In Lotos, processes can send data to each other and data can be of different type. Transitions of a receiving process may depend on the received data.

Processes can handle data in Lotos as in normal programming languages. An essential difference is between Lotos and programming languages is that data types in Lotos are defined algebraically and data types can be used only through operations defined in this algebraic formalism.

In this course we do not define data types, but we use some types defined in CADP software and in examples.
In full Lotos, an action consists of a gate name and one or more send or receive action. For example, in the following expression a process sends an integer through the gate $g$ and receives at the same time a boolean value.

$$g \rightarrow !2 \ ?x: \text{Boolean}$$

Gate $g$ is called action before in basic Lotos. In full Lotos, a gate may exist alone as in basic Lotos, but it is more common that some data is sent or received through that gate.

The first question we must solve is how processes will synchronize with each other, if data is sent. In the following list there are all the different cases and the synchronization rules. For simplicity, we assume that there is only one data action in the gate.
If $g!E$ and $g!D$ try to synchronize, this can succeed, if $E$ and $D$ are of the same type and $\text{value}(E) = \text{value}(D)$. The term for this case is *value matching*.

If the other action is a sending action $g!E$ and the other a receiving action $g?x : T$, then the synchronization happens if $E$ is of type $T$. This case is called *value passing*.

If both actions are receiving actions $g?x : T$ and $g?y : U$, the synchronization can happen, if $U = T$. When the synchronization happens, both $x$ and $y$ get the same arbitrary value from $T$. This case is called *value generation*. In the global state graph there will be one branch for every element in $T$. So $T$ cannot be large!

The above rules must be used with the parallel operator. The following examples show, how transition systems are formed in the presence of data.
Let $P$ be:

$$P[a,b,c] := a \uparrow \text{true}; b \uparrow x:\text{Boolean}; \text{stop}
\begin{array}{c}
\text{[]} \\
a \downarrow \text{false}; c \uparrow y:\text{Boolean} \uparrow z:\text{Boolean}; \text{stop}
\end{array}
$$

endproc

It generates the following transition system:
Example 1. III

- We must take the following things into account when constructing transition systems.

- Transitions are of the form $g < e_1 e_2, \cdots, e_n >$ where $e$ is a sending action.

- If the original process has only a receiving action without any synchronization with another process, there must be a transition for every value. If the type is infinite, the there are an infinite number of transitions. This is of course not possible in practice with a concrete software.

- The synchronization rules can be used for different purposes. Consider for example the following cases.

- The first rule can be used for branching. For example the following process can receive a bolean value and it makes a branching depending of that value. This is sometimes easier than receiving the value to a variable and then using guards (see later).
process P[g, g1,g2] :=
  g!true; Q[g1]
endproc

  g!false; R[g2]
endproc

- The second rule is used for normal sending and receiving.
- The third rule can be used for generating a value. In this way it is possible to test how a protocol behaves with all the possible values.
Example 2. I

\[
\begin{align*}
\text{P}[a] & := a!\text{true}; \ P[a] \\
\text{Q}[a,b] & := a?x:\text{Boolean}; \ b; \ \text{stop} \\
\text{R}[a,c] & := c; \ a?y:\text{Boolean}; \ \text{stop} \\
\text{S}[a,b,c] & := \text{P}[a] | [a] | (\text{Q}[a,b] | [a] | \text{R}[a,c])
\end{align*}
\]

Let us construct a global state graph of $Q$ and $R$ when $a$ is a sole synchronizing action.
Example 2. II

Q[a,b] | [a] | R[a,c]

Q[a,b] | [a] | a?y:Boolean; stop

b; stop | [a] | stop

b; stop | [a] | stop

x=y=true

x=y=false
Notice the branching with an action $a$. The first branch corresponds the case that a generated value is true, the other the case that it is false. $P$ offers the value true. When combining $P$ with the global state graph, the branch with true is chosen.
Example 2. IV

PQR1 \rightarrow \text{c} \rightarrow \text{PQR2} \rightarrow \text{a} \land \text{true} \rightarrow \text{PQR3} \rightarrow \text{b} \rightarrow \text{PQR4}

\begin{align*}
\text{x} &= \text{y} = \text{true} \\
\text{x} &= \text{y} = \text{true}
\end{align*}
When defining a process in Lotos, it is possible to add a parameter list after the gate list. For example, in the following process there are gates $a$, $b$, $c$ and an integer and a boolean value as parameters. The process sends the boolean value and integer from the gate $a$. It receives a new boolean value and new integer from the gates $b$ and $c$. After this it starts from the beginning, but with different parameters.

```
process P[a,b,c](bit:Bool, n:Nat):noexit :=
  a !n !bit;
  b ?x:Bool;
  c ?y:Nat;
  P[a,b,c](x,y)
endproc
```
If a process finishes successfully, it can at the time send values to the process which continues. The values sent are written to the exit process,

$\text{exit}(E_1, \cdots, E_n)$.

This expression offers actions

$\delta!E_1 \cdots !E_n$.

Here $E_1, \cdots, E_n$ are expressions, whose values are sent forward by the finishing process.

Next we define a consumer process which receives one message consisting of a natural number and an even number and finishes successfully sending the even number forward at the same time.
process Consumer[a]: exit(Nat) :=
    a ?Msg:Nat ?Seq:Nat [(Seq mod 2) = 0];
    exit(Seq)
endproc

- The process uses a logical condition which must be true before the synchronization takes place.

- If the parallel operator is used and if the components terminate successfully, then they synchronize at the gate $\delta$. Thus all the components must offer exit with the same number of parameters and the parameters must obey the synchronization rules presented earlier.

- A parameter, which can get an arbitrary value from type $T$, is denoted by any $T$. For example, if $P_1$ and $P_2$ are combined with the parallel operator and
  - $P_1$ terminates exit(any Boolean, 1),
  - $P_2$ terminates exit(true, any Nat),
then when terminating,

- $P_1$ offers $\delta?\text{dummy} : \text{Bool!1}$,
- $P_2$ offers $\delta!\text{true?}\text{dummy} : \text{Nat}$.

These values agree with each other and the system offers

$$\delta!\text{true!1}.$$
Let us construct a process which receives messages with a sequence number and calculates the number of even and odd sequence numbers. At the end the process sends forward the number of messages which can be zero. The termination may come at any moment.
process Consumer[a](Odd_Num_Msg, Even_Num_Msg: Nat):
    exit(Nat,Nat):=
        a ?Msg: Nat ?n: Nat [(n mod 2) = 0];
        Consumer[a](Odd_Num_Msg, Even_Num_Msg+1)
        []
    a ?Msg: Nat ?n: Nat [(n mod 2) <> 0];
        Consumer[a](Odd_Num_Msg+1, Even_Num_Msg)
        []
    exit(any Nat, Odd_Num_Msg + Even_Num_Msg)
endproc
Example 3. III

When using sequential combination, parameters can be delivered with the `accept` expression:

\[ P_1 \gg \text{accept } x_1 : T_1, \cdots, x_n : T_n \text{ in } P_2 \]

If \( P_1 \) terminates successfully with exit, \( P_2 \) continues. \( P_2 \)'s variables \( x_1, \cdots, x_n \) get the values in exit as their initial values. These values must be of type \( T \).

Let us construct a consumer process which generates a value and sends it with a sequence number or terminates and forwards the number of messages.
process Producer[b](n: Nat): exit(Nat,Nat) :=
  (Generate_Ele >>
   accept Msg: Nat in
     b !Msg !n;
     Producer[b](n+1)
  )
[]
exit(n, any Nat)
where
  process Generate_Ele: exit(Nat) :=
    exit(any Nat)
  endproc
endproc
We have already used conditions to restrict actions after receiving a message. For example,

\[ a \ ?\text{Msg} : \text{Nat} \ ?\text{Seq} : \text{Nat} \ [(\text{Seq} \mod 2) = 0]; \]

The condition must be true before the synchronization can happen. This kind of a condition can be written also in the beginning of an expression and then we talk about guards. A simple example is the following:

\[
\begin{align*}
[x=2 \text{ or } x=3] \rightarrow g \ !x \ !0; \ \text{stop} \\
[] \rightarrow g!x \ !1; \ \text{stop}
\end{align*}
\]

We write still a consumer with the help of guards:
process Consumer[a] (Odd_Num_Msg, Even_Num_Msg: Nat):
    exit(Nat,Nat):=

    ( [ (n mod 2) = 0 ] -> Consumer[a](Odd_Num_Msg, Even_Num_Msg+1) )
    []
    [ (n mod 2) <> 0 ] -> Consumer[a](Odd_Num_Msg+1, Even_Num_Msg)

    exit(any Nat, Odd_Num_Msg + Even_Num_Msg)
endproc
Generalized choice I

Generalized choice is of the form

```plaintext
choice X:T [] B
```

which is the same as

```plaintext
[t_1/X]B [ ] [t_2/X]B [ ] · · · [ ] [t_n/X]B.
```

Here \( t_1, t_2, \cdots, t_n \) means all the values of type \( T \) and \( [t_i/X]B \) means that \( X \) is replaced by \( t_i \) in \( B \).

Infinite types are possible in principle. Even in practice they can be used as in the following expression:

```plaintext
choice x:Nat []
    [(x mod 2) = 0] -> g !x; stop
```
Generalized choice II

Many variables can get values in one choice expression:

\[
\text{choice } X_1 : T_1, \cdots, X_n : T_n \rightarrow P(X_1, \cdots, X_n). 
\]

Also choice between gates is possible:

\[
\text{choice } a \text{ in } [a_1, \cdots, a_n] \rightarrow P[a](\cdots). 
\]

Some of the calls

\[
P[a_1], \cdots, P[a_n] 
\]

will be chosen.

Next we write a Lotos process which sorts numbers. We do not use any sorting algorithm, but specify the order. Lotos mechanism takes care of the actual sorting (very slowly!). We need the type \texttt{NatList}, a list of natural numbers. In addition, we need operators \texttt{IsPermuted} and \texttt{IsOrdered}. The former is used in the form
IsPermuted(SortedList, UnsortedList)

and it returns true, if the second list is a permutation of the first list. The latter operator tests, if a list is in order or not.

input ?UndortedList: NatList;

choice SortedList: NatList []

[IsPermuted(SortedList, UnsortedList) and
  IsOrdered(SortedList)] ->

output !SortedList
Variable can be given values with LET expression:

```
gate ?x: Nat;
(let ext_val = 3*((x+3)*x-2)**(x*s),
int_val = (x*x)+2 in
  gate !ext_val !int_val;
  stop
[]
)gate ?y: Nat !int_val;
  stop
[]
gate !ext_val ?y:Nat;
  stop
)
```
Par expression is of the form

\[
\text{par } <\text{gateDeclarations}> <\text{parallelOp}>
\]

It is used to connect several processes by changing one or more gates. For example:

\[
\text{par } g_1 \text{ in } [a_1, a_2, a_3] \text{ parop, B}[g_1, h_1]
\]

where \( B[g_1, h_1] \) is a behavioural expression and \textit{parop} is a parallel operator like \texttt{||} or \texttt{|||}. The corresponding expression with parallel operators would be:

\[
B[a_1, h_1] \text{ parop } B[a_2, h_1] \text{ parop } B[a_3, h_1]
\]
In Lotos, a user can define his own data types.

Lotos uses the algebraic specification of data types which is based on ACT ONE language.

This kind of a method is formal and it avoids all the details connected to the implementation of data types.

Data types are abstract and data is an element in these types.
type Boolean is

sorts Bool

opns

  true, false: -> Bool

  not: Bool -> Bool

eqns

  ofsort Bool

  not(true) = false;
  not(false) = true;

endtype (* Boolean *)
First, the name of the type is announced and after this there is its definition.

In section sort the types used in the definition are given. In this case there is only one set $Bool$. The type Boolean consists of this basic set and operations.

The data type is defined with the help of operators. Operators are defined after the sorts. Operators are either constants or functions. If an operator is a function, its domain and range is given. With constants, only range is given.

The semantics of the operators is defined through equations. Here there is the difficulty: how to guarantee that the equations define just that type which has been thought.
type NaturalNumber is

    sorts Nat

    opns
        0: → Nat
        Succ: Nat → Nat

endtype
Example 3: Stack 1

type BooleanStack is Boolean

sorts Stack

opns
   empty: -> Stack
   push: Bool, Stack -> Stack
   top: Stack -> Bool
   pop: Stack -> Stack
   IsEmpty: Stack -> Bool

eqns
   for all b: Bool, s:Stack
     ofsort Bool
     top(push(b,s)) = b;
Example 3: Stack II

IsEmpty(empty) = true;
IsEmpty(push(b,s)) = false;

ofsort Stack
pop(push(b,s)) = s;
endtype
From a data definition it is possible to generate data elements. These are generated by applying operators to constants. For example, the definition of natural numbers generates the following elements:

\[ 0, \text{Succ}(0), \text{Succ}(<\text{Succ}(0)), \text{Succ}(<\text{Succ}(\text{Succ}(0))), \text{etc} \].

If equations are in use, not all the elements are different. The equations define an equivalence relation in the set of elements and the data type consists of equivalence classes rather than single elements.

For example, there are an infinite number of different elements in the data type Boolean:

\[ \{\text{not}^n(\text{true})| \ n \geq 1\} \cup \{\text{not}^n(\text{false})| \ n \geq 1\} \cup \{\text{true, false}\}. \]

However, there are only two equivalence classes, so the type consists of two elements [true] ja [false].
Term algebra II

In the case of Stack there are an infinite number of elements:
empty, push(true, empty), push(false, empty),
push(true, push(true, empty)),
push(false, push(true, empty)),
push(true, push(false, empty)),
push(false, push(false, empty)),

etc

So we have a set and operators which map elements to elements. We talk about term algebra, if we mean the set of elements, and about quotient term algebra, if we mean the equivalence classes.
Algebraic specification is such a general method that it is not possible to handle purely algorithmically. It may happen that it is an undecidable problem to check, if two expressions are equivalent or not. That is why it is necessary to make sure that a dat type in Lotos is manageable algorithmically. The following instructions are useful in this respect:

In Lotos, an equation is interpreted as a rewrite rule, i.e. how the left side expression can be transformed to the form in the right side. For example, in the type Boolean the rules are:

\[
\text{not(true)} \rightarrow \text{false}, \quad \text{not(false)} \rightarrow \text{true}.
\]

Thus the expression \(\text{not(not(not(true))))\) can be simplified mechanically using the rules:

\[
\text{not(not(not(true))))} \rightarrow \text{not(not(false)}) -> \\
\text{not(true)} ightarrow \text{false}.
\]
Data types in practice II

- Notice that the algebraic specification does not demand this interpretation for the equations, but an equations can be applied from right to left or from left to right. However, if a data type is written for Lotos software, the rewriting interpretation must be applied. Typically, the software works as follows.

- An arbitrary element, or term as it is called, is created by applying operations in some order to constants.

- Term must have a unique normal form that is got by applying suitable rewriting rules to the term until it is no more possible to simplify.

- A rewriting rule can be applied to any subterm of the term. Here we need pattern matching, which is easy with this kind of terms.

- If there is a variable in the right side of an equation, the same variable must be also in the left side.

- Write an equation in such a way that the right side is simpler than the left side.
The normal form should be independent of the order the operators are applied (confluence property).

Every term must have a normal form.

Especially, equations cannot create loops such as $x + y = y + x$.

If a data type follows the above rules, software can check if two terms are equivalent by transforming the terms into normal forms and comparing the normal forms.

With most distributed systems or protocols, it is not necessary to use complicated data types. Natural numbers, boolean values, different constants, lists and records are enough. They are easy to specify algebraically.
Let us define list a la Lisp. The elements in the list are natural numbers.

type NATURAL_LIST is NATURAL, BOOLEAN
  sorts LIST
  opns NIL: -> LIST
    CONS: NAT, LIST -> LIST
    CAR: LIST -> NAT
    CDR: LIST -> LIST
    ATOM: LIST -> BOOL
  eqns forall N:NAT, L:LISLIST
    ofsort LIST
      CAR(CONS(N,L)) = N;
      CDR(CONS(N,L)) = L;
    ofsort BOOL
      ATOM(NIL) = true;
      ATOM(CONS(N,L)) = false
endtype
In CÆSAR/Aldebaran software data types demand extra definitions.

Constants must be shown to the software by writing a comment (* constructor *) after a constant.

The other task is connected to the preliminary handling of data types. CADP software compiles an algebraic specification to C language construction. This happens automatically, but in certain cases a user must inform about restrictions to the software. These restrictions are written into a .t file. After this, data types are executed and a .h file is formed.

Unfortunately the documents are a little obscure at this point. It is best to check ready-made demos. There are a lot of them in the demos directory /opt/cadp/demos in the ”machine” melkki.