Csc72010

Parallel and Distributed Computation and Advanced Operating Systems

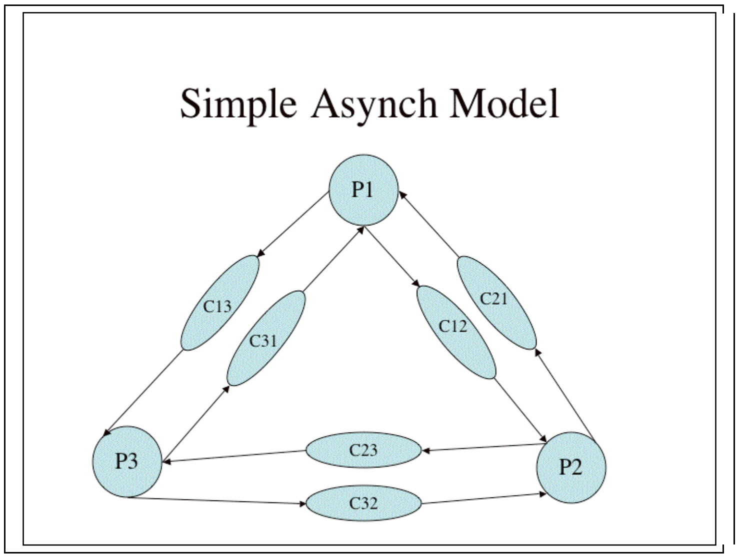
Lecture 2

February 2, 2006

The I/O Automaton Model

I/O automata model
This is a general mathematical model for reactive components. It imposes very little structure – we add structure for various kinds of systems.

For this course, we will model networks as processes communicating via channels



The Pi's and Ci's are reactive components, i.e., modules that interact with their environments using input and output actions (send and receive).

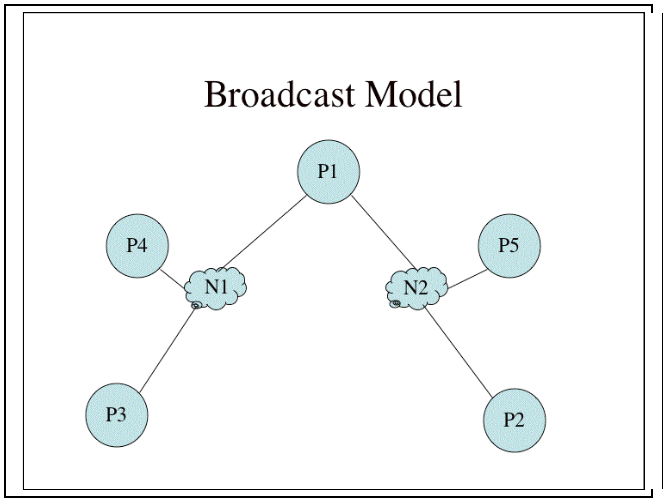
P1, P2, P3 are processes and have input actions receive and output actions send
Cij are channels and have input actions send and output actions receive

The process and channel actions correspond where there is an arrow. So for example P1 has outputs send(m,1,2) and send(m,1,3), while C12 has input send(m,1,2) and C13 has input send(m,1,3).

Also, P1 has inputs receive(m,2,1) and receive(m,3,1) and C21 has output receive(m,2,1) and C31 has output receive(m,3,1).

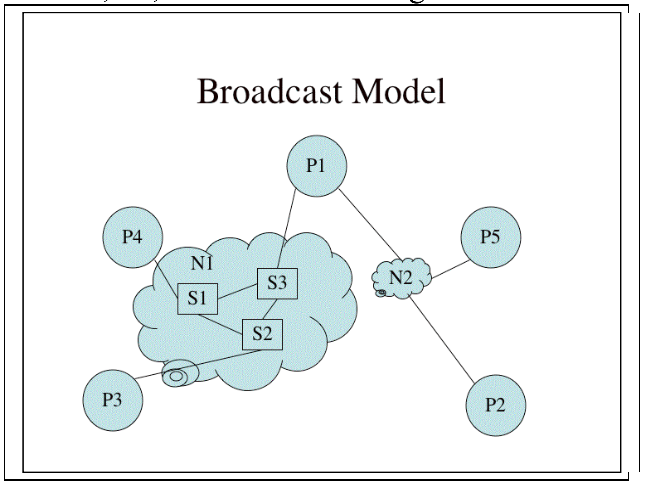
The above diagram is the network we will use for algorithms in the book. However, it is a point-to-point network, and Ethernets are shared (broadcast) networks, so they need to be modeled a little differently. See the diagram below.

A point-to-point network is implicitly specified by giving the endpoints; in the broadcast model, the network needs its own id. So, the actions would be send(n,i,j,m) and receive(n,i,j,m), where n is the network, i is the source, j is the destination, and m is the message. The action send(n,i,j,m) is an ouput of process i and an input of network n. The action receive(n,i,j,m) is an output of network n and an input of process j.



The I/O automaton model is designed to make it easy to organize the description of a system and to prove things about it:

1) We can compose components to form a larger system – we could have started just with P1, P3, and P4, and then added N2, P5, and P2.
2) We can describe systems at different levels of abstraction - e.g., N1 could be a collection of switches running learning bridge and spanning tree algorithms, but all we need to use is that this collection of switches lets P1, P3, and P4 send messages to each other.



3) We have good proof methods
    a. Invariants

b. Composition and projection
c. Simulation relations

Definitions

An I/O automaton A consists of

- sig(A), a *signature*, which specifies the input, the outut, and the internal actions of the automaton.
    - in(A) is the set of input actions
    - out(A) is the set of output actions
    - internal(A) is the set of internal actions
    - local(A) = out(A) $\cup$ internal(A) is the set of locally-controlled actions
    - acts(A) is the set of all actions
- a set of states states(A), which may be infinite
- a set of start states start(A)$\subseteq$states(A)
- a set of transitions trans(A)$\subseteq$states(A)$\times$acts(A)$\times$states(A)
- an equivalence relation tasks(A) on the local actions of A (i.e., internal and output actions).

There's one restriction on all of this: any input is enabled in any state, i.e., there is a transition involving that input.
For all s$\in$ states(A) and $\pi\in$ in(A), there is a transition <s, π, t> in trans(A).
This is because we don't want an automaton to be able to prevent the environment from doing something. This requires us to model behavior in "bad" environments, which do unexpected things. If we really want to restrict inputs, we can model the "good" environment as another automaton that only passes on the good inputs from the real environment.

In other words, inputs are controlled by the environment and can happen at any time. Input and output are external and can be seen by the environment. Output and internal are locally controlled, i.e., happen under control of the automaton.

States can be infinite, to let us model queues that grow without bound, files, and so on. This may not be realistic sometimes, but usually simplifies the model.

Tasks are groups of locally-controlled actions that should get an opportunity to happen. They are use to model "fair" executions. For example, we may want to say that we're only interested in networks where every process gets to send a message infinitely often (that is, it's not blocked forever from sending). Then for each network that a process is connected to, it would have a task containing all possible send(n,i,j,m), i.e., n and i are fixed but j and m can be any values.

Digression to explain tasks: There are two kinds of properties, safety and liveness. Safety properties say that bad things don't happen; liveness properties say that good things will eventually happen. Usually, we have to know that processes continue to interact with each other in order to guarantee that good things happen. Thus we don't try to prove the "good things" for sequences of events in which one process stops doing anything.

Examples

Channel automaton

This is a reliable FIFO channel, unidirectional between 2 processes.
Fix a message alphabet M.
The signature is:
inputs: { send(m): m∈M }
output: { receive(m): m∈M }
states: a FIFO queue of messages, call it queue, initially empty.

transitions: these are described by code fragments

In the executable language:

```
automaton Channel(i, j: Int)

  signature
    input send(const i, const j, m: Int)
    output receive(const i, const j, m: Int)

  states
    queue: Seq[Int] := {}

  transitions

    input send(i, j, m)
      eff queue := queue |- m

    output receive(i, j, m)
      pre m = head(queue)
      eff queue := tail(queue)
```

Process automaton

Here is a trivial process:

```
automaton Process(n: Int)
  signature
    input receive(const n-1, const n, x:Int)
    output send(const n, const n+1, x:Int)
  states
    toSend:Seq[Int]:= {}|-n
  transitions
    input receive(i, j, x)
      eff toSend := toSend |- x

    output send(i, j, x)
      pre x = head(toSend)
      eff toSend := tail(toSend)
```