「ソフトウェア工学」

(3) 形式手法

東大理学部情報科学科講義

石川　冬樹（本務先：国立情報学研究所）
f-ishikawa@nii.ac.jp / @fyufyu
http://research.nii.ac.jp/~f-ishikawa/

大学共同利用機関法人 情報・システム研究機構
国立情報学研究所
National Institute of Informatics

目次

- V&Vと形式手法
- 逐次型プログラム検証の理論（概観）
- 形式仕様記述
- モデル検査

V&V

- Verification（検証）

  Are you building the things right?

  - 定めた正当性（Correctness）の基準を満たしているか？
  - 工程・成果物ごとに考えることが多い

- Validation（妥当性確認）

  Are you building the right things?

  - 作っているものの妥当性（Validity）に問題ないか？
  - 全体評価として考えることが多い

➡ V&V と区別しつつ，品質評価の活動を総称する

V&Vの受け止め方

- Validationは顧客やユーザの真のゴールに関する究極の問い
  - 受け入れテスト（次回）や使用アンケートなどで評価するが，
    不確かでもあり問い続けるもの
- Verificationは，妥当なものを作るために立てた
  （サブ）目標をしっかりと果たすことを確認
  - 形式検証（今回）や，テスト（次回）の一部は，主にここに寄与
  - 副次的にValidationにもつながる

※ あまり気にしないで用語を使っている場もある

モデルの厳密さ・表現力

- ここまで扱ったモデルは「スケッチ」的
  - 記述（図を含む）の文法は定義されていた
  - ➡ 同じ図形が場所により違うものを指すようなことはない
  - 意味論は必ずしも厳密に与えられていないことも
  - ➡ 複数の解釈がなされうる（UMLの古い版はそうだった）
  - 検証のための必要な情報が十分に含まれていないことも
  - ➡ 典型的には，振る舞いの定義において，引数と戻り値の型だけ正確な記法で与えられ，機能は自然言語で定義

形式手法

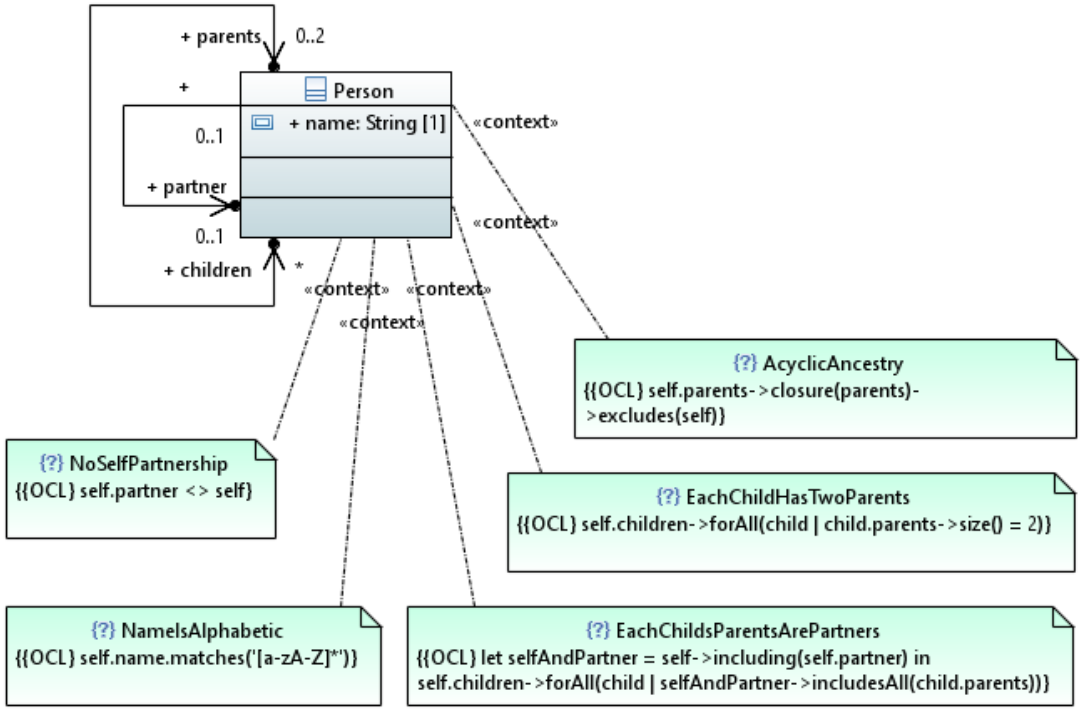
- 形式手法（Formal Methods）
  - 数理論理学に基づき品質の高いソフトウェアを効率よく開発するための科学的・系統的アプローチの「総称」
  - 文法と意味論が数学的に定まったモデルを活用
  - 厳密な記述により，曖昧さや思い込み・誤解を排除
  - 数学的な（特に自動的な）分析・検証を行い，品質を向上

⇨ 特に開発早期において問題を除去し，手戻りを防ぐ
（プログラムに対する強力な「形式検証」を考えることもある）

OCL

OCL (Object Constraint Language)

- UMLにおいて制約を記述するための形式言語
- 以下はクラス図における構造制約追記の例



[https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FOCLExamplesforUML.html] より引用

目次

- V&Vと形式手法
- 逐次型プログラム検証の理論（概観）
- 形式仕様記述
- モデル検査

プログラムの正当性

- 正当性（Correctness）は仕様に対して相対的に定まる
- 手続き型プログラムの場合，以下二つの条件を与える
  - 事前条件 （Precondition）：プログラムを作る側にとっては仮定であり，プログラムを呼び出す側が守るべき制約
  - 事後条件（Postcondition）：プログラムを作る側が保証すべき制約であり，プログラムを呼び出す側にとっては仮定
  - プログラムと合わせてHoare Tripleと呼ばれる

与えられた事前条件 A を満たすときに，
該当プログラム P を実行すると，
事後条件 B を満たすか？

直感的な例（１）

- Swap

```
x = A ∧ y = B
```

（便宜上の）事前条件

```
t = x;

x = y;

y = t;
```

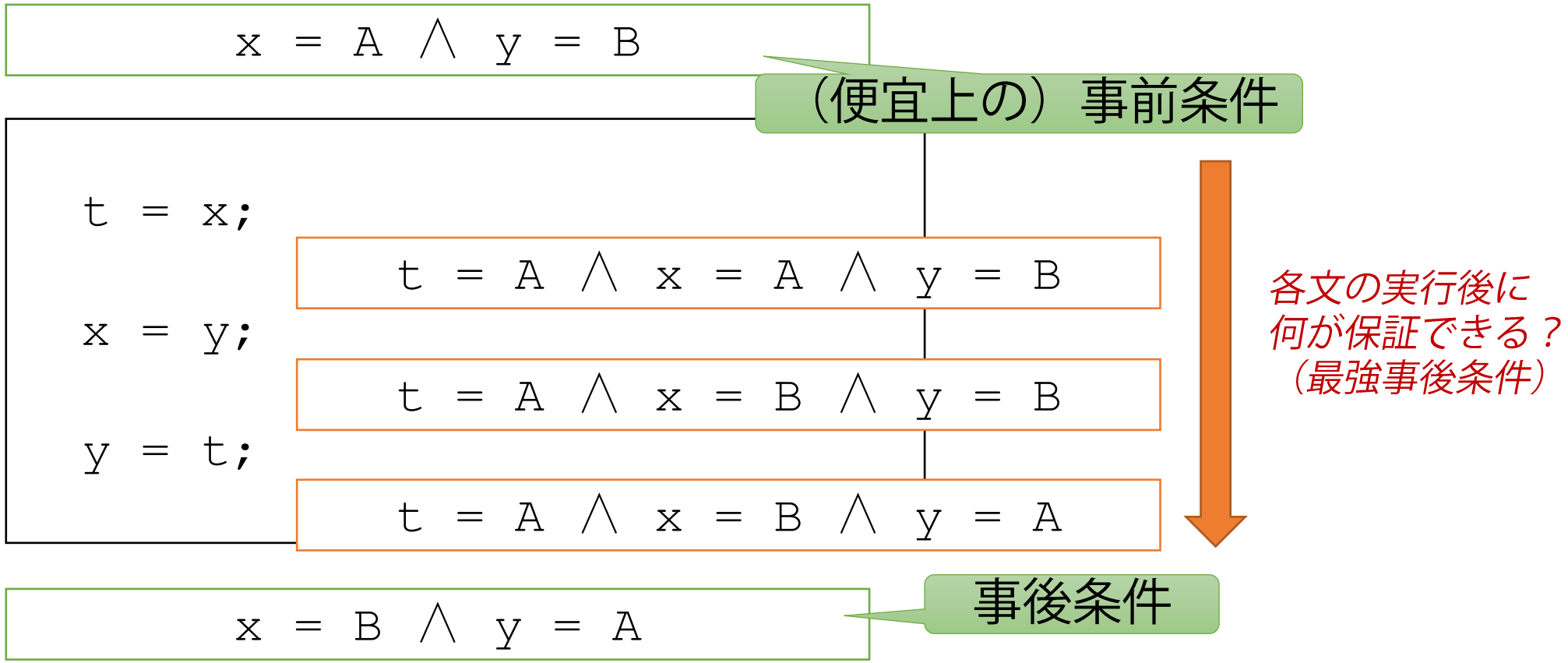
？

```
x = B ∧ y = A
```

事後条件

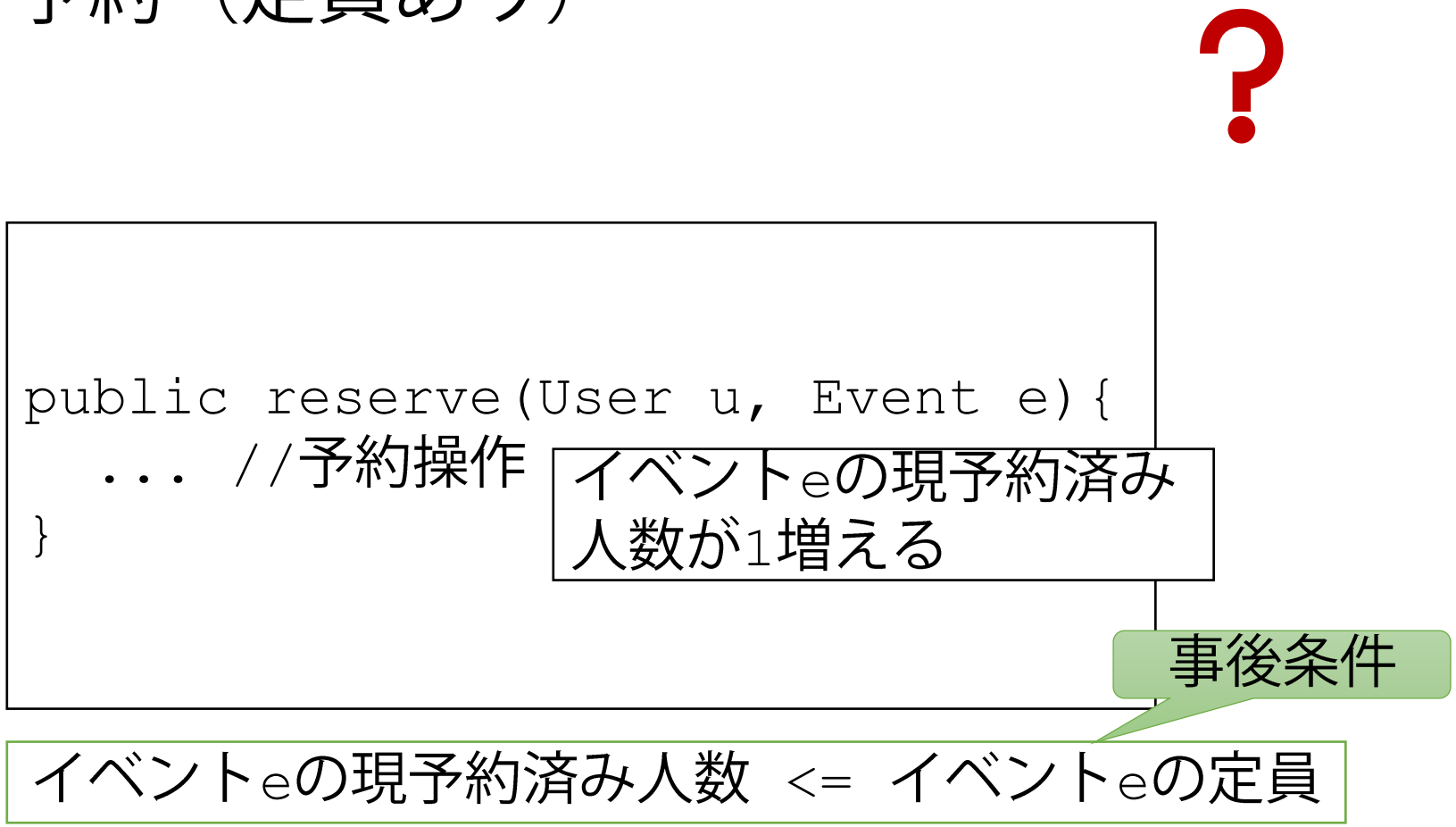
プログラムに対する証明：直感的な例 (1)

Swap



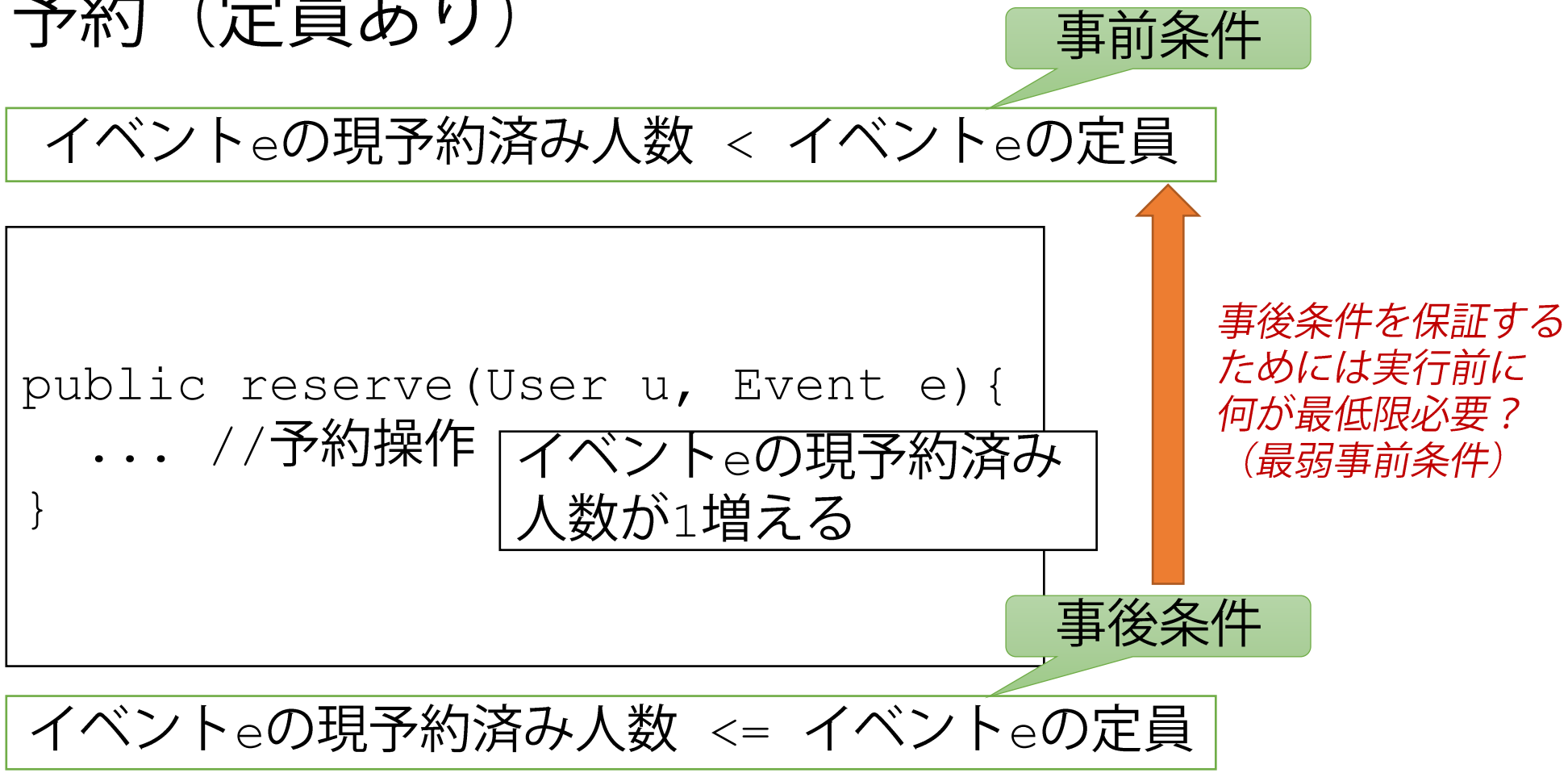
プログラムに対する証明：直感的な例 (2)

- イベント予約（定員あり）



プログラムに対する証明：直感的な例 (2)

- イベント予約（定員あり）



プログラムに対する証明：直感的な例 (3)

- 配列内の値の逐次探索
  - 事後条件の一部：値が見つからなかった場合に負の値が返る

```
public static int search(int a[], int key) {
  int i = 0;
  while(i < a.length) {
    if(a[i] == key) {
      return i;
    }
    i++;
  }
  return -1;
}
```

実行後の時点で以下が成立するには？

```
¥result<0 ==>
  ¥forall int i; 0<=i && i<a.length;
    a[i]!=key;
```

プログラムに対する証明：直感的な例 (3)

- ループの場合，一文の前後変化を考えるのが困難

```
public static int search(int a[], int key) {
  int i = 0;
  while(i < a.length) {
    if(a[i] == key) {
      return i;
    }
    i++;
  }
  return -1;
}
```

?

return文の前では以下が成立すべき

```
-1<0 ==>
  ¥forall int i; 0<=i && i<a.length;
    a[i]!=key;
```

実行後の時点で以下が成立するには？

```
¥result<0 ==>
  ¥forall int i; 0<=i && i<a.length;
    a[i]!=key;
```

プログラムに対する証明：直感的な例 (3)

- ループの場合，一文の前後変化を考えるのが困難
  - 帰納法にて「何周してもある条件が維持される」を考える

```
public static int search(int a[], int key) {
   int i = 0;
   // ループ不変条件 ¥forall int j; 0<=j && j<i; a[j]!=key;
   while(i < a.length) {
      if(a[i] == key) {
         return i;
      }
      i++;
   }
   return -1;
}
```

ループ前には
この条件が成立する？

ループを何周しても
この条件は維持される？

上を示す前提で，
ループを抜けたら
この条件が満たされると
仮定してよい

今回は「すでに調べた範囲には探したい値は
なかった」ことがループの正当性の根幹

プログラムに対する証明：直感的な例 (3)

- この例の場合3つの問題に分かれる

```
public static int search(int a[], i
  int i = 0;
  // ループ不変条件 ¥forall int j; 0<=j && j<i; a[j]!=key;
  while(i < a.length) {
    if(a[i] == key) {
      return i;
    }
    i++;
  }
  return -1;
}
```

前提（事前条件）：
　メソッド全体の事前条件
欲しい結論（事後条件）：
　ループ不変条件

前提（事前条件）：
　ループ不変条件 && (i<a.length)
欲しい結論（事後条件）：
　ループ不変条件

前提（事前条件）：
　ループ不変条件 && !(i<a.length)
欲しい結論（事後条件）：
　メソッド全体の事後条件

プログラムに対する証明：直感的な例 (3)

- ここまでくれば人間には自明

```
public static int search(int a[], int key) {
  int i = 0;
  // ループ不変条件 ¥forall int j; 0<=j && j<i; a[j]!=key;
  while(i < a.length) {
    if(a[i] == key) {
      return i;
    }
    i++;
  }
  return -1;
}
```

（メソッド全体の事前条件はなし）

欲しい結論（事後条件）：
「配列aのi番目より前にkeyはない」

前提（事前条件）：
「配列aのi番目より前にkeyはない」
&& (i<a.length)
欲しい結論（事後条件）：
「配列aのi番目より前にkeyはない」

前提（事前条件）：
「配列aのi番目より前にkeyはない」
&& !(i<a.length)
欲しい結論（事後条件）：
「配列aのどこにもkeyはない」

プログラムに対する証明：直感的な例 (4)

- ところで，無限ループすることはないか？

```
public static int search(int a[], int key) {
   int i = 0;
   while(i < a.length) {
      if(a[i] == key) {
         return i;
      }
      i++;
   }
   return -1;
}
```

ループの「進捗」を表す変数値として
`a.length - i` を考える

- この値は必ず0以上の整数
- ループにおいて必ず減少

よってループが無限に続くことはない

ホーア論理・最弱事前条件

- 以上の直感を扱う数学的体系
- ホーア論理（Hoare Logic）
  - 「事前条件・プログラム・事後条件」の3点セットに対して，証明を行うための公理と推論規則を定義
- 最弱事前条件（Weakest Precondition）
  - 「プログラム・事後条件から，必要最低限の事前条件を求める」方が扱いやすいので，そういう「計算」だと見なすことが多い

プログラム検証のツール

- ここまでの基礎理論を直接用いてプログラムを検証する
  - 静的解析（Static Analysis）に分類される方法：
    プログラムコードを動かさずに分析・検証
- 言語・ツールの例
  - C言語向け： ACSL/Frama-C [https://frama-c.com/]
  - Java言語向け： JML/OpenJML [https://www.openjml.org/]

プログラム仕様記述の例 (1)

銀行口座（JML）

```
public class BankAccount {

  private /*@ spec_public @*/ int balance;
  private /*@ spec_public @*/ static int MIN_BALANCE = 0;

  //@ public invariant balance >= MIN_BALANCE;

  //@ requires amount >= 0;
  //@ requires amount <= balance - MIN_BALANCE;
  //@ ensures balance == ¥old(balance) - amount;
  //@ signals (Exception) amount > balance - MIN_BALANCE;
  public void withdraw(int amount) throws Exception{
    if (balance - amount < MIN_BALANCE) throw new Exception();
    balance = balance - amount;
  }

}
```

内部変数だが仕様の記述のためには
外部からも存在を理解する必要あり

不変条件

事前条件

事後条件
（例外発生も含む）

プログラム仕様記述の例 (2)

■二分探索（JML）

```
//@ requires a != null;
//@ requires ¥forall int i; 0 <= i && i < a.length - 1; (¥forall int j; i < j && j< a.length; a[i] < a[j]);
// requires ¥forall int i; 0 <= i && i < a.length - 1; a[i] < a[i+1];
//@ ensures ¥result >= 0 ==> ¥result < a.length && a[¥result] == key;
//@ ensures ¥result < 0 ==> (¥forall int i; 0 <= i && i < a.length; a[i] != key);
public static int binarySearch(int a[], int key) {
    int low = 0;
    int high = a.length;
    //@ maintaining 0 <= low && low <= a.length && 0 <= high && high <= a.length;
    //@ maintaining (¥forall int i; 0 <= i && i < low; a[i] < key);
    //@ maintaining (¥forall int i; high <= i && i < a.length; a[i] > key);
    //@ decreases high - low;
    while (low < high) {
        int mid = low + (high - low) / 2;
        int midVal = a[mid];
        if (key < midVal) { high = mid; }
        else if (midVal < key) { low = mid + 1; }
        else { return mid; // key found}
    }
    return -low - 1; // key not found.
}
```

おまけ：Infer

- Separation Logic（分離論理）
  - ホーア論理の拡張で，変数（ポインタ）同士の指す先が異なるという情報を維持して推論
  - 例：「f1 と f2 は異なるFileオブジェクトを指しているため，f1をcloseしても，f2はcloseされない」
- Infer

[https://fbinfer.com/]

[https://research.fb.com/publications/moving-fast-with-software-verification/]

  - Separation Logicをベースとした静的解析ツール
  - 研究からのビジネス化，その後Facebookが買収，オープンソース化しつつ，モバイルアプリ開発にて利用

目次

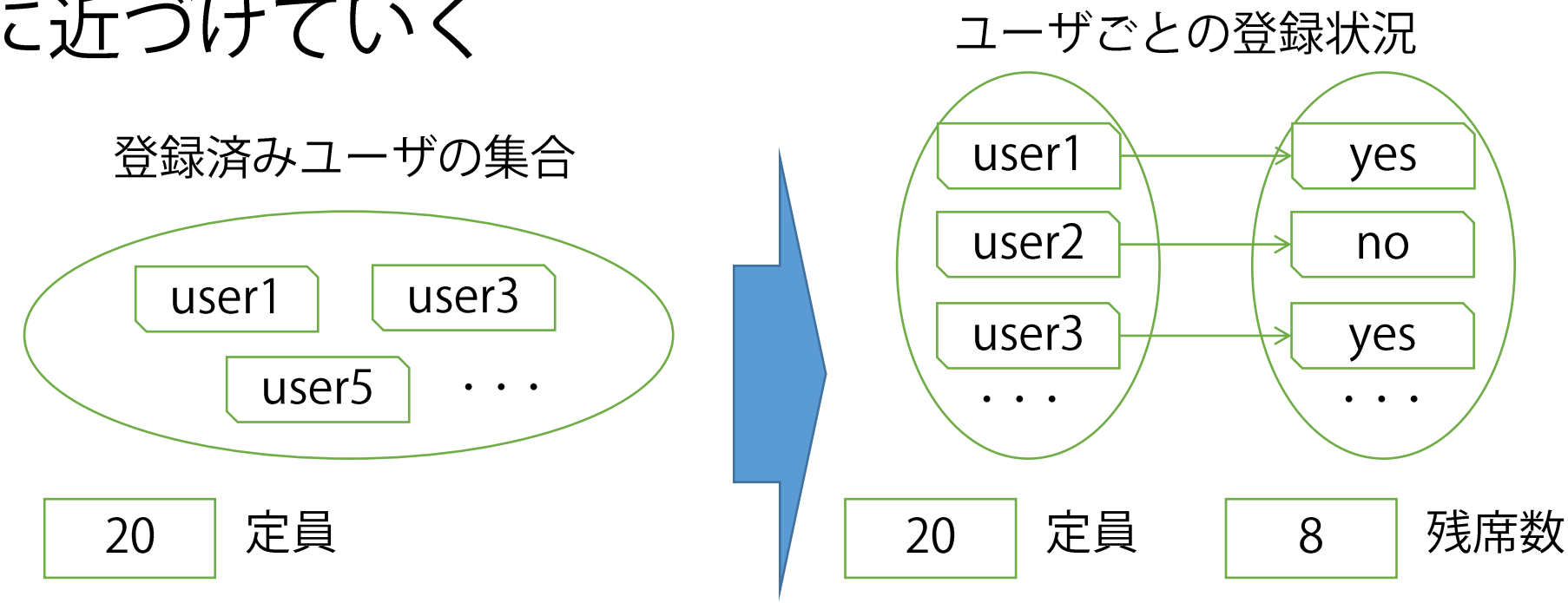
- V&Vと形式手法
- 逐次型プログラム検証の理論（概観）
- 形式仕様記述
- モデル検査

形式仕様記述

- 形式仕様記述（Formal Specification）
  - 機能仕様や設計を形式的に記述，検証するアプローチ
  - 事前条件・事後条件に関するロジックは，プログラムに至らない仕様の時点で存在するもので，その時点で誤りを排除すべき
  - 機能仕様全体を表せるよう表現力が高く，実装詳細を捨象するためプログラミング言語よりも抽象的な言語を用いることが多い
  - VDM, B-Method, Event-B, Alloy, CafeOBJ, Maude など

例題

- イベント管理システム（簡易版）
  - 固定の（1個の）イベントにユーザが参加登録する
  - 定員あり
- 実装に近づけていく



Bメソッドにおける記述例 (1-1)

```
MACHINE
   EventManager(capacity)

CONSTRAINTS
   capacity : NAT

SETS
   USERS

VARIABLES
   registered_users

INVARIANT
   registered_users : POW(USERS) &
   card(registered_users) <= capacity
```

定員はモジュールのパラメーターとしている

パラメーターに関する条件（今回は型宣言のみ）

現時点では詳細を決めない型（集合）を宣言（後に { 0, ⋯, 255 } など定める）

NATはすべての自然数の集合，　:は集合に含まれること（∈）（これも後で，計算機上の整数表現に変える）

Bメソッドにおける記述例 (1-2)

```
VARIABLES
   registered_users

INVARIANT
   registered_users : POW(USERS) &
   card(registered_users) <= capacity

INITIALISATION
   registered_users := {}

OPERATIONS
   register(user) =
   PRE user : USERS & user /: registered_users &
      card(registered_users) <= capacity - 1
   THEN registered_users := registered_users ¥/ {user}
   END

END
```

変数宣言

このシステムが常に満たすべき不変条件（型定義含む）

初期化

操作定義

/ は not を表す（ /: は ∈ ）

事前条件と代入（ ¥/ は union）

Bメソッドにおける記述例 (2-1)

```
REFINEMENT
  EventManager_r(capacity)
REFINES
  EventManager

SETS
  STATUS = {yes, no}

VARIABLES
    user_status, available_num

INVARIANT
    user_status : USERS --> STATUS & available_num : NAT &
    available_num = capacity - card(user_status~[{yes}]) &
    user_status~[{yes}] = registered_users
```

先のモジュールを詳細化

列挙型のように用いる集合を宣言

--> は全域関数

元のモデルにおける変数と，リファインメントにおける変数との関係を表すリンク不変条件

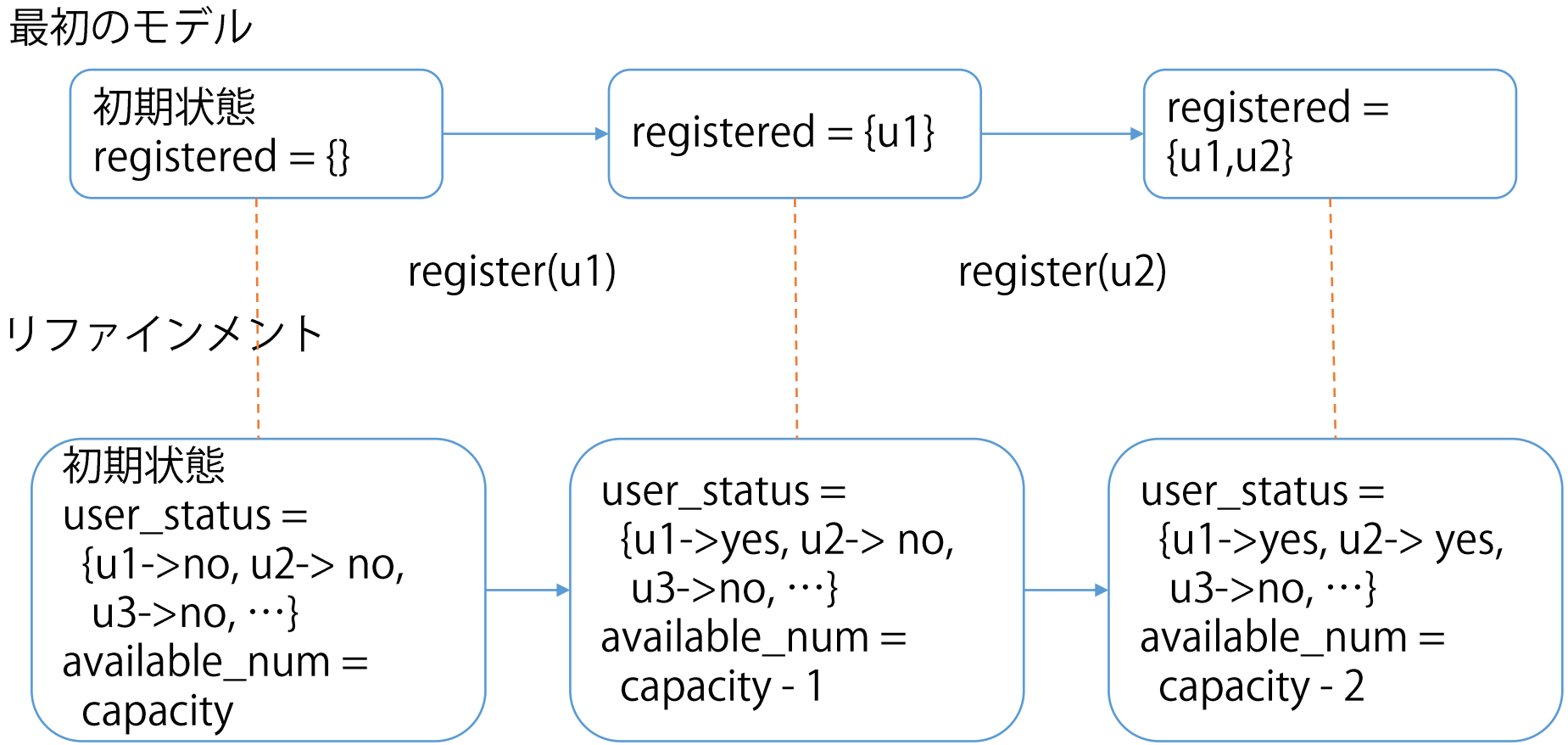
Bメソッドにおける記述例 (2-2)

```
INITIALISATION
  user_status, available_num := USERS * {no}, capacity


OPERATIONS
  register(user) =
  PRE user : USERS & user_status(user) = no & available_num >= 1
  THEN user_status, available_num :=
          user_status <+ {user |-> yes},   available_num - 1
  END

END
```

元のモデルとインターフェースを同じに保っている

<+ は関係における上書き

Bメソッド：リファインメントによる対応関係



（１対１対応でなくてもよく，リファインメントにより非決定性がなくなっていてもよいが，特定の関係（リンク不変条件）を満たす対応がとれるようになっている）

Bメソッドにおける検証

- 定理証明（Theorem Proving）：

  与えられた公理と推論規則のみを用いての結論を導出

  - 各モデルにおいて，初期化により不変条件が成立し，
    各操作は成立している不変条件を壊さないこと
  - 各リファインメントにおいて，各操作の事前条件は，
    リファインメント前よりも強くならないこと
    （呼び出し条件を最初の定義より厳しくしないこと）

➧ Correctness by Construction

  - 最初の仕様モデルに対して正当性を維持しつつコードを得る

Bメソッドの産業事例

- 鉄道事例が有名
  - パリの空港シャトルや地下鉄ホームドア（世界各地に展開）
  - 周期実行されるコア部分のプログラム
- 空港シャトルでの統計
  - 15万行超のコードに対して，4万超の証明
  - うち97%は自動（証明に 4.6 人月）
  - 工数の半数以上は最初の仕様形式モデルに：
    通常とは大きく異なるフロントローディング

[J. R. Abrial, Formal Methods in Industry: Achievements, Problems, Future, 2006]

他の手法の例：VDM

- VDM：ライトウェイトな手法
  - 記述内容はBメソッドに近いが，見た目が C/C++/Java に近い
  - 形式モデルに対して，インタプリタを用いてテストを行う（今のツールでは定理証明はほぼ考えていない）
  - ➡ 曖昧さの排除，系統的なテストによる検証だけでも十分な効果
- モバイルFeliCaチップの事例が有名
  - 外注もされるチップの外部仕様（各コマンドの振る舞い定義）をVDMで記述，検証

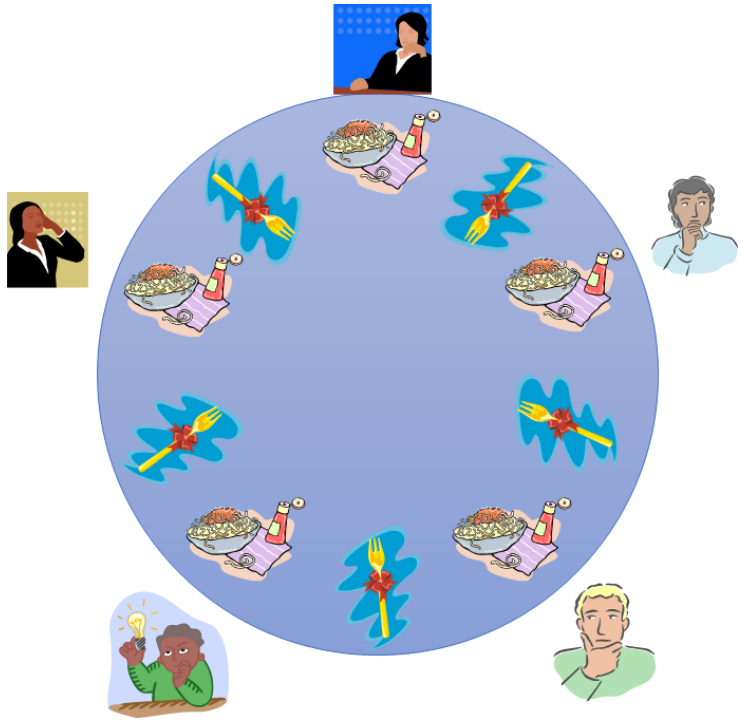
[栗田, 携帯電話組み込み用モバイルFelica ICチップ開発における形式仕様記述手法の適用, 2008]
[Kurita et al., Practices for Formal Models as Documents: Evolution of VDM Application to ``Mobile FeliCa" IC Chip Firmware, 2015]

目次

- V&Vと形式手法
- 逐次型プログラム検証の理論（概観）
- 形式仕様記述
- モデル検査

並行システムにおける正当性

- 特定の実行順序・タイミングでのみ起きる誤りが課題
  - 例：食事する哲学者の問題
    - 哲学者が円卓に座っている
    - 各自の間には1つフォークがある
    - 両手でフォークをとると食事ができる
    - 哲学者間の会話はない
    - 行動は同時には起きない
  - 「右手でとる，左手でとる，
    食事する，左を下ろす，右を下ろす」
    → デッドロックの可能性



モデル検査

- モデル検査（Model Checking）：

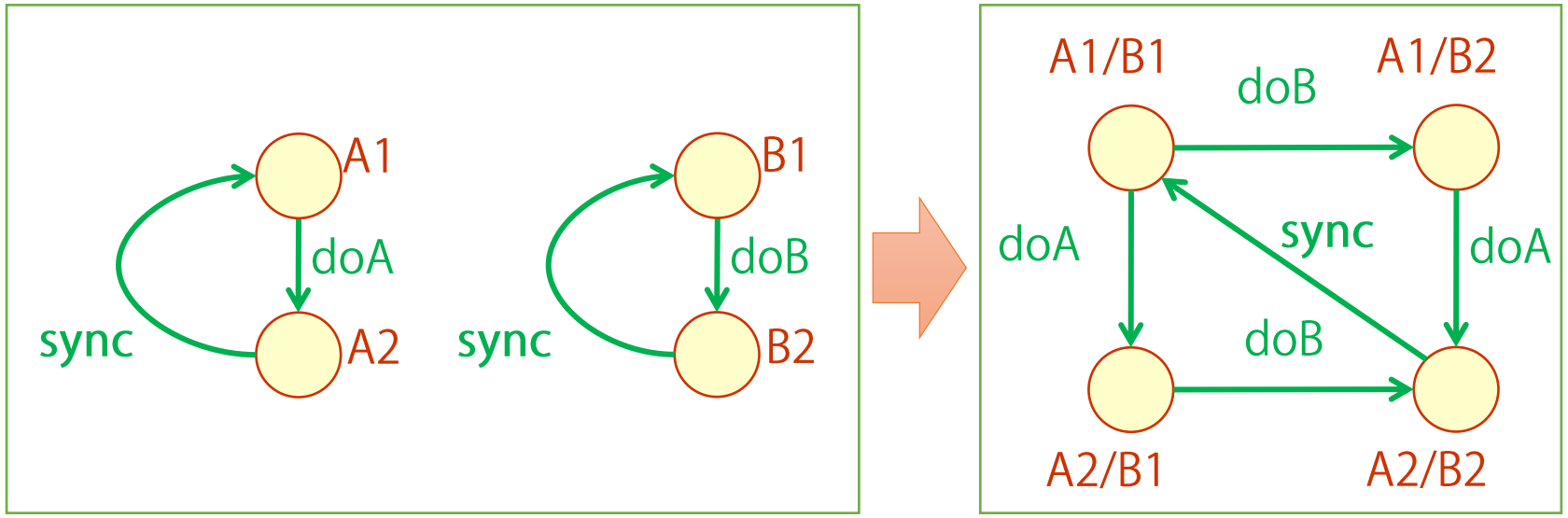
直感的には，状態遷移を網羅的に探索し，検証対象の性質が成り立つかどうかを検証する技術

  - 特に並行システムにおいて，特定の実行パスでしか顕在化しない，デッドロックや，排他制御・同期制御の誤りを検出できる

※ 数理論理学での「モデル」：ある論理式を真とする解釈（変数の真偽値など）
→「モデルであるかどうかを検査」（「状態遷移モデルを検査」ではない）

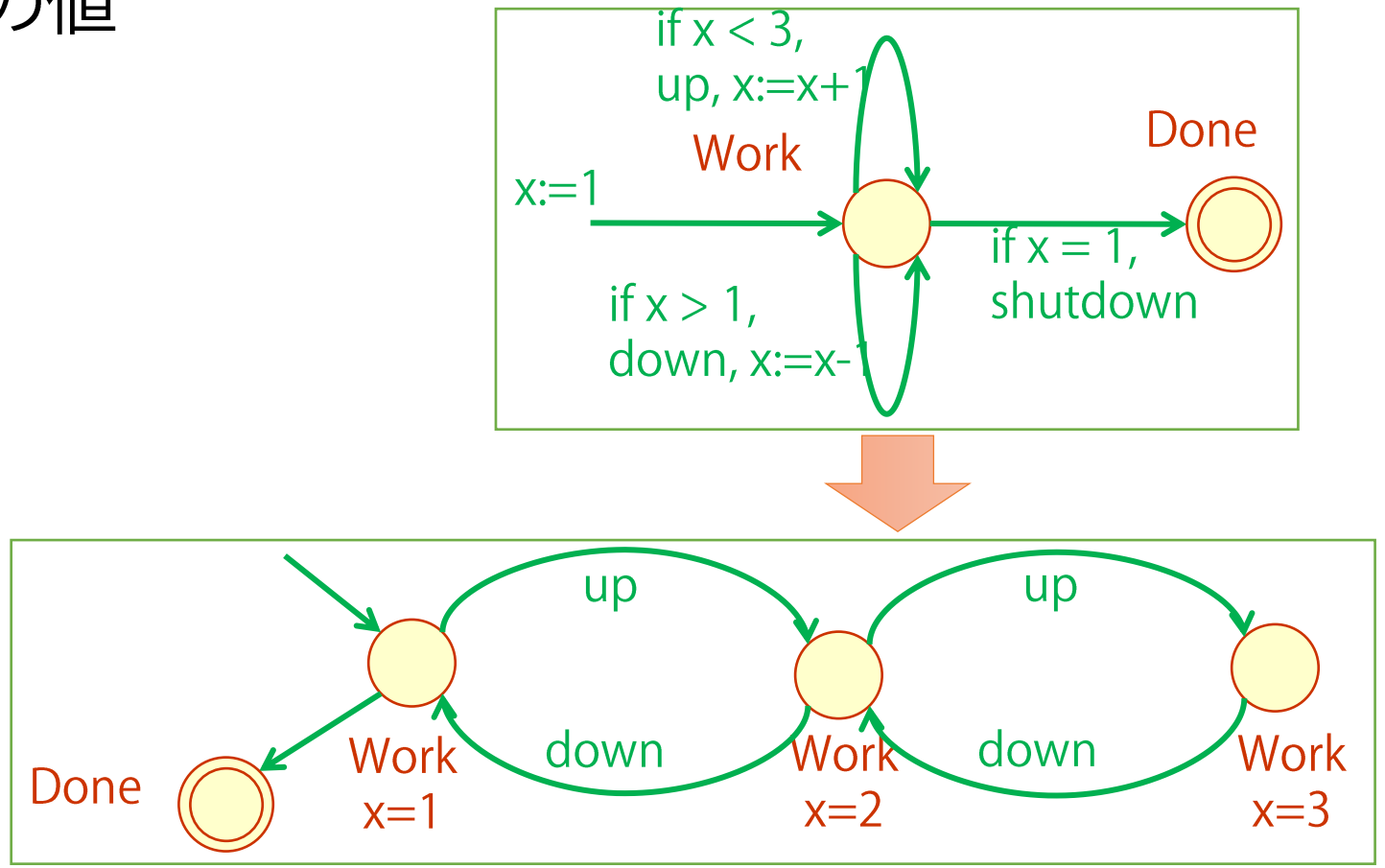
「起きうること」の探索

- 実際の探索空間： 複数プロセスのインターリーブや，変数の値を踏まえて状態遷移を展開
  - 例： 複数プロセスのインターリーブと同期



「起きうること」の探索

- （続）
  - 例： 変数の値



「起きうること」の探索

- 分岐や相互作用の本質に絞り込んで記述
  - 条件分岐にかかわる列挙型（フラグ程度）のデータ
  - 複数プロセスの切り替え，同期・排他制御，通信
  - エラーが起きるのに十分そうな数で試す
    （ユーザ数や送受信回数など）
  - 数値や集合などデータ構造はできる限り捨象する
- 形式仕様やプログラムなどを直接扱う場合
  - 外部要素などを単純化したモックに置き換えたり，探索する数値範囲やステップ数などを絞ったりする

SPINツールにおける記述例（Promela言語）

```
#define SIZE 4

byte msg[SIZE];

chan s2r = [2] of {byte};

proctype Sender() {
  byte i;
  do
    :: i == SIZE -> break;
    :: else -> s2r ! msg[i];
               i++;
  od
}
```

4個のデータを送信してみる

通信チャンネル（有限バッファ長）や送受信に関する語彙あり

この例は分岐だが，基本的に非決定的な振る舞いを書く

```
proctype Receiver() {
  byte j;
  byte rmsg;
  do
    :: j == SIZE -> break;
    :: else -> s2r ? rmsg;
       assert (rmsg == msg[j]);
       j++;
  od
}

proctype Lost() {
  byte drop;
  do
    :: s2r ? drop;
  od
}
```

順番通りに来ているかをアサーションで検証

メッセージを別のプロセスも受信しうる，として通信失敗の可能性を表現

SPINツールにおける記述例（検証式）

- アサーションの検証や，デッドロックの検出
- それ以上の性質については，時相論理式を記述
  （正確にはLTL : Linear Temporal Logic）
  - AとBが同時に真になることはない（安全性）

    `[] !(A && B)`
  - Aが真になった後には必ずBが起きる（活性）

    `[] ( A => <>B)`
  - Aが（ブロックされ続けたりせず）何度でも起きうる
    （ある種の公平性）

    `[] <> A`

おまけ： LTL (Linear Temporal Logic)

- すべての（無限長）実行パスに対して
  何か性質が成り立つかどうかを記述

A . B . A . B . …
A . B . A . C . …
A . C . B . A . …

X p （○ p）	次の状態でpが成り立つ（next）
F p （◇ p）	今の状態かそれ以降のどこかでpが成り立つ （finally）
G p （□ p）	今の状態とそれ以降のすべてでpが成り立つ （globally）
p U q	今の状態かそれ以降のどこかでqが成り立ち， それまではずっとpが成り立つ（until）

SMVツールにおける記述例

```
MODULE main
VAR
  cabin : 0 .. 3 ;
  dir : { up, down }


ASSIGN
  next(cabin) := case
            dir = up & cabin < 3     : cabin + 1 ;
            dir = down & cabin > 0: cabin – 1 ;
            1                  : cabin ;
                  esac
  next(dir)      := case
            dir = up & cabin = 2     : down
            dir = down & cabin = 1: up ;
            1                  : dir ;
                  esac
```

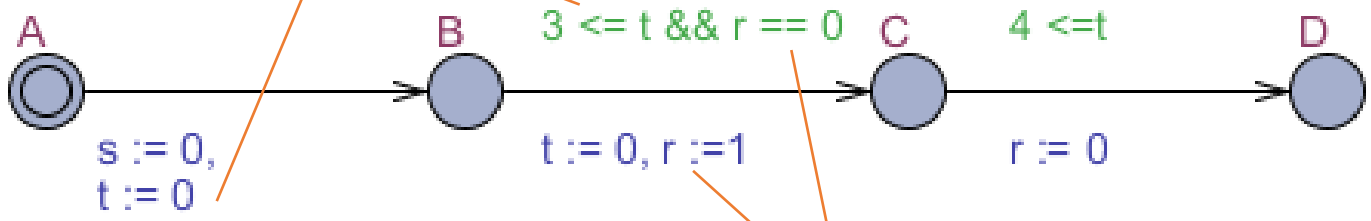
単純に上下を行き来する
エレベータの動きを表現
（実際はともかく3Fまで）

次の状態における各変数の
値をどう決めるか，という
観点で状態遷移を記述

UPPAALツールにおける記述例

状態B到達時に時間変数
（時間カウンター）tをリセット
→ Bを出てCに進むときには
時間が3以上経過している

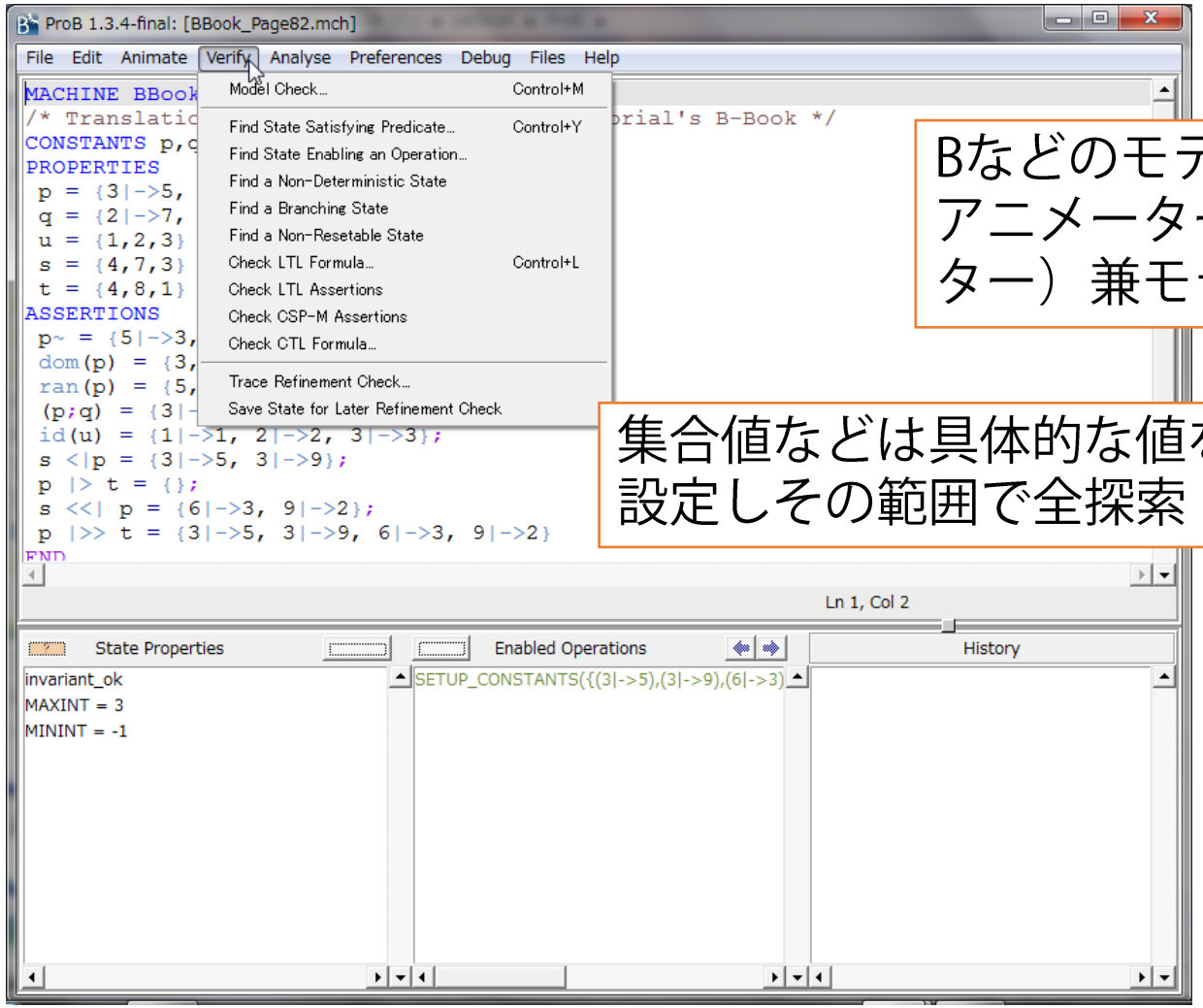
所要時間（のぶれの可能性）
も含めて状態遷移を記述



共有資源が確保できる
（r == 0）ときのみ状態Cに進め，
その際には確保（r := 1）

※ テキスト表現もある

ProBツール



Bなどのモデルを受け付けるアニメーター（シミュレーター）兼モデル検査器

集合値などは具体的な値を設定しその範囲で全探索

モデル検査の結果

- 検証の結果は下記のいずれかである
  - （全探索した場合）与えたモデルにおいて，検証対象の性質が成り立つことを保証
  - 検証対象の性質が成り立たないような実行パスの一例（反例）を提示（最も短いものを探索させることもだいたい可能）
  - 状態爆発により検証が未完了

ソフトウェアモデル検査

- プログラムコードに対してモデル検査
  - プログラムの場合，有限ステップに限る，ループ回数上限を限定するなどが必要（有界モデル検査）
  - Java PathFinder ： Javaコード対象，状態変化の可能性を探索することによる検査
  - CBMC ： Cコード対象，コードを論理式に変換してSAT/SMTソルバーによる制約充足判定により検査

[http://javapathfinder.sourceforge.net/]
[http://www.cprover.org/cbmc/]

今回の参考文献

- 形式手法適用事例調査
  - IPA，2010年
    http://www.ipa.go.jp/sec/softwareengineering/reports/20100729.html
  - 代表的な事例の特徴や体制などを調査
- Bメソッドによる形式仕様記述―ソフトウェアシステムのモデル化とその検証
  - 來間，近代科学社，2007
  - 日本語唯一のBメソッドに関する書籍
- その他各ツールのWebサイト

まとめ

- 形式手法
  - プログラムやその仕様・設計の正当性に関する基礎理論（数理論理学）に基づく手法
  - 厳密な記述を用いて曖昧さを排除
  - 定理証明やモデル検査といった強力な検証手段を提供