経営情報システム学特論１
６．性質の検証（２）

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

（再）性質

- 検証における性質（property）：プログラムやそのモデルに対し，起きうるすべての実行パスにて成り立つべき命題

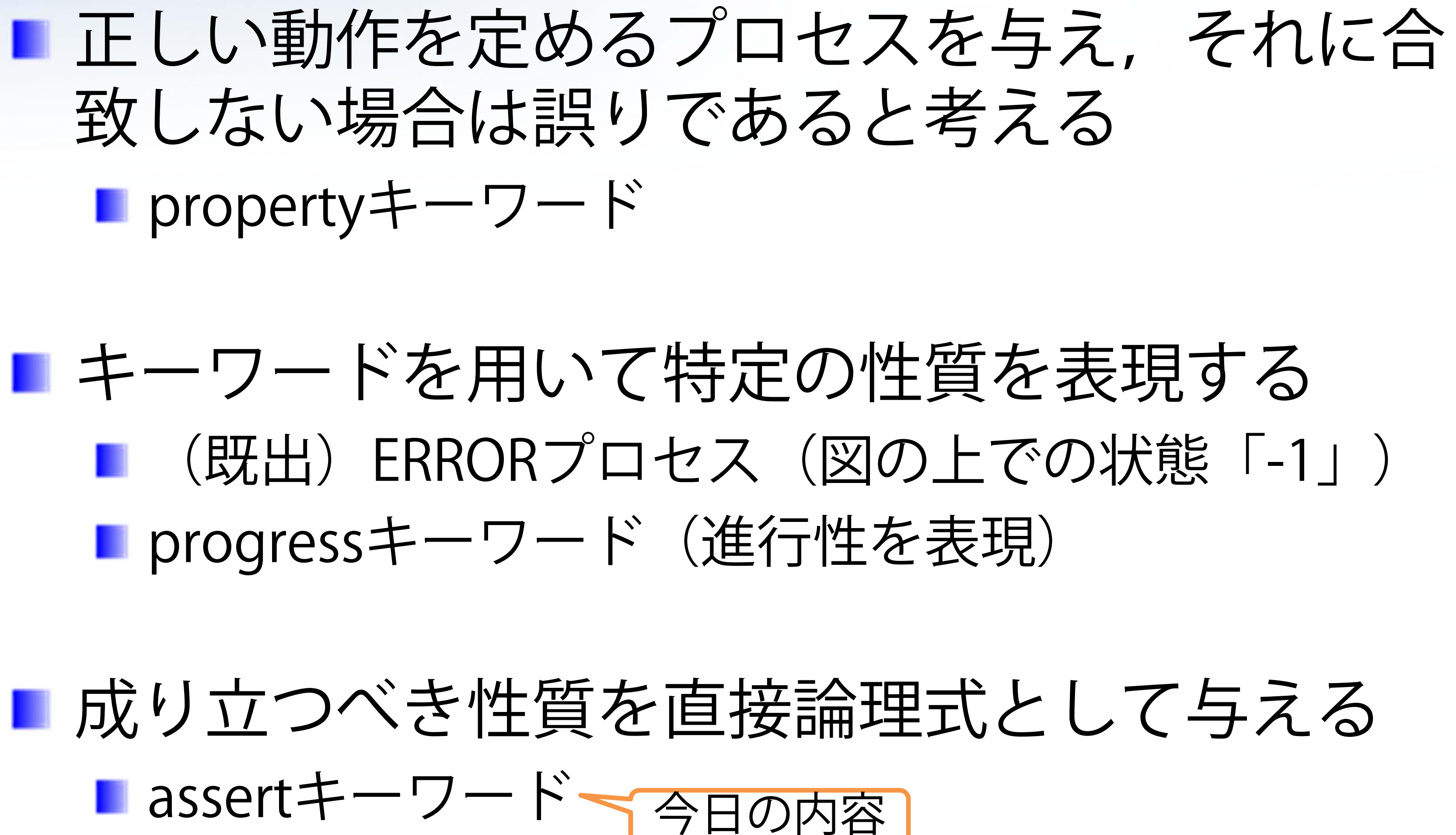
  すなわち検証において「正しさ」の基準となり確認する対象

- 逐次プログラムの場合，

  実行後の結果が期待された条件を満たすこと

  プログラムが停止すること

（再）LTSAの場合



目次

- LTSAにおける性質の記述
- 時相論理に関する一般論

Fluent

- *Fluent*：時間の経過・実行の進捗に応じて真偽が変わる条件
  - （今の状態で）ドアが開いている，部屋に人がいる，フォークを右手に持っている，・・・
  - LTSAの場合は，状態ではなく遷移に名前（ラベル）が付いているので，Fluentを成り立たせる遷移と成り立たせなくする遷移を与えて定義する
  - 例：「ドアが開く」という遷移が起きた後，「ドアが閉まる」という遷移が起きる前の間のみ，「ドアが開いている」というFluentが真であると定義する

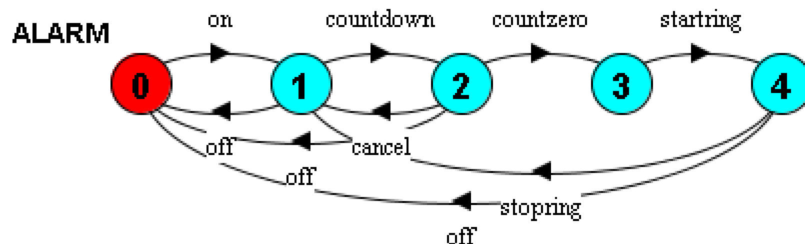
Fluentの定義

```
ALARM = OFF,
OFF = ( on -> ON ),
ON = ( off -> OFF | countdown -> SET ),
SET = ( off -> OFF | cancel -> ON
          | countzero -> startring -> RING ),
RING = ( off -> OFF | stopring -> ON ).

fluent ALARMON = <on, off> initially 0
fluent ALARMRINGING = <startring, {stopring, off}>
```

最初はfalse（デフォルト）

Fluentを成り立たせる開始アクション（の集合）

Fluentを成り立たせる終了アクション（の集合）



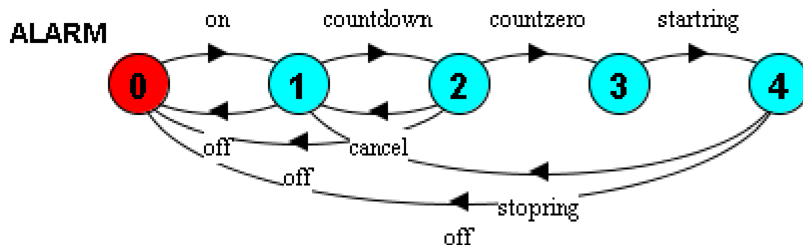
ALARMONは状態1～4にて，ALARMRINGINGは状態4にて成り立つ

Fluentの定義

- すべてのラベルに対して（何も書かなくても）Action Fluentが定義される
  - 名前はラベルと同じ
  - そのラベルを開始アクションとし，そのラベル以外（τ 含む）すべてを終了アクションとするようなFluent
  - 要はそのラベルが発火した直後の状態のみ成り立つ，そのラベルが発火し続けると成り立ち続ける

```
fluent ALARMRINGING = <startring, {stopring, off}>
fluent ALARMRINGING2 = startring
```

同じ



命題（1）

- ***命題***： 真偽が定まる条件式
  - Fluentは命題
  - Fluentを***命題論理***の演算子にてつないだもの

&&　　：論理積（∧）

||　　：論理和（∨）

!　　：否定（¬）

->　　：含意・ならば（⇒）

<->　　：同値（⇔）

命題（2）

- （続）
  - Fluentを*一階述語論理*の演算子にてつないだもの

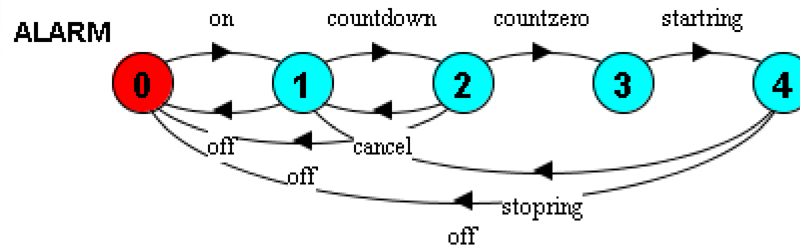
```
forall[i:1..N] FLU[i]
      ⇔ FLU[1] && FLU[2] && …

exists[i:1..N] FLU[i]
      ⇔ FLU[1] || FLU[2] || …
```

時相論理の演算子（1）

- [] FLU は，FLUが現在の状態および以降のすべての状態にて成り立つときのみ真になる
  - 「常に」「いつも」
  - □ と表記されることもある（ [] はこのascii文字での表記）
  - Gと表記されることもある（Globally）

時相論理の演算子（1）

- LTSA上で CHECK > LTL Property



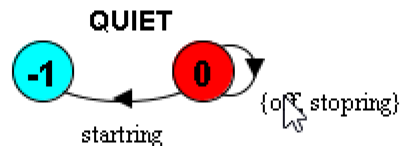

```
fluent ALARMON = <on, off>
fluent ALARMOFF = <off, on>
fluent ALARMRINGING = <startring, {stopring, off}>

assert ONOROFF = [](ALARMON || ALARMOFF)
assert QUIET = []!ALARMRINGING
```

成り立つ

状態4にいたるパスが反例

時相論理の演算子（１）

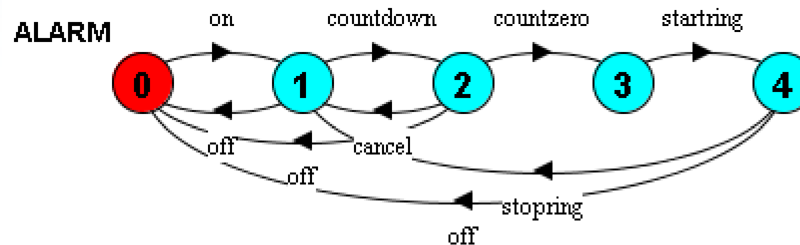
- LTSAにおける安全性検証の内部（１）
  - []FLU を検証する際には，FLUを満たさないときにERRORに遷移するようなオートマトンが構築され，合成されている



時相論理の演算子（２）

- <> FLU は，FLUが現在の状態または以降のいずれかの状態にて成り立つときのみ真になる
  - 「いつか」
  - ◇ と表記されることもある（ <> はこのascii文字での表記）
  - Fと表記されることもある（Finally）
  - LTSAではprogress同様にすべての遷移が均等に起きるとして（ある選択肢だけずっと選ばれないことはない）検証される

時相論理の演算子（2）




```
fluent ALARMON = <on, off>
fluent ALARMOFF = <off, on>
fluent ALARMRINGING = <startring, {stopring, off}>

assert EVENTUALLY_RING = <>ALARMRINGING
assert QUIET2 = !<>ALARMRINGING
```

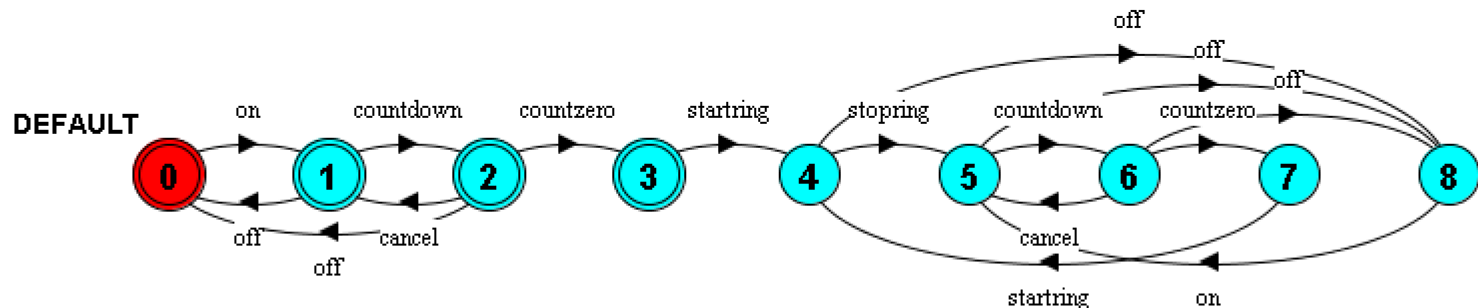
成り立つ

状態4にいたるパスが反例

時相論理の演算子（2）

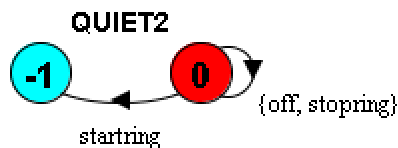
- LTSAにおける活性検証の内部（1）
  - <>FLU を検証する際には，FLUが決して起きないときのみ受理状態になるようなオートマトンが構築されている

  - 元のプロセスと合成したときに受理状態しか通らないようなサイクルができるかどうかを確認する（それがなければ，<>FLUが成り立つ）

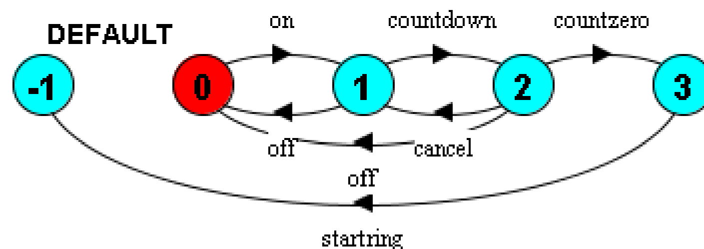


時相論理の演算子（２）

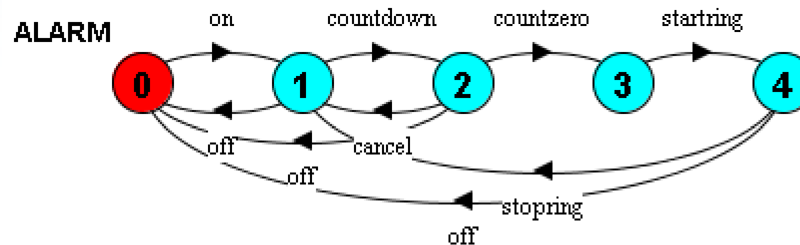
- LTSAにおける活性検証の内部（２）
  - !<>FLU を検証する際には，FLUが起きたときERRORとなるようなプロセスが構築されている



  - 元のプロセスと合成したときにERRORに到達する可能性があるかを確認する
（それがなければ !<>FLU が成り立つ）



時相論理の演算子（２）




```
||SYSTEM = ALARM <<{cancel}.

fluent ALARMON = <on, off>
fluent ALARMOFF = <off, on>
fluent ALARMRINGING = <startring, {stopring, off}>

assert EVENTUALLY_RING = <>ALARMRINGING
assert QUIET2 = !<>ALARMRINGING
```

cancelは他ラベルより最優先

成り立たない

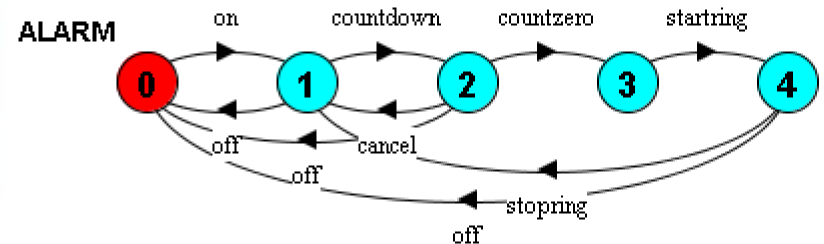
成り立つ

組み合わせ

- 典型的な組み合わせ
  - []<> FLU ： 無限回起きることができる（どの状態から見てもその先いつか起きる）
  - <>[] FLU ： ずっと成り立ったままにいつかなる
  - [] (FLU1 -> <>FLU2) ： FLU1が起きたときにはいつでも，その先いつかFLU2が起きる
- 同値な言い換え
  - ![]FLU ⇔ <>!FLU
  - !<>FLU ⇔ []!FLU

組み合わせ

- それぞれ検査してみよ



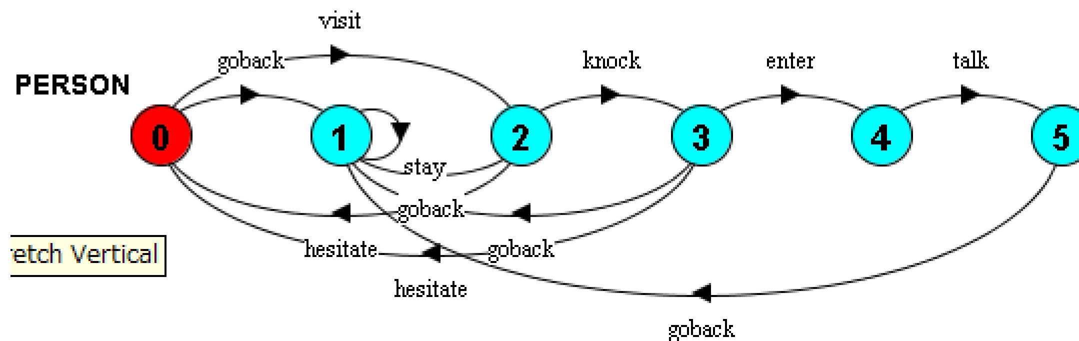

```
assert INIT_ON = ALARMON
assert INIT_OFF = ALARMOFF

assert ALWAYS_ON = [] ALARMON
assert ALWAYS_ON_IFNOTOFF =
  ( (!<>off) -> ([]ALARMON) )

assert PROGRESS_RING = []<> ALARMRINGING
assert PROGRESS_AFTERRING =
  [] ( countzero -> <> (off || stopring) )
```

時相論理の演算子（３）

- FLU1 U FLU2 は，FLU2が現在の状態にて成り立つ，あるいは，FLU2が以降のいずれかの状態にて成り立ちそれまではFLU1が成り立ち続ける
  - 「までは」・Strong Until
- FLU1 W FLU2 は，FLU2が現在の状態および以降のすべての状態にて成り立つ，あるいは，FLU1 U FLU2 が成り立つ
  - Weak Until
- X FLU は，FLUが次の状態にて成り立つ
  - 「次に」・Next
  - ○ と表記されることもある

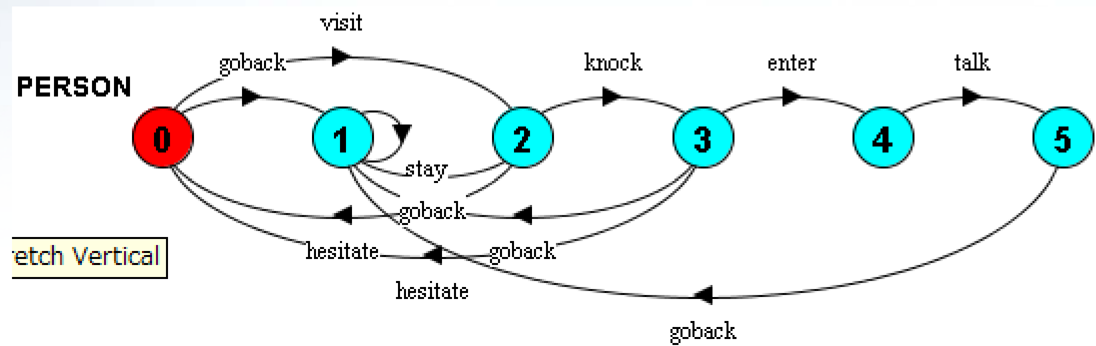
時相論理の演算子（3）

```
PERSON = ROOM,
ROOM = ( visit -> FRONT | goback -> HOME),
FRONT = ( knock -> KNOCKED | hesitate -> ROOM
          | goback -> HOME ),
KNOCKED = (enter -> INROOM | hesitate -> ROOM
           | goback -> HOME ),
INROOM = ( talk -> goback -> HOME ),
HOME = ( stay -> HOME ).
```



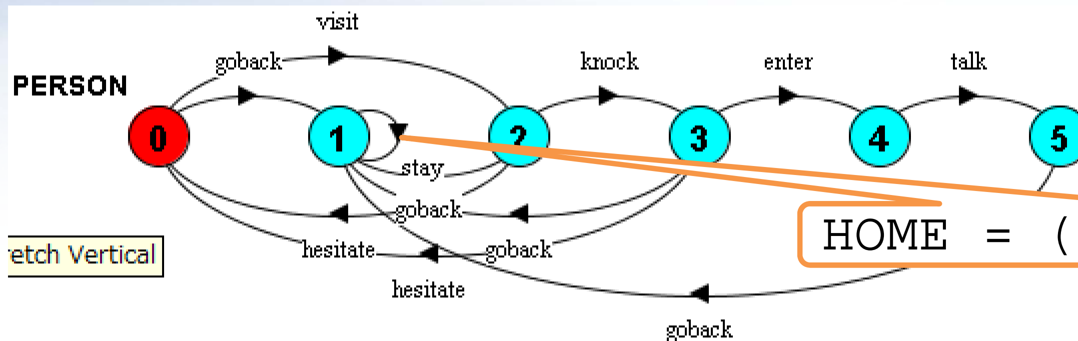
時相論理の演算子（3）

- それぞれ検査してみよ




```
assert POILTE_SAFE1 = ( !enter W knock )
assert POLITE_SAFE2 = ( !enter U knock )
assert POLITE_SAFE3 =
  ( (<>enter) -> (!enter U knock) )

assert POLITE_PROGRESS1 = [] ( knock -> <> enter )
assert POLITE_PROGRESS2 =
  ( (<>enter) -> ( [](knock -> <> enter) ) )
assert POLITE_PROGRESS3 =
  ( (<>enter) -> ( [](knock -> X enter) ) )
```

LTSAでの活性の扱い（復習）



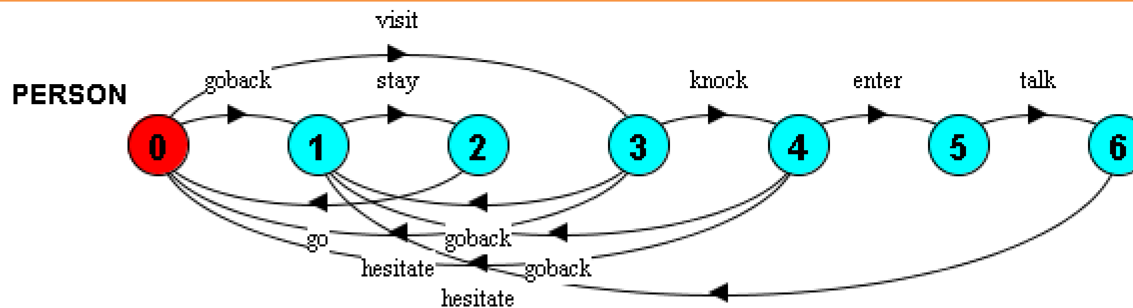

```
HOME = ( stay -> HOME ).
```

```
assert POLITE_PROGRESS1 = [] ( knock -> <> enter )
```

enterを含まないサイクルにて落ち着く可能性があるので成り立たない

```
HOME = ( stay -> go -> ROOM ).
```

として再度enterする可能性があるように変更すると,
enterを含む全体がサイクルとなるため成り立つようになる



LTSAでの活性の扱い（発展）

- 前スライドの更新版においても，Optionの“Fair Choice for LTL check”を無効にすると検査が通らなくなる
  - 再掲：「すべての非決定的な選択肢がどれも『公平に』無限回選ばれるという仮定の下で，無限にいつか起きる」かどうかを検査している
  - この仮定を取り除いたので，たまたま無限に特定の遷移が決して選ばれないような可能性も指摘される

目次

- LTSAにおける性質の記述
- <u>時相論理に関する一般論</u>

時相論理（時相命題論理）

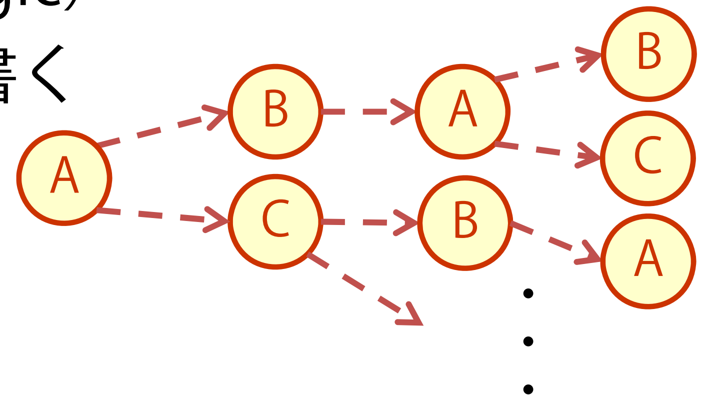
実行の経過に関する論理式を記述

「かつ」「または」「ならば」（命題論理）
＋「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）
  実行を「線」と見て性質を書く
  → LTSAはこちら
- CTL（Computational Tree Logic）
  実行を「木」と見て性質を書く

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



Linear Temporal Logic（LTL）

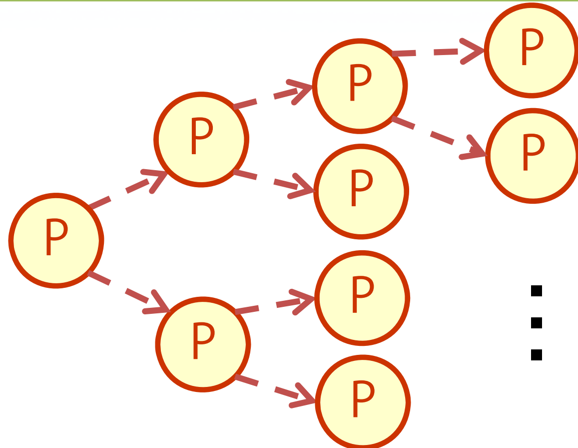
- すべての（無限長）実行パスに対して何か性質が成り立つかどうかを記述

X p（○ 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）

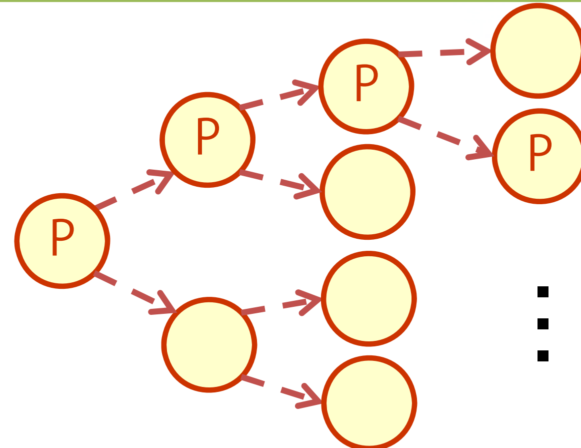
Computational Tree Logic（CTL）

- 実行の可能性を木とみて性質を記述

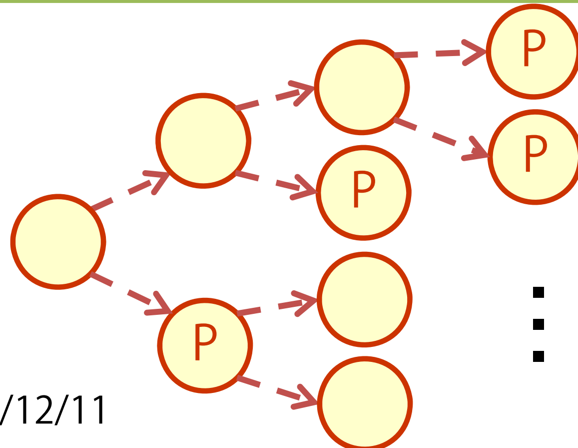
AG P　どのパスでも常に



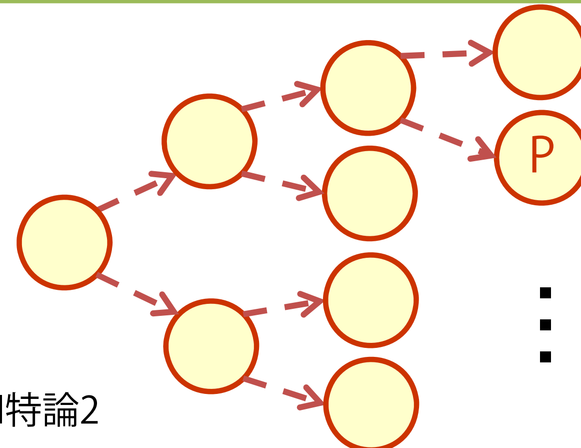
EG P　あるパスでは常に



AF P　どのパスでもいつか



EF P　あるパスではいつか



LTLとCTL

例としてGとFだけに話を絞って説明（UやXも同様）

- LTL： GとFを組み合わせ
  - 「あらゆる実行パスで」ということが前提で付き，「とある実行パスでは」ということが書けない
  - CTLでのAGEFのようなことは書けない（いつか到達する選択肢がある，という性質）
- CTL： AG，AF，EG，EFを組み合わせ
  - GFやFGが書けない
  - 単純な場合はAGAFやAFAGだが，より複雑な「あるパスでは無限回」などは書けない
- CTL*： 上記の任意の組み合わせ

LTLとCTL

- 現実問題
  - 各ツールはいずれか片方をサポートしている
  - サポートしていない方についても，特定キーワードを使ったり，プロセス側に埋め込んだりして，検証はできることが多い
  - LTSAは公平性の仮定（各ラベルが均等に起きると仮定）して活性（F・<>）を考える

⇒ 「最終サイクルの中でEF」相当の意味になっている

優先度を明示的に付ければ，「偏った実行のために決して起きない」可能性を指摘させることもできる

まとめ

- 論理式としての性質の記述
  - 「いつも」「いつか」「までは」といった語彙を扱う時相論理を用いる
  - 時相論理としては，Linear Temporal Logic（LTL），Computational Tree Logic（CTL）があり，利用できる時相演算子の組が異なる
  - 実用上は各ツールの検証機能をよく知ることが重要

- 次回： 検証について例題を通して復習する