# Table of Contents

- **Chapter 2** ............................................................................................................. 3
  - Slide 5 .................................................................................................................. 3
  - Slide 7 .................................................................................................................. 3
  - Slide 9 .................................................................................................................. 3
  - Slide 11 ............................................................................................................... 4
  - Slide 12 ............................................................................................................... 4
  - Slide 13 ............................................................................................................... 4
  - Slide 14 ............................................................................................................... 4
  - Slide 15 ............................................................................................................... 4
  - Slide 28 ............................................................................................................... 4
  - Slide 30-31 ........................................................................................................ 4

- **Chapter 3** ............................................................................................................. 5
  - Slide 4 .................................................................................................................. 5
  - Slide 6 .................................................................................................................. 5
  - Slide 8 .................................................................................................................. 6
  - Slide 10-11 .......................................................................................................... 6
  - Slide 12-14 .......................................................................................................... 6
  - Slide 21 ............................................................................................................... 6
  - Slide 25 ............................................................................................................... 7
  - Slide 29 ............................................................................................................... 7
  - **Summary** ......................................................................................................... 7

- **Chapter 4** ............................................................................................................. 9
  - Slide 11 .............................................................................................................. 9

- **Chapter 5** .......................................................................................................... 11
  - Slide 6 ................................................................................................................ 11
  - Slide 10 .............................................................................................................. 11
  - Slide 13 .............................................................................................................. 11
  - Slide 17 .............................................................................................................. 12
  - Slide 26-36 ........................................................................................................ 12
  - Slide 29 .............................................................................................................. 13
  - Slide 37 .............................................................................................................. 13

- **Chapter 6** .......................................................................................................... 14
  - Slide 2 ................................................................................................................. 14
  - Slide 7 ................................................................................................................. 14
  - Slides 8-18 ........................................................................................................ 14
  - Slide 14 .............................................................................................................. 15
  - Slide 16 .............................................................................................................. 15
  - Slide 18 .............................................................................................................. 15
  - Slide 19 .............................................................................................................. 15

- **Chapter 7** .......................................................................................................... 16
  - Slide 4 ................................................................................................................. 16
  - Slide 6 ................................................................................................................. 16
  - Slide 7 ................................................................................................................. 16
  - Slide 11 .............................................................................................................. 16
Chapter 2

Slide 5
Note how a labelled transition system with only two states can produce an infinite execution trace!

Slide 7
Which one of the three specifications of SWITCH is the best? The first one is the best one, because it’s the most modular. Consider someone who decides to write a Java program and ends up with a huge main() method that does all the work without ever calling another method. This is like the third version of SWITCH. Breaking its specification into multiple parts allows us to better understand what the process is supposed to be doing.

The LTS for SWITCH has two states. This is evident in the first version, since we know that SWITCH = OFF and that there’s another state called ON. But when the LTSA tool is given the third version of SWITCH, how does it know that there should be two different states?
   a) It knows that the initial state (0) will represent the process SWITCH (that is always the case).
   b) It knows that at that initial state, SWITCH can perform action on and move to some unspecified state, from where it can perform action off.
That’s why it creates a second state – because the on action does not move into a previously defined state. That’s also why it doesn’t create a third state – because the off action specifically says that it will take the system to the state SWITCH, which we know already.

Slide 9
Why do we need two states for orange?
Because the action orange is supposed to be followed by two different actions in each case (red & green), so we need to distinguish between these two cases by using separate states.
This would have been easier to see if we had used explicit state names (i.e., subprocesses) instead of relying on implicit states:
---
TRAFFICLIGHT = PREPARE_TO_STOP,
   PREPARE_TO_STOP = (red --> MUST_STOP),
   MUST_STOP = (orange --> PREPARE_TO_GO),
   PREPARE_TO_GO = (green --> MUST_GO),
   MUST_GO = (orange --> PREPARE_TO_STOP).
---

Slide 11
- What’s the probability of observing coffee instead of tea?
We have no idea. Probabilities are not defined on the choices a process can make. A choice can therefore represent all different types of systems – those where an action x is more probable than another action y and those where it’s the opposite.
- What are the possible execution traces of the DRINKS machine?
There are infinite number of traces and each one of them is infinite in length!!!
Slide 12
In the previous slide, actions red & blue determine the choice of the process’s future behaviour. But in the COIN process, toss doesn’t determine the choice of the process’s future behaviour (HEADS or TAILS). So we call this a non-deterministic choice.

Slide 13
What’s the probability of failing to transmit a message according to the CHAN process?

Slide 14
When you write:
BUFF = (in[i:0..3] => out[i] => BUFF).
Then, after the in[i:0..3] action you should realise that there are 4 implicit states, not just 1!!!
This is because the above specification is simply a way to describe a choice for the different possible values of the index i, without writing too much. As the slide says, it’s equivalent to:
BUFF = (in[0] => out[0] => BUFF

Slide 15
Why’s there only one transition from state 0 to state 2 and it has two labels – in.1.0 & in.0.1?
Think: After action in[1][0] process SUM is supposed to behave like which sub-process? What about after action in[0][1]?

Slide 28
Unlike the LAMP process that we’ve implemented in tutorial 1 as a simple Java class with on/off methods, the COUNTDOWN process here is an active one, that’s supposed to be performing actions on its own, instead of simply waiting to respond to calls from others. So we need to implement it as a thread!

Slide 30-31
Why are actions begin & end been implemented as separate, public methods, while tick & beep are private methods?
Think: active vs passive behaviour – who’s supposed to perform each of these?

In the Java implementation of action end, do you understand how private attribute counter is used to terminate the thread?
Chapter 3

Slide 4
- Why do we get an arbitrary relative order of actions from different processes? Because as a previous bullet item says, we assume that the processes are executing with an arbitrary speed! Arbitrary means that it may very well change during the process’s execution, so you may observe lots of actions from one process, then lots of actions from another process, then again lots of actions from the first process, and so on.

This asynchronous model of execution is essentially the opposite of the synchronous model of execution.
In the synchronous model of execution, all processes synchronise and execute with the same speed. There is effectively a global clock that is telling every process when they’re supposed to perform their next action.
Examples of synchronous systems:
- People rowing on a boat – everybody pulls at the same time.
- People passing buckets in a line to put out a fire – everyone is supposed to pass their bucket down/up the line at the same time.
- Soldiers marching.
- Hardware, especially processors – the internal clock tells all the circuits when to read their inputs and produce their outputs. That’s why we’re interested in the Hz of processors (i.e., number of actions per second).

An example of an asynchronous system is a company – workers of that company work at their own speed – there’s no global clock to give them instruction to perform their next action. Another example is soldiers crossing a bridge – they stop marching, so as not to cause the bridge to synchronise and collapse(!) and instead walk each on their own rhythm, so that one’s steps are cancelled out by those of the others.
What is an orchestra – a synchronous or an asynchronous type of a system?

Slide 6
Drawing the LTS of a parallel composition – the parallel composition’s state is essentially a combination (i.e., Cartesian product) of the states of the processes that are being composed in parallel. So the initial state of the system is the state (0,0) when we’re composing two processes. If the first process moves first, then we’ll move to (1,0). If the second process moves first, then we’ll move to (0,1). By keeping track of which process has moved in this way, we can draw the complete LTS of their parallel composition, as is done on the slide.

For two processes, one with 3 states and the other with 4 states, we get a matrix (Cartesian product) like this. Always draw LTSs in this way – impossible to get right otherwise!!!

```
<table>
<thead>
<tr>
<th>0 1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4</td>
</tr>
</tbody>
</table>
```

“x” here corresponds to the (0,0) state, i.e., the one where both processes are at their first state.
**Slide 8**
Since sharing the same action means that two (or more!) processes must synchronise and perform that action together, the only way we’ve got to control process interaction is by changing action names! Sometimes we want to change the names of some actions so that two (or more) processes don’t interact, in other cases, we want to change the names of some actions so that processes do interact.

Try to use the **nuclear bomb** *(aka “alphabet extension”)* at your disposal, to see what happens:

- **MAKER** = (make -> ready -> MAKER).
- **USER** = (ready -> use -> USER) + { make }. // the bomb!
- **MAKER_USER** = (MAKER || USER).

What’s the LTS of **MAKER_USER**? Why is that the LTS?

**IF YOU DON’T UNDERSTAND THIS, POST ON MOODLE!!!**

**Slide 10-11**
Why is the model on slide 11 better than one on slide 10?
Again, because it is more **modular** than the one on slide 10. It breaks down the system into sub-systems (**MAKERS & ASSEMBLE**), instead of presenting everything at the same level.

**Slide 12-14**
These slides present two renaming mechanisms that apply themselves to all the actions of a process.

The single colon (:) is used when we want to get **different, independent** instances of some process:

- **TWO_SWITCHES** = ({a,b}:SWITCH).
This creates two copies of the SWITCH process – the first one has all its actions prefixed with “a.”, while the second one has all its actions prefixed with “b.”. That way, when we perform action a. on process b:SWITCH is not forced to perform action b. on – the two switches are independent of each other.

  So, a **single colon** -> **one** label with **one** prefix, **one process per label**!

The double colon (::) is used when we want a single copy of some process but we’d like that process to be able to interact with many different processes at the same time. It’s mostly used when we want to represent some shared resource, that is going to be used by different processes, as is done on slide 13.

  So, a **double colon** -> **multiple** labels with **one** prefix, **a single process always**!

Another way of specifying **RESOURCE_SHARE** is:

- **RESOURCE_SHARE** = ( {a,b}:USER || {a,b}::RESOURCE ).
The difference is only a single colon but the consequences are completely different – check the LTS’s on slide 14.

**Slide 21**
We didn’t focus much on structure diagrams but what’s the answer here?
Well, the diagram shows two copies of the BUFF process, prefixed with a & b. So:

- **TWOBUFF** =

  ( {a,b}:BUFF ).
The diagram also shows that the out action of a:BUFF is linked to the in action of the b:BUFF and that link is labelled a.out. That means that there we have an action renaming of a.out & b.in into a.out - we only need to rename b.in (since a.out is already called a.out!):

||TWOBUFF = ( {a,b}:BUFF )
  /
  { a.out / b.in }.

The diagram also shows a link between a.in and some external action, where the link label is in, and a similar link for b.out, which is labelled out:

||TWOBUFF = ( {a,b}:BUFF )
  /
  { a.out / b.in,
    in / a.in, out / b.out }.

Since the only externally visible actions are in & out (these are the only circles in the outer box representing TWOBUFF), we need to hide everything else and expose in,out only:

||TWOBUFF = ( {a,b}:BUFF )
  /
  { a.out / b.in, in / a.in, out / b.out }
  @ { in, out }.

Slide 25
Do you understand how the a.stop and b.stop have been renamed into a single stop action?
Do you understand how this sharing of the two stop actions has been implemented in the corresponding Java program? (look up slide 29...)

Slide 29
How’s the pause actions implemented? Check the source code of this applet... (it’s in book_applets/applets/concurrency/display/ThreadPanel.java)

Summary
- In order to compose processes in parallel we use the parallel composition ("||") operator: P || Q || W
- Processes running in parallel *interleave* their actions
- When two or more processes have an action with the same name then *all* of them must perform that action together (they *synchronise* on that action). These are called *shared* actions.
- If even one process is unable to perform a shared action, then no process can perform it.
- In order to control which actions are shared (and thus interaction) we can use a number of operators:
  - Colon: {a, b, c}:P -> creates three instances of P, each prefixed by a different prefix from the set {a, b, c}.
  - Double Colon: {a, b, c}::Q -> creates a *single* instance of Q, where each of its actions x has been replaced by as many actions as
prefixes in the prefix set, each of the actions prefixed with one of the elements of the prefix set (so here: \( a.x, b.x, c.x \)).

- The double colon operator is used to introduce a process that models a *shared resource* (which interacts with many different process instances).

- Action renaming: \( P / \{ \text{new\_name} / \text{old\_name} \} \rightarrow \) replaces all instances of \( \text{old\_name} \) inside \( P \) with \( \text{new\_name} \).

- Action hiding: \( P \setminus \{ \text{internal1, internal2} \} \rightarrow \) hides \( \text{internal1} \) and \( \text{internal2} \) from other processes, so they cannot synchronise with these actions.

- Action exposure (the dual of hiding): \( P \oplus \{ \text{external1, external2} \} \rightarrow \) exposes just \( \text{external1} \) and \( \text{external2} \) from \( P \), hiding all its other actions, so that other processes can only synchronise with these – *can think of these as \( P \)'s API*.

- Alphabet extension (from chapter 2): \( P + \{ \text{action1, action2} \} \rightarrow \) adds \( \text{action1} \) and \( \text{action2} \) to the alphabet of \( P \), even if \( P \) cannot perform them. So, if these are shared actions, we are *effectively forbidding all other processes from ever performing* action1 and action2 (*we're mean...*).
Chapter 4

Slide 11

Slide 11 in chapter 4 is the *most* important slide in the whole book. If you don’t understand it 100% then you’ll struggle with the rest of the material.

The difficult points in it are:

1) How does the TURNSTILE process manage to read the value that is held inside the VAR process?

TURNSTILE's INCREMENT sub-process does a read[x:T] at some point (ignore the value. prefix for the time being).

Given the definition of indexed actions, this means that it has a choice at that point - it can do read[0], read[1], ..., read[N].

But, *surprise, surprise*, these are actions that are *shared* with process VAR. And process VAR *cannot* do them at all states. In fact, at each of the states of VAR, it can do only a single read action - the one indexed by the value it currently holds:

\[ \text{VAR[u:T]} = \text{read[u]} \rightarrow \text{VAR[u]} \]

Therefore, TURNSTILE does not really have a choice – it can only do the read action that VAR can do at this point, so it ends up reading the current value u of VAR!

*(you may want to set N to 1, have the tool draw all the LTSs and simulate to see what's going on - with LTSA there’s an option that allows you to draw multiple LTS)*

**RE-READ THIS EXPLANATION UNTIL YOU'VE UNDERSTOOD IT – POST QUESTIONS ON MOODLE IF YOU DON'T!!!**

2) Why do we use the prefixing we use in the GARDEN composite process?

Well, we need two independent instances of TURNSTILE, so we prefix one with "west:" and the other with "east:". Now their actions are not shared, so they can perform them independently.

They both do actions value.read[x:T] and value.write[x+1] (before the prefixing with east/west). So, the VAR instance needs to be prefixed with "value:".

Since there are two turnstile instances, the VAR instance needs to be prefixed with both east and west (but using the :: operator so as not to get two different copies of VAR, just a single copy).

Does it make sense? No? Set N to 1, and draw the LTSs. Observe the names of their transitions. Experiment with different cases:
3) Why in Heaven’s name are we using alphabet extension in TURNSTILE?!?!

<evil-grin> Heh, heh, heh... I love this part! </evil-grin>

Alphabet extension is used to forbid another process from performing an action.

Still unsure? POST ON MOODLE!!!
Chapter 5

Slide 6
THESE ARE THE RULES TO TURN ANY MODEL INTO A CORRESPONDING JAVA PROGRAM!
- “Active” processes are turned into threads.
- “Passive” processes are turned into monitors.

Note: Sometimes, one cannot say that a process is purely active or purely passive – consider the COUNTDOWN process of Chapter 2, slides 28-31. There we transformed it into a combination of a monitor and a thread. But again, its passive parts (begin, end) were transformed into monitor methods, while its active parts (beep, tick) were transformed into internal actions to be called by the COUNTDOWN’s internal thread.

Slide 10
Here, action arrive has been implemented as a synchronized Java method, which decreases the number of free spaces in the car park. The synchronized part is because car part is a passive process that waits to react to arrive/depart events, so it needs to be implemented as a monitor.
But in the model of the car park says that the arrive action is only possible when the condition (i>0) is true, i.e., when the number of free spaces is greater than 0. So the Java implementation we’ve got on this slide is missing something – it shouldn’t decrease the number of free spaces if there aren’t any!
What we’re missing here is the implementation of the model’s guarded condition “when (i>0)”. This is to be achieved through the wait() method of Java (see next slide).

Slide 13
Just like slide 6, THIS IS THE RULE TO TRANSFORM ALL GUARDED ACTIONS IN A MODEL INTO JAVA CODE!!!

See Table 1 on page 11.
Table 1 FSP guarded actions in Java

```
FSP:
when condition action -> NEW_STATE
Java:
public synchronized void action() throws InterruptedException {
    while (! condition) wait();
    // do whatever action is supposed to do
    notifyAll();
}
```

Why?
FSP’s action is supposed to be something atomic, i.e., it either happens entirely or it doesn’t happen at all. It is supposed to occur only when the guard condition is true.
Therefore, the code:
1) Checks inside a while loop (why a while and not an if?) that condition holds. While it doesn’t it waits.

2) If the wait() is interrupted, then the method throws this interruption exception to whomever called it, without performing any of the instructions for action.

3) When the wait returns and condition holds, the instructions of action can be performed – no exceptions are allowed to be thrown here, as they would break the atomicity of action! All exceptions must be caught and either the instructions performed so far should be undone before leaving method action, or the processing of action should be completed (works like a transaction).

4) At the end we notify all other threads that something has changed in the object and they may want to check if their conditions are now satisfied.

Slide 17
What does “Max” represent?
It represents the maximum number of shared resources, e.g., printers, cinema seats, etc., that can be used concurrently. In a cinema with 100 seats, we’d only allow 100 (sit) down actions...

Why do we initialise the semaphore to 0? Why not initialise it to Max (or some number in between)?
It depends on how many resources are ready to be used initially.
For a cinema, you may want to initialise it to 0 initially, so as to get some time to clean up before you open the doors and do Max up actions to allow the people to enter.
For another system where there are already x (0 < x <= Max) resources that are ready to be used immediately (e.g., x printers), we’d initialise our semaphore to x: SYSTEM = (SEMAPHORE(x) || …).

- Which one is equivalent to a lock, SEMAPHORE(0) or SEMAPHORE(1)?

Slide 26-36
Buffer – an example of which is the car park in the first part of this chapter – is an extremely important notion in asynchronous systems.
In synchronous systems, the data that are produced by one process are immediately consumed by another process, since all processes operate at the same speed.
But in asynchronous systems, the producer & consumer processes operate at different speeds. So we need some mechanism to make sure that the faster process doesn’t need to wait for the slower process all the time. That mechanism is the buffer – the longer it is, the longer it will take the faster process to have to delay so that the slower process can catch up. For differences in speed that are transient, i.e., processes delay a bit only occasionally, a buffer can ensure that the two processes never have to wait (well, at least minimise the probability that they’ll have to wait).
If the producer is much faster than the consumer, then the only way to ensure that it’ll not have to wait is to have an infinite buffer! Identifying the correct length of a buffer is an extremely difficult problem (well, it’s impossible to solve really...).
Slide 29
- *Can you spot the bug? :-)*

Consider the scenario where there are two producers, two consumers, and a buffer with one slot.
Here's a possible execution trace:

Consumer 1 blocks
Consumer 2 blocks
Producer 1 inserts a letter and does a notify
Consumer 1 is notified but fails to get the lock
Producer 1 blocks
Producer 2 blocks
Consumer 1 gets the lock
Consumer 1 consumes the data
Consumer 1 notifies
Consumer 2 wakes up
Consumer 2 blocks
Consumer 1 blocks

**Moral of the story** – use `notifyAll()` instead of `notify()`!!!

Slide 37

*If you don’t know the invariants of your class, it’s most probably wrong.*

- What’s the class constructor(s) supposed to do?
  - *ESTABLISH THE INVARIANT!!!* (i.e., *make it true*)
- What about the class *public* methods?
  - They *can assume* that the invariant is true *when they start.*
  - They *must guarantee* that it is true *when they finish.*
- What about *private* methods?
  - These can break the invariant as they’re meant to be used by public methods as intermediate steps.
Chapter 6

Slide 2
With bad things there are two approaches:

1) Try to identify when they happen and respond to them (e.g., fire alarm).
2) Try to make them impossible/improbable (e.g., use fire-resistant material).

Here we’re trying to design our systems in such a way that deadlocks are \textit{impossible}. Other approaches attempt to identify deadlocks at run-time and repair the system by terminating one of the processes involved in the deadlock.

Slide 7
When checked, the system has no deadlocks – HURRAY!!!
But it has a problem with progress – it fails this test. Yet, we do not bother changing it so as to resolve the progress violation – why?

Because knowing there’s a problem doesn’t mean that we should solve it necessarily. Sometimes the cost of introducing a solution is much higher than the actual cost of the problem itself. Consider for example credit card issuing companies – they know that there are fraudulent transactions and they know that if they improved the technology of the card readers, etc. then the percentage of fraud would drop. But they don’t do it because the cost of replacing all card readers is far higher than the cost of the fraudulent transactions themselves!

Same here – it’s much simpler to introduce some timeout that’s long enough (and random enough) that the progress violation happens rarely.

Slides 8-18
The dining philosophers problem is one of the most classic and well-known problems in operating systems and concurrency.

Note: The modulo (remainder) operator helps constrain a value within a range:

\[ 0 \leq (i \mod N) \leq N-1 \Rightarrow N \leq (i \mod N) + N \leq 2N-1 \quad \text{when both } i \text{ and } N \text{ are positive} \]

\[ 0 \leq (i \mod N) \leq N-1 \Rightarrow N \leq (i \mod N) + N \leq 2N-1 \quad \text{when } i \text{ is positive and } N \text{ is negative} \]

\[-N+1 \leq (i \mod N) \leq 0 \Rightarrow 1 \leq (i \mod N) + N \leq N \quad \text{when both } i \text{ and } N \text{ are negative} \]

\[-N+1 \leq (i \mod N) \leq 0 \Rightarrow 1 \leq (i \mod N) + N \leq N \quad \text{when } i \text{ is negative and } N \text{ is positive} \]

At least that’s what I got with the C++ & Java code in Table 2 (\textit{which might be system-dependent behaviour for all I know}):
Table 2 Testing modulo sign in C++ & Java

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << 1 % 3 << endl
    << 1 % -3 << endl
    << -1 % -3 << endl
    << -1 % 3 << endl;
    return 0;
}
```

```java
class Ideone {
    public static void main (String[] args) {
        System.out.println(1 % 3);
        System.out.println(1 % -3);
        System.out.println(-1% -3);
        System.out.println(-1%3);
    }
}
```

// https://ideone.com/mkXOAg

// https://ideone.com/B7KzEQ

**Slide 14**
Note how field "taken" encodes the state of the fork – initially it's `false`, as the fork is not taken.
Method `put` has not wait guard because it assumes that its callers are well behaved – *an unsafe assumption to make in general!*

**Slide 16**
Compare the initialisation of the philosopher threads with the fork objects against the model itself on slide 10.

**Slide 18**
What other solutions are there?
- Introduce a timeout.
- Introduce a butler process that doesn’t allow all N philosophers to sit at the table – only allows up to N-1 of them to sit at any time.

**Slide 19**
We can actually use the LTSA tool to find a solution to a problem, by representing the solution as a deadlock state.
Chapter 7

Slide 4
An FSP property process encodes the acceptable behaviour of a system. If we have not specified that an action can be performed at some state, then that action becomes an error transition at that state automatically.

Slide 6
Very important to note that property processes *MUST* be deterministic! Otherwise we cannot compose them “transparently” – a non-deterministic property process might accept some trace in one case and reject it in another.

Slide 7
“A safety property must be specified so as to include all the acceptable, valid behaviour in its alphabet.”
If you're interested in specifying a safety property involving actions a1, ..., aN, then a good approach is to start like this:

property MyProp = P
  , P = ? // Fill in this part
  + {a1, ..., aN}.

In this manner, you declare from the start which are the actions that the property cares about, and decrease the chances of making a mistake by forgetting to state when some of these actions are acceptable (the tool’s reported violations will remind you that you forgot to specify these acceptable behaviours).

Slide 11
Note that T starts from 0, while ID starts from 1! So, it’s not the same range.

Slide 14
- What’s RED[0] & BLUE[0])?
As the range of i is ID, and the guard on action exit that goes to RED[i-1] ensures that i is not 1, there is no such state as RED[0]. The model has been specified in such a way that both these two states correspond to ONEWAY actually.

Slide 23
“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.” *after some point* (i.e., the action can actually get executed a few million times – even more – but after some point it never gets executed -> that's starvation).

Slide 24
Note that the notion of “fair” & “infinitely often” here is a bit tricky. It doesn't mean “as many times as” – for example, a coin that when tossed lands heads once every 100 times, is still a “fair” coin as far as we are concerned, since heads will come up an infinite number of times with infinite tosses.

Slide 41
“This is a direct translation from the model.” <- Actually, it's not... And that's why we have a bug... :-(

We have failed to follow the rule for translating guarded actions in models into Java code. We have added some code to be executed *right* before the wait. And that's buggy.

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

- What’s the problem? 
The problem is that if we get interrupted while we’re waiting, we’ll exit method redEnter, but we'll not have undone the increment of waitred.

- How should we have written this piece of code?

```java
synchronized void redEnter() 
    throws InterruptedException { 
        while (nblue>0 || (waitblue>0 && blueturn)) { 
            ++waitred; // attempt re-entry 
            try { wait(); } 
            catch (InterruptedException ie) { throw(ie); } // don’t ignore it! 
            finally { --waitred; } // success or exception – decrement waitred 
        } 
        ++nred; 
} 
```

*Moral of the story:*

*When you divert from the patterns, you’re in *very* dangerous territory!!!*

**Slide 57**
Personally, I think that there’s a bug (potential deadlock) here:

```java
public synchronized void releaseRead() { 
    --readers; 
    if (readers==0) notify(); // BUG? DON’T OPTIMIZE!!!! 
} 
```

We might wake up a reader who's waiting because writers are waiting... :-{
Chapter 8
Chapter 8 discusses in more detail the general approach that we’ve been following so far in the book:

1) do:
   a. **Model** the system in FSP
   b. **Simulate & verify** to make sure the **model** & your **properties**:
      i. Do what you **intended**; and
      ii. Have **no errors**,
2) while ! satisfied (*rinse & repeat*)
3) **Translate** the model into **Java code**, **BEING VERY CAREFUL to follow the patterns!**
Chapter 9
(no commentary yet)