

The University of Sydney  
AUSTRALIA

# FSP Lectures

Alan Fekete  
University of Sydney, 2004

# FSP

- A way to model concurrent systems
  - such as those coded with multithreaded Java code
- Developed by J. Magee and J. Kramer of Imperial College
  - In their book "Concurrency: State Machines and Java Programs" (isbn: 0471987107)
- Supported by a tool: LTSA
  - Download from [www-dse.doc.ic.ac.uk/concurrency](http://www-dse.doc.ic.ac.uk/concurrency)
- Founded on Hoare's CSP model with simplifications to allow automated analysis

FSP Lectures, Alan Fekete, University of Sydney (2004)

# Design approach

- Produce a high-level design
  - much more abstract than code
  - captures key patterns of interaction
- Explore its properties
  - check it avoids errors
    - (normally, it doesn't; re-design and re-check!)
- Write code that follows the design

FSP Lectures, Alan Fekete, University of Sydney (2004)

# Discrete models

- Real systems can change continuously
  - eg: furnace's temperature
  - eg: car's position
  - eg (effectively): thread's program counter
- Model concentrates on major alterations
  - happen in *events* at widely separated and clearly demarcated times
  - eg: furnace switched on
  - eg: car brake applied
  - eg thread is dispatched

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Assumed Knowledge

- Experience with coding Java threads
  - Including common problems like data interference and deadlock
  - Including mechanisms of synchronized blocks and wait/notify calls
- See eg B. Eckel “Thinking in Java (3<sup>rd</sup> ed)”, ch 13

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Overview

- *Syntax and state machine model of an FSP Process*
- Composition of Processes
- Sharing and Mutual Exclusion
- Deadlocks
- Using FSP to Model and Design Java Classes
- Dining Philosophers Example

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Overview of an FSP Process

- Key concepts
  - action
  - trace
  - set of traces
  - process syntax (FSP)
  - state machine (LTS)

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Action

- Each possible event of importance can be identified and named
  - interaction of system with its environment
    - environment does something to system
    - environment observes some aspect of system
    - system does something to environment
  - internal transformation of system
- Action name starting in lower case
  - later we'll see structured names too

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Action examples

### Timebomb

- arm
- disarm
- tick
- boom

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Action examples

- drink machine
  - deposit1dollar
  - deposit2dollars
  - pushcancel
  - pushsodabutton
  - pushjuicebutton
  - return1dollar
  - return2dollars
  - providesoda
  - providejuice

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Trace

- A system executes by taking actions one after another in a sequence
  - what an observer would notice
  - called a trace of the system
  - may be finite (stops eventually) or infinite
- Eg timebomb does
  - arm -> tick -> tick -> disarm
- Eg drink machine does
  - deposit2dollars -> pushjuicebutton -> providejuice -> return1dollar -> deposit1dollar -> pushsodabutton -> providesoda -> ....

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Set of traces

- The same system may execute in different ways on different occasions
  - eg timebomb may also do
    - arm -> tick -> tick -> tick -> tick -> boom
- We regard the important model as the *set* of all possible traces that the system could do
  - “system is correct” means that every trace is acceptable
  - system’s not correct if some traces are acceptable and some are not!

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Why a set?

- In different executions, environment does different things
  - eg user can deposit1dollar or deposit2dollars
  - this leads system to act differently thereafter
- Also, system may have internal non-determinism
  - random choices
  - unpredictable delays, loss etc

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Representing a set of traces

- Different ways to describe a set of sequences
- List the sequences
  - { a->b, a->b->a->b, a->b->a->b->a->b }
  - establish pattern and rely on reader to continue it
- Explain in words
  - do a then b and repeat up to twice more
- Use regular expressions
  - Studied in subjects on theory of computation, formal languages

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Warnings

- The set may have a finite or infinite number of traces in it
  - It is possible to have a set with just one trace
- Some traces in the set may be finite while others are infinite

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Warnings

- Precision matters
  - “set of all finite sequences of a’s” includes an infinite number of sequences such as
    - a->a->a,
    - a->a->a->a,
    - etc
  - but it does not include the infinite sequence
    - a->a->a->a->.....

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Process syntax

- We represent a system by a *process*
  - a piece of text in a precise notation (called FSP)
- The LTSA tool from Imperial College can check that the syntax is correct

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Overview of FSP

- A process is named, beginning in UPPER CASE
- Give a collection of equations, separated by commas, each defining a process in terms of expressions built up from actions and from other processes
  - recursion is allowed (right-hand side can refer to same process as left-hand side, or several processes can be defined in terms of one another)

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Process example

- The following defines the process P1, using P2 and P3 as auxiliary names
  - $P1 = (a \rightarrow P2 \mid b \rightarrow P3),$
  - $P2 = (x \rightarrow y \rightarrow STOP),$
  - $P3 = (z \rightarrow P1).$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Warning

- The syntax is constrained by subtle rules. The following is NOT legal:
  - $P1 = (a \rightarrow b \mid P2),$
  - $P2 = a \rightarrow P3,$
  - $P3 = P2 \rightarrow STOP.$

FSP Lectures, Alan Fekete, University of Sydney (2004)

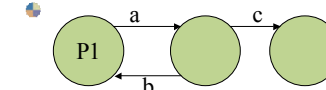
## State machines

- To understand how a process executes, we can build an alternative representation as a state machine or graph (called a *labeled transition system* or LTS)
  - The LTSA tool does this automatically
- Node corresponds to a possible state of the system
- Edge corresponds to the system changing from one state to another in an event
  - label the edge with the name of the action

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Example

```
P1 = ( a ->
      ( b -> P1
      | c -> STOP
      )
    ).
```



Use LTSA to explore

FSP Lectures, Alan Fekete, University of Sydney (2004)

## “Understanding” the model

- You must be able to convert easily between representations (by hand, for easy ones)
  - Eg give LTS graph for FSP process
  - Eg give FSP process for LTS graph
  - Eg give trace set for FSP process
  - Eg give FSP process for trace set
  - Eg give LTS for trace set
  - Eg give trace set for LTS
- Also give any representation from a description of a real system

FSP Lectures, Alan Fekete, University of Sydney (2004)

## More details of FSP

- Some features of the syntax of FSP
  - choice
    - prefix
    - recursion for cycles
  - Indexing
  - Still to come: composition!
- Using the LTSA tool
  - check syntax
  - draw LTS
  - produce traces

FSP Lectures, Alan Fekete, University of Sydney (2004)

## STOP

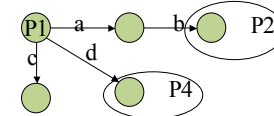
- STOP is the simplest process
  - it represents a system that never does anything
    - eg one in deadlock
- $P1 = STOP.$



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Choice

- A system with several possible next steps can be represented as a choice
  - each branch has one or more actions and then another process which shows how the system behaves after doing those actions
  - branches separated by vertical bars, enclosed in ()
- $P1 = ( a \rightarrow b \rightarrow P2 \mid c \rightarrow STOP \mid d \rightarrow P4 ).$



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Warnings

- Don't omit the parentheses around the choice
  - $P1 = a \rightarrow b \rightarrow STOP \mid c \rightarrow P2$  is wrong
- Each branch must have some actions before one expression
  - $P1 = (P2 \rightarrow a \mid b \rightarrow P3)$  is wrong
  - $P1 = (a \rightarrow P2 \rightarrow b \mid c \rightarrow P3)$  is wrong
  - $P1 = (a \rightarrow b \mid c \rightarrow P2)$  is wrong

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Special case: action prefix

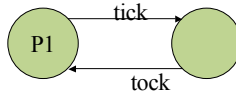
- A choice can have a single branch
  - still need parentheses around it!
- Represents a system whose first action is fixed
- $P1 = (a \rightarrow b \rightarrow STOP).$



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Recursion

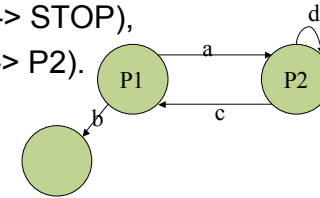
- When a process name occurs after an action, on the right-hand side of an equation, it represents a transition to the start state of the corresponding LTS
- If the process is the same as on the left side, this means a cycle in the LTS
- $P1 = (\text{tick} \rightarrow \text{tock} \rightarrow P1).$



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Multiprocess Recursion

- Each process name always represents the same state, in a single set of equations
- $P1 = (a \rightarrow P2 \mid b \rightarrow \text{STOP}),$
- $P2 = (c \rightarrow P1 \mid d \rightarrow P2).$



FSP Lectures, Alan Fekete, University of Sydney (2004)

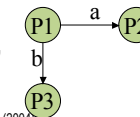
## Non-deterministic choice

- When several branches start with the same action, then the system can follow either branch
  - no way to know which will be taken in a particular execution
- This can model randomness or unpredictable failures

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Expressiveness of FSP

- For any LTS, you can write expressions in FSP to define it
  - several different expressions can define the same LTS
- Give a name to each node, and define it by a choice, showing all transitions out of that node
- $P1 = (a \rightarrow P2 \mid b \rightarrow P3....),$



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Expressiveness of FSP

- Experience shows that while simple FSP can say anything, it becomes very tedious to have a separate name for each possible action and each possible state
  - eg deposit1dollar, deposit2dollars, deposit10cents, etc
  - eg VMWITH1DOLLARSTORED, VMWITH10CENTSSTORED, etc
- Variables that remember state are really useful!

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Indexed Actions

- A family of related actions can be named as action[v] where v is a variable with a predefined range which is finite
  - range given in the action expression eg pushnumber[b: 0..9]
    - or range given before eg ... b: 0..9 ... pushbutton[b]
    - useful trick: define range in one place, so it can be changed easily: range R = 0..9 pushbutton[b:R]

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Indexed processes

- Similarly, define a family of processes whose names are  $P[v:R]$ 
  - on the righthand side, use v also in other processes, recursive mentions of P, and/or in indexed actions
  - $P = P[0]$ ,
  - $P[v:0..2] = (\text{count}[v] \rightarrow P[v+1] \mid \text{restart} \rightarrow P[0])$ ,
  - $P[3] = \text{STOP}$ .

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Guarded actions

- A branch of a choice can be guarded by an expression involving the variables already defined
- This means: the process will not take that branch unless the expression is true
  - this is used to disable activity
  - especially for dealing with boundary of the range while keeping uniform text
  - eg  $P = P[0]$ ,
  - $P[v:0..3] = (\text{when } (v < 3) \text{ count}[v] \rightarrow P[v+1])$ .

FSP Lectures, Alan Fekete, University of Sydney (2004)

## LTSA features

- Standard file menu
  - stores FSP process definitions in a file
  - open a previously stored file
- Build menu
  - compile: check syntax then convert to LTS graph
- Run menu: animate (produce a trace)
- Check properties

FSP Lectures, Alan Fekete, University of Sydney (2004)

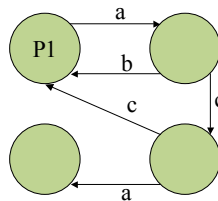
## Exercise

- Draw LTS, and give three traces, for
- $P1 = ( a \rightarrow b \rightarrow P2$   
 $\quad | c \rightarrow STOP$   
 $\quad | c \rightarrow P1),$   
 $P2 = (d \rightarrow P1).$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Exercise

- Give FSP, and three traces, for



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Overview

- Syntax and state machine model of an FSP Process
- *Composition of Processes*
- Sharing and Mutual Exclusion
- Deadlocks
- Using FSP to Model and Design Java Classes
- Dining Philosophers Example

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Key ideas of composition

- Build a process by combining simpler pieces
  - syntax (FSP text)
  - semantics (LTS representation)
    - leads to set of traces
- Issues with alphabets

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Syntax

- A composite process can be defined from several previously defined processes, separated by double-pipes (“||”) and enclosed in parentheses
- The equation for the composition has a double-pipe before the name of the composite process
- Eg  $||PCOMPOSITE = (P1 || P2 || P3).$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Warning

- You can't use a composite process within choice
  - $P = (a \rightarrow (Q || R)).$  is wrong
  - instead, you must define simple processes, then compose them
    - however, you can use composite process within a composition
    - $||P = (P1 || P2).$
    - $||Q = (P || P3).$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## LTS Semantics

- To understand how a composite process behaves, we need to form its LTS (what nodes, what transitions)
  - this is based on the LTS for each component process
- The rules depend on knowing the alphabet of each component
  - that is, what actions are possible for the component

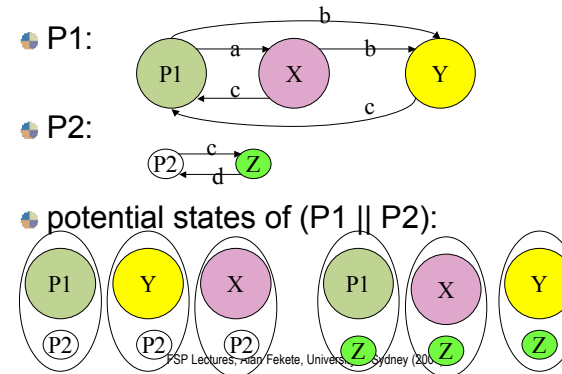
FSP Lectures, Alan Fekete, University of Sydney (2004)

## States of the composition

- A state of the composite process is formed by having each component in one of the states for that component
  - The composition state space is potentially a Cartesian product of separate process state spaces
  - not all combinations may actually occur

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Example



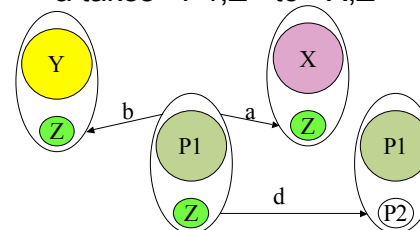
## Non-shared actions

- If one component can take a step by an action which is not shared with other components (ie not in their alphabet)
  - then composite can take this action too
  - state of that component changes appropriately
  - state of other components does not change

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Example

Since *a* takes P1 to X, in the composition *a* takes  $\langle P1, Z \rangle$  to  $\langle X, Z \rangle$



FSP Lectures, Alan Fekete, University of Sydney (2004)

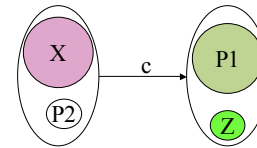
## Shared actions

- If all components that share an action can take it
  - then composite can take this action too
  - state of all components that share the action change simultaneously
  - state of other components (if any) does not change

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Example

- Since  $c$  takes  $X$  to  $P1$ , and also takes  $P2$  to  $Z$ , in the composition  $c$  takes the state  $\langle X, P2 \rangle$  to  $\langle P1, Z \rangle$



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Shared actions block

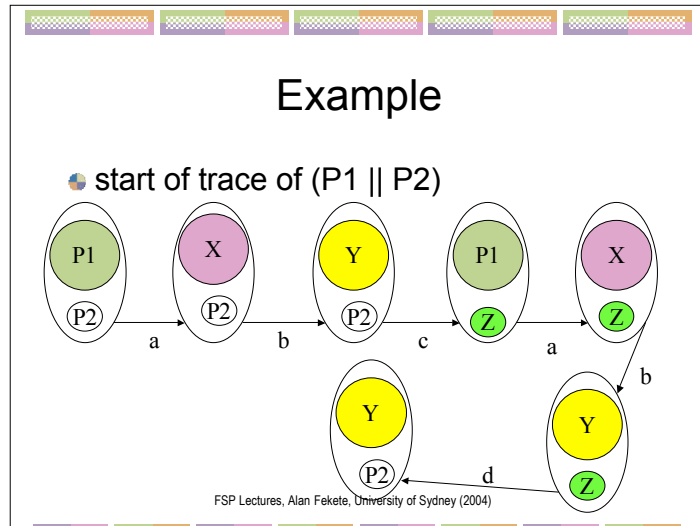
- If there is a shared action, but some component can't take it
  - then composite can't take it either
    - this might block progress in other components which could take this step if they were by themselves
  - eg there is no  $c$  step from state  $P1$ , so there is no  $c$  step in composite from state  $\langle P1, P2 \rangle$ 
    - even though  $c$  takes  $P2$  to  $Z$  when  $P2$  is isolated

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Interleaving traces

- If we take a trace of the composite process, and ignore all actions except those in alphabet of one component,
  - then what is left is a trace of that component
- If all processes have disjoint alphabets (ie no shared actions at all)
  - then every trace of composite is just an interleaving of traces of the components

FSP Lectures, Alan Fekete, University of Sydney (2004)



### Example

- $PA = (u \rightarrow v \rightarrow PA)$ .
  - trace:  $u \rightarrow v \rightarrow u \rightarrow v \rightarrow u \rightarrow v \dots$
- $PB = (w \rightarrow PB \mid x \rightarrow STOP)$ .
  - traces:  $w \rightarrow w \rightarrow \dots w \rightarrow x$
- $PC = (PA \parallel PB)$ .
- Some traces of PC
  - $u \rightarrow w \rightarrow v \rightarrow u \rightarrow v \rightarrow w \rightarrow w \rightarrow u \rightarrow x \rightarrow v \dots$
  - $w \rightarrow u \rightarrow v \rightarrow w \rightarrow u \rightarrow w \rightarrow x \rightarrow v \rightarrow u \dots$

FSP Lectures, Alan Fekete, University of Sydney (2004)

### Model of code with threads

- range  $R = 0..1$
- $FLIP1 = (init1 \rightarrow read1[v:R] \rightarrow write1[1-v] \rightarrow STOP)$ .
- $FLIP2 = (init2 \rightarrow read2[v:R] \rightarrow write2[1-v] \rightarrow STOP)$ .
- $VAR = VAR[0]$ ,
- $VAR[v:R] = (read1[v] \rightarrow VAR[v] \mid read2[v] \rightarrow VAR[v] \mid write1[w:R] \rightarrow VAR[w] \mid write2[w:R] \rightarrow VAR[w])$ .
- $SYS = (FLIP1 \parallel FLIP2 \parallel VAR)$ .
  - note: some processes model threads, and some model shared variables
- show trace corresponding to lost update

FSP Lectures, Alan Fekete, University of Sydney (2004)

### Exercise

- $QA = (a \rightarrow (b \rightarrow QA \mid c \rightarrow STOP) \mid b \rightarrow STOP)$ .
- $QB = (b \rightarrow STOP \mid c \rightarrow d \rightarrow STOP)$ .
- $Q = (QA \parallel QB)$ .
- Is  $a \rightarrow c \rightarrow d$  a trace of Q?
- Is  $a \rightarrow b \rightarrow a \rightarrow c$  a trace of Q?

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Issues with alphabets

- The semantics of composition depend on the alphabet of the components
  - shared actions are treated quite differently than non-shared actions
- We sometimes need to “tweak” process definitions to get the composition we intend
- We also want to build systems with multiple components described by a common definition

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Summary I

- Extension
  - puts an action in the alphabet even though not mentioned in the process text
- Relabeling
  - changes one action name to another
- Hiding
  - removes an action from the alphabet
  - gives corresponding transitions a new private name

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Summary II

- Process Labeling
  - puts extra identifier in front of each action
  - allows multiple components from same definition
- Parameterised processes
  - allow easy change of the scale of the system

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Process Alphabet

- Normally the alphabet of a process is all the action names that are mentioned in the definition
  - this includes those mentioned in “local processes” (ie, descriptions of states within the process; separated by commas in FSP text)
    - expand indexed actions/processes to see what action names occur in the text!
  - not all these actions need be present in traces (if some states are unreachable)
- The alphabet of a composite is the union of the alphabets of the components

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Alphabet example

- $P1 = (a[i:0..2] \rightarrow P[i] \mid b \rightarrow Q),$
- $P[i:0..2] = (\text{when } (i < 2) \text{ } c[i+1] \rightarrow P[i+1]),$
- $Q = (d \rightarrow P[0] \mid d \rightarrow \text{STOP}).$
  
- Alphabet is  $\{a[0], a[1], a[2], b, c[1], c[2], d\}$ 
  - note:  $c[0]$  is not in alphabet

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Alphabet example

- $P1 = (b \rightarrow a \rightarrow P1),$
- $Q = (c \rightarrow Q).$
  
- Alphabet is  $\{a, b, c\}$ 
  - note:  $c$  is in alphabet, but only trace is
    - $b \rightarrow a \rightarrow b \rightarrow a \rightarrow b \rightarrow a \dots$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Extension

- To make a process with more actions in alphabet than are explicit in the definitions
  - why? to allow the process a veto on these actions in a composition
- Syntax:  $P + \{x, y\}.$ 
  - process with same LTS as  $P$ , but alphabet also has  $x$  and  $y$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Extension example

- $\text{INC} = (\text{read}[v:0..3] \rightarrow \text{write}[v+1] \rightarrow \text{INC} \mid \text{read}[4] \rightarrow \text{exception} \rightarrow \text{STOP}) + \{\text{write}[0]\}.$
- $\text{VAR} = \text{VAR}[0],$
- $\text{VAR}[v:0..4] = (\text{read}[v] \rightarrow \text{VAR}[v] \mid \text{write}[w:0..4] \rightarrow \text{VAR}[w]).$
- $\|\text{SYS} = (\text{INC} \|\ \text{VAR}).$
  
- If we omit the extension, then  $\text{SYS}$  can have  $\text{write}[0]$  step at any time
  - This would not be a good model of reality of a thread which repeatedly increments a variable

FSP Lectures, Alan Fekete, University of Sydney (2004)

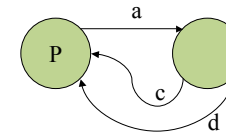
## Relabeling

- Replace one action name by another, everywhere within a process
  - why? usually, to take a predefined process and bring action names to match those needed for composition
- Syntax:  $P / \{ \text{newname}/\text{oldname} \}$ 
  - multiple renamings can be done together
  - even give multiple new names for one old
  - note: in composition, relabeling is done first

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Relabeling example

- $P = (a \rightarrow b \rightarrow P) / \{c/b, d/b\}$ 
  - syntactic sugar: “ $\{ \{c,d\}/b \}$ ”
- equivalent to
- $P = (a \rightarrow (c \rightarrow P \mid d \rightarrow P))$ .



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Hiding

- Change name on all transitions of a given action, to something that is not shown in traces
  - new name written as “tau” but actually is a private name not used in any other process
  - thus can’t be shared in composition
  - why? to ignore internal interactions between components
- Syntax:  $P \setminus \{a,b\}$ . All a and b transitions are changed to tau
- Alternative syntax:  $P @ \{c,d\}$ . All transitions except c and d are changed to tau

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Multiple similar components

- When we model a system, we often want to describe a *type* of component, and then build the system by composing several components of that type (perhaps with other types too)
  - eg network has several PCs and one switch and one LAN and one printer
  - eg bank has several tellers, several accounts, several customers

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Motivation

- We need some way to produce several different components on the same pattern
- NB what's wrong with saying  

$$\parallel \text{SYS} = (\text{PC} \parallel \text{PC} \parallel \text{LAN} \parallel \text{SWITCH} \dots)$$
 answer: PCs would run in lock step; all transitions in two PCs would occur simultaneously due to shared action names!

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Process labeling

- Make a component which is like P but change the action names in a consistent way
- Syntax:  $a:P$  is a component which behaves just like P except that all actions have names with added "a." in front
  - eg action "read" in P becomes "a.read" in  $a:P$
  - So  $\parallel \text{SYS} = (\text{pc1:PC} \parallel \text{pc2:PC} \parallel \text{LAN} \dots)$
  - warning: labeling can only be done within composition (not in simple process)

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Component families

- Use indexed labels for a family of similar components
- $\parallel \text{SYS} = (\text{pc}[0]:\text{PC} \parallel \text{pc}[1]:\text{PC} \parallel \text{pc}[2]:\text{PC} \parallel \dots)$
- equivalent is  

$$\parallel \text{SYS} = (\text{forall } [i:0..3] \text{ pc}[i]:\text{PC} \parallel \dots)$$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Parameterised processes

- Sometimes a system consists of an unknown number of similar components
- So we define a system with a parameter determining the number of components
  - LTSA insists that we give the parameter an exact value before we use it
- eg  $\parallel \text{SYS}(N=3) = (\text{forall } [i:0..N] \text{ pc}[i]:\text{PC} \parallel \dots)$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Families and relabeling

- Sometimes, some actions should be shared between members of a family, in order to be performed simultaneously in all components
  - use process labels which makes separate action names
  - then relabel actions that should be shared to something common

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Example

- $\text{THREAD} = (\text{doit} \rightarrow \text{THREAD} \mid \text{cancel} \rightarrow \text{STOP}).$
- Suppose we want each thread to have its separate `doit`, but all should take cancel step together
- $\|\text{SYS} = (\text{forall } [i:\mathbb{R}] \text{thread}[i]:\text{THREAD}) / \{ \text{forall } [i:\mathbb{R}] \{ \text{cancelall}/\text{thread}[i].\text{cancel} \} \}.$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Overview

- Syntax and state machine model of an FSP Process
- Composition of Processes
- *Sharing and Mutual Exclusion*
- Deadlocks
- Using FSP to Model and Design Java Classes
- Dining Philosophers Example

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Summary

- FSP Model for Threads using a class that has no synchronization
  - model each thread
  - model shared data
  - system is composition
- FSP Model for Threads using a class with synchronization
  - also model lock

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Threads sharing data

- Model each thread as FSP process
  - actions call\_method and return\_method[v] (if needed)
  - also actions var.read[v], var.write[v]
- Model each variable as FSP process
 

```
VAR = VAR[INIT],
VAR[v:R] = (read[v] -> VAR[v]
           |write[w:R] -> VAR[w]).
```
- system is composition (threads and variables)
  - label components properly for composition

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Example

```
class BankAccount {
  private int balance;
  public void deposit(int amt) {
    balance = balance+amt;
  }
  public void withdraw(int amt) {
    if (balance>=amt)
      balance = balance - amt;
  }
  public int getBalance() {
    return balance;
  }
}
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model (Simplified)

- THREAD1 = (call\_deposit[5] ->
 

```
balance.read[v:R] ->
  ( when (v<=MAX-5) balance.write[v+5] ->
    return_deposit -> STOP))+{balance.VarAlpha}.
```
- THREAD2 = (call\_withdraw[3] ->
 

```
balance.read[v:R] ->
  ( when (v>=3) balance.write[v-3] ->
    return_withdraw -> STOP))+{balance.VarAlpha}.
```
- ||SYS = (t1:THREAD1 || t2:THREAD2
 

```
|| {t1,t2}::balance:VAR).
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model (full) I

```
const MAX = 5
const AMT1 = 2
const AMT2 = 1
const INIT = 2
range R = 0..MAX
set VarAlpha = {read[v:R], write[v:R]}

VAR = VAR[INIT],
VAR[v:R] = (read[v] -> VAR[v]
           |write[w:R] -> VAR[w]).
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model (full) II

```

THREAD1 = ( call_deposit[AMT1] ->
            balance.read[v:R] ->
            (when (v<=MAX-AMT1)
              balance.write[v+AMT1] ->
              return_deposit -> STOP)
            )+{balance.VarAlpha}.
  
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model (full) III

```

THREAD2 = ( call_withdraw[AMT2] ->
            balance.read[v:R] ->
            (when (v>=AMT2)
              balance.write[v-AMT2] ->
              return_withdraw ->
              STOP)
            )+{balance.VarAlpha}.
  
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model (full) III

```

||SYS = (t1:THREAD1 || t2:THREAD2
         || {t1,t2}::balance:VAR).
  
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Notes

- Make sure to get proper action name matching
  - label each process, and check that intended sharing happens (and extra sharing doesn't)
    - Typically, each read and write is labeled as thread.variable.read[value], etc
  - use alphabet extension to ensure that each read and write action (with every possible label) is shared so it can't happen spontaneously

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Model versus reality I

- In model we need to keep variable ranges finite (and small)
- In model we must fix one initial value, one argument for each method, etc
- In model we have only small finite number of threads (often just two!)
  - and we must decide in advance which method each thread executes
- In model we often leave out local variables

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Model versus reality II

- In model we treat read and write as atomic (single actions)
  - in reality this is true only for int/float/ref variables;
  - access to long/double is done as two separate word operations, and so thread preemption can happen between!
- In model we leave out most activity except for access to shared variables (and perhaps call/return)

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model with synchronization

- Recall that Java allows code in a method to be in a synchronized block
  - thread will be blocked until it can acquire the lock associated with the object instance named
    - implicitly, object is "this" for a **synchronized** method
  - thread will release the lock at the end of the block
- We can model this with obj.acquire and obj.release actions in the process which represents the thread
- Have an FSP process to represent the lock itself  
LOCK = (acquire -> release -> LOCK).

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model (part)

- THREAD1 = (act.acquire ->
 

```

call_deposit[AMT1] ->
balance.read[v:R] ->
(when (v<=MAX-AMT1)
  balance.write[v+AMT1] ->
  return_deposit ->
  act.release -> STOP)
){balance.VarAlpha}.
      
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model (part) II

- $||\text{SYS} = (\text{t1:THREAD1} || \text{t2:THREAD2} || \{\text{t1,t2}\}::\text{balance:VAR} || \{\text{t1,t2}\}::\text{act:LOCK}).$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Model versus reality

- Our LOCK model ignores thread identity
  - Java allows thread to acquire lock while already holding the lock itself;
  - also Java prevents release by a thread other than the one holding the lock!
- Our model allows no release or extra release
  - Java syntax forces proper placement of release
- Textbook approach is even less accurate, since it models lock with variable instead of with instance of the class whose method is being called

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Checking interference

- Define a TESTER process that observes certain actions of the system, and moves to special state ERROR if something is wrong
- Compose TESTER with system
- Use LTSA tool, which will find whether there exists any trace leading to ERROR

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Example

- $P = (b \rightarrow Q1 \mid c \rightarrow Q2),$   
 $Q1 = (a \rightarrow \text{STOP} \mid c \rightarrow Q2),$   
 $Q2 = (b \rightarrow Q1 \mid a \rightarrow d \rightarrow Q1).$
- Does every trace have at least one b before the first a?
    - $c \rightarrow b \rightarrow c \rightarrow a \rightarrow b.$  would be OK (b then a)
    - $c \rightarrow d \rightarrow c \rightarrow c.$  would be OK (no a, no b)
    - $c \rightarrow b \rightarrow d \rightarrow b.$  would be OK (b but no a)
    - $c \rightarrow d \rightarrow a \rightarrow b.$  would NOT be OK (a before any b)

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Checking example

- Extend system with  

$$\text{TESTER} = ( a \rightarrow \text{ERROR} \\ | b \rightarrow \text{OK}),$$

$$\text{OK} = (a \rightarrow \text{OK} \\ | b \rightarrow \text{OK} \\ | \text{ok} \rightarrow \text{OK}).$$

$$\parallel \text{SYS} = (P \parallel \text{TESTER}).$$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Using LTSA to check

- use Compose item on Build menu
  - output reports “property TESTER violation”
  - this means that there is a trace leading to ERROR state (shown as “-1” on diagram)
  - (sometimes, you can find the trace if you examine diagram of SYS by eye)
- then Check -> Safety
  - LTSA will show you a bad trace
  - Check->Run will then step through it

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Checking interference

- What should TESTER be?
- For interference between writers
  - TESTER reads shared data after all threads are finished
  - move to ERROR if value is wrong
- For interference between reader and writer
  - TESTER observes return value from reader
  - move to ERROR if value is wrong
    - note: several different values would be correct, depending on whether or not writer had run first

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Example (WW interference)

- From previous lecture (deposit and withdraw)
 
$$\text{TESTER} = (\text{done} \rightarrow \text{CHECK}[\text{INIT} + \text{AMT1} - \text{AMT2}],$$

$$\text{CHECK}[v:R] = (\text{display.balance.read}[u:R] \rightarrow$$

$$(\text{when } (u=v) \text{ right} \rightarrow \text{IDLETEST}$$

$$| \text{when } (u \neq v) \text{ wrong} \rightarrow \text{ERROR})),$$

$$\text{IDLETEST} = (\text{idle} \rightarrow \text{IDLETEST})$$

$$+ \{\text{display.balance.VarAlpha}\}.$$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Points to note

- TESTER alphabet includes done action (which we must add to each THREAD)
- TESTER becomes CHECK after seeing done
  - index is the expected value in the variable
- TESTER is written to loop forever when things are OK
  - similarly, each other process is also written without STOP (looping after done)
  - this is not essential, but it makes LTSA tool more useful in telling you the bad trace

FSP Lectures, Alan Fekete, University of Sydney (2004)

## The idle loop

- LTSA reports on whether or not there is a trace leading to ERROR state
- For debugging and understanding, we really want to know what the bad trace is, not simply whether it exists
  - LTSA Check->Safety will give a shortest trace that leads to *either* ERROR or STOP
  - to force it to tell you the trace to ERROR, you need to make sure no trace leads to STOP
    - that means making each component never STOP

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Modified THREAD process

```

THREAD1 = ( call_deposit[AMT1] ->
            balance.read[v:R] ->
            (when (v<=MAX-AMT1)
              balance.write[v+AMT1] ->
              return_deposit -> done -> IDLE1)
            ),
IDLE1 = (idle1 -> IDLE1)+{balance.VarAlpha}.

```

Note: done action, idle loop

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Composition with TESTER

```

||SYS = (t1:THREAD1 || t2:THREAD2
         || {t1,t2,display}::balance:VAR
         || TESTER)
        /{done/{t1.done,t2.done}}.

```

- note use of “display” to label read/write by TESTER
- note renaming of done in each thread, to become shared with done in TESTER

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Checking non-interference

- Similarly compose TESTER with the system that includes locking actions
  - also modify thread process definitions to put in done action and idle loop
- LTSA Build->Compose now does *not* report "property TESTER violation"
  - this means that the model has no trace leading to ERROR state
  - that is, every trace of system leaves the correct value in the shared variable
  - provided model is realistic, this means the real program (with synchronized blocks) has no interference

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Example (RW interference)

```
class ComplexAcct {
  private int savingsBal;
  private int chequeBal;
  public void transferToChq (int amt) {
    savingsBal = savingsBal - amt;
    chequeBal = chequeBal + amt;
  }
  public int totalBalance() {
    return savingsBal+chequeBal;
  }
}
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model

```
THREAD1 = (call_transfer ->
  sav.read[v:R] ->
  (when (v>=AMT) sav.write[v-AMT] ->
  chq.read[w:R] ->
  (when (w<=MAX-AMT) chq.write[w+AMT] ->
  return_transfer -> IDLE1))),
IDLE1 = (idle1 -> IDLE1)+{sav.VarAlpha, chq.VarAlpha}.

THREAD2 = (call_bal ->
  sav.read[v:R] -> chq.read[w:R] ->
  return_bal[v+w] -> IDLE2),
IDLE2 = (idle2 -> IDLE2)+{sav.VarAlpha, chq.VarAlpha}.
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## TESTER

```
TESTER = (t2.return_bal[x:R] ->
  ( when (x==2*INIT) right -> IDLE3
  | when (x!=2*INIT) wrong -> ERROR)),
IDLE3 = (idle3 -> IDLE3).

||SYS = (t1:THREAD1 || t2:THREAD2
  || {t1,t2}::sav:VAR
  || {t1,t2}::chq:VAR
  || TESTER).
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Exercise

- Design model and tester for the case of transferToChq() and depositToChq()
- ```
public void depositToChq(int amt) {
    chequeBal = chequeBal+amt;
}
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Overview

- Syntax and state machine model of an FSP Process
- Composition of Processes
- Sharing and Mutual Exclusion
- *Deadlocks*
- Using FSP to Model and Design Java Classes
- Dining Philosophers Example

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Summary

- Recall deadlock concept
- Deadlock in FSP
- Preventing Deadlock in Java
- Deadlock with wait()

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Transfer method

```
public class Account {
    private int balance;
    /** transfer method */
    public synchronized void transfer
(Account target, int amount) {
        synchronized (target) {
            this.balance -= amount;
            target.balance += amount;
        }
    }
}
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Log of transfer

- t1: call a.transfer(b)
  - t1: get lock on a (synchronized method)
  - t2: call b.transfer(a)
  - t2: get lock on b (synchronized method)
  - t2: block waiting for lock on a
  - t1: block waiting for lock on b
- DEADLOCK!**

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Transfer in FSP

- THREAD1 = (a.acquire -> b.acquire -> ... -> b.release -> a.release -> STOP).
- THREAD2 = (b.acquire -> a.acquire -> ... -> a.release -> b.release -> STOP).
- ||SYS = (t1:THREAD1 || t2:THREAD2 || {t1,t2}::a:LOCK || {t1,t2}::b:LOCK ||...).

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Trace to deadlock

- t1.a.acquire -> t2.b.acquire
- Now no action can happen in SYS
  - recall: a shared action must happen in every component where its in the alphabet
- t1.b.acquire can happen in t1:THREAD1
  - but not in {t1,t2}::b:LOCK
- t2.b.release can happen in {t1,t2}::b:LOCK
  - but not in t2:THREAD2

FSP Lectures, Alan Fekete, University of Sydney (2004)

## In general

- Deadlock is a state where the system can't take any action (ie overall state is like STOP)
  - but each component by itself could take a step (ie no component is in local STOP)

FSP Lectures, Alan Fekete, University of Sydney (2004)

## LTSA checks

- When composing a system, LTSA reports whether there is a trace that leads to STOP
  - as well as whether there is a trace that can lead to ERROR
- Check -> Safety will show the shortest trace of either sort
- To be useful, make sure no component can STOP
  - have an idle loop in each component

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Preventing deadlock in Java

- This discussion deals with deadlocks involving lock acquisition only
- The simplification of blocking for a lock is that only ONE other process can be holding the lock
  - so each thread's progress depends on exactly ONE other thread executing as far as the release of that lock
  - compare with wait()

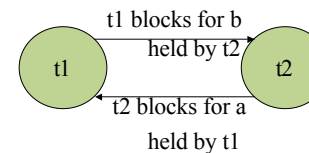
FSP Lectures, Alan Fekete, University of Sydney (2004)

## Waits-for graph

- Draw a graph whose nodes are threads in the system
- If thread t1 is blocking for a lock held by thread t2, then put an edge from t1 to t2
- Deadlock = a cycle in this graph

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Transfer example



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Deadlock principles

- Deadlock involves threads which block trying to acquire resources
  - eg acquire lock for synchronized block of code
- Deadlock involves blocking while still holding other locks
- A thread only releases a resource when it has progressed to the release statement
  - eg lock can't be pre-empted
- Deadlock requires a cycle of blocking

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Sequential synchronized code

- In analysis of possible deadlock, we can ignore threads which hold only one lock at each separate time
  - no other thread could be blocked for t if t itself is blocked
  - so t is not in any cycle
- eg `synchronized(o1) {`

```

      // do something with o1
    }
    synchronized(o2) {
      // do some other thing with o2
    }
  
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Nested locking

- We must worry about situations where a thread can block for a resource o2 while holding a resource o1, like

```

synchronized(o1) {
  synchronized(o2) {
    // do something
  }
}
  
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Indirect nested locking

- Requesting a lock while holding another can be hidden in indirect calls

In class A:

```

B x;
synchronized(o1) {
  x.method1();
}
  
```

In class B

```

void method1() {
  synchronized(o2) {
    // do something with o2
  }
}
  
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Coarse-grained locks

- Avoid deadlock by locking a large part of the system at once
  - eg lock whole bank instead of separately locking each account
- If you lock coarsely enough, each thread will need only one lock
  - no deadlock!
  - however, threads often will wait unnecessarily

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Resource ordering

- Place some fixed agreed order on the resources
- Make sure a thread never asks for a resource if it already holds a later-ordered resource
- Then no cycle can exist (so no deadlock!)
  - if t1 waits-for t2 then the latest-ordered resource held by t1 is less than the latest-ordered resource held by t2
    - because when t1 is blocked on resource a held by t2, then the latest-ordered lock held by t1 is less than a and the latest-ordered resource held by t2 is greater than or equal to a

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Examples

- Lock bank accounts in order of acctId
- Lock pages in a cache in order from position 1 to position N
- Lock objects down a tree
  - lock root, then lock a child of root, then lock a child of child of root, etc

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Warning

- The rule does *not* say either of:
  - make sure that a thread never asks for a resource if it has previously held a later-ordered resource
  - make sure a thread holds all items earlier than x before it requests resource x

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Deadlock with wait()

- It is much harder to analyse deadlock possibilities involving wait() and notifyAll()
  - a thread which is waiting may resume due to any one of a number of other threads
  - after resuming, a thread may wait again
    - this can lead to lockout rather than deadlock
  - resource ordering does not protect you!

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Overview

- Syntax and state machine model of an FSP Process
- Composition of Processes
- Sharing and Mutual Exclusion
- Deadlocks
- *Using FSP to Model and Design Java Classes*
- Dining Philosophers Example

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Summary

- Designing OO classes for concurrency
  - Monitors
    - a class where all public methods are run with exclusive access
      - synchronized
    - can be modeled more simply, with each method modeled as single action
  - Guarded suspension
    - a method that blocks till some condition is valid
- Examples

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Monitors

- In essence, a monitor is a thread-safe class
  - it is accessed through operations
    - called from multiple threads
  - each operation runs indivisibly
    - thus it seems that each operation is a single action

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Implementation of a monitor

- Make every method synchronized
  - actually, only need to have locking on methods that can be called from outside the object itself
- If the code uses any mutable class attributes (“static”) then you must also protect these in synchronized block using Class object

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Modelling a monitor

- Treat a monitor instance as FSP process with one action per method
  - appropriate return value
  - appropriate next state
- Example: Bank account monitor  

$$\text{ACCT}[v:R] = (\text{deposit}[w:R] \rightarrow \text{ACCT}[v+w]$$

$$|\text{balance}[v] \rightarrow \text{ACCT}[v]).$$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Warning

- You should only use the simple FSP model after you have shown that the methods are properly free of interference
- That is, first use complex model with threads and locks
  - show this can simplify to the model with a single action per method

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Guarded suspension

- How to design a method that has a precondition
  - eg remove from collection needs “collection is not empty”
- In traditional design, you prevent method from running when precondition is false
  - client checks before call
  - or, class throws exception or returns error status
- In multithreaded environment, you might just wait till another thread makes condition true

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Examples

- Resource pool
  - obtain: waits till a resource is free
    - when (available>0) obtain ->...
- Sales agent
  - pay\_bonus: waits till target is achieved
    - when (sales>=target) pay\_bonus -> ...
- Work allocation
  - assign\_job: waits till all resources are available
    - when (printer and disk and CPU) assign\_job -> ...

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Code cliche

- ```
synchronized ret_type method(arg_type args)
{
    while (! condition) {
        wait();
    }
    //do the activity;
}

```
- every method that could make condition true must call notifyAll()

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Review of wait/notifyAll

- These are methods of every object
- They can only be called in a synchronized block which holds the lock on the object
- wait(): calling thread is blocked and gives up lock
- notifyAll(): all blocked threads move to queue that is seeking the lock
  - no blocked thread will run until after notifier finishes its synchronized block

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Warnings I

- You should always have wait() inside a while loop
  - the condition may not be true when you resume running
    - even if it were true when notifyAll() was called, another thread may have made condition false again
    - also, Java does not guarantee that you won't resume without anyone calling notifyAll()

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Warnings II

- Always call `notifyAll()` in any method that could make a condition true on which some thread might be waiting
  - it's simplest to call `notifyAll()` whenever relevant fields are changed
    - however, sometimes you are certain a change doesn't make condition true (eg if threads wait for non-empty collection, `replace_elt` need not call `notifyAll()`)
  - if a thread changes state and forgets `notifyAll`, then another thread might wait forever

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Warnings III

- Avoid use of `notify()`
- There is no way to control which waiter is resumed
- With `notifyAll()`, any threads resumed unnecessarily will simply find condition false, and so wait again
  - because wait is inside while loop

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Nested monitor deadlock

- Recall that calling `wait()` on an object `o1` will release the lock on `o1` if the thread has to wait
- But other locks (say on `o2`) held by the thread are not released
- This easily leads to deadlock if the only threads which could call `notify` on `o1` are blocked because they don't have `o2`

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Example

```
In class A:
synchronized(this) {
    x.guarded();
}

In class B:
public synchronized void guarded() {
    while (!cond) {
        wait()
    }
    // do something
    notifyAll();
}
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Preventing nested monitor deadlock

- Do locking and waiting using the outer class

```
In class A:
    x.guarded(A outer);
In class B:
public void guarded(A outer) {
    synchronized(outer) {
        while (!cond) {
            outer.wait()
        }
        // do something
        outer.notifyAll();
    }
}
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Resource Pool

- Several clients need to use a resource, which can be chosen from a collection of equivalent instances
  - eg printers, network connections, file descriptors, cpus (in SMP system)
- After use, the client returns resource to the pool, so other clients can access it

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model

- Abstract by just counting number of available resources, not specific identities

POOL = POOL[INIT],

POOL[n:R] = (when (n>0) obtain -> POOL[n-1]  
|finished -> POOL[n+1]).

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Java code

```
class Pool {
    List resources;

    public synchronized Res obtain() {
        while (resources.isEmpty()) {
            wait();
        }
        return (Res) (resources.removeLast());
    }

    public synchronized void finished(Res r) {
        resources.add(r);
        notifyAll();
    }
}
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Semaphores

- A semaphore is an object which counts the number of a resource available
- Operations are just like in the FSP Model for Pool
- Up (also called V or post or release)
  - $value = value + 1$
- Down (also called P or wait or obtain)
  - when  $(value > 0)$   $value = value - 1$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Barrier

- Parallelizing time-step simulations
  - important for scientific computing
- Each thread calculates next state for part of the system, based on previous state in nearby parts
- Goal: ensure that all parts reach state T, before any thread moves to calculate state T+1

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Multi-party rendezvous in FSP

- In FSP, the problem is trivial because an action can be shared among all the processes
  - but in coding shared action is normally done as call/return which involves just two components
- $THREAD[i,t] = (calculate[i,t] \rightarrow barrier \rightarrow THREAD[i,t+1])$

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Barrier in Java

- Key idea from Doug Lea's book "Concurrent Programming in Java" (2nd ed, p 363)
- ```

class CyclicBarrier {
    synchronized void barrier() {
        reached++;
        if (reached < N) { //not all parties are yet at barrier
            int r = resets;
            do { wait(); } while (resets == r); //wait till someone increases resets
        } else { //all parties are at barrier
            reached = 0;
            ++ resets;
            notifyAll();
        }
    }
}
  
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Waiting for different conditions

- Sometimes there are several conditions on which different threads might be waiting
  - some threads wait till collection is not empty
  - other threads wait till collection is not full
- The simplest design has all waiting on the same queue (associated with the object involved)
  - each checks appropriate condition whenever it's resumed
- Fancy designs have separate "condition variable" for each condition
  - create a new Object to represent each condition
  - each thread will wait on the appropriate object (it must first get lock on that object!)
    - it must still check the condition whenever resumed!
  - deadlock is a real danger (seek lock on condition variable while holding lock on monitor object)

## Overview

- Syntax and state machine model of an FSP Process
- Composition of Processes
- Sharing and Mutual Exclusion
- Deadlocks
- Using FSP to Model and Design Java Classes
- *Dining Philosophers Example*

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Summary

- Explain the "dining philosophers problem"
  - why pay attention to this?
- A simple approach
  - Model it in FSP
  - Express it in Java
  - it can deadlock
- Ways to avoid deadlock

FSP Lectures, Alan Fekete, University of Sydney (2004)

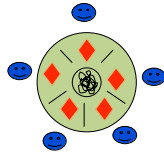
## Dining philosophers

- Five philosophers are sitting around a circular table
- Each philosopher spends some time thinking, then some time eating, then thinking, etc

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Physical arrangement

- On the table is a big bowl of spaghetti
  - each philosopher has a plate
  - there are 5 forks, one between each pair of neighbouring philosophers



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Rules

- In order to eat, a philosopher needs to use two forks
  - one on his left, and another on his right
- A fork can't be used by both neighbouring philosophers at once
  - however, once one philosopher has finished using it, the neighbour can then use it

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Relevance?

- Is this problem *realistic*?
  - not a good description of real philosophers
  - not a good model of any situation likely in a computer system
- Is it *informative*?
  - it has some features of many real resource management situations
  - it's a testbed for approaches to more realistic problems
- The "cutesy" story is typical of theoretical concurrent computing

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Simple approach

- Each philosopher tries to pick up the fork on the right
  - if the fork is being used, the philosopher waits till its available
- Once the philosopher has the right fork, they try to pick up the left one
  - again, wait till it's available
- Once both forks are held, eat
  - once eating is over, put down both forks, then think

FSP Lectures, Alan Fekete, University of Sydney (2004)

## FSP Model

```

FORK = (pickup -> putdown -> FORK).
PHIL = (think -> right.pickup ->
left.pickup -> eat -> left.putdown ->
right.putdown -> PHIL).
||SYS = (forall[i:0..4] p[i]:PHIL
|| forall[i:0..4] {p[i],p[(i+1)%5]}:f[i]:FORK)
/{ forall[i:0..4]
{p[i].right/p[i].f[i],p[i].left/p[i].f[(i+4)%5]}}.

```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Java code

- There are two different ways to provide a Java representation of the dining philosophers
  - have a Fork with pickup() and putdown() methods
  - recognize that a Fork acts like a lock, so just acquire a lock on the Fork

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Java code essence vl

```

class Fork {
    private boolean inUse = false;
    public synchronized void pickup() {
        while(inUse) wait();
        inUse = true;
    }
    public synchronized void putdown() {
        inUse = false;
        notifyAll();
    }
}

```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Java code essence vl

```

class Philosopher implements Runnable {
    private int id;
    private Fork left;
    private Fork right;
    private Spag bowl;
    public Philosopher(int id, Spag bowl, Fork left, Fork right) {
        this.id = id; this.bowl = bowl; this.left = left; this.right = right;
    }
    public void run() {
        while(true) {
            think();
            right.pickup(); left.pickup();
            bowl.eat();
            left.putdown(); right.putdown();
        }
    }
}

```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Java code essence vII

```

class Philosopher implements Runnable {
    // private fields and constructor as in Ib
    public void run() {
        while(true) {
            think();
            synchronized(right) {
                synchronized(left) {
                    bowl.eat();
                }
            }
        }
    }
}

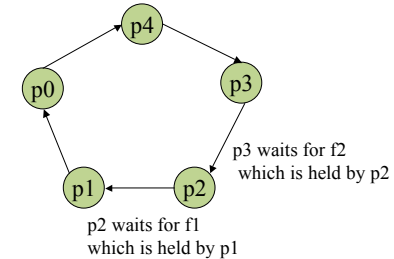
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Deadlock!

Trace to DEADLOCK:

- p.0.think
- p.0.right.pickup
- p.1.think
- p.1.right.pickup
- p.2.think
- p.2.right.pickup
- p.3.think
- p.3.right.pickup
- p.4.think
- p.4.right.pickup



FSP Lectures, Alan Fekete, University of Sydney (2004)

## Avoiding deadlock I

- Use coarse-grained locking
- i.e. have a single lock that allows one philosopher at a time to use any forks
  - the others will block on that lock
  - the one that has that lock will certainly succeed when getting both forks
  - the lock is released only after putting down both forks

FSP Lectures, Alan Fekete, University of Sydney (2004)

FORK = (pickup -> putdown -> FORK).  
 LOCK = (acquire -> release -> LOCK).  
 PHIL = (think -> coarse.acquire -> right.pickup ->  
 left.pickup -> eat -> left.putdown ->  
 right.putdown -> coarse.release -> PHIL).

```

||SYS = (forall[i:0..4] p[i]:PHIL || {p[i:0..4]}::coarse:LOCK
|| forall[i:0..4] {p[i],p[(i+1)%5]}::f[i]:FORK)
/forall[i:0..4] {p[i].right/p[i].f[i],p[i].left/p[i].f[(i+4)%5]}).

```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Avoiding deadlock II

- Make some philosopher left-handed
  - This philosopher tries to pickup the left fork first
- The others are right-handed
- This is equivalent to a resource-ordering strategy
  - which order?

FSP Lectures, Alan Fekete, University of Sydney (2004)

```

FORK = (pickup -> putdown -> FORK).
PHIL = (think -> right.pickup ->
  left.pickup -> eat -> left.putdown ->
  right.putdown -> PHIL).
LHPHIL = (think -> left.pickup ->
  right.pickup -> eat -> right.putdown ->
  left.putdown -> LHPHIL).
||SYS = (forall[i:0..3] p[i]:PHIL || p[4]:LHPHIL
  || forall[i:0..4] {p[i],p[(i+1)%5]}::f[i]:FORK)
  /{forall[i:0..4] {p[i].right/p[i].f[i],p[i].left/p[i].f[(i+4)%5]}}.
  
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Avoiding deadlock III

- Limited permits
- Introduce a limit of 4 philosophers at any one time
  - 4 can be trying to get forks then eat
  - once 4 are trying/eating, the fifth must wait

FSP Lectures, Alan Fekete, University of Sydney (2004)

```

FORK = (pickup -> putdown -> FORK).
PERMITS = PERMITS[0],
PERMITS[n:0..4] =(when (n<4) grant -> PERMITS[n+1]
  |free -> PERMITS[n-1]).
PHIL = (think -> permit.grant -> right.pickup ->
  left.pickup -> eat -> left.putdown ->
  right.putdown -> permit.free -> PHIL).

||SYS = (forall[i:0..4] p[i]:PHIL || {p[i:0..5]}::permit:PERMITS
  || forall[i:0..4] {p[i],p[(i+1)%5]}::f[i]:FORK)
  /{forall[i:0..4] {p[i].right/p[i].f[i],p[i].left/p[i].f[(i+4)%5]}}.
  
```

FSP Lectures, Alan Fekete, University of Sydney (2004)

## Exercise

- For each solution, what is the greatest number of philosophers who can be eating at one time?

FSP Lectures, Alan Fekete, University of Sydney (2004)