First, let us clarify the terminology. Assume that a labelled transition system contains a transition \( S_i \xrightarrow{a} S_j \). Symbol \( a \) is often called *action*.

When we take into account the point where the action happens, we speak about *an event*. For example, if a process has an execution sequence \( abcaade \), then it contains action \( a \) and three different events with \( a \).

Sometimes it is difficult to separate events from each other. In these cases it is possible to use extra marks to denote the event, for example subscripts. Thus in the sequence \( abcaade \) we have events \( a_1, a_2 \) ja \( a_3 \).

In Lotos, an action is divided into two parts, *gate* and *data*. It is thought that a gate is like a socket and data, i.e. data packet, is sent and received through that socket.
Basic Lotos II

- Thus in full Lotos a could be $g!2?x : \text{Boolean}$, where $g$ is a gate, !2 means that 2 is sent synchronously through $g$ and the last part means that Boolean data, coming at the same time synchronously through $g$, is received into variable $x$.

- However, in basic Lotos we do not use data part at all. Thus it is same, if we speak about a gate or a data.

- Simplifying a little, we can say that basic Lotos is an algebraic way to describe labelled transition systems. It resembles CCS and CSP and it contains:
  - two ready-made Lotos processes, stop and exit,
  - mechanism how to call a process,
  - two ready-made actions:
    - $i$ or silent or internal action,
    - $\delta$ or successful termination action,
  - 9 operators.
A specifier can use the internal action \( i \) freely. On the contrary, \( \delta \) is used only when defining the behaviour of the operators. With the help of the operators and action names it is possible to construct all the processes (what can be constructed in Lotos). Next we define the operators.
Lotos has two processes that have been defined beforehand, **stop** and **exit**.  

**Stop**, inaction, is a stopped or empty process that does nothing.

**Exit** is a successful termination. It informs its environment of a successful termination by sending $\delta$ and then stops.

We define the behaviour of the operators formally with the help of *transition rules*. The rules give *operational semantics* to the operators. A rule has three parts:

- process expression $P$,
- action $a$ which can be activated in $P$,
- a new expression $Q$ which is got after $P$ has done $a$. 

We use the notation $P \xrightarrow{a} Q$.

Expression **stop** has no transition rules. Thus it can make no actions or transitions.

Process **exit** is defined by the rule

$$\text{exit} \xrightarrow{\delta} \text{stop}.$$
If $P$ is a process and $a$ an action, then we can construct a new process $a; P$. It makes first $s$ and continues then as $P$. The transition rule is simple:

$$a; P \xrightarrow{a} P.$$ 

It is not allowed to write $P; Q$ or $\delta; P$, where $P$ and $Q$ are Lotos processes. We can already specify simple system in Lotos:
Example 1. Let a transition system be as follows:

\[ S_1 \xrightarrow{a} S_2 \xrightarrow{b} S_3 \xrightarrow{c} S_4. \]

An equivalent Lotos process is

\[
\text{process } P[a,b,c] := a; b; c; \text{stop endproc}
\]

We see the formal definition of a Lotos process. The definition is started with the reserved word \texttt{process}. Then come the name of the process and the action or gate names in brackets. After \texttt{:=} the Lotos expression is written and the reserved word \texttt{endproc} is at the end.
Example 2. A cycle can be realized using recursion. Consider the transition system below:

![Transition System Diagram]

An equivalent process is

```plaintext
process Q[a,b] := a; b; a; Q[a,b] endproc
```
Example 3. One more example.

process R[a,b,c] := a; S[b,c]
where
  process S[b,c] := b;c; S[b,c] endproc
endproc

Usually it is not wise to draw first a labelled transition system and after this translate it mechanically into Lotos. It is better at once to design a Lotos process. The further operators make this task even more straightforward.
Hiding changes visible actions into internal actions. The syntax of the operator is as follows:

\[
\text{hide } a_1, a_2, \cdots, a_n \text{ in } P
\]

This process behaves as \( P \), but makes the action \( i \), when \( P \) makes some of the actions \( a_i \). The transition rules are easy to understand:

- If \( P \xrightarrow{a} Q \) and \( a \in \{a_1, \cdots, a_n\} \), then
  \[
  \text{hide } a_1, \cdots, a_n \text{ in } P \xrightarrow{i} \text{hide } a_1, \cdots, a_n \text{ in } Q.
  \]

- If \( P \xrightarrow{a} Q \) and \( a \not\in \{a_1, \cdots, a_n\} \), then
  \[
  \text{hide } a_1, \cdots, a_n \text{ in } P \xrightarrow{a} \text{hide } a_1, \cdots, a_n \text{ in } Q.
  \]
Example. If $R$ is as in the example 3 of the previous section, then process hide $b, c$ in $R$ as a transition system is
The symbol of the choice operator is '[]', for example $P[ ]Q$. The system composed by using the choice operator is non-deterministic in its initial state. The first event may be either the event of $P$ or $Q$. After the first event, the process continues as $P$ or $Q$ would continue. This can be expressed using transition rules:

- If $P \xrightarrow{a} P'$, then $P[ ]Q \xrightarrow{a} P'$.
- If $Q \xrightarrow{a} Q'$, then $P[ ]Q \xrightarrow{a} Q'$.

Now it is possible to describe any kind of transition systems.
Example 1. Let a transition system be as follows:

\[ P: \]

The corresponding Lotos expression is

\[
\text{process } P[a,b] := (a; \text{stop}) [] (b; i; \text{stop}) \text{ endproc}
\]
Example 2. A labelled transition system is now:

\( R: \)

\[
\begin{align*}
&1 \rightarrow a \rightarrow 4 \\
&1 \leftarrow b \leftarrow 2 \\
&1 \rightarrow a \rightarrow 3 \\
&1 \leftarrow b \leftarrow 4 \\
&2 \rightarrow a \rightarrow 3 \\
&2 \leftarrow b \leftarrow 4 \\
&3 \leftarrow c \leftarrow 4 \\
\end{align*}
\]
Choice IV

The corresponding Lotos expression is

\[
\text{process } R[a,b,c] := (a; b; R[a,b,c]) []
\]
\[
\text{ (b; S[a,c]) []}
\]
\[
\text{ (a; stop)}
\]

where

\[
\text{process } S[a,c] := c; a; S[a,c] \text{ endproc}
\]
\text{endproc}

We show how the process changes when we execute the actions b,c,a,c:

\[
(a; b; R[a,b,c]) [] (b; S[a,c]) [] (a; stop) \rightarrow^b\]
\[
c; a; S[a,c] \rightarrow^c\]
\[
a; S[a,c] \rightarrow^a\]
\[
c; a; S[a,c] \rightarrow^c\]
\[
a; S[a,c]
\]
So far we can create only sequential and non-deterministic processes. With the help of the parallel operator we can also model concurrent behaviours. If $P$ and $Q$ are processes, then the notation

$$P \ |[a_1, \cdots, a_n]| \ Q$$

means as follows:

- $P$ and $Q$ proceed concurrently.
- Actions $a_1, \cdots, a_n$ and $\delta$ are possible only, if both $P$ and $Q$ make them at the same time ($a_1, \cdots, a_n$ are synchronizing actions or gates).
- $P$ and $Q$ perform other actions independently of each other.
The semantics of the parallel operator is defined by the following rules. It is assumed that $a_i \neq i, \delta, i = 1, \cdots, n$.

- If $P \xrightarrow{a} P'$ and $a \notin \{\delta, a_1, \cdots, a_n\}$, then
  
  $$P \parallel [a_1, \cdots, a_n] \xrightarrow{a} P' \parallel [a_1, \cdots, a_n] \parallel Q.$$  

- If $Q \xrightarrow{a} Q'$ and $a \notin \{\delta, a_1, \cdots, a_n\}$, then
  
  $$P \parallel [a_1, \cdots, a_n] \xrightarrow{a} P \parallel [a_1, \cdots, a_n] \parallel Q'.$$  

- If $P \xrightarrow{a} P'$, $Q \xrightarrow{a} Q'$ and $a \in \{\delta, a_1, \cdots, a_n\}$, then
  
  $$P \parallel [a_1, \cdots, a_n] \xrightarrow{a} P' \parallel [a_1, \cdots, a_n] \parallel Q'.$$
If we applied the above rules, then also $P \parallel [a_1, \cdots, a_n] \parallel Q$ can be considered as a transition system, the global state graph or reachability graph of the system consisting of $P$ and $Q$. The next example shows this.
Parallel Operator IV

\[ P \mid [a, b] \mid Q : \]

- \( P1Q1 \) → \( c \) → \( P1Q2 \) → \( a \) → \( P2Q3 \)
- \( a \) → \( P2Q4 \)
- \( b \) → \( P3Q5 \)
If \([a_1, \cdots, a_n]\) consists of all the actions in \(P\) and \(Q\) \((\neq i)\), then we can use the notation

\[ P \parallel Q. \]

If \(P\) and \(Q\) are as the previous example, then \(P \parallel Q\) consists only of transitions:

\[
\begin{align*}
&\text{P1Q1} \xrightarrow{a} \text{P2Q4} \xrightarrow{b} \text{P3Q5} \\
&Q \text{ can not now to perform } c, \text{ because } c \text{ belongs to the synchronization set, but } P \text{ does not have it.}
\end{align*}
\]

If the synchronization set is empty, we can use the notation

\[ P \parallel\parallel Q. \]

This is called \textit{pure interleaving}, because \(P\) and \(Q\) can make all the actions alone, independently of the other process. If there are a lot of
non-synchronizing actions in a system, then the global state graph will often be very large. If we consider the previous examples, then the pure interleaving produces the following graph:
These examples show how concurrency is modelled. First of all, processes proceed independently of each other, interleaving their events.

In synchronization events, both processes perform the same action at the same time. A process cannot proceed before the other has reached the same phase where it can perform the action.

The synchronization is of the form of rendezvous. This kind of synchronous communication is suitable for applications that take place in the same machine or that are really such that processes must wait for other processes before they proceed.

Often the communication is asynchronous. It is possible to model these kind of systems also as systems with synchronous communication, but then it is not possible to detect all the mistakes. However, if a mistake is found in the synchronous version, then it is in asynchronous version as well.
But if we really want to analyse asynchronous versions, then we must model a channel as a separate process.

Before there was a lot of discussion about the interleaving. There are other approaches, so called true concurrency models. Lotos has, for example, this kind of semantics in addition to the interleaving semantics. True concurrency models seem to be quite complicated and that is why they are seldom applied in concrete verifications.
The notation for sequential composition is '>>'. Process $P >> Q$ acts as $P$ until $P$ terminates successfully, and continues after this as $Q$. The termination of $P$ is expressed using the internal action. Transitions rules are simple:

- If $P \xrightarrow{a} P'$ and $a \neq \delta$, then
  \[ P >> Q \xrightarrow{a} P' >> Q. \]

- If $P \xrightarrow{\delta} P'$, then
  \[ P >> Q \xrightarrow{i} Q. \]
Example.

\[
a; \text{exit} \gg \gg b; \text{exit} \quad \xrightarrow{a}
\]
\[
\text{exit} \gg b; \text{exit} \quad \xrightarrow{i}
\]
\[
b; \text{exit} \quad \xrightarrow{b}
\]
\[
\text{exit} \quad \xrightarrow{\delta}
\]
\]

stop
Disabling I

The notation for disabling is $P[>Q$. First, the system acts as $P$. At any time before $P$ has terminated successfully, $Q$ can start. If $Q$ starts, $P$ terminates, but not successfully. $Q$ cannot start, if $P$ has terminated successfully. Transition rules:

- If $P \xrightarrow{a} P'$ and $a \neq \delta$, then

  $$P[>Q \xrightarrow{a} P'[>Q.$$ 

- If $P \xrightarrow{\delta} P'$, then

  $$P[>Q \xrightarrow{\delta} P'.$$

- If $Q \xrightarrow{a} Q'$, then

  $$P[>Q \xrightarrow{a} Q'.$$
Example.

\[
\begin{align*}
  a; b; & \text{ exit } [ > r; \text{ stop } \rightarrow r \text{ stop} \\
  b; & \text{ exit } [ > r; \text{ stop } \rightarrow r \text{ stop} \\
\end{align*}
\]

\[
\begin{align*}
  \text{ exit } [ > r; \text{ stop } \rightarrow r \text{ stop} \\
  \delta \text{ stop}
\end{align*}
\]
The operator precedences are as follows:
action prefix > choice > parallel composition > disabling > enabling > hiding.
Thus

\[
\text{hide a in } a; P [ ] Q >> R || S [ > T]
\]
is the same as

\[
\text{hide a in } (((a; P)[ ]Q) >> ((R || S)[ > T])).
\]

Operators with the same precedence are grouped starting from the right, as in

\[
\]
Process Instantiation I

Calling processes is generally the same as the procedure call in programming languages, but in Lotos even this is defined formally. For this formal definition, we need one extra operator \textit{relabelling}. It is denoted by

\[ [b_1/a_1, b_2/a_2, \cdots, b_n/a_n]. \]

It means that the gates \( a_1, \cdots, a_n \) in the process are replaced by the gates \( b_1, \cdots, b_n \). Relabelling does not belong to Lotos, but it is used only in the definition of process instantiation. The semantics of the operator is defined by the following transition rules:

1. If \( P \xrightarrow{a} P' \) and \( a = a_i \in \{a_1, \cdots, a_n\} \), then
   \[
   P[b_1/a_1, \cdots, b_n/a_n] \xrightarrow{b_i} P'[b_1/a_1, \cdots, b_n/a_n].
   \]

2. If \( P \xrightarrow{a} P' \) and \( a \notin \{a_1, \cdots, a_n\} \), then
   \[
   P[b_1/a_1, \cdots, b_n/a_n] \xrightarrow{a} P'[b_1/a_1, \cdots, b_n/a_n].
   \]
With the help of this operator we can write the rules for process instantiation.

Assume that process $P$ is defined by the expression

$$\text{process } P[a_1, \ldots, a_n] := KL \text{ endproc}$$

where $KL$ is a Lotos process.

If there is a transition

$$KL[b_1/a_1, \ldots, b_n/a_n] \xrightarrow{a} Q$$

then there is the transition

$$P[b_1, \ldots, b_n] \xrightarrow{a} Q,$$

too.
This rule defines process instantiation. In most cases the instantiation behaves as expected. Sometimes one must, however, be careful. We should notice that the process instantiation is dynamic. Relabelings are done continuously during the execution of the process, not statically in the beginning. Consider for example the following process:

$$\text{process } P[a,b,c] := a; b; \text{stop} \mid [a] \mid a;c; \text{stop endproc.}$$

As a transition system it is of the following form:
a ; b ; stop | [a] | a ; c ; stop

\[\begin{array}{c}
\text{a} \\
\text{b} ; \text{stop} | [\text{a}] | \text{c} ; \text{stop}
\end{array}\]

b ; stop | [a] | c ; stop

stop | [a] | c ; stop

b ; stop | [a] | stop

stop | [a] | stop

b ; stop | [a] | stop
Consider the call $P[c, c, a]$. If the relabeling were done statically, the call would change $P$ into

$$c; c; \text{stop} \mid [c] \mid c; a; \text{stop}$$

and this is the same as the transition system $LTS2$:

$$c; c; \textbf{stop} \mid [c] \mid c; a; \textbf{stop}$$

$$\downarrow c$$

$$c; \textbf{stop} \mid [c] \mid a; \textbf{stop}$$

$$\downarrow a$$

$$c; \textbf{stop} \mid [c] \mid \text{stop}$$

The dynamical relabelling in Lotos means that we change action names, but we do not change arcs. We can see in the following diagram how the relabelling is done step by step.
\[ p[c, c, a] \]
\[ \downarrow c \]
\[ (b \; stop | [a] | c \; stop)[c/a, c/b, a/c] \]
\[ \downarrow c \quad \downarrow a \]
\[ \text{stop | [a] | c ; stop} \quad \text{b ; stop | [a] | stop} \]
\[ [c/a, c/b, a/c] \quad [c/a, c/b, a/c] \]
\[ \downarrow a \quad \downarrow c \]
\[ \text{stop | [a] stop} \]
\[ [c/a, c/b, a/c] \]
Exit and noexit

Usually exit or noexit is written after the parameter list of a process. Exit means that it is possible to connect another process to this process using sequential composition operator. Thus there must be a branch in the process which terminates successfully. Noexit says that there is no such branch and that is why sequential composition cannot be used (it is of no use).

**Example.**

process P[a,b]: exit := (a; exit) [] b; stop endproc
process Q[c,d]: noexit := (c; c; d; stop) [] d; c; stop endproc
We will write a complete basic Lotos specification. Our system consists of a producer process which sends two messages to a client process using a channel. The channel may lose one or both of the messages. The order of the messages in the channel cannot change. The client is not confused even if messages are lost in the channel, it takes all into account.
specification Producer_Consumer[pc1, pc2, cc1, cc2]: exit
   behavior
    (Producer[pc1, pc2] ||| Consumer[cc1, cc2])
    ||
    Channel[pc1, pc2, cc1, cc2]
process Producer[pc1, pc2]: exit :=
    pc1; pc2; exit
endproc

process Consumer[cc1, cc2]: exit :=
    cc1;
    (
        cc2;
        exit

        [] exit
    )

    [] exit

    cc2;
    exit

    [] exit endproc
process Channel[pc1, pc2, cc1, cc2]: exit :=
   pc1;
   (pc2;
      cc1;
      exit
   )
   []
   cc1;
   pc2;
   exit
   []
   i;
   pc2;
   exit
)
Specification of a Producer-Client System V

>>
(    
    cc2;
    exit
[]    
i;
    exit
)
endproc
endspec

17. marraskuuta 2014 41 / 59
The next example shows how it is possible to define a Lotos specification which generates an infinite transition system.

```latex
process Counter[is0, not0, inc, dec]: noexit :=
  (inc; C1[not0, inc, dec]
   >>
   Counter[is0, not0, inc, dec])
[]
(is0; Counter[is0, not0, inc, dec])
where
  process C1[not0, inc, dec]: exit :=
    (dec; exit)
[] (inc; C1[not0, inc, dec] >> C1[not0, inc, dec])
[] (not0; C1[not0, inc, dec])
endproc
endproc
```
As a transition system the process is of the following form:

\[
\begin{align*}
&\text{is0} \xrightarrow{i} C \\
&\quad \downarrow \quad \text{inc} \\
&\text{not0} \xrightarrow{i} C1 >> C \\
&\quad \downarrow \quad \text{inc} \\
&\text{not0} \xrightarrow{i} C1 >> C1 >> C \\
&\quad \downarrow \quad \text{inc} \\
&\text{not0} \xrightarrow{i} C1 >> C1 >> C1 >> C \\
&\quad \downarrow \quad \text{inc} \\
\end{align*}
\]
Let us write the system described 2.2 in basic Lotos. This client-server system is a good example where we will see how the parameter lists can be used in a flexible way. We assume that there are three clients in the system. It is enough to describe two clients, of which one models the starting client and the second the other clients. By changing parameters we can describe the whole system.
specification Client_Server_System[t1,t2,t3]: noexit behavior

hide cs1, cs2, cs3, sb1, sb2, sb3, bc1, bc2, bc3 in
     Server[cs1, cs2, cs3, sb1, sb2, sb3]
             |[cs1, cs2, cs3, sb1, sb2, sb3]|
              ( (Client1[bc1, cs1, t1, t2] |[bc1]| Buffer[sb1, bc1])
               |[t1, t2]|)
              ( (Client[bc2, cs2, t2, t3] |[bc2]| Buffer[sb2, bc2])
               |[t3]|)
              (Client[bc3, cs3, t3, t1] |[bc3]| Buffer[sb3, bc3]))
)
where

process Client1[bc, cs, t1, t2]: noexit :=
    cs; (bc; t2; t1; Client1[bc, cs, t1, t2] [])
    t2; bc; t1; Client1[bc, cs, t1, t2] )
endproc

process Client[bc, cs, t1, t2]: noexit :=
    t1; cs; (bc; t2; Client[bc, cs, t1, t2] [])
    t2; bc; Client[bc, cs, t1, t2] )
endproc
process Server[cs1, cs2, cs3, sb1, sb2, sb3]: noexit :=

    cs1; sb1; Server[cs1, cs2, cs3, sb1, sb2, sb3]

    cs2; sb2; Server[cs1, cs2, cs3, sb1, sb2, sb3]

    cs3; sb3; Server[cs1, cs2, cs3, sb1, sb2, sb3]
endproc

process Buffer[sb, bc] : noexit :=

    sb; bc; Buffer[sb, bc]
endproc

endspec
CADP (Caesar/Aldebaran) is a full Lotos software which has been developed mainly in France, but also in Canada and Spain. Its user interface is graphical and easy to use. The only more difficult task is to make the auxiliary files for data types. But in this course we do not use data types.

The software package is large and it knows many equivalences and representations. There is a separate guide, published on the web page of the course, where you can find all the details necessary to start the program and to use it for simple tasks.

One word of warning: Mistakes in Lotos programs are not always visible to the compiler. For example, if some gate name is missing in the gate list, then this can cause a completely different global state graph compared to the situation where all the gate names are in the list.
If the system informs about deadlocks, it is wise to analyse the situation, especially if the graph is smaller or larger than expected.
We show a typical scenario of a verification. We verify the version of the AB-protocol, where there are channels and messages get and give. Let us encode the protocol presented in 4.2. into basic Lotos:
specification AB[get, give]:
  noexit
behavior
hide d0, d1, dd0, dd1, a0, a1, aa0, aa1, st, rt, t in
  ( (Sender[get, d0, d1, aa0, aa1, st, rt, t]
    |[t, st, rt]|
    Timer[st, t, rt])
  |[]|
  Receiver[give, dd0, dd1, a0, a1]
)
  |[d0, d1, dd0, dd1, a0, a1, aa0, aa1]|

Channel[d0, dd0, d1, dd1, a0, aa0, a1, aa1]
where

process Sender[get,d0,d1,aa0,aa1,st,rt,t]: noexit :=
    get; Transmit[d0, aa0, aa1, st,rt,t] >>
    get; Transmit[d1,aa1, aa0, st,rt,t] >>
Sender[get,d0,d1,aa0,aa1,st,rt,t]
where
process Transmit[d0, aa0, aa1, st,rt,t]: exit :=
    aa0; Transmit[d0, aa0, aa1, st, rt, t]
    []
    d0; st; (t; Transmit[d0, aa0, aa1, st,rt,t]
        []
        aa0; rt; exit
    )
endproc
endproc
process Timer[st,t,rt] : noexit :=

   st;(t;Timer[st,t,rt] [] rt; Timer[st,t,rt])
endproc

process Receiver[give, dd0, dd1, a0, a1]: noexit :=

   dd0; give; Ack0[give, dd0, dd1, a0, a1] [] dd1; Ack1[give, dd0, dd1, a0, a1]

where
process Ack1[give, dd0, dd1, a0, a1]: noexit :=

  dd1; Ack1[give, dd0, dd1, a0, a1]

  a1; Receiver[give, dd0, dd1, a0, a1]
endproc

process Ack0[give, dd0, dd1, a0, a1]: noexit :=

  dd0; Ack0[give, dd0, dd1, a0, a1]

  a0; (dd0; Ack0[give, dd0, dd1, a0, a1]

    dd1; give; Ack1[give, dd0, dd1, a0, a1]
  )
endproc
endproc
process Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1] : noexit := 
    d0; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1] 
        [] 
        i; dd0; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1] 
    ) 
    [] 
    d1; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1] 
        [] 
        i; dd1; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1] 
    ) 
    [] 
    a0; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1] 
        [] 
        i; aa0; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1] 
    ) 
    []
a1; (i; Channel[d0, dd0, d1, dd1, a0, aa0, a1, aa1]

[]

i; aa1; Channel[d0, dd0, d1, dd1, a0, aa0, a1, aa1]

)
endproc
endspec
The global state graph consists now of 91 states and 177 transitions. Of the transitions, 156 are $\tau$-transitions. The graph does not contain deadlocks. Now we can compare the global state graph with the service. We do this so that we minimize the global state graph with respect to the bisimulation equivalence. The resulting graph is exactly the same as the service. Thus we have shown that the AB-protocol behaves correctly.