IN2014 Parallel & Concurrent Programming – Study Notes
Getting the Basics Right
Christos Kloukinas © 2010 – 2024

Table of Contents

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., sub-processes) 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 an ***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

DANGER!!!

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
       |in[1] -> out[1] -> BUFF
       |in[2] -> out[2] -> BUFF
       |in[3] -> out[3] -> 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 33 "CountDown timer example"

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 35-36 "CountDown class…"

Why are actions `start` & `stop` 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 `stop`, 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!!!

```
3
2
1
0   x
    0   1   2   3   4
```

"**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.

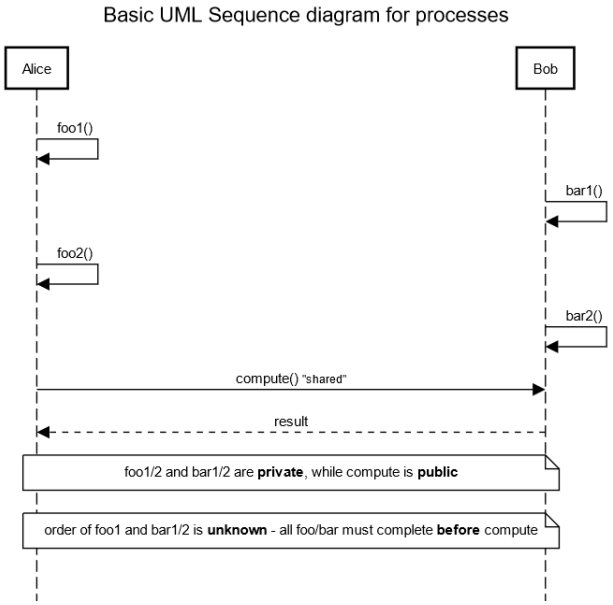
Try to use the ***nuclear bomb*** ™(aka "***alphabet extension***") at your disposal, to see what happens:

```
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!!!

What does a UML sequence diagram look like? Like processes, with local & shared actions!



Slide 10-11

Why is the model on slide 11 better than one on slide 10?

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. For small models this may be overkill sometimes – a bit of judgement is needed.

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 separate** process **per** prefix, all its actions prefixed!

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 -> **only *a single process always,*** its actions prefixed with **all** prefixes!

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 26

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 30… the whole program's `stop` method calls the `stop` method on each object.)

Slide 30

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 independent 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 these new 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 / { new_name / old_name }` -> replaces all instances of `old_name` inside `P` with `new_name`.

    - Action hiding: `P \ {internal1, internal2}` -> hides `internal1` and `internal2` from other processes, so they cannot synchronise with these actions.

    - Action exposure (the dual of hiding): `P @ {external1, external2}` -> exposes just `external1` and `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 + {action1, action2}` -> adds `action1` and `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 12

Slide 12 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 `u`, the value currently held:

```
VAR[u:T] = (read[u] -> 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 `VAR`'s current value `u`!
(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/ASK IN CLASS 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 `value:VAR` instance needs to be prefixed with both `east` and `west` (this time using the `::` operator, so as not to get two different copies of `value: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:

`{east,west}::VAR` (no `value:`)

`value:VAR` (no `{east,west}::`)

`{east,west}:VAR` (no `value:` and `:` instead of `::`)

Still unsure? ASK/POST ON MOODLE!!!

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: used to forbid another process from doing an action."

Set N to 1 – what's the alphabet of `VAR`?

```
{read[0], read[1], write[0], write[1]}
```

What's the alphabet of `TURNSTILE`? (ignoring "`value`".)

```
{go, arrive, end,
 read[0], read[1], _________ write[1], write[2]}
```

MIND THE GAP!!! *Actions `read[0..1]`, and `write[1]` are* ***shared*** *so both processes (`TURNSTILE` & `VAR`) do them together.*

But what about action `write[0]`? It's not shared!!! So process VAR can do it whenever it wants, resetting itself!!!

In other words, in our model `VAR` can **magically** change its value back to zero whenever it feels like it, ignoring the updates that `TURNSTILE` had done. ***<- NOT GOOD!***

So we need to use alphabet extension to add action `write[0]` to the alphabet of `TURNSTILE`.

Now that it has this action in its alphabet it's also shared, so `VAR` can ****never**** do it (because `TURNSTILE` can never do it).

"Alphabet extension: used to forbid another process from doing an action."

Sense much makes? Post on Moodle if not. (I *know* it doesn't...)

If you wanted to model `GARDEN` from scratch, would you have tried to create this LTS in one go?

Check the model and see how the different parts of the system are modelled little by little. First we've got a model of the variable that holds the number of people inside the garden, then a model of a turnstile, both described using a number of sub-processes to make them easier to specify.

NEVER try to specify something as one big blob – it's bound to fail. Try instead to break your system into smaller parts (`VAR`, `TURNSTILE`) and then specify each of them little by little, using sub-processes.

Chapter 5

Slide 3 "5.1 Condition synchronization"

1) "does not permit cars to depart when there are no cars in the carpark" – to model reality, not *really* needed in a ***real*** system (still needed in our Java ***simulation*** of the real system). Even so, one may choose to implement it in a ***real*** system to ensure that the program's view of the world and the world are in sync (Why did you detect a car leaving if your internal counter says there was no car? What's going on – should a human go inspect?).

2) "A controller is required" – a ***very*** ***sad*** fact, since controllers impose centralisation, which ***reduces*** concurrency.
Centralised designs usually lead to *less throughput* and are *less robust* than decentralised ones – sometimes that's the best we can do. Sometimes we choose them to increase other properties, e.g., max speed of processing one item (we accept to waste resources just so we can increase individual response time).
For example, compare the centralised structure of military forces vs the usually far more decentralised structure of most civil societies.

Slide 7 "carpark program"

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 31-34. 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 11 "Carpark program - CarParkControl monitor"

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 slide 13).

Slide 15 "condition synchronization in Java"

Just like slide 7, THIS IS THE RULE TO TRANSFORM ALL GUARDED ACTIONS IN A MODEL INTO JAVA CODE!!!

See Table 1 that follows (on page 12).

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 the ***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.

BEWARE!

This pattern makes a number of assumptions with regards to exceptions!

1) The action implementation doesn't throw any exceptions. The only exception that is potentially thrown is an InterruptedException inside wait(), which we forward.
2) Actions form a kind of self-loop, in the sense that if a subsequent action is interrupted, then the system is fine. **THIS IS A VERY QUESTIONNABLE ASSUMPTION!!!**
   You need to think in terms of **TRANSACTIONS** – what will happen if an action fails?
   a) Is there something that I need to undo? (abandon the transaction)
   b) Should I persist, to ensure that the transaction completes? (force the transaction)

Slide 19 "5.2 Semaphores"

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 28 "SEMADEMO program – MutexLoop"

What happens when the critical section "**while(ThreadPanel.rotate());**" throws an exception?
The mutex.up() statement will not be executed. One resource (the ***only*** resource) has been removed from the system permanently.
For this reason, Java has introduced try-with-resources - https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html
Each resource defines a method close() in its interface, which is called whether the try block terminates normally or through an exception (so it's placed inside a ***finally*** clause).

- Can you re-write this code so that instead of the plain mutex semaphore, we now have a resource that always calls the mutex.up() after the critical section?

Would this code be good enough for all cases? In most cases, the program logic of a critical section goes like this: (a) read things, (b) calculate, and (c) update. If an exception occurs during reading/calculating, then there's no issue – we can simply close() our resource and allow the exception to propagate.
But what happens when the exception happens while we're doing the updating?
TRANSACTIONS:
You must be able to either:

- **Bail Out/Rollback:** Undo (rollback) the effects of any updates so far, to leave things consistent (as they were before you attempted the transaction). You need to figure out *(i)* what you've changed so far, and *(ii)* how you can undo that change (undo handlers).
- **Force Through/Commit:** If the effects of what you've changed so far cannot be undone, then you need to force through the remaining updates and commit.

For this reason, the updates are sometimes done in a local copy and then the actual "commit" of the transaction copies over these updates to the global, live copy using code that CANNOT throw an exception. This uses the Bail Out strategy – actual updates are abandoned (the local copy is discarded).
Check out the **two-phase commit protocol (2PC)** as well for a better appreciation of the complexity of transactions in a distributed setting (2PC is one of the simplest protocols):
https://en.wikipedia.org/wiki/Two-phase_commit_protocol

For more information on how to group actions together into different sections (and potentially pass them as arguments somewhere…), you may want to have a look at Lambda functions:
https://blogs.oracle.com/java/post/java-se-8-lambda-quick-start
https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
https://docs.oracle.com/javase/tutorial/java/javaOO/whentouse.html

Slide 30-35 "5.3 Bounded Buffer"-"bounded buffer prog - producer process"

Buffer – an example of which is the car park in the first part of this chapter – is an extremely important notion in asynchronous systems. It's a SYSTEM DESIGN PATTERN (some others: caching, pre-fetching, pipelining)
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 32 "bounded buffer - a data-independent model"

Being able to model something in a data-independent model is amazing – it allows us to automatically verify very complex systems!
But sometimes the property we care about needs the values, for example FIFO – how can we prove the element/data received at time t was processed/delivered at time after all previous elements/data, if we cannot even compare these elements (because we've abstracted their values away)?

Here come some clever data abstractions to our rescue. Instead of simply abstracting entirely the value of the data away to the point that they're indistinguishable from each other, we can use three values (blue, red, and green), to encode the property we want, along with carefully crafted senders/recipients.

For FIFO, we can have the sender send messages in the order: *blue* red blue* green blue** . Then at the recipient we only need to check that the *red* message gets received before the *green* one.
If that's the case, then we have proved that a message sent before another will always be received first.
Why? That's where the first two *blue** parts come into play… They ensure that we're not just proving FIFO for just a pair of messages but considering all possible pairs!
SENDER = Prefix,
Prefix = (blue -> Prefix | red -> Middle),
Middle = (blue -> Middle | green -> Suffix),

Suffix = (blue -> Suffix).
RECIPIENT = NotSeenRed,
NotSeenRed = (blue -> NotSeenRed |red-> SeenRed |green-> **ERROR**),
SeenRed = ({blue,green} -> SeenRed | red -> **ERROR**).
||SYS = (a:SENDER || {a,some}::IMPLEMENTATION || some:RECIPIENT).
Where the IMPLEMENTATION takes a:X and produces some:X after some point.

Slide 34 "bounded buffer program - buffer monitor"

- *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 46-47 "5.5 Monitor invariants"-"Class Invariant Properties"

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).
3) Buy insurance in case they happen… (usually combined with 1 or 2)

Here we're trying to design our systems in such a way that deadlocks are ***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 8-19

The dining philosophers problem is one of the most famous classic problems in operating,distributed systems and concurrency.

The dining philosophers problem is one of the most famous classic problems in operating/distributed systems and concurrency.

The dining philosophers problem is one of the most famous eclassic problems in operating/distributed systems and concurrency.

Why? Because it's about managing resources, in a way that avoids deadlocks.

Study it well!

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

$0 \leqq (i \,\%\, N) \leqq N\text{-}1 \longrightarrow N \leqq (i \,\%\, N) + N \leqq 2N\text{-}1$	when $0 < i \wedge 0 < N$
	Or when $N < 0 < i$
$-N+1 \leqq (i \,\%\, N) \leqq 0 \longrightarrow 1 \leqq (i \,\%\, N) + N \leqq N$	when $i < 0 \wedge N < 0$
	Or when $i < 0 < N$

The C++ & Java code in Table 2 demonstrates that:

Table 2 Testing modulo sign in C++ & Java

```
#include <iostream>
using namespace std;
int main() {
   cout
   <<  1 %  3 << endl
   <<  1 % -3 << endl
   << -1 % -3 << endl
   << -1 %  3 << endl;
   return 0;
}
//https://ideone.com/mkXOAg
```

```
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/B7KzEQ
```

Slide 14

Note how field "`taken`" encodes the state of the fork – initially it's `false`, as the fork is not taken.

Note that guarded actions need not appear as such in the model – here they're hidden behind the sequence imposed! (see slide 15)

Method `put` has no `wait` guard because it assumes that its callers are well behaved – *an unsafe assumption to make in general!*

Slide 17

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

Slide 19

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 20

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 by definition.

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).

*Designing safety properties – focus on **system** behaviour, not on that of the mechanisms that you use to establish the properties (locks, semaphores, etc.). See how slide 8 does it.*
A possible test – try removing/commenting out the actions of the mechanisms (locks, semaphores, etc.) that you use to make the property true. Do your properties hold? If the mechanisms are indeed needed (always a possibility that they're superfluous), then the property shouldn't hold.

Slide 13

Note that `T` starts from `0`, while `ID` starts from `1`! So, it's not the same range.

Slide 16

- 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 26

"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 27

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 45

"This is a direct translation from the model." <- *But we have a bug…* :-(

It's not the merging of the request & enter actions into one method that's the problem – we'd have a problem even if we hadn't merged them ***(and it'd be worse, as then it'd be harder to control the transaction – one may have introduced extra calls between the request and the enter that we can't!)***.

The pattern to turn models into code doesn't consider exceptions fully – here "request" is a pre-requisite for "enter" and these two atomic actions must happen as a **transaction**. If something happens in-between them, then we need to *(i)* abort the transaction; or *(ii)* force it through.

- How should we have written this piece of code?

Case 1: Abort Tx, undoing partial actions

```
synchronized void redEnter()  // transaction aborts, undoing partial results
        throws InterruptedException {
  try {
    ++waitred;                                          //<-- Tx action 1

    while (nblue>0||(waitblue>0 && blueturn)) wait(); }//<-- Tx action 2
  // don't catch it!
  finally { --waitred; } // success or exception – decrement waitred
  ++nred;
}
/* Undo handler of Tx action 1 happens to be part of Tx action 2, so we can
do it in the finally clause. But more generally: */
synchronized void redEnter()  // transaction aborts, undoing partial results
        throws InterruptedException {
  try {
    ++waitred;                                          //<-- Tx action 1

    while (nblue>0||(waitblue>0 && blueturn)) wait(); }//<-- Tx action 2
  catch (InterruptedException e){ --waitred; throw e;} //catch, undo, throw!
  --waitred; // this is part of Tx action 2
  ++nred;
}
```

Case 2: Force through Tx

```
synchronized void redEnter()  // transaction forces through
        throws InterruptedException {
  ++waitred;                                          //<-- Tx action 1

  while (nblue>0||(waitblue>0 && blueturn))           //<-- Tx action 2
    try { wait(); }
    catch (InterruptedException e){} // catch & ignore it!
  --waitred; // this is part of Tx action 2
  ++nred;
}
```

Moral of the story:
1) When you divert from the patterns, you're in *very* dangerous territory!!!

2) The patterns are not 100% full-proof – check for transactions!!!

Slide 65

First edition had a simple notify, which is no longer safe here, since we might wake up a reader who's waiting because writers are waiting… :-(

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

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

So, "optimisations" can bite even experts in the field – be extremely careful with them (avoid if possible).

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
        How?
        i. Identify actions
        ii. Identify processes
        iii. Identify safety/progress properties
        iv. Do a structural diagram
        v. Model 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!***
    Even then, watch out for a) "optimisations", and b) transactions!

Chapter 9/10

(no commentary yet)