Chapter 8

Model-Based Design

8.1 from requirements to models

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

Any appropriate design approach can be used.

**Model**
- check traces of interest
- check properties of interest

A Cruise Control System - 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 converter.

---

**model - outline design**

- **Outline processes and interactions.**

**Sensors**
- 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.

**Prompts**
- 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**

- **Main events, actions and interactions.**

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

- **Identify main processes.**
  Sensor Scan, Input Speed,
  Cruise Controller, Speed Control and Throttle

- **Identify main properties.**
  safety - disabled when off, brake or accelerator pressed.

- **Define and structure each process.**

**model - structure, actions and interactions**

The CONTROL system is structured as two processes.

The main actions and interactions are as shown.

```plaintext
set Sensors = {engineOn, engineOff, on, off, resume, brake, accelerator}
set Engine = {engineOn, engineOff}
set Prompts = {clearSpeed, recordSpeed, enableControl, disableControl}
```
model elaboration - process definitions

**SENSORSCAN** = {{Sensors} \rightarrow SENSORSCAN}.

// monitor speed when engine on
**INPUTSPEED** = {engineOn \rightarrow CHECKSPEED},
**CHECKSPEED** = {speed \rightarrow CHECKSPEED | engineOff \rightarrow INPUTSPEED}.

// zoom when throttle set
**THROTTLE** = {{setThrottle} \rightarrow {{zoom} \rightarrow THROTTLE}}.

// perform speed control when enabled
**SPEEDCONTROL** = DISABLED,
DISABLED = {{speed,clearSpeed,recordSpeed} \rightarrow DISABLED | enableControl \rightarrow ENABLED},
ENABLED = {speed \rightarrow setThrottle \rightarrow ENABLED | enableControl \rightarrow DISABLED | disableControl \rightarrow DISABLED}.

**set DisableActions** = {off, brake, accelerator}

// enable speed control when cruising, disable when a disable action occurs
**CRUISECONTROLLER** = INACTIVE,
INACTIVE = (engineOn \rightarrow clearSpeed \rightarrow ACTIVE | DisableActions \rightarrow INACTIVE),
ACTIVE = (engineOff \rightarrow INACTIVE | on\rightarrow recordSpeed\rightarrow enableControl\rightarrow CRUISING | DisableActions \rightarrow ACTIVE),
CRUISING = (engineOff \rightarrow INACTIVE | DisableActions\rightarrow disableControl\rightarrow STANDBY | on \rightarrow recordSpeed\rightarrow enableControl \rightarrow CRUISING | DisableActions \rightarrow STANDBY).

- 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?

Animate to check particular traces:

However, we need analysis to check exhaustively:

- **Safety**: Is the control disabled when off, brake or accelerator is pressed?
- **Progress**: Can every action eventually be selected?

model - **CONTROL** subsystem

||CONTROL = (CRUISECONTROLLER || SPEEDCONTROL).

Safety checks are **compositional**. If there is no violation at a subsystem level, 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.
**model - Safety properties**

<table>
<thead>
<tr>
<th>property CRUISESAFETY =</th>
</tr>
</thead>
<tbody>
<tr>
<td>({DisableActions, disableControl} -&gt; CRUISESAFETY</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>) ,</td>
</tr>
<tr>
<td>SAFETYCHECK =</td>
</tr>
<tr>
<td>({on, resume} -&gt; SAFETYCHECK</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>) ,</td>
</tr>
<tr>
<td>SAFETYACTION = (disableControl -&gt; CRUISESAFETY).</td>
</tr>
</tbody>
</table>

---

**model - Safety properties**

Safety analysis using LTSA produces the following **violation**:

Trace to property violation in CRUISESAFETY:

- engineOn
- clearSpeed
- on
- recordSpeed
- enableControl
- engineOff
- off
- off

---

**model - Safety properties**

<table>
<thead>
<tr>
<th>Strange circumstances!</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the system is enabled by switching the engine on and pressing the on button, and then the engine is switched off, it appears that the control system is not disabled.</td>
</tr>
</tbody>
</table>

---

**model - Safety properties**

What if the engine is switched on again?

We can investigate further using animation …

- engineOn
- clearSpeed
- on
- recordSpeed
- enableControl
- engineOff
- engineOn
- speed
- setThrottle
- speed
- setThrottle

The car will accelerate and zoom off when the engine is switched on again!

---

**model - Safety properties**

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

---

**Model LTS for CONTROLMINIMIZED**

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

... using progress?
model - Progress properties

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
Cycle in terminal set:
- speed
- setThrottle
Actions in terminal set:
- {setThrottle, speed}

Check the model for progress properties with no safety property and no hidden actions.

model - revised cruise controller

Modify CRUISECONTROLLER so that control is disabled when the engine is switched off:

```
CRUISING = (engineOff -> disableControl -> INACTIVE
|DisableActions -> disableControl -> STANDBY
|on->recordSpeed->enableControl->CRUISING } ,
```

Modify the safety property:

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

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

SAFEYACTION = (disableControl -> IMPROVEDSAFETY).
```

OK now?

revised CONTROLMINIMIZED

![Diagram of revised CONTROLMINIMIZED]

We can now proceed to compose the whole system:

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

Deadlock?  Safety?  Progress?

No deadlocks/errors
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 behaviour. Hence...

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

Progress? No progress violations detected.

model - system sensitivities

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

||SPEEDHIGH = CRUISECONTROLSYSTEM << {speed}.

Progress violation for actions:
{accelerator, brake, engineOff, engineOn, off, on, resume, setThrottle, zoom}
Trace to terminal set of states:
engineOn
Cycle in terminal set:
speed
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 it is considered that 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.

The central role of design architecture

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

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

Cruise control system - class diagram

Cruise control system - class Controller

```java
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;
        }
    }

    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;
        }
    }
}
```

Controller is a passive entity - it reacts to events. Hence we implement it as a monitor.

This is a 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;
    private int state = DISABLED; // initial state
    private int setSpeed = 0; // target speed
    private Thread speedController;
    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;
        }
    }
    synchronized void disableControl() {
        if (state == ENABLED) {
            disp.disable(); state = DISABLED;
        }
    }
    public void run() {
        // the speed controller thread
        try {
            while (state == ENABLED) {
                double error = (float)(setSpeed - cs.getSpeed())/6.0;
                double steady = (double)setSpeed/12.0;
                cs.setThrottle(steady + error); // simplified feed back control
                wait(500);
            }
        } catch (InterruptedException e) {} 
        speedController = null;
    }
}

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

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 target system model

◆ Practice
  • model interpretation - to infer actual system behavior
  • threads and monitors

Aim: rigorous design process.

Course Outline

2. Processes and Threads
3. Concurrent Execution
4. Shared Objects & Interference
5. Monitors & Condition Synchronization
6. Deadlock
7. Safety and Liveness Properties
8. Model-based Design

The main basic

Concepts
Models
Practice

Advanced topics …

9. Dynamic systems
10. Message Passing
11. Concurrent Software Architectures
12. Timed Systems
13. Program Verification
14. Logical Properties