Resource Allocation

Abstract
This is a small toy example which is well-suited as a first introduction to CP-nets. The CPN model is described in great detail, explaining the basic concepts of CP-nets. Hence, it can be read by people with no/little Petri net background.

The CPN model describes how two different kinds of processes are sharing three different kinds of resources. It is simple to understand and easy to simulate/modify.

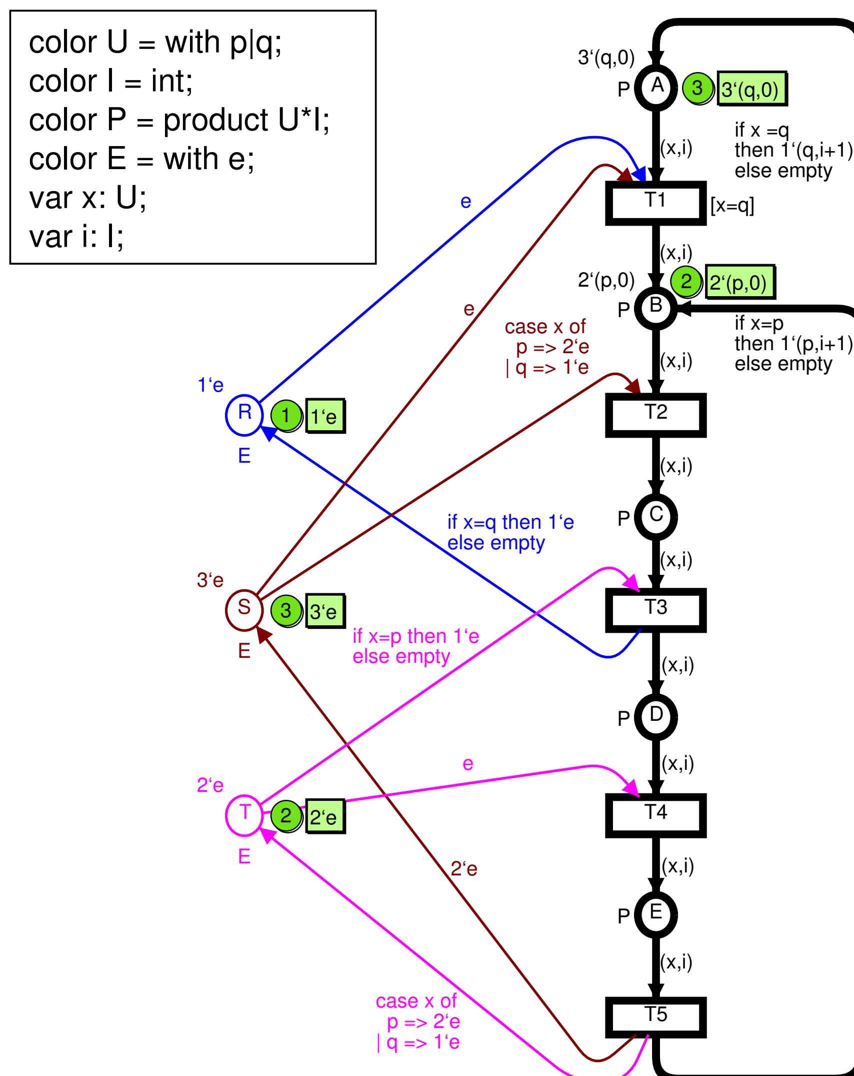
The example is taken from Sect. 1.2 of Vol. 1 of the CPN book.

Developed and Maintained by:
Kurt Jensen, Aarhus University, Denmark (kjensen@daimi.au.dk).

CPN Model

Assume that we have a set of processes, which share a common pool of resources. There are two different kinds of processes (called p-processes and q-processes) and three different kinds of resources (called r-resources, s-resources, and t-resources). The processes could be different computer programs (e.g., text editors and drawing programs) while the resources could be different facilities shared by the programs (e.g., tape drives, laser printers and plotters). Each process is cyclic and during the individual parts of its cycle, the process needs to have exclusive access to a varying amount of the resources. The resource allocation system is modelled by the CP-net shown below.

Initial Marking M_0



The processes can be in five different states, represented by the **places** A–E. Each place may contain one or more markers, called **tokens**. Each token carries a data value – called the **token colour**. The data value may be of arbitrarily complex type (e.g., a record where the first field is a real, the second a text string, while the third is a list of integer pairs). For a given place all tokens must have token colours that belong to a specified type. This type is called the **colour set** of the place.

The use of colour sets in CP-nets is totally analogous to the use of types in programming languages. Colour sets determine the possible values of tokens (analogously to the way in which types determine the possible values of variables and expressions). For historical reasons we talk about "coloured tokens" which can be distinguished from each other – in contrast to the "plain tokens" of an ordinary Petri net.

By convention we write colour sets in italics. From the above figure, it can be seen that the places A–E have the type P as colour set, while the places R–T have the type E as colour set. The **declarations** of the **colour sets** (in the upper left corner of the figure) tell us that each token on A–E has a token colour which is a pair (because the colour set P is declared to be the Cartesian product of two other colour sets U and I). The first element of the pair is an element of U and thus it is either p or q (because the colour set U is declared to be an enumeration type with these two elements). The second element is an integer (because the colour set I is declared by means of the CPN ML standard type int, which contains all integers in an implementation-dependent interval). Intuitively, the first element of a token tells whether the token represents a p-process or a q-process, while the second element tells how many full cycles the process has completed. It can also be seen that all the tokens on R–T have the same token colour (e is the only element of E). Intuitively, this means that these tokens carry no information – apart from their presence/absence at a place.

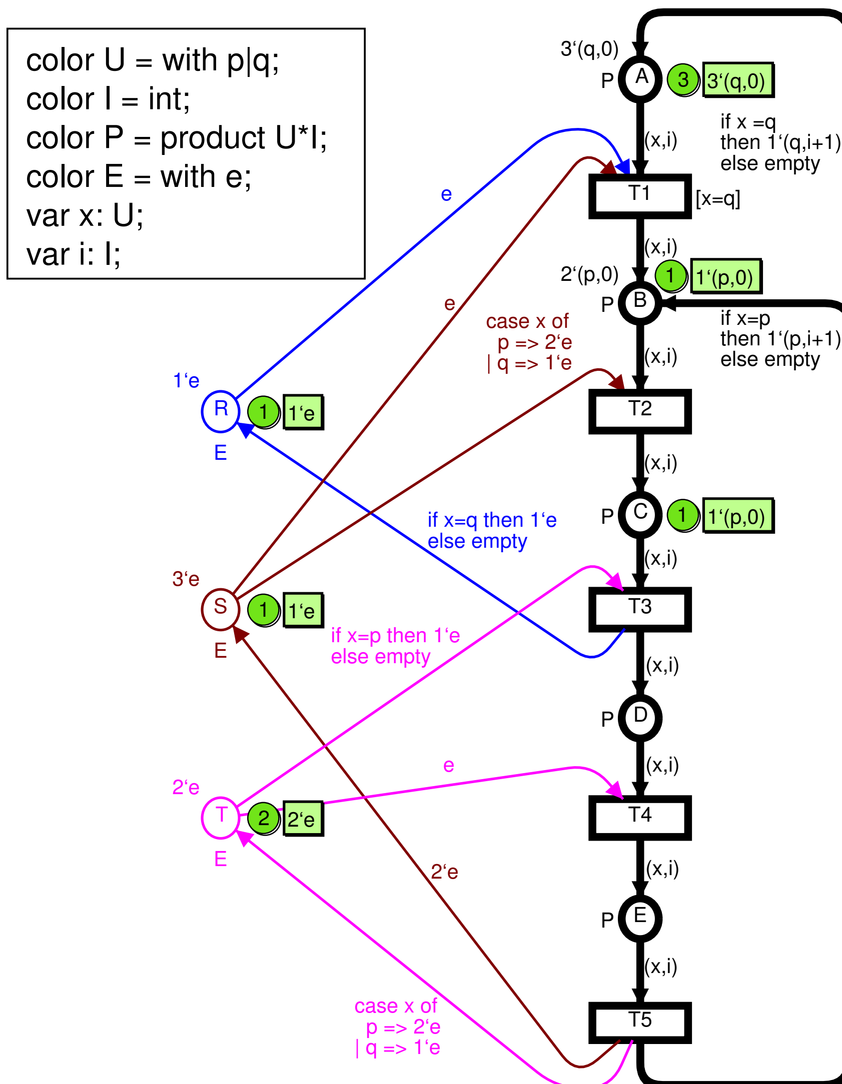
The **initial marking** is determined by evaluating the **initialization expressions**, i.e., the underlined expressions next to the places. In the initial marking there are three (q,0)-tokens on A and two (p,0)-tokens on B, while C, D and E have no tokens (by convention we omit initialization expressions which evaluate to the empty multi-set). Moreover, R has one e-token, S has three e-tokens and T has two e-tokens. The marking of each place is a **multi-set** over the colour set attached to the place. We need multi-sets to allow two or more tokens to have identical token colours. If we only worked with sets it would be impossible, for example, to have three (q,0)-tokens in the initial marking of A.

The **current marking** of a given place is represented by means of a small circle (with an integer saying how many tokens there are) and a text string next to the circle (with a multi-set saying what the individual token colours are, and which coefficients they have). By convention we omit the circle and the text string for places which have no tokens. In the figure above, the current marking

is identical to the initial marking, and this means that the small circles and their text strings contain the same information as the initialization expressions.

Each of the five **transitions** T1–T5 represents a shift from one state to the next. The surrounding arc inscriptions tell us how resources are reserved and released. To see how this works, let us consider transition T2 which has three surrounding arcs. The arc expression “(x,i)” appears twice (on the input arc from B and on the output arc to C) while “case x of p=>2`e | q=>1`e” appears once (on the input arc from S). Together these three arc expressions have two variables, x and i, and from the declarations it can be seen that x has type U while i has type I. At a first glance one might also think that e, p and q are variables, but from the declarations it can be seen that this is not the case: e is an element of the colour set E, while p and q are elements of U. This means that they are constants. Intuitively, the three arc expressions tell us that an occurrence of T2 moves a token from B to C – without changing the colour (because the two arc expressions are identical). Moreover, the occurrence

Marking M_1



removes a multi-set of tokens from S. This multi-set is determined by evaluating the corresponding arc expression. As it can be seen, the multi-set depends upon the kind of process involved. A p-process needs two s-resources to go from B to C (and thus it removes two e-tokens from S), while a q-process only needs one s-resource to go from B to C (and thus it removes only one e-token from S).

Now let us be a little more precise, and explain in detail how the enabling and occurrence of CP-net transitions are determined. The transition T2 has two variables (x and i), and before we can consider an occurrence of the transition these variables have to be bound to colours of the corresponding types (i.e., elements of the colour sets U and I). This can be done in many different ways. One possibility is to bind x to p and i to zero: then we get the **binding** b_1 = <x=p,i=0>. Another possibility is to bind x to q and i to 37: then we get the binding b_2 = <x=q,i=37>.

For each binding we can check whether the transition, with that binding, is **enabled** (in the current marking). For the binding b_1 the two input arc expressions evaluate to (p,0) and 2‘e, respectively. Thus we conclude that b_1 is enabled in the initial marking – because each of the input places contains at least the tokens to which the corresponding arc expression evaluates (one (p,0)-token on B and two e-tokens on S). For the binding b_2 the two arc expressions evaluate to (q,37) and e. Thus we conclude that b_2 is *not* enabled (there is no (q,37)-token on B). A transition can occur in as many ways as we can bind the variables that appear in the surrounding arc expressions (and in the guard – introduced below). However, for a given marking, it is usually only a few of these bindings that are enabled.

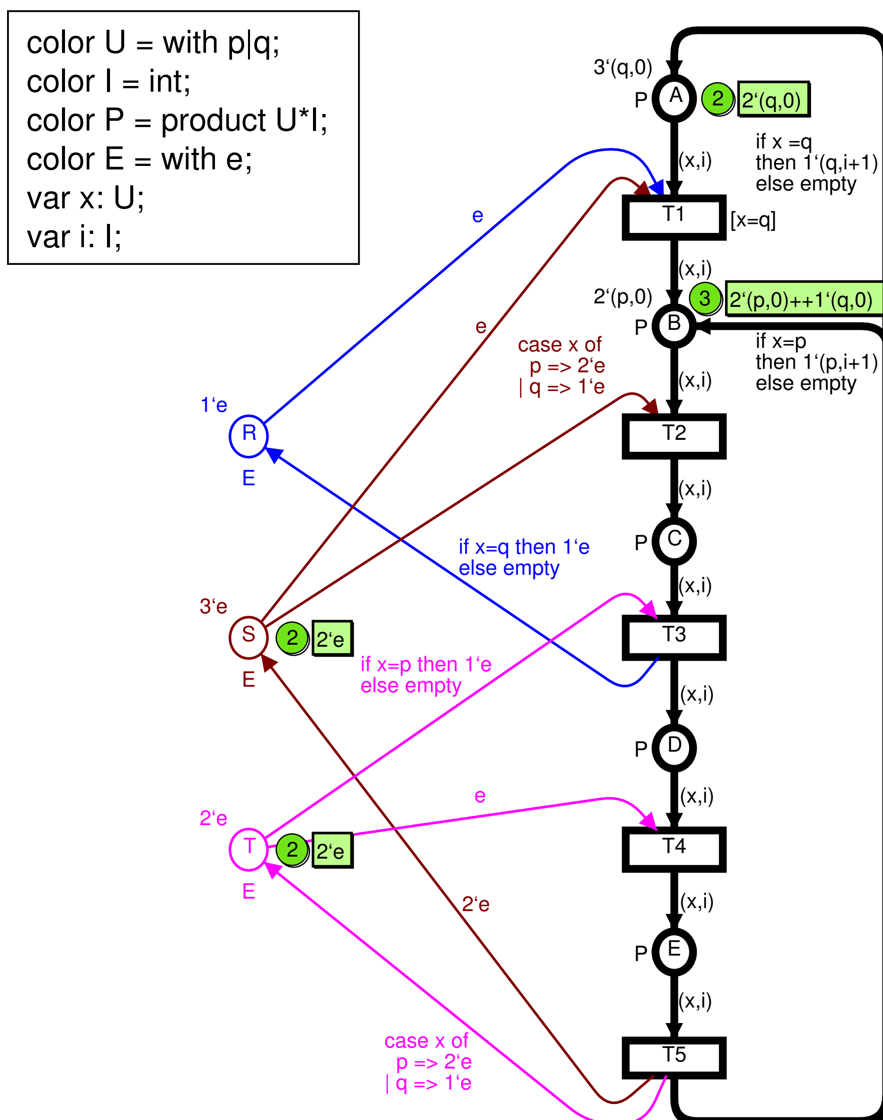
When a transition is enabled (for a certain binding) it may **occur**, and it then removes tokens from its input places and adds tokens to its output places. The number of removed/added tokens and the colours of these tokens are determined by the value of the corresponding arc expressions (evaluated with respect to the binding in question). A pair (t,b) where t is a transition and b a binding for t is called a **binding element.** The binding element (T2,b_1) is enabled in the initial marking M_0 and it transforms M_0 into the marking M_1 (shown below). Analogously, we conclude that the binding element (T1,<x=q,i=0>) is enabled in M_0 and that it transforms M_0 into the marking M_2 (shown below). We say that each of the markings M_1 and M_2 is **directly reachable** from M_0. The binding element (T2,b_2) is *not* enabled in M_0 and thus it cannot occur.

Next, let us take a closer look at transition T5. This transition moves a token from place E to either A or B (p-processes go to B, while q-processes go to A). Simultaneously the transition updates the cycle counter *i*. Notice that different bindings for a transition may not only result in different token colours but also in different *numbers* of tokens. In particular this may mean that the multi-set of tokens which are added/removed, for a given binding, may be empty, as

illustrated by the two thick output arcs of T5. We have positioned the first segments of the two arcs on top of each other to illustrate the close relationship between them. However, it should be stressed that this has no formal meaning. The only purpose is to make the drawing more readable for human beings.

Next let us look at transition T1, which in addition to the arc expressions has a **guard**: “[x=q]”. A guard is a boolean expression (i.e., an expression that evaluates to either true or false). It may have variables in exactly the same way that the arc expressions have. The purpose of a guard is to define an additional constraint which must be fulfilled for a transition to be enabled. In this case the guard tells us that it is only tokens representing q-processes which can move from A to B (because the guard for all bindings <x=p,…> evaluates to false and thus prevents enabling). It is easy to see that we in this case could have omitted the guard, because we never will have p-tokens on place A. However,

Marking M_2



adding the guard makes our description more robust towards errors.

When the same variable name appears more than once, in the guard/arc expressions of a *single* transition, we only have one variable (with multiple appearances). Each binding of the transition specifies a colour for the variable – and this colour is used for all the appearances. However, it should be noted that the appearances of x around T1 are totally independent of the appearances of x around T2 – in the sense that the two sets of appearances in the same step can be bound to different colour values.

It can be shown that the resource allocation system presented above has no deadlock (i.e., no reachable marking in which no binding element is enabled). However what happens if we change the number of processes or the number of resources? As an example, let us assume that the initial marking has an extra s-resource (i.e., an extra e-token on S). One should expect that this small modification cannot lead to a deadlock – because deadlocks appear when we have too few resource tokens, and thus an extra resource token cannot cause a deadlock.

Is the argument above convincing? At a first glance: yes! However, this argument is *wrong*. Adding the extra s-resource actually means that we can reach a deadlock. This can be seen by letting the two p-processes advance from state B to state D, while the q-processes remain in state A.

Hopefully, this small example demonstrates that informal arguments about behavioural properties are dangerous – and this is one of our motivations for the development of the more formal analysis methods, such as occurrence graphs and place invariants.