

CONCURRENT SYSTEMS

Previous lectures have introduced concepts of concurrency, communication and modelling.

In lecture Notes 3 we developed techniques for modelling program objects such as variables and assignment statements.

In this lecture we will use these techniques to model a simple concurrent program, and see how the program can be analysed to determine its correctness.

This lecture will introduce the XCircal language and the Circal System, which is a tool for practical modelling and verification of concurrent systems.

PROPERTIES AND CORRECTNESS

We can often describe what it means for a program or system to be correct by giving a set of *properties* that the system must satisfy. There are several different types of property.

For programs that are supposed to terminate, *correctness* properties assert that a program finishes with the right answer (ie that it correctly maps starting program states to final states).

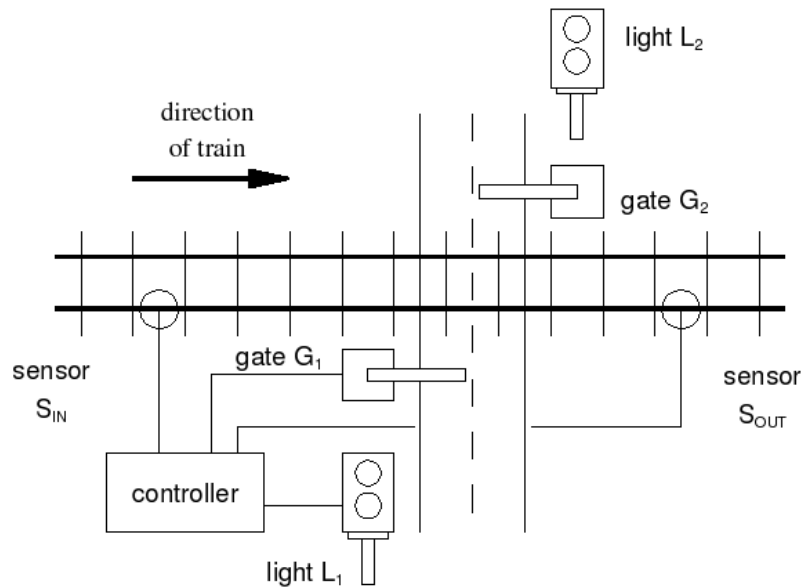
Not all programs are designed to terminate (consider an operating system or web server, for example). For non-terminating programs there are additional types of property.

Safety properties assert that a system never does anything "bad" ie that it remains in a "correct" state at all times.

Liveness properties assert that a system "makes progress", ie that some desirable state will eventually be reached. Note that a system which does nothing is guaranteed to be *safe*, but doesn't satisfy any *liveness* properties.

A RAILWAY LEVEL CROSSING EXAMPLE

As an example of a system with an important safety property, consider a railway level crossing.



Vehicle movement on the road is controlled by the operation of lights and gates. Lights L_1 and L_2 go on simultaneously to stop the traffic and go off when it is clear for traffic to cross the railway track. Gates G_1 and G_2 close simultaneously to physically prevent vehicles crossing the track. The track on either sides of the crossing utilises two sensors and presumes that trains will only ever travel from left to right.

Sensor S_{in} on the left detects when the train enters the region of the crossing while sensor S_{out} on the right of the crossing detects when the train has left the region. The sensors are connected to a control unit called Control, which is also connected to the lights and the gates.

Ignoring the internal details of each component, the complete crossing description is obtained by combining the six components using the concurrent composition operator. The railway crossing behaviour is then given by the Crossing process defined as follows.

```
Crossing=Control*Train*SensorIn*SensorOut*Lights*Gates
```

A necessary property for the safe operation of the railway crossing is to ensure that when the train reaches the crossing the gates are down, so ensuring that there are no vehicles crossing at that point in time. The presence of a train at the crossing point is indicated by a action called `trainCrossing`, which is generated by a process `Train`. This process describes the position of the physical train travelling on the track. It generates actions `trainIn`, `trainCrossing` and `trainOut` to indicate that the train is entering the region of the crossing, is at the middle of the crossing, and is exiting the region of the crossing (respectively).

The *IsSafe* process, which states this safety property, is defined by:

```
Process IsSafe  
sSafe <- gatesDown trainCrossing gatesUp IsSafe
```

This behaviour states that a sequence of actions is rigorously followed such that the `trainCrossing` actions, which indicates that the train has reached the crossing, only occurs between occurrences of the `gatesDown` and `gatesUp` actions. In other words the gates are always lowered before the train reaches the crossing and are never raised before it leaves the crossing. This ensures that the gates are "always down", or equivalently are "never up" when the train is in the crossing.

This property can be verified using the Circal System by checking whether the *IsSafe* property is "internalised" within the Crossing process. Without going into details, this can be achieved by checking the following equivalence:

```
Crossing * IsSafe == Crossing
```

Notice that `==` is the Circal System symbol for equivalence.

THE CIRCAL SYSTEM

The Circal process algebra provides a fully formal language and precise semantics for modelling concurrent systems. A bare process algebra is, however, a very "low level" language (rather like an assembly language).

The Circal System is a practical modelling tool based on the Circal process algebra. It has two important parts:

- * A C-like programming language (XCircal) for building and printing finite state processes
- * An equivalence checking engine for comparing processes

The Circal System can be downloaded from:
<http://www.csse.uwa.edu.au/FormalSpecification/CircalSystem/>

There is a Linux binary package which is simple to install and that will run on the lab machines.

The Circal System is run by invoking the `circ` executable, eg:

```
prompt> ./circ example.xtc
```

where "example.xtc" is an XCircal source file. The Circal System interprets the XCircal code from top to bottom. The Circal System can be used to build Circal processes, display (print out) the state diagram of a process, and check whether two processes are the same.

PROCESS ALGEBRA AND PROGRAMMING LANGUAGES

We are using Circal and Java for studying concurrency in this course, for varying reasons. Modelling with a process algebra such as Circal introduces the concepts of *rigorous description* and *rigorous analysis* of concurrent systems, while the Java component introduces the practice of concurrent programming.

Circal (Process Algebra)	Java (Programming Language)
finite state systems	unbounded state (in theory)
concurrency a primitive (concurrent composition)	concurrency complex (threads, objects, synchronisation)
Have to build memory, arithmetic, logic.	variables, arithmetic, method calls all primitive.
complete control over concurrent behaviour of primitives.	no control of concurrent behaviour of primitives.
simple proof of some properties (equivalence checking).	complex proof of properties (temporal logic etc)

THE MUTUAL EXCLUSION PROBLEM

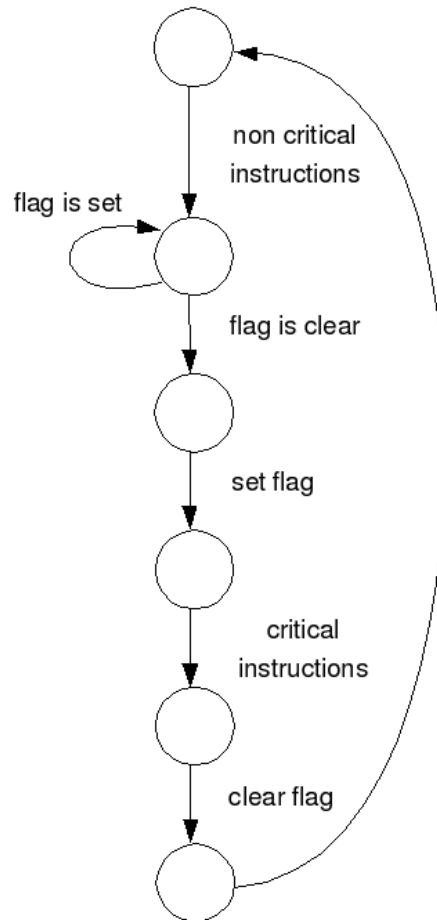
We will introduce XCircal and the Circal System by studying an archetypal concurrent programming problem - the mutual exclusion problem:

N processes are executing in an infinite loop a sequence of instructions which can be divided into two parts, the critical section and the non-critical section. The program must satisfy the *mutual exclusion property*: only one process may be executing instructions in its critical section at a time.

We study this problem because (a) it is a practical problem that arises in real-world concurrent systems (ensuring proper access to a shared resource e.g. a disk) and (b) if we have a solution to the mutual exclusion problem, it can be used as the basis for solutions to other concurrent programming problems.

Clearly, without some coordination or communication between worker processes, the mutual exclusion property will not be satisfied. There is no reason why two workers cannot both advance into their critical sections at the same time.

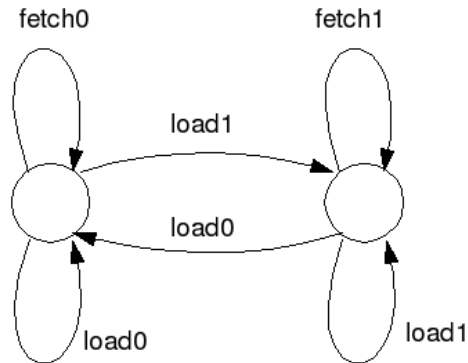
A SHARED VARIABLE SOLUTION



Our attempt to solve the mutual exclusion problem will be to give both worker processes access to a *shared boolean variable*, which shall be used as a flag to indicate that one of the workers is currently in its critical section. Before entering its critical section, a worker will check to see if the flag is set. If it is, the worker will wait and try again later. If the flag is not set, it will set the flag, execute its critical section, and then clear the flag.

To model this system, we require three processes: two worker processes and a process representing the shared variable. Note that we are explicitly modelling the *variable* as a process, not just the workers.

SHARED VARIABLE PROCESS MODEL



The shared variable process is similar to the one given in lecture notes 3, with a few changes. The variable will start with value 0, and the variable will be accessible by two different processes. The XCircal code for the shared variable is:

```
Process V[2]
Event fetchA0, fetchA1, loadA0, loadA1
Event fetchB0, fetchB1, loadB0, loadB1

V[0] <- fetchA0 V[0] + fetchB0 V[0] +
        loadA0 V[0] + loadB0 V[0] +
        loadA1 V[1] + loadB1 V[1]
V[1] <- fetchA1 V[1] + fetchB1 V[1] +
        loadA0 V[0] + loadB0 V[0] +
        loadA1 V[1] + loadB1 V[1]
```

DISPLAYING BEHAVIOURS

As with all programming, it is good to test as we work. We can make the Circal System display the behaviour of a process with the "display" operator:

```
display V[0]
```

which gives:

```
S0 == (((((fetchA0 S0 + fetchB0 S0) + loadA0
S0) + loadB0 S0)+loadA1 S1) + loadB1 S1)
```

```
S1 == (((((fetchA1 S1 + fetchB1 S1) + loadA0
S0) + loadB0 S0)+loadA1 S1) + loadB1 S1)
```

The "display" operator prints out the system behaviour as a *state machine*, with the states labelled as S0, S1, S2, S3...

WORKER PROCESS MODEL

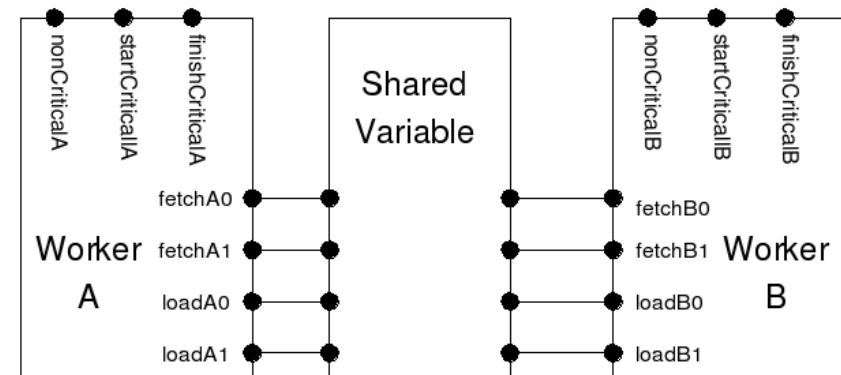
Our XCircal model of the worker processes follows directly from the previous state diagram:

Process A, AWait, B, BWait

```
A      <- nonCriticalA AWait
AWait  <- fetchA1 AWait +
        fetchA0 loadA1 start_CriticalA
        finish_CriticalA loadA0 A
```

```
B      <- nonCriticalB BWait
BWait  <- fetchB1 BWait +fetchB0 loadB1
        start_CriticalB finish_CriticalB
        loadB0 B
```

CONCURRENT COMPOSITION OF SYSTEM



Our system is then the concurrent composition of the shared variable and the two workers.

Process System

```
System <- A*B*V[0]
```

```
display ~(A*B*V[0])
```

The resulting print out of the system behaviour may seem large (having 54 states), but remember that this represents *every possible* sequence of events that the system can perform, not just a single "run".

It turns out that this system is incorrect. If we trace through the sequence of states (you will do this in your laboratory session), we find the following sequence of events is possible:

```
nonCriticalB S1
nonCriticalA S2
fetchB0 S5
fetchA0 S10 // uh-oh
loadB1 S15
loadA1 S22 // this ain't good ...
startCriticalB S21
startCriticalA S30
```

The failure is due to the fact that worker A can find the value of the flag to be 0 *after worker B has decided to enter its critical section but before it has set the flag.*

How can we correct this problem ? Chapter 3 of your textbook covers several solutions using shared variables of the type we have modelled above.

A TEST-AND-SET INSTRUCTION

Looking at the pathological behaviour above, it seems that we could correct the problem if we could eliminate the delay between finding the flag at 0 and setting it to 1. That is, we might be able to solve the problem if we had an *atomic* (indivisible) *test-and-set* instruction.

This is a common theme in concurrent programming: creating sequences of instructions that cannot be interrupted.

A shared variable with a test-and-set instruction can be modelled as follows:

```
V[0]<-fetchAndSetA0 V[1]+loadA0 V[0]+
    fetchAndSetB0 V[1]+loadB0 V[0]
V[1]<-fetchAndSetA1 V[1]+ loadA0 V[0]+
    fetchAndSetB1 V[1]+ loadB0 V[0]
```

Real microprocessors in fact include atomic test-and-set instructions for just this purpose.

Replacing the fetch and load instructions in the worker processes yields:

```
A      <- nonCriticalA AWait
Await  <- fetchAndSetA1 AWait +
      fetchAndSetA0
      startCriticalA finishCriticalA
      loadA0 A

B      <- nonCriticalB BWait
BWait  <- fetchAndSetB1 BWait +
      fetchAndSetB0 startCriticalB
      finishCriticalB loadB0 B
```

If we display the resulting behaviour and examine it, it now looks correct - the mutual exclusion property is now observed in all states.

ABSTRACTING IRRELEVANT EVENTS

The question arises, "how can we be sure that the system is correct" ? Tracing through the behaviour quickly becomes tedious and error-prone for larger behaviours. The Circal System provides several tools to assist with this kind of analysis.

Firstly, we can note that the mutual exclusion property depends *only* on the occurrence of the "start" and "finish" critical section actions. None of the other actions has any effect on the property. We can display the behaviour of just the critical section events by *abstracting* all others:

```
Event varEvents, nonCriticals
varEvents=fetchAndSetA0
fetchAndSetA1
loadA0fetchAndSetB0
fetchAndSetB1 loadB0
nonCriticals=nonCriticalA nonCriticalB
```

```
display ~(System - varEvents - nonCriticals)
```

`varEvents` and `nonCriticals` are the sets of events that we wish to hide. The Circal symbol for abstraction is the minus sign "-".

The resulting behaviour has only 14 states and is simple enough to trace through by hand.

VERIFICATION BY EQUIVALENCE CHECKING

Can we do better, though ? The process of "tracing through" the behaviour and checking that a property is satisfied seems mechanical - it should be able to be automated.

We can get the Circal System to do exactly this, but first we need to express, in XCircal, the property that we are trying to verify.

The essence of the mutual exclusion property is that events marking the beginning and ending of critical sections can only appear in matched pairs -- the finish event for worker X must immediately follow the start event for worker X. We can write this behaviour as a process:

```
Process Mutex
```

```
Mutex <- startCriticalA finishCriticalA Mutex +  
        startCriticalB finishCriticalB Mutex
```

So the question is, is the simplified behaviour of the system (with all the non-critical section events abstracted) the same as the Mutex property ? We can ask the Circal System to check this.

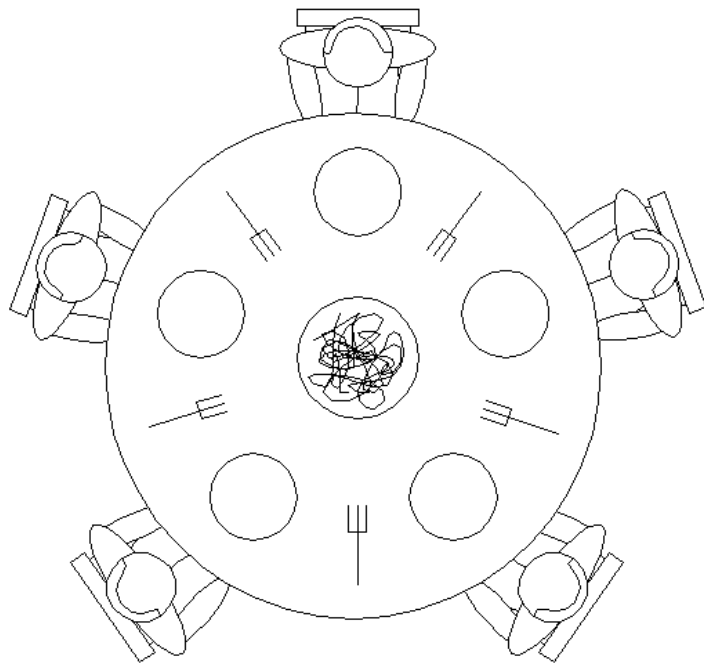
```
print "(System - varEvents - nonCriticals)==may Mutex  
? "  
print (may_eq((System-varEvents-nonCriticals), Mutex))  
print "\n"
```

And we find out that, despite appearances, they have the same behaviour. The "may_eq" function, which checks to see if two processes are equivalent, is built into the Circal System.

```
(System - varEvents - nonCriticals) ==may Mutex ? true
```

The Dining Philosophers Problem

The so called “Dining Philosophers” problem is a piece of computer science folklore (used by Edsger Dijkstra in 1971). It illustrates classic issues of concurrency and contention for shared resources.



A number of philosophers sit around a circular table. The philosophers only do two things: they alternately *think* and *eat*. Between each philosopher is a fork. Because of the tangled supply of noodles in the middle of the table, each philosopher requires two forks in order to eat. Each philosopher thus executes the following process:

Think
Obtain forks
Eat
Release forks
Think
...

The interesting part is the protocol whereby philosophers obtain and release their forks.

Compare this system to the mutual exclusion example in Lecture Notes 4. There, we had 2 worker processes contending for 1 shared resource (the right to enter a critical section). Here, we have N processes worker processes (the philosophers), contending for N shared resources (the forks).

The various processes are linked (and thus *communicate* in a pattern. Philosopher I requires forks (I) and (I+1), while forks (I) is shared between philosopher (I-1) and philosopher (I). This problem is thus (a) bigger (more processes) and (b) has a structured pattern.

This system is more complex, but it does have a regular pattern. We can use XCircal to automatically construct all the processes for a problem of size N and link them correctly together.

First let us model a single fork. It can be grabbed, then freed, from either the left or right.

```
Frk <- gfl ffl Frk + gfr ffr Frk
```

Now, each of our forks has the same behaviour, but with different events. We can write a XCircal function that (a) defines a *prototype* fork and (b) makes a copy of this prototype with 4 specified event names.

```
Process Fork (Event gfl, ffl, gfr, ffr) {  
  Static event gfl', ffl', gfr', ffr' static  
  Process Frk {  
    Frk <- gfl' ffl' Frk + gfr' ffr' Frk }  
  Return Frk [gfl/gfl', ffl/ffl', gfr/gfr',  
             ffr/ffr']  
}
```

We can then use this function to create N fork processes.

```
Process F [n]  
Event gfl [n], gfr[n], ffrl[n], ffr[n]  
  
For (i=0;i<n;i++) // instantiate n forks  
F[i]<-Fork (gfl[i],ffl[i], gfr[i], ffr[i])
```

You may notice that each fork acts like a semaphore: the 'grab' events act as the semaphore 'wait' operation and then 'free' events act as the semaphore 'signal' operation. This suggests the following protocol for the philosopher.

- 1) Wait for left fork to become free and grab it
- 2) Wait for right fork to become free and grab it
- 3) Eat
- 4) Free left fork
- 5) Free right fork

In XCircal, a prototypical philosopher can thus be written:

```
Ph <- think glf grf eat flf frf Ph
```

We will study this model of the dining philosophers problem in the second lab session.