

初めてのモデル検査

正しい実装はどっち？

前編

ゆかたゆ

正しい Mutex の実装はどっち？

A

```
b[me] <- 1
while(turn != me){
    while(b[other] == 1){ }
    turn <- me
}
```

```
// -- critical section -- //
```

```
b[me] <- 0
```

B

```
b[me] <- 1
turn <- other
```

```
while (b[other] = 1
    and turn = other) {
}
```

```
// -- critical section -- //
```

```
b[me] <- 0
```

注意点

- 理論の厳密な解説ではありません
- 説明が不十分な点があります
- 今回は全部の実装は行いません

お品書き

- 形式検証とモデル検査
- 状態の列挙
- 検証
- まとめ

形式検証と モデル検査



形式検証って？

ソフトウェア，ハードウェアの**実装**や**仕様**が

「正しい」ことを機械的に検証する手法

形式検証の分類（私の偏見に基づく）

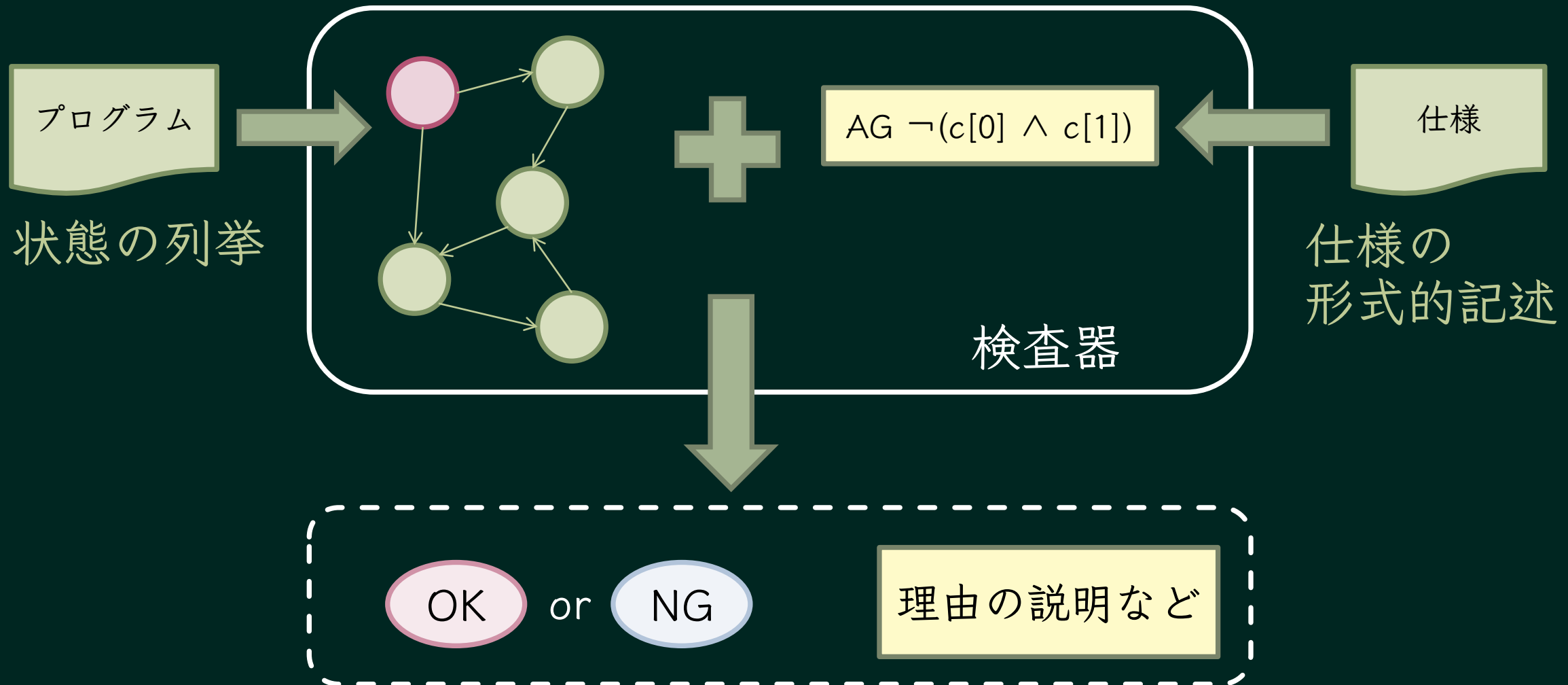
モデル検査

- 全ての状態を列挙して検査する
- 自動化が進んでいる
- 比較的単純なものが対象

定理証明

- 人間が証明を書いて機械を納得させる
- 複雑なものを扱える
- 型システムもこの一種
- Isabelle, Coq, ACL2 など

今回のモデル検査の流れ



状態の列挙

プログラミング言語の定義 (1/3)

- 入力用のプログラミング言語を考える
 - なるべく小さな物
 - 今回のアルゴリズムを表現できる必要
- 欲しい性質
 - 状態遷移図に変換しやすい

プログラミング言語の定義 (2/3)

- 以下の2つで表す

変数 = 即値

```
if( 変数 == 即値 ){  
    goto 行番号  
}else{  
    goto 行番号  
}
```

プログラミング言語の定義 (3/3)

A

```
b[me] <- 1
while(turn != me){
    while(b[other] == 1){ }

    turn <- me
}
// -- critical section -- //
b[me] <- 0
```

A'

```
0 : b[pid] = true
1 : if(turn == pid){ goto 2 }
   else{ goto 4 }
2 : if(b[1-pid] == true){ goto 2 }
   else{ goto 3 }
3 : turn = pid
4 : c[pid] = true
5 : c[pid] = false
6 : b[pid] = false
```

状態遷移図に変換

- 以下の直積を考える
 - 各変数の取りうる組み合わせ
 - 各スレッドがどこまで実行したか
- 今回は 2048 状態

後編につづく



初めてのモデル検査

正しい実装はどっち？

後編

ゆかたゆ

お品書き

- 形式検証とモデル検査

- 状態の列挙

- 検証

- まとめ

検証

仕様の記述

- 表現方法は複数存在
 - LTL (線形時相論理) 人間に扱いやすい
 - CTL (計算木論理) 計算機に扱いやすい
 - CTL* 上記2つを包含する
- 今回はCTLを用いる

CTLとは

- 命題論理の拡張
- 以下の演算子を追加 (被演算子を ϕ, ψ とする)
- この2種類は必ずセットで出現する

経路作用素

$\forall \phi \dots$ 全ての経路で

$\exists \phi \dots$ ある経路で

状態作用素

$\bigcirc \phi \dots$ 次の時点で ϕ $\psi \cup \phi \dots$ ϕ までは ψ

$\square \phi \dots$ 常に ϕ $\psi \text{ W } \phi \dots$ ϕ までは ψ

$\diamond \phi \dots$ いつか ϕ (ϕ の保証なし)

仕様の例

- $\exists \Diamond (<\text{Start}> \wedge \neg <\text{Ready}>)$
準備ができていないのにスタートしてしまうかも
- $\forall \Box (\exists \Diamond <\text{Reset}>)$
いつでもリセットを試行できる
- $\forall \Box (\forall \Diamond <\text{Input}>)$
「次の入力の受け入れ」が常に起こりうる
= いつか入力待ちが解消される

今回検査する仕様

$$\forall \square \neg (c[0] \wedge c[1])$$

どのような条件で
実行しても

スレッド0とスレッド1が同時に
クリティカルセクションに入らない

仕様の検査方法 (1/3)

- それを満たす状態の集合を再帰的に導出

- 論理式 ϕ を満たす状態の集合を $S(\phi)$ とする

$$S(\phi \vee \psi) = S(\phi) \cup S(\psi)$$

$$S(\phi \wedge \psi) = S(\phi) \cap S(\psi)$$

$$S(\neg \phi) = S(\phi)^c$$

⋮

$\forall \square$ や $\exists \square$ 等はどうする？

仕様の検査方法 (2/3)

$\forall \square$ の導出

- それぞれの状態から幅優先探索を行う
 - 到達可能な状態が全て $S(\phi)$ に含まれる
- $\Leftrightarrow S(\forall \square \phi)$ に含まれる

仕様の検査方法 (3/3)

□ の導出

- 基本的にはどこかで無限ループor行き詰まり
 - ϕ を満たさないものを削除したグラフを作る
 - 強連結成分分解を行う
 - 非自明な強連結成分に到達可能な状態を集める

仕様の検査方法 (4/3)

- 残りは読者の宿題とします

実装

- Rustで頑張りました
- 直和型がある言語だと実装しやすい

実行結果

A

- 動作しないパスが存在

```
[A]
-----
0 | b[pid] = true
1 | if(turn == pid){ goto 2 }else{ goto 4 }
2 | if(b[1-pid] == true){ goto 2 }else{ goto 3 }
3 | turn = pid
4 | c[pid] = true
5 | c[pid] = false
6 | b[pid] = false
-----
 $\forall \square \neg (c[0] \wedge c[1])$  : Error!
   $\rightarrow \{([0, 0], [false, false, false, false, false])\}$ 
```

B

- 常に正常なmutexとして動作

```
[B]
-----
0 | b[pid] = true
1 | turn = 1-pid
2 | if(b[1-pid] == true){ goto 3 }else{ goto 4 }
3 | if(turn == 1-pid){ goto 2 }else{ goto 4 }
4 | c[pid] = true
5 | c[pid] = false
6 | b[pid] = false
-----
 $\forall \square \neg (c[0] \wedge c[1])$  : OK!
```


正しい Mutex の実装はどっち？

A

```
b[me] <- 1
while(turn != me){
  while(b[other] == 1){ }
  turn <- me
}

// -- critical section -- //

b[me] <- 0
```



B

```
b[me] <- 1
turn <- other

while (b[other] = 1
  and turn = other) {
}

// -- critical section -- //

b[me] <- 0
```

Aの反例は？

Process 0

```
b[0] <- 1
while(turn != 0){
  while(b[1] == 1){ }
  turn <- 0
}
```

素通りできる

Process 1

```
b[1] <- 1
while(turn != 1){
  while(b[0] == 1){ }
```

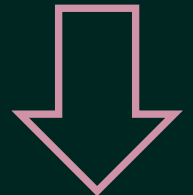
← 素通りできる

抜けられる

```
turn <- 1
```

```
}
```

クリティカル
セクションへ



補足

- 実用的な手法は他にも存在
 - SAT問題にエンコードするなど
- 複雑な問題は計算が終わらない
 - CPU+メモリの状態解析など, 状態数が非常に多い
 - 部分問題に落とし込めると嬉しい
(Mutexなど)

伝えたいこと

- 仕様を厳密に書けると安心
- 厳密にするのはコストが高い

終わり

