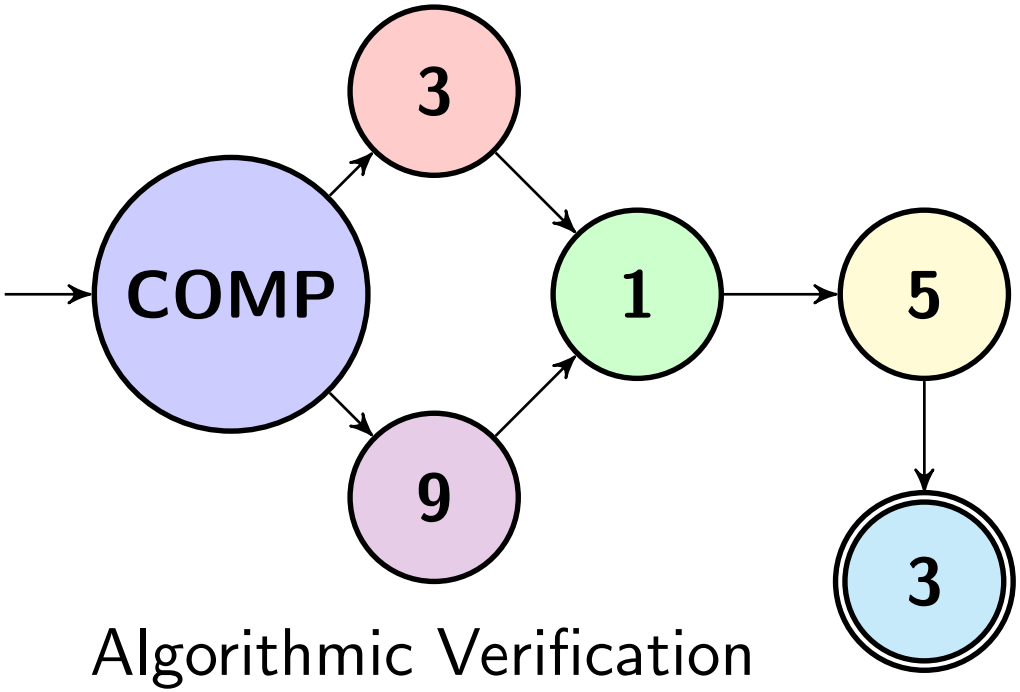


Algorithmic Verification

SPIN and Promela

Dr. Liam O'Connor
CSE, UNSW (for now)
Term 1 2020

SPIN

`http://www.spinroot.com`

Programs are modelled in the Promela language.

Promela in brief

- A kind of weird hybrid of C and Guarded Command Language.
- Models consist of multiple *processes* which may be *non-deterministic*, and may include *guards*.
- Supports structured control using special `if` and `do` blocks, as well as `goto`.
- Variables are either global or process-local. No other scopes exist.
- Variables can be of several types: `bit`, `byte`, `int` and so on, as well as *channels*.
- Enumerations can be approximated with `mtype` keyword.
- Correctness claims can be expressed in many different ways.

Warning

Variables of non-fixed size like `int` are of machine determined size, like C.

Example 1: Hello World

Liam will demonstrate the basics of `proctype` and `run` using some simple examples.

Take-away

You can use SPIN to *randomly simulate* Promela programs as well as model check them.

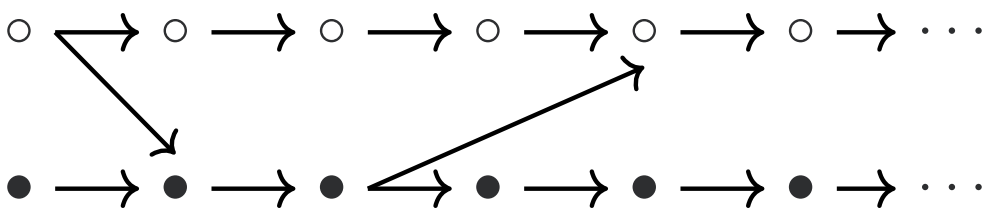
Sequential vs Concurrent

We could consider a *sequential* program as a sequence (or *total order*) of *actions*:

The ordering here is "happens before". For example, processor instructions:

```
LD R0,X ⟶ LDI R1,5 ⟶ ADD R0,R1 ⟶ ST X,R0
```

A concurrent program is not a total order but a partial order.



This means that there are now multiple possible *interleavings* of these actions — our program is non-deterministic where the interleaving is selected by the scheduler.

Example 2: Counters

Liam will demonstrate a program that exhibits non-deterministic behaviour due to scheduling.

Explicit non-determinism

You can also add explicit non-determinism using `if` and `do` blocks:

```
if
:: (n % 2 != 0) -> n = 1;
:: (n >= 0) -> n = n - 2;
:: (n % 3 == 0) -> n = 3;
:: else -> skip;
fi
```

What would happen without the `else` line?

Guards

The arrows in the previous slide are just sugar for semicolons:

```
if
:: (n % 2 != 0); n = 1;
:: (n >= 0); n = n - 2;
:: (n % 3 == 0); n = 3;
fi
```

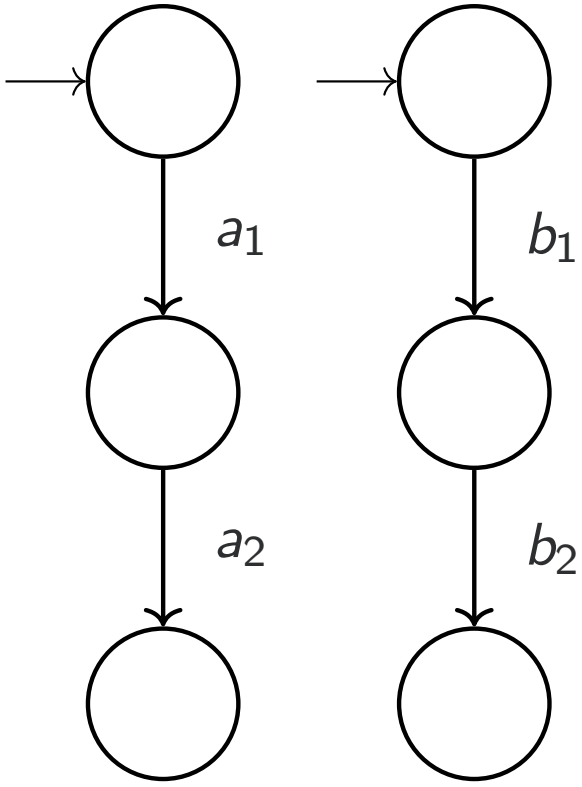
A boolean expression by itself forms a *guard*. Execution can only progress past a guard if the boolean expression evaluates to true (non-zero).
Recall a state with no outgoing transitions is called deadlock.
SPIN can detect deadlock in Promela programs.

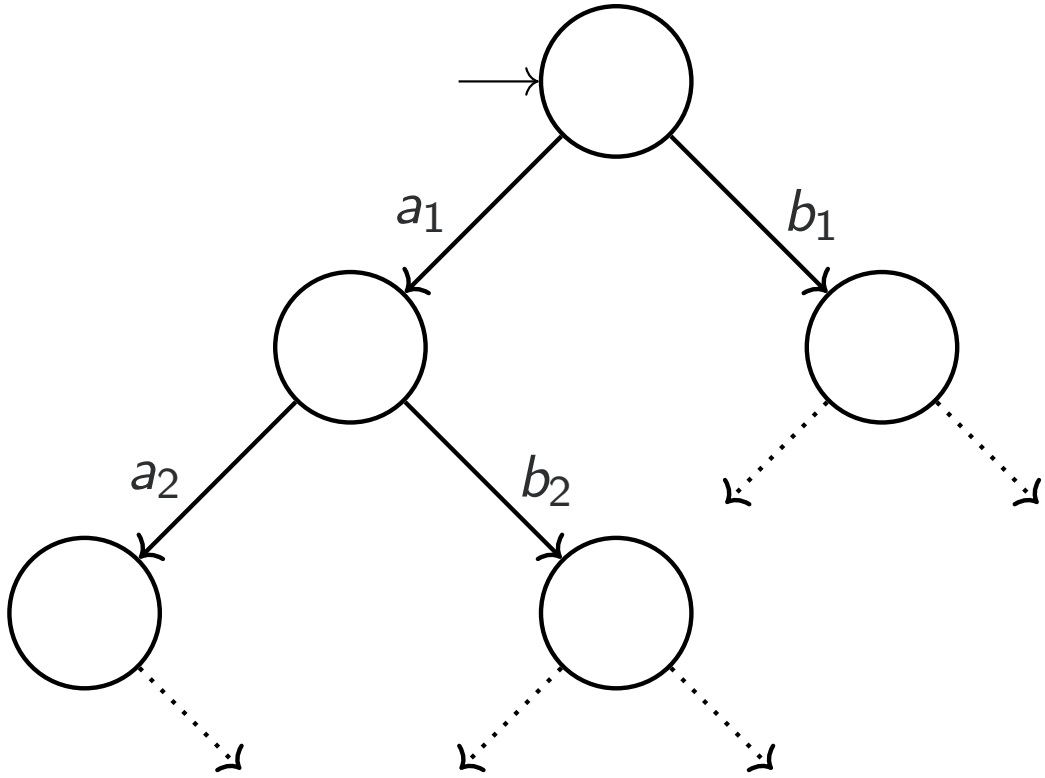
`mtype` and Looping

```
mtype = {RED, YELLOW, GREEN};
active proctype TrafficLight() {
    mtype state = GREEN;
    do
    :: (state == GREEN) -> state = YELLOW;
    :: (state == YELLOW) -> state = RED;
    :: (state == RED) -> state = GREEN;
    od
}
```

Non-determinism can be avoided by making guards mutually exclusive. Exit loops with `break`.

State Space Explosion

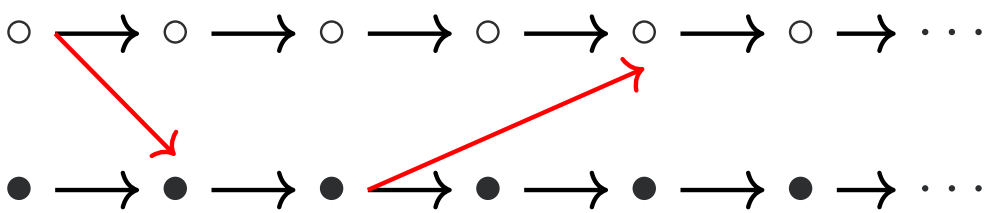




The number of concurrent interleavings gets very large the more processes we add.

Synchronisation

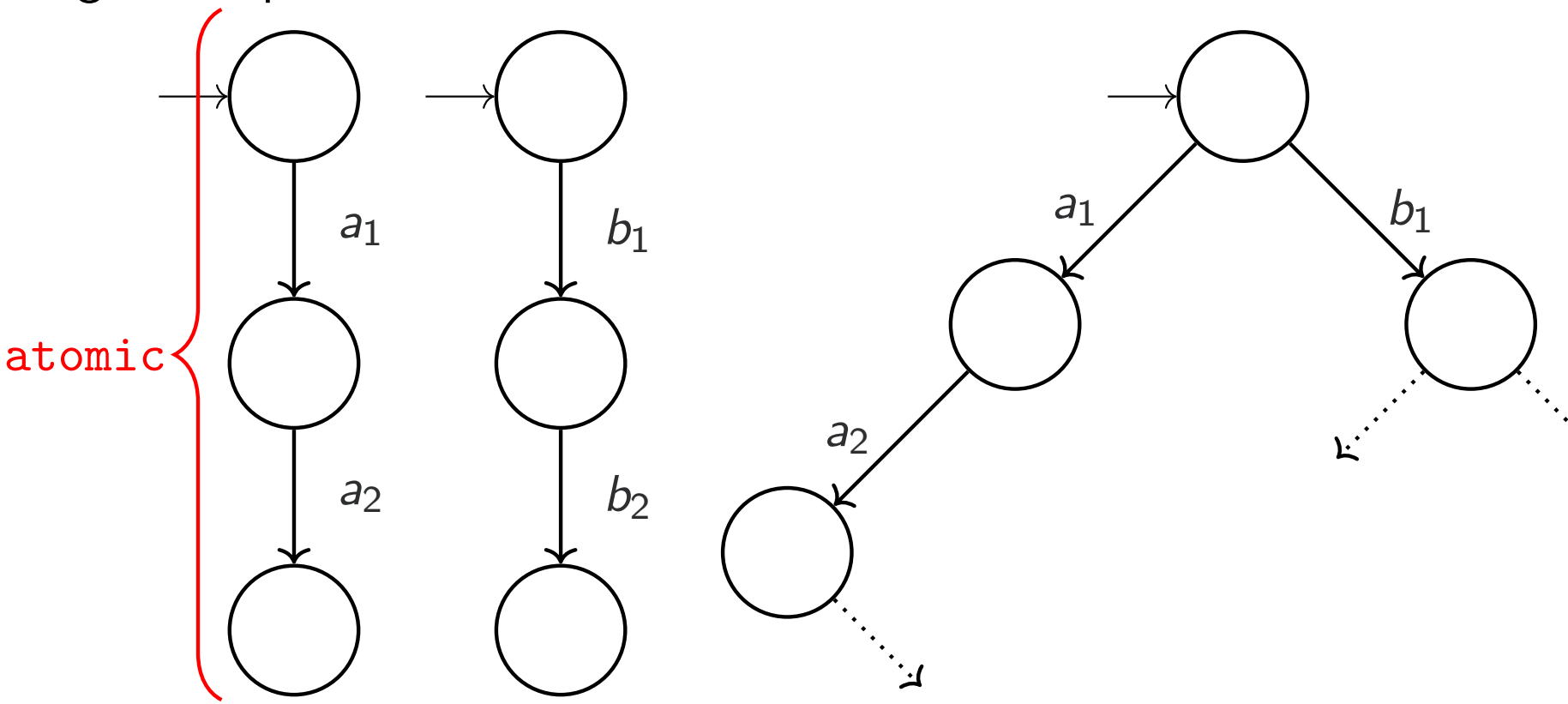
In order to reduce the number of possible interleavings, we must allow processes to synchronise their behaviour, ensuring more orderings (and thus fewer interleavings).



The red arrows are synchronisations.
The most common synchronisation problem is the *critical section problem*, which we will discuss later. Promela includes some synchronisation primitives, however.

`atomic` and `d_step`

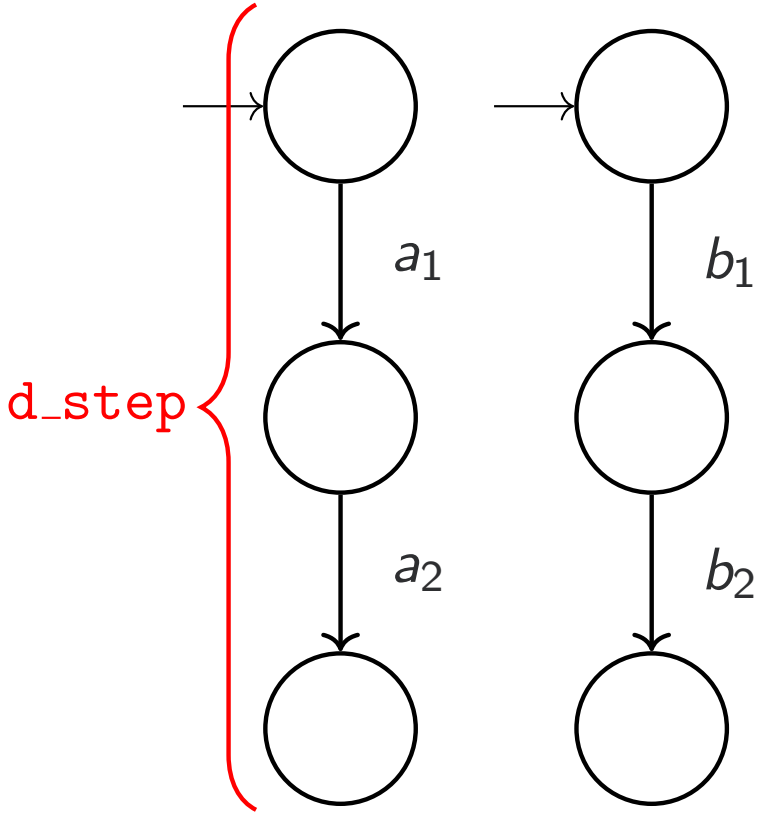
Grouping statements in Promela with `atomic` prevents them from being interrupted.

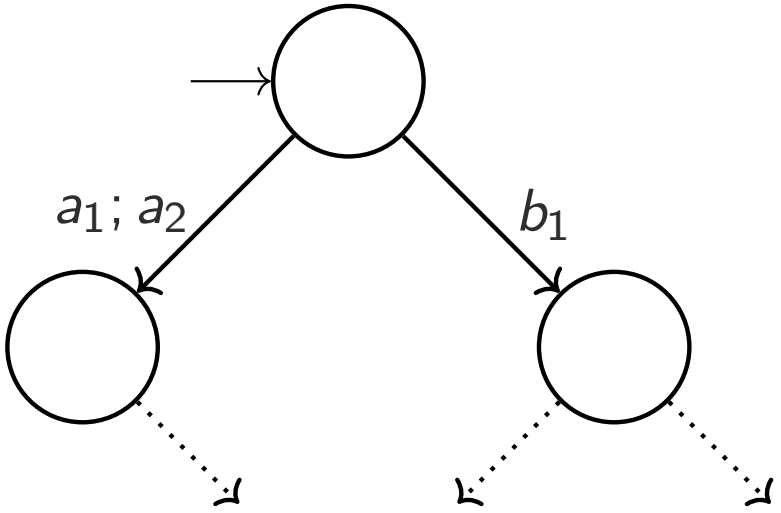


If a statement in an atomic block is blocked, atomicity is temporarily suspended and another process may run.

`atomic` and `d_step`

Grouping statements with `d_step` is more efficient than `atomic`, as it groups them all into **one transition**.





Non-determinism (`if`,`do`) is not allowed in `d_step`. If a statement in the block blocks, a runtime error is raised.

Critical Section Problems

In the Real World™, we don't have the luxury of `atomic` and `d_step` blocks. To solve this for real systems, we need solutions to the *critical section problem*.
A sketch of the problem can be outlined as follows:

forever do	**forever do**
non-critical section	*non-critical section*
pre-protocol	*pre-protocol*
critical section	**critical section**
post-protocol	*post-protocol*

The non-critical section models the possibility that a process may do something else. It can take any amount of time (even infinite).
Our task is to find a pre- and post-protocol such that certain atomicity properties are satisfied.

Desiderata

We want to ensure two main properties:

- **Mutual Exclusion** No two processes are in their critical section at the same time.
- **Eventual Entry** (or *starvation-freedom*) Once it enters its pre-protocol, a process will eventually be able to execute its critical section.

Question

Which is safety and which is liveness?
Mutex is safety, Eventual Entry is liveness.

Let's use SPIN verify our solutions!

Assertions

We can make boolean assertions with `assert`, just like in C, but the SPIN verifier can check our assertions for us.
Combined with a monitor process, it is a useful way to check safety properties.
Liam will use this method to verify a simple mutual exclusion property for a critical section solution.

Labels

You can label lines of code with a label name followed by a colon, as in C. Labels can also be used in boolean expressions, where `P@label` evaluates to true iff process `P` is at the label marked `label`.

Liam: Demonstrate using labels on the critical section solution.

There are also special labels used to denote special states for model checking:

End states occur automatically at the end of processes or on states labelled with labels beginning with `end`. Terminating in these states is not considered as deadlock.

Progress states must be visited infinitely often (liveness). Labels start with `progress`.

Acceptance states must not be visited infinitely often. Labels start with `accept`.

In general these states are for use in *never claims*.

Never Claims

A process defined in a `never` block runs in lock-step with all other processes.

Never Claim execution

Each transition of a regular process must be matched with a transition of the `never` process.

This can be used to restrict the search space to ones where certain properties hold, or to check invariants.

Liam will demonstrate checking Mutual Exclusion with a `never` claim.

Question

What would the following `never` claim do?

```
never {
    true;
}
```

LTL formulae

Spin can translate LTL formulae to Never claims automatically. Liam will demonstrate verifying safety (mutex) and liveness (eventual entry) using LTL claims.

Eventual Entry!

Our naive solution doesn't satisfy eventual entry, even with *weak fairness*.

Liam will try a few more examples, ultimately leading up to Peterson's algorithm.

Channels

A channel is a FIFO queue of messages (which can consist of multiple pieces of data) that is shared between processes.

```
chan c = [10] of { mtype, byte, int }
```

Creates a channel of size 10 with messages consisting of an mtype, a byte and an int.

```
c ? v1, v2, v3;
```

Removes a message from a channel, storing the data in the variables `v1,v2,v3`. If the channel is empty, it blocks. You can also put literal values in place of variables to pattern match.

```
c ! RED, 255, 32767;
```

Sends a message into the channel. If the channel is full, block.

Synchronous communication

If channel size is zero, the channel is *synchronous*. This means that each send must be paired with a matching receive, and they both execute together.
Liam will demonstrate modelling a lock using:

- synchronous message passing, and
- an asynchronous channel with an atomicity token.

Bibliography

- M.Ben-Ari, Principles of Concurrent and Distributed Programming
- `http://spinroot.com`