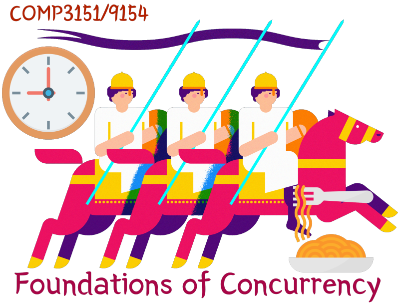


Linear Temporal Logic, Critical Sections and Promela Modelling

Johannes Åman Pohjola
CSE, UNSW
Term 2 2022

Where are we?

Last Lecture

We saw how to treat the semantics of concurrent programs and the properties they should satisfy.

This Lecture

We will give a syntactic way to specify properties (Temporal Logic) and introduce one of two methods we will cover to show properties hold (Model Checking) using the famous Critical Section problem.

Logic

We typically state our requirements with a logic.

Logic

We typically state our requirements with a logic.

Definition

A logic is a formal language designed to express logical reasoning. Like any formal language, logics have a syntax and semantics.

Logic

We typically state our requirements with a logic.

Definition

A logic is a formal language designed to express logical reasoning. Like any formal language, logics have a syntax and semantics.

Example (Propositional Logic Syntax)

- A set of atomic propositions $\mathcal{P} = \{a, b, c, \ldots\}$
- An inductively defined set of formulae:
  - Each $p \in \mathcal{P}$ is a formula.
  - If P and Q are formulae, then $P \wedge Q$ is a formula.
  - If P is a formula, then $\neg P$ is a formula.

(Other connectives are just sugar for these, so we omit them)

Semantics

Semantics

Semantics are a mathematical representation of the meaning of a piece of syntax. We will use models to give semantics to logic.

Semantics

Semantics are a mathematical representation of the meaning of a piece of syntax. We will use models to give semantics to logic.

Example (Propositional Logic Semantics)

A model for propositional logic is a valuation $\mathcal{V} \subseteq \mathcal{P}$, a set of "true" atomic propositions. We can extend a valuation over an entire formula, giving us a satisfaction relation:

$$\begin{array}{lcl} \mathcal{V} \models p & \Leftrightarrow & p \in \mathcal{V} \\ \mathcal{V} \models \varphi \wedge \psi & \Leftrightarrow & \mathcal{V} \models \varphi \text{ and } \mathcal{V} \models \psi \\ \mathcal{V} \models \neg\varphi & \Leftrightarrow & \mathcal{V} \not\models \varphi \end{array}$$

We read $\mathcal{V} \models \varphi$ as $\mathcal{V}$ "satisfies" φ.

LTL

Linear temporal logic (LTL) is a *logic* designed to describe linear time properties.

Linear temporal logic syntax

We have normal propositional operators:

- $p \in \mathcal{P}$ is an LTL formula.
- If φ, ψ are LTL formulae, then $\varphi \wedge \psi$ is an LTL formula.
- If φ is an LTL formula, $\neg\varphi$ is an LTL formula.

LTL

Linear temporal logic (LTL) is a *logic* designed to describe linear time properties.

Linear temporal logic syntax

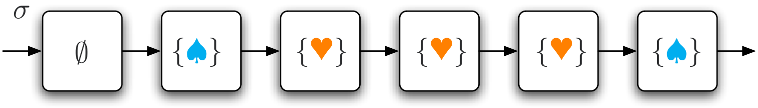
We have normal propositional operators:

- $p \in \mathcal{P}$ is an LTL formula.
- If φ, ψ are LTL formulae, then $\varphi \wedge \psi$ is an LTL formula.
- If φ is an LTL formula, $\neg\varphi$ is an LTL formula.

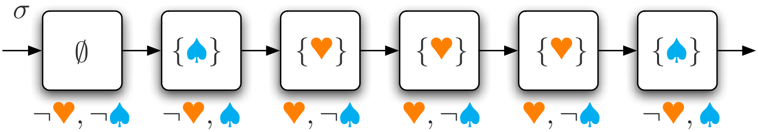
We also have modal or temporal operators:

- If φ is an LTL formula, then $\bigcirc\ \varphi$ is an LTL formula.
- If φ, ψ are LTL formulae, then $\varphi\ \mathcal{U}\ \psi$ is an LTL formula.

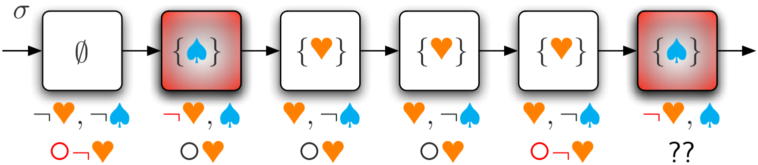
LTL Semantics in Pictures



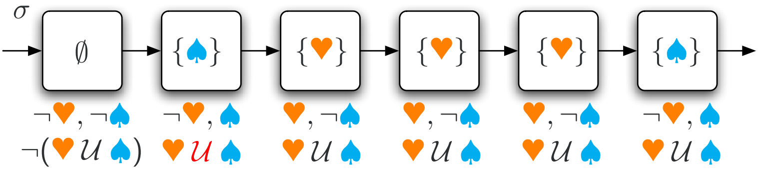
LTL Semantics in Pictures



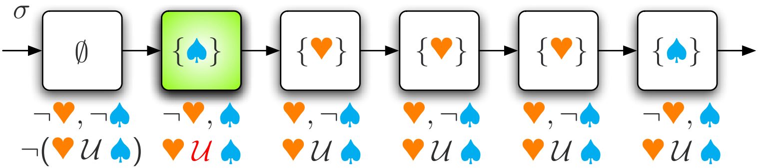
LTL Semantics in Pictures



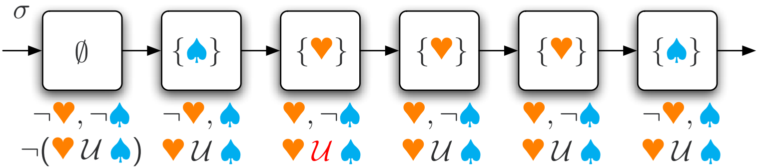
LTL Semantics in Pictures



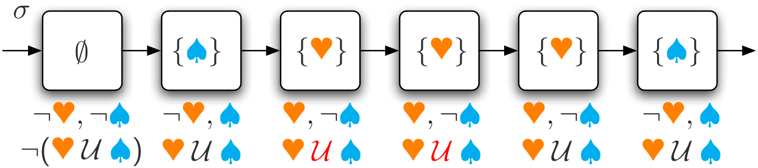
LTL Semantics in Pictures



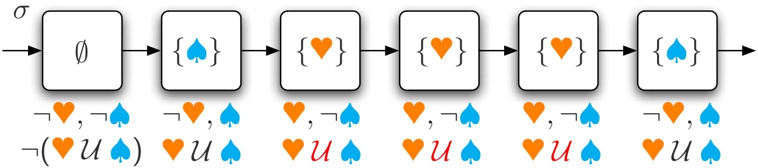
LTL Semantics in Pictures



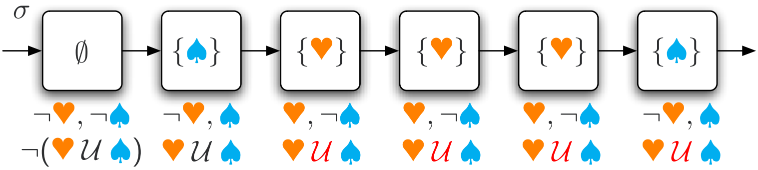
LTL Semantics in Pictures



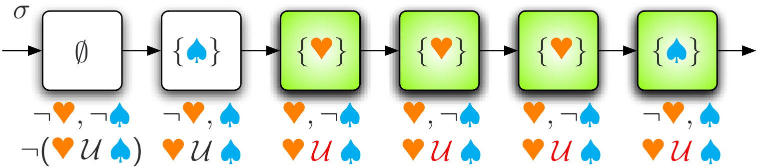
LTL Semantics in Pictures



LTL Semantics in Pictures



LTL Semantics in Pictures



LTL Semantics

Let $\sigma = \sigma_0\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\ldots$ be a behaviour. Then define notation:

- $\sigma|_0 = \sigma$
- $\sigma|_1 = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\ldots$
- $\sigma|_{n+1} = (\sigma|_1)|_n$

LTL Semantics

Let $\sigma = \sigma_0\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\ldots$ be a behaviour. Then define notation:

- $\sigma|_0 = \sigma$
- $\sigma|_1 = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\ldots$
- $\sigma|_{n+1} = (\sigma|_1)|_n$

Semantics

The models of LTL are behaviours. For atomic propositions, we just look at the first state. We often identify states with the set of atomic propositions they satisfy.

$$\sigma \models p \qquad \Leftrightarrow \quad p \in \sigma_0$$

LTL Semantics

Let $\sigma = \sigma_0\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\dots$ be a behaviour. Then define notation:

- $\sigma|_0 = \sigma$
- $\sigma|_1 = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\dots$
- $\sigma|_{n+1} = (\sigma|_1)|_n$

Semantics

The models of LTL are behaviours. For atomic propositions, we just look at the first state. We often identify states with the set of atomic propositions they satisfy.

$$
\begin{array}{lcl}
\sigma \models p & \Leftrightarrow & p \in \sigma_0 \\
\sigma \models \varphi \land \psi & \Leftrightarrow & \sigma \models \varphi \text{ and } \sigma \models \psi \\
\sigma \models \neg\varphi & \Leftrightarrow & \sigma \not\models \varphi
\end{array}
$$

LTL Semantics

Let $\sigma = \sigma_0\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\ldots$ be a behaviour. Then define notation:

- $\sigma|_0 = \sigma$
- $\sigma|_1 = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\ldots$
- $\sigma|_{n+1} = (\sigma|_1)|_n$

Semantics

The models of LTL are behaviours. For atomic propositions, we just look at the first state. We often identify states with the set of atomic propositions they satisfy.

$$
\begin{array}{lcl}
\sigma \models p & \Leftrightarrow & p \in \sigma_0 \\
\sigma \models \varphi \wedge \psi & \Leftrightarrow & \sigma \models \varphi \text{ and } \sigma \models \psi \\
\sigma \models \neg\varphi & \Leftrightarrow & \sigma \not\models \varphi \\
\sigma \models \bigcirc\, \varphi & \Leftrightarrow & \sigma|_1 \models \varphi
\end{array}
$$

LTL Semantics

Let $\sigma = \sigma_0\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\ldots$ be a behaviour. Then define notation:

- $\sigma|_0 = \sigma$
- $\sigma|_1 = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\ldots$
- $\sigma|_{n+1} = (\sigma|_1)|_n$

Semantics

The models of LTL are behaviours. For atomic propositions, we just look at the first state. We often identify states with the set of atomic propositions they satisfy.

$$
\begin{array}{lcl}
\sigma \models p & \Leftrightarrow & p \in \sigma_0 \\
\sigma \models \varphi \wedge \psi & \Leftrightarrow & \sigma \models \varphi \text{ and } \sigma \models \psi \\
\sigma \models \neg\varphi & \Leftrightarrow & \sigma \not\models \varphi \\
\sigma \models \bigcirc\, \varphi & \Leftrightarrow & \sigma|_1 \models \varphi \\
\sigma \models \varphi\ \mathcal{U}\ \psi & \Leftrightarrow & \text{There exists an } i \text{ such that } \sigma|_i \models \psi \\
 & & \text{and for all } j < i,\ \sigma|_j \models \varphi
\end{array}
$$

We say $P \models \varphi$ iff $\forall \sigma \in [\![P]\!].\ \sigma \models \varphi$.

Derived Operators

The operator $\Diamond\ \varphi$ ("finally" or "eventually") says that φ will be true at some point.

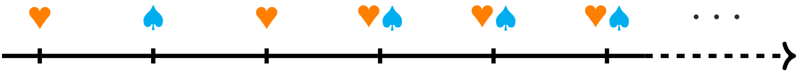
The operator $\Box\ \varphi$ ("globally" or "always") says that φ is always true from now on.

Exercise

- Give the semantics of $\Box$ and $\Diamond$.
- Define $\Box$ and $\Diamond$ in terms of other operators.

More Exercises

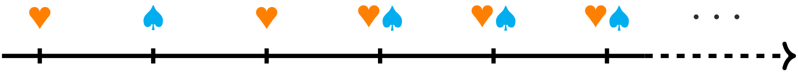
Let ρ be this behaviour:



$\rho \models$ ♥?

More Exercises

Let ρ be this behaviour:

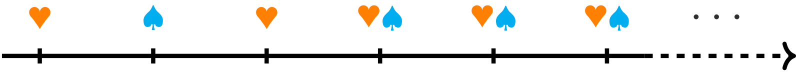


$\rho \models \heartsuit$?

$\rho \models \spadesuit$?

More Exercises

Let ρ be this behaviour:



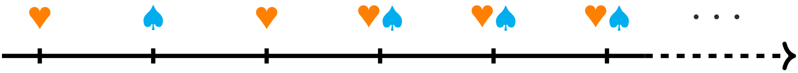
$\rho \models \heartsuit$?

$\rho \models \spadesuit$?

$\rho \models \bigcirc\ \spadesuit$?

More Exercises

Let ρ be this behaviour:



$\rho \models \heartsuit$?

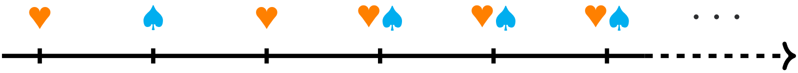
$\rho \models \spadesuit$?

$\rho \models \bigcirc\ \spadesuit$?

$\rho \models \Diamond\ \heartsuit$?

More Exercises

Let ρ be this behaviour:



$\rho \models \heartsuit$?

$\rho \models \spadesuit$?

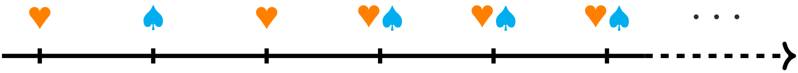
$\rho \models \bigcirc\ \spadesuit$?

$\rho \models \Diamond\ \heartsuit$?

$\rho|_3 \models \Diamond\ (\heartsuit \land \neg\spadesuit)$?

More Exercises

Let ρ be this behaviour:



$\rho \models \heartsuit$?

$\rho \models \spadesuit$?

$\rho \models \bigcirc\ \spadesuit$?

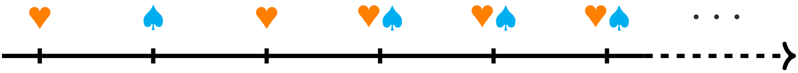
$\rho \models \Diamond\ \heartsuit$?

$\rho|_3 \models \Diamond\ (\heartsuit \wedge \neg\spadesuit)$?

$\rho \models \Diamond\Box\ (\heartsuit \wedge \spadesuit)$?

More Exercises

Let ρ be this behaviour:



$\rho \models \heartsuit$?

$\rho \models \spadesuit$?

$\rho \models \bigcirc\, \spadesuit$?

$\rho \models \Diamond\, \heartsuit$?

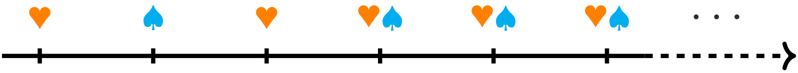
$\rho|_3 \models \Diamond\, (\heartsuit \land \neg\spadesuit)$?

$\rho \models \Diamond\Box\, (\heartsuit \land \spadesuit)$?

$\rho \models \Box\, (\heartsuit\ \mathcal{U}\ \spadesuit)$?

More Exercises

Let ρ be this behaviour:



More Derived Operators

- Define "Infinitely Often" in LTL.
- Define "Almost Globally" in LTL (always true from some point onwards).

$\rho \models \heartsuit$?

$\rho \models \spadesuit$?

$\rho \models \bigcirc\ \spadesuit$?

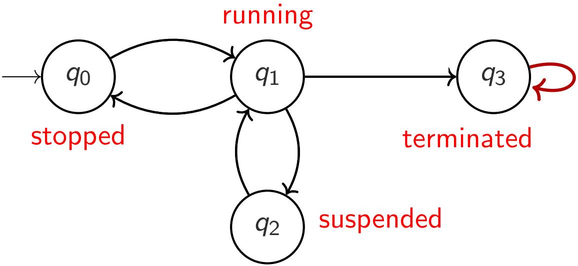
$\rho \models \Diamond\ \heartsuit$?

$\rho|_3 \models \Diamond\ (\heartsuit \land \neg\spadesuit)$?

$\rho \models \Diamond\Box\ (\heartsuit \land \spadesuit)$?

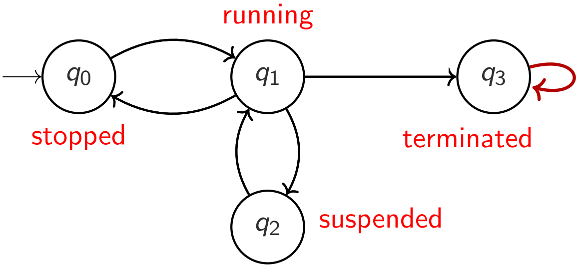
$\rho \models \Box\ (\heartsuit\ \mathcal{U}\ \spadesuit)$?

Possible Futures



We can see that it is always possible for a run to move to the terminated state. How do we express this in LTL?

Possible Futures



We can see that it is always possible for a run to move to the terminated state. How do we express this in LTL? **We can't!** — it is a *branching time* property.

Branching Time

Dealing with branching time properties requires a different logic called CTL (Computation Tree Logic). Learn about it in COMP3153/9153 or COMP6752.

A counting argument for mechanical aids

How many scenarios are there for a program with n finite processes consisting of m atomic actions each?

A counting argument for mechanical aids

How many scenarios are there for a program with n finite processes consisting of m atomic actions each?

$$\frac{(nm)!}{m!^n}$$

A counting argument for mechanical aids

How many scenarios are there for a program with n finite processes consisting of m atomic actions each?

$$\frac{(nm)!}{m!^n}$$

	$n=2$	3	4	5	6
$m=2$	6	90	2520	113400	$2^{22.8}$
3	20	1680	$2^{18.4}$	$2^{27.3}$	$2^{36.9}$
4	70	34650	$2^{25.9}$	$2^{38.1}$	$2^{51.5}$
5	252	$2^{19.5}$	$2^{33.4}$	$2^{49.1}$	$2^{66.2}$
6	924	$2^{24.0}$	$2^{41.0}$	$2^{60.2}$	$2^{81.1}$

A counting argument for mechanical aids

How many scenarios are there for a program with n finite processes consisting of m atomic actions each?

$$\frac{(nm)!}{m!^n}$$

	$n = 2$	3	4	5	6
$m = 2$	6	90	2520	113400	$2^{22.8}$
3	20	1680	$2^{18.4}$	$2^{27.3}$	$2^{36.9}$
4	70	34650	$2^{25.9}$	$2^{38.1}$	$2^{51.5}$
5	252	$2^{19.5}$	$2^{33.4}$	$2^{49.1}$	$2^{66.2}$
6	924	$2^{24.0}$	$2^{41.0}$	$2^{60.2}$	$2^{81.1}$

So, for 6 processes consisting of 6 sequential atomic actions each, that's merely 2 670 177 736 637 149 247 308 800 scenarios.

A counting argument for mechanical aids

How many scenarios are there for a program with n finite processes consisting of m atomic actions each?

$$\frac{(nm)!}{m!^n}$$

	$n = 2$	3	4	5	6
$m = 2$	6	90	2520	113400	$2^{22.8}$
3	20	1680	$2^{18.4}$	$2^{27.3}$	$2^{36.9}$
4	70	34650	$2^{25.9}$	$2^{38.1}$	$2^{51.5}$
5	252	$2^{19.5}$	$2^{33.4}$	$2^{49.1}$	$2^{66.2}$
6	924	$2^{24.0}$	$2^{41.0}$	$2^{60.2}$	$2^{81.1}$

So, for 6 processes consisting of 6 sequential atomic actions each, that's merely 2 670 177 736 637 149 247 308 800 scenarios.

Do come back when you're done testing!

Sobering Conclusion

For any realistic concurrent program, it is *infeasible to test* all possible scenarios.

We need to apply smarter techniques than brute-force testing to establish properties of concurrent programs.
Formal methods let us reason about programs, or, if that is too hard, about *abstractions* of programs.

Industrially applicable formal methods

To verify that program P has property φ (i.e. $P \models \varphi$), we can use:

- *model checking* — exhaustively searching through (an efficient representation of) P's state space to find a *counterexample* to φ
- *theorem proving* — construct a (formal) proof of φ

To be relevant in practice, these techniques must be supported by *tools*.

Model Checking

Pros: easy to use push-button technology; instructive counter examples (error traces) help debugging

Cons: *state (space) explosion problem*

Model Checking

Pros: easy to use push-button technology; instructive counter examples (error traces) help debugging

Cons: *state (space) explosion problem*

Question

Where can I learn more about model checking?

Model Checking

Pros: easy to use push-button technology; instructive counter examples (error traces) help debugging

Cons: *state (space) explosion problem*

Question

Where can I learn more about model checking?

Answer

COMP3153/9153 *Algorithmic Verification* (should run in T2)

(Interactive) Theorem Proving

Pros: no (theoretical) limits on state spaces

Cons: *requires* expert users (e.g. *skilled computer scientists*, mathematicians, or logicians) to hand-crank through proofs

(Interactive) Theorem Proving

Pros: no (theoretical) limits on state spaces

Cons: *requires* expert users (e.g. *skilled computer scientists*, mathematicians, or logicians) to hand-crank through proofs

Question

Where can I learn more about interactive theorem proving?

(Interactive) Theorem Proving

Pros: no (theoretical) limits on state spaces

Cons: *requires* expert users (e.g. *skilled computer scientists*, mathematicians, or logicians) to hand-crank through proofs

Question

Where can I learn more about interactive theorem proving?

Answer

COMP4161 *Advanced Verification* (should run in T3)

SPIN

A model checker for concurrent systems with a lot of useful features and support for LTL model checking.

`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.

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*.

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`.

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.

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*.

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.

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.

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

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

Example 1: Hello World

Johannes 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.

Example 2: Counters

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

Example 2: Counters

Johannes 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
```

Example 2: Counters

Johannes 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).
If the entire system cannot make progress, that 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`.

Volatile Variables

var $y, z \leftarrow 0, 0$	
p_1: **var** x;	q_1: $y \leftarrow 1$;
p_2: $x \leftarrow y + z$;	q_2: $z \leftarrow 2$;

Question

What are the possible final values of x?

Volatile Variables

var $y, z \leftarrow 0, 0$	
p_1: **var** x;	q_1: $y \leftarrow 1$;
p_2: $x \leftarrow y + z$;	q_2: $z \leftarrow 2$;

Question

What are the possible final values of x?

What about $x = 2$? Is that possible?

Volatile Variables

var $y, z \leftarrow 0, 0$			
p_1:	**var** x;	q_1:	$y \leftarrow 1$;
p_2:	$x \leftarrow y + z$;	q_2:	$z \leftarrow 2$;

Question

What are the possible final values of x?
What about $x = 2$? Is that possible?
It is possible, as we cannot guarantee that the statement p_2 is executed atomically — that is, as one step.

Volatile Variables

var $y, z \leftarrow 0, 0$	
p_1: **var** x;	q_1: $y \leftarrow 1$;
p_2: $x \leftarrow y + z$;	q_2: $z \leftarrow 2$;

Question

What are the possible final values of x?
What about $x = 2$? Is that possible?
It is possible, as we cannot guarantee that the statement p_2 is executed atomically — that is, as one step.

Typically, we require that each statement only accesses (reads from or writes to) at most one shared variable at a time. Otherwise, we cannot guarantee that each statement is one atomic step. This is called the *limited critical reference* restriction.

Ensuring Atomicity

We will often have multiple actions that we wish to group into one step, i.e. to execute atomically.

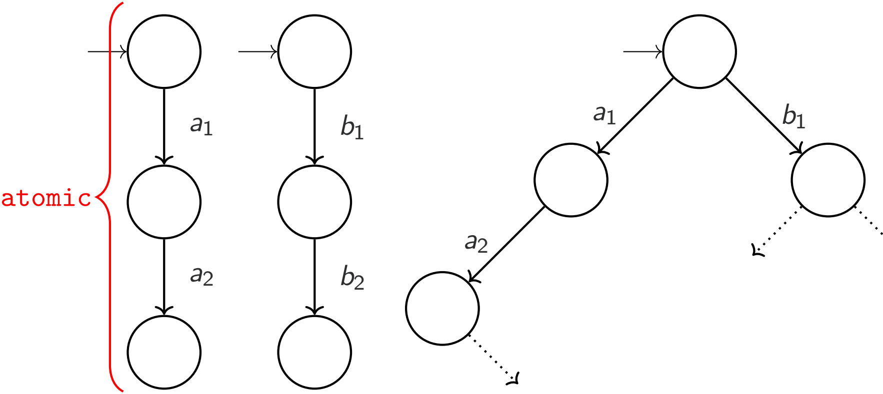
Example (Counters)

In our counter example, if each process executes the loop body atomically the result number can be guaranteed.

In Promela we can simply state this requirement, but in real programming languages we must use synchronisation techniques to achieve this.

`atomic` and `d_step`

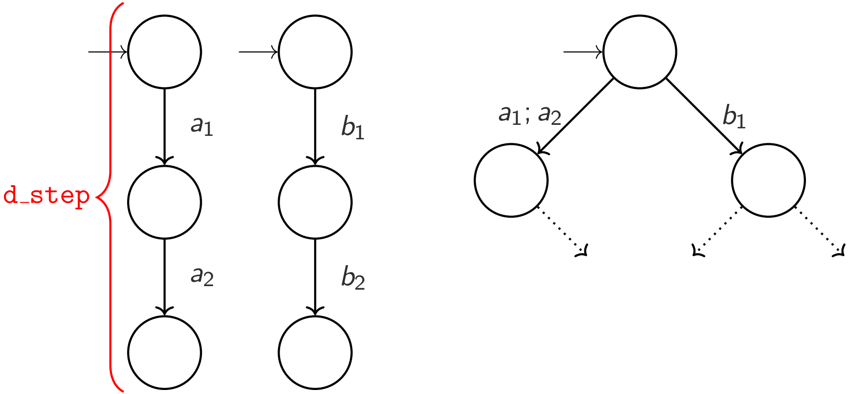
Grouping statements in Promela with `atomic` prevents them from being interrupted.



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.

Atomicity

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*.

Atomicity

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*

Atomicity

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).

Atomicity

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 and two secondary ones:

- **Mutual Exclusion** No two processes are in their critical section at the same time.

Desiderata

We want to ensure two main properties and two secondary ones:

- **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.

Desiderata

We want to ensure two main properties and two secondary ones:

- **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.
- **Absence of Deadlock** The system will never reach a state where no actions can be taken from any process.

Desiderata

We want to ensure two main properties and two secondary ones:

- **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.
- **Absence of Deadlock** The system will never reach a state where no actions can be taken from any process.
- **Absence of Unneccessary Delay** If only one process is attempting to enter its critical section, it is not prevented from doing so.

Desiderata

We want to ensure two main properties and two secondary ones:

- **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.
- **Absence of Deadlock** The system will never reach a state where no actions can be taken from any process.
- **Absence of Unneccessary Delay** If only one process is attempting to enter its critical section, it is not prevented from doing so.

Question

Which is safety and which is liveness?

Desiderata

We want to ensure two main properties and two secondary ones:

- **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.
- **Absence of Deadlock** The system will never reach a state where no actions can be taken from any process.
- **Absence of Unneccessary Delay** If only one process is attempting to enter its critical section, it is not prevented from doing so.

Question

Which is safety and which is liveness?

Eventual Entry is liveness, the rest are safety.

First Attempt

We can implement **await** using primitive machine instructions or OS syscalls, or even using a busy-waiting loop.

var *turn* $\leftarrow 1$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = 1;$	q_2 **await** $turn = 2;$
p_3 **critical section**	q_3 **critical section**
p_4 $turn \leftarrow 2$	q_4 $turn \leftarrow 1$

Question

Mutual Exclusion?

First Attempt

We can implement **await** using primitive machine instructions or OS syscalls, or even using a busy-waiting loop.

var *turn* $\leftarrow 1$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = 1$;	q_2 **await** $turn = 2$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn \leftarrow 2$	q_4 $turn \leftarrow 1$

Question

Mutual Exclusion? Yup!

First Attempt

We can implement **await** using primitive machine instructions or OS syscalls, or even using a busy-waiting loop.

var *turn* $\leftarrow 1$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = 1$;	q_2 **await** $turn = 2$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn \leftarrow 2$	q_4 $turn \leftarrow 1$

Question

Mutual Exclusion? Yup!
Other criteria?

First Attempt

We can implement **await** using primitive machine instructions or OS syscalls, or even using a busy-waiting loop.

var $turn \leftarrow 1$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = 1$;	q_2 **await** $turn = 2$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn \leftarrow 2$	q_4 $turn \leftarrow 1$

Question

Mutual Exclusion? Yup!
Other criteria? Nope!

First Attempt

We can implement **await** using primitive machine instructions or OS syscalls, or even using a busy-waiting loop.

var *turn* $\leftarrow 1$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = 1$;	q_2 **await** $turn = 2$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn \leftarrow 2$	q_4 $turn \leftarrow 1$

Question

Mutual Exclusion? Yup!
Other criteria? Nope! What if q_1 never finishes?

Second Attempt

var *wantp*, *wantq* $\leftarrow$ False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** *wantq* = False;	q_2 **await** *wantp* = False;
p_3 *wantp* $\leftarrow$ True;	q_3 *wantq* $\leftarrow$ True;
p_4 **critical section**	q_4 **critical section**
p_7 *wantp* $\leftarrow$ False	q_7 *wantq* $\leftarrow$ False

Second Attempt

var *wantp*, *wantq* ← False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** *wantq* = False;	q_2 **await** *wantp* = False;
p_3 *wantp* ← True;	q_3 *wantq* ← True;
p_4 **critical section**	q_4 **critical section**
p_7 *wantp* ← False	q_7 *wantq* ← False

Mutual exclusion is violated if they execute in lock-step (i.e. $p_1q_1p_2q_2p_3q_3$ etc.)

Third Attempt

var *wantp*, *wantq* ← False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *wantp* ← True;	q_2 *wantq* ← True;
p_3 **await** *wantq* = False;	q_3 **await** *wantp* = False;
p_4 **critical section**	q_4 **critical section**
p_7 *wantp* ← False	q_7 *wantq* ← False

Third Attempt

var *wantp*, *wantq* ← False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 $\mathit{wantp} \leftarrow \mathrm{True}$;	q_2 $\mathit{wantq} \leftarrow \mathrm{True}$;
p_3 **await** $\mathit{wantq} = \mathrm{False}$;	q_3 **await** $\mathit{wantp} = \mathrm{False}$;
p_4 **critical section**	q_4 **critical section**
p_7 $\mathit{wantp} \leftarrow \mathrm{False}$	q_7 $\mathit{wantq} \leftarrow \mathrm{False}$

Now we have a deadlock (or stuck state) if they proceed in lock step.

Fourth Attempt

var *wantp*, *wantq* ← False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *wantp* ← True;	q_2 *wantq* ← True;
p_3 **while** *wantq* **do**	q_3 **while** *wantp* **do**
p_4 *wantp* ← False;	q_4 *wantq* ← False;
p_5 *wantp* ← True	q_5 *wantq* ← True
p_6 **critical section**	q_6 **critical section**
p_7 *wantp* ← False	q_7 *wantq* ← False

Fourth Attempt

var $wantp, wantq \leftarrow$ False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 $wantp \leftarrow$ True;	q_2 $wantq \leftarrow$ True;
p_3 **while** $wantq$ **do**	q_3 **while** $wantp$ **do**
p_4 $wantp \leftarrow$ False;	q_4 $wantq \leftarrow$ False;
p_5 $wantp \leftarrow$ True	q_5 $wantq \leftarrow$ True
p_6 **critical section**	q_6 **critical section**
p_7 $wantp \leftarrow$ False	q_7 $wantq \leftarrow$ False

We have replaced the deadlock with live lock (looping) if they continuously proceed in lock-step.

Fifth Attempt

var *wantp*, *wantq* ← False, False var *turn* ← 1	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *wantp* = True;	q_2 *wantq* = True;
p_3 **while** *wantq* **do**	q_3 **while** *wantp* **do**
p_4 **if** *turn* = 2 **then**	q_4 **if** *turn* = 1 **then**
p_5 *wantp* ← False;	q_5 *wantq* ← False;
p_6 **await** *turn* = 1;	q_6 **await** *turn* = 2;
p_7 *wantp* ← True	q_7 *wantq* ← True
p_8 **critical section**	q_8 **critical section**
p_9 *turn* ← 2	q_9 *turn* ← 1
p_{10} *wantp* ← False	q_{10} *wantq* ← False

Reviewing this attempt

The fifth attempt (Dekker's algorithm) works well except if the scheduler pathologically tries to run the loop at $q_3 \cdots q_7$ when $turn = 2$ over and over rather than run the process p (or vice versa).

What would we need to assume to prevent this?

Reviewing this attempt

The fifth attempt (Dekker's algorithm) works well except if the scheduler pathologically tries to run the loop at $q_3 \cdots q_7$ when $turn = 2$ over and over rather than run the process p (or vice versa).
What would we need to assume to prevent this?

Fairness

The *fairness assumption* means that if a process can always make a move, it will eventually be scheduled to make that move.

With this assumption, Dekker's algorithm is correct.

Expressing Fairness in LTL

Let $\mathsf{enabled}(\pi)$ and $\mathsf{taken}(\pi)$ be predicates true in a state iff an action π is enabled, resp., taken.

Examples

Weak fairness for action π is then expressible as:

$$\Box(\Box\mathsf{enabled}(\pi) \Rightarrow \Diamond\mathsf{taken}(\pi))$$

Strong fairness for action π is then expressible as:

$$\Box(\Box\Diamond\mathsf{enabled}(\pi) \Rightarrow \Diamond\mathsf{taken}(\pi))$$

Promela can assume weak fairness when checking models.

What now?

- Do the homework exercises and submit them before the Friday lecture.
- Assignment 0 (warm-up) will be out in W2. You have enough knowledge to start it, but not yet enough to finish it.
- Get spin (and ispin) working on your development environment (or use VLAB/ssh)