# システム検証論
## ー モデル検査によるシステム検証 ー

亀山幸義

筑波大学コンピュータサイエンス専攻

## 目次





## 検証と反証

- 検証=「仕様が正しい」ことを示すこと。
- 反例=仕様が成立しない具体的な実行系列。

モデル検査法の利点の 1 つ: 検証に失敗したとき反例を返す。
⇒ システム設計の初期段階でのディバッグに役立つ。
⇒ 検証のための効率的手法だけでなく、反例の発見に適した効率的手法にも意義がある。



## 有界モデル検査

有界モデル検査 (Bounded Model Checking) の概要:

- $k = 1, 2, \ldots$ に対して、長さ $k$ の反例がないかを探す。
- 有限長でのモデル検査問題は、(時相演算子のない) 命題論理式で表現可能。
- ⇒充足可能性を SAT solver で判定。

有界モデル検査の良い点、悪い点:

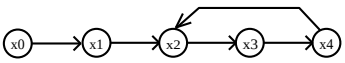
- 反例があるときは、高速。
- 最小の反例 (の 1 つ) が得られる。
- 検証にも一応使える (ただし、遅いことが多い)。



## 一本道+ループ状のモデル (lasso)

- LTL モデル検査: 反例があれば、lasso の形をした反例あり。
- CTL モデル検査: 反例は、木の形など複雑になり得る。

$(l, k)$-ループ (lasso) :

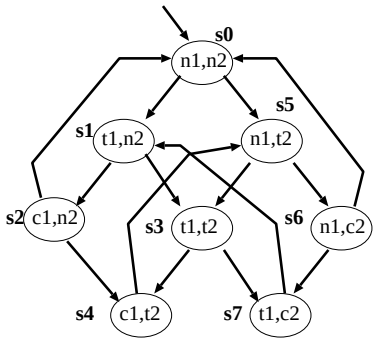



ここでは、$l = 2$ and $k = 4$ で、$\pi = x_0, x_1, x_2, x_3, x_4, x_2, x_3, x_4, \ldots$
全ての反例がこの形とは限らないが、LTL モデル検査では、$\phi$ が不成立であれば、必ず、この形の $\phi$ の反例がある。



## いつもの例

クリプケ構造 $M$:






## 有界モデル検査の手法-1

仕様を $\phi = G(t1 \Rightarrow Fc1)$ とする。

- $(l, k)$-ループ $\pi$ を用意。ここでは、$l = 1, k = 3$ とする。
- 「$(l, k)$-ループで $\neg\phi$ が真」を意味する命題:

$$\begin{aligned} & M, \pi \models \neg\phi \\ \Rightarrow\ & M, \pi \models \neg(G(t1 \Rightarrow Fc1)) \\ \Rightarrow\ & M, \pi \models F(t1 \wedge G\neg c1) \end{aligned}$$

- 従って、

$$\begin{aligned} (M, \pi \models \neg\phi) = {} & (t1(x_0) \wedge \overline{c1}(x_0) \wedge \overline{c1}(x_1) \wedge \overline{c1}(x_2) \wedge \overline{c1}(x_3)) \\ & \vee (t1(x_1) \wedge \overline{c1}(x_1) \wedge \overline{c1}(x_2) \wedge \overline{c1}(x_3)) \\ & \vee (t1(x_2) \wedge \overline{c1}(x_1) \wedge \overline{c1}(x_2) \wedge \overline{c1}(x_3)) \\ & \vee (t1(x_3) \wedge \overline{c1}(x_1) \wedge \overline{c1}(x_2) \wedge \overline{c1}(x_3)) \end{aligned}$$

ただし、$t1(x_0)$ は「状態 $x_0$ で $t1$ が真」



## 有界モデル検査の手法-2

$t1(x)$ の具体的な表現:

- モデル $M$ では、$(x = s1) \vee (x = s3) \vee (x = s7)$.

$\overline{c1}(x)$ の具体的な表現:

- モデル $M$ では、$\neg((x = s2) \vee (x = s4))$.

## 有界モデル検査の手法-3

$(l, k)$ ループが、「初期状態から始まり、パスになっている」ことを意味する命題:

$$W(\pi) = (x_0 \in S_0) \wedge (x_0 \to x_1) \wedge (x_1 \to x_2) \wedge (x_2 \to x_3) \wedge (x_3 \to x_1)$$

具体的には、

$x \in S_0$ は $x = s0$.
$x \to y$ は $(x = s0 \wedge (y = s1 \vee y = s5))$
$\vee (x = s1 \wedge (y = s2 \vee y = s3))$
$\vee (x = s2 \wedge (y = s0 \vee y = s4)) \vee (x = s3 \wedge (y = s4 \vee y = s7))$
$\vee (x = s4 \wedge y = s5) \vee (x = s5 \wedge (y = s3 \vee y = s6))$
$\vee (x = s6 \wedge (y = s0 \vee y = s7)) \vee (x = s7 \wedge y = s1)$



## 有界モデル検査の手法-4

$(l, k)$-ループが反例であることを意味する命題:

$$F(l, k) = (M, \pi \models \neg\phi) \wedge W(\pi)$$

これは、状態を表す変数 $(x_0, x_1, x_2, x_3$ など) を含む命題である。
「$(l, k)$-ループの形の反例が存在する」ことと、「ある $x_0, x_1, \ldots, x_k$ に対して、命題 $F(l, k)$ が充足可能」が同値。



## 有界モデル検査のアルゴリズム

- クリプケ構造 $M$ と LTL 論理式 $\phi$ が与えられる。
- $k = 0, 1, 2, \ldots, |S|$ に対して以下を繰返す。
  - 命題 $\bigvee_{l=0}^{k} F(l, k)$ の充足可能性を判定する。
  - 充足可能であれば、長さ $k$ の反例となり、アルゴリズムを終了する。
  - 充足不能であれば、次の繰返しへ。
- $k = |S|$ となっても充足可能な場合がなければ、反例はなく、$\phi$ は真である。

SAT solver は、与えられた CNF が充足可能な場合に、充足する割当て (原子命題への真理値の割当て) を返すので、それを利用して、$\phi$ の反例を具体的に構成できる。
得られた命題を、SAT solver にかけるために、structure-preserving CNF conversion (初日の授業参照) を利用。



## 有界モデル検査の正しさ

**Theorem**
*LTL* モデル検査では、反例が存在すれば、$0 \le l \le k \le |S|$ である $l, k$ に対して、$(l, k)$-ループの形の反例が存在する。

**Theorem**
*LTL* モデル検査で、$(l, k)$-ループが $\phi$ の反例であることと、命題 $F(l, k)$ が充足可能なことは同値である。



## SAT-solver は速い

| k | CNF gen. red. (sec) | CNF gen. dir. (sec) | SAT solving red. (sec) | SAT solving direct (sec) | # of clauses 1-bit | # of clauses red. | # of clauses direct | result |
|---|---|---|---|---|---|---|---|---|
| 4 | 20 | 1 | 3.2 | 1.8 | 59432 | 61472 | 59783 | unsat. |
| 6 | 38 | 1 | 4.1 | 24.1 | 85704 | 88560 | 86157 | unsat. |
| 8 | 74 | 3 | 9.8 | 24.9 | 112000 | 115672 | 112555 | unsat. |
| 10 | 95 | 5 | 20.5 | 32.1 | 142808 | 138977 | 138320 | unsat. |
| 12 | 124 | 5 | 25.9 | 46.9 | 164664 | 169968 | 165423 | unsat. |
| 14 | 157 | 7 | 35.0 | 81.7 | 191032 | 197152 | 191893 | unsat. |
| 16 | 204 | 9 | 22.9 | 0.2 | 217424 | 224360 | 218387 | sat. |
| 18 | 289 | 13 | 24.1 | 29.8 | 243840 | 251592 | 244905 | sat. |

MiniSAT [Een and Sorensen] の実行速度 (case n=32, m=128)



## 有界モデル検査のまとめ

- 通常のモデル検査とは全く異なる原理による検証手法。
- 実際には、検証よりも、反例の発見に重点。
- lasso 形の反例の中で、長さが最も短いもの。
- SAT solver を使った高速判定アルゴリズムに帰着。



## 目次





## 状態爆発問題

状態爆発問題 (State Explosion Problem)

- 現実に現れる問題 (検証を要するシステムやソフトウェア) はしばしば、状態数が多くなり過ぎ、モデル検査が現実的な時間内に終了しない。
  - e.g. 3 か月に一度新機種を出すときに、検証に 1 年かかっては使いものにならない。
- システムの状態数は、コンポーネント (部品) 数の指数関数。
  - 2 状態のコンポーネントが 100 個あるシステムは、$2^{100}$ の状態数を持つ。
- ⇒状態数削減が必要。

## クリプケ構造の抽象化-1

クリプケ構造の抽象化 (abstraction)：与えられたクリプケ構造 $C$ に対して、(状態数が少ない) クリプケ構造 $A$ を導出すること。

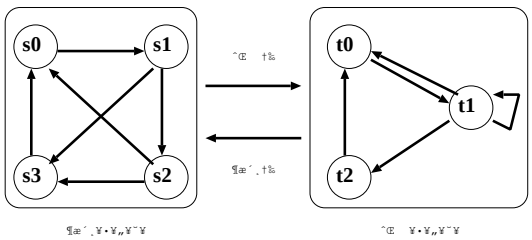



Figure: 抽象化と具体化



## クリプケ構造の抽象化-2

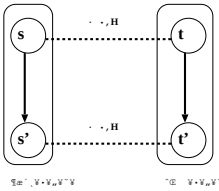
- $C$ に対するモデル検査問題を、$A$ に対するモデル検査問題に帰着したい。
- 具体モデル $C$ と抽象モデル $A$ の間に、シミュレーション関係が成立することを要求。
  - $A$ が $C$ をシミュレートする。
  - $C \leq A$.
- $C \leq A$ であり、仕様 (検証条件)$\phi$ が一定のクラスにはいれば、$A \models \phi$ ならば、$C \models \phi$ となる。(シミュレーション定理)



## シミュレーション (模倣) の定義-1

クリプケ構造 $M_i = (S^i, S^i_0, A^i, R^i, V^i)$ $(i = 1, 2)$.
ただし、$A^1 \supset A^2$ とする。
2 項関係 $H \subset S^1 \times S^2$ が以下を満たすとき、$M_1$ と $M_2$ のシミュレーション関係と言う。

- $H(s,t) \Rightarrow \forall p \in A^2.\ V^1(s,p) = V^2(t,p)$.
- $\forall s \in S^1_0.\ \exists t \in S^2_0.\ H(s,t)$.
- $\forall s, s' \in S^1. \forall t \in S^2.\ H(s,t) \wedge (s \to s') \Rightarrow \exists t' \in S^2.\ H(s',t') \wedge (t \to t')$.





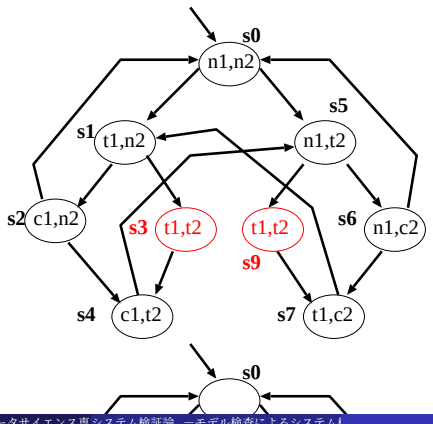

## シミュレーションの定義-2

$M_1$ と $M_2$ の間に、シミュレーション関係が存在するとき、
$M_2$ は $M_1$ はシミュレートする $(M_1 \leq M_2)$ という。



## 抽象化の例

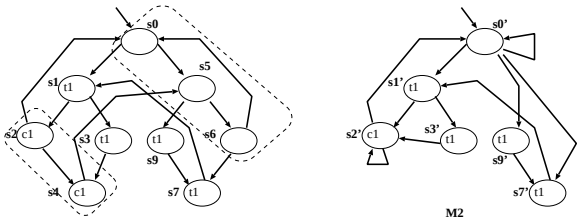
$M_1$: 相互排除の例 (改良後) で、$AG(t1 \Rightarrow AFc1)$ を検証したい。
ステップ 1. $t1$ と $c1$ 以外の原子命題を削除。






## 抽象化の例-2

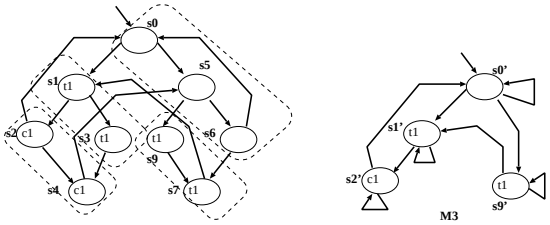
ステップ 2. $M_1$ のいくつかの状態をまとめてグループ化する。
ステップ 3. まとめた状態の間の遷移関係を定める。






## 抽象化の例-3

状態のグループ化を変えると、異なる抽象化になる。






## 特別なシミュレーション関係

シミュレーション関係 $H \subset S^1 \times S^2$ が、関数的 (functional) とは、全域的 (total) かつ、値が唯一 (univalent) であること：

$$\forall s \in S^1. \exists! t \in S^2.\ H(s,t).$$

抽象化のために使うほとんどのシミュレーションは関数的。

## シミュレーション定理

ACTL 論理式 は CTL 論理式 $\phi$ で以下を満たすもの：

- $\phi$ には E オペレータが含まれない。
- $\phi$ に含まれる否定記号は、原子命題の直前のみ。

**Theorem**

$M_1 \leq M_2$ とする。$\phi$ が *ACTL* 論理式または *LTL* 論理式で、$\phi$ に含まれる原子命題は $A^2$ に含まれ、$M_2 \models \phi$ ならば、$M_1 \models \phi$ である。

証明は、$\phi$ の構成に関する帰納法。



## 偽の反例

偽の反例 (spurious counterexample)
抽象化の程度を上げすぎると、抽象クリプケ構造では検証条件が成立せずに反例が生じるが、もとの具体クリプケ構造には対応する実行系列が存在しないことがある。これを、偽の反例という。
例: $M_3$ における反例。



## バイシミュレーション定理

クリプケ構造 $M_1$ と $M_2$ の間にバイシミュレーション関係 $H$ があるとき、$M_1 \sim M_2$ と書く。

**Theorem**

$M_1 \sim M_2$ とする。$\phi$ が *CTL* 論理式または *LTL* 論理式であれば、$M_2 \models \phi$ と $M_1 \models \phi$ は同値である。

証明は、$\phi$ に関する帰納法。

注意。$M_1 \leq M_2$ かつ $M_2 \leq M_1$ であっても、$M_1 \sim M_2$ とは限らない。

バイシミュレーションは制約が厳しいため、状態数の大幅な削減は望めないことが多い。



## 目次





## シミュレーション定理の応用

$\phi$ を ACTL 論理式 $AG(t1 \Rightarrow AFc1)$ とする。

- $M_1 \leq M_2$ かつ $M_2 \models \phi$ なので、$M_1 \models \phi$ である。
- $M_1 \leq M_3$ であるが $M_3 \models \phi$ でないので、$M_1 \models \phi$ は結論できない。
- シミュレーションは片方向なので、抽象化をし過ぎると (状態数を減らし過ぎると)、検証条件が成立しないことがある。
- 抽象モデルで、検証条件が成立しないからといって、具体モデルで検証条件が成立するかどうかはわからない。
- 抽象化の程度をコントロールする必要がある。



## バイシミュレーション (双模倣)

バイシミュレーションの定義:
クリプケ構造 $M_i = (S^i, S^i_0, A, R^i, V^i)$ $(i = 1, 2)$.
2 項関係 $H \subset S^1 \times S^2$ が以下を満たすとき、$M_1$ と $M_2$ のバイシミュレーション関係と言う。

$$H(s,t) \Rightarrow \forall p \in A.\ V^1(s,p) = V^2(t,p)$$
$$\forall s \in S^1_0.\ \exists t \in S^2_0.\ H(s,t)$$
$$\forall t \in S^2_0.\ \exists s \in S^1_0.\ H(s,t)$$
$$\forall s, s' \in S^1. \forall t \in S^2.\ H(s,t) \wedge (s \to s') \Rightarrow \exists t' \in S^2.\ H(s',t') \wedge (t \to t')$$
$$\forall t, t' \in S^2. \forall s \in S^1.\ H(s,t) \wedge (t \to t') \Rightarrow \exists s' \in S^1.\ H(s',t') \wedge (s \to s')$$



## 様々な抽象化

- データ抽象化 (data abstraction): 整数などの (無限あるいは非常に大きな有限の) データ領域を、{ 正整数、ゼロ、負整数 } などの小さなデータ領域に対応付ける。
- 述語抽象化 (predicate abstraction): 後述。
- スライシング (slicing) あるいは cone of influence reduction: プログラムの出力・検証条件に関係ある部分だけを抽出。
- 半順序抽象化 (partial order reduction): 複数のプロセスの非決定的動作の順序を抽象化。
- ...



## ソフトウェアは複雑

- システム設計に比べて、ソフトウェアの状態数は非常に多い。
  - 整数型の変数 (32 ビットとする) を 10 個持つプログラムは、$(2^{32})^{10}$ 個の状態を持つ。
  - 整数型の要素を 1000 個持つ配列があれば、$(2^{32})^{1000}$ 個の状態を持つ。
  - メモリ 1GB の PC の状態を表現するには、$2^{(2^{30})}$ 個の状態が必要。
  - 無限に大きい整数を扱えるシステム (例: Lisp の bignum) では、無限個の状態が必要。

## ソフトウェアモデル検査の技法

1990 年代後半以降、ソフトウェアへの適用が急激に進展。

- ソースコードの直接検証へ
  - SPIN モデル検査器: 広く使われている LTL モデル検査器。
  - C 言語や Java 言語ではなく、独自のプログラム言語 (PROMELA) で、プログラムをモデル化する必要がある。
  - 1997 年頃から BLAST, SLAM (C 言語に対応), Bandera/JavaPathfinder (Java 言語に対応) など、プログラム (ソースコード) を直接検証対象とするモデル検査器が開発された。
- 述語抽象化
  - $x = 0$ などの述語に基づいて抽象化。
- 反例に基づく抽象化の精密化
  - 抽象化の程度を自動的に発見・コントロールする仕組み。



```
1:Example(){
7: if (*) {                    lock(){
    do {got_lock = 0;            if (LOCK==0){
8:      if (*) {                   LOCK = 1;
9:        lock();                } else {
          got_lock++;}             ERROR
10:     if (got_lock) {          }
11:       unlock();}           }
12: } while (*)
   }                           unlock(){
2: do {lock();                   if (LOCK==1){
       old = new;                 LOCK = 0;
3:     if (*) {                  } else {
4:       unlock();                ERROR
         new++;}                  }
5: } while (new != old);     }
6: unlock();
   return;
}
```



## ロック・プログラムの検証-1

前ページの C 言語のプログラム例 [Henzinger et al. POPL 2002]
検証条件: $AG\neg ERROR$. (安全性)

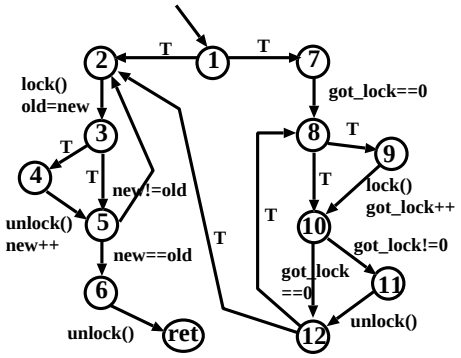



Figure: コントロールフロー・オートマトン



## ロック・プログラムの検証-2

具体クリプケ構造の 1 つの状態：

- プログラムがどの場所を実行中か (12 状態)
- 変数 got_lock の値 (整数型)
- 変数 old の値 (整数型)
- 変数 new の値 (整数型)
- 変数 LOCK の値 (整数型)

整数型の変数を $\{0,1\}$ に限定したとしても，$12*2*2*2*2 = 192$ 個の状態をもつ.



## ロック・プログラムの検証-3

具体クリプケ構造の遷移の例:

| | 遷移前 | → | 遷移後 |
|---|---|---|---|
| プログラムの場所 | 2 | | 3 |
| got_lock | 0 | | 0 |
| old | 1 | | 0 |
| new | 0 | | 0 |
| LOCK | 0 | | 1 |

コントロールフロー・オートマトンは，上記の遷移をコンパクトに表現したもの.



## ロック・プログラムの検証-4

Henzinger らのソフトウェアモデル検査アルゴリズム

- ステップ 1. 抽象モデル上でモデル検査，検証成功したら終了.
- ステップ 2. 反例が見つかったら対応する反例が具体モデル上にあるかどうか検査する.
- ステップ 3. 具体モデル上に対応する反例がない時，その情報に基づいて抽象化のレベルを落とし (精密化)．ステップ 1 を繰返す.



## ロック・プログラムの検証-5

ステップ 1-1. 抽象モデルの構築. (述語 $LOCK = 0$ と $LOCK = 1$ による抽象化).

抽象クリプケ構造の 1 つの状態：

- プログラムがどの場所を実行中か (12 状態)
- 述語 $LOCK = 0$ と $LOCK = 1$ の値 (それぞれ true/false)

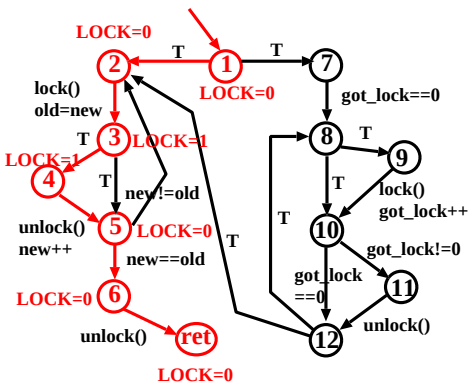
抽象クリプケ構造の 1 つの遷移：

| | 遷移前 | → | 遷移後 |
|---|---|---|---|
| プログラムの場所 | 2 | | 3 |
| $LOCK = 0$ と $LOCK = 1$ の値 | (true,false) | | (false,true) |



## ロック・プログラムの検証-6

ステップ 1-2. 抽象モデル上で，$AG\neg ERROR$ を検査.
⇒反例を発見.



## ロック・プログラムの検証-7

ステップ 2. 反例を具体モデルへ反映.
反例の実行系列を，ERROR 状態から逆方向へ「実行」する.

- 場所 ERROR: $true$.
- 場所 6: $LOCK = 0$.
- 場所 5: $LOCK = 0 \wedge new = old$.
- 場所 4: $LOCK = 1 \wedge new + 1 = old$.
- 場所 3: $LOCK = 1 \wedge new + 1 = old$.
- 場所 2: $LOCK = 0 \wedge new + 1 = new$.

場所 2 において，充足不能な論理式が生成された．(偽の反例)
さらに，充足不能性の証明から，$new = old$ が重要な述語であることを判定.



## ロック・プログラムの検証-8

- ステップ 3. 偽の反例から得られた述語を追加して，ステップ 1 へ.
  - 述語 $LOCK = 0$, $LOCK = 1$ に，$new = old$ を追加して，ステップ 1 へ.
- ステップ 1'. 抽象モデルを構築して，$AG\neg ERROR$ を検査すると，反例が得られる.
- ステップ 2'. 反例を具体モデルに反映.
- ステップ 3'. 述語 $got\_lock = 0$ を獲得.
- ステップ 1''. 4 つの述語のもとで抽象モデルを構築すると，$AG\neg ERROR$ の検証に成功.



## ロック・プログラムの検証-9

この検証アルゴリズムに必要なもの.

- ソースコードの直接検証.
- 述語抽象化
- モデル検査.
- 反例を具体モデルで逆順に実行.
- 一階述語論理式の充足可能性判定.
  - 最弱事前条件 (逐次プログラムの検証手法)
  - 一階述語論理の定理証明器
- 充足不能な一階述語論理式から,「重要な述語」を抽出する方法.



## ロック・プログラムの検証-10

この手法でできること

- C 言語のソースコードのまま検証.
- 検証条件 (仕様) は，単純な安全性 ($AGp$).
- たとえば,「デッドロックしない」,「閉じたファイルに書き込もうとしない」等.

この手法の限界:

- 検証条件 (仕様) が単純なもののみ.
- アルゴリズムが停止しないことがある.
- 定理証明器は完全ではない.



## 論理学との関連

補間定理 (Interpolation Theorem) [Craig 1957].

**Theorem**

一階述語論理式 $A \Rightarrow C$ が証明可能なとき，以下を満たす一階述語論理式 $B$ *(interpolant)* が存在する.

- $A \Rightarrow B$ が証明可能。
- $B \Rightarrow C$ が証明可能。
- $B$ に含まれる自由変数は，$A$ にも $C$ にも含まれる。
- $B$ に含まれる述語記号は，$A$ にも $C$ にも含まれる。

偽反例から，新しい述語を抽出するとき，補間定理を使って，探索範囲を狭めることができる.



## ソフトウェアモデル検査ツール

- SPIN
  - 古典的な LTL モデル検査器．PROMELA 言語でソフトウェアを記述.
- BLAST:
  - California 大学 Berkeley 校で開発，C 言語に対するもの．今回紹介した Lazy Abstraction など.
- SLAM:
  - Microsoft が開発，C 言語に対するもの．Windows に組み込むデバイスドライバは SLAM の検査を受けなければならない.
- Bandera/JavaPathfinder:
  - NASA が開発，Java に対するもの.



## 目次





## モデル検査の適用分野

- 成功した分野
  - システムの設計検証: 例. 鉄道の信号システム
  - ハードウェア検証: 例. CPU
  - 通信プロトコル検証: 例. IEEE Futurebus+
- ある程度成功した分野
  - 単純な実時間システムの検証: UPPAAL モデル検査器
  - ソフトウェアの簡単な性質の検証: 例. Microsoft SLAM によるデバイスドライバ検証
- まだまだこれからの分野
  - 連続的な実時間システムの検証
  - 多値モデル検査
  - 本格的なソフトウェア検証
  - 確率モデル検査

## モデル検査ツール

- SMV, NuSMV: CTL に基づく (最近は LTL 有界モデル検査へも対応)
- SPIN: LTL に基づく
- UPPAAL: 実時間モデル検査
- BLAST, SLAM, Bandera/Java Pathfinder ···: ソフトウェアモデル検査
- (enumerous model checkers ...)

無料で使えるものが多い。



## この授業でやったこと

- モデル検査の基礎概念 (CTL と LTL, クリプケ構造意味論)
- モデル検査アルゴリズム
  - CTL モデル検査アルゴリズムの詳細
  - LTL モデル検査アルゴリズム (Büchi オートマトン上の判定問題への帰着)
  - 有界モデル検査アルゴリズム (SAT solving への帰着)
- モデル検査器の利用
  - NuSMV2 ツール
  - 例題 (相互排除、制約問題の解の探索)
- 命題論理に関するアルゴリズム (CNF への変換と SAT solver)
- 抽象化の基礎とソフトウェアモデル検査
- おまけ: アルゴリズムにおける計算量の視点
- おまけ: 研究の紹介



## まとめ

モデル検査法:

- システム検証・ソフトウェア検証の手法の 1 つで今日、最も成功しているもの。
- 論理に基盤を置いた手法、アルゴリズムの正当性の保証。
- 極めて幅広い応用: 「使えるようになる」だけでも有用。

様々な理論・技術の利用:

- 定理証明器 (SAT-solver, 一階述語論理定理証明)
- グラフ理論のアルゴリズム (強連結成分の計算)
- Büchi オートマトンの理論
- 論理学的知見 (Craig 補間定理など)
- プログラム言語の意味論・解析手法



## 目次





## 今回カバーできなかった重要な話題

- 記号モデル検査アルゴリズム (OBDD, Ordered Binary Decision Diagram に基づく記号モデル検査)
- 様々な抽象化技法