形式手法

2022/11/28

海谷 治彦

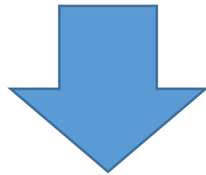
目次

- 形式手法で解決しようとする問題点と手段
- 形式手法の分類
- モデルチェック

再録

コーディングの入力は？

- アークテクチャが決まりました.
  - 配置図等
- モジュール設計もできました.
  - クラス図等



- 無論，これらをプログラム群にしないと動きません.
  - システムは複数の要素(プログラム群，ハード群，人等)から構成されて,
  - それらが，相互に連携しあって要求を満たします.
- 本日は, プログラミングに関するソフトウェア工学の話題を取り上げます.

ソフトウェアの仕様

- Specification
- 当該のソフトの振る舞いを規定する文書.
- 関数ならば，何を入れると何が出てくるかの記述.

- ソフト全体，個々のクラス，メソッドや関数等は仕様通りに作らなければならない.
- 設計段階で仕様は明確に記述されている（はず）.
- 仕様はソフト等が何をするかを記述しているもので，それによって何が満たされるか（要求）とは異なる.
  - 要求と仕様は別物．要求はソフト，機械，人等の連携によって満たされる人の願望.

実際の仕様

- 基本的に英語や日本語の文書で記述してあることがほとんど.
- クラス図等のモデルでは, クラス名やメソッド名から, 意図をくみ取れや, という空気もある.
  - 例 getBalance なら預金残高を返せっていう機能だろうなぁと, くみ取る等.
- ソフト(アプリ)全体は複数の機能群を外部に提供する存在である.
  - 各機能の仕様も文書で記述される. ユースケース記述等.
  - 文書でなくても, 機能を処理手順として手続き的に記述する.

仕様にかかわる問題点

1. 仕様通りにコード等が実装されているかわからない.
   - すなわち，検証(Verification)できてない.
   - 実装忘れ(incompleteness)も含むが，それはわりと自明.
2. 個々の関数，クラス等の仕様が，相互に矛盾してないかわからない.
   - 矛盾 inconsistency
   - そもそも要求段階で矛盾があるかもしれない.
3. 仕様通りに実装されたとして，要求が満たされるのかわからない.
   - すなわち，妥当性確認(Validation)ができてない.
   - 仕様の抜け(これもincompleteness)も含む.
   - 要求の抜けの発見は困難.

形式手法とその基盤(数学)

- 前頁の問題群を数学的なツールで解決しようとするソフトウェア工学の手法群が形式手法.

利用する主な数学ツール

✓論理学 ¬ ∧ ∨ ≡ ⇒ ⇔ ∀ ∃ □◇

- 命題論理, 述語論理, 時相論理, 様相論理

✓集合 ⊂ ⊃ ⊆ ⊇ ∪ ∩ ∈ ∋

- 主にデータや状態の規定につかう

✓代数 =

- 等式の集合と, それに基づく式や項の書き換え

✓グラフ

- 有向グラフ等. 集合と論理で記述可能

手続き的 VS 宣言的

- 多くの形式手法は，仕様を手続き的に記述しない.
- ある機能が満たす条件を宣言的に書く事が多い.
- 前述のように論理学が中心となるので，宣言的な記述が優勢というところもある.
- 手続き的に仕様を書くと，実装もその手順にひっぱられるという悪い副作用を回避するという視点もあるのかもしれない.

宣言的な仕様の基本

- 機能を手順で説明するのではなく，以下の三種類の条件で規定する様式が宣言的仕様.
- 手順を書かないので，一般には分かりにくい.

1. 事前条件 pre-condition
   - 該当する機能を実施する前に成り立たせる必要のある条件.
2. 事後条件 post-condition
   - 機能の実施後に，何が成り立っているかの条件を書く.
3. 不変命題 invariant
   - 機能の処理中に成り立ってないといけない条件.
   - クラス（オブジェクト）やシステム全体等，ある一定時間に存在している部分の仕様には必要.

例

- 機能概要
  - 正の整数の数列を大きい順に並べ替える.
- 事前条件
  1. 数列に含まれる全ての数が，0より大きい整数.
- 事後条件
  1. 並べ替える前の数列の要素は，全て，並べ替えた後の数列に含まれる.
  2. 並べ替える前の数列に，同じ数が複数あった場合，並べ替えた後の数列にも当該数が同じ数だけ含まれる.
  3. 並べ変えた数列の要素それぞれについて，最後の要素を除き，当該要素は，次の要素より大きいか等しい.
- 上記のように，別に(論理)式等を使わなくても宣言的な仕様は記述できる.
  - ただし，かなり慎重に文章を書く必要があるので，式で書いたほうが往々にして楽.

数学等によって問題点は解決されるか？

- 仕様にまつわる問題点を前の方で挙げた.
- 数学的なツール（論理式や集合等）を使い，その問題点が解決されるのか？

1. 仕様と実装の問題
   - 仕様の数学的性質を実装が引き継いでいると示せれば，これは解決できる.
2. 矛盾の問題
   - これは数学系ツールの得意分野.
3. 仕様と要求の問題
   - 要求が数学的な言葉で記述できれば，仕様を公理系とみなし，証明が可能.
   - ただし，ソフト以外の所も公理の一部として，数学の言葉で定式化する必要がある.

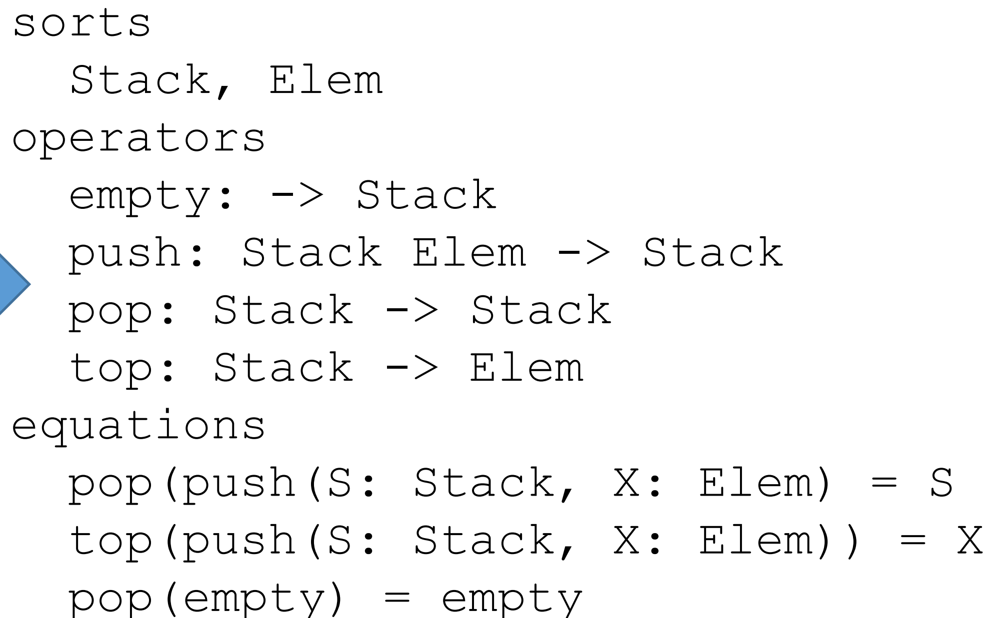
形式手法の分類

- モデル規範 Model-oriented
  - 仕様の記述を既知の数学的な体系に対応付ける手法.
  - 完備な数学的体系（命題論理等）は,
    - 字面の操作で導けることは意味的に正しく,
    - 意味的に正しいことは，字面の操作で全て導ける.
  - そんな体系に仕様の記述を対応付けることで，記述の正しさを担保する.
  - 例 VDM, Z, B, Event-B
  - モデル検査(spinやSMV)の手法もこっちのグループ
- 性質規範 Property-oriented
  - 記述の要素が満たす関係式を定義し，要素の意味を他の記述との関係性で定義する.
  - 代数（等式の集合）を用いた記述等が代表例.
  - その記述が既知の数学体系と対応付いているわけではないのが上記との違い.

代数仕様の例

- Stack, Elem という二つのデータを定義型を定義
- empty, push, pop, top の四つの操作(関数)を定義
- 操作を適用した場合に成り立つ等式を定義

- コレを仕様とするなら，コレを見て，プログラマがプログラムを作る・・・
- アリエルのか？
- ありえないのか？
- 文章のほうがマシか否か？



```
sorts
  Stack, Elem
operators
  empty: -> Stack
  push: Stack Elem -> Stack
  pop: Stack -> Stack
  top: Stack -> Elem
equations
  pop(push(S: Stack, X: Elem) = S
  top(push(S: Stack, X: Elem)) = X
  pop(empty) = empty
```

対象を記述する観点の分類

- 前述のモデルと性質は実際は混ぜて使われる.
- 実用的には，ソフトをどのような視点から仕様として記述するかの流儀の分類がある.

a. 状態に基づく (VDM, Z, B等)
   - ほぼモデル規範に対応

b. 履歴に基づく (モデルチェック系)
   - 実行経路の集まりから振る舞いを規定し，そこでの性質を扱う.

c. 状態遷移
   - ほぼオートマトンの利用のこと

d. 関数として (プロセス代数等)
   - 前頁の代数仕様みたいなもの.

e. 操作として (色々あった)
   - 仕様をプログラム的にみなし，実行してみること.

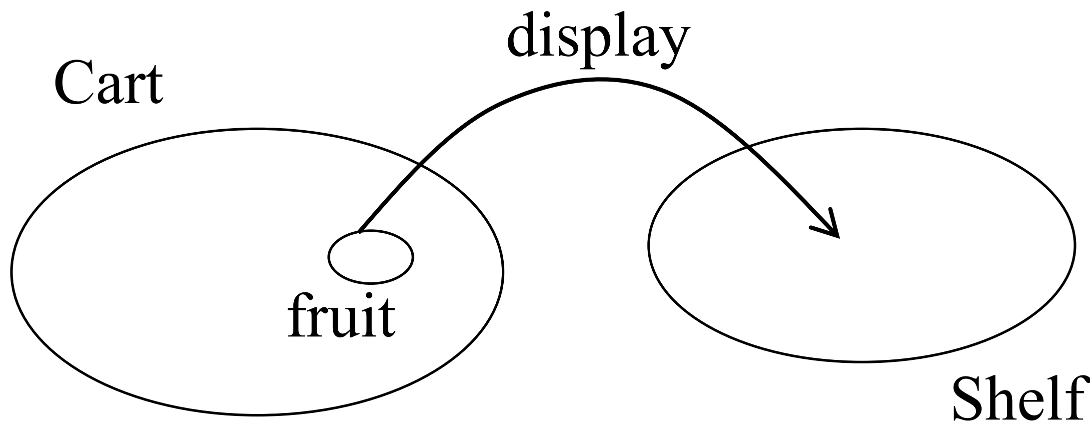
モデル規範の例: Event-B

- 典型的なモデル規範の記法
- イベントの概念によって並行性等を扱いやすくなった. (ただのBよりも)
- ツールもある RODIN

- なんだかかっこいい（？）数学の記号を一杯使う.
  - コレは Z, VDM 等も同じ.
  - ギリシャ文字とか謎の数学関係式とか.
    - 自己満なのか・・・

例題による記述を利用紹介

- スーパーマーケット売り場で，果物(fruit)をカート(cart)から取り出して，陳列棚(shelf)に並べる(display)・・・・
- というシステム（というか業務）の仕様.
- 上記の仕様を模式図に表したのが以下.



コンテクストとマシンM0

- 果物全体を示す有限集合 Fruit を考える.
- Context に集合の宣言 sets と 公理 axioms を導入する.

```
CONTEXT C0
SETS Fruits
AXIOMS
  axm1: finite(Fruits)
END
```

- Cart と Shelf を 果物の集まり参照する変数とする.
- displayは集合間で果物を移動するイベントとみなせるが後述.
- Cart, Shelf 間の不変命題を以下のように書く.

```
MACHINE M0
SEES C0
VARIABLES
  Cart
  Shelf
INVARIANTS
  inv1: Cart ⊆ Fruits
  inv2: Shelf ⊆ Fruits
  inv3 Cart ∩ Shelf = Φ
EVENTS
  次頁以降
END
```

陳列する機能の仕様

- 陳列する display を Even-B では，<u>イベント</u>と呼ぶ.
- システムの状態を変化させる<u>機能</u>のようなもの.
- 左の例では，where にある条件において，then にある状態変化が起こると書いてある.
- ≡ は，＝ の上に△があるみたいな記号だが，フォントがないので，≡ で代用.

```
M0-Display ≡
any x
where
  grd1: x ∈ Cart
then
  act1: Cart := Cart ＼ |x|
  act2: Shelf := Shelf ∪ |x|
end
```

マシンM0の初期設定

- マシンM0の最初の状態の設定.
- act1 は，べき集合 P を構成する演算子を使っているが，実質，Cart は Fruits の部分集合といってるのと同じ.
- 以降，こんなこと書いて，何がうれしいかを示す.

```
MD-Initialization ≡
begin
  act1: Cart :∈ P(Fruits)
  act2: Shelf := Φ
end
```

形式検証

- 前頁までで書いたシステムの仕様内に矛盾が無いことが証明できる.
- 一応，前の方で述べた問題点2を解決できる.
- 以下，主な証明すべき条件.
- INV証明条件
  - 定義したイベントがマシンの不変命題を壊すことが無いことを証明できる.
- FIS証明条件
  - イベント実施によって遷移する状態が，存在しうることを証明できる.

マシン M0 の場合

- INV証明条件
  - Cartの要素 x は当然 Fruits の要素である.
  - それを Cart や Shelf から除外したり追加したりするのだから，inv1, inv2 は満たされる.
  - inv3 に関しても，同一要素 x を Cart から除外して，Shelf に追加してるので，積集合は空のまま.
  - 初期値は Shelf が空なので問題ない.
- FIS証明条件
  - イベント Display の act1 act2 によって更新される Cart Shelf の集合は，存在しうるので，FIS条件はOK
- 上記の文章で書いた内容を論理式の証明で，ごりごり書くらしい．だから，Formal に証明されたことになるらしい.

垂直リファインメント Refinement

- Vertical refinement, or stepwise refinement
- 仕様を段階的に少しずつ，プログラムに変換してゆく手法.
- プログラムに近づく毎に詳細化(refine)されると考える.
- 一回の詳細化の前後で，前の記述の持つ性質が，後の記述に引き継がれていることを証明する.
- これによって，仕様の性質をちゃんと反映したプログラムを導出できると考えられている.
- 最初の仕様が正しいかどうかは対象外.

水平リファインメント

- Horizontal Refinement
- 機能を含めた仕様の構成要素を追加する方法
- 追加した仕様が，先にある仕様の持つ性質を壊さないことを確認できる．

水平リファインメントの例

- 以下の仕様がM0に追加されたとする.
- 「果物の陳列する際に，開店前か売り場が混雑していないときに陳列を行う.」

コンテクストの拡張

- スーパーの営業状態を，Open と Close に分ける.
- Openの間の特殊な状態として，Busy を導入.
- partition(x,y,z)はxがyとzに分割されてることを表す略記.
- $(y \cap z=\Phi) \wedge (y \cup z=x)$ と書くとタルいので.

```
CONTEXT C1
SETS Status
CONSTANTS
  Open
  Close
  Busy
AXIOMS
  axm1: partition(Status, Open, Close)
  axm2: Busy ⊂ Open
  axm3: Open ≠ Φ
  axm4: Close ≠ Φ
END
```

MR0, M0の拡張版

- M0 に status 状態を追加した拡張マシン MR0

MACHINE MR0
REFINES M0
SEES C0 C1
VARIABLES
  Cart
  Shelf
  status
INVARIANTS
  inv4: status $\in$ Status
EVENTS
  次頁以降
END

参考

MACHINE M0
SEES C0
VARIABLES
  Cart
  Shelf
INVARIANTS
  inv1: Cart $\subseteq$ Fruits
  inv2: Shelf $\subseteq$ Fruits
  inv3 Cart $\cap$ Shelf = Φ
EVENTS
  次頁以降
END

イベントの拡張

- Close もしくは Busyじゃない時，このイベントは実行できるよ，という記述にする.
- 抽象側のM0より具体側のMR0のほうが条件がきびしくなっているので，このイベント発生の頻度は狭まっている.
- 段階的詳細化の観点からは，これで矛盾なく詳細化されたといえるらしい.

MR0-Display ≡
extends M0-Display
any x
where
  grd1: $x \in Cart$
  grd2: $status \in Open \setminus Busy \lor status \in Close$
then
  act1: $Cart := Cart \setminus |x|$
  act2: $Shelf := Shelf \cup |x|$
end

参考

M0-Display ≡
any x
where
  grd1: $x \in Cart$
then
  act1: $Cart := Cart \setminus |x|$
  act2: $Shelf := Shelf \cup |x|$
end

垂直リファインメントの例

- 前述の水平に以下の仕様が追加されたとする.
- 「果物は国産か輸入物のどちらかで，国産は左の棚，輸入は右の棚に並べる」

- 機能追加っぽく見えるが，まぁ，棚や果物に関して，より具体的な分類を導入したということで，垂直リファインメントの例となる.
- 無論，コードには，ほど遠い.

コンテクストの拡張

- 輸入品の集合，Oversea と，果物から生産地を得る関数みたいなもの label を導入する.

CONTEXT CR0
EXTENDS C0
SETS Origin
CONSTANTS
  label
  Oversea
AXIOMS
  axm1: Oversea $\subset$ Origin
  axm2: label $\in$ Fruits $\rightarrow$ Origin
END

参考

CONTEXT C0
SETS Fruits
AXIOMS
  axm1: finite(Fruits)
END

マシンの拡張

- 左右の棚の導入
- inv0 によって，M0とMR1の対応付けがされている.

```
MACHINE MR1
REFINES MR0
SEES CR0, C0
VARIABLES
  Cart
  LeftShelf
  RightShelf
INVARIANTS
  inv0: partition(Shelf, LeftShelf, RightShelf)
EVENTS
  次頁以降
END
```

参考

```
MACHINE MR0
SEES C0 C1
VARIABLES
  Cart
  Shelf
  status
INVARIANTS
  inv1: Cart ⊆ Fruits
  inv2: Shelf ⊆ Fruits
  inv3 Cart ∩ Shelf = Φ
  inv4: status ∈ Status
EVENTS
  次頁以降
END
```

イベントの拡張

- 国産は左の例のみ，右の例もほぼ同様.

```
MR1-Display-Left ≡
extends MR0-Display
any x
where
  grd1: x ∈ Cart
  grd2: status ∈ Open∖Busy ∨ status ∈Close
  grd3: label(x) !∈ Oversea
then
  act1: Cart := Cart ∖ |x|
  act2: LeftShelf := LeftShelf ∪ |x|
end
```

```
参考
MR0-Display ≡
extends M0-Display
any x
where
  grd1: x ∈ Cart
  grd2: status ∈ Open∖Busy ∨ status ∈Close
then
  act1: Cart := Cart ∖ |x|
  act2: Shelf := Shelf ∪ |x|
end
```

形式手法の適用例，成功例

- 1973年 IBMがPL/1のコンパイラ開発にVDL/VDMを用いた．これが最初の適用例かも．
- 2000年前後 Bメソッド: パリの地下鉄，バルセロナの地下鉄，パリ国際空港(CDG)内のターミナル移動列車 CDG-VALのソフト開発．
- フェリカのファームや周辺ツールの開発

モデル検査

論理式を使う

- モデル検査の大前提として，システムの仕様は論理式の集まり（定理）として記述する.
- その論理式群を前提として，成り立ってほしい性質（これも論理式）が成り立つか否かを自動チェックする.
  - この性質が，ある意味，要求項目に相当する.
- ある前提（定理）下で，ある論理式（性質）が真か否かをチェックする方法の流儀の違いを理解する必要する.

モデルと証明

モデル (恒真)

- 論理学のモデル，UML等のモデルとは違う

- ある論理式が複数の変数から構成される際，その式全体が真となるような，変数それぞれへの真偽値の割り当てをモデルと呼ぶ.
- $\models$ という記号を使う. |= で代用するかも.

証明 (証明可能)

- ある論理式を，前提となる論理式群（定理）から推論規則を用いて字面を変換することで真偽を判定すること.
- $\vdash$ という記号を使う. |- で代用するかも.

モデルの例とvalid

- 論理式 $(p \to q) \land \neg(\neg p \land q)$ に注目.
- p, q ともに真(Tとする，偽はFとする)の場合，
  - $(T \to T) \land \neg(\neg T \land T) = (\neg T \lor T) \land \neg(F \land T) = T \land \neg F = T$
- よって，式全体はTとなり，p,qともに真という割り当ては，この式のモデルとなる.
- $p \land q \models (p \to q) \land \neg(\neg p \land q)$ とかける.
- $X \models Y$ の場合，$\models X \to Y$ としてよく，この場合，$X \to Y$ は，含まれる論理変数にT,Fをどのように割り当ててもTとなる.
- 上記のようにどんな割り当てでも真となる式をvalid と呼ぶ.

証明の例

- 以下の式1, 式2, 式3, 式4考える.
  - $p \to (q \to r)$, p, $\neg r$, $\neg q$
- 式1, 式2, 式3 を前提として，式4は推論規則から導ける.
  - 式1と式2から $q \to r$ は真で，これは $\neg q \vee r$. 式3で $\neg r$ なので， $\neg q$ のみ残り，式4となる.
- このことを以下のようにかく.
  - 式1, 式2, 式3 |- 式4
- 上記を以下に変換してよい.
  - |- 式1 →(式2 → （式3 →式4)

分かりやすい対比

式 $(\neg A \lor B) \to (A \to B)$ の真偽をチェック.

- モデルを使う，以下のように4通りのインプットに対して，式の真偽を計算していくだけ.

入力（A, B）　結果

A	B	¬A	¬A ∨ B	A → B	上記の式
T	T	F	T	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	F	T	T	T	T

全部真だから恒真 (valid)

- 証明，右のルールで字面を変換してゆくと，式全体が真になる.
- ルール1で $(A \to B) \to (A \to B)$
- $(A \to B)$ を X とすると $X \to X$
- ルール1で $\neg X \lor X$
- ルール2で 全体が真

ルール
1. $A \to B = \neg A \lor B$
2. $\neg A \lor A =$ 真

本来は命題論理の公理を利用すべきだが，上記の式の書き換えがコンパクトに行える正しい書き換えルールをここでは用いる.

証明せずにモデルをみつける

- 証明でもモデルでも式の真偽の判定を同様に行える．
  - ただし，健全で完全な論理体系の場合．
- ただし，モデルの場合，多数の変数があると，それらにT/Fを割り当てた全ての組み合わせをチェックするのが計算量的に困難だった．
  - intractable（手におえない）と呼ばれる
- 命題論理ならなんとかなるが，述語論理を用いると，全て(∀)の変数の割り当てをチェックするのは，そもそも不可能．
  - 可能な体系を decidable（決定可能）と呼ぶ．
  - 不可能な体系は undecidable と呼ぶ．

モデルチェックとは？

- ある論理式の（論理学の）モデルを計算機で機械的にみつける手法やツール.
- マシンが早くなったので，実用的に使えるようになった.
- decidable な体系なら網羅ができる.
- undecidableなら適当にタイムアウトさせる.

- 命題論理では表現能力が心もとないので，実際のモデルチェック手法では，時相論理が用いられる.

証明ベースの手法/ツールは？

- 無論，ある.
- しかし，最近はマシンパワーに頼ったモデルチェックのほうが流行ってる.
- 証明はあくまで，字面の変換によって式の真偽を判定するが，モデルはダイレクトに判定している.
  - ダイレクト 構成要素の真偽値を網羅的に当てはめる.
- 証明の場合，もとになる公理系が完全(|=なら|-)かつ健全(|-なら|=)でないと結果がきびしい.
  - 完全 全ての真である式を字面の変換(証明)で導ける.
  - 健全 真偽を導いた結果(証明結果)に誤りは無い.
- 命題論理は大丈夫だが，複雑な論理体系において，完全かつ健全な公理系は概ね無い.

LTLとCTL 代表的な時相論理

- LTL 線形時間論理 Linier Temporal Logic
  - 時間を直線の集合として表現
- CTL 分岐時間論理 Computational Temporal Logic
  - 時間を木で表現し，時間分岐を表現できる.
  - ある性質を持つパスが存在するということを記述しやすい.
- 双方を統合したCTL*という体系もあるが，対応するツールが無い.

何故，時相論理がソフト開発に？

- ソフトは基本的に状態機械とみなせる.
- 時間経過上で成り立ってほしい性質として要求項目を表現しやすい. (以下，例)
  - 走行中，電車のドアは開かない.
  - ボタンを押せばエレベーターはいつか来る.
  - 注文すればウーバーイーツは来る.
- 6ページで述べた仕様の問題点3「設計通りに作って要求が満たされるかのチェック」ができるのが売りといえば売りである.
- 仕様を時相論理で記述でき，確認したい要求項目を論理式で書けるなら，強力なツールとなる.

ツール NuSMVを インストールしてみる

興味がある方は，授業ページからリンクのある公式ページからダウンロードしてください．

LTL概要

- 命題論理の拡張版
- システムの仕様は状態遷移として表現される.
- 状態を規定する変数があり，その変数が等しい大きい等の条件として命題を記述できる.
- 同じ命題でも，状態毎に真偽値は異なる場合がある.（時間変化によって真偽が変わる）
- 真偽をチェックしたい性質の式は，全ての遷移のパスでも成り立ってないといけない.
  - 特定のパスで成り立つ等はチェックできない.
- ある状態である命題が真等は示せない.
  - ずっと成り立つ，いつか成り立つ，次に成り立つ等の性質しかかけない.

LTLの論理式

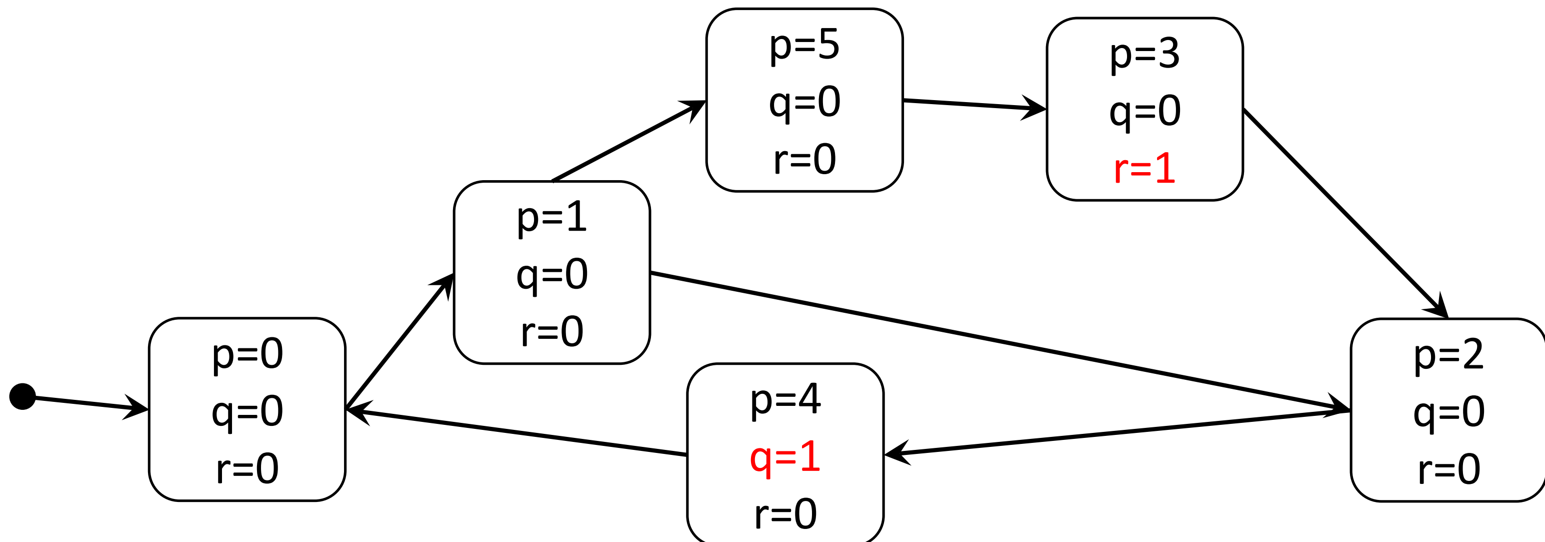
- G p 現在以降，ずっとpが成り立つ.
- F p 現在以降，いつかpが成り立つ状態になる.
- p U q qが成り立つまで，ずっとpが成り立つ.
- X p 次の状態でpが成り立つ.

- 論理演算子 ! & | xor -> <->
- 変数比較 = != < > <= =>
- 演算 + - mod

- C言語と異なる部分も多い.

簡単な例 ltl1.smv

- 変数, p, q, r の値がかわる以下のような状態遷移を考える.
- G F (q=1) どの状態からいってもq=1になる状態にたどりつけるは真.
- G F (r=1) r=1になるパスは通らない場合もあるので, これは成り立たない.



ツール(NuSMV)で自動チェックできる

```
*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification  G ( F q = 1)  is true
-- specification  G ( F r = 1)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 1.1 <-
    p = 0
    q = 0
    r = 0
  -> State: 1.2 <-
    p = 1
  -> State: 1.3 <-
    p = 2
  -> State: 1.4 <-
    p = 4
    q = 1
  -> State: 1.5 <-
    p = 0
    q = 0
-- specification  G (p = 5 ->  F r = 1)  is true

C:¥LEC¥SE¥NuSMV¥work>
```

仕様と検査式

```
MODULE main

VAR
    p: {0, 1, 2, 3, 4, 5};
    q: {0, 1};
    r: {0, 1};

ASSIGN
    init(p) :=0;
    next(p) := case
        p = 0 : 1;
        p = 1 : {2, 5};
        p = 3 : 2;
        p = 2 : 4;
        p = 4 : 0;
        p = 5 : 3;
        esac;
```

非決定的な遷移

```
-- 続き
    init(q) :=0;
    next(q) := case
        p=2 : 1;
        TRUE : 0;
        esac;

    init(r) :=0;
    next(r) := case
        p=5 : 1;
        TRUE : 0;
        esac;

LTLSPEC G F (q=1)
LTLSPEC G F (r=1)
LTLSPEC G(p=5 -> F (r=1))
```

状態遷移のシステムの仕様

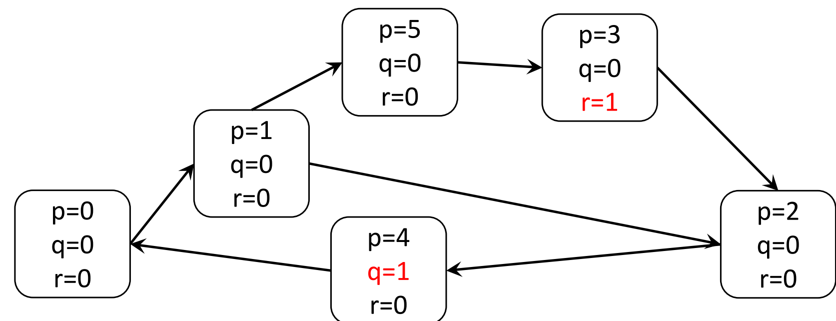
- 前頁の例をみればわかる通り，わりと単純なswitch文でかける．
  - 直前の状態値によって，次の状態を決定．
- 説明のため数ページ前の状態図を素直に書いてあるが，この形式にあってれは，明確な状態図のイメージがなくても仕様はかける．

Counterexample 反例

- 検査式が偽になるということは，その式が真とならない事例が少なくとも存在することを意味する.
- Counterexampleはそれを提示してくれる.

```
-- specification  G ( F r = 1)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 1.1 <-
    p = 0
    q = 0
    r = 0
  -> State: 1.2 <-
    p = 1
  -> State: 1.3 <-
    p = 2
  -> State: 1.4 <-
    p = 4
    q = 1
  -> State: 1.5 <-
    p = 0
    q = 0
-- specification  G (p = 5 ->  F r = 1)  is true
```

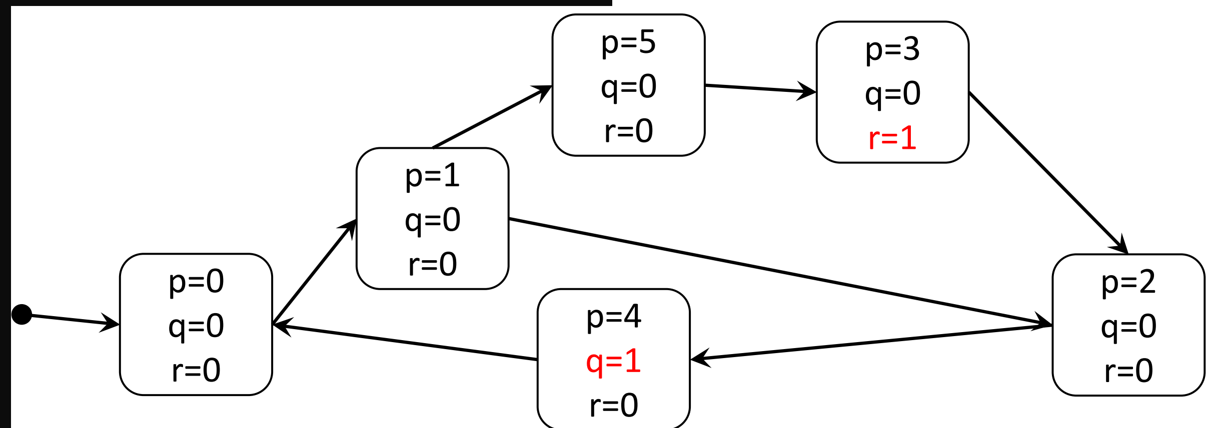
- G (F r=1) は，全ての時点で，将来 r=1 となる状態にたどり着けるということ.
- しかし，左のように，状態 p=0, 1, 2, 4, 0 で元に戻るパスがあり，このパスでは r=1とならない.
- よって，G(F r=1)は成り立たない.



Untilの例

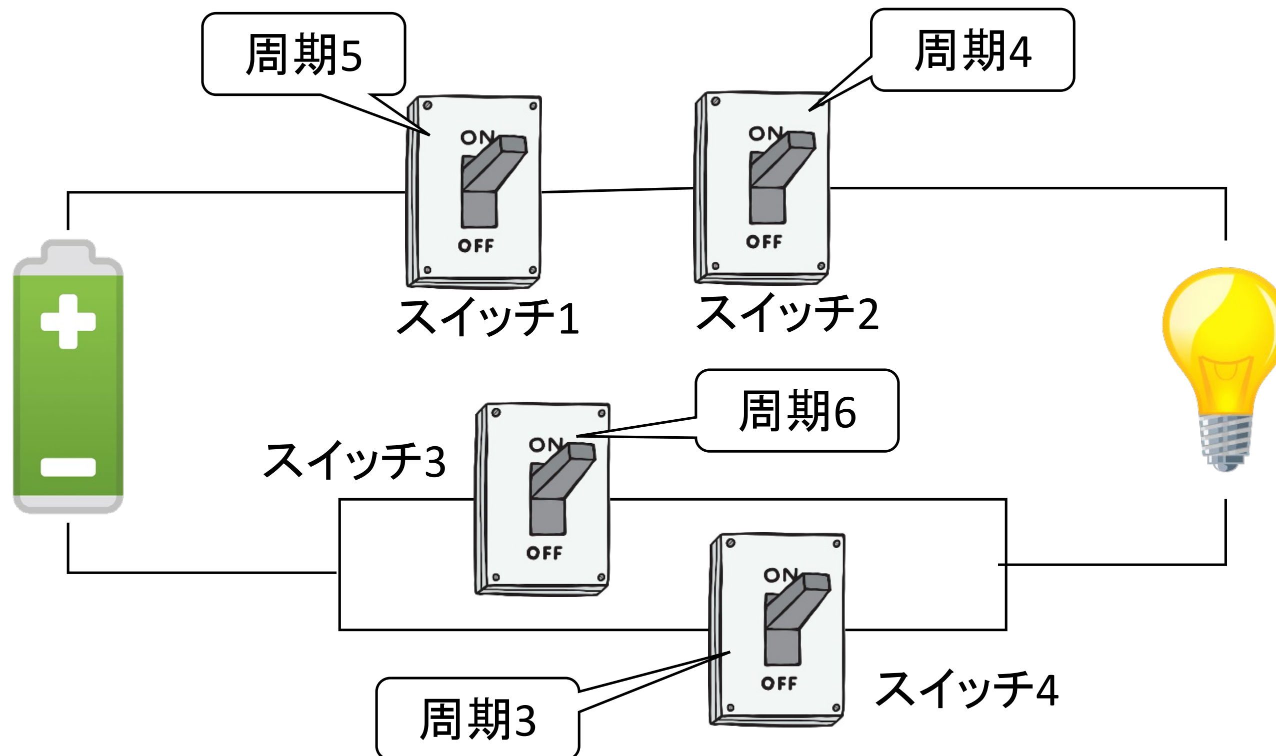
```
-- specification (p < 3 U q = 1)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 2.1 <-
    p = 0
    q = 0
    r = 0
  -> State: 2.2 <-
    p = 1
  -> State: 2.3 <-
    p = 5
  -> State: 2.4 <-
    p = 3
    r = 1
  -> State: 2.5 <-
    p = 2
    r = 0
  -> State: 2.6 <-
    p = 4
    q = 1
  -> State: 2.7 <-
    p = 0
    q = 0
-- specification  F (p < 3 U q = 1)  is true
```

F(p<3 U q=1)は，初期状態から出発する
q=1となるパスがある意味ではなく，
p<3 U q=1 が成り立つ状態が将来に存在するという意味.
具体的には，p=2の状態，だから真.



電球が付くかの検査

- 各スイッチは周期的に 1 から N まで数値を変える.
- 最大値になったら次のタイミングで通電する.
- 以下の配線で電球が付くことはありえるか？



状態仕様と検査式

```
---- lamp.smv
MODULE main

VAR
    Tm1: 1..5;
    Tm2: 1..4;
    Tm3: 1..6;
    Tm4: 1..3;

    Sw1: {On, Off};
    Sw2: {On, Off};
    Sw3: {On, Off};
    Sw4: {On, Off};

ASSIGN

    init(Tm1):=2;
    init(Tm2):=3;
    init(Tm3):=1;
    init(Tm4):=2;

    init(Sw1):=Off;
    init(Sw2):=Off;
    init(Sw3):=Off;
    init(Sw4):=Off;
```

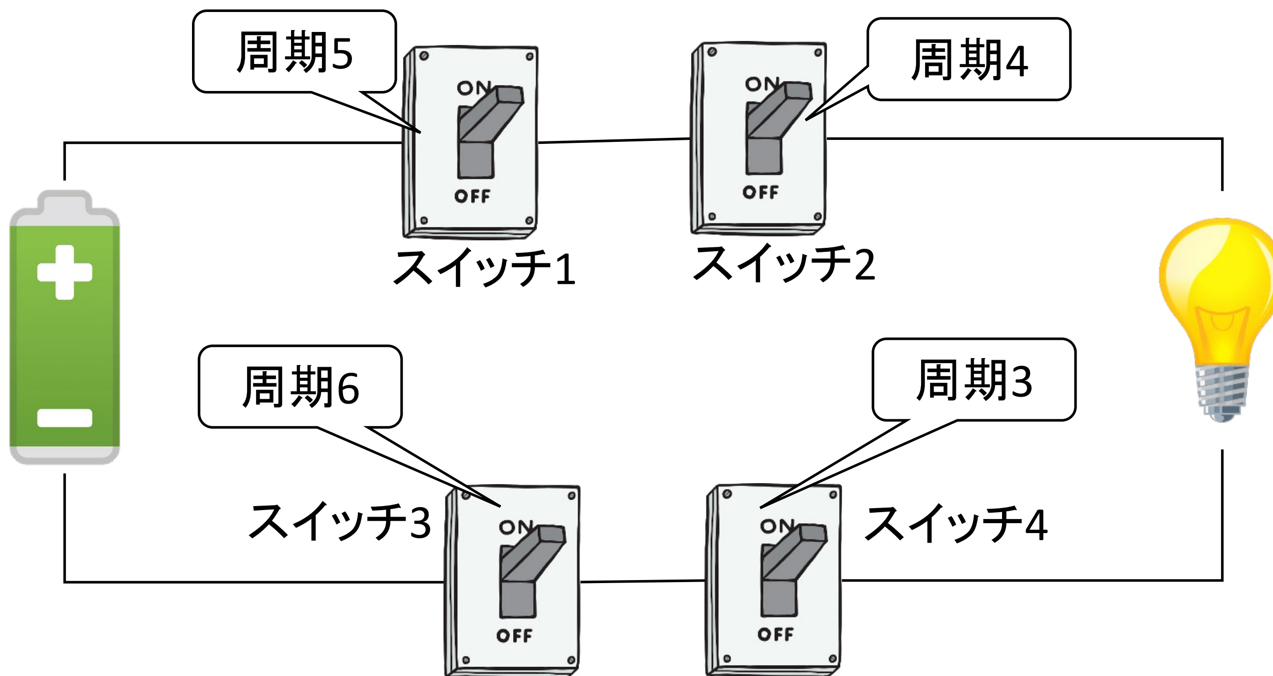
```
next(Tm1) := case
        Tm1 < 5: Tm1 + 1;
        TRUE: 1;
        esac;
    next(Tm2) := case
        Tm2 < 4: Tm2 + 1;
        TRUE: 1;
        esac;
    next(Tm3) := case
        Tm3 < 6: Tm3 + 1;
        TRUE: 1;
        esac;
    next(Tm4) := case
        Tm4 < 3: Tm4 + 1;
        TRUE: 1;
        esac;
```

```
    next(Sw1) := case
        Tm1 = 5: On;
        TRUE: Off;
        esac;
    next(Sw2) := case
        Tm2 = 4: On;
        TRUE: Off;
        esac;
    next(Sw3) := case
        Tm3 = 6: On;
        TRUE: Off;
        esac;
    next(Sw4) := case
        Tm4 = 3: On;
        TRUE: Off;
        esac;

-- Fig 3.1 on page 38, book1
LTLSPEC F(Sw1=On & Sw2=On &(Sw3=On | Sw4=On))
```

```
-- specification  F ((Sw1 = On & Sw2 = On) & (Sw3 = On | Sw4 = On))  is true

C:¥LEC¥SE¥NuSMV¥work>
```

別のつなぎ方にしてみる

- 各スイッチは周期的に 1 から N まで数値を変える.
- 最大値になったら次のタイミングで通電する.
- 以下の配線で電球が付くことはありえるか？




```
-- specification  F (((Sw1 = On & Sw2 = On) & Sw3 = On) & Sw4 = On)  is false
-- as demonstrated by the following execution sequence
```

CTL概要

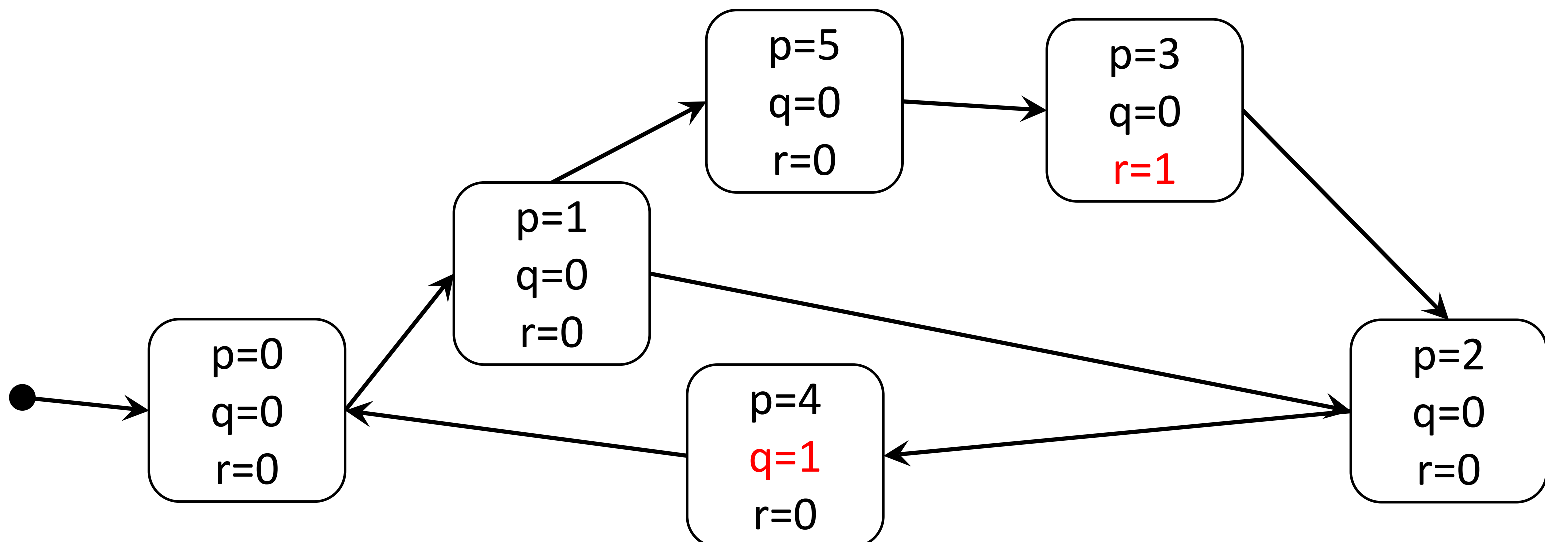
- やはり命題論理の拡張版
- システムの仕様は状態遷移として表現される.
- 状態を規定する変数があり，その変数が等しい大きい等の条件として命題を記述できる.
- 同じ命題でも，状態毎に真偽値は異なる場合がある.（時間変化によって真偽が変わる）
- とある状態遷移パスが存在することを素直に検査できる.
- 状態遷移上の条件を表す論理オペレーター(F,G,X,U等)の組み合わせに制限がある.

CTL論理式

- LTLのF,G,X,Uの前に以下のパス量化記号をつける. (つけなければいけない)
- A 全てのパスで成り立つ
- E 成り立つパスがある.
- すなわち，AF, EF, AG, EG, AX, EX, A f U g, E f U g の8種類しか使えない，入れ子はOK.
- 命題論理と値に関する演算子はLTLに同じ.

簡単な例 ctl1.smv

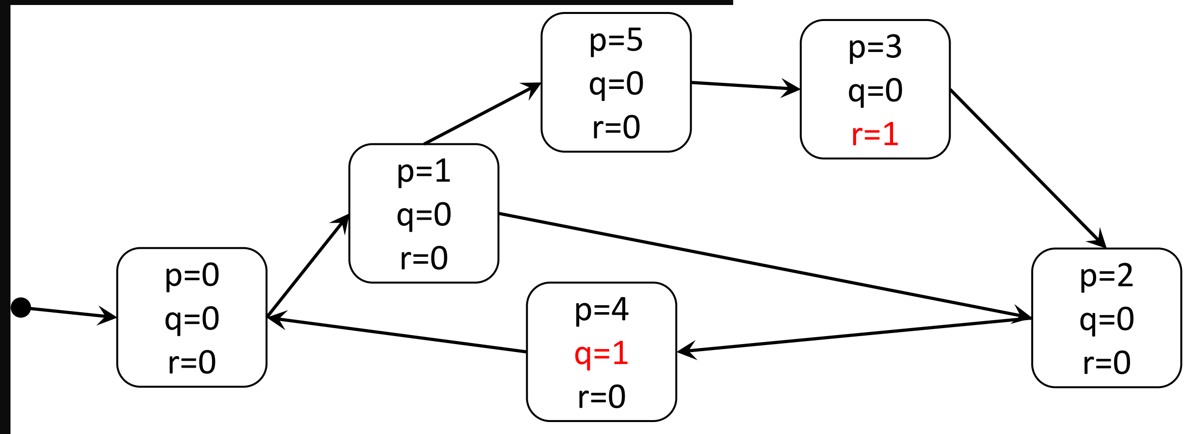
- 変数, p, q, r の値がかわる以下のような状態遷移を考える.
- AF (q=1) 全ての状態分岐で, いつかq=1となる.
- EF (r=1) r=1になるパスが存在する.
- AF (r=1) これは成り立たない.



ツールで実行チェックしてみる

```
*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification AF q = 1  is true
-- specification EF r = 1  is true
-- specification AF r = 1  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 1.1 <-
    p = 0
    q = 0
    r = 0
  -> State: 1.2 <-
    p = 1
  -> State: 1.3 <-
    p = 2
  -> State: 1.4 <-
    p = 4
    q = 1
  -> State: 1.5 <-
    p = 0
    q = 0

C:¥LEC¥SE¥NuSMV¥work>
```

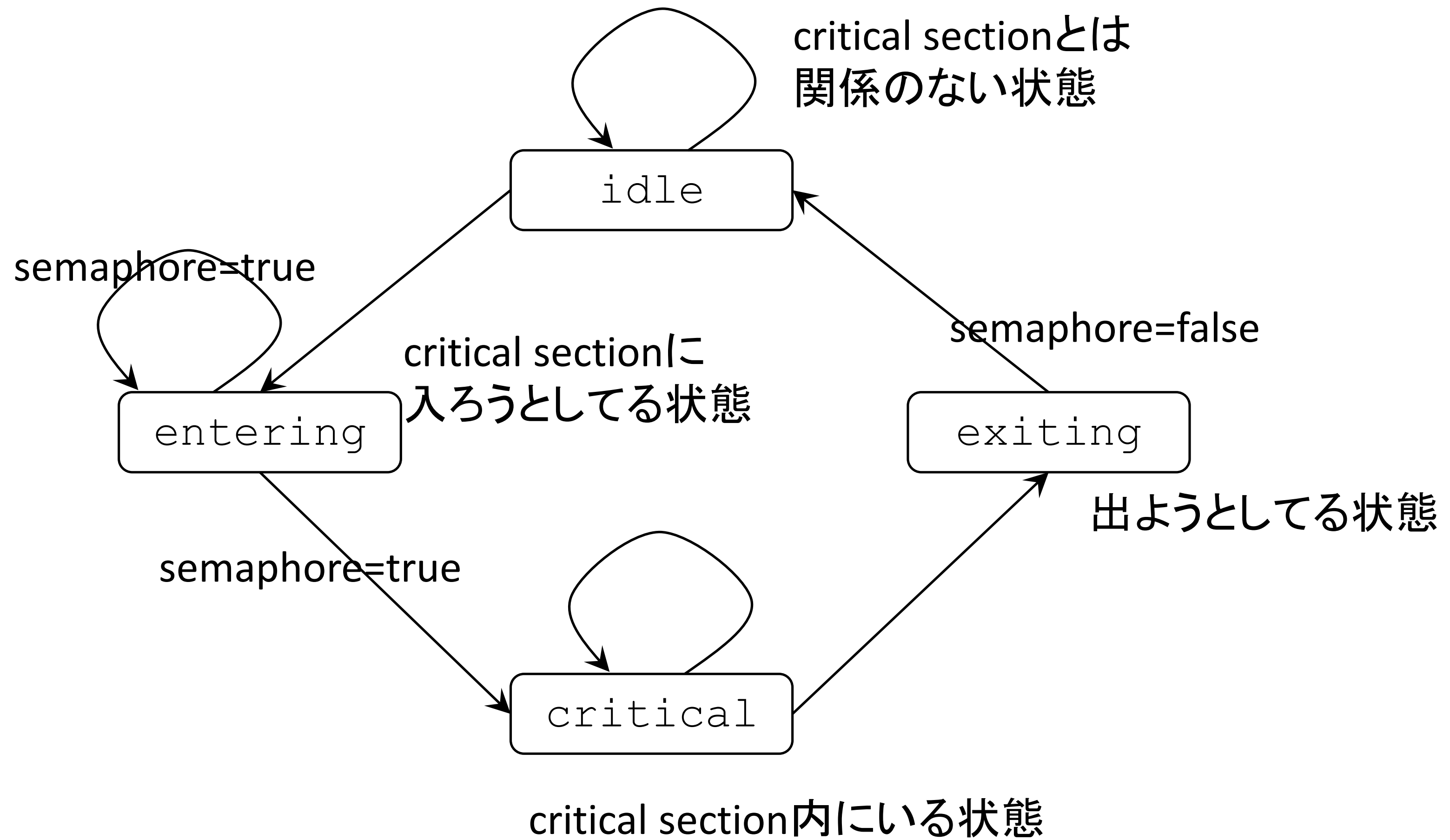
```
-- 状態仕様は ltl1と同じ

SPEC AF (q=1)
SPEC EF (r=1)
SPEC AF (r=1)
```



セマフォ

- 二つ(以上)のプロセスにおいて，一つのプロセスが独占したい資源があるとする.
  - 資源を独占していることを「クリティカルセクションに入っている」と呼んでいる.
  - 複数のプロセスがクリティカルセクションに同時に入らないようにするのが，この機構の役目.
- 独占が可能なように排他制御を行う方法として，各資源に対して「使用中/空き」を示す札をセマフォと(歴史的に)呼ぶ.
- 機構のミソは，いきなりクリティカルセクションに入るのではなく，一旦，セマフォの札を使用中にしてから入る所.
- OSでの定番の話題･･･

プロセスの状態



状態仕様

```
MODULE proc(semaphore)
VAR
  state : {idle,entering,critical,exiting};
ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle,entering};
      state = entering & !semaphore : critical;
      state = critical : {critical,exiting};
      state = exiting : idle;
      TRUE : state;
    esac;
  next(semaphore) :=
    case
      state = entering : TRUE;
      state = exiting : FALSE;
      TRUE : semaphore;
    esac;

JUSTICE
  running
```

```
MODULE main
VAR
  semaphore : boolean;
  proc1 : process proc(semaphore);
  proc2 : process proc(semaphore);
ASSIGN
  init(semaphore) := FALSE;
```

二つが同時にcsに入るを否定

- EF(proc1.state=critical & proc2.state=critical)
- 上記は二つのプロセスが同時にcritical section (cs)に入るパスがあることを示している.
- 下記のように否定されてるので安心.

```
-- specification EF (proc1.state = critical & proc2.state = critical)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    semaphore = FALSE
    proc1.state = idle
    proc2.state = idle
```

enterしてもcsにいけるとは限らない

- AG (proc1.state = entering -> AF proc1.state = critical)
- も否定される.
- proc1が先にenterしても，後からenterしたproc2が，先にsemaphoreを真にして，csに入る可能性があるから.

- 別途，実行例参照.

モデルチェック利用のポイント

- システムの設計なり仕様なりから，それを正しく反映したシステムの状態仕様が書けるか？
- 確認したい性質を時相論理式で正しく書けるか？

- 技術自体は確かなものだが，そこへのインプットは人間が準備するので，その辺が弱点といえる．

誤解の多いモデル検査での用語

- モデル検査での「モデル」は，証明と対比される「論理学でのモデル」のことである.
  - 命題変数に網羅的に真偽値を割り当て真になる組み合わせをチェックすること.
- だが，システムの仕様を書いた状態遷移表現を，ついついモデルと呼んでしまうが，これは誤解のもと.
- 同様に検査すべき時相論理式は仕様ではなく，確認したいシステムの性質であり，むしろ要求に近い.
- しかし，SMVでも，コレをSPECと呼んでしまっているのは誤解のもと.

Spin について

- NuSMV よりも有名なモデルチェックツール
- LTL しか使えない.
- 開発も利用もNuSMVより盛んに見える.

http://spinroot.com/spin/whatispin.html

以上