Concurrency: State Models & Java Programs

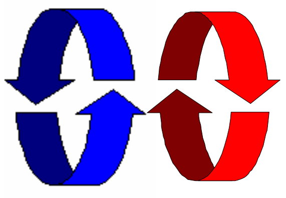
2 authors:

Jeff Magee
Imperial College London
210 PUBLICATIONS **11,068** CITATIONS
SEE PROFILE

Jeff Kramer
Imperial College London
372 PUBLICATIONS **15,220** CITATIONS
SEE PROFILE

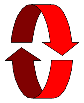
Concurrency

State Models and Java Programs

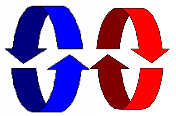


Jeff Magee and **Jeff Kramer**

What is a Concurrent Program?



A **sequential** program has a single thread of control.



A **concurrent** program has multiple threads of control allowing it 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.

Do I need to know about concurrent programming?

Concurrency is widespread but error prone.

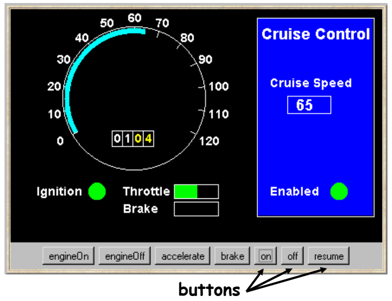
- Therac - 25 computerised radiation therapy machine

  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 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?***

models

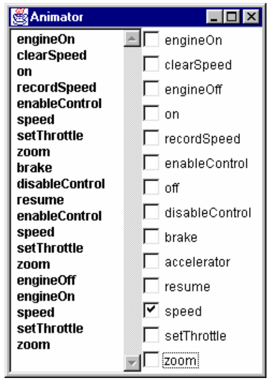
A model is a simplified representation of the real world.

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.

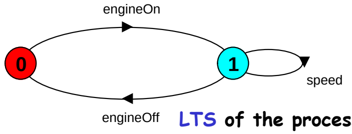
modeling the Cruise Control System



engineOn
clearSpeed
on
recordSpeed
enableControl
speed
setThrottle
zoom
brake
disableControl
resume
enableControl
speed
setThrottle
zoom
engineOff
engineOn
speed
setThrottle
zoom

- ☐ engineOn
- ☐ clearSpeed
- ☐ engineOff
- ☐ on
- ☐ recordSpeed
- ☐ enableControl
- ☐ off
- ☐ disableControl
- ☐ brake
- ☐ accelerator
- ☐ resume
- ☑ speed
- ☐ setThrottle
- ☐ zoom

LTSA **Animator to step through system actions and events.**



LTS of the process that monitors speed.

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

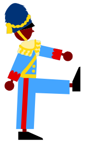
programming practice in Java

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.

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

Web based course material

http://www.doc.ic.ac.uk/~jnm/book/

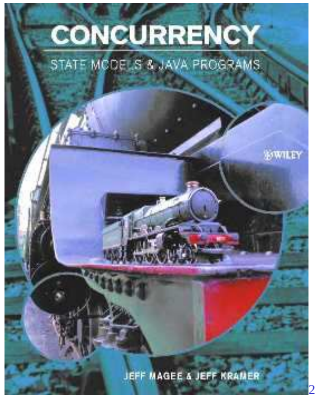
- Java examples and demonstration programs
- State models for the examples
- Labelled Transition System Analyser (***LTSA***) for modeling concurrency, model animation and model property checking.

Book

**Concurrency:
State Models &
Java Programs**

Jeff Magee &
Jeff Kramer

WILEY



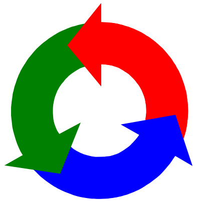
Summary

- Concepts
  - **we adopt a model-based approach for the design and construction of concurrent programs**
- Models
  - **we use finite state models to represent concurrent behavior.**
- Practice
  - **we use Java for constructing concurrent programs.**

Examples are used to illustrate the concepts, models and demonstration programs.

Chapter 2

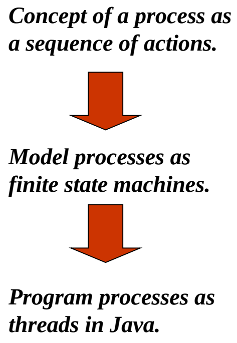
Processes & Threads



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 and threads

Concepts: processes - units of sequential execution.

Models: **finite state processes (FSP)**
to model processes as sequences of actions.
labelled transition systems (LTS)
to analyse, display and animate behavior.

Practice: Java threads

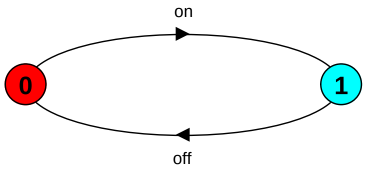
2.1 Modeling Processes

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

modeling processes

A process is the execution of a sequential program. It is modeled as a finite state machine which transits from state to state by executing a sequence of atomic actions.



a light switch
LTS

on→off→on→off→on→off→ ……….

a sequence of actions or *trace*

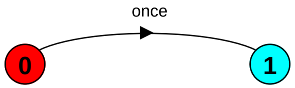
FSP - action prefix

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

ONESHOT = (once -> STOP).

ONESHOT state machine

(terminating process)

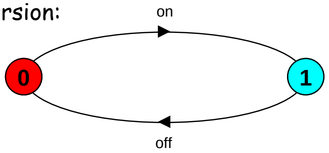


Convention: actions begin with lowercase letters
PROCESSES begin with uppercase letters

FSP - action prefix & recursion

Repetitive behaviour uses recursion:

```
SWITCH = OFF,
OFF    = (on -> ON),
ON     = (off-> OFF).
```



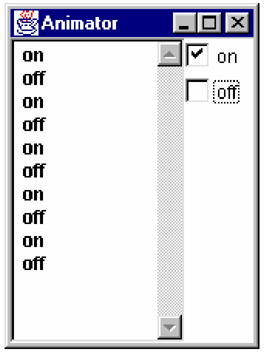
Substituting to get a more succinct definition:

```
SWITCH = OFF,
OFF    = (on ->(off->OFF)).
```

And again:

```
SWITCH = (on->off->SWITCH).
```

animation using LTSA



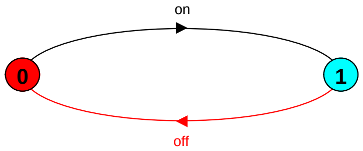
on
off
on
off
on
off
on
off
on
off

☑ on
□ off

The *LTSA* animator can be used to produce a trace.

Ticked actions are eligible for selection.

In the LTS, the last action is highlighted in red.

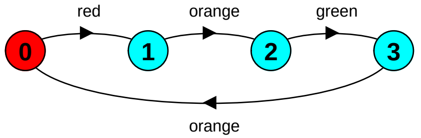


FSP - action prefix

FSP model of a traffic light :

```
TRAFFICLIGHT = (red->orange->green->orange
              -> TRAFFICLIGHT).
```

LTS generated using *LTSA*:



Trace:

red→orange→green→orange→red→orange→green …

FSP - choice

If **x** and **y** are actions then **(x-> P | y-> 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**.

Who or what makes the choice?

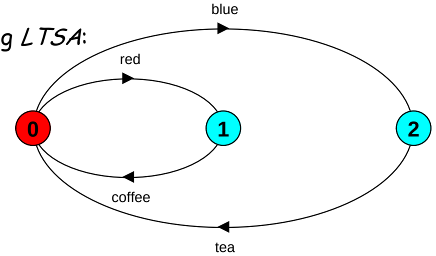
Is there a difference between input and output actions?

FSP - choice

FSP model of a drinks machine :

```
DRINKS = (red->coffee->DRINKS
         |blue->tea->DRINKS
         ).
```

LTS generated using *LTSA*:



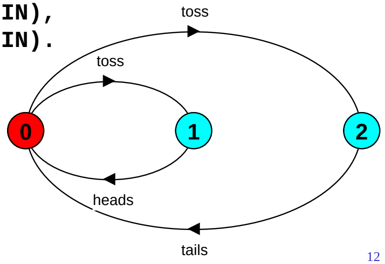
Possible traces?

Non-deterministic choice

Process (x-> P | x -> Q) describes a process which engages in **x** and then behaves as either P or Q.

```
COIN = (toss->HEADS|toss->TAILS),
HEADS= (heads->COIN),
TAILS= (tails->COIN).
```

Tossing a coin.

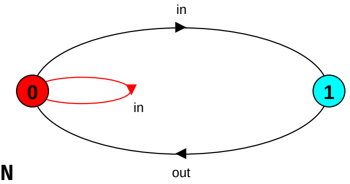


Possible traces?

Modeling 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
       ).
```

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

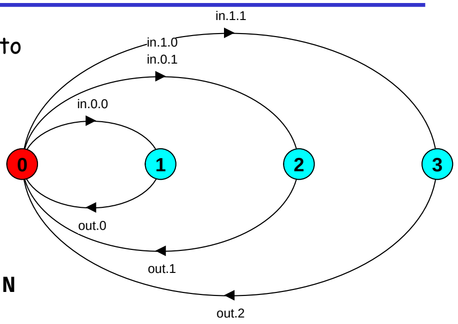
```
    BUFF = (in[0]->out[0]->BUFF
           |in[1]->out[1]->BUFF
           |in[2]->out[2]->BUFF
           |in[3]->out[3]->BUFF
           ).
```

or using a process parameter with default value:

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

FSP - constant & range declaration

index expressions to model calculation:

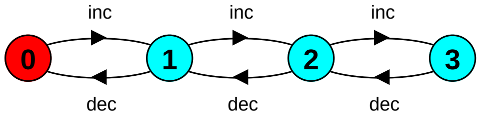


```
const N = 1
range T = 0..N
range R = 0..2*N

SUM        = (in[a:T][b:T]->TOTAL[a+b]),
TOTAL[s:R] = (out[s]->SUM).
```

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.

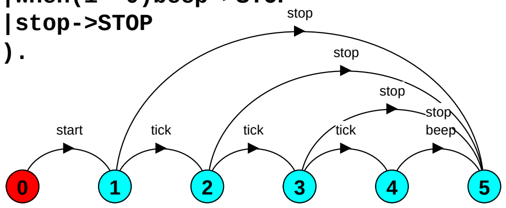
```
COUNT (N=3)    = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                |when(i>0) dec->COUNT[i-1]
                ).
```



FSP - guarded actions

A countdown timer which 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

What is the following FSP process equivalent to?

```
const False = 0
P = (when (False) doanything->P).
```

Answer:

STOP

FSP - process alphabets

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

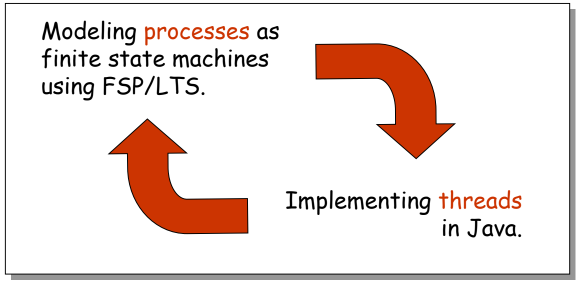
Alphabet extension can be used to extend the **implicit** alphabet of a process:

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

Alphabet of **WRITER** is the set **{write[0..3]}**

(we make use of alphabet extensions in later chapters)

2.2 Implementing processes

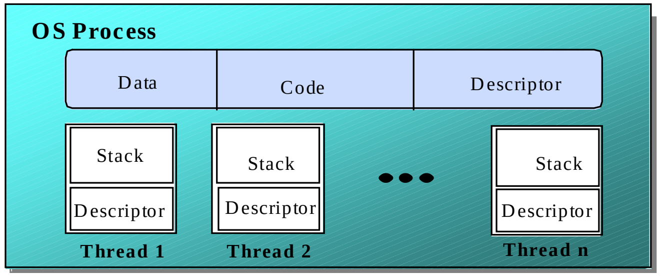


Modeling processes as finite state machines using FSP/LTS.

Implementing threads in Java.

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

Implementing processes - the OS view



OS Process

Data | Code | Descriptor

Stack, Descriptor — Thread 1

Stack, Descriptor — Thread 2

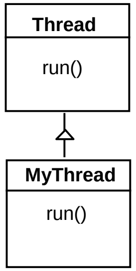
• • •

Stack, Descriptor — Thread n

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) **threads of control**, it has multiple stacks, one for each thread.

threads in Java

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



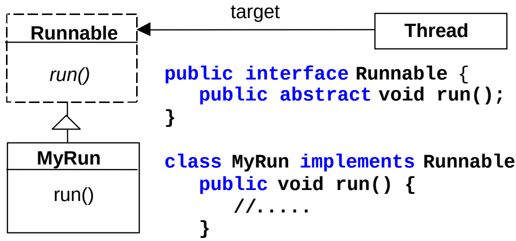
The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

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

```
Thread x = new MyThread();
```

threads in Java

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.

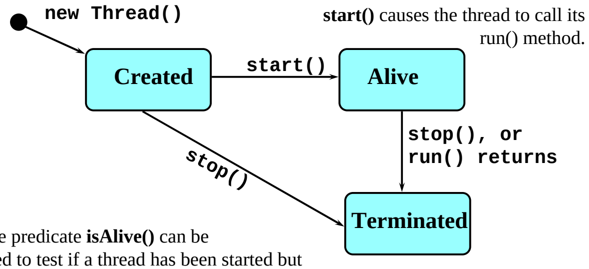


```
public interface Runnable {
   public abstract void run();
}

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

```
Thread x = new Thread(new MyRun());
```

thread life-cycle in Java

An overview of the life-cycle of a thread as state transitions:

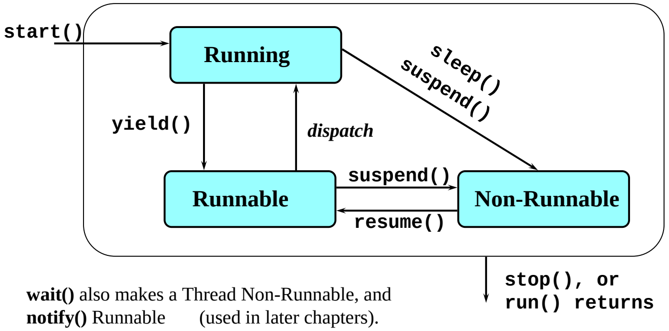


start() causes the thread to call its run() method.

The predicate **isAlive()** can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted (cf. mortals).

thread alive states in Java

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

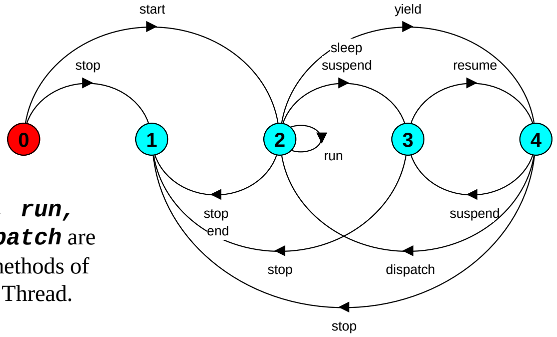


wait() also makes a Thread Non-Runnable, and **notify()** Runnable (used in later chapters).

Java thread lifecycle - an FSP specification

```
THREAD       = CREATED,
CREATED      = (start           ->RUNNING
               |stop            ->TERMINATED),
RUNNING      = ({suspend,sleep}->NON_RUNNABLE
               |yield           ->RUNNABLE
               |{stop,end}      ->TERMINATED
               |run             ->RUNNING),
RUNNABLE     = (suspend         ->NON_RUNNABLE
               |dispatch        ->RUNNING
               |stop            ->TERMINATED),
NON_RUNNABLE = (resume          ->RUNNABLE
               |stop            ->TERMINATED),
TERMINATED   = STOP.
```

Java thread lifecycle - an FSP specification



end, run, dispatch are not methods of class Thread.

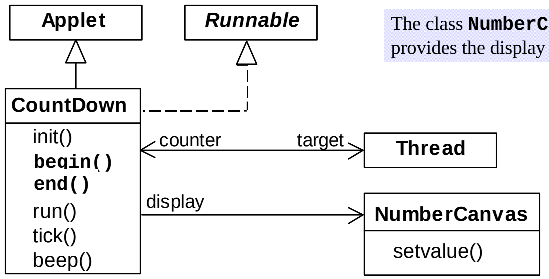
States 0 to 4 correspond to **CREATED, TERMINATED, RUNNING, NON-RUNNABLE,** and **RUNNABLE** respectively.

CountDown timer example

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

Implementation in Java?

CountDown timer - class diagram



The class **NumberCanvas** provides the display canvas.

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

CountDown class

```
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 begin() {...}
  public void end()  {...}
  public void run()   {...}
  private void tick() {...}
  private void beep() {...}
}
```

CountDown class - start(), stop() and run()

```
public void begin() {
   counter = new Thread(this);
   i = N; counter.start();
 }
 public void end() {
   counter = null;
 }
 public void run() {
   while(true) {
     if (counter == null) return;
     if (i>0)  { tick(); --i; }
     if (i==0) { beep(); return;}
   }
 }
```

COUNTDOWN Model

`begin ->`

`end ->`

`COUNTDOWN[i]` process

`recursion` as a `while` loop

```
STOP
when(i>0) tick -> CD[i-1]
when(i==0)beep -> STOP
```

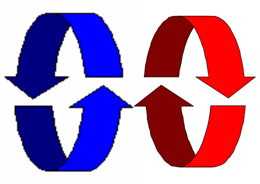
`STOP` when `run()` `returns`

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.

Chapter 3

Concurrent Execution



Concurrent execution

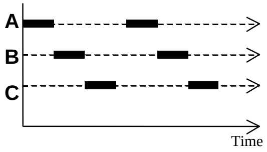
Concepts: processes - concurrent execution and interleaving.
process interaction.

Models: **parallel composition** of asynchronous processes - interleaving
interaction - shared actions
process labeling, and action relabeling and hiding
structure diagrams

Practice: Multithreaded Java programs

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.

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)

parallel composition - action interleaving

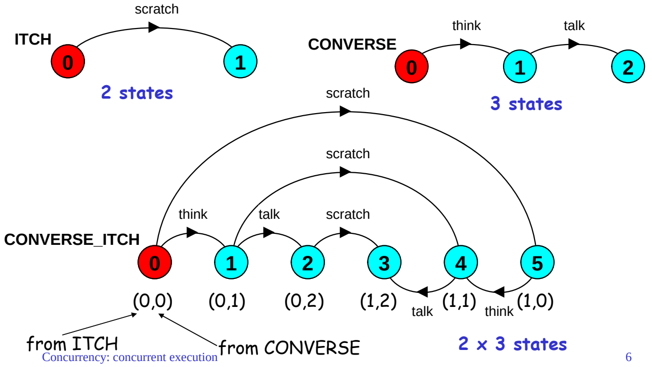
If P and Q are processes then (P||Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

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

||CONVERSE_ITCH = (ITCH || CONVERSE).
```

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

Possible traces as a result of action interleaving.

parallel composition - action interleaving



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?

modeling interaction - handshake

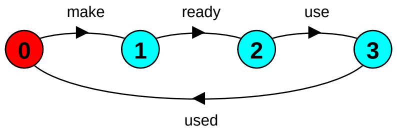
A handshake is an action acknowledged by another:

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

||MAKER_USERv2 = (MAKERv2 || USERv2).
```

3 states

3 states

3 x 3 states?



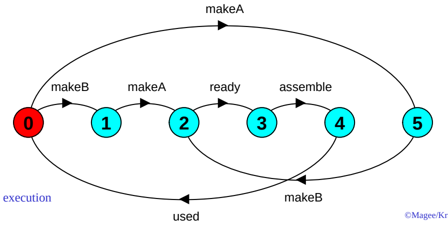
4 states

Interaction constrains the overall behaviour.

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).
```



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.

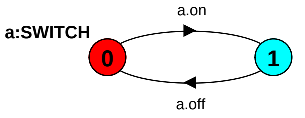
```
||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
```

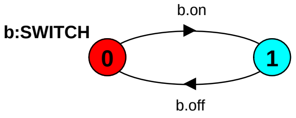
process labeling

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).
```





An array of **instances** of the switch process:

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

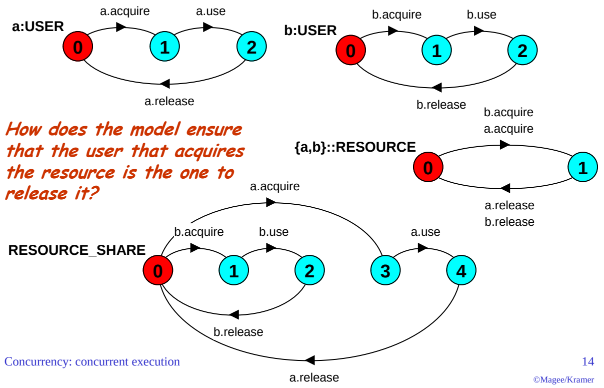
process labeling by a set of prefix labels

{a1,..,ax}::P replaces every action label n in the alphabet of P with the labels a1.n,…,ax.n. Further, every transition (n->X) in the definition of P is replaced with the transitions ({a1.n,…,ax.n} ->X).

Process prefixing is useful for modeling **shared** resources:

```
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).

||RESOURCE_SHARE = (a:USER || b:USER
                    || {a,b}::RESOURCE).
```

process prefix labels for shared resources



How does the model ensure that the user that acquires the resource is the one to release it?

action relabeling

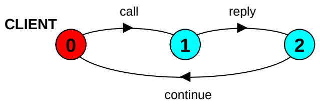
Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:
/{*newlabel_1*/*oldlabel_1*,… *newlabel_n*/*oldlabel_n*}.

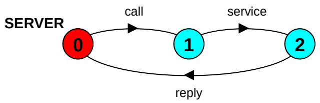
Relabeling to ensure that composed processes synchronize on particular actions.

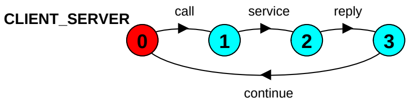
```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
```

action relabeling

```
||CLIENT_SERVER = (CLIENT || SERVER)
                  /{call/request, reply/wait}.
```







action relabeling - prefix labels

An alternative formulation of the client server system is described below using qualified or prefixed labels:

```
SERVERv2 = (accept.request
           ->service->accept.reply->SERVERv2).
CLIENTv2 = (call.request
           ->call.reply->continue->CLIENTv2).


||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
                    /{call/accept}.
```

action hiding - abstraction to reduce complexity

When applied to a process P, the hiding operator \{a1..ax} removes the action names a1..ax from the alphabet of P and makes these concealed actions "silent". These silent actions are labeled tau. Silent actions in different processes are not shared.

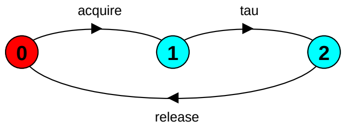
Sometimes it is more convenient to specify the set of labels to be exposed....

When applied to a process P, the interface operator @{a1..ax} hides all actions in the alphabet of P not labeled in the set a1..ax.

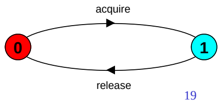
action hiding

The following definitions are equivalent:

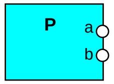
```
USER = (acquire->use->release->USER)
       \{use}.

USER = (acquire->use->release->USER)
        @{acquire,release}.
```



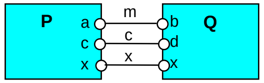
Minimization removes hidden **tau** actions to produce an LTS with equivalent observable behavior.



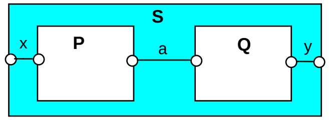
structure diagrams



Process P with alphabet {a,b}.



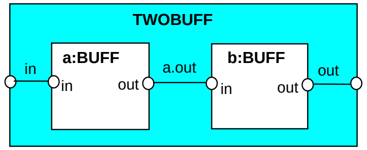
Parallel Composition (P||Q) / {m/a,m/b,c/d}



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

structure diagrams

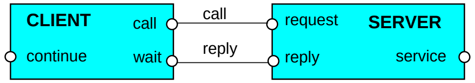
We use structure diagrams to capture the structure of a model expressed by the static combinators: *parallel composition*, *relabeling* and *hiding*.



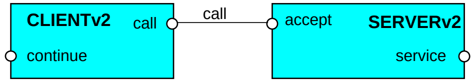
```
range T = 0..3
BUFF = (in[i:T]->out[i]->BUFF).

||TWOBUF = ?
```

structure diagrams

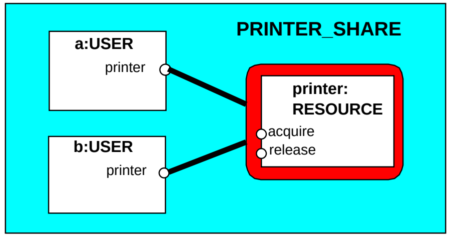
Structure diagram for **CLIENT_SERVER** ?



Structure diagram for **CLIENT_SERVERv2** ?



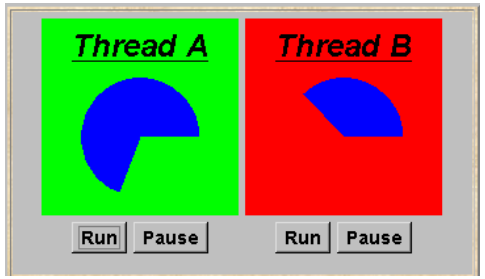
structure diagrams - resource sharing



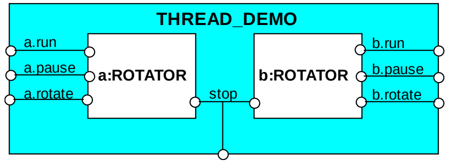
```
RESOURCE = (acquire->release->RESOURCE).
USER =     (printer.acquire->use
             ->printer.release->USER).

||PRINTER_SHARE
  = (a:USER||b:USER||{a,b}::printer:RESOURCE).
```

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

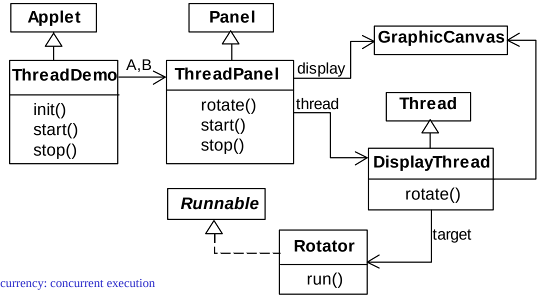


```
ROTATOR = PAUSED,
PAUSED  = (run->RUN | pause->PAUSED
          |interrupt->STOP),
RUN     = (pause->PAUSED |{run,rotate}->RUN
          |interrupt->STOP).

||THREAD_DEMO = (a:ROTATOR || b:ROTATOR)
                 /{stop/{a,b}.interrupt}.
```

Interpret run, pause, interrupt as inputs, rotate as an output.

ThreadDemo implementation in Java - class diagram

`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.



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()`.

ThreadPanel class

```
public class ThreadPanel extends Panel {

  // construct display with title and segment  color c
  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();}
}
```

`ThreadPanel` manages the display and control buttons for a thread.

Calls to `rotate()` are delegated to `DisplayThread`.

Threads are created by the `start()` method, and terminated by the `stop()` method.

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.

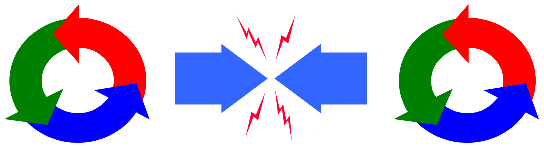
`ThreadPanel` is used extensively in later demonstration programs.

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.

Chapter 4

Shared Objects & Mutual Exclusion



Shared Objects & Mutual Exclusion

Concepts: process interference.
mutual exclusion.

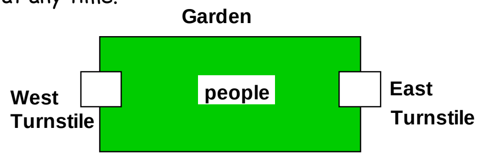
Models: model checking for interference
modeling mutual exclusion

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

4.1 Interference

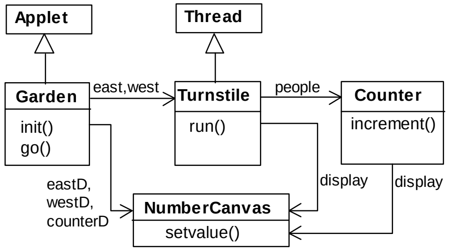
Ornamental garden problem:

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.

ornamental garden Program - class diagram



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.

ornamental garden program

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

```
    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.

Turnstile class

```
class Turnstile extends Thread {
  NumberCanvas display;
  Counter people;

  Turnstile(NumberCanvas n,Counter c)
    { display = n; people = c; }

  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) {}
  }
}
```

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

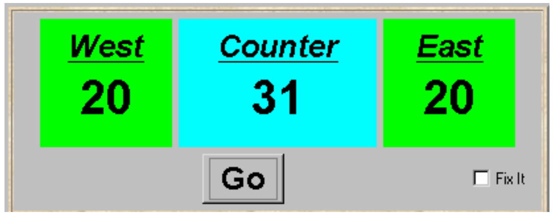
Counter class

```
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.

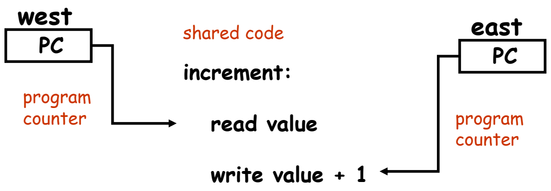
ornamental garden program - display



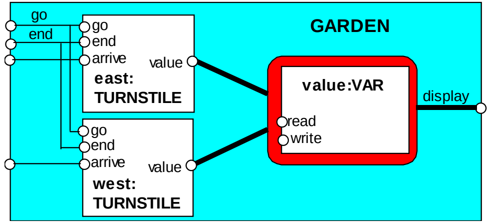
After the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. *Why?*

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



Process **VAR** models read and write access to the shared counter **value**.

Increment is modeled inside **TURNSTILE** since Java method activations are not atomic i.e. thread objects **east** and **west** may interleave their **read** and **write actions**.

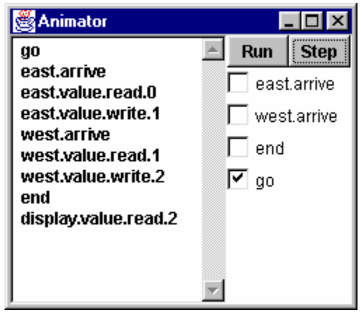
ornamental garden model

```
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),
RUN       = (arrive-> INCREMENT
            |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** ie. all actions in **VAR** must be controlled by a **TURNSTILE**.

checking for errors - animation



```
go
east.arrive
east.value.read.0
east.value.write.1
west.arrive
west.value.read.1
west.value.write.2
end
display.value.read.2
```

Scenario checking - use animation to produce a trace.

Is this trace correct?

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

Like **STOP**, **ERROR** is a predefined FSP local process (state), numbered -1 in the equivalent LTS.

ornamental garden model - checking for errors

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

Use ***LTSA*** to perform an exhaustive search for **ERROR**.

```
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**.

Interference and Mutual Exclusion

Destructive update, caused by the arbitrary interleaving of read and write actions, is termed ***interference***.

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.

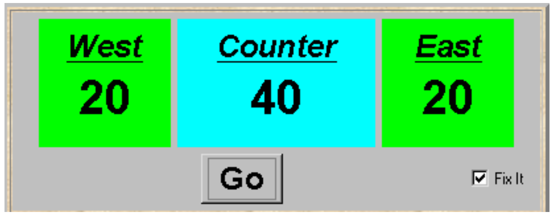
4.2 Mutual exclusion in Java

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:

```
class SynchronizedCounter extends Counter {

  SynchronizedCounter(NumberCanvas n)
     {super(n);}

   synchronized void increment() {
        super.increment();
   }
}
```

mutual exclusion - the ornamental garden



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.

Java synchronized statement

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

```
    synchronized (object) { statements }
```

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

```
synchronized(counter) {counter.increment();}
```

Why is this "less elegant"?

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

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 :

```
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).

set VarAlpha = {value.{read[T],write[T],
                acquire, release}}
```

Modify **TURNSTILE** to acquire and release the lock:

```
TURNSTILE = (go    -> RUN),
RUN       = (arrive-> INCREMENT
            |end   -> TURNSTILE),
INCREMENT = (value.acquire
             -> value.read[x:T]->value.write[x+1]
             -> value.release->RUN
            )+VarAlpha.
```

Revised ornamental garden model - checking for errors

A sample animation execution trace

```
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
```

Use TEST and ***LTSA*** to perform an exhaustive check.

Is TEST satisfied?

COUNTER: Abstraction using action hiding

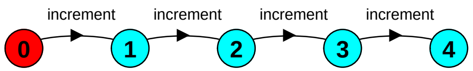
To model shared objects directly in terms of their synchronized methods, we can abstract the details by hiding.

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

```
const N = 4
range T = 0..N

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

LOCK = (acquire->release->LOCK).

INCREMENT = (acquire->read[x:T]
             -> (when (x<N) write[x+1]
                 ->release->increment->INCREMENT
                )
             )+{read[T],write[T]}.

||COUNTER = (INCREMENT||LOCK||VAR)@{increment}.
```

COUNTER: Abstraction using action hiding

Minimized LTS:



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

```
COUNTER = COUNTER[0]
COUNTER[v:T] = (when (v<N) increment -> COUNTER[v+1]).
```

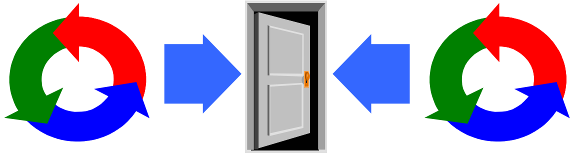
This therefore exhibits "equivalent" behavior i.e. has the same observable behavior.

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).**

Chapter 5

Monitors & Condition Synchronization



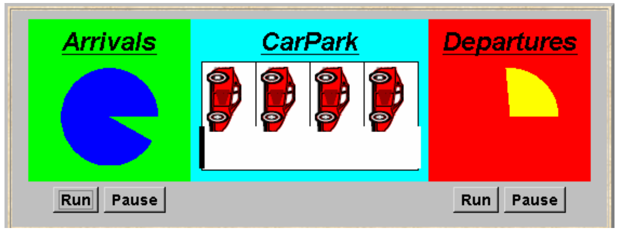
monitors & condition synchronization

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 enter when the carpark is not full and does not permit cars to leave when there are no cars in the carpark. Car arrival and departure are simulated by separate threads.

carpark model

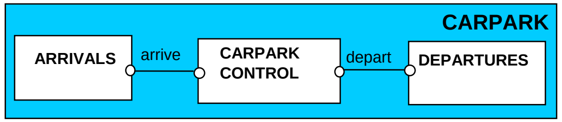
- Events or actions of interest?

  arrive and depart

- Identify processes.

  arrivals, departures and carpark control

- Define each process and interactions (structure).



carpark model

```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 |when(i<N) depart->SPACES[i+1]
                 ).

ARRIVALS   = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

||CARPARK =
      (ARRIVALS||CARPARKCONTROL(4)||DEPARTURES).
```

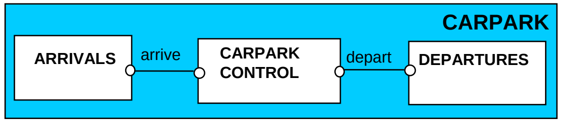
Guarded actions are used to control `arrive` and `depart`.

LTS?

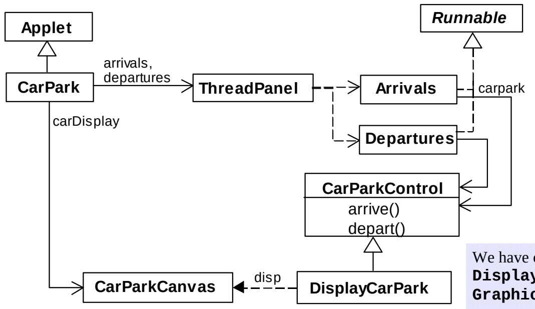
carpark program

- 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?



carpark program - class diagram



We have omitted **DisplayThread** and **GraphicCanvas** threads managed by **ThreadPanel**.

carpark program

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

Instances of these are created by the **start()** method of the **CarPark** applet **:**

```
public void start() {
  CarParkControl c =
     new DisplayCarPark(carDisplay,Places);
  arrivals.start(new Arrivals(c));
  departures.start(new Departures(c));
}
```

carpark program - Arrivals and Departures threads

```
class Arrivals implements Runnable {
  CarParkControl carpark;

  Arrivals(CarParkControl c) {carpark = c;}

  public void run() {
    try {
      while(true) {
        ThreadPanel.rotate(330);
        carpark.arrive();
        ThreadPanel.rotate(30);
      }
    } catch (InterruptedException e){}
  }
}
```

Similarly **Departures** which calls **carpark.depart()**.

How do we implement the control of **CarParkControl**?

Carpark program - CarParkControl monitor

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() {
    …  --spaces; …
    }

  synchronized void depart() {
    … ++spaces; …
    }
}
```

mutual exclusion by synch methods

condition synchronization?

block if full? (spaces==0)

block if empty? (spaces==N)

condition synchronization in Java

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

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.

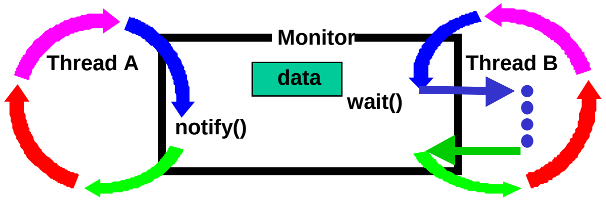
public final void wait()
throws InterruptedException

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.

condition synchronization in Java

We refer to a thread ***entering*** a monitor when it acquires the mutual exclusion lock associated with the monitor and ***exiting*** the monitor when it releases the lock.

Wait() - causes the thread to exit the monitor, permitting other threads to enter the monitor.



condition synchronization in Java

```
FSP:   when cond act -> NEWSTAT
```

```
Java:  public synchronized void act()
               throws InterruptedException
       {
        while (!cond) wait();
        // modify monitor data
        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.

CarParkControl - condition synchronization

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notify();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces;
    notify();
  }
}
```

Why is it safe to use **notify()** *here rather than* **notifyAll()***?*

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()**.

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.

```
down(s): if s >0 then
             decrement s
         else
             block execution of  the calling process

up(s):   if processes blocked on s then
             awaken one of them
         else
             increment s
```

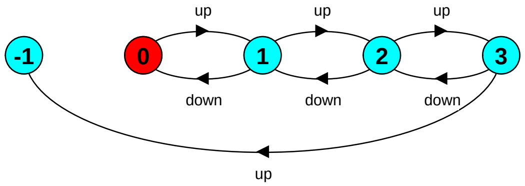
modeling semaphores

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.

```
const Max = 3
range Int = 0..Max

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

LTS?

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

semaphore demo - model

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

```
LOOP = (mutex.down->critical->mutex.up->LOOP).

||SEMADEMO = (p[1..3]:LOOP
             ||{p[1..3]}::mutex:SEMAPHORE(1)).
```

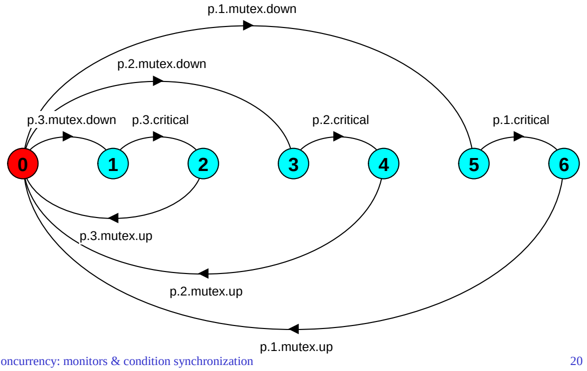
For mutual exclusion, the semaphore initial value is 1. ***Why?***

Is the **ERROR** *state reachable for* `SEMADEMO`*?*

Is a binary semaphore sufficient (i.e. `Max=1`*) ?*

LTS?

semaphore demo - model



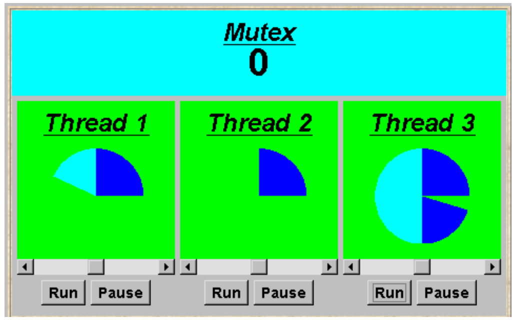
semaphores in Java

Semaphores are passive objects, therefore implemented as **monitors**.

(In practice, semaphores are a low-level mechanism often used in implementing the higher-level monitor construct.)

```
public class Semaphore {
  private int value;

  public Semaphore (int initial)
    {value = initial;}

  synchronized public void up() {
     ++value;
     notify();
  }

  synchronized public void down()
      throws InterruptedException {
    while (value== 0) wait();
    --value;
  }
}
```

SEMADEMO display



current semaphore value

thread 1 is executing critical actions.

thread 2 is blocked waiting.

thread 3 is executing non-critical actions.

SEMADEMO

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.

SEMADEMO program - revised ThreadPanel class

```
public class ThreadPanel extends Panel {
  // construct display with title and rotating arc color c
  public ThreadPanel(String title, Color c) {…}
  // hasSlider == true creates panel with slider
  public ThreadPanel
  (String title, Color c, boolean hasSlider) {…}
  // rotate display of currently running thread 6 degrees
  // return false when in initial color, return true when in second color
  public static boolean rotate()
         throws InterruptedException {…}
  // rotate display of currently running thread by degrees
  public static void rotate(int degrees)
         throws InterruptedException {…}
  // create a new thread with target r and start it running
  public void start(Runnable r) {…}
  // stop the thread using Thread.interrupt()
  public void stop() {…}
}
```

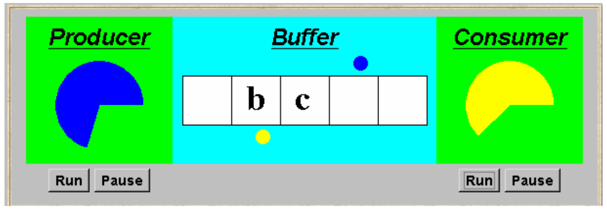
SEMADEMO program - MutexLoop

```
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){}
  }
}
```

Threads and semaphore are created by the applet **start()** method.

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

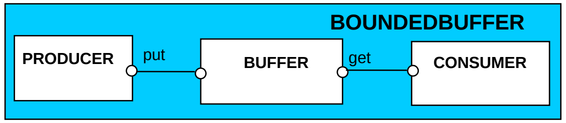
5.3 Bounded Buffer



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*.

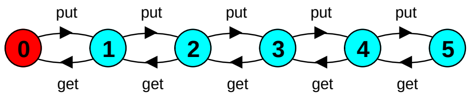
(see car park example)

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:



bounded buffer - a data-independent model

```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
    = (when (i<N) put->COUNT[i+1]
      |when (i>0) get->COUNT[i-1]
      ).

PRODUCER = (put->PRODUCER).
CONSUMER = (get->CONSUMER).

||BOUNDEDBUFFER =
(PRODUCER||BUFFER(5)||CONSUMER).
```

bounded buffer program - buffer monitor

```
public interface Buffer {…}

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

We separate the interface to permit an alternative implementation later.

bounded buffer program - producer process

```
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){}
  }
}
```

Similarly **Consumer** which calls **buf.get()**.

5.4 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.

```
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);
  }
…
}
```

nested monitors - bounded buffer program

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

```
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}.
```

Does this behave as desired?

nested monitors - bounded buffer model

LTSA analysis predicts a possible **DEADLOCK**:

```
Composing
 potential DEADLOCK
States Composed: 28 Transitions: 32 in 60ms
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 - 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.

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

nested monitors - revised bounded buffer model

```
BUFFER =  (put -> BUFFER
          |get -> BUFFER
          ).

PRODUCER =(empty.down->put->full.up->PRODUCER).
CONSUMER =(full.down->get->empty.up->CONSUMER).
```

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?

Minimized LTS?

5.5 Monitor invariants

An **invariant** for a monitor is an assertion concerning the variables it encapsulates. This assertion must hold whenever there is no thread executing inside the monitor i.e. on thread **entry** to and **exit** from a monitor .

CarParkControl Invariant: $\mathbf{0 \leq \mathit{spaces} \leq \mathit{N}}$

Semaphore Invariant: $\mathbf{0 \leq \mathit{value}}$

Buffer Invariant: $\mathbf{0 \leq \mathit{count} \leq \mathit{size}}$
and $\mathbf{0 \leq \mathit{in} < \mathit{size}}$
and $\mathbf{0 \leq \mathit{out} < \mathit{size}}$
and $\mathbf{\mathit{in} = (\mathit{out} + \mathit{count})}$ **modulo** ***size***

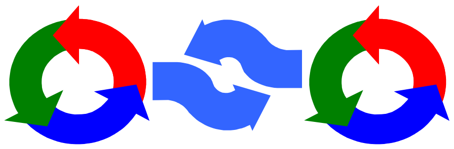
Invariants can be helpful in reasoning about correctness of monitors using a logical *proof-based* approach. Generally we prefer to use a *model-based* approach amenable to mechanical checking .

Summary

- Concepts
  - **monitors: encapsulated data + access procedures**
    mutual exclusion + condition synchronization
  - **nested monitors**
- Model
  - **guarded actions**
- Practice
  - **private data and synchronized methods in Java**
  - **wait(), notify() and notifyAll() for condition synchronization**
  - **single thread active in the monitor at a time**

Chapter 6

Deadlock



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.

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.

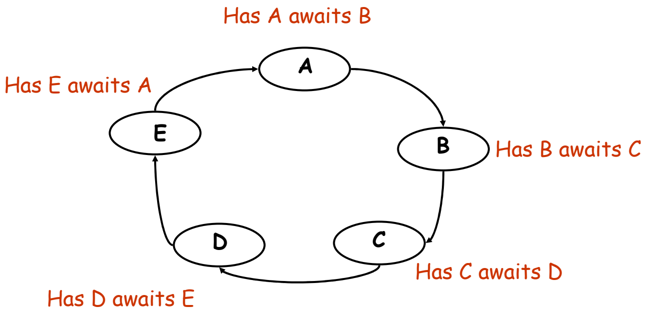
- No pre-emption:

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.

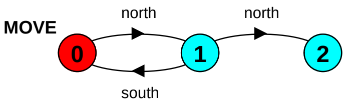
Wait-for cycle



6.1 Deadlock analysis - primitive processes

- deadlocked state is one with no outgoing transitions
- in FSP: STOP process

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



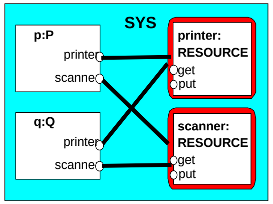
- animation to produce a trace.
- analysis using *LTSA*:

(shortest trace to STOP)

```
Trace to DEADLOCK:
	north
	north
```

deadlock analysis - parallel composition

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



```
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
     ).
```

Deadlock Trace?

Avoidance?

deadlock analysis - avoidance

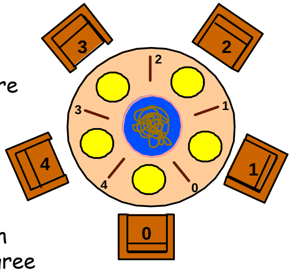
- acquire resources in the same order?
- Timeout:

```
P          = (printer.get-> GETSCANNER),
GETSCANNER = (scanner.get->copy->printer.put
                          ->scanner.put->P
             |timeout -> printer.put->P
             ).
Q          = (scanner.get-> GETPRINTER),
GETPRINTER = (printer.get->copy->printer.put
                          ->scanner.put->Q
             |timeout -> scanner.put->Q
             ).
```

Deadlock? *Progress?*

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.

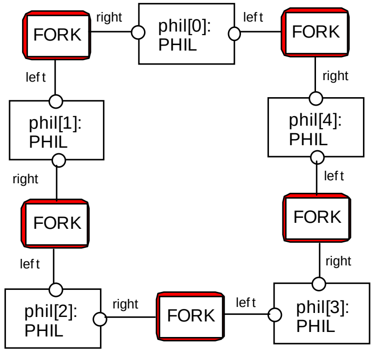


One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

Dining Philosophers - model structure diagram

Each FORK is a shared resource with actions **get** and **put**.

When hungry, each PHIL must first get his right and left forks before he can start eating.



Dining Philosophers - model

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

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 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
```

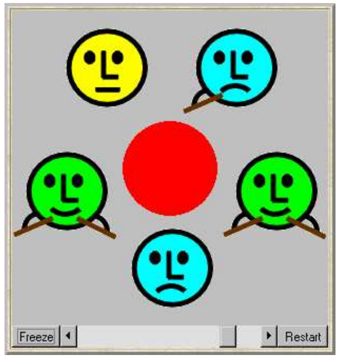
This is the situation where all the philosophers become hungry at the same time, sit down at the table and each philosopher picks up the fork to his right.

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!

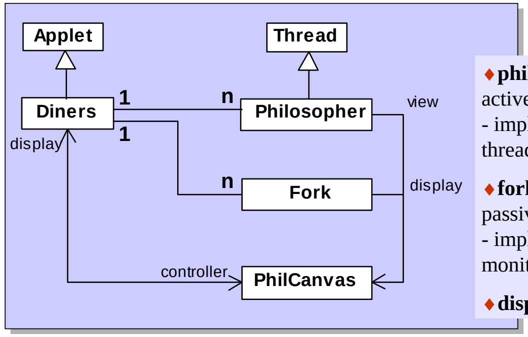
Dining Philosophers

Deadlock is easily detected in our model.

How easy is it to detect a potential deadlock in an implementation?



Dining Philosophers - implementation in Java



♦ **philosophers**: active entities - implement as threads

♦ **forks**: shared passive entities - implement as monitors

♦ **display**

Dining Philosophers - Fork monitor

```
class Fork {
  private boolean taken=false;
  private PhilCanvas display;
  private int identity;

  Fork(PhilCanvas disp, int id)
    { display = disp; identity = id;}

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

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

taken encodes the state of the fork

Dining Philosophers - Philosopher implementation

```
class Philosopher extends Thread {
  ...
  public void run() {
    try {
      while (true) {                          // thinking
        view.setPhil(identity,view.THINKING);
        sleep(controller.sleepTime());  // hungry
        view.setPhil(identity,view.HUNGRY);
        right.get();                           // gotright 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.InterruptedException e){}
  }
}
```

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

Dining Philosophers - implementation in Java

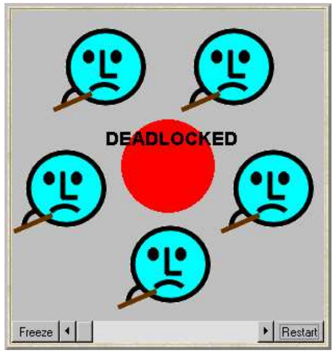
Code to create the philosopher threads and fork monitors:

```
for (int i =0; i<N; ++i)
  fork[i] = new Fork(display,i);
for (int i =0; i<N; ++i){
  phil[i] =
    new Philosopher
        (this,i,fork[(i-1+N)%N],fork[i]);
  phil[i].start();
}
```

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.



Deadlock-free Philosophers

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

Introduce an *asymmetry* into our definition of philosophers.

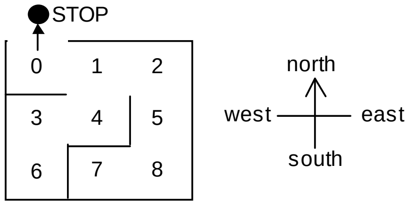
Use the identity I of a philosopher to make even numbered philosophers get their left forks first, odd their right first.

Other strategies?

```
PHIL(I=0)
  = (when (I%2==0) sitdown
       ->left.get->right.get
       ->eat
       ->left.put->right.put
       ->arise->PHIL
    |when (I%2==1) sitdown
       ->right.get->left.get
       ->eat
       ->left.put->right.put
       ->arise->PHIL
    ).
```

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**.

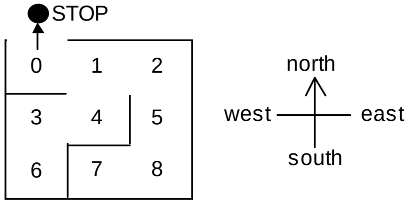
Each position can be modelled by the moves that it permits. The **MAZE** parameter gives the starting position.

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

Maze example - shortest path to “deadlock”

```
||GETOUT = MAZE(7).
```

Shortest path escape trace from position 7 ?



```
Trace to
DEADLOCK:
    east
    north
    north
    west
    west
    north
```

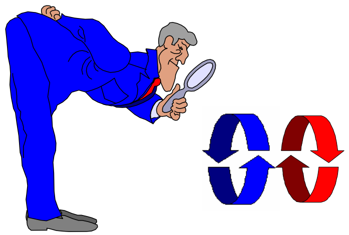
Summary

- Concepts
  - **deadlock: no futher progress**
  - **four necessary and sufficient conditions:**
    - serially reusable resources
    - incremental acquisition
    - no preemption
    - wait-for cycle
- Models
  - **no eligable actions (analysis gives shortest path trace)**
- Practice
  - **blocked threads**

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

Chapter 7

Safety & Liveness Properties



safety & liveness properties

Concepts: properties: true for every possible execution
safety: nothing bad happens
liveness: something good *eventually* happens

Models: safety: no reachable ERROR/STOP state
progress: an action is *eventually* executed
fair choice and action priority

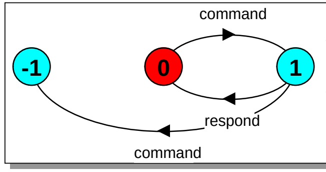
Practice: threads and monitors

Aim: property satisfaction.

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



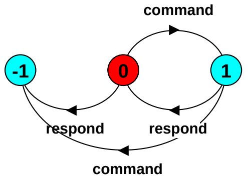
```
ACTUATOR
   =(command->ACTION),
ACTION
   =(respond->ACTUATOR
    |command->ERROR).
```

- analysis using LTSA:
  (shortest trace)

```
Trace to ERROR:
  command
  command
```

Safety - property specification

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



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

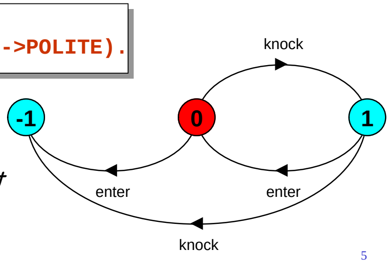
- analysis using ***LTSA*** as before.

Safety properties

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

Traces: **knock→enter** ☑ **enter** ☒
knock→knock

```
property POLITE
  = (knock->enter->POLITE).
```



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

Safety properties

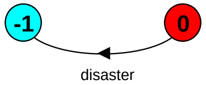
Safety **property P** defines a deterministic process that asserts that any trace including actions in the alphabet of **P**, is accepted by **P**.

Thus, if **P** is composed with **S**, then traces of actions in the alphabet of **S** ∩ alphabet of **P** must also be valid traces of **P**, otherwise **ERROR** is reachable.

Transparency of 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 behavior. However, if a behavior can occur which violates the safety property, then **ERROR** is reachable. Properties must be deterministic to be transparent.*

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.

Safety - mutual exclusion

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

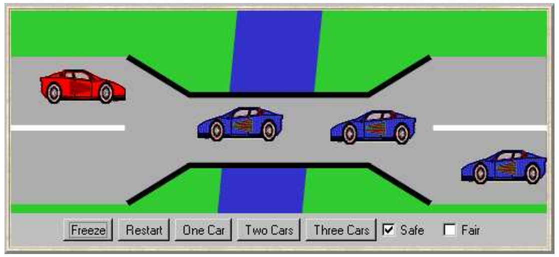
How do we check that this does indeed ensure mutual exclusion in the critical section?

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

*Check safety using **LTSA**.*

What happens if semaphore is initialized to 2?

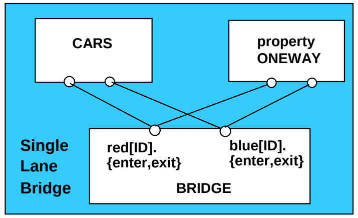
7.2 Single Lane Bridge problem



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.

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

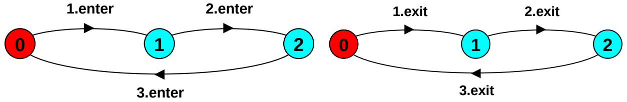
```
const N = 3        // number of each type of car
range T = 0..N     // type of car count
range ID= 1..N     // car identities

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 - CONVOY model

```
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 = ([ID]: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 - BRIDGE model

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.

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

We now specify a **safety** property 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.

```
property ONEWAY =(red[ID].enter   -> RED[1]
                 |blue.[ID].enter -> BLUE[1]
                 ),
RED[i:ID] = (red[ID].enter -> RED[i+1]
            |when(i==1)red[ID].exit  -> ONEWAY
            |when(i>1) red[ID].exit  -> RED[i-1]
            ),        //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 - model analysis

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

Is the safety property **ONEWAY** *violated?*

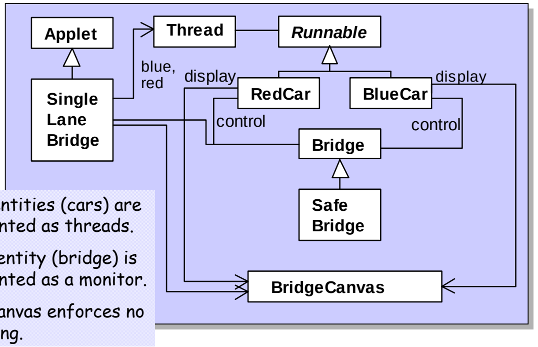
```
No deadlocks/errors
```

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

Without the **BRIDGE** *contraints, is the safety property* **ONEWAY** *violated?*

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

Single Lane Bridge - implementation in Java



Active entities (cars) are implemented as threads.

Passive entity (bridge) is implemented as a monitor.

BridgeCanvas enforces no overtaking.

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.

```
class BridgeCanvas extends Canvas {

  public void init(int ncars) {…}    //set number of cars

  //move red car with the identity i  a step
  //returns true for the period from just before,until just after car on bridge
  public boolean moveRed(int i)
         throws InterruptedException{…}

  //move blue car with the identity i  a step
  //returns true for the period from just before,until just after car on bridge
  public boolean moveBlue(int i)
         throws InterruptedException{…}

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

Single Lane Bridge - RedCar

```
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) {
        while (!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) {}
  }
}
```

Similarly for the **BlueCar**

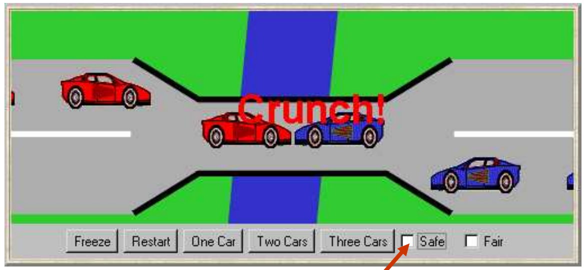
Single Lane Bridge - class `Bridge`

```
class Bridge {
  synchronized void redEnter()
     throws InterruptedException {}
  synchronized void redExit()  {}
  synchronized void blueEnter()
     throws InterruptedException {}
  synchronized void blueExit() {}
}
```

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

Result………… ?

Single Lane Bridge



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

Single Lane Bridge - `SafeBridge`

```
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()
      throws InterruptedException {
    while (nblue>0) wait();
    ++nred;
 }
 synchronized void redExit(){
    --nred;
    if (nred==0)notifyAll();
 }
```

This is a direct translation from the **BRIDGE** model.

Single Lane Bridge - `SafeBridge`

```
     synchronized void blueEnter()
          throws InterruptedException {
        while (nred>0) wait();
        ++nblue;
      }
      synchronized void blueExit(){
        --nblue;
        if (nblue==0)notifyAll();
      }
   }
```

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.*

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?*

ie. make **PROGRESS?**

A progress property asserts that it is *always* the case that an action is *eventually* executed. Progress is the opposite of *starvation*, the name given to a concurrent programming situation in which an action is never executed.

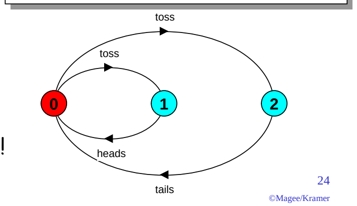
Progress properties - fair choice

Fair Choice: If a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.

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** !

```
COIN =(toss->heads->COIN
      |toss->tails->COIN).
```



Progress properties

`progress P = {a1,a2..an}` defines a progress property **P** which asserts that in an infinite execution of a target system, at least **one** of the actions **`a1,a2..an`** will be executed infinitely often.

⇨ COIN system: **`progress HEADS = {heads}`** ☑

`progress TAILS = {tails}` ☑

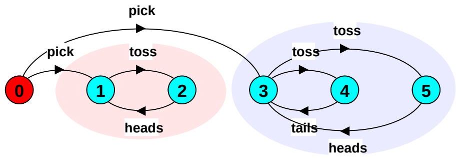
LTSA check progress:

```
No progress violations detected.
```

Progress properties

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

a trick coin and a regular coin……

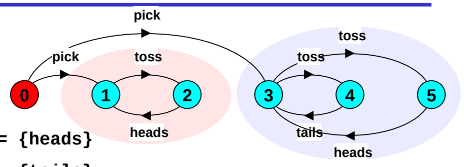


```
TWOCOIN = (pick->COIN|pick->TRICK),
TRICK   = (toss->heads->TRICK),
COIN    = (toss->heads->COIN|toss->tails->COIN).
```

⇨ TWOCOIN: **`progress HEADS = {heads}`** ☑

`progress TAILS = {tails}` ✗

Progress properties



`progress HEADS = {heads}`

`progress TAILS = {tails}`

LTSA check progress ⇨

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

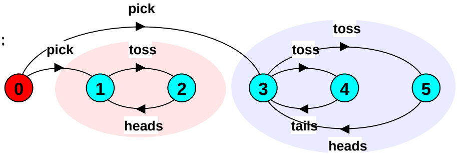
`progress HEADSorTails = {heads,tails}` ☑

Progress analysis

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} and {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!

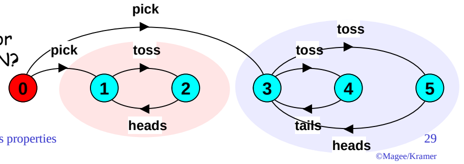
Progress analysis

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

⇨ **`progress TAILS = {tails}`** **in {1,2}**

Default: given fair choice, for *every* action in the alphabet of the target system, that action will be executed infinitely often. This is equivalent to specifying a separate progress property for every action.

⇨ Default analysis for TWOCOIN?



Progress analysis

Default analysis for TWOCOIN: separate progress property for every action.

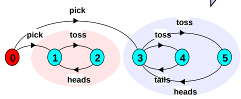
⇨

```
Progress violation for actions:
{pick}
Path to terminal set of states:
     pick
Actions in terminal set:
{toss, heads, tails}
```

and

⇨

```
Progress violation for actions:
{pick, tails}
Path to terminal set of states:
     pick
Actions in terminal set:
{toss, heads}
```

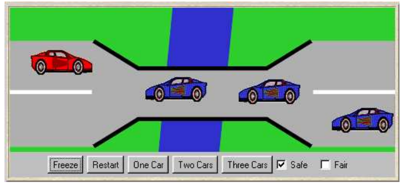


If the default holds, then every other progress property holds i.e. every action is executed infinitely often and system consists of a single terminal set of states.

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?



```
progress BLUECROSS = {blue[ID].enter}
progress REDCROSS =  {red[ID].enter}
No progress violations detected.
```

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 ("<<")

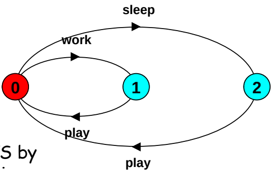
`||C = (P||Q)<<{a1,…,an}` specifies a composition in which the actions **`a1,..,an`** have higher priority than any other action in the alphabet of **`P||Q`** including the silent action **`tau`**. *In any choice in this system which has one or more of the actions* ***`a1,..,an`*** *labeling a transition, the transitions labeled with lower priority actions are discarded.*

Low Priority (">>")

`||C = (P||Q)>>{a1,…,an}` specifies a composition in which the actions **`a1,..,an`** have lower priority than any other action in the alphabet of **`P||Q`** including the silent action **`tau`**. *In any choice in this system which has one or more transitions not labeled by* ***`a1,..,an`****, the transitions labeled by* ***`a1,..,an`*** *are discarded.*

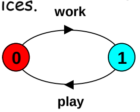
Progress - action priority

```
NORMAL =(work->play->NORMAL
        |sleep->play->NORMAL).
```

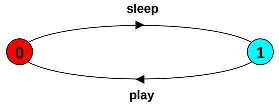


Action priority simplifies the resulting LTS by discarding lower priority actions from choices.

```
||HIGH =(NORMAL)<<{work}.
```



```
||LOW  =(NORMAL)>>{work}.
```



7.4 Congested single lane bridge

```
progress BLUECROSS = {blue[ID].enter}
progress REDCROSS =  {red[ID].enter}
```

`BLUECROSS` - eventually one of the blue cars will be able to enter

`REDCROSS` - eventually one of the red cars will be able to enter

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.*

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

Progress Analysis ? LTS?

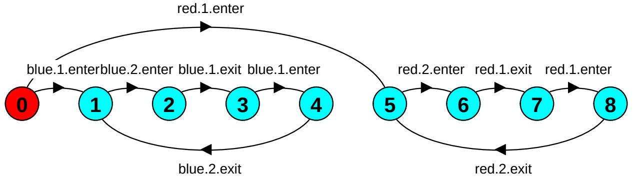
congested single lane bridge model

```
Progress violation: BLUECROSS
Path to terminal set of states:
     red.1.enter
     red.2.enter
Actions in terminal set:
{red.1.enter, red.1.exit, red.2.enter,
red.2.exit, red.3.enter, red.3.exit}

Progress violation: REDCROSS
Path to terminal set of states:
     blue.1.enter
     blue.2.enter
Actions in terminal set:
{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.

congested single lane bridge model

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


red.1.enter
blue.1.enterblue.2.enter blue.1.exit blue.1.enter
red.2.enter red.1.exit red.1.enter
0
1
2
3
4
5
6
7
8
blue.2.exit
red.2.exit

Will the results be the same if we model congestion by giving car entry to the bridge high priority?

Can congestion occur if there is only one car moving in each direction?

Progress - revised single lane bridge model

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

Modify **CAR**:

```
CAR = (request->enter->exit->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 - revised single lane bridge model

```
/* nr– number of red cars on the bridge  wr  – number of red cars waiting to enter
   nb– number of blue cars on the bridge wb  – number of blue cars waiting to enter
*/
BRIDGE = BRIDGE[0][0][0][0],
BRIDGE[nr:T][nb:T][wr:T][wb:T] =
  (red[ID].request  -> BRIDGE[nr][nb][wr+1][wb]
  |when (nb==0 && wb==0)
     red[ID].enter  -> BRIDGE[nr+1][nb][wr-1][wb]
  |red[ID].exit     -> BRIDGE[nr-1][nb][wr][wb]
  |blue[ID].request -> BRIDGE[nr][nb][wr][wb+1]
  |when (nr==0 && wr==0)
     blue[ID].enter -> BRIDGE[nr][nb+1][wr][wb-1]
  |blue[ID].exit    -> BRIDGE[nr][nb-1][wr][wb]
  ).
```

OK now?

Progress - analysis of revised single lane bridge model

```
    Trace to DEADLOCK:
        red.1.request
        red.2.request
        red.3.request
        blue.1.request
        blue.2.request
        blue.3.request
```

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.

Progress - 2 nd revision of single lane bridge model

➡ ***Analysis ?***

```
const True = 1
const False = 0
range B = False..True
/*  bt - true indicates blue turn,   false indicates red turn */
BRIDGE = BRIDGE[0][0][0][0][True],
BRIDGE[nr:T][nb:T][wr:T][wb:T][bt:B] =
  (red[ID].request  -> BRIDGE[nr][nb][wr+1][wb][bt]
  |when (nb==0 && (wb==0||!bt))
     red[ID].enter  -> BRIDGE[nr+1][nb][wr-1][wb][bt]
  |red[ID].exit     -> BRIDGE[nr-1][nb][wr][wb][True]
  |blue[ID].request -> BRIDGE[nr][nb][wr][wb+1][bt]
  |when (nr==0 && (wr==0||bt))
     blue[ID].enter -> BRIDGE[nr][nb+1][wr][wb-1][bt]
  |blue[ID].exit    -> BRIDGE[nr][nb-1][wr][wb][False]
  ).
```

Revised single lane bridge implementation - FairBridge

```
class FairBridge extends Bridge {

  private int nred  = 0; //count of red cars on the bridge
  private int nblue = 0; //count of blue cars on the bridge
  private int waitblue = 0;   //count of waiting blue cars
  private int waitred = 0;    //count of waiting red cars
  private boolean blueturn = true;

  synchronized void redEnter()
      throws InterruptedException {
    ++waitred;
    while (nblue>0||(waitblue>0 && blueturn)) wait();
    --waitred;
    ++nred;
  }

  synchronized void redExit(){
    --nred;
    blueturn = true;
    if (nred==0)notifyAll();
  }
```

This is a direct translation from the model.

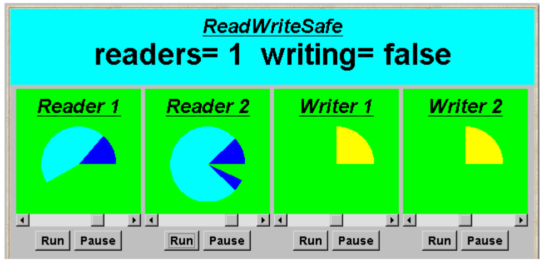
Revised single lane bridge implementation - FairBridge

```
  synchronized void blueEnter(){
      throws InterruptedException {
    ++waitblue;
    while (nred>0||(waitred>0 && !blueturn)) wait();
    --waitblue;
    ++nblue;
  }

  synchronized void blueExit(){
    --nblue;
    blueturn = false;
    if (nblue==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.

7.5 Readers and Writers



Light blue indicates database access.

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

readers/writers model

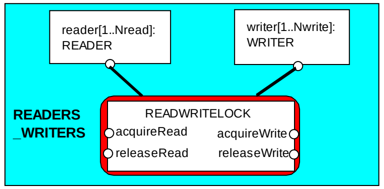
- Events or actions of interest?

  acquireRead, releaseRead, acquireWrite, releaseWrite
- Identify processes.

  Readers, Writers & the RW_Lock
- Identify properties.

  RW_Safe

  RW_Progress
- Define each process and interactions (structure).



readers/writers model - READER & WRITER

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

READER = (acquireRead->examine->releaseRead->READER)
  + Actions
  \ {examine}.

WRITER = (acquireWrite->modify->releaseWrite->WRITER)
  + Actions
  \ {modify}.
```

Alphabet extension is used to ensure that the other access actions cannot occur freely for any prefixed instance of the process (as before).

Action hiding is used as actions **examine** and **modify** are not relevant for access synchronisation.

readers/writers model - RW_LOCK

```
const False = 0   const True  = 1
range Bool  = False..True
const Nread = 2             // Maximum readers
const Nwrite= 2             // Maximum writers

RW_LOCK = RW[0][False],
RW[readers:0..Nread][writing:Bool] =
     (when (!writing)
          acquireRead  -> RW[readers+1][writing]
     |releaseRead      -> RW[readers-1][writing]
     |when (readers==0 && !writing)
          acquireWrite -> RW[readers][True]
     |releaseWrite     -> RW[readers][False]
     ).
```

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

readers/writers model - safety

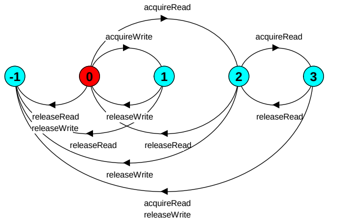
```
property SAFE_RW
  = (acquireRead  -> READING[1]
    |acquireWrite -> WRITING
    ),
READING[i:1..Nread]
  = (acquireRead -> READING[i+1]
    |when(i>1) releaseRead  -> READING[i-1]
    |when(i==1) releaseRead -> SAFE_RW
    ),
WRITING = (releaseWrite -> SAFE_RW).
```

We can check that RW_LOCK satisfies the safety property……

```
||READWRITELOCK = (RW_LOCK || SAFE_RW).
```

➡ *Safety Analysis ? LTS?*

readers/writers model



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
   = (reader[1..Nread] :READER
     || writer[1..Nwrite]:WRITER
     ||{reader[1..Nread],
        writer[1..Nwrite]}::READWRITELOCK).
```

➡ *Safety and Progress Analysis ?*

readers/writers - 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.

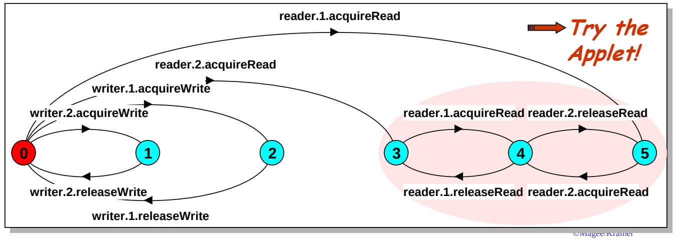
```
||RW_PROGRESS = READERS_WRITERS
              >>{reader[1..Nread].releaseRead,
                 writer[1..Nread].releaseWrite}.
```

Progress Analysis ? LTS?

readers/writers model - progress

```
Progress violation: WRITE
Path to terminal set of states:
     reader.1.acquireRead
Actions in terminal set:
{reader.1.acquireRead, reader.1.releaseRead,
 reader.2.acquireRead, reader.2.releaseRead}
```

Writer starvation: The number of readers never drops to zero.



Try the Applet!

readers/writers implementation - monitor interface

We concentrate on the monitor implementation:

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

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

Firstly, the safe **READWRITELOCK.**

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();
  }
```

Unblock a single writer *when no more readers.*

readers/writers implementation - ReadWriteSafe

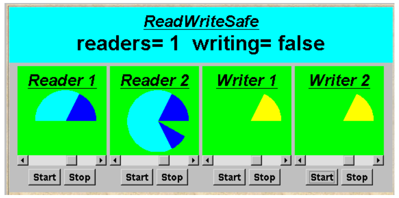
```
  public synchronized void acquireWrite()
               throws InterruptedException {
    while (readers>0 || writing) wait();
    writing = true;
  }

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

Unblock all readers

However, this monitor implementation suffers from the WRITE progress problem: possible ***writer starvation*** if the number of readers never drops to zero.

Solution?

readers/writers - writer priority


ReadWriteSafe

readers= 1 writing= false

Strategy: *Block readers if there is a writer waiting.*

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

WRITER =(requestWrite->acquireWrite->modify
               ->releaseWrite->WRITER
        )+Actions\{modify}.
```

readers/writers model - writer priority

```
RW_LOCK = RW[0][False][0],
RW[readers:0..Nread][writing:Bool][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 ?*

readers/writers model - writer priority

property RW_SAFE:

```
No deadlocks/errors
```

progress READ and WRITE:

```
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 implementation - ReadWritePriority

```
class ReadWritePriority implements ReadWrite{
  private int readers =0;
  private boolean writing = false;
  private int waitingW = 0; // no of waiting Writers.

  public synchronized void acquireRead()
             throws InterruptedException {
    while (writing || waitingW>0) wait();
     ++readers;
  }

  public synchronized void releaseRead() {
    --readers;
    if (readers==0) notify();
  }
```

readers/writers implementation - ReadWritePriority

```
synchronized public void acquireWrite() {
    ++waitingW;
    while (readers>0 || writing) try{ wait();}
          catch(InterruptedException e){}
    --waitingW;
    writing = true;
  }

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

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

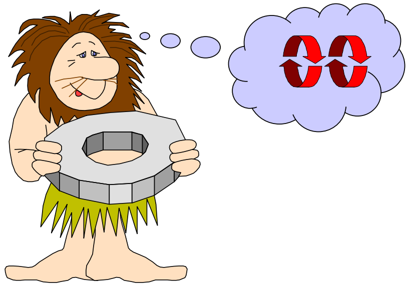
Summary

- Concepts
  - **properties**: true for every possible execution
  - **safety**: nothing bad happens
  - **liveness**: something good *eventually* happens
- Models
  - **safety**: no reachable ERROR/STOP state
    compose safety properties at appropriate stages
  - **progress**: an action is eventually executed
    fair choice and action priority
    apply progress check on the final target system model
- Practice
  - threads and monitors

Aim: property satisfaction

Chapter 8

Model-Based Design



Model-based Design

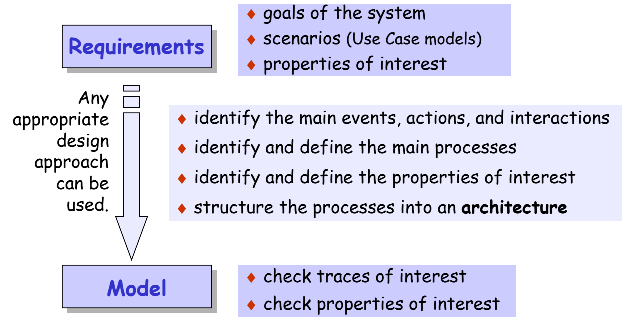
Concepts: **design process:**
requirements to **models** to implementations

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

Aim: rigorous design process.

8.1 from requirements to models



Requirements

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

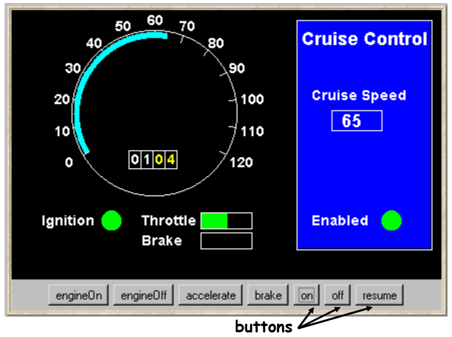
Any appropriate design approach can be used.

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

Model

- check traces of interest
- check properties of interest

a Cruise Control System - requirements

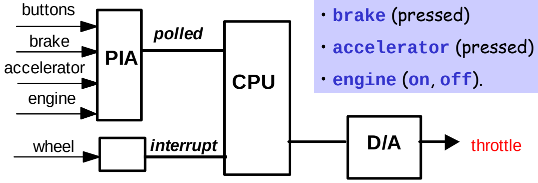


When the car ignition is switched on and the **on** button is pressed, the current speed is recorded and the system is enabled: *it maintains the speed of the car at the recorded setting.*

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

a Cruise Control System - hardware

Parallel Interface Adapter (PIA) is polled every 100msec. It records the actions of the sensors:
- buttons (**on**, **off**, **resume**)
- **brake** (pressed)
- **accelerator** (pressed)
- **engine** (**on**, **off**).

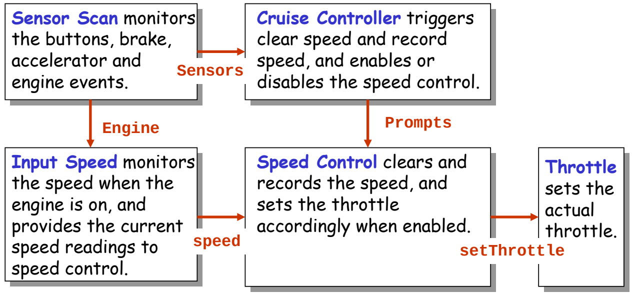


Wheel revolution sensor generates interrupts to enable the car **speed** to be calculated.

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

model - outline design

- outline processes and interactions.



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

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

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

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

Throttle sets the actual throttle.

model -design

- Main events, actions and interactions.

```
on, off, resume, brake, accelerator     }  Sensors
engine on, engine off,
speed, setThrottle
clearSpeed,recordSpeed,                 }  Prompts
enableControl,disableControl
```

- Identify main processes.

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

- Identify main properties.

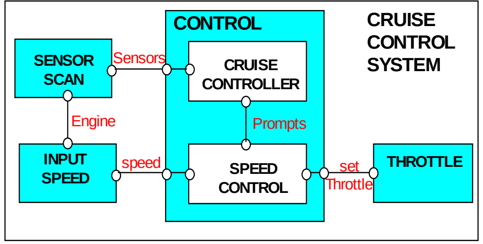
  safety - disabled when off, brake or accelerator pressed.

- Define and structure each process.

model - structure, actions and interactions

The CONTROL system is structured as two processes.

The main actions and interactions are as shown.



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

model elaboration - process definitions

```
SENSORSCAN = ({Sensors} -> SENSORSCAN).
      // monitor speed when engine on
INPUTSPEED = (engineOn -> CHECKSPEED),
CHECKSPEED = (speed -> CHECKSPEED
             |engineOff -> INPUTSPEED
             ).
      // zoom when throttle set
THROTTLE =(setThrottle -> zoom -> THROTTLE).
      // perform speed control when enabled
SPEEDCONTROL = DISABLED,
DISABLED =({speed,clearSpeed,recordSpeed}->DISABLED
           | enableControl -> ENABLED
           ),
ENABLED = ( speed -> setThrottle -> ENABLED
           |{recordSpeed,enableControl} -> ENABLED
           | disableControl -> DISABLED
           ).
```

model elaboration - process definitions

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

model - CONTROL subsystem

```
||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 - 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).
```

LTS?

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

Is CRUISESAFETY violated?

model analysis

We can now compose the whole system:

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

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

Deadlock?
Safety?

```
No deadlocks/errors
```

Progress?

model - Progress properties

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

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

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

model - Progress properties

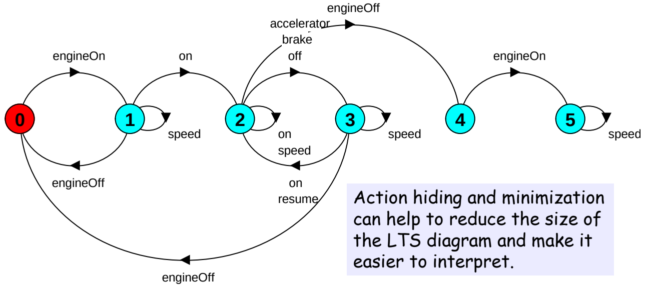
Check with no hidden actions

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

Control is not disabled when the engine is switched off !

cruise control model - minimized LTS

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



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

model - revised cruise control system

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

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

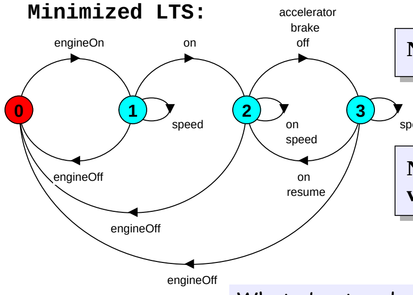
OK now?

Modify the safety property:

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

model - revised cruise control system

Minimized LTS:



No deadlocks/errors

No progress violations detected.

What about under adverse conditions? Check for system sensitivities.

model - system sensitivities

```
||SPEEDHIGH = CRUISECONTROLSYSTEM << {speed}.
```

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

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

model interpretation

Models can be used to indicate system sensitivities.

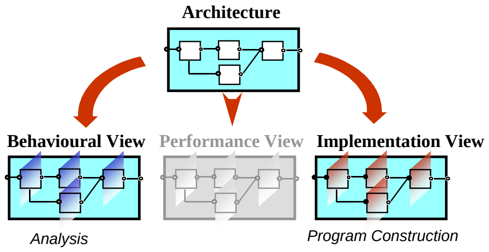
If it is possible that erroneous situations detected in the model may occur in the implemented system, then the model should be revised to find a design which ensures that those violations are avoided.

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

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

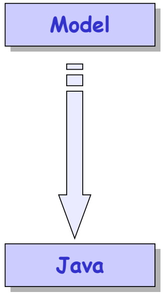
The central role of design architecture

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



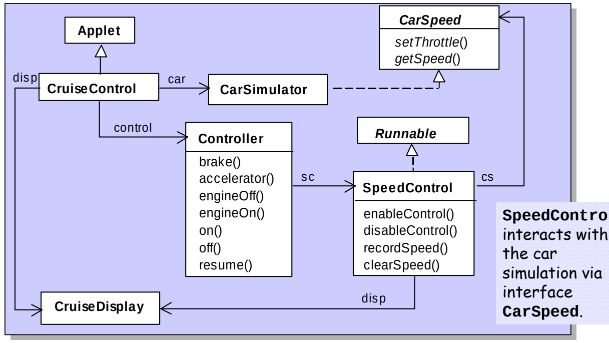
We consider that the models for analysis and the implementation should be considered as elaborated views of this basic design structure.

8.2 from models to implementations



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

cruise control system - class diagram



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

↖ **CRUISECONTROLLER** ↖ **SPEEDCONTROL**

cruise control system - class Controller

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

  Controller(CarSpeed cs, CruiseDisplay disp)
    {sc=new SpeedControl(cs,disp);}

  synchronized void brake(){
    if (controlState==CRUISING )
      {sc.disableControl(); controlState=STANDBY; }
  }

  synchronized void accelerator(){
    if (controlState==CRUISING )
      {sc.disableControl(); controlState=STANDBY; }
  }
 synchronized void engineOff(){
    if(controlState!=INACTIVE) {
      if (controlState==CRUISING) sc.disableControl();
      controlState=INACTIVE;
    }
  }
```

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

cruise control system - class Controller

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

This is a direct translation from the model.

cruise control system - class SpeedControl

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

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

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

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

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

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

cruise control system - class SpeedControl

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

  public void run() {     // the speed controller thread
    try {
      while (state==ENABLED) {
        Thread.sleep(500);
        if (state==ENABLED) synchronized(this) {
          double error = (float)(setSpeed-cs.getSpeed())/6.0;
          double steady = (double)setSpeed/12.0;
          cs.setThrottle(steady+error); //simplified feed back control
        }
      }
    } catch (InterruptedException e) {}
    speedController=null;
  }
}
```

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

Summary

- ◆ Concepts
  - ● design process:

    from requirements to models to implementations
  - ● design architecture
- ◆ Models
  - ● check properties of interest

    safety: *compose safety properties at appropriate (sub)system*

    progress: *apply progress check on the final target system model*
- ◆ **Practice**
  - ● model interpretation - to infer actual system behavior
  - ● threads and monitors

Aim: rigorous design process.

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

Chapter 10

Message Passing

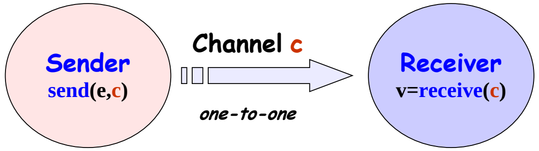
Message Passing

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)

10.1 Synchronous Message Passing - channel



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

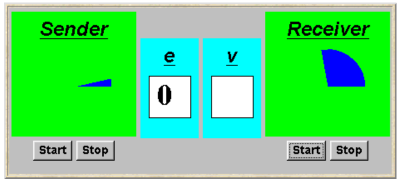
♦ *v* = **receive(*c*)** - receive a value into local variable *v* from channel *c*. The process calling the receive operation is ***blocked*** waiting until a message is sent to the channel.

cf. distributed assignment ***v = e***

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.



```
Channel chan = new Channel();
tx.start(new Sender(chan,senddisp));
rx.start(new Receiver(chan,recvdisp));
```

Instances of ThreadPanel

Instances of SlotCanvas

Java implementation - channel

```
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);
  }
}
```

The implementation of Channel is a monitor that has synchronized access methods for send and receive.

Selectable is described later.

Java implementation - sender

```
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

```
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){}
  }
}
```

model

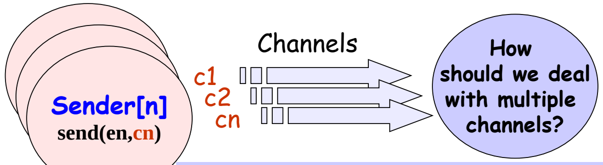
```
range M = 0..9                // messages with values up to 9

SENDER = SENDER[0],           // shared channel chan
SENDER[e:M] = (chan.send[e]-> SENDER[(e+1)%10]).

RECEIVER = (chan.receive[v:M]-> RECEIVER).

                              // relabeling to model synchronization
||SyncMsg = (SENDER || RECEIVER)
            /{chan/chan.{send,receive}}.
```

LTS?

How can this be modelled directly without the need for relabeling?

message operation	*FSP model*
send(e,*chan*)	?
v = **receive**(*chan*)	?

selective receive

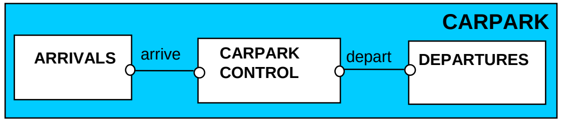


Select statement...

select
  when G_1 **and** v_1=**receive**($chan_1$) => S_1;
or
  when G_2 **and** v_2=**receive**($chan_2$) => S_2;
or
  when G_n **and** v_n=**receive**($chan_n$) => S_n;
end

How would we model this in FSP?

selective receive



```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 |when(i<N) depart->SPACES[i+1]
                 ).

ARRIVALS   = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

||CARPARK = (ARRIVALS||CARPARKCONTROL(4)
                     ||DEPARTURES).
```

Implementation using message passing?

Java implementation - selective receive

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

  public MsgCarPark(Channel a, Channel l,
                   StringCanvas d,int capacity) {
    depart=l; arrive=a; N=spaces=capacity; disp=d;
  }
  …
  public void run() {…}
}
```

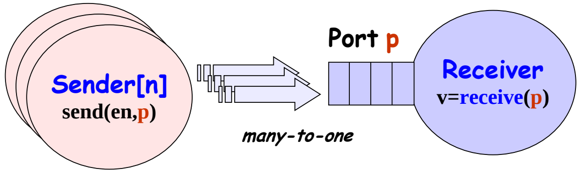
Implement CARPARKCONTROL *as a thread* MsgCarPark *which receives signals from channels* arrive *and* depart.

Java implementation - selective receive

```
public void run() {
    try {
      Select sel = new Select();
      sel.add(depart);
      sel.add(arrive);
      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;
        }
      }
    } catch InterrruptedException{}
}
```

See Applet

10.2 Asynchronous Message Passing - port



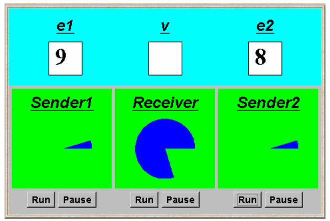
- **send**(e,p) - send the value of the expression *e* to port *p*. The process calling the send operation is **not** ***blocked***. The message is queued at the port if the receiver is not waiting.

- *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.

asynchronous message passing - applet

Two senders communicate with a receiver via an "unbounded" port.

Each sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.



```
Port port = new Port();
tx1.start(new Asender(port,send1disp));
tx2.start(new Asender(port,send2disp));
rx.start(new Areceiver(port,recvdisp));
```

Instances of ThreadPanel

Instances of SlotCanvas

Java implementation - port

```
class Port extends Selectable {
 Vector queue = new Vector();

   public synchronized void send(Object v){
     queue.addElement(v);
     signal();
   }

   public synchronized Object receive()
          throws InterruptedException {
     block(); clearReady();
     Object tmp = queue.elementAt(0);
     queue.removeElementAt(0);
     return(tmp);
   }
}
```

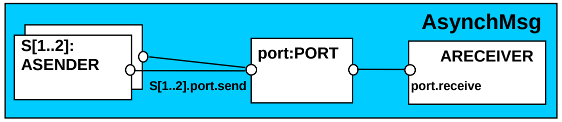
The implementation of Port is a monitor that has synchronized access methods for send and receive.

port model

```
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]),
PORT[h:M]                 //one message queued to port
  = (send[x:M]->PORT[x][h]
    |receive[h]->PORT
    ),
PORT[t:S][h:M]            //two or more  messages queued to port
  = (send[x:M]->PORT[x][t][h]
    |receive[h]->PORT[t]
    ).

// minimise to see result of abstracting from data values
||APORT = PORT/{send/send[M],receive/receive[M]}.
```

LTS?

model of applet

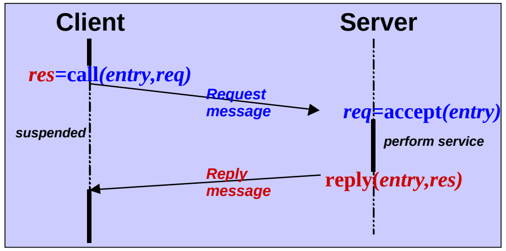


```
ASENDER = ASENDER[0],
ASENDER[e:M] = (port.send[e]->ASENDER[(e+1)%10]).

ARECEIVER = (port.receive[v:M]->ARECEIVER).

||AsyncMsg = (s[1..2]:ASENDER || ARECEIVER||port:PORT)
             /{s[1..2].port.send/port.send}.
```

Safety?

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.



Rendezvous

♦ ***res=call(e,req)*** - send the value ***req*** as a request message which is queued to the entry ***e***.

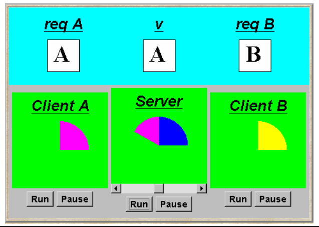
♦ ***req=accept(e)*** - receive the value of the request message from the entry ***e*** into local variable ***req***. The calling process is ***blocked*** if there are no messages queued to the entry.

♦ The calling process is ***blocked*** until a reply message is received into the local variable ***req***.

♦ ***reply(e,res)*** - send the value ***res*** as a reply message to entry ***e***.

asynchronous message passing - applet

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



```
Entry entry = new Entry();
clA.start(new Client(entry,clientAdisp,"A"));
clB.start(new Client(entry,clientBdisp,"B"));
sv.start(new Server(entry,serverdisp));
```

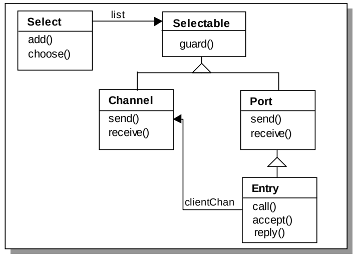
Instances of ThreadPanel

Instances of SlotCanvas

Java implementation - entry

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.

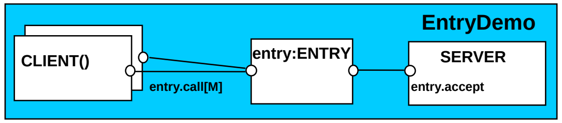
Java implementation - entry

```
public class Entry extends Port {
  private CallMsg cm;
  public Object call(Object req) throws InterruptedException {
    Channel clientChan = new Channel();
    send(new CallMsg(req,clientChan));
    return clientChan.receive();
  }
  public Object accept()throws InterruptedException {
    cm = (CallMsg) receive();
    return cm.request;
  }
  public void reply(Object res) throws InterruptedException {
    cm.replychan.send(res);
  }
  private class CallMsg {
    Object  request; Channel replychan;
    CallMsg(Object m, Channel c)
      {request=m; replychan=c;}
  }
}
```

Do `call`*,* `accept` *and* `reply` *need to be synchronized methods?*

model of entry and applet

We reuse the models for ports and channels …



```
set M = {replyA,replyB}         // reply channels
||ENTRY = PORT/{call/send, accept/receive}.
CLIENT(CH='reply) = (entry.call[CH]->[CH]->CLIENT).
SERVER = (entry.accept[ch:M]->[ch]->SERVER).
||EntryDemo = (CLIENT('replyA)||CLIENT('replyB)
                 || entry:ENTRY || SERVER  ).
```

Action labels used in expressions or as parameter values must be prefixed with a single quote.

rendezvous Vs monitor method invocation

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)?

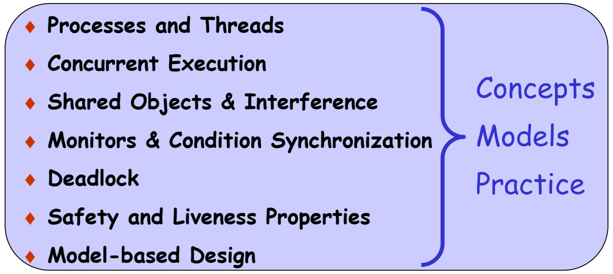
Summary

- 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**

Concepts
Models
Practice

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

LTSA User Manual

www.doc.ic.ac.uk/~jnm/book/firstbook/ltsa/ltsa-doc/User-manual.html

User manual

It is the hope of the designers of LTSA that this manual should be largely unnecessary. In most cases, the user simply has to enter a specification in the FSP window and invoke one of the analysis functions from the **Check** menu. LTSA will perform the necessary compilation and LTS composition.

Contents

LTSA Window

The LTSA window has the following controls in addition to the menubar.

Edit Button

This brings the FSP window to the front. The FSP window is used to enter the FSP specification text to be analysed. Text can be loaded from file (using the File menu) if LTSA is running as an application or it may be pasted into the window if LTSA is running as an applet.

Results Button

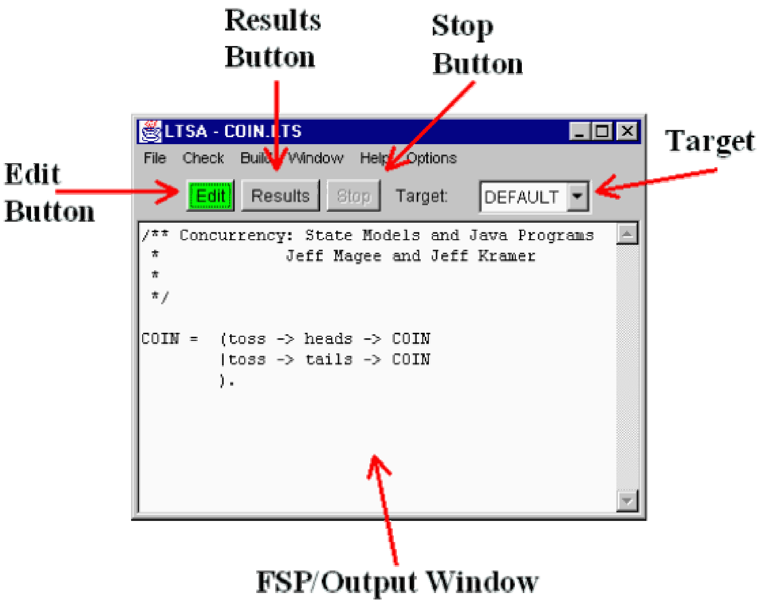
Brings the Output window to the front. This displays the results of analysis, error messages etc.

Stop Button

This is highlighted when LTSA is performing a computation, which could potentially take a long time such as minimisation. Clicking on the **Stop** button will abort the activity.

Target

The target choice box is used to select the composite process to be analysed. If there is only one composite process then this is set automatically. If no composite process is specified then the target displays "DEFAULT" which is the composite process consisting of the composition of all primitive processes in the current specification. For a specification with multiple composite processes, it is necessary to initialise the target choice when a specification is first loaded by invoking **Parse** from the **Build** menu.



```
/** Concurrency: State Models and Java Programs
 *             Jeff Magee and Jeff Kramer
 *
 */

COIN =  (toss -> heads -> COIN
        |toss -> tails -> COIN
        ).
```

LTS construction - Build

This menu contains all the functions necessary to generate an LTS from an FSP description. However, usually, it is not necessary to invoke these functions directly. For example, compilation is implicitly invoked if the description is changed between successive calls to the safety check function.

Parse

Performs a syntax check on the text in the FSP window. The location of the first error detected is highlighted and an error message displayed in the output window. Consequently, errors are located and fixed one at a time.

After a successful parse, the *target* choice will contain the list of composite processes. The visible process is the target for compilation etc. In addition, the list of actions menu's available from **Check/Run** is updated.

Compile

Generates an LTS for each of the component processes (whether primitive or composite) of the target composite process. After compilation, the component processes may be viewed either graphically or textually - see Window. Compile will automatically invoke parse if the FSP window has been changed since the last **Compile** (or **Parse**). However, if a new target or action menu is added, **Parse** must be invoked explicitly to update the target and run lists

Compose

Generates a single LTS by composing the component LTSs produced by compile for a specified target. After composition, the LTS may be viewed graphically or textually. Error messages produced during composition indicate safety property violations and deadlocks.

Minimise

Minimises the LTS produced by composition according to Milner's observation equivalence relation.

Analysis functions - Check

The analysis functions operate on the target composite process indicated in the **Target** choice box. If this has not been compiled or composed, compilation and composition are automatically invoked.

Safety

Performs a breadth first search on the target LTS. If a property violation or deadlock is found, the shortest trace of actions that would lead to the property violation or deadlock is displayed in the output window.

Progress

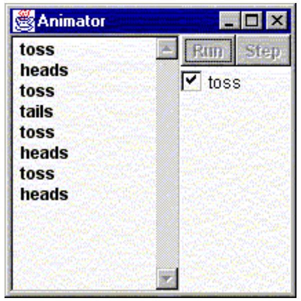
Computes the connected components for the target LTS. Checks for progress violations with respect to the declared progress properties. If no progress properties are declared then a check with respect to a default property is performed. The default property has all the actions in the alphabet of the current target.

Run

Performs a user-controlled animation of the target composite process. Uses the component LTSs rather than the composite LTS, so larger systems can be animated even if they cannot be exhaustively checked. DEFAULT is the alphabet of the target composite process and allows explicit contol of all actions. This may be reduced by declaring an explicit **menu** e.g.

```
menu RUN = {toss}
```

The current state of each component LTS displayed in a Draw window is updated by the animator. By default, the Animator window includes **Run** and **Step** buttons. These are used to control actions which are not in the action menu and consequently do not have click boxes. These buttons do not appear if the autorun option is selected in the **Options** menu. The **Run** button permits a sequence of actions to occur where these actions are not explicitly controlled. The **Step** button permits a single such action to occur.



```
toss
heads
toss
tails
toss
heads
toss
heads
```

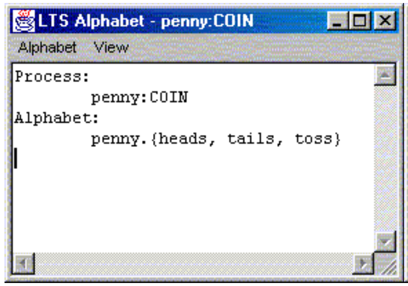
Reachable

Performs an "on-the-fly" depth first search on the set of component LTS for the target. Since the composite LTS is not required, this uses less storage than **Safety**. Property violations and deadlocks are detected, however, no counter examples (traces) are produced.

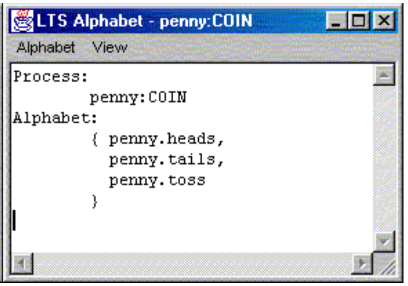
Display functions - Window

Alphabet

Displays the action alphabet for either the component process of a target or the target LTS itself. The alphabet is by default displayed concisely - actions with common prefixes are collected into sets or ranges. The alphabet may also be viewed in an expanded form by choosing **Expanded** from the **View** menu. The view can be adjusted between fully expanded and the concise view using **Expand** and **Contract** from **View**.



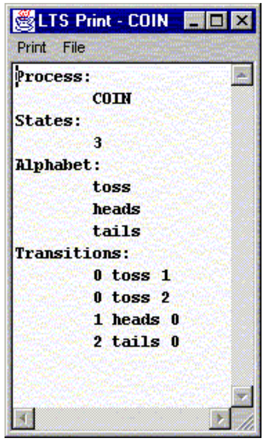
```
Process:
        penny:COIN
Alphabet:
        penny.{heads, tails, toss}
```



```
Process:
        penny:COIN
Alphabet:
        { penny.heads,
          penny.tails,
          penny.toss
        }
```

Text

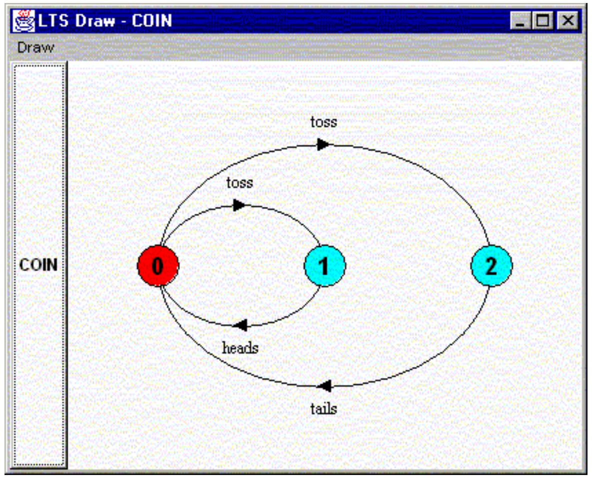
Displays as text the LTS for either the component process of a target or the target LTS itself. When LTSA is running as an application, the textual representation may be saved to file.



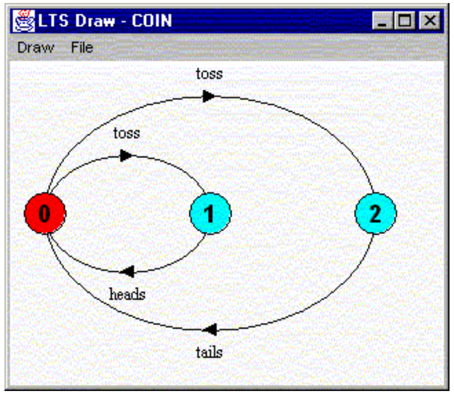
```
Process:
        COIN
States:
        3
Alphabet:
        toss
        heads
        tails
Transitions:
        0 toss 1
        0 toss 2
        1 heads 0
        2 tails 0
```

Draw

Displays graphically the LTS for either the component process of a target or the target LTS itself. A button for each process in the target appears at the left of the window. Clicking on the process button displays the LTS for that process. During animation the process buttons for each process participating in the last action to occur are coloured red.



A separate window for the process is created when the process is selected from the **Draw** menu. When LTSA is running as an application, the graphical representation may be saved to file (in PICT format) from this window. The **Freeze Drawing while adjusting Window** option under the **File** menu allows the LTS graph to be repositioned inside the window before the PICT image is saved.



Options

Display warning messages

Displays a warning message when an undefined state is mapped to the ERROR state during compilation. Default is set.

Treat warnings as errors

Halts compilation with an error message when an undefined state is found. Default is not set.

Minimise during Composition

If set, composite processes are minimised by default during compilation (ie. Composite processes which are components of the target). Default is not set.

Use big font

Use large font in all windows. Useful for demonstrations and presentations. Default is not set.

Use arrows when drawing LTS

Transitions are always drawn clockwise, consequently, arrows are not strictly necessary. The LTS will sometimes be clearer without arrows. Default is set.

Display name when drawing LTS

Displays the process name for an LTS in the Draw window. Defaults is not set.

Process Buttons in LTS Draw window

Enables process buttons in Draw window. Default is set.

Autorun actions in Animator

Removes **Run** and **Step** buttons from Animator window. Actions are selected until an action in the menu set is enabled. Default is not set.

FSP Quick Reference Guide

http://www.doc.ic.ac.uk/~jnm/book/ltsa/Appendix-A-2e.html

Appendix A

FSP Quick Reference

A.1 Processes

A process is defined by a one or more local processes separated by commas. The definition is terminated by a full stop. **STOP** and **ERROR** are primitive local processes.

Example

Process = (a -> Local),

Local = (b -> STOP).

Action Prefix **->**	If x is an action and P a process then (x->P) describes a process that initially engages in the action x and then behaves exactly as described by P.
Choice **|**	If x and y are actions then (x->P|y->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.
Guarded Action **when**	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.
Alphabet Extension **+**	The alphabet of a process is the set of actions in which it can engage. P + S extends the alphabet of the process P with the actions in the set S.

Table A.1 – Process operators

A.2 Composite Processes

A composite process is the parallel composition of one or more processes. The definition of a composite process is preceded by **||**.

Example

||Composite = (P || Q).

Parallel Composition **||**	If P and Q are processes then (P||Q) represents the concurrent execution of P and Q.
Replicator **forall**	**forall** [i:1..N] P(i) is the parallel composition (P(1) || … || P(N))
Process Labeling **:**	a:P prefixes each label in the alphabet of P with a.
Process Sharing **::**	{a_1, . ., a_x}::P replaces every label n in the alphabet of P with the labels a_1.n,…,a_x.n. Further, every transition (n->Q) in the definition of P is replaced with the transitions ({a_1.n,…,a_x.n}->Q).
Priority High **<<**	||C =(P||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||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.
Priority Low **>>**	||C=(P||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||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.

Table A.2 – Composite Process Operators

A.3 Common Operators

The operators in Table A.3 may be used in the definition of both processes and composite processes.

Conditional **if then else**	The process **if** B **then** P **else** Q behaves as the process P if the condition B is true otherwise it behaves as Q. If the **else** Q is omitted and B is false, then the process behaves as STOP.
Re-labeling **/**	Re-labeling is applied to a process to change the names of action labels. The general form of re-labeling is: /{*newlabel*_1/*oldlabel*_1,… *newlabel*_n/*oldlabel*_n}.
Hiding \	When applied to a process P, the hiding operator \\{a_1..a_x} removes the action names a_1..a_x from the alphabet of P and makes these concealed actions "silent". These silent actions are labeled tau. Silent actions in different processes are not shared.
Interface **@**	When applied to a process P, the interface operator @{a_1..a_x} hides all actions in the alphabet of P not labeled in the set a_1..a_x.

Table A.3 – Common Process Operators

A.4 Properties

Safety **property**	A safety **property** P defines a deterministic process that asserts that any trace including actions in the alphabet of P, is accepted by P.
Progress **progress**	**progress** P = {a_1,a_2..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..a_n will be executed infinitely often.

Table A.4 – Safety and Progress Properties

A.5 *FLTL* – Fluent Linear Temporal Logic

Fluent **fluent**	**fluent** FL = <{s_1,…s_n}, {e_1..e_n}> **initially** *B* defines a fluent FL that is initially true if the expression *B* is true and initially false if the expression *B* is false. FL becomes true immediately any of the initiating actions {s_1,…s_n}occur and false immediately any of the terminating actions {e_1..e_n} occur. If the term **initially** *B* is omitted then FL is initially false.
Assertion **assert**	**assert** PF = FLTL_Expression defines an FLTL property.
&&	conjunction (*and*)
||	disjunction (*or*)
!	negation (*not*)
->	implication ((A->B)≡ (!A || B))
<->	equivalence ((A<->B) ≡(A->B)&&(B->A))
next time **X *F***	iff ***F*** holds in the next instant.
always **[]*F***	iff ***F*** holds now and always in the future.
eventually **<>*F***	iff ***F*** holds at some point in the future.
until ***P* U *Q***	iff ***Q*** holds at some point in the future and ***P*** holds until then.
weak until ***P* W *Q***	iff ***P*** holds indefinitely or ***P* U *Q***
forall	**forall** [i:R] FL(i) conjunction of FL(i)
exists	**exists** [i:R] FL(i) disjunction of FL(i)

Table A.5 – Fluent Linear Temporal Logic