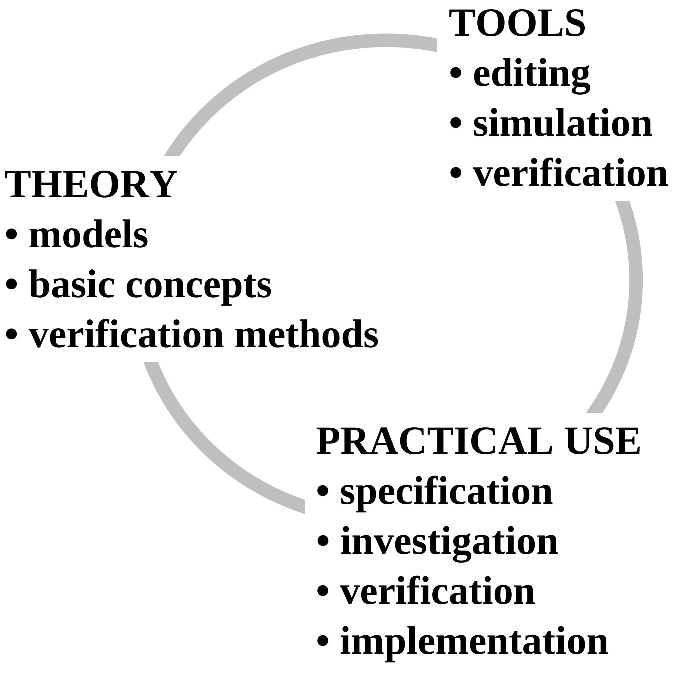
Coloured Petri Nets

Kurt Jensen

Computer Science Department
University of Aarhus

Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark

Phone: +45 89 42 32 34
Telefax: +45 89 42 32 55
E-mail: kjensen@daimi.aau.dk
URL: http://www.daimi.aau.dk/~kjensen



TOOLS
- **editing**
- **simulation**
- **verification**

THEORY
- **models**
- **basic concepts**
- **verification methods**

PRACTICAL USE
- **specification**
- **investigation**
- **verification**
- **implementation**

Part 1: Introduction to CP-nets

An ordinary Petri net (PT-net) has *no types* and *no modules:*

- Only one kind of tokens and the net is flat.

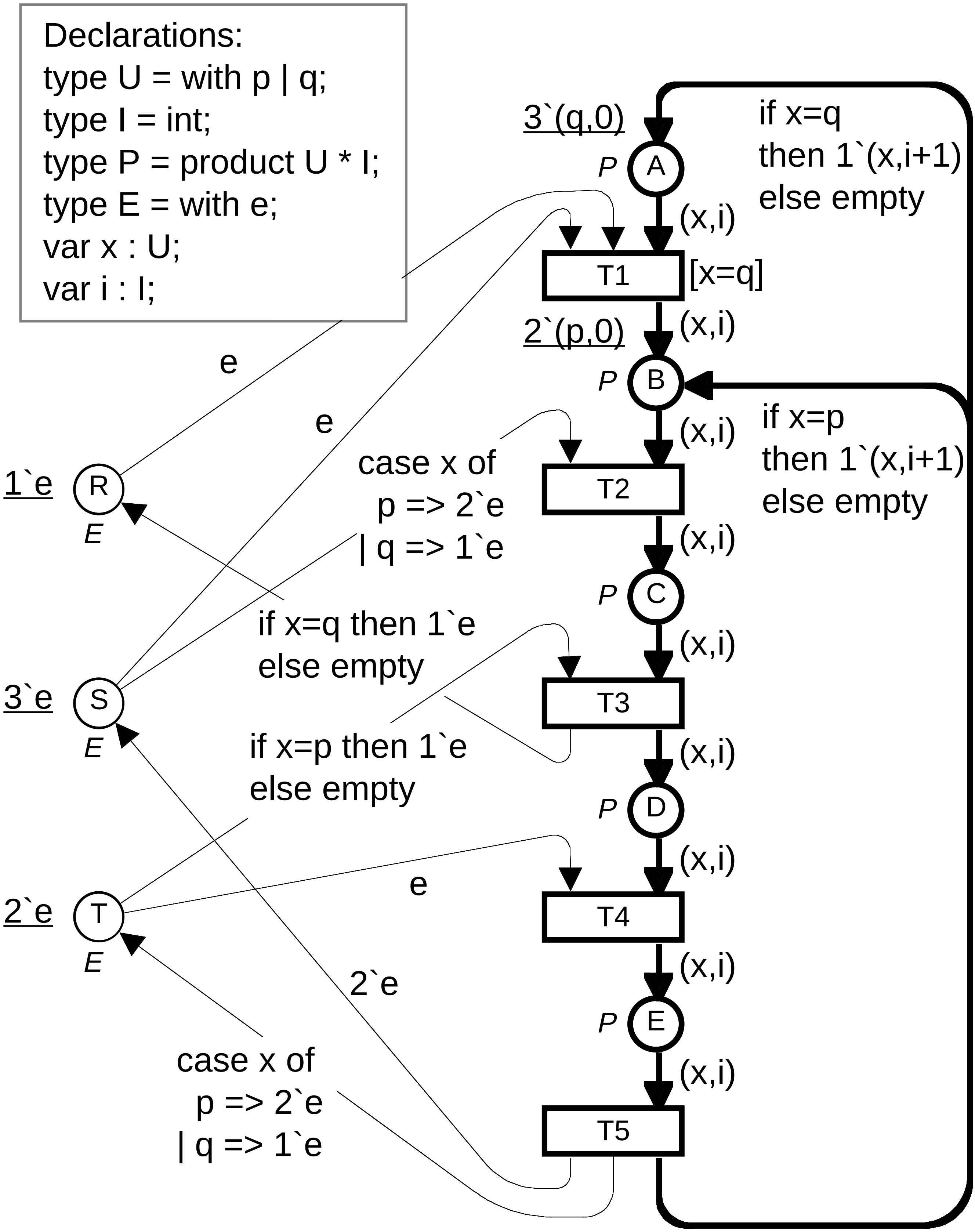
With Coloured Petri Nets (CP-nets) it is possible to use *data types* and complex *data manipulation:*

- Each token has attached a data value called the *token colour*.
- The token colours can be *investigated* and *modified* by the occurring transitions.

With CP-nets it is possible to make *hierarchical* descriptions:

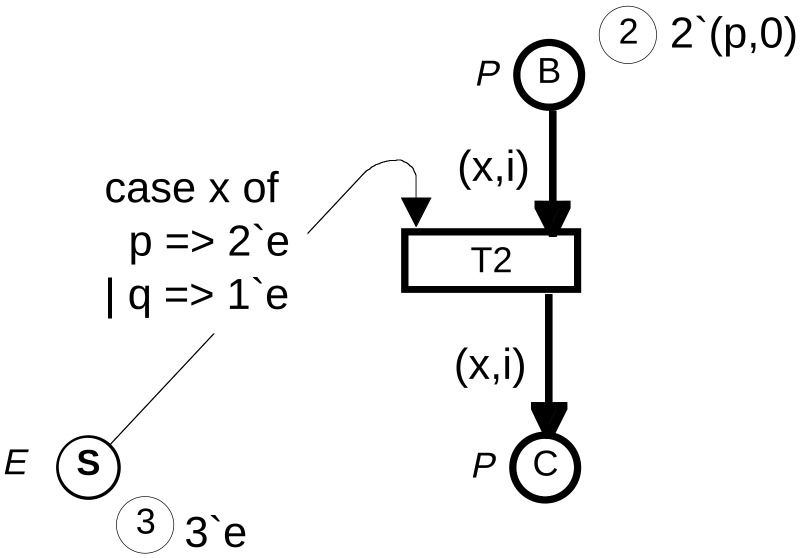
- A large model can be obtained by *combining* a set of *submodels.*
- Well-defined *interfaces* between the submodels.
- Well-defined *semantics* of the combined model.
- *Submodels* can be reused.

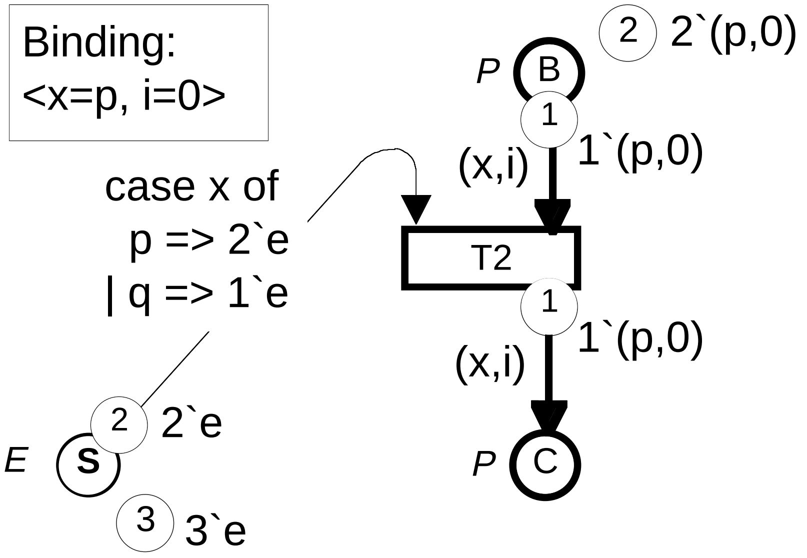
Resource allocation example

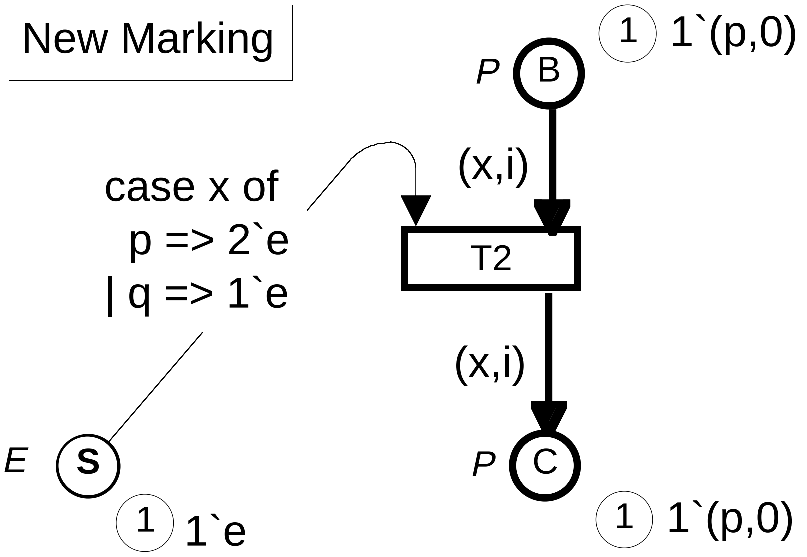


```
Declarations:
type U = with p | q;
type I = int;
type P = product U * I;
type E = with e;
var x : U;
var i : I;
```

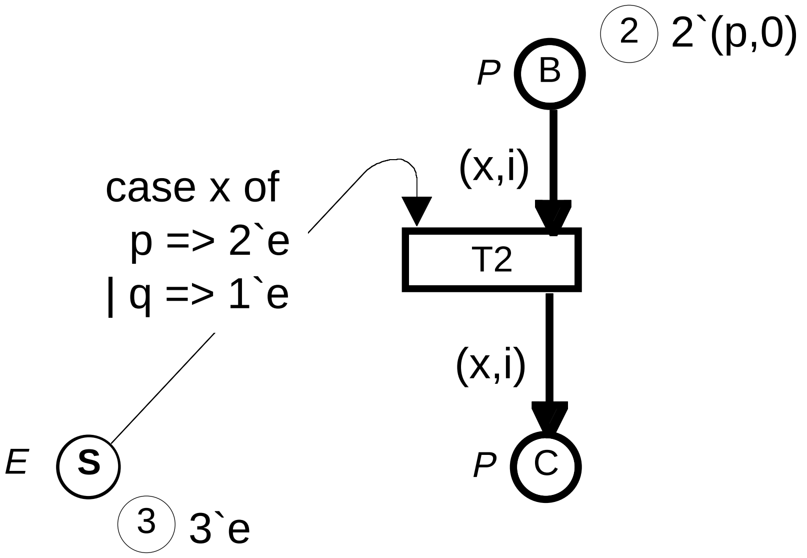
Occurrence of enabled binding

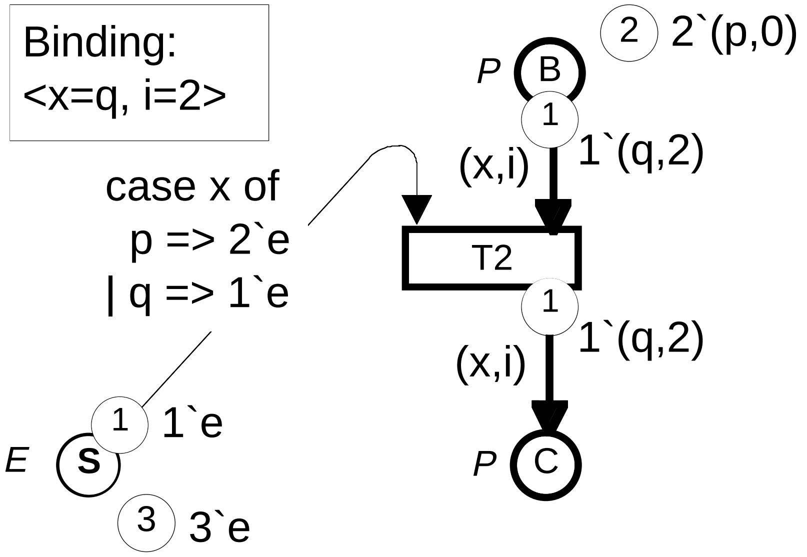



Binding:
<x=p, i=0>


New Marking

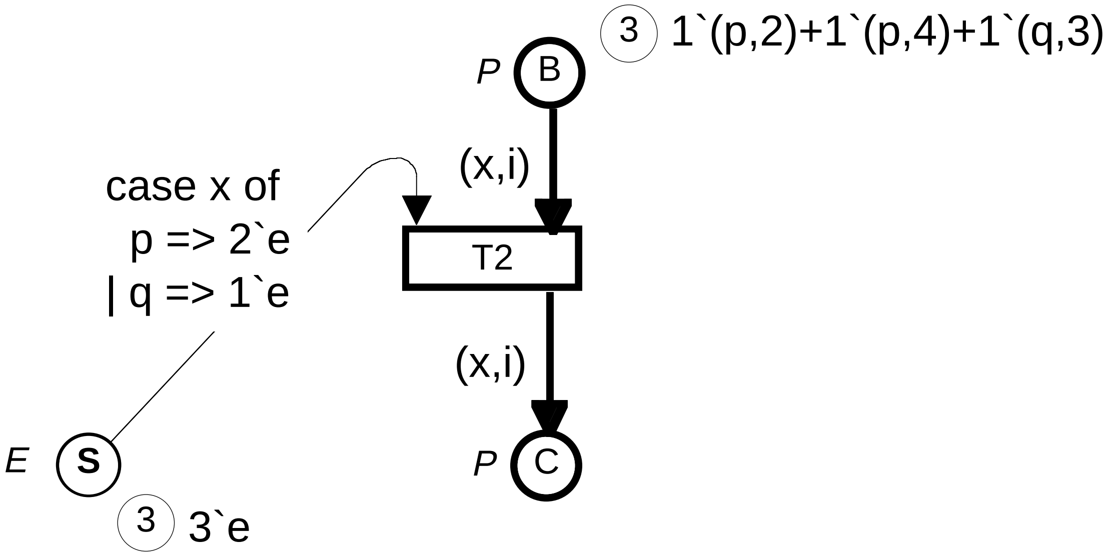
Binding which is not enabled

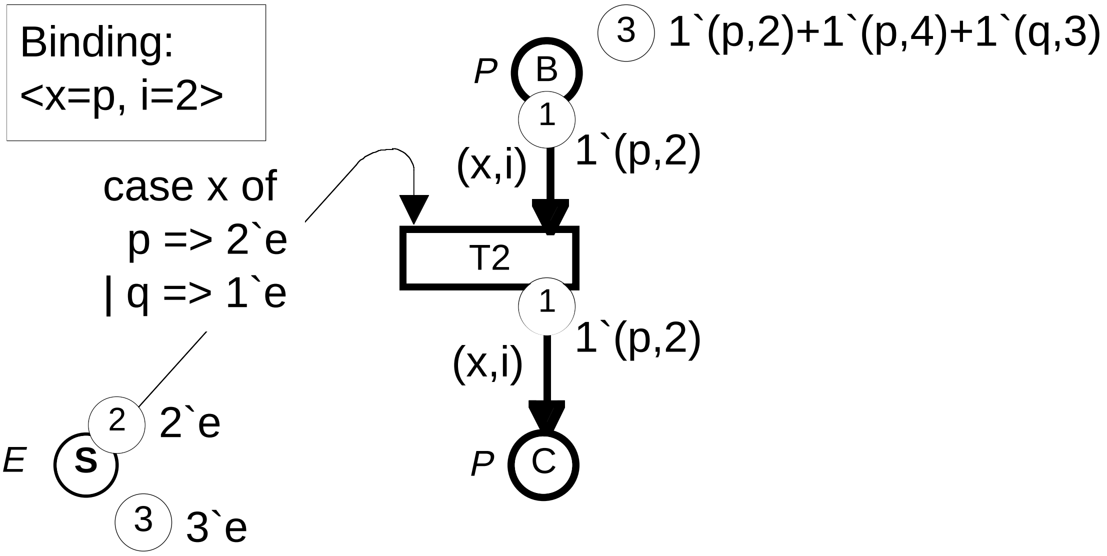



Binding:
<x=q, i=2>

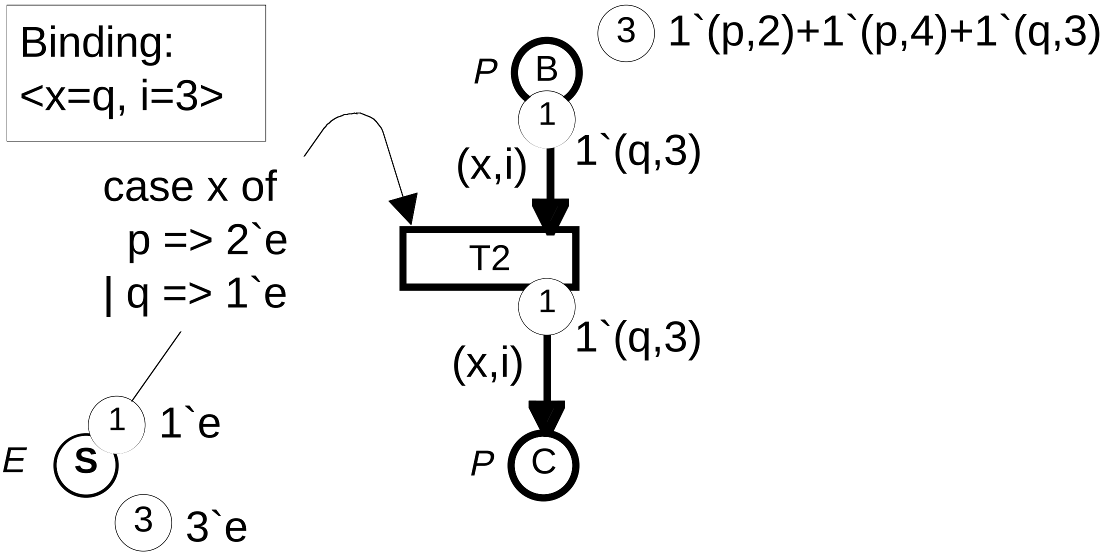
Binding cannot occur

A more complex example



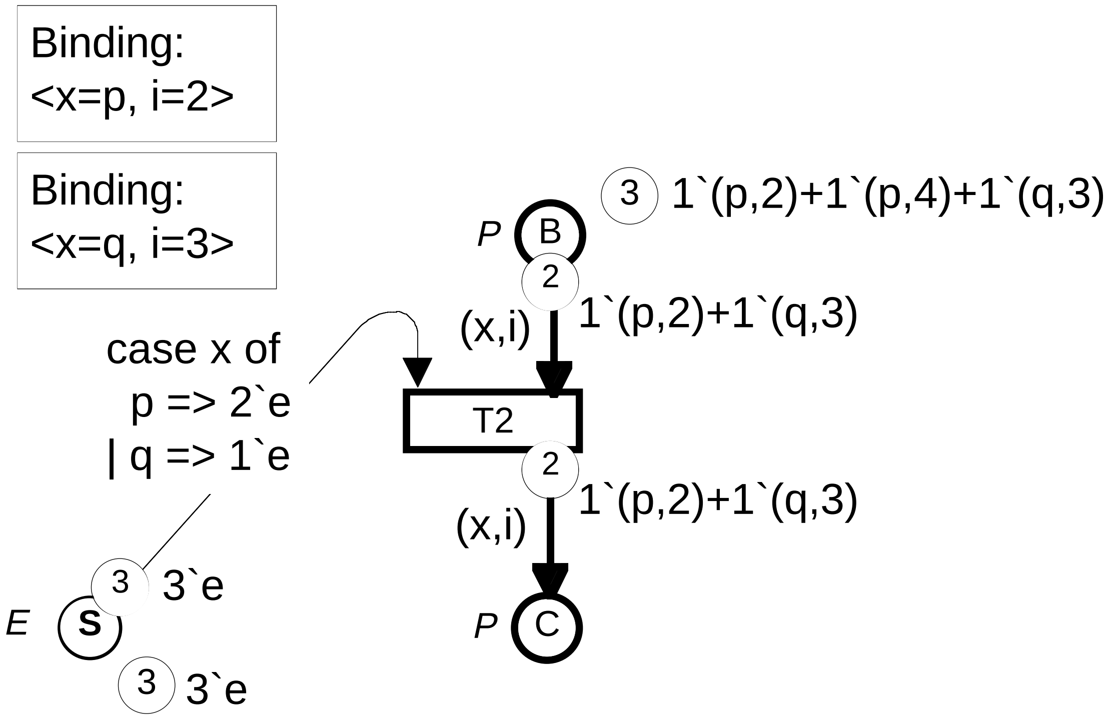


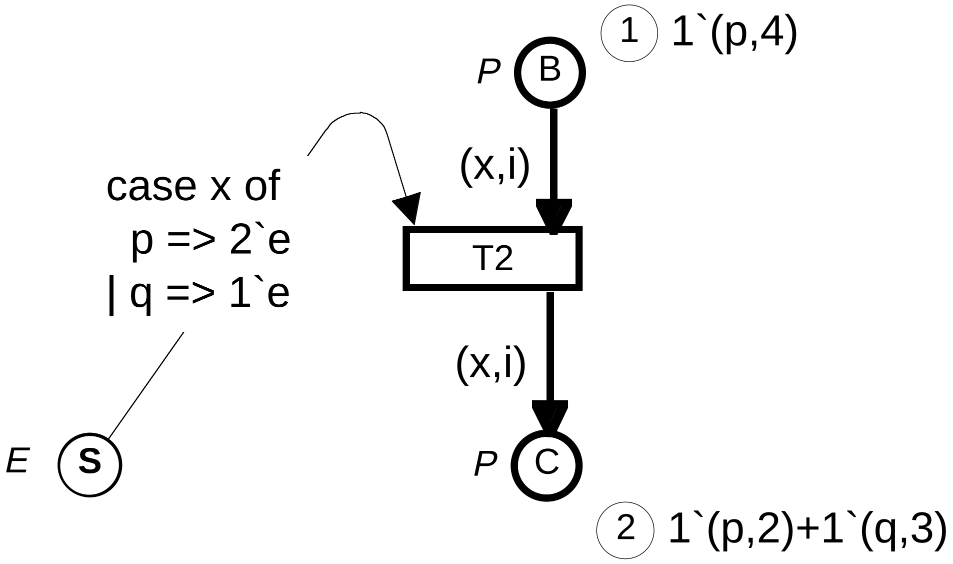
Binding:
<x=p, i=2>



Binding:
<x=q, i=3>

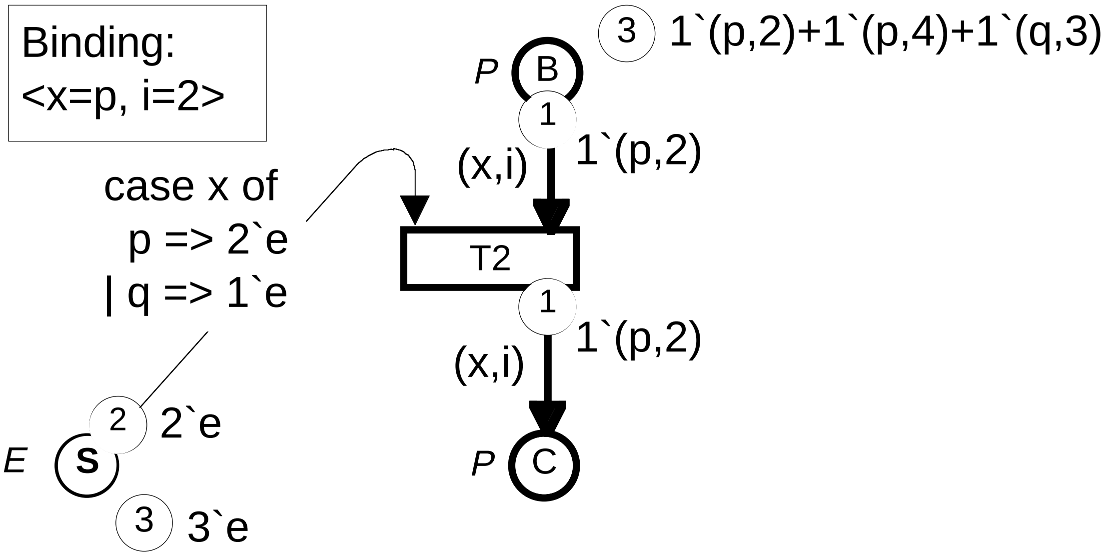
Concurrency

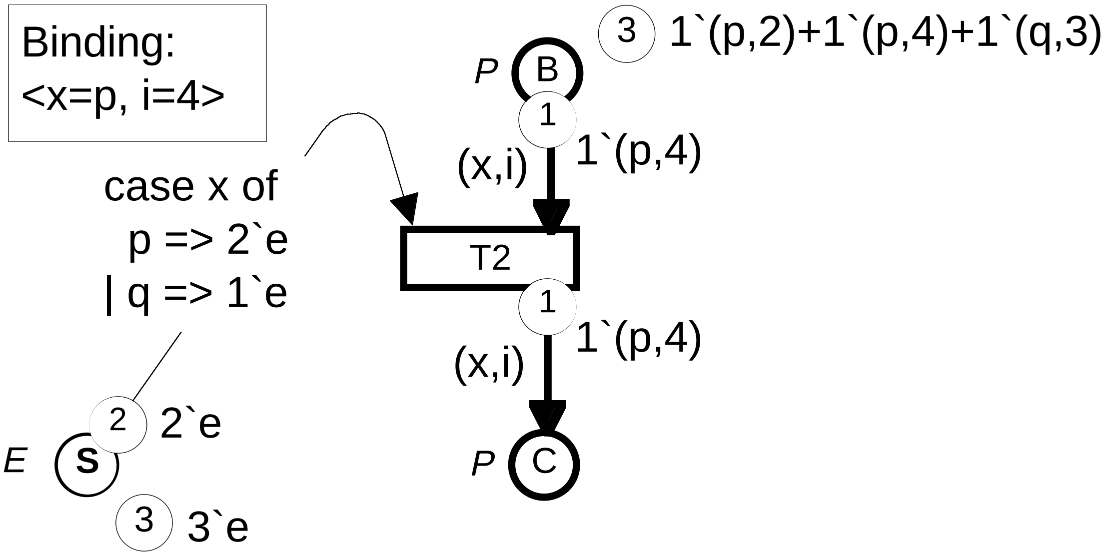




- The two bindings may occur *concurrently*.
- This is possible because they use *different tokens*.

Conflict





- These two bindings cannot occur *concurrently*.
- The reason is that they need the *same tokens*.

Resource allocation system

Two kinds of *processes:*

- Three cyclic q-processes (states A,B,C,D and E).
- Two cyclic p-processes (states B,C,D and E).

Three kinds of *resources:*

- Represented by the places R, S and T.

During a *cycle* a process *reserves* some resources and *releases* them again:

- Tokens are *removed* from and *added* to the resource places R, S and T.

A *cycle counter* is increased each time a process completes a full cycle.

It is rather straightforward to prove that the resource allocation system *cannot deadlock.*

- What happens if we add an additional token to place S – i.e., if we start with four S-resources instead of three?

Coloured Petri Nets

Declarations:

- *Types*, *functions*, *operations* and *variables*.

Each *place* has the following inscriptions:

- *Name* (for identification).
- *Colour set* (specifying the type of tokens which may reside on the place).
- *Initial marking* (multi-set of token colours).

Each *transition* has the following inscriptions:

- *Name* (for identification).
- *Guard* (boolean expression containing some of the variables).

Each *arc* has the following inscriptions:

- *Arc expression* (containing some of the variables). When the arc expression is evaluated it yields a multi-set of token colours.

Enabling and occurrence

A *binding* assigns a *colour* (i.e., a value) to each *variable* of a transition.

A *binding element* is a pair (t,b) where t is a *transition* while b is a *binding* for the variables of t. Example: (T2,<x=p, i=2>).

A binding element is *enabled* if and only if:

- There are *enough tokens* (of the correct colours on each input-place).
- The *guard* evaluates to true.

When a binding element is enabled it may *occur:*

- A multi-set of tokens is *removed* from each input-place.
- A multi-set of tokens is *added* to each output-place.

A binding element may occur *concurrently* to other binding elements – iff there are so many tokens that each binding element can get its "own share".

Main characteristics of CP-nets

Combination of *text* and *graphics.*

Declarations and *net inscriptions* are specified by means of a formal language, e.g., a *programming language.*

- Types, functions, operations, variables and expressions.

Net structure consists of places, transitions and arcs (forming a bi-partite graph).

- To make a CP-net *readable* it is important to make a nice graphical layout.
- The graphical layout has *no formal meaning*.

CP-nets have the same kind of *concurrency properties* as Place/Transition Nets.

Formal definition of CP-nets

Definition: A Coloured Petri Net is a tuple CPN = (Σ, P, T, A, N, C, G, E, I) satisfying the following requirements:

(i) Σ is a finite set of non-empty types, called **colour sets**.

(ii) P is a finite set of **places**.

(iii) T is a finite set of **transitions**.

(iv) A is a finite set of **arcs** such that:

- $P \cap T = P \cap A = T \cap A = \emptyset$.

(v) N is a **node** function. It is defined from A into $P \times T \cup T \times P$.

(vi) C is a **colour** function. It is defined from P into Σ.

(vii) G is a **guard** function. It is defined from T into expressions such that:

- $\forall t \in T$: [Type(G(t)) = Bool $\wedge$ Type(Var(G(t))) $\subseteq \Sigma$].

(viii) E is an **arc expression** function. It is defined from A into expressions such that:

- $\forall a \in A$: [Type(E(a)) = $C(p(a))_{MS}$ $\wedge$ Type(Var(E(a))) $\subseteq \Sigma$]

where p(a) is the place of N(a).

(ix) I is an **initialization** function. It is defined from P into closed expressions such that:

- $\forall p \in P$: [Type(I(p)) = $C(p)_{MS}$].

Formal definition of behaviour

Definition: A **step** is a multi-set of binding elements.

A step Y is **enabled** in a marking M iff the following property is satisfied:

$$\forall p \in P: \sum_{(t,b) \in Y} E(p,t)\langle b\rangle \leq M(p).$$

When a step Y is enabled in a marking M_1 it may **occur**, changing the marking M_1 to another marking M_2, defined by:

$$\forall p \in P: M_2(p) = (M_1(p) - \sum_{(t,b) \in Y} E(p,t)\langle b\rangle) + \sum_{(t,b) \in Y} E(t,p)\langle b\rangle.$$

The first sum is called the **removed** tokens while the second is called the **added** tokens. Moreover we say that M_2 is **directly reachable** from M_1 by the occurrence of the step Y, which we also denote: $M_1 [Y\rangle M_2$.

An **occurrence sequence** is a sequence of markings and steps:

$$M_1 [Y_1\rangle M_2 [Y_2\rangle M_3 \ldots M_n [Y_n\rangle M_{n+1}$$

such that $M_i [Y_i\rangle M_{i+1}$ for all $i \in 1..n$. We then say that M_{n+1} is **reachable** from M_1. We use $[M\rangle$ to denote the set of markings which are reachable from M.

Formal definition

The existence of a *formal definition* is very important:

- It is the basis for *simulation*, i.e., execution of the CP-net.
- It is also the basis for the *formal verification* methods (e.g., state spaces and place invariants).
- Without the formal definition, it would have been impossible to obtain a *sound* net class.

It is *not necessary* for a *user* to know the formal definition of CP-nets:

- The correct *syntax* is checked by the CPN editor, i.e., the computer tool by which CP-nets are constructed.
- The correct use of the *semantics* (i.e., the enabling rule and the occurrence rule) is guaranteed by the CPN simulator and the CPN tools for formal verification.

High-level contra low-level nets

The relationship between CP-nets and Place/Transition Nets (PT-nets) is *analogous* to the relationship between high-level programming languages and assembly code.

- In theory, the two levels have exactly the same *computational power*.
- In practice, high-level languages have much more *modelling power* – because they have better structuring facilities, e.g., types and modules.

Each CP-net has an *equivalent* PT-net – and vice versa.

- This equivalence is used to derive the definition of *basic properties* and to establish the *verification methods*.
- In practice, we *never* translate a CP-net into a PT-net – or vice versa.
- Description, simulation and verification are done *directly* in terms of CP-nets.

Other kinds of high-level nets

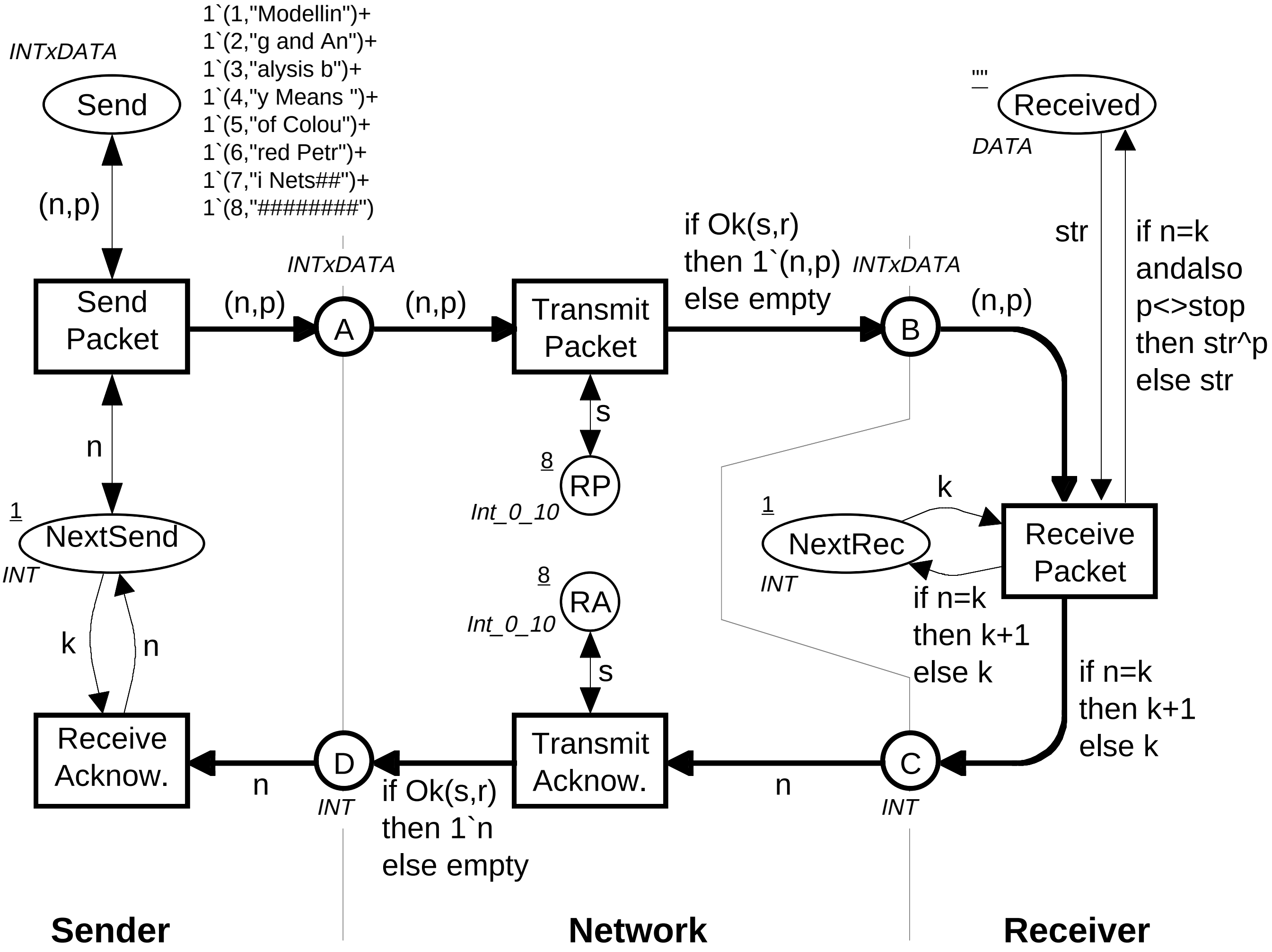
CP-nets have been developed from Predicate/Transition Nets.

- *Hartmann Genrich & Kurt Lautenbach*.
- With respect to *description* and *simulation* the two models are nearly identical.
- With respect to *formal verification* there are some differences.

Several other kinds of high-level Petri Nets exist.

- They all build upon the same *basic ideas*, but use *different languages* for declarations and inscriptions.
- A detailed comparison is outside the scope of this talk.

Simple protocol

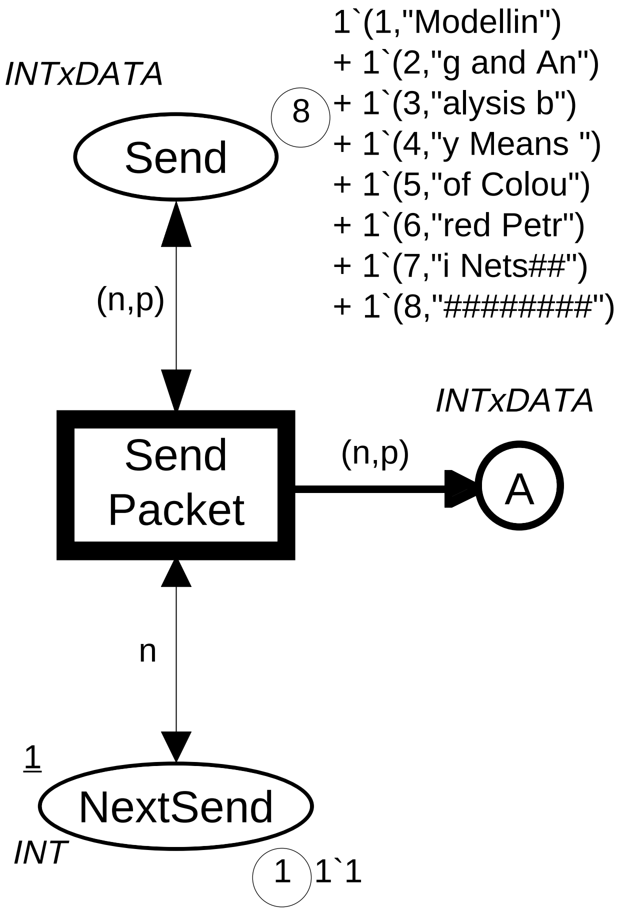


```
type INT = int;
type BOOL = bool;
type DATA = string;
type INTxDATA = product INT * DATA;
var n, k : INT;
var p,str : DATA;
val stop = "########";

type Int_0_10 = int with 0..10;
type Int_1_10 = int with 1..10;
var s : Int_0_10;
var r : Int_1_10;

fun Ok(s : Int_0_10, r : Int_1_10) = (r≤s);
```

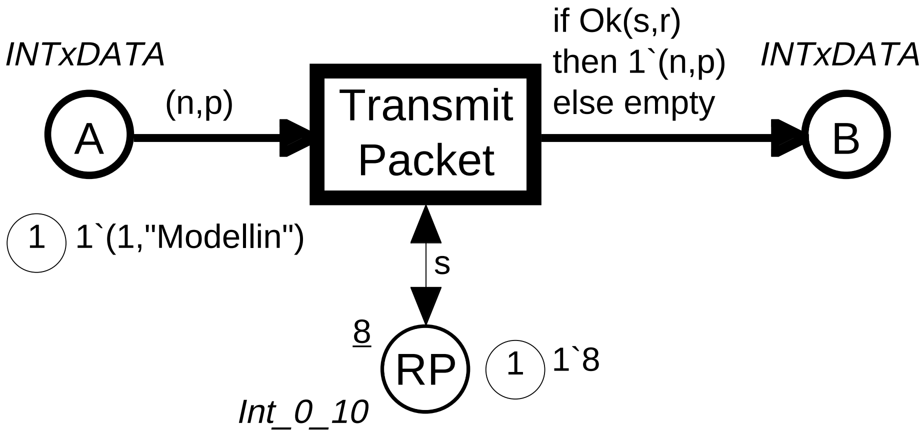
Send packet



Only the binding <n=1, p="Modellin"> is *enabled*.

- When the binding *occurs* it adds a token to place A. The token represents that the packet (1,"Modellin") is sent to the network.
- The packet is *not* removed from place *Send* and the *NextSend* counter is *not* changed.

Transmit packet



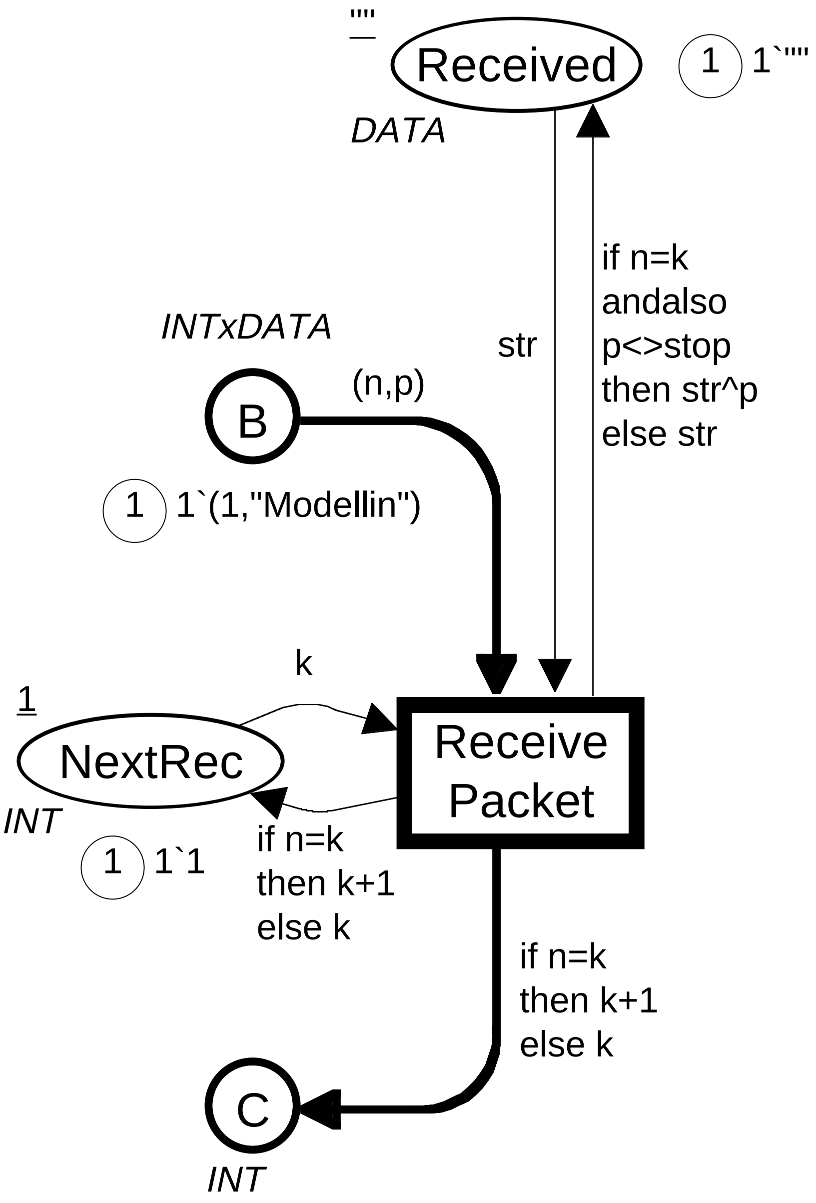
There are now *10 enabled bindings:*

- They are all of the form
  <n=1, p= "Modellin", s=8, r=…>.
- The variable r can take *10 different values*, because the type of r is defined to contain the integers 1..10.

The *function* Ok(s,r) checks whether r ≤ s.

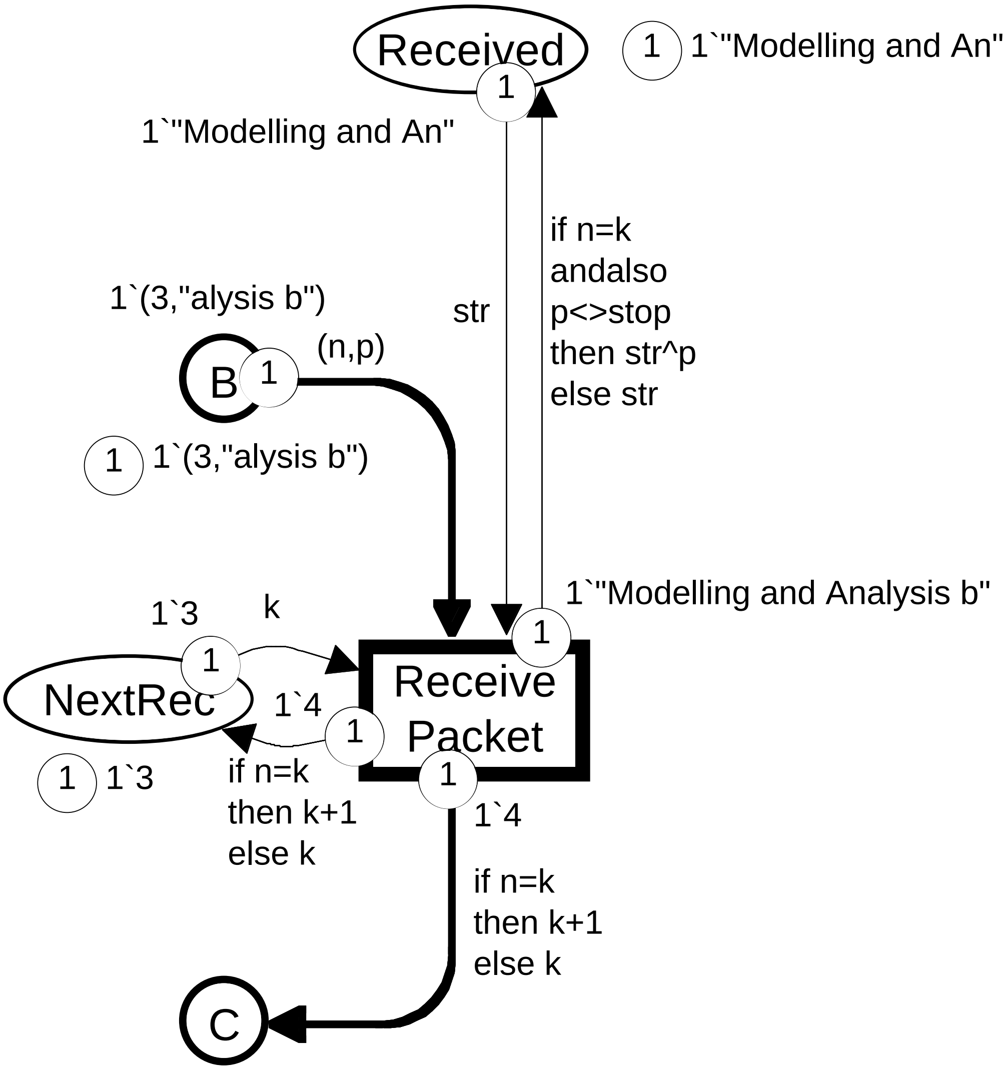
- For r ∈ 1..8, *Ok(s,r) = true*. This means that the token is moved from A to B, i.e., that the packet is *successfully transmitted* over the network.
- For r ∈ 9..10, *Ok(s,r) = false*. This means that no token is added to B, i.e., that the packet is *lost*.
- The CPN simulator make *random* choices between enabled bindings. Hence there is *80%* chance for *successful transfer*.

Receive packet



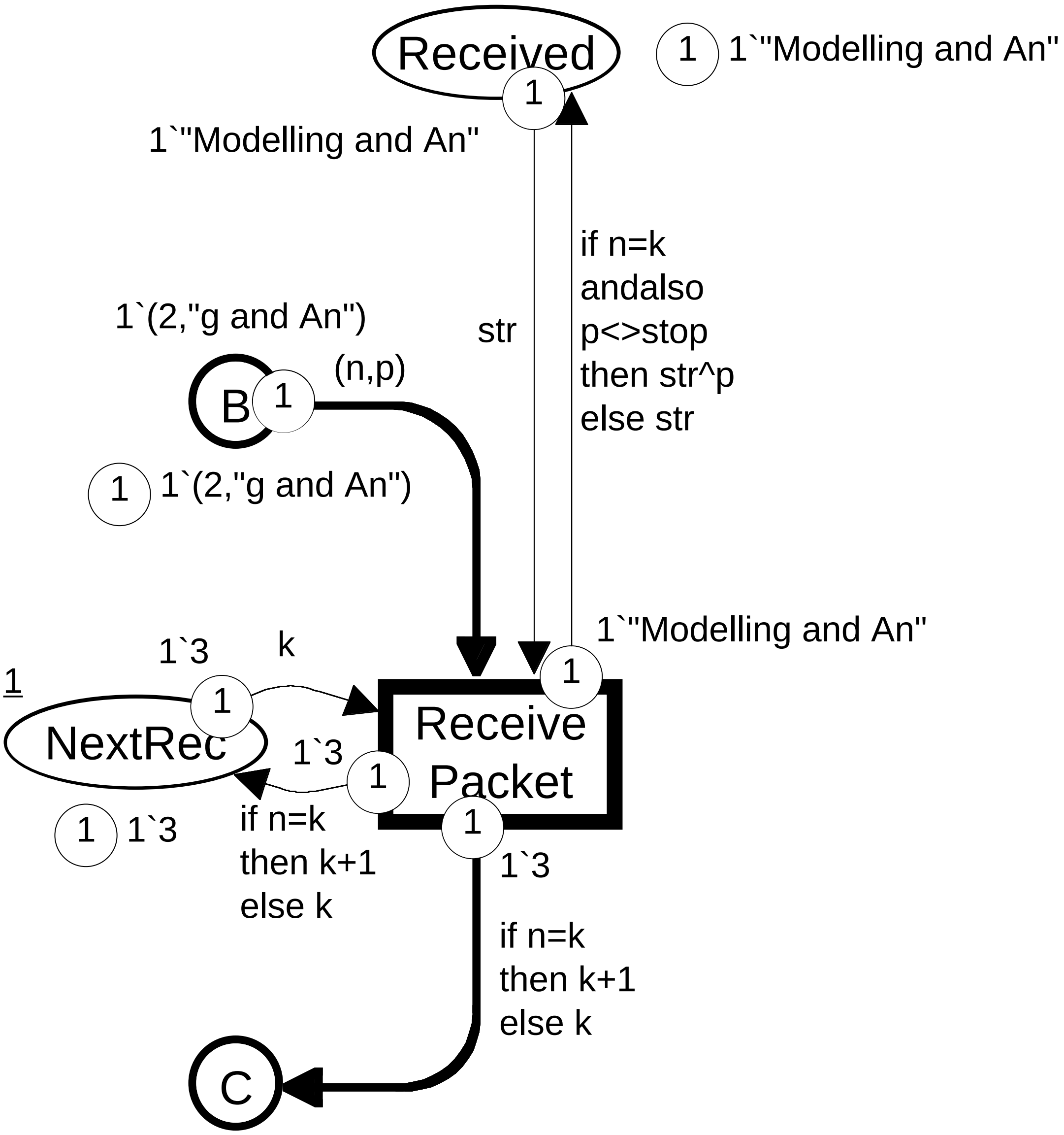
It is checked whether the number of the incoming packet n *matches* the number of the expected packet k.

Correct packet number



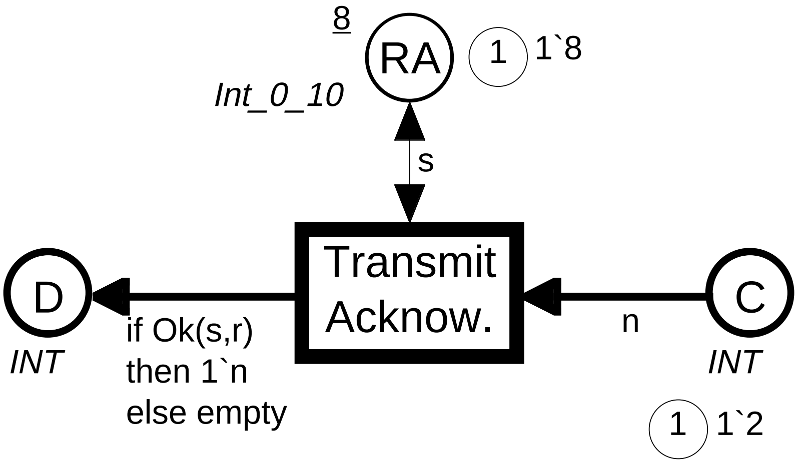
- The data in the packet is *concatenated* to the data already received.
- The *NextRec* counter is increased by one.
- An *acknowledgement message* is sent. It contains the number of the next packet which the receiver wants to get.

Wrong packet number



- The data in the packet is *ignored.*
- The *NextRec* counter is unchanged.
- An *acknowledgement message* is sent. It contains the number of the next packet which the receiver wants to get.

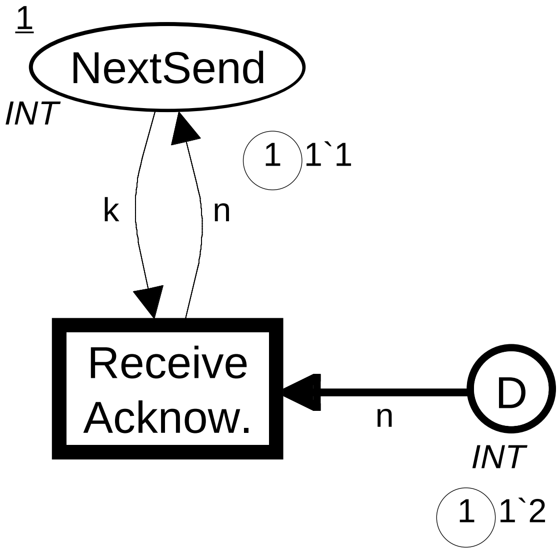
Transmit acknowledgement



This transition works in a similar way as *Transmit Packet*.

- The token on place *RA* determines the success rate for transmission of acknowledgements.
- When *RA* contains a token with value 8, the success rate is *80%.*
- When *RA* contains a token with value 10, *no* acknowledgements are lost.
- When *RA* contains a token with value 0, *all* acknowledgements are lost.

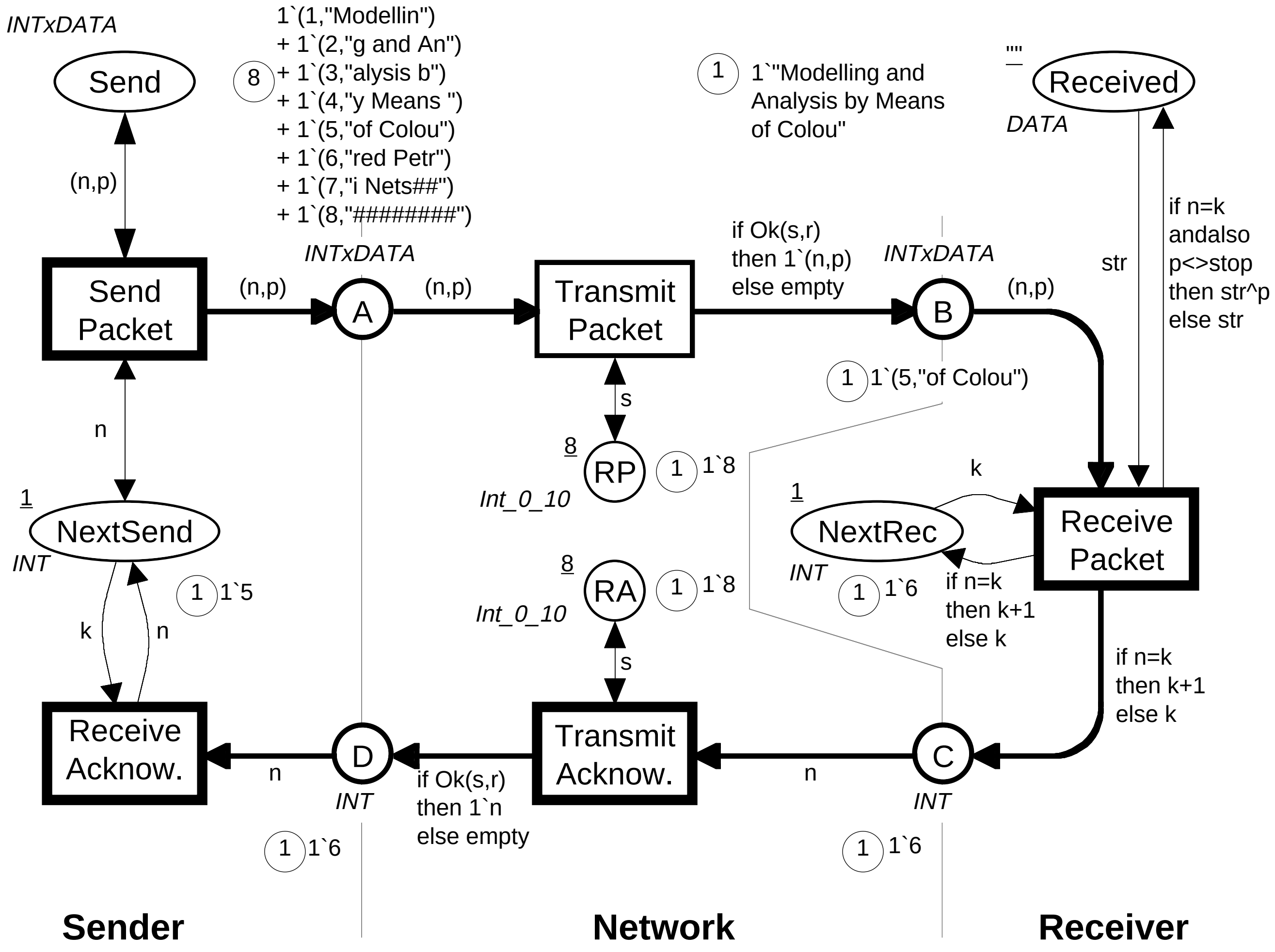
Receive acknowledgement



When an acknowledgement *arrives* to the *Sender* it is used to update the *NextSend* counter.

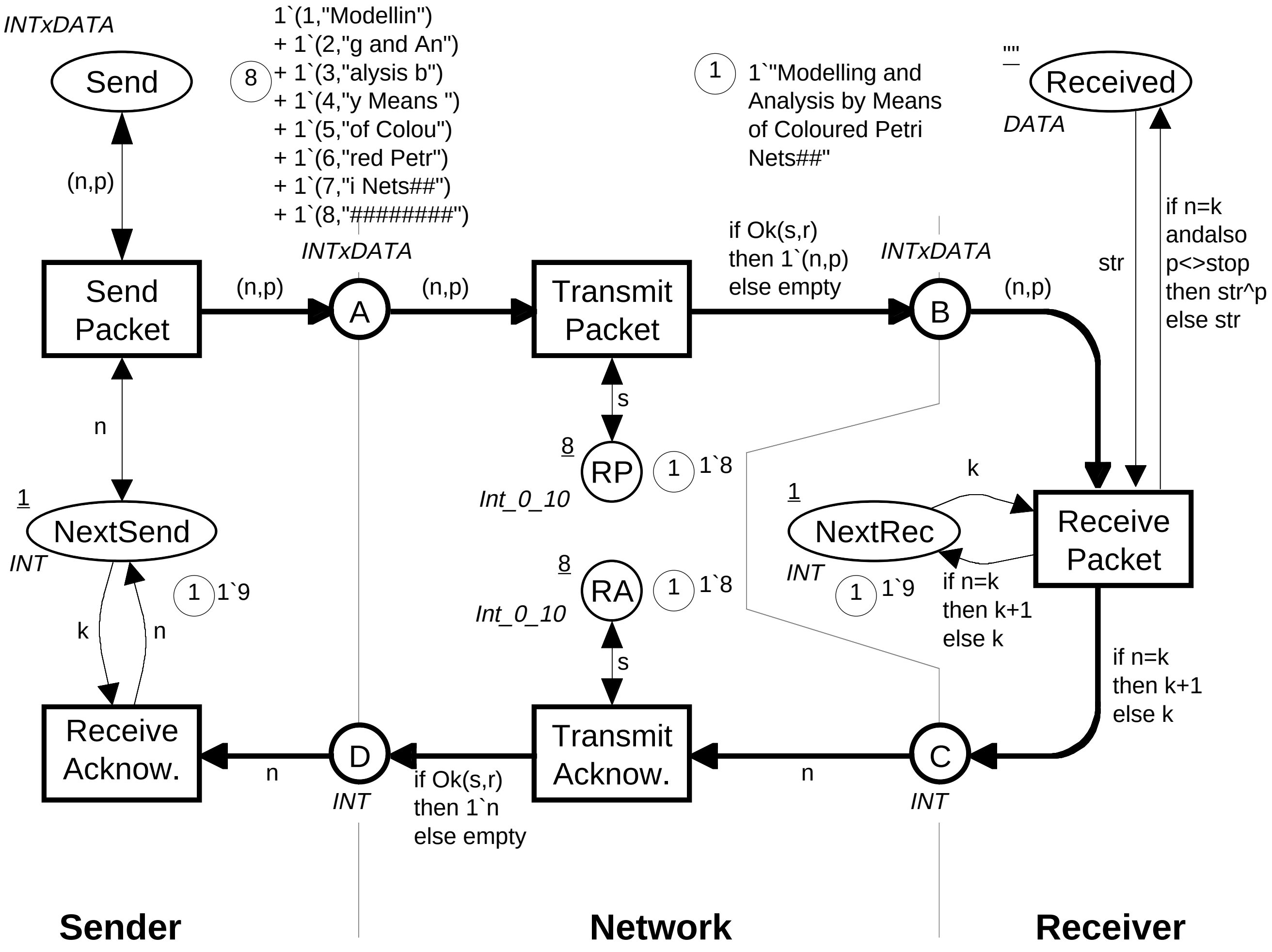
- In this case the counter value becomes 2, and hence the *Sender* will begin to send packet number 2.

Intermediate state



- The *Receiver* is expecting package no. 6. This means that it has successfully received the first 5 packets.
- The *Sender* is still sending packet no. 5. In a moment it will receive an acknowledgement containing a request for packet no. 6.
- When the acknowledgement is received the *NextSend* counter is updated and the *Sender* will start sending packet no. 6.

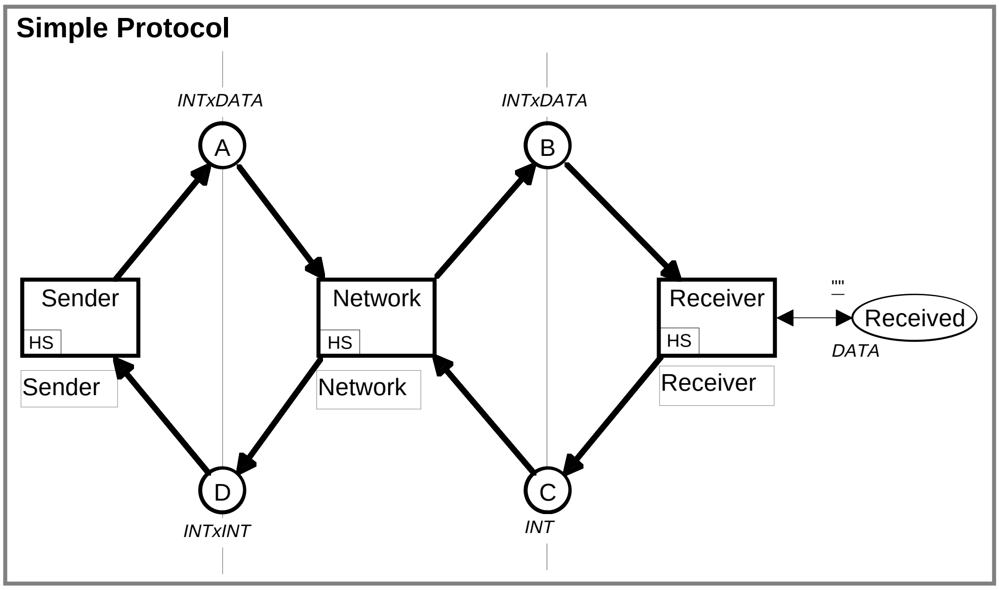
Final state

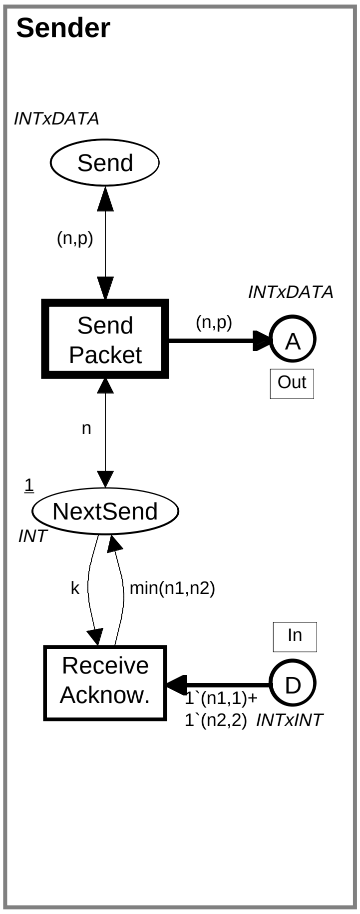


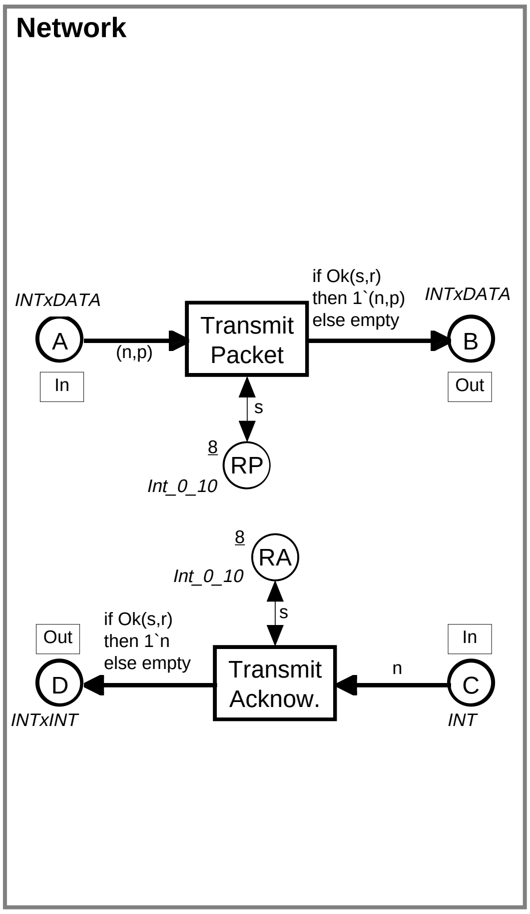
- When the last packet, i.e., packet no. 8 reaches the *Receiver* an acknowledgement with value 9 is sent.
- When this acknowledgement reaches the *Sender* the *NextSend* counter is updated to 9.
- This means that the *Send Packet* transition no longer can occur, and hence the transmission stops.

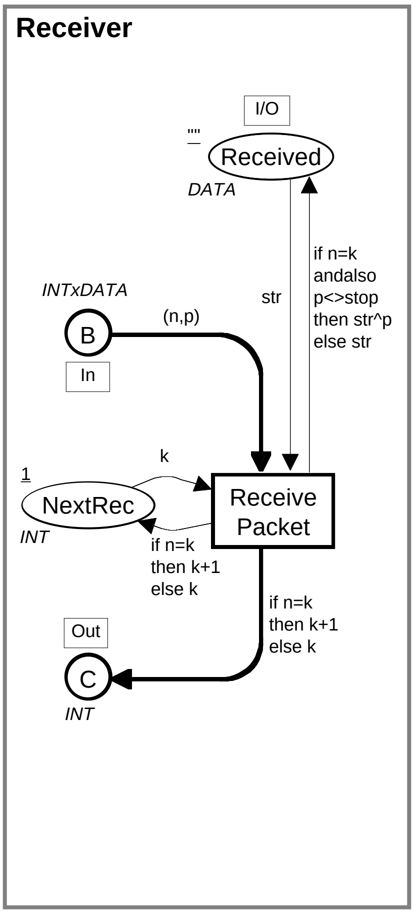
Part 2: Hierarchical CP-nets

A hierarchical CP-net contains a number of *interrelated subnets*– called *pages*.









Substitution transitions

A page may contain one ore more *substitution transitions*.

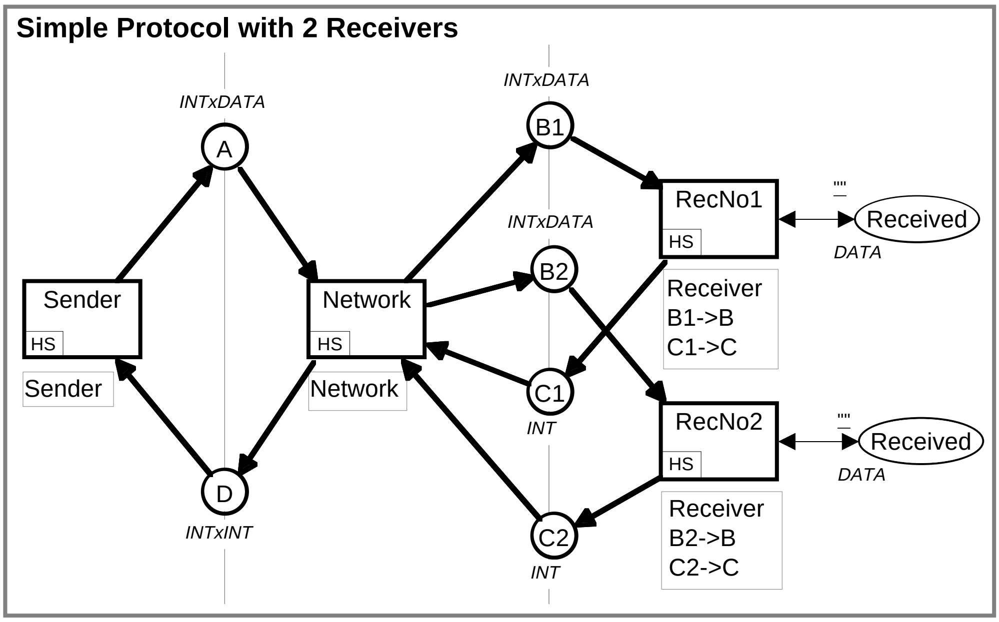
- Each substitution transition is related to a *page*, i.e., a *subnet* providing a *more detailed description* than the transition itself.
- The page is a *subpage* of the substitution transition.

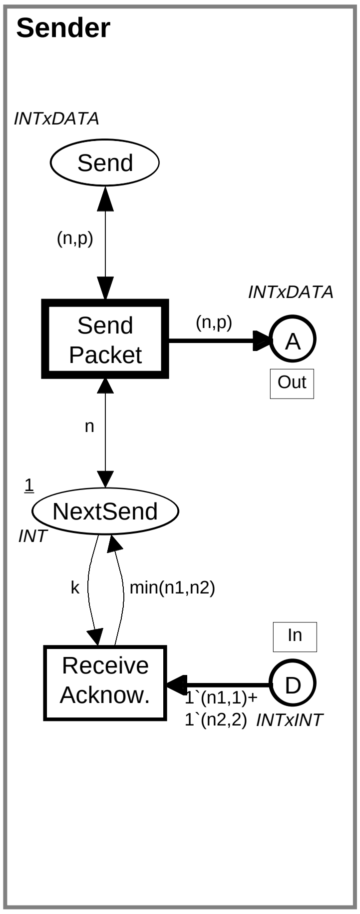
There is a *well-defined interface* between a substitution transition and its subpage:

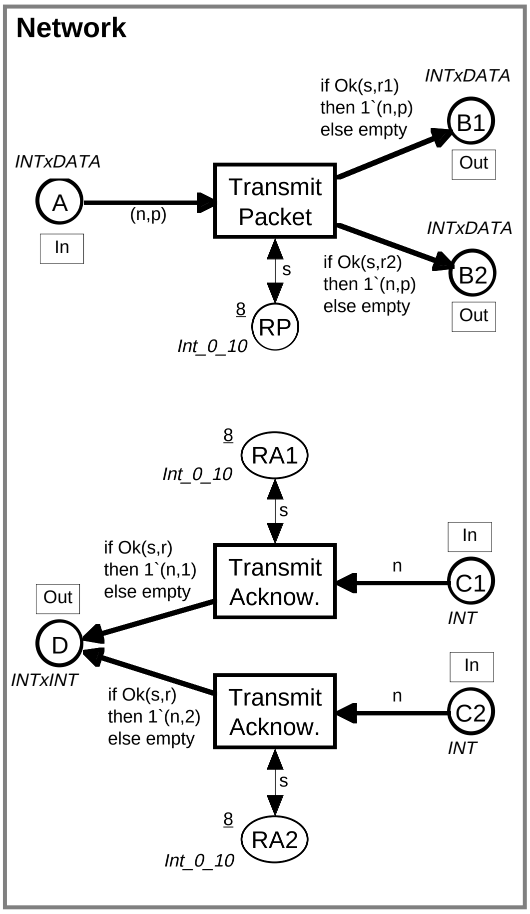
- The places surrounding the substitution transition are *socket places.*
- The subpage contains a number of *port places.*
- Socket places are *related* to port places – in a similar way as actual parameters are related to formal parameters in a procedure call.
- A socket place has always the *same marking* as the related port place. The two places are just *different views* of the same place.

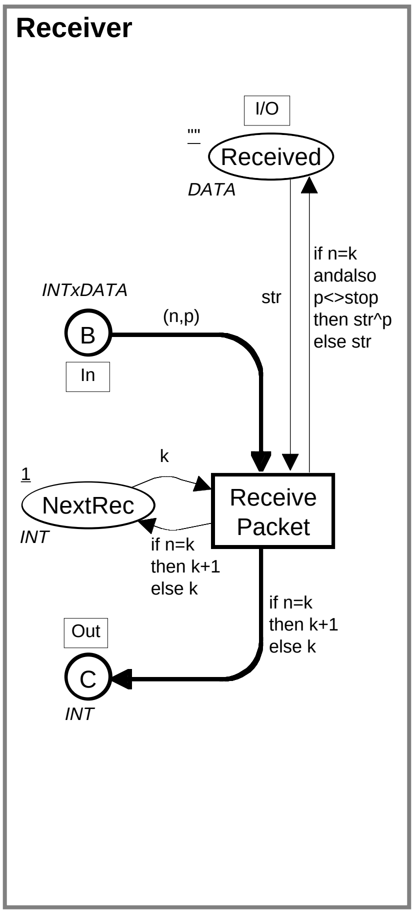
Substitution transitions work in a similar way as the refinement primitives found in many system description languages – e.g., SADT diagrams.

Pages can be used more than once



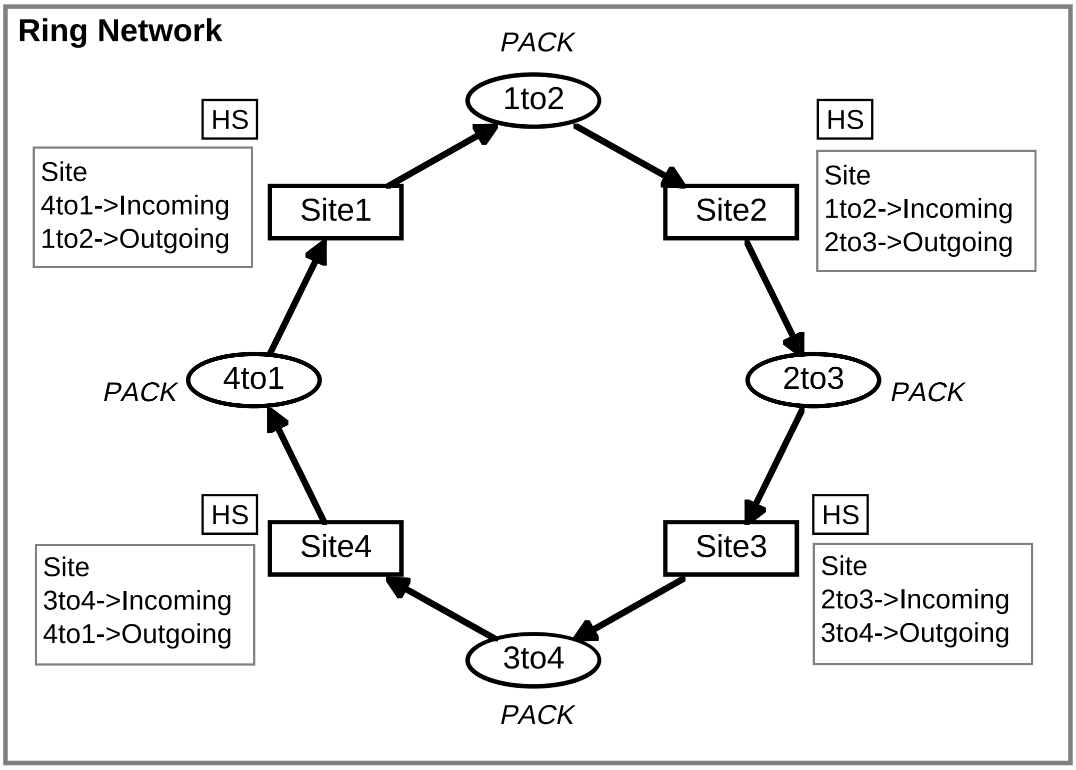


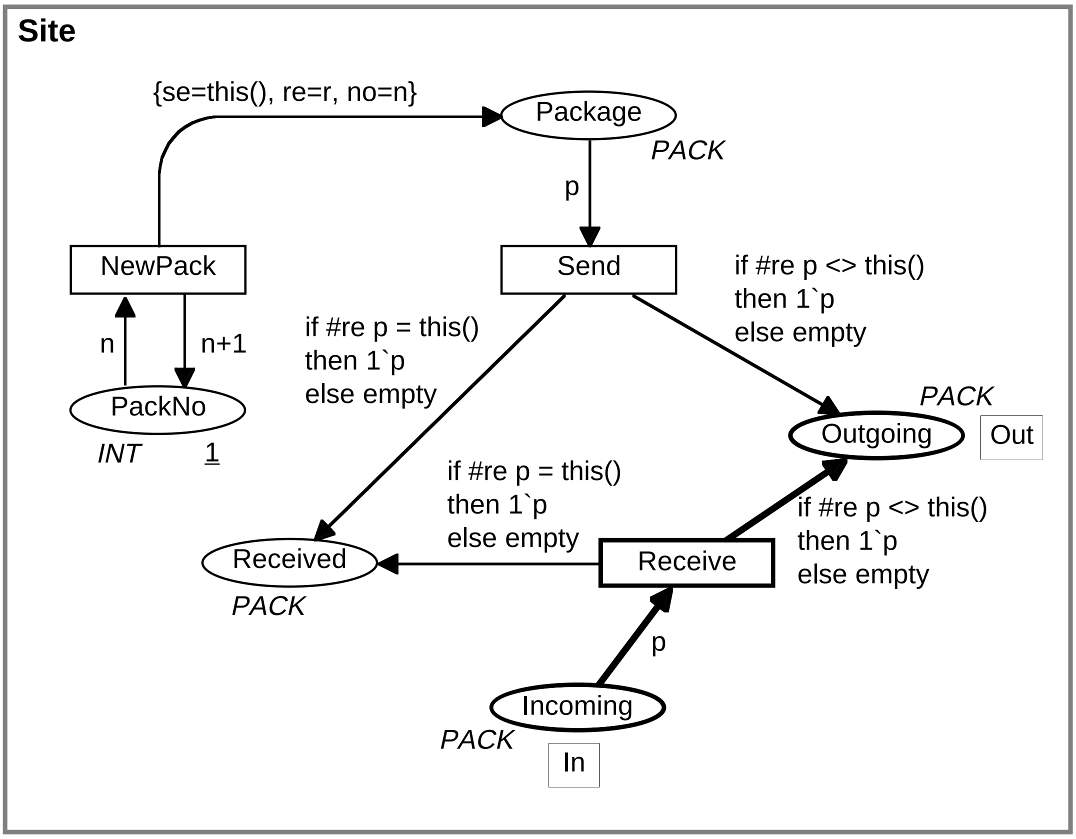




There are *two* different *instances* of the *Receiver* page. Each instance has its *own marking*.

Ring network





Formal definition of hierarchical CP-nets

The *syntax* and *semantics* of hierarchical CP-nets have *formal definitions* – similar to the definitions for non-hierarchical CP-nets

Each hierarchical CP-net has an *equivalent* non-hierarchical CP-net – and vice versa.

- The two kinds of nets have the same *computational power* – but hierarchical CP-nets have much more *modelling power*.
- The *equivalence* is used for *theoretical purposes*.
- In practice, we *never* translate a hierarchical CP-net into a non-hierarchical CP-net – or vice versa.

CP-nets may be large

A typical *industrial application* of CP-nets contains:

- 10-200 *pages*.
- 50-1000 *places and transitions*.
- 10-200 *colour sets*.

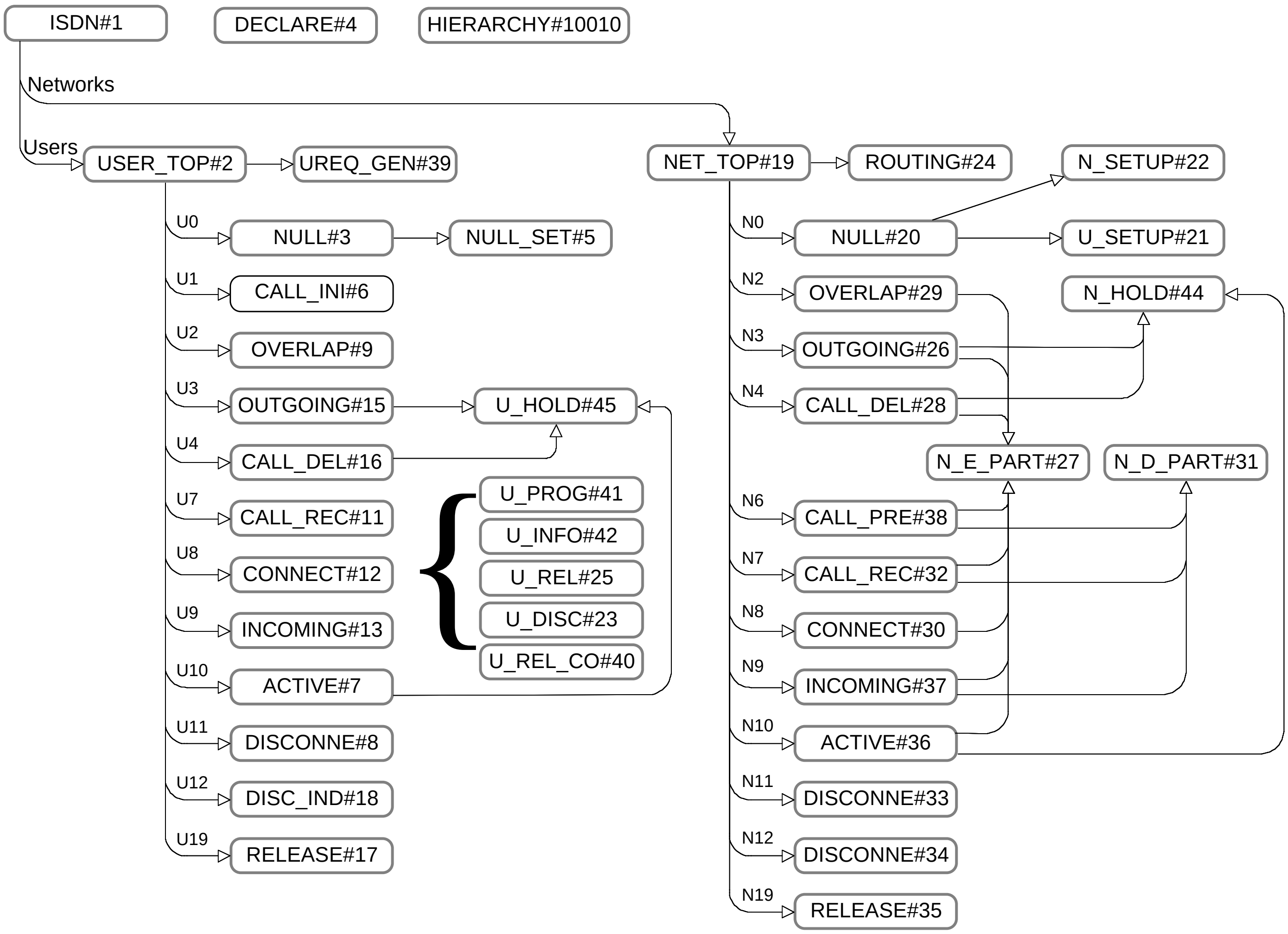
This corresponds to *thousands/millions of nodes* in a Place/Transition Net.

Most of the industrial applications would be *totally impossible* without:

- Colours.
- Hierarchies.
- Computer tools.

Protocol for telephone network

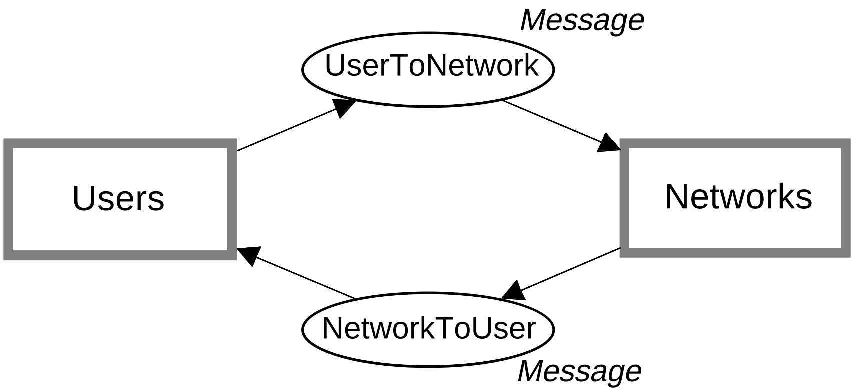
Transport layer of a protocol for *digital telephone communication*.

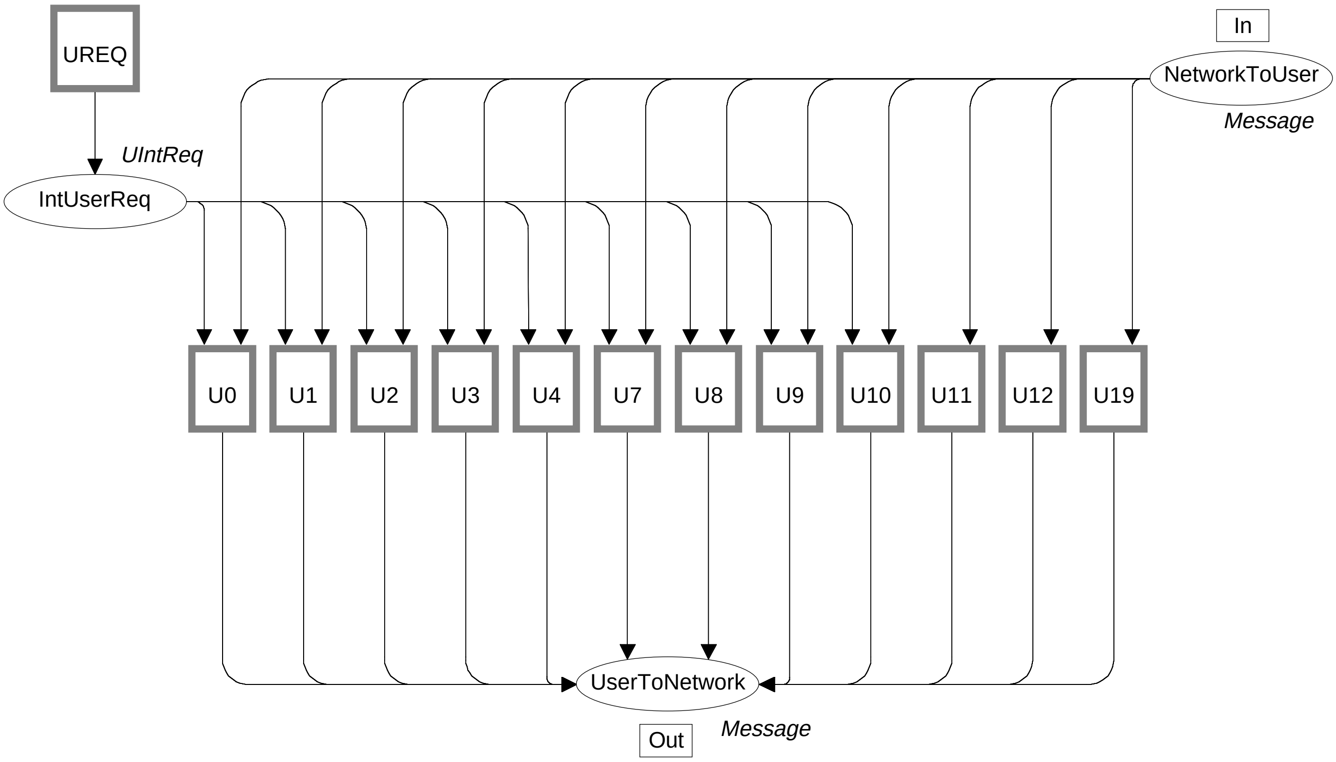


Overview of the hierarchy structure:

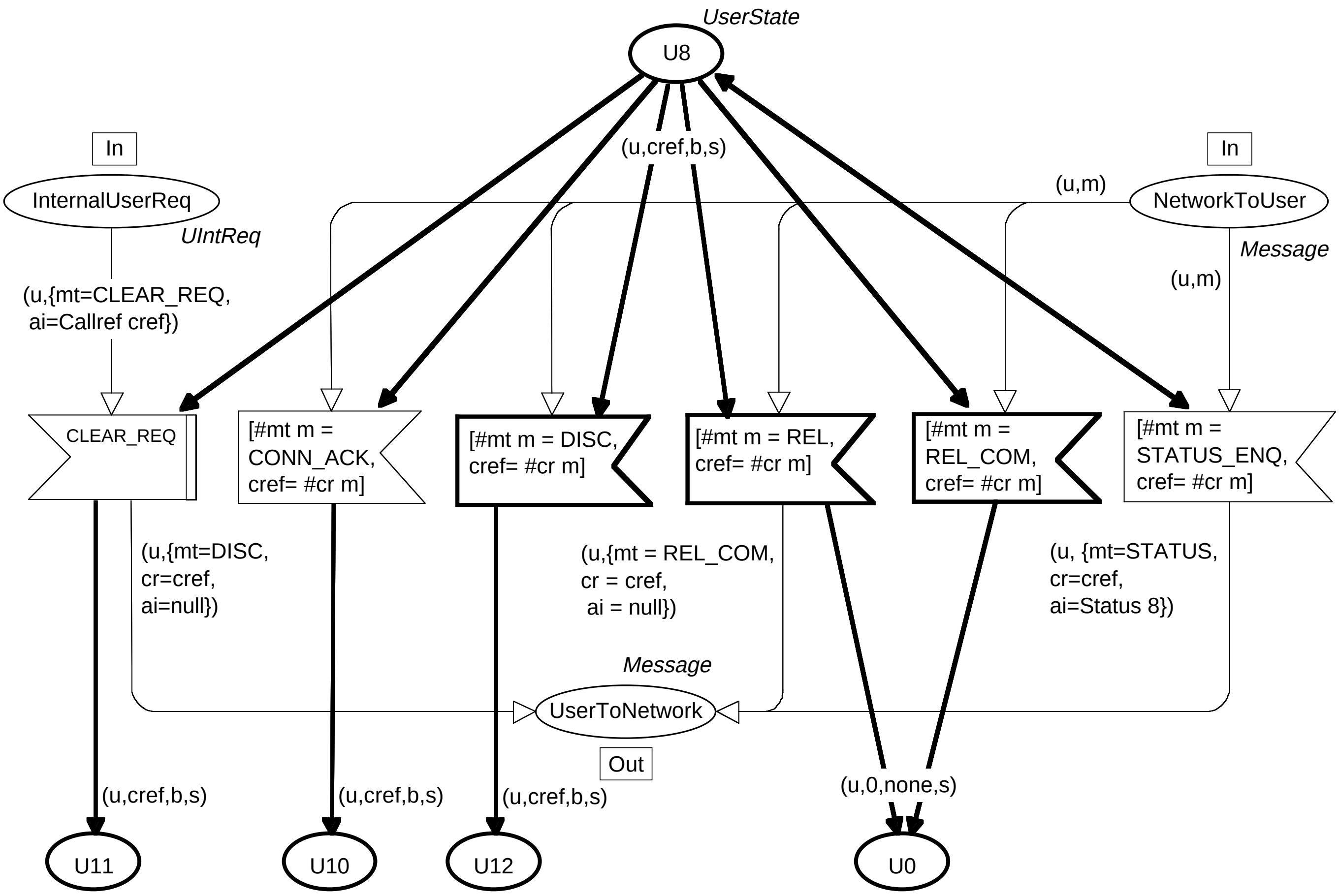
- Each *node* represents a *page*, i.e., a subnet.
- Each *arc* represents a *transition substitution*.

Two of the most abstract pages





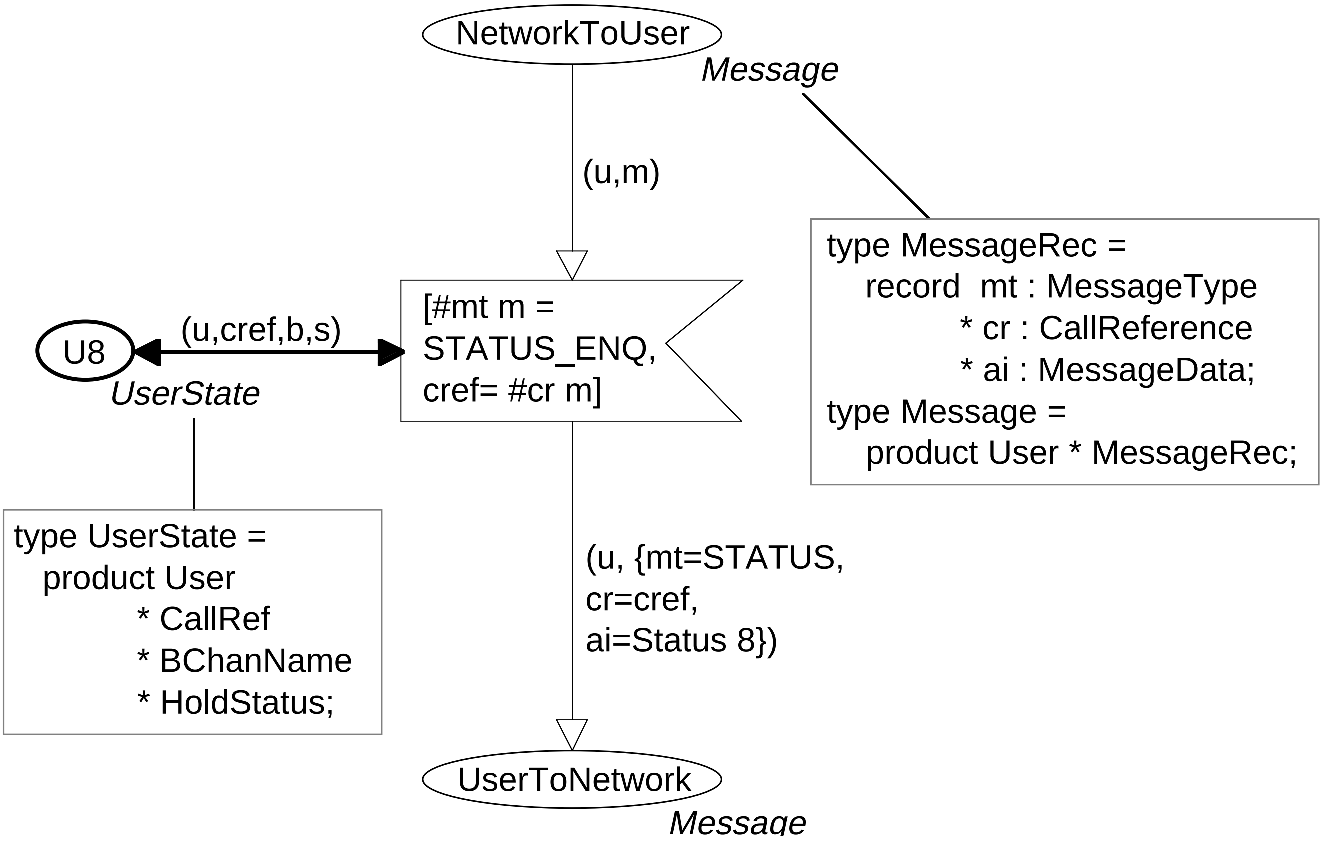
Typical page for the user site



This page describes the *possible actions* that can happen when the user site is in state *U8:*

- From the *network* five different kinds of messages may be received.
- In addition there is one kind of *internal user request.*
- In three of the cases a *new message* is sent to the *network site.*

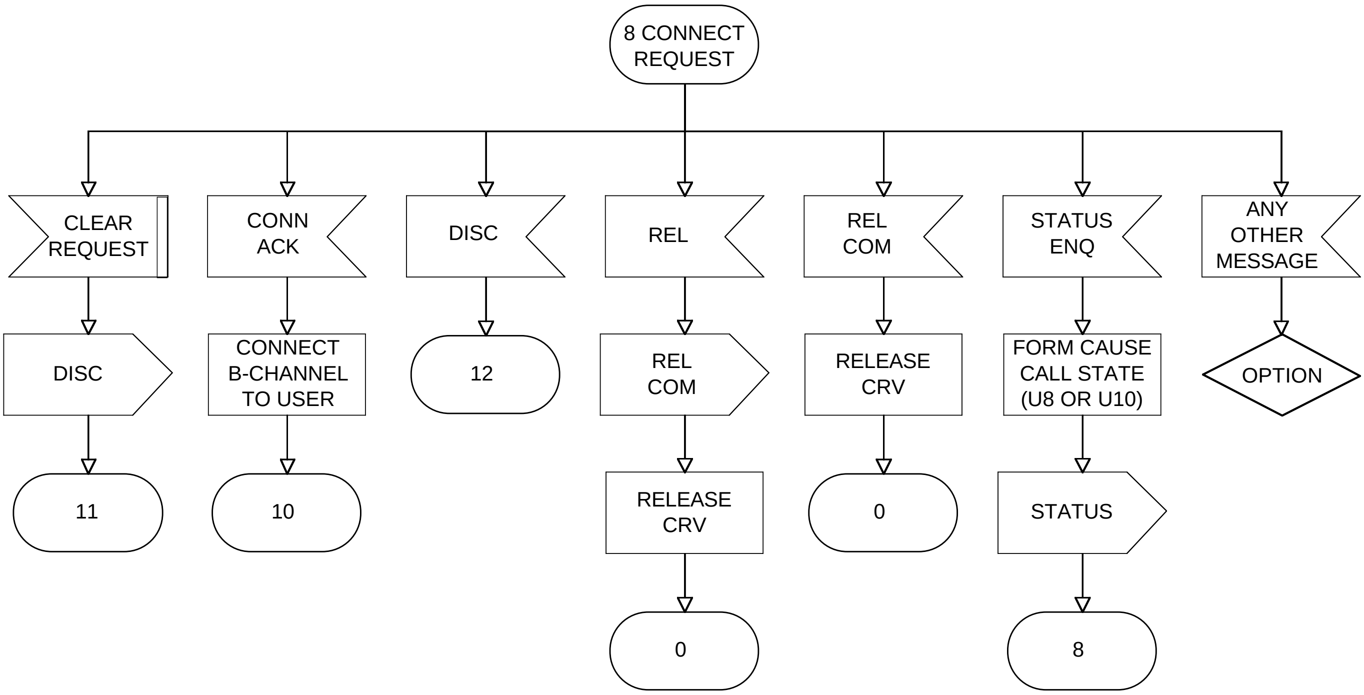
Typical transition



This transition describes the *actions* that are taken when a *Status Enquiry* message is received in state *U8:*

- The guard checks that the message is a *Status Enquiry* message. It also checks that the *Call Reference* is correct (i.e., matches the one in the *User State* token at place U8).
- A *Status message* is sent to the *network site.* It tells that the user site is in state U8.

SDL description of user page



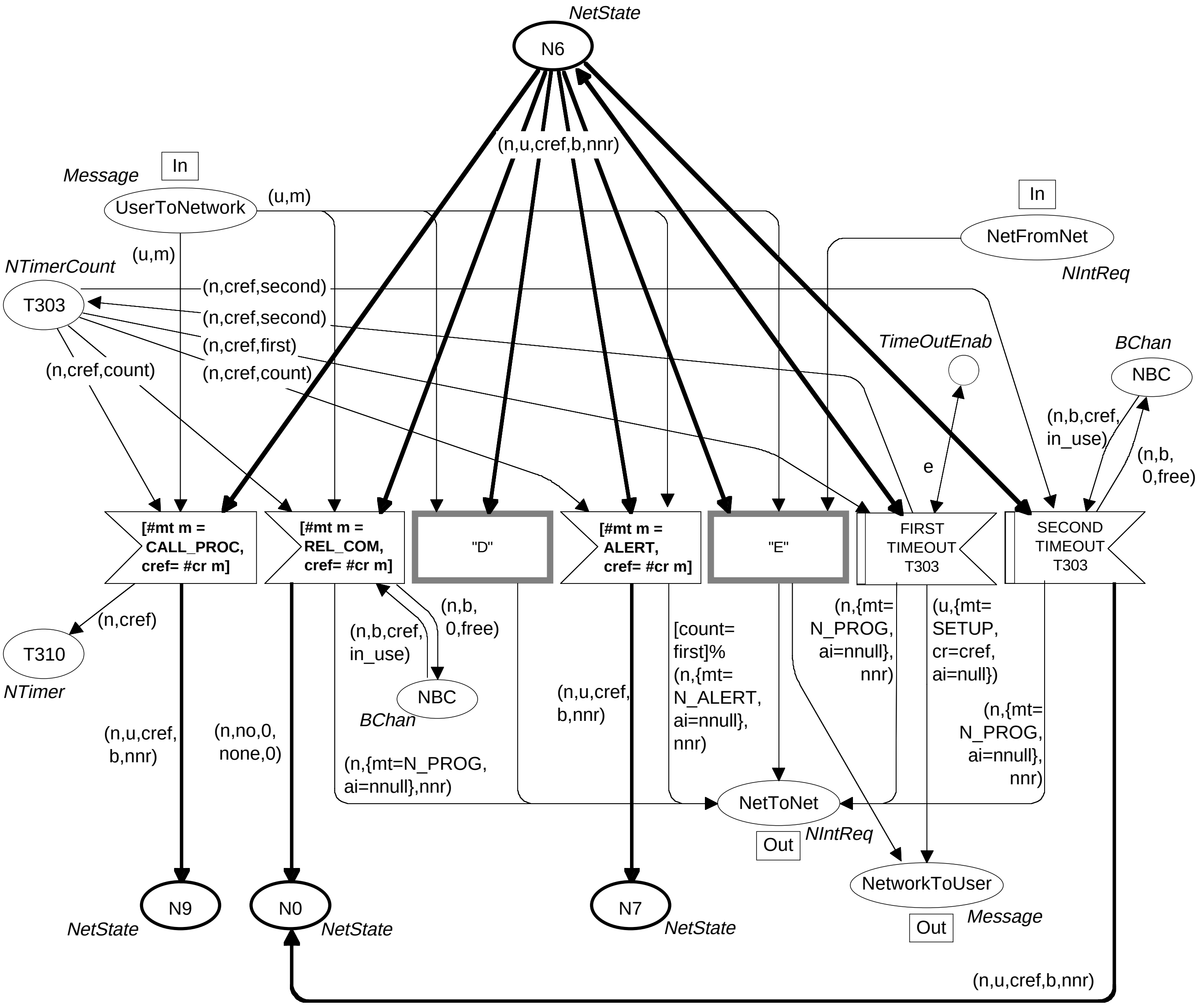
Each *vertical string of SDL symbols* describes a sequence of actions – which is translated into a *single CPN transition.*

- The *translation* from SDL to CPN was done *manually*.
- The translation is straightforward and it could easily be *automated*.

The graphical shape of a node has a *well-defined* meaning in SDL.

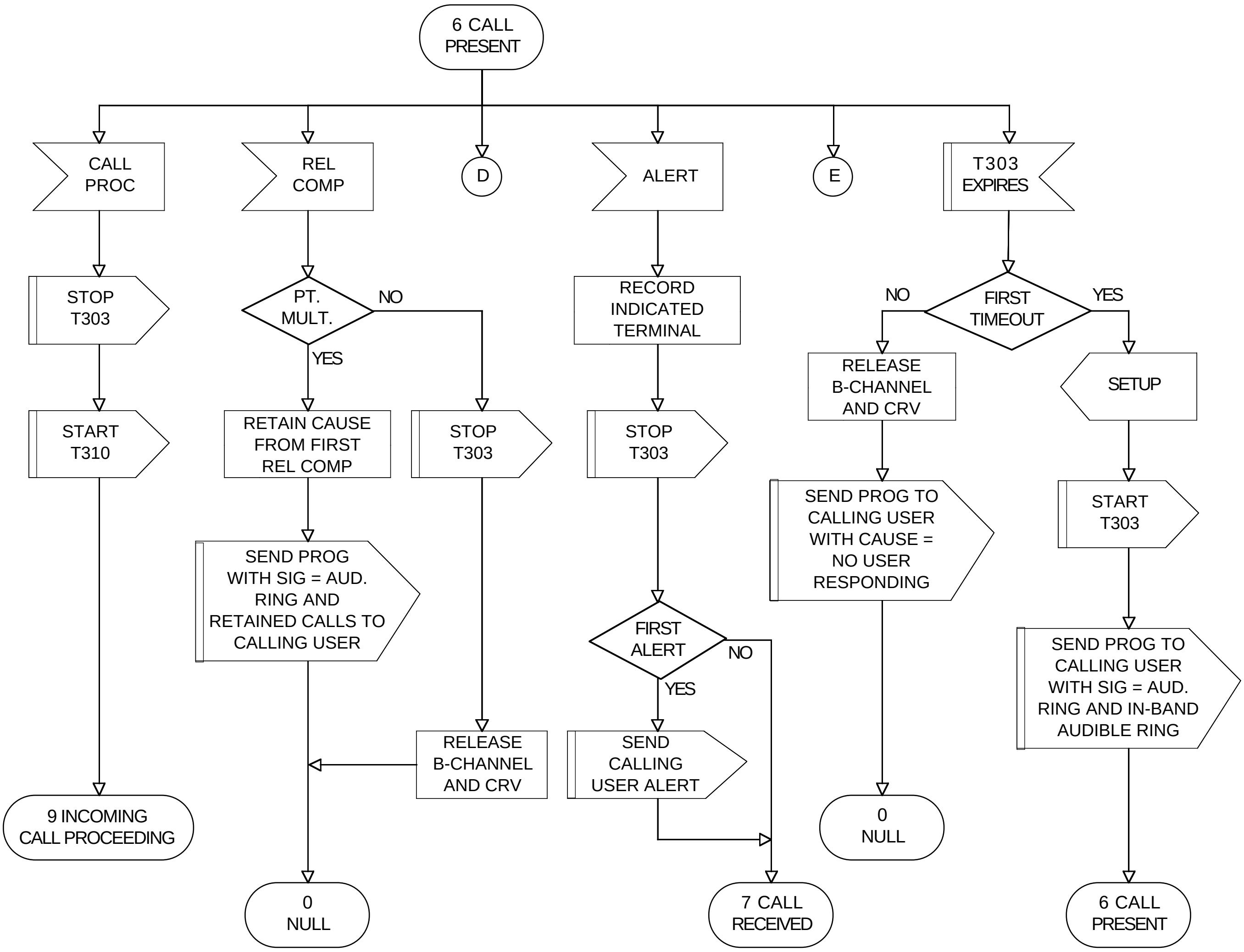
- In the CP-net the shape is retained – to improve the *readability*. It has no formal meaning.

Typical page for the network site



Similar structure as for the user page – but slightly more complex.

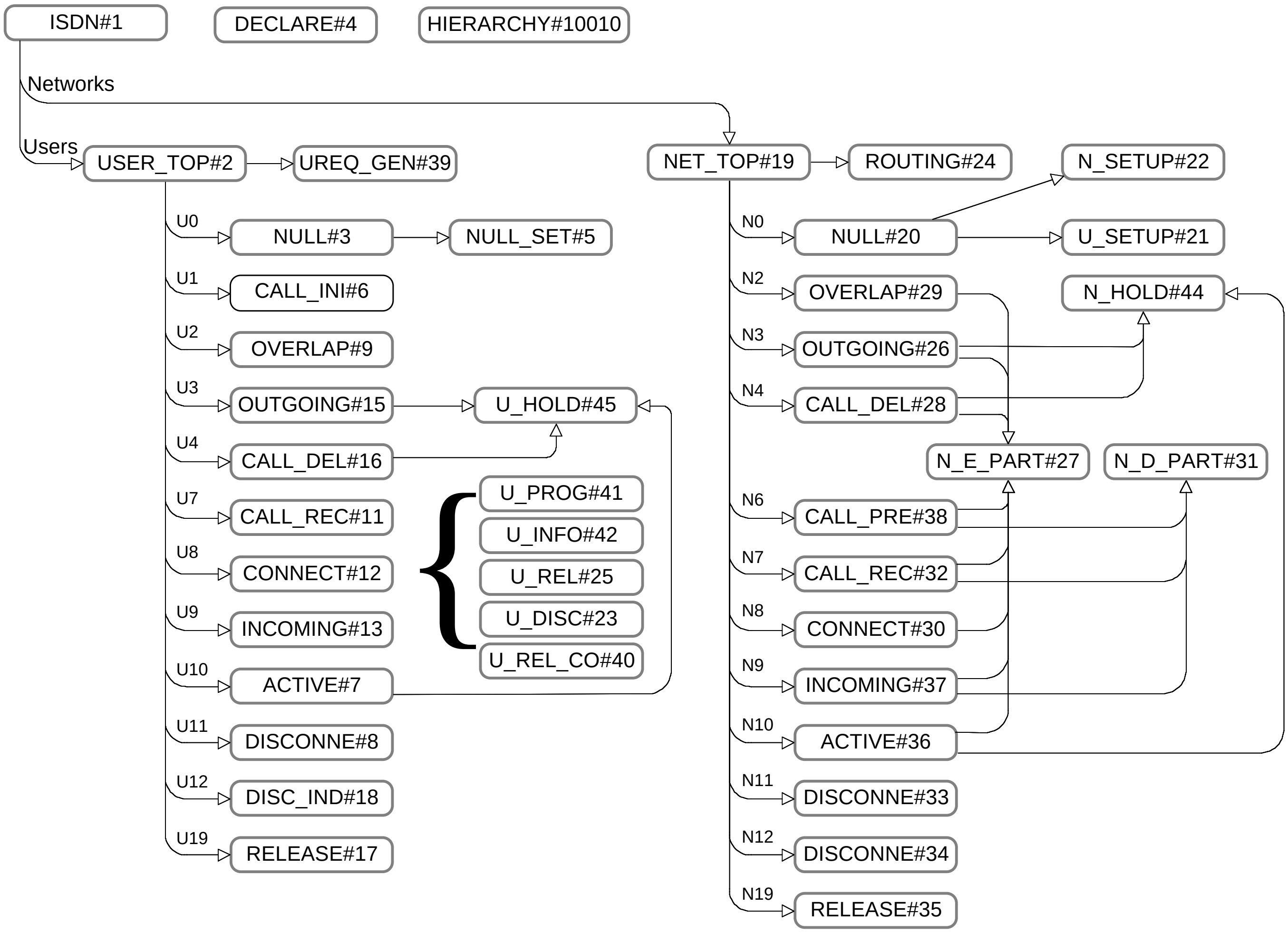
SDL description of network page



Similar structure as for the user page – but slightly more complex.

It is easy to see that there is a very straightforward relationship between the *SDL page* and the corresponding *CPN page*.

Some pages are used many times



- 43 pages with more than 100 page instances.
- The entire modelling of this – fairly complex protocol – was made in only *3 weeks* (by a single person).
- According to engineers at the participating telecommunications company, the CPN model was the *most detailed* behavioural model that they had ever seen for such protocols.

Practical use of CP-nets

CP-nets are used in *many different areas*. A few selected examples are:

- Communication protocols (BRI, DQDB, ATM).
- VLSI chips (clocked and self-timed).
- Banking procedures (check processing and funds transfer).
- Correctness of ADA programs (rendezvous structure).
- Teleshopping systems.
- Military systems (radar control post and naval vessel).
- Security systems (intrusion alarms, etc.).
- Flexible manufacturing.

Summary of practical experiences

Graphical representation and *executability* are extremely important.

Most practical models are *large*.

- They cannot be constructed without the *hierarchy concepts*.
- Neither can they be constructed or verified without the *computer tools*.

CP-nets are often used *together* with other graphical description languages, such as SADT, SDL and block diagrams.

- This means that the user does not have to learn a completely *new language*.

CP-nets are well-suited for *verification* of existing designs – in particular concurrent systems.

- CP-nets can also be used to *design* new systems.
- Then it is possible to use the *insight* gained through the modelling, simulation and verification activities – to *improve* the design itself.

Part 3: Construction and Simulation of CP-nets

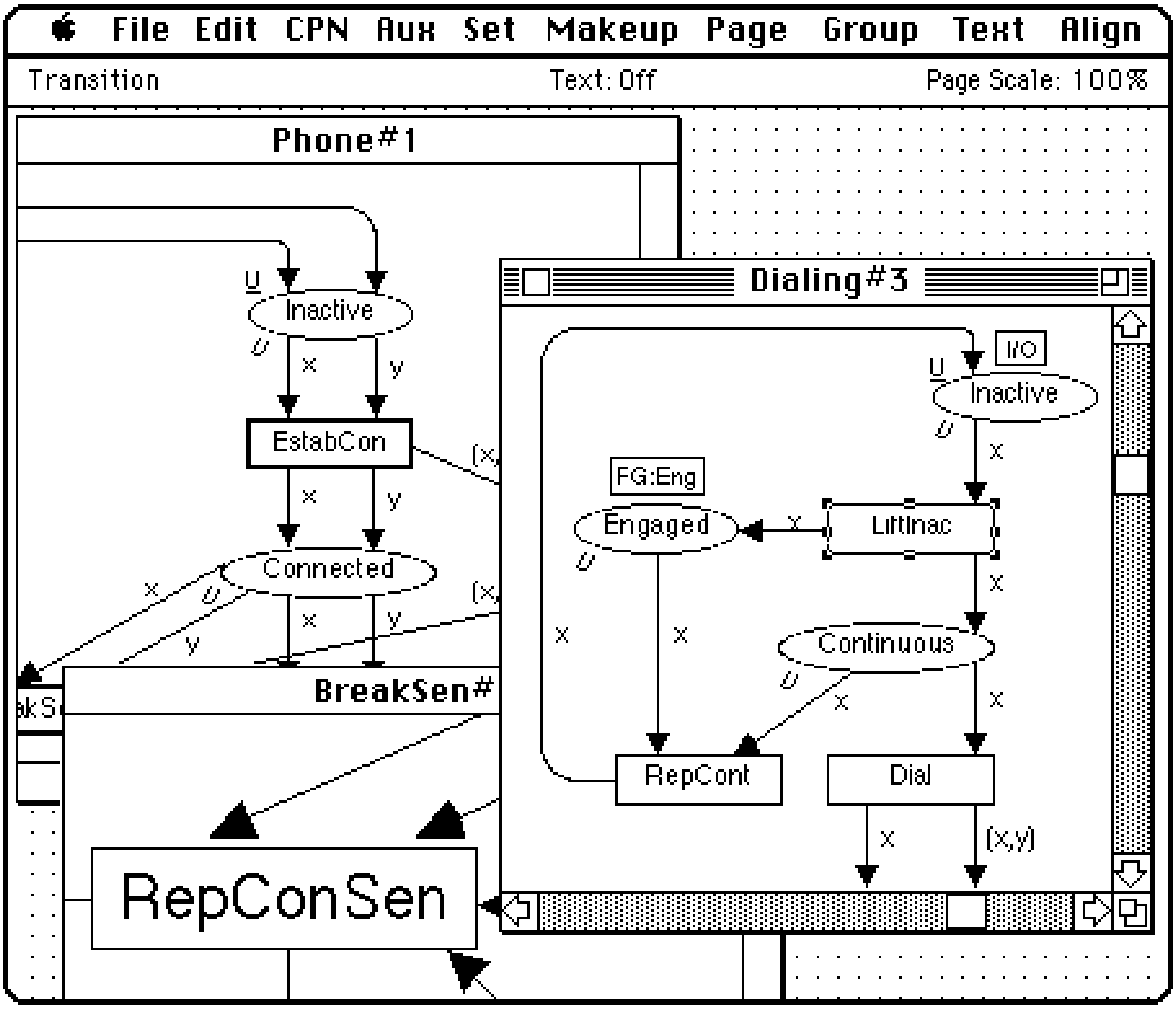
CP-nets have an *integrated* set of *robust* computer tools with *reliable support:*

- *Construction* and *modification* of CPN models.
- *Syntax checking* (e.g., types and module interfaces).
- *Interactive simulation*, e.g., to gain additional understanding of the modelled system. Can also be used for *debugging*.
- *Automatic simulations*, e.g., to obtain performance measures. Can also be used for *prototyping*.
- *Verification* to *prove* behavioural properties.
  - *State spaces* (also called reachability graphs and occurrence graphs).
  - *Place invariants*.

The computer tools are available on different platforms:

- Sun Sparc with Solaris.
- Macintosh with Mac OS.

CPN editor



Each *page* is shown in its own *window*.

The user performs an operation by selecting an *object* and a *command* for it, e.g.:

- Select a *transition* (by pointing with the mouse).
- Select the desired *command* (by pointing in the corresponding drop-down menu).

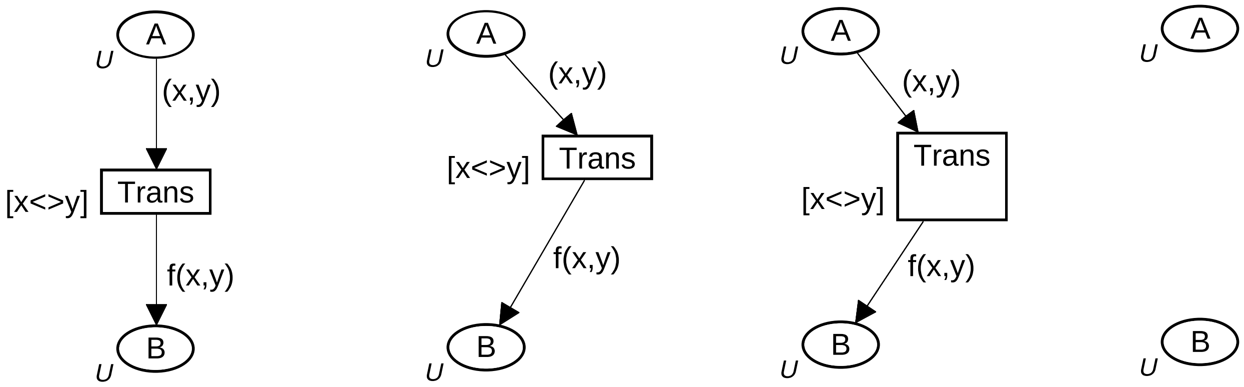
Commands can be performed on a *set of objects*.

Editor knows syntax of CP-nets

Some kinds of errors are *impossible*, e.g.:

- An arc between *two places* or *two transitions*.
- A place with *two colour sets* or an arc with *two arc expressions*.
- A transition with a *colour set*.
- Port assignment involving a place which is a *non-socket* or a *non-port*.
- A *cyclic* set of *substitution transitions*.

The editor behaves *intelligently:*



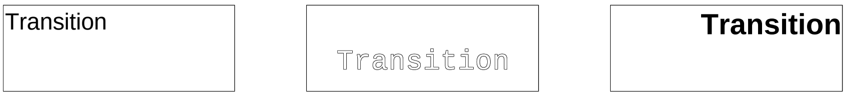
- When a node is *repositioned* or *resized* the surrounding arcs and inscriptions are *automatically adjusted*.
- When a node is *deleted* the surrounding arcs are *automatically deleted*.

Attributes

Each graphical object has its own *attributes.*

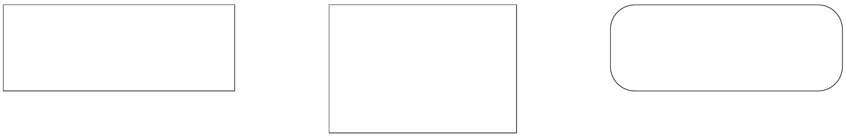
They determine how the object appears on the screen/print-outs:

- Text attributes

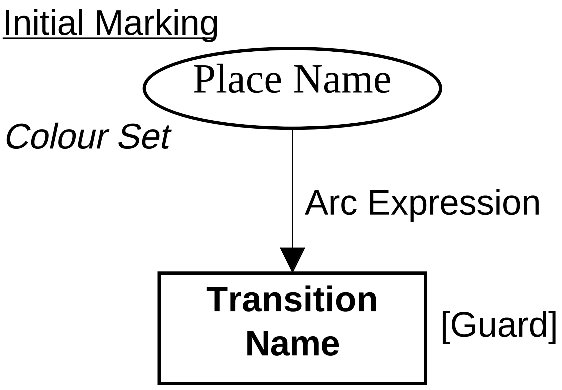


- Graphical attributes

- Shape attributes

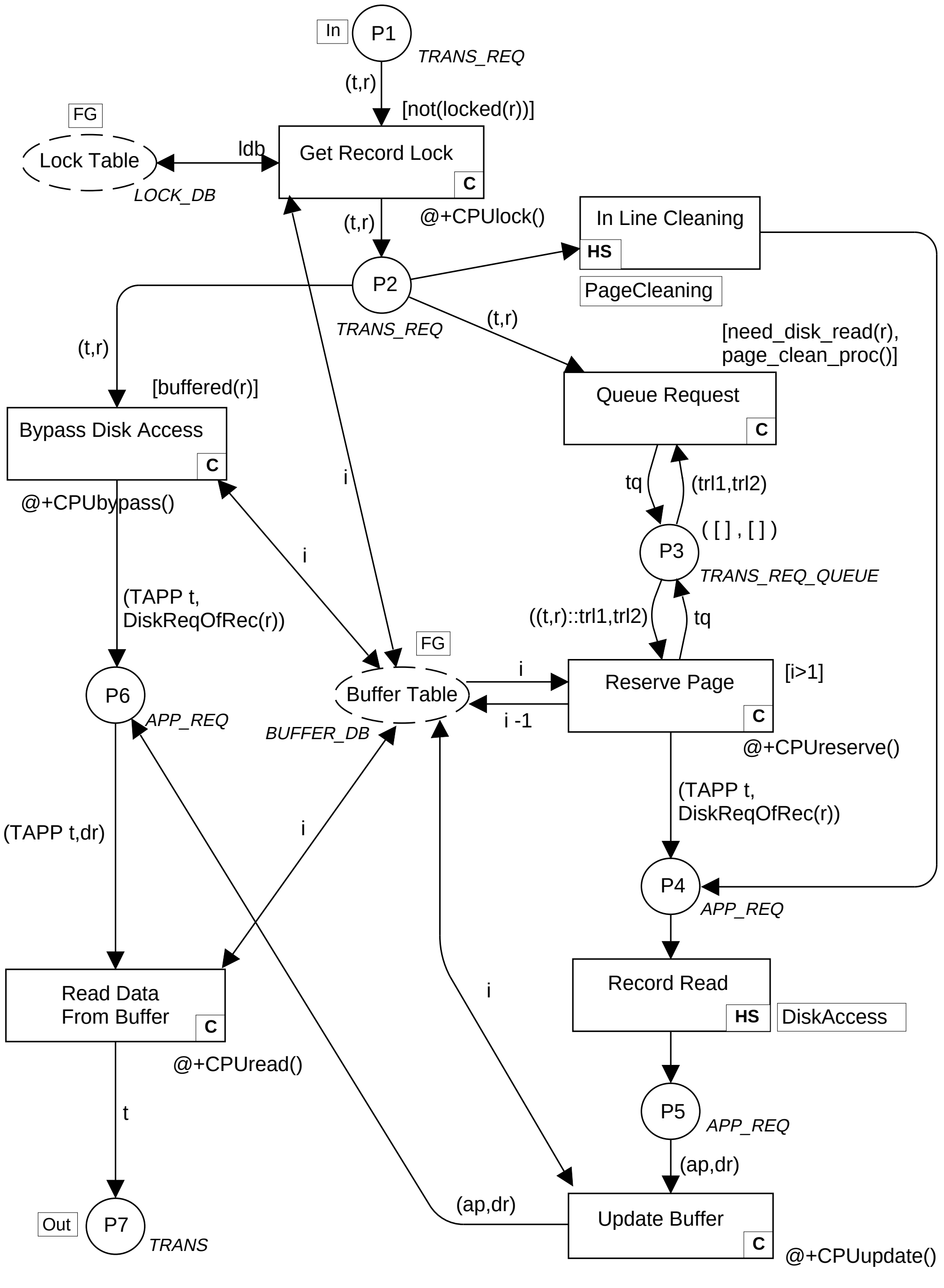


Each *kind of objects* has its own *defaults:*



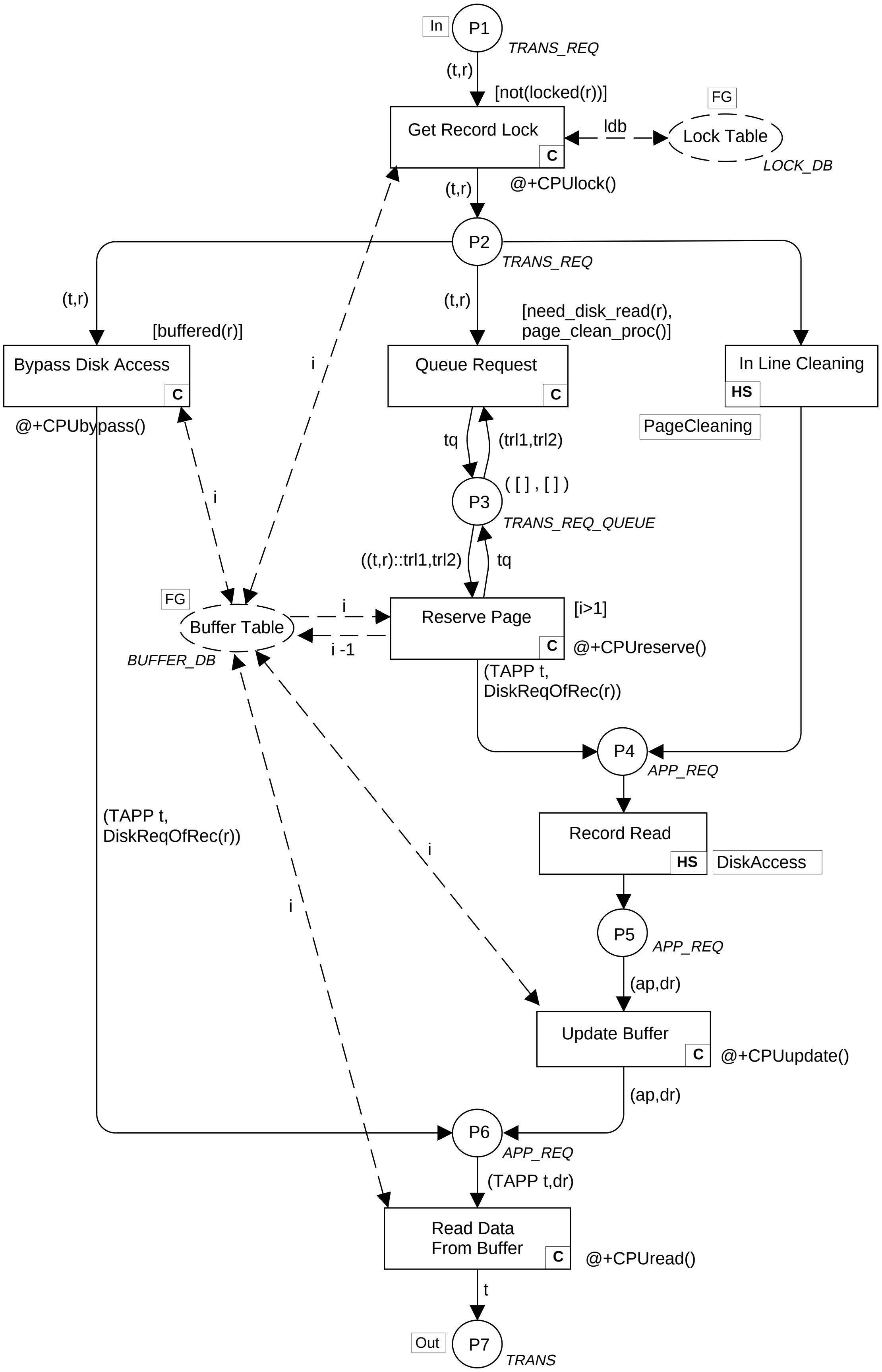
Defaults can be *changed* and they can be *overwritten* (for individual objects).

Easy to experiment

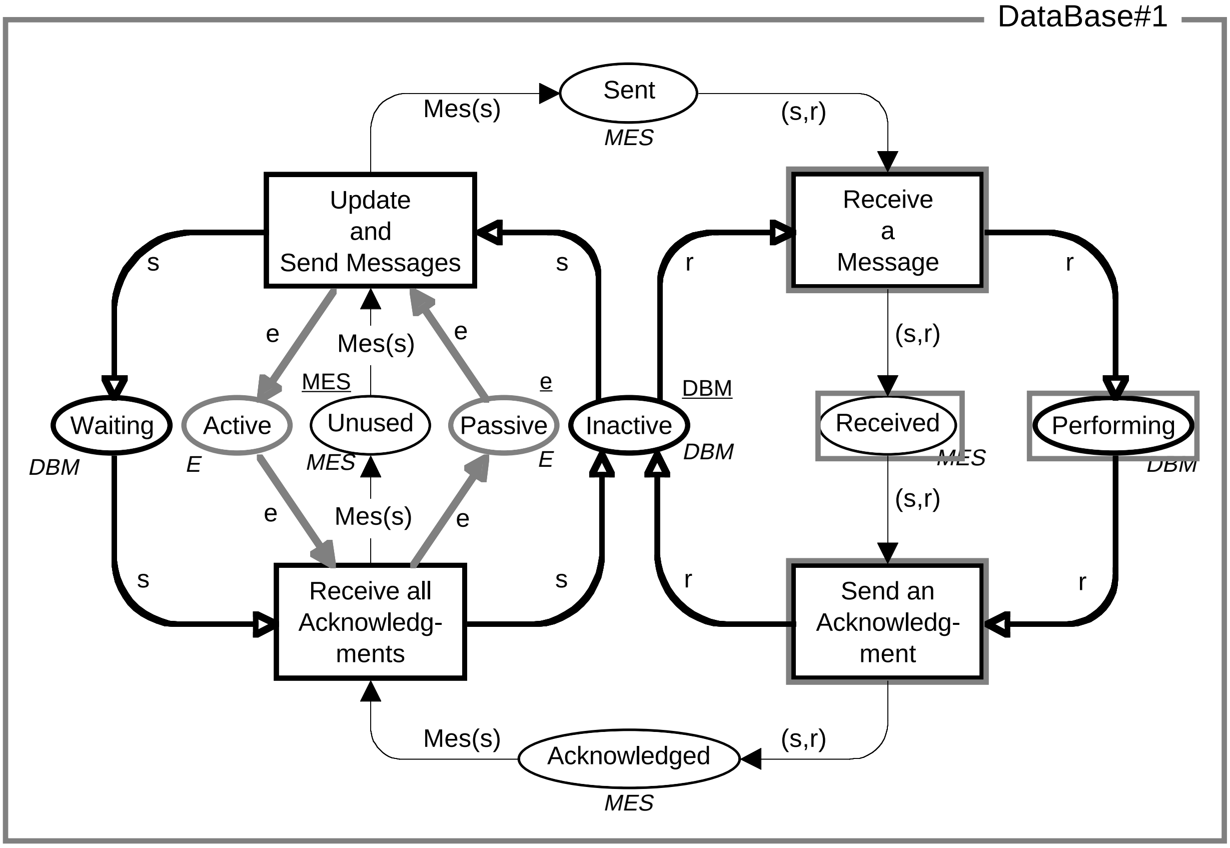


Can we improve the *layout* of this page?

Improved layout



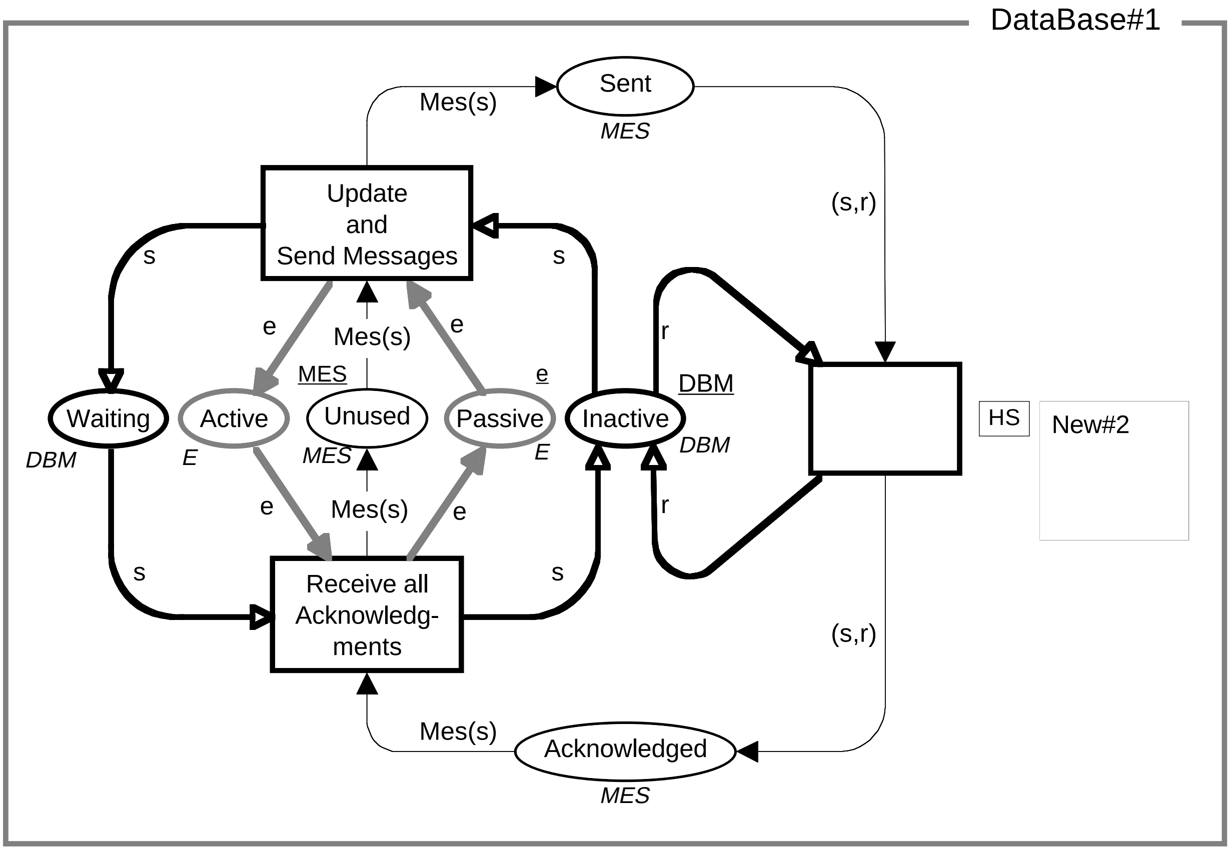
How to make a new subpage

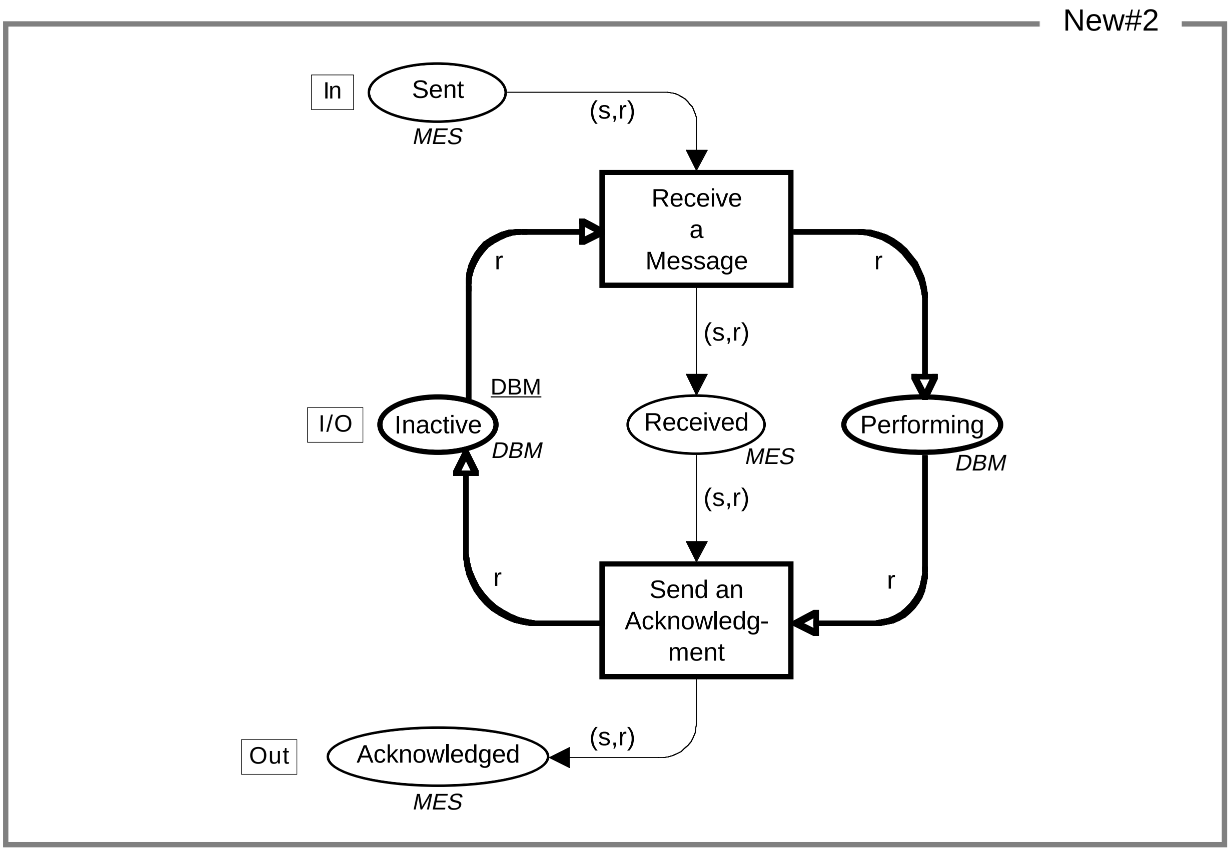


We want to *move* the four selected nodes to a *new page* – and replace them by a *substitution transition:*

- This is done by a single command – called *Move to Subpage*.

Result of Move to Subpage





Move to Subpage is complex

The *Move to Subpage* command is *complex.* The command:

- Checks the *legality of the selection* (all border nodes must be transitions).
- Creates the *new page.*
- *Moves the subnet* to the new page.
- *Prompts the user* to create a new transition which becomes the supernode for the new subpage.
- Creates the *port places* by copying those places which were next to the selected subnet.
- Calculates the *port types* (in, out or in/out).
- Creates the corresponding *port inscriptions.*
- Constructs the necessary *arcs* between the port nodes and the selected subnet.
- Draws the *arcs* surrounding the new transition.
- Creates a *hierarchy inscription* for the new transition.
- Updates the *hierarchy page.*

All these things are done in a *few seconds.*

Top-down and bottom-up

Move to Subpage supports *top-down* development. However, it is also possible to work *bottom-up* – or use a *mixture* of top-down and bottom-up.

The *Substitution Transition* command is used to relate a substitution transition to an *existing page.* The command:

- Makes the *hierarchy page active.*
- *Prompts the user* to select the desired subpage; when the mouse is moved over a page node it blinks, unless it is illegal (because selecting it would make the page hierarchy cyclic).
- *Waits* until a blinking *page node* has been selected.
- Tries to deduce the *port assignment* by means of a set of rules which looks at the port/socket names and the port/socket types.
- Creates the *hierarchy inscription* with the name and number of the subpage and with those parts of the port assignment which could be automatically deduced.
- Updates the *hierarchy page.*

Syntax checking

When a CPN diagram has been constructed it can be *syntax checked.*

The most common errors are:

- Syntax errors in the *declarations.*
- Syntax errors in *arc expressions or guards.*
- *Type mismatch* between arc expressions and colour sets.

Syntax checking is *incremental:*

- When a colour set, guard or an arc expression is changed, it is *sufficient* to recheck the *nearest surroundings.*
- Analogously, if an *arc* is added or removed.

All CPN diagrams in this set of lecture notes are made by means of the CPN editor.

CPN simulator

When a *syntactical correct* CPN diagram has been constructed, the CPN tool generates the necessary *code to perform simulations*.

The simulation code:

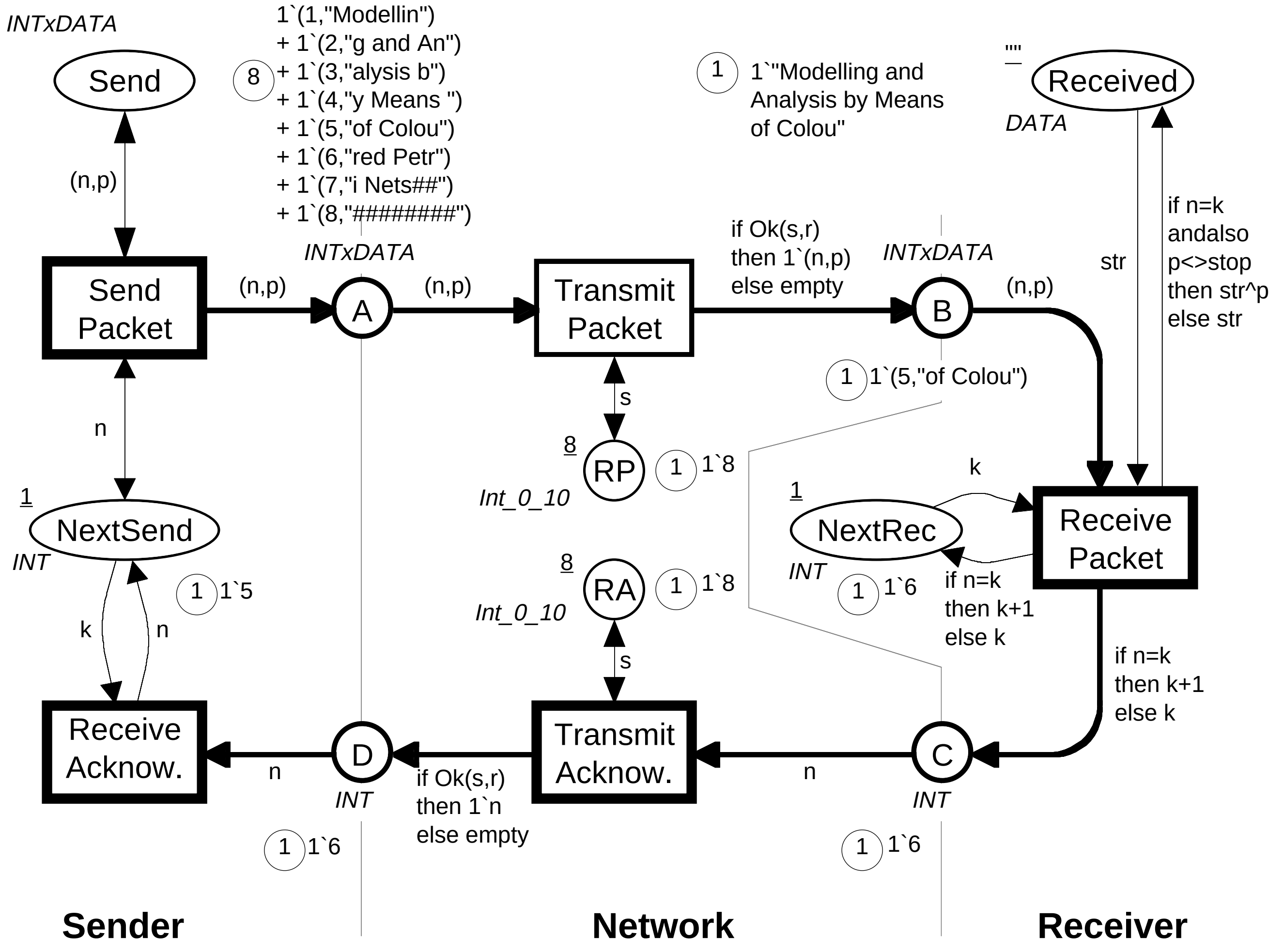
- Calculates whether the individual transitions and bindings are *enabled*.
- Calculates the *effect of occurring transitions and bindings*.

The code generation is *incremental*. Hence it is fast to make small changes to the CPN diagram.

We distinguish between two kinds of simulations:

- In an *interactive* simulation the user is in control, but most of the work is done by the system.
- In an *automatic* simulation the system does all the work.

Interactive simulation



Simulation results are shown directly on the CP-net:

- The user can see the *enabled transitions* and the *markings* of the individual places.

To *execute a step*, the user:

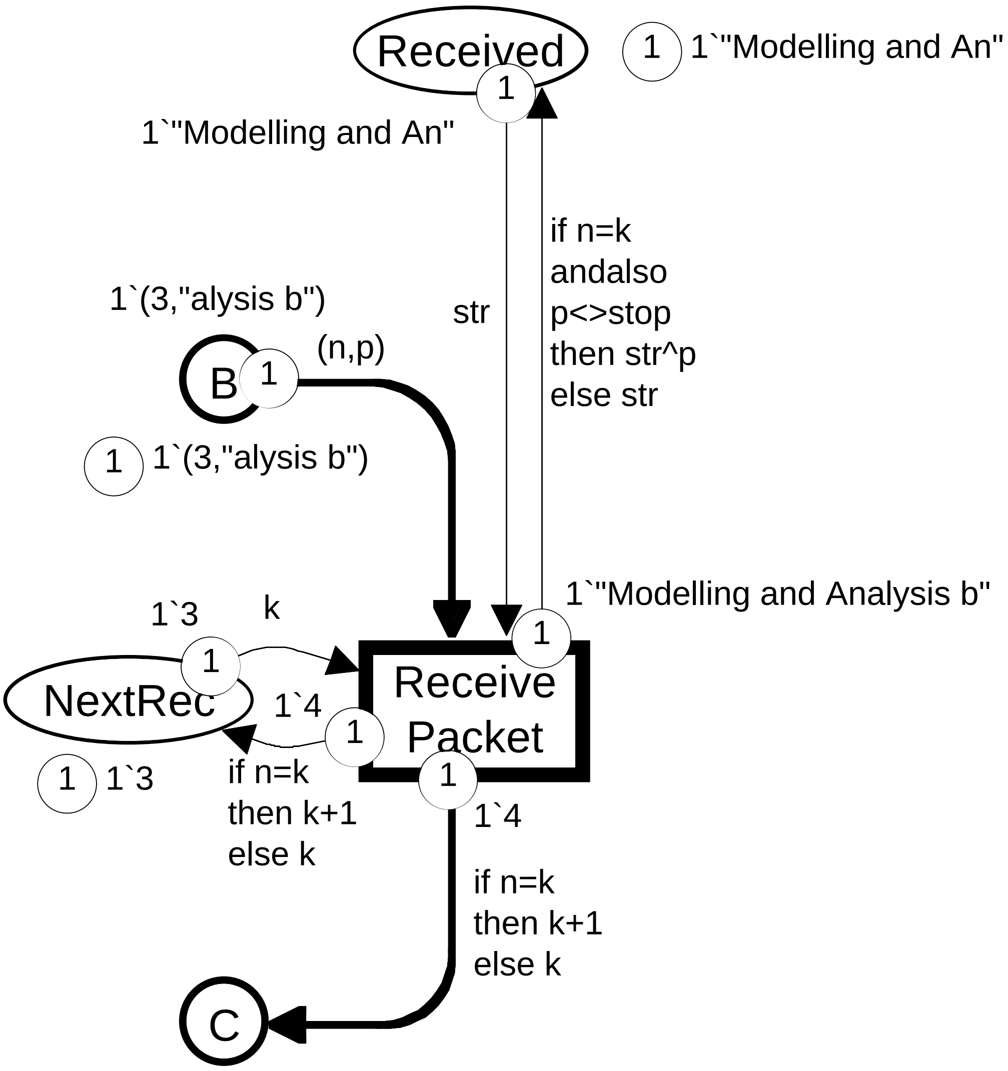
- *Selects* one of the enabled transitions.
- Then he *either* enters a binding or asks the simulator to calculate all the enabled bindings, so that he can select one.

Execution of a step

The simulator:

- Checks the *legality and enabling* of the binding.
- Calculates the *result of the execution.*

The *user determines* whether the simulator displays the tokens which are added/removed:



Interactive simulation with random selection of steps

The simulator *chooses* between conflicting transitions and bindings (by means of a *random number generator*).

- The user can *observe* all details, e.g., the markings the enabling and the added/removed tokens.
- The simulator *shows the page* on which each step is executed – by moving the corresponding window to the top of the screen.
- The user can set *breakpoints* so that he has the necessary time to inspect markings, enablings, etc.

A simulation with this amount of graphical feedback is *slow* (typically a few transitions per minute):

- It takes a lot of time to update the graphics.
- A user has no chance to follow a fast simulation.

It is possible to *turn off* selected parts of the *graphical feedback*, e.g.:

- Added and removed *tokens*.
- Observation of *uninteresting pages*.

Automatic simulation

The simulator *chooses* between conflicting transitions and bindings (by means of a *random number generator*).

The user does *not* intend to follow the simulation:

- The simulation can be *very fast* – several hundred steps per second.
- The user specifies some *stop criteria*, which determine the duration of the simulation.
- When the simulation stops the graphics of the CP-net is *updated*.
- Then the user can inspect all details of the graphics, e.g., the *enabling* and the *marking*.
- Automatic simulations can be *mixed* with interactive simulations.

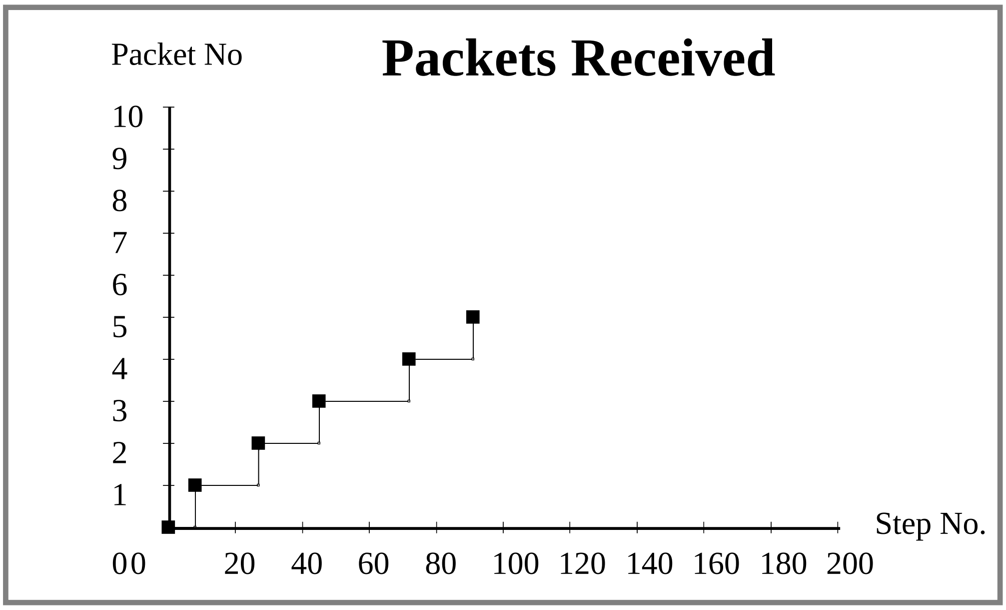
To find out what happens during an *automatic simulation* the user has a number of choices.

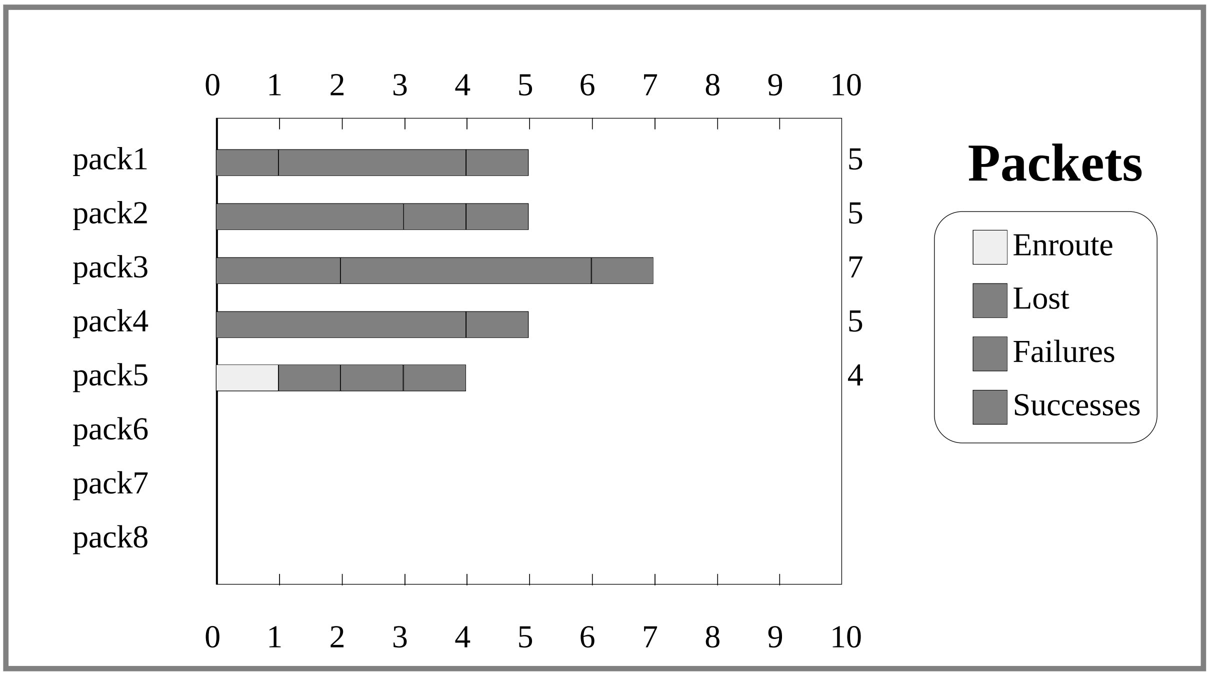
Simulation report

```
1    SendPack@(1:Top#1)   {n = 1, p = "Modellin"}
2    TranPack@(1:Top#1)   {n = 1, p = "Modellin", r = 6, s = 8}
3    SendPack@(1:Top#1)   {n = 1, p = "Modellin"}
4    TranPack@(1:Top#1)   {n = 1, p = "Modellin", r = 3, s = 8}
5    RecPack@(1:Top#1)    {k = 1, n = 1, p = "Modellin",
                           str = ""}
6    SendPack@(1:Top#1)   {n = 1, p = "Modellin"}
7    SendPack@(1:Top#1)   {n = 1, p = "Modellin"}
8    TranAck@(1:Top#1)    {n = 2, r = 2, s = 8}
9    TranPack@(1:Top#1)   {n = 1, p = "Modellin", r = 7, s = 8}
10   RecPack@(1:Top#1)    {k = 2, n = 1, p = "Modellin",
                           str = "Modellin"}
11   RecAck@(1:Top#1)     {k = 1, n = 2}
12   RecPack@(1:Top#1)    {k = 2, n = 1, p = "Modellin",
                           str = "Modellin"}
13   TranAck@(1:Top#1)    {n = 2, r = 7, s = 8}
14   TranPack@(1:Top#1)   {n = 1, p = "Modellin", r = 6, s = 8}
15   RecAck@(1:Top#1)     {k = 2, n = 2}
16   SendPack@(1:Top#1)   {n = 2, p = "g and An"}
17   TranAck@(1:Top#1)    {n = 2, r = 6, s = 8}
18   RecPack@(1:Top#1)    {k = 2, n = 1, p = "Modellin",
                           str = "Modellin"}
19   RecAck@(1:Top#1)     {k = 2, n = 2}
20   SendPack@(1:Top#1)   {n = 2, p = "g and An"}
```

The *simulation report* shows the *transitions* which have occurred. The user determines whether he also wants to see the *bindings*.

Charts

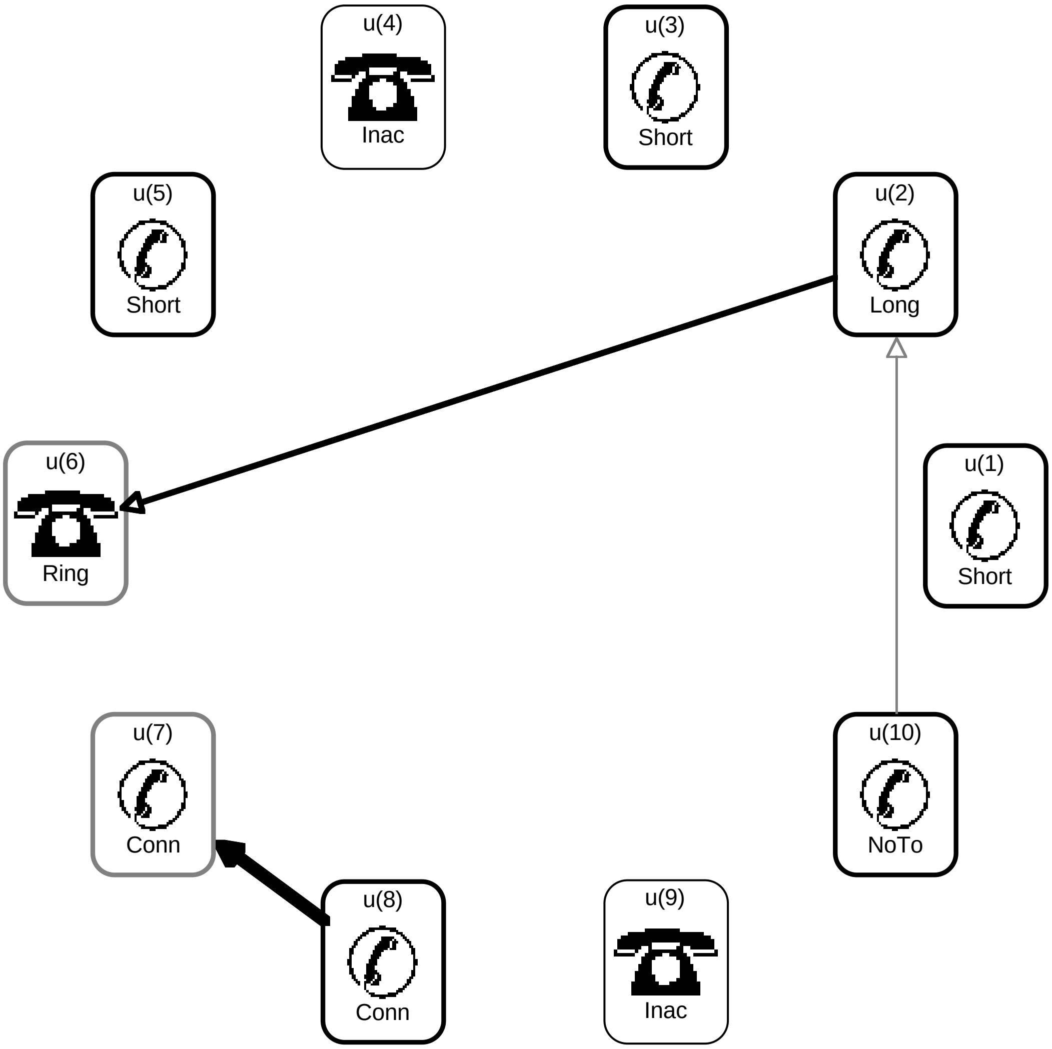




These charts are used to show the *progress* of a simulation of the simple protocol:

- The upper chart is updated each time a new packet is *successfully received.*
- The lower chart is updated for *each 50 steps.*

Other kinds of graphics

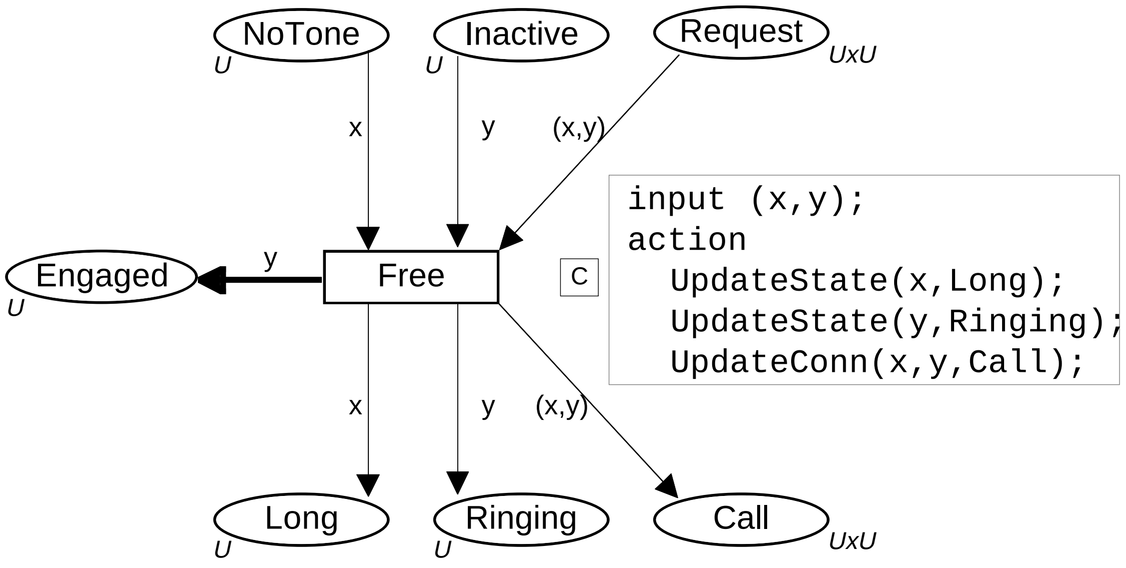


This graphic is used to display the state of a *simple telephone system.* The graphics is updated each time one of the telephones changes to a new state:

- Telephones u(7) and u(8) are *connected.*
- Telephone u(2) is calling u(6) which is *ringing.*
- Telephone u(10) is calling u(2). This call will *not succeed* because u(2) already is engaged.

Code segments

Each transition may have a code segment, i.e., a sequence of *program instructions* which are executed each time the transition occurs.



```
input (x,y);
action
  UpdateState(x,Long);
  UpdateState(y,Ringing);
  UpdateConn(x,y,Call);
```

- The instructions in code segment are used to *update charts and graphics.*
- This is done by calling a number of *library functions.*
- Usually, the code segment does *not* influence the *behaviour* of the CP-net (i.e., the enabling and occurrence).
- However, a code segment may *read and write* from *files.*
- In this way it is possible to *input values* to be used during the simulation, or to *output simulation results.*

Standard ML

Declarations, net inscriptions and code segments are specified in a *programming language* called *Standard ML*.

- *Strongly typed*, *functional* language.
- *Data types* can be:
  - *Atomic* (integers, reals, strings, booleans and enumerations).
  - *Structured* (products, records, unions, lists and subsets).
- Arbitrary complex *functions* and *operations* can be defined (polymorphism and overloading).
- Computational power of expressions are equivalent to *lambda calculus* (and hence to Turing machines).
- Developed at *Edinburgh University* by Robin Milner and his group.
- Standard ML is well-known, well-tested and very general. Several *text books* are available.

Time analysis

CP-nets can be extended with a *time concept*. This means that the *same language* can be used to investigate:

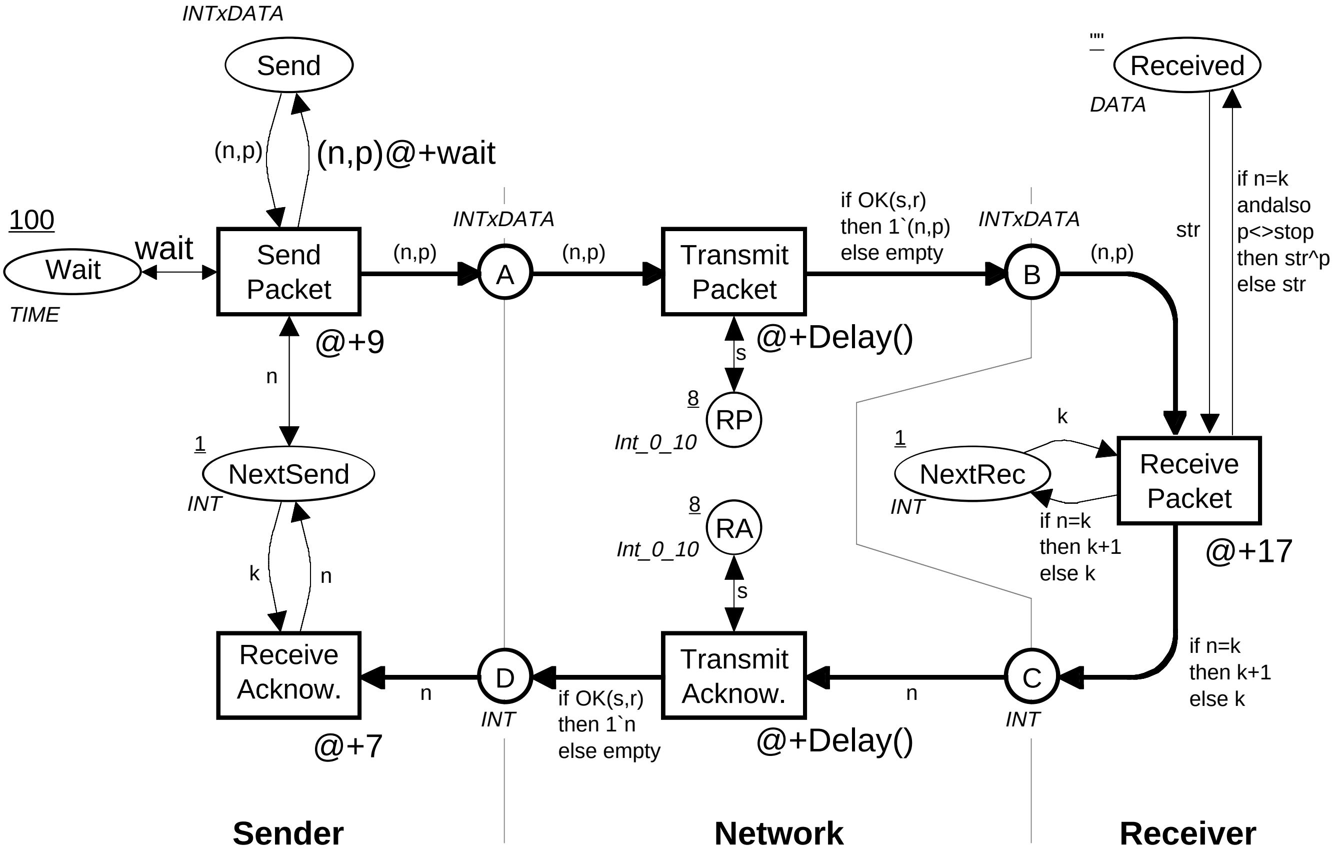
- *Logical correctness.*
  Desired functionality, absence of deadlocks, etc.
- *Performance.*
  Remove bottlenecks. Predict mean waiting times and average throughput. Compare different strategies.

In a timed CP-net each token carries a *colour* (data value) and a *time stamp* (telling when it can be used).

Time stamps are specified by expressions:

- Time stamps can depend upon *colour values*.
- Time stamps can be specified by *probability distributions.*
- This means that we, e.g., can specify *fixed* delays, *interval* delays and *exponential* delays.

A timed CP-net for protocol

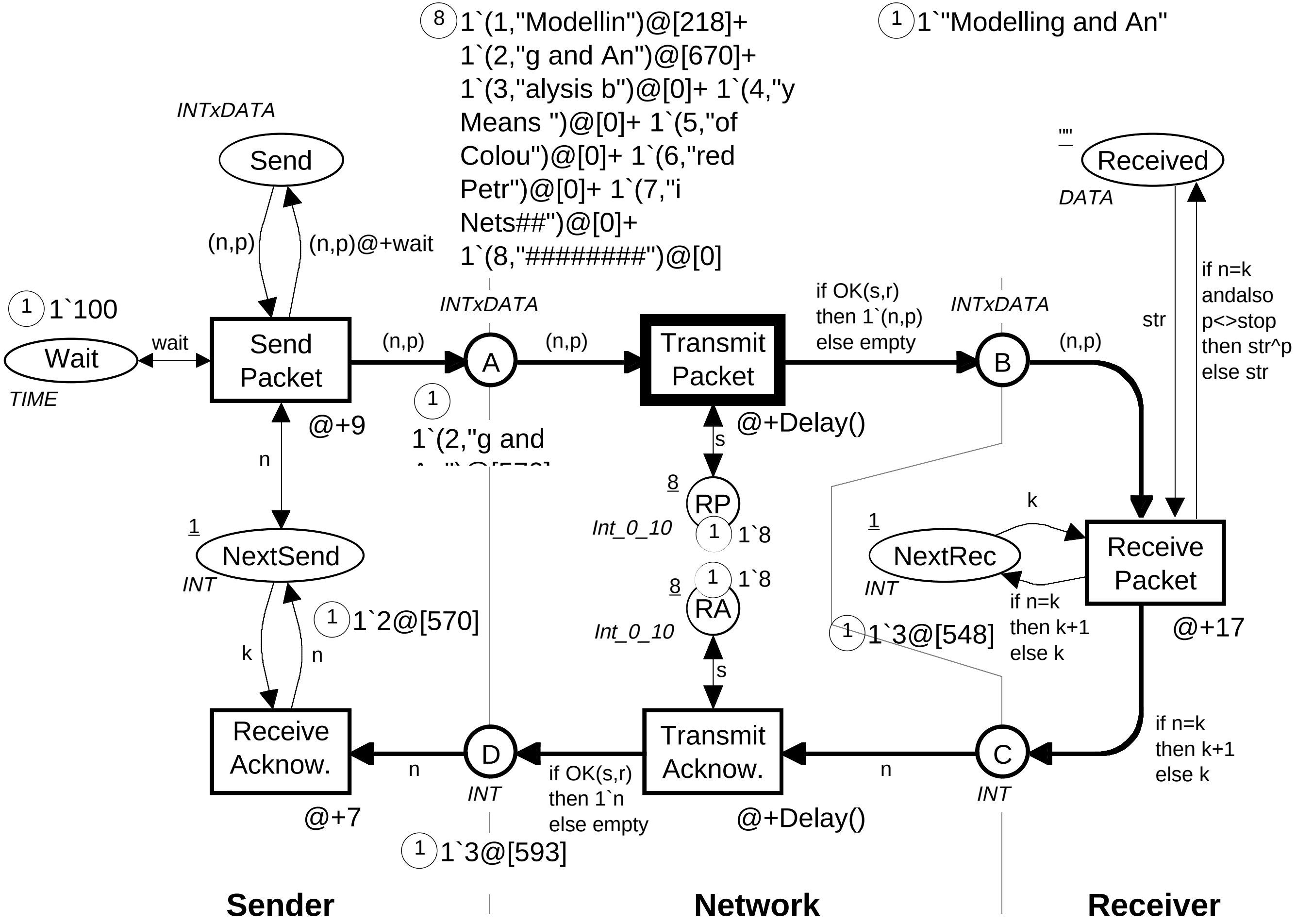


- For the three *Send* and *Receive* operations we specify a *fixed delay.*
- For the *network* we specify an *interval delay*, i.e., random delay between 25 and 75 time units.
- The token colour on place *Wait* specifies the delay between two *retransmissions* of the same packet.

The computer tools for CP-nets also support simulation of *timed* CP-nets.

Timed simulation of protocol

Time: 570



- Model time is now 570.
- *Send Packet* has sent a copy of packet no. 2 at *time 570.*
- If no acknowledgement arrives *another copy* of packet no. 2 will be sent at *time 670.*
- The only transition which is enabled at time 570 is *Transmit Packet*.

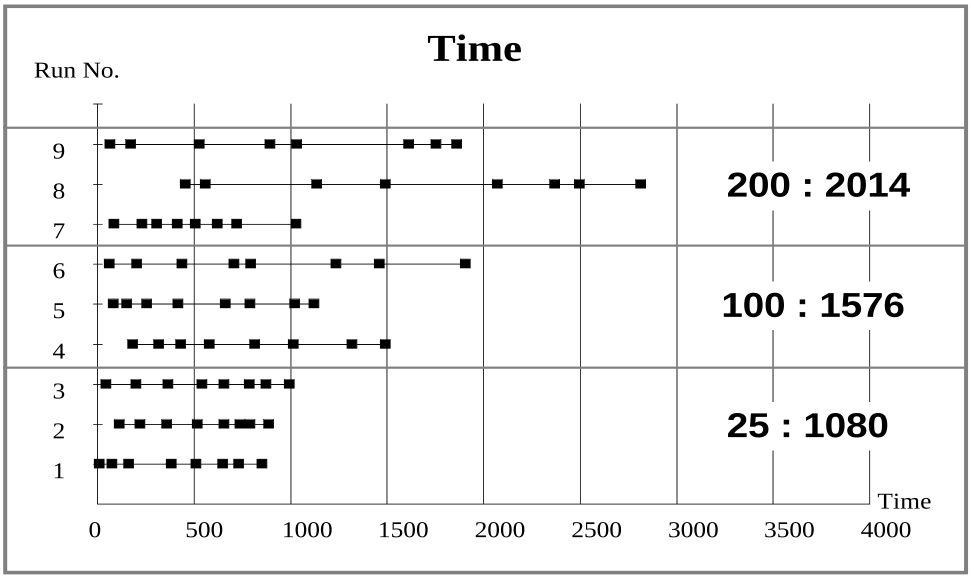
Timed simulations

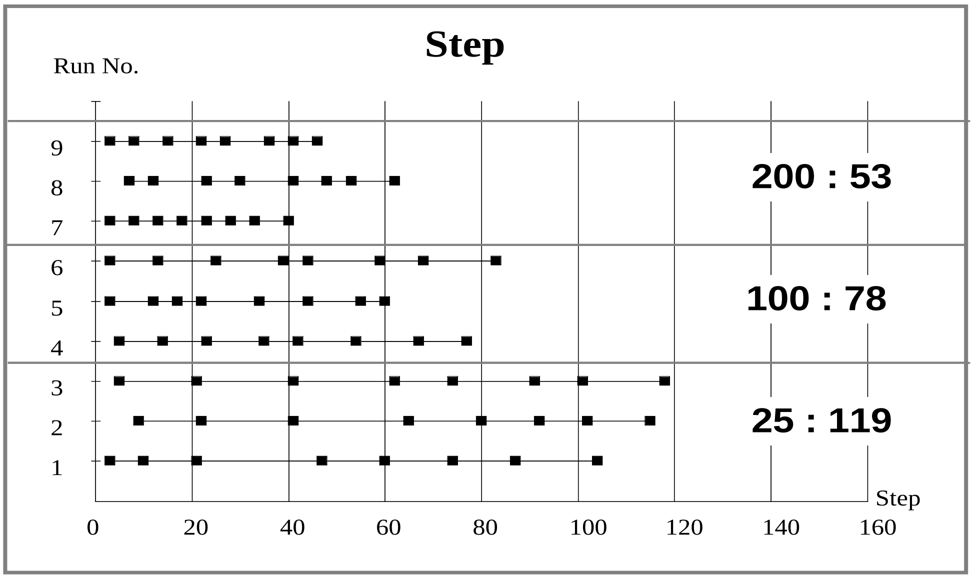
Timed simulations have the *same facilities* as untimed simulations, e.g.:

- We can *switch* between *interactive* and *automatic* simulation.
- *Simulation reports* tell the time at which the individual transitions occurred.
- We can use *charts* and other kinds of *reporting facilities.*

It is easy to *switch* between a *timed* and an *untimed simulation.*

Charts for a timed simulation





- *Short interval* between retransmisions implies *fast transmission* with *heavy use of the network.*
- *Long interval* between retransmisions implies *slow transmission* with *less use of the network.*
- To get *reliable results* it is necessary to make a *large number* of *lengthy* simulation runs.

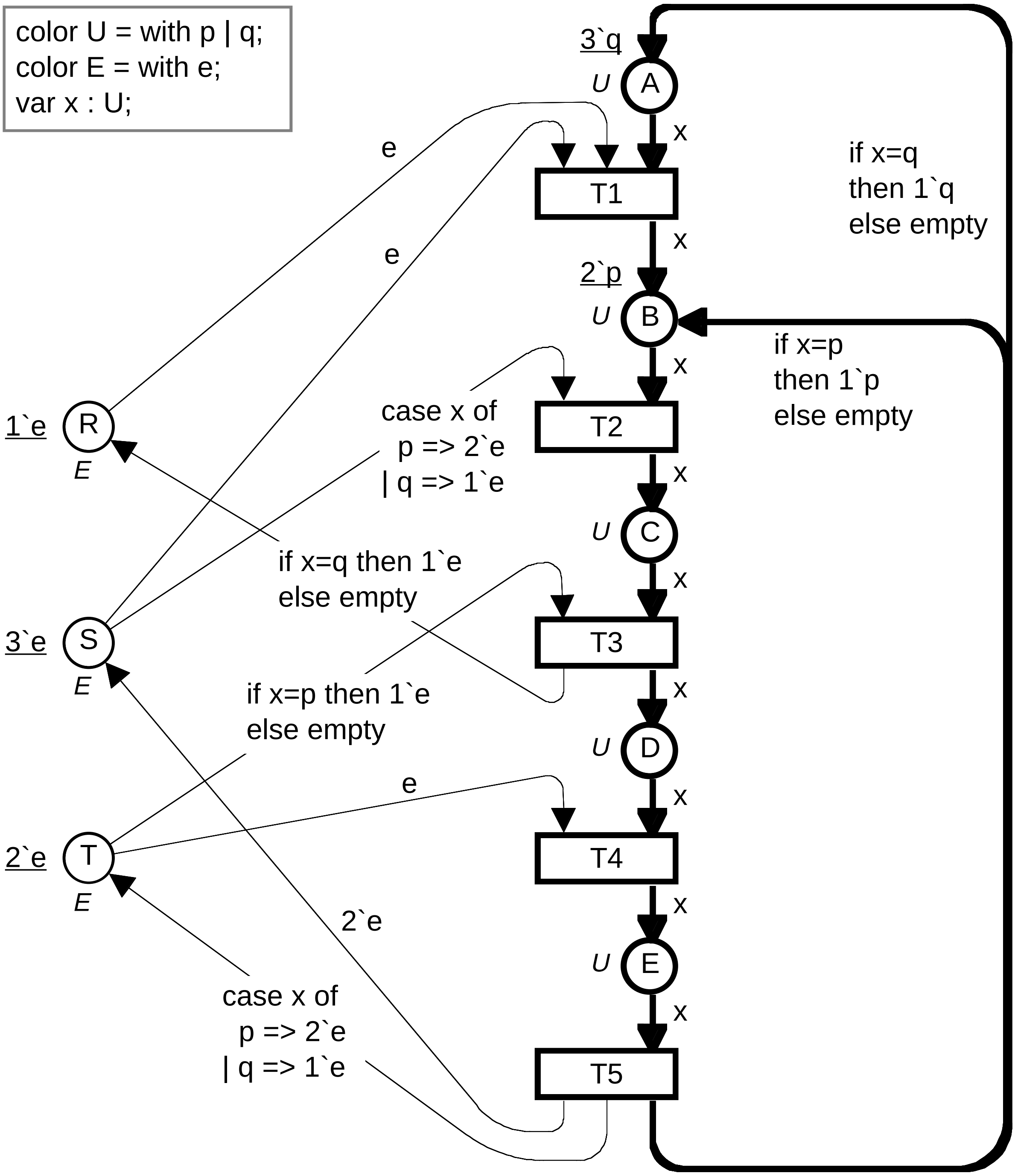
Part 4: Verification of CP-nets

In this part of the talk we describe the two most important methods for *verification* of CP-nets:

- *State spaces* (also called reachability graphs and occurrence graphs).
- *Place invariants*.

We also describe how the verification methods are supported by *computer tools*.

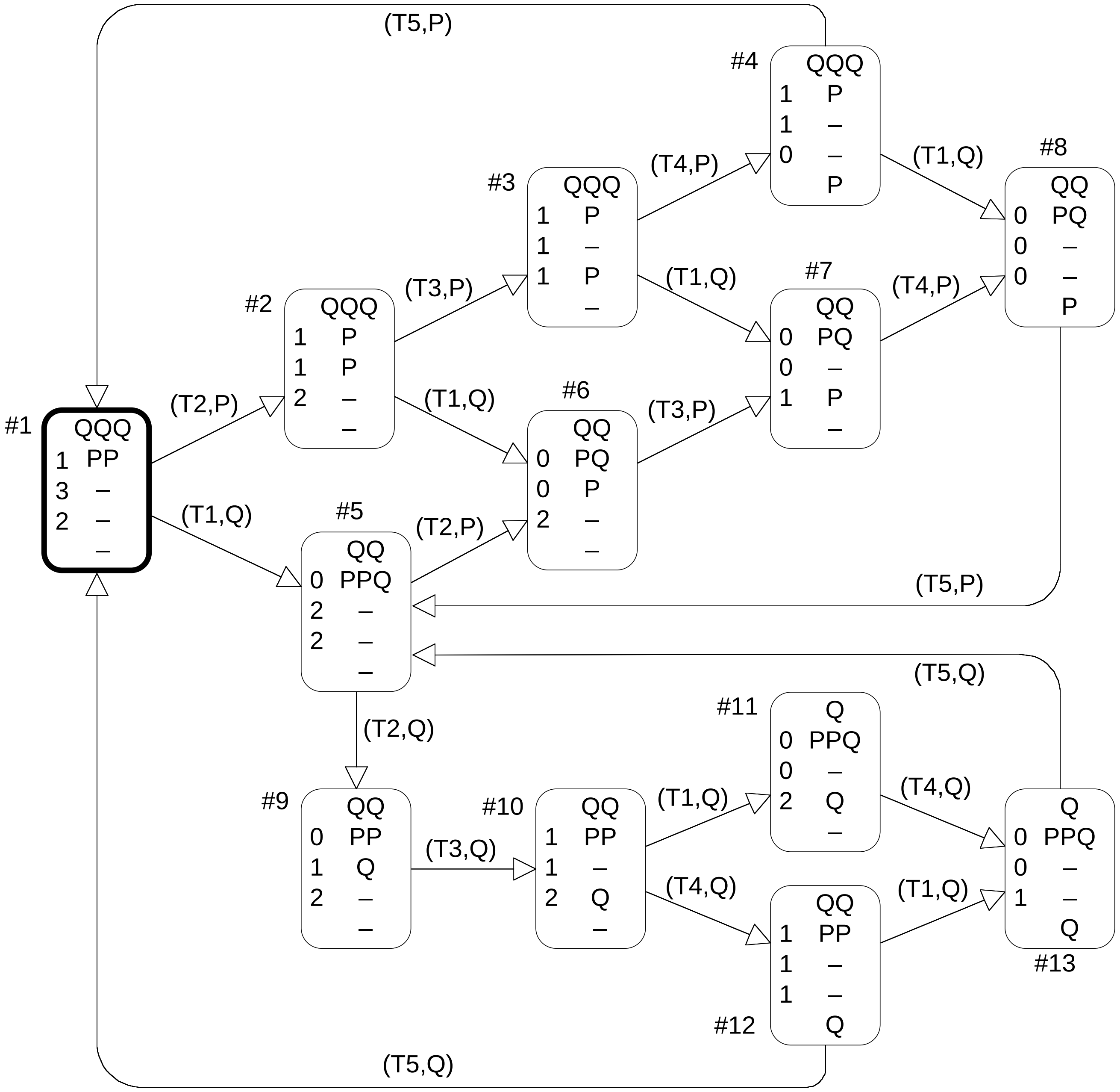
State space analysis



```
color U = with p | q;
color E = with e;
var x : U;
```

To obtain a *finite* state space we remove the cycle counters. Otherwise there would be an *infinite* number of reachable markings.

State space for resource allocation



Directed graph with:

- A *node* for each *reachable marking* (i.e., state).
- An *arc* for each *occurring binding element.*

Some questions that can be answered from state spaces

Boundedness properties:

- What is the *maximal number* of tokens on the different places?
- What is the *minimal number* of tokens on the different places?
- What are the *possible token colours*?

Home properties:

- Is it *always* possible to *return* to the initial marking?

Liveness properties:

- Are all transitions live, i.e., can they *always* become enabled *again*?

State space report for resource allocation system

Statistics

Occurrence Graph

Nodes: 13
Arcs: 20
Secs: 1
Status: Full

Scc Graph

Nodes: 1
Arcs: 0
Secs: 1

Boundedness Properties

Upper Integer Bounds

A: 3
B: 3
C: 1
D: 1
E: 1

R: 1
S: 3
T: 2

Lower Integer Bounds

A: 1
B: 1
C: 0
D: 0
E: 0

R: 0
S: 0
T: 0

Upper Multi-set Bounds

A: 3`q
B: 2`p+ 1`q
C: 1`p+ 1`q
D: 1`p+ 1`q
E: 1`p+ 1`q

R: 1`e
S: 3`e
T: 2`e

Lower Multi-set Bounds

A: 1`q
B: 1`p
C: empty
D: empty
E: empty

R: empty
S: empty
T: empty

State space report (continued)

Home Properties

Home Markings: All

Liveness Properties

Dead Markings: None

Live Transitions: All

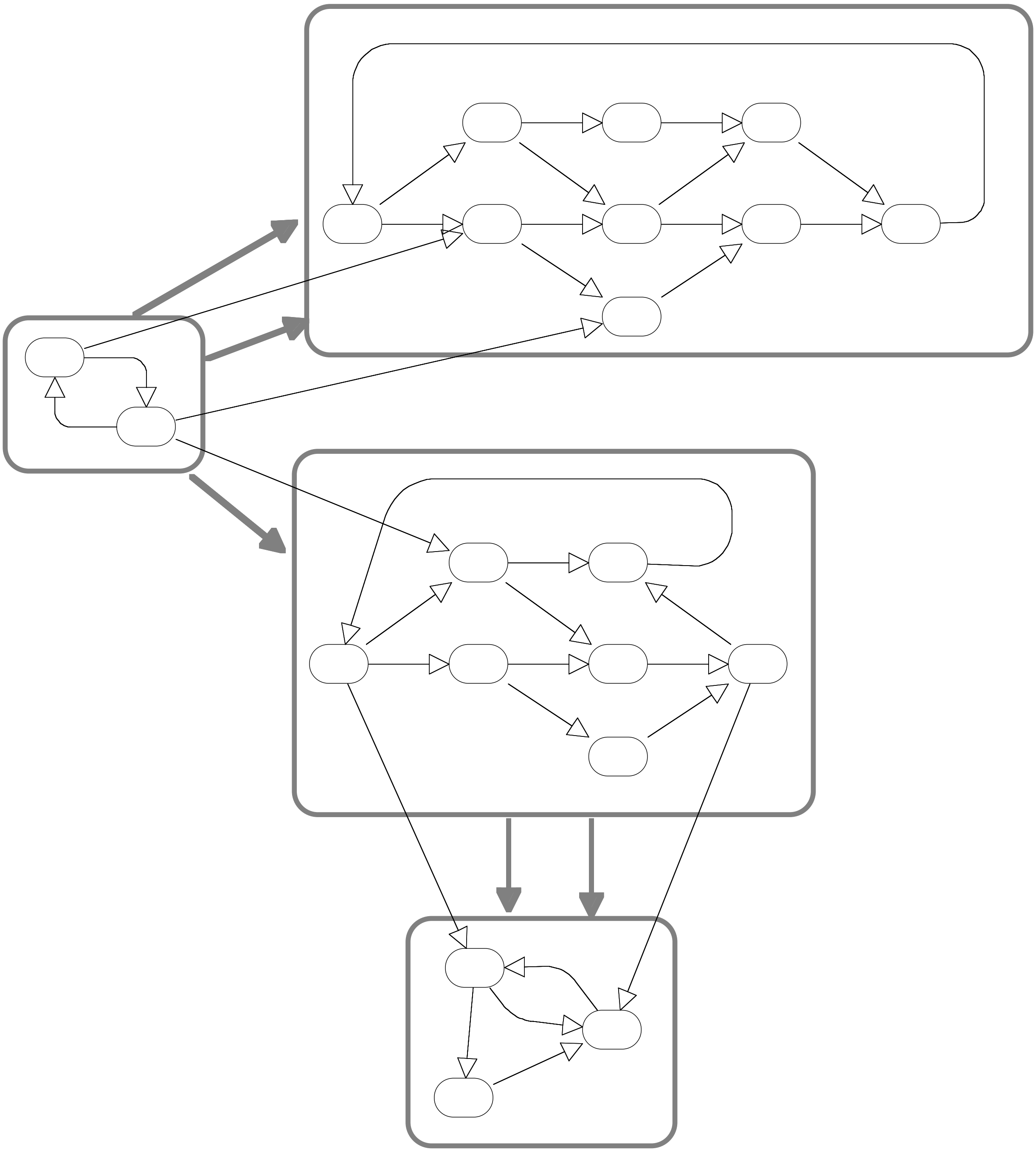
Fairness Properties

T1:	No Fairness
T2:	Impartial
T3:	Impartial
T4:	Impartial
T5:	Impartial

Generation of the state space report takes only a *few seconds*.

- The report contains a lot of *useful information* about the *behaviour* of the CP-net.
- The report is excellent for *locating errors* or to *increase our confidence* in the correctness of the system.

Strongly connected components



- Subgraph where *all nodes are reachable from each other*.
- *Maximal* subgraph with this property.

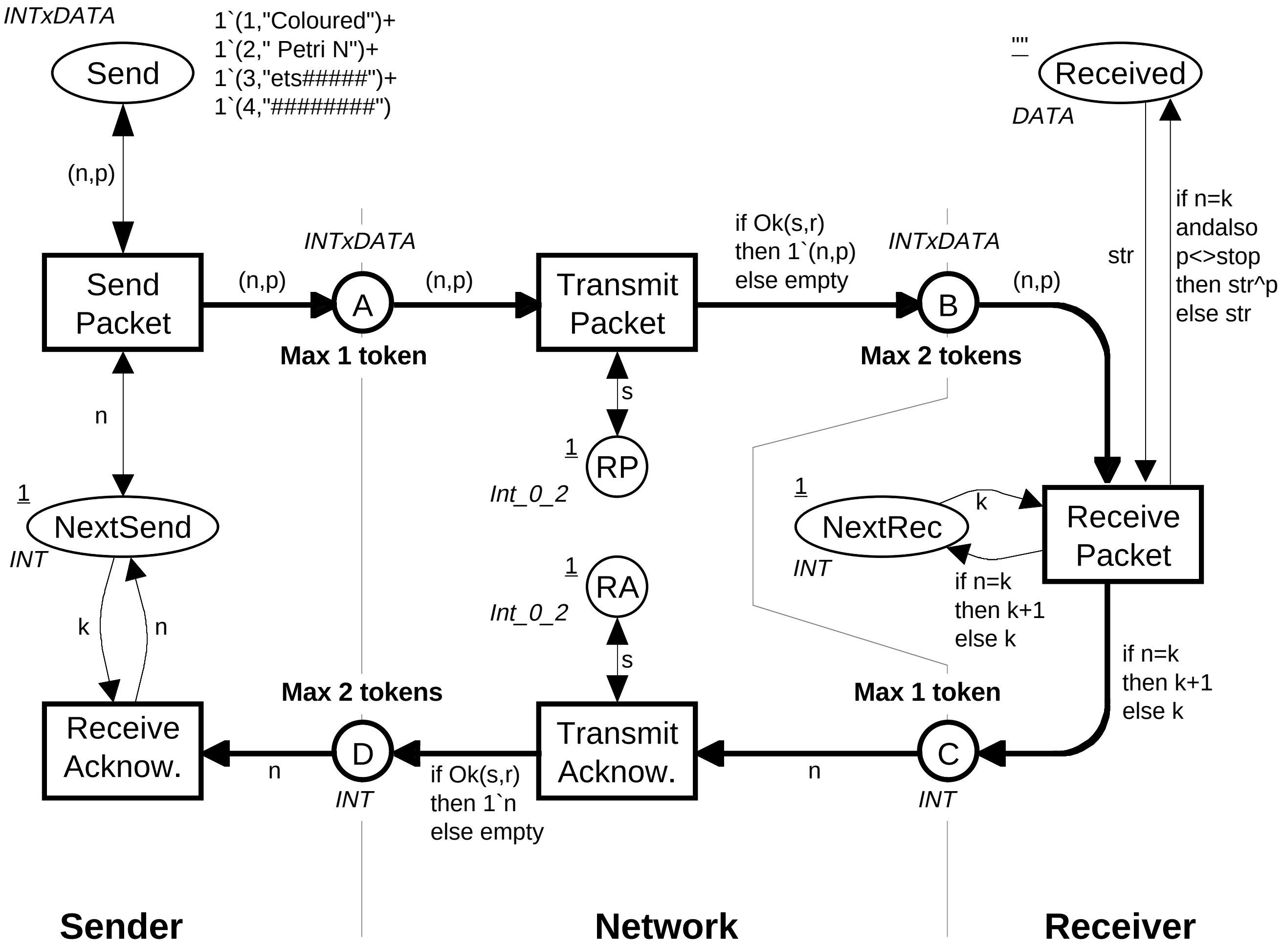
Strongly connected components are very useful

There are often *much fewer* strongly connected components than nodes:

- A *cyclic system* has only *one* strongly connected component.
- This is, e.g., the case for the resource allocation system.
- The *strongly connected components* can be determined in *linear time*, e.g., by Tarjan’s algorithm.

Strongly connected components can be used to answer questions about *home properties* and *liveness properties*.

State space for simple protocol



To obtain a *finite* state space we limit the number of tokens on the “buffer” places A, B, C and D. Otherwise there would be an *infinite* number of reachable markings.

Moreover, we now only have *4 packets* and a *binary choice* between success and failure.

State space report for protocol

Statistics

Occurrence Graph

Nodes:	4298
Arcs:	15887
Secs:	53
Status:	Full

Scc Graph

Nodes:	2406
Arcs:	11677
Secs:	17

Boundedness Properties

Upper Integer Bounds

A:	1
B:	2
C:	1
D:	2
NextRec:	1
NextSend:	1
RA:	1
RP:	1
Received:	1
Send:	4

Lower Integer Bounds

A:	0
B:	0
C:	0
D:	0
NextRec:	1
NextSend:	1
RA:	1
RP:	1
Received:	1
Send:	4

State space report (continued)

Upper Multi-set Bounds

```
A:          1`(1,"Coloured")+ 1`(2," Petri N")+
            1`(3,"ets#####")+ 1`(4,"########")
B:          2`(1,"Coloured")+ 2`(2," Petri N")+
            2`(3,"ets#####")+ 2`(4,"########")
C:          1`2+ 1`3+ 1`4+ 1`5
D:          2`2+ 2`3+ 2`4+ 2`5

NextRec:    1`1+1`2+1`3+1`4+1`5
NextSend:   1`1+1`2+1`3+1`4+1`5

RA:         1`1
RP:         1`1

Received:   1`""+ 1`"Coloured"+ 1`"Coloured Petri N"+
            1`"Coloured Petri Nets#####"
Send:       1`(1,"Coloured")+ 1`(2," Petri N")+
            1`(3,"ets#####")+ 1`(4,"########")
```

Lower Multi-set Bounds

```
A:          empty
B:          empty
C:          empty
D:          empty

NextRec:    empty
NextSend:   empty

RA:         1`1
RP:         1`1

Received:   empty
Send:       1`(1,"Coloured")+ 1`(2," Petri N")+
            1`(3,"ets#####")+ 1`(4,"########")
```

State space report (continued)

Home Properties

Home Markings:	1	[452]

Liveness Properties

Dead Markings:	1	[452]
Live Transitions:	None	

Fairness Properties

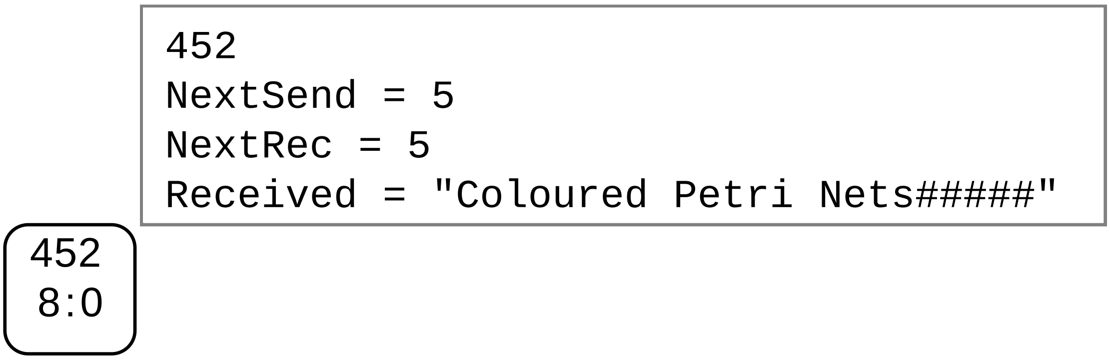
Send Packet:	Impartial
Transmit Packet:	Impartial
Receive Packet:	No Fairness
Transmit Acknow:	No Fairness
Receive Acknow:	No Fairness

Generation of the state space report takes only a *few seconds*.

- The report contains a lot of *useful information* about the *behaviour* of the CP-net.
- The report is excellent for *locating errors* or to *increase our confidence* in the correctness of the system.

Investigation of dead marking

We ask the system to display marking number 452.



```
452
NextSend = 5
NextRec = 5
Received = "Coloured Petri Nets#####"
```

Marking no. 452 is the *desired final marking* (all packets has been received in the correct order)

Marking 452 is *dead:*

- This implies that the protocol is *partially correct* (if execution stops it stops in the desired final marking).

Marking 452 is a *home marking:*

- This implies that we *always have a chance to finish correctly* (it is impossible to reach a state from which we cannot reach the desired final marking).

Investigation of shortest path

We ask the system to calculate one of the *shortest paths* from the initial marking to the dead marking:

```
val path =
NodesInPath(1,452);
```

```
> val path =
[1,2,3,5,8,11,15,20,27,38,50,
64,80,102,133,164,199,243,
301,375,452] : Node list
```

```
Length(path);
```

```
> 20 : int
```

The calculated path contains *20 transitions*.

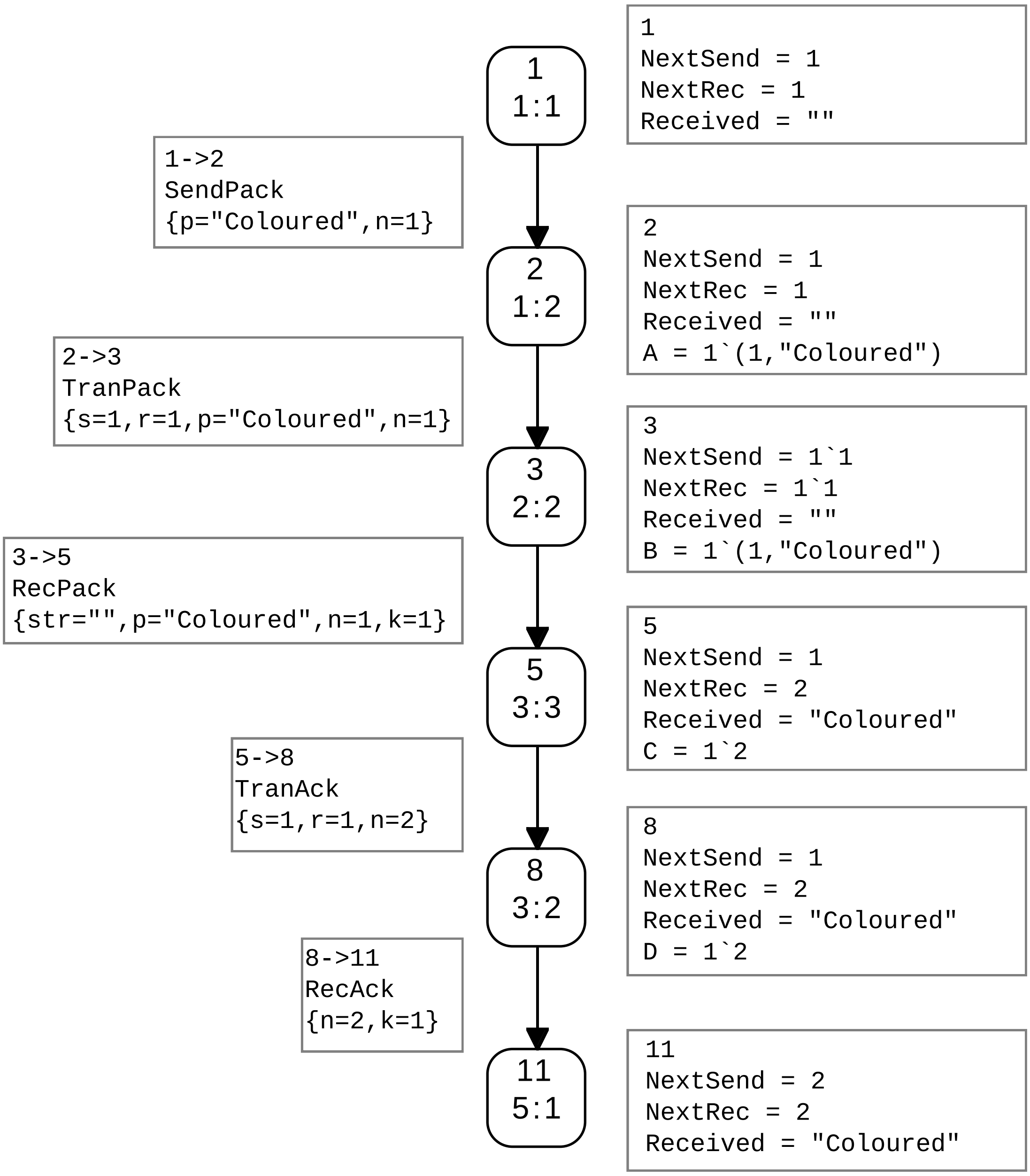
- This is as expected because there are *4 packets* which each need *5 transitions* to occur.

Drawing of shortest path

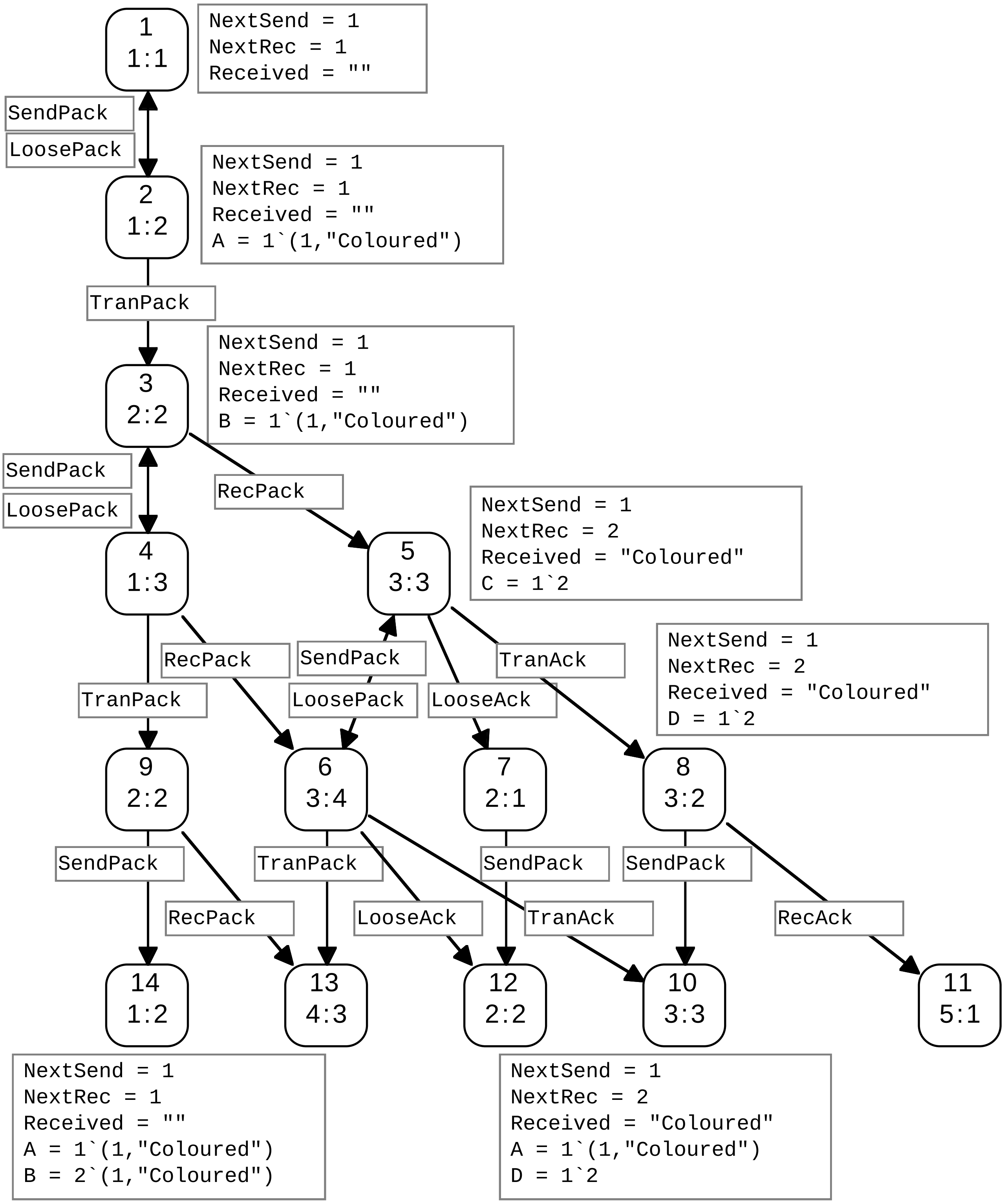
We ask the system to draw the *first six nodes* in the calculated shortest path:

```
DisplayNodePath; [1,2,3,5,8,11];
```

```
> () : unit
```



Draw subgraph

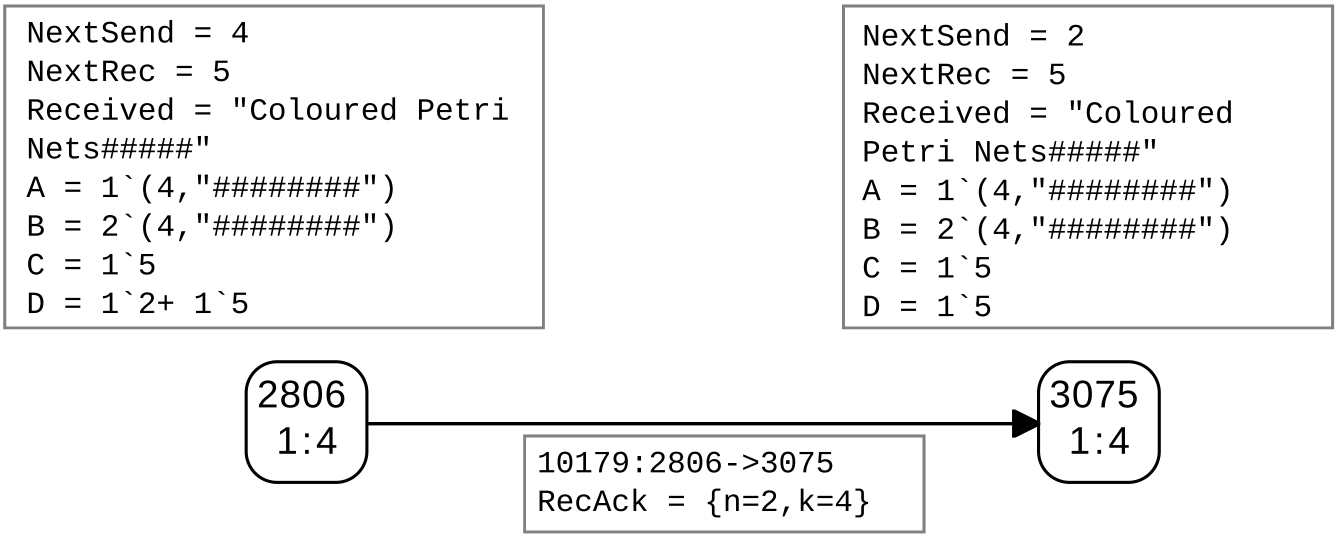


Non-standard questions

We ask the system to search *all arcs* in the *entire graph* and return the *first 10 arcs* where *NextSend* has a *larger* value in the *source marking* than it has in the *destination marking*.

```
PredArcs
      (EntireGraph,
       fn a => ((ms_to_col(Mark.NextSend 1
                  (SourceNode a))) >
                (ms_to_col(Mark.NextSend 1
                  (DestNode a)))),
       10)
end;
```

```
>[10179,10167,10165,10159,10055,10052,10035,
10031,10019,10007] : Arc list
```



```
NextSend = 4
NextRec = 5
Received = "Coloured Petri
Nets#####"
A = 1`(4,"########")
B = 2`(4,"########")
C = 1`5
D = 1`2+ 1`5
```

```
NextSend = 2
NextRec = 5
Received = "Coloured
Petri Nets#####"
A = 1`(4,"########")
B = 2`(4,"########")
C = 1`5
D = 1`5
```

```
10179:2806->3075
RecAck = {n=2,k=4}
```

Temporal logic

It is also possible to make questions by means of a CTL-like *temporal logic*.

Usually CTL focuses on queries about *state properties*, e.g.:

- Inv(Pos(M))
  checks whether M is a *home marking*.
- Ev(dead)
  checks whether there are any infinite occurrence sequences.

Our version of CTL also allows queries about *transitions* and *binding elements*.

- Inv(Pos(t in Arc))
  checks whether transition t is *live*.

Timed CP-nets

The computer tools for CP-nets also support state space analysis of *timed* CP-nets.

State spaces – pro/contra

State spaces are *powerful* and *easy* to use.

- The main drawback is the *state explosion, i.e., the size of the state space.*
- The present version of our tool handles graphs with 100,000 nodes and 500,000 arcs. For many systems this is *not sufficient.*

Fortunately, it is sometimes possible to construct much more *compact* state spaces – *without loosing information.*

- This is done by exploiting the inherent *symmetries* of the modelled system.
- We define two *equivalence relations* (one for markings and one for binding elements).
- The condensed state spaces are often *much smaller* (polynomial size instead of exponential).
- The condensed state spaces contain the *same information* as the full state spaces.

Place invariants analysis

The basic idea is similar to the use of *invariants* in *program verification*.

- A place invariant is an *expression* which is satisfied for all reachable markings.
- The expression *counts* the tokens of the marking – using a specified set of weights.

We first *construct* a set of place invariants.

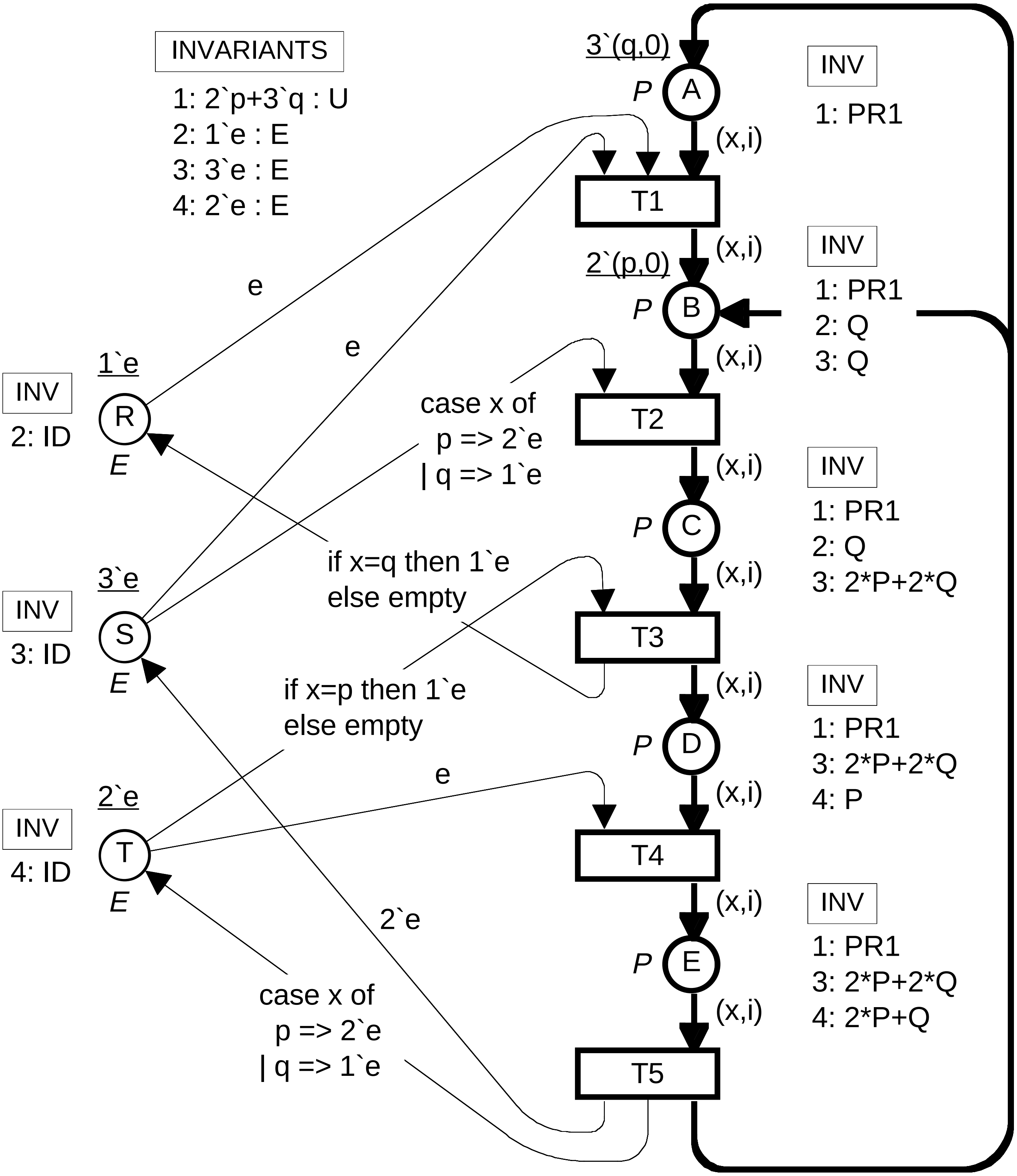
Then we check whether they are *fulfilled*.

- This is done by showing that each occurring binding element *respects* the invariants.
- The *removed* set of tokens must be identical to the *added* set of tokens – when the weights are taken into account.

Finally, we use the place invariants to *prove* behavioural properties of the CP-net.

- This is done by a *mathematical proof*.

Example of place invariants



Place invariants for resource allocation system

To specify the weights we use *three functions:*

- PR_1 is a *projection* function: (x,i) --> x.
- P is an *indicator* function: (p,i) --> 1`e; (q,i) -->Ø.
- Q is an *indicator* function: (p,i) --> Ø; (q,i) -->1`e.
- P and Q “counts” the number of p and q tokens.

$$PR_1(M(A)+M(B)+M(C)+M(D)+M(E)) = 2`p+3`q$$

$$M(R) + Q(M(B)+M(C)) = 1`e$$

$$M(S) + Q(M(B)) + (2*P+2*Q)(M(C)+M(D)+M(E)) = 3`e$$

$$M(T) + P(M(D)) + (2*P+Q)(M(E)) = 2`e$$

A more readable version of the place invariants

$PR_1(A+B+C+D+E) = 2`p+3`q$

$R + Q(B+C) = 1`e$

$S + Q(B) + (2*P+2*Q)(C+D+E) = 3`e$

$T + P(D) + (2*P+Q)(E) = 2`e$

The place invariants can be used to *prove* properties of the resource allocation system, e.g., that it is *impossible to reach a dead marking*.

Tool support for place invariants

Check of place invaritans:

- The *user* proposes a set of weights.
- The *tool* checks whether the weights constitute a place invariant.

Automatic calculation of all place invariants:

- This is possible, but it is a very *complex* task.
- Moreover, it is difficult to represent the results on a *useful form*, i.e., a form which can be used by the system designer.

Interactive calculation of place invaritans:

- The *user* proposes some of the weights.
- The *tool* calculates the *remaining weights* – if possible.

Interactive calculation of place invariants is *much easier* than a fully automatic calculation.

How to use place invariants

Invariants in ordinary *programming languages:*

- No one would construct a large program – and then expect *afterwards* to be able to calculate invariants.
- Instead invariants are constructed *together* with the program.

For *CP-nets* we should do the same:

- During the system specification and modelling the designer gets a lot of *knowledge* about the system.
- Some of this knowledge can easily be formulated as *place invariants.*
- The invariants can be *checked* and in this way it is possible to find *errors*.
- It can be seen *where* the errors are.

Some *prototypes* of computer tools for invariants analysis do exist. However, none of them are at a state where they can be widely used.

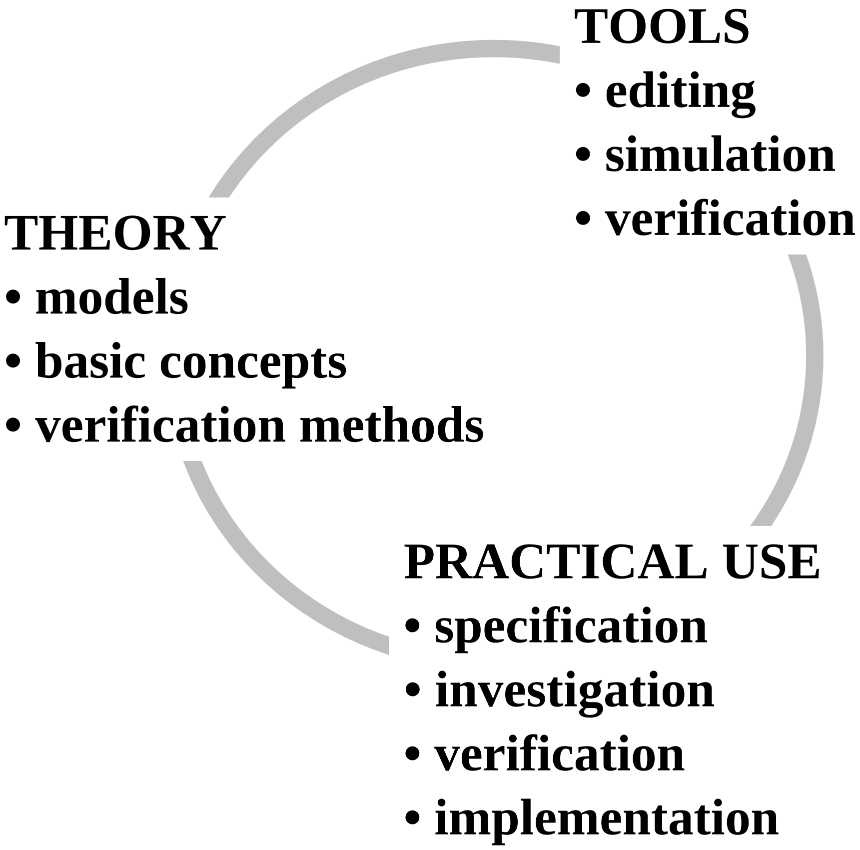
Place invariants – pro/contra

From place invariants it is possible to prove many kinds of *behavioural properties*.

- Invariants can be used to make *modular verification* – because it is possible to combine invariants of the individual pages.
- Invariants can be used to verify *large systems* – without computational problems.
- The user needs some ingenuity to *construct* invariants. This can be supported by *computer tools* – interactive process.
- The user also needs some ingenuity to *use* invariants. This can also be supported by *computer tools* – interactive process.
- Invariants can be used to verify a system – without fixing the *system parameters* (such as the number of sites in the data base system).

Conclusion

One of the main reasons for the success of CP-nets is the fact that we – *simultaneously* – have worked with:



TOOLS
- **editing**
- **simulation**
- **verification**

THEORY
- **models**
- **basic concepts**
- **verification methods**

PRACTICAL USE
- **specification**
- **investigation**
- **verification**
- **implementation**

More information on CP-nets

The following WWW pages contain a lot of information about CP-nets and their computer tools:

http://www.daimi.aau.dk/CPnets/

A detailed introduction to CP-nets can be found in the following papers/books:

K. Jensen: Coloured Petri Nets: *A High-level Language for System Design and Analysis.* In: G. Rozenberg (ed.): Advances in Petri Nets 1990, Lecture Notes in Computer Science Vol. 483, Springer-Verlag 1991, 342–416. Also in K. Jensen, G. Rozenberg (eds.): High-level Petri Nets. Theory and Application. Springer-Verlag, 1991, 44–122.

K. Jensen: *An Introduction to the Theoretical Aspects of Coloured Petri Nets*. In: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): A Decade of Concurrency, Lecture Notes in Computer Science vol. 803, Springer-Verlag 1994, 230-272.

K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Monographs in Theoretical Computer Science, Springer-Verlag.

- Vol. 1: Basic Concepts, 1992, ISBN: 3-540-60943-1.
- Vol. 2: Analysis Methods, 1994, ISBN: 3-540-58276-2.
- Vol. 3: Practical Use, 1996.

Some of the most important papers on high-level nets, their verification methods and applications have been reprinted in:

K. Jensen, G. Rozenberg (eds.): *High-level Petri Nets. Theory and Application.* Springer-Verlag, 1991, ISBN: 3-540-54125-X.

Different examples of the industrial use of CP-nets can be found in:

G. Balbo, S.C. Bruell, P. Chen, G. Chiola: *An Example of Modelling and Evaluation of a Concurrent Program Using Colored Stochastic Petri Nets: Lamport's Fast Mutual Exclusion Algorithm.* IEEE Transactions on Parallel and Distributed Systems, 3 (1992). Also in K. Jensen, G. Rozenberg (eds.): High-level Petri Nets. Theory and Application. Springer-Verlag, 1991, 533–559.

J. Berger, L. Lamontagne: *A Colored Petri Net Model for a Naval Command and Control System.* In: M. Ajmone-Marsan (ed.): Application and Theory of Petri Nets 1993. Proceedings of the 14th International Petri Net Conference, Chicago 1993, Lecture Notes in Computer Science Vol. 691, Springer-Verlag 1993, 532–541.

C. Capellmann, H. Dibold: *Petri Net Based Specifications of Services in an Intelligent Network. Experiences Gained from a Test Case Application.* In: M. Ajmone-Marsan (ed.): Application and Theory of Petri Nets 1993. Proceedings of the 14th International Petri Net Conference, Chicago 1993, Lecture Notes in Computer Science Vol. 691, Springer-Verlag 1993, 542–551.

L. Cherkasova, V. Kotov, T. Rokicki: *On Net Modelling of Industrial Size Concurrent Systems.* In: M. Ajmone-Marsan (ed.): Application and Theory of Petri Nets 1993. Proceedings of the 14th International Petri Net Conference, Chicago 1993, Lecture Notes in Computer Science Vol. 691, Springer-Verlag 1993, 552–561.

L. Cherkasova, V. Kotov, T. Rokicki: *On Scalable Net Modeling of OLTP.* In PNPM93: Petri Nets and Performance Models. Proceedings of the 5th International Workshop, Toulouse, France 1993, IEEE Computer Society Press, 270–279.

S. Christensen, L.O. Jepsen: *Modelling and Simulation of a Network Management System Using Hierarchical Coloured Petri Nets.* In: E. Mosekilde (ed.): Modelling and Simulation 1991. Proceedings of the 1991 European Simulation Multiconference, Copenhagen, 1991, Society for Computer Simulation 1991, 47–52.

H. Clausen, P.R. Jensen: *Validation and Performance Analysis of Network Algorithms by Coloured Petri Nets.* In PNPM93: Petri Nets and Performance Models. Proceedings of the 5th International Workshop, Toulouse, France 1993, IEEE Computer Society Press, 280–289.

G. Florin, C. Kaiser, S. Natkin: *Petri Net Models of a Distributed Election Protocol on Undirectional Ring.* Proceedings of the 10th International Conference on Application and Theory of Petri Nets, Bonn 1989, 154 –173.

H.J. Genrich, R.M. Shapiro: *Formal Verification of an Arbiter Cascade.* In: K. Jensen (ed.): Application and Theory of Petri Nets 1992. Proceedings of the 13th International Petri Net Conference, Sheffield 1992, Lecture Notes in Computer Science Vol. 616, Springer-Verlag 1992, 205–223.

P. Huber, V.O. Pinci: *A Formal Executable Specification of the ISDN Basic Rate Interface.* Proceedings of the 12th International Conference on Application and Theory of Petri Nets, Aarhus 1991, 1–21.

W.W. McLendon, R.F. Vidale: *Analysis of an Ada System Using Coloured Petri Nets and Occurrence Graphs.* In: K. Jensen (ed.): Application and Theory of Petri Nets 1992. Proceedings of the 13th International Petri Net Conference, Sheffield 1992, Lecture Notes in Computer Science Vol. 616, Springer-Verlag 1992, 384–388.

K.H. Mortensen, V. Pinci: *Modelling the Work Flow of a Nuclear Waste Management Program.* Proceedings of the 15th International Petri Net Conference, Zaragoza 1994, Lecture Notes in Computer Science, Springer-Verlag 1994

V.O. Pinci, R.M. Shapiro: *An Integrated Software Development Methodology Based on Hierarchical Colored Petri Nets.* In: G. Rozenberg (ed.): Advances in Petri Nets 1991, Lecture Notes in Computer Science Vol. 524, Springer-Verlag 1991, 227–252. Also in K. Jensen, G. Rozenberg (eds.): High-level Petri Nets. Theory and Application. Springer-Verlag, 1991, 649– 667.

G. Scheschonk, M. Timpe: *Simulation and Analysis of a Document Storage System.* In: R. Valette (ed.): Application and Theory of Petri Nets 1994. Proceedings of the 15th International Petri Net Conference, Zaragoza 1994, Lecture Notes in Computer Science vol. 815, Springer-Verlag 1992, 454–470.

R.M. Shapiro: *Validation of a VLSI Chip Using Hierarchical Coloured Petri Nets.* Journal of Microelectronics and Reliability, Special Issue on Petri Nets, Pergamon Press, 1991. Also in K. Jensen, G. Rozenberg (eds.): High-level Petri Nets. Theory and Application. Springer-Verlag, 1991, 667– 687.