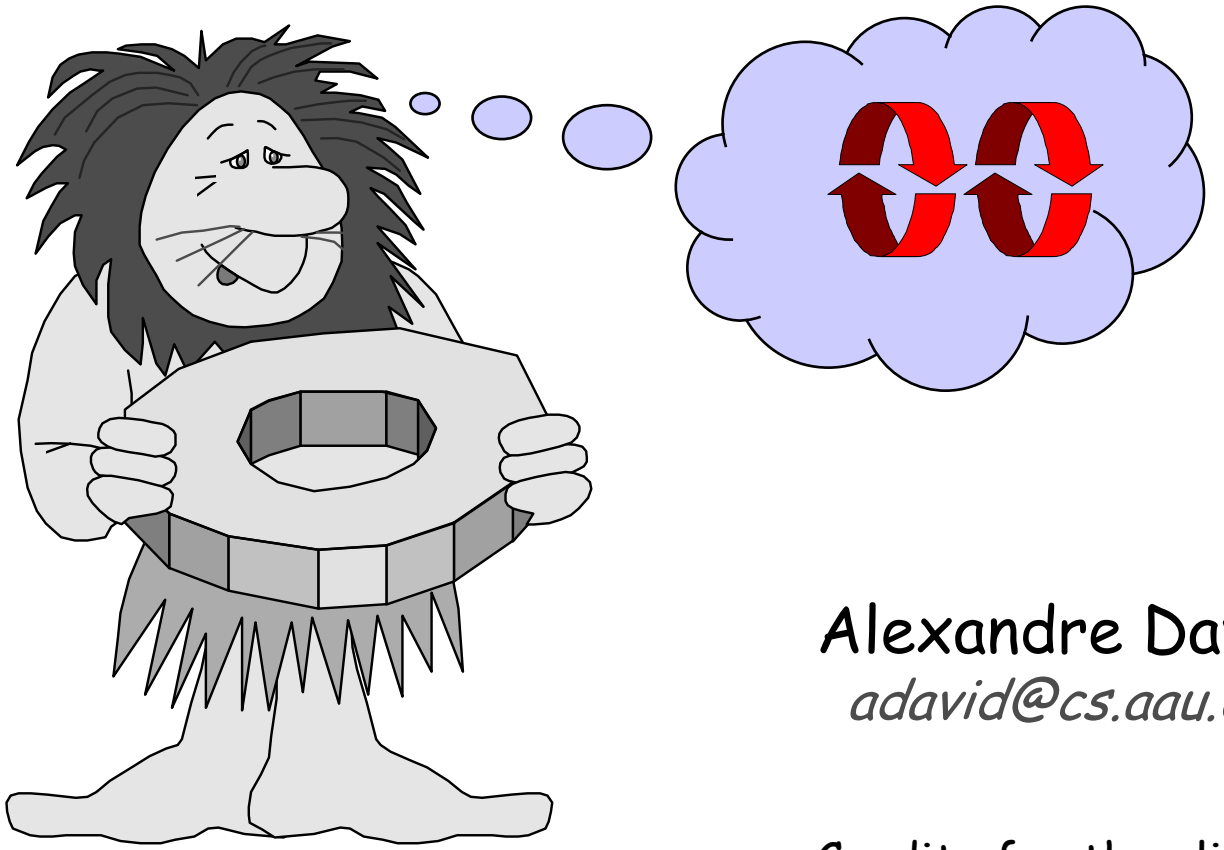
8 - Model-Based Design



Alexandre David
adavid@cs.aau.dk

Credits for the slides:
Claus Braband
Jeff Magee & Jeff Kramer

Repetition - Safety & Liveness Properties

Concepts:

Properties: true for every possible execution
Safety: nothing bad happens
Liveness: something good *eventually* happens

Models:

Safety: no reachable ERROR/STOP state
Progress: an action is *eventually* executed
(fair choice and action priority)

Practice:

Threads and monitors

Aim: property satisfaction.

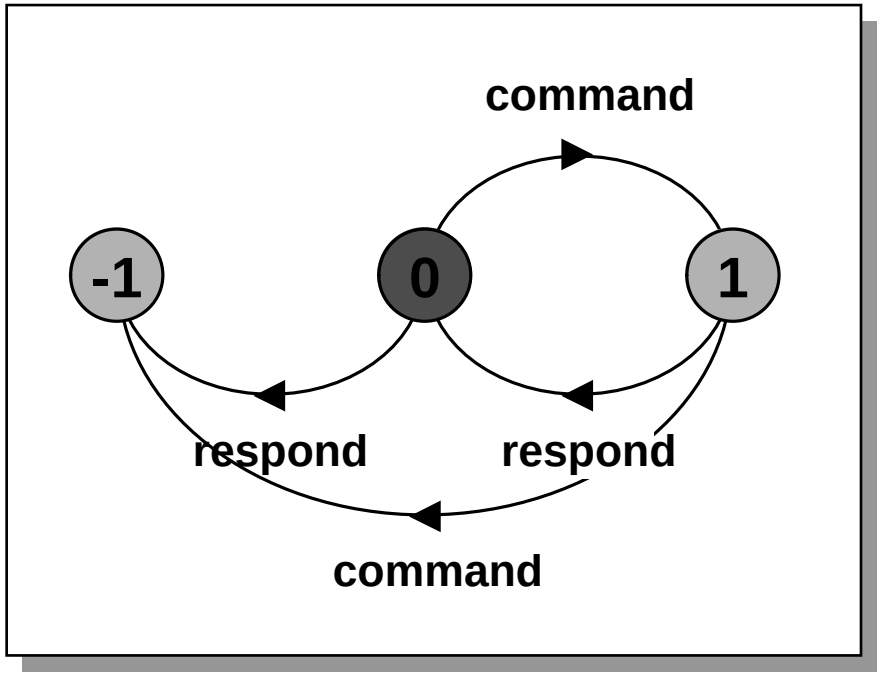
Repetition - Safety

A **safety property** asserts that nothing bad happens.

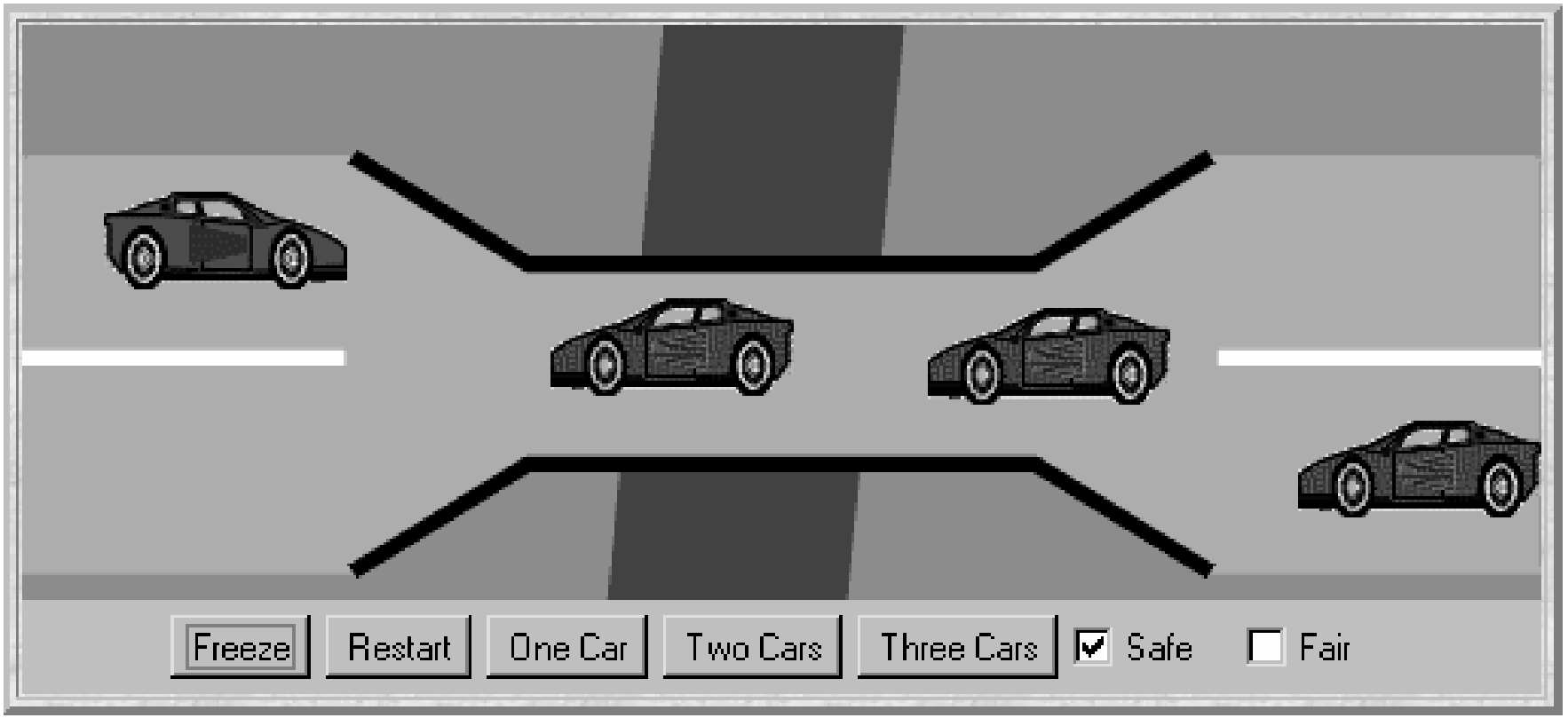
- **ERROR** conditions state what is **not** required (~ exceptions).
- In complex systems, it is usually better to specify **safety properties** by stating directly what **is** required.

```
property SAFE_ACTUATOR =
    (command ->
        respond ->
            SAFE_ACTUATOR).
```

```
ACTUATOR =
  (command ->
    (respond -> ACTUATOR
    |command -> ERROR)
  |respond -> ERROR).
```



Repetition - Single Lane Bridge Problem



Repetition - Safety Property "ONEWAY"

We now specify a **safety property** to check that cars do not collide!:

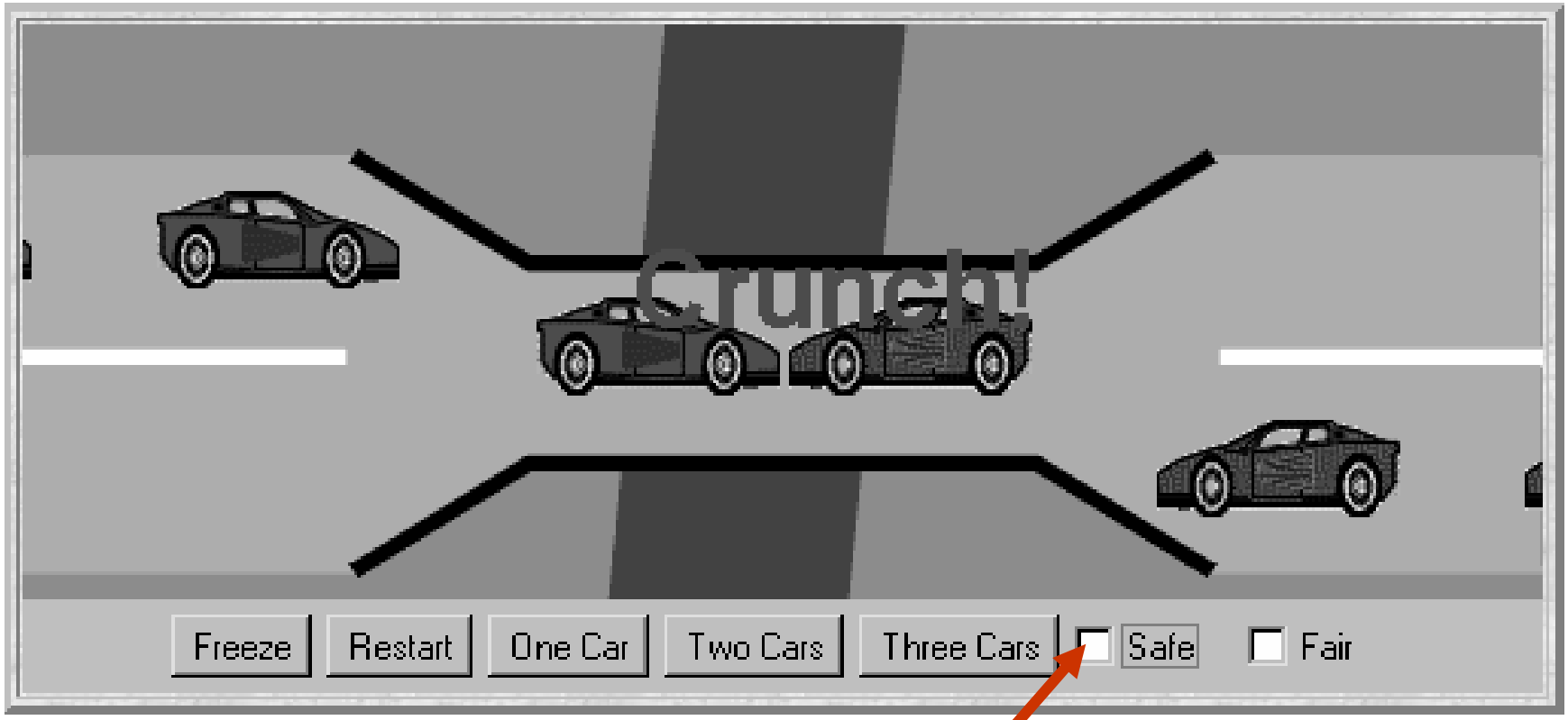
```
property ONEWAY = (red[ID].enter  -> RED[1]
                  |blue[ID].enter -> BLUE[1]),

RED[i:ID] = (red[ID].enter  -> RED[i+1]
            |when (i==1) red[ID].exit  -> ONEWAY
            |when (i>1)  red[ID].exit  -> RED[i-1]),

BLUE[j:ID]= (blue[ID].enter -> BLUE[j+1]
            |when (j==1) blue[ID].exit -> ONEWAY
            |when (j>1)  blue[ID].exit -> BLUE[j-1]).
```

When the bridge is empty, either a red or a blue car may enter. While red cars are on the bridge only red cars can enter; similarly for blue cars.

Repetition - Single Lane Bridge

Without bridge controller =>

```
Trace to property violation in ONEWAY:
     red.1.enter
     blue.1.enter
```



Repetition - Liveness

A **safety** property **asserts** that nothing bad happens.

A **liveness** property **asserts** that something good *eventually* happens.

*Does every car eventually get an opportunity to cross the bridge (i.e., make **progress**)?*

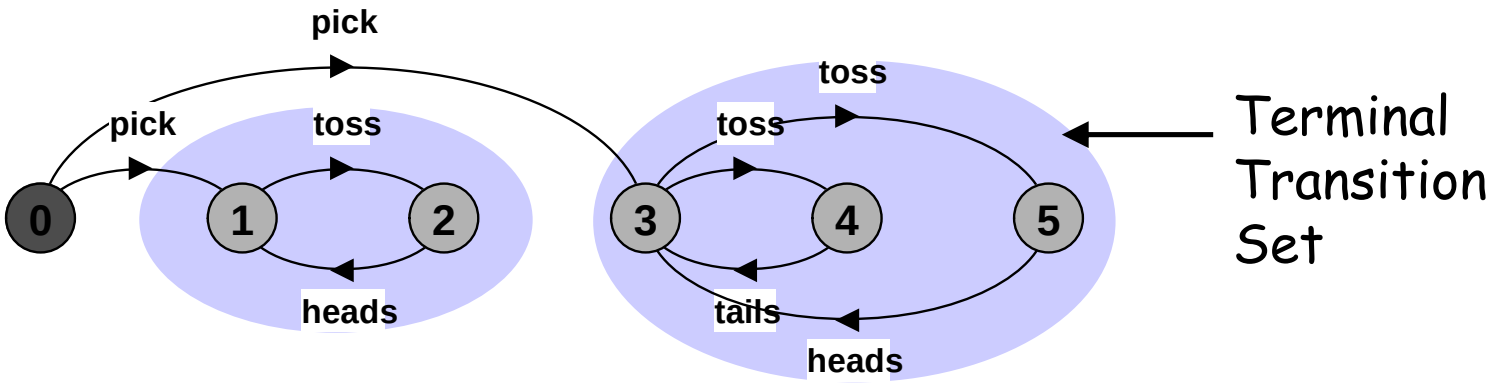
A progress property **asserts** that it is *always* the case that an action is *eventually* executed.

Progress is the opposite of *starvation* = the name given to a concurrent programming situation in which an action is never executed.

Repetition - Progress Properties

Suppose that there were **two** possible coins that could be picked up: *a regular coin* and *a **trick** coin*

```
TWOCOIN = (pick->COIN | pick->TRICK),
COIN    = (toss->heads->COIN | toss->tails->COIN).
TRICK   = (toss->heads->TRICK),
```



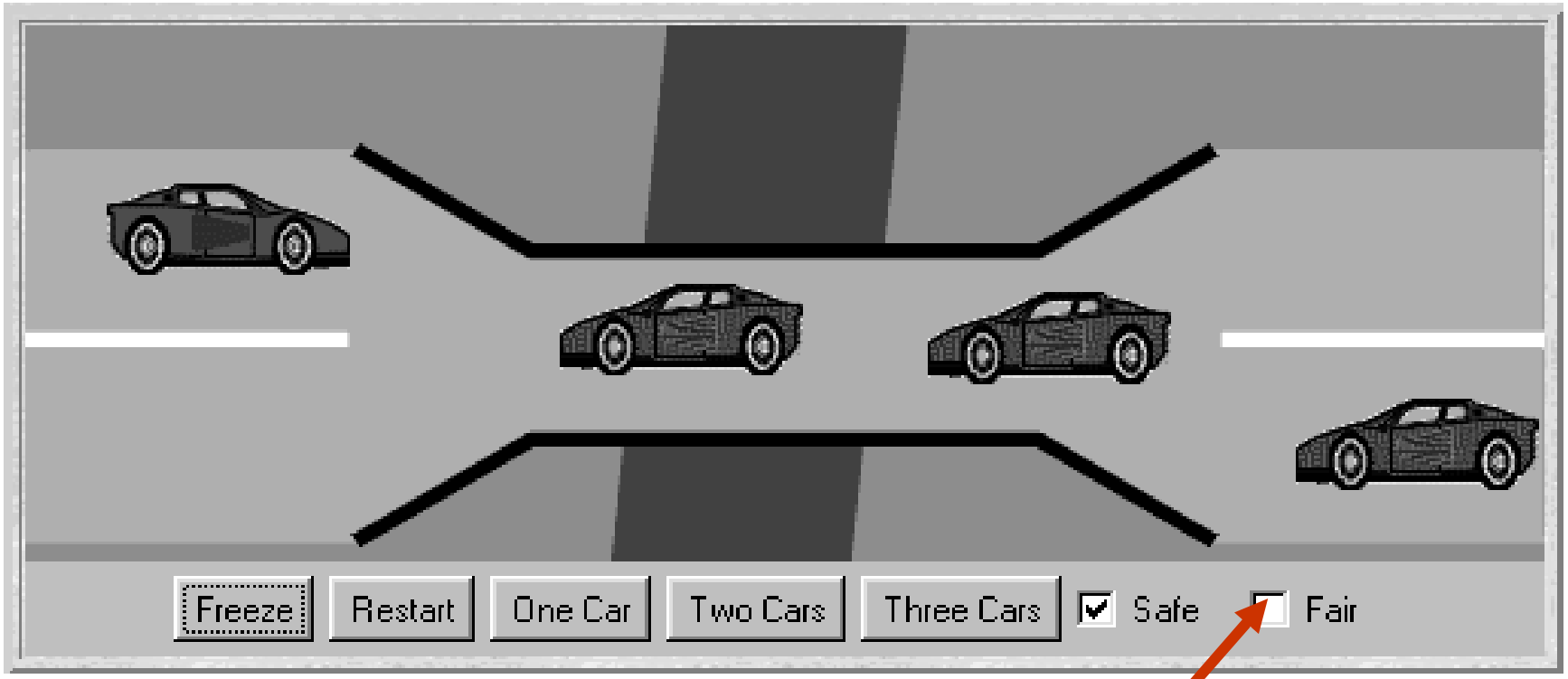

```
progress HEADS = {heads} ?   ☺
progress TAILS = {tails} ?   ☹
```

Repetition - FairBridge

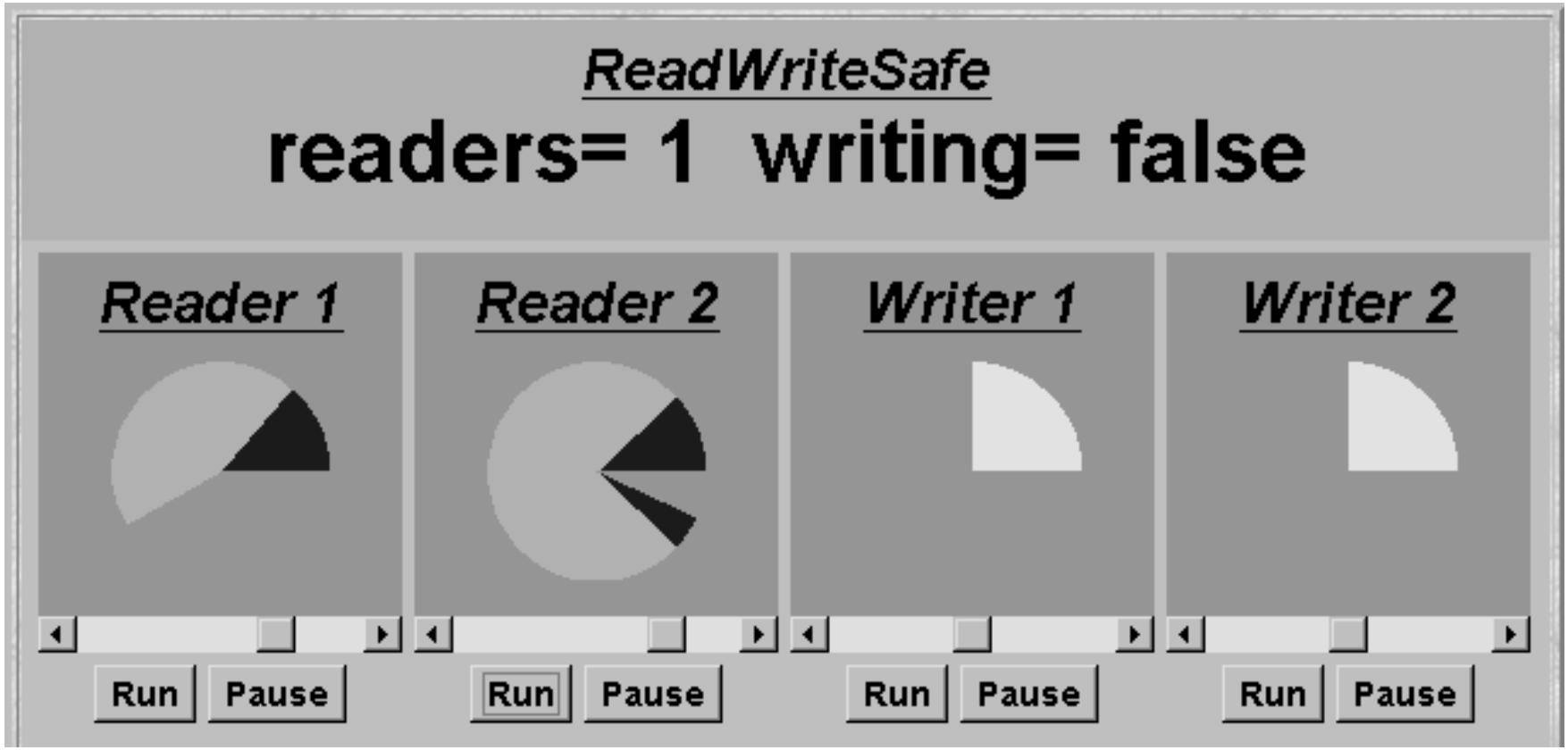
Low/high action priorities:

```
P >> {low1, …, lowN}
P << {high1, …, highN}
```

```
||CongestedBridge = (SingleLaneBridge)
                    >> {red[ID].exit,blue[ID].exit}.
```



Repetition - Readers and Writers



Model-Based Design

Concepts:

design process:

- from requirements to **models**
- from **models** to implementations

Models:

check properties of interest:

- safety of the appropriate (sub-)system
- progress of the overall system

Practice:

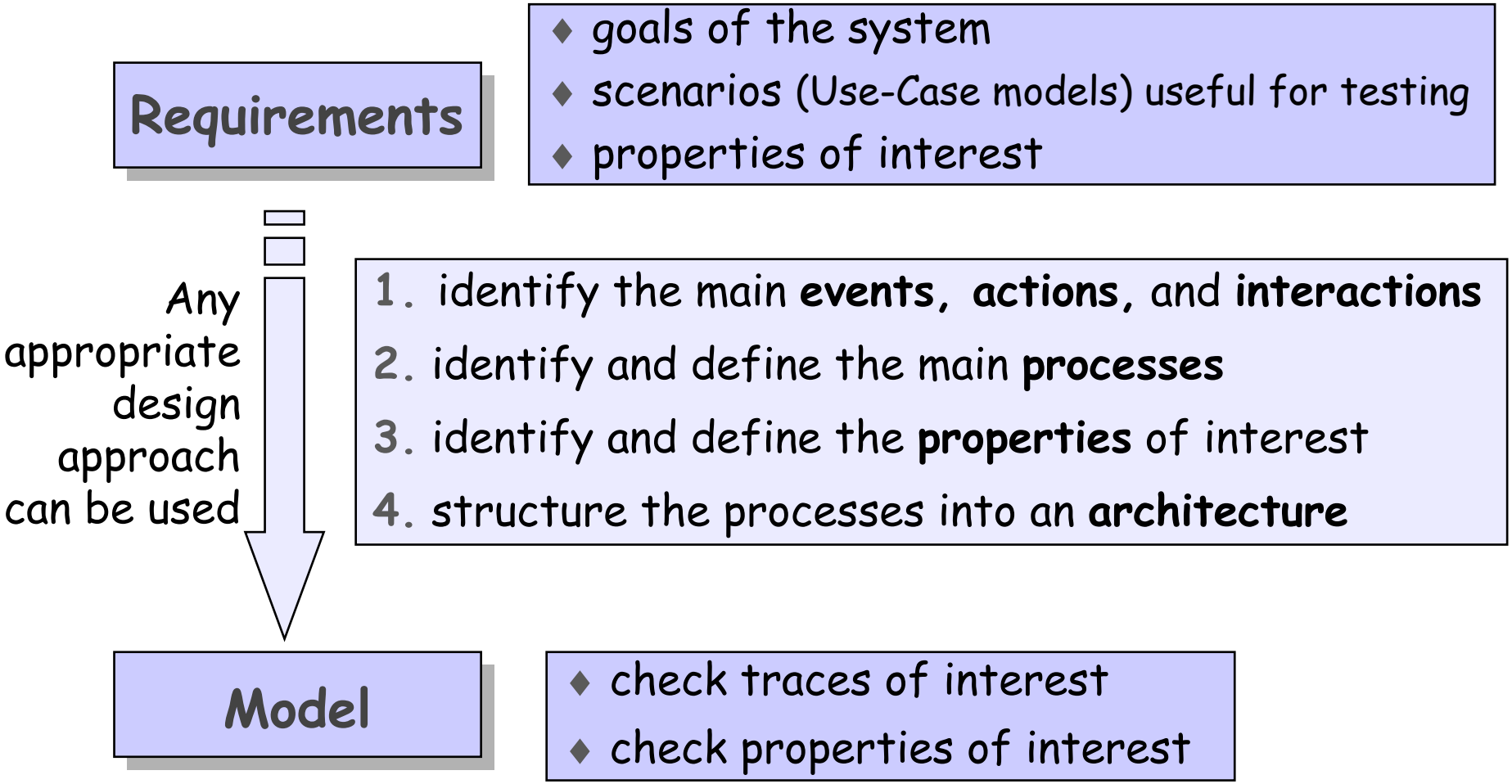
model "interpretation":

- to infer actual system behaviour

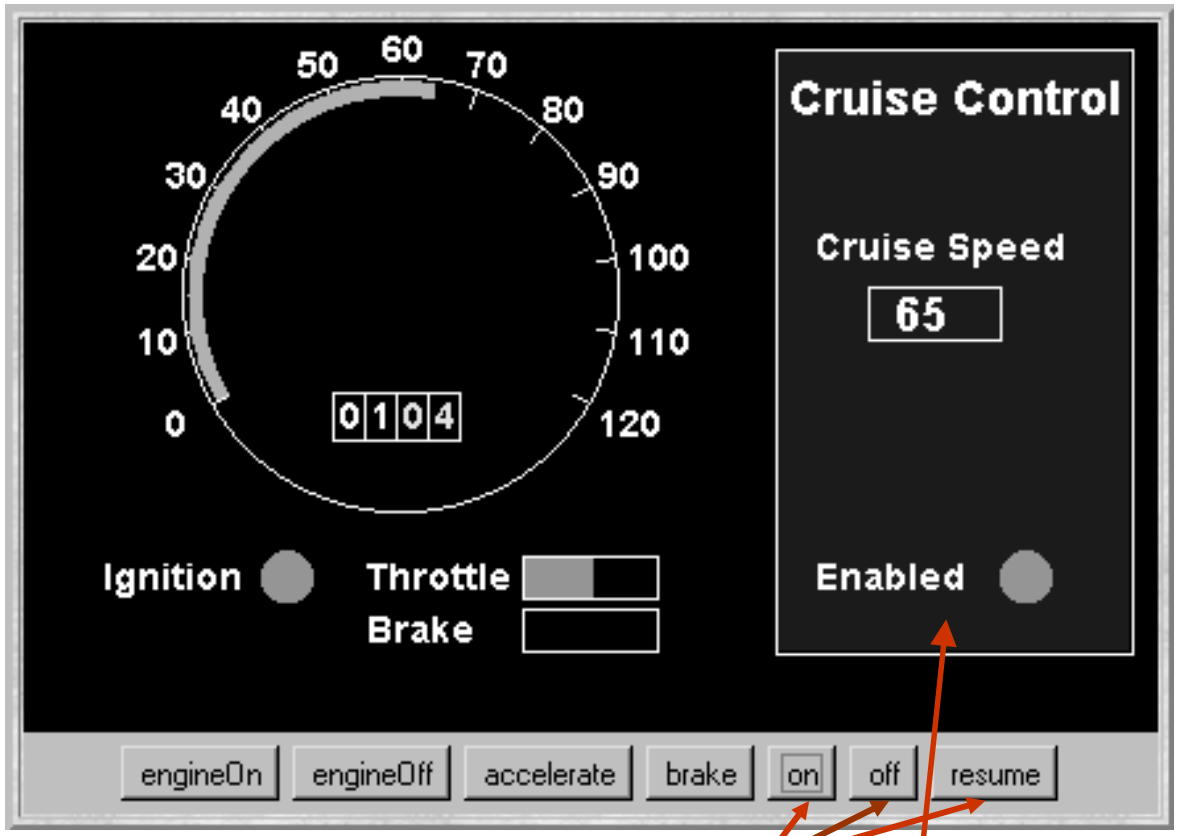
active threads and *passive monitors*

Aim: rigorous design process.

8.1 From Requirements to Models



Example: A Cruise Control System - Requirements



Requirements:

When the car ignition is switched on and the **'on'** button is pressed, the current speed is recorded and the system is *enabled:*

It maintains the speed of the car at the recorded setting.

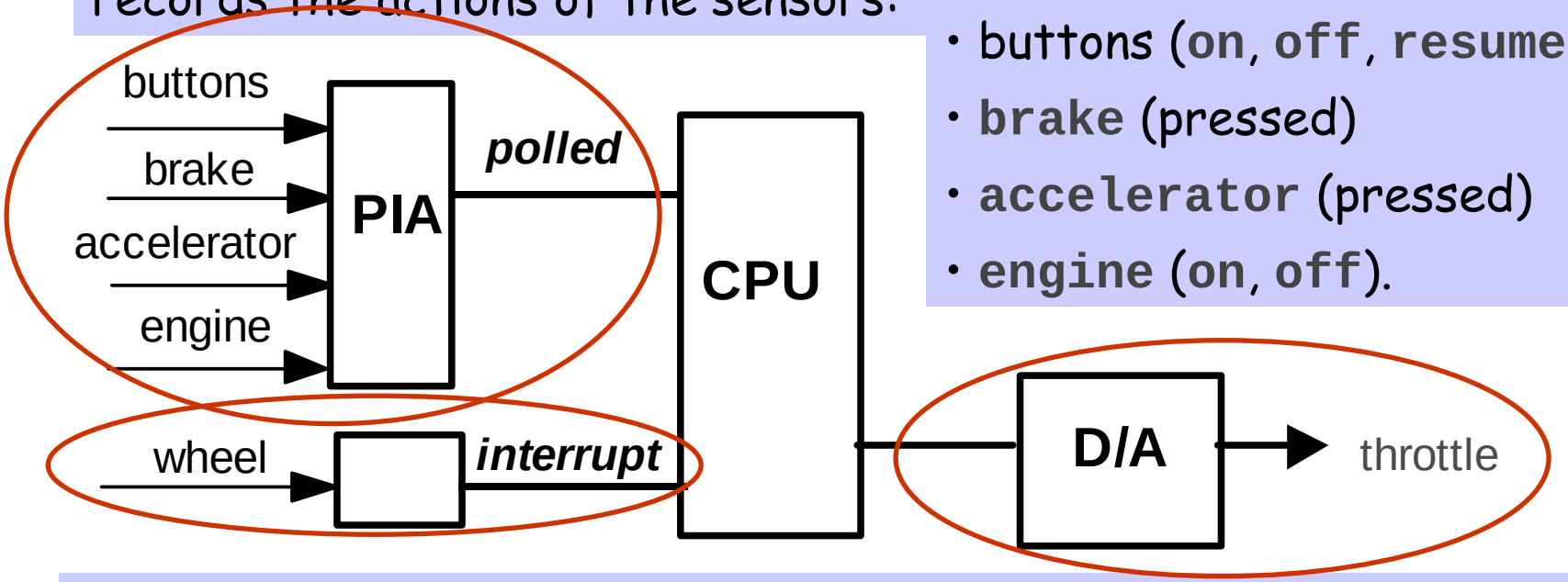
Pressing the **'brake'**, **'accelerator'** or **'off'** button *disables* the system.

Pressing **'resume'** or **'on'** *re-enables* the system.

A Cruise Control System - Hardware

Parallel Interface Adapter (PIA) is polled every 100msec. It records the actions of the sensors:

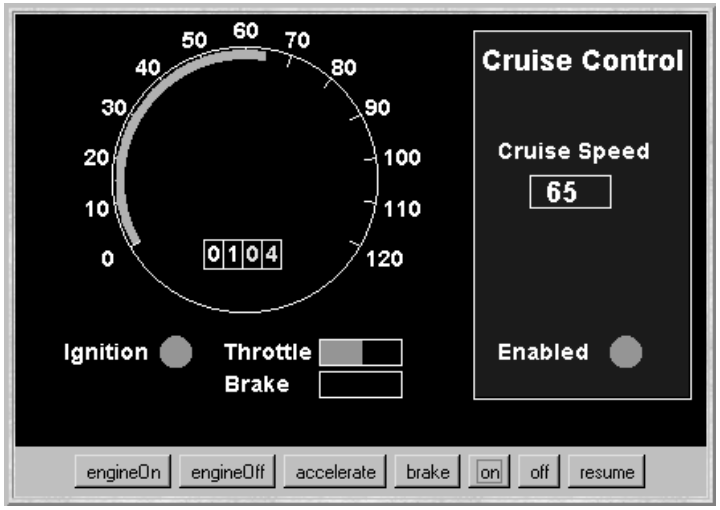
- buttons (`on`, `off`, `resume`)
- `brake` (pressed)
- `accelerator` (pressed)
- `engine` (`on`, `off`).

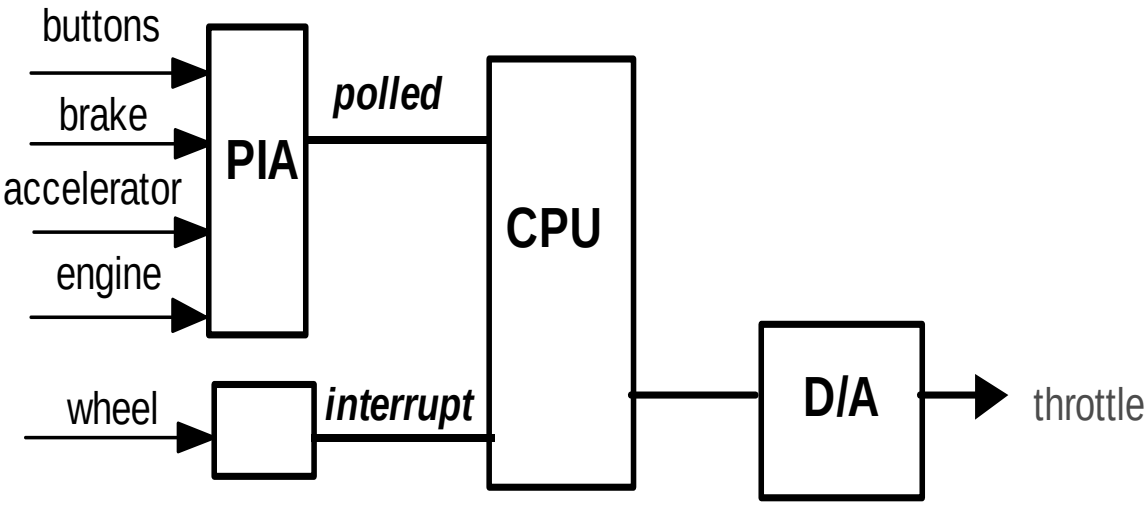


Wheel revolution sensor generates interrupts to enable the car `speed` to be calculated.

Output: The cruise control system controls the car speed by setting the `throttle` via the digital-to-analogue (D/A) converter.

Model - Design





1. Identify the main **events**, **actions**, and **interactions**:

`engineOn`, **`engineOff`**, **`accelerator`**, **`brake`**, **`on`**, **`off`**, **`resume`**, } **Sensors**

`speed`, **`setThrottle`**, **`zoom`**
`clearSpeed`, **`recordSpeed`**,
`enableControl`, **`disableControl`**. } **Prompts**

Model - Design

1. Identify the main **events**, **actions**, and **interactions**:

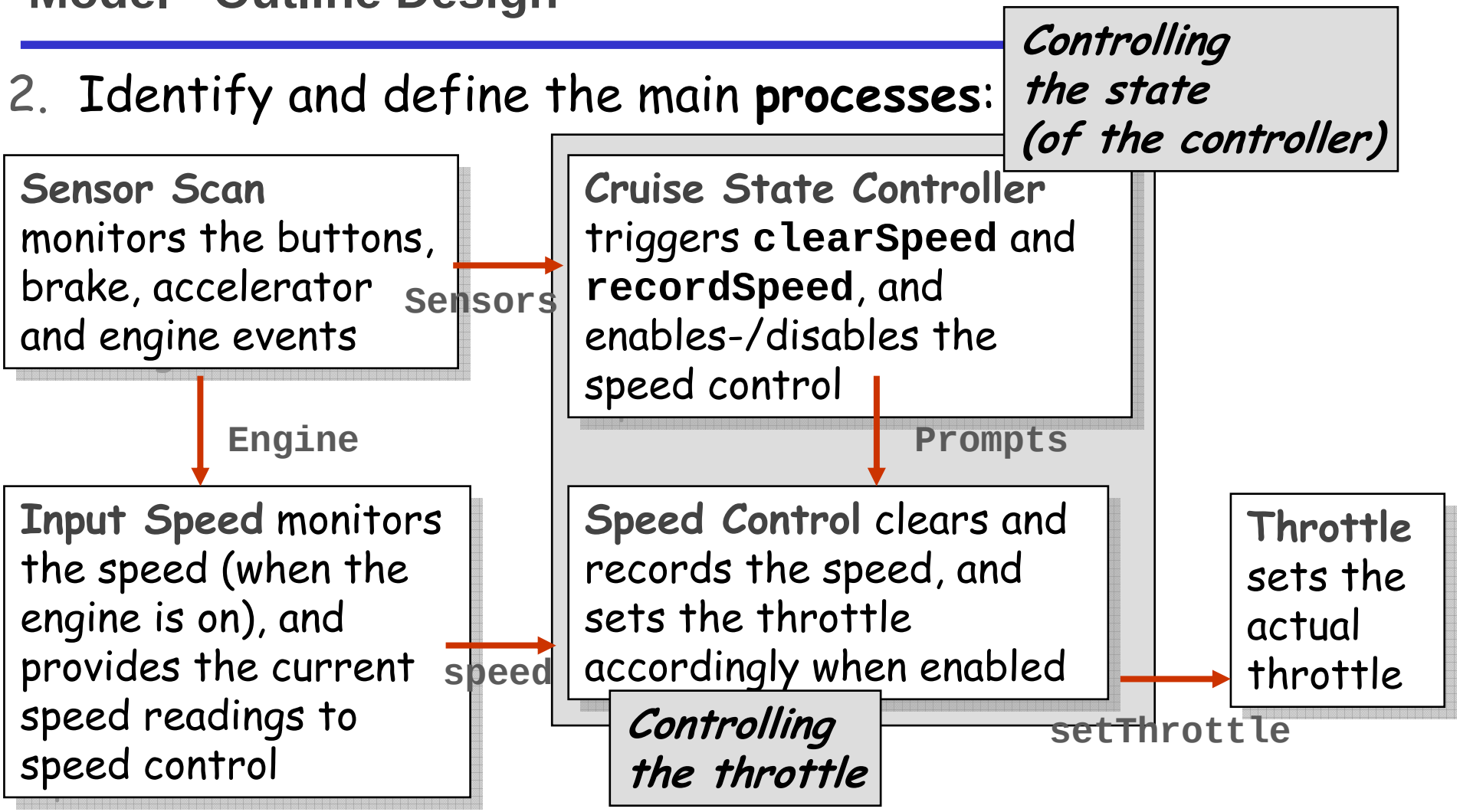
   `engineOn`, `engineOff`, `accelerator`, `brake`, `on`, `off`, `resume`, } **Sensors**

   `speed`, `setThrottle`, `zoom`
   `clearSpeed`, `recordSpeed`,
   `enableControl`, `disableControl`. } **Prompts**

```
set Sensors = {engineOn, engineOff, accelerator, brake,
               on, off, resume}
set Engine  = {engineOn, engineOff}
set Prompts = {enableControl, disableControl,
               clearSpeed, recordSpeed}
```

Model - Outline Design

2. Identify and define the main **processes**:




```
set Sensors = {engineOn, engineOff, accelerator, brake, on, off, resume}
set Engine  = {engineOn, engineOff}
set Prompts = {enableControl, disableControl, clearSpeed, recordSpeed}
```

Model – Design (Step 1+2+3)

1. Identify the main **events**, **actions**, and **interactions**:

   `on`, `off`, `resume`, `brake`, `accelerator` } **Sensors**

   `engineOn`, `engineOff`,

   `speed`, `setThrottle`

   `clearSpeed`, `recordSpeed`, } **Prompts**

   `enableControl`, `disableControl`

2. Identify and define the main **processes**:

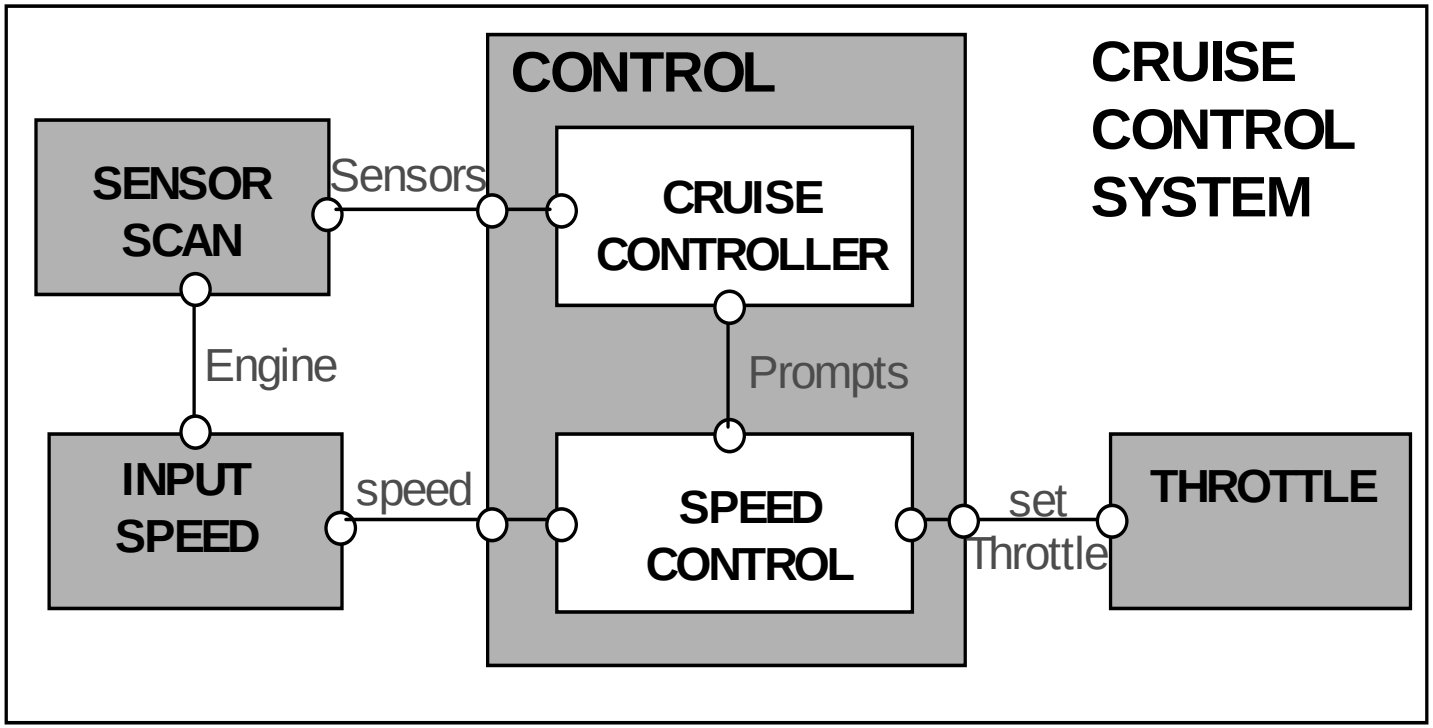
   `Sensor Scan`, `Input Speed`,
   `Cruise Controller`, `Speed Control` and
   `Throttle`

3. Identify and define the main **properties** of interest:

   safety - disabled when `off`, `brake` or `accelerator` pressed.

Model - Design (Step 4)

4. Structure the processes into an **architecture**:



The CONTROL system is structured as two processes:

- **CRUISECONTROLLER** (controlling the state); *and*
- **SPEEDCONTROL** (controlling the throttle)

Model Elaboration - Process Definitions

```
SENSORSCAN = ({Sensors} -> SENSORSCAN).

     // monitor speed when engine on
INPUTSPEED = (engineOn  -> CHECKSPEED),
CHECKSPEED = (speed     -> CHECKSPEED  // monitor speed
             |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).
```

Model Elaboration - Process Definitions

```
CRUISECONTROLLER = INACTIVE,

INACTIVE =(engineOn  -> clearSpeed          -> ACTIVE),

ACTIVE   =(engineOff                        -> INACTIVE
          |on-> recordSpeed-> enableControl-> CRUISING),

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

STANDBY  =(engineOff                        -> INACTIVE
          |resume           -> enableControl-> CRUISING
          |on-> recordSpeed-> enableControl-> CRUISING).
```

Model - CONTROL Sub-System

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

Animate will check *particular* traces:

- Is control enabled after the engine is switched on and the **'on'** button is pressed?
- Is control disabled when the brake is then pressed?
- Is control re-enabled when resume is then pressed?

Analysis (a.k.a. verification/model-checking) will exhaustively check *all possible* traces:

Safety: Is the control always disabled when **off**, **brake**, or **accelerator** is pressed?

Progress: Can every action eventually be selected?

Model - Safety Properties

Safety checks are compositional !

If there is no violation at a sub-system level, then there cannot be a violation when the sub-system is composed with other sub-systems.

This is because, if the **ERROR** state of a particular safety property is unreachable in the LTS of the sub-system, it remains unreachable in any subsequent parallel composition which includes the sub-system.

Thus: Safety properties should be composed with the *appropriate (sub-)system* to which the property refers.

In order for the property to be able to check the actions in its alphabet, the actions must not be hidden in the system.

Model - Safety Properties

Is the control always disabled when off/brake/acc pressed?

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

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


SAFETYACTION = (disableControl->CRUISESAFETY).
```

Composition with CONTROL processes:

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

Verify CRUISESAFETY? ☺

Model Analysis

We can now compose the whole system:

```
||CONTROL =
   (CRUISECONTROLLER||SPEEDCONTROL||CRUISESAFETY)
                    @ {Sensors,speed,setThrottle}.

||CRUISECONTROLSYSTEM =
     (CONTROL||SENSORSCAN||INPUTSPEED||THROTTLE).
```

We know there can be no safety violations due to composition

What about deadlocks?

```
No deadlocks/errors
```

What about **progress**…?

Model - Progress Properties

Progress checks are **not** compositional !

Even if there is no progress violation at a sub-system level, a progress violation may "appear" when the sub-system is composed with other sub-systems.

This is because an action in the sub-system may satisfy progress yet be unreachable when the sub-system is composed with other sub-systems which constrain system behaviour.

Thus: Progress checks should be conducted on the complete target system after satisfactory completion of the safety checks.

Model - Progress Properties

Progress check *(with hidden actions)*:

```
Progress violation for actions:
{engineOn, clearSpeed, engineOff, on, recordSpeed,
enableControl, off, disableControl, brake,
accelerator.....}
Path to terminal set of states:
     engineOn
     tau
     on
     tau
     tau
     engineOff
     engineOn
Actions in terminal set:
{speed, setThrottle, zoom}
```

Model - Progress Properties

Progress check *(without hidden actions):*

```
Progress violation for actions:
{engineOn, clearSpeed, engineOff, on, recordSpeed,
enableControl, off, disableControl, brake,
accelerator..........}
Path to terminal set of states:
     engineOn
     clearSpeed
     on
     recordSpeed
     enableControl
     engineOff
     engineOn
Actions in terminal set:
{speed, setThrottle, zoom}
```

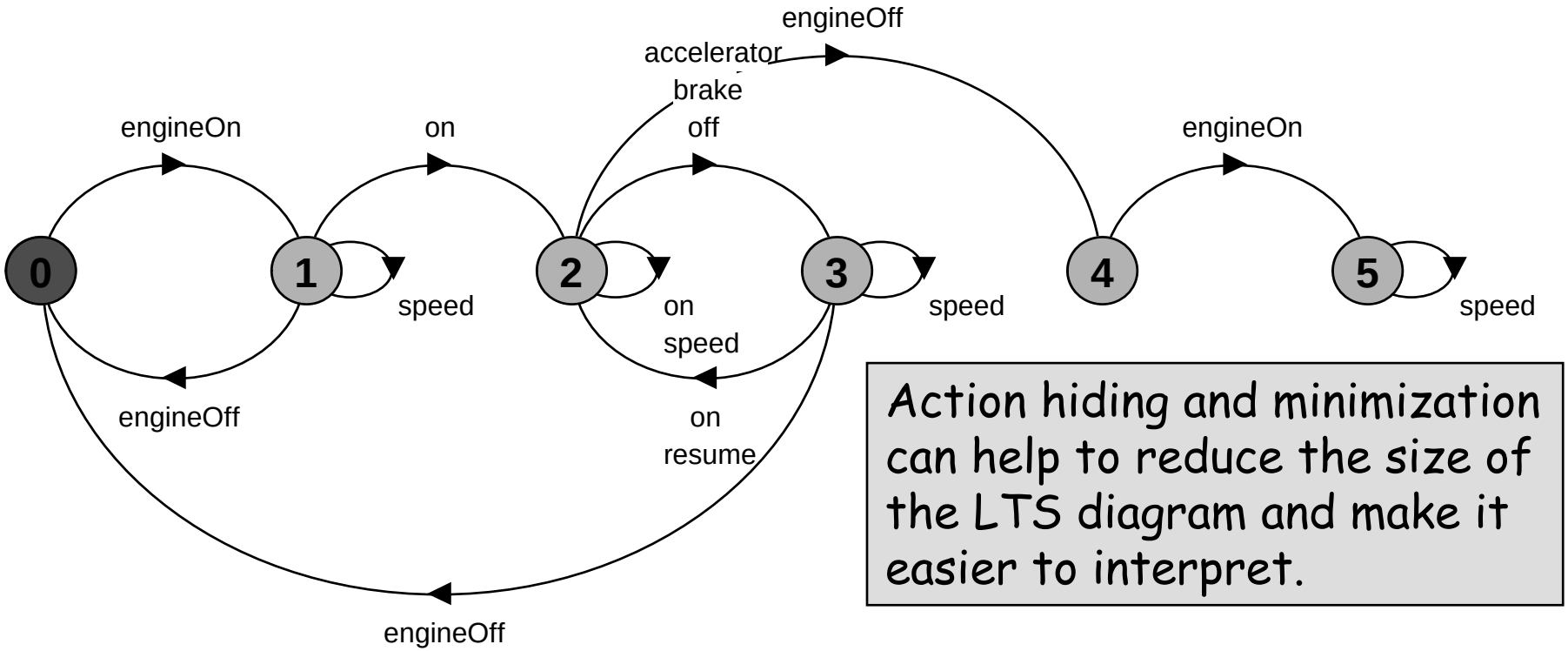
When the engine is switched off:

- CruiseController becomes inactive, whereas
- SpeedControl is not disabled!

Cruise Control Model - Minimized LTS

```
Progress violation trace: engineOn -> clearSpeed -> on ->
recordSpeed -> enableControl -> engineOff -> engineOn.
```

```
||CRUISEMINIMIZED = (CRUISECONTROLSYSTEM)
                    @ {Sensors,speed}.
```



Action hiding and minimization can help to reduce the size of the LTS diagram and make it easier to interpret.

Model – Revised Cruise Control System

Fix **CRUISECONTROLLER** so that it disables the **SPEEDCONTROLLER** when the engine is switched off:

```
      // enable speed control when cruising,
      // disable when off, brake, or accelerator pressed
      //         or when the engine is turned off!!!
CRUISING =(engineOff    -> disableControl   -> INACTIVE
          |{off,brake,accelerator}
                              ->disableControl-> STANDBY
          |on-> recordSpeed-> enableControl-> CRUISING),
```

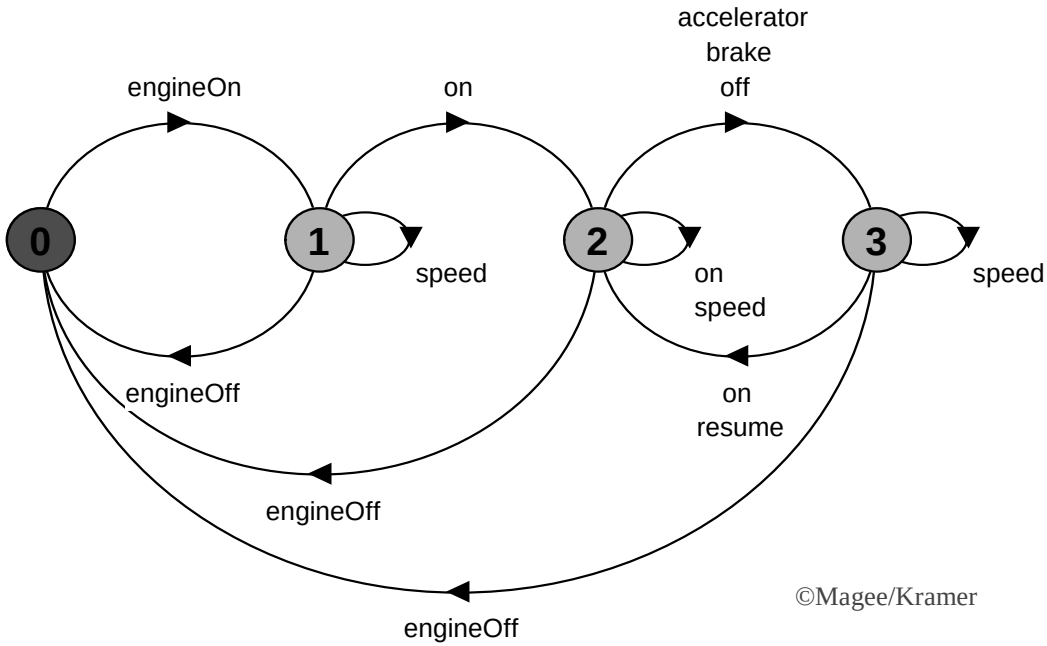
OK now?

Model – Revised Cruise Control System (Properties)

```
property CRUISESAFETYv2 =
   ({off,accelerator,...,engineOff} -> CRUISESAFETYv2
                |{on,resume}          -> SAFETYCHECK),
SAFETYCHECK  = ({on,resume}          -> SAFETYCHECK
                |{off,...,engineOff} -> SAFETYACTION
                |disableControl      -> CRUISESAFETYv2),
SAFETYACTION = (disableControl       -> CRUISESAFETYv2).
```

No deadlocks/errors

No progress violations detected



Model - System Sensitivities *(under Adverse Conditions)*

```
||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}
```

Indicates that the system may be sensitive to the priority of the action "**speed**".

Model Interpretation

Models can be used to indicate system sensitivities!

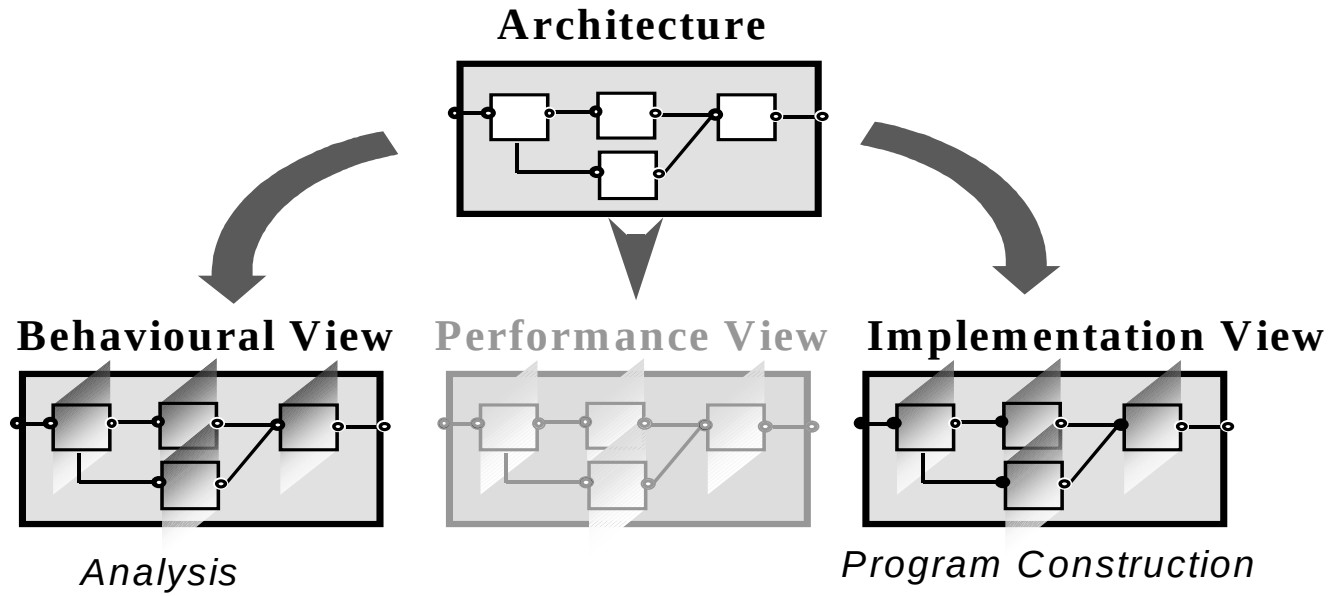
If it is possible that erroneous situations detected in the model may occur in the implemented system, then the model should be revised to find a design which ensures that those violations are avoided.

However, if it is considered that the real system will not exhibit this behaviour, then no further model revisions are necessary.

Model **interpretation** and correspondence to the implementation are important in determining the relevance and adequacy of the model design and its analysis.

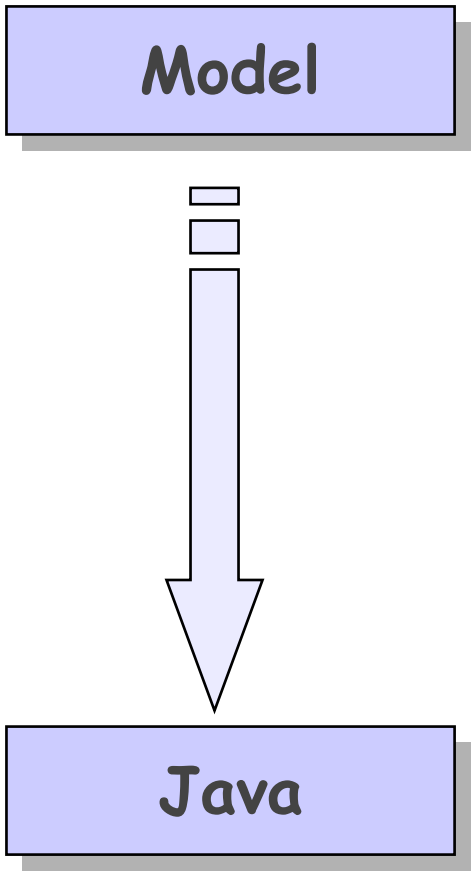
The Central Role of Design Architecture

Design architecture describes the gross organization and global structure of the system in terms of its constituent components. See UML.



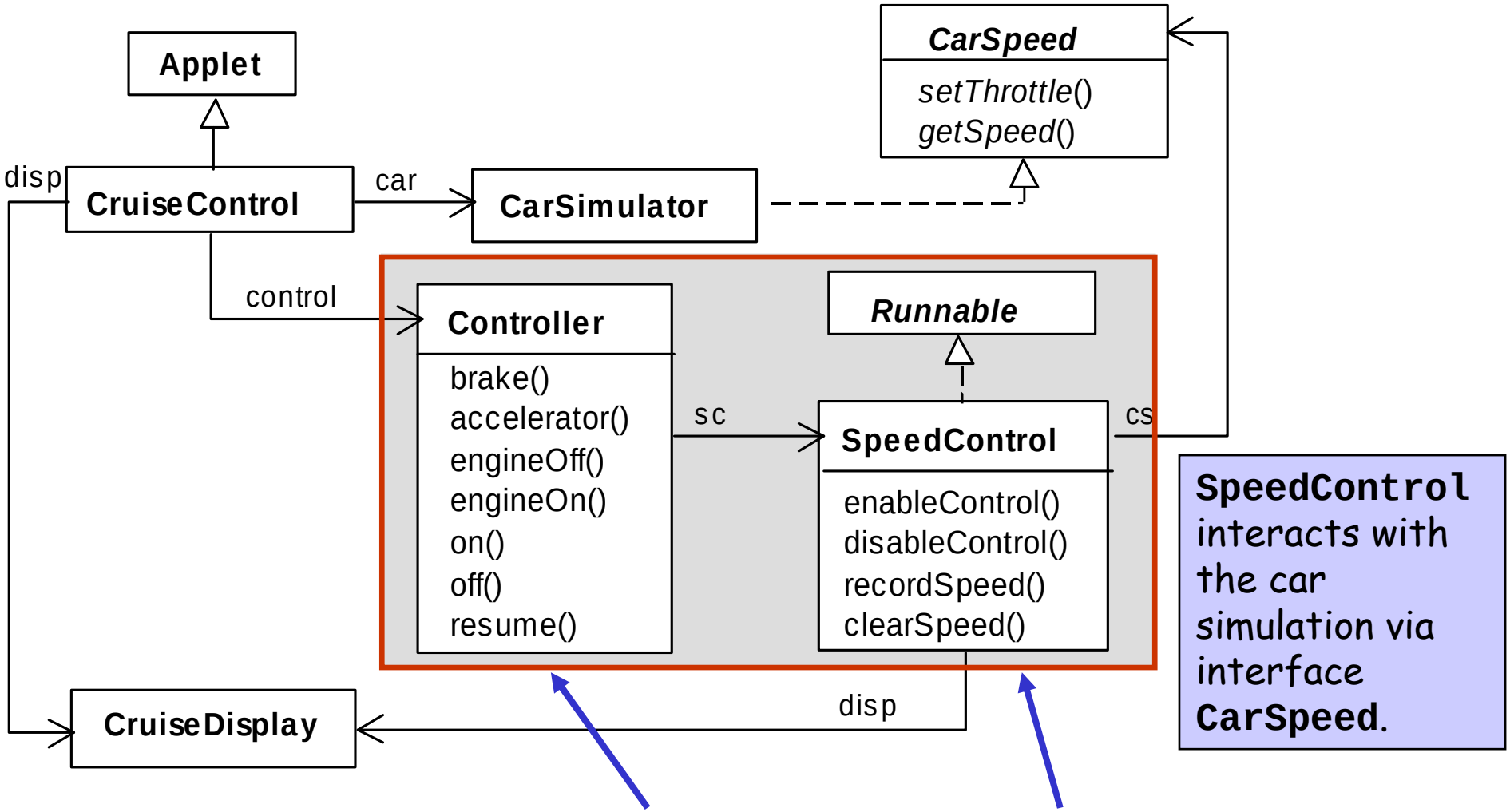
We consider that the implementation should be considered as an **elaborated view** of the basic design architecture.

8.2 From Models to Implementations



- identify the main *active* entities
  - to be implemented as threads
- identify the main (shared) *passive* entities
  - to be implemented as monitors
- (identify the interactive display environment
  - to be implemented as associated classes)
- structure the classes as a (UML) class diagram
  - to be implemented

Cruise Control System - Class Diagram



Cruise Control System - Class `Controller`

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

    Controller(CarSpeed cs, CruiseDisplay disp) {
        sc = new SpeedControl(cs, disp);
    }
    synchronized void brake() {
        if (state == CRUISING)
            { sc.disableControl(); state = STANDBY; }
    }
    synchronized void accelerator() {
        if (state == CRUISING)
            { sc.disableControl(); state = STANDBY; }
    }
    synchronized void engineOff() {
        if (state != INACTIVE) {
            if (state == CRUISING) sc.disableControl();
            state = INACTIVE;
        }
    }
```

Controller is a *passive* entity (it reacts to events) and thus implemented as a **monitor**

Cruise Control System - Class `Controller`

```
    ...
    synchronized void engineOn() {
        if (state == INACTIVE) {
            sc.clearSpeed(); state = ACTIVE;
        }
    }
    synchronized void on() {
        if (state != INACTIVE) {
            sc.recordSpeed();
            sc.enableControl(); state = CRUISING;
        }
    }
    synchronized void off() {
        if (state == CRUISING) {
            sc.disableControl(); state = STANDBY;
        }
    }
    synchronized void resume() {
        if (state == STANDBY) {
            sc.enableControl(); state = CRUISING;
        }
    }
}
```

Direct translation from the model.

Cruise Control System - Class SpeedControl

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

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

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

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

SpeedControl is an *active* entity; when enabled, a **new thread** is created (which periodically obtains car speed and sets the throttle).

Cruise Control System - Class `SpeedControl`

```
...

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

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

...
```

Cruise Control System - Class `SpeedControl`

```
    public void run() { // the speed controller thread
        try {
            while (state == ENABLED) {
                synchronized (this) {
                    // calculate and set new throttle speed
                    double throttle = ...cs.getSpeed()...;
                    cs.setThrottle(throttle);
                }
                Thread.sleep(500);
            }
        } catch (InterruptedException _) {}
        sc = null; // throw away SpeedController thread
    }
}
```

`SpeedControl` is an example of a class that combines both

- synchronized methods (to update local vars); *and*
- a thread.

Summary: Model-Based Design

Concepts:

design process:

- from requirements to **models**
- from **models** to implementations

Models:

check properties of interest:

- safety on the appropriate (sub)system
- progress on the overall system

Practice:

model "interpretation":

- to infer actual system behavior

active threads and *passive monitors*

Aim: rigorous design process.