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

Real World

Abstraction

Formal Requirements Specification

Formal Execution Model
Formal Models for Software

Real World

over-simplification

e.g., zero delay

Formal Requirements Specification

Formal Execution Model
Formal Models for Software

- Real World
- Formal Requirements Specification
- Formal Execution Model

Missing requirement: e.g., max stack size
Formal Models for Software

Real World

wrong modeling

\[ \mathbb{Z} \text{ vs int} \]

Formal Requirements Specification

Formal Execution Model
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

<table>
<thead>
<tr>
<th>Abstract programs, Simple properties</th>
<th>Abstract programs, Complex properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concrete programs, Simple properties</td>
<td>Concrete programs, Complex properties</td>
</tr>
</tbody>
</table>
Main Approaches

<table>
<thead>
<tr>
<th>SPIN today</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract programs, Simple properties</td>
</tr>
<tr>
<td>Concrete programs, Simple properties</td>
</tr>
</tbody>
</table>
Main Approaches

<table>
<thead>
<tr>
<th>SPIN</th>
<th>KeY</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>not today</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract programs, Simple properties</th>
<th>Abstract programs, Complex properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concrete programs, Simple properties</td>
<td>Concrete programs, Complex properties</td>
</tr>
</tbody>
</table>
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
Model Checking

System Model

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

System Property

\[ \square ! (\text{criticalSectP} && \text{criticalSectQ}) \]

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
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

```c
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

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

One possible computation of this program

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

```c
1 active [2] proctype P() {
2 byte n;
3 n = 1;
4 n = 2
5 }
```
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**
     - Mutual 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.

⇒ 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
= reserving 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”
SPIN: “Simple Promela Interpreter”

The name is a serious understatement!
Model Checker for This Course: \texttt{SPIN}

\texttt{SPIN}: “Simple Promela Interpreter”

The name is a serious understatement!

main functionality of \texttt{SPIN}:

\begin{itemize}
\item simulating a model (randomly/interactively)
\item generating a verifier
\end{itemize}
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: \texttt{SPIN}

\texttt{SPIN}: "Simple Promela Interpreter"

The name is a serious understatement!

main functionality of \texttt{SPIN}:
- simulating a model (randomly/interactively)
- generating a verifier

verifier generated by \texttt{SPIN} is a C program performing model checking:
- exhaustively checks \texttt{PROMELA} model against correctness properties
- in case the check is negative: generates a failing run of the model
**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

- model name.pml
- correctness properties

SPIN

- -a
- -i
- -t

random/interactive/guided simulation

verifier pan.c

C compiler

executable verifier pan

failing run name.pml.trail

"errors: 0"

or

either
Plain Simulation with \textbf{SPIN}

- model \texttt{name.pml}
- correctness properties

\textbf{SPIN}

- verifier \texttt{pan.c}
- C compiler
- executable verifier \texttt{pan}

random/interactive/simulation

- \texttt{-i}

failing run \texttt{name.pml.trail}
Model Checking with SPIN

- Model: `name.pml`
- Correctness properties

- SPIN
- `verifier pan.c`
- C compiler
- Executable verifier `pan`

- `-a`
- Failing run `name.pml.trail`
- "errors: 0"

- Either
- Or
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 $M$. 

But how to state Correctness Properties?
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 $C_i$. ($R_{M,C_i} \subseteq R_M$)

But how to state Correctness Properties?
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
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 w.r.t. $C_1, \ldots, C_n$ 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.
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

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.
Correctness properties can be stated *within*, or *outside*, the model.
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
Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form
\[
\text{assert}(\text{expr})
\]
were \text{expr} is any PROMELA expression.
Assertion Statements

**Definition (Assertion Statements)**

Assertion statements in PROMELA are statements of the form

\[
\text{assert}(\text{expr})
\]

were \textit{expr} is any PROMELA expression.

Typically, \textit{expr} is of type bool.
Assertion Statements

**Definition (Assertion Statements)**

Assertion statements in PROMELA are statements of the form

\[ \text{assert}(expr) \]

were \( expr \) is any PROMELA expression.

Typically, \( expr \) is of type bool.

\text{assert}(expr)\ can appear wherever a statement is expected.
Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

\[ \text{assert} (expr) \]

were \( expr \) is any PROMELA expression.

Typically, \( expr \) is of type bool.

\( \text{assert}(expr) \) can appear wherever a statement is expected.

\[ \ldots \]
\[ \text{stmt1;} \]
\[ \text{assert}(\text{max} == \text{a}); \]
\[ \text{stmt2;} \]
\[ \ldots \]
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**: 

```plaintext
/* 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

Spin generates Verifier in C, called `pan.c` (plus helper files)

Command Line Execution

Generate Verifier in C

> spin -a max.pml
Compile To Executable Verifier

Command Line Execution

`compile to executable verifier`

```bash
> gcc -o pan pan.c
```
**Compile To Executable Verifier**

![Diagram showing the process of compiling a verifier from a C program.]

**Command Line Execution**

*compile to executable verifier*

```bash
> 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 "protocol analyzer", now "process analyzer"
Run Verifier (= Model Check)

**Executable**

`pan`

either

“errors: 0”

or

failing run `max.pml.trail`

---

**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)

Executable verifier `pan` either prints "errors: 0", or prints "errors: n" (n > 0)

Command Line Execution

```
run verifier pan
> ./pan  or  > pan
```

- prints "errors: 0", or
- prints "errors: n" (n > 0)
Run Verifier (= Model Check)

```
run verifier pan
> ./pan  or  > pan
```

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

Command Line Execution
Run Verifier (= Model Check)

```plaintext
run verifier pan
> ./pan  or  > pan
```

- prints “errors: 0”, or
- prints “errors: n” \((n > 0)\)  \(\Rightarrow\) 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):\(\text{max} = 1\)

spin: line 25 "max.pml", Error: assertion violated
spin: text of failed assertion:
\texttt{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\leq b))]\)
3: proc 0 (P) line 23 "max.pml" (state 14) \([\text{max} = a]\)
   P(0):\text{max} = 1
spin: line 25 "max.pml", Error: assertion violated
spin: text of failed assertion:
   \text{assert}(((\text{max}==((a>b)) \rightarrow (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) \([\text{max} = a]\)  
   \(P(0):\text{max} = 1\)

spin: line 25 "max.pml", Error: assertion violated
spin: text of failed assertion:
   \texttt{assert((max==((a>b)) \rightarrow (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)

model name.pml
correctness properties

SPIN
- a
- i
- t

verifier pan.c
C compiler
executable verifier pan

random/interactive/guided simulation

failing run name.pml.trail

"errors: 0"
What did we do so far?

following whole cycle (most primitive example, assertions only)

model name.pml

correctness properties

SPIN

verifier pan.c

C compiler

eXecutable verifier pan

random/interactive/guided simulation

-f p -l -g ...

failing run name.pml.trail

"errors: 0"
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.

```plaintext
byte n;

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

n is global

ë 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).
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)
}
byte n = 0;

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

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

```c
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

```c
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 resources\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

\texttt{SPIN} (./pan -E) shows no assertion is violated
\implies \text{mutual exclusion is verified}

still \texttt{SPIN} (without -E) reports (invalid end state)
\implies \text{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

\[
\text{chan } name = \left[\text{capacity}\right] \text{ of } \{\text{type}_1, \ldots, \text{type}_n\}
\]

Creates a channel, which is stored in \(name\)
Channels in Promela

\[ \text{chan } name = [\text{capacity}] \text{ of } \{\text{type}_1, \ldots, \text{type}_n\} \]

Creates a channel, which is stored in \textit{name}

Messages communicated via the channel are \( n \)-tuples \( \in \text{type}_1 \times \ldots \times \text{type}_n \)
Channels in Promela

```plaintext
chan name = [capacity] of {type₁, ..., typeₙ}
```

Creates a channel, which is stored in `name`

Messages communicated via the channel are $n$-tuples $\in type₁ \times \ldots \times typeₙ$

Can buffer up to `capacity` messages, if $capacity \geq 1$

$\Rightarrow$ “buffered channel”
Channels in Promela

\[
\text{chan } \text{name} = [\text{capacity}] \text{ of } \{\text{type}_1, \ldots, \text{type}_n\}
\]

Creates a channel, which is stored in \text{name}

Messages communicated via the channel are \(n\)-tuples \(\in \text{type}_1 \times \ldots \times \text{type}_n\)

Can buffer up to \text{capacity} messages, if \text{capacity} \(\geq 1\)
\(\Rightarrow \) “buffered channel”

The channel has \text{no} buffer, if \text{capacity} = 0
\(\Rightarrow \) “rendezvous channel”
Example:

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

Creates a channel, which is stored in `ch`
Channels in **Promela** cont’d

**Example:**

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

Creates a channel, which is stored in `ch`

Messages communicated via `ch` are 3-tuples \( \in \text{mtype} \times \text{byte} \times \text{bool} \)
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:
Example:

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

Creates a channel, which is stored in `ch`

Messages communicated via `ch` are 3-tuples ∈ `mtype × byte × 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:

\[ name ! expr_1, \ldots, expr_n \]

- \( name \): channel variable
- \( expr_1, \ldots, expr_n \): sequence of expressions, where number and types match message type
- sends values of \( expr_1, \ldots, expr_n \) as one message
- example: \( ch ! \) green, 20, false

**receive statement** has the form:

\[ name ? var_1, \ldots, var_n \]

- \( name \): channel variable
- \( var_1, \ldots, var_n \): sequence of variables, where number and types match message type
- assigns values of message to \( var_1, \ldots, var_n \)
- example: \( ch ? \) color, time, flash
Rendezvous Channels

```c
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?

⇒ ask SpinSpider
Rendezvous Channels

```c
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

```c
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?  ⇒ ask \texttt{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**

\[
\begin{array}{c}
\text{Sender} \\
\downarrow \\
(11,45) \\
\end{array} \quad \rightarrow \quad \begin{array}{c}
\text{Receiver} \\
\downarrow \\
\text{hour,minute} \\
\end{array}
\]
Reply Channels - Single Server

```haskell
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
}
```
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

```plaintext
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) 
    Is the assertion valid? Ask SPIN.
}
active proctype RudeClient() {
    mtype msg;
    request ! rude; reply ? msg
}
More realistic with several servers:

```prolog
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:

```plaintext
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
}
```

Analyse with Spin.
Several Servers

More realistic with several servers:

```plaintext
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:

```prolog
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
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

Real World

Formalisation

Formal Model
Formalisation: Syntax, Semantics

Real World

Formal Language

Formal Semantics

Syntax

Semantics
Formalisation: Syntax, Semantics

Real World

Formal Language

Syntax

Semantics

Formal Semantics

has model
Formalisation: Syntax, Semantics

Real World

Propositional Logic

Valuation

Syntax

Semantics

has model
Formalisation: Syntax, Semantics

Real World

Promela + Temporal Logic

All Runs $\sigma +$

Valuation in $\sigma$

Syntax

Semantics

has model
Formalisation: Syntax, Semantics

Real World

Syntax
Semantics

Temporal Logic
Promela
All Runs $\sigma =$
Transition System
Formalisation: Syntax, Semantics, Proving

Real
World

Syntax
Semantics

Temporal Logic
Promela
All Runs $\sigma =$
Transition System

How to do proving?
Formal Verification: Model Checking

Real World

Syntax

Sem.

Promela

Transition System

Translation of Negation

Büchi Automaton
Formal Verification: Model Checking

Real World → Syntax → Promela → Translation of Negation → Büchi Automaton → Intersection accepts no run?

Syntax → Semantics → Transition System
Transition systems (aka Kripke Structures)

\[
\begin{array}{cccc}
 s_0 & s_1 & s_2 & s_3 \\
 F F & T F & T T & F T \\
p = T; & q = p; & p = F; & \end{array}
\]

Notation

- name: update
- interp.: x
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

Real World

TL

Promela

Transition System

Syntax

Sem.
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 $\phi$ is a formula then so is $\Box \phi$

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

**Until** If $\phi$ and $\psi$ are formulas then so is $\phi U \psi$

Concrete Syntax

<table>
<thead>
<tr>
<th></th>
<th>text book</th>
<th>SPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Always</strong></td>
<td>$\Box$</td>
<td>$[\ ]$</td>
</tr>
<tr>
<td><strong>Eventually</strong></td>
<td>$\Diamond$</td>
<td>$&lt;&gt;$</td>
</tr>
<tr>
<td><strong>Until</strong></td>
<td>$U$</td>
<td>U</td>
</tr>
</tbody>
</table>
Formal Verification: Model Checking

- Real World
- Syntax
- Sem.
- TL
- Promela
- Transition System
- Büchi Automaton
- Translation of Negation
Given a finite alphabet (vocabulary) $\Sigma$

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

$$w = a_o \cdots a_n$$

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

$L \subseteq \Sigma^*$ is called a language
\(\omega\)-Languages

Given a finite alphabet (vocabulary) \(\Sigma\)

An \(\omega\)-word \(w \in \Sigma^\omega\) is an infinite sequence

\[ w = a_0 \cdots a_k \cdots \]

with \(a_i \in \Sigma, i \in \mathbb{N}\)

\(L^\omega \subseteq \Sigma^\omega\) is called an \(\omega\)-language
Büchi Automaton

Definition (Büchi Automaton)

A (non-deterministic) Büchi automaton over an alphabet $\Sigma$ 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

$\Sigma = \{a, b\}, Q = \{q_1, q_2, q_3\}, I = \{q_1\}, F = \{q_2\}$
Definition (Execution)

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

- $w = a_0 \cdots a_k \cdots \in \Sigma^\omega$
- $v = q_0 \cdots q_k \cdots \in Q^\omega$

where $q_0 \in I$, and $(q_i, a_i, q_{i+1}) \in \delta$, for all $i \in \mathbb{N}$.
Definition (Execution)
Let $B = (Q, I, F, \delta)$ be a Büchi automaton over alphabet $\Sigma$. An execution of $B$ is a pair $(w, v)$, with

- $w = a_0 \cdots a_k \cdots \in \Sigma^\omega$
- $v = q_0 \cdots q_k \cdots \in Q^\omega$

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

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

denotes the $\omega$-language recognised by $\mathcal{B}$. 
Let $\mathcal{B} = (Q, I, F, \delta)$ be a Büchi automaton, then

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

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

An $\omega$-language for which an accepting Büchi automaton exists is called $\omega$-regular language.
Example, $\omega$-Regular Expression

Which language is accepted by the following Büchi automaton?

$\omega$-regular expressions like standard regular expression

$\omega$-

$n$: $ab \omega = a(ba \omega)$
Which language is accepted by the following Büchi automaton?

Solution: $(a + b)^* (ab)\omega$   

[NB: $(ab)^\omega = a(ba)^\omega$]
Example, $\omega$-Regular Expression

Which language is accepted by the following Büchi automaton?

$$\begin{array}{c}
\text{start} \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \\
a, b \quad a \quad b
\end{array}$$

Solution: $(a + b)^* (ab)^\omega$  

[NB: $(ab)^\omega = a(ba)^\omega$]

$\omega$-regular expressions like standard regular expression

- $ab$ a then $b$
- $a + b$ a or $b$
- $a^*$ arbitrarily, but finitely often $a$
- new: $a^\omega$ infinitely often $a$
Formal Verification: Model Checking

Real World → Syntax of Promela → Semantics → Translation of Negation → Büchi Automaton

Intersection accepts no run?
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 $\phi$ satisfied in all runs of $\mathcal{T}$, i.e.,

$$\mathcal{T} \models \phi$$
\( \mathcal{T} \models \phi \) ?

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

\( T \models \phi \ ? \)

1. Represent transition system \( T \) as Büchi automaton \( B_T \) such that \( B_T \) accepts exactly those words corresponding to runs through \( T \)

2. Construct Büchi automaton \( B_{\neg \phi} \) for negation of formula \( \phi \)
Spin Model Checking—Overview

\[ \mathcal{T} \models \phi \ ? \]

1. Represent transition system \( \mathcal{T} \) as Büchi automaton \( B_{\mathcal{T}} \) such that \( B_{\mathcal{T}} \) accepts exactly those words corresponding to runs through \( \mathcal{T} \).

2. Construct Büchi automaton \( B_{\neg \phi} \) for negation of formula \( \phi \).

3. If

\[ \mathcal{L}^\omega (B_{\mathcal{T}}) \cap \mathcal{L}^\omega (B_{\neg \phi}) = \emptyset \]

then \( \phi \) holds.
Spin Model Checking—Overview

\[ \mathcal{T} \models \phi \, ? \]

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

2. Construct Büchi automaton \( \mathcal{B}_{\neg \phi} \) for negation of formula \( \phi \)

3. If

\[ \mathcal{L}^\omega(\mathcal{B}_T) \cap \mathcal{L}^\omega(\mathcal{B}_{\neg \phi}) = \emptyset \]

then \( \phi \) holds.

If

\[ \mathcal{L}^\omega(\mathcal{B}_T) \cap \mathcal{L}^\omega(\mathcal{B}_{\neg \phi}) \neq \emptyset \]

then each element of the set is a counterexample for \( \phi \).
Spin  Model Checking—Overview

\[ \mathcal{T} \models \phi \quad ? \]

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

2. Construct Büchi automaton \( \mathcal{B}_{\neg \phi} \) for negation of formula \( \phi \)

3. If

\[ \mathcal{L}(\mathcal{B}_T) \cap \mathcal{L}(\mathcal{B}_{\neg \phi}) = \emptyset \]

then \( \phi \) holds.

If

\[ \mathcal{L}(\mathcal{B}_T) \cap \mathcal{L}(\mathcal{B}_{\neg \phi}) \neq \emptyset \]

then each element of the set is a counterexample for \( \phi \).

To check \( \mathcal{L}(\mathcal{B}_T) \cap \mathcal{L}(\mathcal{B}_{\neg \phi}) \) 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 $B_\mathcal{T}$ accepting exactly those words representing a run of $\mathcal{T}$

Example

```plaintext
active procotpe 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 $T$ as Büchi automaton $B_T$ accepting exactly those words representing a run of $T$

Example

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

The property we want to check is $\phi = \square \neg Pcs$ (which does not hold)
B"uchi Automaton $B_{\neg \phi}$ for $\neg \phi$

Second Step:
Construct B"uchi Automaton corresponding to negated LTL formula

$T \models \phi$ holds iff there is no accepting run of $T$ 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

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

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

Büchi Automaton $B_{\neg \phi}$

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

```
start ----> qa -- \Sigma_{Pcs} ----> qb
\sum_{Pcs}
```

$$\Sigma_{Pcs} = \{l | l \in \Sigma, Pcs \in l\}, \quad \sum_{Pcs}^c = \Sigma - \Sigma_{Pcs}$$
Checking for Emptiness of Intersection Automaton

Third Step: \( \mathcal{L}^\omega(B_T) \cap \mathcal{L}^\omega(B_{\neg\phi}) = \emptyset \) ?
Checking for Emptiness of Intersection Automaton

Third Step: $\mathcal{L}^\omega(B_T) \cap \mathcal{L}^\omega(B_{\neg \phi}) = \emptyset$?
Checking for Emptiness of Intersection Automaton

**Third Step:** \( L^\omega(B_T) \cap L^\omega(B_{\neg \phi}) \neq \emptyset \)

Counterexample

**Intersection Automaton**

```
\begin{center}
\begin{tikzpicture}

\node (start) at (0,0) [draw, circle, thick, inner sep=0.3cm] {start};
\node (0a) at (1,1) [draw, circle, thick, inner sep=0.3cm] {0a};
\node (1a') at (2,2) [draw, circle, thick, inner sep=0.3cm] {1a'};
\node (3b') at (3,3) [draw, circle, thick, inner sep=0.3cm] {3b'};
\node (0b) at (4,4) [draw, circle, thick, inner sep=0.3cm] {0b};
\node (2a') at (-1,1) [draw, circle, thick, inner sep=0.3cm] {2a'};
\node (4a') at (-2,2) [draw, circle, thick, inner sep=0.3cm] {4a'};
\node (1b) at (1,3) [draw, circle, thick, inner sep=0.3cm] {1b};
\node (3b) at (2,2) [draw, circle, thick, inner sep=0.3cm] {3b};
\node (1b') at (1,1) [draw, circle, thick, inner sep=0.3cm] {1b'};

\draw [->, thick, red] (start) -- (0a) node [midway, above] \{wP\};
\draw [->, thick, red] (0a) -- (1a') node [midway, above] \{wP, Pcs\};
\draw [->, thick, red] (1a') -- (3b') node [midway, above] \{wP, Pcs\};
\draw [->, thick, red] (3b') -- (0b) node [midway, above] \{wP\};
\draw [->, thick, red] (0a) -- (2a') node [midway, above] \{wQ\};
\draw [->, thick, red] (2a') -- (4a') node [midway, above] \{wQ, Qcs\};
\draw [->, thick, red] (4a') -- (0a) node [midway, above] \{wQ\};
\draw [->, thick, red] (1a') -- (1b) node [midway, above] \{wP\};
\draw [->, thick, red] (1b) -- (3b) node [midway, above] \{wP\};
\draw [->, thick, red] (3b) -- (1b') node [midway, above] \{wP, Pcs\};
\draw [->, thick, red] (0b) -- (0b') node [midway, above] \{wP\};
\draw [->, thick, red] (3b') -- (3b) node [midway, above] \{wP, Pcs\};
\draw [->, thick, red] (0b') -- (1b') node [midway, above] \{wP\};
\end{tikzpicture}
\end{center}
```
Checking for Emptiness of Intersection Automaton

Third Step: \( L^\omega(B_T) \cap L^\omega(B_{\neg \phi}) \neq \emptyset \)

Counterexample Construction of intersection automaton

**Intersection Automaton**

- **Start state:** 0a
  - Transition: 0a' with label \( \{wP\} \)
  - Transition: 2a' with label \( \{wQ\}, \{wQ, Qcs\} \)
  - Transition: 4a' with label \( \emptyset \)

- **State 1a'**
  - Transition: 1a' with label \( \{wP, Pcs\} \)
  - Transition: 3b' with label \( \{wP, Pcs\} \)

- **State 3b'**
  - Transition: 3b' with label \( \emptyset \)
  - Transition: 1b' with label \( \{wP\} \)

- **State 0b'**
  - Transition: 0b' with label \( \emptyset \)
  - Transition: 3b with label \( \{wP, Pcs\} \)

- **State 1b'**
  - Transition: 1b' with label \( \{wP\} \)

Model Checking: A Taster

CHALMERS/GU 131014 66 / 72
Applying Temporal Logic to Critical Section Problem

We want to verify $[] (\text{critical} \leq 1)$ as a correctness property of:

```c
int critical = 0;

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

active proctype Q() {
  ...
correspondingly...
}
```
Model Checking a Safety Property with \textit{jSpin}

\textbf{edit ‘LTL fomula’ field of \textit{jSpin}}

1. load \texttt{Promela} file in \textit{jSpin} (not necessarily containing \texttt{ltl} \ldots )
2. enter \texttt{\([\text{critical} \leq 1]\)} in LTL text field of \textit{jSpin}
3. select Translate to create a ‘\texttt{never claim}’, corresponding to the negation of the formula
4. ensure Safety is selected
5. select Verify
6. (if necessary) select \texttt{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 $M$
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 $\phi$ to be verified
Theory behind SPIN

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

2. Construct Büchi automaton (never claim) $\mathcal{N}C_{\neg \phi}$ for negation of TL formula $\phi$ to be verified

3. If

$$L^\omega(M) \cap L^\omega(\mathcal{N}C_{\neg \phi}) = \emptyset$$

then $\phi$ holds in $M$,

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 $\phi$ to be verified

3. If

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

then $\phi$ 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
We want to verify mutual exclusion without using ghost variables

```cpp
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 \textbf{SPIN}?

- \textbf{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 \textbf{SPIN} user workshops series held since 1995
- based on standard theory of (\(\omega\)-)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