Model Checking Concurrent Programs

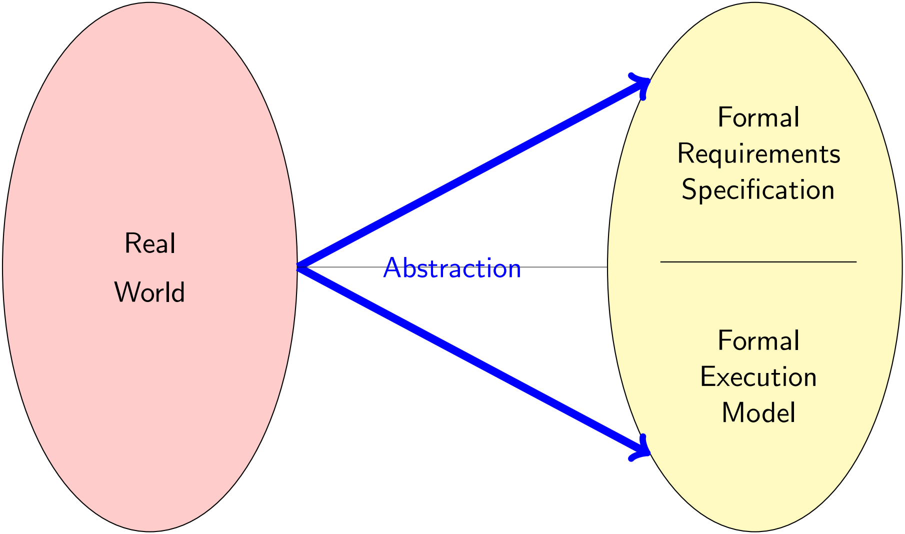
A Taster

Wolfgang Ahrendt

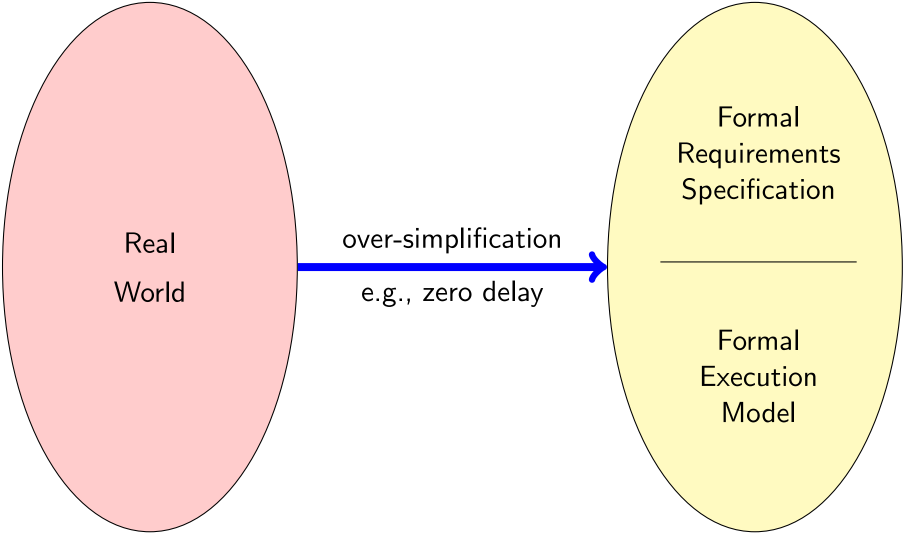
Department of Computer Science and Engineering
Chalmers University of Technology
and
University of Gothenburg

14 October 2013

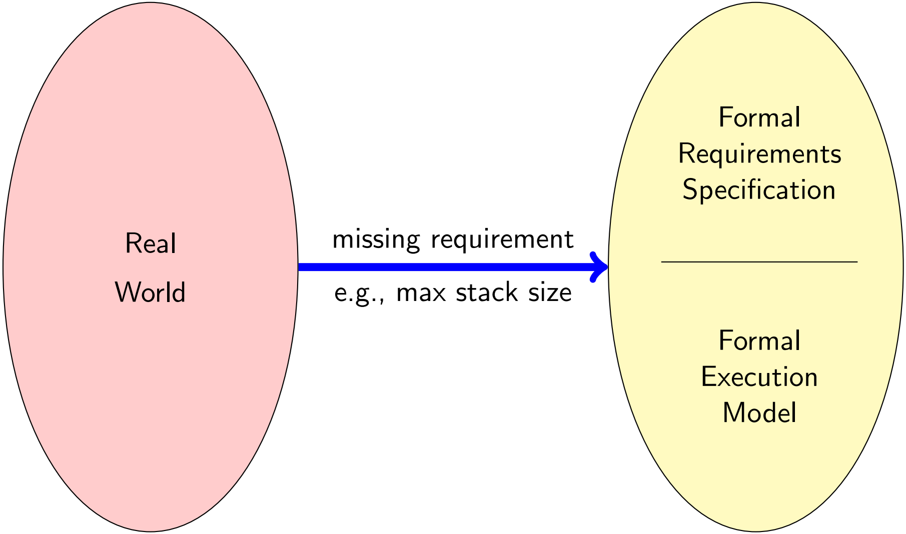
Formal Models for Software



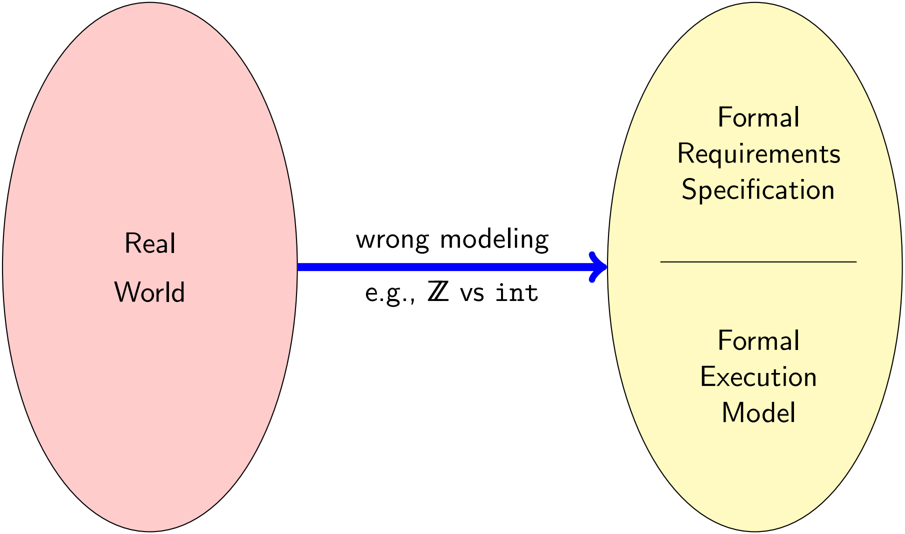
Formal Models for Software



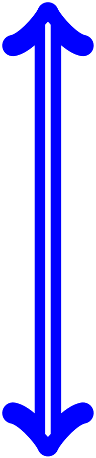
Formal Models for Software



Formal Models for Software



Level of System (Implementation) Description



- Abstract level
  - Finitely many states (finite datatypes)
  - Automated proofs are (in principle) possible
  - Simplification, unfaithful modeling inevitable
- Concrete level
  - Infinite datatypes
    (pointer chains, dynamic arrays, streams)
  - Complex datatypes and control structures,
    general programs
  - Realistic programming model (e.g., Java)
  - Automated proofs (in general) impossible!

Expressiveness of Specification

- Simple
  - Simple or general properties
  - Finitely many case distinctions
  - Approximation, low precision
  - Automated proofs are (in principle) possible
- Complex
  - Full behavioural specification
  - Quantification over infinite domains
  - High precision, tight modeling
  - Automated proofs (in general) impossible!

Main Approaches

Abstract programs, Simple properties	Abstract programs, Complex properties
Concrete programs, Simple properties	Concrete programs, Complex properties

Main Approaches

SPIN
today

Abstract programs, Simple properties	Abstract programs, Complex properties
Concrete programs, Simple properties	Concrete programs, Complex properties

Main Approaches

SPIN
today

Abstract programs, Simple properties	Abstract programs, Complex properties
Concrete programs, Simple properties	Concrete programs, Complex properties

KeY
not today

Proof Automation

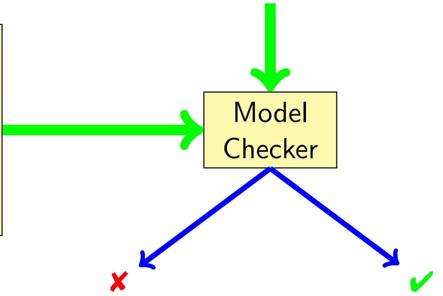
- "Automated" Proof
  ("batch-mode")
  - No interaction during verification necessary
  - Proof may fail or result inconclusive
    Tuning of tool parameters necessary
  - Formal specification still "by hand"

- "Semi-Automated" Proof
  ("interactive")
  - Interaction may be required during proof
  - Need certain knowledge of tool internals
    Intermediate inspection can help
  - Proof is checked by tool

System Model | **System Property**

[] ! (criticalSectP && criticalSectQ)

```
active proctype P() {
   ...
}
active proctype Q() {
   ...
}
```



criticalSectP = 0 1 1
criticalSectQ = 1 0 1

Model Checking in Industry

- Hardware verification
  - Good match between limitations of technology and application
  - Intel, Motorola, AMD, ...
- Software verification
  - Specialized software: control systems, protocols
  - Typically no checking of executable source code, but of abstractions
  - Bell Labs, Ericsson, Microsoft

What is PROMELA?

PROMELA is an acronym

Process meta-language

What is PROMELA?

PROMELA is an acronym

Process meta-language

PROMELA is a language for modeling concurrent systems

- multi-threaded

What is PROMELA?

PROMELA is an acronym

Process meta-language

PROMELA is a language for modeling concurrent systems

- multi-threaded
- synchronisation and message passing

What is PROMELA?

PROMELA is an acronym

Process meta-language

PROMELA is a language for modeling concurrent systems

- multi-threaded
- synchronisation and message passing
- few control structures, pure (no side-effects) expressions

What is PROMELA?

PROMELA is an acronym

Process meta-language

PROMELA is a language for modeling concurrent systems

- multi-threaded
- synchronisation and message passing
- few control structures, pure (no side-effects) expressions
- data structures with finite and fixed bounds

What is PROMELA **Not**?

PROMELA is not a programming language

Very small language, not intended to program real systems

- No pointers
- No methods/procedures
- No libraries
- No GUI, no standard input
- No floating point types
- Fair scheduling policy (during verification)
- No data encapsulation
- Non-deterministic

Guarded Commands: Selection

```
active proctype P() {
  byte a = 5, b = 5;
  byte max, branch;
  if
    :: a >= b -> max = a; branch = 1
    :: a <= b -> max = b; branch = 2
  fi
}
```

Guarded Commands: Selection

```
active proctype P() {
  byte a = 5, b = 5;
  byte max, branch;
  if
    :: a >= b -> max = a; branch = 1
    :: a <= b -> max = b; branch = 2
  fi
}
```

Observations

- Guards may "overlap" (more than one can be true at the same time)
- Any alternative whose guard is true is randomly selected
- When no guard true: process blocks until one becomes true

Guarded Commands: Repetition

```
active proctype P() { /* computes gcd */
  int a = 15, b = 20;
  do
    :: a > b -> a = a - b
    :: b > a -> b = b - a
    :: a == b -> break
  od
}
```

Guarded Commands: Repetition

```
active proctype P() { /* computes gcd */
  int a = 15, b = 20;
  do
    :: a > b -> a = a - b
    :: b > a -> b = b - a
    :: a == b -> break
  od
}
```

'

Observations

- Any alternative whose guard is true is randomly selected
- Only way to exit loop is via **break** or **goto**
- When no guard true: loop blocks until one becomes true

Sources of Non-Determinism

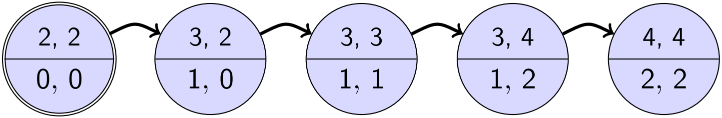
1. Non-deterministic choice of alternatives with overlapping guards
2. Scheduling of concurrent processes

PROMELA Computations

```
active [2] proctype P() {
  byte n;
  n = 1;
  n = 2
}
```

```
1 active [2] proctype P() {
2   byte n;
3   n = 1;
4   n = 2
5 }
```

One possible computation of this program



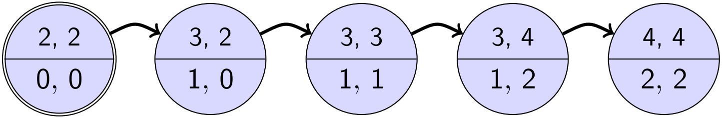
Notation

- Program pointer (line #) for each process in upper compartment
- Value of all variables in lower compartment

PROMELA **Computations**

```
1 active [2] proctype P() {
2   byte n;
3   n = 1;
4   n = 2
5 }
```

One possible computation of this program



Notation

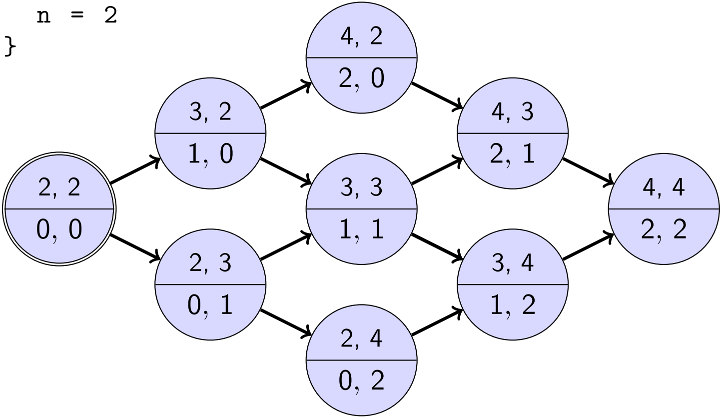
- Program pointer (line #) for each process in upper compartment
- Value of all variables in lower compartment

Computations are either infinite or terminating or blocking

Interleaving

Can represent possible interleavings in a DAG

```
active [2] proctype P() {
  byte n;
  n = 1;
  n = 2
}
```



Usage Scenario of PROMELA

1. Model the essential features of a system in PROMELA
   - abstract away from complex (numerical) computations
     - make usage of non-deterministic choice of outcome
   - replace unbounded data structures with finite approximations
   - assume fair process scheduler
2. Select properties that the PROMELA model must satisfy
   - Generic Properties
     - Mutal exclusion for access to critical resources
     - Absence of deadlock
     - Absence of starvation
   - System-specific properties
     - Event sequences (e.g., system responsiveness)

What Does A Model Checker Do?

Model Checker (MC) is designed to prove the user wrong.

MC does *not* try to prove correctness properties.
It tries the opposite.

MC tuned to find counter example to correctness property.

What Does A Model Checker Do?

Model Checker (MC) is designed to prove the user wrong.

MC does *not* try to prove correctness properties.
It tries the opposite.

MC tuned to find counter example to correctness property.

Why can an MC also prove correctness properties?

What Does A Model Checker Do?

Model Checker (MC) is designed to prove the user wrong.

MC does *not* try to prove correctness properties.
It tries the opposite.

MC tuned to find counter example to correctness property.

Why can an MC also prove correctness properties?

MC's search for counter examples is exhaustive.

What Does A Model Checker Do?

Model Checker (MC) is designed to prove the user wrong.

MC does *not* try to prove correctness properties.
It tries the opposite.

MC tuned to find counter example to correctness property.

Why can an MC also prove correctness properties?

MC's search for counter examples is exhaustive.

$\Rightarrow$ Finding no counter example proves stated correctness properties.

What does 'exhaustive search' mean here?

exhaustive search

=

resolving non-determinism in all possible ways

What does ‘exhaustive search’ mean here?

exhaustive search

=

resolving non-determinism in all possible ways

For model checking PROMELA code,
two kinds of non-determinism to be resolved:

What does ‘exhaustive search’ mean here?

exhaustive search
$=$
resolving non-determinism in all possible ways

For model checking PROMELA code,
two kinds of non-determinism to be resolved:

- explicit, local:
  if/**do** statements

```
:: guardX -> ...
:: guardY -> ...
```

What does ‘exhaustive search’ mean here?

exhaustive search
=
resolving non-determinism in all possible ways

For model checking PROMELA code,
two kinds of non-determinism to be resolved:

- explicit, local:
  if/**do** statements

```
:: guardX -> ...
:: guardY -> ...
```

- implicit, global:
  scheduling of concurrent processes

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of Spin:

- simulating a model (randomly/interactively)
- generating a verifier

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of Spin:

- simulating a model (randomly/interactively)
- generating a verifier

verifier generated by Spin is a C program performing

Model Checker for This Course: SPIN

SPIN: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of SPIN:

- simulating a model (randomly/interactively)
- generating a verifier

verifier generated by SPIN is a C program performing model checking:

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of Spin:

- simulating a model (randomly/interactively)
- generating a verifier

verifier generated by Spin is a C program performing model checking:

- exhaustively checks Promela model against correctness properties

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of Spin:

- simulating a model (randomly/interactively)
- generating a verifier

verifier generated by Spin is a C program performing model checking:

- exhaustively checks Promela model against correctness properties
- in case the check is negative:
  generates a failing run of the model

Model Checker for This Course: Spin

Spin: "Simple Promela Interpreter"

The name is a serious understatement!

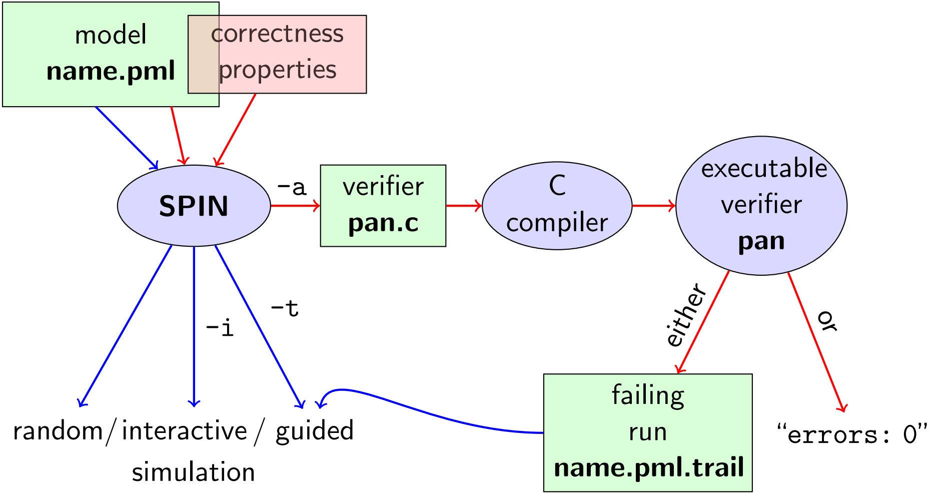
main functionality of Spin:

- simulating a model (randomly/interactively/guided)
- generating a verifier

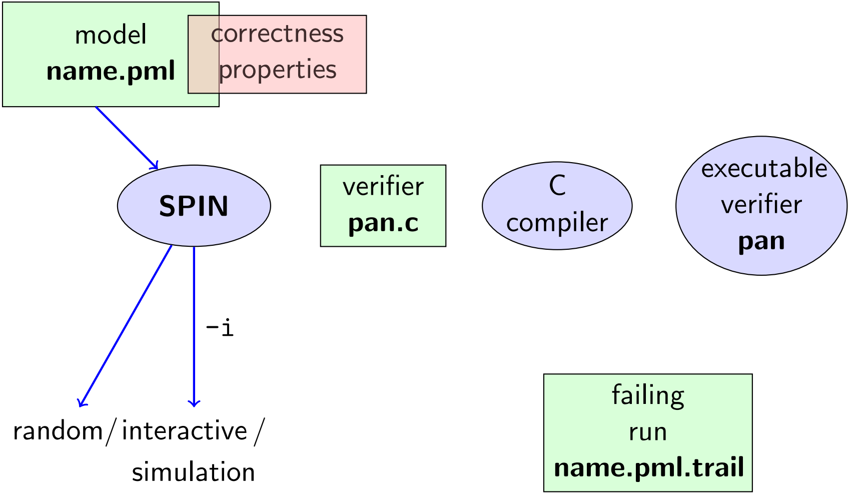
verifier generated by Spin is a C program performing model checking:

- exhaustively checks Promela model against correctness properties
- in case the check is negative:
  generates a failing run of the model, to be simulated by Spin

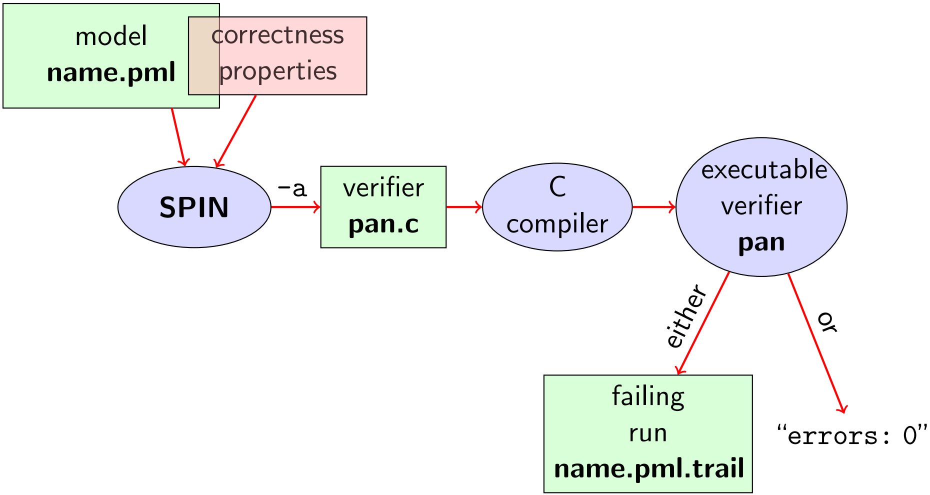
SPIN **Workflow: Overview**



Plain Simulation with SPIN



Model Checking with Spin



Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

- Be R_M the set of all possible runs of M.

Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  ($R_{M,C_i} \subseteq R_M$)

Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \dots, C_n$.

- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  ($R_{M,C_i} \subseteq R_M$)
- M is correct wrt. $C_1, \dots, C_n$ iff

Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  $(R_{M,C_i} \subseteq R_M)$
- M is correct wrt. $C_1, \ldots, C_n$ iff $R_M = (R_{M,C_1} \cap \ldots \cap R_{M,C_n})$.

Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  $(R_{M,C_i} \subseteq R_M)$
- M is correct wrt. $C_1, \ldots, C_n$ iff $R_M = (R_{M,C_1} \cap \ldots \cap R_{M,C_n})$.
- If M is not correct, then
  each $r \in (R_M \setminus (R_{M,C_1} \cap \ldots \cap R_{M,C_n}))$ is a counter example.

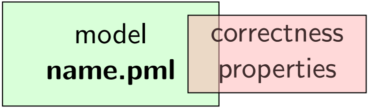
Meaning of Correctness w.r.t. Properties

Given PROMELA model M, and correctness properties $C_1, \ldots, C_n$.

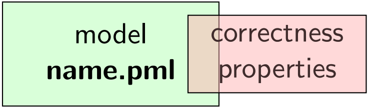
- Be R_M the set of all possible runs of M.
- For each correctness property C_i,
  R_{M,C_i} is the set of all runs of M satisfying C_i.
  ($R_{M,C_i} \subseteq R_M$)
- M is correct wrt. $C_1, \ldots, C_n$ iff $R_M = (R_{M,C_1} \cap \ldots \cap R_{M,C_n})$.
- If M is not correct, then
  each $r \in (R_M \setminus (R_{M,C_1} \cap \ldots \cap R_{M,C_n}))$ is a counter example.

But how to state Correctness Properties?

Stating Correctness Properties

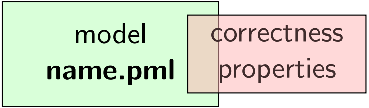


Stating Correctness Properties



Correctness properties can be stated within, or outside, the model.

Stating Correctness Properties

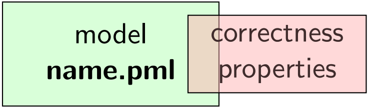


Correctness properties can be stated within, or outside, the model.

stating properties within model using

- assertion statements

Stating Correctness Properties

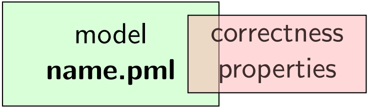


Correctness properties can be stated within, or outside, the model.

stating properties within model using

- assertion statements
- meta labels
  - `end` labels
  - `accept` labels
  - `progress` labels

Stating Correctness Properties



Correctness properties can be stated within, or outside, the model.

stating properties within model using

- assertion statements
- meta labels
  - `end` labels
  - `accept` labels
  - `progress` labels

stating properties outside model using

- never claims
- temporal logic formulas

Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

$$\textbf{assert}(\textit{expr})$$

were *expr* is any PROMELA expression.

Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

$$\mathbf{assert}(\mathit{expr})$$

were *expr* is any PROMELA expression.

Typically, *expr* is of type **bool**.

Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

$$\mathbf{assert}(\mathit{expr})$$

were *expr* is any PROMELA expression.

Typically, *expr* is of type **bool**.

assert(*expr*) can appear wherever a statement is expected.

Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

$$\textbf{assert}(\textit{expr})$$

were *expr* is any PROMELA expression.

Typically, *expr* is of type **bool**.

assert(*expr*) can appear wherever a statement is expected.

```
...
stmt1;
assert(max == a);
stmt2;
...
```

Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

assert(*expr*)

were *expr* is any PROMELA expression.

Typically, *expr* is of type **bool**.

assert(*expr*) can appear wherever a statement is expected.

```
...
stmt1;
assert(max == a);
stmt2;
...
```

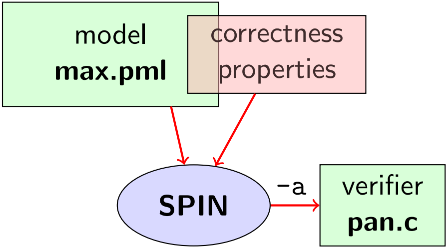
```
...
if
  :: b1 -> stmt3;
           assert(x < y)
  :: b2 -> stmt4
...
```

Employing Assertions

quoting from file **max.pml**:

```
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = b
  :: a <= b -> max = a
fi;

assert( max == (a>b -> a : b) )
```

Generate Verifier in C



Command Line Execution

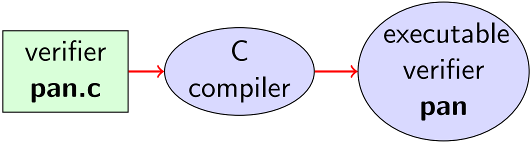
Generate Verifier in C

```
> spin -a max.pml
```

SPIN generates Verifier in C, called **pan.c**

(plus helper files)

Compile To Executable Verifier

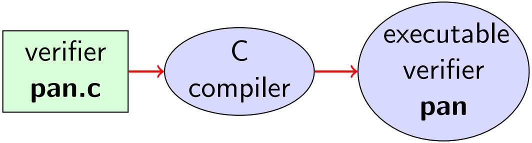


Command Line Execution

compile to executable verifier

```
> gcc -o pan pan.c
```

Compile To Executable Verifier



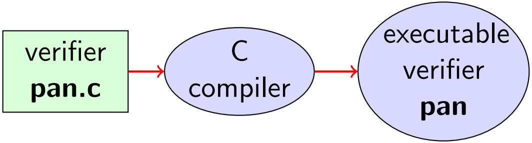
Command Line Execution

compile to executable verifier

```
> gcc -o pan pan.c
```

C compiler generates executable verifier **pan**

Compile To Executable Verifier



Command Line Execution

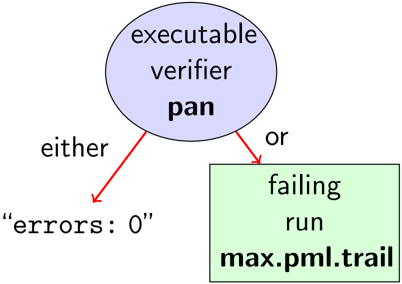
compile to executable verifier

```
> gcc -o pan pan.c
```

C compiler generates executable verifier **pan**

pan: historically "**p**rotocol **an**alyzer", now "**p**rocess **an**alyzer"

Run Verifier (= Model Check)

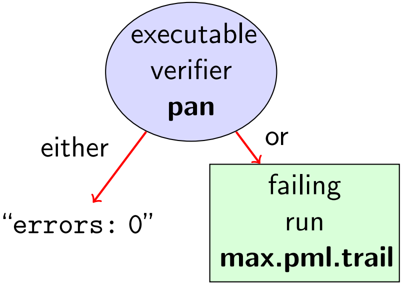


Command Line Execution

run verifier **pan**

> ./pan or > pan

Run Verifier (= Model Check)



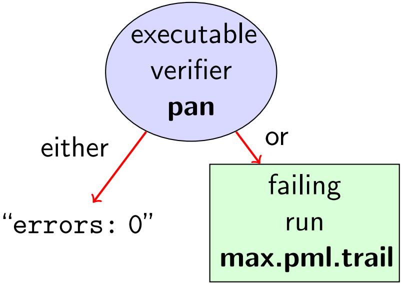
Command Line Execution

run verifier **pan**

> ./pan or *> pan*

- prints “errors: 0”

Run Verifier (= Model Check)



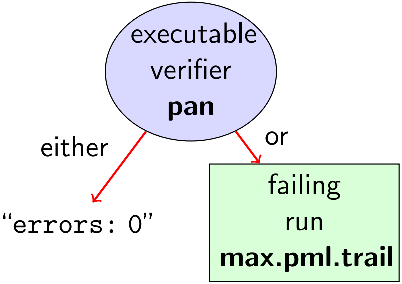
Command Line Execution

run verifier **pan**

`> ./pan`   *or*   `> pan`

- prints “errors: 0” ⇒ Correctness Property verified!

Run Verifier (= Model Check)



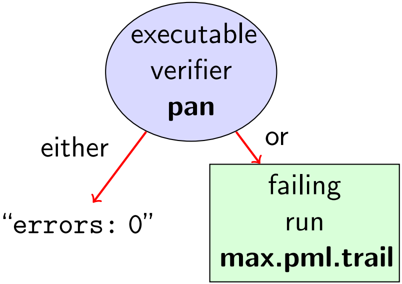
Command Line Execution

run verifier **pan**

`> ./pan` *or* `> pan`

- prints "`errors: 0`", or
- prints "`errors:` n" ($n > 0$)

Run Verifier (= Model Check)



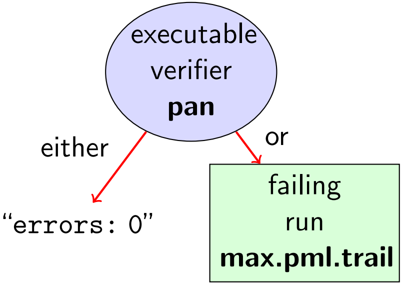
Command Line Execution

run verifier **pan**

> ./pan or > pan

- prints "errors: 0", or
- prints "errors: n" $(n > 0)$ $\Rightarrow$ counter example found!

Run Verifier (= Model Check)



Command Line Execution

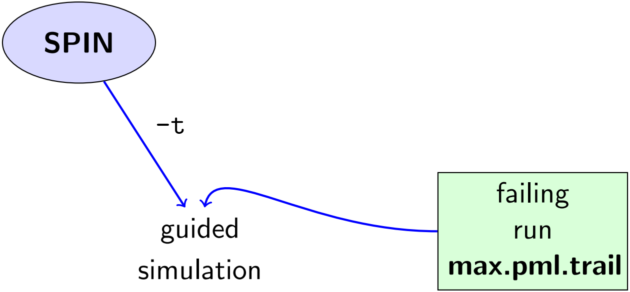
run verifier **pan**

`> ./pan` *or* `> pan`

- prints “`errors: 0`”, or
- prints “`errors:` n” ($n > 0$) ⇒ counter example found!
  records failing run in **max.pml.trail**

Guided Simulation

To examine failing run: employ simulation mode, “guided” by trail file.



Command Line Execution

inject a fault, re-run verification, and then:

```
> spin -t -p -l max.pml
```

Output of Guided Simulation

can look like:

```
Starting P with pid 0
1: proc 0 (P) line  8 "max.pml" (state 1) [a = 1 ]
                P(0):a = 1
2: proc 0 (P) line 14 "max.pml" (state 7) [b = 2 ]
                P(0):b = 2
3: proc 0 (P) line 23 "max.pml" (state 13) [((a<=b))]
3: proc 0 (P) line 23 "max.pml" (state 14) [max = a ]
                P(0):max = 1
spin: line  25 "max.pml", Error: assertion violated
spin: text of failed assertion:
      assert((max==( ((a>b)) -> (a) : (b) )))
```

Output of Guided Simulation

can look like:

```
Starting P with pid 0
1: proc 0 (P) line  8 "max.pml" (state 1) [a = 1 ]
                P(0):a = 1
2: proc 0 (P) line 14 "max.pml" (state 7) [b = 2 ]
                P(0):b = 2
3: proc 0 (P) line 23 "max.pml" (state 13) [((a<=b))]
3: proc 0 (P) line 23 "max.pml" (state 14) [max = a ]
                P(0):max = 1
spin: line  25 "max.pml", Error: assertion violated
spin: text of failed assertion:
      assert((max==( ((a>b)) -> (a) : (b) )))
```

assignments in the run

Output of Guided Simulation

can look like:

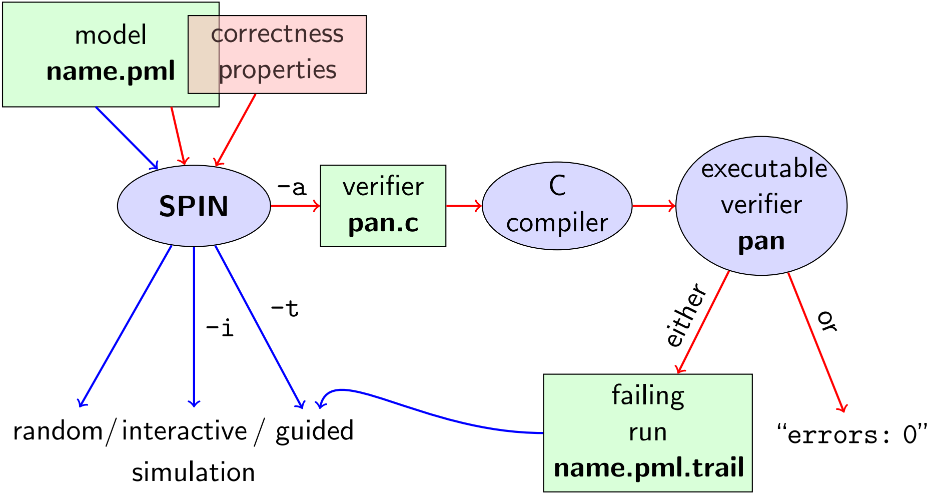
```
Starting P with pid 0
1: proc 0 (P) line  8 "max.pml" (state 1) [a = 1 ]
                P(0):a = 1
2: proc 0 (P) line 14 "max.pml" (state 7) [b = 2 ]
                P(0):b = 2
3: proc 0 (P) line 23 "max.pml" (state 13) [((a<=b))]
3: proc 0 (P) line 23 "max.pml" (state 14) [max = a ]
                P(0):max = 1
spin: line  25 "max.pml", Error: assertion violated
spin: text of failed assertion:
      assert((max==( ((a>b)) -> (a) : (b) )))
```

assignments in the run

values of variables whenever updated

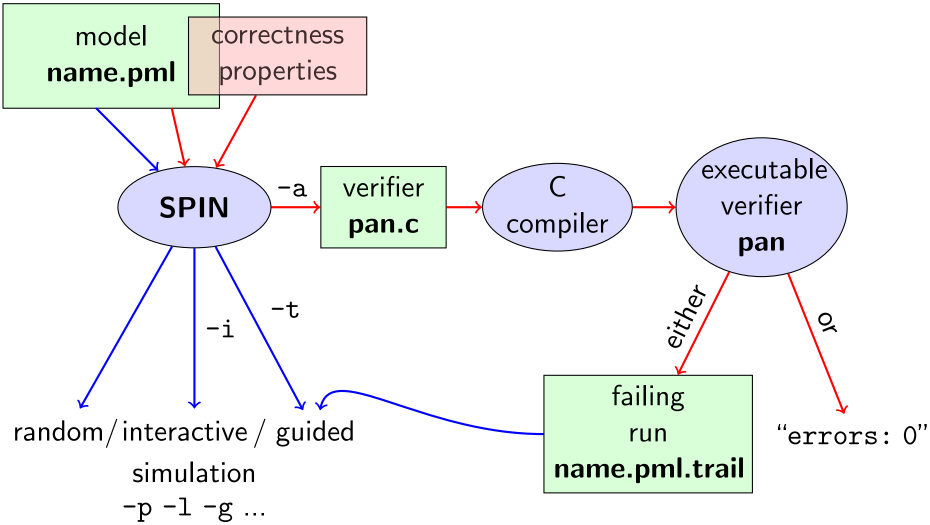
What did we do so far?

following whole cycle (most primitive example, assertions only)



What did we do so far?

following whole cycle (most primitive example, assertions only)



Local and Global Data

Variables declared outside of the processes are global to all processes.

Variables declared inside a process are local to that processes.

```
byte n;

proctype P(byte id; byte incr) {
  byte t;
  ...
}
```

`n` is global

`t` is local

Modeling with Global Data

pragmatics of modeling with global data:

shared memory of concurrent systems often modeled
by global variables of numeric (or array) type

status of shared resources (printer, traffic light, ...) often modeled
by global variables of Boolean or enumeration type
(**bool**/**mtype**).

communication mediums of distributed systems often modeled
by global variables of channel type (**chan**).

Interference on Global Data

```
byte n = 0;

active proctype P() {
  n = 1;
  printf("Process␣P,␣n␣=␣%d\n", n)
}
```

Interference on Global Data

```
byte n = 0;

active proctype P() {
  n = 1;
  printf("Process␣P,␣n␣=␣%d\n", n)
}

active proctype Q() {
  n = 2;
  printf("Process␣Q,␣n␣=␣%d\n", n)
}
```

Interference on Global Data

```
byte n = 0;

active proctype P() {
  n = 1;
  printf("Process␣P,␣n␣=␣%d\n", n)
}

active proctype Q() {
  n = 2;
  printf("Process␣Q,␣n␣=␣%d\n", n)
}
```

how many outputs possible?

Interference on Global Data

```
byte n = 0;

active proctype P() {
  n = 1;
  printf("Process␣P,␣n␣=␣%d\n", n)
}

active proctype Q() {
  n = 2;
  printf("Process␣Q,␣n␣=␣%d\n", n)
}
```

how many outputs possible?

different processes can interfere on global data

Examples

1. `interleave0.pml`
   SPIN simulation, SPINSPIDER automata + transition system
2. `interleave1.pml`
   SPIN simulation, adding assertion, fine-grained execution model, model checking
3. `interleave5.pml`
   SPIN simulation, SPIN model checking, trail inspection

Show Mutual Exclusion

```
int critical = 0;

active proctype P() {
  do :: printf("P non-critical actions\n");
        P_in_CS = true;
        !Q_in_CS;
        /* begin critical section */
        critical++;
        printf("P uses shared recourses\n");
        assert(critical < 2);
        critical--;
        /* end critical section */
        P_in_CS = false
  od
}

active proctype Q() {
  ...correspondingly...
}
```

Verify Mutual Exclusion of this

SPIN (`./pan -E`) shows no assertion is violated
$\Rightarrow$ mutual exclusion is verified

Verify Mutual Exclusion of this

SPIN (`./pan -E`) shows no assertion is violated
⇒ mutual exclusion is verified

still SPIN (without `-E`) reports (`invalid end state`)
⇒ deadlock

Deadlock Hunting

Invalid End State:

- A process does not finish at its end
- Two or more inter-dependent processes do not finish at the end
  Real deadlock

Deadlock Hunting

Invalid End State:

- A process does not finish at its end
- Two or more inter-dependent processes do not finish at the end
  Real deadlock

Find Deadlock with SPIN:

- Verify to produce a failing run trail
- Simulate to see how the processes get to the interlock
- Fix the code

Atomicity against Deadlocks

solution:

checking and setting the flag in one atomic step

Atomicity against Deadlocks

solution:

checking and setting the flag in one atomic step

```
atomic {
  !Q_in_CS;
  P_in_CS = true
}
```

Channels in PROMELA

chan *name* = [*capacity*] **of** {$type_1, ..., type_n$}

Creates a channel, which is stored in *name*

Channels in Promela

chan *name* = [*capacity*] **of** {$type_1, ..., type_n$}

Creates a channel, which is stored in *name*

Messages communicated via the channel are *n*-tuples $\in type_1 \times \ldots \times type_n$

Channels in PROMELA

chan *name* = [*capacity*] **of** $\{type_1, ..., type_n\}$

Creates a channel, which is stored in *name*

Messages communicated via the channel are n-tuples $\in type_1 \times \ldots \times type_n$

Can buffer up to *capacity* messages, if $capacity \geq 1$
$\Rightarrow$ *"buffered channel"*

Channels in Promela

chan *name* = [*capacity*] **of** {$type_1, ..., type_n$}

Creates a channel, which is stored in *name*

Messages communicated via the channel are *n*-tuples $\in type_1 \times \ldots \times type_n$

Can buffer up to *capacity* messages, if $capacity \geq 1$
$\Rightarrow$ *"buffered channel"*

The channel has *no* buffer, if $capacity = 0$
$\Rightarrow$ *"rendezvous channel"*

Channels in PROMELA cont'd

Example:

```
chan ch = [2] of { mtype, byte, bool }
```

Creates a channel, which is stored in `ch`

Channels in PROMELA cont'd

Example:

chan `ch` = [2] **of** { **mtype**, **byte**, **bool** }

Creates a channel, which is stored in `ch`

Messages communicated via `ch` are 3-tuples $\in$ **mtype** $\times$ **byte** $\times$ **bool**

Channels in PROMELA cont'd

Example:

chan `ch = [2]` **of** { **mtype**, **byte**, **bool** }

Creates a channel, which is stored in `ch`

Messages communicated via `ch` are 3-tuples $\in$ **mtype** $\times$ **byte** $\times$ **bool**

Given, e.g., **mtype** `{red, yellow, green}`,
an example message can be:

Channels in Promela cont'd

Example:

chan `ch = [2]` **of** { **mtype**, **byte**, **bool** }

Creates a channel, which is stored in `ch`

Messages communicated via `ch` are 3-tuples $\in$ **mtype** $\times$ **byte** $\times$ **bool**

Given, e.g., **mtype** `{red, yellow, green}`,
an example message can be: `green, 20,` **false**

Channels in PROMELA cont'd

Example:

```
chan ch = [2] of { mtype, byte, bool }
```

Creates a channel, which is stored in `ch`

Messages communicated via `ch` are 3-tuples $\in$ **mtype** $\times$ **byte** $\times$ **bool**

Given, e.g., **mtype** `{red, yellow, green}`,
an example message can be: `green, 20,` **false**

`ch` is a *buffered channel*, buffering up to 2 messages

Sending and Receiving

send statement has the form:

name ! $expr_1$, ... , $expr_n$

- *name*: channel variable
- $expr_1$, ... , $expr_n$: sequence of expressions, where number and types match message type
- sends *values* of $expr_1$, ... , $expr_n$ as *one* message
- example: `ch ! green, 20,` **false**

receive statement has the form:

name ? var_1, ... , var_n

- *name*: channel variable
- var_1, ... , var_n: sequence of variables, where number and types match message type
- *assigns* values of message to var_1, ... , var_n
- example: `ch ? color, time, flash`

Rendezvous Channels

```
chan ch = [0] of { byte, byte };

/* global to make visible in SpinSpider */
byte hour, minute;

active proctype Sender() {
  printf("ready\n");
  ch ! 11, 45;
  printf("Sent\n")
}

active proctype Receiver() {
  printf("steady\n");
  ch ? hour, minute;
  printf("Received\n")
}
```

Rendezvous Channels

```
chan ch = [0] of { byte, byte };

/* global to make visible in SpinSpider */
byte hour, minute;

active proctype Sender() {
  printf("ready\n");
  ch ! 11, 45;
  printf("Sent\n")
}

active proctype Receiver() {
  printf("steady\n");
  ch ? hour, minute;
  printf("Received\n")
}
```

Which interleavings can occur?

Rendezvous Channels

```
chan ch = [0] of { byte, byte };

/* global to make visible in SpinSpider */
byte hour, minute;

active proctype Sender() {
  printf("ready\n");
  ch ! 11, 45;
  printf("Sent\n")
}

active proctype Receiver() {
  printf("steady\n");
  ch ? hour, minute;
  printf("Received\n")
}
```

Which interleavings can occur? $\Rightarrow$ ask SpinSpider

Demo

through jSpin:
SpinSpider on `ReadySteady.pml`

Rendezvous are Synchronous

On a rendezvous channel:

transfer of message from sender to receiver is synchronous,
i.e., one single operation

Rendezvous are Synchronous

On a rendezvous channel:

transfer of message from sender to receiver is synchronous,
i.e., one single operation

Sender		Receiver
$\vdots$		$\vdots$
`(11,45)`	$\longrightarrow$	`(hour,minute)`
$\vdots$		$\vdots$

Reply Channels - Single Server

```
chan request = [0] of { mtype };
chan reply = [0] of { mtype };
mtype = { nice, rude };

active proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
  od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;

}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Reply Channels - Single Server

```
chan request = [0] of { mtype };
chan reply = [0] of { mtype };
mtype = { nice, rude };

active proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
  od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)
}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Reply Channels - Single Server

```
chan request = [0] of { mtype };
chan reply = [0] of { mtype };
mtype = { nice, rude };

active proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
  od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)
}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Is the assertion valid?

Reply Channels - Single Server

```
chan request = [0] of { mtype };
chan reply = [0] of { mtype };
mtype = { nice, rude };

active proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
  od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)
}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Is the assertion valid? Ask SPIN.

Several Servers

More realistic with several servers:

```
active [2] proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
  od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;

}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Several Servers

More realistic with several servers:

```
active [2] proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
  od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)
}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Several Servers

More realistic with several servers:

```
active [2] proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
  od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)                  And here?
}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Several Servers

More realistic with several servers:

```
active [2] proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
  od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)
}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

And here? Analyse with SPIN.

Sending Channels via Channels

One way to fix the protocol:

clients declare local reply channel + send it to server

Sending Channels via Channels

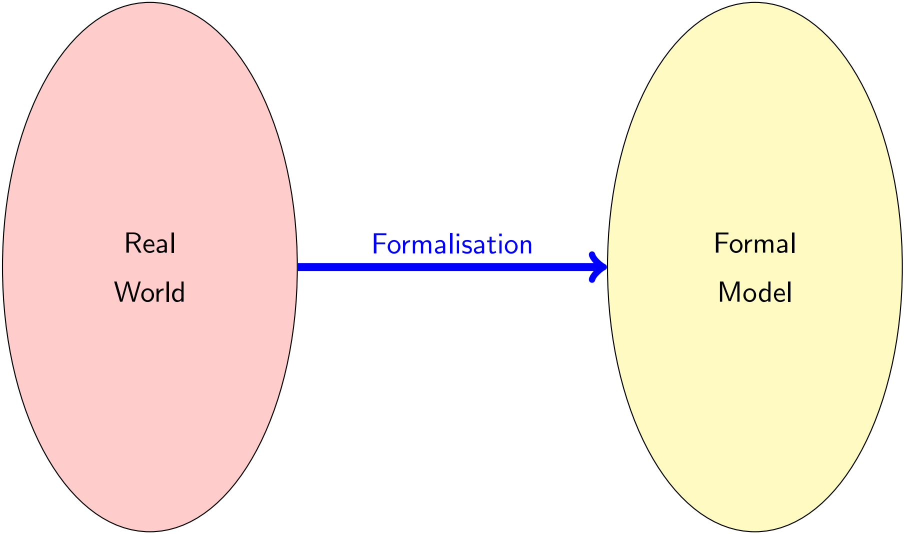
```
mtype = { nice, rude };
chan request = [0] of { mtype, chan };

active [2] proctype Server() {
  mtype msg; chan ch;
  do :: request ? msg, ch;
        ch ! msg
  od
}
active proctype NiceClient() {
  chan reply = [0] of { mtype };  mtype msg;
  request ! nice, reply;   reply ? msg;
  assert( msg == nice )
}
active proctype RudeClient() {
  chan reply = [0] of { mtype };  mtype msg;
  request ! rude, reply;   reply ? msg
}
```

Sending Channels via Channels

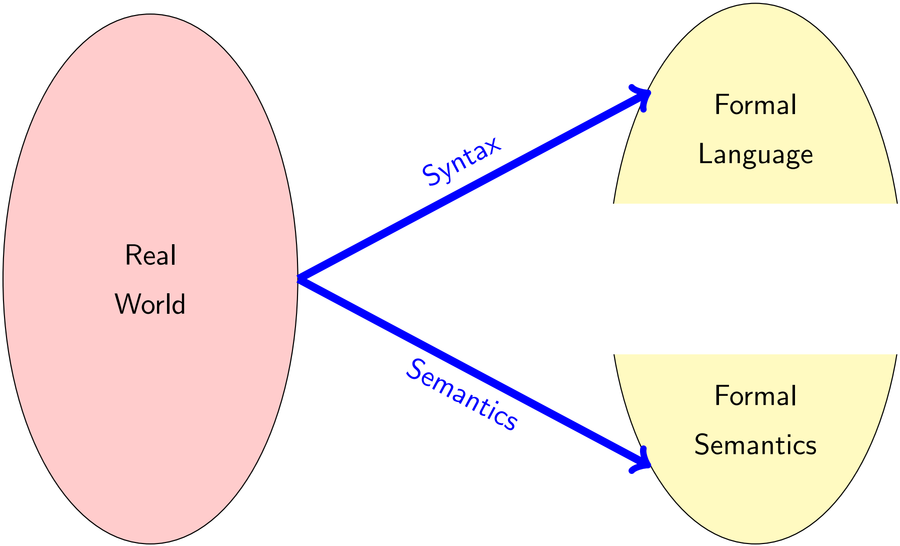
```
mtype = { nice, rude };
chan request = [0] of { mtype, chan };

active [2] proctype Server() {
  mtype msg; chan ch;
  do :: request ? msg, ch;
        ch ! msg
  od
}
active proctype NiceClient() {
  chan reply = [0] of { mtype };  mtype msg;
  request ! nice, reply;   reply ? msg;
  assert( msg == nice )
}
active proctype RudeClient() {
  chan reply = [0] of { mtype };  mtype msg;
  request ! rude, reply;   reply ? msg
}
```

verify with SPIN

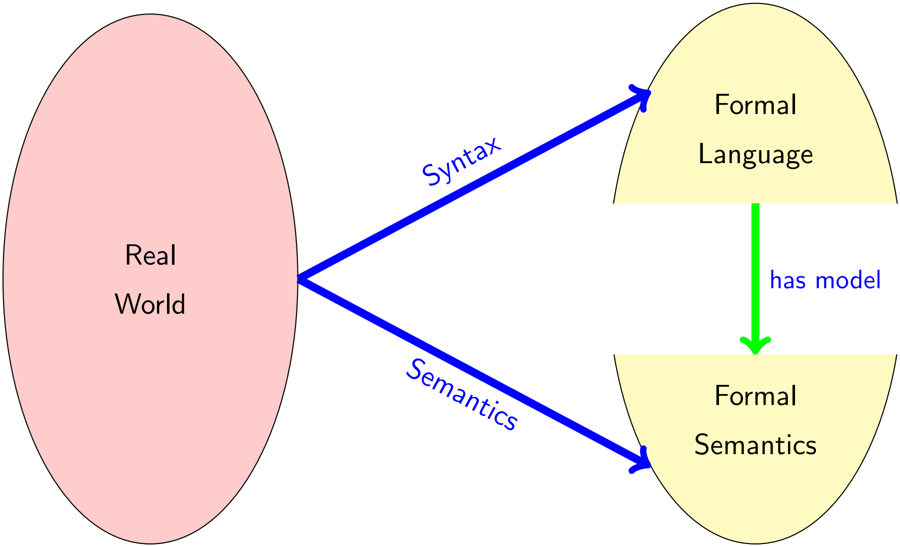
Recapitulation: Formalisation



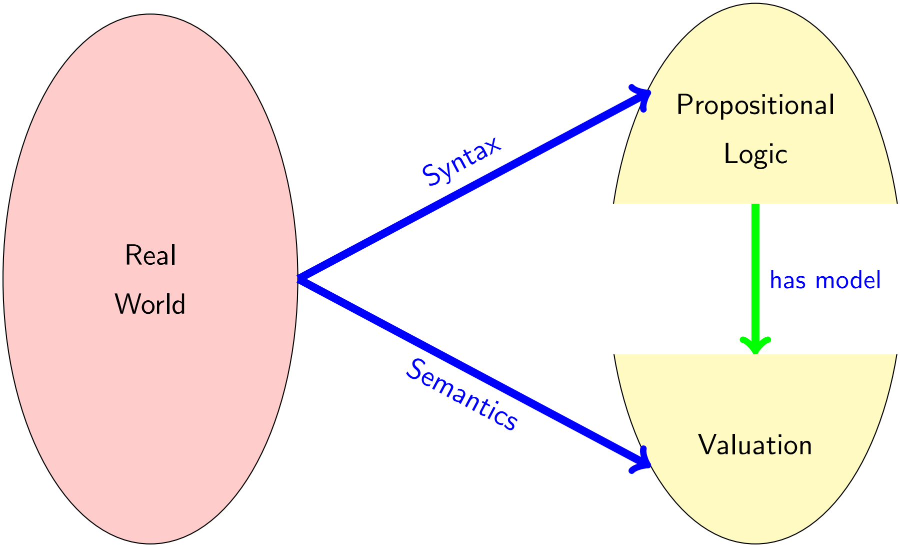
Formalisation: Syntax, Semantics



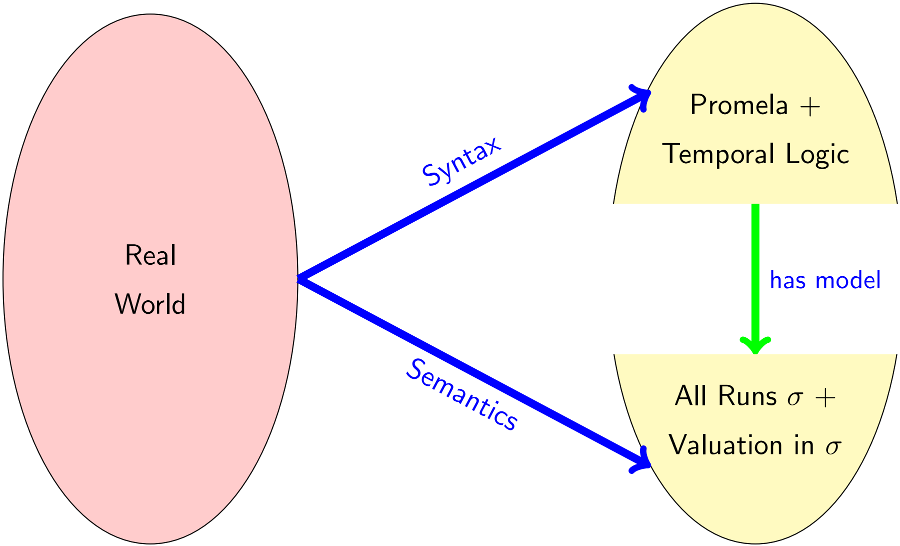
Formalisation: Syntax, Semantics



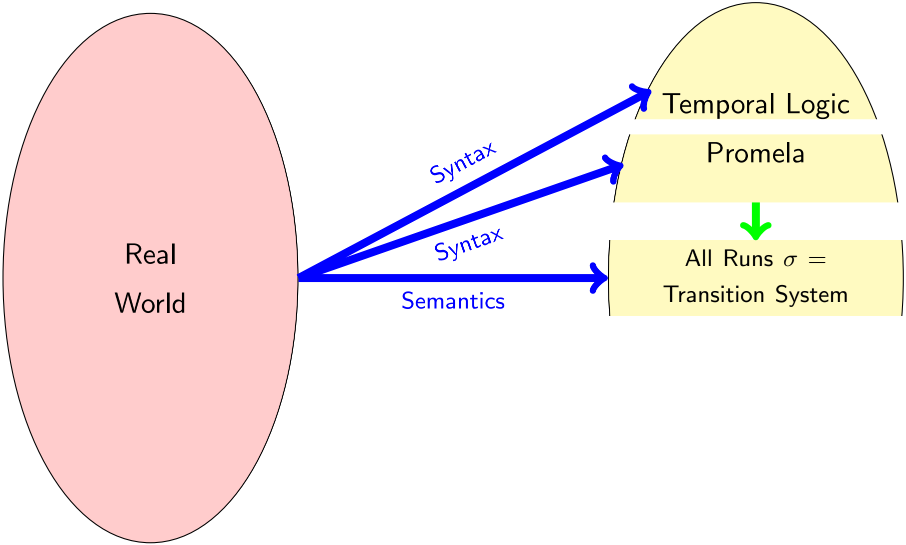
Formalisation: Syntax, Semantics



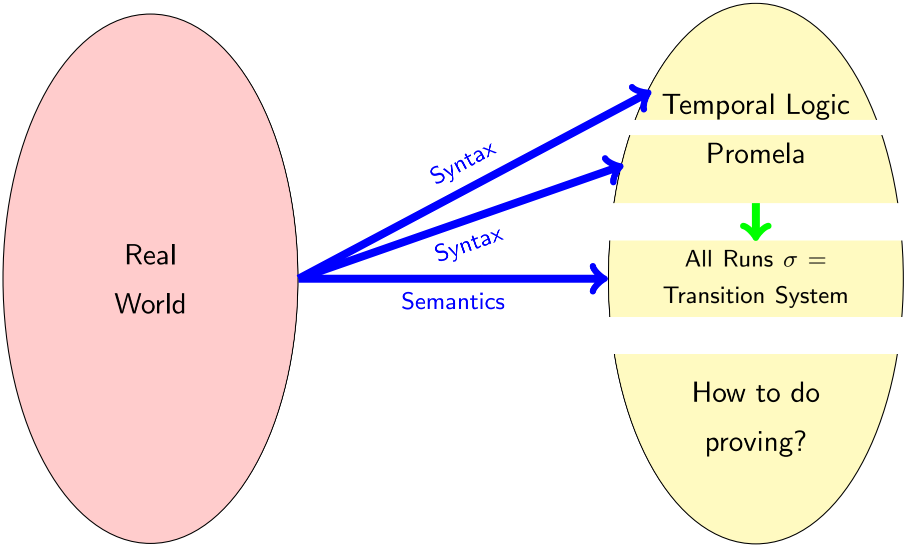
Formalisation: Syntax, Semantics



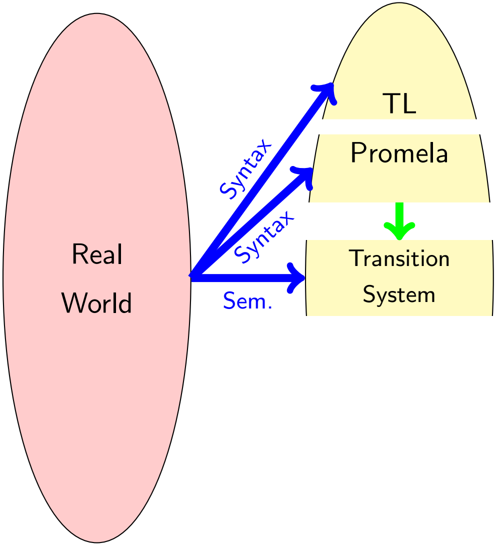
Formalisation: Syntax, Semantics



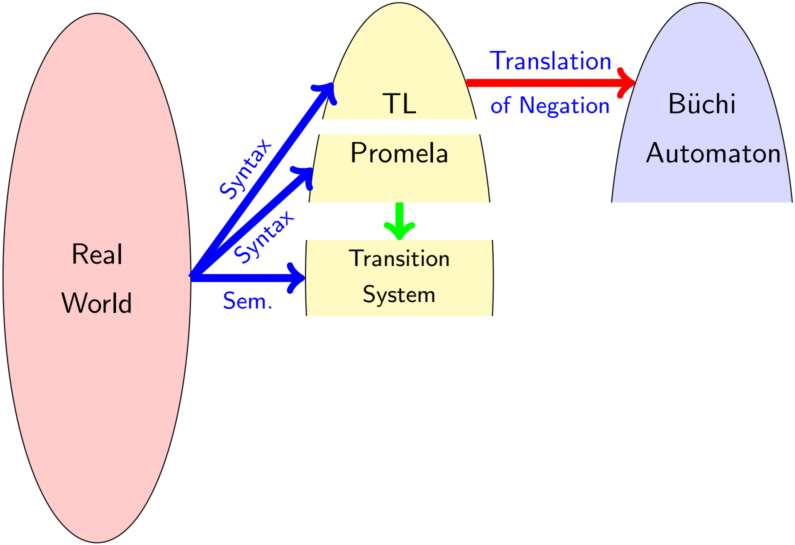
Formalisation: Syntax, Semantics, Proving



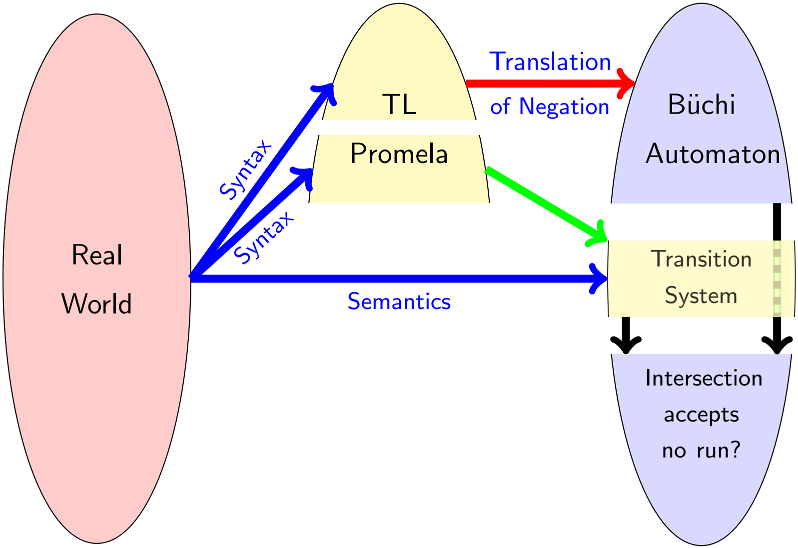
Formal Verification: Model Checking



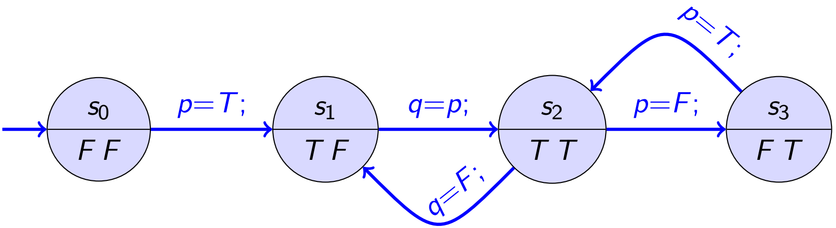
Formal Verification: Model Checking



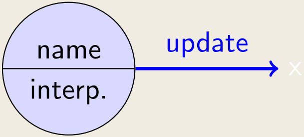
Formal Verification: Model Checking



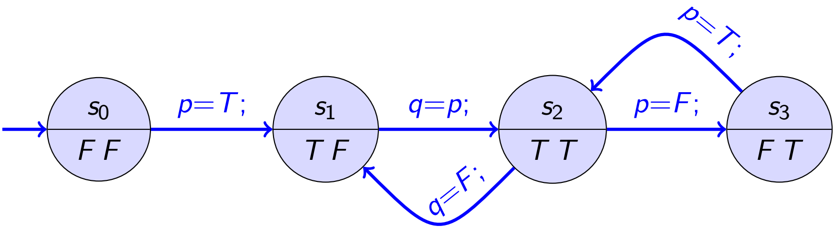
Transition systems (aka Kripke Structures)



Notation

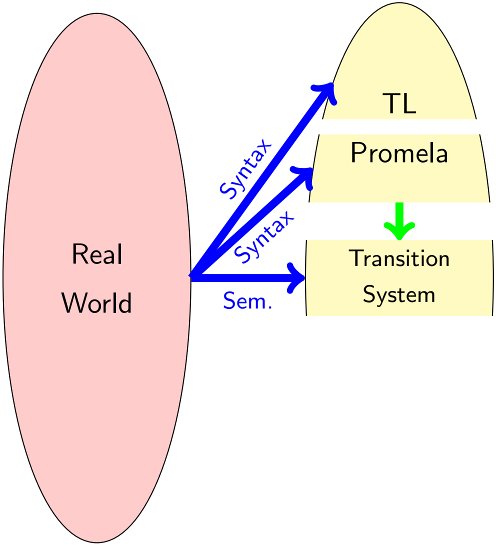


Transition systems (aka Kripke Structures)



- Each state s_i has its own propositional interpretation I_i
  - Convention: list values of variables in ascending lexicographic order
- Computations, or runs, are *infinite* paths through states
  - Intuitively 'finite' runs modelled by looping on final states
- In general, infinitely many different runs possible
- How to express (for example) that p changes its value infinitely often in each run?

Formal Verification: Model Checking



(Linear) Temporal Logic

An extension of propositional logic that
allows to specify properties of all runs

(Linear) Temporal Logic—Syntax

An extension of propositional logic that
allows to specify properties of all runs

Syntax

Based on propositional signature and syntax

Extension with three connectives:

Always If ϕ is a formula then so is $\Box\phi$

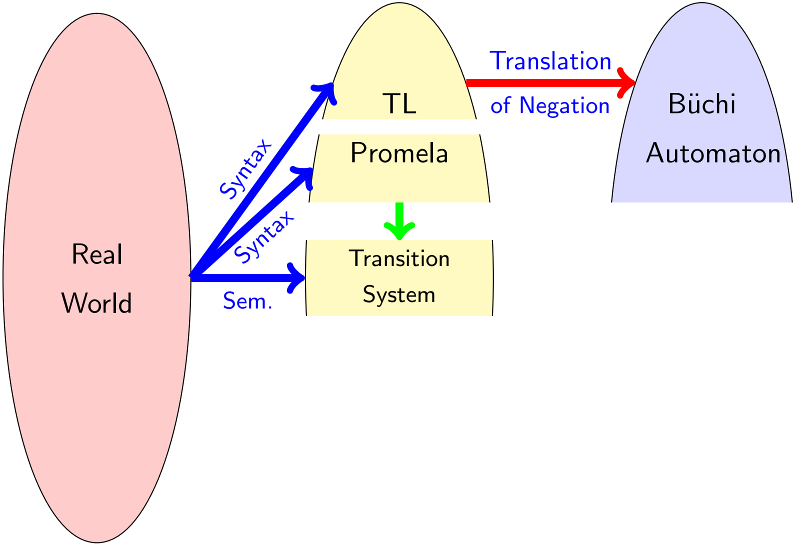
Eventually If ϕ is a formula then so is $\Diamond\phi$

Until If ϕ and ψ are formulas then so is $\phi\mathcal{U}\psi$

Concrete Syntax

	text book	SPIN
Always	$\Box$	[]
Eventually	$\Diamond$	<>
Until	$\mathcal{U}$	U

Formal Verification: Model Checking



ω-**Languages**

Given a finite alphabet (vocabulary) Σ

A word $w \in \Sigma^*$ is a finite sequence

$$w = a_o \cdots a_n$$

with $a_i \in \Sigma, i \in \{0, \ldots, n\}$

$\mathcal{L} \subseteq \Sigma^*$ is called a language

ω-Languages

Given a finite alphabet (vocabulary) Σ

An ω-word $w \in \Sigma^\omega$ is an infinite sequence

$$w = a_o \cdots a_k \cdots$$

with $a_i \in \Sigma, i \in \mathbb{N}$

$\mathcal{L}^\omega \subseteq \Sigma^\omega$ is called an ω-language

Büchi Automaton

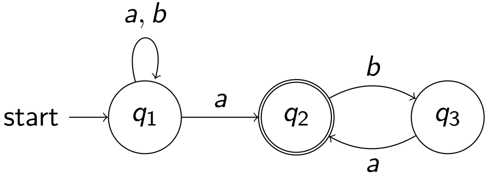
Definition (Büchi Automaton)

A (non-deterministic) Büchi automaton over an alphabet Σ consists of a

- finite, non-empty set of locations Q
- a non-empty set of initial/start locations $I \subseteq Q$
- a set of accepting locations $F = \{F_1, \ldots, F_n\} \subseteq Q$
- a transition relation $\delta \subseteq Q \times \Sigma \times Q$

Example

$\Sigma = \{a, b\}, Q = \{q_1, q_2, q_3\}, I = \{q_1\}, F = \{q_2\}$



Büchi Automaton—Executions and Accepted Words

Definition (Execution)

Let $\mathcal{B} = (Q, I, F, \delta)$ be a Büchi automaton over alphabet Σ.
An execution of $\mathcal{B}$ is a pair (w, v), with

- $w = a_o \cdots a_k \cdots \in \Sigma^\omega$
- $v = q_o \cdots q_k \cdots \in Q^\omega$

where $q_0 \in I$, and $(q_i, a_i, q_{i+1}) \in \delta$, for all $i \in \mathbb{N}$

Büchi Automaton—Executions and Accepted Words

Definition (Execution)

Let $\mathcal{B} = (Q, I, F, \delta)$ be a Büchi automaton over alphabet Σ.
An execution of $\mathcal{B}$ is a pair (w, v), with

- $w = a_o \cdots a_k \cdots \in \Sigma^\omega$
- $v = q_o \cdots q_k \cdots \in Q^\omega$

where $q_0 \in I$, and $(q_i, a_i, q_{i+1}) \in \delta$, for all $i \in \mathbb{N}$

Definition (Accepted Word)

A Büchi automaton $\mathcal{B}$ accepts a word $w \in \Sigma^\omega$, if there exists an execution (w, v) of $\mathcal{B}$ where some accepting location $f \in F$ appears infinitely often in v

Büchi Automaton—Language

Let $\mathcal{B} = (Q, I, F, \delta)$ be a Büchi automaton, then

$$\mathcal{L}^\omega(\mathcal{B}) = \{w \in \Sigma^\omega | w \in \Sigma^\omega \text{ is an accepted word of } \mathcal{B}\}$$

denotes the ω-language recognised by $\mathcal{B}$.

Büchi Automaton—Language

Let $\mathcal{B} = (Q, I, F, \delta)$ be a Büchi automaton, then

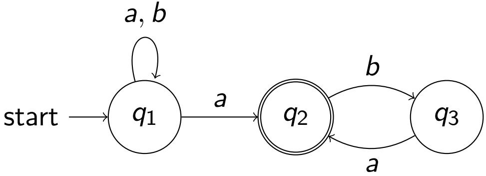
$$\mathcal{L}^\omega(\mathcal{B}) = \{w \in \Sigma^\omega | w \in \Sigma^\omega \text{ is an accepted word of } \mathcal{B}\}$$

denotes the ω-language recognised by $\mathcal{B}$.

An ω-language for which an accepting Büchi automaton exists is called ω-regular language.

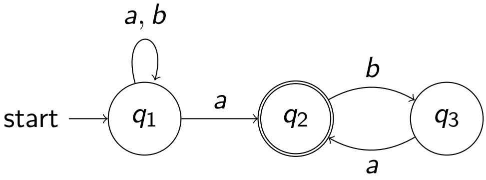
Example, ω-Regular Expression

Which language is accepted by the following Büchi automaton?



Example, ω-Regular Expression

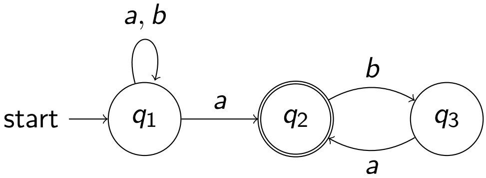
Which language is accepted by the following Büchi automaton?



Solution: $(a+b)^*(ab)^\omega$ [NB: $(ab)^\omega = a(ba)^\omega$]

Example, ω-Regular Expression

Which language is accepted by the following Büchi automaton?



Solution: $(a+b)^*(ab)^\omega$ [NB: $(ab)^\omega = a(ba)^\omega$]

ω-regular expressions like standard regular expression

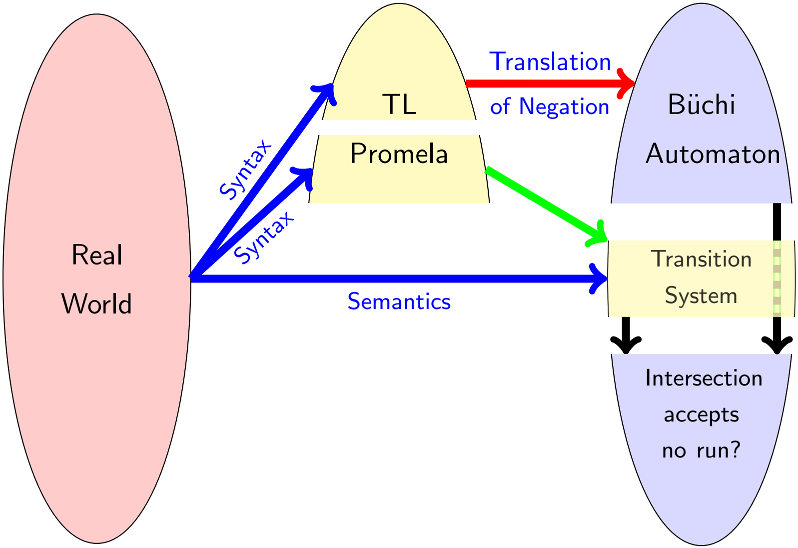
ab a **then** b

$a+b$ a **or** b

a^* arbitrarily, but finitely often a

new: a^ω infinitely often a

Formal Verification: Model Checking



Model Checking

Check whether a formula is valid in all runs of a transition system

Given a transition system $\mathcal{T}$ (e.g., derived from a PROMELA program)

Verification task: is the LTL formula ϕ satisfied in all runs of $\mathcal{T}$, i.e.,

$$\mathcal{T} \models \phi \quad ?$$

Spin Model Checking—Overview

$$\mathcal{T} \models \phi \quad ?$$

1. Represent transition system $\mathcal{T}$ as Büchi automaton $\mathcal{B}_{\mathcal{T}}$ such that $\mathcal{B}_{\mathcal{T}}$ accepts exactly those words corresponding to runs through $\mathcal{T}$

SPIN Model Checking—Overview

$$\mathcal{T} \models \phi \quad ?$$

1. Represent transition system $\mathcal{T}$ as Büchi automaton $\mathcal{B}_{\mathcal{T}}$ such that $\mathcal{B}_{\mathcal{T}}$ accepts exactly those words corresponding to runs through $\mathcal{T}$
2. Construct Büchi automaton $\mathcal{B}_{\neg\phi}$ for negation of formula ϕ

Spin Model Checking—Overview

$$\mathcal{T} \models \phi \quad ?$$

1. Represent transition system $\mathcal{T}$ as Büchi automaton $\mathcal{B}_{\mathcal{T}}$ such that $\mathcal{B}_{\mathcal{T}}$ accepts exactly those words corresponding to runs through $\mathcal{T}$
2. Construct Büchi automaton $\mathcal{B}_{\neg\phi}$ for negation of formula ϕ
3. If

$$\mathcal{L}^{\omega}(\mathcal{B}_{\mathcal{T}}) \cap \mathcal{L}^{\omega}(\mathcal{B}_{\neg\phi}) = \emptyset$$

then ϕ holds.

SPIN Model Checking—Overview

$$\mathcal{T} \models \phi \quad ?$$

1. Represent transition system $\mathcal{T}$ as Büchi automaton $\mathcal{B}_{\mathcal{T}}$ such that $\mathcal{B}_{\mathcal{T}}$ accepts exactly those words corresponding to runs through $\mathcal{T}$
2. Construct Büchi automaton $\mathcal{B}_{\neg\phi}$ for negation of formula ϕ
3. If
$$\mathcal{L}^{\omega}(\mathcal{B}_{\mathcal{T}}) \cap \mathcal{L}^{\omega}(\mathcal{B}_{\neg\phi}) = \emptyset$$
then ϕ holds.

   If
$$\mathcal{L}^{\omega}(\mathcal{B}_{\mathcal{T}}) \cap \mathcal{L}^{\omega}(\mathcal{B}_{\neg\phi}) \neq \emptyset$$
then each element of the set is a counterexample for ϕ.

SPIN Model Checking—Overview

$$\mathcal{T} \models \phi \quad ?$$

1. Represent transition system $\mathcal{T}$ as Büchi automaton $\mathcal{B}_{\mathcal{T}}$ such that $\mathcal{B}_{\mathcal{T}}$ accepts exactly those words corresponding to runs through $\mathcal{T}$
2. Construct Büchi automaton $\mathcal{B}_{\neg\phi}$ for negation of formula ϕ
3. If
$$\mathcal{L}^{\omega}(\mathcal{B}_{\mathcal{T}}) \cap \mathcal{L}^{\omega}(\mathcal{B}_{\neg\phi}) = \emptyset$$
then ϕ holds.

   If
$$\mathcal{L}^{\omega}(\mathcal{B}_{\mathcal{T}}) \cap \mathcal{L}^{\omega}(\mathcal{B}_{\neg\phi}) \neq \emptyset$$
then each element of the set is a counterexample for ϕ.

   To check $\mathcal{L}^{\omega}(\mathcal{B}_{\mathcal{T}}) \cap \mathcal{L}^{\omega}(\mathcal{B}_{\neg\phi})$ construct intersection automaton and search for cycle through accepting state

Representing a Model as a Büchi Automaton

First Step: Represent transition system $\mathcal{T}$ as Büchi automaton $\mathcal{B}_{\mathcal{T}}$ accepting exactly those words representing a run of $\mathcal{T}$

Example

```
active proctype P () {
do
  :: atomic {
      !wQ; wP = true
     };
     Pcs = true;
     atomic {
      Pcs = false;
      wP = false
     }
od }
```

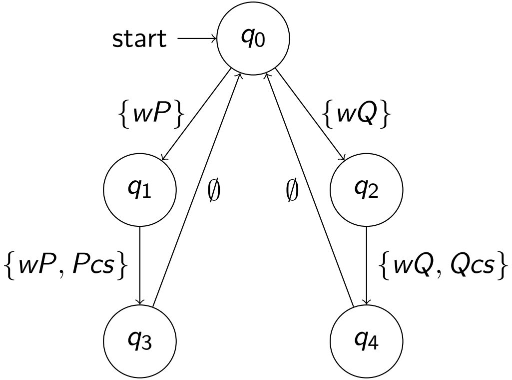
First location skipped and second made **atomic** just to keep automaton small; similar code for process `Q`

Representing a Model as a Büchi Automaton

First Step: Represent transition system $\mathcal{T}$ as Büchi automaton $\mathcal{B}_{\mathcal{T}}$ accepting exactly those words representing a run of $\mathcal{T}$

Example

```
active proctype P () {
do
  :: atomic {
      !wQ; wP = true
     };
     Pcs = true;
     atomic {
      Pcs = false;
      wP = false
     }
od }
```



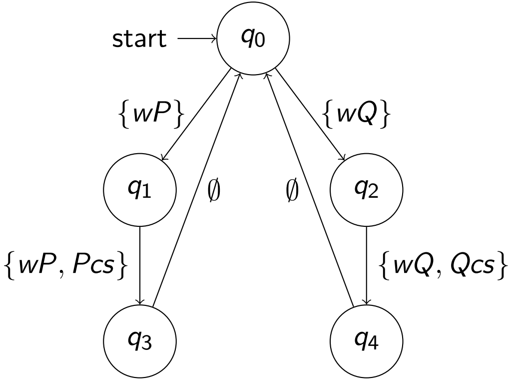
First location skipped and second made **atomic** just to keep automaton small; similar code for process `Q`

Representing a Model as a Büchi Automaton

First Step: Represent transition system $\mathcal{T}$ as Büchi automaton $\mathcal{B}_{\mathcal{T}}$ accepting exactly those words representing a run of $\mathcal{T}$

Example

```
active proctype P () {
do
  :: atomic {
      !wQ; wP = true
     };
     Pcs = true;
     atomic {
      Pcs = false;
      wP = false
     }
od }
```



The property we want to check is $\phi = \Box\neg Pcs$ (which does not hold)

Büchi Automaton $B_{\neg\phi}$ for $\neg\phi$

Second Step:

Construct Büchi Automaton corresponding to negated LTL formula

$\mathcal{T} \models \phi$ holds iff there is no accepting run of $\mathcal{T}$ for $\neg\phi$

Simplify $\neg\phi = \neg\Box\neg Pcs = \Diamond Pcs$

Büchi Automaton $B_{\neg\phi}$ for $\neg\phi$

Second Step:

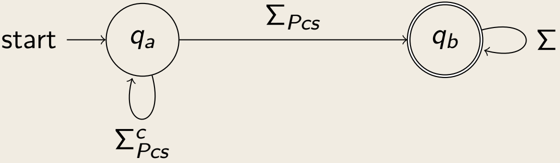
Construct Büchi Automaton corresponding to negated LTL formula

$\mathcal{T} \models \phi$ holds iff there is no accepting run of $\mathcal{T}$ for $\neg\phi$

Simplify $\neg\phi = \neg\Box\neg Pcs = \Diamond Pcs$

Büchi Automaton $\mathcal{B}_{\neg\phi}$

$$\mathcal{P} = \{wP, wQ, Pcs, Qcs\},\ \Sigma = 2^{\mathcal{P}}$$



$$\Sigma_{Pcs} = \{I | I \in \Sigma, Pcs \in I\}, \quad \Sigma^c_{Pcs} = \Sigma - \Sigma_{Pcs}$$

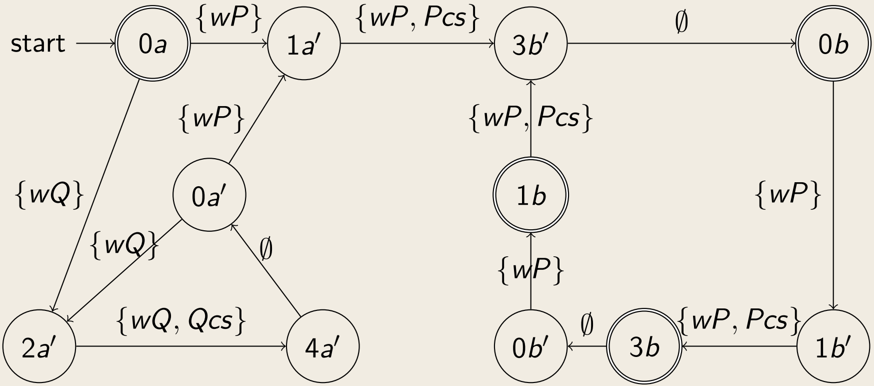
Checking for Emptiness of Intersection Automaton

Third Step: $\mathcal{L}^\omega(\mathcal{B}_\mathcal{T}) \cap \mathcal{L}^\omega(\mathcal{B}_{\neg\phi}) = \emptyset$?

Checking for Emptiness of Intersection Automaton

Third Step: $\mathcal{L}^\omega(\mathcal{B}_\mathcal{T}) \cap \mathcal{L}^\omega(\mathcal{B}_{\neg\phi}) = \emptyset$?

Intersection Automaton

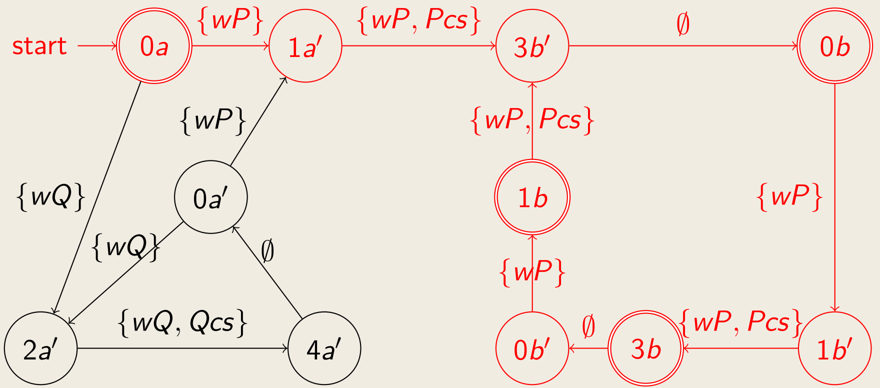


Checking for Emptiness of Intersection Automaton

Third Step: $\mathcal{L}^\omega(\mathcal{B}_\mathcal{T}) \cap \mathcal{L}^\omega(\mathcal{B}_{\neg\phi}) \neq \emptyset$

Counterexample

Intersection Automaton

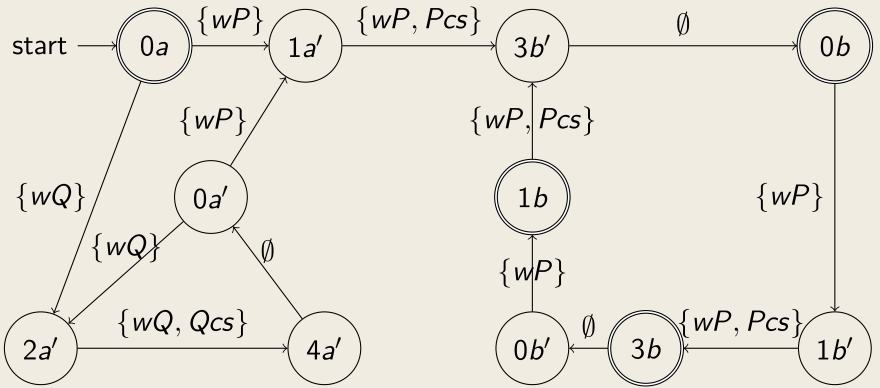


Checking for Emptiness of Intersection Automaton

Third Step: $\mathcal{L}^\omega(\mathcal{B}_\mathcal{T}) \cap \mathcal{L}^\omega(\mathcal{B}_{\neg\phi}) \neq \emptyset$

Counterexample Construction of intersection automaton

Intersection Automaton



Applying Temporal Logic to Critical Section Problem

We want to verify `[](critical<=1)` as a correctness property of:

```
int critical = 0;

active proctype P() {
  do :: printf("P␣non-critical␣actions\n");
        atomic {
          !Q_in_CS;
          P_in_CS = true
        }
        critical++;
        printf("P␣uses␣shared␣recourses\n");
        critical--;
        P_in_CS = false
  od
}

active proctype Q() {
  ...correspondingly...
}
```

Model Checking a Safety Property with JSPIN

edit 'LTL fomula' field of JSPIN

1. load PROMELA file in JSPIN (not necessarily containing `ltl ...`)
2. enter `[](critical <= 1)` in LTL text field of JSPIN
3. select `Translate` to create a 'never claim', corresponding to the negation of the formula
4. ensure `Safety` is selected
5. select `Verify`
6. (if necessary) select `Stop` to terminate too long verification

Demo: `csGhostLTL.pml`

Model Checking against Temporal Logic Property

Theory behind SPIN

1. Represent the interleaving of all processes as a single automaton (only one process advances in each step), called $\mathcal{M}$

Model Checking against Temporal Logic Property

Theory behind SPIN

1. Represent the interleaving of all processes as a single automaton (only one process advances in each step), called $\mathcal{M}$
2. Construct Büchi automaton (never claim) $\mathcal{NC}_{\neg\phi}$ for negation of TL formula ϕ to be verified

Model Checking against Temporal Logic Property

Theory behind Spin

1. Represent the interleaving of all processes as a single automaton (only one process advances in each step), called $\mathcal{M}$
2. Construct Büchi automaton (never claim) $\mathcal{NC}_{\neg\phi}$ for negation of TL formula ϕ to be verified
3. If
$$\mathcal{L}^{\omega}(\mathcal{M}) \cap \mathcal{L}^{\omega}(\mathcal{NC}_{\neg\phi}) = \emptyset$$
then ϕ holds in $\mathcal{M}$,
otherwise we have a counterexample

Model Checking against Temporal Logic Property

Theory behind SPIN

1. Represent the interleaving of all processes as a single automaton (only one process advances in each step), called $\mathcal{M}$
2. Construct Büchi automaton (never claim) $\mathcal{NC}_{\neg\phi}$ for negation of TL formula ϕ to be verified
3. If
$$\mathcal{L}^{\omega}(\mathcal{M}) \cap \mathcal{L}^{\omega}(\mathcal{NC}_{\neg\phi}) = \emptyset$$
then ϕ holds in $\mathcal{M}$,
otherwise we have a counterexample
4. To check $\mathcal{L}^{\omega}(\mathcal{M}) \cap \mathcal{L}^{\omega}(\mathcal{NC}_{\neg\phi})$ construct intersection automaton (both automata advance in each step) and search for accepting run

Temporal Model Checking without Ghost Variables

We want to verify mutual exclusion without using ghost variables

```
bool inCriticalP = false, inCriticalQ = false;

active proctype P() {
  do :: atomic {
          !inCriticalQ;
          inCriticalP = true
        }
cs:     /* critical activity */
        inCriticalP = false
  od
}

/* similar for process Q with same label cs: */

ltl m { []!(P@cs && Q@cs) }
```

Demo: `noGhost.pml`

Why SPIN?

- SPIN targets software, instead of hardware verification ("*Software* Engineering using Formal Methods")
- 2001 ACM Software Systems Award (other winning software systems include: Unix, TCP/IP, WWW, Tcl/Tk, Java)
- used for safety critical applications
- distributed freely as research tool, well-documented, actively maintained, large user-base in academia and in industry
- annual SPIN user workshops series held since 1995
- based on standard theory of (ω-)automata and linear temporal logic

Interested?

In order to

- learn more about Software Model Checking (SPIN)
- learn about Deductive Verification (KeY) of
  - a real-world language, here Java (without abstraction)
  - w.r.t. more complex, problem specific properties

Interested?

In order to

- learn more about Software Model Checking (Spin)
- learn about Deductive Verification (KeY) of
  - a real-world language, here Java (without abstraction)
  - w.r.t. more complex, problem specific properties

you are welcome to my course:

Software Engineering using Formal Methods