Formale Methoden im Softwareentwurf

Modellierung von Nebenläufigkeit / Modeling Concurrency

Richard Bubel
(in Vertretung von R. Hähnle)

5 November 2018

Formale Methoden im Softwareentwurf
Modellierung von Nebenläufigkeit / Modeling Concurrency
Richard Bubel
(in Vertretung von R. Hähnle)
5 November 2018

Concurrent Systems — The Big Picture

Concurrent System:
"doing things at the same time trying not to get into each others way"

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

http://www.youtube.com/watch?v=JgMB6nEv7K0
http://www.buzzfeed.com/svoip/good-parallel-parking-4y59

shared resource = crossing/lane, mopeds/cars = 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 circling over an airport waiting to land)

These need to be carefully designed and verified, otherwise . . .

Concurrent Systems — The Big Picture
Concurrent System:
"doing things at the same time trying not to get into each others way"
Doing things at the same time can mean many things, crucial for us is:
sharing computational resources, mainly memory
http://www.youtube.com/watch?v=JgMB6nEv7K0
http://www.buzzfeed.com/svoip/good-parallel-parking-4y59
shared resource = crossing/lane, mopeds/cars = 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 circling over an airport waiting to land)
These need to be carefully designed and verified, otherwise . . .

Students trying to find a seat in the lecture hall

Concurrent Systems — A Deadlock

2018-11-05

FMiSE
└─This Lecture
└─Concurrent Systems — A Deadlock

Concurrent Systems — A Deadlock

Focus of this Lecture

Goal of SPIN-style model checking methodology:

To exhibit design flaws in concurrent and distributed software systems

Focus of today's lecture:

- Modeling and analyzing concurrent systems

Focus of next week's lecture:

- Modeling and analyzing distributed systems

Focus of this Lecture

Goal of SPIN-style model checking methodology:

To exhibit design flaws in concurrent and distributed software systems

Focus of today's lecture:

- Modeling and analyzing concurrent systems

Focus of next week's lecture:

- Modeling and analyzing distributed systems

Concurrent/Distributed Systems: Hard to Get Right

Some Problems of Concurrent/Distributed Systems

- Hard to predict, hard to form correct intuition about them
- Enormous combinatorial explosion of possible behavior
- Interleaving prone to unsafe operations ("data races")
- 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

Concurrent/Distributed Systems: Hard to Get Right
Some Problems of Concurrent/Distributed Systems
- Hard to predict, hard to form correct intuition about them
- Enormous combinatorial explosion of possible behavior
- Interleaving prone to unsafe operations ("data races")
- 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 (scheduling, delays, ...)
  ⇒ we miss failures in test phase
- Lack of reproducibility
  ⇒ even if failures appear in test phase,
  often impossible to analyze/debug defect
- Lack of resources
  ⇒ exhaustive testing exhausts the testers long before
  it exhausts behavior of the system ...

2018-11-05

FMiSE
└─This Lecture
└─Testing Concurrent or Distributed System is Hard

Testing Concurrent or Distributed System is Hard

We cannot exhaustively test concurrent/distributed systems
- Lack of controllability (scheduling, delays, ...)
  ⇒ we miss failures in test phase
- Lack of reproducibility
  ⇒ even if failures appear in test phase,
  often impossible to analyze/debug defect
- Lack of resources
  ⇒ exhaustive testing exhausts the testers long before
  it exhausts behavior of the system ...

Mission of SPIN-style Model Checking

To offer a model-based methodology for

- improving the design and
- to exhibit defects

of concurrent and distributed systems

Mission of SPIN-style Model Checking

To offer a model-based methodology for

- improving the design and
- to exhibit defects

of concurrent and distributed systems

Activities in Spin-style Model Checking

1. Model (critical aspects of) concurrent/distributed system in Promela
2. Use assertions, temporal logic, . . . to model crucial properties
3. Use Spin to check all possible runs of the model
4. Analyze result, possibly re-work **1.** and **2.**

Observations

- The hardest aspect of Model Checking tends to be **1.**
- **1.** and **2.** need to go hand in hand
- Only **3.** is—sometimes—“push-button”

Separation of concerns (system vs. property) is essential:
verify the property you want a system to have, not the one it already has

Activities in Spin-style Model Checking

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

Observations
- The hardest aspect of Model Checking tends to be 1.
- 1. and 2. need to go hand in hand
- Only 3. is—sometimes—“push-button”

Separation of concerns (system vs. property) is essential:
verify the property you want a system to have, not the one it already has

Main Challenge of Modeling

Conflicting Goals

Richness	Simplicity
Model must be rich enough to encompass defects the real system could have	Model must be simple enough to be checkable, both theoretically and in practice

Main Challenge of Modeling
Conflicting Goals
Richness
Model must be rich enough to encompass defects the real system could have
Simplicity
Model must be simple enough to be checkable, both theoretically and in practice

Modeling Concurrent Systems in PROMELA

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

PROMELA processes

Modeling Concurrent Systems in PROMELA

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

PROMELA processes

Initial Process

There is always exactly one initial process prior to all others

- Often declared implicitly using "**active**"

Initial process can be declared explicitly with keyword "**init**"

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

- If keyword **init** is supplied then this process can start other processes with **run** statement

2018-11-05

Initial Process

There is always exactly one initial process prior to all others
- Often declared implicitly using "active"

Initial process can be declared explicitly with keyword "init"

init {
  printf("Hello␣world\n")
}

- If keyword init is supplied then this process can start other processes with run statement

Starting Processes

Processes may be started explicitly from **init** using **run**

```
proctype P() { // not declared active
  byte local;
  ...
}

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

- Each **run** operator starts copy of process (with own local variables)
- **run** `P()` does not wait for `P` to finish (asynchronous behavior)

(PROMELA's **run** corresponds to JAVA's **`start`**, not to JAVA's **`run`**)

2018-11-05

Starting Processes

Processes may be started explicitly from init using run

proctype P() { // not declared active
byte local;
...
}

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

Each run operator starts copy of process (with own local variables)
run P() does not wait for P to finish (asynchronous behavior)

(PROMELA's run corresponds to JAVA's start, not to JAVA's run)

Atomic Start of Multiple Processes

Recommended to enclose **run** operators in **atomic** block
(otherwise, interleaving with other processes possible)

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

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

Effect: processes only start executing once all are created

(more on **atomic** later)

Atomic Start of Multiple Processes
Recommended to enclose run operators in atomic block
(otherwise, interleaving with other processes possible)
proctype P() {
byte local;
...
}
init {
atomic {
run P();
run P()
}
}
Effect: processes only start executing once all are created
(more on atomic later)

Joining (Synchronizing) Processes

A trick allows "join" of processes: waiting for all processes to finish

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

init {
  atomic {
    run P();
    run P()
  }
  (_nr_pr == 1) ->
     printf("ready")
}
```

- `_nr_pr` built-in variable holding number of running processes
- `_nr_pr == 1` only one process (**init**) still running

Joining (Synchronizing) Processes
A trick allows "join" of processes: waiting for all processes to finish
proctype P() { ... }
init { atomic { run P(); run P() } (_nr_pr == 1) -> printf("ready") }
_nr_pr built-in variable holding number of running processes
_nr_pr == 1 only one process (init) still running

Process Parameters

Processes may have arguments, instantiated by **run**

```
proctype P(byte i; bool b) {
  ...
}

init {
  run P(7, true);
  run P(8, false)
}
```


Process Parameters

Processes may have arguments, instantiated by run

proctype P(byte i; bool b) {
...
}

init {
run P(7, true);
run P(8, false)
}

Active (Set of) Processes

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

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

- implicit **init** process will **run** exactly one copy of `P`

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

- implicit **init** process will **run** *n* copies of `P`

Active (Set of) Processes
init can be made implicit by using the active modifier
active proctype P() { ... }
implicit init process will run exactly one copy of P
active [n] proctype P() { ... }
implicit init process will run n copies of P

Local and Global Data

Variables declared outside of any process are global to all processes

Variables declared inside a process are local to that process

```
byte n ;

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

n is global
t is local

Local and Global Data

Variables declared outside of any process are global to all processes

Variables declared inside a process are local to that process

byte n ;

proctype P() {
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

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

Communication media of distributed systems often modeled by global variables of channel type (**chan**) (next lecture)

Never use global variables to model process-local data!

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
Shared resources state of (printer, traffic light, ...) often modeled by global variables of Boolean or enumeration type (bool/mtype)
Communication media of distributed systems often modeled by global variables of channel type (chan) (next lecture)
Never use global variables to model process-local data!

Interference on Global Data

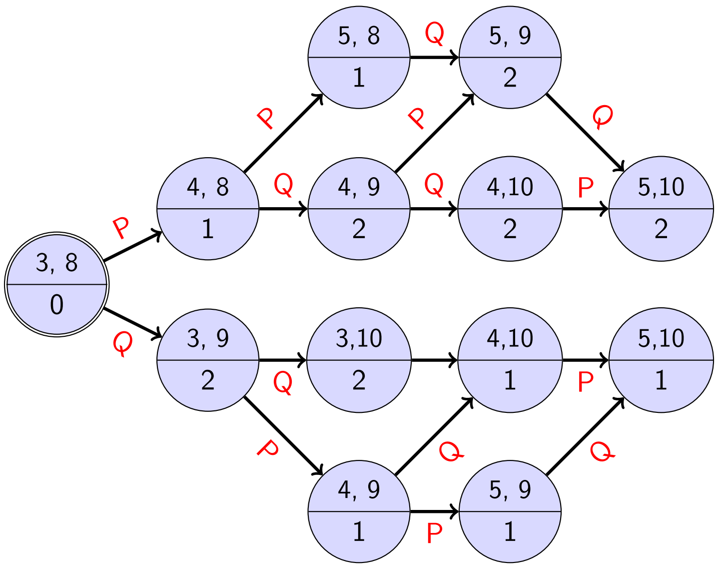
```
byte n = 0;

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

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

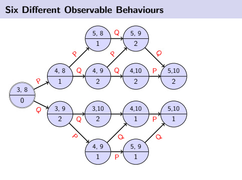
How many outputs possible? `interleave0.pml`

Processes can interfere on global data

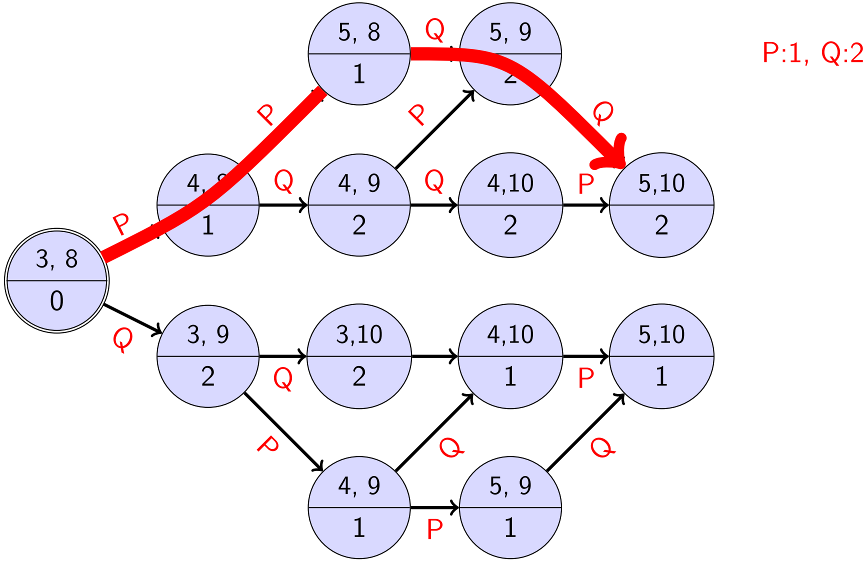
Interference on Global Data
byte n = 0;
active proctype P() {
n = 1;
printf("Process␣P,␣n␣=␣%d\n", n)
}
active proctype Q() {
n = 2;
printf("Process␣Q,␣n␣=␣%d\n", n)
}
How many outputs possible? interleave0.pml
Processes can interfere on global data

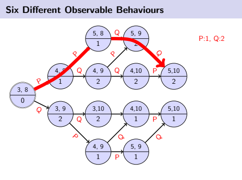
Six Different Observable Behaviours



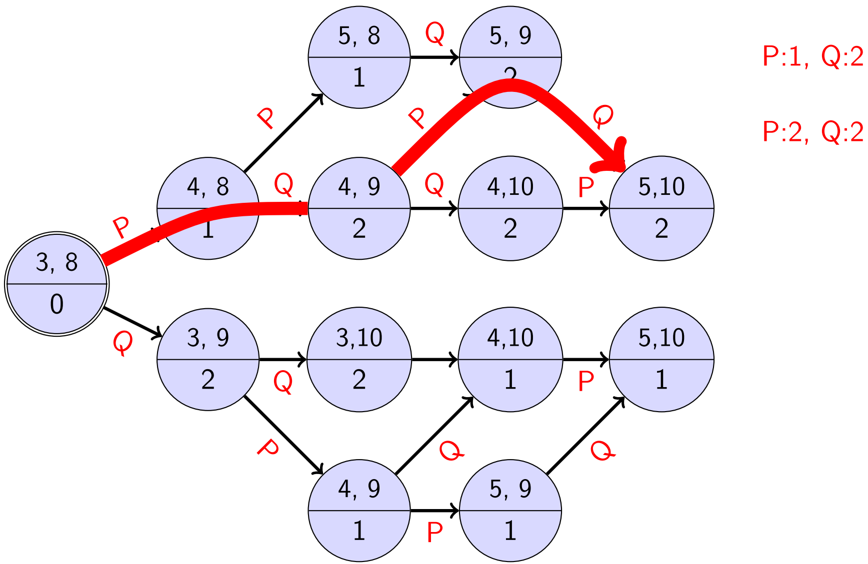


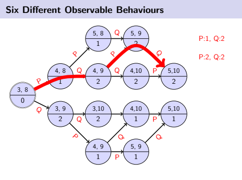
Six Different Observable Behaviours



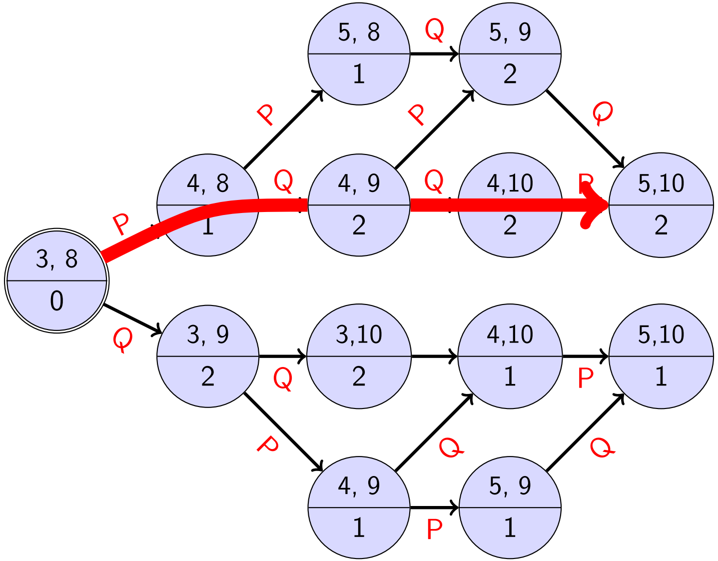


Six Different Observable Behaviours





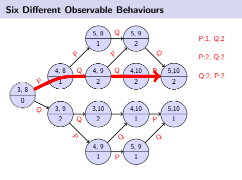
Six Different Observable Behaviours



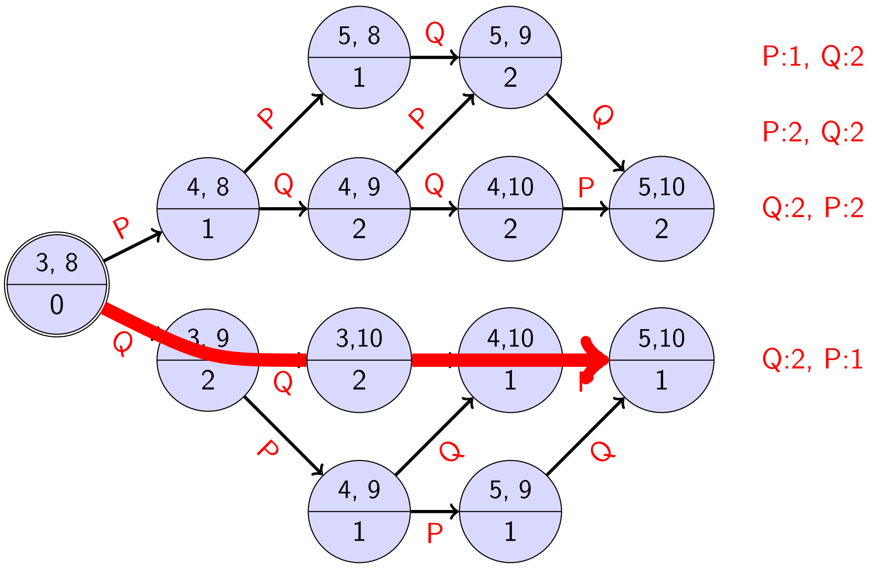
P:1, Q:2

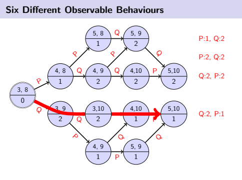
P:2, Q:2

Q:2, P:2

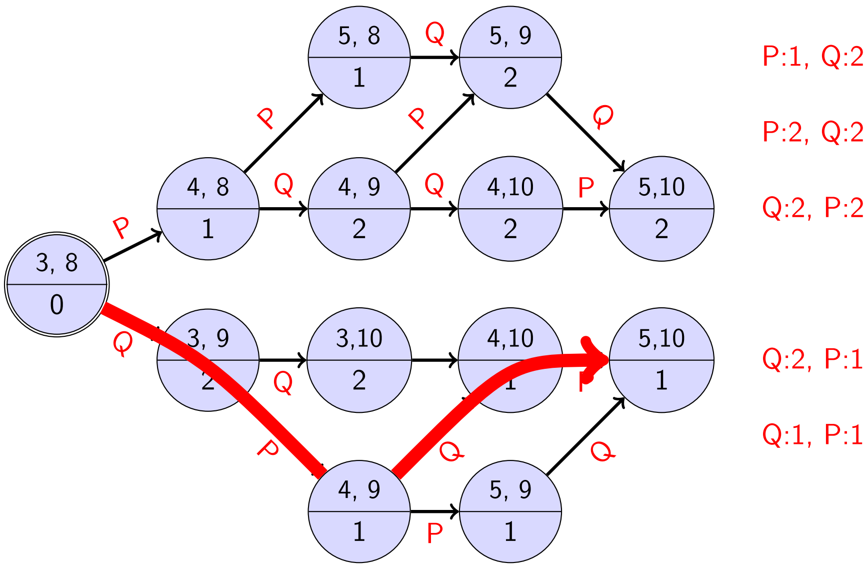


Six Different Observable Behaviours

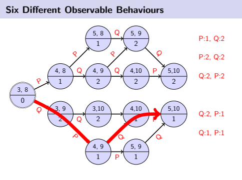




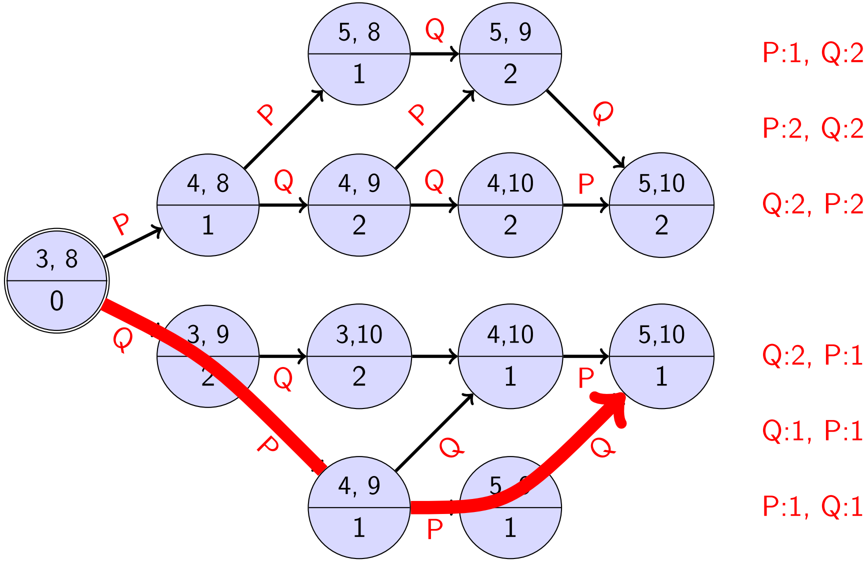
Six Different Observable Behaviours

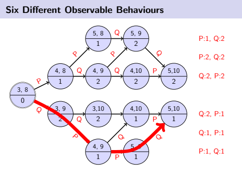


2018-11-05



Six Different Observable Behaviours





Examples

1. `interleave0.pml`
   SPIN simulation, automata
2. `interleave1.pml`, `interleave1A.pml`
   Adding assertion about `n`, model checking
3. `interleave5.pml`, `interleave5F.pml`, `interleave5A.pml`
   SPIN simulation, assertion, SPIN model checking, trail inspection
   show generated graph `interleave5.pdf`, modify assertion, verify

Examples

1. interleave0.pml
   SPIN simulation, automata
2. interleave1.pml, interleave1A.pml
   Adding assertion about n, model checking
3. interleave5.pml, interleave5F.pml, interleave5A.pml
   SPIN simulation, assertion, SPIN model checking, trail inspection
   show generated graph interleave5.pdf, modify assertion, verify

Atomicity

Limit possibilities of being interrupted ("pre-empted") by other processes

- Decrease the possible number of interleavings

Weakly atomic sequence
: can only be interrupted if a statement is not executable
$\Rightarrow$ defined in PROMELA by **atomic**{ ... }

Strongly atomic sequence
: cannot be interrupted at all
$\Rightarrow$ defined in PROMELA by **d_step**{ ... }

Atomicity
Limit possibilities of being interrupted ("pre-empted") by other processes
Decrease the possible number of interleavings
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**)
- non-determinism resolved in fixed way (always take the first option)
  ⇒ good style to avoid non-determinism 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

Deterministic Sequences
d_step:
strongly atomic
deterministic (like a single step)
non-determinism resolved in fixed way (always take the first option)
⇒ good style to avoid non-determinism 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 control is passed to another process

(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 control is passed to another process

Example for Limiting Interference by Atomicity

- `interleave5D.pml`
  Show assertion, verify

Example for Limiting Interference by Atomicity
interleave5D.pml
Show assertion, verify

Synchronization on Global Data

PROMELA has no synchronization primitives:

- semaphores
- locks
- monitors
- ...

Instead, PROMELA controls statement executability (absence of blocking)

- Non-executable statements in atomic sequences permit pre-emption

> Most known synchronization primitives (test & set, compare & swap, semaphores, ...) can be modelled using executability and atomicity

Synchronization on Global Data

PROMELA has no synchronization primitives:

- semaphores
- locks
- monitors
- ...

Instead, PROMELA controls statement executability (absence of blocking)

- Non-executable statements in atomic sequences permit pre-emption

Most known synchronization primitives (test & set, compare & swap, semaphores, ...) can be modelled using executability and atomicity

Executability

Each PROMELA statement has the property "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

Each PROMELA statement has the property "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

statement type	*executable iff*
atomic, **d_step**	guard (first statement of scope) executable
if, **do**	any of its alternatives is executable
alternative of **if**, **do**	guard (first statement of scope) executable (recall: "->" syntactic sugar for ";")
`for`	always (body can block, of course)

Executability (Cont'd)

Executability of compound statements

statement type	*executable iff*
atomic, **d_step**	guard (first statement of scope) executable
if, **do**	any of its alternatives is executable
alternative of **if**, **do**	guard (first statement of scope) executable (recall: "->" syntactic sugar for ";")
`for`	always (body can block, of course)

Executability and Blocking

Definition (Blocking)

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

For the next step of execution, the scheduler chooses non-deterministically one of the non-blocking statements in a process

> Executability/blocking are the basic concepts in PROMELA-style modeling of solutions to synchronization problems

Executability and Blocking
Definition (Blocking)
A statement is blocking iff it is not executable.
A process is blocking iff its location counter points to a blocking statement.
For the next step of execution, the scheduler chooses non-deterministically one of the non-blocking statements in a process
Executability/blocking are the basic concepts in PROMELA-style modeling of solutions to synchronization problems

The Critical Section Problem

Archetypical problem of concurrent systems

Definition (Critical Section)

The critical section (CS) of a process is the block of code where shared state (e.g., global variables) are accessed and possibly manipulated

Example

The PROMELA models `interleave?.pml` with global variable `n`

CS Problem (Data Race, Race Condition, "kritischer Wettlauf")

Given a set of processes each containing at least one critical section:

The result of the computation performed by the processes might depend on their execution order

The Critical Section Problem
Archetypical problem of concurrent systems
Definition (Critical Section)
The critical section (CS) of a process is the block of code where shared state (e.g., global variables) are accessed and possibly manipulated
Example
The PROMELA models interleave?.pml with global variable n
CS Problem (Data Race, Race Condition, "kritischer Wettlauf")
Given a set of processes each containing at least one critical section:
The result of the computation performed by the processes might depend on their execution order

The Critical Section Problem: Solutions

Given a number of looping processes, each containing a critical section

Mutual Exclusion

At most one process is executing its critical section at any given time

Challenges

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

Absence of Starvation If any process tries to enter its critical section, then that process must eventually succeed

The Critical Section Problem: Solutions

Given a number of looping processes, each containing a critical section

Mutual Exclusion

At most one process is executing its critical section at any given time

Challenges

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

Absence of Starvation If any process tries to enter its critical section, then that process must eventually succeed

Critical Section Pattern

For demo purposes, model (non-)critical sections by **printf** statements:

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

active proctype Q() {
  do :: printf("Q␣non-critical␣action\n");
        /* begin critical section */
        printf("Q␣uses␣shared␣resource\n")
        /* end critical section */
  od
}
```


Critical Section Pattern

For demo purposes, model (non-)critical sections by printf statements:

Mutual Exclusion: First Attempt

Simple idea: use Boolean flags to control access to critical section

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

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

active proctype Q() {
  analogous
}
```


Mutual Exclusion: First Attempt
Simple idea: use Boolean flags to control access to critical section
bool enterCriticalP = false;
bool enterCriticalQ = false;
active proctype P() {
do :: printf("P␣non-critical␣action\n");
enterCriticalP = true;
/* begin critical section */
printf("P␣uses␣shared␣resource\n");
/* end critical section */
enterCriticalP = false
od
}
active proctype Q() {
analogous
}

Verification of Mutual Exclusion Not Yet Possible

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

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

active proctype Q() {
    analogous
}
```

(csAssert.pml)

Verification of Mutual Exclusion Not Yet Possible

bool enterCriticalP = false;
bool enterCriticalQ = false;

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

active proctype Q() {
    analogous
}

(csAssert.pml)

Mutual Exclusion: Second Attempt

"Busy Waiting" `csBusy.pml`

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

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


Mutual Exclusion: Second Attempt
"Busy Waiting" csBusy.pml
bool enterCriticalP = false;
bool enterCriticalQ = false;
active proctype P() {
do :: printf("P␣non-critical␣action\n");
enterCriticalP = true;
do :: !enterCriticalQ -> break
:: else -> skip
od;
/* begin critical section */
printf("P␣uses␣shared␣resource\n");
assert(!enterCriticalQ);
/* end critical section */
enterCriticalP = false
od
}
active proctype Q() { analogous }

Discussion

Failed verification — Busy waiting is problematic

- Does not block execution, even if exclusion property fails
- Wasteful on resources

Instead of busy waiting, use blocking to:

- release control when exclusion property not fulfilled
- continue only when exclusion properties are fulfilled

Don't use assignment, but expression statement `!enterCriticalQ` to let process `P` block where it should not proceed!

Discussion

Failed verification — Busy waiting is problematic

- Does not block execution, even if exclusion property fails
- Wasteful on resources

Instead of busy waiting, use blocking to:

- release control when exclusion property not fulfilled
- continue only when exclusion properties are fulfilled

Don't use assignment, but expression statement !enterCriticalQ to let process P block where it should not proceed!

Mutual Exclusion: Third Attempt

Use `!enterCriticalQ` as a guard that blocks execution

```
// csBlocking.pml
bool enterCriticalP;
bool enterCriticalQ;

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

active proctype Q() { analogous }
```


Mutual Exclusion: Third Attempt
Use !enterCriticalQ as a guard that blocks execution
// csBlocking.pml
bool enterCriticalP;
bool enterCriticalQ;
active proctype P() {
do :: printf("P␣non-critical␣action\n");
enterCriticalP = true;
!enterCriticalQ;
/* begin critical section */
printf("P␣uses␣shared␣resource\n");
assert(!enterCriticalQ);
/* end critical section */
enterCriticalP = false
od
}
active proctype Q() { analogous }

Verifying Mutual Exclusion

Mutual Exclusion (ME) cannot be shown by SPIN

- `enterCriticalP/Q` sufficient for achieving ME
- `enterCriticalP/Q` insufficient for proving ME

Global vs. Local Properties

To verify ME one needs to ensure that at any time at most one process is in a critical section

- `assert` statements are code-local and insufficient for this
- Need mechanism that can express system-global properties

Some typical mechanisms to express global system properties

Ghost Variables global variables used only for specification/verification

Invariants properties that hold at certain times ⇒Temporal Logic

Verifying Mutual Exclusion
Mutual Exclusion (ME) cannot be shown by SPIN
enterCriticalP/Q sufficient for achieving ME
enterCriticalP/Q insufficient for proving ME
Global vs. Local Properties
To verify ME one needs to ensure that at any time at most one process is in a critical section
assert statements are code-local and insufficient for this
Need mechanism that can express system-global properties
Some typical mechanisms to express global system properties
Ghost Variables global variables used only for specification/verification
Invariants properties that hold at certain times ⇒Temporal Logic

Verify Mutual Exclusion with Ghost Variables

```
int critical = 0; // nr of processes in CS

active proctype P() {
  do :: printf("P␣non-critical␣action\n");
        enterCriticalP = true;
        !enterCriticalQ;
        /* begin critical section */
        critical++;
        printf("P␣uses␣shared␣resource\n");
        assert(critical <= 1);
        critical--;
        /* end critical section */
        enterCriticalP = false
  od
}

active proctype Q() { analogous }
```


Verify Mutual Exclusion with Ghost Variables
int critical = 0; // nr of processes in CS
active proctype P() {
do :: printf("P␣non-critical␣action\n");
enterCriticalP = true;
!enterCriticalQ;
/* begin critical section */
critical++;
printf("P␣uses␣shared␣resource\n");
assert(critical <= 1);
critical--;
/* end critical section */
enterCriticalP = false
od
}
active proctype Q() { analogous }

Verify Mutual Exclusion with SPIN

- Attempt to verify `csGhost.pml`

```
spin -a csGhost.pml; gcc -o pan pan.c; ./pan
```

- Simulate guided by trail

```
spin -g -p -t csGhost.pml
```

  - Both processes have set `enterCritical`
  - Both processes are at guard `!entercritical`
  - Neither can proceed
- Make pan ignore deadlocks (invalid end states)

```
./pan -E;
```


2018-11-05

FMiSE
└─Mutual Exclusion
└─Verify Mutual Exclusion with SPIN

Verify Mutual Exclusion with SPIN
- Attempt to verify csGhost.pml
spin -a csGhost.pml; gcc -o pan pan.c; ./pan
- Simulate guided by trail
spin -g -p -t csGhost.pml
- Both processes have set enterCritical
- Both processes are at guard !entercritical
- Neither can proceed
- Make pan ignore deadlocks (invalid end states)
./pan -E;

Deadlock Hunting

Invalid End State

- A process does not finish in an end state
- OK, if it is not crucial to continue (see previous lecture)
- Two or more inter-dependent processes do not finish at the end: Real deadlock

Finding Deadlocks with Spin

- Attempt verification to produce a failing run trail
- Guided simulation to see how the processes get to the deadlock
- Fix the code, but don't use `endXXX:`-labels or `-E` switch

Deadlock Hunting
Invalid End State
A process does not finish in an end state
OK, if it is not crucial to continue (see previous lecture)
Two or more inter-dependent processes do not finish at the end: Real deadlock
Finding Deadlocks with Spin
Attempt verification to produce a failing run trail
Guided simulation to see how the processes get to the deadlock
Fix the code, but don't use endXXX:-labels or -E switch

Atomicity against Deadlocks

Deadlock-free solution to ME problem with only flags/blocking is hard

Atomicity

- More powerful and general mechanism
- Often leads to conceptually simpler solutions
- But is not always a realistic system assumption

Idea for Solution of ME Problem by Atomicity

Check and set the critical section flag in one atomic step

```
atomic {
  !enterCriticalQ; // use as guard, must come first
  enterCriticalP = true
} // csGhostAtomic.pml
```


Atomicity against Deadlocks
Deadlock-free solution to ME problem with only flags/blocking is hard
Atomicity
More powerful and general mechanism
Often leads to conceptually simpler solutions
But is not always a realistic system assumption
Idea for Solution of ME Problem by Atomicity
Check and set the critical section flag in one atomic step
atomic {
!enterCriticalQ; // use as guard, must come first
enterCriticalP = true
} // csGhostAtomic.pml

Variations of Critical Section Problem

At most *n* processes allowed in critical section

Modeling possibilities include:

- counters instead of booleans
- semaphores
- test & set instructions (primitive for atomic block on previous slide)

Refined mutual exclusion conditions

- several critical sections (Leidseplein 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!

Variations of Critical Section Problem
At most n processes allowed in critical section
Modeling possibilities include:
counters instead of booleans
semaphores
test & set instructions (primitive for atomic block on previous slide)
Refined mutual exclusion conditions
several critical sections (Leidseplein 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!

Use Atomicity with Good Judgment

There is a trivial solution of the CS problem using atomicity (`csAtomic.pml`)

Using atomicity in such an extreme way has serious drawbacks

- Not generalizable to variations of the CS problem
- **atomic** only weakly atomic, blocking still possible
- **d_step** excludes any non-determinism

Use Atomicity with Good Judgment

There is a trivial solution of the CS problem using atomicity (csAtomic.pml)

Using atomicity in such an extreme way has serious drawbacks
- Not generalizable to variations of the CS problem
- atomic only weakly atomic, blocking still possible
- d_step excludes any non-determinism

Literature for this Lecture

Ben-Ari Chapter 3
Sections 4.1–4.4

Literature for this Lecture
Ben-Ari Chapter 3
Sections 4.1–4.4