Contents

1 Petri-net 3
2 Transition Systems 4
3 At-Most-Once Property 5
4 Semaphore 6
5 Monitor 7
6 Invariants 8
   6.1 The Inductive Invariance Technique 8
   6.2 Semaphore Invariants 9
   6.3 Monitor Invariants 10
7 Live Temporal Logic (LTL) 11
8 CSP 12
9 Asynchronous Message Passing 13
10 Remote Procedure Call - RPC 14
11 Rendezvous 15
12 Fairness 16
13 Banker’s algorithm 17
1 Petri-net

Reference:
See [Løv05] page 1.

Comments:
• Remember to initialize the petri-net with tokens.
• An arc can only go from:
  – place to transition,
  – or transition to place.

Example:

The following petri net represents the following algorithm:

First A is executed then B and C are executed concurrently. After B and C finished their execution the process is repeated.

Figure 1: Example on the use of Petri nets.
2 Transition Systems

Reference:
See [Løv05] page 11.

Comments:

• Distinguish between transition graphs and transition diagrams.
  – Transition graphs show reachable states. (See [Løv05] page 17)
  – Transition diagrams model processes. (See [Løv05] page 13)

• All transitions (representing statements) must be atomic or satisfy At-Most-Once property:
  – If not atomic the statement must be subdivided into multiple transitions, which then are atomic!

Example:

```c
/* The following code is represented by the transition diagram to the right where "loop" is a shared variable modified by another process: */
process A;
repeat
  loop := 1;
  while loop != 5 do
    loop := loop + 1;
  od;
forever
```

Figure 2: Example on the use of transition diagrams.
3 At-Most-Once Property

References:
See [And00] page 52.

Comments:
• If a statement has at most one critical reference, then it satisfies At-Most-Once property and can be treated as being atomic.

Critical References
• If a statement reads a variable which is modified in another process, then it counts as one critical reference.

• If a statement modifies a variable which is accessed in another process, then it counts as one critical reference.

Example:

```cpp
/* The following code represents "another process", where x and y are shared variables: */
Process A;
repeat;
    x := 2; // x is now modified
    ta := y; // y is now read
forever

Process B;
repeat;
y := 1;
    /* We are modifying y, which is accessed in process A. So there is one critical reference in this statement*/
x := y;
    /* We are modifying x, which is accessed in process A. So there is one critical reference in this statement. Reading y does not constitute a critical reference as it is not modified in A.*/
tb := x + y;
    /* We are reading x, which is modified in process A. So there is one critical reference in this statement. Writing to tb is not a critical reference as it is a local variable.*/
x := x + tb;
    /* We are reading and writing to x, which is modified in process A. So there are two critical references in this statement.*/
forever
```

Figure 3: Example on counting critical references.
4 Semaphore

References:
See [And00] page 153.

Comments:
• The V-operation increments the semaphores atomically without delaying.
• The P-operation decrements the semaphore atomically if it has a positive value or it delays until it gets a positive value.
• Remember to give the semaphores initial values.

Example:

```
semA := 2; // initialization
semBC := 0;

Process X;
repeat;
    P(semA)
    A;
    V(semBC)
forever

Process Y;
repeat;
    P(semBC)
    B;
    V(semA)
forever

Process Z;
repeat;
    P(semBC)
    C;
    V(semA)
forever
```

The programs to the right represents the following program:
First A is executed then B or C are executed. After B or C has finished its execution the process is repeated.

Figure 4: Example on the use of semaphores.
5 Monitor

References:
See [And00] page 203.

Comments:
- All monitor operations are inherently mutually exclusive.
- The wait\( (q) \)-operation makes the thread wait on the queue, \( q \), and releases the monitor.
- The empty\( (q) \)-operation returns whether or not the queue, \( q \), is empty.
- The signal\( (q) \)-operation wakes up one thread from the queue, \( q \), if any threads are waiting.
- The signal\( _\text{all}(q) \)-operation wakes up all threads from the queue, \( q \), if any threads are waiting.

Signal and Continue semantics: When a thread calls signal or signal\( _\text{all} \), it continues its own execution.

Signal and Wait semantics: When a thread calls signal or signal\( _\text{all} \), executions transfers to the threads that are woken up.

Signal and Urgent Wait semantics: When a thread calls signal or signal\( _\text{all} \), executions transfers to the threads that are woken up and when they are done in the monitor then the thread calling signal has priority to continue execution.

Example:

```plaintext
The monitor to the right represents a resource-manager, where one can acquire a resource and place resources.

If a process tries to take a resource, when none is present, then it will wait on the condition queue.

NB! This is a naive implementation e.g. when 1000 processes are waiting on the queue, and a process adds just one resource, then 999 processes will be woken up just to go back to sleep. For a better version see Figure 6.

```
6 Invariants

Reference:
See [Løv05] page 21.

6.1 The Inductive Invariance Technique

Reference:
See [Løv05] page 26.

Comments:
- To make a proof using the inductive invariance technique you must:
  - Proof or argue that the invariant holds initially e.g. when starting.
  - Proof or argue that the invariant holds every time the relevant variables is changed.
- If asked whether or not a statement preserves an invariant, one must assume the invariant hold initially and then determine if it always holds after executing the statement.

Example:

This example will proof that for the program to the right the following invariant holds:

\[ I \triangleq j > 0 \]

The invariant holds initially as \( j \) is initialized to 1.

The only place \( j \) is changed is in the statement inside the loop, where \( j \) is incremented. This means that \( j \) increases its value monotonically, so the invariant also holds after executing of the this statement.
6.2 Semaphore Invariants

Reference:
See [Lov05] page 34.

Comments:
- \( #V(s) \) denotes the number of completed V-operations on a semaphore \( s \).
- \( #P(s) \) denotes the number of completed P-operations on a semaphore \( s \).
- It always holds that \( #P(s) \leq s_0 + #V(s) \), where \( s_0 \) is the initial value of the semaphore \( s \).

Example:

We will use semaphore invariants to prove that the program works as intended.
If one looks at the semaphore example in Figure 4, then we know that:

\[
\begin{align*}
#P(semA) & \leq 1 + #V(semA) \quad (1) \\
#P(semBC) & \leq #V(semBC) \quad (2)
\end{align*}
\]

We can see from process X, that the following invariant holds:

\[
#V(semBC) \leq #P(semA) \quad (3)
\]

In both process Y and Z, we see that:

\[
#V(semA) \leq #P(semBC) \quad (4)
\]

Combining (2), (3) and (4), we end up with:

\[
#V(semA) \leq #P(semA) \quad (5)
\]

Now we can see from (1) and (5) that:

\[
0 \leq #P(semA) - #V(semA) \leq 1 \quad (6)
\]

This means we will at most make one more \( P \)-operation than \( V \)-operations on \( semA \), e.g. we can never make two \( P(semA) \)-operations in a row. So by looking at the program we can conclude that every time we make an \( A \)-operation, we will make either an \( B \)- or \( C \)-operation. So now it is proven that the program does, what was actually intended.
6.3 Monitor Invariants

Reference:

See [Lov05] page 53.

Comments:

- To state monitor invariants the following notions may be used:
  - \( waiting(c) \), denotes the number of processes waiting on the condition queue \( c \).
  - \( woken(c) \), denotes the number of processes woken up from the condition queue \( c \) (only for Signal-Continue semantics).

Example:

For the monitor given in Figure 5 one can express the following invariant:

\[
waiting(q) > 0 \Rightarrow res = 0
\]

This invariant just says that if a process is waiting on the queue, then there are no resources.

Now for the modified version of the program on the right this invariant does not hold and needs to be modified:

\[
waiting(q) > 0 \land woken(q) = 0 \Rightarrow res = 0
\]

Now the invariant says that if a process is waiting on the queue and there are no awakened processes from the queue, then there are no resources left.

Figure 6: Example on the use of monitor invariants.
7 Live Temporal Logic (LTL)

Reference:
See [Løv05] page 37.

Comments:
- The most noticeable operators in live temporal logic are:
  - $\square P$, which states that the predicate $P$ always holds.
  - $\Diamond P$, which states that the predicate $P$ will hold eventually.
- It is also worth remembering:
  - $\square \Diamond P$, which states that the predicate $P$ holds infinitely often.
  - $\Diamond \square P$, which states that the predicate $P$ will eventually hold forever.
  - $P \rightarrow Q \triangleq \square (P \Rightarrow \Diamond Q)$, which states that when the predicate $P$ holds then $Q$ will eventually hold.

Example:

For the program on the right the following properties can be formulated using live temporal logic.

$$
\square (x > 0) \\
\Diamond (y = 1) \\
\square \Diamond (y = 0) \\
\Diamond \square (z = 10) \\
(y = 0) \rightarrow (y \neq 0)
$$

Process A

```
x := 1;
y := 1;
z := 1;
while (true)
  x := x + 1;
y := y + 1 mod 2;
  if (z != 10)
    z := z + 1;
```

Figure 7: Example on the use of live temporal logic.
8  CSP

References:
See [And00] page 320.

Comments:

• $P!(e_1, \ldots, e_n)$-operator, denotes that the process is “waiting” for process $P$ to receive the value of the expressions $e_1, \ldots, e_n$.

• $P?(x_1, \ldots, x_n)$-operator, denotes that the process is waiting to receive input from another process $P$. After reception of the input it will be present in the variables $x_1, \ldots, x_n$.

• $[]$-notation is used in $if$- and $do$-statements together with “guarded communication statements” (See [And00] page 323) and it is used to express that the process can choose non-deterministically between guards, which can succeed.

• Sending messages is done synchronously, so the sending process waits for the receiving process to receive a message.

Example:

The program to the right contains four processes. Process A and B both try to send a value to process D and will block until they can do so. Process C waits to receive input from process D.

Process D has a loop containing three guards, which says:

• If A is trying to send then it should be received and added to the value $r$.

• If B is trying to send then it should be received and subtracted from the value $r$.

• If C is waiting to receive input, then the value from $r$ should be sent.

If none of the guards can succeed then the loop will block. If more than one guard can succeed then it will choose non-deterministically between these guards.

```
process A{
    int x;
    do true -> D!x; od
}

process B{
    int y;
    do true -> D!y; od
}

process C{
    int z;
    do true -> D?z; od
}

process D{
    int r,t;
    do A?t -> r := r + t;
    [] B?t -> r := r - t;
    [] C!r -> skip;
    od
}
```

Figure 8: Example on the use of CSP.
9 Asynchronous Message Passing

References:
See [And00] page 296.

Comments:

- **chan ch(type1 id1, ..., type n id n)**, declares a channel, ch, for communicating messages, which contains the values, id, of type type. Note the identifiers id can be omitted in the declaration.

- **send ch(expr1, ..., expr n)**-operator, denotes a process sending a message over the channel, ch, which contains the value of the expressions expr.

- **receive ch(var1, ..., var n)**-operator, denotes a process waiting to receive a message over the channel, ch, where its contents will be saved in the variables var.

- Sending messages is done asynchronously, so the sending process does not wait for the receiving process to receive a message.

Example:

The program to the right contains three processes. Process A and B both send two values to process C through the channel from and then both wait to receive a message from channel to[0] and to[1] respectively.

Process C waits to receive two messages from the channel from, when both messages are received, it calculates the sum of the four values from the two messages. The sum is then first sent over the channel to[0], and then again over to[1].

Note that both A and B send messages over the channel from, which means that C may receive the messages in any order. However when C replies, it first sends a message over to[0], which is used by A only. Thus the reply is always first sent to A and then to B.

```plaintext
chan from( int, int);
chan to[2]( int);
process A{
  int x,y,z;
  while ( true ){
    send from( x,y);
    receive to[0]( z);
  }
}
process B{
  int x,y,z;
  while ( true ){
    send from( x,y);
    receive to[1]( z);
  }
}
process C{
  int a,b,c,d,z;
  while ( true ){
    receive from( a,b);
    receive from( c,d);
    z := a + b + c + d;
    send to[0]( z);
    send to[1]( z);
  }
}
```

Figure 9: Example on the use of asynchronous message passing.
10 Remote Procedure Call - RPC

References:
See [And00] page 362.

Comments:
- call mname.opname(arguments), states that a remote procedure call is made to the procedure opname on the module mname.
- RPC contains no synchronization on the server-side.
- Remember to declare procedures available for remote procedure calls explicitly using the op-keyword.

Example:

```
module resman
  // Declaring RPC procedures
  op take() returns int;
  op place(int);
body
  sem res = 0;
  proc take() returns int{
    P(res);
    return 1; // one resource
  }
  proc place(int i){
    for(int j=0; j < i; j++)
      V(res)
  }
end resman

process taker[i = 0 to n]{
  int res, total=0;
  while(true){
    res = call resman.take();
    total = total + res;
  }
}

process giver[i = 0 to m]{
  while(true)
    call resman.give(i);
}
```

The program to the right defines a resource manager like the one proposed in Figure 5 and 6 using RPC. It also defines two different kinds of calling processes, taker and giver, which use remote procedure calls on the module.

Note the calling processes will block execution until the remote procedure call has finished their own execution.

Figure 10: Example on the use of remote procedure call.
11 Rendezvous

References:
See [And00] page 373.

Comments:

• `in opname(formal identifiers) -> $S; ni`, makes a server process delay until a process calls the operation `opname` on the server. When the operation is called the statement list $S$ is then executed.
  
  - Rendezvous can use guarded communication statements, similar to those from CSP, using the `[]`-notion in a `in`-statement (See [And00] page 373).

• `?opname`, denotes the number of pending invocations of the procedure `opname`

• Remember to declare procedures available for rendezvous explicitly using the `op`-keyword.

Example:

```plaintext
module resman
  // Declaring rendezvous procedures
  op take() returns int;
  op place(int);

  body
  int r = 0; //resources
  process Manager{
    while(true){
      in take() returns t and r > 0
      -> r = r - 1; t = 1;
      [] place(int i) -> r = r + i;
    ni
    }
  }
end resman

process taker[i = 0 to n]{
  int res, total=0;
  while(true){
    res = call resman.take();
    total = total + res;
  }
}

process giver[i = 0 to m]{
  while(true)
    call resman.give(i);
}
```

The program to the right defines a resource manager like the one proposed in Figure 11 now using rendezvous.

NB! If the server can handle both types of invocations, then it will pick one non-deterministically. Note that the server will only handle an invocation of `take` if the number of resources are greater than 0. So if there are no resources, then the calling process will block until its call has been handled.

Figure 11: Example on the use of remote procedure call.
12 Fairness

References:
See [And00] page 74.

Comments:
- A scheduler or algorithm will,
  - eventually execute conditional atomic actions, when the delay condition becomes true and remains true, if it is weakly fair.
  - eventually execute conditional atomic actions, when the delay condition becomes true infinitely often, if it is strongly fair.

Example:

The program on the right contains three process and two shared variables $x$ and $y$. It is worth noting that $x$ increases monotonically, while $y$ alternates between 1 and 0 all the time.

Process B will execute its conditional statement, when $x$ is greater than 10. We can see that this will eventually happen and when it does it will always be greater than 10. Thus, this statement is ensured to be executed under weak fairness and therefore also strong fairness.

Process C will execute its conditional statement, when $y$ is equal to 0, which happens infinitely often. But $y$ does not stay 0, so the statement is not ensured to be executed under weak fairness, however because it is equal to 0 infinitely often it will execute under strong fairness.

Process D will execute its conditional statement, when $x$ is equal to 0, which never happens. The statement will therefore not be executed under both weak and strong fairness.

Figure 12: Example on weak and strong fairness.
13 Banker’s algorithm

References:
See [ASG91] page 195.

Comments:
• An allocation graph shows the distribution of resources among processes and their individual needs for finishing their tasks.
  – Allocations are drawn with solid arrows from resource to process.
  – Requests are drawn with solid arrows from process to resource.
  – Expected, but not yet requested, needs are drawn with dashed arrows from process to resource.
• A situation is safe if it is possible for all processes to finish their tasks in some order.
• Banker’s algorithm will only go from one safe state to another safe state.

Example:

We have two instances of resource A, one instance of resource B and two instances of resource C.
The allocation graph on the right represents the processes with the following needs, allocations and request:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>1 0 1 0 0 1</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

The situation is safe as $P_2$ can finish and then release all resources, and hereby letting $P_1$ finish as well.
Now if $P_1$ is granted the requested $C$-resource in the given state, then the state becomes unsafe as none of the processes are able to finish. Banker’s algorithm will therefore not grant such a request.

Figure 13: Example on the use of allocation graphs.
References

