Concurrent Programming
19530-V (WS01)

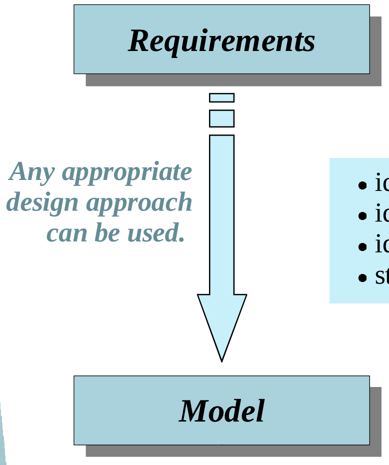
Lecture 11:
Model-based Design

Dr. Richard S. Hall
`rickhall@inf.fu-berlin.de`

Model-based Design

- Concept
  - *requirements → models → implementation*
- Models
  - Allow us to check properties of interest before implementation
    - *Safety* for the appropriate (sub)system
    - *Progress* on the overall system
- Practice
  - Interpret model behavior to infer actual system behavior (e.g., which will be composed of threads and monitors).

From Requirements to Models



Requirements

- goals of the system
- scenarios (Use Case models)
- properties of interest

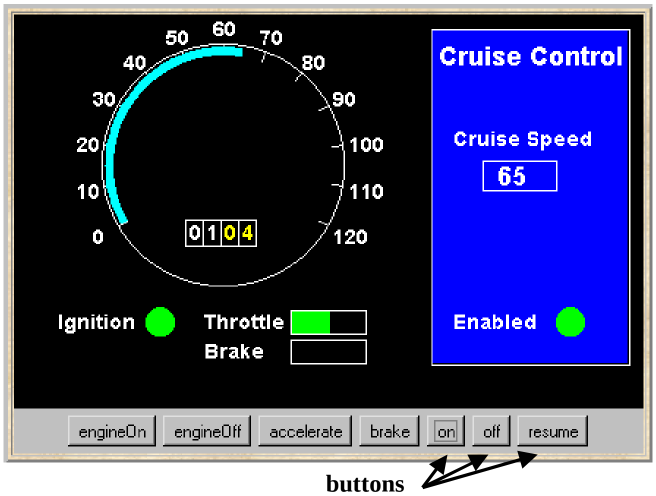
Any appropriate design approach can be used.

- identify the main events, actions, and interactions
- identify and define the main processes
- identify and define the properties of interest
- structure the processes into an ***architecture***

Model

- check traces of interest
- check properties of interest

Cruise Control Requirements

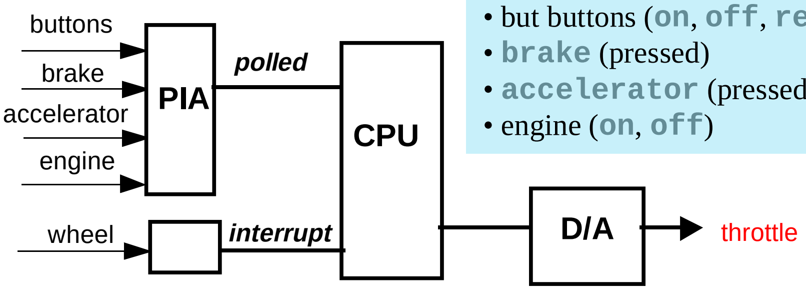


When the car ignition is switched on and the **`on`** button is pressed, the current speed is recorded and the system is enabled: ***it maintains the speed of the car at the recorded setting***.

Pressing the **`brake`**, **`accelerator`**, or **`off`** button disables the system. Pressing **`resume`** or **`on`** re-enables the system.

Cruise Control System Hardware

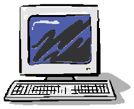
Parallel Interface Adapter (PIA) is polled every 100msec. It records the actions of the sensors:



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

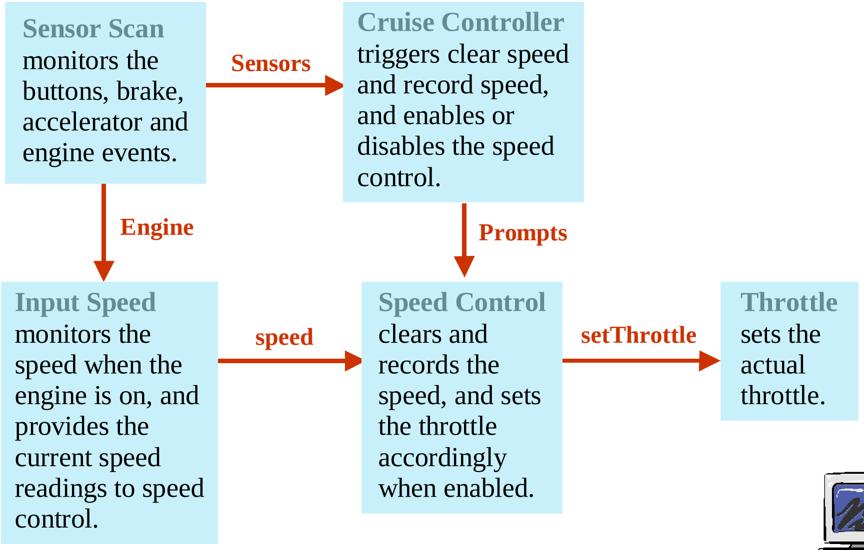
Wheel revolution sensor generates interrupts to enable the car **speed** to be calculated.

Output: The cruise control system controls the car speed by setting the **throttle** via the digital-to-analog converter.



Model Design Outline

Outline processes and interactions:



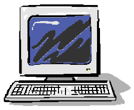
Sensor Scan monitors the buttons, brake, accelerator and engine events.

Cruise Controller triggers clear speed and record speed, and enables or disables the speed control.

Input Speed monitors the speed when the engine is on, and provides the current speed readings to speed control.

Speed Control clears and records the speed, and sets the throttle accordingly when enabled.

Throttle sets the actual throttle.



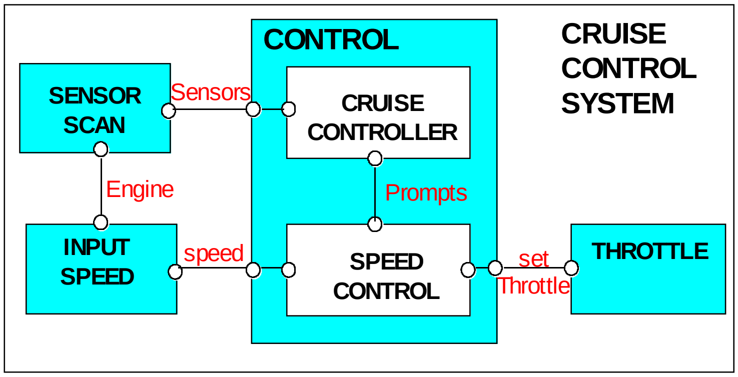
Model Design Overview

- Main processes
  - **`SENSORSCAN`**, **`INPUTSPEED`**, **`CRUISECONTROLLER`**, **`SPEEDCONTROL`**, and **`THROTTLE`**
- Main events, actions, and interactions
  - **`engineOn`**, **`engineOff`**, **`on`**, **`off`**, **`resume`**, **`brake`**, and **`accelerator`** (monitored by sensors)
  - **`clearSpeed`**, **`recordSpeed`**, **`enableControl`**, **`disableControl`** (interact with speed control)
  - **`speed`** and **`setThrottle`** (input/output of speed control)
- Main properties
  - *Safety* – system is disabled when **`off`**, **`brake`**, or **`accelerator`** is pressed

Cruise Control Structure

The **`CONTROL`** system is structured as two processes.

The main actions and interactions are as shown.



```
// Simplify keeping track of sensor events
set Sensors = {engineOn,engineOff,on,off,
               resume,brake,accelerator}
```

Cruise Control Model

```
// "Listen" for all sensor events
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).
```

Cruise Control Model

```
// 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).
```

Cruise Control Model

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

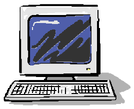
Animate to check particular traces:

- 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?

However, we need to analyze for:

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

Progress: Can every action eventually be selected?



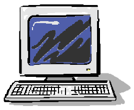
Model Safety Properties

Safety checks are *compositional*. If there is no violation within a particular subsystem, then there cannot be a violation when the subsystem is composed with other subsystems.

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

Hence...

Safety properties should be composed with the appropriate system or subsystem to which the property refers. In order that the property can check the actions in its alphabet, these actions must not be hidden in the system.



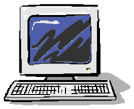
Cruise Control Safety Property

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

LTS?

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

Is ***CRUISESAFETY*** *violated?*



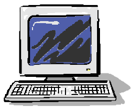
Cruise Control Safety Property

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

// Complete cruise control system
||CRUISECONTROLSYSTEM =
   (CONTROL || SENSORSCAN || INPUTSPEED || THROTTLE).
```

Deadlock?
Safety?

```
No deadlocks/errors
```

Progress?



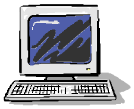
Model Progress Properties

Progress checks are ***not compositional***. Even if there is no violation at a subsystem level, there may still be a violation when the subsystem is composed with other subsystems.

This is because an action in the subsystem may satisfy progress yet be unreachable when the subsystem is composed with other subsystems which constrain its behavior.

Hence...

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

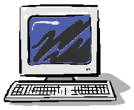


Cruise Control Progress Property

Since the cruise control system should always work, we would expect no action to starve, thus we can use the *default progress property*. *When a system specifies no progress properties, then LTSA uses the default progress property; it is equivalent to defining a progress property for each action.*

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

Hidden actions appear as **`tau`**



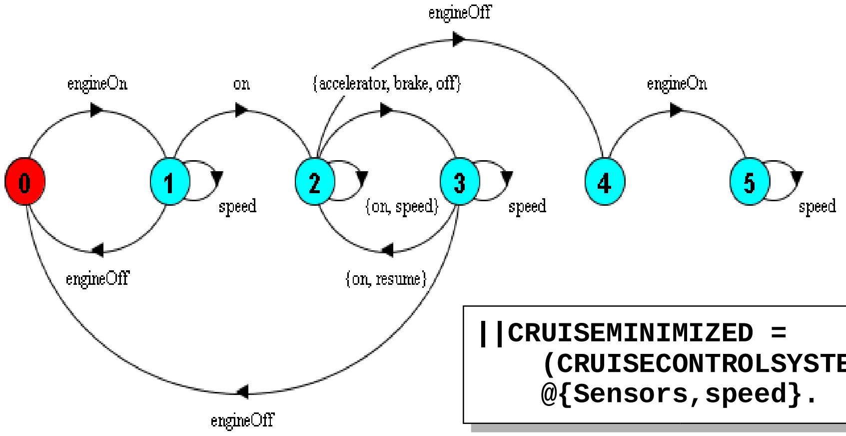
Cruise Control Progress Property

Removing the hidden actions...

```
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
Actions in terminal set:
     {setThrottle, speed, zoom}
```

Why is this happening?

Minimized Cruise Control LTS



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

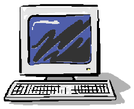
We can easily see here that in state 2, the cruise control is not disabled when the engine is turned off (via `engineOff`).

Revised Cruise Control System

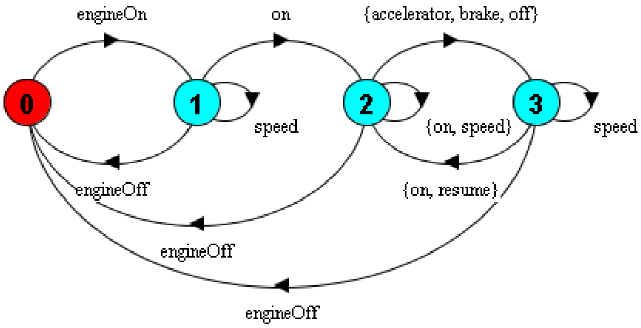
```
property IMPROVEDSAFETY =
    ({off,accelerator,brake,disableControl,engineOff}
           ->IMPROVEDSAFETY
    |{on,resume}->SAFETYCHECK),
SAFETYCHECK =
    ({on,resume}-> SAFETYCHECK
    |{off,accelerator,brake,engineOff}->SAFETYACTION
    |disableControl->IMPROVEDSAFETY),
SAFETYACTION =(disableControl->IMPROVEDSAFETY).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

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

Okay now?



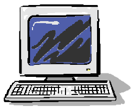
Revised Cruise Control System



No deadlocks/errors

No progress violations detected.

What about under *adverse* conditions? Check for system sensitivities.

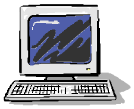


Cruise Control Sensitivities

```
||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}
```

The system may be sensitive to the priority of the action **`speed`**.



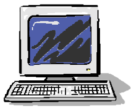
Model Interpretation

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.

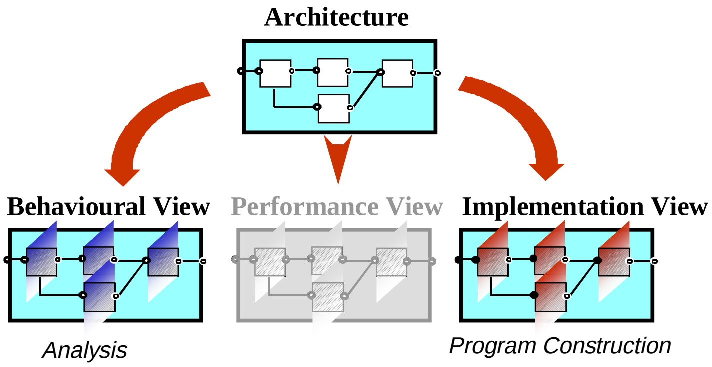
However, if the real system will ***not*** exhibit this behavior, then no further model revisions are necessary.

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



Central Role of a Design Architecture

Design architecture describes the overall organization and structure of the system in terms of its components; we have been using FSP and structure diagrams for our design architecture.



Architecture

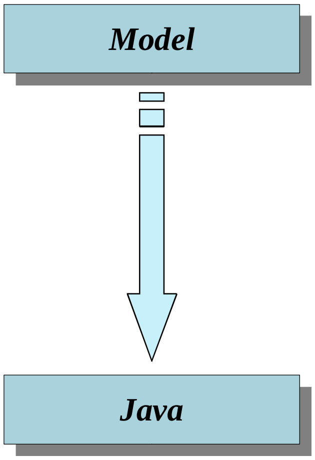
Behavioural View Performance View **Implementation View**

Analysis

Program Construction

We consider that the models for analysis and the implementation should be considered as elaborated views of this basic design structure.

Models to Implementations

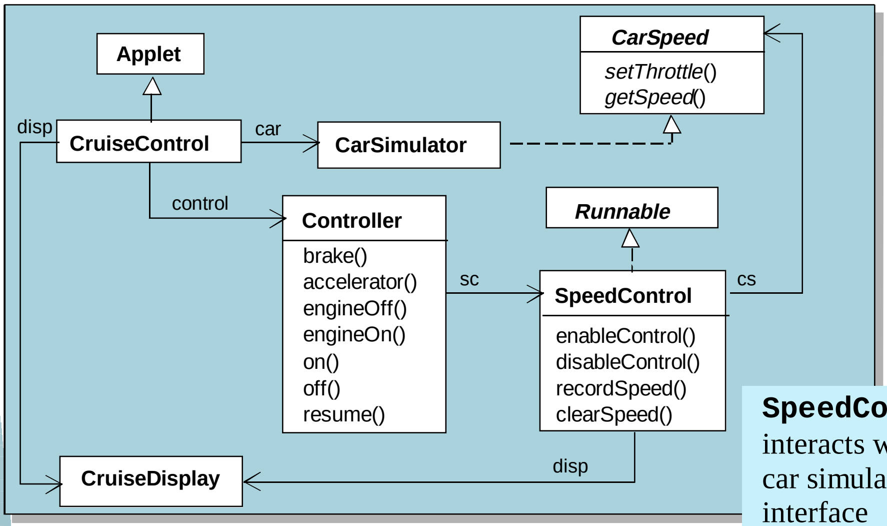


Model

Java

- Identify the main active entities
  - Implemented as threads
- Identify the main (shared) passive entities
  - Implemented as monitors
- Identify the interactive display environment
  - Implemented as associated classes
- Structure the classes as a class diagram

Cruise Control Class Diagram



`SpeedControl` interacts with the car simulation via interface **`CarSpeed`**.

Controller Class

```
class Controller {
  final static int INACTIVE = 0;
  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; }
  }

// continued on next slide...
```

`Controller` is a passive entity - it reacts to events; hence we implement it as a ***monitor***

Controller Class

```
// continued from previous slide...
  synchronized void engineOff() {
    if (controlState != INACTIVE) {
      if (controlState == CRUISING) sc.disableControl();
      controlState = INACTIVE;
    }
  }
  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; }
  }
}
```

This is a direct translation from the model.

SpeedControl Class

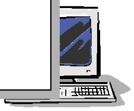
```
class SpeedControl implements Runnable {
  final static int DISABLED = 0; // Speed control states
  final static int ENABLED  = 1;
  private int state = DISABLED; // Initial state
  private int setSpeed = 0;     // Target speed
  private Thread speedController;
  private CarSpeed cs;     // Interface to car
  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); }
  }

// continued on next slide...
```

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

SpeedControl Class

```
// continued from previous slide...
  synchronized void enableControl() {
    if (state == DISABLED)
    { disp.enable(); speedController = new Thread(this);
      speedController.start(); state = ENABLED; }
  }
  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); // feed back control
        }
      }
    } catch (InterruptedException e) { }
    speedController=null;
  }
}
```

`SpeedControl` is an example of a class that combines both synchronized access methods (to update local variables) and a thread.



Summary

- Concepts
  - *Design process*
    - → from requirements to models to implementations
  - *Design architecture*
- Models
  - Check properties of interest
    - *Safety*: compose safety properties at appropriate (sub)system
    - *Progress*: apply progress check on the final system model
- Practice
  - *Model interpretation* to infer actual system behavior
  - *Implement* using threads and monitors