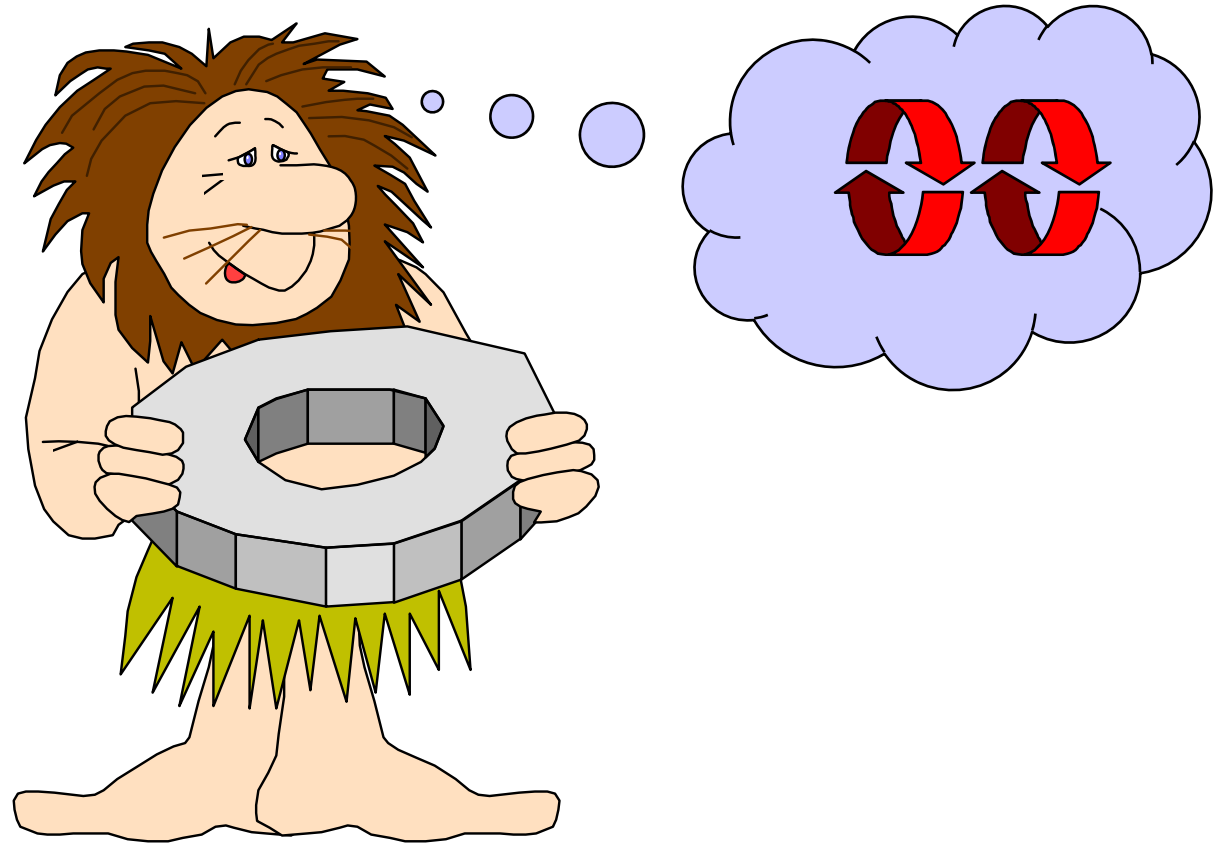


モデルに基づく設計

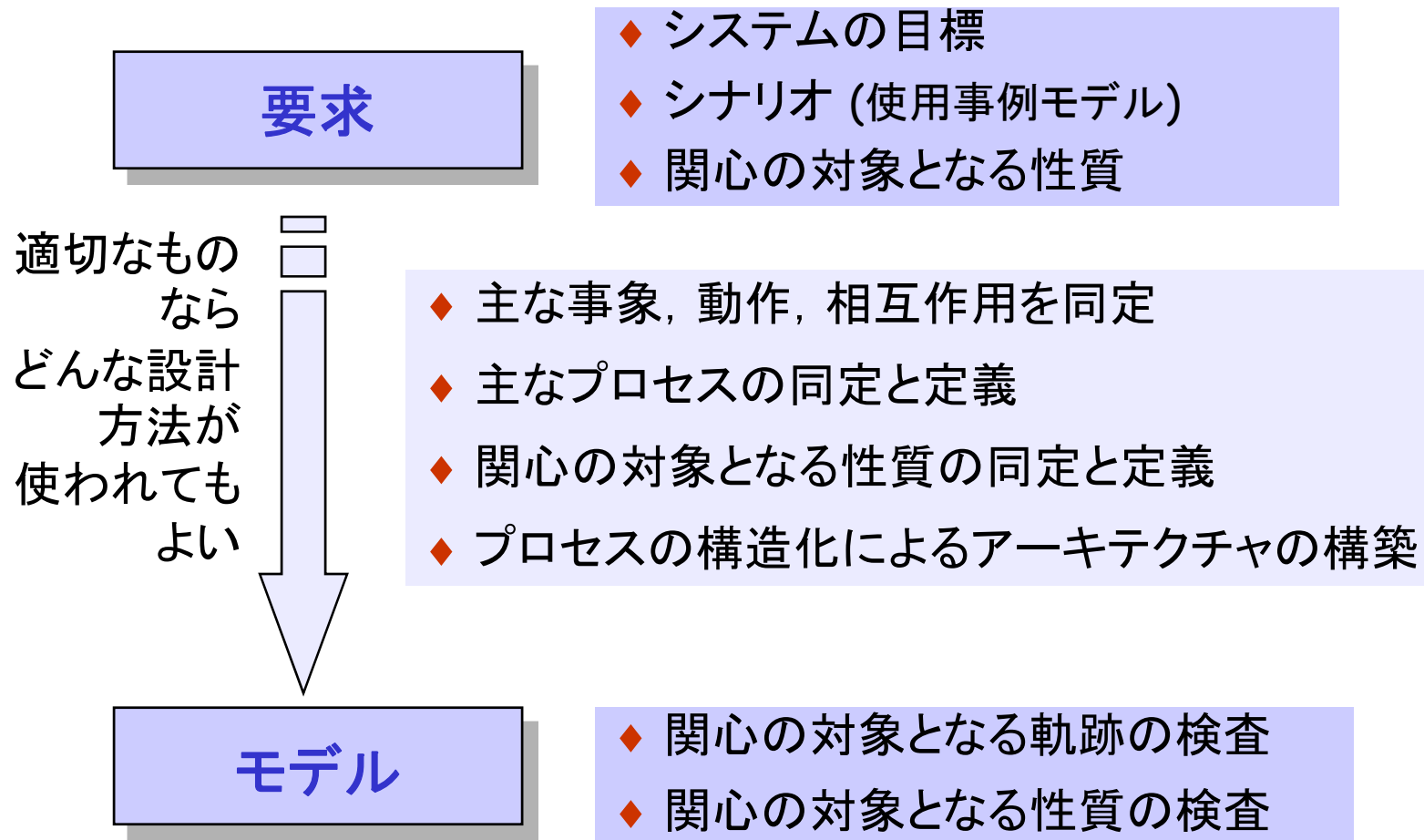


モデルに基づく設計

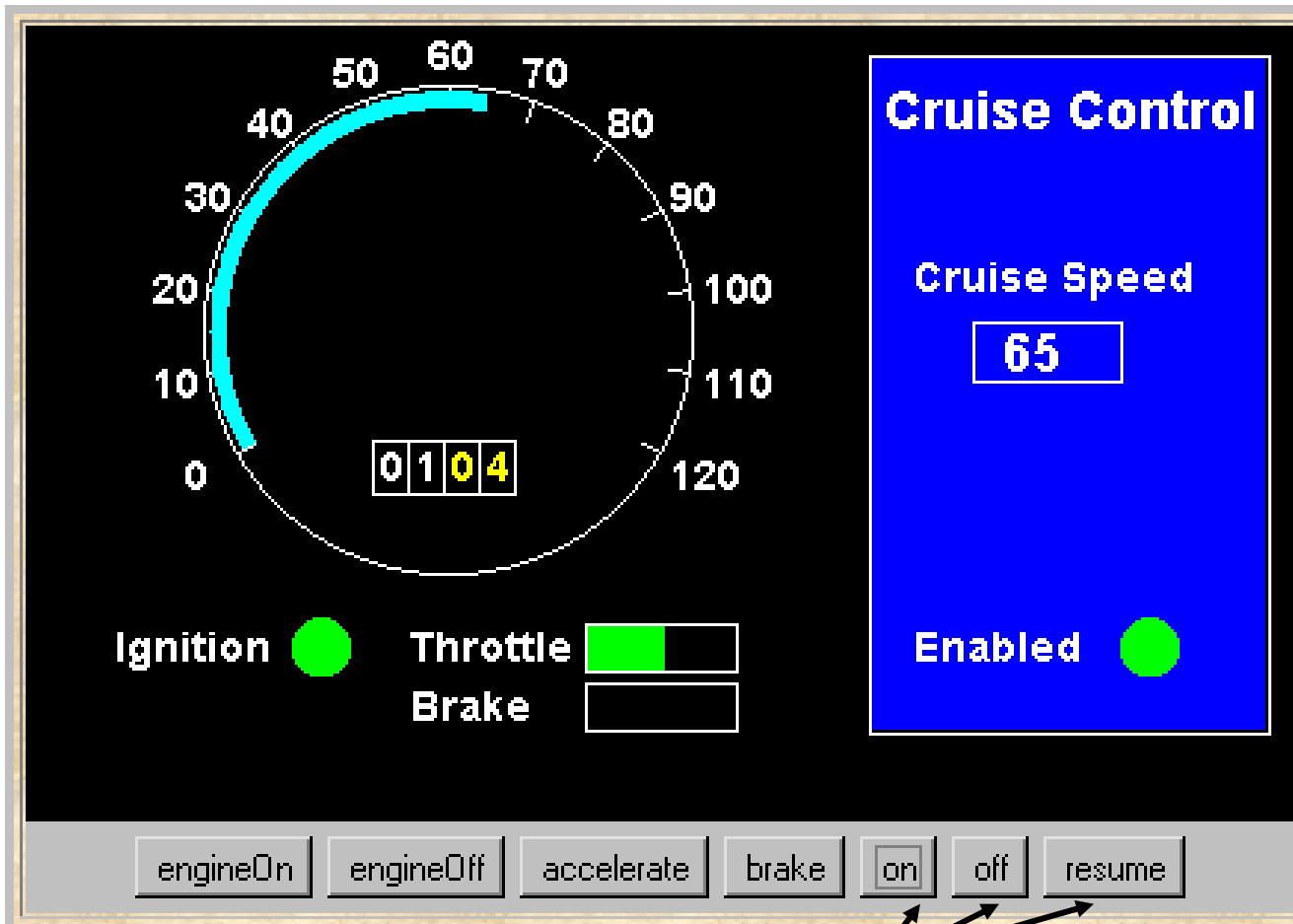
- 概念:** 設計プロセス:
要求からモデルをへて実装へ
- モデル:** 関心の対象となる性質の検査:
- 適切な(サブ)システムの**安全性**
- システム全体の**進行性**
- 実践:** モデルの解釈 - 実際のシステムの振舞いを推論
スレッドとモニタ

ねらい: 厳密な設計プロセス

8.1 要求からモデルへ



巡航制御システム – 要求



ボタン

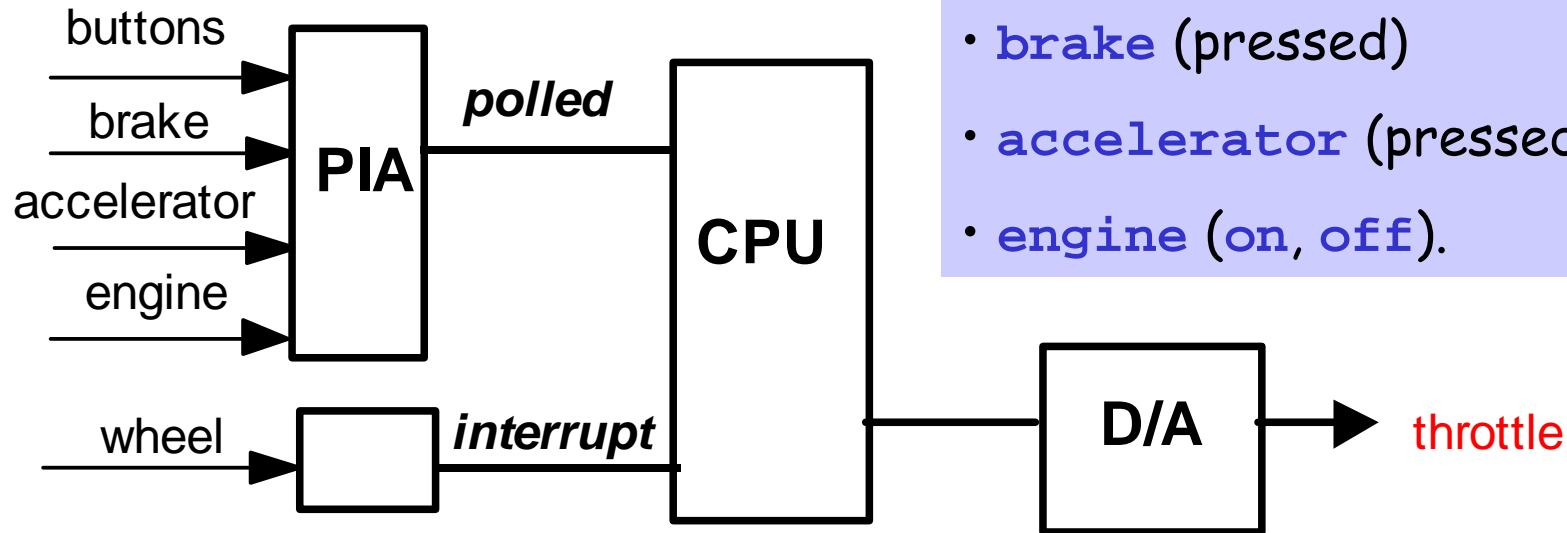
車がエンジンが点火している状態で **on** ボタンが押されると、現在の速度が記録され、制御システムが動作可能となる: *車の速度を記録された値に維持する。*

ブレーキかアクセルが踏まれるか、**off** ボタンが押されると、制御システムは動作不能となる. **resume** か **on** ボタンが押されると、制御システムは再び動作可能となる.

巡航制御システム – ハードウェア

並列インタフェース・アダプタ (PIA) が100msecごとにポーリングされる。センサーの動作が記録される:

- buttons (**on**, **off**, **resume**)
- **brake** (pressed)
- **accelerator** (pressed)
- **engine** (**on**, **off**).

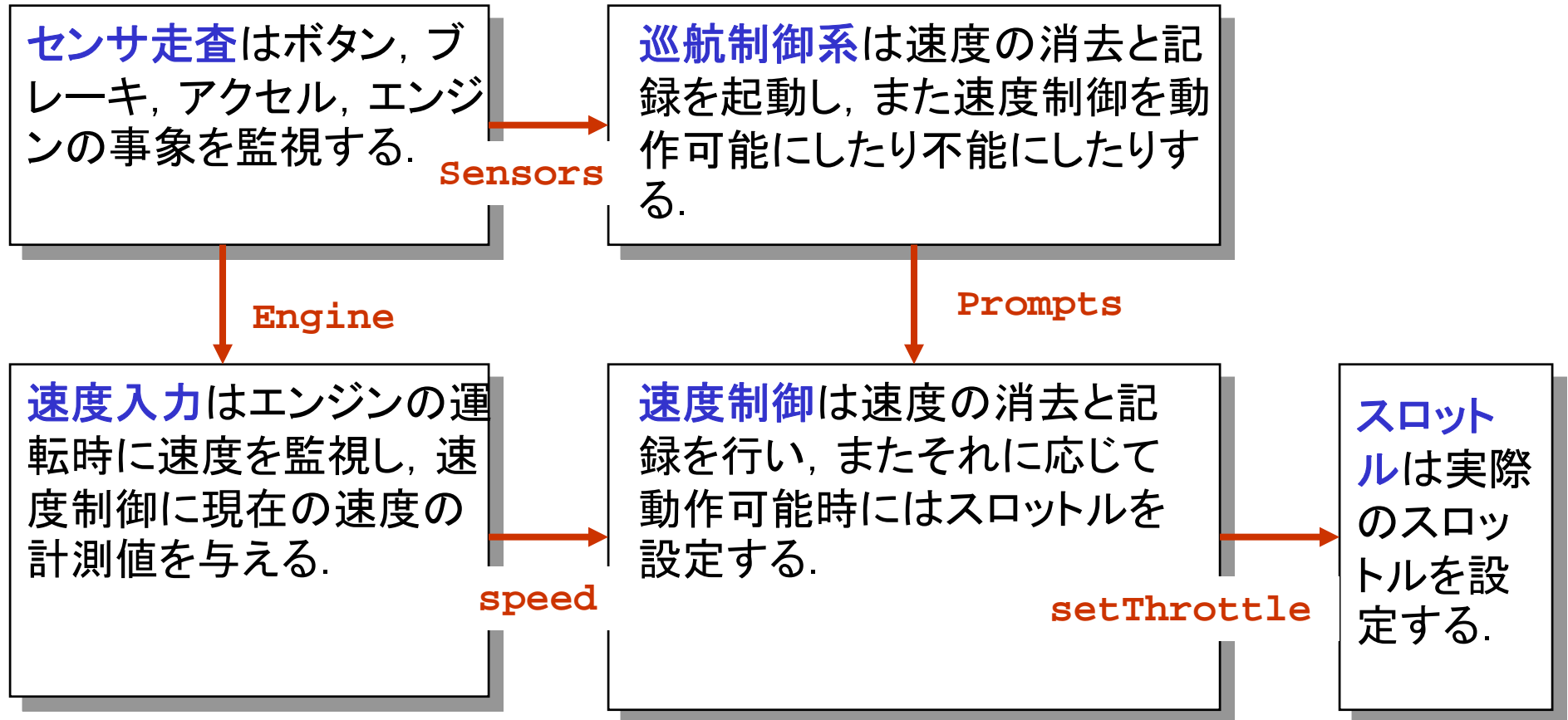


車輪回転センサーは、車の**速度** W を計算するために割り込みを生成する。

出力: 巡航制御システムは、デジタル/アナログ変換器を介して**スロットル**を設定することにより、車の速度を制御する。

モデル – 概要設計

◆ プロセスと相互作用の概要を決める.



モデル – 設計

◆ 主な事象, 動作, 相互作用

`on, off, resume, brake, accelerator`
`engine on, engine off,`
`speed, setThrottle`
`clearSpeed, recordSpeed,`
`enableControl, disableControl`

Sensors

Prompts

◆ 主なプロセスの同定

`Sensor Scan, Input Speed,`
`Cruise Controller, Speed Control and`
`Throttle`

◆ 主な性質の同定

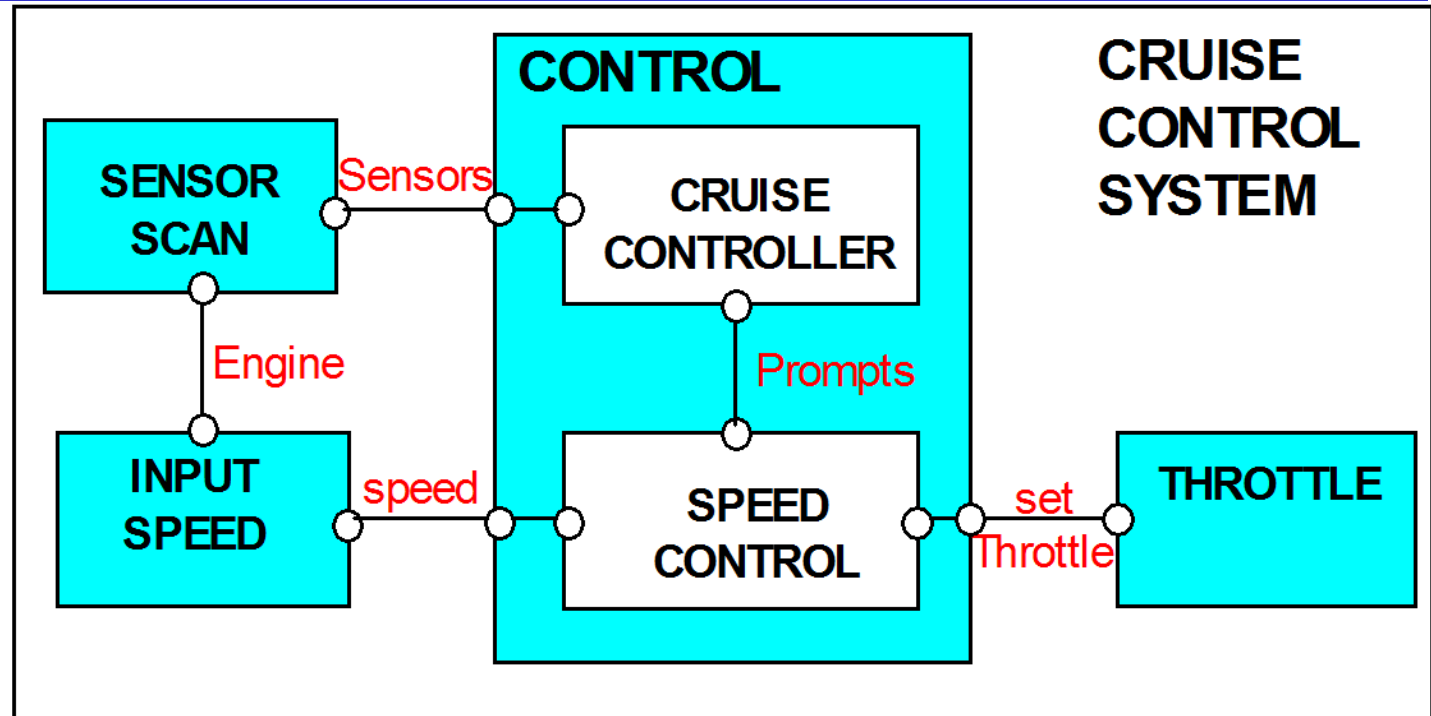
安全性 - `off, brake, accelerator`が押されている時は無効にする.

◆ 各プロセスの定義と構造化

モデル – 構造, 動作, 相互作用

CONTROL
システムは2
つのプロセス
として構造化
され.

主な動作と相
互作用は示し
た通り.



set **Sensors** = {engineOn,engineOff,on,off,
resume,brake,accelerator}

set **Engine** = {engineOn,engineOff}

set **Prompts** = {clearSpeed,recordSpeed,
enableControl,disableControl}

モデルの詳細化 – プロセスの定義

```
SENSORSCAN = ({Sensors} -> SENSORSCAN).  
    // monitor speed when engine on  
INPUTSPEED = (engineOn -> CHECKSPEED),  
CHECKSPEED = (speed -> CHECKSPEED  
    | engineOff -> INPUTSPEED  
    ).  
    // zoom when throttle set  
THROTTLE = (setThrottle -> zoom -> THROTTLE).  
    // perform speed control when enabled  
SPEEDCONTROL = DISABLED,  
DISABLED = ({speed,clearSpeed,recordSpeed}->DISABLED  
    | enableControl -> ENABLED  
    ),  
ENABLED = ( speed -> setThrottle -> ENABLED  
    | {recordSpeed,enableControl} -> ENABLED  
    | disableControl -> DISABLED  
    ).
```

モデルの詳細化 – プロセスの定義

```
// enable speed control when cruising,  
// disable when off, brake or accelerator pressed  
CRUISECONTROLLER = INACTIVE,  
INACTIVE = (engineOn -> clearSpeed -> ACTIVE),  
ACTIVE   = (engineOff -> INACTIVE  
            | on -> recordSpeed -> enableControl -> CRUISING  
            ),  
CRUISING = (engineOff -> INACTIVE  
            | { off, brake, accelerator }  
              -> disableControl -> STANDBY  
            | on -> recordSpeed -> enableControl -> CRUISING  
            ),  
STANDBY  = (engineOff -> INACTIVE  
            | resume -> enableControl -> CRUISING  
            | on -> recordSpeed -> enableControl -> CRUISING  
            ).
```

モデル - CONTROL サブシステム

```
|| CONTROL = (CRUISECONTROLLER
               || SPEEDCONTROL
               ).
```

特定の軌跡を検査するため動かしてみよう:

- エンジンのスイッチが入れられ, onボタンが押されれば, 制御は動作可能となるか
- ブレーキが踏まれれば 制御は動作不能となるか
- resume が押されれば, 制御はふたたび動作可能となるか

しかし, 網羅的な検査をするためには分析が必要である:

安全性: off, brake
または accelerator
が押されたら, 制御は
動作不能となるか
進行性: どの動作もい
つかは選択されるか

モデル – 安全性の性質

安全性の検査は**合成的**である。サブシステム・レベルで違反がなければ、それが他のサブシステムと合成されても違反は起こさない。

その理由は、ある安全性の性質の**ERROR**状態がサブシステムのLTSで到達不能ならば、そのサブシステムを含むどんな並列合成でも、やはりその**ERROR**状態には到達しないからである。したがって...

安全性の性質は、その性質が関わる適切なシステムまたはサブシステムと合成されなければならない。その性質がアルファベット内の動作を検査できるようにするため、それらの動作はシステムにおいて隠蔽してはならない。

モデル – 安全性の性質

```
property CRUISESAFETY =  
  ( {off, accelerator, brake, disableControl} -> CRUISESAFETY  
  | {on, resume} -> SAFETYCHECK  
  ),  
SAFETYCHECK =  
  ( {on, resume} -> SAFETYCHECK  
  | {off, accelerator, brake} -> SAFETYACTION  
  | disableControl -> CRUISESAFETY  
  ),  
SAFETYACTION = ( disableControl -> CRUISESAFETY ).
```

LTSは?

```
|| CONTROL = (CRUISECONTROLLER  
              || SPEEDCONTROL  
              || CRUISESAFETY  
              ).
```

CRUISESAFETY の
違反は起こるか?

モデル – 安全性の性質

LTSAを用いて安全性分析を行うと以下の**違反**が見つかる:

```
Trace to property violation in CRUISESAFETY:  
  engineOn  
  clearSpeed  
  on  
  recordSpeed  
  enableControl  
  engineOff  
  off  
  off
```

異常な状況!

エンジンがかけられオン・ボタンが押されてシステムが起動された後、**エンジンのスイッチが切られても**、制御システムは動作不能とされないようだ。

モデル - 安全性の性質

エンジンのスイッチが再び入れられたらどうなるか? さらにアニメーションを使って調べることができる ...

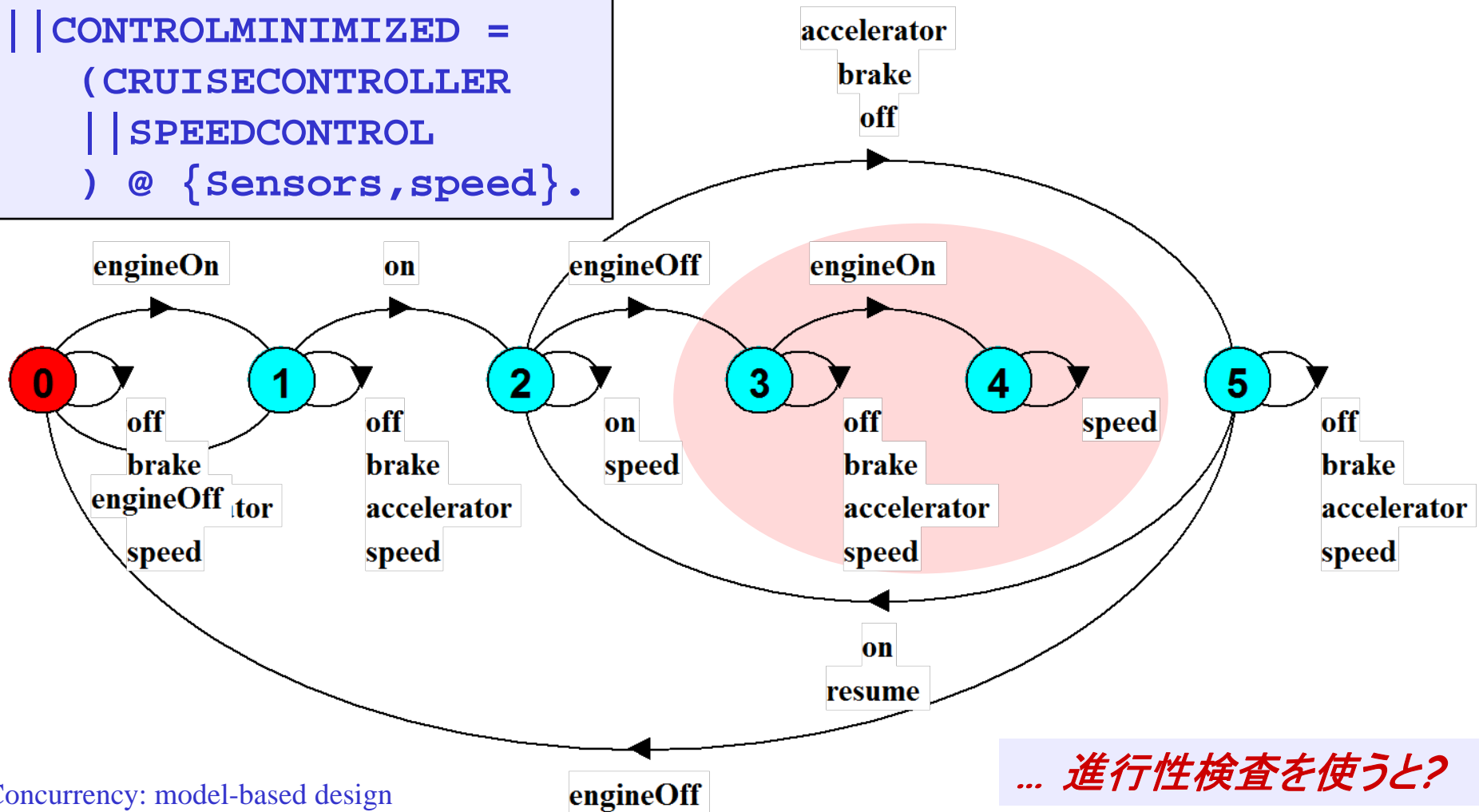
```
engineOn  
clearSpeed  
on  
recordSpeed  
enableControl  
engineOff  
engineOn  
speed  
setThrottle  
speed  
setThrottle  
...
```

エンジンのスイッチが再び入れられると、車は加速し振り切れてしまう。

... LTSを使うと? 動作隠蔽と最小化によりLTS図の大きさを縮小し解釈しやすくなる ...

CONTROLMINIMIZEDモデルのLTS

```
minimal
|| CONTROLMINIMIZED =
  ( CRUISECONTROLLER
  || SPEEDCONTROL
  ) @ { Sensors, speed }.
```



Concurrency: model-based design

... 進行性検査を使うと?

モデル - 進行性の性質

Progress violation for actions:

```
{accelerator, brake, clearSpeed, disableControl,
enableControl, engineOff, engineOn, off, on,
recordSpeed, resume}
```

Trace to terminal set of states:

```
engineOn
clearSpeed
on
recordSpeed
enableControl
engineOff
engineOn
```

Cycle in terminal set:

```
speed
setThrottle
```

Actions in terminal set:

```
{setThrottle, speed}
```

安全性と動作隠蔽を入れ
ずにモデルの進行性を検
査すると

モデル – 改訂版巡航制御システム

CRUISECONTROLLERを変更し, エンジンが止められたら制御は**動作不能**となるようにする.

```
...
CRUISING =(engineOff -> disableControl -> INACTIVE
           |{ off,brake,accelerator} -> disableControl -> STANDBY
           |on->recordSpeed->enableControl->CRUISING
           ),
...

```

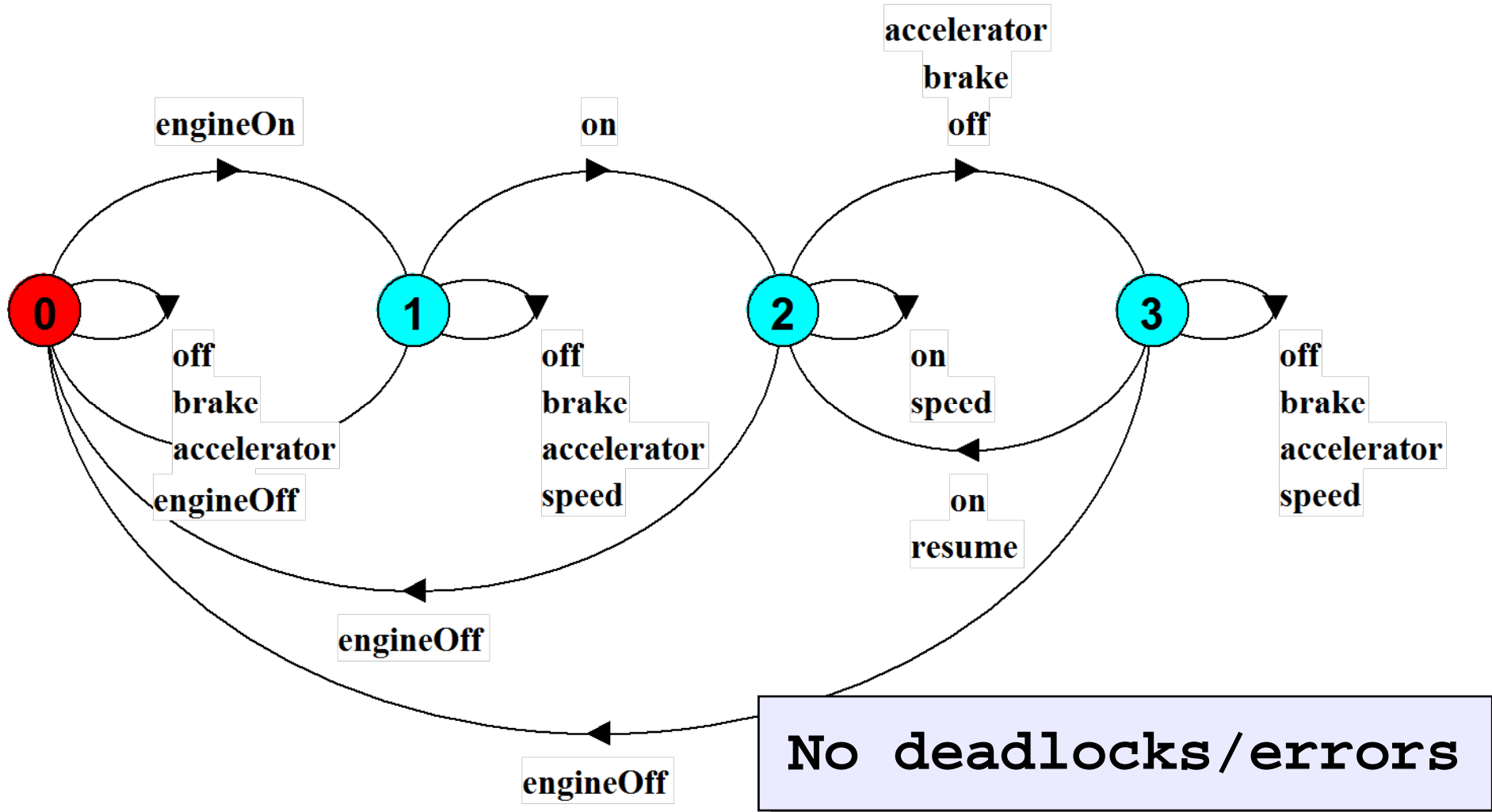
安全性の性質を変更:

```
property IMPROVEDSAFETY =
  (DisableActions,disableControl,engineOff} -> IMPROVEDSAFETY
  |{on,resume} -> SAFETYCHECK
  ),
SAFETYCHECK = ({on,resume} -> SAFETYCHECK
               |{DisabelActions,engineOff} -> SAFETYACTION
               |disableControl -> IMPROVEDSAFETY
               ),
SAFETYACTION =(disableControl -> IMPROVEDSAFETY)

```

これでOK か?

改訂版 CONTROLMINIMIZED



モデル 分析

これでシステム全体を構成できる.

```
|| CONTROL =  
  ( CRUISECONTROLLER | | SPEEDCONTROL | | CRUISESAFETY  
    )@ { sensors, speed, setThrottle } .  
|| CRUISECONTROLSYSTEM =  
  ( CONTROL | | SENSORSCAN | | INPUTSPEED | | THROTTLE ) .
```

デッドロックは?
安全性は?

No deadlocks/errors

進行性は?

モデル - 進行性の性質

進行性の検査は**合成的ではない**。サブシステム・レベルで違反がなくても、他のサブシステムと合成した時に違反が起こるかもしれない。

これはサブシステムの動作が進行性を満たしても、そのサブシステムがその振舞いを制約するような他のサブシステムと合成されたときに到達不能になるかもしれないからである。したがって...

進行性の検査は、安全性の検査を十分に行った後の完成した対象システムに対して実施しなければならない。

進行性は？

No progress
violations detected.

モデル - システムの感度

不利な条件下での進行性はどうか? システムの感度を調べてみよう.

```
|| SPEEDHIGH = CRUISECONTROLSYSTEM << {speed}.
```

Progress violation for actions:

```
{engineOn, engineOff, on, off, brake, accelerator,  
resume, setThrottle, zoom}
```

Path to terminal set of states:

```
engineOn
```

```
tau
```

Actions in terminal set:

```
{speed}
```

speed 動作の優先度
に, システムは影響を
受けるかもしれない.

モデルの解釈

モデルを用いてシステムの感度を示すことができる。

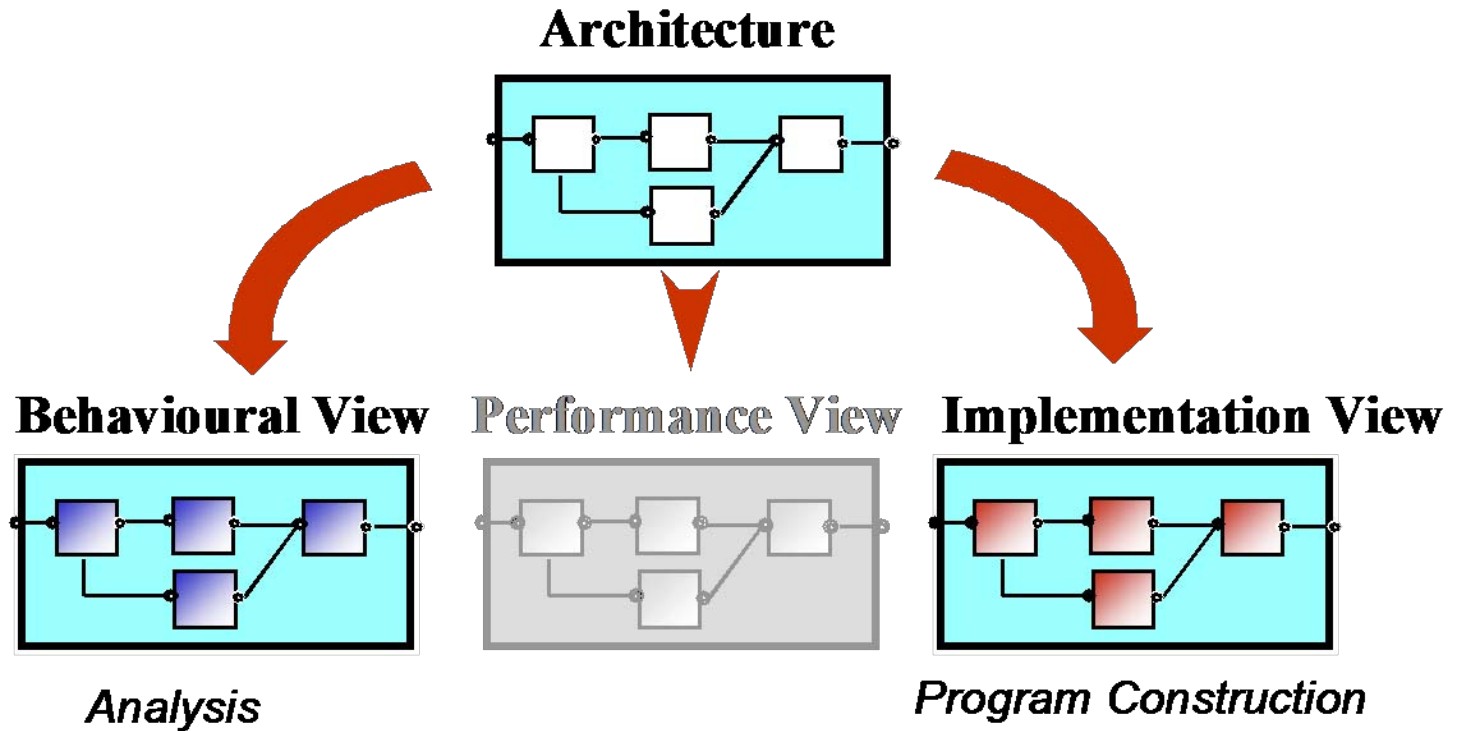
もしモデルで検出された誤りのある状況が実装システムで起こりうる場合は、モデルを改訂してそのような違反が回避されることが保証できる設計に変えなければならない。

しかし、もし実際のシステムがそのような振舞いを示さないと考えられるなら、モデルの改訂は不要である。

モデルの解釈と実装との対応は、モデルの設計と分析がどれほど意味をもちまたどれほど十分なものであるかを決定するのに、重要である。

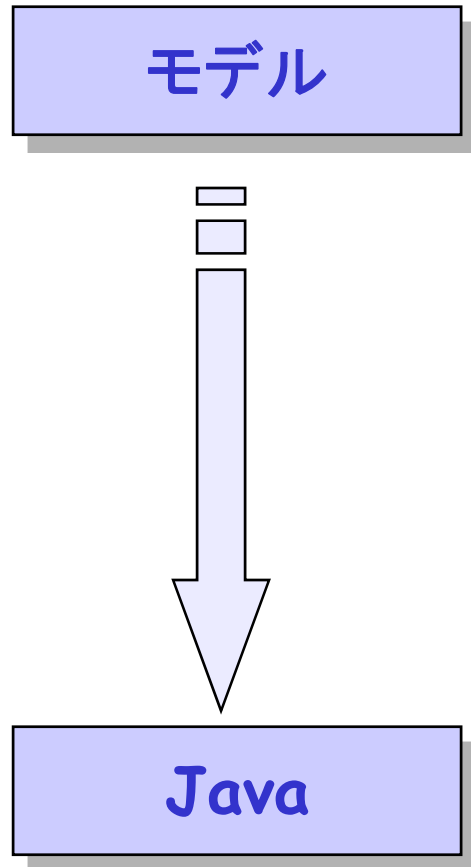
設計アーキテクチャの中心的役割

設計アーキテクチャは、システムの大まかな組織と大域的構造を、その構成要素によって記述するものである。



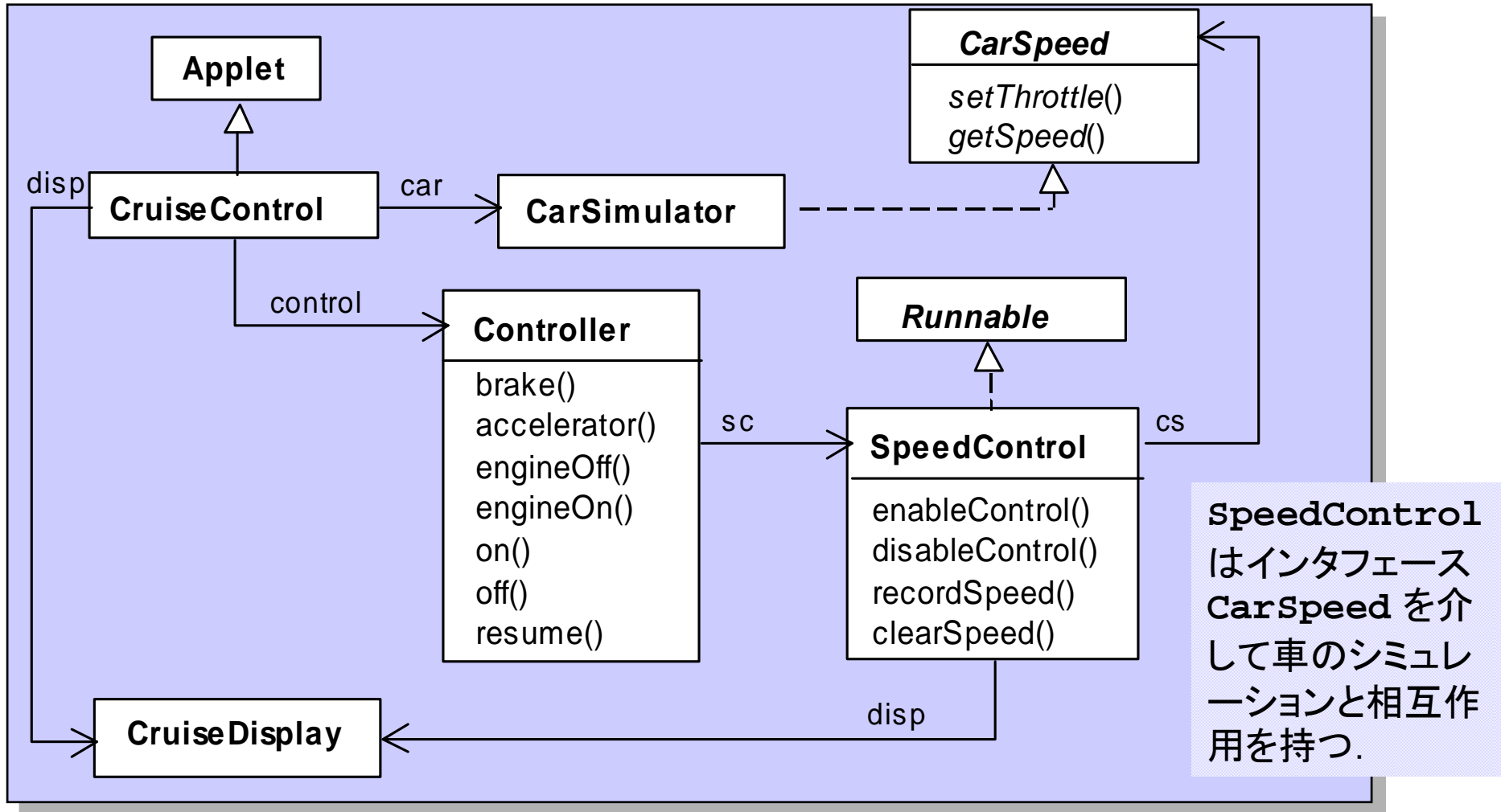
分析モデルとその実装は、この基本設計構造のそれぞれの視点からの詳細化と考えるべきである。

8.2 モデルから実装へ



- ◆ 主な能動的実体を同定する
 - スレッドとして実装すべきもの
- ◆ 主な(共有される)受動的実体を同定する
 - モニタとして実装すべきもの
- ◆ 相互作用のある表示環境を同定する
 - 関連づけられたクラス群として実装すべきもの
- ◆ それらのクラスをクラス図として構造化する

巡航制御システム - クラス図



巡航制御システム - Controller クラス

```
class Controller {
    final static int INACTIVE = 0; // cruise controller states
    final static int ACTIVE   = 1;
    final static int CRUISING = 2;
    final static int STANDBY  = 3;
    private int controlState = INACTIVE; // initial state
    private SpeedControl sc;

    Controller(CarSpeed cs, CruiseDisplay disp)
        {sc=new SpeedControl(cs,disp);}

    synchronized void brake(){
        if (controlState==CRUISING )
            {sc.disableControl(); controlState=STANDBY; }
    }

    synchronized void accelerator(){
        if (controlState==CRUISING )
            {sc.disableControl(); controlState=STANDBY; }
    }

    synchronized void engineOff(){
        if(controlState!=INACTIVE) {
            if (controlState==CRUISING) sc.disableControl();
            controlState=INACTIVE;
        }
    }
}
```

Controller
は受動的実体
である。すな
わち事象に応
答する。した
がって、モニタ
として実装す
る。

巡航制御システム – Controller クラス

```
synchronized void engineOn(){
    if(controlState==INACTIVE)
        {sc.clearSpeed(); controlState=ACTIVE;}
}

synchronized void on(){
    if(controlState!=INACTIVE){
        sc.recordSpeed(); sc.enableControl();
        controlState=CRUISING;
    }
}

synchronized void off(){
    if(controlState==CRUISING )
        {sc.disableControl(); controlState=STANDBY;}
}

synchronized void resume(){
    if(controlState==STANDBY)
        {sc.enableControl(); controlState=CRUISING;}
}
}
```

これはモデル
からの直接の
翻訳である。

巡航制御システム - SpeedControlクラス

```
class SpeedControl implements Runnable {
    final static int DISABLED = 0; //speed control states
    final static int ENABLED = 1;
    private int state = DISABLED; //initial state
    private int setSpeed = 0; //target speed
    private Thread speedController;
    private CarSpeed cs; //interface to control speed
    private CruiseDisplay disp;

    SpeedControl(CarSpeed cs, CruiseDisplay disp){
        this.cs=cs; this.disp=disp;
        disp.disable(); disp.record(0);
    }

    synchronized void recordSpeed(){
        setSpeed=cs.getSpeed(); disp.record(setSpeed);
    }

    synchronized void clearSpeed(){
        if (state==DISABLED) {setSpeed=0;disp.record(setSpeed);}
    }

    synchronized void enableControl(){
        if (state==DISABLED) {
            disp.enable(); speedController= new Thread(this);
            speedController.start(); state=ENABLED;
        }
    }
}
```

SpeedControlは能動的実体である。実行可能とされた時は、新しいスレッドが作られ、それが周期的に車の速度を得てスロットルを設定する。

巡航制御システム - SpeedControlクラス

```
synchronized void disableControl(){
    if (state==ENABLED) {disp.disable(); state=DISABLED;}
}

public void run() {           // the speed controller thread
    try {
        while (state==ENABLED) {
            Thread.sleep(500);
            if (state==ENABLED) synchronized(this) {
                double error = (float)(setSpeed-cs.getSpeed())/6.0;
                double steady = (double)setSpeed/12.0;
                cs.setThrottle(steady+error); //simplified feed back control
            }
        }
    } catch (InterruptedException e) {}
    speedController=null;
}
}
```

SpeedControl は(局所変数を更新する)同期アクセスメソッドとスレッドの両者を組み合わせたクラスの例である。

まとめ

◆ 概念

- 設計プロセス:
 要求からモデルを経て実装へ
- 設計アーキテクチャ

◆ モデル

- 関心の対象となる性質を検査
 安全性: 安全性の性質を適切な(サブ)システムに結合
 進行性: 最終的な対象システムモデルに進行検査を適用

◆ 実践

- モデルの解釈 - 実際のシステムの振舞いを推論
- スレッドとモニタ

ねらい: 厳密な設計プロセス