AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Test of Distributed Systems
Lecture 2

Correctness and Concurrency:

Interleaving

Interference

Atomicity

Today's lecture

- Sequential programming
- Concurrent programming
- Interleaving
- Interference and non-determinism
- Atomicity
- Synchronization
- Blocking
- Semaphores

Today's lecture

- Sequential programming
- Concurrent programming
- Interleaving
- Interference and non-determinism
- Atomicity
- Synchronization
- Blocking
- Semaphores

Testing of a sequential program

- Specification
  - The integer logarithm program:

    k = log n

    if and only if

    $2^k <= n <= 2^{k+1}$
- Model and test the program in SPIN.
- Used assertions to state
  - Pre and post conditions
  - Expected properties of intermediate states

Skeleton of a program

- mostly C-syntax
- int k = 0; /* global variable */
- active proctype P() { /* process */

```
  active proctype P() {  /* process */
        …
  }
```

Conditional

- if
  :: branch1
  :: branch2
  …
  :: else …
  fi

Loop

- do
  :: branch1
  :: branch2
  …
  :: else …
  od

Expressions and Assertions

- expressions are statements
  e.g., x > 0

- assertions are written
  assert *expression*
  e.g., assert x > 0

Concurrency vs distribution

- A ***concurrent*** program is composed of ***processes*** that may communicate by means of ***shared variables***

- Contrast: a ***distributed*** program is composed of ***processes*** that can only ***communicate*** by means of dedicated ***channels***

A simple concurrent program 1

```
byte n=3;

active proctype P() {
        n=0;
        printf("Process P, n = %d\n", n)
}

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

Add
pre- and post-
conditions
to P() and Q()

A simple concurrent program 2

```
byte n=3;

active proctype P() {
        n=0;
        printf("Process P, n = %d\n", n)
}

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

Add
pre- and post-
conditions
to P() and Q()

What is the
final value of
variable n?

A simple concurrent program 3

```
byte n=3;

active proctype P() {
        n=0;
        printf("Process P, n = %d\n", n)
}

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

Add
pre- and post-
conditions
to P() and Q()

What is the
final value of
variable n?

Interleaving of processes

- How are P() and Q() executed?
  - Execution of P() and Q() is alternated arbitrarily, for example looking at program 1:

    P(): n=0

    Q(): n=2

    Q(): **printf**("Process Q, n = %d\n", n)

    P(): **printf**("Process P, n = %d\n", n)
  - Six possible executions
  - The alternating execution is called ***interleaving***

Interleaving, more formally

- An **interleaved computation** of two processes P and Q is a sequence of states

  $(s_0, s_1, s_2, \ldots)$

  where s_{j+1} follows s_j in the sequence *if* s_{j+1} *is obtained by executing in state* s_j *the statement at the location counter of* P *or* Q

Interference

- ***What is the difference*** between program 1, program 2 and program 3?
  - Program 1 and program 2 yield the same result
  - Program 3 may yield additionally ``n=1''
  - In all cases the result whether 0,1 or 2 is computed **non-deterministically**:
    it depends on the *execution order* in the interleaved computation sequence which is ***arbitrary***
- Can we make program 3 behave similarly to programs 1 and 2?

Atomicity

- Interleaving is limited by the (smallest) units of a program that are executed atomically
- In Promela all ***statements are atomic***
  - Attention: expressions are statements in Promela!

```
if
:: a != 0 ->
        c = b / a
:: else ->
        c = b
fi
```

Introducing Atomicity

Program 2:

```
byte n=3;

active proctype P() {
        n=0;
        printf("Process P, n = %d\n", n)
}

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

Program 3:

```
byte n=3;

active proctype P() {
        n=0;
        printf("Process P, n = %d\n", n)
}

active proctype Q() {
        atomic { n=n+1; n=n+1; }
        printf("Process Q, n = %d\n", n)
}
```

More on Atomicity

- How to make ``a != 0 -> c = b / a'' atomic?

```
if
:: atomic { a != 0 ->
        c = b / a }
:: else ->
        c = b
fi
```

More on interference

```
byte n=0;

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

active proctype Q() {
  byte temp;
  temp=n+1;
  n = temp;
  printf("Process Q, n = %d\n", n)
}
```

Add
pre- and post-conditions
to P() and Q()

What is the final value of variable n?

Notation for sets of processes

- We can write the preceding more concisely

```
byte n=0;

active [2] proctype P() {
  byte temp;
  temp=n+1;
  n = temp;
  printf("Process P, n = %d\n", n)
}
```

What is the final range of n?

```
byte n = 0;

proctype P() {
  byte temp;
  byte i;
  i = 1;
  do
  :: i > 10 -> break
  :: else ->
    temp = n+1;
    n = temp;
    i = i+1
  od
}
```

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

  (_nr_pr == 1) ->
    printf("The value is %d\n", n)

}
```

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2)

Correctness: critical sections

- Processes are divided in ***critical*** sections (CS) and ***noncritical*** sections (NS)
- A process may halt in its NS, but not in its CS
- ***Correctness properties*** to be verified:
  - **Mutual exclusion (ME)**: At most one process is executing its CS at any time
  - **Absence of deadlock (AD)**: Whenever some processes attempt to enter their CS, one of them succeeds
  - **Absence of starvation (AS)**: Any process attempting to enter its CS eventually succeeds

Example: critical sections

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: printf("Noncritical section P\n");
     wantP = true;
     printf("Critical section P\n");
     wantP = false
  od
}

active proctype Q() {
  do
  :: printf("Noncritical section Q\n");
     wantQ = true;
     printf("Critical section Q\n");
     wantQ = false
  od
}
```

How can we verify ME?

Answer: use ghost variables

```
bool wantP = false, wantQ = false;
byte critical = 0;

active proctype P() {
 do
 :: printf("Noncritical section P\n");
    wantP = true;
    critical++;
    printf("Critical section P\n");
    assert (critical <= 1);
    critical--;
    wantP = false
 od
}

active proctype Q() {
 do
 :: printf("Noncritical section Q\n");
    wantQ = true;
    critical++;
    printf("Critical section Q\n");
    assert (critical <= 1);
    critical--;
    wantQ = false
 od
}
```

Today's lecture

- Concurrent programming
- Interleaving
- Interference and non-determinism
- Atomicity
- Synchronization
- Blocking
- Semaphores

Critical section with *busy-waiting*

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: printf("Noncritical section P\n");
     wantP = true;
     do
     :: !wantQ -> break
     :: else -> skip
     od;
     printf("Critical section P\n");
     wantP = false
  od
}

active proctype Q() {
  do
  :: printf("Noncritical section Q\n");
     wantQ = true;
     do
     :: !wantP -> break
     :: else -> skip
     od;
     printf("Critical section Q\n");
     wantQ = false
  od
}
```

Critical section with *blocking*

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: printf("Noncritical section P\n");
     wantP = true;
     do
     :: !wantQ -> break
     od;
     printf("Critical section P\n");
     wantP = false
  od
}

active proctype Q() {
  do
  :: printf("Noncritical section Q\n");
     wantQ = true;
     do
     :: !wantP -> break
     od;
     printf("Critical section Q\n");
     wantQ = false
  od
}
```

Critical section with *blocking*

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: printf("Noncritical section P\n");
     wantP = true;
     !wantQ;
     printf("Critical section P\n");
     wantP = false
  od
}

active proctype Q() {
  do
  :: printf("Noncritical section Q\n");
     wantQ = true;
     !wantP;
     printf("Critical section Q\n");
     wantQ = false
  od
}
```

Critical section with *blocking*

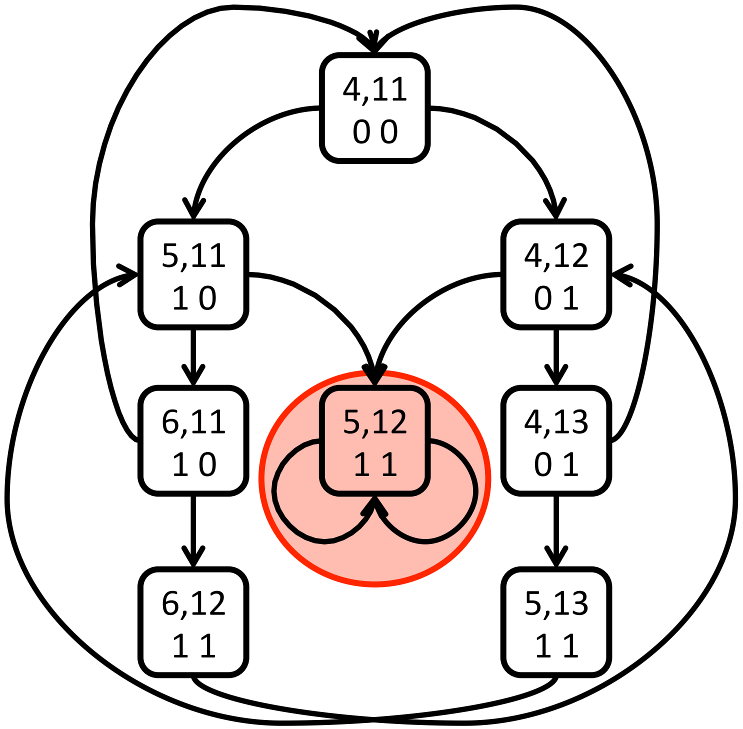
```
bool wantP = false, wantQ = false;

active proctype P() {
  do :: wantP = true;
        !wantQ;
        wantP = false
  od
}

active proctype Q() {
  do :: wantQ = true;
        !wantP;
        wantQ = false
  od
}
```

Explore reachable states, use the ***state transition diagram***:

1. Let $S = \{s_0\}$ where s_0 is the initial state; mark s_0 unexplored.
2. For each unexplored state $s \in S$, let t be a state that results from executing an executable statement in state s; if $t \notin S$, add t to S and mark it unexplored, mark s as explored.
3. Terminate when all states in S are marked explored.

Critical section with *blocking*

```
1.  bool wantP = false, wantQ = false;
2.
3.  active proctype P() {
4.    do :: wantP = true;
5.          !wantQ;
6.          wantP = false
7.    od
8.  }
9.
10. active proctype Q() {
11.   do :: wantQ = true;
12.         !wantP;
13.         wantQ = false
14.   od
15. }
```



Deadlock!

Critical section with *blocking*

```
bool wantP = false, wantQ = false;

active proctype P() {
  do :: wantP = true;
        !wantQ;
        wantP = false
  od
}

active proctype Q() {
  do :: wantQ = true;
        !wantP;
        wantQ = false
  od
}
```

What can we do?

Use atomic sequence of statements.

Where?

Critical section with *blocking*

```
bool wantP = false, wantQ = false;

active proctype P() {
  do :: atomic { !wantQ;
                 wantP = true };
        wantP = false
  od
}

active proctype Q() {
  do :: atomic { !wantP;
                 wantQ = true };
        wantQ = false
  od
}
```

Semaphores

- A ***semaphore*** is a construct for synchronizing concurrent programs
- It is a variable of type **byte** with two atomic operations:
  - wait(sem): *the operation is executable when the value of* sem *is positive; it decrements the value of* sem.
  - signal(sem): *the operation is always executable; it increments the value of* sem.

Critical section with *semaphore*

```
byte sem = 1;

active proctype P() {
  do ::
    printf("Noncritical section P\n");
    atomic { /* wait(sem) */
      sem > 0;
      sem--
    };
    printf("Critical section P\n");
    sem++ /* signal(sem) */
  od
}
```

```
active proctype Q() {
  do ::
    printf("Noncritical section Q\n");
    atomic { /* wait(sem) */
      sem > 0;
      sem--
    };
    printf("Critical section Q\n");
    sem++ /* signal(sem) */
  od
}
```

Analyse, improve and document:

```
bool turn, flag[2];
byte ncrit;

active [2] proctype user() {
  assert _pid == 0 || _pid == 1;
  again:
  flag[_pid] = 1;
  turn = _pid;
  flag[1 - _pid] == 0 || turn == 1 - _pid;
  ncrit++;
  assert ncrit == 1;
  ncrit--;
  flag[_pid] = 0;
  goto again
}
```