6. Colored Petri Nets: The Language

6.1. Introduction

In the previous chapter we introduced the extensions with color and time at a conceptual level. Before introducing the third extension (i.e., the addition of hierarchy), we continue with the first two extensions in more detail. In a classical Petri net, the diagram is a complete specification of a process. The colored and timed models as they have been introduced, however, require additional descriptions to result in a complete specification. This is undesirable because these informal descriptions may result in incomplete, ambiguous, and even inconsistent models. Therefore, we provide a concrete language in chapter. For this purpose we use the Colored Petri Net (CPN) language as introduced by Kurt Jensen. This language is supported by an abundance of documentation and tools (e.g., Design/CPN and its successor CPN Tools). Other languages such as ExSpect, CPN/AMI, PEP, PROD, Renew, Artifex, etc. use a slightly different language. These languages also support the concepts as they have been introduced in previous chapter. However, they use a different language to describe place types and transitions.

The reason we choose the CPN language is that it is completely graphical, i.e., the whole model can be expressed in the diagram. The language has formal semantics, i.e., the meaning of every concept is defined in a complete and unambiguous manner. Moreover, an abundance of analysis methods is available for CPN in addition to software products like CPN Tools.

The CPN language, also referred to as CPN-ML, is based on the functional language (Standard) ML. Therefore, CPN inherits the basic types, type constructors, basic functions, operators, and expressions from ML. Therefore, we introduce parts of ML. The goal of this chapter is twofold. On the one hand, it provides a concrete syntax for the concepts introduced in the previous sections. On the other hand, it demonstrates the application of colored and timed Petri nets.

After studying this chapter you are expected to be able to

- explain and use the following concepts: color set, timed color set, variable, arc inscription, multiset, initialization expression, guard, binding element, and function;
- use the basic types and types constructors of CPN-ML, i.e., `int`, `string`, `bool`, `unit`, `product`, `record`, and `list`;
- use the basic operators for these types;

- formulate arc inscriptions, guards and initialization expressions;
- formulate colored and timed Petri nets using the CPN language.

The study load of this chapter is about six hours.

6.2. Introduction

To illustratable the CPN language we start by revisiting an example discussed in previous chapters: The punch card desk. As demonstrated in the previous chapter, the classical Petri net falls short when it comes to describing data and time aspects.

As before we model the punch card desk such that there are two relevant events: *start* (start making a punch card) and *stop* (hand over punch card). Patients are waiting, being served, or done. Employees are either busy or free. For making the punch card it is necessary that the name, address, the date of birth, and the gender of the patient are known. Employees are characterized by an employee number and their experience (expressed in years of experience). The time required to make a punch card depends on the experience of the employee. If (s)he has more than 5 years of experience, the processing time is 3 minutes, otherwise 4.

```
color STR = string;
color INT = int;
color Pat = record Name:STR * Address:STR *
            DateOfBirth:STR * Gender:STR timed;
color Emp = record EmpNo:INT * Experience:INT timed;
color EP = product Pat * Emp timed;
var p:Pat;
var emp:Emp;
val Klaas = {Name="Klaas", Address="Plein 10",
             DateOfBirth="13-Dec-1962", Gender="M"};
val Ann = {EmpNo=641112, Experience=7};
fun d(emp:Emp) = if #Experience(emp) > 5 then 3 else 4;
```

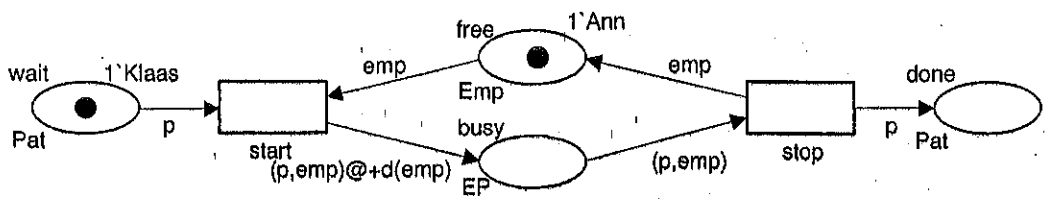


Figure 6.1.: A CPN model of the punch card desk process.

Figure 6.1 shows a CPN model of the punch card desk. It has been modeled as much in the spirit of the previous chapter as possible. Each of the elements in this model will be described in detail in the remainder of this chapter. However, to introduce the main ideas we provide an informal introduction of the main concepts of the CPN language using the example.

The box in Figure 6.1 specifies the *declarations* used in this model. Four *color sets* are introduced in this box: `STR`, `INT`, `Pat`, `Emp`, and `EP`. Each of the color set definitions introduces a new type that can be used in the model. The first two color sets are directly derived from the built-in types `string` and `int`. The other three color sets are more involved. Besides the color set definitions, the declarations box also specifies four *variables* and one *function*. Variable `p` is of color set `Pat` and will be used to describe the flow of patients. Variable `emp` is of color set `Emp` and will be used to describe the flow of employees. The remaining two variables are constants representing respectively a patient and an employee. Function `d` calculates the processing time based on the number of experience of a given employee.

The places in Figure 6.1 are typed by specifying the corresponding color sets, e.g., the place on the right has name `done` and color set `Pat`. The place on the left also has color set `Pat` but in addition a token is present in the initial marking. The color of this token is `Klaas`, i.e., the constant defined in the box. Place `free` also holds a token initially as is indicated by the inscription right of/above the place. This place is of type `Emp`, and the remaining place `busy` is of type `EP` (i.e., the combination of a patient and an employee).

In the previous chapter, we did not specify the behavior of transitions explicitly because a concrete language was lacking. The language presented in this chapter uses *arc inscriptions* to specify this. Figure 6.1 shows six arc inscriptions: one for each arc. The inscription on the left-most arc shows that transition `start` consumes a token with value `p`, where `p` refers to a variable declared in the box. The other input arc of `start` shows that transition `start` also consumes a token with value `emp`, where `emp` refers to another variable. The output arc of `start` is the most interesting one. The first part of this inscription, i.e., `(p,emp)`, shows the value/color of the token produced for place `busy`. The second part, i.e., `+d(e)`, shows that the token will get a delay of `d(e)` time units. Note that the inscription uses the function `d`. The other three arc inscriptions are self explanatory: the token taken from busy is split into two parts: the patient `p` and the employee `emp`. This is done without any delay.

Using a simple example such as the one shown in Figure 6.1 the CPN language is almost self-explanatory. However, for modeling more realistic examples, we need introduce the language in much more detail, starting with the definitions of color sets (types), variables, and arc inscriptions.

6.3. Types and Values

CPN is based on functional language Standard ML. In this section we review the basic types and type constructors offered by Standard ML.

The basic types are: `int`, `real`, `string`, `bool`, `unit`.

- Type `int` is used to represent integers, e.g., `1`, `33`, `~1231`, `234324`. Note that `~1231` represents -1231.
- Type `real` represents the reals. Reals can have a dot and can use the unary minus

(~) and exponential notation (e), e.g., 65.0, 0.2312, ~33.31 (i.e., -33.31), 12e4 (i.e., 120000), and 3e~3 (i.e., 0.003). CPNML, i.e., the functional language used in this chapter, does not allow for reals.[1] Therefore, we will not use any reals in the remainder.

- Type string is used to represent strings. Strings are written between double quotes ("). Examples of quotes are "Ping", "17-01-2003", and "Hello World".
- Type bool (Booleans) has only two values: true and false. This type is used for logical expressions and variables.
- Type unit has only one value: (). This type can be used to define "black" tokens, i.e., tokens that do not carry any information.

For the basic types the following basic operators are defined.

- ~ is the unary minus operator.
- * is the multiplication operator and can be used for integers.
- div is the division operator for integers and mod is the remainder of integer division.
- + is the addition operator and - is the (binary) subtraction operator for integers.
- ^ is the concatenation operator for strings.
- =, >, <, >=, <=, and <> are the basic comparison operators applicable to most types. Each of then has result type bool.
- not is the logical negation.
- andalso is the logical AND.
- orelse is the logical OR.
- if then else is a ternary operator: The first operand should be of type bool and the second and third should be of the same type.

Note that if then else is a ternary operator, i.e., it uses three arguments. The first argument is a Boolean expression to decide whether the "then" part or the "else" part should be taken. The second and third argument should be of the same type. This is the result type of the expression, i.e., the type of result one obtains when applying the if then else operator. The other operators are self-explaining. (Note that the logical AND is named andalso to reflect the fact that the second operand is only evaluated if the first one is true. Similarly, the logical OR is named orelse to reflect that the second operand is only evaluated if the first one is false.)

[1] Note that earlier versions of CPN Tools and Design/CPN did support reals. They are no longer supported because of technical reasons, e.g., it is not possible to test the quality of two reals.

Example 6.1 *Expression* `(5+10) div 5` *is of type* `int` *(i.e., 3). Expression* `(5+10) mod 5` *is of type* `int` *(i.e., 0). Expression* `"Hello" ^ " " ^ "World"` *is of type* `string` *(i.e., "Hello World"). Expression* `(1=1) orelse (2=2)` *is of type* `bool` *(i.e., true). Expression* `if 1=1 then "OK" else "NOK"` *is of type* `string` *(i.e."OK"). Finally, the last expression* `if 2=3 then "YES" else 6+2` *is incorrect because the second and third operand should are not of the same type.*

Exercise 6.1 *Determine the type and result of the followihg expressions:*

1. `5 > 6`
2. `"Hello" ^ " " ^ "World" = "Bye"`
3. `not(true andalso (1=0))`
4. `if "OK"="NOK" then 4 div 2 else 6 div 2`

6.4. Defining Color Sets

Places are typed, i.e., all tokens on a place have a value of some common type. In CPN-terms this means that all tokens in a given place should belong to the same color set. This implies that each place has a color set (i.e., type). Color sets are defined in a straightforward manner. Using the basic types, the color sets `I`, `S`, `B`, and `U` are defined as follows:

```
color I = int;
color S = string;
color B = bool;
color U = unit;
```

Using the basic types one can construct subsets of these types:

```
color Age = int with 0..130;
color Temp = int with ~30..40;
color Alphabet = string with "a".."z";
```

The `Age` color set is a subset of `int`, only the values $0, 1, 2, \ldots 130$ are allowed. The `Temp` color set is the set of integers between -30 and 40. `Alphabet` corresponds to all strings using only characters of the alphabet.

The `with` construct can also be used to introduce color sets not based on one of the standard types. In this case, the values of the type are simply enumerated.

```
color E = with e;
color YN = with Y|N;
color Sex = with Male|Female;
color Beer = with Heineken|Tuborg|Corona|Miller;
```

The color set E only has only one value and can be used like the basic type unit, i.e., to simulate "black tokens". However, the value is e rather than (). The color set YN has two values representing yes and no. The color set Sex also has two values, while the color set Beer has four values. It is important to note that the enumerated values become reserved words. For example, after defining color set E, e can no longer be used in all kinds of expressions.[2]

It is also possible to redefine the basic types bool and unit:

```
color YN = bool with (no,yes);
color BlackToken = unit with null;
```

Color set YN has two possible values: no corresponds to false and yes corresponds to true. The BlackToken color set renames it only value, i.e., (), to null.

More interesting is the possibility to create new types by explicit enumeration:

```
color Human = with man | woman | child;
color ThreeColors = with Green | Red | Yellow;
```

Both color set Human and color set ThreeColors have three possible values.

Finally, there is the possibility to construct new types using the type constructors product, record, and list:

```
color Coordinates = product I * I * I;
color HumanAge = product Human * Age;
color CoordinatesR = record x:I * y:I * z:I;
color CD = record artists:S * title:S * noftracks:I;
color Names = list S;
color ListOfColors = list ThreeColors;
```

Possible colors (i.e., values) of these newly constructed color sets are (colors are separated by commas):

```
Coordinates: (1,2,3), (~4,66,0), ...
HumanAge: (man,50), (child,3), ...
CoordinatesR: {x=1, y=2, z=3}, {x=~4, y=66, z=0}, {y=2, x=1, z=3}, ...
CD: {artists="Havenzangers", title="La La", noftracks=10}, ...
Names: ["John", "Liza", "Paul"], [], ...
ListOfColors = [Green], [Red, Yellow], ...
```

The main difference between products and records is the way the attributes are represented. In a product the position matters while in a record the label matters, i.e., {x=1, y=2, z=3} and {y=2, x=1, z=3} refer to the same value but (1,2,3) and (2,1,3) are different. The type constructors product, record, and list are used to construct more complex types from simpler ones. The product and record constructs are used to structure values with a fixed content.

[2]Note that in CPN Tools, E is a default color set to facilitate the modeling of classical Petri nets, and therefore e cannot be used as a variable unless this color set is explicitly removed.

Although we introduce both records and products, we advocate the use of products for reasons of simplicity. All things expressed in terms of records can also be expressed in products (and vice versa). Therefore, there is no need to learn both. The notation using products is typically more compact. Hence, most of the examples will use products rather than records.

The `list` construct is more dynamic because one list may have an arbitrary number of elements, e.g., `ListOfColors` may have no elements (value `[]`) or many like in `[Green, Red, Yellow, Red, Yellow, Green, Red, Yellow, Red, Yellow, Red, Yellow, Green, Red, Yellow]`.

Example 6.2 *For a Formula 1 race the following color sets may be useful:*

```
color Driver = string;
color Lap = int with 1..80;
color TimeMS = int with 0..100000;
color LapTime = product Lap * TimeMS;
color LapTimes = list LapTime;
color DriverResults = record d:Driver * r:Laptimes;
color Race = List DriverResults;
```

TimeMS is used to express time in milliseconds. A possible color of type Race is:

```
[{d="Jos Verstappen", r=[(1,31000),(2,33400),(3,32800)]},
 {d="Michael Schumacher", r=[(1,32200),(2,31600),(3,30200),(4,29600)]},
 {d="Rubens Barrichello", r=[(1,34500),(2,32600),(3,37200),(4,42600)]}]
```

The type constructors `product`, `record`, and `list` are highly relevant for practical applications. To support the use of lists, products and records, the following operators/constants have been defined:

- `[]` denotes the empty list.
- `^^` concatenates two lists, e.g., `[1,2]^^[3,4]`.
- `::` adds an element to the front of the list, e.g., `"a"::["b","c"]`.
- `#` extracts the field of a record, e.g., `#x({x=1,y=2})`.

The result of `[1,2]^^[3,4]` is the list `[1,2,3,4]`. The result of `"a"::["b","c"]` is the list `["a","b","c"]`. The result of `#x({x=1,y=2})` is the integer `1`.

In CPN it is also possible to define *constants*. Some examples:

```
val jv = "Jos Verstappen" : Driver;
val lap1 = 1 : Lap;
val start = 0 : TimeMS;
```

Example 6.3 *Monaco is a constant of type Race that is constructed from other constants:*

```
val emptyrace = [] : Race;
val jv = "Jos Verstappen" : Driver;
val r1jos =(1,31000) : LapTime;
val r2jos =(2,33400) : LapTime;
val r3jos =(3,32800) : LapTime;
val r123jos = ((1,31000)::[(2,33400)])^^[(3,32800)] : LapTimes;
val jos = {d=jv,r=123jos}: DriverResults;
val michael = {d="Michael Schumacher", r=[(1,32200),(2,31600),
    (3,30200),(4,29600)]}:DriverResults;
val rubens = {d="Rubens Barrichello", r=[(1,34500),(2,32600),(3,37200),
    (4,42600)]}:DriverResults;
val Monaco = jos :: ([michael]^^[rubens]) : Race;
```

Exercise 6.2 *Let Monaco and the other constants be defined as in the example. What is the value of Monaco? Give also the value and type of the following constants:*

```
val e1 = r1jos::[];
val e2 = #d(michael);
val e3 = #r(jos)^^#r(Rubens);
```

6.5. Initial Marking and Multisets

Each place has a type. The type should be a color set defined in the same context. Places should only contain tokens of the corresponding type. At any point in time a place can hold multiple tokens. The same holds for the initial state. Note that CPN uses the term *marking* rather than state. We will use both terms interchangeably. To specify the initial marking we need to be able to express the fact that there are multiple tokens in one place. Moreover, a place can hold multiple tokens having the same value. Therefore, we need to be able to refer to multiple tokens in one expression. For this purpose we need *bags* also known as *multisets*. In a multiset the same element can appear multiple times. However, just like in an ordinary set the order does not matter. In CPN multisets are denoted using the following notation:

```
x1`v1 ++ x2`v2 ++ ... ++ xn`vn
```

where `v1` is a value and `x1` the number of times this element appears in the multiset, etc. Each place has a so-called *initialization expression*. If the initialization expression is not present, the place is empty in the initial marking. If the initialization expression matches the type of the place, the place contains one initial token. If the initialization expression yields a multiset, then the place may contain multiple tokens.

Consider Figure 6.2 illustrating the three possible situations ((a), (b), and (c)). The two places listed above (a) are empty in the initial marking. In situation (b) both places contains one token. In situation (c) each of the two places contains 6 tokens: place p1 holds 6 tokens (one token of value 2 and five of value 4) and place p2 also holds 6 tokens (one token of value "John" and five of value "Sara").

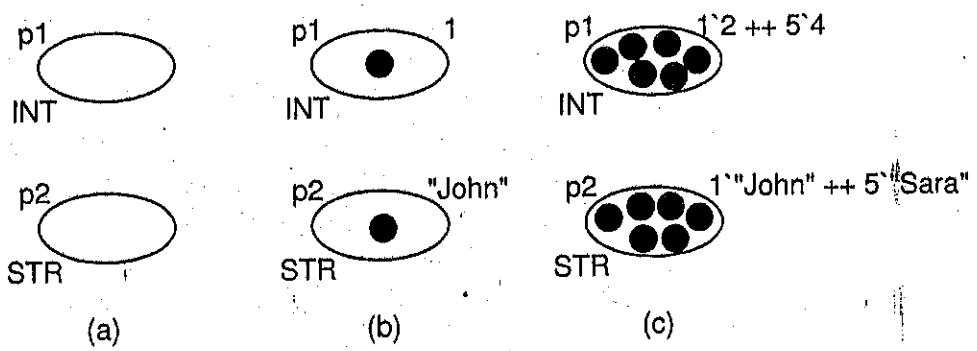
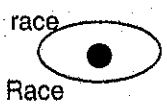


Figure 6.2.: Three types of initialization expressions: (a) empty (no token), (b) value as indicated by the type of the place (single token), and (c) multiset (multiple tokens).

Note that the places in Figure 6.2 are depicted as ovals rather than circles. The reason for this is that this is the typical representation of CPNs or colored Petri nets in general. There is no semantic difference between ovals and circles. However in this chapter we stick as much to the "look and feel" of CPN and tools like Design/CPN and CPN Tools.[3] It is also customary to show the color set definition in the diagram. To illustrate this we also show a place with a more complex type, cf. Figure 6.3.

```
color Driver = string;
color Lap = int with 1..80;
color Time = real with 0.0..1000.0;
color LapTime = product Lap * Time;
color LapTimes = list LapTime;
color DriverResults = record d:Driver * r:LapTimes;
color Race = List DriverResults;
```




```
1`[{d="Jos Verstappen",
r=[(1,31.0),(2,33.4),(3,32.8)]},
{d="Michael Schumacher",
r=[(1,32.2),(2,31.6),(3,30.2),(4,29.6)]],
{d="Rubens Barrichello",
r=[(1,34.5),(2,32.6),(3,37.2),(4,42.6)]]]
```

Figure 6.3.: A typed place with an initialization expression and the declarations of the color sets involved.

[3] Although the notation of tools like Design/CPN and CPN Tools has been adopted as much as possible, there are differences with respect to the positioning of inscriptions and the presentation of the (initial) marking. The differences allow for a clearer representation of the model and the marking.

6.6. Variables, Arc Inscriptions, and Bindings

Firing transitions not only changes the distribution of tokens over the places but also changes the colors (i.e., values) of the tokens flowing through the network. This requires a specification of the transitions involved. In the previous sections the behavior of each transition was described informally. This is inadequate because it allows for all kinds of inconsistencies and ambiguities to remain undetected. To specify the input/output behavior of transitions we use *arc inscriptions*.

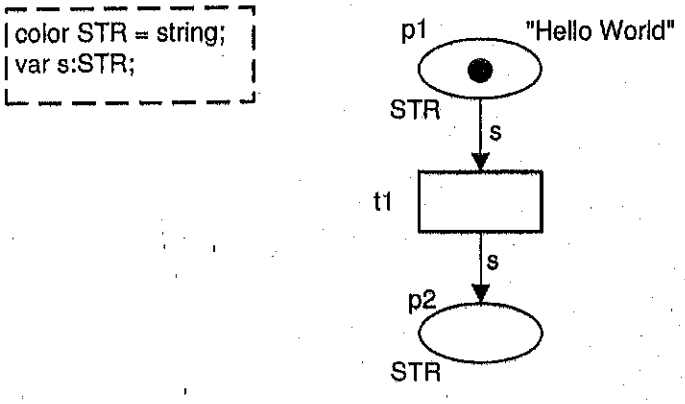


Figure 6.4.: Variable s is used as an arc inscription from p1 to t1 and from t1 to p2.

Figure 6.4 shows an example with very simple arc inscriptions. s is declared as a variable by the declaration var s:STR;. Moreover, s is used in the inscriptions of the arcs from p1 to t1 and from t1 to p2. The moment t1 fires, a token with value s is consumed from p1 and a token with the same value is produced for p2, i.e., transition simply passes a token without changing its value.

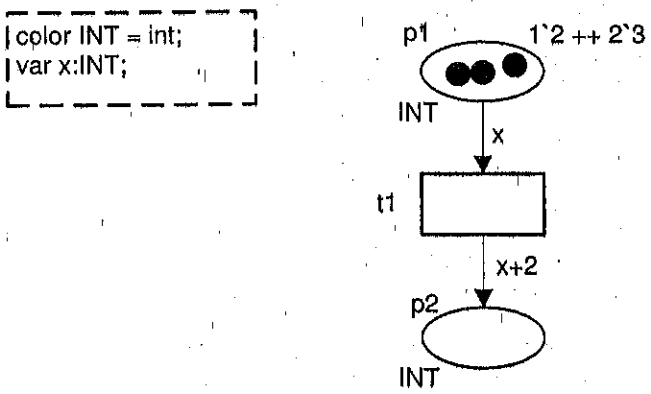


Figure 6.5.: The three tokens are moved from p1 to p2 and the values are incremented by 2.

Figure 6.5 shows another example with a very simple arc inscriptions. x is declared

as a variable by the declaration `var x:INT;`.[4]

When transition `t1` fires it consumes a token with value `x` and produces a token with value `x+2`. Given the initial marking shown in Figure 6.5 transition `t1` will fire three times resulting in three tokens in place `p2` which can be described by the following multiset: `1'4 ++ 2'5`.

In Figure 6.5 there are three tokens in place `p1` therefore it is not sufficient to simply state things like "`t1` fires": `t1` is enabled for three different tokens of which the two tokens with value 3 are undistinguishable. To make this a bit more precise we introduce the concept of binding.

Definition 6.1 *Given a transition t with variables x1, x2, ..., xn on its input and output arcs, a* binding *of t allocates a concrete value to each of these variables. These values should be of the corresponding type. A binding is* enabled *if there are tokens matching the values of the arc inscriptions. If a binding is enabled, it can* occur, *i.e., the transition fires while consuming and producing the corresponding tokens.*

In Figure 6.5 transition `t1` has two enabled bindings: binding `<x=2>` and binding `<x=3>`. The pair `(t1,<x=2>)` is called a binding element. Binding element `(t1,<x=2>)` can occur and results in the state with two tokens in `p1` both having value 3 and one token in `p2` having value 4.

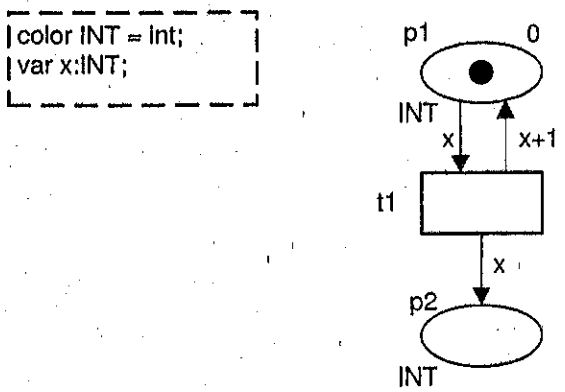


Figure 6.6.: Variable x is used in three arc inscriptions: Binding element `(t1,<x=0>)` is enabled.

Figure 6.6 shows an example which starts in the state with just a token in place `p1` having value 0. Transition `t1` is enabled and will continue to fire. Each time it fires a token is produced for both places. The token returned to `p1` is incremented by one. In the state shown in Figure 6.6 there is one enabled binding element: `(t1,<x=0>)`. The occurrence of this binding element will consume a token with value 0 from `p1` and produce a token for both `p1` (value 1) and `p2` (value 0). In the resulting marking, binding element `(t1,<x=1>)` is enabled, etc.

[4]Note that only "free names" can be used as variables, i.e. the elements enumerated in a `with` statement used to create a new color set may not be declared as a variable. For example, after defining `color E = with e`, `e` cannot be used as a variable name.

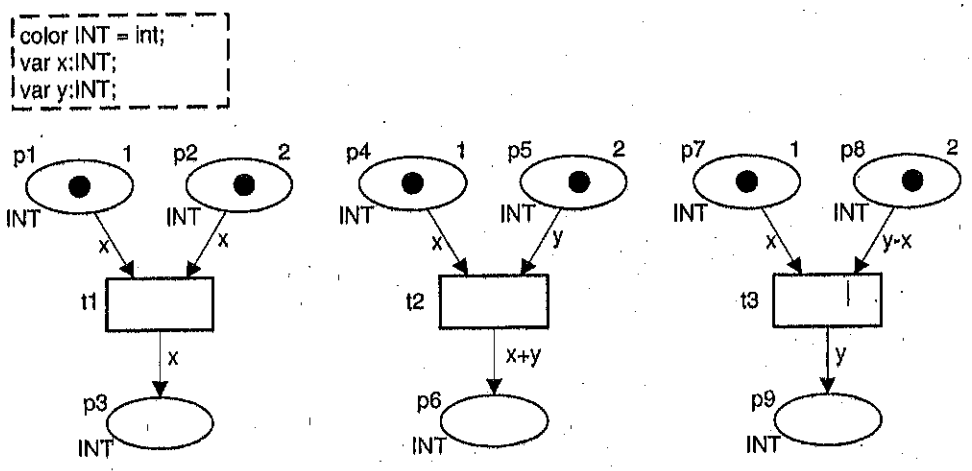


Figure 6.7.: Three examples to illustrate the binding concept (1).

Figure 6.7 shows three transitions to illustrate the binding concept. Transition `t1` is *not* enabled because it is not possible to find an enabled binding. Place `p1` only holds a token with value 1 and therefore `x` should be bound to 1. However, place `p2` only holds a token with value 2 and therefore `x` should be bound to 2 instead of 1. This results in a contradiction showing that there is no enabled binding element. Hence `t1` is not enabled. Transition `t2` in Figure 6.7 is enabled because `x` can be bound to 1 and `y` can be bound to 2. Hence, binding element `(t2,<x=1,y=2>)` is enabled and `t2` can fire while producing a token with value 3 (the output arc evaluates to `x+y`, i.e., 3). Transition `t3` in Figure 6.7 is also enabled. In this case `x` can be bound to 1 and `y` can be bound to 3. The occurrence of binding element `(t3,<x=1,y=3>)` consumes a token with value 1 from place `p7`, consumes a token with value 2 from place `p8` (i.e., `y-x`), and produces a token with value 3 for place `p9`. Note that the behavior of transition `t2` is identical to transition `t3` although the arc inscriptions are different.

Figure 6.8 also shows three transitions. Transition `t1` is enabled. In fact there are two binding elements `(t1,<x=5>)` and `(t1,<x=7>)`. In both cases two tokens are produced: one for `p2` and one for `p3`. The first binding element `(t1,<x=5>)` can occur two times and the second binding element `(t1,<x=7>)` can occur three times. The order of these occurrences is undetermined, i.e., there is a non-deterministic choice. In fact one can think of the five bindings being enabled concurrently. However, for simplicity and because of the so-called diamond rule[5], we assume an interleaving semantics, i.e., the occurrences are ordered. Transition `t2` in Figure 6.8 is enabled because `x` can be bound to 5 or to 7, i.e., again there are two bindings: `(t2,<x=5>)` and `(t2,<x=7>)`. However, unlike `t1` only one token will be produced for either `p5` or `p6`. The two if-then-

[5]The diamond rule states that if two binding elements are enabled concurrently (i.e., the occurrence of one one element does not disable the other or depend on it and vice versa) the result of their concurrent occurrence is identical to their "interleaved" occurrence, i.e., the order in which they fire does not matter. Because of this property it often does not make sense to consider concurrent occurrences (they do not yield additional states).

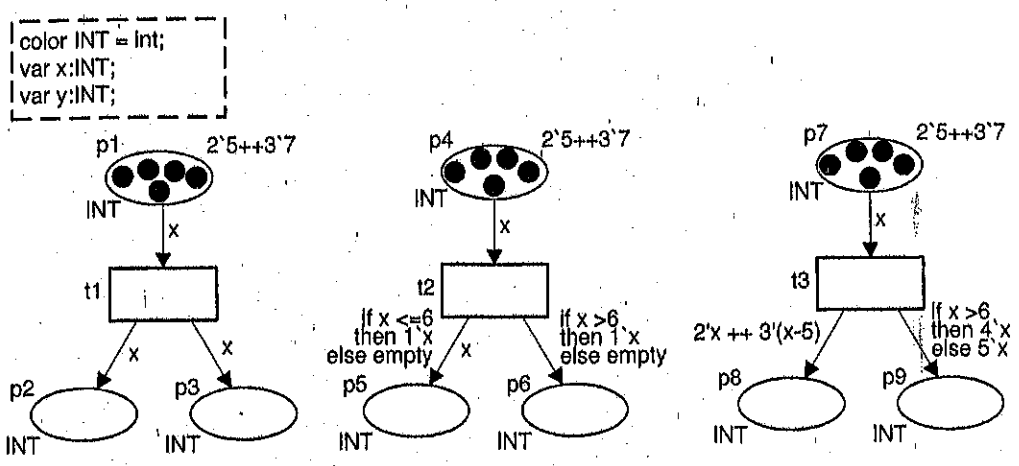


Figure 6.8.: Three examples to illustrate the binding concept (2).

else constructs inspect the value of x and based on this value evaluate to either 1`x or empty. empty is the empty multiset and 1`x is the multiset containing one value x. Note that when using the if-then-else in an arc inscription, the "then" part and the "else" part should be of the same type. Also note that the overall construct should evaluate to a multiset. In fact, every arc inscription should evaluate to a multiset. However, just like in initialization expressions a single value (e.g, (x)) is mapped onto a multiset with just one value (1`x). The fact that arc inscriptions evaluate to multisets is illustrated by transition t3 in Figure 6.8. Again there are two bindings: (t3,<x=5>) and (t3,<x=7>). When binding (t3,<x=5>) occurs one token with value 5 is consumed from p7, five tokens are produced for p8 (two with value 5 and three with value 0), and five tokens are produced for p9 (all with value 5). When binding (t3,<x=7>) occurs one token with value 7 is consumed from p7, five tokens are produced for p8 (two with value 7 and three with value 2), and four tokens are produced for p9 (all with value 7).

Clearly, the examples given in this section are rather abstract and do not correspond to practical situations. They are given to show the syntax and semantics of CPN.

Finally, Figure 6.9 shows three more transitions to illustrate concepts like binding, variable and occurrence when strings, lists, and records are involved. Transition t1 in Figure 6.9 is enabled because both x and y can be bound, i.e., binding element (t1,<x="Hello ",y=" World">) is enabled. Place p3 is of type STR and place p4 is of type S. The occurrence of the enabled binding will result in two tokens: one having value "Hello World" and the other having value ["Hello "," World"]. Transition t2 in Figure 6.9 has two enabled bindings: (t2,<x="Hi",z=0,s=[]>) and (t2,<x="Ho",z=0,s=[]>). Given the initial marking shown in Figure 6.9, t2 will fire five times. In the resulting marking, place p6 holds a token with value 5 and place p7 holds a token representing a list of five strings. The order of the three "Hi" and two "Ho" strings depends on the order in which the tokens are consumed from p5. Transition t3 in Figure 6.9 splits tokens in p8, i.e., tokens in p8 have a record value and the a field is produced for p9 and the b field is produced for p10. The expression #a(r) produces

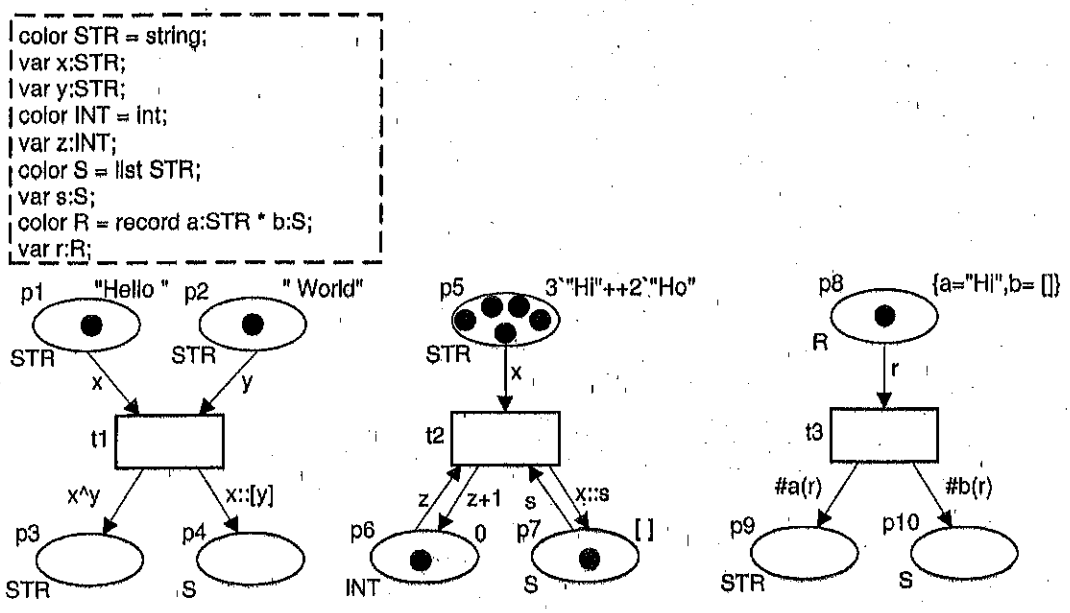


Figure 6.9.: Three examples to illustrate the binding concept (3).

the a field of `r` and `#b(r)` produces the `b` field of `r`.

Example 6.4 *Consider three of the political parties in The Netherlands: CDA, PVDA, and VVD. People can vote for these parties. Figure 6.10 shows a simple CPN model consisting for just two places and one transition modeling this. Tokens in place* `vote` *represent votes. For each party there is also a token in place* `votes`*. Place* `votes` *has a product type to keep track of the votes for each party. For each vote the counter of the corresponding party is incremented by 1.*

Exercise 6.3 *Consider a simple banking system. There are 1000 accounts numbered from 1 to 1000. People can deposit or withdraw money. Only amounts between 1 EURO and 5000 EURO can be deposited or withdrawn. The account may have a negative balance. Model this in terms of a CPN model.*

Exercise 6.4 *Consider a database system where authors can submit articles. The articles are stored in such a way that it is possible to get a sequential list of articles per author. The list is ordered in such a way that the oldest articles appear first. Note that the system should support two actions: submit articles (with name of author and article) and get articles. We assume that each article has a single author and that only authors already registered in the database can submit articles. Model this in terms of a CPN model.*

Exercise 6.5 *Extend the CPN model such that each article can have multiple authors, i.e., the article is stored once for each author, and that there is an explicit action to add authors to the database.*

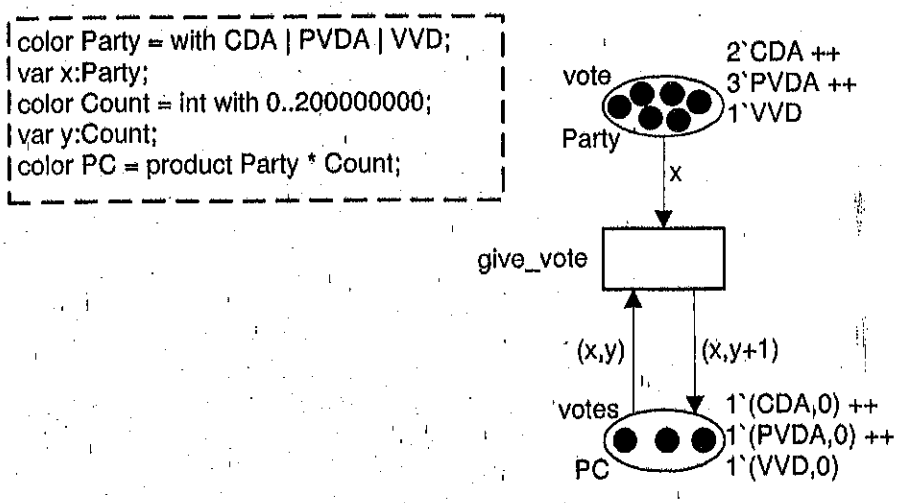


Figure 6.10.: A simple CPN model of the voting process.

6.7. Guards and Functions

In this section we introduce two new concepts: *guards* and *functions*. First we define the concept of guards.

Definition 6.2 *A transition can have a guard. A guard is a Boolean expression (i.e., an expression which evaluates to either true of false) and it may have variables in exactly the same way that arc inscriptions have.*

The purpose of a guard is to *block* a transition when it should not fire for some reason, i.e., the guard defines an additional constraint which must be fulfilled before the transition is enabled. More precisely: a binding element is only enabled if its corresponding guard evaluates to false. Another term for guard is *precondition*. One can think of each transition we have seen so far as a transition with guard `true`. By convention we omit guards which always evaluate to true (in a similar way we omit initialization expressions which evaluate to the empty multiset). For clarity we will enclose guards by square brackets.

Figure 6.11 shows three simple guards. Note that as a result of the guards both `t1` and `t1` cannot fire, i.e., given the initial marking there is no binding for which the guards evaluate to true. Transition `t3` is enabled in Figure 6.11. Note that the value of the token produced (i.e., `z`) is restricted by the guard, i.e., the value of the produced token is 3 (`[z=x+y]`).

Figure 6.11 also shows that variables of the same time do not need to declared separately, i.e., `x`, `y`, and `z` are of type `INT` and are declared by one expression: `var x,y,z:INT`.

Guards can also be used to model situations where if-the-else constructs are not suitable. Figure 6.12 shows two situations where guards are useful. The first situation

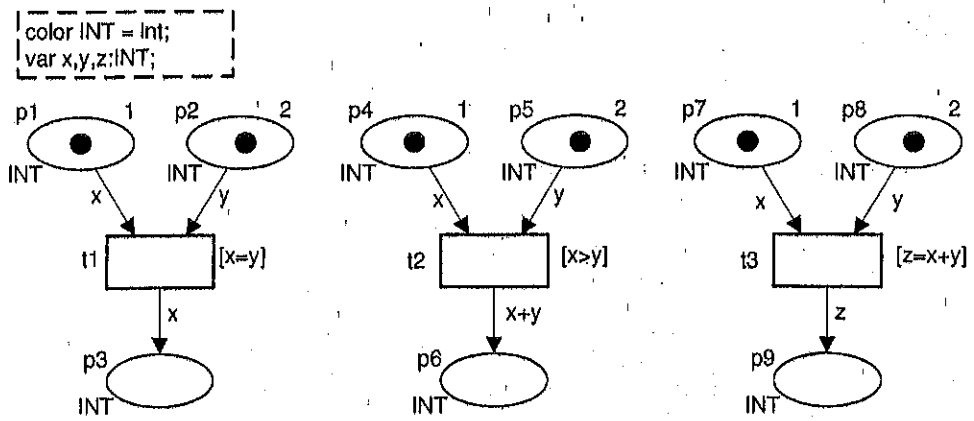


Figure 6.11.: Three examples using guards.

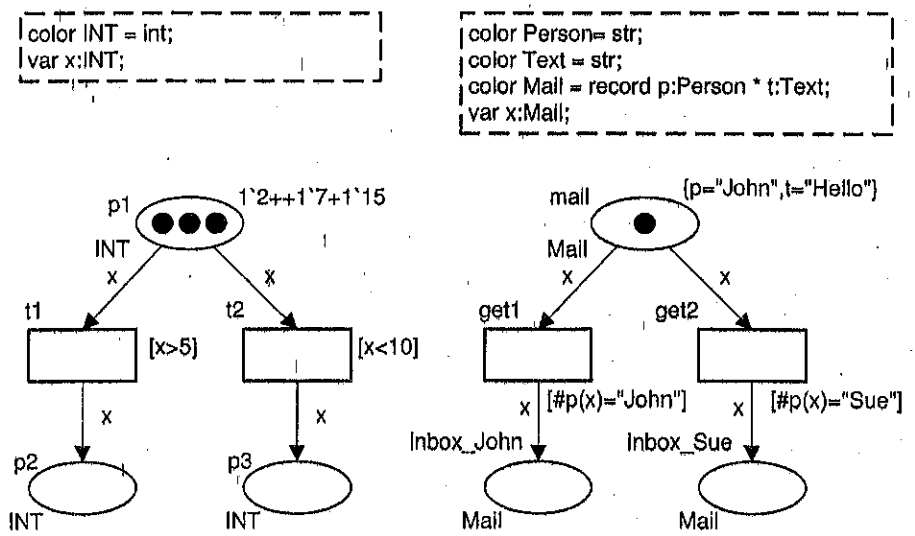


Figure 6.12.: Two additional examples using guards.

(left-hand side of Figure 6.12) shows the situation where a choice is not deterministic but also not completely non-deterministic. Tokens with value 6, 7, 8 or 9 can be consumed by both transitions (i.e., a non-deterministic choice). Tokens with a value of 5 or less are consumed by transition `t2` and tokens with a value of 10 or more are consumed by transition `t1`. Note that there are four enabled binding elements for the initial marking shown in Figure 6.12 (left-hand side only): `(t2,<x=2>)`, `(t1,<x=7>)`, `(t2,<x=7>)`, and `(t1,<x=15>)`. If the two guards are omitted, two additional binding elements come into play: `(t1,<x=2>)` and `(t2,<x=15>)`. Note that a mixed deterministic/non-deterministic choice is not possible without guards. Another situation where guards come in handy is shown in the right-hand side of Figure 6.12. Based on the guards each mail is distributed to the right place. Note that it is possible to add new mail recipients without changing the logic or any of the expressions: Simply add another output transition similar to the one for John or Sue. Note that in this case there is a deterministic choice. Nevertheless, the notation is preferable since it is possible to add additional recipients more easily than through a series of if-the-else constructs.

Exercise 6.6 *Consider again the simple banking system introduced earlier. Thus far we assumed that an account could have a negative balance. Change the model such that the balance cannot become negative, i.e., do not accept transactions which lead to a negative balance. Model this in terms of a CPN model.*

For the simple examples we have seen so far all the logic can be put into guards and arc inscriptions. For more complex problems this is not longer possible. Consider for example the banking system with the requirement that a withdrawal is only possible if the total amount of money in the bank is at least 10000 EURO. In such a situation it is often convenient to use *functions*.

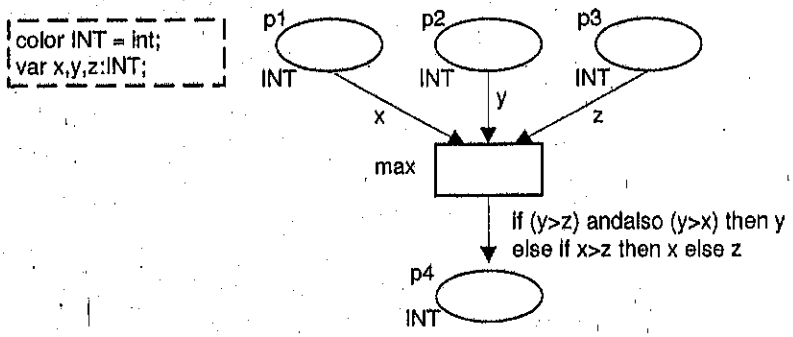


Figure 6.13.: An example not using functions.

To illustrate the role of functions consider Figure 6.13. Transition `max` takes the maximum of three tokens, i.e., the value of the token produced for place `p4` is the maximum of the values of the three tokens consumed. To do this the arc inscription on the output arc has a nested if-then-else statement:

```
if (y>z) andalso (y>x) then y else if x>z then x else z
```

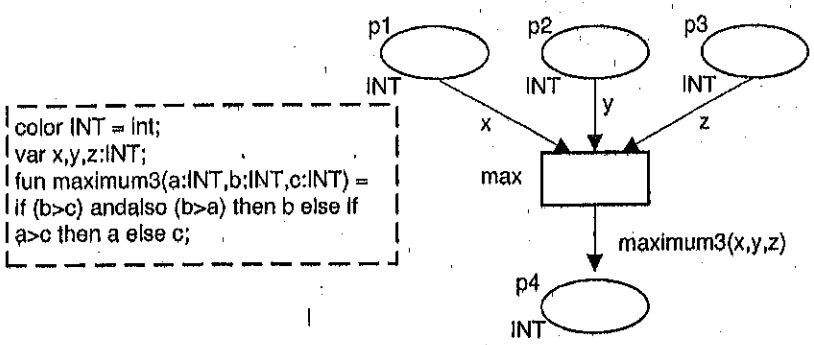


Figure 6.14.: The same example but now using the dedicated function `maximum3`.

To allow for more compact arc inscriptions, this expression can be replaced by a function call as shown in Figure 6.14. The function `maximum3` is defined as follows:

```
fun maximum3(a:INT,b:INT,c:INT) =
if (b>c) andalso (b>a) then b else if a>c then a else c;
```

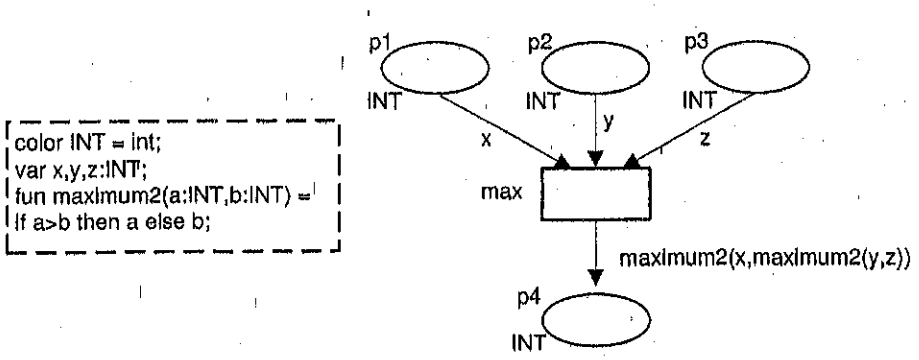


Figure 6.15.: Taking the maximum of three integers can be decomposed into two calls of the function `maximum2`.

Observing this definition, we see the same two if-then-else statements. These are needed because `maximum3` takes the maximum of three integers rather than two. However, it is also possible to decompose function `maximum3` into two calls of the function `maximum2` as shown in Figure 6.15. Note that `maximum2` is applied to `x` and the maximum of `y` and `z`. The latter value is obtained by applying `maximum2` to `y` and `z`.

Functions can be used to make arbitrary complex calculations. There are *standard functions* such as `hd` and `tl`. `hd` returns the head of a (non-empty) list. `tl` returns the tail of a (non-empty) list, (i.e., `hd(x)::tl(x)=x`). Besides the standard functions it is possible to define new functions, cf. `maximum2` and `maximum3`. Some additional examples are given below:

```
color INT = int;
```

```
fun fac(x:INT) = if x>1 then x*fac(x-1) else 1;
fun fib(x:INT) = if x<2 then 1 else fib(x-1) + fib(x-2);
color L = list INT;
fun odd(x:L) = if x=[] then [] else hd(x)::even(tl(x));
fun even(x:L) = if x=[] then [] else odd(tl(x));
```

Function `fac` calculates the factorial of a given integer. Function `fib` calculates Fibonaci numbers. Function odd results the even elements of a list. Note that function odd calls function even. In this section, we do not describe how functions should be defined. At this point in time it is sufficient to know that they can be used to do the more complex calculations. Functions can be used in guards, arc inscriptions, and initialization expressions. Later in this chapter, we show how to define more complex functions.

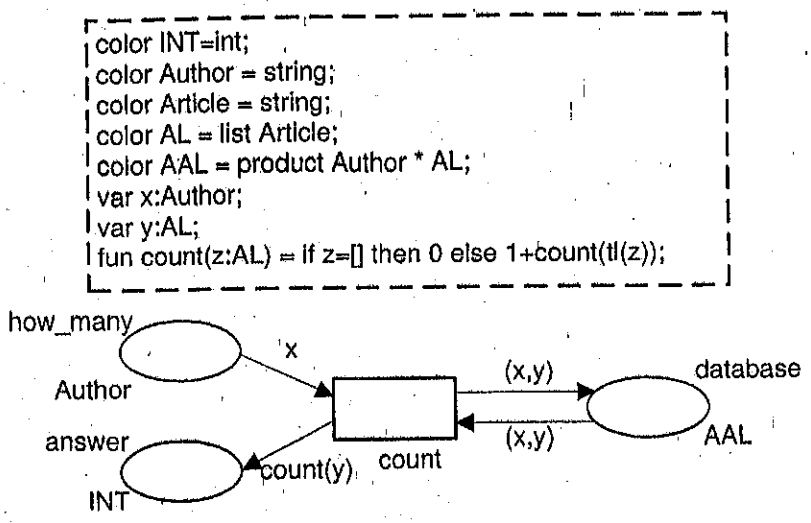


Figure 6.16.: An example using functions.

Figure 6.16 shows an example which uses a function to count the number of articles for a given author. For any token put in place how many the database is inspected whether the author appears there. If the author exists arc inscription `count(y)` is evaluated. As a result function `count` is used to determine the length of the list bound to `y`.

Exercise 6.7 *Calculate `fac(fib(3))` and modify odd and even such that the odd lines are returned in reversed order.*

To conclude this section, we answer a number of questions that may have popped up while introducing concepts like variables, arc inscriptions, and guards:

- Is it possible to have multiple arcs connecting a place and a transitions?
- Is it possible to have multisets as arc inscriptions on input arcs?
- Is it possible to use constants or other expressions without variables as arc inscriptions?

- Is it possible to use records, lists, etc. with variables (e.g., `{a=x,b=y}` and `{a=x::y}`) in arc inscriptions?

The answer to each of these questions is yes. The only requirement is that expressions in guards and output arcs can be bound to a concrete value, i.e., typically it is not allowed to have a variable on an output arc or guard which does not appear on one of the input arcs. The reason for this restriction is that otherwise it is not possible to bind this variable given a specific marking because the number of possible bindings is unrestricted. Figure 6.17 shows examples that demonstrate that the above questions can be answered positively. Moreover, it also shows two situations which are not allowed. Note that in both cases variable `z` is unbound.

6.8. Time

To investigate the performance of processes and systems or to simply model temporal aspects, CPN allows for timed tokens as introduced in the previous chapter. In addition to the token color, CPN allows each token to have a time stamp attached to it. This time stamp indicates the earliest time at which a token can be removed. A color set can be made a *timed* color set by adding the term `timed`, e.g., declaration `color TimedINT = int timed;` defines a timed color set for integers. Tokens in a place having a timed color set have a time stamp. All other tokens have no time stamp and are therefore always available for consumption. Delays can be put in the arc inscription of an *outgoing* arc, e.g., `x@+44` denotes that a token with value `x` is produced with a delay of 44 time units. Note that `x@+44` is *relative* to the firing time, i.e., if the corresponding transition fired at time 22 the resulting time stamp of the produced token is 66. Timed tokens have *absolute* time stamps denoted `@` followed by the value of the time stamp.

Figure 6.18 shows two transitions using the timed color set `STR` and the untimed color set `INT`. Transition `t1` produces a token for place `p3` with a delay of 1 time unit. The delay of the other token produced for `p3` depends on the value of `y` (in this case 4). From the marking shown, the token for `p3` will have a time stamp of 1 while the token for `p2` will have a time stamp of 4. In the other example shown in Figure 6.18 the delays of subsequent tokens produced for place `p7` increase as the value of the token in place `p6` increases.

Exercise 6.8 *Calculate the time stamps of the tokens in p7 in the final marking. Modify the left-hand side of Figure 6.18 such that for positive values of y the delay is 10 while for other values the delay is 20. Modify the right-hand side of Figure 6.18 such that the delay for "Hi" is 10 while for other values the delay is y.*

In Section 6.10.3 we will discuss special functions related to time. For example, the function `time()` can be used to obtain the "current time" in the model. Section 8.5 will discuss the use of CPN models for simulation purposes.

This completes our introduction to the CPN language. Note that a CPN model consists of two parts: declarations and the net structure. Figure 6.19 summarizes the main ingredients of a CPN model.

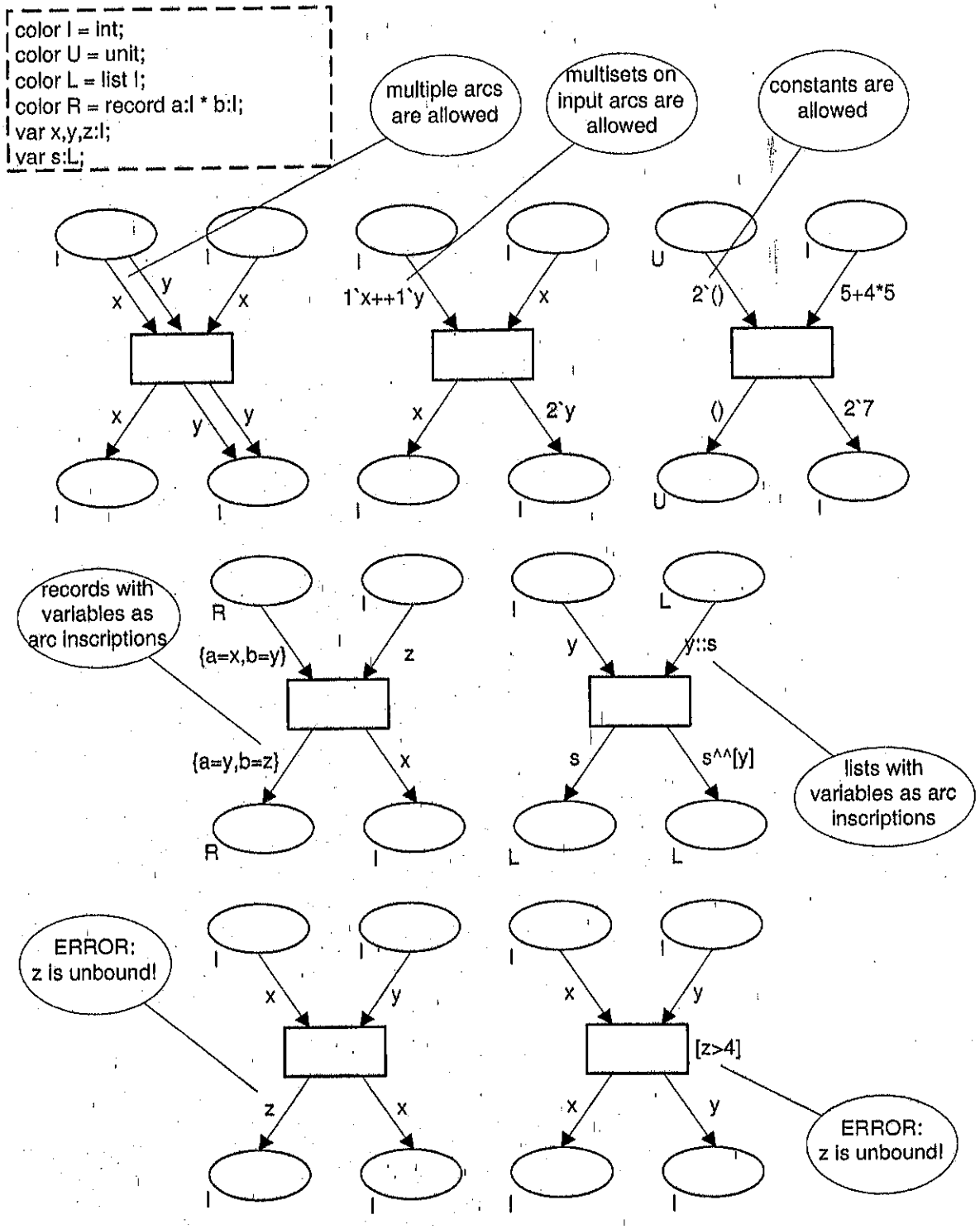


Figure 6.17.: Constructs that are allowed and constructs that are not allowed.

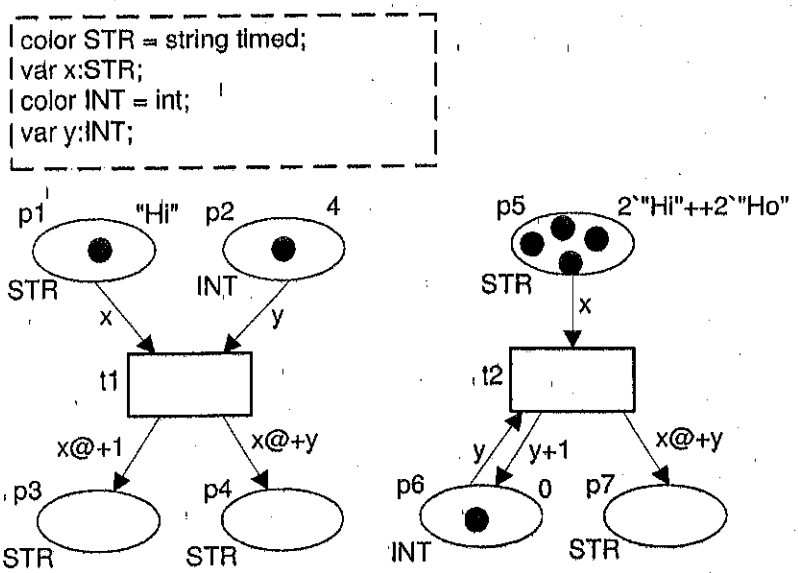


Figure 6.18.: Two timed CPN models.

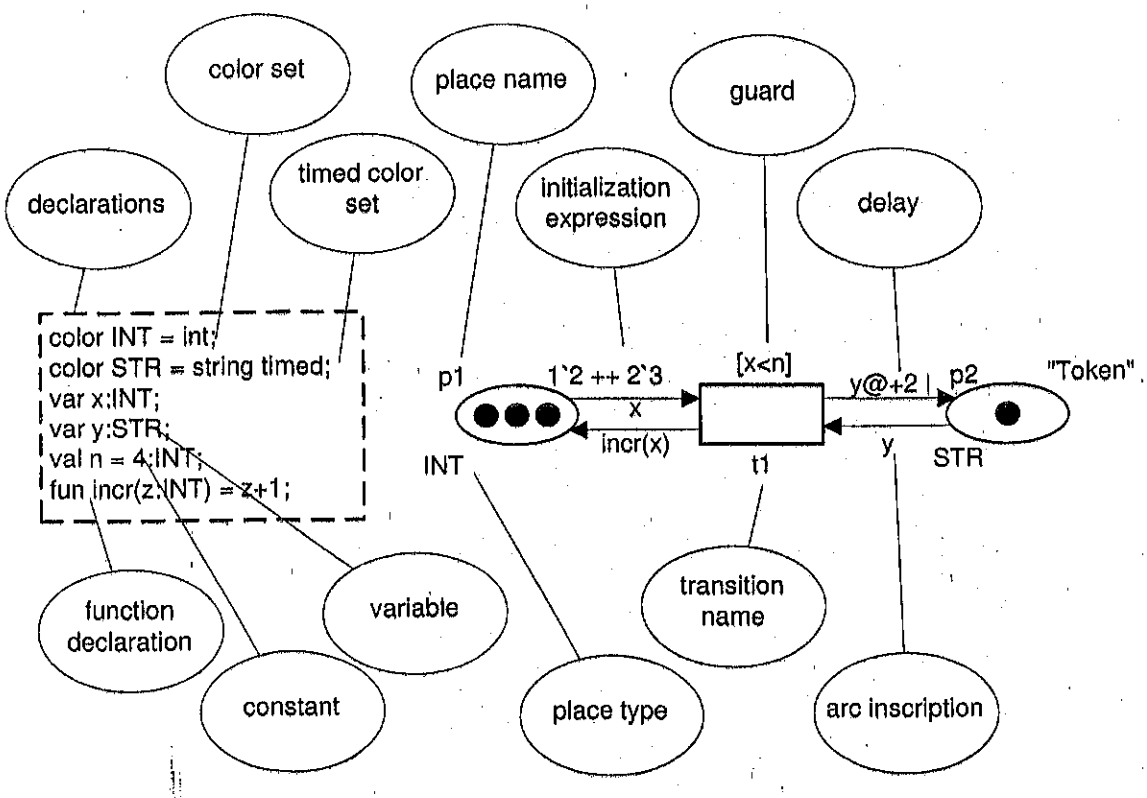


Figure 6.19.: The elements of a CPN model.

6.9. Examples Revisited

In the previous chapter we used a rather informal notation to model Petri nets with time and/or color. In this section, we return two examples (punch card desk, and keeping stocks).

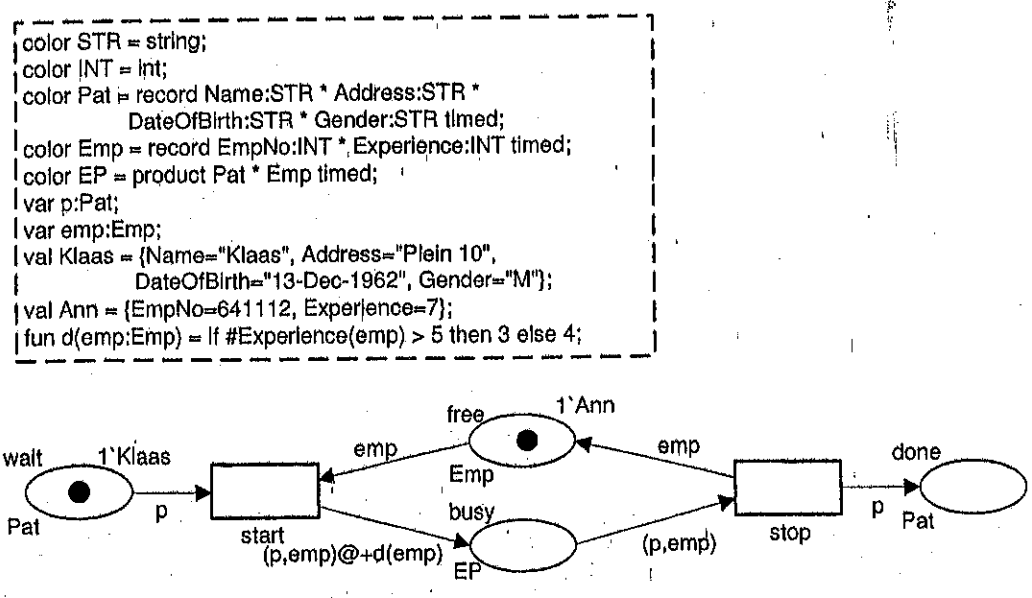


Figure 6.20.: A CPN model of the punch card desk process.

First we consider the punch card desk example. When we introduced the CPN language we already showed a CPN model (cf. Section 6.1). This model is again shown in Figure 6.20. As indicated before, it has been modeled as much in the spirit of the previous process as possible. Note that the two constants (`Klaas` and `Ann`) and the function `d` (to calculate the delay) have only been introduced to simplify the diagram. `Klaas`, `Ann`, and `d` can be simply replaced by their definitions. It is also possible to improve the model. In the previous chapter, we choose simple types like `string` and `int`. Using the types and type constructors presented in this chapter, we can change the declarations in Figure 6.20 into:

```
color Name = string;
color Street = string;
color Number = int;
color Town = string;
color Address = record s:Street * n:Number * t:Town;
color Day = int with 1..31;
color Month = with Jan | Feb | Mar | Apr | May | Jun |
                   Jul | Aug | Sep | Oct | Nov | Dec;
color Year = int with 0..2100;
color Date = record d:Day * m:Month * y:Year;
```

```
color Gender = with male | female;
color EmpNo = int with 100000..999999;
color Pat = record name:Name * address:Address *
                   birthdate:Date * gender:Gender timed;
color Emp = record empno:EmpNo * experience:Year timed;
color EP = product Pat * Emp timed;
var p:Pat;
var e:Emp;
val Klaas = {name="Klaas",
             address={s="Plein",n=10,t="Unknown"},
             birthdate={d=13,m=Dec,y=1962},
             gender=male}:Pat;
val Ann = {empno=641112, experience=7}:Emp;
fun d(x:Emp) = if #experience(x) > 5 then 3 else 4;
```

Note that these declarations give more structure to the token colors. For example it is not possible to omit the month in a date or use a negative employee number (in fact the number always has 6 digits).

Exercise 6.9 *How to change the model shown in Figure 6.20 such that female patients cannot be served by employees with less than 5 years of experience? How to model that the employee takes a rest after each customer? If the employee served a male patient, it takes 5 minutes to recover, otherwise only 3 minutes. (Note that these extensions do not make any sense. They only serve as practice.)*

Second, we model the database system to keep track of the products in stock (i.e., bikes, wheels, frames, etc.) presented informally in the previous chapter. Recall that there are two actions: increase stock of a specific product and decrease stock of a specific product. In the previous chapter, all information about the products in stock was put into a single token. Figure 6.21 models the database system in the spirit of the approach used in the previous chapter.

The network structure of the CPN model shown in Figure 6.21 is very simple. The arc inscriptions are rather involved but their complexity is hidden in functions. We need three recursive functions: one to increase the stock (`incrs`), one to decrease the stock (`decrs`), and one to check whether the stock does not become negative (`check`). In this chapter we do not focus on function declarations. Therefore, we do not describe these functions in detail. (Recall that `hd` returns the head of a (non-empty) list and `tl` returns the tail of a (non-empty) list.) The intended effect of these functions speaks for itself. A design choice in the model of Figure 6.21 is to put all information in a single token: the token in `stock`. The advantage is that it is easy to address the stock as a whole. For example the function `totalstock` can be used to count the total number of items:

```
fun totalstock(s:Stock) =
if s=[]
then 0
else (#number(hd(s)))+totalstock(tl(s));
```

```
color Product = string;
color Number = int;
color StockItem = record prod:Product * number:Number;
color Stock = list StockItem;
var x:StockItem;
var s:Stock;
fun incrs(x:StockItem,s:Stock) = if s=[] then [x] else (if (#prod(hd(s)))=(#prod(x))
  then {prod=(#prod(hd(s))),number=((#number(hd(s)))+(#number(x))))}::tl(s)
  else hd(s)::incrs(x,tl(s)));
fun decrs(x:StockItem,s:Stock)= incrs({prod=(#prod(x)),number=(~(#number(x))))},s);
fun check(s:Stock)= if s=[] then true else if (#number(hd(s)))<0 then false
  else check(tl(s));
val initstock = [{prod="bike", number=4},{prod="wheel", number=2},
  {prod="bell", number=3}, {prod="steering wheel", number=3},
  {prod="frame", number=2}];
```

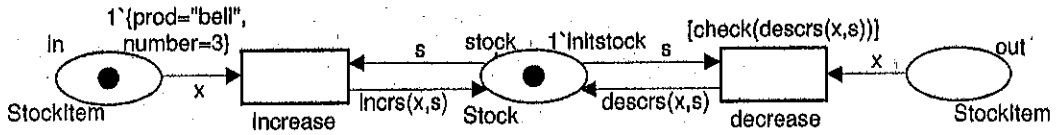


Figure 6.21.: A CPN model of the stock-keeping system.

The drawback is that for any modification of the stock (even the ones referring to only one product) we need to add a recursive function. In this particular case it is possible to simplify the model by using multiple tokens in the place `stock` and using a product type rather than a record type for color set `StockItem`. Figure 6.22 shows the result.

```
color Product = string;
color Number = int;
color StockItem = product Product*Number;
var p:Product;
var x,y:Number;
```

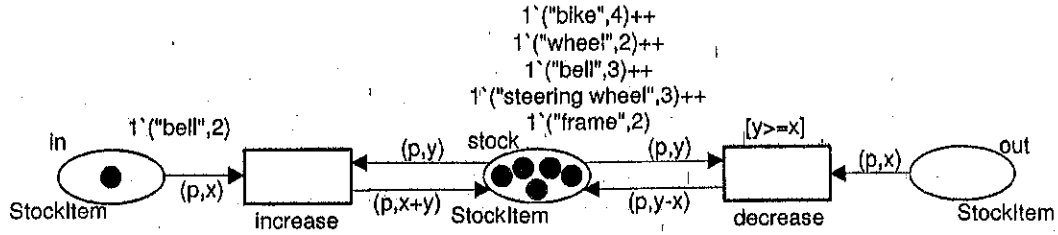


Figure 6.22.: A simplified CPN model of the stock-keeping system.

The CPN model in Figure 6.22 is indeed less complicated and illustrates that selecting the right color sets and choosing the right level of detail (e.g., Does a token represent one stock level or the multiple stock levels?) are important design decisions. The only way to learn to make the right design decisions is through experience and trail-and-error. Therefore, it is important to try and model as many systems as possible.

Exercise 6.10 *Extend the model shown in Figure 6.22 such that the total number of products is also recorded. Moreover, in the warehouse individual products can get lost or damaged. Assume that such accidental reductions of the stock are known and registered directly.*

6.10. More on functions

For real-life process models, not only the network structure gets complex, also complicated functions are needed. In this section, we focus on the definition of functions. As indicated before, after beging defined, functions can be used in arc inscriptions, guards, and initialization expressions. At the end of this section, we also discuss some of the probabilistic functions (i.e., random distribution functions) provided by CPN Tools. These functions are often used in combination with time, e.g., in simulation models.

6.10.1. Recursive functions

Recursion is the construct where a functions calls itself. This way some form of iteration is created. To illustrate the concept of recursion we consider the factorial of a given integer n:

$$n! = \prod_{i=1}^{n} i$$

For example $10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 3628800$. Let us define a recursive function fac to calculate this. Recursion always has (1) a base of the induction and (2) the induction step. The base of the induction is some trivial value for which we know the answer, e.g., for function fac we know that $fac(1) = 1$. The general induction step describes the relation between results of the function of "higher" parameter values and results of the function of "lower" parameter values. For function fac there is a clear relation between two subsequent parameter values i. For parameter values of at least 2, i.e., $i \geq 2$, $fac(i) = i * fac(i-1)$. Note that the result for the "higher" parameter value i is expressed in terms of the result for the "lower" parameter value $i-1$. By repeatedly applying the general induction step until the base induction is reached, the factorial of any positive integer can be derived. For example: $fac(10) = 10 * fac(9)$, $fac(9) = 9 * fac(8)$, ..., $fac(2) = 2 * fac(1)$, and finally $fac(1) = 1$ (base step). In the CPN language we use an if-the-else statement to distinguish between the general induction step and the base induction step:

```
color INT = int;
fun fac(i:INT) = if i>1 then i*fac(i-1) else 1;
```

Note that this function was already given before. Another recursive function already shown before is the function `totalstock`:

```
color Product = string;
color Number = int;
```

```
color StockItem = record prod:Product * number:Number;
color Stock = list StockItem;
fun totalstock(s:Stock) =
  if s=[]
  then 0
  else (#number(hd(s)))+totalstock(tl(s));
```

This function works on parameters of type Stock, i.e., a list of stock items each represented by a product attribute and a number attribute. Function totalstock takes the sum over all products. The base induction step is the situation that the list is empty, i.e., there are no products left in the list and therefore totalstock applied to the empty list results in 0. In the general induction step the head of the list is taken ($hd(s)$) and the number of products of this stock item is added to totalstock applied to the tail of the list ($tl(s)$).

Question 6.1 *How to construct the function maxstock which calculates the maximal stock of a single product?*

Again the base step is the empty list and the general step is the non-empty list. If the list is empty, the function should return 0. If not, the number of products of the first product in the list should be compared with the function maxstock applied to the tail of the list. Through recursion the result can be obtained. In terms of the CPN language:

```
fun maxstock(s:Stock) =
  if s=[]
  then 0
  else if (#number(hd(s))) >= maxstock(tl(s)) then #number(hd(s))
                                              else maxstock(tl(s));
```

To obtain the product that has a maximum number, function maxstockname can be defined as follows:

```
fun maxstockname(s:Stock) =
  if s=[]
  then "no product found"
  else if (#number(hd(s)))=maxstock(tl(s)) then #prod(hd(s))
                                           else maxstockname(tl(s));
```

Note that the function uses function maxstock. Also note that the base step returns the string "no product found". This string is only returned if the list is actually empty, i.e., there are no products in stock. If there are multiple products having the same quantity, the first product in the list is selected.

Functions can also have two or more arguments. As an example consider the function enoughstock:

```
fun enoughstock(s:Stock,n:Number) =
  if s=[]
  then []
  else if (#number(hd(s)))>= n then hd(s)::enoughstock(tl(s),n)
                               else enoughstock(tl(s),n);
```

This function results a list with just the products where there are at least a given number of items, e.g., `enoughstock(s,8)` returns a list of stock items with at least 8 items in stock (i.e., selected parts of list `s`).

Question 6.2 *How to construct the function `enoughstockn(s,n)` which calculates the number of products having at least `n` items in stock?*

Given the above the answer to this question is not difficult:

```
fun enoughstockn(s:Stock,n:Number) =
  if s=[]
  then 0
  else if (#number(hd(s)))>= n then 1+enoughstockn(tl(s),n)
                               else enoughstockn(tl(s),n);
```

An alternative answer to the question would be to use the function `enoughstock` and a function to calculate the length of a list. In ML there is a built-in function `length` to calculate the length of any list. Using these two functions an alternative definition for function `enoughstock` is:

```
fun enoughstockn(s:Stock,n:Number) = length(enoughstock(s,n));
```

6.10.2. Pattern matching

CPN also supports the concept of *pattern matching*. This concept does not extend the expressive power of the language but allows for a more compact notation. For example, instead of using the built-in `hd` (head) and `tl` (tail) functions and explicit if-the-else statements, the notation `[]` could be used for the empty list and `x::y` for lists containing at least one element where `x` is the head and `y` is the tail. Consider for example the following function definition without pattern matching:

```
fun lenlist1(s:Stock) =
  if s=[]
  then 0
  else 1+lenlist(tl(s));
```

This function calculates the number of different products. This corresponds to simply calculating the length of list `s`. This function could also be re-written using pattern matching:

```
fun lenlist2([]) = 0 |
 lenlist2(si::s) = 1+lenlist2(s);
```

Note that now there are two cases/patterns. The first one considers the empty list. The second one considers a list containing at least one element as indicated by the pattern `si::s`. If we compare the two definitions (`lenlist1` and `lenlist2`), there are several differences. First, the if-then-else statement is replaced by a horizontal bar. Second, there is no need to use the `tl` (tail) function because through the pattern `si::s` the term `s` suffices. Finally, function `lenlist1` is explicitly defined for lists of type `Stock` while function `lenlist2` can de applied to any list. Through the pattern matching mechanism it is clear that `lenlist2` only accepts lists. In the definition of `lenlist2` the type of elements in that list is not important. Therefore, the definition applies to any list. This way CPN allows for *polymorphism*, i.e., one function is defined independent of the exact type (in this case any list).

Pattern matching allows for a more compact definition of functions. First we redefine function `totalstock`.

```
fun totalstock([]:Stock) = 0 |
  totalstock(si::s) = (#number(si))+totalstock(s);
```

Another example using pattern matching is an alternative definition of function `maxstock`.

```
fun maxstock([]:Stock) = [] |
 maxstock(si::s) = if (#number(si))>maxstock(s) then #number(si)
                                                else maxstock(s);
```

Similarly the other functions already given can be translated. As an example consider the function `incrs` used in the CPN model shown in Figure 6.21. This function can also be defined as follows:

```
fun incrs(x:StockItem,[]:Stock) = [x] |
 incrs (x,(si::s)) =
  if  (#prod(si))=(#prod(x))
  then {prod=(#prod(si)),number=((#number(si))+(#number(x)))}::incrs(x,s)
  else (si::incrs(x,s));
```

To conclude this section on functions we show some example definitions of polymorphic functions.

```
fun reverse([]) = [] |
 reverse(x::y) = reverse(y)^^[x];
fun odd([]) = [] |
 odd(x::y) = x::even(y);
fun even([]) = [] |
 even(x::y) = odd(y);
```

Function `reverse` reverses any list, e.g., `reverse([1,2,3]) = [3,2,1]`. Functions odd and even have been discussed before but are now defined using pattern matching. Note that they can be applied to any list now.

6.10.3. Random distribution functions in CPN Tools

CPN Tools can be used to simulate CPN models. Section 8.5 will discuss the use of CPN models for simulation purposes in detail. In this section, we highlight several functions that are useful for simulation purposes.

The function `step` returns the number of the last step, i.e., `step()` returns an integer n when for the n-th time some transition fires.

The function `time` returns the current time in the model. This may be useful for measuring cycle times etc. Function `time` cannot be used on input arcs and in transition guards because this would lead to semantical problems. `time()` returns an infinite integer (`IntInf`) and needs to be cast to the proper type, e.g., use `IntInf.toString(time())` to cast the time onto a string value that can be passed on to a token. The expression `IntInf.toInt(time())` can be used to cast the time onto an integer. After casting the time onto an integer it can be used to calculate service times, waiting times, and flow times.

When using a CPN model as a simulation model it is necessary to attach probabilities to certain events. For example, the duration of some production step may be sampled from some probability distribution or a path in the process is taken with a certain probability. There are two ways to achieve this: (1) using the function `ran` on a finite color set or (2) using one of the predefined random distribution functions.

Examples of finite color sets are `color B = bool`, `color Age = int with 0..99`, `color Sex = with M|F`, etc. These color sets have in common that the number of possible values is limited. It is possible to apply the function `ran` to these color set to obtain a random value from these sets. For example, `B.ran()` yields a boolean, `Age.ran()` yields a integer between 0 and 99, and `Sex.ran()` yields `M` or `F`. Every value of the multiset has an equal probability. The function `ran` can be used in other functions or on output arcs. An output arc with an inscription `Age.ran()` produces a token with a value between 0 and 99. An output arc with an inscription `if B.ran() then Age.ran()` produces a token with a value between 0 and 99 but only in 50 percent of the cases. An output arc with an inscription `Sex.ran()@+Age.ran()` produces a token with a delay between 0 and 99 and value `M` or `F`.

Alternatively, it is possible to use predefined random distribution functions. For example, `uniform(a:real, b:real) : real` is a function that yields a random number between a and b. The probability distribution is uniform, i.e., any value between a and b has the same likelihood of being chosen by `uniform(a,b)`. Note that both the parameters a and b and the result are reals. Since tokens and delays cannot be of type `real`, the result of `uniform(a,b)` needs to be cast onto another type, typically `int`. For this purpose, one can use the functions `floor`, `ceil`, and `round`. The first function produces `floor(r)`, the largest integer not larger than r. The second function produces `ceil(r)`, the smallest integer not less than r. The third function yields the integer nearest to r. Some examples: `floor(5.2)=5`, `ceil(5.2)=6`, and `round(5.2)=5`. `floor(uniform(1.0,5.0))` yields 1, 2, 3, or 4 (each with equal probability). `ceil(uniform(1.0,5.0))` yields 2, 3, 4, or 5 (each with equal probability). `round(uniform(1.0,5.0))` yields 1, 2, 3, 4, or 5. In the latter case, 1 and 5 have a smaller probability. To be precise, the probability of

1 or 5 is equal to the probability of 2.

The function `bernoulli(p:real) : int` returns an integer taken from a Bernoulli distribution with probability p of success, i.e., `bernoulli(p) = 1` with probability p. Note that p is real number between 0 and 1.

The function `binomial(n:int, p:real) : int` returns a sample from a binomial distribution with n experiments and probability p for success. Note that n is an integer and p a real number between 0 and 1. Throwing a dice 50 times and observe how many times a six was thrown, corresponds to a binomial distribution with parameters $n = 50$ and $p = 1/6$, i.e., `binomial(50,0.166666)`.

The sum of the squares of n independent normally distributed random variables with mean 0.0 and standard deviation 1.0 is a chi-squared distribution with n degrees of freedom. The corresponding function is `chisq(n:int) : real`. The distribution is mainly used for statistical tests.

The function `discrete (a:int, b:int) : int` is the discrete version of function `uniform(a,b)`, i.e., `discrete(1,5)` yields 1, 2, 3, 4, or 5 (each with equal probability). Note that `discrete (a,b)` more or less equals `floor(uniform(real(a),real(b)))` and `discrete(floor(a),floor(b))` more or less equals `floor(uniform(a,b))`. (Functions `real()` and `floor()` cast some value onto the proper type.)

The function `erlang (n:int, r:real) : real` can be used to get values from an Erlang distribution with parameters n and r where n is a positive integer and r a positive real. An Erlang distribution with parameters n and r can be derived by adding of n random numbers from a exponential distribution with parameter r.

Probably the most widely used random distribution function for simulating inter-arrival times in a simulation is the function `exponential(r:real) : real` where r is a positive real. This function can be used to generate a so-called Poisson arrival process.

Three other random distribution functions are `normal(n:real, v:real) : real`, `poisson(m:real) : int`, and `student (n:int) : real`. `normal(n,v)` returns a sample from a normal distribution with mean n and variance v. (Note that v should be a positive real.) `poisson(m)` returns an integer taken from a Poisson distribution with intensity m. (Note that m should be a positive real.) `student(n)` returns a sample from a Student distribution (also called t-distribution) with n degrees of freedom. (Note that n should be a positive integer.)

CPN Tools does not allow for reals and uses an infinite integer ("IntInf.int" in Standard ML terms) to represent time. Therefore, one needs to choose an appropriate time scale and cast reals to integers when assigning a value or timestamp to a token. Choose a time scale of milliseconds rather than seconds, or seconds rather than minutes, or minutes rather than hours. For example, if the sending of a message takes between 1 and 2 seconds, do not choose the time scale of seconds because both `discrete(1,2))` and `floor(uniform(1.0,3.0))` yield 1 or 2, i.e., it is not possible to have a real number in-between these two integers. If a timescale of milliseconds is used, we can use `discrete(1000,2000)` or `floor(uniform(1000.0,3000.0))` and get a more continuous view on time.

6.11. More Examples

The only way to learn to model complex processes is through practice. Moreover, it is important to see many examples using various constructs. When looking at an example one should not just try to understand the solution but also ask the question "How would I have modeled this?". This question is very important because models in general are easy to understand once they are ready and correct but difficult to construct.[6] In this section we present two new examples. Please observe not only the model but also the design choices that where made to construct the model.

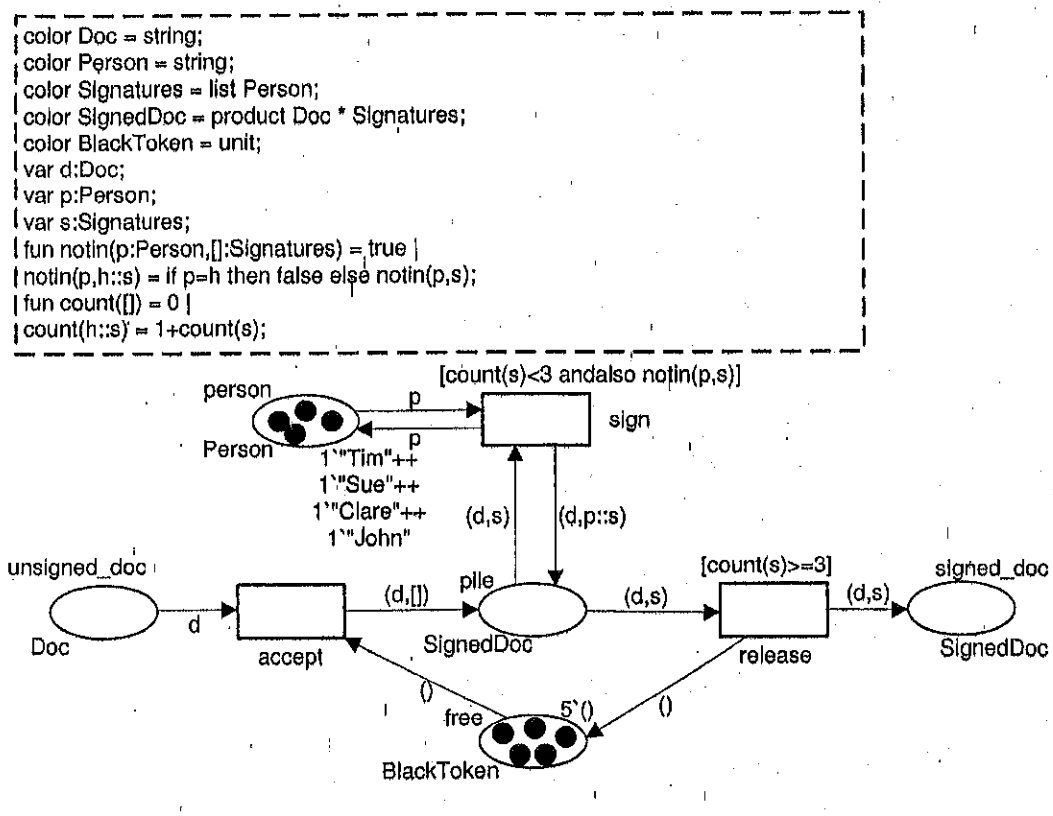


Figure 6.23.: A CPN model for signing documents.

Figure 6.23 shows a small CPN which involves documents that need to be signed and persons that sign these documents. For simplicity, both entities (documents and persons) are represented as strings. Each document needs to be signed by precisely three persons. It is not allowed that the same person signs the same document multiple times. Moreover, work-in-progress is limited to five, i.e., not more than five documents can be in the "signing phase" at the same time. To keep track of the signatures color set `SignedDoc`

[6] An interesting analogy is the difference between watching a sports like soccer and actually playing it. Understanding the game does not imply that one actually masters it. Similarly, it is much more difficult to construct a model than it is to understand one.

includes a list of signatures. Each signature is represented by the corresponding person. Transitions `accept` and `release` are used to control the work-in-progress. Each token in `free` represents a free slot. Initially the number of tokens in this place is five, thus limiting the work-in-progress, i.e., the number of token in place `pile` to five. `accept` puts a token in `free` with an empty list of signatures. Transition `sign` adds signatures to this list by matching a person and a document. The guard of `sign` specifies a precondition for such a match: there should be less than three signatures on the document and only persons that did not sign yet are allowed to put their signature on the document. The guard of `sign` uses the recursive function `notin` which checks whether person `p` appears in `s`. The way this function is specified is not important at this point. (In fact `notin` could be replaced by a standard ML function.) `release` has a guard to make sure that only completed documents (i.e., document with three signatures) are released.

Exercise 6.11 *Replace place `free` in Figure 6.23 by a place holding always one token with a value to indicate the number of documents being processed.*

The second example we consider in this section is a simple thermostat system. This example illustrates the use of time in CPN. Consider a room which at any point has some temperature. There is a heater to warm up the house and there is a door which opens every hour such that part of the warmth escapes. When the door opens the temperature in the room suddenly drops by three degrees centigrade. The heater has a capacity of heating the room 1 degree centigrade every 15 minutes. When the heater would be on the whole time the temperature would continue to rise by 1 degree per hour. Therefore, there is a control system, i.e., the thermostat, which switches off the heater. The thermostat uses the following rules. If the temperature drops below 18, the heater is switched on. If the temperature rises above 22, the heater is switched off. Figure 6.24 shows the corresponding CPN model. Place `temp` records the temperature. Initially it is 15 degrees. Places `on` and `off` represent the two states of the thermostat. These places hold uncoloured tokens (i.e., tokens of type `unit` which has only one value `()`). Place `heater` is only added for timing purposes. This place is has a timed color set (`BT`) and through the delay via this place it is made sure that the room temperature can only rise 1 degree centigrade every 15 minutes. Similarly, place `door`, also of type `BT`, is only added for timing reasons (once every hour the door opens). Note that variables `a` and `b` are introduced for the arc instructions relating to uncolored tokens. These variables can also be replaced by constants of value `()`, i.e., simply replace `a` and `b` by `()`.

Exercise 6.12 *Describe the room temperature in time starting in the initial state shown, i.e., play a timed, colored "token game".*

Note that playing the "timed, colored token game" is exactly what happens in a simulation tool.

Exercise 6.13 *Extend the model shown in Figure 6.24 such that there is a day program and a night program. From midnight to 8am, the thermostat tries to keep the temperature between 14 and 18 degrees centigrade. (If the temperature drops below 14 the heater is*

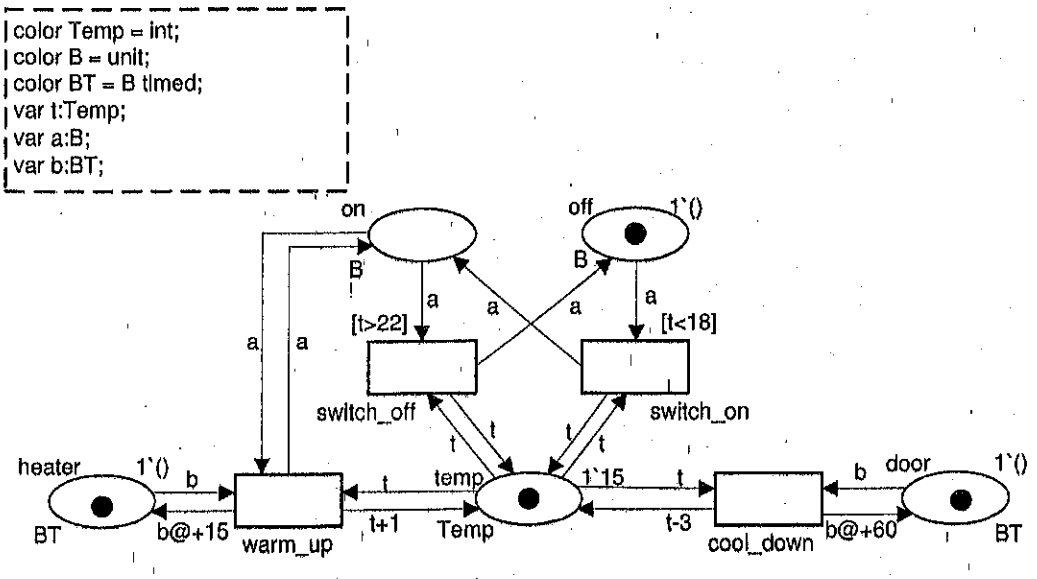


Figure 6.24.: Thermostat CPN model.

switched on. If the temperature rises above 18 the heater is switched off.) From 8am to midnight, the temperature is kept between 18 and 22 degrees, like in Figure 6.24.

6.12. More information about ML and CPN

Standard ML is a popular functional programming language created in the 1980's. In 1987, Robin Milner and his team won the BCS Award for Technical Excellence for work on Standard ML. The language was standardized in 1990 and is one of a very few programming languages with a fully formal definition, giving it significant appeal for both research purposes, and industrial-strength applications. One of the important features of the language is that it is safe: all errors that could "crash" an ML program are detected at compile-time or handled neatly at run-time. This property makes program development and debugging much easier than in most other programming languages.

For more information:

1. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. The Definition of Standard ML: Revised 1997. The MIT Press, 1997.

2. J. D. Ullman. Elements of ML Programming (ML 97 edition). Prentice-Hall, 1998.

Coloured Petri Nets (CP-nets or CPNs) is a modeling language developed for systems in which communication, synchronization and resource sharing play an important role. It combines the strengths of ordinary Petri nets with the strengths of a high-level programming language. Petri nets provide the primitives for process interaction, while the programming language provides the primitives for the definition of data types and the manipulations of data values. There have been several proposals to extend Petri nets

with color and/or time. However, the proposal by Kurt Jensen has been most successful and is supported by both a strong theoretical basis and tools.

For more information, the reader is referred to

1. K. Jensen: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997.

2. K. Jensen: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods. Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997.

3. K. Jensen: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use. Monographs in Theoretical Computer Science, Springer-Verlag, 1997.

4. K. Jensen and G. Rozenberg (eds.): High-level Petri Nets. Theory and Application. Springer-Verlag, 1991.

6.13. Summary

In this chapter we presented a precise notation for the concepts introduced in the previous chapter. For this purpose we used the CPN language. Tokens carry data, referred to as color, and may have a time stamp. Places are typed by a color set and may have an initialization expression to specify the initial marking of the place. The input/output behavior of transitions is specified by arc inscriptions. Both initialization expressions and arc inscriptions may yield multisets to indicate the presence of multiple tokens. Arc inscriptions typically have variables which are bound to concrete values at execution. Variables have a type as indicated by their color set. To restrict the binding of these variables, transitions may have a guard. The guard is an additional precondition which needs to be satisfied in order to enable the transition. The combination of tokens in a given marking, arc inscriptions, and guards results in a set of binding elements. A binding element corresponds to an enabled transition with a fixed consumption/production pattern.

A CPN model consists of a graphical structure and declarations. There are four types of declarations: (timed) color sets, variables, constants, and functions. Later we will show that a CPN model can also be structured hierarchically as a collection of pages.

After studying this chapter you should be able to apply all the concepts mentioned in this chapter. For example, in the remainder we assume detailed knowledge of the basic types and type constructors used in CPN. However, we do not assume the ability to design complex (recursive) functions. This topic will also be addressed in one of the next chapter.

Test Yourself

Problem 6-1 A small model railway has a circular track with two trains A and B, which move in the same direction. The track is, for safety reasons, divided into seven different sectors $S = \{S_1, S_2, \ldots, S_7\}$ (see Figure 6.25). At the start of each sector a signal post indicates whether a train may proceed or not. By means of a set of sensors situated at the signal posts it can be automatically determined whether a given sector is empty or not. To allow a train to enter a sector S_i it is of course necessary to require that the sector is empty. However, it is also necessary to require the next sector S_{i+1} to be empty. Otherwise a stopped train could be situated at the very beginning of S_{i+1} and it would be impossible for the incoming train to stop before colliding with the train ahead.

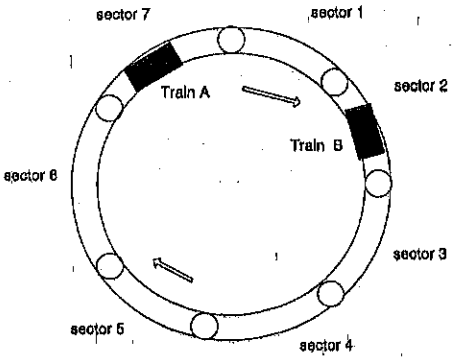


Figure 6.25.: A railway system with 7 sectors and two trains.

(a) Describe the train system by a classical Petri net. Each sector S_i may be represented by three places O_{ia}, O_{ib}, and E_i (where O_{ix} is shorthand for "sector S_i is occupied by train x" and E_i is shorthand for "sector S_i is empty". Make a simulation of the constructed model, i.e., play the "token game".

(b) Describe the same system by a CPN model where each sector is described by two places O_i and E_i. O_i has the set $\{a, b\}$ as possible token colours while E_i is just a placeholder. Make a simulation of the constructed CPN model. Compare the CPN model with the classical Petri net in (a).

(c) Describe the same system by a CPN model which only has two places O and E, and a single transition *Move to next sector*. Make a simulation of the constructed model.

(Example is taken from K. Jensen: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 1997.)

Problem 6-2 Imagine a system where five Chinese philosophers are situated around a circular table (see Figure 6.26). In the middle of the table there is a delicious dish

of rice, and between each pair of philosophers there is a single chopstick. Each philosopher alternates between thinking and eating. To eat, the philosopher needs two chopsticks, and she is only allowed to use the two which are situated next to her (on her left-hand and right-hand side). It is obvious that this restriction (lack of resources) prevents two neighbours from eating at the same time.

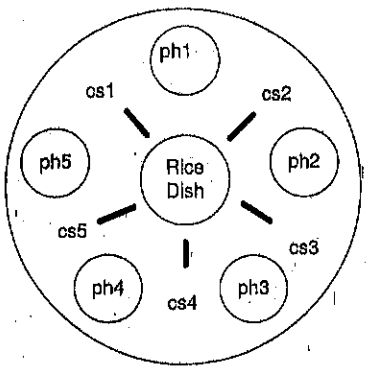


Figure 6.26.: Five philosophers $PH = \{1, 2, \ldots 5\}$ eating with chopsticks $CS = \{1, 2, \ldots, 5\}$.

(a) Describe the philosopher system by a classical Petri net. It is assumed that each philosopher simultaneously (and indivisibly) picks up his pair of chopsticks. Analogously, he puts them down in a single indivisible action. Each philosopher may be represented by two places (*Think* and *Eat*) and two transitions (*Take_chopsticks* and *Put_down_chopsticks*) Each chopstick may be represented by a single place (which has a token when the chopstick is unused). Make a simulation of the constructed model.

(b) Describe the same system by a CPN model which contains the two colour sets $PH = \{ph1, ph2, ..., ph5\}$ and $CS = \{cs1, cs2, ...cs5\}$, representing the philosophers and the chopsticks, respectively. It is only necessary to use three places (*Think*, *Eat* and *Unused_chopsticks*) and two transitions (*Take_chopsticks* and *Put_down_chopsticks*). Make a simulation of the constructed model.

(c) Modify the CPN model created in (b), so that each philosopher first takes her right chopstick and next the left one. Analogously, she first puts down his left chopstick and next the right one. Make a simulation of the constructed CPN model. Does this modification change the overall behaviour of the system (e.g., with respect to deadlocks)?

(d) Modify the CPN model created in (c), so that each philosopher takes the two chopsticks one at a time (in an arbitrary order). Make a simulation of the constructed CPN model. Does this modification change the overall behaviour of the system (e.g., with respect to the deadlocks)?

(Example is also taken from K. Jensen: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 1997.)

Answers and Solutions

Solutions to Exercises

1. The types and results of the expresions:

   a) `5 > 6` has type `bool` and value `false`.

   b) `"Hello" ^ " " ^ "World" = "Bye"` has type `bool` and value `false`.

   c) `not(true andalso (1=0))` has type `bool` and value `true`.

   d) `if "OK"="NOK" then 4 div 2 else 6 div 2` has type `int` and value 3.

2. The value of `Monaco` is

```
[{d="Jos Verstappen", r=[(1,31000),(2,33400),(3,32800)]},
{d="Michael Schumacher", r=[(1,32200),(2,31600),(3,30200),(4,29600)]},
{d="Rubens Barrichello", r=[(1,34500),(2,32600),(3,37200),(4,42600)]}]
```

   The value and types of the three constants:

```
e1 = [(1,31000)] : LapTimes;
e2 = "Michael Schumacher" : Driver;
e3 = [(1,31000),(2,33400),(3,32800),(1,34500),(2,32600),
      (3,37200),(4,42600)] : LapTimes;
```

3. See Figure 6.27. Given its similarities with the voting process, it should be self-explaining.

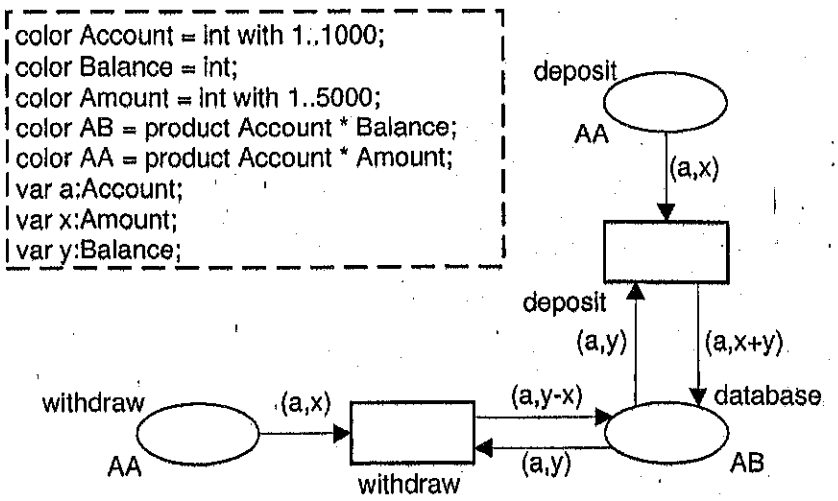


Figure 6.27.: A simple CPN model of the banking system.

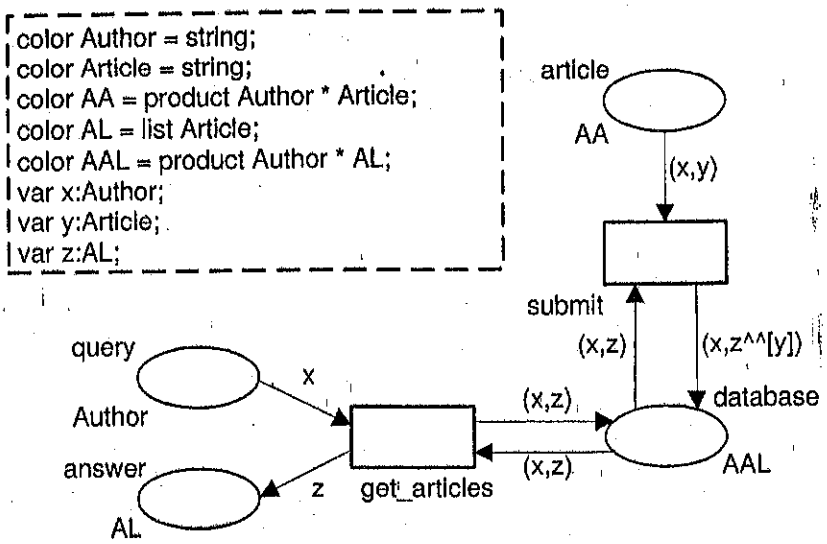


Figure 6.28.: A database system for articles.

4. See Figure 6.28.

5. See Figure 6.29. Compared to Figure 6.28 a new place and transition have been added to support the addition of new authors to the database. More complex are the changes related to multiple authors per article. Transition `submit` now "peals off" the list of authors. If there is just a single author, no token is returned to place `article`. However, if there is a second author, the first author is removed and a token is returned to place `article`, etc. The rest of the model did not change.

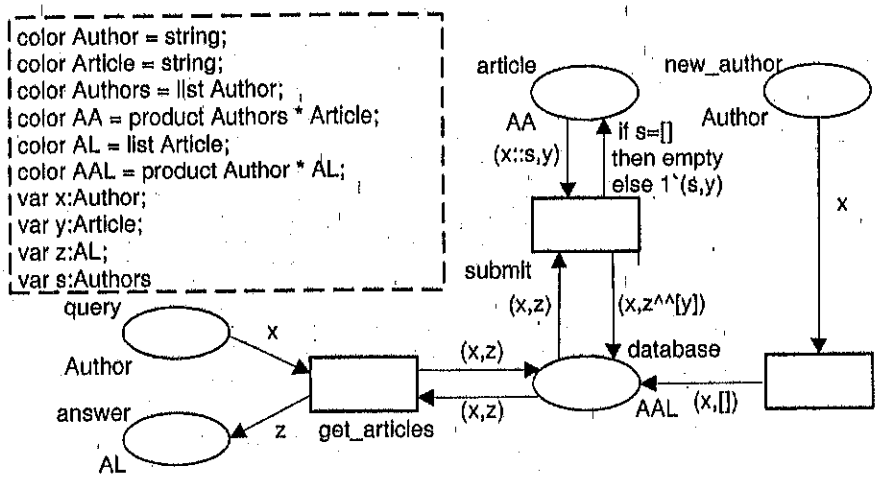


Figure 6.29.: A more complex CPN model of the banking system.

6. See Figure 6.30. Note the addition of the guard.

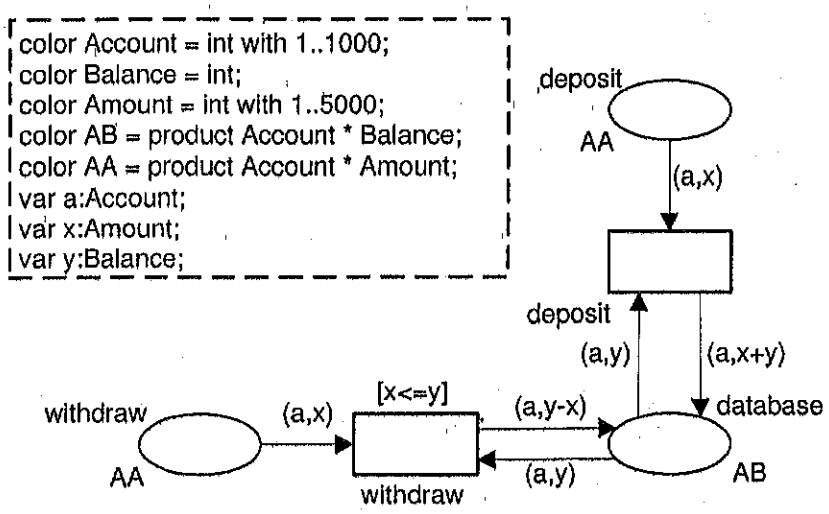


Figure 6.30.: A CPN model of the banking system using a guard.

7. `fib(3)` equals 3. `fac(fib(3))` equals 6. Functions `odd` should be modified as follows:

```
fun odd(x:L) = if x=[] then [] else even(tl(x))^^[hd(x)];
fun even(x:L) = if x=[] then [] else odd(tl(x));
```

8. The time stamps of the tokens in p7 in the final marking are: 0, 1, 2, and 3. The left-hand side of Figure 6.31 is modified such that for positive values of y the delay is 10 while for other values the delay is 20. The right-hand side of Figure 6.31 is modified such that the delay for "Hi" is 10 while for other values the delay is y.

9. To model that female patients cannot be served by employees with less than 5 years of experience, simply add the following guard to transition `start`:

```
[#Gender(p)="M" orelse #Experience(e)>=5]
```

To model that an employee takes a rest after each customer, add a delay to the arc from `stop` to `busy`, i.e., change arc inscription `emp` into:

```
emp@+if #Gender(p)="M" then 5 else 3
```

If the employee served a male patient, it takes 5 minutes to recover, otherwise only 3 minutes.

10. See Figure 6.32. The total number of products is also recorded in an additional place. A transition is added to model accidental reductions of the stock.

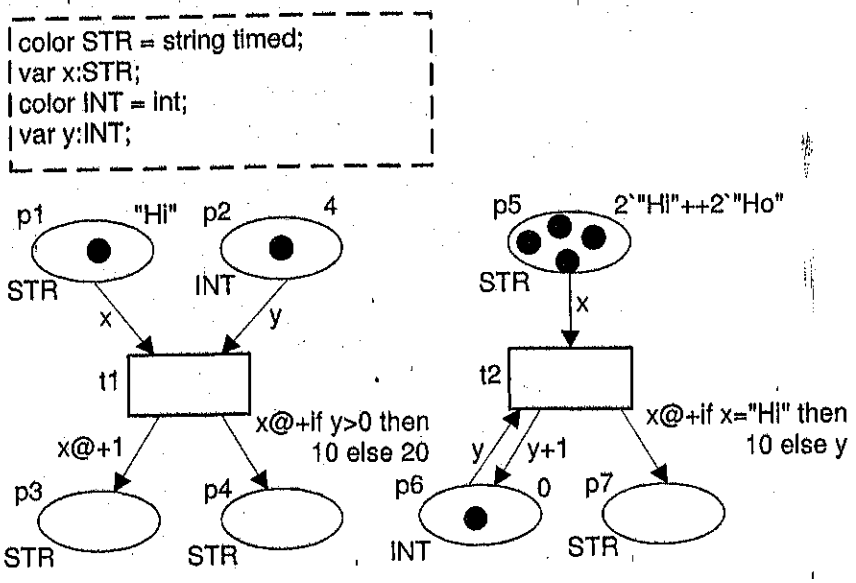


Figure 6.31.: The modified examples.

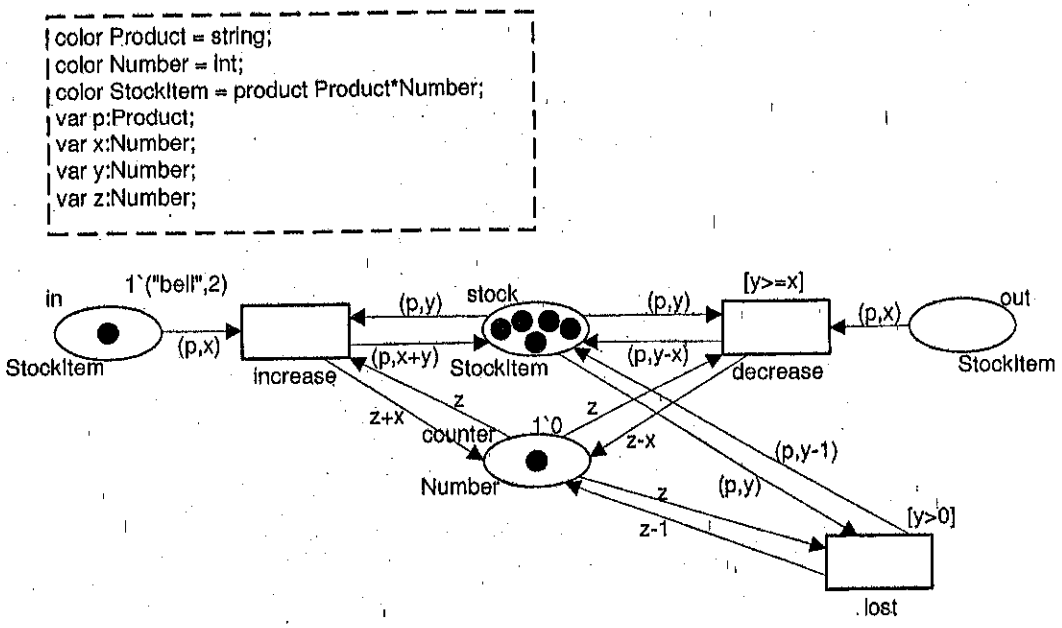


Figure 6.32.: A CPN model that also counts the products in stock and allows for missing products.

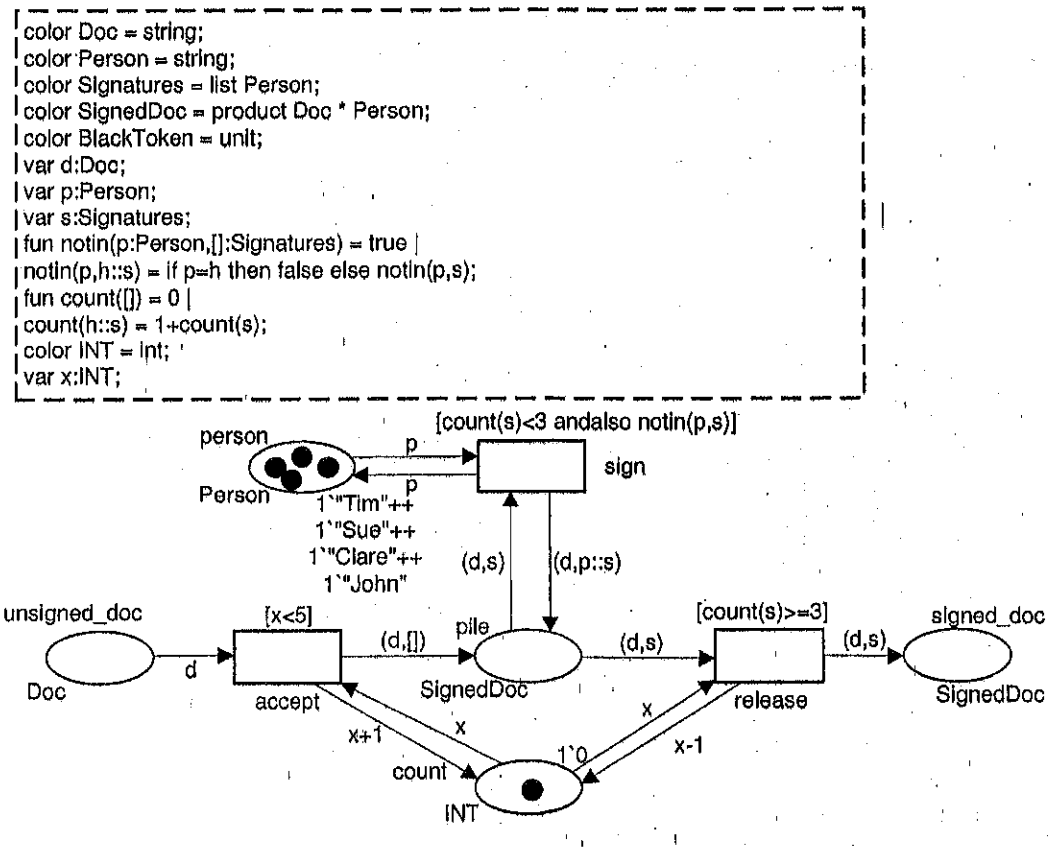


Figure 6.33.: A CPN model uses one colored token rather than multiple black tokens to record the work-in-progress.

11. See Figure 6.33. Note the guard that has been added to transition `accept`.

12. At time 0 the room is 15 degrees (0:15). The thermostat switches on at the same time and it becomes instantly 16 degrees (0:16). At the same time the door opens and it gets 13 degrees (0:13). Note that the two events both take place at the same time but their order could be reversed and still result in 13 degrees at time 0. Now there is a delay of 15 minutes to get 14 degrees (15:14), etc. A possible path is thus (0:15), (0:16), (0:13), (15:14), (30:15), (45:16), (60:17), (60:14), (75:15), (90:16), (105:17), etc.

13. See Figure 6.34.

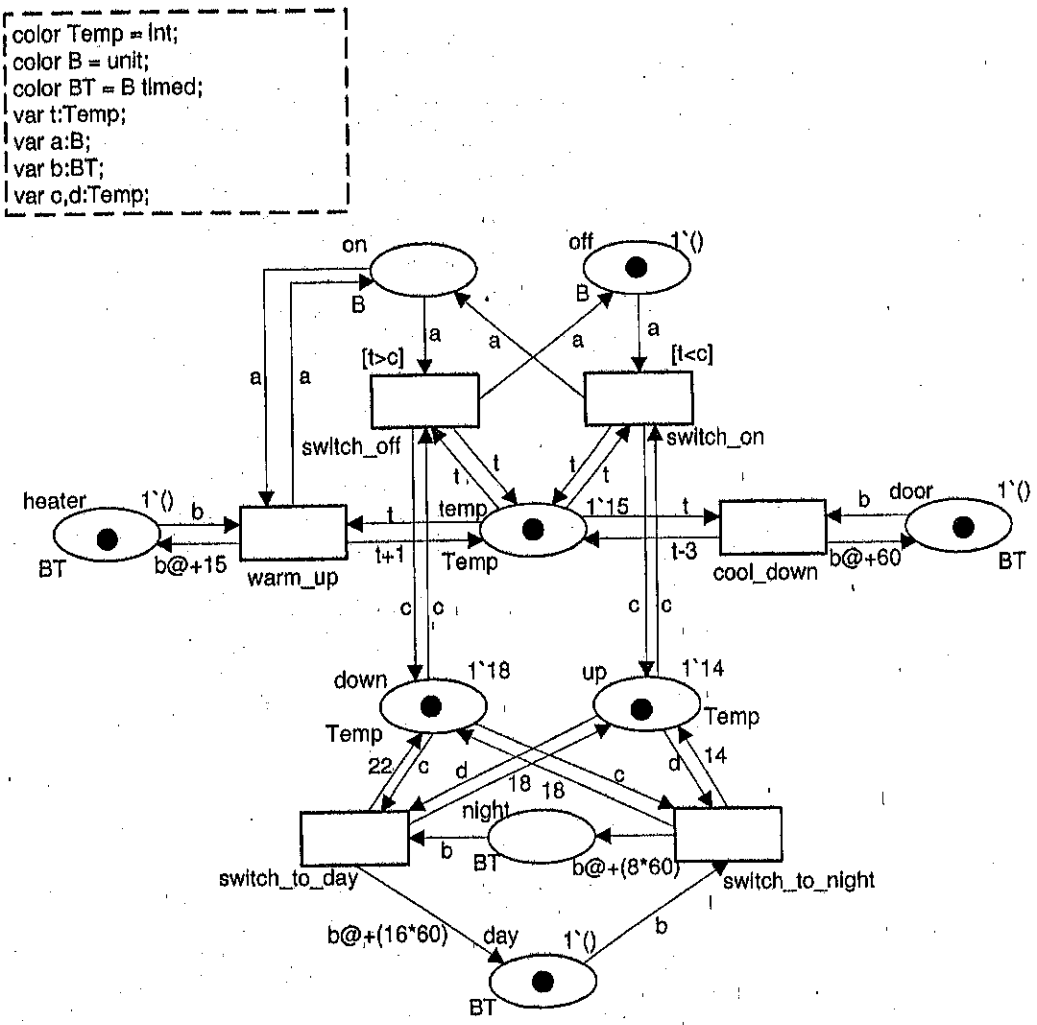


Figure 6.34.: A CPN model of the thermostat with day and night program.

Solutions to the Test

6-1 (a) Figure 6.35 models the railway as a classical Petri net. Note that only a part of the model is shown because of the repetitive nature of the model. The model is linear in the number of tracks and trains, e.g., if the number of tracks is doubled the network gets twice as large.

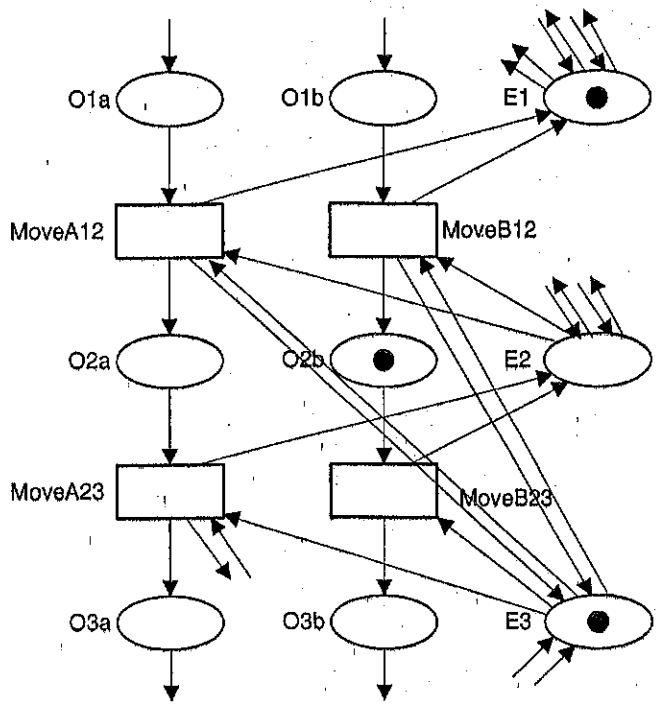


Figure 6.35,: The railway system modeled as a classical Petri net.

(b) Figure 6.36 models the railway in CPN with limited use of token colors. Note that only a part from the model is shown because of the repetitive nature of the model. The model is smaller but still linear in the number of tracks. However, unlike (a) the size does not depend on the number of trains.

(c) Figure 6.37 models the railway using also colored tokens for the location of the train and empty sectors. Note the compactness of the model compared to (a) and (b). The network does not change if sectors or trains are added.

6-2 (a) Figure 6.38 models the philosophers as a classical Petri net. Only one philosopher is shown because of the repetitive nature of the model. Note that philosopher 5 uses chopstick 5 and 1 thus making the model circular.

(b) Figure 6.39 models the philosophers using the CPN language. Note that the size of the network is independent of the number of philosophers. Also note the use of the mod operator: 1 mod 5 = 1, 2 mod 5 = 2, ... 5 mod 5 = 0.

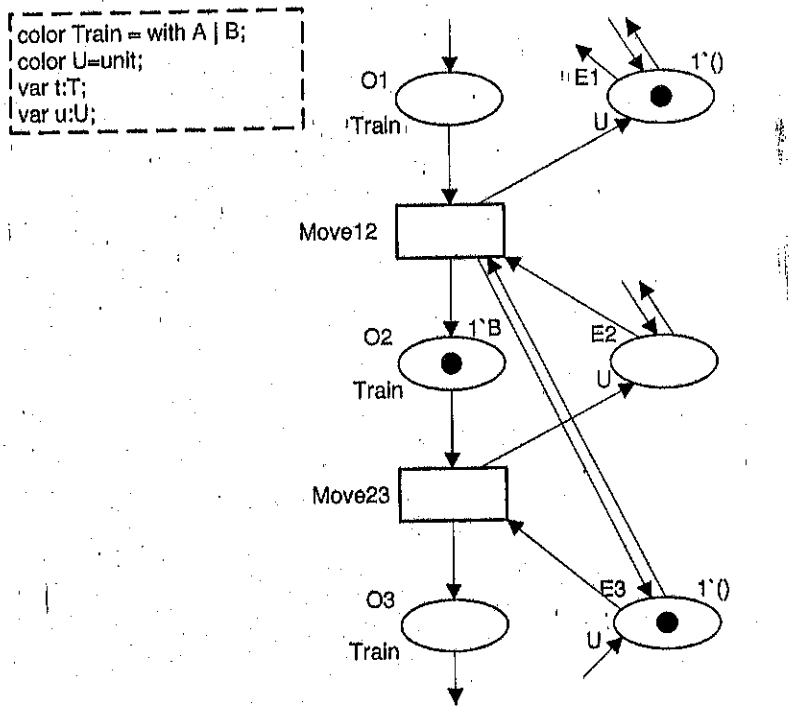


Figure 6.36.: The railway system modeled in CPN with limited colors.

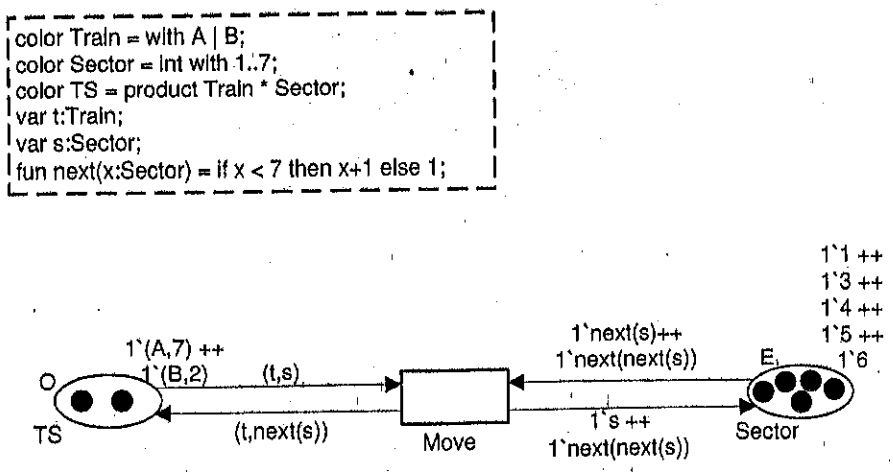


Figure 6.37.: The railway system modeled in term of a CPN.

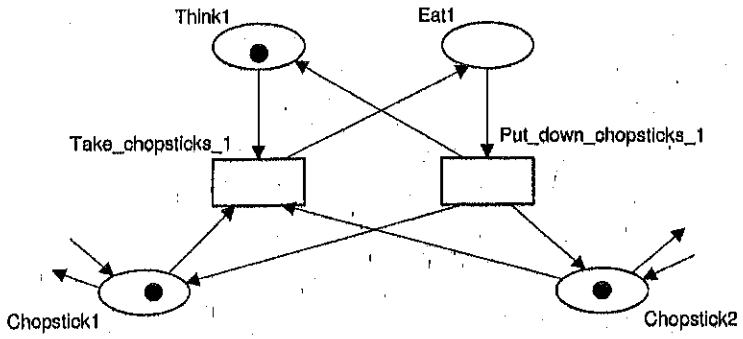


Figure 6.38.: The philosophers modeled as a classical Petri net.

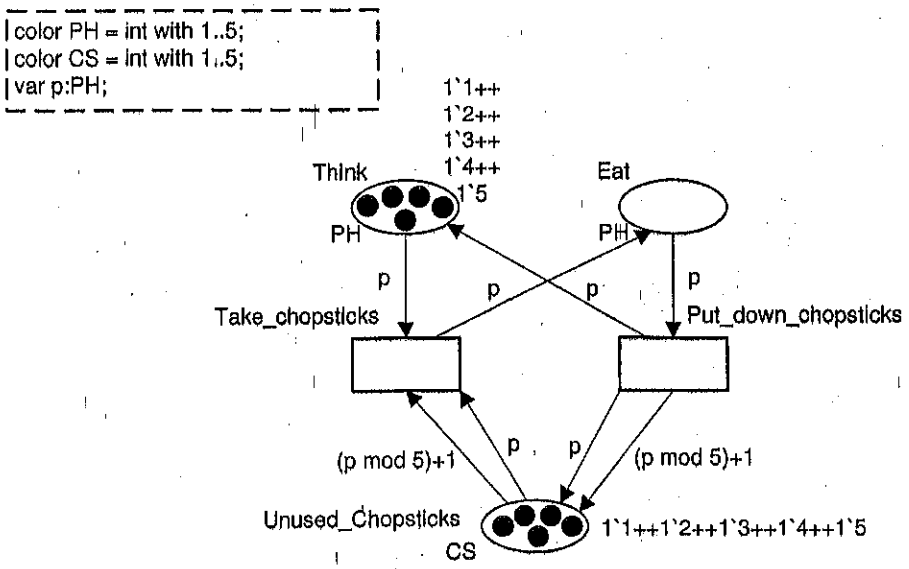


Figure 6.39.: A CPN model of the philosophers taking both chopsticks at the same time.

(c) Figure 6.40 models the situation where philosophers take their chopsticks in two steps (first right, the left). The system has several deadlocks. For example, if each philosopher takes its right chopstick, the system deadlocks because nobody can take a second chopstick.

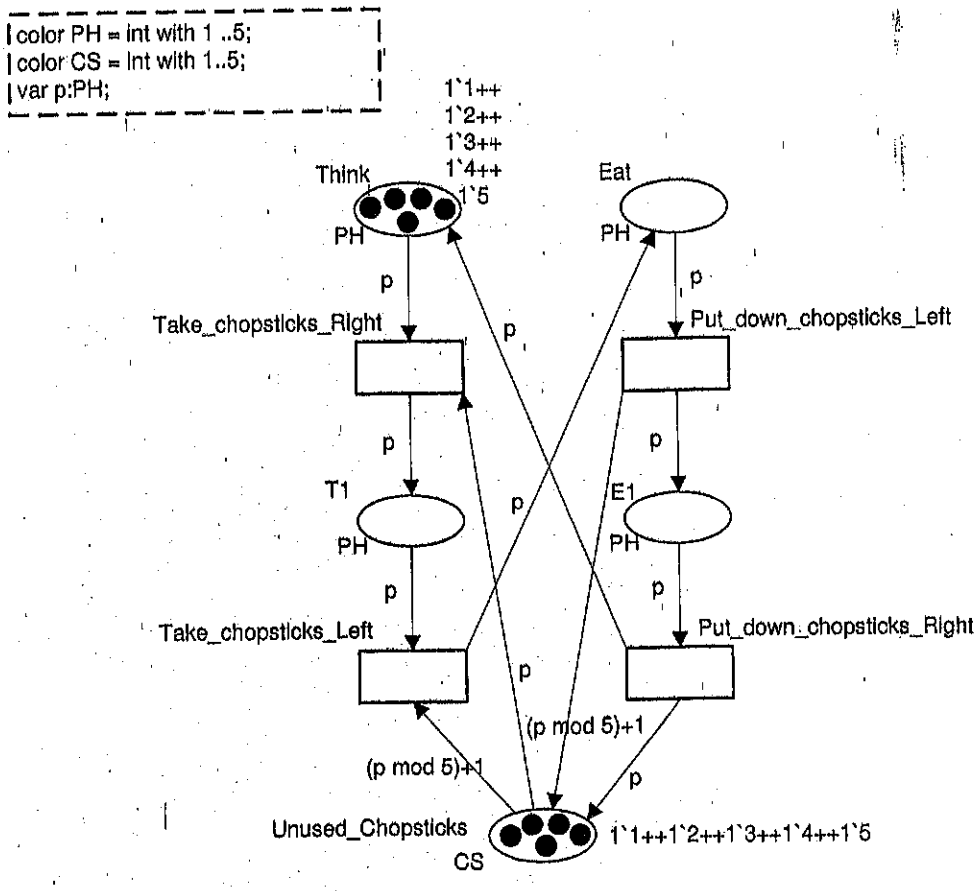


Figure 6.40.: A CPN model of the philosophers taking the right chopstick first.

(d) Figure 6.41 models the situation where philosophers take their chopsticks in two steps but without a predetermined order. Still there is a deadlock problem because if each philosopher takes its right chopstick, the system deadlocks because nobody can take a second chopstick.

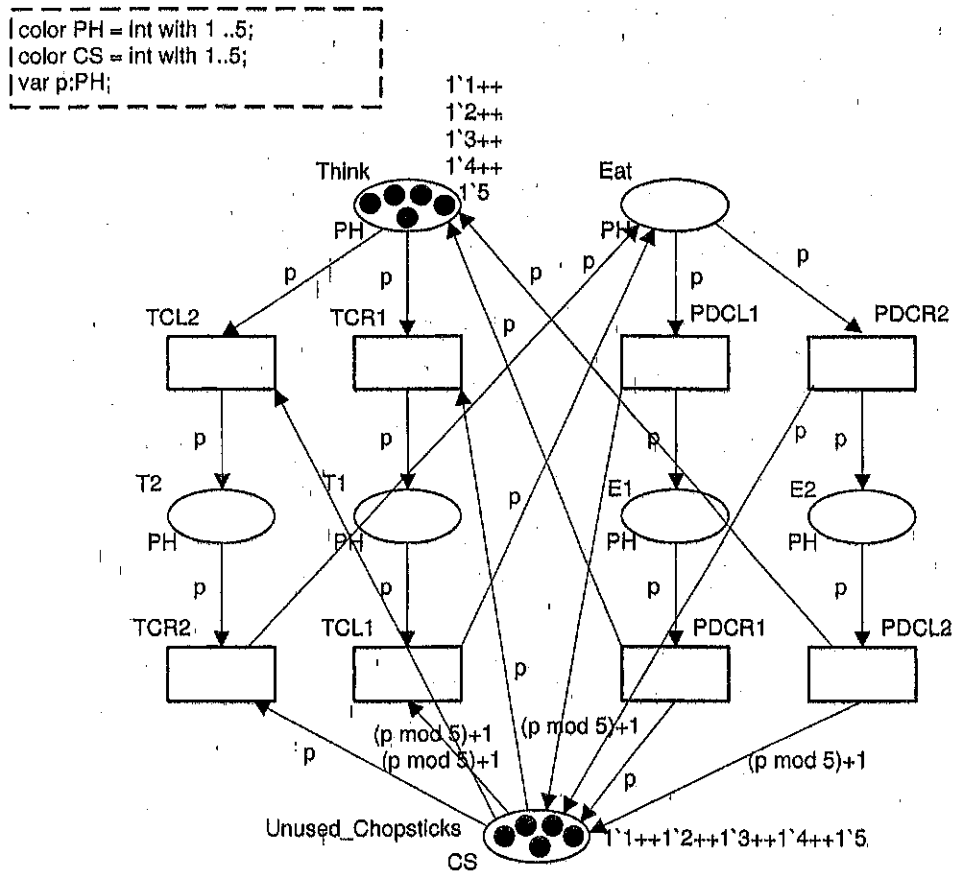


Figure 6.41.: A CPN model of the philosophers taking one (left or right) chopstick first.