Concurrency

State Models and Java Programs

Jeff Magee and Jeff Kramer

Concurrency: introduction

©Magee/Kramer

What is a Concurrent Program?

A sequential program has a single thread of control.

A concurrent program has multiple threads of control allowing it to perform multiple computations in parallel and to control multiple external activities which occur at the same time.

Why Concurrent Programming?

◆ Performance gain from multiprocessing hardware
  ● parallelism.
◆ Increased application throughput
  ● an I/O call need only block one thread.
◆ Increased application responsiveness
  ● high priority thread for user requests.
◆ More appropriate structure
  ● for programs which interact with the environment, control multiple activities and handle multiple events.

models

A model is a simplified representation of the real world. Engineers use models to gain confidence in the adequacy and validity of a proposed design.

◆ focus on an aspect of interest - concurrency
◆ model animation to visualise a behaviour
◆ mechanical verification of properties (safety & progress)

Models are described using state machines, known as Labelled Transition Systems (LTS). These are described textually as finite state processes (FSP) and displayed and analysed by the LTSA analysis tool.

a Cruise Control System

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 re-enables the system.

Is the system safe?

Would testing be sufficient to discover all errors?

Do I need to know about concurrent programming?

Concurrency is widespread but error prone.

◆ Therac - 25 computerised radiation therapy machine
  Concurrent programming errors contributed to accidents causing deaths and serious injuries.
◆ Mars Rover
  Problems with interaction between concurrent tasks caused periodic software resets reducing availability for exploration.

A model is a simplified representation of the real world. Engineers use models to gain confidence in the adequacy and validity of a proposed design.

◆ focus on an aspect of interest - concurrency
◆ model animation to visualise a behaviour
◆ mechanical verification of properties (safety & progress)

Models are described using state machines, known as Labelled Transition Systems (LTS). These are described textually as finite state processes (FSP) and displayed and analysed by the LTSA analysis tool.
modeling the Cruise Control System

Later chapters will explain how to construct models such as this so as to perform animation and verification.

programming practice in Java

Java is

- widely available, generally accepted and portable
- provides sound set of concurrency features

Hence Java is used for all the illustrative examples, the demonstrations and the exercises. Later chapters will explain how to construct Java programs such as the Cruise Control System.

“Toy” problems are also used as they crystallize particular aspects of concurrent programming problems!

course objective

This course is intended to provide a sound understanding of the concepts, models and practice involved in designing concurrent software.

The emphasis on principles and concepts provides a thorough understanding of both the problems and the solution techniques. Modeling provides insight into concurrent behavior and aids reasoning about particular designs. Concurrent programming in Java provides the programming practice and experience.

Learning outcomes...

After completing this course, you will know

- how to model, analyze, and program concurrent object-oriented systems.
- the most important concepts and techniques for concurrent programming.
- what are the problems which arise in concurrent programming.
- what techniques you can use to solve these problems.

Book

Concurrency: State Models & Java Programs, 2nd Edition
Jeff Magee & Jeff Kramer
WILEY

Course Outline

- Processes and Threads
- Concurrent Execution
- Shared Objects & Interference
- Monitors & Condition Synchronization
- Deadlock
- Safety and Liveness Properties
- Model-based Design

Concepts
Models
Practice

- Dynamic systems
- Concurrent Software Architectures
- Message Passing
- Timed Systems
Summary

\begin{itemize}
\item Concepts
  \begin{itemize}
  \item we adopt a model-based approach for the design and construction of concurrent programs
  \end{itemize}
\item Models
  \begin{itemize}
  \item we use finite state models to represent concurrent behavior.
  \end{itemize}
\item Practice
  \begin{itemize}
  \item we use Java for constructing concurrent programs.
  \end{itemize}
\end{itemize}

Examples are used to illustrate the concepts, models and demonstration programs.
concurrent processes
We structure complex systems as sets of simpler activities, each represented as a sequential process. Processes can overlap or be concurrent, so as to reflect the concurrency inherent in the physical world, or to offload time-consuming tasks, or to manage communications or other devices.

Designing concurrent software can be complex and error prone. A rigorous engineering approach is essential.

Concept of a process as a sequence of actions.
Model processes as finite state machines.
Program processes as threads in Java.

Processes & Threads

2.1 Modelling Processes
Models are described using state machines, known as Labelled Transition Systems LTS. These are described textually as finite state processes (FSP) and displayed and analysed by the LTSA analysis tool.

- LTS - graphical form
- FSP - algebraic form

Can finite state models produce infinite traces?

FSP - action prefix
If \( x \) is an action and \( P \) a process then \((x \rightarrow P)\) describes a process that initially engages in the action \( x \) and then behaves exactly as described by \( P \).

\[
\text{ONESHOT} = \text{(once} \rightarrow \text{STOP)}.
\]

ONESHOT state machine (terminating process)

ONESHOT = (once \( \rightarrow \) STOP).

ONESHOT state machine (terminating process)

Convention: actions begin with lowercase letters
PROCESSES begin with uppercase letters
Repetitive behaviour uses **recursion**:

\[
\text{SWITCH} = \text{OFF,}
\]
\[
\text{OFF} = (\text{on} \rightarrow \text{ON}),
\]
\[
\text{ON} = (\text{off} \rightarrow \text{OFF}).
\]

Substituting to get a more succinct definition:

\[
\text{SWITCH} = \text{OFF,}
\]
\[
\text{OFF} = (\text{on} \rightarrow (\text{off} \rightarrow \text{OFF})).
\]

And again:

\[
\text{SWITCH} = (\text{on} \rightarrow \text{OFF} \rightarrow \text{SWITCH}).
\]

---

**FSP - action prefix & recursion**

---

**FSP - action prefix**

FSP model of a traffic light:

\[
\text{TRAFFICLIGHT} = (\text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \text{orange} \rightarrow \text{red} \rightarrow \text{orange} \rightarrow \text{green} \ldots)
\]

LTS generated using LTSA:

Trace:

\[
\text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \text{orange} \rightarrow \text{red} \rightarrow \text{orange} \rightarrow \text{green} \ldots
\]

---

**FSP - choice**

If \(x\) and \(y\) are actions then \((x \rightarrow P \mid y \rightarrow Q)\) describes a process which initially engages in either of the actions \(x\) or \(y\). After the first action has occurred, the subsequent behavior is described by \(P\) if the first action was \(x\) and \(Q\) if the first action was \(y\).

---

**Non-deterministic choice**

Process \((x \rightarrow P \mid x \rightarrow Q)\) describes a process which engages in \(x\) and then behaves as either \(P\) or \(Q\).

---

**FSP - choice**

FSP model of a drinks machine:

\[
\text{DRINKS} = (\text{red} \rightarrow \text{coffee} \rightarrow \text{DRINKS}) \mid (\text{blue} \rightarrow \text{tea} \rightarrow \text{DRINKS}).
\]

LTS generated using LTSA:

---

Who or what makes the choice?

Is there a difference between input and output actions?

---

Tossing a coin.

Tossing a coin.

---

Could we make this deterministic and trace equivalent?

Would it really have equivalent behaviour?
Modelling failure

How do we model an unreliable communication channel which accepts in actions and if a failure occurs produces no output, otherwise performs an out action?

Use non-determinism...

CHAN = (in->CHAN |
in->out->CHAN ).

Deterministic?

FSP - indexed processes and actions

Single slot buffer that inputs a value in the range 0 to 3 and then outputs that value:

BUFF = (in[i:0..3]->out[i]->BUFF).

equivalent to

BUFF = (in[0]->out[0]->BUFF |
in[1]->out[1]->BUFF |
in[2]->out[2]->BUFF |
in[3]->out[3]->BUFF ).

or using a process parameter with default value:

BUFF(N=3) = (in[i:0..N]->out[i]->BUFF).

FSP - guarded actions

The choice (when B x -> P | y -> Q) means that when the guard B is true then the actions x and y are both eligible to be chosen, otherwise if B is false then the action x cannot be chosen.

COUNT (N=3) = COUNT[0],
COUNT[i:0..N] = (when (i<N) inc->COUNT[i+1]
|when (i>0) dec->COUNT[i-1] )..

FSP - guarded actions

A countdown timer which, once started, beeps after N ticks, or can be stopped.

COUNTDOWN (N=3) = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
|when (i>0) tick->COUNTDOWN[i-1]
|when (i=0) beep->STOP
|stop->STOP).

FSP - guarded actions

A countdown timer which, once started, beeps after N ticks, or can be stopped.

COUNTDOWN (N=3) = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
|when (i>0) tick->COUNTDOWN[i-1]
|when (i=0) beep->STOP
|stop->STOP).

FSP - guarded actions

Local indexed process definitions are equivalent to process definitions for each index value

const N = 1
range T = 0..N
range R = 0..2*N
SUM = (in[a:T][b:T]->TOTAL[a+b]),
TOTAL[s:R] = (out[s]->SUM).

index expressions to model calculation:

BUF(N=3) = (in[i:0..N]->out[i]->BUFF).
FSP - guarded actions

What is the following FSP process equivalent to?

\[
\text{const False = 0} \\
P = (\text{when (False) do anything}) \rightarrow P.
\]

Answer:

STOP

FSP - process alphabets

The alphabet of a process is the set of actions in which it can engage.

Process:
COUNTDOWN
Alphabet:
\{ beep, start, stop, tick \}

FSP - process alphabet extension

Alphabet extension can be used to extend the implicit alphabet of a process:

\[
\text{WRITER = (write[1] -> write[3] -> WRITER) + \{write[0..3]\}}.
\]

(we make use of alphabet extensions in later chapters to control interaction between processes)

Revision & Wake-up Exercise

In FSP, model a process \text{FILTER}, that filters out values greater than 2:

ie. it inputs a value \( v \) between 0 and 5, but only outputs it if \( v \leq 2 \), otherwise it discards it.

\[
\text{FILTER = (in[v:0..5] \rightarrow DECIDE[v])}, \\
\text{DECIDE[v:0..5] = ( ? )}.
\]

2.2 Implementing processes

Modeling processes as finite state machines using FSP/LTS.

Implementing threads in Java.

Note: to avoid confusion, we use the term \text{process} when referring to the models, and \text{thread} when referring to the implementation in Java.

Implementing processes - the OS view

A (heavyweight) process in an operating system is represented by its code, data and the state of the machine registers, given in a descriptor. In order to support multiple (lightweight) \text{threads of control}, it has multiple stacks, one for each thread.

\[
\begin{array}{ccc}
\text{OS Process} & \text{Data} & \text{Code} & \text{Descriptor} \\
\text{Thread 1} & \text{Stack} & \text{Descriptor} & \text{Stack} \\
\text{Thread 2} & \text{Stack} & \text{Descriptor} & \text{Stack} \\
\text{...} & \text{...} & \text{...} & \text{...} \\
\text{Thread n} & \text{Stack} & \text{Descriptor} & \text{Stack}
\end{array}
\]
threads in Java

A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.

The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

```java
class MyThread extends Thread {
    public void run() {
        //......
    }
}
```

Creating and starting a thread object:

```java
Thread a = new MyThread();
a.start();
```

Since Java does not permit multiple inheritance, we often implement the `run()` method in a class not derived from Thread but from the interface Runnable. This is also more flexible and maintainable.

```java
public interface Runnable {
    public abstract void run();
}
```

```java
class MyRun implements Runnable {
    public void run() {
        //.....
    }
}
```

Creating and starting a thread object:

```java
Thread b = new Thread(new MyRun());
b.start();
```

thread alive states in Java

Once started, an alive thread has a number of substates:

- Runnable
- Non-Runnable

Java thread lifecycle - an FSP specification

```
<table>
<thead>
<tr>
<th>Thread</th>
<th>Created</th>
<th>Runnable</th>
<th>Running</th>
<th>Terminated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Created</td>
<td>(start)</td>
<td>(dispatch)</td>
<td>(sleep, wait)</td>
<td>(end)</td>
</tr>
<tr>
<td>Runnable</td>
<td>(timeout, notify)</td>
<td>(yield, timeslice)</td>
<td>(run)</td>
<td>(timeout)</td>
</tr>
<tr>
<td>Running</td>
<td>(timeout, notify)</td>
<td>(yield, timeslice)</td>
<td>(run)</td>
<td>(timeout)</td>
</tr>
</tbody>
</table>
```

Dispatch, timeslice, end, run, and timeout are not methods of class Thread, but model the thread execution and scheduler.

| THREAD | Created
|--------|--------|
| Created | Runnable
| Runnable | Running
| Running | Terminated

States 0 to 4 correspond to Created, Runnable, Running, Terminated and Non-Runnable respectively.
CountDown timer example

COUNTDOWN (N=3) = (start->COUNTDOWN[N]),
COUNTDOWN[1..N] =
  (when(i>0) tick->COUNTDOWN[i-1]
   |when(i==0)beep->STOP
   |stop->STOP
).

Implementation in Java?

CountDown timer - class diagram

The class CountDown derives from Applet and contains the implementation of the run() method which is required by Thread.

CountDown class

```java
public class CountDown extends Applet implements Runnable {
    Thread counter; int i;
    final static int N = 10;
    AudioClip beepSound, tickSound;
    NumberCanvas display;
    public void init() {...}
    public void start() {...}
    public void stop() {...}
    public void run() {...}
    private void tick() {...}
    private void beep() {...}
}
```

Summary

◆ Concepts
  ● process - unit of concurrency, execution of a program

◆ Models
  ● LTS to model processes as state machines - sequences of atomic actions
  ● FSP to specify processes using prefix "->", choice "|" and recursion.

◆ Practice
  ● Java threads* to implement processes.
  ● Thread lifecycle - created, running, runnable, non-runnable, terminated.

* see also java.util.concurrent
* cf. POSIX pthreads in C
Chapter 3

Concurrent Execution

Concepts: processes - concurrent execution and interleaving, process interaction.
Models: parallel composition of asynchronous processes - interleaving interaction - shared actions process labeling, and action relabeling and hiding structure diagrams
Practice: Multithreaded Java programs

3.1 Modeling Concurrency

◆ How should we model process execution speed?
  - arbitrary speed (we abstract away time)
◆ How do we model concurrency?
  - arbitrary relative order of actions from different processes (interleaving but preservation of each process order)
◆ What is the result?
  - provides a general model independent of scheduling (asynchronous model of execution)

Definitions

◆ Concurrency
  - Logically simultaneous processing. Does not imply multiple processing elements (PEs). Requires interleaved execution on a single PE.
◆ Parallelism
  - Physically simultaneous processing. Involves multiple PEs and/or independent device operations.

Both concurrency and parallelism require controlled access to shared resources. We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-parallel execution.

parallel composition - action interleaving

If P and Q are processes then (P || Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

ITCH = (scratch->STOP).
CONVERSE = (think->talk->STOP).

||CONVERSE_ITCH = (ITCH || CONVERSE).

For example:

ITCH = (0,0) (0,1) (0,2)
CONVERSE = (1,0) (1,1) (1,2)

CONVERSE_ITCH = (0,0) (0,1) (0,2) (1,0) (1,1) (1,2)

Possible traces as a result of action interleaving:

think→talk→scratch
think→scratch→talk
scratch→think→talk

Parallel composition - action interleaving

ITCH = (0,0) (0,1) (0,2)
CONVERSE = (1,0) (1,1) (1,2)

ITCH_ITCH = (0,0) (0,1) (0,2) (1,0) (1,1) (1,2)

CONVERSE_ITCH = (0,0) (0,1) (0,2) (1,0) (1,1) (1,2)

Possible traces as a result of action interleaving:

think→talk→scratch
think→scratch→talk
scratch→think→talk

Both concurrency and parallelism require controlled access to shared resources. We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-parallel execution.
Concurrency: concurrent execution

parallel composition - algebraic laws

Commutative: $P || Q = Q || P$
Associative: $(P || (Q || R)) = ((P || Q) || R) = (P || Q || R)$.

Clock radio example:

- CLOCK = (tick->CLOCK).
- RADIO = (on->off->RADIO).

||CLOCK_RADIO = (CLOCK || RADIO).

LTS? Traces? Number of states?

modeling interaction - shared actions

If processes in a composition have actions in common, these actions are said to be shared. Shared actions are the way that process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

MAKER = (make->ready->MAKER).
USER = (ready->use->USER).

| | | MAKER_USER = (MAKER || USER).

MAKER synchronizes with USER when ready.

LTS? Traces? Number of states?

composite processes

A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

||MAKERS = (MAKE_A || MAKE_B).
||FACTORY = (MAKERS || ASSEMBLE).

Substituting the definition for MAKERS in FACTORY and applying the commutative and associative laws for parallel composition results in the original definition for FACTORY in terms of primitive processes.

||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).

modeling interaction - handshake

A handshake is an action acknowledged by another:

MAKERv2 = (make->ready->used->MAKERv2).
USERv2 = (ready->use->used ->USERv2).

||MAKER_USERv2 = (MAKERv2 || USERv2).

process labeling

a:P prefixes each action label in the alphabet of P with a.

Two instances of a switch process:

SWITCH = (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).

a:SWITCH
0 a.on 1 a.off
b:SWITCH
0 b.on 1 b.off

An array of instances of the switch process:

||SWITCHES(N=3) = (forall[i:1..N] a[i]:SWITCH).
||SWITCHES(N=3) = (s[i:1..N]:SWITCH).

modeling interaction - multiple processes

Multi-party synchronization:

MAKE_A = (makeA->ready->used->MAKE_A).
MAKE_B = (makeB->ready->used->MAKE_B).
ASSEMBLE = (ready->assemble->used->ASSEMBLE).
||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).

Clock radio example:

- CLOCK = (tick->CLOCK).
- RADIO = (on->off->RADIO).

||CLOCK_RADIO = (CLOCK || RADIO).

LTS? Traces? Number of states?

modeling interaction - shared actions

If processes in a composition have actions in common, these actions are said to be shared. Shared actions are the way that process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

MAKER = (make->ready->MAKER).
USER = (ready->use->USER).

| | | MAKER_USER = (MAKER || USER).

MAKER synchronizes with USER when ready.

LTS? Traces? Number of states?

composite processes

A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

||MAKERS = (MAKE_A || MAKE_B).
||FACTORY = (MAKERS || ASSEMBLE).

Substituting the definition for MAKERS in FACTORY and applying the commutative and associative laws for parallel composition results in the original definition for FACTORY in terms of primitive processes.

||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
Process prefixing is useful for modeling shared resources:

\[
\text{RESOURCE} = (\text{acquire} \rightarrow \text{release} \rightarrow \text{RESOURCE}) .
\]

\[
\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) .
\]

\[
\text{||RESOURCE\_SHARE} = (a: \text{USER} \mid | b: \text{USER} \mid | \{a, b\}::\text{RESOURCE}) .
\]

Action relabeling:

\[
\text{||CLIENT\_SERVER} = (\text{CLIENT} \mid | \text{SERVER}) .
\]

where \( \{\text{call/request, reply/wait}\} \).
action hiding

The following definitions are equivalent:

\[
\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \ \backslash \{\text{use}\}.
\]

\[
\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \ \@\{\text{acquire}, \text{release}\}.
\]

structure diagrams

Process P with alphabet \{(a,b)\}.

Parallel Composition \((P || Q) / \{m/a, m/b, c/d\}\)

Composite process \(|S = (P || Q) \@ \{x, y\}\)

structure diagrams - resource sharing

RESOURCE = (acquire \rightarrow \text{release} \rightarrow \text{RESOURCE})

USER = (\text{printer.acquire} \rightarrow \text{printer.release} \rightarrow \text{USER})

\(|\text{PRINTER\_SHARE} = (a: \text{USER} \| | b: \text{USER} \| | (a,b)::\text{printer}:\text{RESOURCE})\).

structure diagrams

We use structure diagrams to capture the structure of a model expressed by the static combinators: parallel composition, relabeling and hiding.

range T = 0..3

BUFF = (\text{in}[i:T] \rightarrow \text{out}[i] \rightarrow \text{BUFF})

||\text{TWOBUF} = ?

3.2 Multi-threaded Programs in Java

Concurrency in Java occurs when more than one thread is alive. ThreadDemo has two threads which rotate displays.
ThreadDemo model

ThreadDemo creates two ThreadPanel displays when initialized.
ThreadPanel manages the display and control buttons, and delegates calls to rotate() to DisplayThread Rotator implements the runnable interface.

ThreadPanel class

public class ThreadPanel extends Panel {
    // construct display with title and segment color
    public ThreadPanel(String title, Color c) {...}
    // rotate display of currently running thread 6 degrees
    // return value not used in this example
    public static boolean rotate() throws InterruptedException {...}
    // create a new thread with target r and start it running
    public void start(Runnable r) {...
        thread = new DisplayThread(canvas,r,...);
        thread.start();
    }
    // stop the thread using Thread.interrupt()
    public void stop() {thread.interrupt();}
}

ThreadDemo implementation in Java - class diagram

Rotator class

class Rotator implements Runnable {
    public void run() {
        try {
            while(true) ThreadPanel.rotate();
            catch(InterruptedException e) {} 
        }
    }
}

Rotator implements the runnable interface, calling ThreadPanel.rotate() to move the display.
run() finishes if an exception is raised by Thread.interrupt.

ThreadDemo class

public class ThreadDemo extends Applet {
    ThreadPanel A; ThreadPanel B;
    public void init() {
        A = new ThreadPanel("Thread A",Color.blue);
        B = new ThreadPanel("Thread B",Color.blue);
        add(A); add(B);
    }
    public void start() {
        A.start(new Rotator());
        B.start(new Rotator());
    }
    public void stop() {
        A.stop();
        B.stop();
    }
}

ThreadDemo creates two ThreadPanel displays when initialized and two threads when started.
ThreadPanel is used extensively in later demonstration programs.

Summary

◆ Concepts
  - concurrent processes and process interaction

◆ Models
  - Asynchronous (arbitrary speed) & interleaving (arbitrary order).
  - Parallel composition as a finite state process with action interleaving.
  - Process interaction by shared actions.
  - Process labeling and action relabeling and hiding.
  - Structure diagrams

◆ Practice
  - Multiple threads in Java.
Shared Objects & Mutual Exclusion

4.1 Interference

Ornamental garden problem:
People enter an ornamental garden through either of two turnstiles. Management wish to know how many are in the garden at any time.

The concurrent program consists of two concurrent threads and a shared counter object.

The Turnstile thread simulates the periodic arrival of a visitor to the garden every second by sleeping for a second and then invoking the increment() method of the counter object.

The run() method exits and the thread terminates after Garden.MAX visitors have entered.

The Counter object and Turnstile threads are created by the go() method of the Garden applet:

```java
private void go() {
    counter = new Counter(counterD);
    west = new Turnstile(westD,counter);
    east = new Turnstile(eastD,counter);
    west.start();
    east.start();
}
```

Note that counterD, westD and eastD are objects of NumberCanvas used in chapter 2.

The ornamental garden program:
The Counter object and Turnstile threads are created by the go() method of the Garden applet:

```java
private void go() {
    counter = new Counter(counterD);
    west = new Turnstile(westD,counter);
    east = new Turnstile(eastD,counter);
    west.start();
    east.start();
}
```

Turnstile class:

```java
class Turnstile extends Thread {
    public void run() {
        try {
            display.setValue(0);
            for (int i=1; i<=Garden.MAX; i++){
                Thread.sleep(500); // 0.5 second between arrivals
                display.setValue(i);
                people.increment();
            }
        } catch (InterruptedException e) {} 
    }
}
```
Counter class

```java
class Counter {
    int value = 0;
    NumberCanvas display;

    Counter(NumberCanvas n) {
        display = n;
        display.setValue(value);
    }

    void increment() {
        int temp = value;
        read value
        Simulate.HWinterrupt();
        value = temp + 1;
        write value
        display.setValue(value);
    }
}
```

Hardware interrupts can occur at arbitrary times.

The counter simulates a hardware interrupt during an increment(), between reading and writing to the shared counter value. Interrupt randomly calls Thread.yield() to force a thread switch.

After the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. Why?

concurrent method activation

Java method activations are not atomic - thread objects east and west may be executing the code for the increment method at the same time.

ornamental garden model

```
const N = 4
range T = 0..N
set VarAlpha = { value.read[T], write[T] } 

VAR = VAR[0],
VAR[u:T] = (read[u] -> VAR[u]
| write[v:T] -> VAR[v]).

TURNSTILE = (go -> RUN)
 | (end -> TURNSTILE),
INCREMENT = (value.read[x:T] -> value.write[x+1] -> RUN
 | VarAlpha).
||GARDEN = (east:TURNSTILE || west:TURNSTILE
 | { east, west, display :: value: VAR})
|{ go / { east, west } . go,
end / { east, west } . end } .
```

The alphabet of process VAR is declared explicitly as a set constant, VarAlpha.

The alphabet of TURNSTILE is extended with VarAlpha to ensure no unintended free actions in VAR i.e. all actions in VAR must be controlled by a TURNSTILE.

checking for errors - animation

Scenario checking - use animation to produce a trace.

Is this trace correct?
The Java™ Tutorials: Concurrency

**Immutable Objects**

An object is considered immutable if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

(The fewer moving things when juggling, the better)

---

**Interference and Mutual Exclusion**

Destructive update, caused by the arbitrary interleaving of read and write actions, is termed interference. (aka a “data race”)

Interference bugs are extremely difficult to locate. The general solution is to give methods mutually exclusive access to shared objects.

Mutual exclusion can be modeled as atomic actions. (functional programming: no updates / interference)

---

**4.2 Mutual exclusion in Java**

Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword `synchronized`.

We correct `COUNTER` class by deriving a class from it and making the increment method `synchronized`:

```java
class SynchronizedCounter extends Counter {
    SynchronizedCounter(NumberCanvas n) {
        super(n);
    }
    synchronized void increment() {
        super.increment();
    }
}
```

Java associates a lock with every object. The Java compiler inserts code to acquire the lock before executing the body of the `synchronized` method and code to release the lock before the method returns. Concurrent threads are blocked until the lock is released.

---

**Concurrent: shared objects & mutual exclusion**

©Magee/Kramer

checking for errors - exhaustive analysis

Exhaustive checking - compose the model with a TEST process which sums the arrivals and checks against the display value:

```
TEST[v:T] = TEST[0],
(TEST[v:T] =
  (when (v<N) {east.arrive, west.arrive} -> TEST[v+1]
   | end -> CHECK[v])}
  )+(display.VarAlpha).
```

---

**Concurrent: shared objects & mutual exclusion**

©Magee/Kramer

**ornamental garden model - checking for errors**

Use LTSA to perform an exhaustive search for ERROR.

```
||TESTGARDEN = (GARDEN || TEST).

Trace to property violation in TEST: 
go east.arrive east.value.read.0 west.arrive west.value.read.0 east.value.write.1 west.value.write.1 end display.value.read.1 wrong
```

LTSA produces the shortest path to reach ERROR.

---

**Concurrent: shared objects & mutual exclusion**

©Magee/Kramer

Interference and Mutual Exclusion

Destructive update, caused by the arbitrary interleaving of read and write actions, is termed interference. (aka a “data race”)

Interference bugs are extremely difficult to locate. The general solution is to give methods mutually exclusive access to shared objects.

Mutual exclusion can be modeled as atomic actions. (functional programming: no updates / interference)

---

**Concurrent: shared objects & mutual exclusion**

©Magee/Kramer

**The Java™ Tutorials: Concurrency**

**Immutable Objects**

An object is considered immutable if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

(The fewer moving things when juggling, the better)
4.3 Modeling mutual exclusion

To add locking to our model, define a LOCK, compose it with the shared VAR in the garden, and modify the alphabet set:

\[
\text{LOCK} = (\text{acquire} \to \text{release} \to \text{LOCK}),
\]
\[
||\text{LOCKVAR} = (\text{LOCK} \mid \mid \text{VAR}).
\]
\[
\text{set VarAlpha} = \{\text{value}.\{\text{read}[T],\text{write}[T],
\text{acquire, release}\}\}
\]

Modify TURNSTILE to acquire and release the lock:

\[
\text{TURNSTILE} = (\text{go} \to \text{RUN}),
\]
\[
\text{RUN} = (\text{arrive} \to \text{INCREMENT}
\text{end} \to \text{TURNSTILE}),
\]
\[
\text{INCREMENT} = (\text{value}.\text{acquire}
\to \text{value.read}[x:T]\to\text{value.write}[x+1]
\to \text{value.release}\to\text{RUN})+\text{VarAlpha}.
\]

Why is this “less elegant”?

To ensure mutually exclusive access to an object, all object methods should be synchronized.

Use TEST and LTSA to perform an exhaustive check.

Is TEST satisfied?

Revised ornamental garden model - checking for errors

Go
east.arrive
east.value.acquire
east.value.read.0
east.value.write.1
east.value.release
west.arrive
west.value.acquire
west.value.read.1
west.value.write.2
west.value.release
end
display.value.read.2
right

We can give a more abstract, simpler description of a COUNTER which generates the same LTS:

\[
\text{COUNTER} = \text{COUNTER}[0]
\]
\[
\text{COUNTER}[v:T] = (\text{when (v<N) increment} \to \text{COUNTER}[v+1]).
\]

This therefore exhibits "equivalent" behavior i.e. has the same observable behavior.

Summary

◆ Concepts
  • process interference
  • mutual exclusion

◆ Models
  • model checking for interference
  • modeling mutual exclusion

◆ Practice
  • thread interference in shared Java objects
  • mutual exclusion in Java (synchronized objects/methods).

Java synchronized statement

Access to an object may also be made mutually exclusive by using the synchronized statement:

\[
synchronized (\text{object}) \{ \text{statements} \}
\]

A less elegant way to correct the example would be to modify the Turnstile.run() method:

\[
synchronized(counter) \{\text{counter.increment();}\}
\]

To model shared objects directly in terms of their synchronized methods, we can abstract the details by hiding.

For SynchronizedCounter we hide read, write, acquire, release actions.

const N = 4
range T = 0..N
VAR = VAR[0],
VAR[u:T] = \{\text{read}[u] \to \text{VAR}[u]
| \text{write}[v:T] \to \text{VAR}[v]\}.

LOCK = (\text{acquire} \to \text{RELEASE} \to \text{LOCK}).

INCREMENT = (\text{acquire} \to \text{read}[x:T]
\to (\text{when (x<N) write}[x+1]
\to \text{RELEASE} \to \text{INCREMENT}
\text{end})
\to \text{VAR}[T],\text{write}[T]\}.

||\text{COUNTER} = (\text{INCREMENT} \mid \text{LOCK} \mid \text{VAR}) @\{\text{increment}\}.

COUNTER: Abstraction using action hiding

Minimized

LTS:

<table>
<thead>
<tr>
<th>0</th>
<th>increment</th>
<th>increment</th>
<th>increment</th>
<th>increment</th>
</tr>
</thead>
</table>

To add locking to our model, define a LOCK, compose it with the shared VAR in the garden, and modify the alphabet set:

\[
\text{LOCK} = (\text{acquire} \to \text{release} \to \text{LOCK}).
\]
\[
||\text{LOCKVAR} = (\text{LOCK} \mid \mid \text{VAR}).
\]
\[
\text{set VarAlpha} = \{\text{value}.\{\text{read}[T],\text{write}[T],
\text{acquire, release}\}\}
\]

Modify TURNSTILE to acquire and release the lock:

\[
\text{TURNSTILE} = (\text{go} \to \text{RUN}),
\]
\[
\text{RUN} = (\text{arrive} \to \text{INCREMENT}
\text{end} \to \text{TURNSTILE}),
\]
\[
\text{INCREMENT} = (\text{value}.\text{acquire}
\to \text{value.read}[x:T]\to\text{value.write}[x+1]
\to \text{value.release}\to\text{RUN})+\text{VarAlpha}.
\]

Why is this “less elegant”?

To ensure mutually exclusive access to an object, all object methods should be synchronized.

Use TEST and LTSA to perform an exhaustive check.

Is TEST satisfied?

Revised ornamental garden model - checking for errors

Go
east.arrive
east.value.acquire
east.value.read.0
east.value.write.1
east.value.release
west.arrive
west.value.acquire
west.value.read.1
west.value.write.2
west.value.release
end
display.value.read.2
right

We can give a more abstract, simpler description of a COUNTER which generates the same LTS:

\[
\text{COUNTER} = \text{COUNTER}[0]
\]
\[
\text{COUNTER}[v:T] = (\text{when (v<N) increment} \to \text{COUNTER}[v+1]).
\]

This therefore exhibits "equivalent" behavior i.e. has the same observable behavior.

Summary

◆ Concepts
  • process interference
  • mutual exclusion

◆ Models
  • model checking for interference
  • modeling mutual exclusion

◆ Practice
  • thread interference in shared Java objects
  • mutual exclusion in Java (synchronized objects/methods).
Monitors & Condition Synchronization

Concepts:
- Monitors: encapsulated data + access procedures
- Mutual exclusion + condition synchronization
- Single access procedure active in the monitor
- Nested monitors

Models:
- Guarded actions

Practice:
- Private data and synchronized methods (exclusion).
- `wait()`, `notify()` and `notifyAll()` for condition synch.
- Single thread active in the monitor at a time

5.1 Condition Synchronization

A controller is required for a carpark, which only permits cars to arrive when the carpark is not full and does not permit cars to depart when there are no cars in the carpark. Car arrival and departure are simulated by separate threads.

Carpark model

Here? Events or actions of interest? arrive and depart

Define each process and interactions (structure).

Guarded actions are used to control arrive and depart.
carpark program

- Model - all entities are processes interacting by actions
- Program - need to identify threads and monitors
  - thread - active entity which initiates (output) actions
  - monitor - passive entity which responds to (input) actions.

For the carpark?

```
class Arrivals implements Runnable {
    CarParkControl carpark;
    Arrivals(CarParkControl c) {carpark = c;}
    public void run() {
        try {
            ThreadPanel.rotate(330); // ARRIVALS
            Arrivals arrive
            carpark.arrive();
            ThreadPanel.rotate(30);
        } catch (InterruptedException e){}
    }
}
```

```
class CarParkControl {
    protected int spaces;
    protected int capacity;
    CarParkControl(int n)
        {capacity = spaces = n;}
    synchronized void arrive() {
        ++spaces;
    }
    synchronized void depart() {
        --spaces;
    }
    synchronized void depart() {
        ++spaces;
    }
}
```

```
carpcar program - class diagram

Arrivals and Departures implement Runnable, CarParkControl provides the control (condition synchronization).
```

```java
public void start() {
    CarParkControl c = 
    new DisplayCarPark(carDisplay,Places);
    arrivals.start(new Arrivals(c));
    departures.start(new Departures(c));
}
```
Concurrent monitors & condition synchronization

Java provides a thread wait set per monitor (actually per object) with the following methods:

```java
public final void notify()
    Wakes up a single thread that is waiting on this object’s set.

public final void notifyAll()
    Wakes up all threads that are waiting on this object’s set.

public final void wait()
    Blocks the current thread and releases the lock.
    The thread will wait until another thread invokes notify() or notifyAll() on the same object.
    Reacquires the lock when being notified.

public final void wait(long timeout, int nanos)
    Similar to wait() but with a timeout.
```

Concurrent monitors & condition synchronization

FSP: when cond act -> NEWSTAT

Java:
```java
public synchronized void act()
    throws InterruptedException {
        while (!cond) wait(); // NO EXCEPTIONS!
        // modify monitor data // NO EXCEPTIONS!
        notifyAll();
    }
```

The while loop is necessary to retest the condition cond to ensure that cond is indeed satisfied when it re-enters the monitor.

notifyAll() is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

Part II

models to monitors - summary

Active entities (that initiate actions) are implemented as threads.
Passive entities (that respond to actions) are implemented as monitors.

```
Each guarded action in the model of a monitor is implemented as a synchronized method which uses a while loop and wait() to implement the guard. The while loop condition is the negation of the model guard condition.

Changes in the state of the monitor are signaled to waiting threads using notify() or notifyAll().
```

Why is it safe to use notify() here rather than notifyAll()?
### 5.2 Semaphores

Semaphores are widely used for dealing with inter-process synchronization in operating systems. Semaphore s is an integer variable that can take only non-negative values.

The only operations permitted on s are **up(s)** and **down(s)**. Blocked processes are held in a FIFO queue.

#### modeling semaphores

To ensure analyzability, we only model semaphores that take a finite range of values. If this range is exceeded then we regard this as an ERROR. N is the initial value.

```java
const Max = 3 const TRUE = 1 range Int = 0..Max SEMAPHORE (N=0) = SEMA [N], SEMA [v:Int] = (when (TRUE) up->SEMA [v+1] | when (v>0) down->SEMA [v-1] ), SEMA [Max+1] = ERROR.
```

#### semaphore demo - model

Three processes p[1..3] use a shared semaphore mutex to ensure mutually exclusive access (action critical) to some resource.

```
LOCK = (mutex.down->critical->mutex.up->LOOP). || SEMADEMO = (p[1..3]:LOOP || {p[1..3]}::mutex:SEMAPHORE(1)).
```

For mutual exclusion, the semaphore initial value is 1. **Why?**

Is the ERROR state reachable for SEMADEMO? **Is a binary semaphore sufficient (i.e. Max=1)?** **LTS?**

#### semaphore demo - model

```
0 1 2 3 4 5 6
```

---

### modeling semaphores

Action down is only accepted when value v of the semaphore is greater than 0.

Action up is not guarded.

Trace to a violation:

```
up → up → up → up
```

---

### semaphores in Java

Semaphores are passive objects, therefore implemented as monitors.

(Note: In practice, semaphores are a low-level mechanism often used for implementing the higher-level monitor construct. Java SE5 provides general counting semaphores.)

```java
public class Semaphore {
    private int value;
    public Semaphore (int initial) {value = initial;}
    synchronized public void up () {
        //while (!true) wait();????
        ++value;
        notifyAll();
    }
    synchronized public void down () {
        throws InterruptedException {while (value== 0) wait();
        --value;
        // notifyAll():????
    }
}
```
What if we adjust the time that each thread spends in its critical section?

- Large resource requirement - more conflict?
  (eg. more than 67% of a rotation)?
- Small resource requirement - no conflict?
  (eg. less than 33% of a rotation)?

Hence the time a thread spends in its critical section should be kept as short as possible.

**Part III**

5.3 Bounded Buffer

A bounded buffer consists of a fixed number of slots. Items are put into the buffer by a producer process and removed by a consumer process. It can be used to smooth out transfer rates between the producer and consumer.

(see car park example)

```java
class MutexLoop implements Runnable {
    Semaphore mutex;
    MutexLoop (Semaphore sema) {mutex=sema;}
    public void run() {
        try {
            while(true) {
                while(!ThreadPanel.rotate());
                mutex.down(); // get mutual exclusion
                while(ThreadPanel.rotate()); // critical actions
                mutex.up(); // release mutual exclusion
            }
        } catch(InterruptedException e){}
    }
}
```

ThreadPanel.rotate() returns false while executing non-critical actions (dark color) and true otherwise.
Some *System* Design Patterns

- Smooth out spikes:
  - Buffers (trade space for time)
- Increase throughput:
  - Parallelism:
    - SIMD (e.g., GPUs)
    - MIMD (e.g., Pipeline, threads)
  - Play the odds:
    - Pre-fetching (trade space for time)
    - Caching (trade space for time)
- Make changes easier:
  - Add indirection (pointers)

bounded buffer - a data-independent model

The behaviour of BOUNDEDBUFFER is independent of the actual data values, and so can be modelled in a data-independent manner.

LTS:

PRODUCER = (put->PRODUCER).
CONSUMER = (get->CONSUMER).
||BOUNDEDBUFFER = (PRODUCER||BUFFER(5)||CONSUMER).

bounded buffer program - buffer monitor

```java
public interface Buffer {
    void put(Object o) throws InterruptedException;
    Object get() throws InterruptedException;
}

public class BufferImpl implements Buffer {
    public synchronized void put(Object o) throws InterruptedException {
        while (count==size) wait();
        buf[in] = o; ++count; in=(in+1)%size;
        notify(); // notifyAll() ?
    }

    public synchronized Object get() throws InterruptedException {
        while (count==0) wait();
        Object o =buf[out];
        buf[out]=null; --count; out=(out+1)%size;
        notify(); // notifyAll() ?
        return (o);
    }
}
```

bounded buffer program - producer process

```java
public class Producer implements Runnable {
    Buffer buf;
    String alphabet= "abcdefghijklmnopqrstuvwxyz";
    Producer(Buffer b) {buf = b;}
    public void run() {
        try {
            int ai = 0;
            while(true) {
                ThreadPanel.rotate(12);
                buf.put(new Character(alphabet.charAt(ai)));
                ai=(ai+1) % alphabet.length();
                ThreadPanel.rotate(348);
            }
        } catch (InterruptedException e){}
    }
}
```

Part IV
Concurrency: monitors & condition synchronization

Each Java object has a thread wait set and the following methods:

```java
public final void notify(notifyAll)()
Wakes up a single thread that is waiting on this object’s set.
Notifying threads have no idea what the others are waiting for.
public final void wait()
throws InterruptedException
Waits to be notified by another thread. The waiting thread
releases the synchronization lock associated with the monitor.
When notified, the thread must wait to reacquire the monitor
before resuming execution.
```

Can’t we tell notifying threads what the others are waiting for?

### Nested Monitors!

Suppose that, in place of using the `count` variable and condition synchronization directly, we instead use two semaphores `full` and `empty` to reflect the state of the buffer.

```java
class SemaBuffer implements Buffer {
    Semaphore full;  // counts number of items
    Semaphore empty;  // counts number of spaces
    SemaBuffer(int size) {
        this.size = size; buf = new Object[size];
        full = new Semaphore(0);
        empty = new Semaphore(size);
    }
    // Semaphore’re value = # available resources
    // Cannot count in critical region.
}
```

nested monitors - bounded buffer model

```java
const Max = 5
range Int = 0..Max
SEMAPHORE ... as before...
BUFFER = (put -> empty.down -> full.up -> BUFFER
| get -> full.down -> empty.up -> BUFFER).
PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).
|| BOUNDEDBUFFER = (PRODUCER || BUFFER || CONSUMER
| || empty: SEMAPHORE(5)
| || full: SEMAPHORE(0))
@ {put, get}.
```

LTSA analysis predicts a possible DEADLOCK:

Composing potential DEADLOCK
States Composed: 28 Transitions: 32 in 60 ms
Trace to DEADLOCK:
```
get
```

The Consumer tries to get a character, but the buffer is empty. It blocks and releases the lock on the semaphore `full`. The Producer tries to put a character into the buffer, but also blocks. Why?

This situation is known as the nested monitor problem.

### Nested Monitors - Bounded Buffer Program

```java
synchronized public void put(Object o) throws InterruptedException {
    empty.down();
    buf[put] = o;
    ++count; in = (in+1)%size;
    full.up();
}
```

We signal only those who care about our signal!

```java
synchronized public Object get() throws InterruptedException {
    Object o = buf[get]; buf[get] = null;
    --count; out = (out+1)%size;
    empty.up() return (o);
}
```

Does this behave as desired?

empty is decremented during a put operation, which is blocked if empty is zero; full is decremented by a get operation, which is blocked if full is zero.

### Nested Monitors - Revised Bounded Buffer Program

The only way to avoid it in Java is by careful design. In this example, the deadlock can be removed by ensuring that the monitor lock for the buffer is not acquired until after semaphores are decremented.

```java
public void put(Object o)
throws InterruptedException {
    empty.down(); /* do I have the resources I need to proceed? */
    synchronized(this) {
        buf[in] = o; ++count; in = (in+1)%size;
    }
    full.up(); /* not inside the monitor; must keep critical region as short as possible. */
}
```
The semaphore actions have been moved to the producer and consumer. This is exactly as in the implementation where the semaphore actions are outside the monitor.

Does this behave as desired?

5.5 Monitor invariants

An invariant for a monitor is an assertion on its fields. Invariants must hold (=non-variant) whenever no thread executes inside the monitor, i.e., on thread entry to and exit from a monitor.

- CarParkControl Invariant: \( 0 \leq \text{spaces} \leq N \)
- Semaphore Invariant: \( 0 \leq \text{value} \)
- Buffer Invariant: \( 0 \leq \text{count} \leq \text{size} \) and \( 0 \leq \text{in} < \text{size} \) and \( 0 \leq \text{out} < \text{size} \) and \( \text{in} = (\text{out} + \text{count}) \mod \text{size} \)

Invariants can be helpful in reasoning about correctness of monitors using a logical proof-based approach. Generally, we prefer to use a model-based approach, as it’s amenable to mechanical checking.
### Deadlock Concepts:
- **System deadlock:** no further progress
- **Four necessary & sufficient conditions**

### Models:
- Deadlock - no eligible actions

### Practice:
- Blocked threads

### Aim:
- Deadlock avoidance - to design systems where deadlock cannot occur.

---

#### Wait-for cycle

- Has A awaits B
- Has E awaits A
- Has B awaits C
- Has C awaits D
- Has D awaits E

---

#### 6.1 Deadlock analysis - primitive processes

- Deadlocked state is one with no outgoing transitions
- In FSP: **STOP** process

```plaintext
MOVE = (north->x(south->MOVE|north->STOP)).
```

- Animation to produce a trace.
- Analysis using **LTSA:**
  
  **Trace to DEADLOCK:**
  - Shortest trace to **STOP**:
    - North
    - North

---

#### Deadlock: four necessary and sufficient conditions

- **Serially reusable resources:**
  - The processes involved share resources which they use under mutual exclusion.
- **Incremental acquisition:**
  - Processes hold on to resources already allocated to them while waiting to acquire additional resources.
- **No pre-emption:**
  - Once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.
- **Wait-for cycle:**
  - A circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

---

#### Deadlock analysis - parallel composition

- In systems, deadlock may arise from the parallel composition of interacting processes.

```plaintext
RESOURCE = (get->put->RESOURCE).

P = (printer.get->scanner.get
     ->copy
     ->printer.put->scanner.put
     ->P).

Q = (scanner.get->printer.get
     ->copy
     ->scanner.put->printer.put
     ->Q).

SYS = (p:P||q:Q
      ||{p,q}::printer:RESOURCE
      ||{p,q}::scanner:RESOURCE).
```
6.2 Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

Table of philosophers:

| {DINERS (N=5)= forall [i:0..N-1] (phil[i]:PHIL | {phil[i].left.phil[((i-1)+N)%N].right::FORK )}.

Can this system deadlock?

Dining Philosophers - model

FORK = (get -> put -> FORK).
PHIL = (sitdown ->right.get->left.get ->eat ->right.put->left.put ->arise->PHIL).

Dining Philosophers - model analysis

Trace to DEADLOCK:
phil.0.sitdown
phil.0.right.get
phil.1.sitdown
phil.1.right.get
phil.2.sitdown
phil.2.right.get
phil.3.sitdown
phil.3.right.get
phil.4.sitdown
phil.4.right.get

This is the situation where all the philosophers become hungry at the same time, sit down at the table and each philosopher picks up the fork to his right.

The system can make no further progress since each philosopher is waiting for a fork held by his neighbor i.e. a wait-for cycle exists!

Deadlock is easily detected in our model.

How easy is it to detect a potential deadlock in an implementation?
**Dining Philosophers - implementation in Java**

```java
class Philosopher extends Thread {
    // PHIL = (sitdown ->right. get -> left. get -> right. put -> left. put -> arise -> PHIL).

    public void run() {
        try {
            while (true) {
                // thinking
                view.setPhil(identity, view.THINKING);
                sleep(controller.sleepTime()); // hungry
                view.setPhil(identity, view.HUNGRY);
                right.get(); // got right chopstick
                view.setPhil(identity, view.GOTRIGHT);
                sleep(500);
                left.get(); // eating
                view.setPhil(identity, view.EATING);
                sleep(controller.eatTime());
                right.put();
                left.put();
            }
        } catch (java.lang.InterruptedIOException e) { }
    }
}
```

Follows from the model (sitting down and leaving the table have been omitted).

**Dining Philosophers - Fork monitor**

```java
class Fork {// FORK = (get -> put -> FORK).
    private boolean taken=false;
    private PhilCanvas display;
    private int identity;
    Fork(PhilCanvas disp, int id) {
        display = disp; identity = id;
    }

    synchronized void put() { // WHY ?
        taken=false;
        display.setFork(identity,taken);
        notify(); // WHY ?
    }

    synchronized void get() throws java.lang.InterruptedIOException {
        while (taken) wait(); // WHY ?
        taken=true;
        display.setFork(identity,taken);
    }
}
```

Guarded actions may be hidden in a model.

Actions get & put cannot happen at all times - they're guarded!

Encode the state of the LTS as an explicit variable to expose them:

```java
FTOK = TAKEN[0];
TAKEN[b:0..1] = (when (b) get -> TAKEN[b]) | (when (b) put -> TAKEN[b])
```

**Dining Philosophers**

To ensure deadlock occurs eventually, the slider control may be moved to the left. This reduces the time each philosopher spends thinking and eating. This "speedup" increases the probability of deadlock occurring.

---

**Concurrency: Deadlock**

- **Philosophers**: active entities - implement as threads
- **Forks**: shared passive entities - implement as monitors

**Philosopher implementation**

```java
for (int i =0; i<N; ++i)
    phil[i].start();
```

Code to create the philosopher threads and fork monitors:

```java
for (int i =0; i<N; ++i) { 
    fork[i] = new Fork(display,i);
}
```
Deadlock-free Philosophers

Deadlock can be avoided by ensuring that a wait-for cycle cannot exist. How?

Introduction an asymmetry into our definition of philosophers.

Use the identity I of a philosopher to make even numbered philosophers get their left forks first, odd their right first.

Other strategies?

Maze example - shortest path to “deadlock”

We can exploit the shortest path trace produced by the deadlock detection mechanism of LTSA to find the shortest path out of a maze to the STOP process.

We must first model the MAZE. Each position can be modelled by the moves that it permits. The MAZE parameter gives the starting position.

eg. MAZE(Start=8) = P[Start], P[0] = (north->STOP|east->P[1]),...

Maze example - shortest path to “deadlock”

Shortest path escape trace from position 7?

Trace to DEADLOCK:

\[
\begin{array}{c|c|c}
\text{north} & \text{east} & \text{west} \\
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8 \\
\end{array}
\]

Summary

◆ Concepts
  ● deadlock: no further progress
  ● four necessary and sufficient conditions:
    ◆ serially reusable resources
    ◆ incremental acquisition
    ◆ no preemption
    ◆ wait-for cycle
  
  Aim: deadlock avoidance
  - to design systems where deadlock cannot occur.

◆ Models
  ● no eligible actions (analysis gives shortest path trace)

◆ Practice
  ● blocked threads
Safety & Liveness Properties

Concepts:

- **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.

7.1 Safety

A safety property asserts that nothing bad happens.

- STOP or deadlocked state (no outgoing transitions)
- ERROR process (-1) to detect erroneous behaviour

Property POLITE

Property that it is polite to knock before entering a room.

Traces: knock→enter ✓ enter

property POLITE

= (knock→enter→POLITE).

In all states, all the actions in the alphabet of a property are eligible choices.

Safety properties

Since all actions in the alphabet of a property are eligible choices, composing a property with a set of processes does not affect their correct behaviour. However, if a behaviour can occur which violates the safety property, then ERROR is reachable.

Properties must be deterministic to be transparent.

Trace to ERROR: (shortest trace)

Analysis using LTSA:

STOP or deadlocked state (no outgoing transitions)
Safety properties

- How can we specify that some action, disaster, never occurs?

property CALM = STOP + \{disaster\}.

A safety property must be specified so as to include all the acceptable, valid behaviors in its alphabet.

Part II – Single Lane Bridge

7.2 Single Lane Bridge problem

A bridge over a river is only wide enough to permit a single lane of traffic. Consequently, cars can only move concurrently if they are moving in the same direction. A safety violation occurs if two cars moving in different directions enter the bridge at the same time.

Safety - mutual exclusion

LOOP = (mutex.down -> enter -> exit
-> mutex.up -> LOOP).
||SEMADEMO = (p[1..3]:LOOP
||{p[1..3]}::mutex::SEMAPHORE(1)).

How do we check that this indeed ensures mutual exclusion in the critical section?

property MUX = (p[i:1..3].enter
-> p[i].exit
-> MUX).
||CHECK = (SEMADEMO || MUX).

Check safety using LTSA.

What happens if semaphore is initialized to 2?

The property focuses on system actions ONLY!
Property doesn't care about the mechanism used to achieve it (here mutex.down/up)!

Safety - mutual exclusion

property MUX = (p[i:1..3].enter
-> p[i].exit
-> MUX).
||CHECK = (SEMADEMO || MUX).

Single Lane Bridge - model

- Events or actions of interest? enter and exit
- Identify processes. cars and bridge
- Identify properties. oneway
- Define each process and interactions (structure).
**Single Lane Bridge - CARS model**

```javascript
const N = 3; // number of each type of car
range T = 0..N; // type of car count
range ID = 1..N; // car identities

CAR = (enter->exit->CAR).
```

To model the fact that cars cannot pass each other on the bridge, we model a CONVOY of cars in the same direction. We will have a red and a blue convoy of up to N cars for each direction:

||CARS = (red:CONVOY || blue:CONVOY).

**Single Lane Bridge - safety property ONEWAY**

We now specify safety properties to check that cars do not collide. While red cars are on the bridge only red cars can enter; similarly for blue cars. When the bridge is empty, either a red or a blue car may enter.

```javascript
property ONEWAY = (RED[i].enter -> RED[i+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]
), //i is a count of red cars on the bridge
BLUE[i:ID]= (blue[ID].enter -> BLUE[i+1]
| when(i==1)blue[ID].exit -> ONEWAY
| when(i>1) blue[ID].exit -> BLUE[i-1]
), //i is a count of blue cars on the bridge
```

**Single Lane Bridge - CONVOY model**

```javascript
NOPASS1 = C[1], //preserves entry order
C[i:ID] = (i.enter-> C[i+N+1])
 NOPASS2 = C[1], //preserves exit order
C[i:ID] = (i.exit-> C[i+N+1]).
||CONVOY = (i:CAR||NOPASS1||NOPASS2).
```

Permits 1.enter→2.enter→1.exit→2.exit but not 1.enter→2.enter→2.exit→1.exit
ie. no overtaking.

**Single Lane Bridge - model analysis**

```javascript
||SingleLaneBridge = (CARS|| BRIDGE||ONEWAY).
```

Is the safety property ONEWAY violated?

```javascript
||SingleLaneBridge = (CARS||ONEWAY).
```

Without the BRIDGE constraints, is the safety property ONEWAY violated?

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

**Single Lane Bridge - implementation in Java**

Cars can move concurrently on the bridge only if in the same direction. The bridge maintains counts of blue and red cars on the bridge. Red cars are only allowed to enter when the blue count is zero and vice versa.

```java
Bridge = BRIDGE[0][0], //initially empty
BRIDGE[nr:T][nb:T] = //nz is the red count, nb the blue
(red[ID].enter -> BRIDGE[nr+1][nb] //nb==0
| red[ID].exit -> BRIDGE[nr-1][nb]
|when (nb==0)
|blue[ID].enter -> BRIDGE[nr][nb+1] //nr==0
| blue[ID].exit -> BRIDGE[nr][nb-1]
).
```

Even when 0, exit actions permit the car counts to be decremented. LTSA maps these undefined states to ERROR.
Single Lane Bridge - BridgeCanvas

An instance of BridgeCanvas class is created by SingleLaneBridge applet - ref is passed to each newly created RedCar and BlueCar object.

```java
public void init(int ncars) {...} // set number of cars

public boolean moveRed(int i) {...} // move red car with the identity i a step

public boolean moveBlue(int i) {...} // move blue car with the identity i a step

public synchronized void freeze(){...} // freeze display

public synchronized void thaw(){...} // unfreeze display
```

To ensure safety, the "safe" check box must be chosen in order to select the SafeBridge implementation.

Single Lane Bridge - RedCar

```java
class RedCar implements Runnable {
    BridgeCanvas display; Bridge control; int id;
    RedCar(Bridge b, BridgeCanvas d, int id) {
        display = d; this.id = id; control = b;
    }
    public void run() {
        try {
            while (true) {
                if (!display.moveRed(id)) { // not on bridge
                    control.redEnter(); // request access to bridge
                    while (display.moveRed(id)) { // move over bridge
                        control.redExit(); // release access to bridge
                    }
                }
            }
        } catch (InterruptedException e) {} //release access to bridge
    }
}
```

Similarly for the BlueCar

Single Lane Bridge - class Bridge

```java
class Bridge {
    synchronized void redEnter() {...}
    synchronized void redExit() {...}
    synchronized void blueEnter() {...}
    synchronized void blueExit() {...}
}
```

Class Bridge provides a null implementation of the access methods i.e. no constraints on the access to the bridge.

Result: 

```
Concurrent: safety & liveness properties
©Magee/Kramer
```

Single Lane Bridge - SafeBridge

```java
class SafeBridge extends Bridge {
    private int nred = 0; //number of red cars on bridge
    private int nblue = 0; //number of blue cars on bridge

    // Monitor Invariant: nred0 and nblue0 and
    // not (nred>0 and nblue>0)

    synchronized void redEnter() {...}
    synchronized void blueEnter() {...}

    synchronized void redExit() {
        --nred;
    }
    synchronized void blueExit() {
        --nblue;
    }
```

This is a direct translation from the BRIDGE model.

To avoid unnecessary thread switches, we use conditional notification to wake up waiting threads only when the number of cars on the bridge is zero i.e. when the last car leaves the bridge.

But does every car eventually get an opportunity to cross the bridge? This is a liveness property.
Part III – Liveness and Progress

7.3 Liveness

A safety property asserts that nothing bad happens. A liveness property asserts that something good eventually happens.

Single Lane Bridge: Does every car eventually get an opportunity to cross the bridge?

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.

Progress properties - fair choice

COIN = (toss>heads>COIN \| toss>tails>COIN).

If a coin were tossed an infinite number of times, we would expect that heads would be chosen infinitely often and that tails would be chosen infinitely often.

This requires Fair Choice!

Note: $2 \times \infty = \infty$.

Progress properties

progress $P = \{a_1, a_2, \ldots, a_N\}$ defines a progress property $P$ which asserts that in an infinite execution of a target system, at least one of the actions $a_1, a_2, \ldots, a_N$ will be executed infinitely often.

COIN system: progress HEADS = {heads} ✓ progress TAILS = {tails} ✓

LTSA check progress: No progress violations detected.

Suppose that there were two possible coins that could be picked up:

- a trick coin
- and a regular coin......

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

progress HEADS = {heads} ✓ progress TAILS = {tails} ✗

Progress violation: TAILS Path to terminal set of states:
pick Actions in terminal set:
{toss, heads}

progress HEADSorTails = {heads, tails} ✓

Progress properties
Concurrent processes can cause problems, including those in which cars do not starve. To detect progress problems we must superimpose some scheduling policy for actions, which models the situation in which the bridge is congested.

**Fair choice** means that eventually every possible execution occurs, including those in which cars do not starve. To detect progress problems we must superimpose some scheduling policy for actions, which models the situation in which the bridge is congested.

**Progress - single lane bridge**

The Single Lane Bridge implementation can permit progress violations. However, if default progress analysis is applied to the model then no violations are detected!

**Why not?**

Fair choice means that eventually every possible execution occurs, including those in which cars do not starve. To detect progress problems we must superimpose some scheduling policy for actions, which models the situation in which the bridge is congested.

**Progress - action priority**

Action priority expressions describe scheduling properties:

- High Priority ("<<")
  - $|C = P_1| Q << a_1, ..., a_n$ specifies a composition in which the actions $a_1, ..., a_n$ have higher priority than any other action in the alphabet of $P_1| Q$ including the silent action tau. In any choice in this system which has one or more of the actions $a_1, ..., a_n$ labeling a transition, the transitions labeled with lower priority actions are discarded.

- Low Priority (">>")
  - $|C = P_1| Q >> a_1, ..., a_n$ specifies a composition in which the actions $a_1, ..., a_n$ have lower priority than any other action in the alphabet of $P_1| Q$ including the silent action tau. In any choice in this system which has one or more transitions not labeled by $a_1, ..., a_n$, the transitions labeled by $a_1, ..., a_n$ are discarded.

---

**Part IV – Checking Progress in the Single Lane Bridge**

A terminal set of states is one in which every state is reachable from every other state in the set via one or more transitions, and there is no transition from within the set to any state outside the set.

Terminal sets for TWOCOIN:

- $(1,2)$
- $(3,4,5)$

Given fair choice, each terminal set represents an execution in which each action used in a transition in the set is executed infinitely often. Since there is no transition out of a terminal set, any action that is not used in the set cannot occur infinitely often in all executions of the system - and hence represents a potential progress violation!

**Progress analysis**

A progress property is violated if analysis finds a terminal set of states in which none of the progress set actions appear.

**Default analysis for TWOCOIN?**

Path to terminal set of states:

- Action priority expressions describe scheduling properties:
  - $|C = P_1| Q << a_1, ..., a_n$ specifies a composition in which the actions $a_1, ..., a_n$ have higher priority than any other action in the alphabet of $P_1| Q$ including the silent action tau. In any choice in this system which has one or more of the actions $a_1, ..., a_n$ labeling a transition, the transitions labeled with lower priority actions are discarded.

- $|C = P_1| Q >> a_1, ..., a_n$ specifies a composition in which the actions $a_1, ..., a_n$ have lower priority than any other action in the alphabet of $P_1| Q$ including the silent action tau. In any choice in this system which has one or more transitions not labeled by $a_1, ..., a_n$, the transitions labeled by $a_1, ..., a_n$ are discarded.

**Progress - action priority**

Action priority expressions describe scheduling properties:

- High Priority ("<<")
  - $|C = P_1| Q << a_1, ..., a_n$ specifies a composition in which the actions $a_1, ..., a_n$ have higher priority than any other action in the alphabet of $P_1| Q$ including the silent action tau. In any choice in this system which has one or more of the actions $a_1, ..., a_n$ labeling a transition, the transitions labeled with lower priority actions are discarded.

- Low Priority (">>")
  - $|C = P_1| Q >> a_1, ..., a_n$ specifies a composition in which the actions $a_1, ..., a_n$ have lower priority than any other action in the alphabet of $P_1| Q$ including the silent action tau. In any choice in this system which has one or more transitions not labeled by $a_1, ..., a_n$, the transitions labeled by $a_1, ..., a_n$ are discarded.
Action priority simplifies the resulting LTS by discarding lower priority actions from choices.

Would giving car entry to the bridge high priority make congestion worse or better?

Progress - action priority

Lecture 9 stopped here!

Progress - revised single lane bridge model

The bridge needs to know whether or not cars are waiting to cross.

Modify CAR:

\[
\text{CAR} = (\text{request} \rightarrow \text{enter} \rightarrow \text{exit} \rightarrow \text{CAR}).
\]

Modify BRIDGE:

Red cars are only allowed to enter the bridge if there are no blue cars on the bridge and there are no blue cars waiting to enter the bridge.

Blue cars are only allowed to enter the bridge if there are no red cars on the bridge and there are no red cars waiting to enter the bridge.

Progress violation: BLUECROSS
Path to terminal set of states:
\[ \text{red.1.enter, red.1.exit, red.2.enter, red.2.exit, red.3.enter, red.3.exit} \]
Actions in terminal set:
\[ \text{blue.1.enter, blue.2.enter, blue.2.exit, blue.3.enter, blue.3.exit} \]

Progress violation: REDCROSS
Path to terminal set of states:
\[ \text{red.1.enter, red.1.exit, red.2.enter, red.2.exit} \]
Actions in terminal set:
\[ \text{blue.1.enter, blue.1.exit, blue.2.enter, blue.2.exit, blue.3.enter, blue.3.exit} \]

This corresponds with the observation that, with more than one car, it is possible that whichever color car enters the bridge first will continuously occupy the bridge preventing the other color from ever crossing.

Congestion using action priority?

Could give red cars priority over blue (or vice versa)? In practice neither has priority over the other. Instead we merely encourage congestion by lowering the priority of the exit actions of both cars from the bridge.
The trace is the scenario in which there are cars waiting at both ends, and consequently, the bridge does not allow either red or blue cars to enter.

**Solution?**

Introduce some *asymmetry* in the problem (cf. Dining philosophers).

This takes the form of a boolean variable (bt) which breaks the deadlock by indicating whether it is the turn of blue cars or red cars to enter the bridge.

 Arbitrarily set bt to true initially giving blue initial precedence.

```java
class FairBridge implements Bridge { // This is a direct translation from the model!//***
    private int nred = 0; // count of red cars on the bridge
    private int nblue = 0; // count of blue cars on the bridge
    private int waitred = 0; // count of waiting red cars
    private int waitblue = 0; // count of waiting blue cars
    private boolean blueturn = true;

    // synchronized void redRequest() { ++waitred; while (nblue>0 || (waitblue>0 && blueturn)) wait(); --waitred; ++nred; } //***
    synchronized void redExit() { //***
        --nred;
        blueturn = true;
        if (nred==0) notifyAll();
    } // Tx undo handler!

    synchronized void blueEnter() { //***
        ++waitblue;
        while (nblue>0 || (waitblue>0 && blueturn)) wait();
        --waitblue;
        ++nred;
        synchronized void blueExit() { //***
            --nred;
            blueturn = true;
            if (nred==0) notifyAll();
        }
    }
}
```

The “fair” check box must be chosen in order to select the FairBridge implementation.

Note that we did not need to introduce a new request monitor method. The existing enter methods can be modified to increment a wait count before testing whether or not the caller can access the bridge.
Part V – Readers & Writers

7.5 Readers and Writers

A shared database is accessed by two kinds of processes, Readers and Writers, which execute transactions that examine and update the database. A Writer must have exclusive access to the database; any number of Readers may concurrently access it.

The readers/writers model defines each process (as before).

- Events or actions of interest?
  - acquireRead, releaseRead, acquireWrite, releaseWrite
- Identify processes.
  - Readers, Writers & the RW_Lock
- Identify properties.
  - RW_Safe
  - RW_Progress
- Define each process and interactions (structure).

The lock maintains a count of the number of readers, and a Boolean for the writers.

The lock maintains a count of the number of readers, and a Boolean for the writers.

property SAFE_RW = (acquireRead → READING[i])
| acquireWrite → WRITING |
| READING[i].Readers | 1
| when (i>1) releaseRead → READING[i-1] |
| when (i=1) releaseRead → SAFE_RW |
| WRITING = (releaseWrite → SAFE_RW). |

We can check that RW_LOCK satisfies the safety property ...

\[ | | READWRITELOCK = (RW_LOCK | | SAFE_RW). \]
Concurrency: safety & liveness properties

An ERROR occurs if a reader or writer is badly behaved (release before acquire or more than two readers). We can now compose the READWRITELOCK with READER and WRITER processes according to our structure... ...

Readers/Writers model - progress

progress WRITE = {writer[1..Nwrite].acquireWrite}
progress READ = {reader[1..Nread].acquireRead}

WRITE - eventually one of the writers will acquireWrite
READ - eventually one of the readers will acquireRead

Adverse conditions using action priority?
We lower the priority of the release actions for both readers and writers.

Readers/Writers implementation - ReadWriteSafe

public synchronized void acquireWrite()
throws InterruptedException {
while (writers > 0) {
++writers;
write = true;
}
}

public synchronized void releaseWrite()
writing = false;
notifyAll();
}

Readers/Writers implementation - ReadWriteSafe

We define an interface that identifies the monitor methods that must be implemented, and develop a number of alternative implementations of this interface.

interface ReadWrite {
  public void acquireRead()
  throws InterruptedException;
  public void releaseRead();
  public void acquireWrite()
  throws InterruptedException;
  public void releaseWrite();
}

Readers/Writers implementation - ReadWriteSafe

class ReadWriteSafe implements ReadWrite {
  private int readers = 0;
  private boolean writing = false;
  public synchronized void acquireRead()
  throws InterruptedException {
    while (writing) wait();
    ++readers;
  }
  public synchronized void releaseRead() {
    --readers;
    if (readers == 0) notify(); // notifyAll()
  }

Unblock a single writer when no more readers.

(How do I know only writers are waiting?)
Part V – Readers & Writers – Priority

Readers/Writers - Writer Priority

Strategy:
Block readers if there is a writer waiting.

set Actions = {acquireRead, releaseRead, acquireWrite, 
releaseWrite, requestWrite}

RW_LOCK = RW[0][False][0], 
RW[readers:0..Nread][writing:Boolean][waitingW:0..Nwrite] = 
{when (!writing & & waitingW==0) 
acquireRead -> RW[readers+1][writing][waitingW] 
|releaseRead -> RW[readers-1][writing][waitingW] 
|when (readers==0 & & !writing) 
acquireWrite-> RW[readers][True][waitingW-1] 
|releaseWrite-> RW[readers][False][waitingW] 
|requestWrite-> RW[readers][writing][waitingW+1] }.

Safety and Progress Analysis?

Progress violation: READ
Path to terminal set of states:
writer.1.requestWrite 
writer.2.requestWrite 
Actions in terminal set:
{writer.1.requestWrite, writer.1.acquireWrite, 
writer.1.releaseWrite, writer.2.requestWrite, 
writer.2.acquireWrite, writer.2.releaseWrite}

Reader starvation: if always a writer waiting.

In practice, this may be satisfactory as is usually more read access than write, and readers generally want the most up to date information.

Readers/Writers Model - Writer Priority

Both READ and WRITE progress properties can be satisfied by introducing a turn variable as in the Single Lane Bridge.
Both **READ** and **WRITE** progress properties can be satisfied by introducing a turn variable as in the Single Lane Bridge.

### Concurrency: safety & liveness properties

**Summary**

- **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 always eventually executed
    - fair choice and action priority
    - apply progress check on the final target system model

- **Practice**
  - threads and monitors

---

**Single Lane Bridge problem – NOT ALL PROBLEMS HAVE A CENTRALISED CONTROLLER!!!**

Here it's implied.

But not every problem has one.

In distributed systems this is particularly the case! (no centralised solutions desired, to minimise contention on that centralised controller)
Chapter 8

Model-Based 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

Concurrency: model-based design

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

Any appropriate design approach can be used.

Concurrency: model-based design

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).

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

Concurrency: model-based design

Aim: rigorous design process.

Concurrency: model-based design

8.1 from requirements to models

Concurrency: model-based design

Concurrency: model-based design
**model - design**

- **Main events, actions and interactions.**
  
  ```plaintext
  on, off, resume, brake, accelerator
  engine on, engine off, speed, setThrottle
  clearSpeed, recordSpeed, enableControl, disableControl
  ```

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

- **Identify main properties.**
  
  ```plaintext
  Safety: disabled when off, brake or accelerator pressed.
  ```

- **Define and structure each process.**

**model elaboration - process definitions**

- **CONTROL subsystem**

```plaintext
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 analyse to exhaustively check:

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

**Progress:** Can every action eventually be selected?

**model structure, actions and interactions**

The CONTROL system is structured as two processes. The main actions and interactions are as shown.

- **SENSOR SCAN**
  - Sensors
  - Prompts
  - Control
  - Cruise
  - System

- **INPUT SPEED**
  - Engine
  - Speed

- **SPEED CONTROL**
  - Prompts
  - Control

- **THROTTLE**
  - zoom

**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 )
| engineOff -> INPUTSPEED ).
```

```
ENABLED = ( speed -> setThrottle -> ENABLED |
| recordSpeed, enableControl ) -> ENABLED |
disableControl -> DISABLED ).
```

**model - Safety properties**

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

model - Progress properties

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
disableControl(engineOn)
Actions in terminal set:
{speed, setThrottle, zoom}

Cruise control model - minimized LTS

Sugar model: change property IMPROVEDSAFETY =
{(off,accelerator,brake,disableControl, engineOff) -> IMPROVEDSAFETY
{(on,resume) -> IMPROVEDSAFETY }
},
SAFETYCHECK =
{(off,accelerator,brake,engineOff) -> SAFETYACTION
{disableControl -> IMPROVEDSAFETY }
},
SAFETYACTION ={disableControl -> IMPROVEDSAFETY}.

We can now compose the whole system:

model analysis

CRUISEMINIMIZED = (CRUISECONTROLSYSTEM) @ (Sensors,speed).

Deadlock? Safety? No deadlocks/progress?

OK now?
Concurrency: model-based design

Minimized LTS:

1. \( \text{engineOn} \)
2. \( \text{on} \)
3. \( \text{speed} \)
4. \( \text{engineOff} \)

- No deadlocks/errors
- No progress violations detected.

What about under adverse conditions? Check for system sensitivities.

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

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 cruise control system - class diagram

- Identify the main active entities
- Identify the main (shared) passive entities
- Identify the interactive display environment
- Structure the classes as a class diagram
Concurrent Execution

- Processes and Threads
- Concurrent Execution
- Shared Objects & Interference
- Monitors & Condition Synchronization
- Deadlock
- Safety and Liveness Properties
- Model-based Design
- Dynamic systems
- Concurrent Software Architectures
- Message Passing
- Timed Systems

Aim: rigorous design process.

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)
        (cs,cs) {sc=new SpeedControl(cs,disp);}
    synchronized void engineOn(){
        if (controlState==CRUISING) {sc.enableControl(); controlState=CRUISING;}
    }
    synchronized void engineOff(){
        if (controlState==CRUISING) {sc.disableControl(); controlState=STANDBY;}
    }
    synchronized void setThrottle(double error) {
        double steady = (double)setSpeed/12.0;
        cs.setThrottle(steady+error); // simplified feedback control
    }
    synchronized void recordSpeed() {sc.record(setSpeed); controlState=DISABLED;}
    synchronized void clearSpeed() {sc.clearSpeed(); controlState=ACTIVE;}
    synchronized void resume() {if (controlState==STANDBY) {sc.enableControl(); controlState=CRUISING;}}
    synchronized void off() {if (controlState==CRUISING) {sc.disableControl(); controlState=STANDBY;}}
    synchronized void brake() {if (controlState==CRUISING) ac.disableControl(); controlState=INACTIVE;}
}
```

SpeedControl

```java
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; // initial state
    private CarSpeed cs;
    private CruiseDisplay disp;
    SpeedControl(CarSpeed cs, CruiseDisplay disp)
        (cs,cs) this.cs=cs; this.disp=disp; disp.disable(); disp.record(0);
    synchronized void recordSpeed(){
        if (state==DISABLED) {setSpeed=cs.getSpeed(); disp.record(setSpeed);
        }
    }
    synchronized void clearSpeed(){
        if (state==DISABLED) {setSpeed=cs.getSpeed(); disp.record(setSpeed);
        }
    }
    synchronized void enableControl(){
        if (state==DISABLED) {setSpeed=cs.getSpeed(); disp.record(setSpeed);
        }
    }
    synchronized void disableControl(){
        if (state==DISABLED) {setSpeed=cs.getSpeed(); disp.record(setSpeed);
        }
    }
    SpeedControl start(); state=ENABLED;
}
```

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

Course Outline

- Processes and Threads
- Concurrent Execution
- Shared Objects & Interference
- Monitors & Condition Synchronization
- Deadlock
- Safety and Liveness Properties
- Model-based Design
- Dynamic systems
- Concurrent Software Architectures
- Message Passing
- Timed Systems
Chapter 10
Message Passing

Concepts: synchronous message passing - channel
asynchronous message passing - port
- send and receive / selective receive
rendezvous bidirectional comms - entry
- call and accept ... reply

Models: channel: relabelling, choice & guards
port: message queue, choice & guards
entry: port & channel

Practice: distributed computing (disjoint memory)
threads and monitors (shared memory)

synchronous message passing - applet
A sender communicates with a receiver using a single channel.
The sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.

Java implementation - channel
These are the Java implementations:

```java
class Channel extends Selectable {
    Object chann = null;
    public synchronized void send(Object v) throws InterruptedException {
        chann = v;
        signal();
        while (chann != null) wait();
    }
    public synchronized Object receive() throws InterruptedException {
        block(); clearReady(); //part of Selectable
        Object tmp = chann; chann = null;
        notifyAll(); //could be notify()
        return(tmp);
    }
}
```

10.1 Synchronous Message Passing - channel

Sender send(e,c)
Channel c
Receiver v = receive(c)

♦ send(e,c) - send the value of the expression e to channel c. The process calling the send operation is blocked waiting until the message is received from the channel.

♦ v = receive(c) - receive a value into local variable v from channel c. The process calling the receive operation is blocked waiting until a message is sent to the channel.

cf. distributed assignment v = e

one-to-one

Java implementation - sender
These are the Java implementations:

```java
class Sender implements Runnable {
    private Channel chan;
    private SlotCanvas display;
    Sender(Channel c, SlotCanvas d)
    {chan=c; display=d;}
    public void run() {
        try { int ei = 0;
            while(true) {
                display.enter(String.valueOf(ei));
                ThreadPanel.rotate(12);
                chan.send(new Integer(ei));
                display.leave(String.valueOf(ei));
                ei=(ei+1)%10; ThreadPanel.rotate(348);
            }
        } catch (InterruptedException e){}
    }
}
```
Java implementation - receiver

```java
class Receiver implements Runnable {
  private Channel chan;
  private SlotCanvas display;

  Receiver(Channel c, SlotCanvas d)
  {chan=c; display=d; }

  public void run() {
    try { Integer v=null;
      while (true) {
        ThreadPanel.rotate(180);
        if (v!=null) display.leave(v.toString());
        v = (Integer)chan.receive();
        display.enter(v.toString());
        ThreadPanel.rotate(180);
      }
    } catch (InterruptedException e) {} 
  }
}
```

Java implementation - selective receive

```java
class MsgCarPark implements Runnable {
  private Channel arrive,depart;
  private int spaces,R;
  private StringCanvas disp;

  MsgCarPark(Channel a, Channel l,
            StringCanvas d, int capacity) {
    depart=а; arrive=a; R=spaces=capacity; disp=;
  }

  public void run() {
    CARPARKCONTROL as a thread MsgCarPark
    which receives signals from channels arrive and depart.
    /*CARPARKCONTROL3 N-1 spaces+0L*/
  }
```

How should we deal with multiple channels?

**LTS?**

<table>
<thead>
<tr>
<th>Message operation</th>
<th>FSP model</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(e,chan)</td>
<td>?</td>
</tr>
<tr>
<td>v = receive(chan)</td>
<td>?</td>
</tr>
</tbody>
</table>

How can this be modelled directly without the relabeling?

```
Select sel = new Select();
sel.add(depart);
while (true) {
  ThreadPanel.rotate(12);
  arrive.guard(spaces>0);
  depart.guard(spaces<N);
  switch (sel.choose()) {
  case 1: depart.receive().display(+spaces); break;
  case 2: arrive.receive().display(-spaces); break;
  }
}
```

See Applet
Concurrency: message passing

10.2 Asynchronous Message Passing - port

- **send(e,p)** - send the value of the expression e to port p. The process calling the send operation is not blocked. The message is queued at the port if the receiver is not waiting.

- **v = receive(p)** - receive a value into local variable v from port p. The process calling the receive operation is blocked if there are no messages queued to the port.

**Port**

- **range M = 0..9** // messages with values up to 9
- **set S = (M, [M] [M])** // queue of up to three messages

**PORT**

- **empty state, only send permitted**
- **= (send[x:M] -> PORT[x])**
- **= (send[x:M] -> PORT[x][h]**
- **| receive[h] -> PORT)**

**sense:**

- **Port t:S [h:M]** // two or more messages queued to port
- **Port t:S [h:M]** // two or more messages queued to port
- **= (send[x:M] -> PORT[x][t][h]**
- **| receive[h] -> PORT[t]**

10.3 Rendezvous - entry

Rendezvous is a form of request-reply to support client server communication. Many clients may request service, but only one is serviced at a time.

**Java implementation - port**

```java
import java.util.*;

class Port extends Selectable {
    Vector queue = new Vector();
    public synchronized void send(Object v) {
        queue.addElement(v);
        signal();
    }
    public synchronized Object receive() throws InterruptedException {
        Object tmp = queue.elementAt(0);
        queue.removeElementAt(0);
        return(tmp);
    }
}
```

**model of applet**

- **ASENDER = ASENDE[0]**
- **ASENDER[e:M] = (port.send[e] -> ASENDE[el %10])**

- **ARECEIVER = (port.receive[v:M] -> ARECEIVER)**
- **ANDAsyncMsg = (s[1..2] : ASENDER || ARECEIVER | port:PORT)**
- **/s[1..2] . port.send/port.send**

Safety?

Safety?

Safety?
Rendezvous

- res=call(e,req) - send the value req as a request message which is queued to the entry e.
- req=accept(e) - receive the value of the request message from the entry e into local variable req. The calling process is blocked if there are no messages queued to the entry.
- reply(e,res) - send the value res as a reply message to entry e.

Two clients call a server which services a request at a time.

Instances of threadPanel

Instances ofSlotCanvas

Java implementation - entry

```java
public class Entry extends Port {
  private CallMsg cm;
  public Object call(Object req) throws InterruptedException {
    cm = (CallMsg) receive();
    return cm.request;
  }
  public void reply(Object res) throws InterruptedException {
    cm.replychan.send(res);
  }
}
```

Entries are implemented as extensions of ports, thereby supporting queuing and selective receipt.

The call method creates a channel object on which to receive the reply message. It constructs and sends to the entry a message consisting of a reference to this channel and a reference to the req object. It then awaits the reply on the channel.

The accept method keeps a copy of the channel reference; the reply method sends the reply message to this channel.

What is the difference?

... from the point of view of the client?
... from the point of view of the server?
... mutual exclusion?

Which implementation is more efficient?

... in a local context (client and server in same computer)?
... in a distributed context (in different computers)?
Summary

Concepts
- **synchronous** message passing - *channel*
- **asynchronous** message passing - *port*
  - send and receive / selective receive
- **rendezvous** bidirectional comms - *entry*
  - call and accept ... reply

Models
- channel : relabelling, choice & guards
- port : message queue, choice & guards
- entry : port & channel

Practice
- distributed computing (disjoint memory)
- threads and monitors (shared memory)

Course Outline

- Processes and Threads
- Concurrent Execution
- Shared Objects & Interference
- Monitors & Condition Synchronization
- Deadlock
- Safety and Liveness Properties
- Model-based Design

Practice
- Dynamic systems
- Concurrent Software Architectures
- Message Passing
- Timed Systems