

7

ソフトウェア工学

Software Engineering

モデル検査

MODEL CHECKING

1 モデル検査の概要

並行システム：相互排他，デッドロック，スタベーションなどの現象

➡ 入出力関係に着目した「停止性＋部分正当性」のみでは正当性を言えない

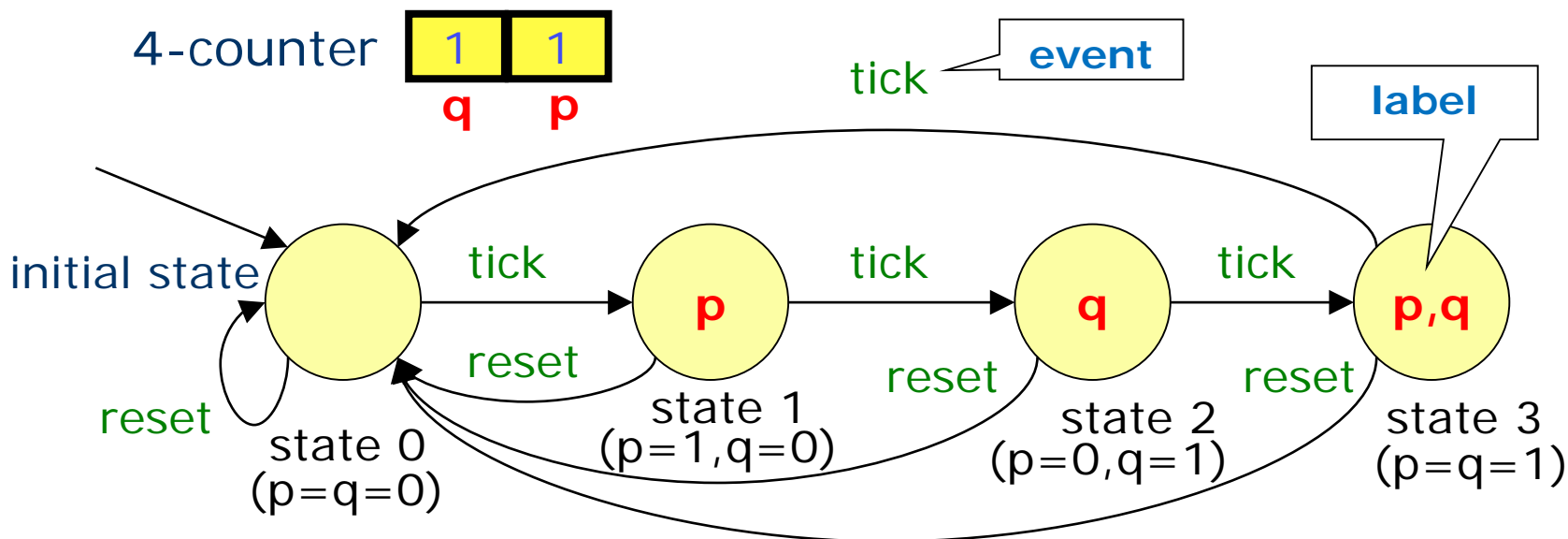
➡ 振る舞い(途中の状態遷移)の考慮の必要性
behaviors

➡ モデル検査：有限状態遷移系の振る舞いの検証を自動で行う技術
model checking

モデル = 有限状態遷移系：状態数が有限個の状態遷移系
finite state transition system

検査 = 検証：システムが期待される性質(仕様)を満たすことの確認
verification properties (specifications)

モデル: 状態遷移系



状態遷移系は次の4項目からなる.

- 状態(ノード)の有限集合
- 初期状態の集合(ふつうは1個の初期状態)
- 遷移関係(イベントに対応する有向辺の集合. イベントが状態遷移を引き起こす)
- ラベル(各状態で真(=1)となる原子命題の集合. これにより状態を区別)

状態数は数百万くらいはOK

検証できる主な性質

モデル検査では、状態遷移系の振る舞いについて、主につぎの2つの性質を検証する

□ 安全性 (safety)

「悪いことは決して起こらない」

例: このエレベータは、ドアが開いたまま昇降することは決してない

□ 活性 (liveness)

「良いことはいつか必ず起こる」

例: 緊急停止ボタンを押したら、いつか必ず停止する

「0.5秒以内に」というようなリアルタイム性を調べられることも

なにが「悪いこと」で、なにが「良いこと」かは、応用目的にあわせて設計者が決定

モデル検査の実施手順

モデル検査はつぎの3ステップで実施する

- (1) **モデリング** : システムを状態遷移系として表現
 - (2) **性質の記述** : 検査したい性質を具体的に表現
 - (3) **モデル検査** : モデル検査器で自動検査
-

(1) モデリング

- システムの振る舞いを表すモデル（状態遷移系）を記述する.
 - 記述には一般に、モデル記述言語を用いる.
 - モデル記述言語の分類
 - プロセス代数に基づく数理的な言語
process algebra
 - プログラミング言語風の言語
-

(2) 性質の記述

- システムが満たすべき（検査したい）**性質**を具体的に記述する.
property
- 記述には一般的に、時間の概念を扱う**時相論理**を用いる.
temporal logic
- 時相論理として、**LTL**と**CTL**がよく知られている.
LTL (Linear Temporal Logic) CTL (Computation Tree Logic)

【LTL の例】 **G(Req → F Ack)**



Reqが真になると、その後いつか必ずAckが真になる

It is **globally (always)** true that

if Req is true then in **future** Ack will be true.

(3) モデル検査

モデルが、与えられた性質を満たすかどうか、自動的に検査する。

モデル検査器

model checker

- 性質が成り立つ場合：検査終了
- 性質が成り立たない場合：反例（エラートレース）を出力
counterexample (error trace)

モデル検査アルゴリズムの基本原理

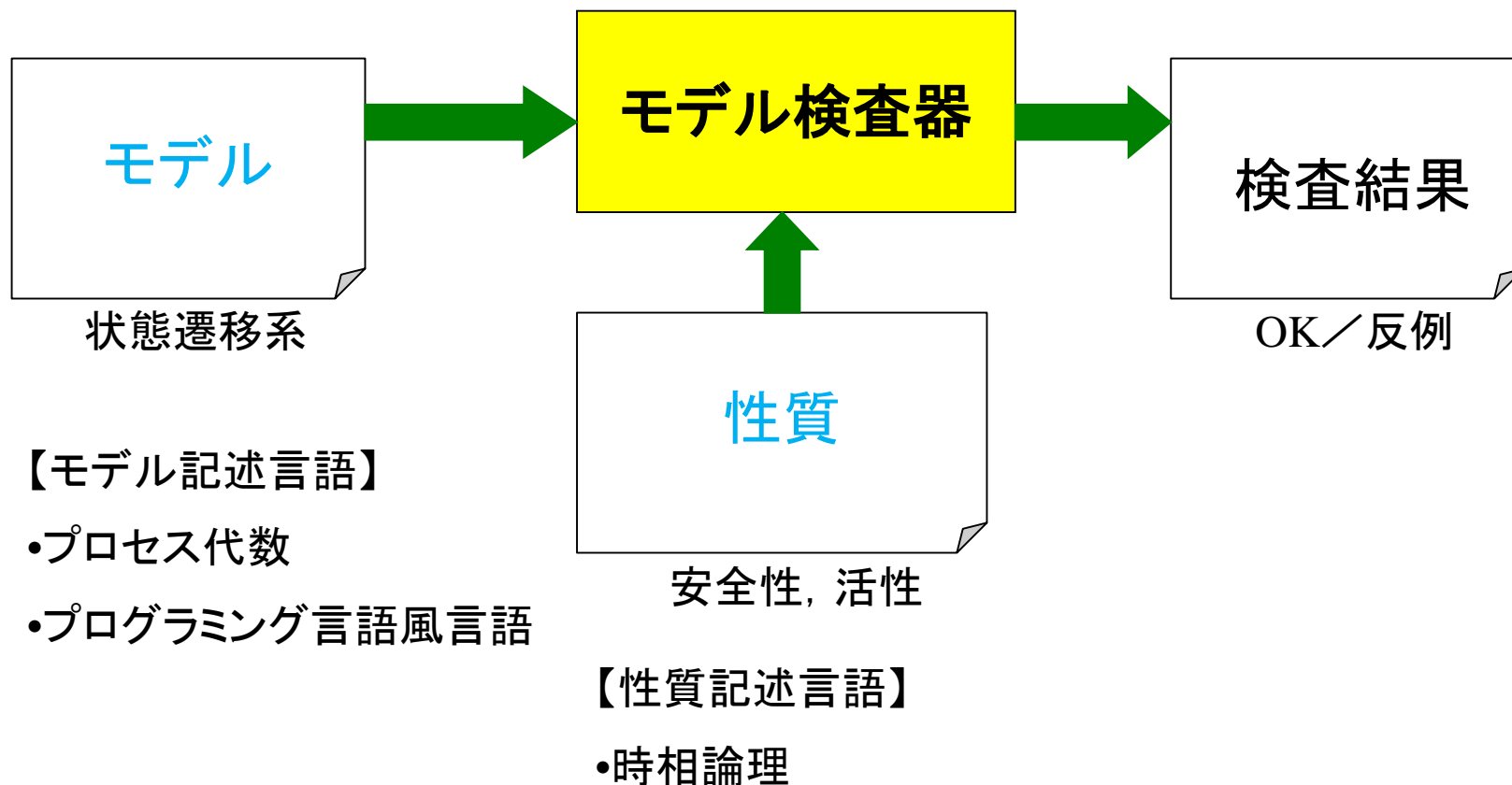
- ➡ すべての入力とすべての状態遷移列について
振る舞いを網羅的に検査
-

モデル検査器の例

- SMV, NuSMV
カーネギーメロン大学
IEEE Futurebus+ standardのバグ発見
- SPIN
ベル研究所
ACM Software System Award受賞
- LTSA
ロンドン王立大学
FSP言語(プロセス代数), アニメーション
- Java PathFinder (JPF)
NASA
Javaのバイトコードの検査

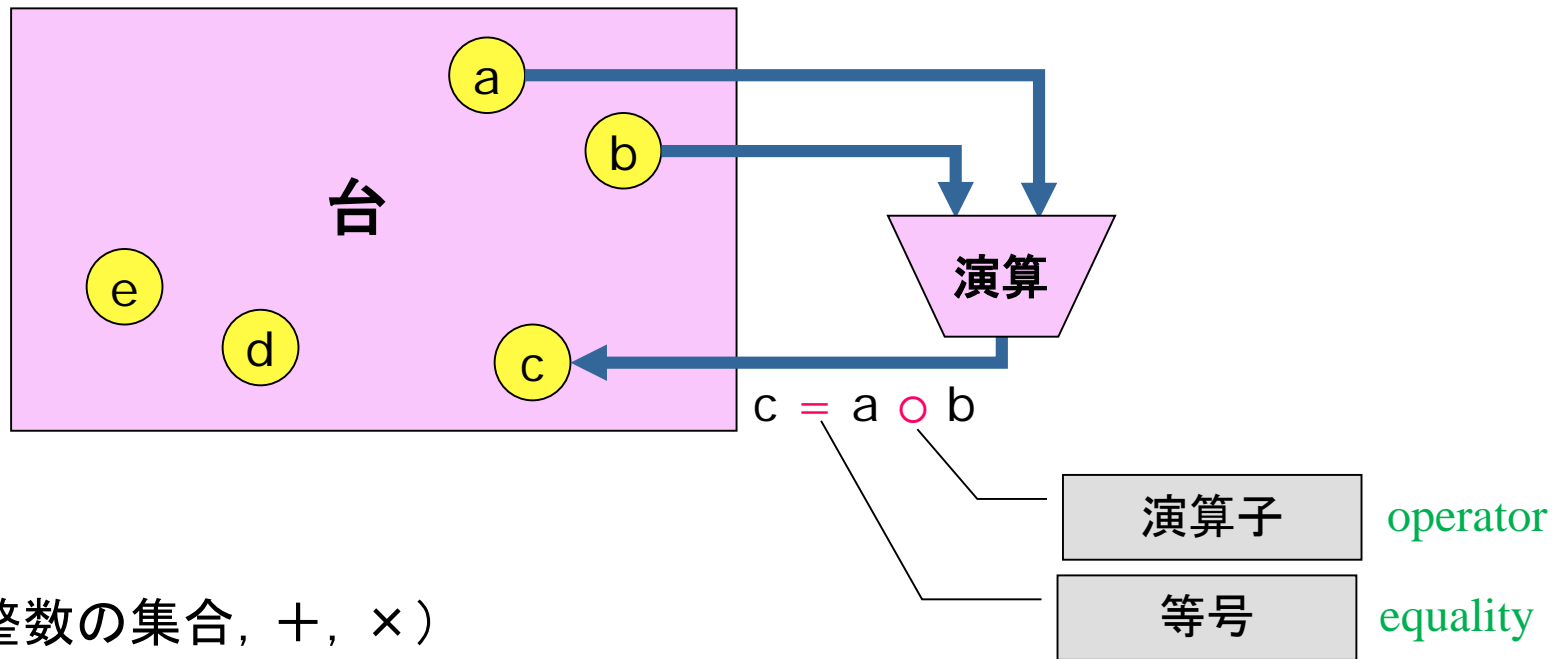
} きのうの授業では
これを紹介

モデル検査器の概要



2 逐次プロセスのモデリング

代数: 要素の集合(台)およびその上での演算の集まりからなる計算系
algebra universe operation

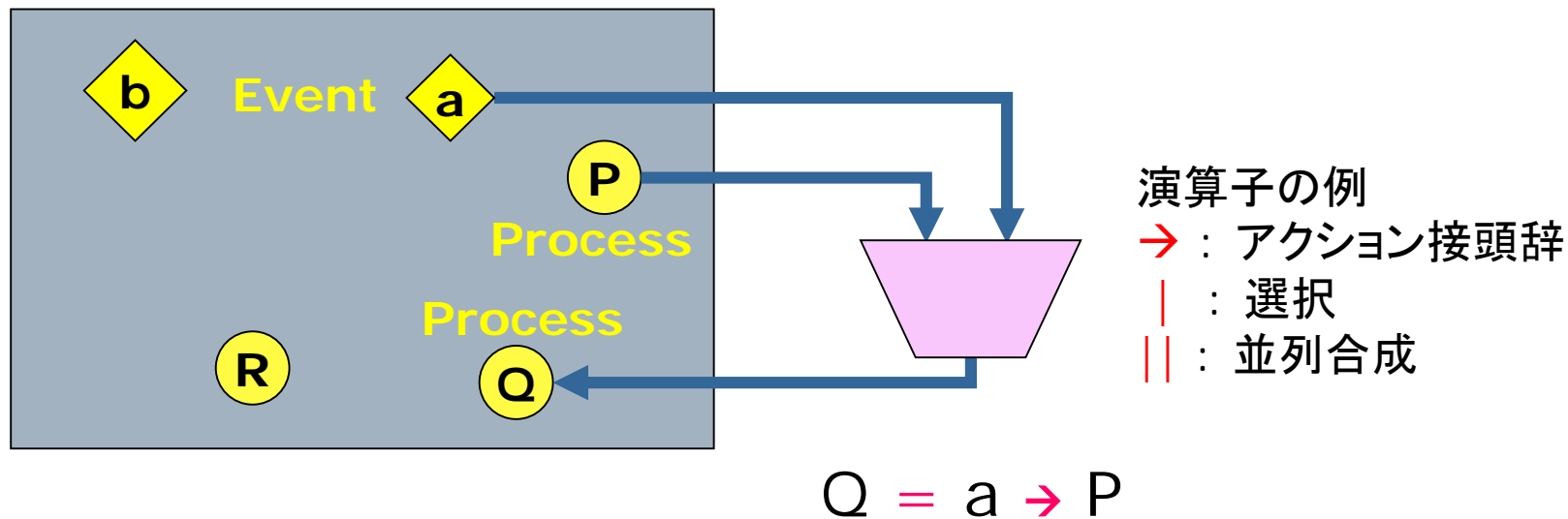


例: (整数の集合, +, ×)

process algebra

プロセス代数

プロセス代数: プロセスやイベント等の集合を台とし,
プロセス合成などの演算を定義した代数



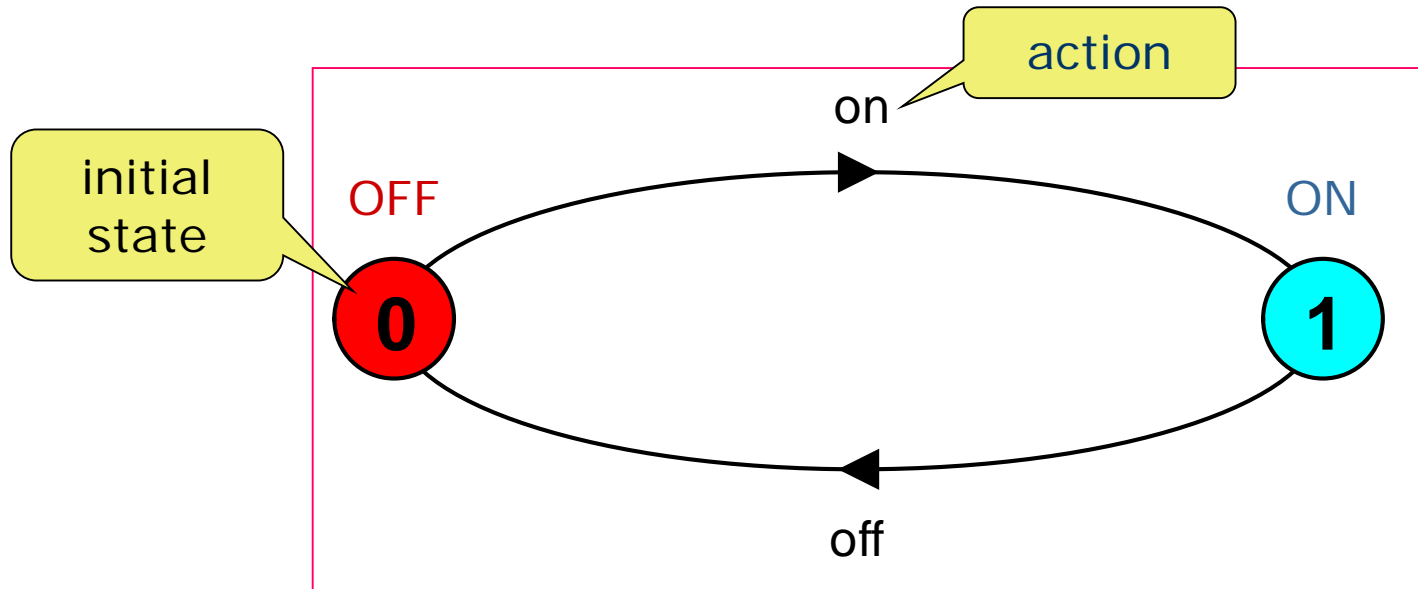
action prefix

アクション接頭辞

$(a \rightarrow P)$

最初にアクション a を実行し、つぎにプロセス P と同じ振る舞いをするプロセス.

スイッチの例(1)



プロセス = 実行可能なアクションの列(トレース)の集合

OFF = { on → off → on → off → on → off → }

ON = { off → on → off → on → off → on → }

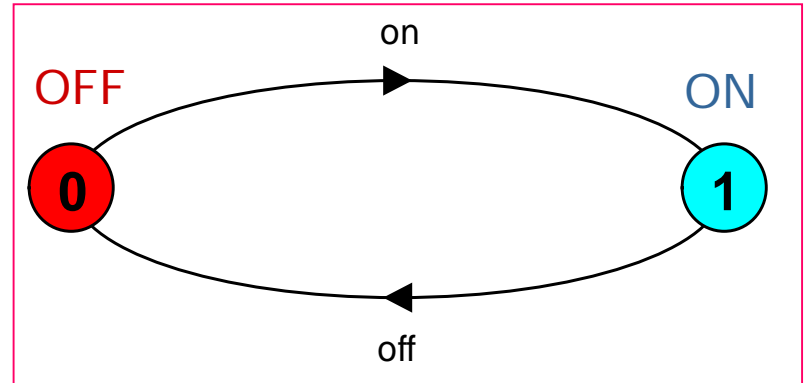
(FSP: Finite State Processes)

スイッチの例(2) FSPによる記述

```
SWITCH = OFF,  
OFF     = (on → ON),  
ON      = (off → OFF).
```

プロセス名
(大文字)

アクション名
(小文字)



代入によってより簡潔な表現を得る

```
SWITCH = OFF,  
OFF     = (on → (off → OFF)).
```

```
SWITCH = (on → off → SWITCH).
```

(→ は右結合演算子)

choice

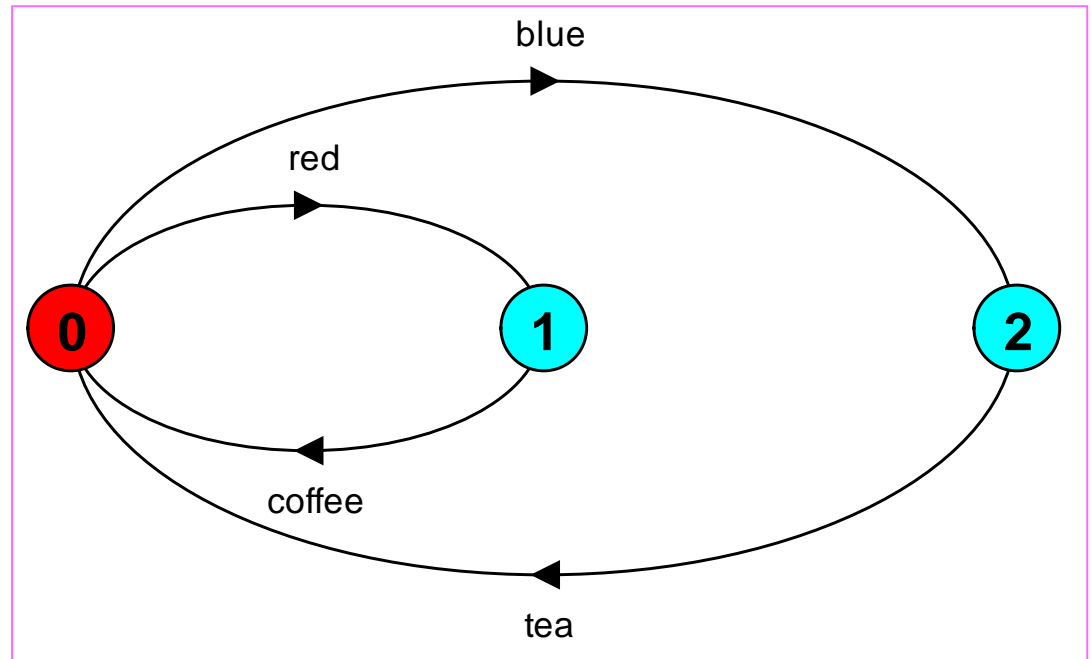
選択

$(a \rightarrow P \mid b \rightarrow Q)$

最初にアクション a, b のいずれかを実行する.
そのアクションが a ならつぎにプロセス P を実行し,
 b ならつぎにプロセス Q を実行するプロセス.

自動販売機の例

ボタンの色で飲み物を指定



DRINKS = (red → coffee → DRINKS
| blue → tea → DRINKS
).

3 並行プロセスのモデリング

プロセスの並列合成

$(P \parallel Q)$

プロセス P と Q の並行実行を表すプロセス。

交換則: $(P \parallel Q) = (Q \parallel P)$

結合則: $(P \parallel (Q \parallel R)) = ((P \parallel Q) \parallel R)$
 $= (P \parallel Q \parallel R).$

action interleaving

アクションのインタリーブ

プロセス間で共有されないアクションはインタリーブされる

かゆいところ
をかく

```
ITCH = (scratch → STOP).  
CONVERSE = (think → talk → STOP).
```

} 共有アクション
無し

会話する

```
|| CONVERSE_ITCH = (ITCH || CONVERSE).
```

並列合成は || で書き始める

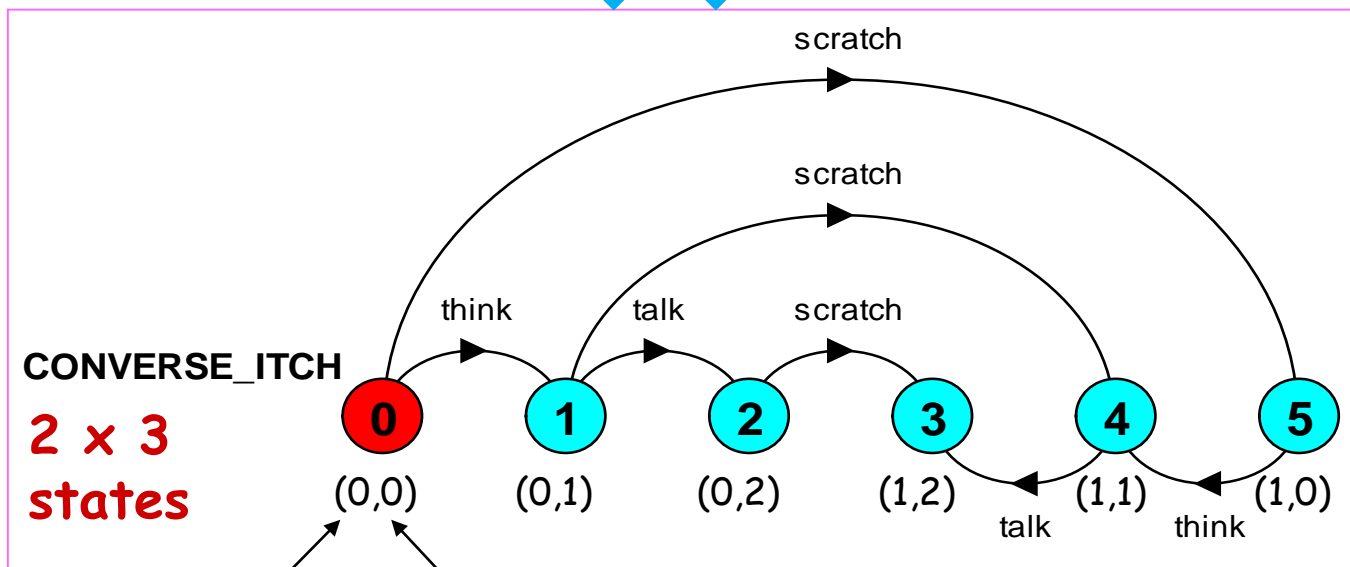
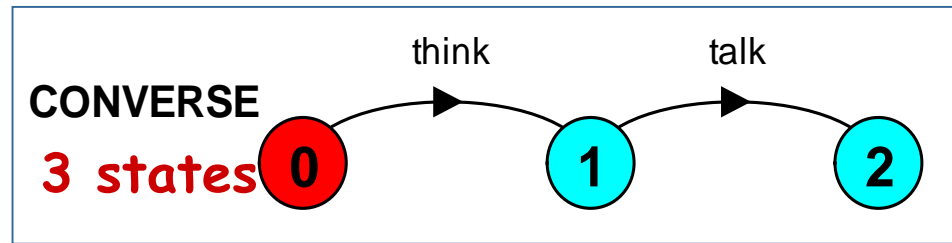
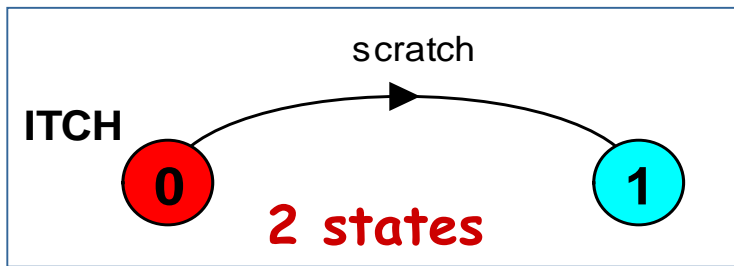
インタリーブで可能となるトレース

```
CONVERSE_ITCH = {  
  think → talk → scratch  
  think → scratch → talk  
  scratch → think → talk  
}
```

インタリーブで不可能なトレース

```
talk → think → scratch  
talk → scratch → think  
scratch → talk → think
```

並列合成の結果



from ITCH

from CONVERSE

Pの状態pとQの状態qの組(p,q)が,
P||Qの状態となる

共有アクションによるインタラクションと同期

共有アクション: 並列合成されたプロセスがもつ共通のアクション
shared action

共有アクションで, プロセスの**インタラクション**(相互作用)をモデル化
interaction

共有アクションは, それを共有する全プロセスにおいて同時に**同期実行**
synchronization

インタラクションの例

共有アクション
ready, used

MAKER = (make → ready → used → MAKER).

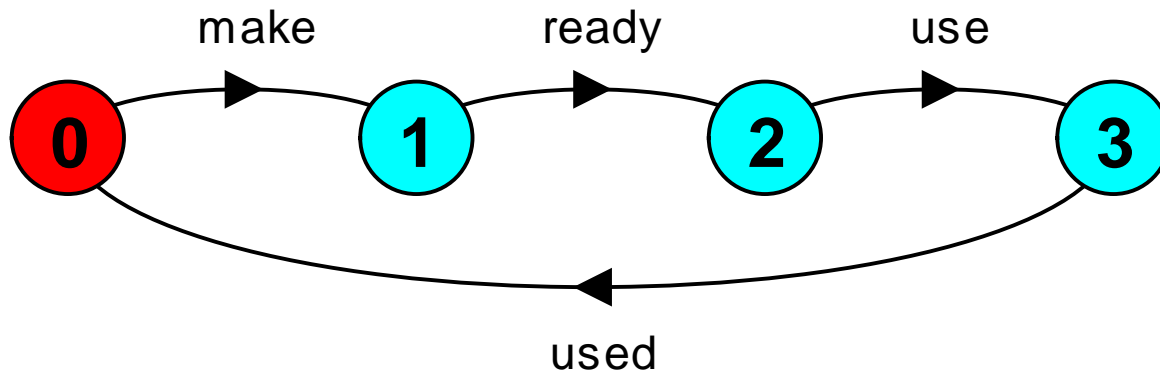
USER = (ready → use → used → USER).

|| MAKER_USER = (MAKER || USER).

3 状態

3 状態

3 × 3 状態?



4 状態

インタラクションは
振舞いを制約する

演習問題 7

二人のユーザーを表すプロセス USER_A, USER_B,
共有資源を表すプロセス RESOURCE,
それらを並列合成したプロセス RESOURCE_SHARE
が、つぎのように定義されている。

USER_A = (a_acquire -> a_use -> a_release -> USER_A).

USER_B = (b_acquire -> b_use -> b_release -> USER_B).

RESOURCE = IDLE,

IDLE = (a_acquire -> BUSY | b_acquire -> BUSY),

BUSY = (a_release -> IDLE | b_release -> IDLE).

|| RESOURCE_SHARE = (USER_A || USER_B || RESOURCE).

ユーザーAが資源を獲得(a_acquire)しているときにはユーザーBは資源を獲得(b_acquire)できない仕組みになっていることを説明しなさい。

また、これら4つのプロセスの状態遷移系をそれぞれ描画しなさい。
