Coloured Petri Nets

Modelling and Validation of Concurrent Systems

Chapter 3: CPN ML Programming

Kurt Jensen &
Lars Michael Kristensen

{kjensen,lmkristensen}
@cs.au.dk

```
colset PACKETS  = list PACKET;
var packets : PACKETS;
fun member (e,l) =
    let
       fun equal x = (e=x)
    in
       exists (equal,l)
    end;
```

CPN ML programming language

- Based on the functional programming language Standard ML

- CPN ML extends the Standard ML environment with
  - Constructs for defining colour sets and declaring variables
  - Concept of multisets and associated functions and operators

- Standard ML plays a major role in CPN modelling and CPN Tools
  - Provides the expressiveness required to model data and data manipulation as found in typical industrial projects
  - Used to implement simulation, state space analysis, and performance analysis in CPN Tools
  - Supports a flexible and open architecture that makes it possible to develop extensions and prototypes in CPN Tools

Why Standard ML?

- Formal definition of CP-nets uses types, variables, and evaluation of expressions, which are basic concepts from functional programming
- Patterns in functional programming languages provide an elegant way of implementing enabling inference
- Standard ML is based on the lambda-calculus which has a formal syntax and semantics implying that CPN Tools get an expressive and sound formal foundation
- Standard ML is supported by mature compilers, associated documentation and textbooks

Functional programming and CPN ML

- Computation proceeds by evaluation of expressions not by executing statements making modifications to memory locations

- Strong typing means that all expressions have a type that can be determined at compile time which eliminates many run-time errors
- Types of expressions are inferred by the type system rather than being declared by the user

- Functions are first-order values and is treated in the same way as basic types such as integers, Booleans, and strings
- Functions can be polymorphic and hence operate on different types of values
- Recursion is used to express iterative constructs

Simple colour sets

- A set of basic types for defining simple colour sets
  - Integers - **int**: `{..., ~2 , ~1 , 0 , 1 , 2 ,...}`
  - Strings - **string**: `{"a" , "abc" ,...}`
  - Booleans - **bool**: `{true,false}`
  - Unit - **unit**: `{()}`
- Standard colour set definitions

```
colset INT    = int;
colset STRING = string;
colset BOOL   = bool;
colset UNIT   = unit;
```

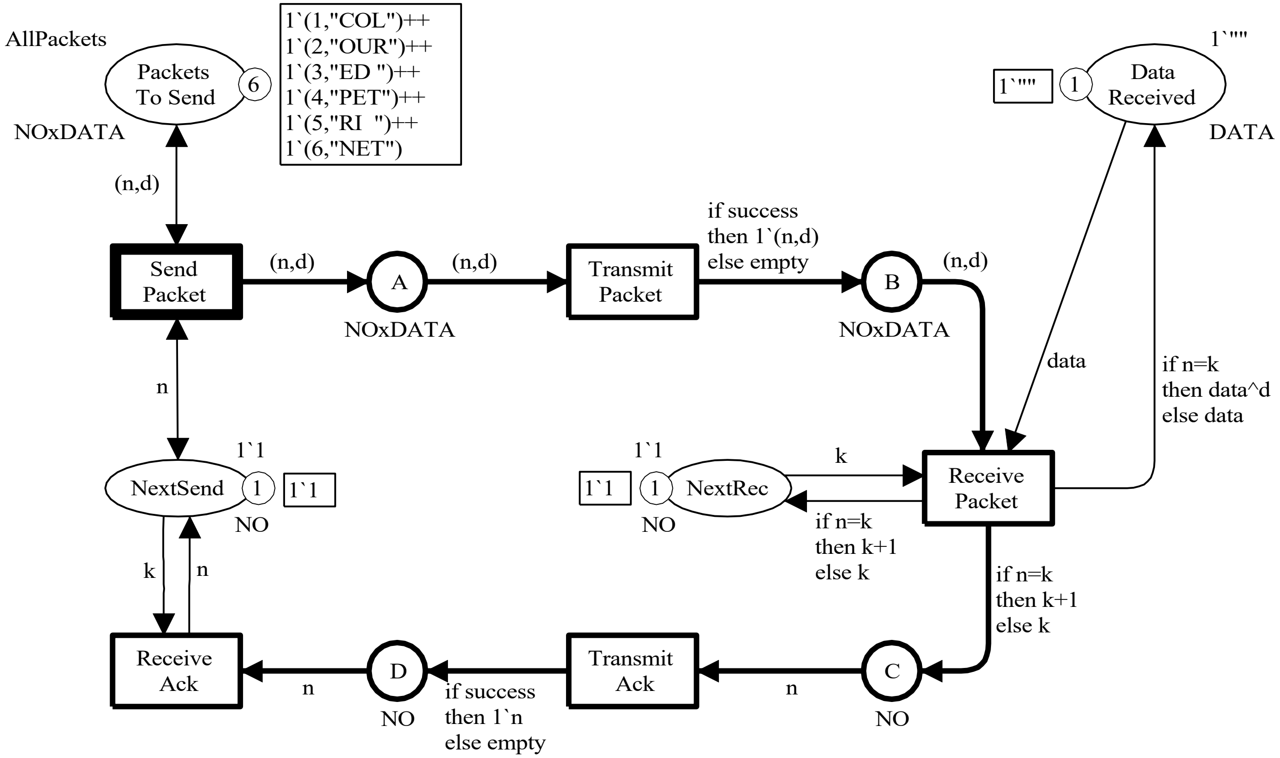
- Two other kinds of simple colour sets
  - enumeration colour sets
  - indexed colour sets

Structured colour sets

- Structured colours sets are defined using colour set constructors:
  - Products
  - Records
  - Unions
  - Lists
  - Subsets

```
colset NOxDATA   = product NO * DATA;
colset DATAPACK = record seq:NO * data:DATA;
colset PACKET    = union Data:DATAPACK + Ack:ACKPACK;
colset PACKETS   = list PACKET;
```

Simple protocol



- This version uses products to represent data packets

- We will now develop a new version where
  - Data packets are modelled as a record colour set
  - Data packets and acknowledgement packets are modelled by a common union colour set
  - We have duplication of packets
    – in addition to loss and successful transmission

Revised colour set definitions

- Old definitions

```
colset DATA    = string;
colset NO      = int;
colset NOxDATA = product NO * DATA;
```

- New definitions

Record field names

```
colset DATAPACK = record seq : NO * data : DATA;
colset ACKPACK  = NO;
colset PACKET   = union Data : DATAPACK + Ack : ACKPACK;
```

Data constructors

Enumeration colour set (with three explicitly specified data values)

```
colset RESULT   = with success | failure | duplicate;
```

Example values

- Record colour set

```
colset DATAPACK = record seq : NO * data : DATA;
```

```
{seq=1,data="COL"}
```

```
{data="COL",seq=1,}
```

Same data value

- Union colour set

Data constructors

```
colset PACKET = union Data : DATAPACK + Ack : ACKPACK;
```

```
Data {seq=1,data="COL"}
```

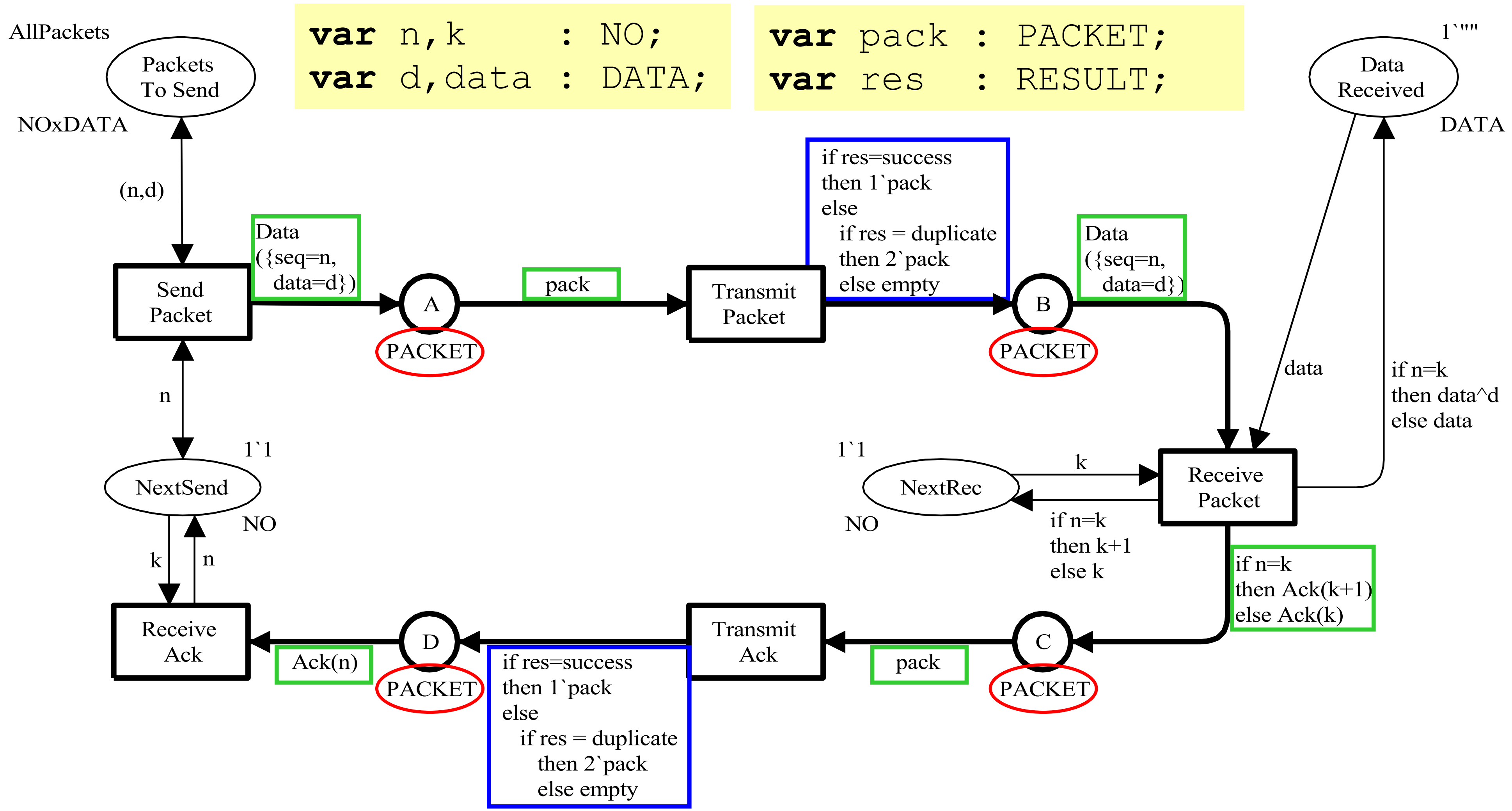
Data packet

```
Ack (2)
```

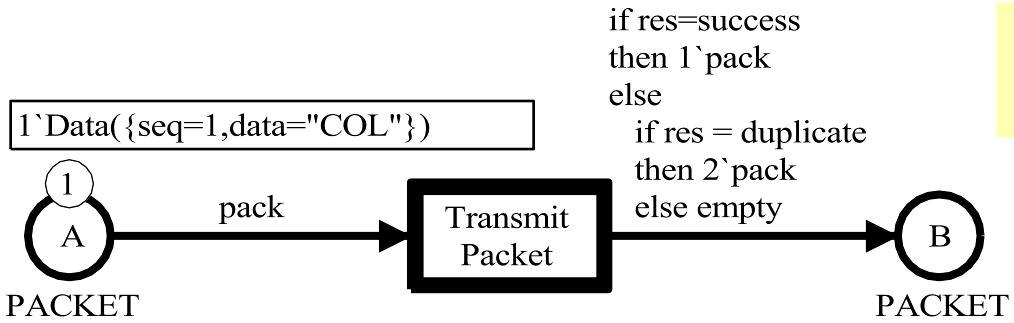
Acknowledgement packet

```
colset ACKPACK = NO;
```

Revised CPN model



Transmit Packet transition

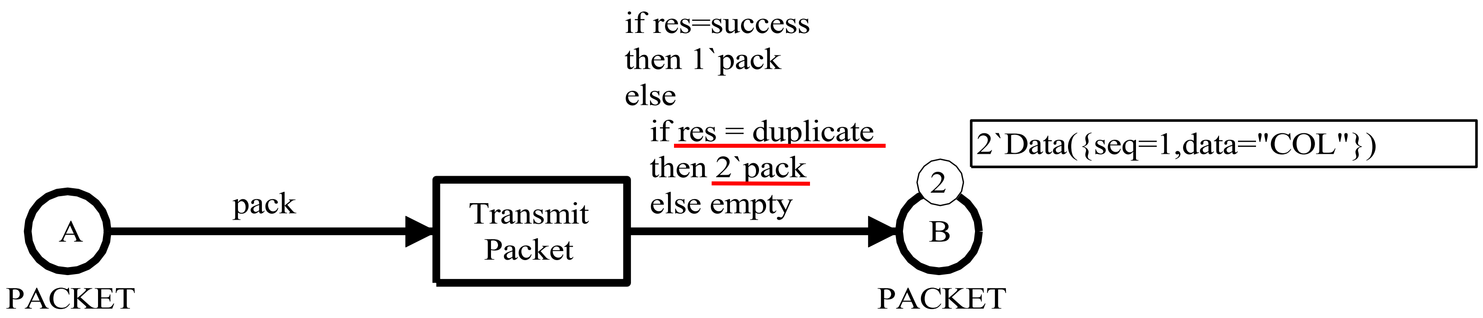



```
var pack : PACKET;
var res  : RESULT;
```

b^{+} = <pack=Data({seq=1,data="COL"}), res=success>

b^{-} = <pack=Data({seq=1,data="COL"}), res=failure>

b^{++} = <pack=Data({seq=1,data="COL"}), res=duplicate>

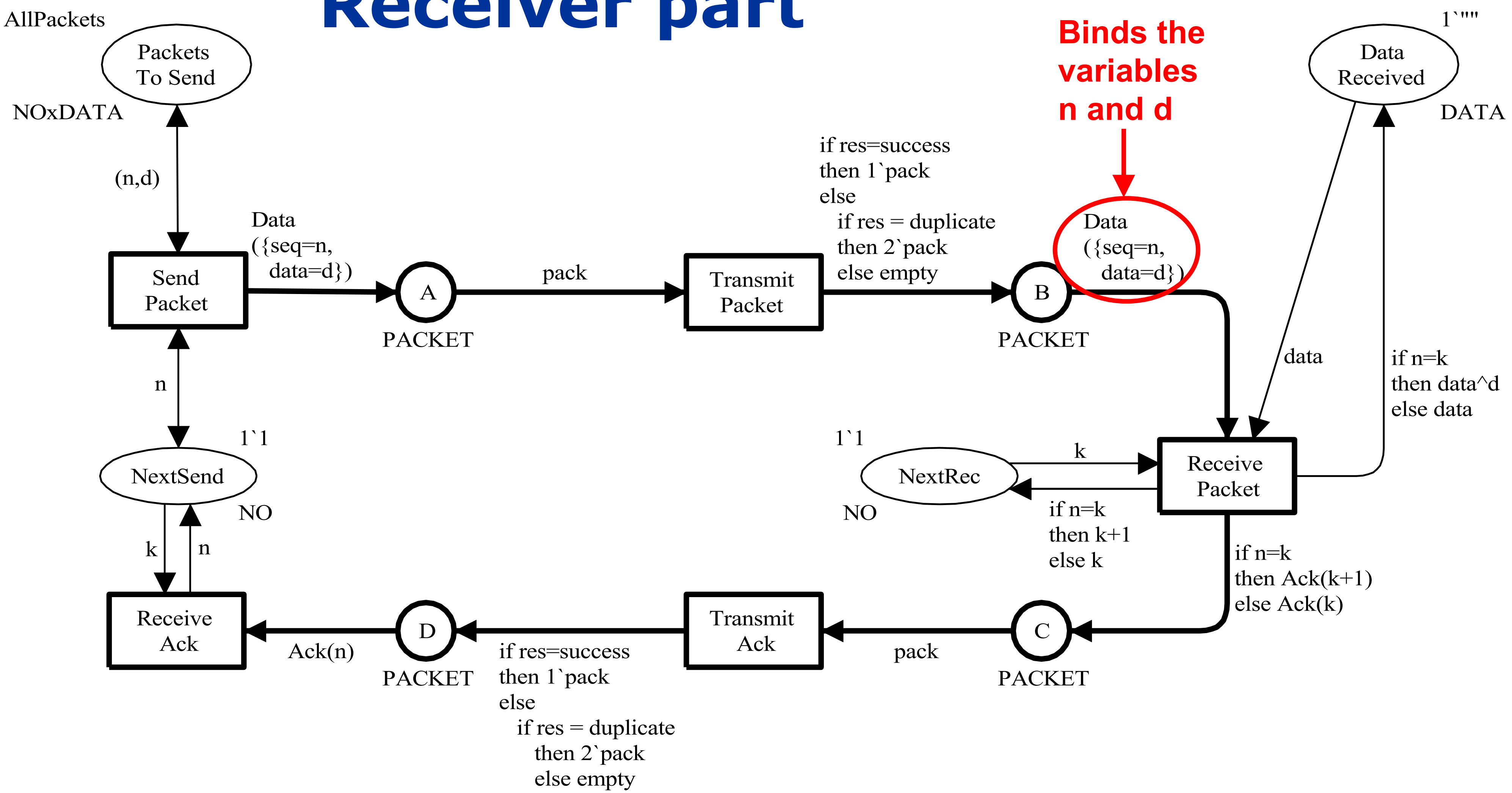


Tuples and records

- Tuple components and record fields can be accessed using the family of # operators
- Examples

`#seq {seq=1,data="COL"}`	1	Records
`#data {seq=1,data="COL"}`	"COL"	
`#1 (3,"ED ")`	3	Products
`#2 (3,"ED ")`	"ED "	

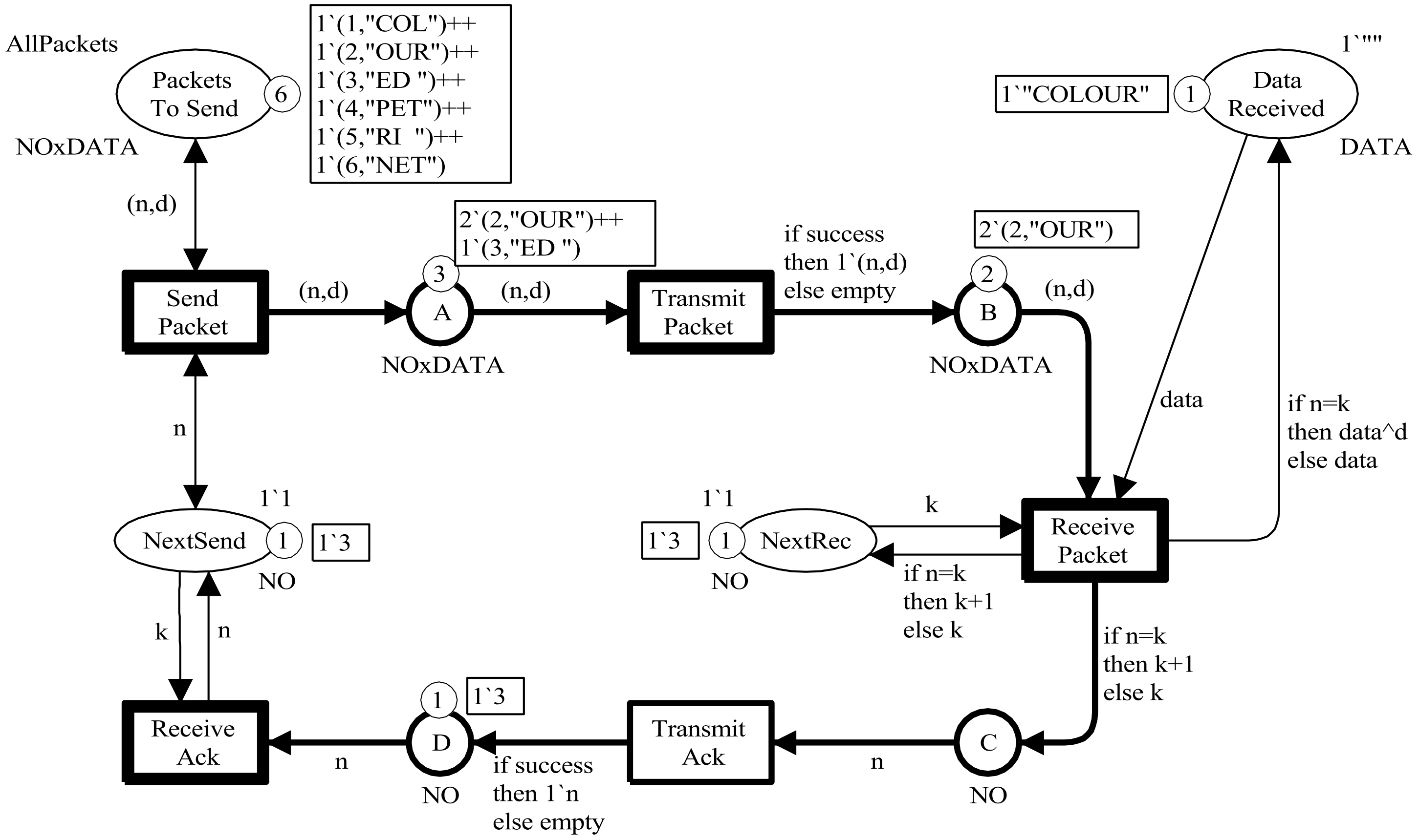
Receiver part



Products or records?

- There is a always a choice between using product or record colour sets
- Products may give shorter net inscriptions, because we avoid the selector names used in records
- Records may give more readable net inscriptions due to the mnemonic selector names. The same effect can often be achieved for products by using variables with mnemonic names, e.g. (seq,data)
- As a rule of thumb we do not recommend using products with more than 4-5 components. In such cases it is better to use records

Overtaking is possible



- We will develop a new version where overtaking of data packets and acknowledgements is impossible

List colour sets

- Colour set definitions

```
colset DATAPACKS = list NOxDATA;
colset ACKPACKS  = list NO;
```

- Example values

`[(1,"COL"),(1,"COL"),(2,"OUR")]` ← **Three data packets**

`[2,2,3,3]` ← **Four acknowledgement packets**

`[]` ← **Empty list (polymorphic)**

List concatenation (^^)

- Application

```
[(1,"COL"),(1,"COL")]^^[(2,"OUR"),(3,"ED ")]
```

List List

- Result

```
[(1,"COL"),(1,"COL"),(2,"OUR"),(3,"ED ")]
```

List

List construction (::)

- Application

```
(1,"COL")::[(1,"COL"),(2,"OUR")]
```

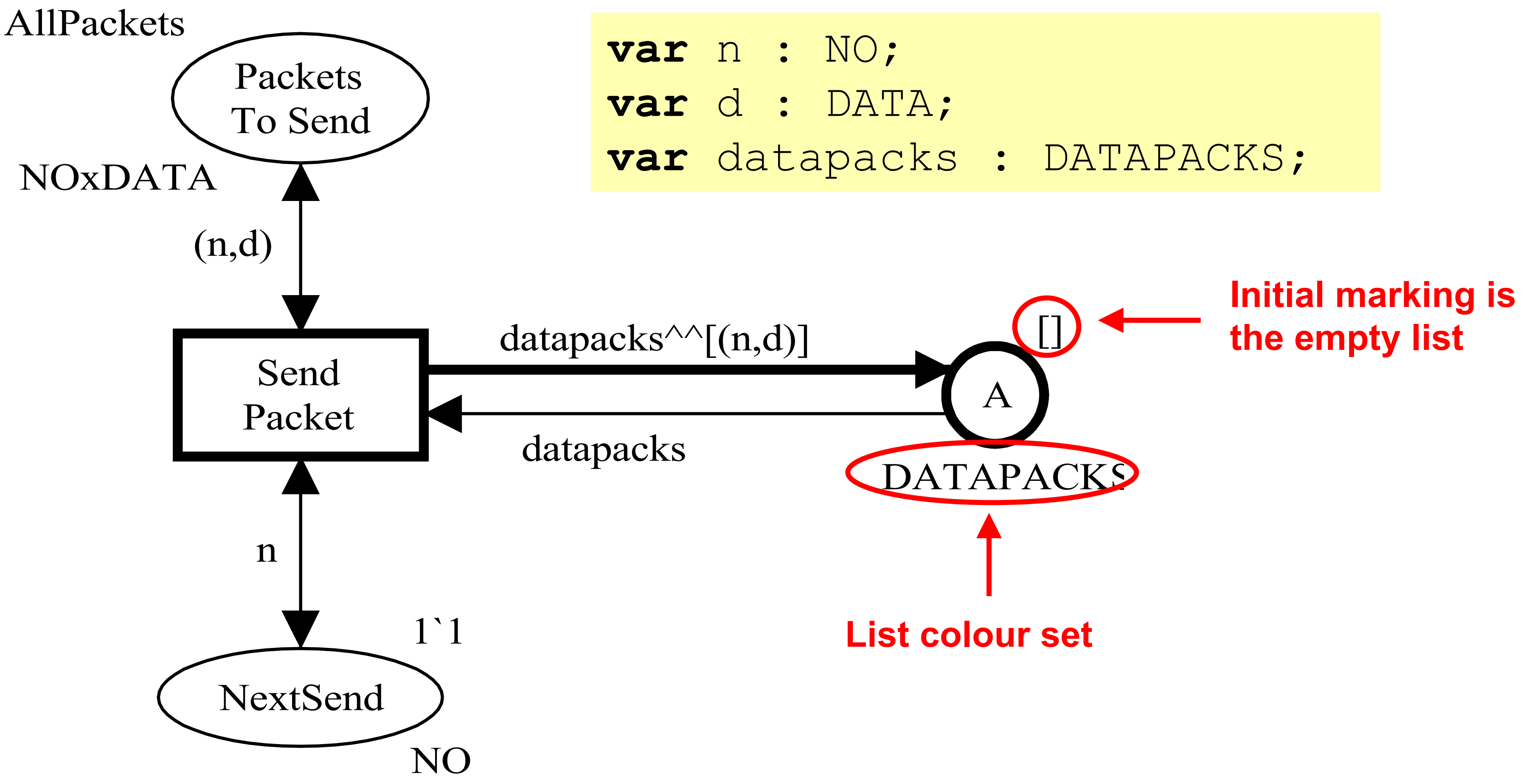
Element List

- Result

```
[(1,"COL"),(1,"COL"),(2,"OUR")]
```

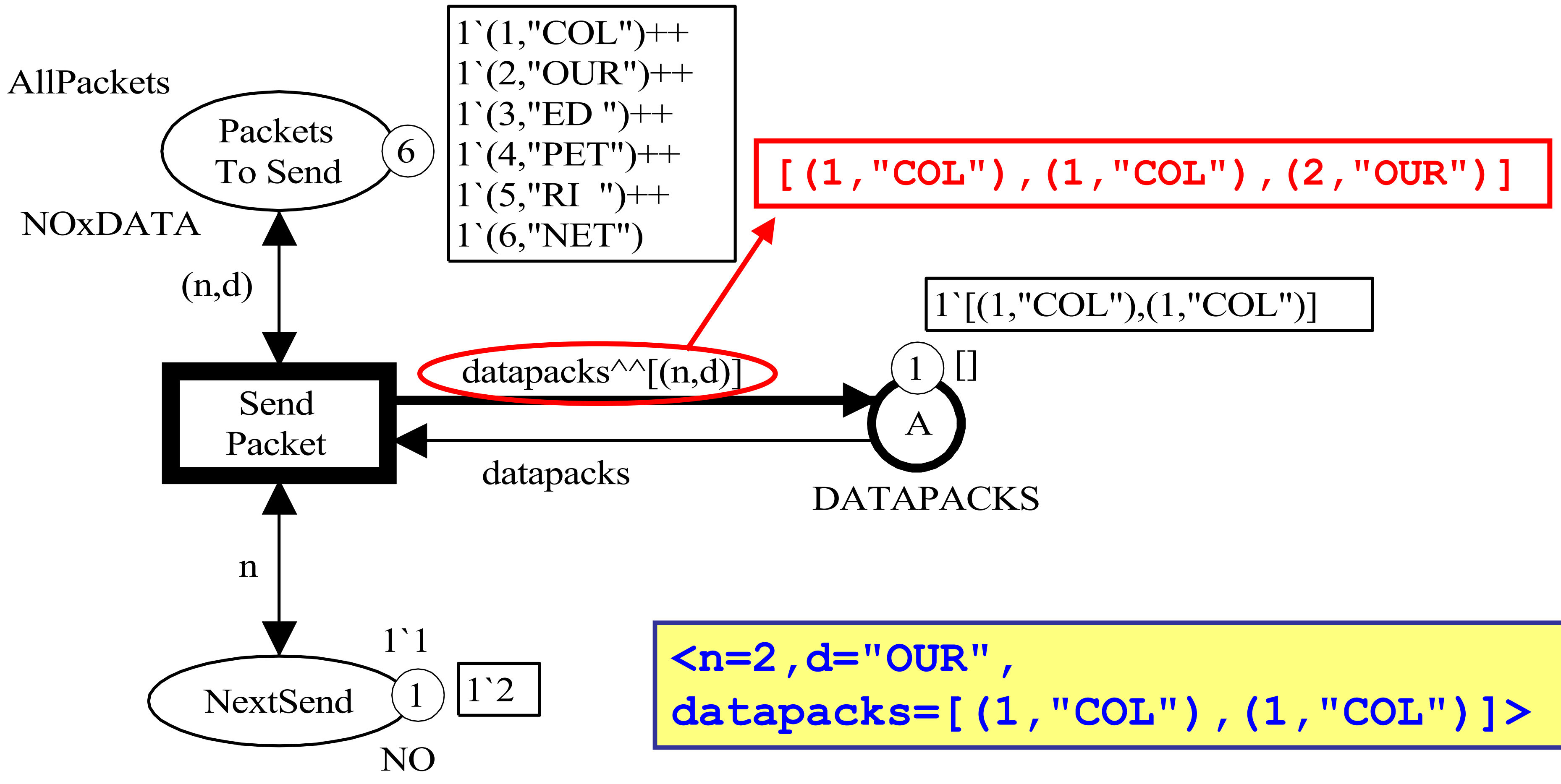
List

Revised SendPacket



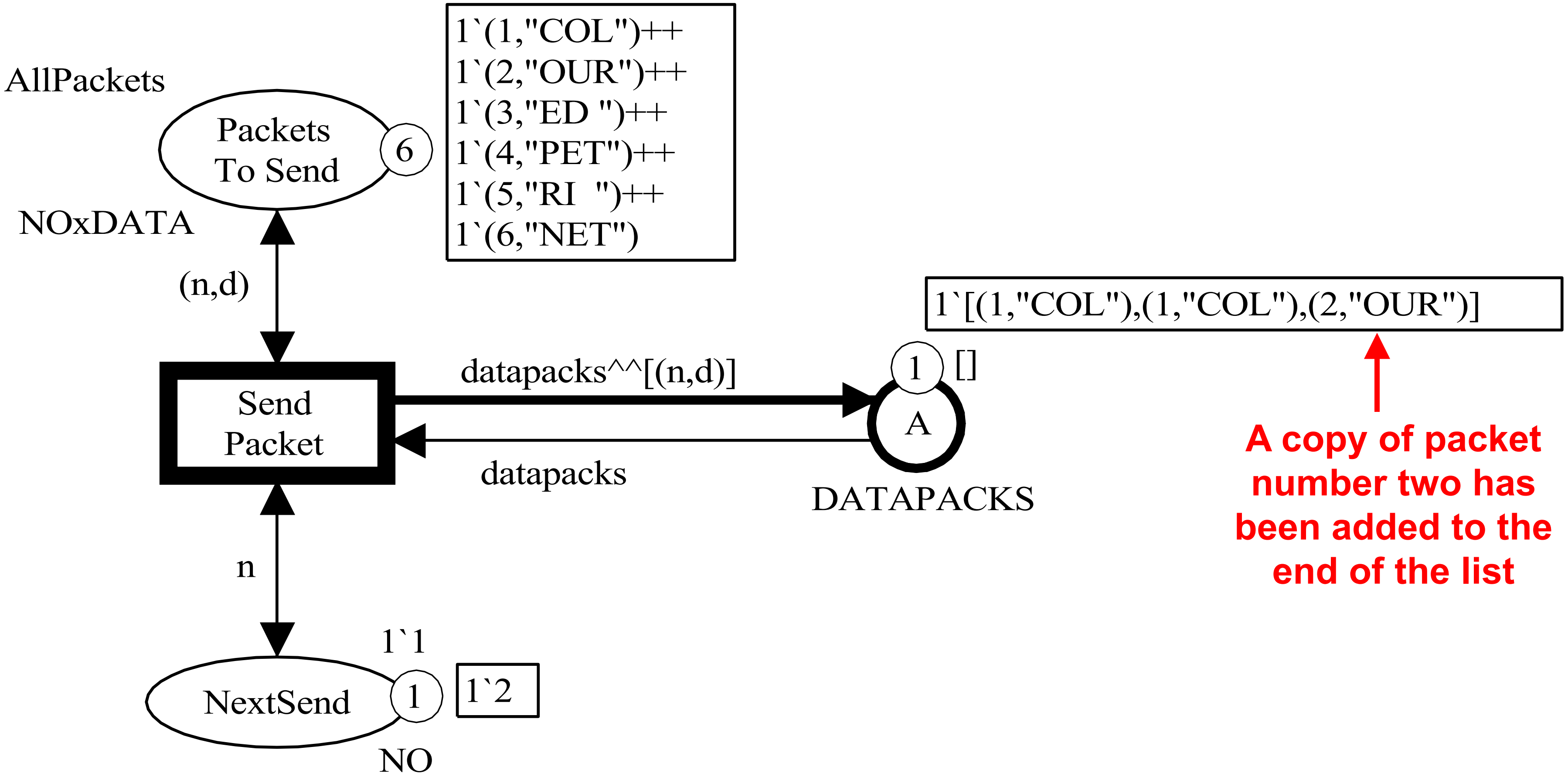

```
var n : NO;
var d : DATA;
var datapacks : DATAPACKS;
```

Enabling of SendPacket

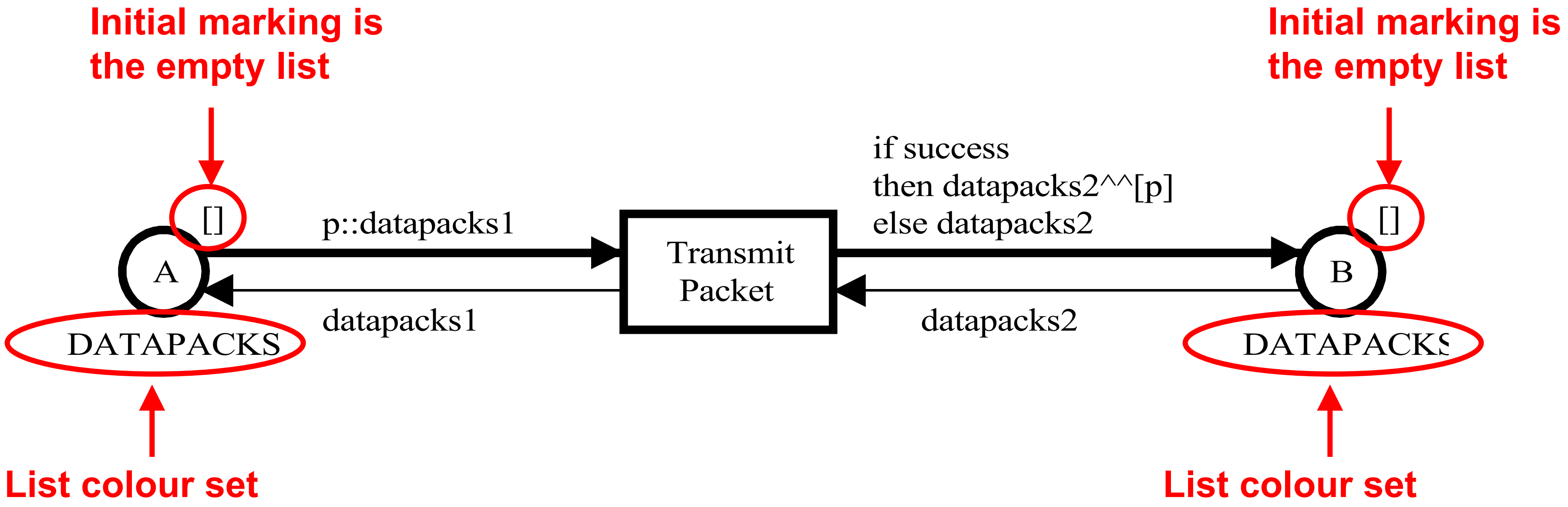


<n=2,d="OUR",
datapacks=[(1,"COL"),(1,"COL")]>

Occurrence of SendPacket

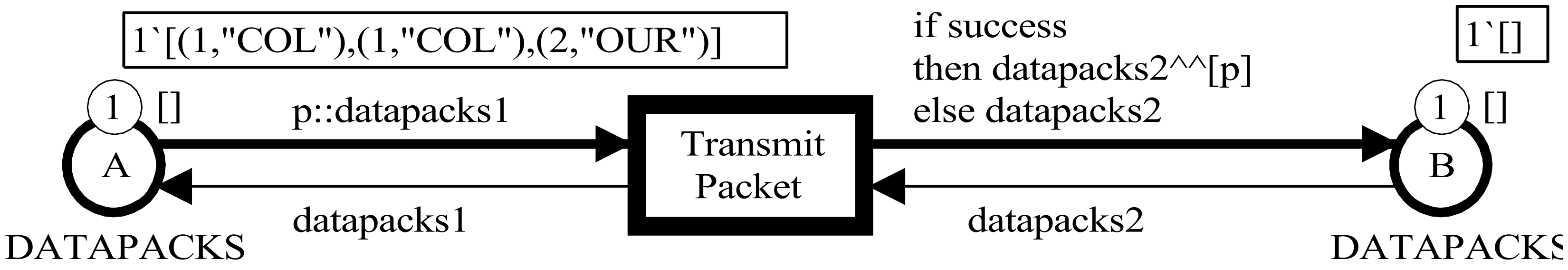


Revised TransmitPacket




```
var p        : NOxDATA;
var success : BOOL;
var datapacks1,datapacks2 : DATAPACKS;
```

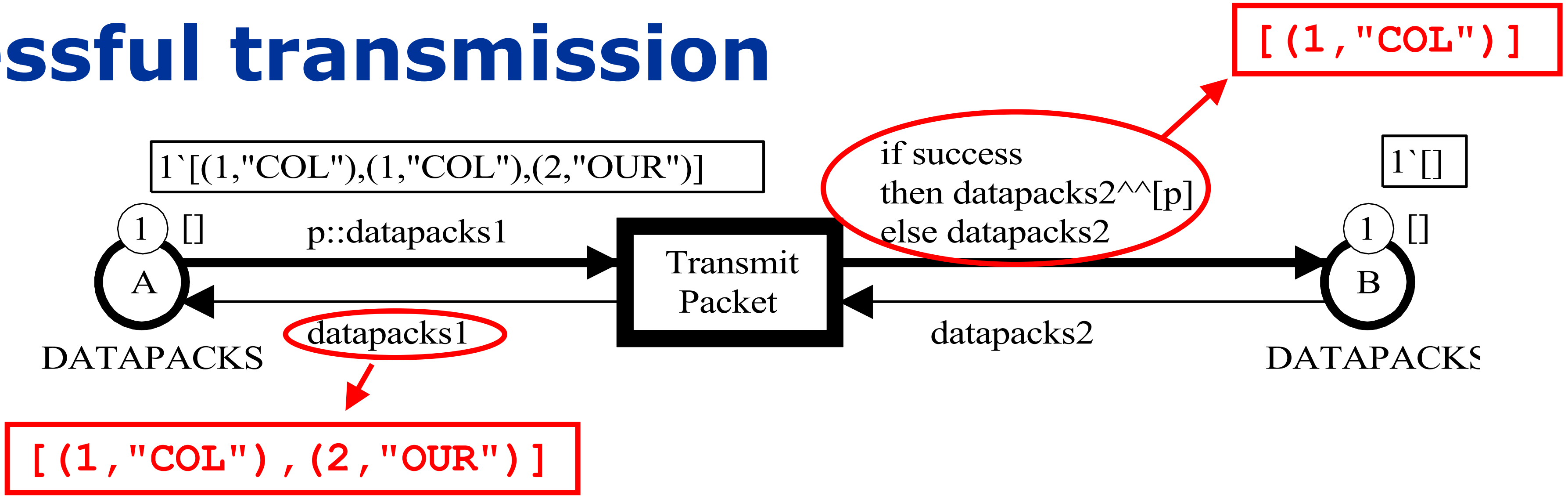
Enabling of TransmitPacket



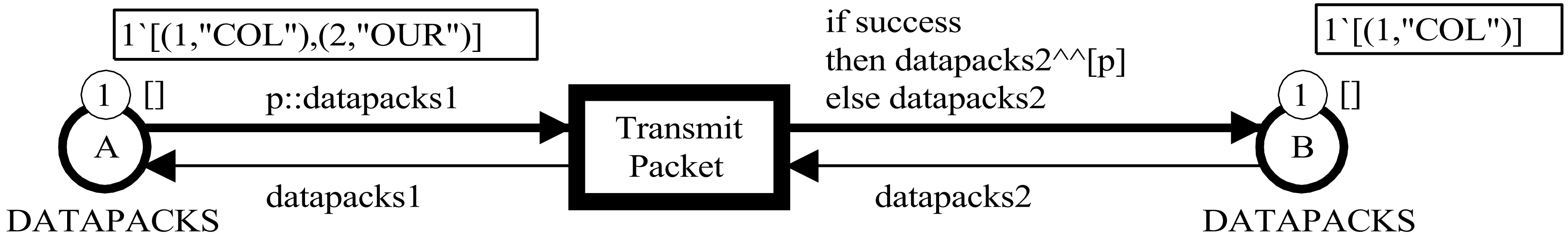

```
b+ = <p=(1,"COL"),datapacks1=[(1,"COL"),(2,"OUR")],
      success=true,datapacks2=[ ]>

b- = <p=(1,"COL")},datapacks1=[(1,"COL"),(2,"OUR")],
      success=false,datapacks2=[ ]>
```

Successful transmission

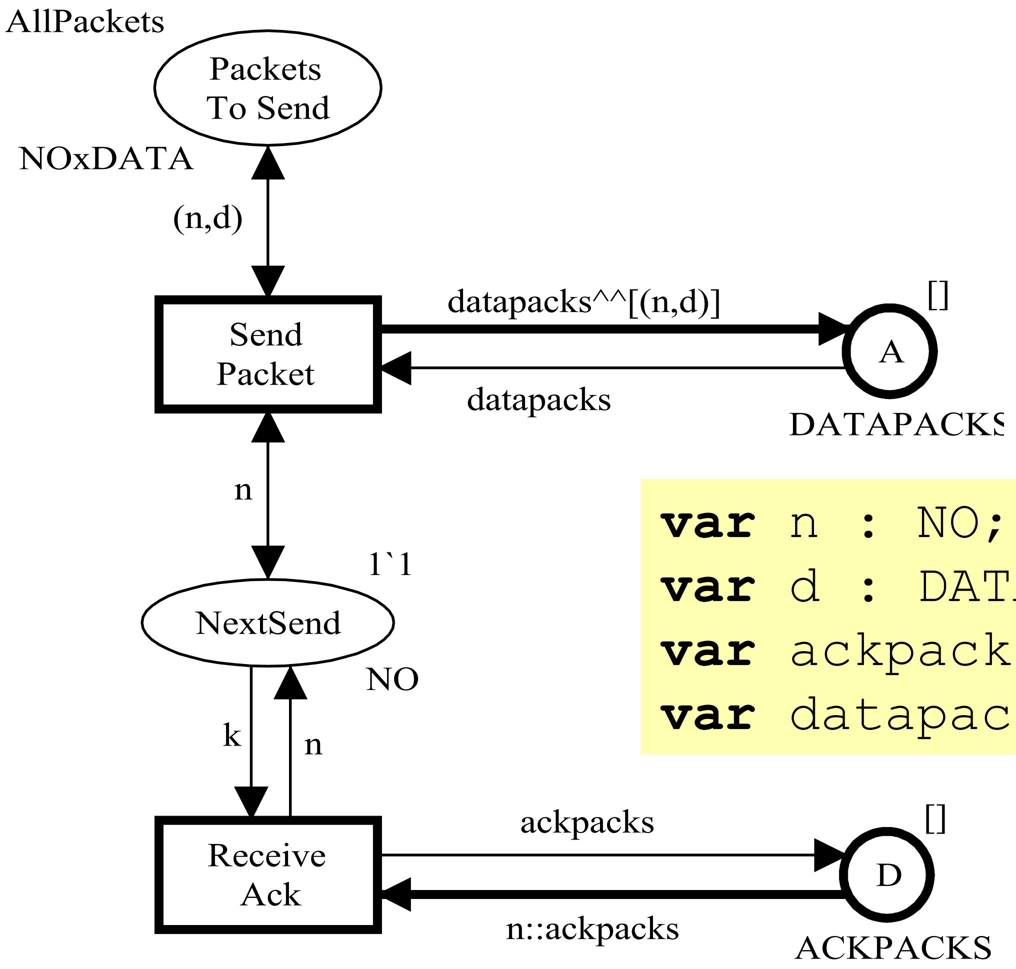


$$b^{+} = \langle p=(1,\text{"COL"}), datapacks1=[(1,\text{"COL"}),(2,\text{"OUR"})], success=true, datapacks2=[\,]\rangle$$



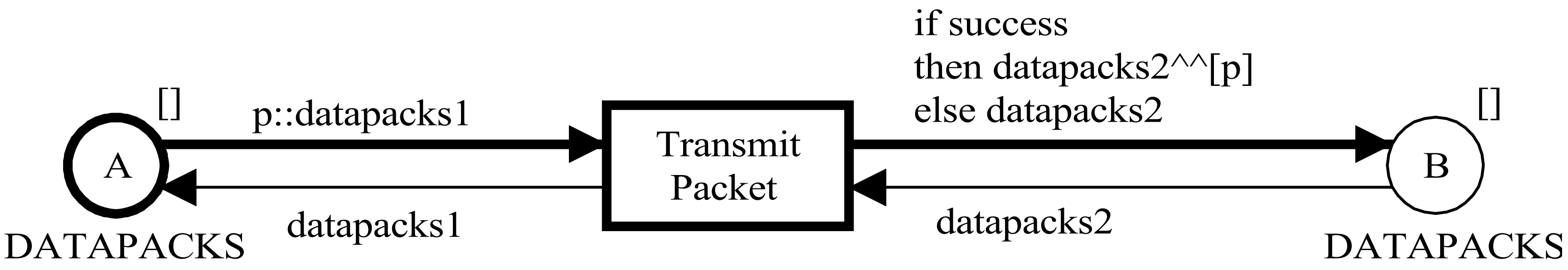
The first element from the A-list has been moved to the end of the B-list

Revised sender

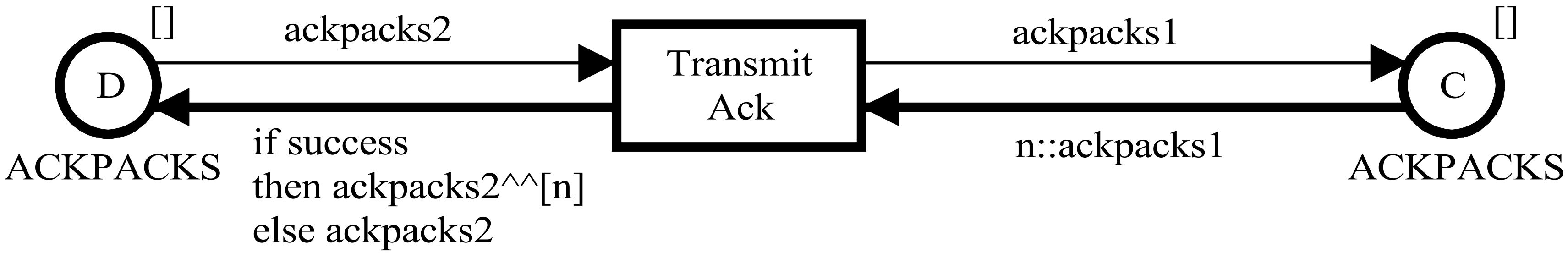



```
var n : NO;
var d : DATA;
var ackpacks : ACKPACKS;
var datapacks : DATAPACKS;
```

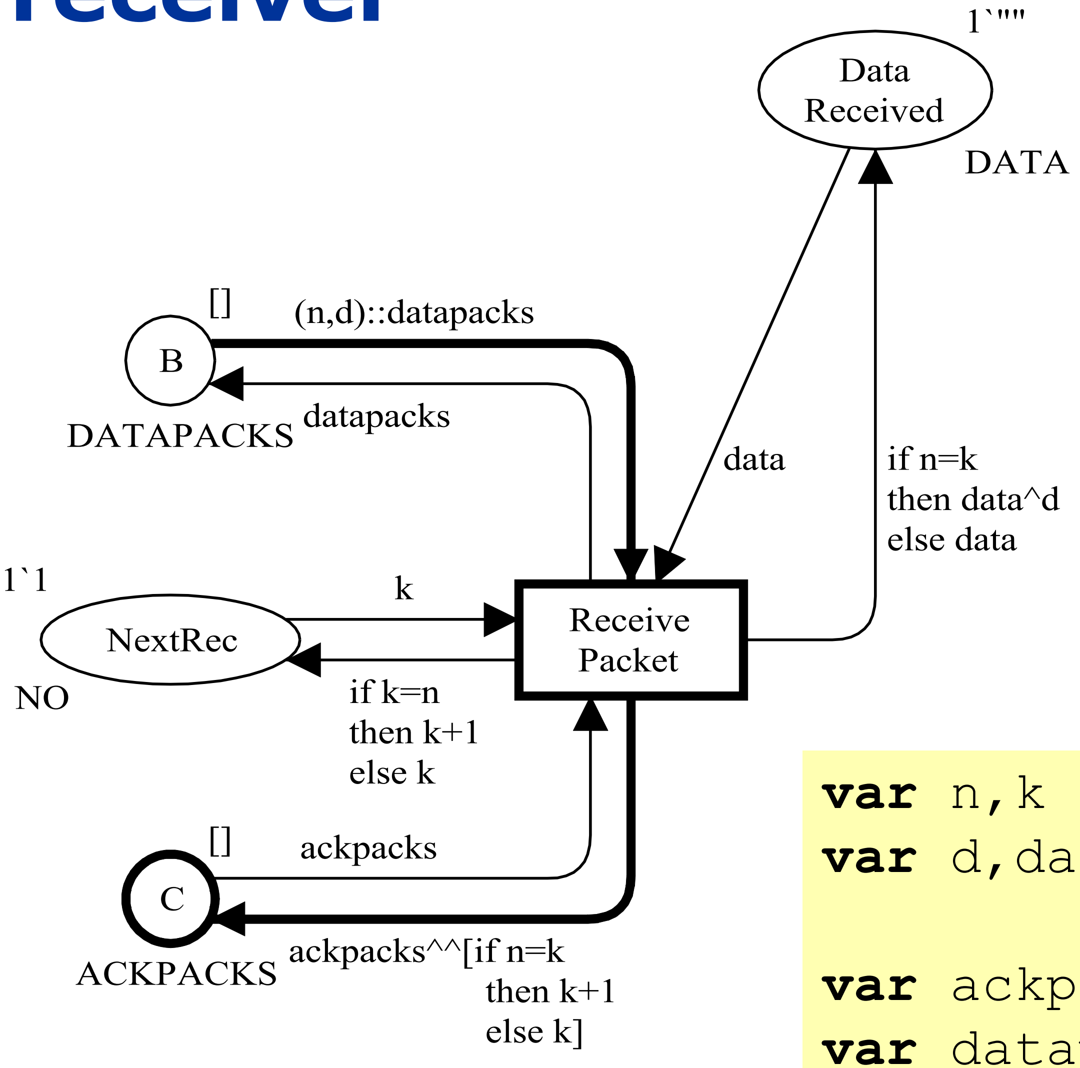
Revised network




```
var n : NO;
var p : DATAPACK;
var success : BOOL;
var ackpacks1,ackpacks2 : ACKPACKS;
var datapacks1,datapacks2 : DATAPACKS;
```



Revised receiver



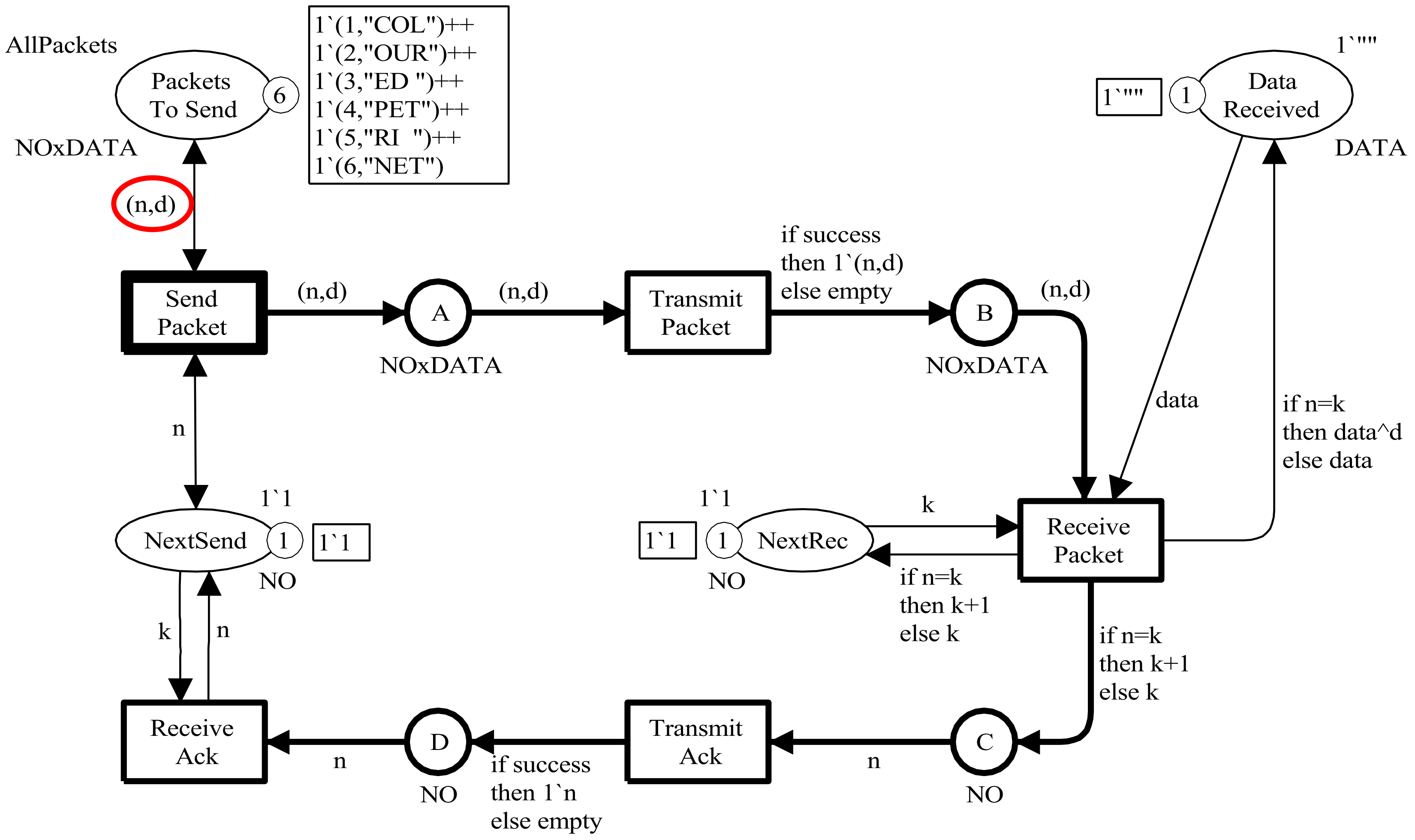

```
var n,k      : NO;
var d,data   : DATA;

var ackpacks : ACKPACKS;
var datapacks: DATAPACKS;
```

Expressions and types

- The complete set of Standard ML expressions can be used in net inscriptions provided that they have the proper type
  - The type of an arc expression must be equal to the colour set of the place connected to the arc (or a multiset over the colour set of the place)
  - The type of an initial marking must be equal to the colour set of the place (or a multiset over the colour set of the place)
  - A guard must be a Boolean expression (or a list of Boolean expressions)
- The CPN ML type system checks that all net inscriptions are type consistent and satisfies the above type constraints
- This is done by automatically inferring the types of expressions

Example of type checking



Type checking of (n,d)

```
colset NOxDATA = product NO * DATA;
```

```
var n : NO;
var d : DATA;
```

Arc expression

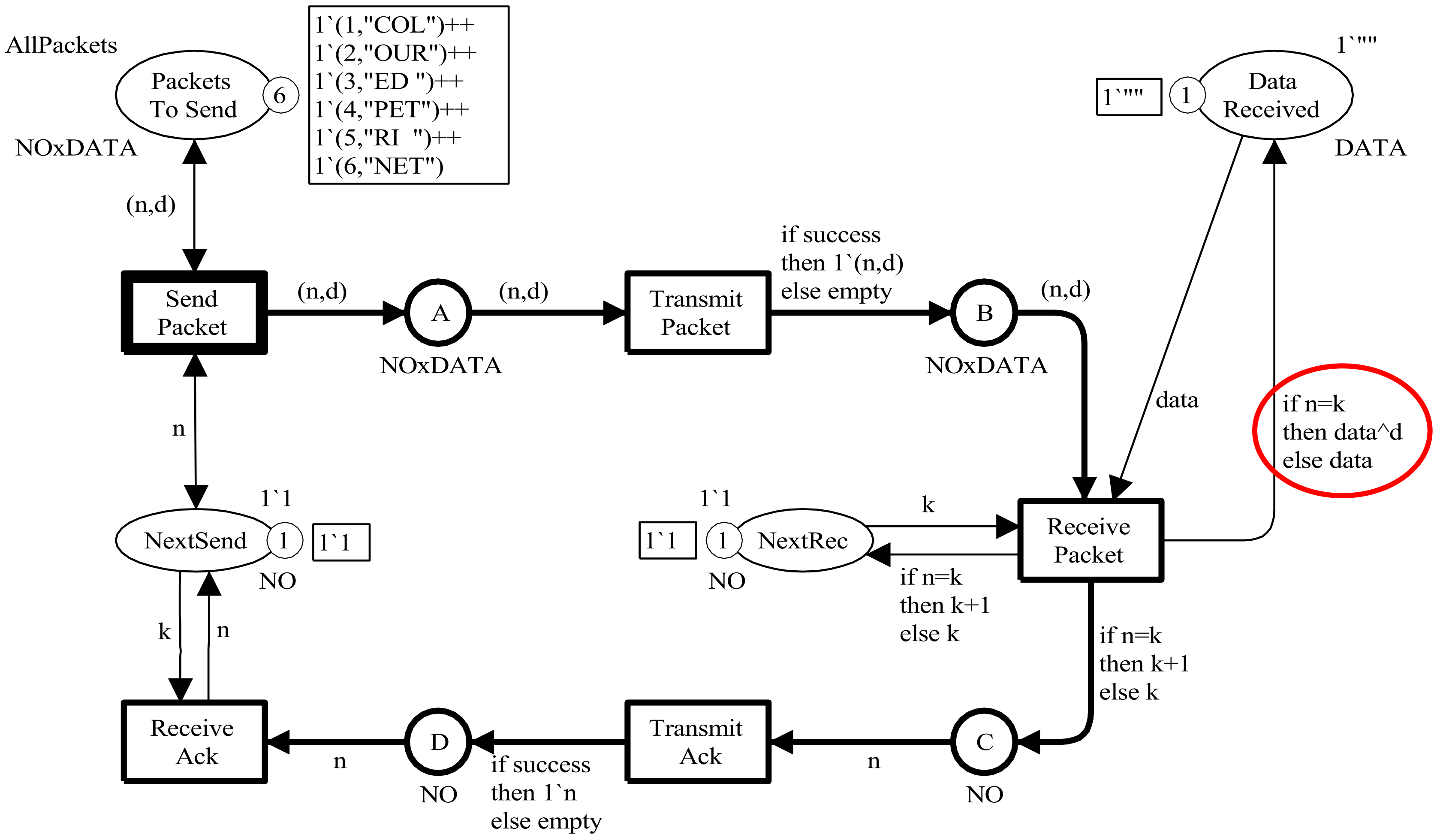
`(n,d)` `NO * DATA`

`NO` `n` `d` `DATA`

Sub-expressions

- (n,d) is type consistent and of type NO * DATA which is the colour set of the connected place

Second example of type checking



AARHUS UNIVERSITY

Kurt Jensen
Lars M. Kristensen

Type checking of if expression

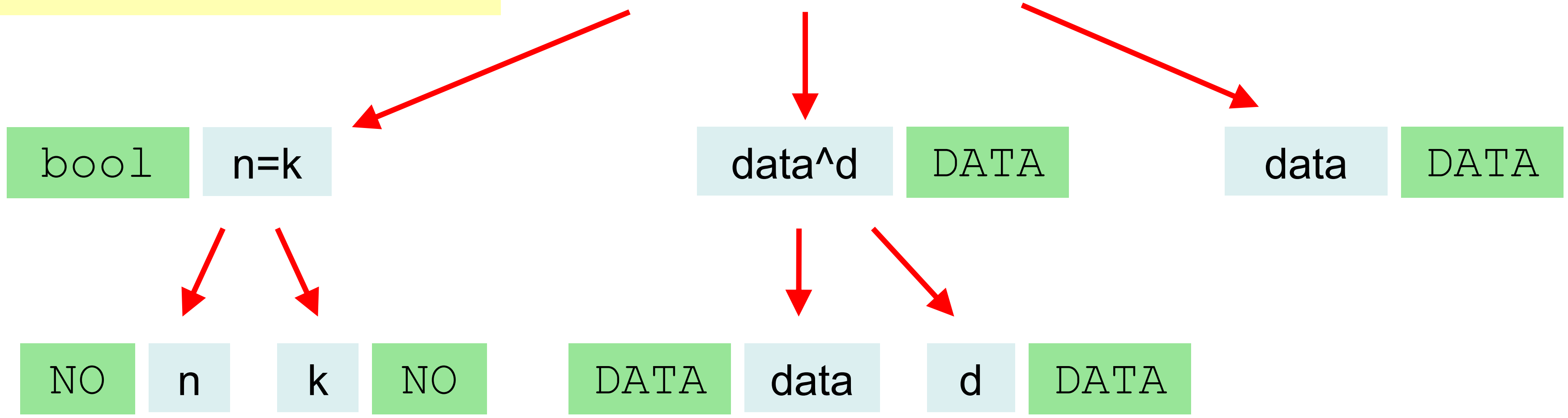
```
colset DATA = string;
```

```
var n,k : NO;
var d,data : DATA;
```

Arc expression

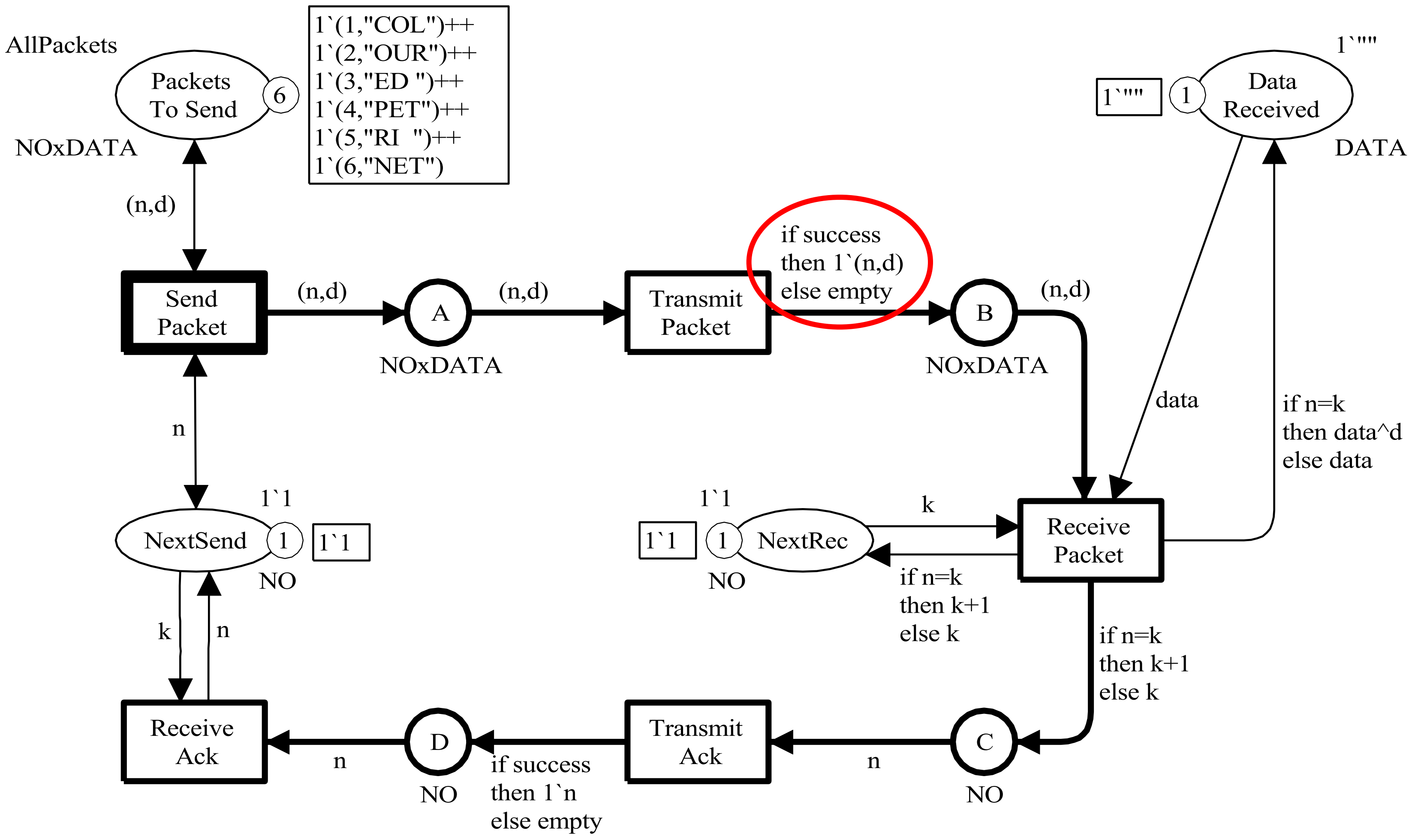
```
if n=k
then data^d
else data
```

DATA



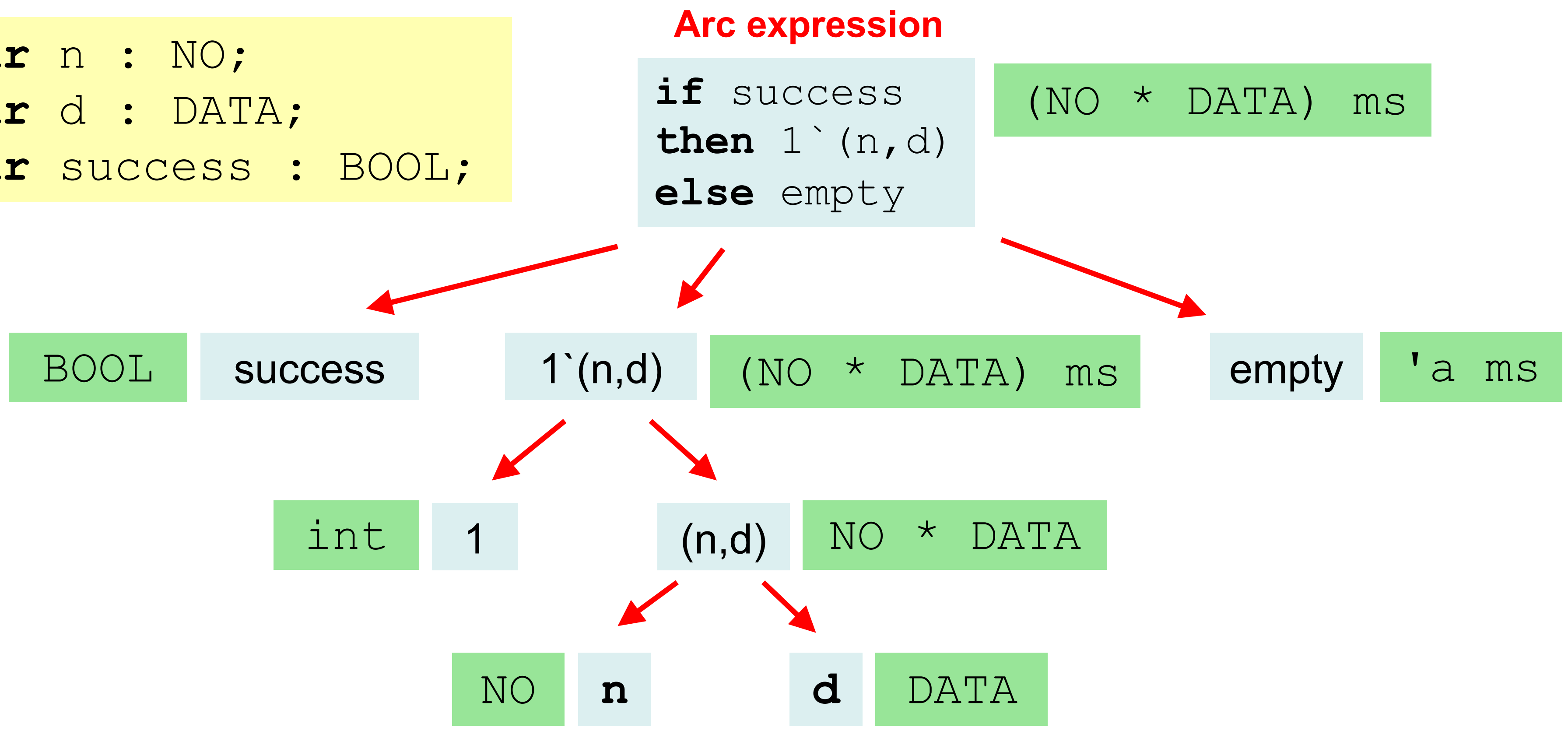
- If expression is type consistent and of type DATA which is the colour set of the connected place

Third example of type checking



Type checking of if expression

```
var n : NO;
var d : DATA;
var success : BOOL;
```



- If expression is type consistent and of type NO * DATA ms - multisets over the colour set of the connected place

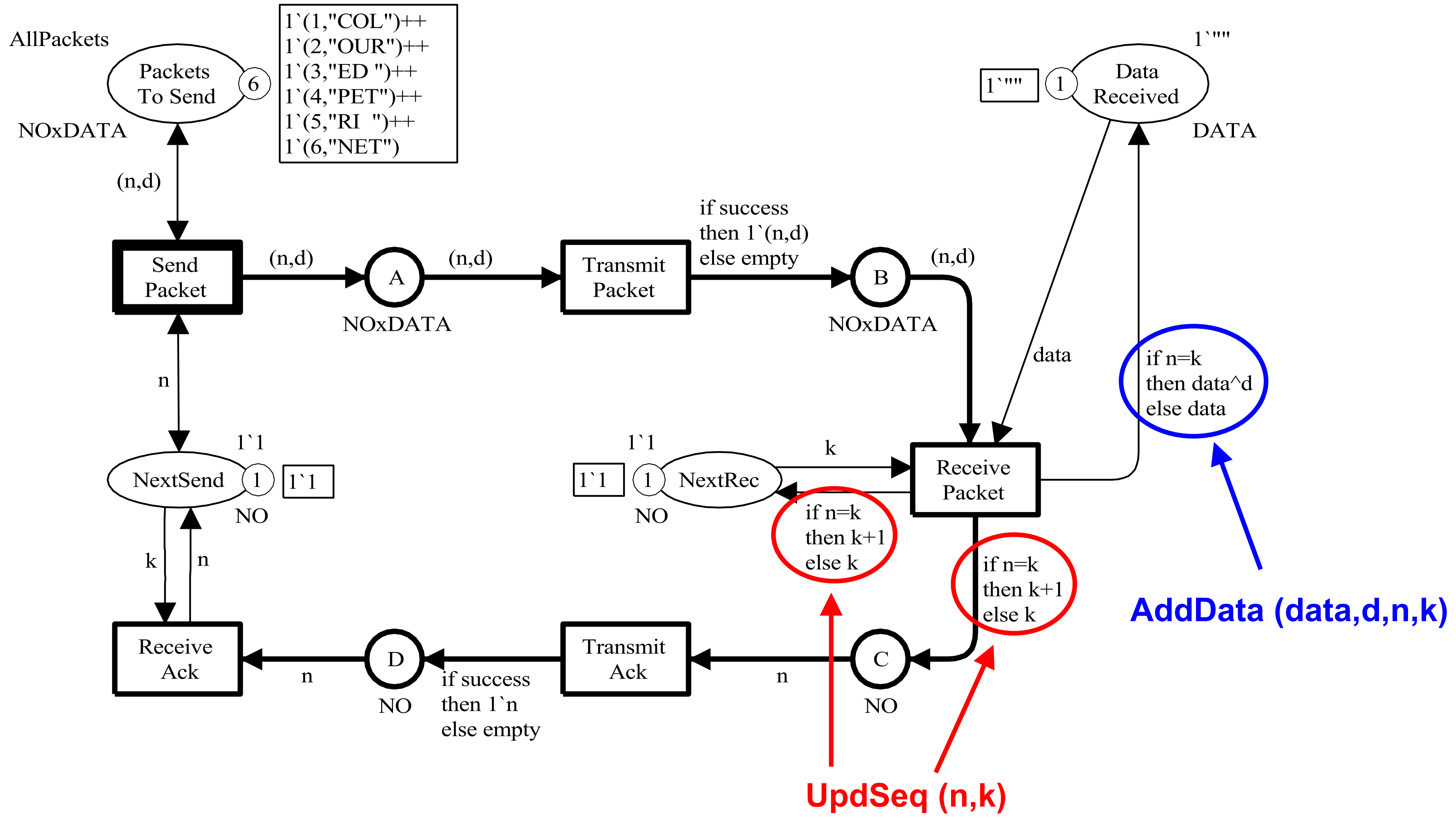
Functions

- Functions can be used in all kinds of net expressions
  - Guards
  - Arc expressions
  - Initial markings

- Functions are used when
  - Complex expressions takes up too much space in the graphical representation.
  - Same functionality is required in different parts of the model

- Functions make CPN models easier to read and maintain

Simple protocol



Definition of two functions

```
fun UpdSeq (n,k) = if n=k
                   then k+1
                   else k;
```

Function Name **Function Parameter**

```
fun AddData (data,d,n,k) = if n=k
                           then data^d
                           else data;
```

- All functions in Standard ML take a single parameter which may be a tuple

Inference of function type

```
fun UpdSeq (n,k) = if n=k
                   then k+1
                   else k;
```

n : INT

k : INT

```
int * int -> int
```

Function evaluates to an integer

- The variables n and k are local to the function definition
- They should not be confused with the variables n and k of type NO used as arguments in the function call

Inference of function type

```
fun AddData (data,d,n,k) = if n=k
                           then data^d
                           else data;
```

n and k must have the same type

data : string
d : string

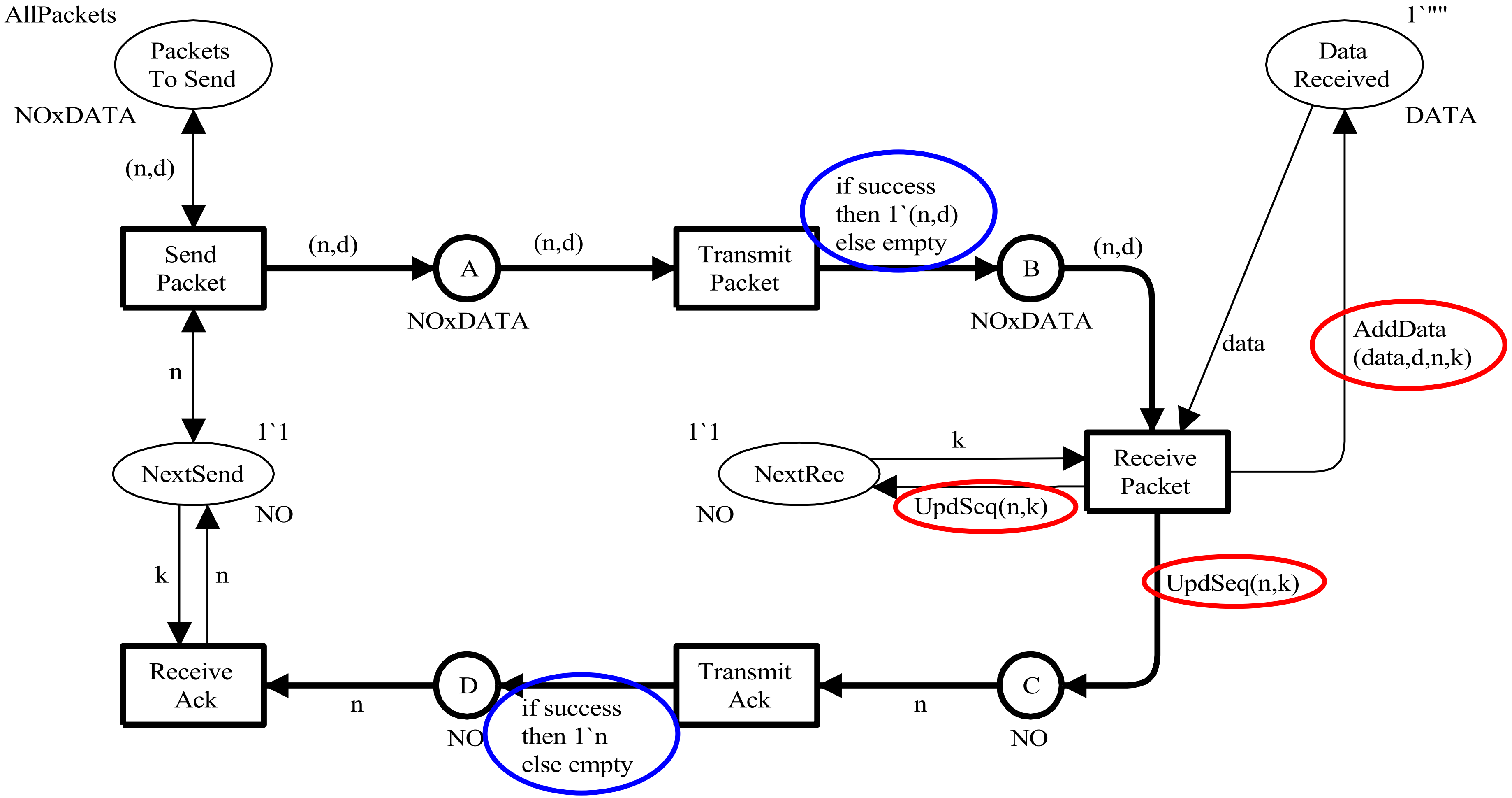
Function evaluates to a string

```
string * string * ''a * ''a -> string
```

Type variable:
Some type with equality operation

- Polymorphic function
- Can be called with different types of arguments

CPN model with functions



Exploiting polymorphism

```
fun Transmit (success,pack) = if success
                              then 1`pack
                              else empty;
```

success : bool

```
bool * 'a -> 'a ms
```

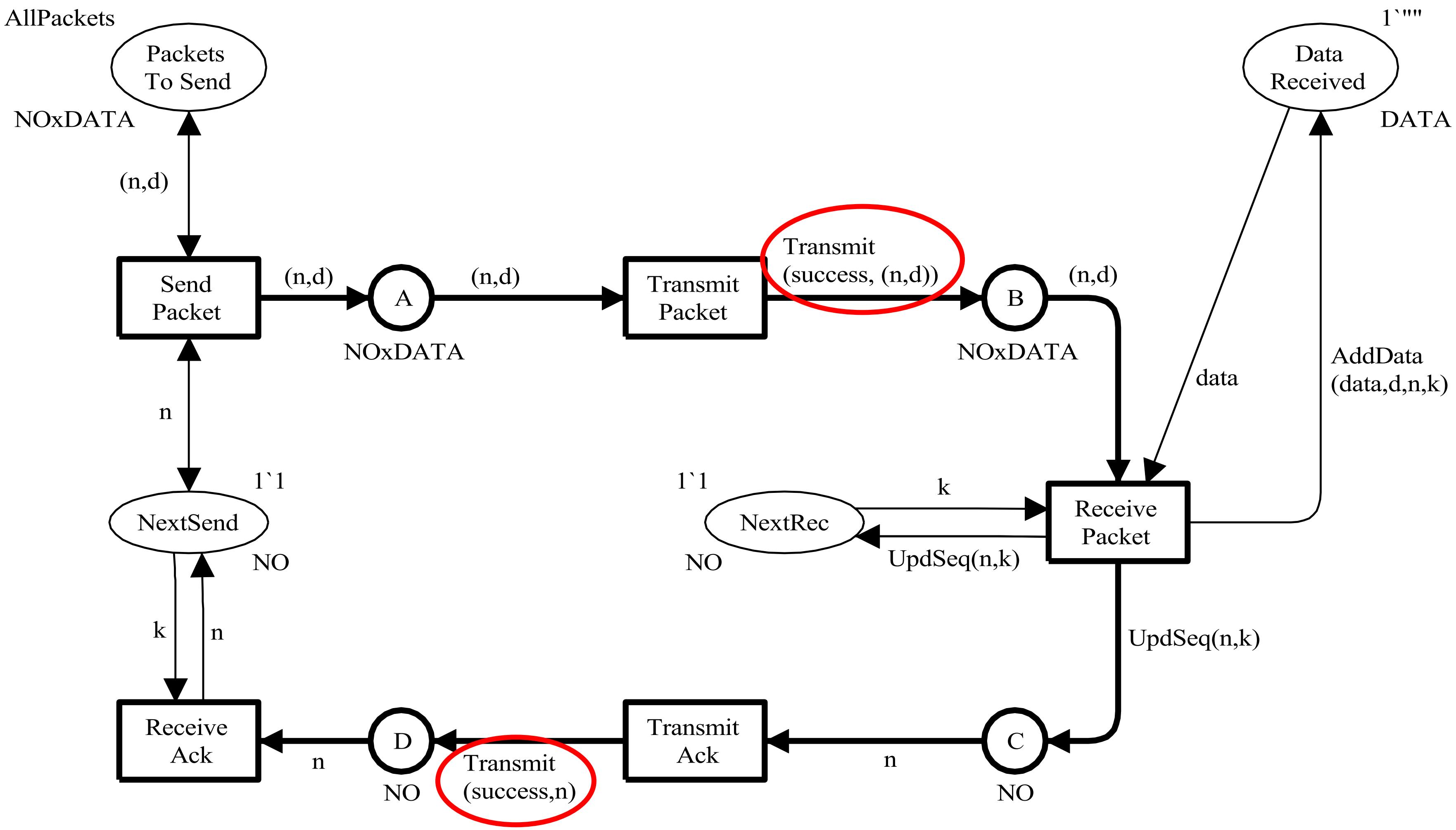
Function evaluates to a multiset over the type of pack

Multiset

Type variable: Some type where equality operation not required

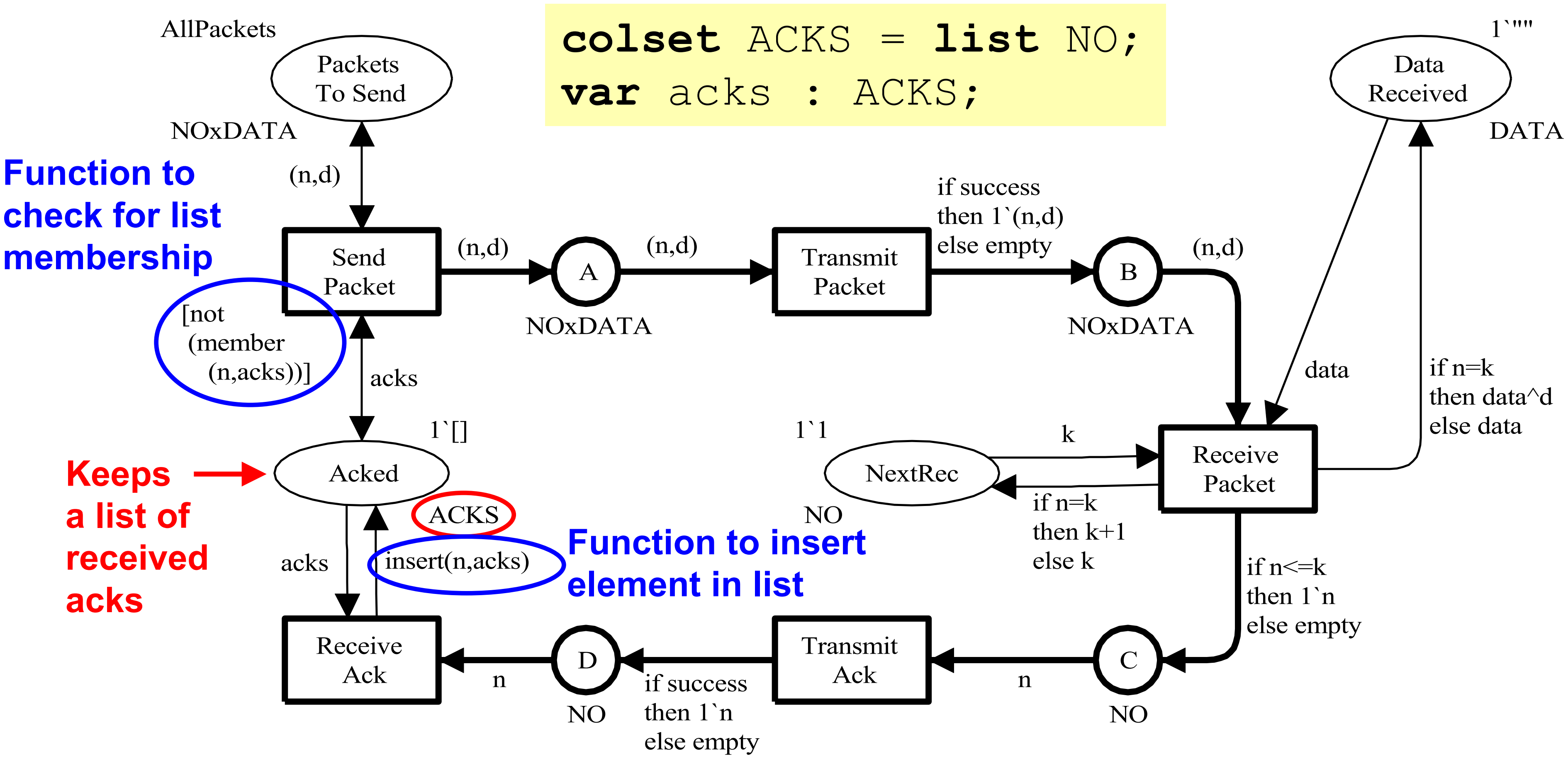
- Polymorphic function
- Can be called with different types of arguments
  - Transmit (success,(n,d)) ← To transmit data packets
  - Transmit (success,n) ← To transmit acknowledgments

CPN model with polymorphic function



Revised protocol

- Sender can send any unacknowledged data packet



Function member

- Checks whether the element e is present in the list l

```
fun member (e,l) =
      if l = []
      then false
      else
          if (e = List.hd l)
          then true
          else member (e,List.tl l);
```

Library functions

Recursive call

Function insert

- Inserts the element e in the list l if it is not already present

```
fun insert (e,l) =
        if member (e,l)
        then l
        else e::l;
```

Uses the member function

Local environments

- Can be introduced using a let expression

Comments

```
fun member (e,l) =
    if l = []
    then false (* if list empty, e is not a member *)
    else       (* list is not empty *)
      let
        (* extract head and tail of the list *)
        val head = List.hd l
        val tail = List.tl l
      in
        if e = head
        then true     (* e was equal to the head *)
        else member (e,tail) (* check the tail *)
      end;
```

Even short ML functions can be tricky to read and understand
Hence it is a very good idea to use comments

Higher-order functions

- A function taking a function as parameter or returning a function is a higher-order function
- Member is a special case of determining whether there exists an element in the list `l` satisfying a Boolean predicate `p`

```
fun exists (p,l) =        (''a -> bool) * ''a list -> bool
    if l = []
    then false
    else
       if p (List.hd l)
       then true
       else exists (p,List.tl l);
```

```
fun member (e,l) =        ''a * ''a list -> bool
    let
       fun equal x = (e=x)
    in
       exists (equal,l)
    end;
```

Anonymous and curried functions

- Anonymous functions are specified without an explicit name

```
fn x => (e=x);
```

```
fun member (e,l)  = exists (fn x => (e=x),l);
```

- Curried functions take their parameters one at a time

```
fun equal e x = (e=x);
```

```
''a -> ''a -> bool
```

```
equal e;
```

```
''a -> bool
```

```
fun member (e,l)  = exists (equal e,l);
```

Patterns in function applications

- Expressions are built from constants, constructors, and variables
- Can be matched with arguments to bind values to the variables

```
fun member (e,l) =
    if l = []
    then false
    else
       if (e = List.hd l)
       then true
       else member (e,List.tl l);
```

Pattern

```
member (2,[1,3,4])
```

Function call

- The argument `(2,[1,3,4])` is matched with the pattern `(e,l)`

Patterns in function definitions

Not used

Matches the empty list

```
fun member (e,[])    = false
 |  member (e,x::l) =
    if (x = e)
          then true
          else member (e,l);
```

Matches a non-empty list

Wilcard (matches everything)

```
fun member (_,[])    = false
 |  member (e,x::l) =
    if (x = e)
          then true
          else member (e,l);
```


Kurt Jensen
Lars M. Kristensen

Patterns in case expressions

- Case expressions can be used instead of nested if expressions

```
case res of
    success   => 1`p
  | duplicate => 2`p
  | failure   => empty;
```

Three patterns

```
if res = success
then 1`pack
else if res = duplicate
     then 2`pack
     else empty;
```

- Alternative

```
(case res of
     success => 1
   | duplicate => 2
   | failure => 0)`pack
```

Common patterns pitfalls

- Redundant match

```
case res of
    _              => empty
  | success        => 1`p
  | duplicate      => 2'p;
```

Warning!

Programming error

- Everything will match the first clause.
- The other clauses will never be used

- Non-exhaustive match

```
fun member (e,x::l) =
    if (e = x)
    then true
    else member (e,l);
```

Warning! – Is it wise to ignore the warning?

NO

- Recursion will always end with a call involving the empty list

Questions