Software Engineering using Formal Methods
Modeling Concurrency

Wolfgang Ahrendt

08 September 2015
Concurrency: different processes trying not to run into each others’ way
Concurrent Systems – The Big Picture

Concurrency: different processes trying not to run into each others’ way

Main problem of concurrency: sharing computational resources

http://www.youtube.com/watch?v=JgMB6nEv7K0
http://www.youtube.com/watch?v=G8eqymwUFi8
Concurrency: different processes trying not to run into each others’ way

Main problem of concurrency: sharing computational resources

http://www.youtube.com/watch?v=JgMB6nEv7K0
http://www.youtube.com/watch?v=G8eqymwUFi8

Shared resource = crossing, bikers = processes,
Concurrent Systems – The Big Picture

Concurrency: different processes trying not to run into each others’ way

Main problem of concurrency: sharing computational resources

http://www.youtube.com/watch?v=JgMB6nEv7K0
http://www.youtube.com/watch?v=G8eqymwUFi8

Shared resource = crossing, bikers = processes, and a (data) race in progress, approaching a disaster.
Concurrency: different processes trying not to run into each others’ way

Main problem of concurrency: sharing computational resources

http://www.youtube.com/watch?v=JgMB6nEv7K0
http://www.youtube.com/watch?v=G8eqymwUFi8

Shared resource = crossing, bikers = processes, and a (data) race in progress, approaching a disaster.

Solutions to this must be carefully designed and verified, otherwise...
Concurrent Systems – The Big Picture
Focus of this Lecture

Aim of **SPIN**-style model checking methodology:

- exhibit flaws in software systems
Focus of this Lecture

Aim of \textit{SPIN}-style model checking methodology:

exhibit design flaws in \hspace{1cm} software systems
Focus of this Lecture

Aim of SPIN-style model checking methodology:

exhibit design flaws in concurrent and distributed software systems
Focus of this Lecture

Aim of SPIN-style model checking methodology:

- exhibit design flaws in **concurrent** and **distributed** software systems

Focus of this lecture:

- Modeling and analyzing concurrent systems
Aim of SPIN-style model checking methodology:

**exhibit design flaws in concurrent and distributed software systems**

Focus of this lecture:

- Modeling and analyzing concurrent systems

Focus of next lecture:

- Modeling and analyzing distributed systems
Concurrent/Distributed systems difficult to get right

problems:
  - hard to predict, hard to form faithful intuition
Concurrent/Distributed systems difficult to get right

problems:

- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
Concurrent/Distributed systems difficult to get right

problems:
- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
Concurrent/Distributed systems difficult to get right

problems:

- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
Concurrent/Distributed systems difficult to get right

problems:

- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
- limited control—from within applications—over ‘external’ factors:
Concurrent/Distributed systems difficult to get right

problems:

- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
- limited control—from within applications—over ‘external’ factors:
  - scheduling strategies
Concurrent/Distributed systems difficult to get right

problems:
- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
- limited control—from within applications—over ‘external’ factors:
  - scheduling strategies
  - relative speed of components
Concurrent/Distributed systems difficult to get right

problems:
- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
- limited control—from within applications—over ‘external’ factors:
  - scheduling strategies
  - relative speed of components
  - performance of communication mediums
Concurrent/Distributed systems difficult to get right

problems:

- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
- limited control—from within applications—over ‘external’ factors:
  - scheduling strategies
  - relative speed of components
  - performance of communication mediums
  - reliability of communication mediums
We cannot exhaustively test concurrent/distributed systems

- lack of controllability
  - we miss failures in test phase
We cannot exhaustively test concurrent/distributed systems

- lack of controllability
  ⇒ we miss failures in test phase
- lack of reproducability
  ⇒ even if failures appear in test phase, often impossible to analyze/debug defect
We cannot exhaustively *test* concurrent/distributed systems

- lack of controllability
  ⇒ we miss failures in test phase

- lack of reproducability
  ⇒ even if failures appear in test phase,
    often impossible to analyze/debug defect

- lack of time
  exhaustive testing exhausts the testers long before it exhausts behavior of the system...
Mission of Spin-style Model Checking

offer an efficient methodology to
  ▶ improve the design
  ▶ exhibit defects

of concurrent and distributed systems
Activities in Spin-style Model Checking

1. model (critical aspects of) concurrent/distributed system with Promela
2. state crucial properties with assertions, temporal logic, ...
3. use Spin to check all possible runs of the model
4. analyze result, possibly re-work 1. and 2.
Activities in Spin-style Model Checking

1. model (critical aspects of) concurrent/distributed system with Promela
2. state crucial properties with assertions, temporal logic, . . .
3. use Spin to check all possible runs of the model
4. analyze result, possibly re-work 1. and 2.

Separate concerns of model vs. property! Check the property you want the model to have, not the one it happens to have.
Main Challenges of Modeling

**expressiveness**
model must be expressive enough to ‘embrace’ defects the real system could have

**simplicity**
model must be simple enough to be ‘model checkable’, theoretically and practically
in the \textbf{SPIN} approach, the cornerstone of modeling concurrent/distributed systems are \textbf{PROMELA processes}
Initializing Processes

there is always an initial process prior to all others
often declared *implicitly* using ‘active’
there is always an initial process prior to all others
often declared *implicitly* using ‘active’

can be declared *explicitly* with key word ‘init’

```c
init {
    printf("Hello\nworld\n")
}
```

if *explicit*, `init` is used to start other processes with `run` statement
Starting Processes

processes can be started *explicitly* using `run`

```plaintext
proctype P() {
  byte local;
  ...
}
```

```plaintext
init {
  run P();
  run P();
  run P();
}
```

each `run` operator starts copy of process (with copy of local variables)
Starting Processes

processes can be started *explicitly* using `run`

```procy
type P() {
    byte local;
    ...
}
```

```procy
init {
    run P();
    run P();
}
```

each `run` operator starts copy of process (with copy of local variables)

`run P()` does *not* wait for `P` to finish
Starting Processes

processes can be started *explicitly* using `run`

```latex
proctype P() {
    byte local;
    ...
}

init {
    run P();
    run P();
    run P()
}
```

each `run` operator starts copy of process (with copy of local variables)

`run P()` does *not* wait for `P` to finish

(*PROMELA’s run corresponds to JAVA’s `start`, *not* to JAVA’s `run*)
Atomic Start of Multiple Processes

by convention, run operators enclosed in atomic block

```
proctype P() {
    byte local;
    ...
}

init {
    atomic {
        run P();
        run P();
    }
}
```
by convention, run operators enclosed in atomic block

```
proctype P() {
    byte local;
    ...
}
```

```
init {
    atomic {
        run P();
        run P()
    }
}
```

effect: processes only start executing once all are created
Atomic Start of Multiple Processes

by convention, run operators enclosed in atomic block

```c
proctype P() {
    byte local;
    ... }
```

```c
init {
    atomic {
        run P();
        run P();
    }
}
```

effect: processes only start executing once all are created

(more on atomic later)
Joining Processes

following trick allows ‘joining’, i.e., waiting for all processes to finish

byte result;

proctype P() {
    ...
}

init {
    atomic {
        run P();
        run P()
    }
    (_nr_pr == 1); /*blocks until join*/
    printf("result\n=%d", result)
}
Joining Processes

following trick allows ‘joining’, i.e., waiting for all processes to finish

```c
byte result;

proctype P() {
    ...
}

init {
    atomic {
        run P();
        run P()
    }
    (_nr_pr == 1); /*blocks until join*/
    printf("result␣=%d", result)
}
```

_nr_pr built-in variable holding number of running processes
_nr_pr == 1 only ‘this’ process (init) is (still) running
Processes may have formal parameters, instantiated by `run`:

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

init {
    run P(7, 10);
    run P(8, 15)
}
```
Active (Sets of) Processes

init can be made implicit by using the active modifier:

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

implicit init will run one copy of P
Active (Sets of) Processes

init can be made *implicit* by using the active modifier:

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

*implicit* init will run **one copy** of \( P \)

```c
active \([n]\) proctype P() {
    ...
}
```

*implicit* init will run **\( n \) copies** of \( P \)
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
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). (next lecture)
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 \texttt{P}, \texttt{n} = \texttt{%d}\n", n)
}

active proctype Q() {
    n = 2;
    printf("Process \texttt{Q}, \texttt{n} = \texttt{%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?
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
Atomicity

limit the possibility of sequences being interrupted by other processes

**weakly atomic sequence**

can *only* be interrupted if a statement is not executable

**strongly atomic sequence**

cannot be interrupted at all
Atomicity

limit the possibility of sequences being interrupted by other processes

weakly atomic sequence
    can only be interrupted if a statement is not executable
defined in PROMELA by atomic{ ... }

strongly atomic sequence
    cannot be interrupted at all
defined in PROMELA by d_step{ ... }
Deterministic Sequences

d_step:
  - strongly atomic
  - deterministic (like a single step)
Deterministic Sequences

d_step:

- strongly atomic
- deterministic (like a single step)
- choices resolved in fixed way (always take the first possible option)
  ⇒ avoid choices in d_step
Deterministic Sequences

d_step:

- strongly atomic
- deterministic (like a single step)
- choices resolved in fixed way (always take the first possible option)
  ⇒ avoid choices in d_step
- it is an error if any statement within d_step, other than the first one (called ‘guard’), blocks

```
d_step  {
    stmt1;  ← guard
    stmt2;
    stmt3
}
```

If stmt1 blocks, d_step is not entered, and blocks as a whole.

It is an error if stmt2 or stmt3 block.
(Weakly) Atomic Sequences

atomic:
  ▶ weakly atomic
  ▶ can be non-deterministic

atomic { 
  stmt1; ← guard
  stmt2;
  stmt3
}

If guard blocks, atomic is not entered, and blocks as a whole.

Once atomic is entered, control is kept until a statement blocks, and only then passed to another process.
Prohibit Interference by Atomicity

apply \texttt{atomic} or \texttt{d_step} to interference examples
PROMELA has *no synchronization primitives*, like semaphores, locks, or monitors.
**Synchronization on Global Data**

Promela has *no synchronization primitives*, like semaphores, locks, or monitors.

Instead, Promela inhibits concept of statement *executability*
**Synchronization on Global Data**

**Promela** has *no synchronization primitives*, like semaphores, locks, or monitors.

Instead, **Promela** inhibits concept of statement **executability**.

Executability addresses many issues in the interplay of processes.
PROMELA has *no synchronization primitives*, like semaphores, locks, or monitors.

Instead, PROMELA inhibits concept of statement executability.

Executability addresses many issues in the interplay of processes.

Most known synchronization primitives (e.g. test & set, compare & swap, semaphores) can be modelled using executability and atomicity.
Each statement has the notion of executability.

### Executability of basic statements:

<table>
<thead>
<tr>
<th>statement type</th>
<th>executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>always</td>
</tr>
<tr>
<td>assertion</td>
<td>always</td>
</tr>
<tr>
<td>print statement</td>
<td>always</td>
</tr>
<tr>
<td><strong>expression statement</strong></td>
<td>iff value not 0/false</td>
</tr>
<tr>
<td>send/receive statement</td>
<td>(next lecture)</td>
</tr>
</tbody>
</table>
Executability (Cont’d)

Executability of compound statements:
Executability (Cont’d)

Executability of **compound statements**:

- atomic resp. `d_step` statement is executable iff
- guard (i.e., the first inner statement) is executable
Executability (Cont’d)

Executability of **compound statements**:

atomic resp. `d_step` statement is executable
  iff
guard (i.e., the first inner statement) is executable

if resp. `do` statement is executable
  iff
any of its alternatives is executable
Executability of compound statements:

atomic resp. d_step statement is executable
iff
guard (i.e., the first inner statement) is executable

if resp. do statement is executable
iff
any of its alternatives is executable

an alternative is executable
iff
its guard (the first statement) is executable
Executability of compound statements:

atomic resp. d_step statement is executable
iff
guard (i.e., the first inner statement) is executable

if resp. do statement is executable
iff
any of its alternatives is executable

an alternative is executable
iff
its guard (the first statement) is executable

(recall: in alternatives, “->” syntactic sugar for “;”)
**Definition (Blocking)**

A statement blocks iff it is not executable.
A process blocks iff its location counter points to a blocking statement.

For each step of execution, the scheduler nondeterministically chooses a process to execute.
Exectability and Blocking

Definition (Blocking)

A statement blocks iff it is not executable.
A process blocks iff its location counter points to a blocking statement.

For each step of execution, the scheduler nondeterministically chooses a process to execute among the non-blocking processes.
Executability and Blocking

**Definition (Blocking)**

A statement blocks iff it is *not* executable.

A process blocks iff its location counter points to a blocking statement.

For each step of execution, the scheduler nondeterministically chooses a process to execute among the non-blocking processes.

Executability, resp. blocking are the key to PROMELA-style modeling of solutions to synchronization problems. (to be discussed in the following)
archetypical problem of concurrent systems
given a number of looping processes, each containing a critical section
design an algorithm such that:
The Critical Section Problem

archetypical problem of concurrent systems
given a number of looping processes, each containing a critical section
design an algorithm such that:

**Mutual Exclusion** At most one process is executing its critical section at any time.
The Critical Section Problem

archetypical problem of concurrent systems

given a number of looping processes, each containing a critical section
design an algorithm such that:

**Mutual Exclusion**  At most one process is executing its critical section at any time.

**Absence of Deadlock**  If *some* processes are trying to enter their critical sections, then *one* of them must eventually succeed.
The Critical Section Problem

archetypical problem of concurrent systems
given a number of looping processes, each containing a critical section
design an algorithm such that:

**Mutual Exclusion**  At most one process is executing its critical section at any time.

**Absence of Deadlock**  If some processes are trying to enter their critical sections, then one of them must eventually succeed.

**Absence of (individual) Starvation**  If any process tries to enter its critical section, then that process must eventually succeed.
Critical Section Pattern

for demonstration and simplicity:
(non)critical sections only printf statements

active proctype P() {
  do :: printf("P\non-critical\actions\n");
  /* begin critical section */
  printf("P\uses\shared\recourses\n")
  /* end critical section */
  od
}

active proctype Q() {
  do :: printf("Q\non-critical\actions\n");
  /* begin critical section */
  printf("Q\uses\shared\recourses\n")
  /* end critical section */
  od
}
No Mutual Exclusion Yet

More infrastructure to achieve ME.
Adding two Boolean flags:

```c
bool P_in_CS = false;
bool Q_in_CS = false;
```

```c
active proctype P() {
    do :: printf("P␣non-critical␣actions\n");
        P_in_CS = true;
    /* begin critical section */
    printf("P␣uses␣shared␣recourses\n");
    /* end critical section */
    P_in_CS = false
    od
}
```

```c
active proctype Q() {
    ...correspondingly...
}
```
Show Mutual Exclusion VIOLATION with **SPIN**

adding assertions

```c
bool P_in_CS = false;
bool Q_in_CS = false;

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

active proctype Q() {
    ..........assert(!P_in_CS);........
}
```
Mutual Exclusion by Busy Waiting

bool P_in_CS = false;
bool Q_in_CS = false;

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

active proctype Q() { ...correspondingly... }
Mutual Exclusion by Blocking

instead of Busy Waiting, process should

- yield control
- continuing to run only when exclusion properties are fulfilled
Mutual Exclusion by Blocking

instead of Busy Waiting, process should

- yield control
- continuing to run only when exclusion properties are fulfilled

We can use expression statement \(!Q\_in\_CS\),
to let process \(P\) block where it should not proceed!
Mutual Exclusion by Blocking

active proctype P() {
    do :: printf("P\nnon-critical\nactions\n");
    P_in_CS = true;
    !Q_in_CS;
    /* begin critical section */
    printf("P\nuses\nshared\nresources\n");
    assert(!Q_in_CS);
    /* end critical section */
    P_in_CS = false
    od
}

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

Verify Mutual Exclusion of this

Verify with $\text{SPIN}$
Verify Mutual Exclusion of this

Verify with $\text{SPIN}$

$\text{SPIN}$ error (invalid end state)
⇒ deadlock
Verify Mutual Exclusion of this

Verify with Spin

Spin error (invalid end state)
⇒ deadlock

can make pan ignore the deadlock: ./pan -E
Verify Mutual Exclusion of this

Verify with SPIN

SPIN error (invalid end state)  ⇒  deadlock

can make pan ignore the deadlock:  ./pan  -E

SPIN still reports assertion violation(!)
Proving Mutual Exclusion

- mutual exclusion (ME) cannot be shown by SPIN
Proving Mutual Exclusion

- mutual exclusion (ME) cannot be shown by Spin
- P/Q_in_CS sufficient for achieving ME
Proving Mutual Exclusion

- mutual exclusion (ME) cannot be shown by Spin
- $P/Q\_in\_CS$ sufficient for achieving ME
- $P/Q\_in\_CS$ not sufficient for proving ME
Proving Mutual Exclusion

- mutual exclusion (ME) cannot be shown by Spin
- P/Q in CS sufficient for achieving ME
- P/Q in CS not sufficient for proving ME

need more infrastructure:
ghost variables, only for proving / model checking
int critical = 0;

active proctype P() {
    do :: printf("P\nnon-critical\nactions\n");
    P_in_CS = true;
    !Q_in_CS;
    /* begin critical section */
    critical++;
    printf("P\nuses\nshared\nrecourses\n");
    assert(critical < 2);
    critical--;
    /* end critical section */
    P_in_CS = false

    od
}

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

Verify Mutual Exclusion of this

\texttt{Spin (./pan -E)} shows no assertion is violated
\Rightarrow \text{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
- OK if it is not crucial to continue – see end.pml
- If it is crucial to continue:
  Real deadlock
Invalid End State:

- A process does not finish at its end
- OK if it is not crucial to continue – see end.pml
- If it is crucial to continue:
  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 (not using the end...: labels or -E option)
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

(demonstrae that in csGhost.pml)
Atomicity against Deadlocks

solution:

checking and setting the flag in one atomic step

(demonstrae that in csGhost.pml)

atomic {
    !Q_in_CS;
    P_in_CS = true
}
Variations of Critical Section Problem

designated artifacts for verification:

- ghost variables (‘verification only’ variables)
- temporal logic (later in the course)
  - $\max \ n$ processes allowed in critical section
  - modeling possibilities include:
    - counters instead of booleans
    - semaphores (see demo)
  - more fine grained exclusion conditions, e.g.
    - several critical sections (Leidestraat in Amsterdam)
    - writers exclude each other and readers
    - readers exclude writers, but not other readers
    - FIFO queue semaphores, for fairly choosing processes to enter
  - ... and many more
Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (‘verification only’ variables)
Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables ('verification only' variables)
  - temporal logic (later in the course)
Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)
- max $n$ processes allowed in critical section

modeling possibilities include:

- counters instead of booleans
- semaphores (see demo)
- more fine grained exclusion conditions, e.g.
  - several critical sections (Leidestraat in Amsterdam)
  - writers exclude each other and readers
  - readers exclude writers, but not other readers
  - FIFO queue semaphores, for fairly choosing processes to enter
  - ... and many more
Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables ('verification only' variables)
  - temporal logic (later in the course)
- max $n$ processes allowed in critical section
  modeling possibilities include:
  - counters instead of booleans
Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables ('verification only' variables)
  - temporal logic (later in the course)

- max $n$ processes allowed in critical section

modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)
Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)

- max $n$ processes allowed in critical section
- modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)

- more fine grained exclusion conditions, e.g.
Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)
- max \( n \) processes allowed in critical section
- modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)
- more fine grained exclusion conditions, e.g.
  - several critical sections (Leidestraat in Amsterdam)
Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)
- max $n$ processes allowed in critical section
- modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)
- more fine grained exclusion conditions, e.g.
  - several critical sections (Leidestraat in Amsterdam)
  - writers exclude each other and readers
    - readers exclude writers, but not other readers
Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)

- max $n$ processes allowed in critical section
  modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)

- more fine grained exclusion conditions, e.g.
  - several critical sections (Leidestraat in Amsterdam)
  - writers exclude each other and readers
    readers exclude writers, but not other readers
  - FIFO queue semaphores, for fairly choosing processes to enter
Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)
- max $n$ processes allowed in critical section
- modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)
- more fine grained exclusion conditions, e.g.
  - several critical sections (Leidestraat in Amsterdam)
  - writers exclude each other and readers
    - readers exclude writers, but not other readers
  - FIFO queue semaphores, for fairly choosing processes to enter
- ... and many more
Why Not Critical Section in Single Atomic Block?

Actually possible in this case. Also in interleaving example (counting via temp, see above).

But:
- does not carry over to variations (see previous slide)
- atomic only weakly atomic!
- d step excludes any nondeterminism!

Using atomic and d step too heavily, for too large blocks, can result in well-behaved models, while modelling the wrong system.
Why Not Critical Section in Single Atomic Block?

Actually possible in this case. Also in interleaving example (counting via `temp`, see above).

But:
- does not carry over to variations (see previous slide)
- `atomic` only weakly atomic!
- `d_step` excludes any nondeterminism!
Why Not Critical Section in Single Atomic Block?

Actually possible in this case.
Also in interleaving example (counting via \texttt{temp}, see above).

But:

- does not carry over to variations (see previous slide)
- \texttt{atomic} only weakly atomic!
- \texttt{d\_step} excludes any nondeterminism!

Using \texttt{atomic} and \texttt{d\_step} too heavily, for too large blocks, can result in well-behaved models, while modelling the wrong system.