# 経営情報システム学特論１
# １．イントロダクション

SS専攻 経営情報システム学講座 客員
石川 冬樹
f-ishikawa@nii.ac.jp

# 自己紹介

- 2012年6月よりSS専攻客員
- 国立情報学研究所 コンテンツ科学研究系 准教授
  - ここでの「トップエスイー」と電通大との連携

    http://www.topse.jp/

    「ソフトウェア工学」に対する企業の
    方々からのニーズに対応
    （それっぽい方法でやることが限界？）
- 専門
  - Webサービス，サービス指向，クラウド
  - ソフトウェア工学（特に形式仕様，形式検証）

  http://research.nii.ac.jp/~f-ishikawa/

# 目次

- 本講義の範囲
  - 並行システム
  - 検証
  - プログラミングとモデリング
- 関連した動向
- 講義の進め方

# 並行システム

- 並行：「同時に」実行されている
  - 様々なソフトウェアの様々な部分で活用
  - 一例：裏で通信や処理をしながらユーザに状況表示
  - 物理世界で「同時」（こちらは「並列」）とは限らない
  - 1つのプロセッサで複数スレッドを少しずつ切り替えながら実行すると，「同時」に見える（疑似並列）
- 難しさ：資源の共有，待ち合わせなどの相互作用において，実行タイミングなどに起因して無数の状態変化の可能性が存在し，その一部で問題が発生しうる（実行しても再現できない）

# 並行システムの基本例題（１）

- グローバル変数などを更新する簡単な関数

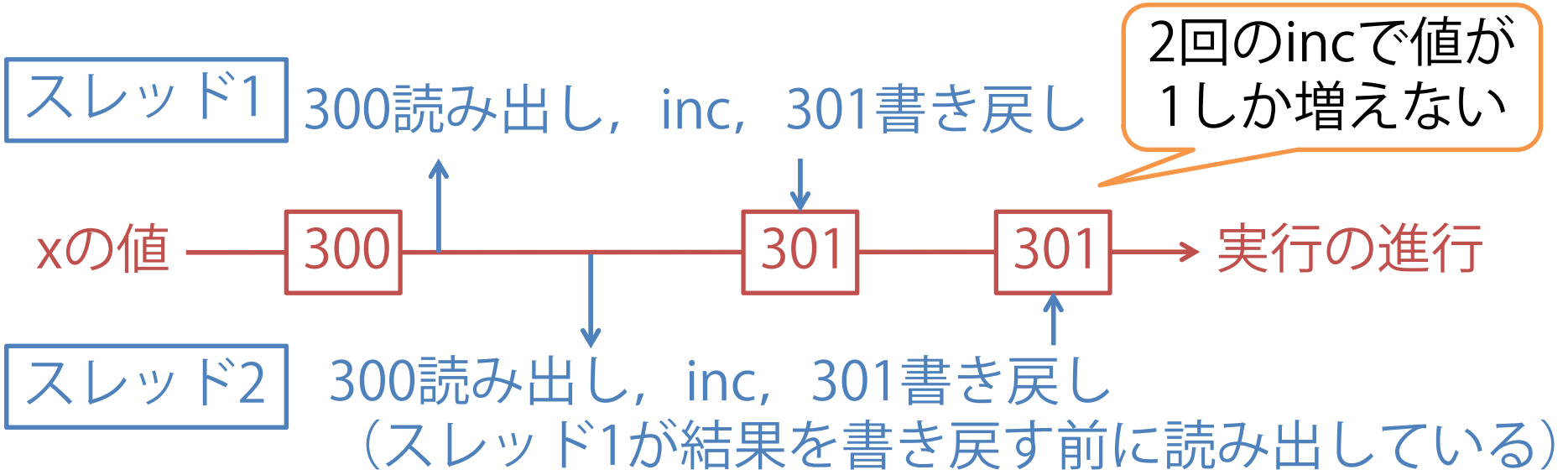
```
void increment(){
   x = x + 1;
}
```

  - これをマルチスレッド化すると・・・

# 並行システムの基本例題（１）

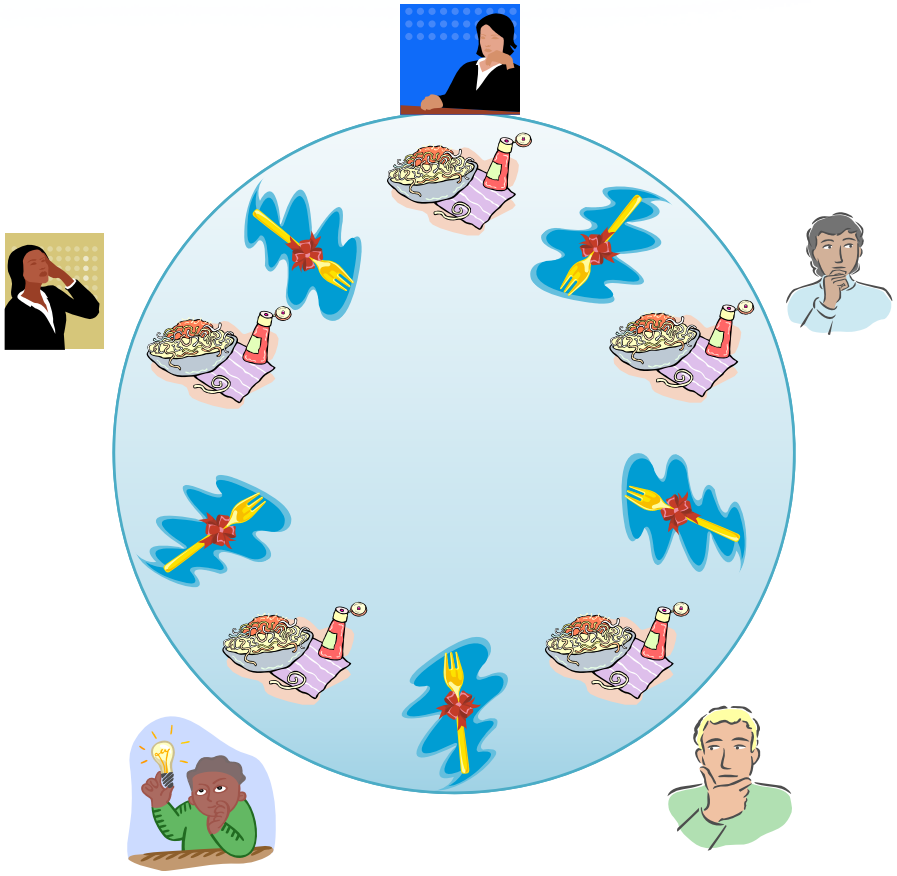
- テスト
  - 変数xの初期値を 0 とする
  - 前スライドのincrementを1000回実行するスレッドを2個作成し，並列に動作させてみる

➡ 結果は2000より小さい「ことがある」



# 並行システムの基本例題（２）

- もう少し大きな粒度での例題： 食事する哲学者（プログラムの比喩）



哲学者が円卓に座っており，各自の間には1つずつフォークが置いてある

各哲学者は，左右どちらかのフォークをとった後，もう片方をとり，両方を用いて食事をし，フォークを1つずつ置く
（会話はしない）

（左図は5人の場合）

# 並行システムの基本例題（２）

- 「哲学者」のプログラムの振る舞いループ例
  - 右のフォークが空くまで待って，空いたら確保する
  - それに成功したら，左のフォークが空くまで待って，空いたら確保する
  - 両手分が揃ったら食べて，右のフォークを置く
  - 左のフォークを置く
- ⇨うまくいく？

  「うまくいく」とは何をもってそう決める？

# 並行システムの基本例題（２）

- 「哲学者」のプログラムの振る舞いループ例
  - 右のフォークが空くまで待って，空いたら確保する
  - それに成功したら，左のフォークが空くまで待って，空いたら確保する
  - 両手分が揃ったら食べて，右のフォークを置く
  - 左のフォークを置く
- デッドロックする「ことがある」
  - 全員が右のフォークを持った状態になると，誰もが左のフォークを確保できず待ち続ける

# 並行システムならではの要件

- 例題（２）における並行システム固有の性質
  - 誰も何もできなくなる状況に到達することがない（デッドロックフリー）
  - 誰も食べることができないような状況変化を，繰り返し続けることがない（ライブロックフリー）
    - 「タイムアウトして手に取ったフォークを置く」を全員で同じタイミングで繰り返すなど
  - 誰か1人がずっと食べられないような状況変化を，繰り返し続けることがない（公平性）

# 目次

- 本講義の範囲
  - 並行システム
  - 検証
  - プログラミングとモデリング
- 関連した動向
- 講義の進め方

# 本講義の焦点

- 「バグ」？「正しくない（間違っている）」？
  - 機能に関する不具合を扱う
    「関数（メソッド）の入力が・・・出力が・・・
    結果として画面に表示された情報が・・・」
    （実行速度などの非機能的側面は扱わない）
  - 定めた仕様に照らし合わせて定まる不具合を扱う

```
int multiply(int x, int y){
  return x + y;
}
```

正しい？

# 用語：「正当性」・「検証」

**V&V**
**検証（Verification）**
**妥当性確認（Validation）**

- **（正当性）検証**：「成果物を正しく作っている？」
  その成果物が満たすべき性質（基準）を満たす？

こちらを扱う！

- **妥当性確認**：「妥当な成果物を作っている？」
  自身や顧客の本来の要求に合致している？

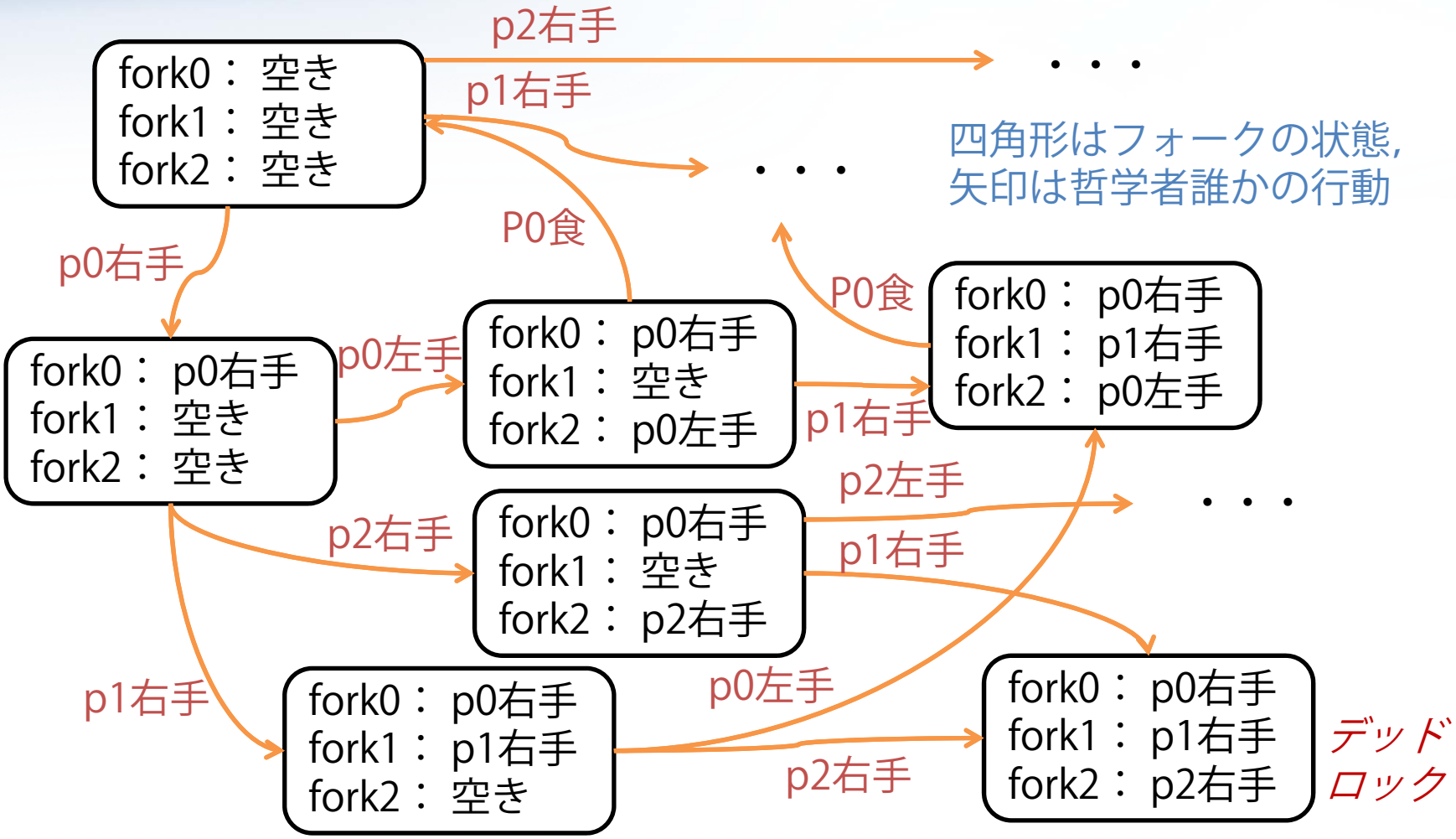
# モデル検査の基礎： 検証する性質

| | |
|---|---|
| *Reachability*<br>*到達可能性* | ある条件の下ある状況に到達しうる<br>「初期画面に戻る操作の列が常にある」 |
| *Safety*<br>*安全性* | ある条件の下ある状況に到達することがない<br>「踏切が空いた状態で電車が通過することはない」 |
| *Liveness*<br>*活性* | ある条件の下ある状況にいつか必ず到達する<br>「登録をするといつか必ず確認メールが届く」 |
| *Deadlock-freeness* | デッドロックが起きない |
| *Fairness*<br>*公平性* | ある条件の下ある状況が無限回起きる（ない）<br>「ユーザが無数にリクエストを送れば無数に返事が来る（他のユーザ等にブロックされ続けることはない）」 |

# 検証アプローチは？

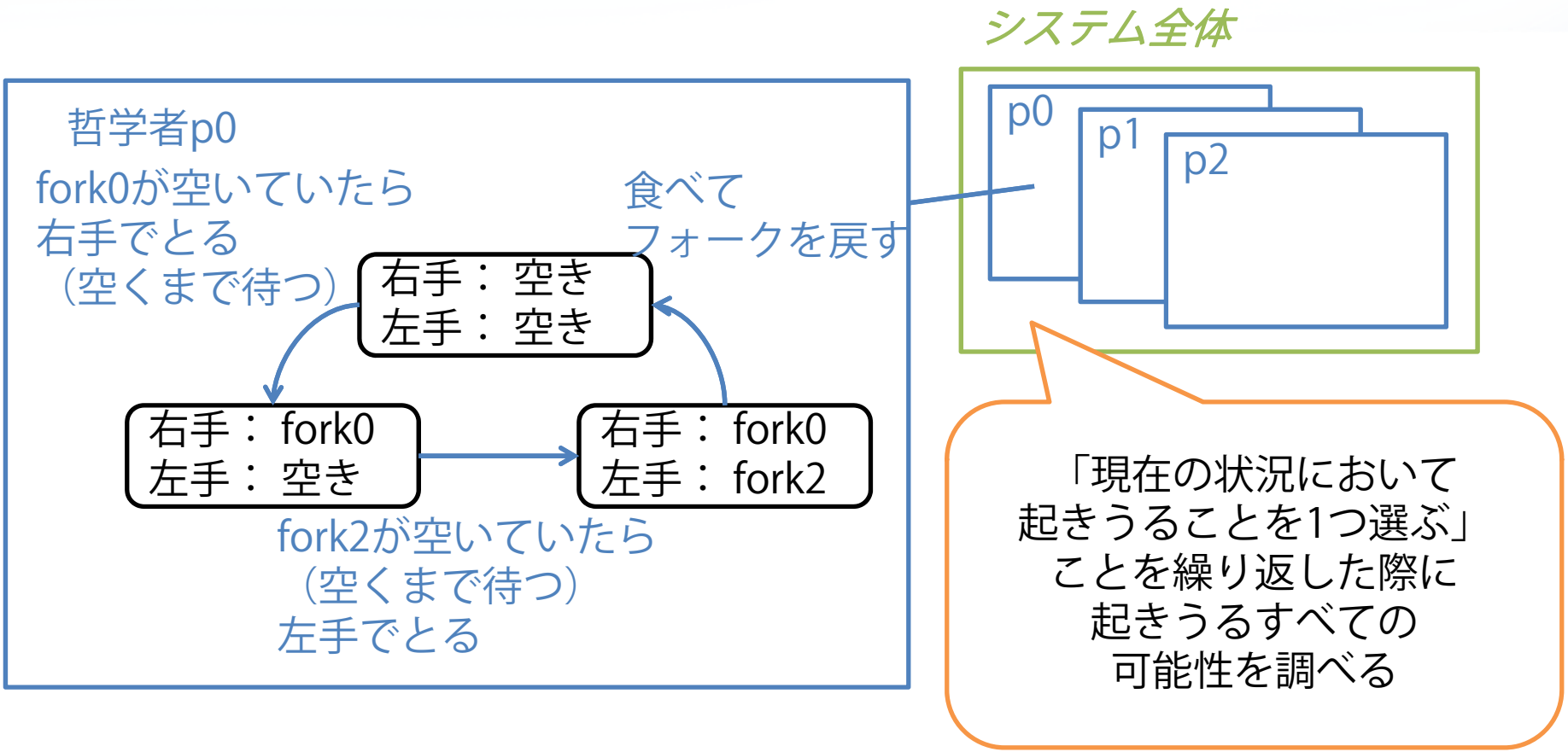
- システム全体で起きうる状態変化が非常に多数で複雑である
  - スレッドがいつ切り替わるか
  - 複数プロセス（哲学者）のどれが先に起動し，どれのどの行動がどういう順序で起こるか
  - その他通信の失敗やタイムアウトなど
- これらをすべて考えなければ，保証はできない
  - 特定のケースでのみ，誤りが顕在化したり，デッドロックが発生したりするため
  - 人の頭では賢く「こういう場合だけ考えれば十分」と絞れるかもしれない（難しい・間違えうる）

# 例題（２） 3人版の状態遷移



# 例題（２） 3人版の状態遷移

- 今やったこと

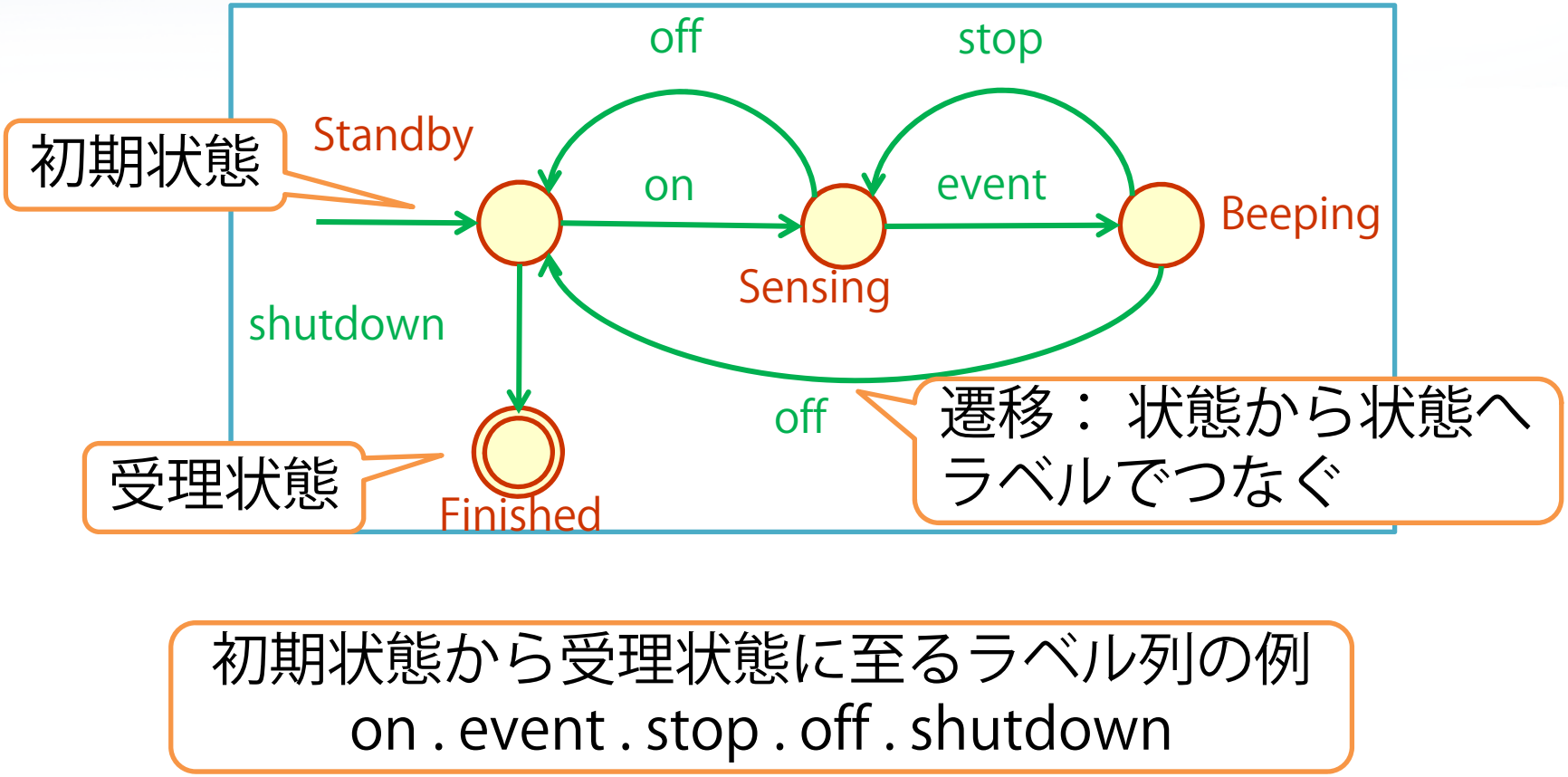


# 何を探索するのか？

- 現実に起きうることすべてを探索したわけではない
  - 哲学者が食べるスパゲッティーは，ナポリタンとボンゴレとカルボナーラから選ばれる
  - ⇨ この可能性も網羅して図を描き直す！？
- 検証内容に応じて本質的な情報を抜き出し，その範囲で網羅している
  - すなわち対象システムのモデルを構築して分析している
  - もしも「ナポリタンを食べたら，次に3人が食事を終えるまでフォークを採ってはならない」という謎ルールがあったらモデルをどう変える？

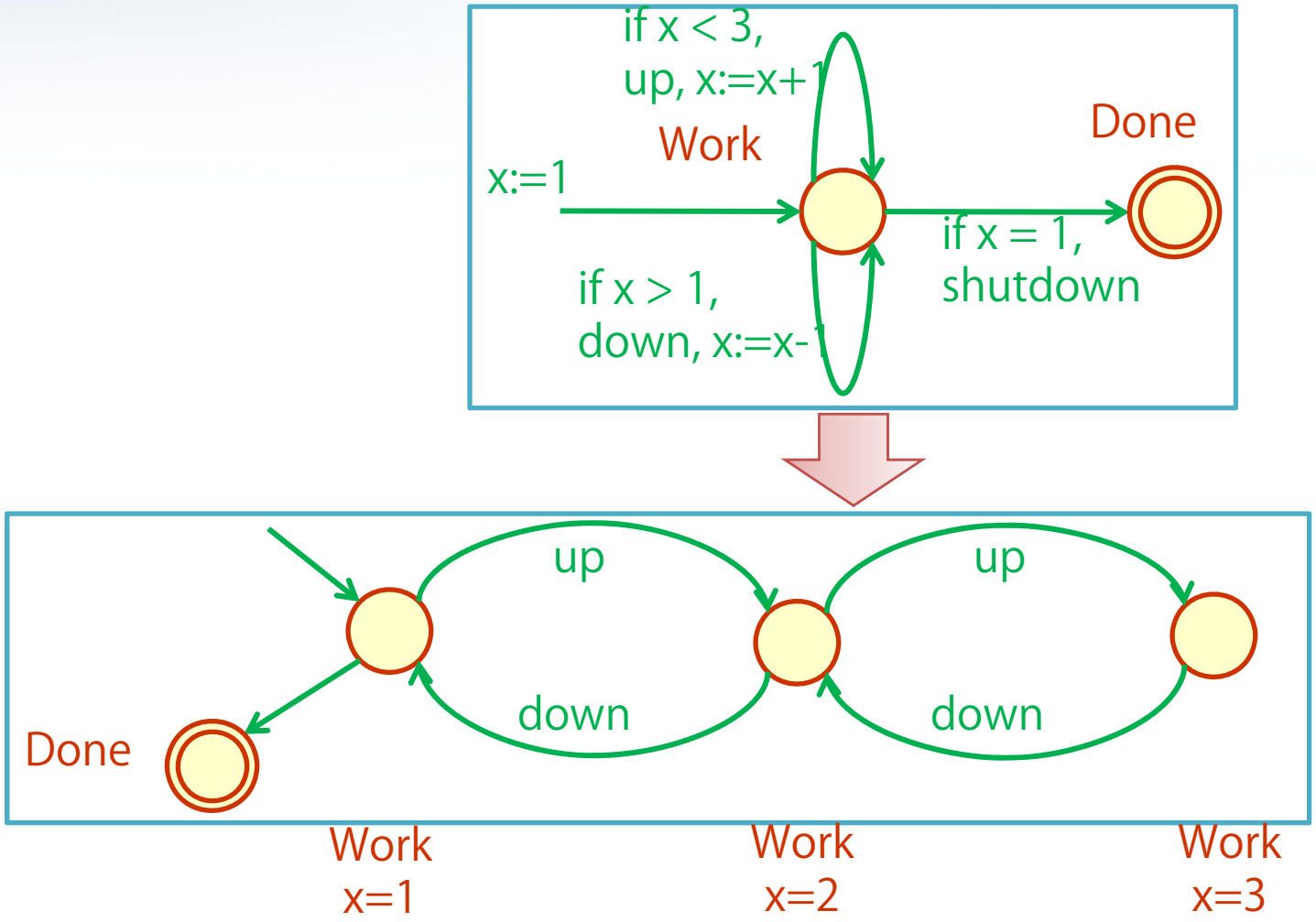
# 基礎理論： オートマトン

アラームの例

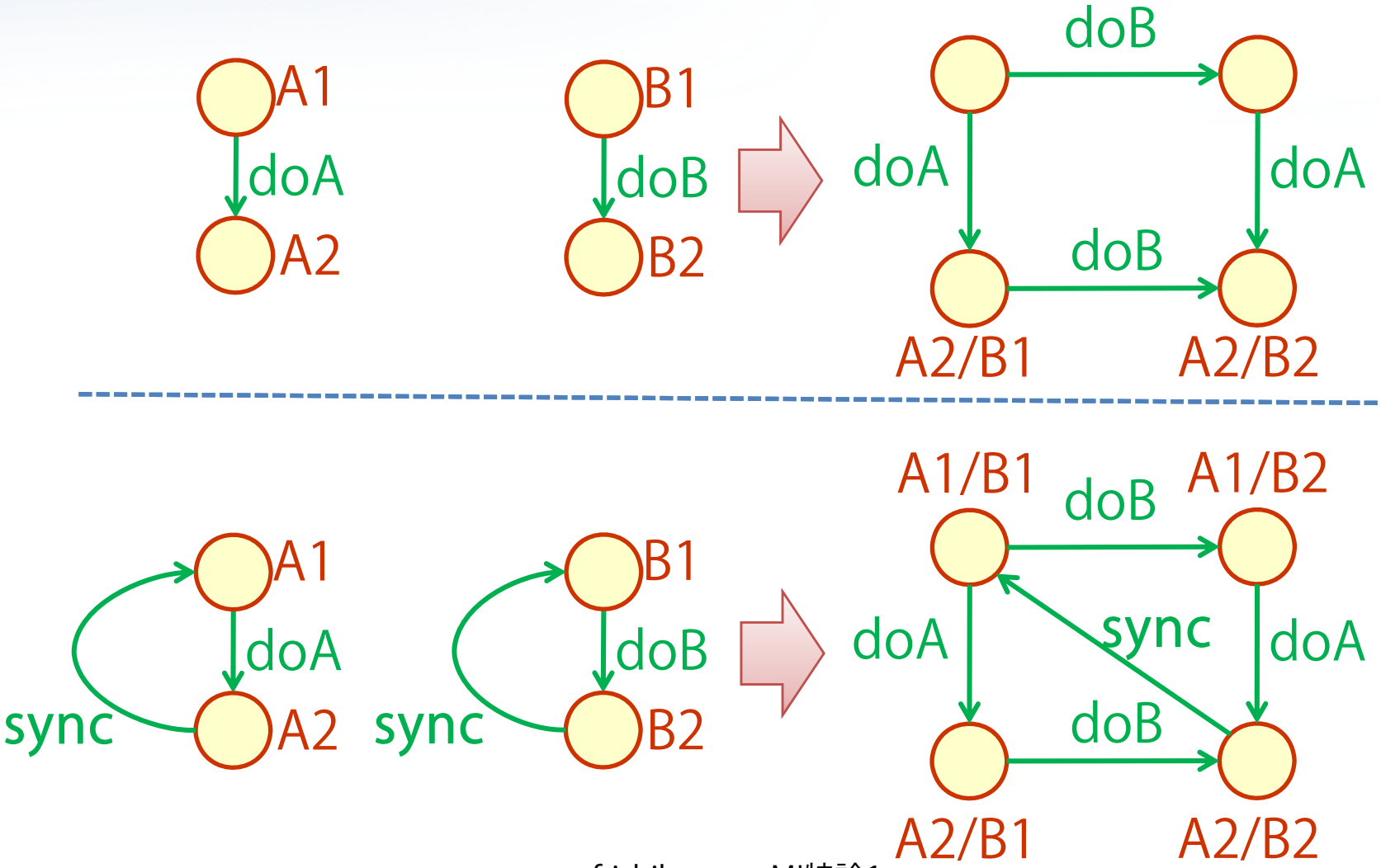



初期状態から受理状態に至るラベル列の例
on . event . stop . off . shutdown

# 基礎理論：オートマトン


if x < 3,
up, x:=x+1
Work
x:=1
Done
if x = 1,
shutdown
if x > 1,
down, x:=x-1
up
down
up
down
Done
Work
x=1
Work
x=2
Work
x=3

# 基礎理論： 並行動作の合成



# おまけ：ソフトウェア「工学」

- 多くのソフトウェアが大規模であり，様々な人に利用され，使い捨てではない
  - 多人数で，ときには組織をまたがり開発する
  - 個人の能力や思い込みにすべてを委ねる，全責任を押しつけるべきではない
- 作る人のセンスや好みで，頭の良さで，各自適宜頑張ればよい，わけではない

# 並行プログラムの検証アプローチ

- 検証アプローチ
  - 理屈をもって説明することが簡潔にできることもあるが，可能な状態遷移を洗い出して論じなければならないことも多い
    - ただし実装では通常意図的に再現できない
- 「（モデル上において）起きうることすべて洗い出して，試す」仕組みを持つ検査ツールの有効性が高い
  - 変数値の変化なども併せて，「すべて試す」

モデル検査

※「モデル検査」の「モデル」は，ある性質を満たす構造という数学用語（「モデルかどうか検査」）

# 目次

- 本講義の範囲
  - 並行システム
  - 検証
  - プログラミングとモデリング
- 関連した動向
- 講義の進め方

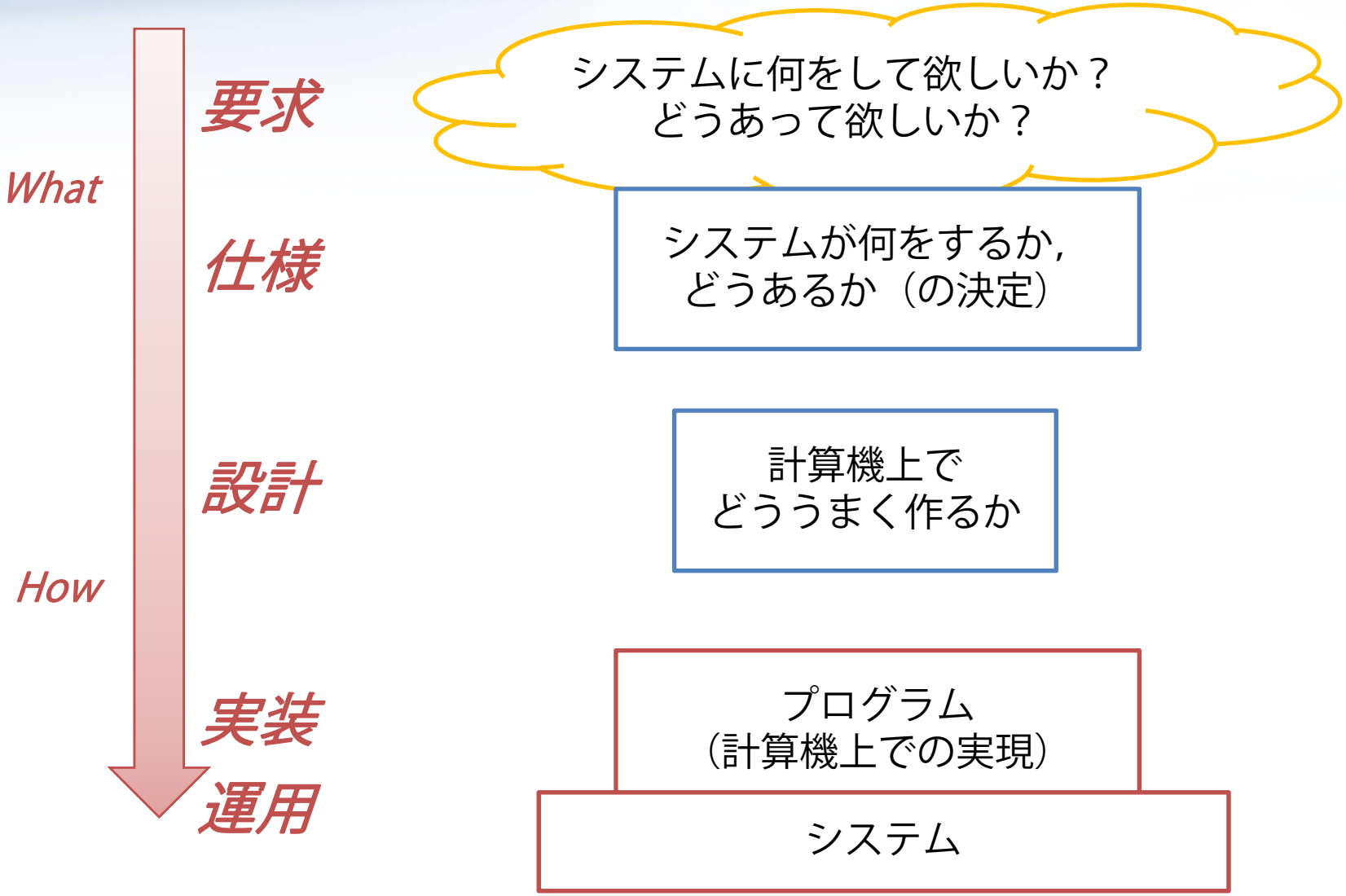
# 用語：「モデル」

モデル：計算や予測を助けるために用いられる，システムやプロセスの単純化された記述（しばしば数学的な）

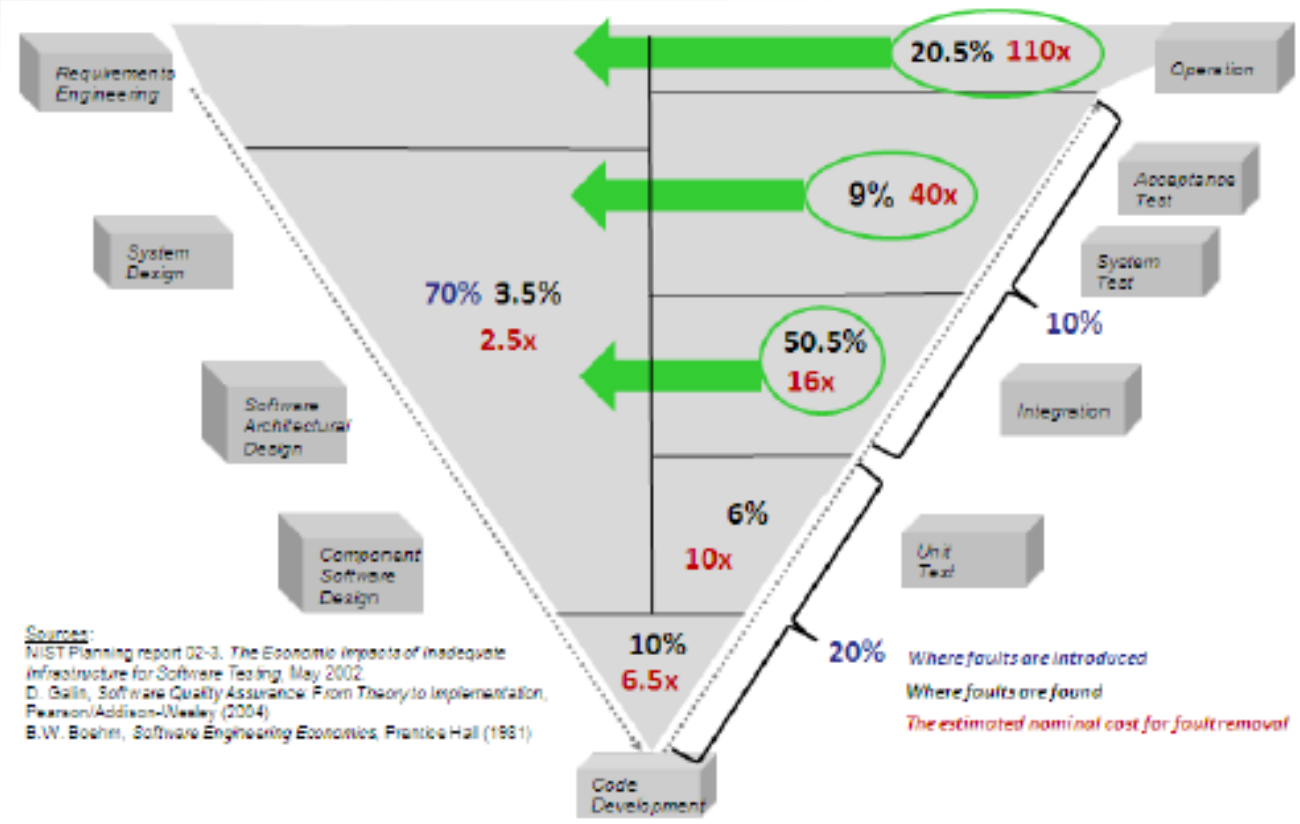
（Oxford英英辞典の石川訳）

- 注目する側面に特化し，抽象化・単純化
  （不要な実現詳細を捨象）
- 効率的，効果的に分析，検証
  （人手にしろ，自動にしろ）

# ソフトウェア開発における抽象度



# コストの考え方：「手戻り」の影響




青字：
不具合が埋め込まれる場所

黒字：
不具合が発見される場所

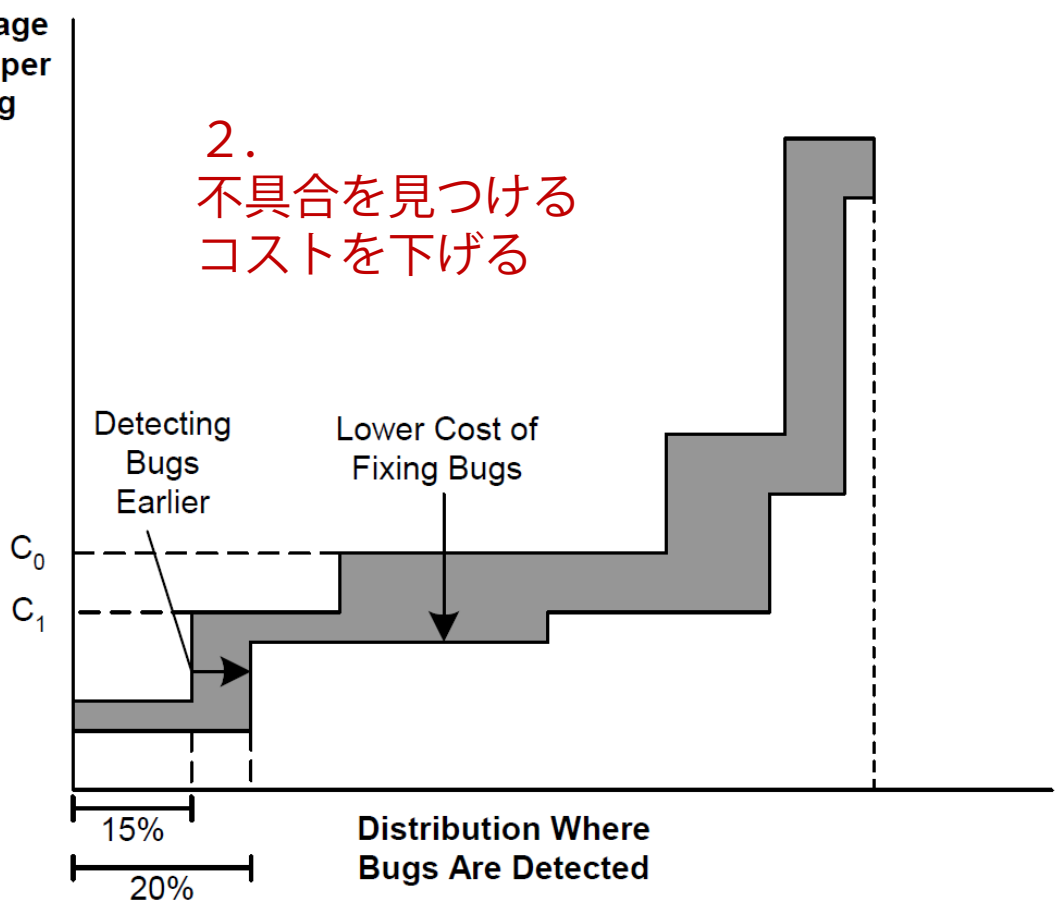
赤字：
不具合の除去におけるコストの増加率

[P. H. Feilerら, System Architecture Virtual Integration: An Industrial Case Study, 2009]

# コストの考え方： 対応策

1．
より早く不具合
を見つける

2．
不具合を見つける
コストを下げる




[The Economic Impacts of Inadequate Infrastructure for Software Testing, NIST, 2002]

# コストの考え方： 対応策

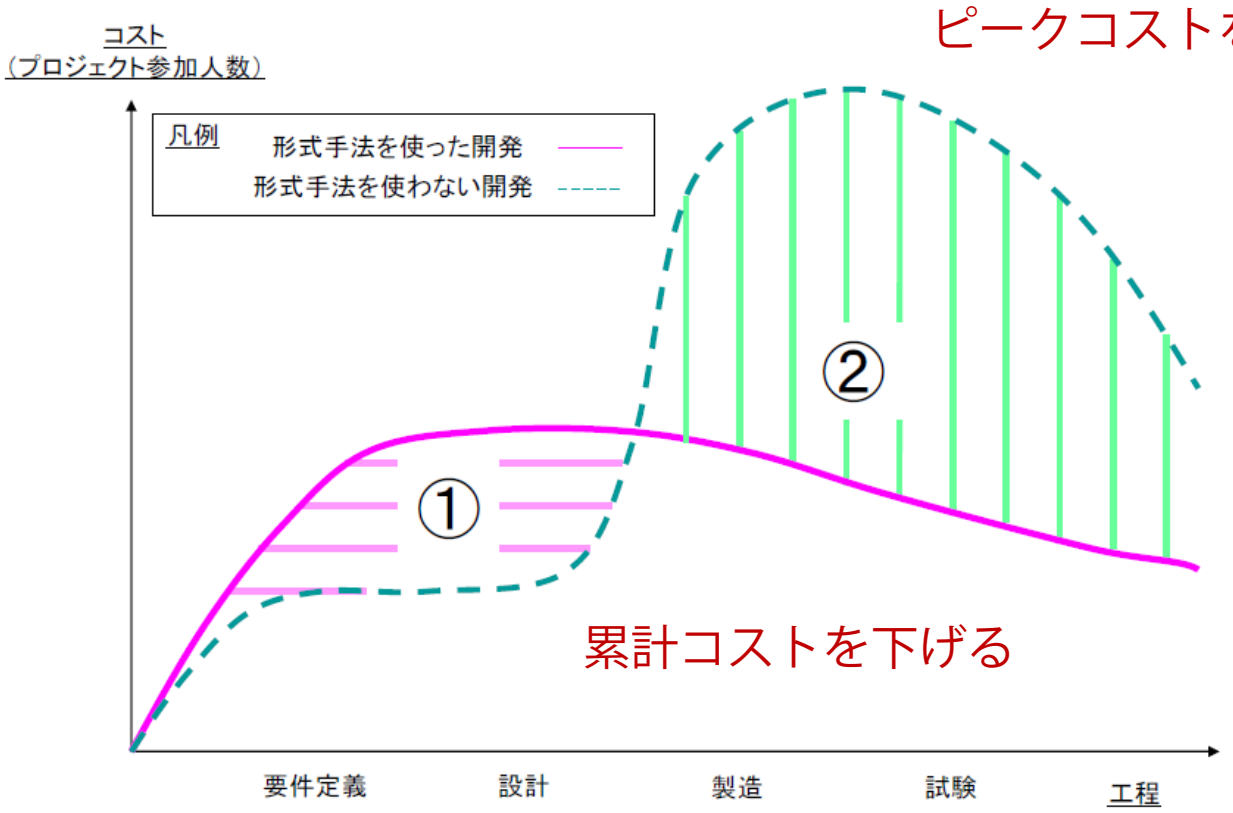



**図 1　形式手法導入によるフロントローディング効果イメージ**

[http://www.nttdata.com/jp/ja/dsf/]

# 目次

- 本講義の範囲
  - 並行システム
  - 検証
  - プログラミングとモデリング
- 関連した動向
- 講義の進め方

# おまけ：形式手法

形式手法（の大ざっぱな説明）

- プログラムを書くと
  - 対象が厳密，明確に，
    ある基準を満たすよう「一通り」書き出される
  - 型チェック，テスト等分析・検証を行える
- 上流の文書（仕様・設計）でも！（形式記述）

  厳密な文法と意味論がある言語を用いて記述し，
  分析や検証を行う
  （ただしHowは書かずWhatだけ書く）
- プログラムであれ上流であれ，より強力な検証技術を用いる（形式検証）

# おまけ：形式手法と国内産業界

- 最近産業界でも様々な活動が行われている
  - 適用事例調査（2010年7月）
    IPA http://sec.ipa.go.jp/reports/20100729.html
  - 導入ガイダンス（2011年5月）
    経産省（MRI，NII） http://formal.mri.co.jp/
  - 適用手順・イディオム（2011年7月）
    Dependable Software Forum （DSF）
    http://www.nttdata.com/jp/ja/dsf/
    http://www.ipa.go.jp/sec/softwareengineering/reports/20120928.html
  - 実証実験（2012年4月）
    IPA・東証 http://sec.ipa.go.jp/reports/20120420.html

# モデル検査の代表的な事例

例えば・・・

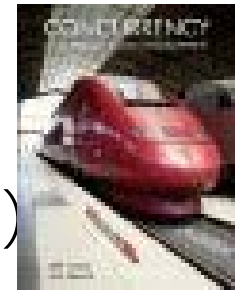
- 北米トヨタリコールに対するNASAによる調査
- IEEE Futurebus+ （キャッシュプロトコル）[1993]やNeedham-Schroeder（認証プロトコル）[1996]における欠陥の発見
- その他一般的なシステムでの活用など

# 目次

- 本講義の範囲
  - 並行システム
  - 検証
  - プログラミングとモデリング
- 関連した動向
- 講義の進め方

# 進め方

- マルチスレッドプログラミングで体験しつつ，モデルの記述・検証も行っていく
  - 分散システム（コンピューター複数）の場合は，プログラミング演習まではしない
- 教科書（買わなくてもよい）
  - Concurrency: State Models and Java Programs, Jeff Magee and Jeff Kramer, Wiley 2006



- 演習
  - Java（講師はEclipseにて説明，他でもよい）
  - LTSA

http://www.doc.ic.ac.uk/~jnm/book/

# 評価

- 小演習を行うレポート課題
- 出席状況
  - 社会人学生の方はご相談下さい

# まとめ

- 並行システム
  - プログラミング技術としても，より上流の設計においても，非常に重要性が高い
  - 固有の難しさ： 資源の共有，待ち合わせなどの相互作用において，実行タイミングなどに起因して無数の状態変化の可能性が存在し，その一部で問題が発生しうる（実行しても再現できない）
  - 抽象化し，制御しやすいモデルを対象として検証を行うことが有効
- 次回
  - マルチスレッドの基礎（１）