CSCI 5828: Foundations of Software Engineering

Lecture 19-20: Model Based Design

Slides created by Magee and Kramer for the Concurrency textbook
Chapter 8

Model-Based Design
Design

Concepts: design process: requirements to 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
threads and monitors

Aim: rigorous design process.
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**
- 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**

- check traces of interest
- check properties of interest
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.

**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.

---

Concurrency: model-based design
model -design

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

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

}  

Sensors

}  

Prompts

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

The main actions and interactions are as shown.

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

set Engine = {engineOn, engineOff}

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

Concurrent: model-based design
model elaboration - process definitions

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).
model elaboration - process definitions

\begin{verbatim}
set DisableActions = \{ off, brake, accelerator \}
// enable speed control when cruising, disable when a disable action occurs
CRUISECONTROLLER = INACTIVE,
INACTIVE  = ( engineOn -> clearSpeed -> ACTIVE
  | DisableActions -> INACTIVE ),
ACTIVE     = ( engineOff -> INACTIVE
  | on->recordSpeed->enableControl->CRUISING
  | DisableActions -> ACTIVE ),
CRUISING   = ( engineOff -> INACTIVE
  | DisableActions->disableControl->STANDBY
  | on->recordSpeed->enableControl->CRUISING ),
STANDBY    = ( engineOff -> INACTIVE
  | resume -> enableControl -> CRUISING
  | on->recordSpeed->enableControl->CRUISING
  | DisableActions -> STANDBY )
\end{verbatim}
model - CONTROL subsystem

\[
| \) = (\text{CRUISECONTROLLER} | \text{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 analysis to check exhaustively:

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

property CRUISESAFETY =
  ({DisableActions,disableControl} -> CRUISESAFETY |
  {on,resume} -> SAFETYCHECK
  ),
SAFETYCHECK =
  ({on,resume} -> SAFETYCHECK |
  DisableActions -> SAFETYACTION |
  disableControl -> CRUISESAFETY
  ),
SAFETYACTION = (disableControl->CRUISESAFETY).

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

Is CRUISESAFETY violated?
Safety analysis using LTSA produces the following violation:

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

Strange circumstances!
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.
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!

... 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}.

Concurrency: model-based design

... using progress?
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}

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

OK now?
revised CONTROLMINIMIZED

No deadlocks/errors
We can now proceed to compose the whole system:

\[
| | \text{CONTROL} =
\]
\[
(CRUISECONTROLLER \mid \text{SPEEDCONTROL} \mid \text{CRUISESAFETY}) @ \{\text{Sensors, speed, setThrottle}\}.
\]

\[
| | \text{CRUISECONTROLSYSTEM} =
\]
\[
(CONTROL \mid \text{SENSORSCAN} \mid \text{INPUTSPEED} \mid \text{THROTTLE}).
\]

**Deadlock?**

**Safety?**

**No deadlocks/errors**

**Progress?**
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.

**Progress?**

No progress violations detected.
model - system sensitivities

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

\[ \| \text{SPEEDHIGH} = \text{CRUISECONTROLSYSTEM} \ll \{ \text{speed} \} \]

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 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

- **Applet**
- **CarSpeed**
  - `setThrottle()`
  - `getSpeed()`
- **Controller**
  - `brake()`
  - `accelerator()`
  - `engineOff()`
  - `engineOn()`
  - `on()`
  - `off()`
  - `resume()`
- **Runnable**
- **SpeedControl**
  - `enableControl()`
  - `disableControl()`
  - `recordSpeed()`
  - `clearSpeed()`

**Interactions**
- **CruiseControl** interacts with **CarSimulator** via **interface CarSpeed**.
- **SpeedControl** interacts with the car simulation via interface **CarSpeed**.

Concurrency: model-based design

Concurrent elements:
- **CRUISECONTROLLER**
- **SPEEDCONTROL**
cruise control system - class **Controller**

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;
        }
    }
}
cruise control system - class **Controller**

```java
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.
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 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;
        }
    }
}
cruise control system - class **SpeedControl**

```java
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 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 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

Concurrency: model-based design

12. Timed Systems
13. Program Verification
14. Logical Properties