Σχεδιασμός Πρωτοκόλλων

Χειμερινό εξάμηνο

Περίληψη

- Η γλώσσα PROMELA
- Processes – channels - variables
- Executability of statements
- Modeling Timeouts
- Παραδείγματα

SPIN & PROMELA

- http://spinroot.com/
- Manuals
- Οδηγίες εγκατάστασης σε Windows-UNIX
- Papers
- Παραδείγματα

PROMELA

- Γλώσσα για την προδιαγραφή (specification) και την επαλήθευση (verification) πρωτοκόλλων
- Μας ενδιαφέρει η πληρότητα και η λογική ορθότητα των πρωτοκόλλων
- Μοντέλο επιβεβαίωσης (validation model) → μερική περιγραφή (βάρος στους διαδικαστικούς κανόνες)
- Δεν μας ενδιαφέρουν λεπτομέρειες υλοποίησης (π.χ., ακριβής μορφοποίηση)
- Μπορεί όμως να περιλαμβάνει α) υποθέσεις για τη συμπεριφορά του περιβάλλοντος, β) απαιτήσεις ορθότητας

Validation Models

- Διαφορά με μοντέλο υλοποίησης?
- Η διαφορά στο μοντέλο και στο κτίριο που φτιάχνει ένας πολιτικός μηχανικός → έλεγχος ότι οι σχεδιαστικές αρχές είναι σωστές
- Η PROMELA μοιάζει αρκετά με τη C

Τύποι αντικειμένων

- Διεργασίες (καθολικά αντικείμενα)
- Κανάλια μηνυμάτων (καθολικά ή τοπικά σε διεργασία)
- Μεταβλητές (καθολικά ή τοπικά σε διεργασία)

Διεργασίες

proctype A() { byte state; state = 3 }

- The process type is named A. The body of the declaration, enclosed in parentheses, consists of local variable or channel declarations and statements. The declaration above contains one local variable declaration and a single statement: an assignment of the value 3 to variable state. The semicolon is a statement *separator (not a statement terminator*, *hence there is no* semicolon after the last statement).
- PROMELA defines two different statement separators: an arrow, ->, and a semicolon, ;. The two separators are equivalent. The arrow is sometimes used as an informal way to indicate a causal relation between two statements.

Διεργασίες

THE INITIAL PROCESS

- Ο ορισμός proctype δηλώνει τη συμπεριφορά της διεργασίας και δεν την εκτελεί. Αρχικά εκτελείται μια διεργασία : init process στην Promela, η οποία είναι αντίστοιχη της function main() ενός standard C προγράμματος.
- The smallest possible PROMELA specification is : init { skip }
  - where skip is a null statement. Only slightly more complicated is the PROMELA equivalent of the famous ‘‘hello world’’ program from C:
    - init { printf("hello world\n") }
- More interestingly, however, the initial process can initialize global variables, create message channels, and instantiate processes. An init declaration for a two-process system:

init { run A(); run B() }

- This init process starts two processes, which will run concurrently with the init process from then on. In the above case, the init process terminates after starting the second process, but it need not do so.
  - Run is a unary operator that instantiates a copy of a given process type (for example, A). It does not wait for the process to terminate. The run statement is executable and returns a positive result only if the process can effectively be instantiated. It is unexecutable and returns zero if this cannot be done, for instance if too many processes are already running.

Διεργασίες

```
active [2] proctype you_run ()
{
    printf("my pid is: %d\n", _pid
}
```

```
$spin you_run.pml
My pid is 0
    My pid is 1
2 processes created
```

Διεργασίες

```
proctype you_run(byte x)
{
    printf("x = %d, pid = %d\n", x, _pid)
}

init {
    run you_run(0);
    run you_run(1)
}
```

```
-----------------------------------------------------------------------------
$spin you_run2.pml
    x = 1, pid=2
X=0, pid=1
```

Διεργασίες

```
proctype you_run(byte x)
{
	printf("x = %d, pid = %d\n", x, _pid)
}

init { pid p0, p1;

	p0 = run you_run(0);
	p1 = run you_run(1);
	printf("pids: %d and %d\n", p0, p1)
}
```

```
---------------------------------------------------------------------------
$spin you_run2.pml
          x = 1, pid=2
Pids: 1 and 2
        x=0, pid=1
```

Διεργασίες

```
active proctype splurge(int n)
{    int p;
     printf("%d\n", n);
     p = run splurge(n+1)
}
---------------------------
$spin –T splurge
0
1
…..
254
Spin: too many processes created (255 max)
255 processes created
```

Διεργασίες

- Μια διεργασία τερματίζει όταν φτάσει στο τέλος του σώματος του proctype “}”
- Μια διεργασία πεθαίνει όταν πεθάνουν πρώτα όσες δημιουργήθηκαν μετά από αυτή

Διεργασίες

```
bool    toggle = true;
int cnt;

active proctype A() provided (toggle == true)
{
    do
    :: cnt++; printf("A: cnt=%d\n", cnt); toggle = false
    od
}

active proctype B() provided (toggle == false)
{
    do
    :: cnt--; printf("B: cnt=%d\n", cnt); toggle = true
    od
}
```

```
$spin toggle.pml
A: cnt=1
    B:cnt=0
A: cnt=1
    B:cnt=0
```

Κανάλια μηνυμάτων

```
chan qname =          [16] of          {short, byte, bool}
                            ^
              user defined ,predefined type
                          (even other channels)
qname!expr1, expr2, expr3
qname?var1,var2,var3
qname?constant1, var2,constant2
qname?eval(var1),var2,var3
len(qname)
(a>b && qname?[msg0])
q?<eval(y), x> /* there is message and the first field has value equals the value of y*/
qname!!msg0
qname??msg0
```

Μεταβλητές και τύποι δεδομένων

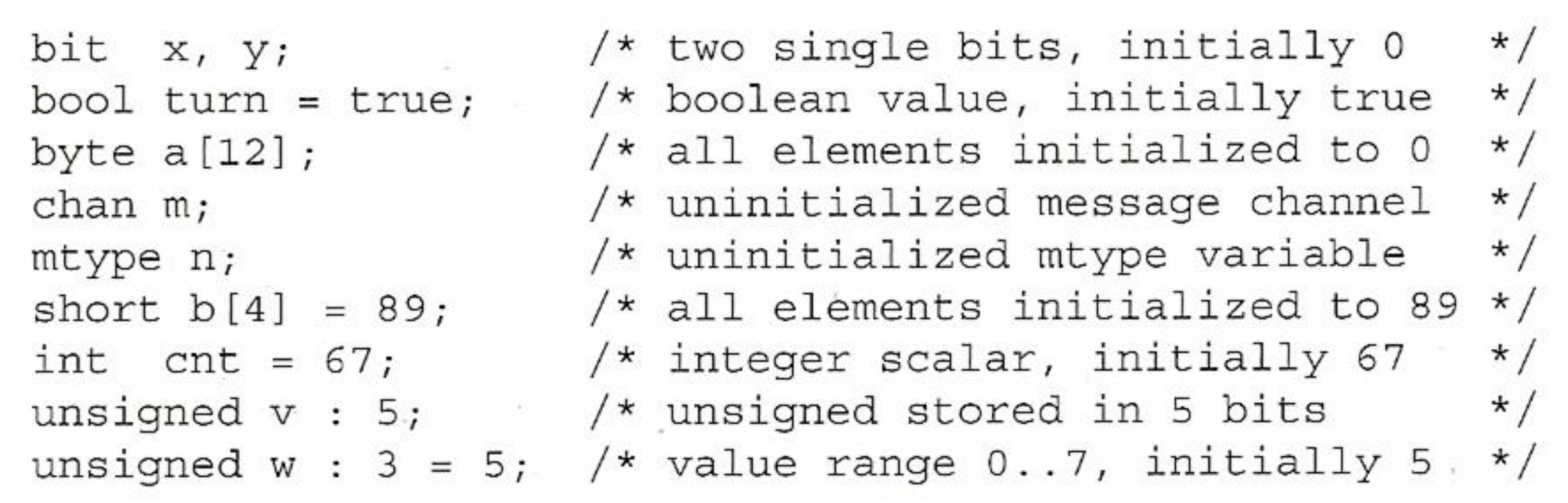
Type	Typical Range
bit	0,1
bool	false, true
byte	0..255
chan	1..255
mtype	1..255
pid	0..255
short	$-2^{15}\ldots2^{15}-1$
int	$-2^{31}\ldots2^{31}-1$
unsigned	$0..2^{n}-1$

Μεταβλητές και τύποι δεδομένων

```
init {
        /* x declared in outer block */
        int x;
        {/* y declared in inner block */
                int y;
                printf("x = %d, y = %d\n", x, y);
                x++;
                y++;
        }
        /* y remains in scope */
        printf("x = %d, y = %d\n", x, y);
}
```

```
$spin scope.pml
x=0, y=0
x=1,y=1
1 process created
```

Μεταβλητές και τύποι δεδομένων



```
bit   x, y;              /* two single bits, initially 0   */
bool turn = true;        /* boolean value, initially true  */
byte a[12];              /* all elements initialized to 0  */
chan m;                  /* uninitialized message channel  */
mtype n;                 /* uninitialized mtype variable   */
short b[4] = 89;         /* all elements initialized to 89 */
int   cnt = 67;          /* integer scalar, initially 67   */
unsigned v : 5;          /* unsigned stored in 5 bits      */
unsigned w : 3 = 5;      /* value range 0..7, initially 5  */
```

mtype

```
mtype = { appel, pear, orange, banana };
mtype = { fruit, vegetables, cardboard };

init {
    mtype n = pear;     /* initialize n to pear */
    printf("the value of n is %e\n", n);
     printf("the value of n is %d\n", n);
}
```

```
$spin mtype.pml
    the value of n is pear
    the value of n is 3
```

Πίνακες δύο διαστάσεων

```
typedef Array {
    byte el[4]
}

Array a[4]

a[i].el[j]
```

Σημείωση: δεν μπορούμε να περάσουμε σαν όρισμα ένα πίνακα σε μία νέα διεργασία

Εκτελεσιμότητα δηλώσεων

- Δεν υπάρχει διαφορά μεταξύ statements (print, assignment, i/o) & expressions
- Όλες οι δηλώσεις είναι εκτελέσιμες ή μπλοκαρισμένες ανάλογα με τις τιμές των μεταβλητών και τα περιεχόμενα των καναλιών
- Μια διεργασία περιμένει για ένα γεγονός να συμβεί περιμένοντας μία δήλωση να γίνει εκτελέσιμη

Π.χ., while (a!=b) skip ←→ (a==b)

Assignments & Expressions

- c= c+1; c=c-1 /* valid */
- c++;c-- /* valid */
- b=c++ /* not valid */
- --c; ++c/* not valid */
- variable = expression (first cast to integer then cast to type of variable, possibility of truncation)
- (expr1 ? expr2 : expr3) /*not valid*/
- (expr1 -> expr2 : expr3) /*valid*/

Control Flow

- atomic sequences
- deterministic steps
- Selections
- Repetitions
- Escape Sequences

Atomic sequences

- A sequence of statements enclosed in parentheses prefixed with the keyword atomic indicates that the sequence is to be executed as one indivisible unit, non-interleaved with any other processes. It is an error if any statement, other than the first one, can block in an atomic sequence. The executing process will abort in that case.

```
atomic { /* swap the values of a and b*/
            tmp=b;
            b=a;
            a=tmp
}


-----------------------------------------------------------------------
init{            /* none can start executing until all have been initialized*/
      atomic{
            run A(1,2);
            run B(2,3)
      }
}
```

Data structures

```
typedef Field {
        short f = 3;
        byte  g
};

typedef Record {
        short a[3];
        int fld1;
        Field fld2;
        chan p[3];
        bit b
};

proctype me(Field z) {
       z.g = 12
}

init {
       Record goo;
       Field  foo;

       run me(foo)
}
```

goo.a[2] = goo.fld2.f + 12

Deterministic steps

```
dstep { /* swap the values of a and b*/
            tmp=b;
            b=a;
            a=tmp
}
```

Εκτελείται πάντα ως μία εντολή

1. Η εκτέλεση είναι πάντα αιτιοκρατική. Ακόμα και αν υπάρχουν μη αιτιοκρατικά βήματα πάντα θα επιλύονται με τον ίδιο τρόπο

- Δεν επιτρέπονται goto
- Η εκτέλεση τους δεν μπορεί να διακοπεί από μη εκτελέσιμα statements

Selection

```
if
:: (a<b) -> option1
:: (a==b)->option2
fi
```

- A sequence can be selected only if its first statement is executable. The first statement is therefore called a *guard*.

Selection

```
#define a 1
#define b 2
chan ch = [1] of { byte };
proctype A() { ch!a }
proctype B() { ch!b }
proctype C()
        { if
        :: ch?a
        :: ch?b
        fi
        }
init { atomic { run A(); run B(); run C() } }
```

- This example defines three processes and one channel. The first option in the selection structure of the process of type C is executable if the channel contains a message a, where a is a constant with value 1, as defined in a macro definition at the start of the program. The second option is executable if it contains a message b, where b is a constant. Which message will be available depends on the relative speeds of the processes.

Selection

```
byte count;

active proctype counter()
{
    if
    :: count++
    :: count--
    fi
}
```

Repetition

```
byte count;

active proctype counter()
{
    do
    :: count++
    :: count--
    :: (count == 0) -> break
    od
}
```

- Only one option can be selected for execution at a time. After the option completes, the execution of the structure is repeated. The normal way to terminate the repetition structure is with a break statement. In the example, the loop can be broken when the count reaches zero.

Repetition

Για την επιβολή termination, μπορουμε να κανουμε την ακολουθη μετατροπή:

```
byte count;

active proctype counter()
{
    do
    :: (count != 0) ->
      if
      :: count++
      :: count--
      fi
    :: (count == 0) -> break
    od
}
```

Repetition

```
byte count;

active proctype counter()
{
      do
      :: (count != 0) ->
        if
        :: count++
        :: count--
        :: else
        fi
      :: else -> break
      od
}
```

Jump

Another way to break the loop is with an unconditional jump: the infamous goto statement. This is illustrated in the following implementation of Euclid’s algorithm for finding the greatest common divisor of two positive numbers:

```
proctype Euclid(int x, y)
{
    do
    :: (x >  y) -> x = x - y
    :: (x <  y) -> y = y - x
    :: (x == y) -> goto done
    od;
done:
    printf("answer: %d\n", x)
}

init { run Euclid(36, 12) }
```

The goto in this example jumps to a label named done

Παραδείγματα

- The following example specifies a filter that receives messages from a channel in and divides them over two channels large and small depending on the values attached. The constant N is defined to be 128, and size is defined to be 16 in two macro definitions.

Παραδείγματα

```
#define N 128
#define size 16
chan in = [size] of { short };
chan large = [size] of { short };
chan small = [size] of { short };
proctype split()
{ short cargo;
do
:: in?cargo ->
if
:: (cargo >= N) -> large!cargo
:: (cargo < N) -> small!cargo
fi
od
}
init { run split() }
```

Παραδείγματα

A process type that merges the two streams back into one, most likely in a different order, and writes it back to the channel in could be specified as:

```
proctype merge()
{ short cargo;
do
if
:: large?cargo
:: small?cargo
fi;
in!cargo
od
}
```

MODELING PROCEDURES AND RECURSION

Procedures, even recursive ones, can be modeled as processes. The return value can be passed back to the calling process via a global variable or via a message. Example:

```
proctype fact(int n; chan p)
{ int result;
if
:: (n <= 1) -> p!1
:: (n >= 2) ->
chan child = [1] of { int };
run fact(n-1, child);
child?result;
p!n*result
fi
}
init
{ int result;
chan child = [1] of { int };
run fact(7, child);
child?result;
printf("result: %d\n", result)
}
```

The process fact(n, p) recursively calculates the factorial of n, communicating the result to its parent process via channel p.

MESSAGE-TYPE DEFINITIONS

Constants can be defined using C-style macros. PROMELA allows for message type definitions of the form:

```
mtype = { ack, nak, err, next, accept }
```

The definition is equivalent to the following sequence of macro definitions.

```
#define ack 1
#define nak 2
#define err 3
#define next 4
#define accept 5
```

A formal message-type definition is the preferred way of specifying the message types since it defers any decision on the specific values to be used. At the same time, it makes the names of the constants, rather than the values, available to an implementation, which can improve error reporting. There can be only one message-type definition per specification.

MODELING TIMEOUTS

2 types of statements with a predefined meaning in PROMELA: skip and break.

Another predefined statement is timeout. The timeout statement allows a process to abort the waiting for a condition that can no longer become true, for example, an input from an empty channel. The timeout provides an escape from a hang state. It can be considered an artificial, predefined condition that becomes true only when no other statements in the distributed system are executable.

Note that it carries no value: it does not specify a timeout interval, but a timeout possibility.

A simple example is the following process that sends a reset message to a channel named guard whenever the system comes to a standstill.

```
proctype watchdog()
{ do
:: timeout -> guard!reset
od
}
```

STATEMENT TYPES

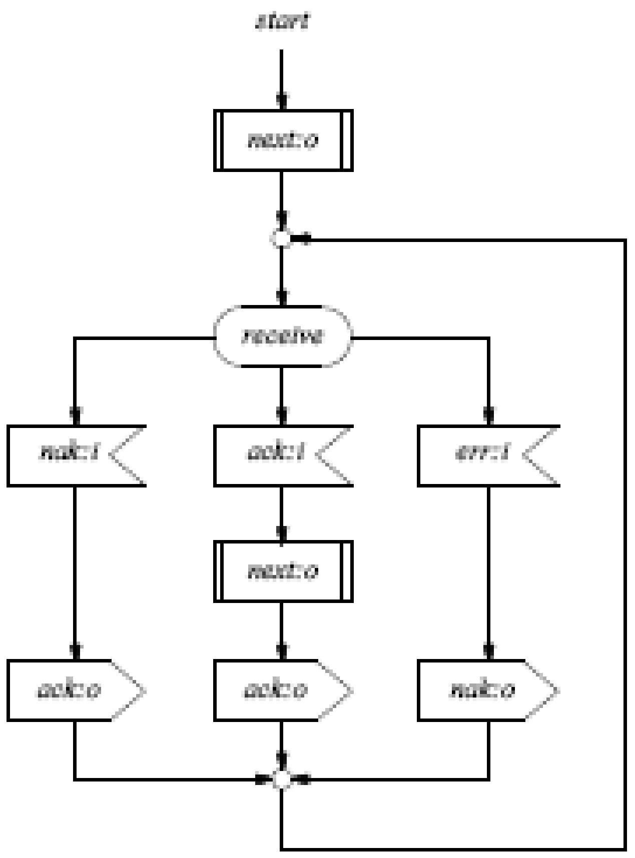
Basic types of statements defined in PROMELA:

- Assignments and conditions
- Selections and repetitions
- Send and receive
- Goto and break statements
- Timeout

The skip statement was introduced as a filler to satisfy syntax requirements. It is not formally part of the language but a *pseudo-statement, a synonym for another statement* with the same effect. Trivially, skip is equivalent to the condition (1); it is always executable and has no effect.

LYNCH´s PROTOCOL REVISITED

```
mtype = { ack, nak, err, next, accept }
proctype transfer(chan in, out, chin, chout)
{ byte o, i;
in?next(o);
do
:: chin?nak(i) -> out!accept(i); chout!ack(o)
:: chin?ack(i) -> out!accept(i); in?next(o); chout!ack(o)
:: chin?err(i) -> chout!nak(o)
od
}
init
{ chan AtoB = [1] of { byte, byte };
chan BtoA = [1] of { byte, byte };
chan Ain = [2] of { byte, byte };
chan Bin = [2] of { byte, byte };
chan Aout = [2] of { byte, byte };
chan Bout = [2] of { byte, byte };
atomic {
run transfer(Ain, Aout, AtoB, BtoA);
run transfer(Bin, Bout, BtoA, AtoB)
};
AtoB!err(0)
}
```



The channels Ain and Bin are to be filled with token messages of type next and arbitrary values (e.g., ASCII character values) by unspecified background processes: the users of the transfer service. Similarly, these user processes can read received data from the channels Aout and Bout. The processes are initialized in an atomic statement and started with the dummy err message.

Escape Sequences

```
{P} unless {Q}
A;
do
:: b1->B1
::b2->B2
….
od unless {c->C}
D
```

Escape Sequences

```
A;
do
:: b1->B1
::b2->B2
….
::c - >break
od
C;
D
```

Other message types

```
mtype = { offhook, dialtone, number, ringing,
   busy, connected, hangup, hungup };
```

Example phone line

```
active proctype pots()
{    chan who;
idle:  line?offhook,who;
     {         who!dialtone;
       who?number;
       if
       :: who!busy; goto zombie
       :: who!ringing ->
              who!connected;
              if
              :: who!hungup; goto zombie
              :: skip
              fi
       fi
     } unless
     {if
       :: who?hangup -> goto idle
       :: timeout -> goto zombie
       fi
     }
zombie:     who?hangup; goto idle
}
```

Example phone line

```
active proctype subscriber()
{    chan me = [1] of { mtype };
idle: line!offhook,me;
      me?dialtone;
      me!number;
      if
      :: me?busy
      :: me?ringing ->
        if
        :: me?connected;
                if
                :: me?hungup
                :: timeout
                fi
        :: skip
        fi
      fi;
      me!hangup; goto idle
}
```

Repetition

```
mtype = { msg, ack };

chan to_sndr = [2] of { mtype, bit };
chan to_rcvr = [2] of { mtype, bit };

active proctype Sender()
{       int seq_out, seq_in;

        /* obtain first message */
        do
        :: to_rcvr!msg(seq_out) ->
           to_sndr?ack(seq_in);
           if
           :: seq_in == seq_out ->
                    /* obtain new message */
                    seq_out = 1 - seq_out;
           :: else
           fi
        od
}

active proctype Receiver()
{       int seq_in;

        do
        :: to_rcvr?msg(seq_in) ->
           to_sndr!ack(seq_in)
        :: timeout ->      /* recover from msg loss */
           to_sndr!ack(seq_in)
        od
}
```

Repetition

```
mtype = { msg, ack };

chan  to_sndr = [2] of { mtype, bit };
chan  to_rcvr = [2] of { mtype, bit };

active proctype Sender()
{       int seq_out, seq_in;

        /* obtain first message */
        do
        :: to_rcvr!msg(seq_out) ->
           to_sndr?ack(seq_in);
           if
           :: seq_in == seq_out ->
                   /* obtain new message */
                   seq_out = 1 - seq_out;
           :: else
           fi
        ::timeout->to_rcvr!msg(seq_out) ->
           to_sndr?ack(seq_in);
           if
           :: seq_in == seq_out ->
                   /* obtain new message */
                   seq_out = 1 - seq_out;
           :: else
           fi
        od
}

active proctype Receiver()
{       int seq_in;

        do
        :: to_rcvr?msg(seq_in) ->
           to_sndr!ack(seq_in)
        od
}
```

Synthetic version

```
mtype = { msg, ack };

chan  to_sndr = [2] of { mtype, bit };
chan  to_rcvr = [2] of { mtype, bit };

active proctype Sender()
{       int seq_out, seq_in;

        /* obtain first message */
        do
        :: to_rcvr!msg(seq_out) ->
          if
          ::to_sndr?ack(seq_in);
                  if
                  :: seq_in == seq_out ->
                          /* obtain new message */
                          seq_out = 1 - seq_out;
                  :: else
                  fi
          ::timeout-> to_rcvr!msg(seq_out)
          fi
        od
}

active proctype Receiver()
{       int seq_in;

        do
        :: to_rcvr?msg(seq_in) ->
          to_sndr!ack(seq_in)
        ::to_rcvr?msg(seq_in)
        od
}
```

Producer-Consumer

```
mtype = { P, C };

mtype turn = P;

active proctype producer()
{
      do
      :: (turn == P) ->
        printf("Produce\n");
        turn = C
      od
}

active proctype consumer()
{
      do
      :: (turn == C) ->
        printf("Consume\n");
        turn = P
      od
}
```

Data Transfer Protocol

```
mtype = { ini, ack, dreq, data, shutup, quiet, dead };

chan M = [1] of { mtype };
chan W = [1] of { mtype };

active proctype Mproc()
{
    W!ini;              /* connection     */
    M?ack;              /* handshake      */

    timeout ->          /* wait           */
    if                  /* two options:   */
    :: W!shutup         /* start shutdown */
    :: W!dreq;          /* or request data */
        do
        :: M?data ->W!data        /* send data      */
        :: M?data ->W!shutup;     /* or shutdown    */
            break
        od
    fi;

    M?shutup;           /* shutdown handshake */
    W!quiet;
    M?dead
}
```

```
active proctype Wproc()
{
    W?ini;              /* wait for ini   */
    M!ack;              /* acknowledge    */

    do                  /* 3 options:     */
    :: W?dreq ->        /* data requested */
        M!data          /* send data      */
    :: W?data ->        /* receive data   */
        skip            /* no response    */
    :: W?shutup ->
        M!shutup;       /* start shutdown */
        break
    od;

    W?quiet;
    M!dead
}
```

Data Transfer Protocol

```
mtype = { ini, ack, dreq, data, shutup, quiet, dead };

chan M = [1] of { mtype };
chan W = [1] of { mtype };

active proctype Mproc()
{
   W!ini;              /* connection     */
   M?ack;              /* handshake      */

   timeout ->          /* wait           */
   if                  /* two options:   */
   :: W!shutup         /* start shutdown  */
   :: W!dreq;          /* or request data */
      M?data ->        /* receive data   */
      do
      :: W!data        /* send data      */
      :: W!shutup;     /* or shutdown    */
         break
      od
   fi;

   M?shutup;           /* shutdown handshake */
   W!quiet;
   M?dead
}
```

```
active proctype Wproc()
{
   W?ini;              /* wait for ini   */
   M!ack;              /* acknowledge    */

   do                  /* 3 options:     */
   :: W?dreq ->        /* data requested  */
      M!data           /* send data      */
   :: W?data ->        /* receive data   */
M!data
   :: W?shutup ->
      M!shutup;        /* start shutdown  */
      break
   od;

   W?quiet;
   M!dead
}
```

Alternating bit protocol

```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sndr = [2] of { mtype };
chan to_rcvr = [2] of { mtype };

active proctype Sender()
{
again:      to_rcvr!msg1;
      to_sndr?ack1;
      to_rcvr!msg0;
      to_sndr?ack0;
      goto again
}

active proctype Receiver()
{
again:      to_rcvr?msg1;
      to_sndr!ack1;
      to_rcvr?msg0;
      to_sndr!ack0;
      goto again
}
```

Rendezvous Communication

```
mtype = { msgtype };

chan name = [2] of { mtype, byte };

active proctype A()
{   name!msgtype(124);
    name!msgtype(121)
}

active proctype B()
{   byte state;
    name?msgtype(state)
}
```