Next we present the AB protocol specification in full Lotos. The specification can be found in CADP demos. First we must define the alternating bit type, because it is now a parameter. This demands the following definitions in the file BITALT.lib:

library
    BOOLEAN, NATURAL
endlib

type BIT is
    sorts BIT
    opns 0 (*! constructor *),
        1 (*! constructor *) : -> BIT
    not : BIT -> BIT
    eqns

forall X, Y:BIT
ofsort BIT
    not (0) = 1;
    not (1) = 0;
endtype

type MESSAGES is NATURAL renamedby
    sortnames MSG for NAT
endtype

We have used existing definitions BOOLEAN and NATURAL when defining the type BIT. We present also the definitions of these type:
type Boolean is

sorts
  Bool (*! implemented by ADT_BOOL compared by ADT_CMP_BOOL
  enumerated by ADT_ENUM_BOOL printed by ADT_PRINT_BOOL *)

opns
  false (*! implemented by ADT_FALSE constructor *),
  true (*! implemented by ADT_TRUE constructor *) : -> Bool
  not (*! implemented by ADT_NOT *) : Bool -> Bool
  _and_ (*! implemented by ADT_AND *),
  _or_ (*! implemented by ADT_OR *),
  _xor_ (*! implemented by ADT_XOR *),
  _implies_ (*! implemented by ADT_IMPLIES *),
  _iff_ (*! implemented by ADT_IFF *),
\_eq\_  (*! implemented by ADT\_EQ\_BOOL *),
\_ne\_  (*! implemented by ADT\_NE\_BOOL *): Bool, Bool \rightarrow Bool

eqns
  \forall x, y : Bool
  \text{ofsort Bool}
  \begin{align*}
  \text{not (true)} &= \text{false}; \\
  \text{not (false)} &= \text{true}; \\
  x \text{ and true} &= x; \\
  x \text{ and false} &= \text{false}; \\
  x \text{ or true} &= \text{true}; \\
  x \text{ or false} &= x; \\
  x \text{ xor } y &= (x \text{ and not (y)}) \text{ or (y and not (x))}; \\
  x \text{ implies } y &= y \text{ or not (x)}; \\
  x \text{ iff } y &= (x \text{ implies } y) \text{ and (y implies x)}; \\
  x \text{ eq } y &= x \text{ iff } y; \\
  x \text{ ne } y &= x \text{ xor } y;
  \end{align*}
AB protocol in full Lotos V

endtype

type Natural is Boolean

  sorts Nat (*! implemented by
    ADT_NAT compared by ADT_CMP_NAT
    enumerated by
      ADT_ENUM_NAT printed by ADT_PRINT_NAT *)

opns 0 (*! implemented by ADT_N0 constructor *),
  1 (*! implemented by ADT_N1 *),
  2 (*! implemented by ADT_N2 *),
  3 (*! implemented by ADT_N3 *),
  4 (*! implemented by ADT_N4 *),
  5 (*! implemented by ADT_N5 *),
  6 (*! implemented by ADT_N6 *),
  7 (*! implemented by ADT_N7 *),
AB protocol in full Lotos VI

8 (*! implementedby ADT_N8 *),
9 (*! implementedby ADT_N9 *): -> Nat
Succ (*! implementedby ADT_SUCC constructor *):
    Nat -> Nat
_+_ (*! implementedby ADT_PLUS *),
_*_* (*! implementedby ADT_MULT *),
_**_* (*! implementedby ADT_POWER *),
_-_- (*! implementedby ADT_MINUS *),
_div_ (*! implementedby ADT_DIV *),
_mod_ (*! implementedby ADT_MOD *):
    Nat, Nat -> Nat
_eq_ (*! implementedby ADT_EQ_NAT *),
_ne_ (*! implementedby ADT_NE_NAT *),
_lt_ (*! implementedby ADT_LT_NAT *),
_le_ (*! implementedby ADT_LE_NAT *),
_gt_ (*! implementedby ADT_GT_NAT *),
AB protocol in full Lotos VII

\[\begin{align*}
\_\text{ge}_\_ (\text{implementedby ADT\_GE\_NAT}), \\
\_\text{==}_\_ (\text{implementedby ADT\_EQ\_BIS\_NAT}), \\
\_\langle\rangle\_ (\text{implementedby ADT\_NE\_BIS\_NAT}), \\
\_\text{<_}\_ (\text{implementedby ADT\_LT\_BIS\_NAT}), \\
\_\text{<=}\_ (\text{implementedby ADT\_LE\_BIS\_NAT}), \\
\_\text{>_}\_ (\text{implementedby ADT\_GT\_BIS\_NAT}), \\
\_\text{>=}\_ (\text{implementedby ADT\_GE\_BIS\_NAT}): \\
&\text{Nat, Nat} \to \text{Bool} \\
\text{min} (\text{implementedby ADT\_MIN}), \\
\text{max} (\text{implementedby ADT\_MAX}), \\
\text{gcd} (\text{implementedby ADT\_GCD}), \\
\text{scm} (\text{implementedby ADT\_SCM}): \\
&\text{Nat, Nat} \to \text{Nat}
\end{align*}\]

eqns

forall m, n : \text{Nat}

ofsort \text{Nat}
AB protocol in full Lotos VIII

1 = Succ (0);
2 = Succ (1);
3 = Succ (2);
4 = Succ (3);
5 = Succ (4);
6 = Succ (5);
7 = Succ (6);
8 = Succ (7);
9 = Succ (8);

ofsort Nat
m + 0 = m;
m + Succ(n) = Succ(m) + n;

ofsort Nat
m * 0 = 0;
m * Succ(n) = m + (m * n);

ofsort Nat
\[m \times 0 = \text{Succ}(0)\];
\[m \times \text{Succ}(n) = m \times (m \times \text{Succ}(n))\]
ofsort Nat
\[m - 0 = m;\]
\[\text{Succ}(m) - \text{Succ}(n) = m - n;\]
ofsort Nat
\[n \neq 0, \ m \lt n \Rightarrow m \div n = 0;\]
\[n \neq 0, \ m \geq n \Rightarrow m \div n = 1 + ((m - n) \div n);\]
ofsort Nat
\[n \neq 0, \ m \lt n \Rightarrow m \mod n = m;\]
\[n \neq 0, \ m \geq n \Rightarrow m \mod n = ((m - n) \mod n);\]
ofsort Bool
\[0 \text{ eq } 0 = \text{true};\]
\[0 \text{ eq } \text{Succ}(n) = \text{false};\]
AB protocol in full Lotos X

\[
\begin{align*}
\text{Succ} (m) \text{ eq } 0 & = \text{ false}; \\
\text{Succ} (m) \text{ eq } \text{Succ} (n) & = m \text{ eq } n; \\
ofsort \text{ Bool} \\
m \text{ ne } n & = \text{ not } (m \text{ eq } n); \\
ofsort \text{ Bool} \\
0 \text{ lt } 0 & = \text{ false}; \\
0 \text{ lt } \text{Succ} (n) & = \text{ true}; \\
\text{Succ} (n) \text{ lt } 0 & = \text{ false}; \\
\text{Succ} (m) \text{ lt } \text{Succ} (n) & = m \text{ lt } n; \\
ofsort \text{ Bool} \\
m \text{ le } n & = (m \text{ lt } n) \text{ or } (m \text{ eq } n); \\
ofsort \text{ Bool} \\
m \text{ ge } n & = \text{ not } (m \text{ lt } n); \\
ofsort \text{ Bool} \\
m \text{ gt } n & = \text{ not } (m \text{ le } n);
\end{align*}
\]
ofsort Bool
  m == n = m eq n;
  m <> n = m ne n;
  m <= n = m le n;
  m < n = m lt n;
  m > n = m gt n;
  m >= n = m ge n;

ofsort Nat
  m le n => min (m, n) = m;
  m gt n => min (m, n) = n;

ofsort Nat
  m ge n => max (m, n) = m;
  m lt n => max (m, n) = n;

ofsort Nat
  m eq n, m ne 0 => gcd (m, n) = m;
  m lt n, m ne 0 => gcd (m, n) =
gcd (m, n - m);
m gt n, n ne 0 => gcd (m, n) =
gcd (m - n, n);

ofsort Nat

scm (m, n) = (m * n) div gcd (m, n);

endtype

The type BOOLEAN seems simple, but NATURAL can confuse at first. Natural numbers are represented by expressions succ(succ(succ(\ldots(0)\ldots)). So the numbers are represented by expressions with the exception of numbers 0, \ldots, 9, which have a normal representation.

Caesar-adt, which handles data types in CADP, needs information about restrictions which shrink the number of possible elements. In this case, the
restrictions are connected to the data part of messages which are natural numbers from 0 to 4. Restrictions are written in BITALT.t file:

```c
#define CAESAR_AD T_EXPERT_T 4.4

/* This file restricts the message numbers to the integer range 0..4 */

#define ADT_ENUM_NAT(CAESAR_AD T_0) for ((CAESAR_AD T_0) = 0; (CAESAR_AD T_0) < 5; ++(CAESAR_AD T_0))
```
As we see, the definitions are quite cryptic. That is why we do not write our own data types during this course, but use existing data type in the CADP software.

Data types take the most part of the specification. The behavioural part, on the other hand, is not very long:

```plaintext
specification ALTERNATING_BIT_PROTOCOL [PUT, GET] : noexit
library BITALT endlib

behaviour

    hide SDT, RDT, RDTe, RACK, SACK, SACKe in
    (
        (}
```
AB protocol in full Lotos XV

TRANSMITTER [PUT, SDT, SACK, SACKe] (0 of BIT)
|||
RECEIVER [GET, RDT, RDTe, RACK] (0 of BIT)
)
|[SDT, RDT, RDTe, RACK, SACK, SACKe]| |
( 
  MEDIUM1 [SDT, RDT, RDTe]
|||
  MEDIUM2 [RACK, SACK, SACKe]
)
)

where

(*--------------------------------------------------------------

November 21, 2014 15 / 57

(*)
process TRANSMITTER [PUT, SDT, SACK, SACKe] (B:BIT) :
    noexit :=
        PUT ?M:MSG; (* acquisition d’un message *)
    TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
where
process TRANSMIT [PUT, SDT, SACK, SACKe] (B:BIT, M:MSG) noexit :=
    SDT !M !B; (* emission du message *)
    SACK !B; (* bit de controle correct *)
    TRANSMITTER [PUT, SDT, SACK, SACKe] (not (B)) []
    SACK !(not (B)); (* bit de controle incorrect => reemission *)
    TRANSMIT [PUT, SDT, SACK, SACKe] (B, M) []
SACKe;  (* indication de perte => reemission *)
TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
[]
i;  (* timeout => reemission *)
TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
)
endproc
endproc

process RECEIVER [GET, RDT, RDTe, RACK] (B:BIT) :
noexit :=
    RDT ?M:MSG !B;  (* bit de controle correct *)
    GET !M;  (* livraison du message *)
RACK !B;  (* envoi d’un acquittement correct *)

RECEIVER [GET, RDT, RDTe, RACK] (not (B))

[]

RDT ?M:MSG !(not (B));
(* bit de controle incorrect => *)

RACK !(not (B));
(* envoi d’un acquittement incorrect *)

RECEIVER [GET, RDT, RDTe, RACK] (B)

[]

RDTe;
(* indication de perte => *)

RACK !(not (B));
(* envoi d’un acquittement incorrect *)

RECEIVER [GET, RDT, RDTe, RACK] (B)

[]

i;
(* timeout => *)
RACK !(not (B));
   (* envoi d’un acquittement incorrect *)
   RECEIVER [GET, RDT, RDTe, RACK] (B)
endproc

(*----------------------------------------------------------*)

process MEDIUM1 [SDT, RDT, RDTe] : noexit :=
  SDT ?M:MSG ?B:BIT;
   (* reception d’un message *)
  (RD
    RDT !M !B;   (* transmission correcte *)
    MEDIUM1 [SDT, RDT, RDTe]
  []
    RDTe;  (* perte avec indication *)
    MEDIUM1 [SDT, RDT, RDTe]
AB protocol in full Lotos XX

[]
i; (* perte silencieuse *)
MEDIUM1 [SDT, RDT, RDTe]
)
endproc

(*---------------------------------------------------------------

process MEDIUM2 [RACK, SACK, SACKe] : noexit :=
RACK ?B:BIT;
(* reception d’un acquittement *)
(
SACK !B;  (* transmission correcte *)
   MEDIUM2 [RACK, SACK, SACKe]
[]
SACKe;  (* perte avec indication *)

(*---------------------------------------------------------------*)
MEDIUM2 [RACK, SACK, SACKe]
[]
i; (* perte silencieuse *)
MEDIUM2 [RACK, SACK, SACKe]
)
endproc

endspec
As the second example we have a different type of distributed system. We do not consider communication protocols, but distributed algorithms, in this mutual exclusion algorithms. First we show a couple of incorrect algorithms, whose mistakes are revealed in Lotos analysis. At the end we analyse a correct algorithm. All solutions are purely based on algorithms, hardware features (semaphores, interruptions) are not used.
Solution 1

```plaintext
var DOOR: (OPEN, CLOSED); (* shared variable *)

DOOR := OPEN;

process 1

repeat (* continue testing *)
until DOOR = OPEN;

DOOR := CLOSED;
    go to critical area;
    exit critical area;

DOOR := OPEN;
```

process 2
  repeat (* continue testing *)
  UNTIL DOOR = OPEN;

  DOOR := CLOSED;
  go to critical area;
  exit critical area;
  DOOR := OPEN;

This distributed algorithm uses a shared variable DOOR which can get only two values. If the value of the variable is OPEN, the process can move to the critical area, otherwise not.
The solution is not working, because the processes may read the shared variable at the same time which causes both of them to move to the critical area. Anyway, we write the algorithm in Lotos and find that mistake automatically.

One problem which must solved when writing the Lotos program is the shared variable. Lotos processes cannot transmit information through a shared variable, so we must represent it as a process.

Process DOOR communicates with the environment, i.e. with processes 1 and 2, through the gate $D$. The value of the shared variable is now the value the parameter $VAL_D$ in the process DOOR.

If a process wants to read the value, it synchronizes itself at the gate $D$ with action $D!\text{READ } ?VAL:\text{Bool}$. After the synchronization, the variable $VAL$ gets the value of $VAL_D$ and DOOR returns to its initial state.
If a process wants to change the value of $VAL_D$, the process synchronizes with the action $D \ \text{!WRITE} \ \text{!VAL}$ to DOOR’s action $D \ \text{!WRITE} \ \text{?VAL:Bool}$. After this, the process has variable DOOR’s value in its variable $VAL$ and the process DOOR starts from the beginning, but now with the parameter value $VAL$.

specification mutex1[enter1, exit1, enter2, exit2]: noexit

library
    BOOLEAN
endlib

type COMMAND is
Solution 1 V

sorts COMMAND

opns

   READ (*! constructor *),
   WRITE (*! constructor *) : ->COMMAND

endtype

behavior

hide NCS0, NCS1, D in

(  
P[NCS0, D, enter1, exit1]
   |||
P[NCS1, D, enter2, exit2]
)
|D|
DOOR[D](true)

where

process P[NCS, D, entern, exitn]:noexit :=

NCS;
P_AUX[NCS, D, entern, exitn]
endproc
process P_AUX[NCS, D, entern, exitn]:noexit :=

D !READ ?VAL_D:Bool;

(VAL_D] ->
    D !WRITE !false;
    entern;
    exitn;
    D !WRITE !true;
    P[NCS, D, entern, exitn]

[]

[not(VAL_D)]->
Solution 1 VIII

\[
P_{\text{AUX}}[\text{NCS, D, enter, exit}] \\
\]
endproc

process DOOR[D](VAL_D:Boolean):noexit :=

D !READ !VAL_D;
  DOOR[D](VAL_D)
[]
D !WRITE ?VAL:Boolean;
  DOOR[D](VAL)
endproc

endspec
The global state graph is quite large, 94 states and 188 transitions. It can be minimized with respect to the trace equivalence when the following graph is obtained:
We can see that both process can go to the critical area at the same time. Thus the algorithm is incorrect. This can also be seen automatically by giving the service:

Clearly, the service is not trace equivalent with the algorithm.
In the next version we do not use a shared variable, but both processes have their own variables. Anyway, the processes can read each other’s variable.
Algoritmi on seuraava:

```plaintext
var PROCESS1, PROCESS2 : (INSIDE, OUTSIDE);

PROCESS1 := OUTSIDE;

(*process 1*)

    PROCESS1 := INSIDE;
    repeat
    until PROCESS2 = OUTSIDE;
    enter1;
    exit1;
    PROCESS1 := OUTSIDE;
```
PROCESS2 := OUTSIDE;

(*process 2*)

PROCESS2 := INSIDE;
repeat
until PROCESS1 = OUTSIDE;
enter2;
exit2;
PROCESS2 := OUTSIDE;
This solution leads to a deadlock, if both processes make \texttt{PROCESS := INSIDE} at the same time. Let us write the algorithm in Lotos and check, how the mistake is revealed.

Local variables present now a problem. A local variable could be a parameter in a Lotos process. The process itself can change the value of the parameter with \texttt{LET} expression and the other process can read the value through some gate.

One problem is that a process should be able read the variable at any moment. That is why it is easier to model a local variable as a process. Only one can change the value but both can read the value. Thus we need two gates.
specification mutex2[enter1, exit1, enter2, exit2]: noexit

library
  BOOLEAN
endlib

type COMMAND is
  sorts COMMAND
  opns
    READ (*! constructor *),
    WRITE (*! constructor *) : -> COMMAND
endtype
behavior

hide NCS1, NCS2, G1, G2 in

(P[NCS1, enter1, exit1, G1, G2]
  |||
  P[NCS2, enter2, exit2, G2, G1])

P(G1, G2)

(Proc[G1](false)
  |||
  Proc[G2](false))
where
process P[NCS, entern, exitn, G1, G2]:noexit :=

    NCS;
    G1 !WRITE !true;
    P_AUX[NCS, entern, exitn, G1, G2]
endproc
process P_AUX[NCS, entern, exitn, G1, G2]:noexit :=

G2 !READ ?VAL_P2:Bool;
(
  [VAL_P2] -> P_AUX[NCS, entern, exitn, G1, G2]
  []
  [not(VAL_P2)] ->
    entern;
    exitn;
    G1 !WRITE ! false;
    P[NCS, entern, exitn, G1, G2]
)

endproc
process PROC[G](VAL_PROC:Boolean):noexit :=

G !READ !VAL_PROC;
PROC[G](VAL_PROC)
[]
G !WRITE $?VAL:Boolean;
PROC[G](VAL)
endproc

endspec
When the global state graph is constructed, it has no deadlocks, because a process can continue testing its local variable even if its value never changes. Neither trace equivalence reveals problem. Let us try the minimization with the weak bisimulation equivalence:

Now the deadlock is visible, because minimizing with respect to the weak bisimulation equivalence changes live locks with only internal actions to deadlocks.
The first algorithmic solution was invented by Dekker. The solution is the combination of the two previous attempts. So processes have their own variables and in addition there is a shared variable.
var PROCESS1, PROCESS2: (INSIDE, OUTSIDE);
TURN: 1..2; (* jaettu muuttuja *)

TURN := 1;
PROCESS1 := OUTSIDE;

(* prosessi 1 *)

PROCESS1 := INSIDE;
if PROCESS2 = INSIDE then
  if TURN = 2 then
    PROCESS1 := OUTSIDE;
    repeat until TURN = 1;
    PROCESS1 := INSIDE;
  end if;
end if;
repeat
    until PROCESS2 = OUTSIDE;
end if;

critical section;

TURN := 2;
PROCESS1 := OUTSIDE;
Dekker's algorithm IV

PROCESS2 := OUTSIDE;

(* prosessi 2 *)

PROCESS2 := INSIDE;
if PROCESS1 = INSIDE then
  if TURN = 1 then
    PROCESS2 := OUTSIDE;
    repeat until TURN = 2;
    PROCESS2 := INSIDE;
  end if;
  repeat
    until PROCESS1 = OUTSIDE;
end if;
critical section;

TURN := 1;
PROCESS2 := OUTSIDE;

In the Lotos description, the shared variable is represented as a process. The local variables PROCESS1 and PROCESS2 are represented also as processes, but their names are now FLAG[F0] and FLAG[F1].
specification DEKKER [NCS0, CS0, NCS1, CS1, F0, F1, T] :
noexit

library
   BOOLEAN,
   NATURAL
endlib

type COMMAND is
   sorts COMMAND
   opns
      READ (*! constructor *),
      WRITE (*! constructor *) : -> COMMAND
endtype
Dekker’s algorithm VII

behaviour

hide NCS0, NCS1, F0, F1, T in

(  
P [NCS0, enter1, exit1, F0, F1, T] (0)  
|||  
P [NCS1, enter2, exit2, F1, F0, T] (1)  
)

|[F0, F1, T]|  

(  
FLAG [F0] (false)  
|||  
FLAG [F1] (false)  
|||  
TURN [T] (0)
Dekker’s algorithm VIII

)

where
Dekker’s algorithm IX

process P [NCS, entern, exitn, FJ, FI, T] (J : Nat) :
    noexit :=
        NCS; (* non critical section *)
        FJ !WRITE !true;
        P_AUX_1 [NCS, entern, exitn, FJ, FI, T] (J)
endproc
process P_AUX_1 [NCS, entern, exitn, FJ, FI, T]
(J : Nat) : noexit :=
    FI !READ ?VAL_FI:Bool;
    (]
        [VAL_FI] ->
            T !READ ?VAL_T:Nat;
            (]
                [VAL_T <> J] ->
                    FJ !WRITE !false;
                    P_AUX_2 [NCS, entern, exitn, FJ, FI, T] (J)
            []
        [VAL_T == J] ->
            P_AUX_1 [NCS, entern, exitn, FJ, FI, T] (J)
    )
Dekker's algorithm XI

[]
[not (VAL_FI)] ->
    entern; (* critical section *)
    exitn;
    T !WRITE !(J + 1) mod 2;
    FJ !WRITE !false;
    P [NCS, entern, exitn, FJ, FI, T] (J)
endproc
Dekker’s algorithm XII

process P_AUX_2 [NCS, entern, exitn, FJ, FI, T] (J : Nat) :
noexit :=

T !READ ?VAL_T:Nat;
( [VAL_T <> J] ->
   P_AUX_2 [NCS, entern, exitn, FJ, FI, T] (J)
) [ ]
[VAL_T == J] ->
   FJ !WRITE !true;
   P_AUX_1 [NCS, entern, exitn, FJ, FI, T] (J)
)
endproc
process FLAG [F] (VAL_FLAG : BOOL) : noexit :=
  F !READ !VAL_FLAG;
  FLAG [F] (VAL_FLAG)
[]
  F !WRITE ?VAL:BOOL;
  FLAG [F] (VAL)
endproc

process TURN [T] (VAL_TURN : NAT) : noexit :=
  T !READ !VAL_TURN;
  TURN [T] (VAL_TURN)
[]
  T !WRITE ?VAL:NAT;
  TURN [T] (VAL)
Dekker’s algorithm XIV

endproc

endspec

The global state graph is quite large, 1138 states and 2276 transitions. When it is minimized with respect to weak bisimulation equivalence, the result is the following graph:
We notice that there are no deadlocks. Moreover, the minimized graph is trace equivalent with the service. However, it is not weakly bisimulation equivalent with the service, but this is not a requirement. Thus it depends on the case what kind of equivalences are used.