



モデル検証入門 ～ツールに振る舞いを検査させる～

国立情報学研究所 石川冬樹

many thanks to 長久 勝さん

f-ishikawa@nii.ac.jp

2008/09/10



概要

システム検証の難しさ

不具合が起きる状況の検証漏れ

多数のフラグによる
複雑な条件分岐により
発生する状況

並列・分散システムで
特定の実行タイミング
でのみ発生する状況



モデル検査

基本的な概念を紹介

ツールを用いて網羅的な検査を行う手法

デッドロックしない？

進めなくならない？

リクエストが放置されない？



1. モデル検査を知る
2. モデル検査を使う
3. モデル検査を役立てる

1. モデル検査を知る



システムを検証する

「設計や製品（実装）が期待される性質を満たすか」調べ、示す

- 「正しさの基準」に対する相対評価
 - 仕様として定められる性質
 - 「デッドロックしない」といった基本的な性質
- 様々な手段の組み合わせ
 - ピアレビューとテストが主流

➡ 「誤り」が起きる可能性の検出漏れをどう効率よく少なくする？



システムを検証する

最近注目されている

形式手法

(少なくともツールが) 理論に基づき
高品質なシステムを効率よく開発

(検証以外にもいろいろな側面があるが・・・)

- 検証方法 1 : 観測ベース「モデル検査」

今日は
こっち

「性質が成り立つかすべての場合を調べる」

(変数の値, スレッド切替タイミング, ...)

- 検証方法 2 : 法則ベース「定理証明」

「性質が成り立つことを理屈で説明する」

(整数の性質, 三段論法, ...)



進行の非決定性が難しい

特定の実行の分岐・順序で発生する誤りを把握するのが難しい！

■ 並列・分散システムにおける相互作用

- 複数スレッドが異なる資源のロックを確保して、互いに待ち合ってデッドロックする

- 複数ノードがほぼ同時に互いに対しメッセージを送り、返事を待ち合ってデッドロックする

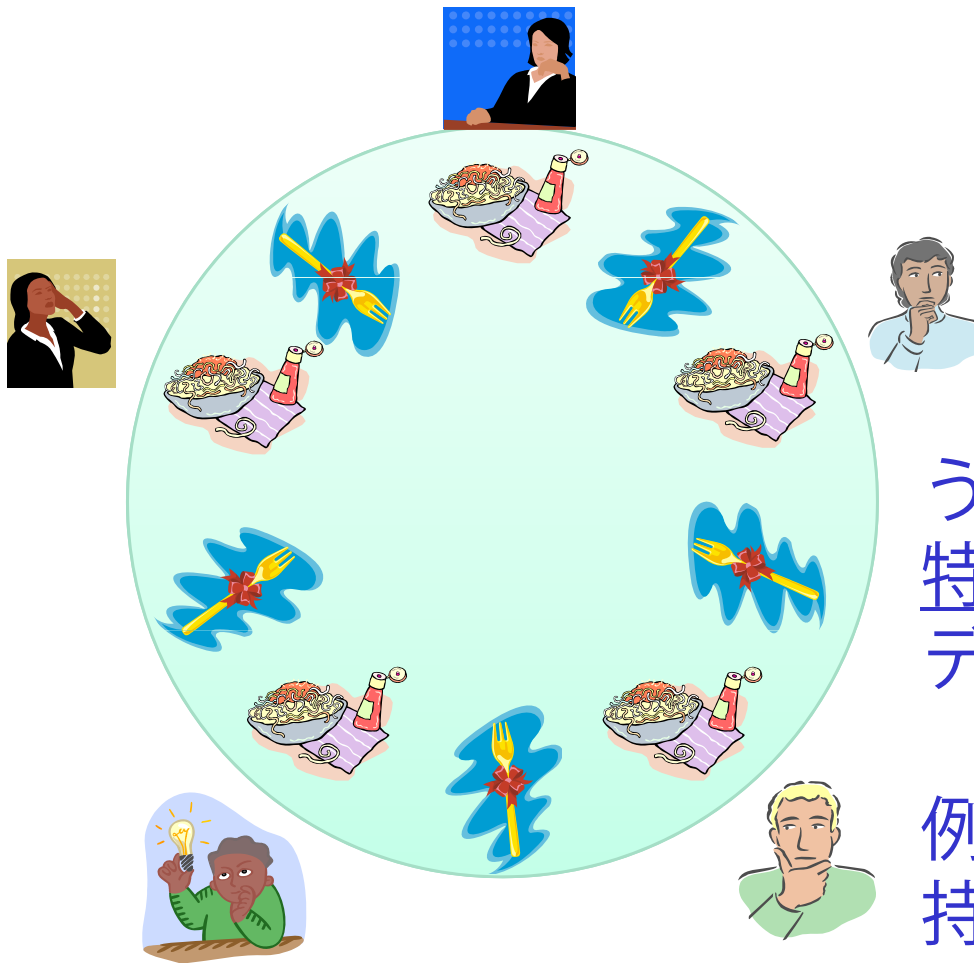
■ 複雑な条件分岐

- 多数の行動・フラグの管理において、想定外の操作により予期しない状況に到達してしまう



進行の非決定性が難しい

有名な例：食事する哲学者



左右どちらかのフォークをとった後、もう片方を取り、食事をしてフォークを戻す
(会話はしない)

うまく行動を定めないと、特定の振る舞いの順序で
デッドロックやライブロック

例：全員が左のフォークを持ち右が空くのを待ち続ける



網羅的に調べる？

- 進行の可能性を網羅的に調べればよい？
 - 人手でレビューするには複雑過ぎ、思い込みや誤りが混入してしまう
 - 実際のプログラムの実行（テスト）では実質的に不可能
 - 単純に「すべての可能性」を調べようとしたら、設定も実行もいくら時間があっても足りない
 - 並列・分散システムではそもそも意図的に進行の順序を制御、再現できず、起きうる状況を洗い出せない



モデル検査が使える

モデル検査

注目する側面（システムの振る舞い）を
抜き出した「モデル」に対し、
ツールを用い厳密、網羅的な検査を行う

おおざっぱには2通りの使い方

- 設計モデルを検証

「検証・修正コストがはるかに小さい」

- 実装から特定の動作を抜き出し検証

「手軽に使い、ポインタ等のバグも扱える」



モデル検査を使う

システムの振る舞いを記述する

「ボタン1が押されるとランプ2が赤になる」
「クライアントからのリクエストは同時に1つだけ処理し残りは処理待ちでブロックされる」

システムの性質を記述する

「デッドロックしない」
「リクエストを送るといつか返事が来る」

ツールに検証させる

「性質は満たされる」
「この順序で処理が実行された場合にデッドロックが起きる」 (性質を満たさない反例)



モデル検査は使われている

- 25年以上の歴史を持つ技術
 - 代表的なツール： SPIN
 - ACM Software System Award 2001を受賞
(2000： Apache, 2002： Java, 2003： MAKE)
 - 研究開発グループは2007年チューリング賞受賞
 - 産業界での様々な応用事例
 - ネットワークプロトコル検証が代表例： 多くの誤りを発見し標準の修正につながったり, 17年見つからなかった誤りを発見したり
 - 検証対象の状態数は 2^{30} , 2^{100} といったオーダーに



モデル検査を使う・・・？

- モデル検査を知り，活用するために
 - 振る舞いをどう記述する？
 - 検証する性質をどう記述する？
 - どう検証する？
 - 何に気をつけないといけない？
 - ゲーム開発にどう使える？



1. モデル検査を知る
2. モデル検査を使う
 1. 振る舞いを書く
 2. 性質を書く
 3. ツールを使う
3. モデル検査を役立てる

2. モデル検査を使う

1. 振る舞いを書く



振る舞いを書く

注目する側面：システムの振る舞い

「どういう条件で何が起こる可能性があり、状態がどう変化するのか？」



「オートマトン」形式で表現可能

- 様々なツールがありそれぞれの記述言語があるが、「表現されている情報」はほぼ共通
 - 正確には「**オートマトン」といった区別があるがここでは共通した考え方を見してみる
 - 実際はツールごとの記述言語（や図形式）を使う

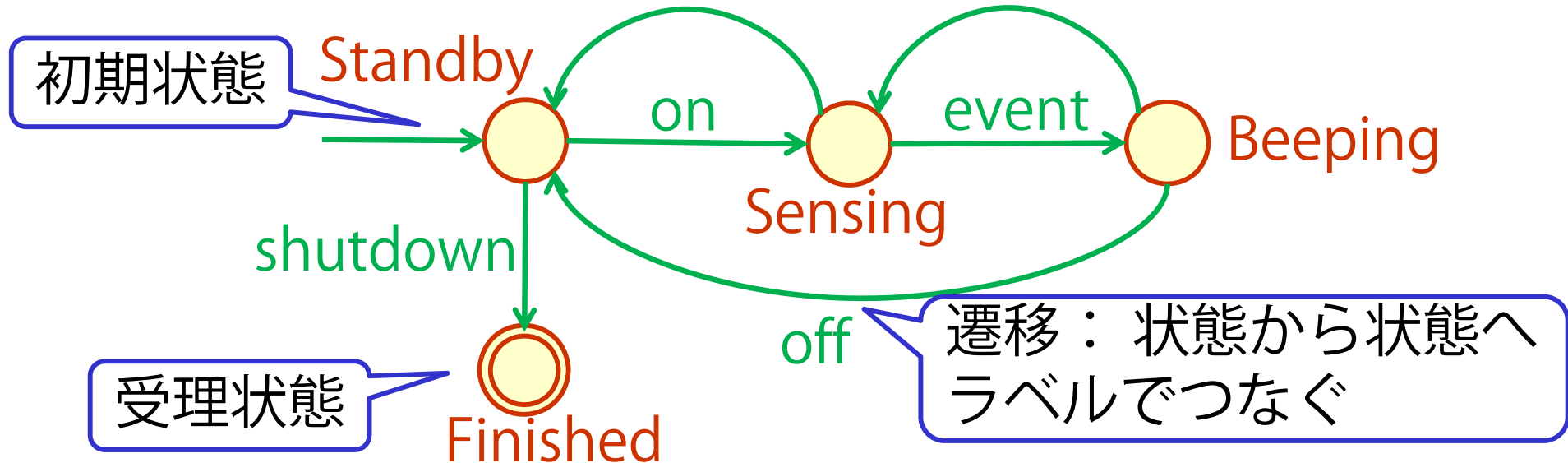


振る舞いを書く

オートマトン

- 全状態, 初期状態・受理状態それぞれの集合
- 状態遷移のラベル・状態遷移の集合

アラームの例

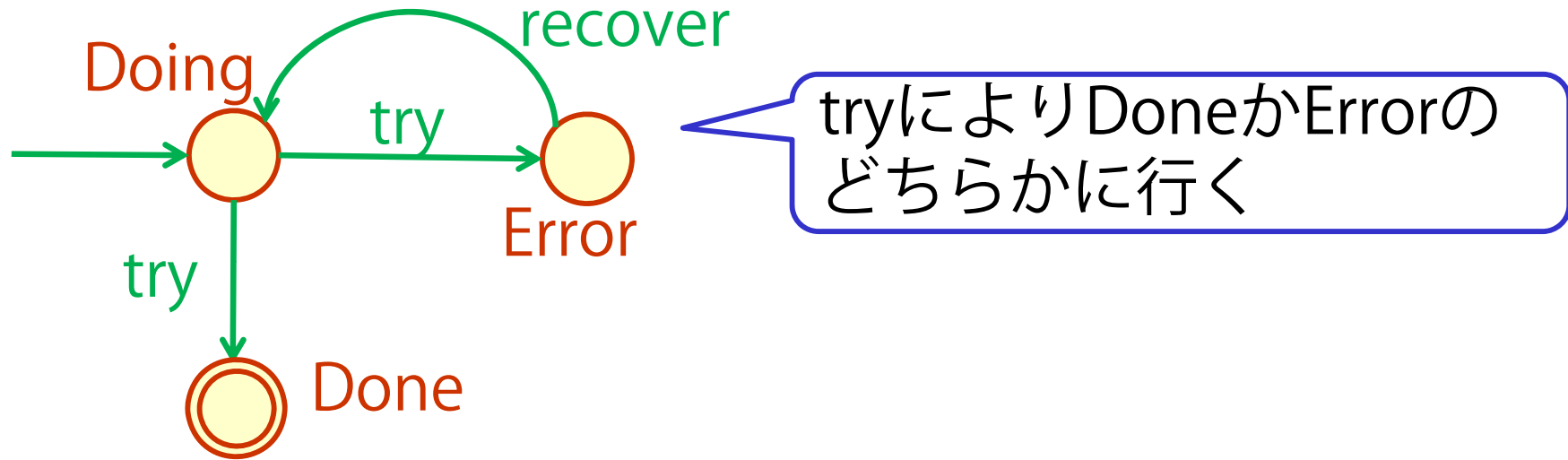


初期状態から受理状態に至るラベル列の例
on . event . stop . off . shutdown

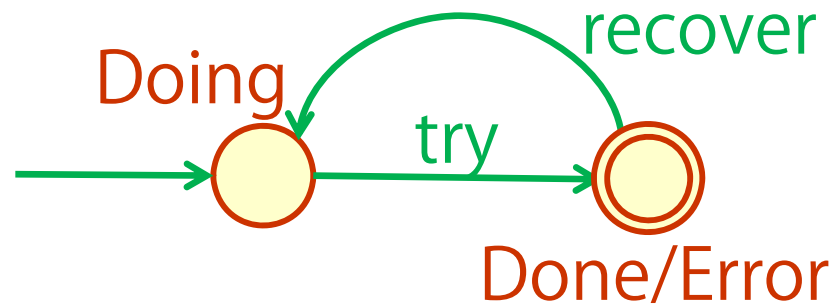


振る舞いを書く

非決定的な振る舞いを書く



(ちなみに同じラベル列を受理するような決定的な書き方に変換できる)



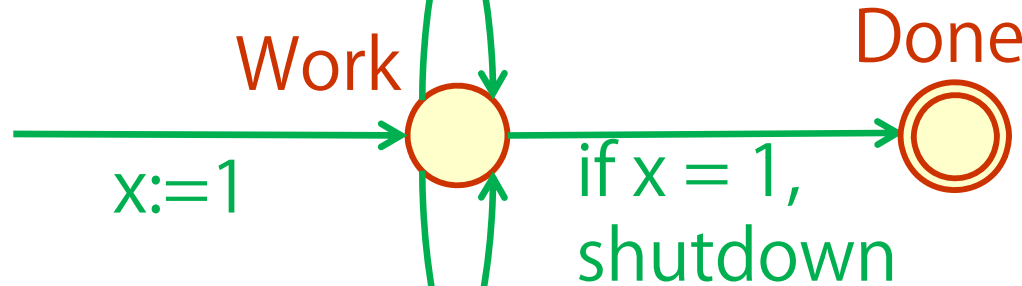


振る舞いを書く

変数による遷移を書く

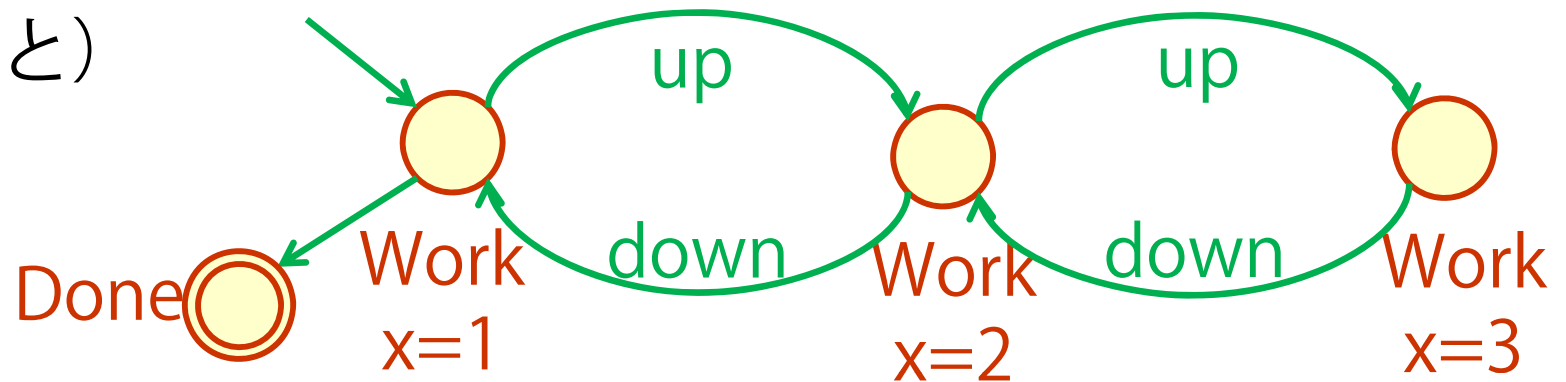
if $x < 3$, up, $x := x + 1$

$x < 3$ ならばupして
 x を1増やす



if $x > 1$, down, $x := x - 1$

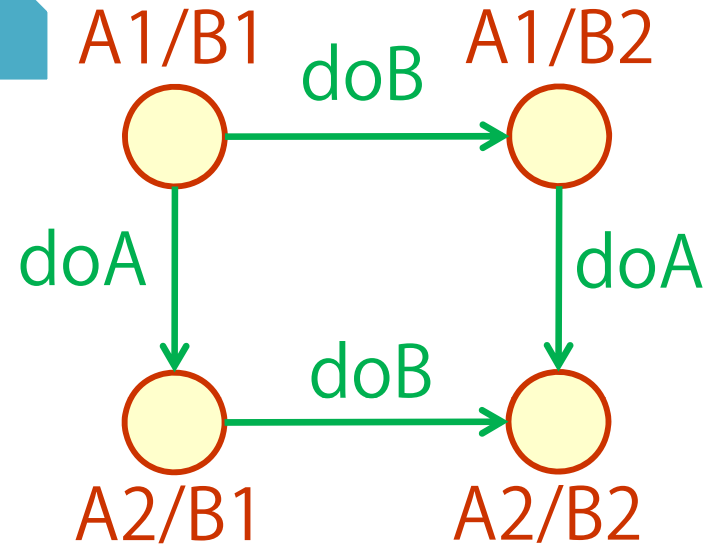
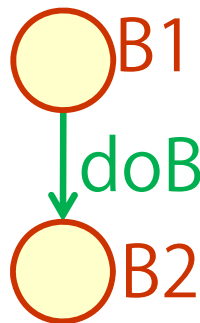
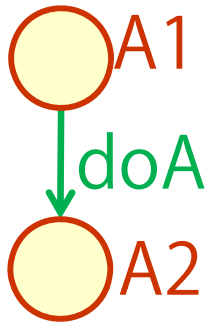
(展開すると)



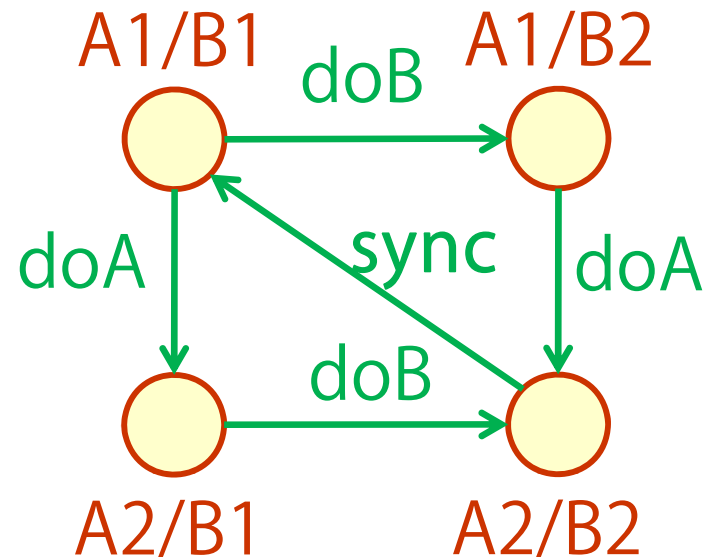
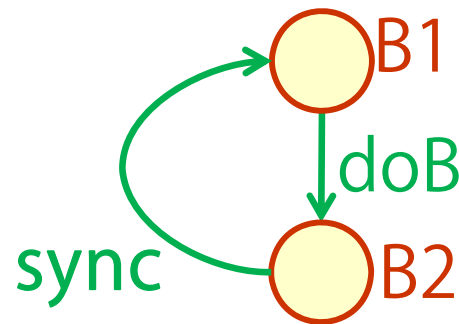
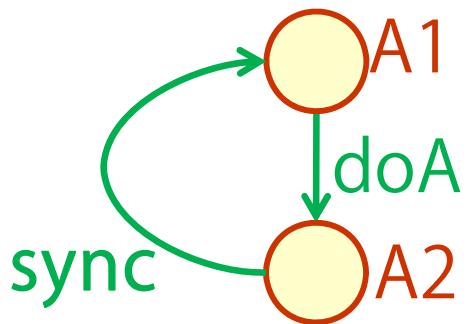


振る舞いを組み合わせる

様々な順序でそれぞれ遷移する



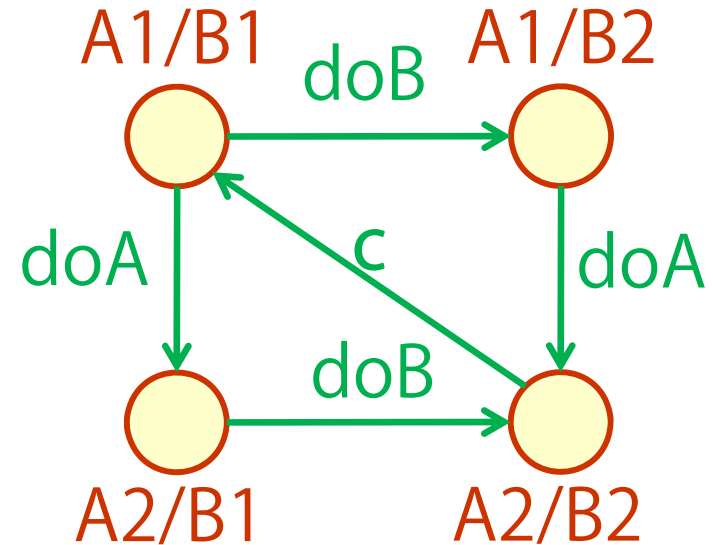
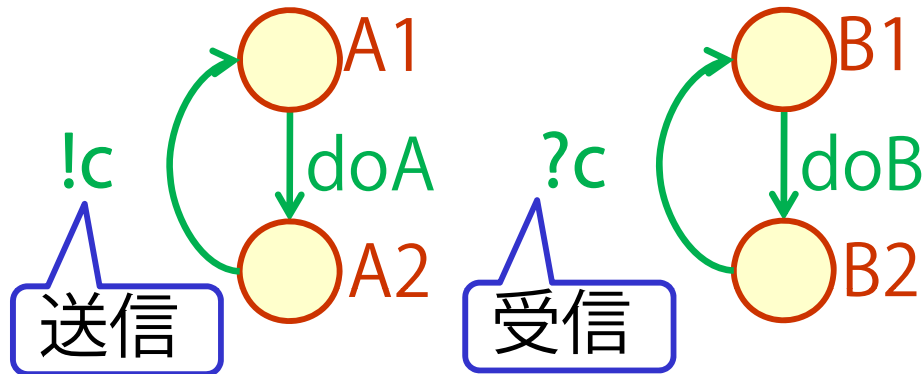
共通のラベルで同期する



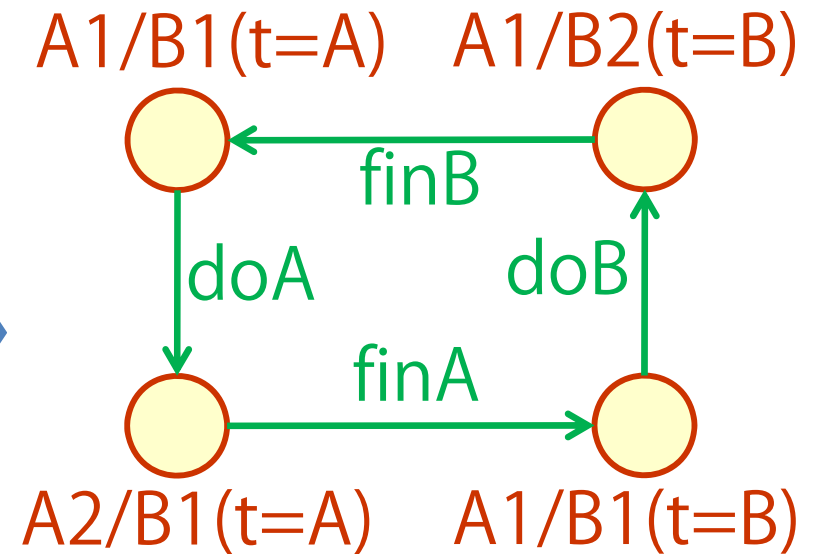
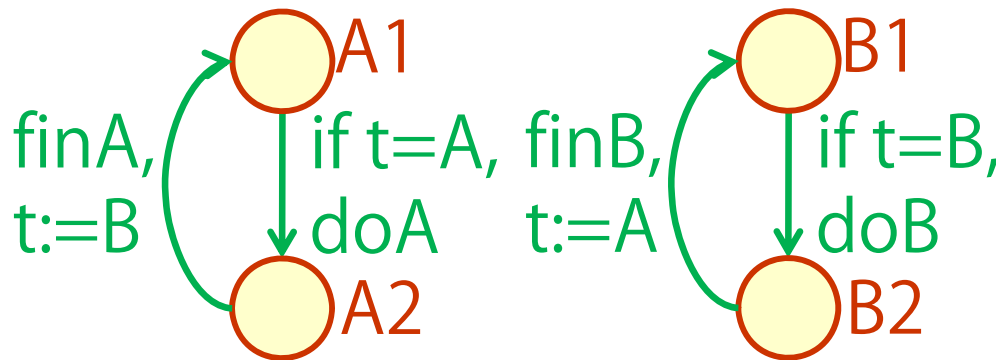


振る舞いを組み合わせる

ラベルの対で同期する



共通変数を用いる





実際の言語で書く

SPINにおける振る舞い記述イメージ (Promela言語)

通信チャンネル
(バッファ長と
データ型)

```
mtype { msg1, msg2 }  
byte count = 1;  
chan c1 = [1] of {mtype}  
proctype SampleSender{
```

列挙型定義

プロセス定義

ループ

```
do  
  :: (count == 0) -> c1 ! msg1; count++  
  :: (count != 0) ->  
    c1 ! msg2;  
  if  
    :: count++  
    :: count--  
  fi  
od  
}
```

チャンネルc1を
通してmsg1 (msg2)
を送信

「どれかを実行」

条件付きの場合
条件を満たすもの

実行可能なものが
なければブロック,
複数あれば非決定
的に選択



1. モデル検査を知る
2. モデル検査を使う
 1. 振る舞いを書く
 2. 性質を書く
 3. 検証する
3. モデル検査を役立てる

2. モデル検査を使う

2. 性質を書く



実行の経過を調べたい

オートマトンに対して考える性質

どういう条件で、どういう状態に
到達するか（しないか）？

- 実際のシステムで検証したい性質
- ➡ よく知られた分類・パターンがある
- 性質を記述する方法
- ➡ 検証方法の効率とセットで検討されている
 - 特定の性質を検証する機能
 - 論理式で書かれた性質を検証する機能



実行の経過を調べたい

典型的な性質

Reachability	ある条件の下ある状況に到達しうる 「初期画面に戻る操作の列が常にある」
Safety	ある条件の下ある状況に到達することがない 「AとBが同時に仲間であることはない」
Liveness	ある条件の下ある状況にいつか必ず到達する 「登録をするといつか必ず記念品が届く」
Deadlock-freeness	デッドロックが起きない
Fairness	ある条件の下ある状況が無限回起きる（ない） 「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）」



性質をきちんと書く

時相論理

実行の経過に関する論理式を記述

「かつ」「または」「ならば」
+ 「いつも」「いつか」「までは」...

もっともよく用いられている2種類

■ LTL (Linear Temporal Logic)

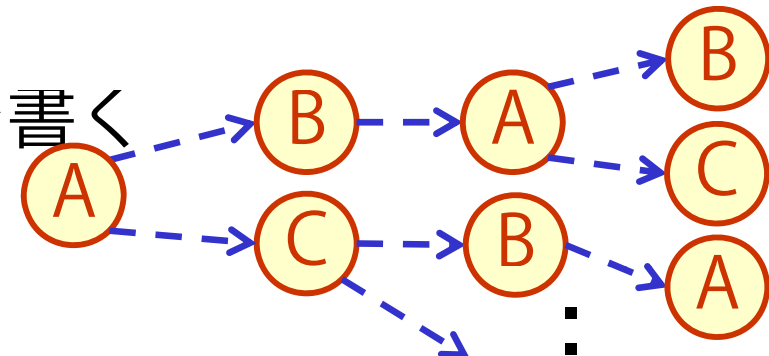
A . B . A . B

実行を「線」と見て性質を書く



実行を「木」と見て性質を書く

今日はこっちはおまけ資料
(配布資料末尾)





実行パスを見る

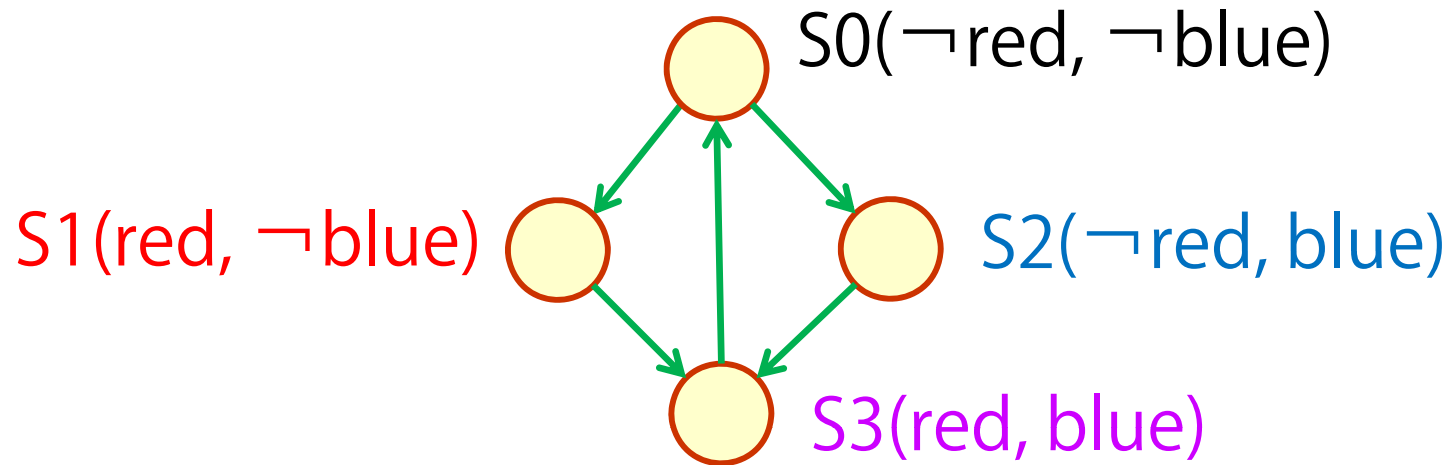
LTL

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

$X p$ ($\bigcirc p$)	次の状態でpが成り立つ (next)
$F p$ ($\blacklozenge p$)	今の状態かそれ以降のどこかでpが成り立つ (finally)
$G p$ ($\square p$)	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)



実行パスを見る



(各状態で成り立つ, これ以上分解できない
基本的な性質に注目している)

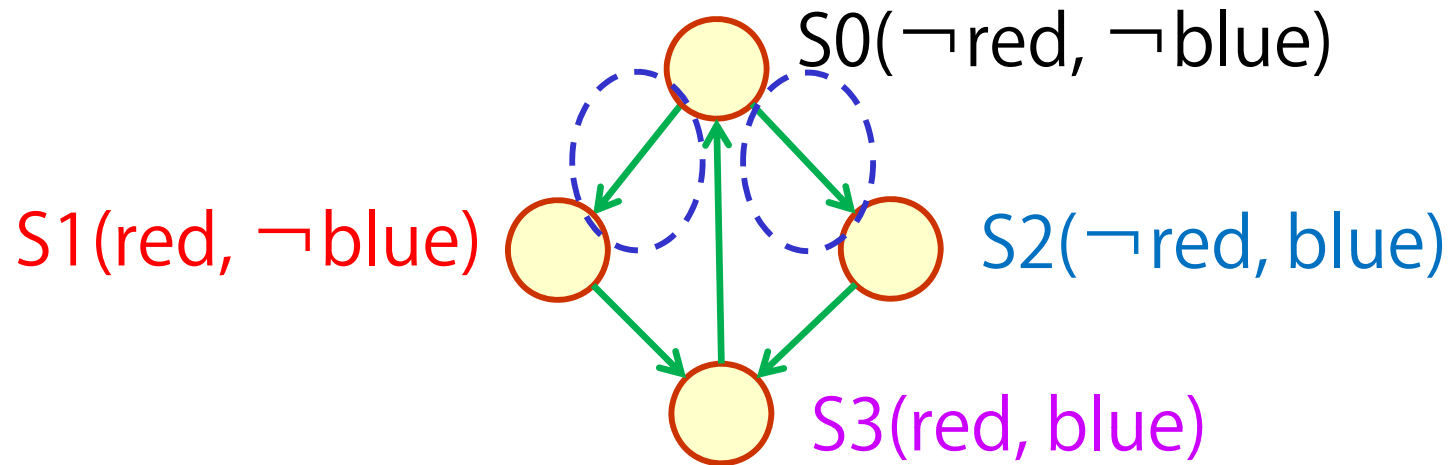
S0から始まる実行パスの例
(通過する状態の無限列)

S0 . S1 . S3 . S0 . S1 . S3 . S0 . S1 . S3 . …

S0 . S2 . S3 . S0 . S1 . S3 . S0 . S2 . S3 . …



実行パスを見る



S0から始まる実行パスを考えたときの例

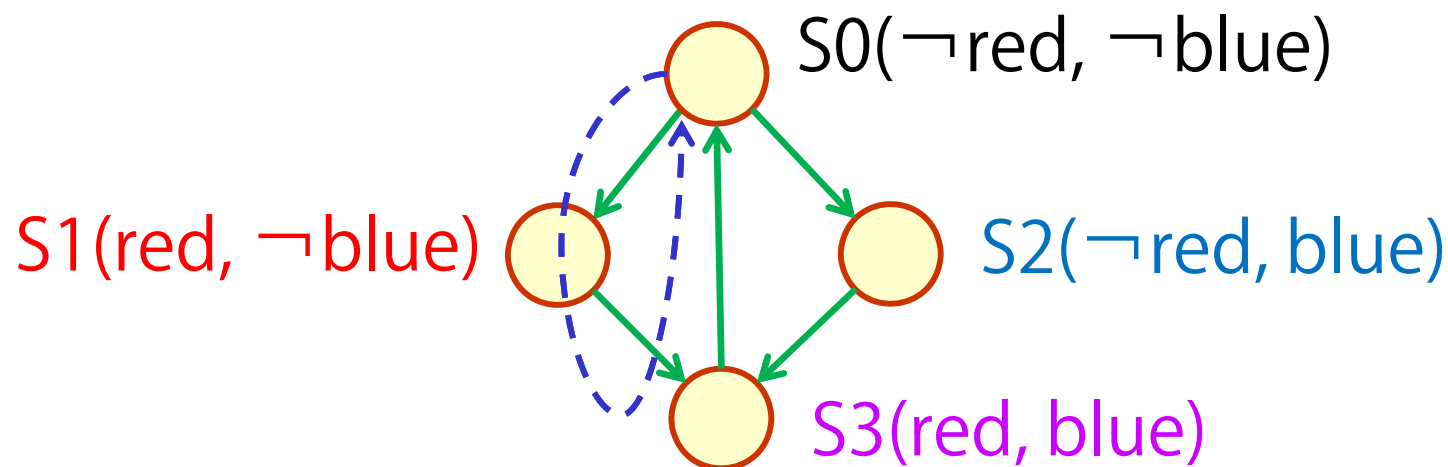
■ X (red \vee blue)

「次の状態ではredまたはblueが成り立つ」

➡ どの実行パスでも成り立つ



実行パスを見る



S0から始まる実行パスを考えたときの例

■ $F (\neg \text{red} \wedge \text{blue})$

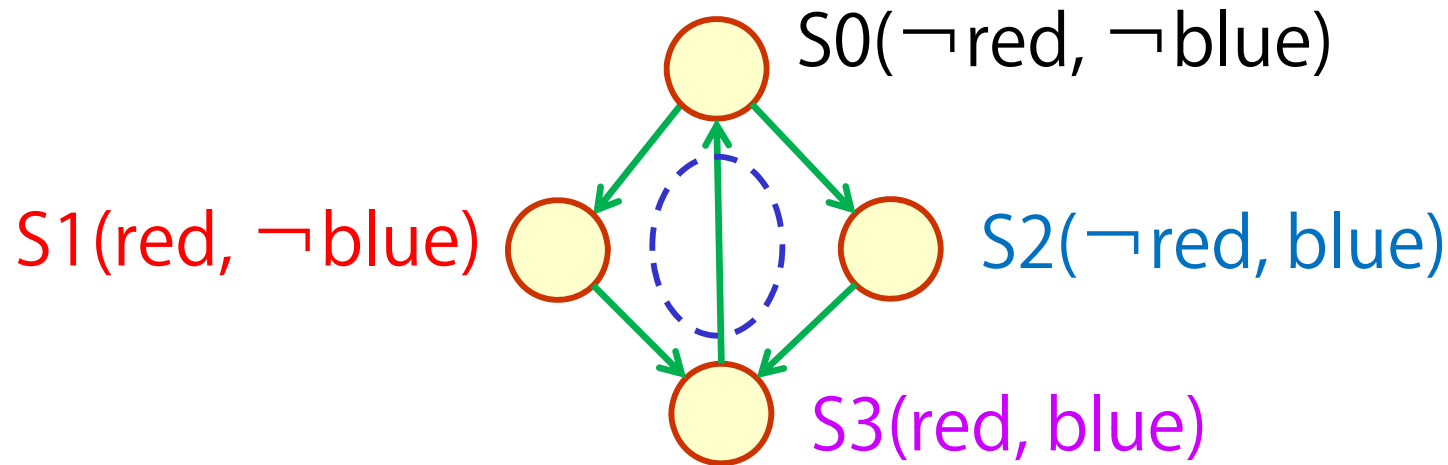
「redではなくblueである状態にいつかなる」

➡ 満たさない実行パスがある

(S2を通らずにS1を経由し続けるパス)



実行パスを見る



S0から始まる実行パスを考えたときの例

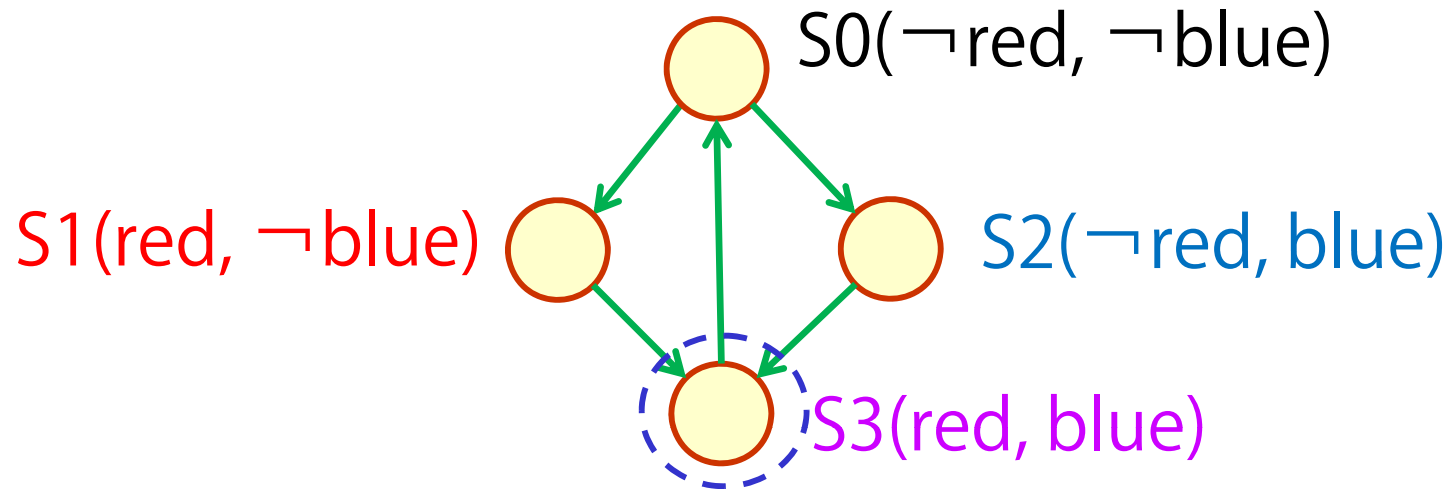
■ $G((red \wedge blue) \Rightarrow X(\neg red \wedge \neg blue))$

「実行パス上のどの状態でも、今redかつblueであるならば、次はredでもblueでもない」

➡ どの実行パスでも成り立つ



実行パスを見る



S0から始まる実行パスを考えたときの例

■ G F (red \wedge blue)

「実行パス上のどの状態から見ても, redかつ blueである状態にいつかなる」

➡ どの実行パスでも成り立つ



1. モデル検査を知る
2. モデル検査を使う
 1. 振る舞いを書く
 2. 性質を書く
 3. 検証する
3. モデル検査を役立てる

2. モデル検査を使う

3. 検証する



検証・「デバッグ」する

■ 振る舞いと性質をツールに与えて検証

➡ ① OK, その性質は成り立つ

➡ ② NG, その性質が成り立たない場合がある

➡ 振る舞いを修正する or 性質を修正する

➡ ③ ??, 状態数が大きすぎて検査しきれない

➡ 状態数やメモリ利用を減らす工夫をする



成り立たない場合を調べる

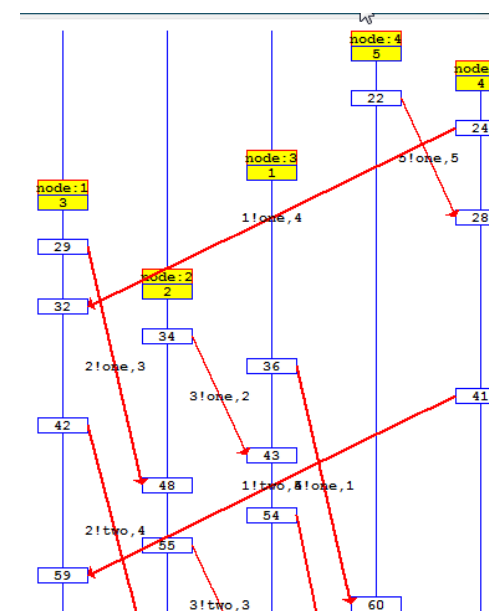
■ 例：SPINの機能を用いた場合

■ 検証において、与えられた性質が成り立たない実行パス（反例）を検出

➡ 性質が成り立たない状況に、最も少ないステップで到達する実行パスを検索

➡ その実行パスを図示

➡ 振る舞いまたは性質を修正





何を修正する？

- 振る舞い（例：ロック確保手順）でなく性質を修正する場合も多い

- $G(\text{user1_req} \Rightarrow F \text{user1_res})$

「どの状態にいても、user1からのリクエストが起きているならその後必ずuser1への返信が起きる」

- ➡ 状況例：エラーが返る場合を見落とし
- ➡ 求めている性質が強すぎたので修正

$G(\text{user1_req} \Rightarrow (F \text{user1_res} \vee F \text{user1_err}))$

「返信またはエラー通知が必ず発生」



何を修正する？（続）

- 別の状況例：他のユーザによる占有の可能性
 - 「他のユーザuser2がリクエストを無限に送り、user1のリクエストがブロックされ続ける」
- ➡ 振る舞いを強化することも考えられる
 - 「ブロックされている他のユーザがいる場合、同じユーザのリクエストを続けて処理しない」
- ➡ 性質を緩めることも考えられる
 - サーバの制御機能や運用ルールで解決するとして「user2が無限リクエストを送らなければ」とする
 - $\neg G F \text{ user2_req} \Rightarrow G (\text{user1_req} \Rightarrow F \text{ user1_res})$



問題を小さくする

■ 様々な手法やツール設定により状態削減

■ 検証シナリオの設定

- 例：ユーザは2名に限る

■ 様々な抽象化手法

どの解説本でも
「抽象化」の章がある

- 例：int型変数 x があり，条件分岐では $x > 0$, $x = 0$, $x < 0$ の3つの場合を考えている

➡ $x = \{\text{neg}, \text{zero}, \text{pos}\}$ と列挙型に変更

```
x := 0    → x := zero
if x > 0  → if x = positive
x++      → if x = neg, x := neg or x := zero;
           else if x = zero, x := pos;   else skip;
```

非決定的にどちらか



問題を小さくする

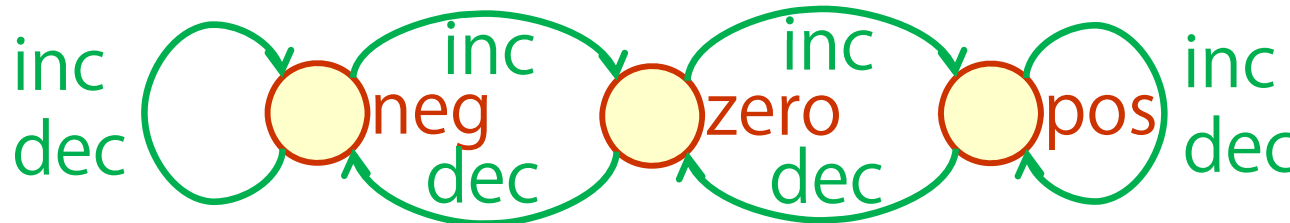
■ 検証の意味が変わるのが難しい！

■ 例：ユーザは2名に限る

➡ 3名が関わって初めて起きる不整合はない？

■ 例：int x を $x = \{\text{neg}, \text{zero}, \text{pos}\}$ に

➡ 制約となる情報減 → 遷移の可能性増



-1でも-100でもneg
→ 1回のincで
zeroに飛びうる

➡ 今回は抽象版のモデルでしか起きないことがあり、
抽象版で起きないことは元のモデルでも起きない



実際のツールを使う

■ 様々なツールがある

■ モデル検査

SPIN, SMV, LTSA, Design/CPN, . . .

■ ソフトウェアモデル検査 (プログラム検証)

CBMC, Java PathFinder, VARVEL, . . .

■ リアルタイムモデル検査 (今日は割愛)

UPPAAL, KRONOS, . . .

「通信に1秒以上かかる」 「9秒以内に返事が来る」



実際のツールを使う

- ツールの例： SPIN （Bell Lab, 1980）
 - 代表的なモデル検査ツール
 - 25年以上の歴史，近年各種の受賞
 - ついに日本語の解説本が
 - 豊富な機能
 - デッドロック検証，進行性検証，LTL式検証
 - 最短ステップの反例探し
 - GUIによる性質記述（起きるべきシナリオ記述）
 - Cコードの埋め込み



実際のツールを使う

- ツールの例：VARVEL (NEC, 2007)
 - ごく最近のCコード向けモデル検査ツール
 - 最近の (国内での研究開発の) 話題なので紹介
 - Cコードから性質に関連する部分のみ抜き出してモデルを構築し, (ある定義された範囲内で) 網羅的に検証
 - ポインタ管理, メモリ管理等Cにありがちなバグ
 - 変数の値が常にある条件を満たすか
 - 関数の実行後に常に期待された状況になるか



ツールの中を知る

- 各ツールは特定の検証手法を効率よく実装

- ある性質に対する検証 (例: デッドロック)

- ある記述言語に対する検証 (例: LTL)

ほんとにすべての実行パスを
見てるわけではない (無限列だし)

- オプションで細かい設定もできるので, 最適化手法を知っておくともっと使いこなせる

- Partial Order Reduction (状態数削減方式)

- 「遷移の起きる順序を変えても性質に影響しない部分は, 起きる順序を網羅しない」

- Binary Decision Diagram (メモリ詰め込み方式)



1. モデル検査を知る
2. モデル検査を使う
3. モデル検査を役立てる

3. モデル検査を役立てる



何が嬉しい？何が難しい？

嬉しさ

- システムのある部分のみに対し適用できる
- レアな、気づきにくい不整合も検出できる
- 反例の分析も支援されている

難しさ

- ある程度の専門的知識を要する
- 状態爆発の問題が伴う
- 数値やデータ構造はうまく扱えない
- 実システムの「近似」の検証でしかない点に注意して意味を考える必要がある



「モデル」が肝になる

■ 抽象化・単純化の嬉しさ

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（Oxford English Dictionary）

■ 注目する側面に特化し， 抽象化・単純化

→ ツールが効率的に， 網羅的に検査を行える

→ 人間も重要な（難しい）本質に集中して
問題の明確化， 分析を行える



「モデル」が肝になる

■ 厳密さ・正確さの嬉しさ

モデル：ある論理式を満たす解釈（構造）
（数理論理学）

- システム記述が与えられた性質の「モデル」であるか厳密な理論に基づき「検査」する
- ➡ ツールによる検証は厳密，系統的に行われ，その結果の意味は明確に定まっている
- ➡ 記述過程において，要求・仕様の曖昧さ・不明確さ・不整合が排除される

この効果も実は大きい



「モデル」が肝になる

ツールに対する「指示」の、目的・意図に対する意義・妥当性の確認は開発者の責任
ツールは与えられた「正しさ」を検証

■ 抽象化・単純化の難しさ

- モデルは表現したい側面を十分に含めている？
- 満たす性質・満たさない性質が変わっていない？

■ 厳密さ・正確さの難しさ

- 表現されていることは意図していることか？
- 結果のOK/NGの意味するところは何か？

P ⇒ QはOK！ となったが P が成り立たないだけだったり



ゲーム開発に使う？

(ゲーム開発素人の疑問)

- 一般的な使い方はできそう
 - 並列・分散処理の設計モデル検証
 - ➡ 検証コストをかける価値が特に大きいのは？
 - ➡ 複雑なオリジナル部品を作り使い回す場合 (シリーズものの共通部品とか？)
 - プログラム検証 (ポインタ・メモリ管理)
 - ➡ デバッグ強化に手軽に導入でき、Cの利用が多そうなので一番とっつきやすい？



ゲーム開発に使う？

(ゲーム開発素人の疑問)

■ 特にRPGでのシナリオ・フラグ管理はまさにモデル検査の活きるところでは？

■ 常に状態が変わり続け、フラグが複雑

「ちょっと遷移、いつもの状態に戻って繰り返し」
ではなく、昔の多くのフラグに依存した遷移

■ ユーザが想定（シナリオ一直線）から外れた
様々な行動選択をとれる どう調べてるの??

「ある町に入らず先の町に行こうとしてみたり、イベント発生中に戻ったり移動呪文唱えたり」



ゲーム開発に使う？

(ゲーム開発素人の疑問)

■ より手軽に使うためには

■ 「橋渡し」を用意することも多い

「UMLステートチャートから振る舞いモデルのテンプレートを生成」

「プログラムからOSのAPI呼び出しを抜き出して振る舞いモデルを生成」

➡ ゲーム開発においては？シナリオ記述から振る舞いモデルを生成？

(例：長久さんのイントロでの事例)



最後に

モデル検査

システムの振る舞いを抜き出し、
ツールを用い徹底、網羅的な検査を行う

- ソフトウェア一般では最近注目を集めている
 - 近年の品質要求の高まり
 - ツールの効率化とマシン的高速化、ノウハウの蓄積による実用性の向上

最初の一歩、まずはとにかく使ってみて、
ゲーム開発に使えるか考えてみませんか？



参考文献（最小限）

- SPINモデル検査—検証モデリング技法
中島 震, 近代科学社, 2008
待望のSPIN日本語本！
- 特集「フォーマルメソッドの新潮流」
情報処理学会 学会誌2008年5月号（Vol.49 No.5）
VARVELの解説記事を含む, 形式手法関連の7つの
記事
- 英語の良書多数については下記サイトに載せます
<http://fmug.grace-center.jp/>



参考文献 (最小限)

- SPIN <http://www.spinroot.com/>
- NuSMV <http://nusmv.fbk.eu/>
- LTSA <http://www.doc.ic.ac.uk/ltsa/>
- CBMC
<http://www.cs.cmu.edu/~modelcheck/cbmc/>
- Java Pathfinder
<http://javapathfinder.sourceforge.net/>
- UPPAAL <http://www.uppaal.com/>



今日の情報

■ 日々mnagaku

<http://kgr-lab.ddo.jp:10880/wordpress/>

イントロをしてくださった長久さんのサイト。ゲーム適用事例を含むイントロ資料はこちら。

■ 形式手法実践ポータル

<http://fmug.grace-center.jp/>

モデル検査を含め形式手法に関する日本語の情報が少なすぎるところがあるので、勉強がてら少しずつまとめていっています。今日の内容の先を知りたい場合ここからたどれるようにします。



宣伝

- トップエスイー <http://www.topse.jp/>
開発者向けのソフトウェア工学教育コース
講義（座学＋グループ演習）
修了制作（実問題への適用）

本講演や午前の「開発方法論」の内容をきちんと
今年度まで文科省のプロジェクト，来年度は自立

1年コース，年間50万円程度
基本は企業から国立情報学研究所への委託
個人応募もあり（特待生制度あり）

今月 Webに情報公開

10月 説明会

1月 選考

4月 コース開始



ご清聴ありがとうございました

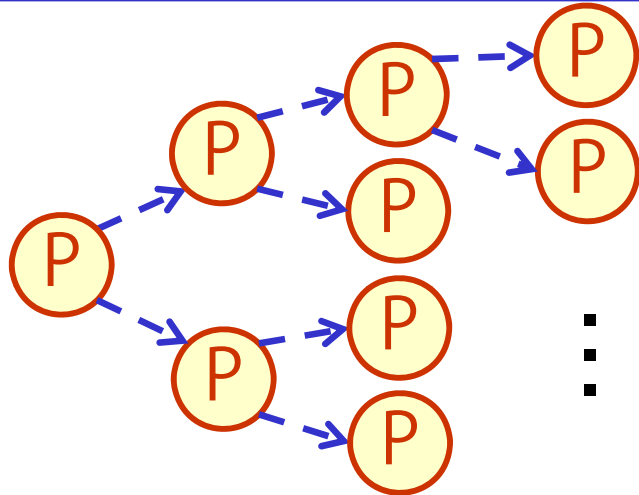


おまけ： CTL

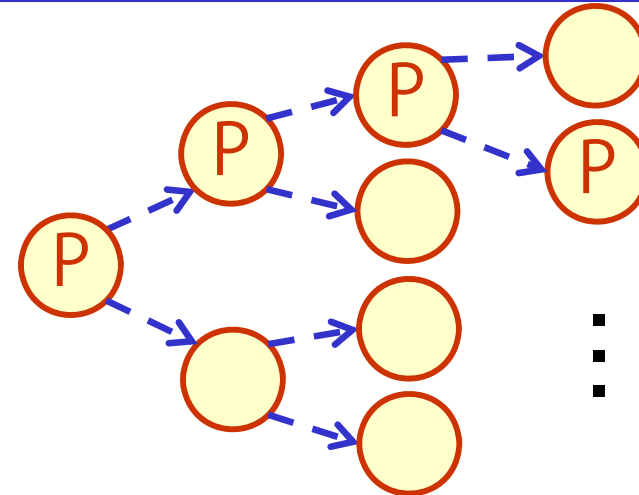


実行の分岐を見る

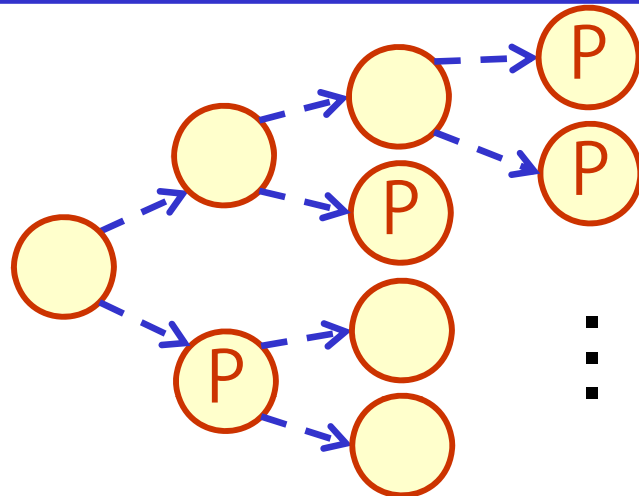
AGP どのパスでも常に



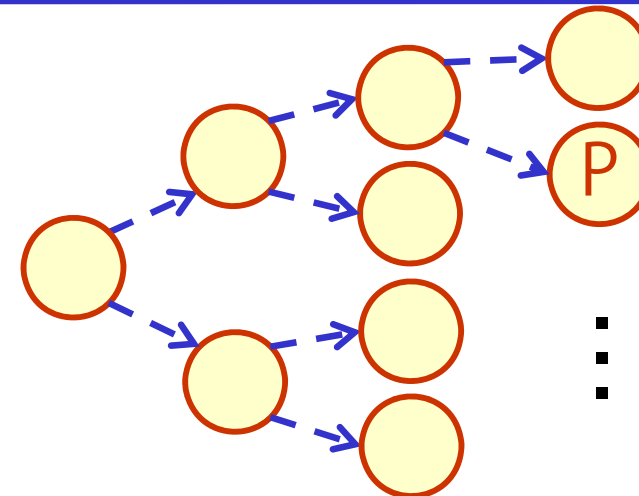
EGP あるパスでは常に



AFP どのパスでもいつか

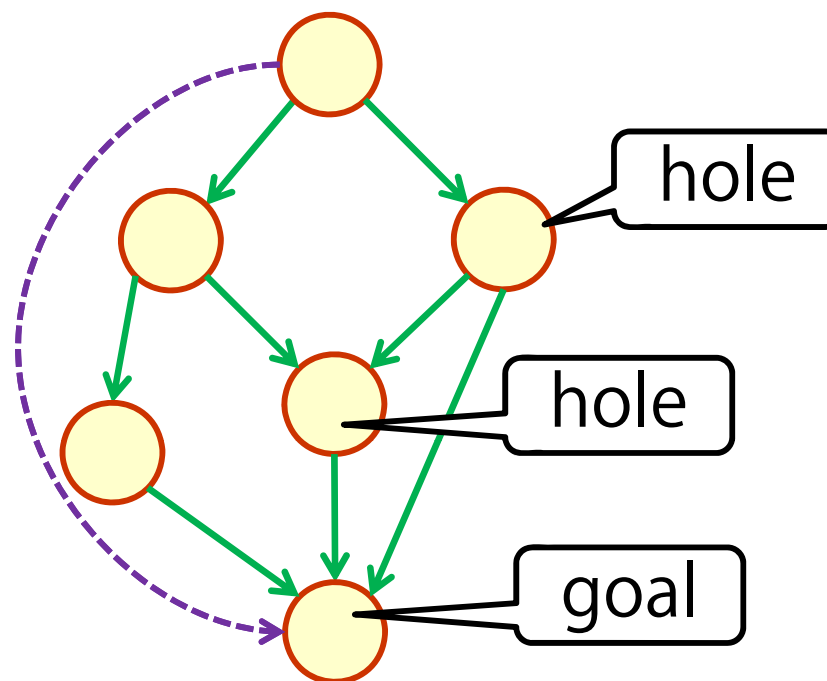


EGP あるパスではいつか





実行の分岐を見る



■ $E (\neg \text{hole} \cup \text{goal})$

「goalとなるまでholeではない状態が続くようなパスが存在する」

➡ 成り立つ



LTLとCTL

- LTLでは将来の分岐の可能性に言及できない
- ➔ Reachabilityの一部はLTLでは直接書けない
(CTL) AG EF goal
「どのパスに行っても常に、あるパスからいつかgoalに到達できる」
- CTLではFとGを連続して書けない
- ➔ Fairnessの一部はCTLでは直接書けない
(LTL) GF open
「どの状態でもいつかopenが起こる」



LTLとCTL

- 検証方法が全く異なる
 - 用いるツールによって扱える性質記述が異なる
 - 一般的にCTLの方が効率がよい
- 表現できる性質が少し異なる
 - ツールによっては補完する機能を設けている
(例：CTL+Fairness)
 - 振る舞い記述の方に明示的にアサーションを入れる等、だいたい対応する手段がある

現実的な時間で網羅的な検証ができるかの方が大きな問題