Software Engineering using Formal Methods

Modeling Concurrency

Wolfgang Ahrendt & Richard Bubel & Wojciech Mostowski

7 September 2011

Concurrent Systems – The Big Picture

Concurrent system means doing things at the same time trying not to run into each others way.

Concurrent Systems – The Big Picture

Concurrent system means doing things at the same time trying not to run into each others way.

Doinig things at the same time can mean many things, crucial for us is sharing computational resources, mainly the memory.

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

Concurrent Systems – The Big Picture

Concurrent system means doing things at the same time trying not to run into each others way.

Doinig things at the same time can mean many things, crucial for us is sharing computational resources, mainly the memory.

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

shared resource = crossing, bikers = processes. . .

Concurrent Systems – The Big Picture

Concurrent system means doing things at the same time trying not to run into each others way.

Doinig things at the same time can mean many things, crucial for us is sharing computational resources, mainly the memory.

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, waiting for a disaster.

To control this one employs:

- blocking, locks (e.g. railway crossing)
- semaphores (traffic lights)
- busy waiting (a plane circulating an airport waiting to land)

These need to 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 SPIN-style model checking methodology:

exhibit design flaws in 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

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

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 about

Concurrent/Distributed systems difficult to get right

problems:

- hard to predict, hard to form faithful intuition about
- enormous combinatorial explosion of possible behavior

Concurrent/Distributed systems difficult to get right

problems:

- hard to predict, hard to form faithful intuition about
- 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 about
- 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 about
- 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 about
- 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 about
- 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 about
- 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 about
- 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

Testing Concurrent or Distributed System is Hard

We cannot exhaustively test concurrent/distributed systems

- lack of controllability
  $\Rightarrow$ we miss failures in test phase

Testing Concurrent or Distributed System is Hard

We cannot exhaustively test concurrent/distributed systems

- lack of controllability
  $\Rightarrow$ we miss failures in test phase
- lack of reproducability
  $\Rightarrow$ even if failures appear in test phase,
  often impossible to analyze/debug defect

Testing Concurrent or Distributed System is Hard

We cannot exhaustively test concurrent/distributed systems

- lack of controllability
  $\Rightarrow$ we miss failures in test phase
- lack of reproducability
  $\Rightarrow$ 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. use assertions, temporal logic, ... to model crucial properties
3. use Spin to check all possible runs of the model
4. analyze result, and possibly re-work **1.** and **2.**

Activities in SPIN-style Model Checking

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

Wolfgang claims:
The hardest part of Model Checking is **1.**

Activities in SPIN-style Model Checking

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

Wolfgang claims:
The hardest part of Model Checking is **1.**

I claim:
1. and **2.** go strongly side by side.

Activities in SPIN-style Model Checking

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

Wolfgang claims:
The hardest part of Model Checking is **1.**

I claim:
1. and **2.** go strongly side by side.

There still needs to be sepration of concerns:
model vs. property! I.e. verify the property you want the system to have, not the one it already has.

Main Challenges of Modeling

expressiveness
: model must be expressive enough to 'embrace' defects the real system could have

simplicity
: model simple enough to be 'model checkable', theoretically and practically

Modeling Concurrent Systems in Promela

cornerstone of
modeling concurrent and distributed systems in the SPIN approach are

PROMELA processes

Initializing Processes

there is always an initial process prior to all others

often declared *implicitly* using '**active**'

Initializing Processes

there is always an initial process prior to all others

often declared *implicitly* using '**active**'

can be declared *explicitly* with key word '**init**'

```
init {
  printf("Hello␣world\n")
}
```

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

Starting Processes

processes can be started *explicitly* using **run**

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

init {
  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**

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

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**

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

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

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()
  }
}
```

Atomic Start of Multiple Processes

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

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

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) ->
     printf("result␣=%d", result)
}
```

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) ->
     printf("result␣=%d", result)
}
```

`_nr_pr` built-in variable holding number of running processes

`_nr_pr == 1` only ‘this’ process (**init**) is (still) running

Process Parameters

Processes may have formal parameters, instantiated by **run**:

```
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:

```
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:

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

implicit **init** will **run** one copy of `P`

```
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

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**). (next lecture)

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

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
- **d**eterministic (like a single **step**)
- nondeterminism resolved in fixed way (always take the first option)
  ⇒ good style to avoid nondeterminism 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 **atomic** or **d_step** to interference examples

Synchronization on Global Data

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

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

most of known synchronization primitives (e.g. test & set, compare & swap, semaphores) can be modelled using executability and atomicity anyhow

Executability

Each statement has the notion of executability.
Executability of basic statements:

statement type	*executable*
assignments	always
assertions	always
print statements	always
expression statements	iff value not 0/**false**
send/receive statements	(next lecture)

Executability (Cont'd)

Executability of compound statements:

Executability (Cont'd)

Executability of compound statements:

atomic resp. **d_step** statement is executable
iff
guard (the first statement within) is executable

Executability (Cont'd)

Executability of compound statements:

atomic resp. **d_step** statement is executable
iff
guard (the first statement within) is executable

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

Executability (Cont'd)

Executability of compound statements:

atomic resp. **d_step** statement is executable
iff
guard (the first statement within) 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 (Cont'd)

Executability of compound statements:

atomic resp. **d_step** statement is executable
iff
guard (the first statement within) 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 ";")

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

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 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)

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:

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 it's critical section 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 it's critical section 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 it's critical section 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

need more infrastructure to achieve it:
adding two Boolean flags:

```
bool enterCriticalP = false;
bool enterCriticalQ = false;

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

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

Show Mutual Exclusion VIOLATION with SPIN

adding assertions

```
bool enterCriticalP = false;
bool enterCriticalQ = false;

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

active proctype Q() {
    ........assert(!enterCriticalP);........
}
```

Mutual Exclusion by Busy Waiting

```
bool enterCriticalP = false;
bool enterCriticalQ = false;

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

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

Mutual Exclusion by Blocking

instead of Busy Waiting, process should

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

Mutual Exclusion by Blocking

instead of Busy Waiting, process should

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

We can use expression statement `!enterCriticalQ`,
to let process `P` block where it should not proceed!

Mutual Exclusion by Blocking

```
int critical = 0;

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

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

Proving Mutual Exclusion

- mutual exclusion (ME) cannot be shown by SPIN

Proving Mutual Exclusion

- mutual exclusion (ME) cannot be shown by SPIN
- `enterCriticalP/Q` sufficient for *achieving* ME

Proving Mutual Exclusion

- mutual exclusion (ME) cannot be shown by Spin
- `enterCriticalP/Q` sufficient for *achieving* ME
- `enterCriticalP/Q` *not* sufficient for *proving* ME

Proving Mutual Exclusion

- mutual exclusion (ME) cannot be shown by Spin
- `enterCriticalP/Q` sufficient for *achieving* ME
- `enterCriticalP/Q` *not* sufficient for *proving* ME

Proving Mutual Exclusion

- mutual exclusion (ME) cannot be shown by SPIN
- `enterCriticalP/Q` sufficient for *achieving* ME
- `enterCriticalP/Q` *not* sufficient for *proving* ME

need more infrastructure:
ghost variables, only for proving / model checking

Show Mutual Exclusion with Ghost Variable

```
int critical = 0;

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

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

Verify Mutual Exclusion of this

SPIN
still errors (`invalid end state`)
⇒ deadlock
can make `pan` ignore the deadlock: `./pan -E`
SPIN then proves mutual exclusion

Deadlock Hunting

Invalid End State:

- A process does not finish at its end
- OK if it is not crucial to continue – see last lecture
- 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
- OK if it is not crucial to continue – see last lecture
- 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, not using the `end...:` labels or `-E` switch ;)

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 {
  !enterCriticalQ;
  enterCriticalP = true
}
```

Variations of Critical Section Problem

- designated artifacts for verification:

Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (verification only)

Variations of Critical Section Problem

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

Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (verification only)
  - temporal logic (later in the course)
- max n processes allowed in critical section
  modeling possibilities include:

Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (verification only)
  - 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)
  - 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)
  - 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)
  - 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)
  - 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)
  - 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 queues for entering sections (full semaphores)

Variations of Critical Section Problem

- designated artifacts for verification:
  - ghost variables (verification only)
  - 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 queues for entering sections (full semaphores)
- ... and many more

Solving CritSectPr with atomic/d_step only?

actually possible in this case (demo)
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!