phils, __code next(...))
    struct AtomicT_int* left_fork = phils->Leftfork;
    goto left_fork->set(-1, thinking);
}
```

ソースコード 5.5 は DPP の例題のもので、Philospher の Leftfork を置く部分である。フォークは各 Philosher 間で共有されるデータのため、競合が起きないように CAS を行う必要がある。このソースコードは以下のソースコード 5.6 に変換される。Gearef は Context から Data Gaer を参照するマクロになっている。goto 先が mcMeta になっている。

Listing 6: putdownlforkImpl

```
__code putdown_lforkPhilsImpl(struct Context *context,
        struct PhilsImpl* phils, enum Code next) {
    struct AtomicT_int* left_fork = phils->Leftfork;
    Gearef(context, AtomicT_int)->atomicT_int = (union Data*) l
    Gearef(context, AtomicT_int)->newData = -1;
    Gearef(context, AtomicT_int)->next = C_thinkingPhilsImpl;
    goto mcMeta(context, left_fork->set);
}
```

Gears OS のノーマルレベルの code は変換されるが、mcMeta は __ncode と記述されており、これは meta として扱い変換しない意味である。9 行目にある mcti にタスクを渡している。

2

Listing 7: mcMeta

```
__ncode mcMeta(struct Context* context, enum Code next) {
    // 次の実行を context に覚えておく
    context->next = next; // remember next Code Gear
    // Worker (複数)と TaskManager(singleton)を context から取ってくる
    struct MCWorker* mcWorker =
(struct MCWorker*) context->worker->worker;
    StateNode st ;
    StateDB out = &st;
    struct Element* list = NULL;
    struct MCTaskManagerImpl* mcti = (struct MCTaskManagerImpl *)mcWorker->taskManager->taskManager;
    out->memory = mcti->mem;
    out->hash = get_memory_hash(mcti->mem,0);
    if (dump) {
        dump_memory(mcti->mem); printf("\n");
    }
    // state を db から探す
    int found = visit_StateDB(out, &mcti->state_db, &out, mcWorker->visit);

    // モデル検査フラグの更新
    mcti->statefunc(mcti, mcWorker, mcWorker->parent, out, mcWorker->checking);
    if (found) {
      // 既に状態 db にあった場合
      // iterator を探す、終わっていたら上に戻る
      while(!(list = takeNextIterator(mcWorker->task_iter))) {
          // no more branch, go back to the previous one
          TaskIterator* prev_iter = mcWorker->task_iter->prev;
          if (!prev_iter) {
            //  もう上がないので全部探した
            printf("All done count %d repeat %d\n",mcWorker->count,mcWorker->visit);
            memory_usage();
            // flag の更新を見る
            if (! mcWorker->change && mcWorker->checking) {
              exit(0);
            } else if (! mcWorker->change ) {
                // flag の更新は終わったので、フラグを調べる
                mcWorker->checking = 1;
            }

            // 最初から始める
            mcWorker->change = 0;
            mcWorker->visit++;
            // start from root state and iterator
            mcWorker->depth = 0;
            struct SingleLinkedQueue* mcSingleQueue = (struct SingleLinkedQueue*)mcWorker->mcQueue->queue;
            mcWorker->task_iter = createQueueIterator(mcSingleQueue->top->next,mcWorker->rho,NULL);
          } else {
            // 一つ上に戻る
            mcWorker->depth--;
            freeIterator(mcWorker->task_iter);
            mcWorker->task_iter = prev_iter;
          }
      }
      // 戻った時にメモリを書き戻す
      restore_memory(mcWorker->task_iter->state->memory);
      context = (Context *)list->data;
      // printf("restore list %x next %x\n",(int)list,(int)
    } else {
      // 一段、深く実行するので、新しく iterator を作る
      mcWorker->depth++;
      struct SingleLinkedQueue* mcSingleQueue = (struct Sin
      mcWorker->task_iter = createQueueIterator(mcSingleQue
    }
    // normal level に戻る
    mcWorker->parent = out;
    mcWorker->count++;
    mcWorker->mcContext = context;
    //goto list->phils->next(list->phils,list);
    goto meta(context, context->next);
}
```

11 モデル検査のフラグの管理

ソースコード 4.3 の mcDPP.h はモデル検査で使われるフラグの宣言をしている。

Listing 8: mcDPP.h

```
#ifndef MCDPP_H
#define MCDPP_H 0
#include "context.h"
#include "ModelChecking/state_db.h"
/*
  00 don't care
  01 true
  11 false
 */
enum DPPMC_F {
  t_eating = 0x1,          // eating
  f_eating = 0x3,          // ¬eating
  t_F_eating  = 0x4,       // <> eating
  f_F_eating  = 0xc,       // ¬<> eating
  t_GF_eating  = 0x10,     // [] <> eating
  f_GF_eating  = 0x30,
// ¬[] <> eating
};
extern void mcDPP(struct MCTaskManagerImpl* mcti, struct MC
```

しかし今回の DPP の例題においては t_eating と f_F_eating のフラグしか使用してしいない。DPP の例題は食事とそれ以外の状態を循環しているため t_eating フラグの「食事中」と f_F_eating フラグの「いつかは食事にする」という 2 つの状態で表すことが可能となる。

現在 eating かどうかを調べて、t_eating をまず設定する。導出木の生成を繰り返し行い、次の状態が t_eating

または f_F_eating の時に f_F_eating を設定する。フラグが変化しなくなったら終了である。この方法だと導出木を F フラグの深さの分だけ繰り返すことになる。

f_F_eating のないノードがあれば、それはライブロックまたはデッドロックということになる。以下のノードで一部 f_F_eating がないノードがあれば、それはライブロックということになる。

フラグは now と next を見比べながら update する。すべての状態は 今の context にのっているが、過去は add_memory_range で記録されたものあるいはフラグしか見れないようになっている。

ソースコード 4.4 の mcDPP.cbc ではフラグの比較によるモデル検査を行っている。

Listing 9: mcDPP.cbc

```
void mcDPP(struct MCTaskManagerImpl* mcti, struct MCWorker* mcWorker,
           StateDB now,StateDB next, int check) {
    PhilsImpl* phils =
        (PhilsImpl*)GearImpl(mcWorker->mcContext, Phils, phils);
    int prev_now = now->flag;
    int prev_next = next->flag;
        if (phils->self != 1) return;
        enum Code nextc = mcWorker->mcContext->next;
        if (nextc == C_putdown_rforkPhilsImpl ) {
          next->flag |= t_eating;
        }
        if ((next->flag & t_eating )||(next->flag & t_F_eating)) {
          now->flag |= t_F_eating;
        }
        if ( prev_now != now->flag || prev_next != next->flag )
            mcWorker->change = 1;
        if (check) {
            if (!(now->flag & t_F_eating)) {
                 printf("not <> eating\n");
           }
    }
}
```

最初に今のフラグと次のフラグを取得し、次に **phils->self** が 1 である場合は return でぬける。12 行目では 次と今のフラグが et_ating または t_F_eating であれば t_F_eating; 15 行目では直前のフラグと今のフラグ、または直前のフラグと次のフラグが違っていると change に 1 が入り、動き続け、そうでなければ 18 行目で今のフラグと t_F_eating を比較し、あっている場合には **not<>eating** となり、終了する。

12 実行結果

次にモデル検査を行った結果を一部抜き出して掲載する。

not<>eating はデッドロックフラグである次の 5 つの値はフォークを持っているスレッドを表しており、この場合は各スレッドが一本ずつ持っている状態を表している。食事をしているスレッドがある場合は 5 つの値のうち 2 つは同じ値になっている。その次の 5 つの値が各スレッドの状態を示している。flag 0 の後の部分が stateDB によって同じ状態のものを探しハッシュ値で抽象化している部分となり、最後にこの処理を行っていたスレッド番号となっている。

モデル検査の実行結果を一部抜粋

```
not <> eating
00000000 01000000 02000000 03000000 04000000
01000000 01000000 01000000 01000000 01000000
flag 0 0x7fb22255e090 -> 0x7fb22255e090 hash 5feba6a0 iter 5

not <> eating
00000000 01000000 02000000 03000000 04000000
01000000 01000000 01000000 01000000 01000000
flag 0 0x7fb22255e090 -> 0x7fb22255e090 hash 5feba6a0 iter 4

not <> eating
00000000 01000000 02000000 03000000 04000000
01000000 01000000 01000000 01000000 01000000
flag 0 0x7fb22255e090 -> 0x7fb22255e090 hash 5feba6a0 iter 3

not <> eating
00000000 01000000 02000000 03000000 04000000
01000000 01000000 01000000 01000000 01000000
flag 0 0x7fb22255e090 -> 0x7fb22255e090 hash 5feba6a0 iter 2

not <> eating
00000000 01000000 02000000 03000000 04000000
01000000 01000000 01000000 01000000 01000000
flag 0 0x7fb22255e090 -> 0x7fb22255e090 hash 5feba6a0 iter 1
```

13 評価

今回の実装はフラグの設定に繰り返し導出木を生成するアルゴリズムを用いてるのであまり高速な実行になっていない。CTL 検査ではなく、すべての場合を尽くすだけが目的であれば繰り返し生成は必要ない。繰り返し実行自体は状態 DB に次の状態を記録すれば避けられるがメモリ使用量は増える。

登録するメモリ領域、そして深さ探索する部分の指定は手動で行う必要がある。さらに、CTL 用のフラグ

管理もメタ計算として自分で書く必要がある。この部分を自動生成すること自体は容易だと思われる。

現在はモデル検査の並列実行および Gears OS 自体のモデル検査は行っていない。

14 今後の展開

この OS を含むモデル検査は OS の拡張性をデバイスドライバの開発などに向いていると考えられる。この場合は、デバイス自体の仕様が codeGear/dataGear で書かれている必要がある。

状態の展開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。

スレッド数がランダムで決まる、または途中でスレッドが増える例題がある場合について考える。その例題を走査するためには、スレッド の状態を memoryTree として保管する iterator をスレッドの数だけ用意する必要がある。しかし現在の Data Gear は予め生成しておいたものであり、実行中に生成をする方法がない。また生成される Data Gear は iterator であるためノーマルレベルからは呼び出せないようにしておく必要がある。このため Data Gear の自動生成 には工夫が必要となる。

モデル検査を行った際に、メモリの状態を iterator で保管している。この memoryTree を実行履歴として trace し遡ることが出来れば、展開された状態から任意の実行状態を作る事が可能であると考えらる。さらに mcMeta に対して debugger を埋め込む事によって、状態の展開から bug の発生箇所を発見し、debug することが可能であると考えられる。

Red Black Tree は平衡二分木の一種で複雑な構造ではあるが、探索、挿入、削除の最悪計算量 O(logn) となるものである。この例題のモデル検査したいと考えている。

Red Black Tree をモデル検査するためにはノードを循環構造に事によって状態を有限で表す必要があり。またノードの値を iterator で整合性の検証の仕方について考察する必要がある。

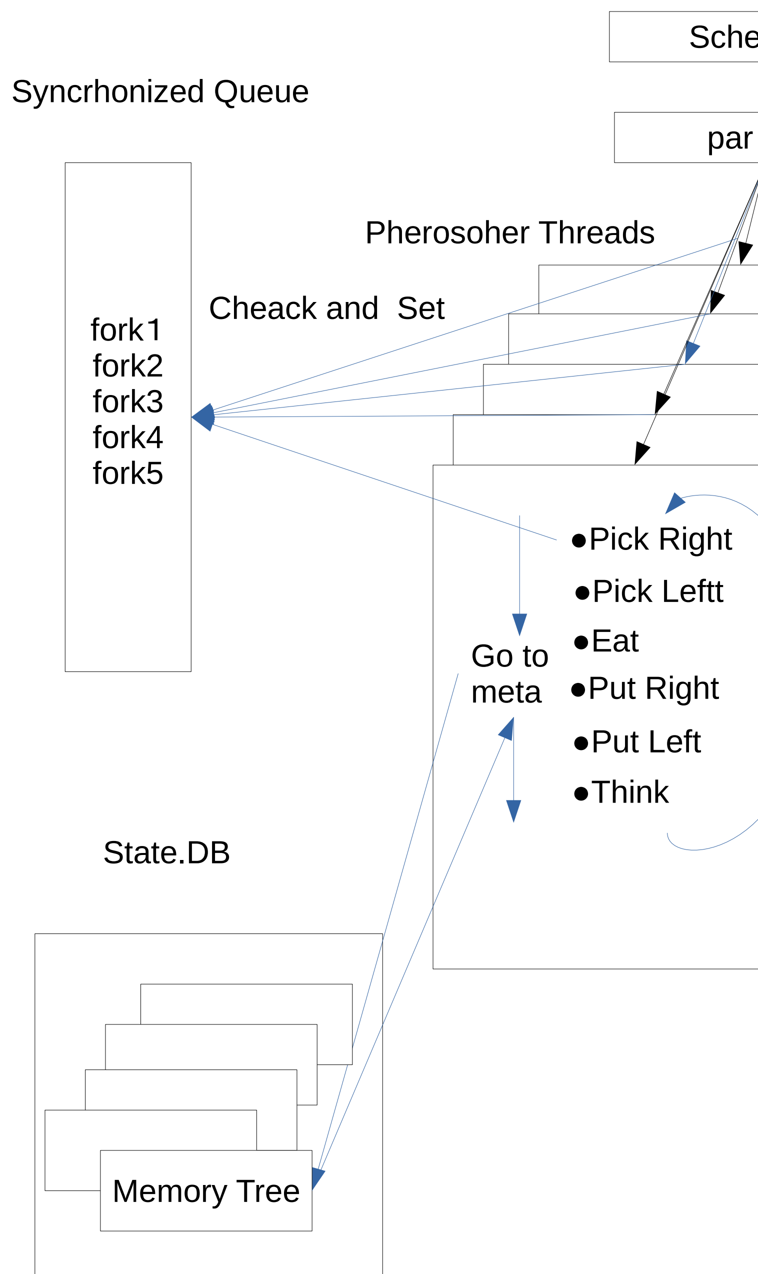

Figure 8: DPP on Gears OS