Moderné vzdelávanie pre vedomostnú spoločnosť/

Projekt je spolufinancovaný zo zdrojov EÚ

MODELOVANIE A SIMULÁCIA

MODELING AND SIMULATION

Fakulta elektrotechniky a informatiky

Štefan Korečko

Európska únia
Európsky sociálny fond

Agentúra Ministerstva školstva, vedy, výskumu a športu SR pre štrukturálne fondy EÚ

Operačný program VZDELÁVANIE

Contents

I Introduction

The discipline of Modelling and Simulation deals with processes that utilize models of systems to study properties of the systems. This involves observing behaviour of a real system, creation of an appropriate model, checking whether the model corresponds to the original, performing experiments with the model and analysing results of these experiments. Before we go deeper into these processes we define basic terms, namely system, model, modelling and simulation. We will also deal with advantages and disadvantages of simulation, cases when it is appropriate to use simulation, tasks related to modelling and simulation and a typical course of a simulation study. This chapter is written primarily on the basis of (Banks et al., 1998), (Banks et al., 2001), (Cassandras & Lafortune, 2008), (Křivý & Kindler, 2001) and lectures from Modelling and Simulation as taught by assoc. prof. Milan Šujanský at the home institution of the author.

Chapter II gives an overview of different types of systems and corresponding mathematical models and the rest of the book deals with modelling and simulation of particular types of discrete-state systems. Chapter III briefly describes modelling and simulation of digital circuits and chapters IV to VI are dedicated to discrete event systems (DES). Chapter IV introduces random numbers and their generators, which are necessary for stochastic DES and chapter V overviews a frequently used and well-researched class of DES, called queuing systems. Chapter VI deals with timed Coloured Petri nets (tCPN), a formal language suitable for modelling and simulation of DES. It also contains a complete simulation study conducted using tCPN. Modelling and simulation of continuous-state systems are not treated here, an interested reader can find exhaustive information about it in (Cellier. & Greifeneder, 1991) and (Cellier & Kofman, 2006) or in (Neuschl et al, 1988).

I.1 System

The modelling and simulation is about *systems*, so we should fix the meaning of this word first. The word system is understood in many different ways, depending on the specific area of human activity or scientific discipline where it is used. Here are some definitions of the term from well-known dictionaries and encyclopaedia:

- An aggregation or assemblage of things so combined by nature or man as to form an integral or complex whole (Encyclopedia Americana).
- A group of related parts that move or work together (Merriam-Webster).
- A set of things working together as parts of a mechanism or an interconnecting network; a complex whole (Oxford Dictionary).
- A combination of components that act together to perform a function not possible with any of the individual parts (IEEE Standard Dictionary of Electrical and Electronic Terms).

Other famous encyclopaedia, the Encyclopaedia Britannica doesn't even define the word system in general, but talks only about different types of systems. Each of the definitions is different, but there are some common features: all of them identify a system as a whole, composed of parts working together. On the basis of this we will fix the meaning of a *system* as follows:

*A **system** is an organized, purposeful structure regarded as a whole and consisting of interconnected elements, which exists and operates in time and space through the interaction of the elements.*

The *elements* can be named in many ways, such as components, entities, factors, members or parts. The elements are regarded basic parts, which cannot be divided. Usually we can identify groups of elements in a system that have common properties and can be described separately. These are called *subsystems*.

Basic characteristics, or properties, of a system are:

- Its elements are interrelated and interdependent.
- It displays properties not possessed by any of the individual elements.
- It is definable. The boundary between a system and its environment has to be clearly and unambiguously defined.

We can rarely think about a system as something that is totally isolated from its surroundings and in the modelling and simulation we usually have to take the surroundings into account. We call it an ***environment of the system*** and it is defined as a set of elements outside the system. A set of environment elements related to the system forms its *significant environment*, which is also called *system environment*. Interaction between a system and its significant environment is usually defined by inputs and outputs of the system.

System configuration is defined by number of elements, type of elements and relations (interconnections) between them. It represents qualitative system characteristics. ***System parameters*** are quantitative system characteristics and ***system structure*** is defined by both system configuration and parameters.

Realization of system properties in time and space defines its ***behaviour***. The behaviour of a system is observed as a dependency of system outputs on system inputs.

I.2 Modelling

Modelling is a process of producing a model and a ***model*** is a representation of the construction (structure) and working (behaviour) of a system of interest. It is similar to but simpler than the original system. A good model is a judicious trade-off between realism and simplicity. Every model is, naturally, also a system. Relation between a system and its model is that of ***similarity*** and we can have two kinds of similarities.

- *Similarity in structure.* This means that a model is composed of elements with the same or similar parameters as those in the original system and similar interconnections. This similarity doesn't have to be 1:1 correspondence; an element of a model can represent a subsystem in the original system.
- *Similarity in behaviour.* Systems similar in behaviour respond to the same inputs with similar outputs.

Similarity in structure in most cases implies similarity in behaviour but systems similar in behaviour often have very different structure.

We distinguish three basic types of models:

- *Physical model* is a smaller or larger physical copy of an object, usually similar in structure and in behaviour to the original. For example a diesel locomotive is a system and its

model for model railway is its physical model. They are similar in behaviour and also in structure, but only to some extent (i.e. the engine of original is different from the model).

- *Mathematical model* describes behaviour of the original by means of a mathematical apparatus. It can be used for formal analysis (analytical computations) or creation of a simulation model of the original. This type of model is also called *conceptual model*. The term conceptual model is also used in cases when such a model is not fully formalised, for example some parts are described by natural language and not by a mathematical apparatus.
- *Simulation model* is the mathematical (or conceptual) model transformed to an executable program. They are similar to the original in behaviour.

In principal, it is always possible to construct a model of a real system. The opposite, however, is not true as we can have mathematical models of systems that are not physically realisable. We call them ***abstract systems***.

I.3 Simulation

The word ***simulation*** can be defined as a manipulation of a model in such a way that it operates in time or space. What is very important, the purpose of the manipulation should be to study properties of the original system (provided that it exists). Because a model is similar to it, we can experiment with the model and apply results to the original. It is not necessary to use computers for simulation; it can be also done on paper or using physical models. But simulation on computers is the most popular nowadays and we will deal exclusively with this kind of simulation in the rest of the book.

Simulation can have various goals and in general we distinguish four types of simulation tasks:

- *System analysis.* The original system exists and its mathematical model is known. The task is to create a simulation model and perform experiments on it. Aim of the experiments can be, for example, optimization of performance of the system.
- *System synthesis*. A mathematical model exists, simulation model is created from it, simulated and results analysed. After a suitable simulation model is created a real system is constructed on its basis. This process usually involves several modifications of the simulation model and repeated analysis.
- *System identification.* The original exists; a task is to find its mathematical model. The extreme case here is so-called "black box problem“, where we can control inputs and read outputs and we have to determine the mathematical relationship between the inputs and outputs.
- *System simulator.* Here a part of the system is in its original form and another part is a simulation model. The task is to interconnect them. An example of such system is a flight trainer.

To perform simulation on digital computers, ***simulation systems*** are used. A simulation system usually consists of a simulation program and a simulation language. A *simulation language* is used to describe simulation models and simulation experiments and a *simulation program* is software for creating simulation models and defining and performing simulation experiments. In some cases a simulation model is directly programmed in some general-purpose programming language (gppl). Then the corresponding gppl serves as a simulation language.

A concrete case of simulation and modelling is called ***simulation study***. Each study consists of several steps, which, according to (Banks et al., 2001), are as follows:

1. *Problem formulation.* The problem, which the study should solve, is defined. It is also necessary to ensure that the problem is understood in the same way by those that have it (i.e. *client*) and by those who will solve it (i.e. *simulation analyst*).
2. *Setting of objectives and overall project plan.* This step starts by defining objectives of the study; that is questions the study should answer. After they are specified, it should be determined whether the simulation is an appropriate technique. If yes, the project plan is prepared. The plan includes statements about systems and their modifications to be considered and modelled, a method for evaluating them, number of people involved, estimated cost of the study and duration and anticipated results of its phases.
3. *Model conceptualization.* Creation of mathematical or conceptual model of the system. The crucial decision here is about a level of detail that will be captured in the model. On the basis of this we can select appropriate mathematical apparatus. Here it is also decided what parameters of the system will be considered fixed, what will constitute (adjustable) input and what (measurable) output of the model. Unfortunately, there is no universal set of instructions to be followed when constructing a model, but it is advised to start with a simple, very abstract, model and refine it step-by-step by adding more and more details until the requested level is reached. It is also recommended to involve the model user in this phase.
4. *Data collection.* Collecting of data about the original system, about its inputs and outputs. The data should cover all situations under which the original system operates that are important for the study. The data are not only important for building of the model but also for its validation. This phase usually occurs simultaneously with the model conceptualization.
5. *Model translation.* Creation (programming) of a simulation model of the system on the basis of the conceptual model.
6. *Verification.* Checking whether the simulation model corresponds to the mathematical or conceptual one.
7. *Validation.* Determination that a model is an accurate representation of the original system. Simulated behaviour of the model is compared to behaviour of the original system.
8. *Simulation experiments and analysis of results.* Simulation experiments are prepared and run (so-called *simulation runs*), results are analysed and answers to the project objectives are formulated on their basis.
9. *Documentation and reporting.* In (Banks et al., 2001) two types of documentation are identified: program and progress. The program documentation describes simulation models in a way similar to the standard software documentation. The progress documentation is a recorded history of simulation experiments. A final report contains analysis results that should clearly answer the project objectives.
10. *Implementation.* Implementation of the study results to the original system.

The steps described fit primarily the system analysis type of simulation study, for other types they need to be modified accordingly. For example, the model conceptualization step is reduced and the validation step is absent in the case of the system synthesis.

As the second step indicates, it is not always appropriate to use simulation. According to (Banks et al., 2001) the simulation should be avoided if

1. *The problem in question can be solved analytically.* Analytically means that we can use some mathematical apparatus, for example differential calculus or trigonometry, to solve the problem in general - that is for any circumstances. This also includes situations when some simple mathematical computation, based on the common sense, is satisfactory.
2. *It is easier to perform direct experiments.* This is, for example, the case when experimenting with the original system doesn't significantly interrupt or influence its normal operation. Or there is enough time to adopt and test changes when the system is not used.
3. *Too expensive.* If the costs of corresponding simulation study exceed savings that will come from implementation of study results (if positive).
4. *Resource and time is not available.* Limited time or access to data about the system to be modelled cause that a true-to-original model cannot be constructed.

Before ending this section we summarize advantages and disadvantages of simulation, again according to (Banks et al., 2001). The advantages are

- Modifications of a system, including new policies, operating procedures and decision rules, can be evaluated without disrupting normal operation of the system or investments to physical realisation of the modifications.
- Systems can be tested before their acquisition.
- Hypotheses about how or why certain phenomena occur can be tested for feasibility.
- Time can be compressed or expanded.
- Insight about internal workings of the system, i.e. about interactions between its elements, importance of their parameters with respect to performance of the system or about understanding how the system operates, can be obtained.
- Bottleneck analysis can be performed to identify elements decreasing efficiency of the system.
- "What-if"" questions can be answered. This is particularly useful in tasks of the system synthesis type.

We can also identify some disadvantages:

- Model building requires special training.
- It may be difficult to interpret simulation results.
- Realisation of a simulation study may be time consuming and expensive.

An example of a complete simulations study can be found in section VI.8.

II Taxonomy of Systems and Models

In the introductory chapter we briefly defined what system and model is. In this one we explore their nature more deeply and present a taxonomy of systems and models, together with several examples. Different taxonomies can be found in related literature, here we will stick to the one presented in (Cassandras & Lafortune, 2008), which deals with discrete event systems (DES) from the point of view of the systems and control theory. The primary reason why we picked up this view on the taxonomy is that it clearly describes position of discrete event systems (DES) and DES in a class of systems with which we deal primarily in this book. The choice of notation and examples in this chapter is also based on (Cassandras & Lafortune, 2008).

It should be noted that when we speak about taxonomy of systems we in fact mean types of models that are the most suitable for given types of systems. And from the basic types of models, this taxonomy is primary related to mathematical models. How these models can be obtained and what is their form is explained in the next two sections.

II.1 Input-Output Modelling

In the process of modelling we try to create something (a device) that mimics the behaviour of the original system. To make a simulation or formal analysis of the model possible, we need to describe the model in an exact, unambiguous, way. That is, by mathematical means. When observing a real system we can identify parameters whose values can be altered and other parameters whose values change when some of the parameters from the first group are altered. For example, consider an automobile. The positions of its clutch, brake and throttle pedals, selected gear, the position of its steering wheel and amount of remaining fuel belong to the first group while rpm (revolutions per minute) of its engine, its speed and direction belong to the second. When creating a mathematical model we identify the parameters from the first group with ***input variables*** and the ones from the second group with ***output variables***. Both of these parameters have to be measurable, because our task is to define a relationship between them. Their values change as time passes, so they can be defined as time functions. Assume that we have p input and m output variables. Then the input variables form the set (1) and they can be written as a column vector (2). The output variables are members of (3) and their vector form is (4). The symbol T in upper index in (2) and (4) stands for vector or matrix transposition. The vector $\vec{\mathrm{u}}(\mathrm{t})$ can be simply called an ***input*** and the vector $\vec{\mathrm{y}}(\mathrm{t})$ an ***output***.

$$\{u_1(t), \dots, u_p(t)\} \quad (1)$$

$$\vec{u}(t) = \left[u_1(t), \dots, u_p(t)\right]^T \quad (2)$$

$$\{y_1(t), \dots, y_m(t)\} \quad (3)$$

$$\vec{y}(t) = [y_1(t), \dots, y_m(t)]^T \quad (4)$$

The variable t is called the ***time variable*** and its values are from an interval $< t_0, t_f >$. The interval represents a period of time during which we study behaviour of the system, i.e. we modify input variables and measure output variables. We can also have variables that don't belong to the input of output ones. These are called ***suppressed output variables*** (Cassandras & Lafortune, 2008).

On the basis of an observation of the real system and adjustment and measuring of its parameters we should be able to define a mathematical relationship between the input and output variables. Suppose that we can establish this relationship in the form of m functions

$$g_1, \dots, g_m$$

defined as

$$y_i(t) = g_i\left(u_1(t), \dots, u_p(t)\right),$$

where $1 \leq i \leq \mathrm{m}$. Then we obtain a mathematical model in the form (5).

$$\vec{y}(t) = \vec{g}(\vec{u}(t)) = [g_1(u_1(t), \dots, u_p(t)) \dots g_m(u_1(t), \dots, u_p(t))]^T \qquad (5)$$

To shorten the inscription we usually write the relationship between input and output as (6). In general, $\vec{\mathrm{g}}$ can explicitly depend on t, which we designate as (7).

$$\vec{y} = \vec{g}(\vec{u}) \qquad (6)$$

$$\vec{y} = \vec{g}(\vec{u}, t) \qquad (7)$$

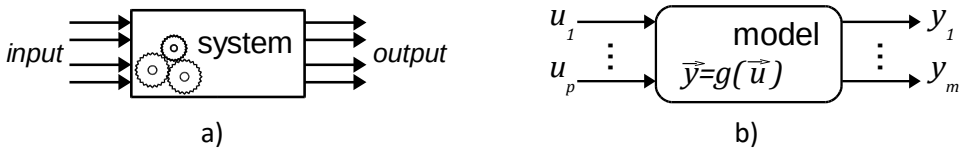


Figure 1. A system (a) and its model (b).

Usually, there is more than one way of creating a model of a given system. Depending on the purpose of the model and required level of detail we can pick up different output and input variables and the function $\vec{\mathrm{g}}$ will also be different. Even on the same level of detail and using the same mathematical apparatus we can create several models that will differ in selection of input and output variables. This is illustrated by the following example.

Example 1. *Current divider circuit.*

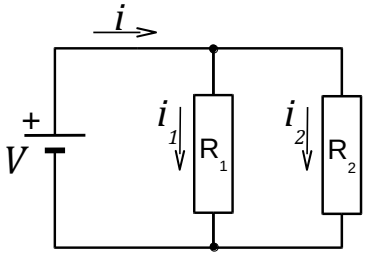


Figure 2. A simple current divider circuit.

Figure 2 shows a simple electric circuit with a DC power supply delivering voltage V and two resistors with resistances R_1 and R_2, connected in parallel. A simple mathematical model of this circuit can be built on the basis of Kirchhoff's current law and Ohm's law. Application of these laws results in equations (8) and (9).

$$i = i_1 + i_2 \qquad (8)$$

$$i_1 = \frac{V}{R_1}, \quad i_2 = \frac{V}{R_2} \qquad (9)$$

For this system we can build several models, depending on what parameters we consider fixed, what we would like to control and what we would like to measure. For example

- The power supply voltage V is fixed, we can adjust R_1 and R_2 to control i_1 and i_2. We get a model with two input variables u_1 and u_2 and two output variables y_1 and y_2:

$$u_1(t) = R_1, \qquad u_2(t) = R_2, \quad y_1(t) = i_1 = \frac{V}{R_1}, \quad y_2(t) = i_2 = \frac{V}{R_2}$$

- The power supply voltage V and the resistance R_1 are fixed, we can adjust R_2 and we are interested in measuring i. Now we will have a model with one input variable u_1 and one output variable y_1:

$$u_1(t) = R_1, \quad y_1(t) = i = V\frac{R_1 + R_2}{R_1 R_2}$$

□

II.2 State Space Modelling

When creating a model of a system it is not always possible to define relation between its input $\vec{u}$ and output $\vec{y}$ in such a way that $\vec{y}(t)$ will depend only on $\vec{u}(t)$. Sometimes an additional information is necessary – an information about previous inputs and outputs of the system (i.e. about $\vec{u}(t')$ and $\vec{y}(t')$ for some $t' < t$). This information forms a *state of the system*, which we can more precisely define as (Cassandras & Lafortune, 2008):

State of a system at time t_0 *is the information required at t_0 such that $\vec{y}(t)$, for all $t \geq t_0$, is uniquely determined from this information and from $\vec{u}(t)$, $t \geq t_0$.*

Like in the case of the input and output, we can define the state as a vector $\vec{x}$ of n state variables x_1 to x_n.

$$\vec{x}(t) = [x_1(t), \dots, x_n(t)]^T \qquad (10)$$

The set of all possible values the state $\vec{x}$ of given system may take is the *state space of the system*, and we will denote it as X. Normally, X is ***finite-dimensional***, i.e. n is a final number, but there are cases where it is ***infinite-dimensional***.

The state $\vec{x}(t)$ is determined by the set of ***state equations***. They compute $\vec{x}(t)$ on the basis of the input $\vec{u}(t)$ an the ***initial state*** $\vec{x}(t_0)$, $t_0 \leq t$. Their form may vary, but in the systems and control theory they are mostly differential equations (Cassandras & Lafortune, 2008) of the form (11).

$$\dot{\vec{x}}(t) = \vec{f}(\vec{x}(t), \vec{u}(t), t) \qquad (11)$$

$$\vec{x}(t_0) = x_0 \qquad (12)$$

$$\vec{y}(t) = \vec{g}(\vec{x}(t), \vec{u}(t), t) \qquad (13)$$

A ***state space model*** of a system will then consist of the state equations (11), initial conditions (12) and output equations (13). The relationship between input, state and output, expressed by them, is what we usually call *dynamics* of a system. Assuming that input and output vectors are in the form (2) and (4) and state vector in the form (11) , the model (11) - (13) actually contains n *state equations* with initial conditions:

$$\dot{x}_i(t) = f_i\big(x_1(t), \dots, x_n(t), u_1(t), \dots, u_p(t), t\big), \qquad x_i(t_0) = x_{i0}, \qquad 1 \le i \le n$$

and m output equations:

$$y_j(t) = g_j\big(x_1(t), \dots, x_n(t), u_1(t), \dots, u_p(t), t\big), \qquad 1 \le j \le m$$

As in the case of the input-output modelling there are several ways of defining state-space model of the same system. The concept of the state-space model is illustrated by the following example.

Example 2. *Spring-mass system*

Consider a spring, characterized by the spring constant k and with an object of mass m attached to it. Together they form a spring-mass system as depicted in Figure 3. Supposing that the spring is not de-

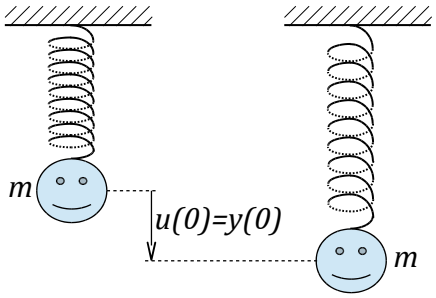


Figure 3. Spring-mass system.

formed (that is stretched or compressed beyond its elastic limit), the system behaves according to the Hooke's law (14), that states that the force F with which the spring pushes back is linearly proportional to the displacement y from its equilibrium length. Since force equals to mass times acceleration and acceleration is the second derivative of the displacement, we end up with (15) with initial conditions (16) and (17). We assume that $t_0 = 0$.

$$F = -ky \qquad (14)$$
$$m\ddot{y} = -ky \qquad (15)$$
$$y(0) = u_0 \qquad (16)$$
$$\dot{y}(0) = 0 \qquad (17)$$

The conditions state that at the beginning the object is pushed by the distance $|u_0|$ downwards (16) and its velocity is zero (17). As in the previous example, we can define several models of the system with different inputs and outputs. Probably the most common case is that we would like to set the initial displacement u_0 and observe the actual displacement y of the object over time. In this case the input function will be defined as

$$u(t) = \begin{cases} u_0 & t = 0 \\ 0 & t > 0 \end{cases} \quad (18)$$

and the output function $y(t)$ is the solution of (15). Because the differential equation (15) contains the second derivative of y and there are no other variables we will need two state variables, $x_1(t)$, $x_2(t)$ that will form two state equations (19), (20), initial conditions (21) and output equation (22). The variable $x_1(t)$ is the displacement of the object and $x_2(t)$ its speed.

$$\dot{x_1}(t) = x_2(t) \quad (19)$$

$$\dot{x_2}(t) = -\frac{k}{m}x_1(t) \quad (20)$$

$$x_1(0) = u_0, \;\; x_2(0) = 0 \quad (21)$$

$$y(t) = x_1(t) \quad (22)$$

The model can be also written in the matrix form (23) to (25).

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (23)$$

$$\begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} u_0 \\ 0 \end{bmatrix} \quad (24)$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (25)$$

□

II.2.1 Derivative

In state space modelling we use derivatives quite often, so it's the right place to recall definition of this term. The ***derivative*** is a measure of how a function changes as its input changes. The process of finding a derivative of a function is called ***differentiation*** and the reverse process is *antidifferentiation*, which is the same as ***integration***.

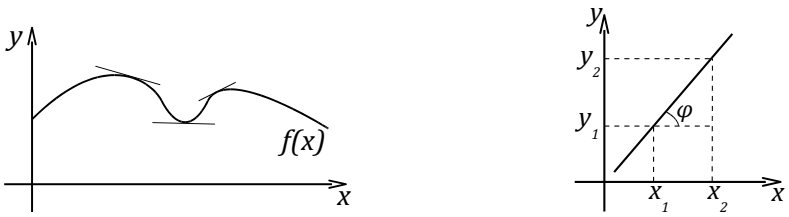


Figure 4. Tangent lines for function f(x) and slope of a line (b).

Suppose that we have a real function f of a single variable (26).

$$y = f(x) \quad (26)$$

Then its derivative at a point a is the slope of the tangent line to the graph of f at the point a. The concepts of tangent line and slope are visualized in Figure 4. The slope, or gradient, of a line is a

number that describes both the direction and the steepness of the line. If the line is as in Figure 4 then its slope is defined as (27).

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = tan(\varphi) \qquad (27)$$

The expression (27) is called Newton's difference quotient. As the tangent line can be different in each point of the function, the derivative is the slope for Δx approaching 0 and is defined as:

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h} \qquad (28)$$

The derivative in (28) is called the *first derivative of* f. We can also have the *second derivative* of f, denoted f'', which is the first derivative of f'. Derivatives of higher degree are defined similarly, third derivative is denoted as f''' and the n-th as $f^{(n)}$. Here we used Lagrange's notation for derivatives, there are also three other notations:

- Leibniz's notation. The first derivative is denoted as $\frac{dy}{dx}$, $\frac{df}{dx}(x)$ or $\frac{d}{dx}f(x)$ and the n-th as $\frac{d^n y}{dx^n}$, $\frac{d^n f}{dx^n}(x)$ or $\frac{d^n}{dx^n}f(x)$
- Euler's notation. The first derivative is $D_x y$ or $D_x f(x)$ the n-th is $D_x^n y$ or $D_x^n f(x)$.
- Newton's notation is used exclusively for time derivatives, that is for derivatives of $y = f(t)$). For the first derivative we have $\dot{y}$ and for the second $\ddot{y}$. The first derivative is defined as

$$\dot{y}(t) = \lim_{\tau \to 0} \frac{f(t+\tau) - f(t)}{\tau}$$

If the function (26) has a derivative at a, i.e. $f'(a)$, then we say that f is *differentiable* at a. To be differentiable at a, the function f also have to be continuous at a, but this is only a mandatory condition. For example, the absolute value function $y = |x|$ is continuous at $x = a$, but it doesn't have a derivative there.

II.3 Taxonomy

After establishing more specifically how (mathematical) models look like we can proceed to the taxonomy presented in (Cassandras & Lafortune, 2008). The taxonomy is based on several criteria and each criterion divides systems into two disjoint classes.

II.3.1 Static vs. Dynamic

The first criterion is whether output of the system in question depends only on its input at the same time instant or also on inputs at some previous time instants.

The systems that fulfil this criterion, i.e. the system where $\vec{y}(t)$ depends only on $\vec{u}(t)$, are called ***static systems***. In their case the input-output models are sufficient as the state equation will always be in the form $\dot{\vec{x}}(t) = 0$. This means that the state is fixed and output depends only on input values in the same moment of time. The electric circuit from Example 1 is a static system. Other group of static systems are, for example, combinational logic circuits.

In ***dynamic systems*** or models $\vec{y}(t)$ depends also on $\vec{u}(t')$, $t' < t$ (except of $t' = t_0$). Here a model of the state-space type is required. The spring-mass system from Example 2 is a dynamic system. Other example are sequential logic circuits.

II.3.1.1 Sample Paths

In dynamic systems for each particular vector of functions $\overrightarrow{u^i}(t) = \left[u_1^i(t), \dots, u_p^i(t)\right]^T$ that are input variables we get a particular vector of functions $\overrightarrow{x^i}(t) = \left[x_1^i(t), \dots, x_n^i(t)\right]$ that are state variables. Each of the functions $x_1^i(t)$ to $x_n^i(t)$ is then called a ***sample path*** or a *state trajectory*. Of course, for different input variables we get different state trajectories.

Sample paths can be also described as curves in n-dimensional space. This representation is in more detail treated in (Cassandras & Lafortune, 2008). Examples of graphical representation of sample paths can be found in Figure 6 and Figure 8.

II.3.2 Time-invariant vs. Time-varying

Time-invariant systems are systems where the output is always the same when the same input is applied. So, when at some time instant t the input is $\vec{u}(t) = \vec{a}$ and the output is $\vec{y}(t) = \overrightarrow{r_a}$ then when we will have the same input $\vec{a}$ at any different time instant t' (i.e. $\vec{u}(t') = \vec{a}$) then the output will always be the same as at t (i.e. $\vec{y}(t') = \overrightarrow{r_a}$). State and output of time-invariant systems don't explicitly depend on time therefore the corresponding equations have the form (29).

$$\begin{aligned} \dot{\vec{x}}(t) &= \vec{f}\left(\vec{x}(t), \vec{u}(t)\right) \\ \vec{y}(t) &= \vec{g}\left(\vec{x}(t), \vec{u}(t)\right) \end{aligned} \qquad (29)$$

In ***time-varying systems*** we can get different outputs for the same inputs at different time moments. This means that their state and output explicitly depend on time. The corresponding equations have the form (30).

$$\begin{aligned} \dot{\vec{\mathrm{x}}}(t) &= \vec{\mathrm{f}}(\vec{\mathrm{x}}(t), \vec{u}(t), \mathrm{t}) \\ \vec{y}(t) &= \vec{g}(\vec{\mathrm{x}}(t), \vec{u}(t), \mathrm{t}) \end{aligned} \qquad (30)$$

The spring-mass system from Example 2 is a time-invariant system. But we can create a time-varying system from it by assuming that the mass m changes as time passes. For example, the object attached to the spring can be an open bucket full of a quickly evaporating liquid. Then the mass m decreases with time and when we push the object by the same distance again and release it we will observe different behaviour (output) as for the first time. However, we can make such a system time-invariant again by adding a parameter on which the evaporation rate of the liquid depends to the input variables.

II.3.3 Linear vs. Nonlinear

A ***linear function*** is a function that preserves *vector addition* (also called *superposition*) and *scalar multiplication* (*homogeneity of degree 1*). For example, let's have a function h that maps k-dimensinal real vectors to k-dimensinal real vectors (31), real vectors $\vec{v}, \vec{w} \in \mathbb{R}^{\mathrm{k}}$ and real constants $c, c_1, c_2 \in \mathbb{R}$. Then h preserves vector addition if (32) holds and scalar multiplication if (33) holds. To sum it up, h is linear when (34) is true.

$$h: \mathbb{R}^{\mathrm{k}} \to \mathbb{R}^{\mathrm{k}} \quad (31)$$

$$\vec{h}(\vec{v} + \vec{w}) = \vec{h}(\vec{v}) + \vec{h}(\vec{w}) \quad (32)$$

$$\vec{h}(c.\vec{v}) = \mathrm{c}.\vec{h}(\vec{v}) \quad (33)$$

$$\vec{h}(c_1.\vec{v} + c_2.\vec{w}) = c_1.\vec{h}(\vec{v}) + c_2.\vec{h}(\vec{w}) \quad (34)$$

The linearity property is defined in the same way for the functions $\vec{f}$ and $\vec{g}$ from our state-space model (11) - (13) despite of the fact that these functions map vectors of functions to vectors of functions.

Then a ***linear system*** is a system where both $\vec{\mathrm{f}}$ and $\vec{g}$ are linear. Assuming that $\vec{u}$, $\vec{x}$ and $\vec{y}$ are as defined in (2), (4) and (10) the state model of a linear system can be written in the form (35).

$$\begin{aligned} \dot{\vec{x}}(t) &= \mathrm{A}(t)\vec{x}(t) + \mathrm{B}(t)\vec{u}(t) \\ \vec{y}(t) &= \mathrm{C}(t)\vec{x}(t) + \mathrm{D}(t)\vec{u}(t) \end{aligned} \quad (35)$$

where $\mathrm{A}(t)$ is an $n \times n$ matrix, $\mathrm{B}(t)$ is an $n \times p$ matrix, $\mathrm{C}(t)$ is an $m \times n$ matrix and $\mathrm{D}(t)$ is an $m \times p$ matrix. If the system is also time-invariant then all four matrices are constant and we can rewrite (35) as (36).

$$\begin{aligned} \dot{\vec{x}}(t) &= \mathrm{A}\vec{x}(t) + \mathrm{B}\vec{u}(t) \\ \vec{y}(t) &= \mathrm{C}\vec{x}(t) + \mathrm{D}\vec{u}(t) \end{aligned} \quad (36)$$

The entries stored in these matrices are usually called *model parameters* and it is often the task of simulation to find the most suitable values for them.

The preservation of the superposition in linear systems means that if for a concrete stimulus, that is for some vector $\vec{a}$ of values of input variables, we get a response (vector of values of output variables) $\overrightarrow{r_a}$ and for some $\vec{b}$ we get $\overrightarrow{r_b}$ then for an input $\vec{c}$ that is the superposition of these two stimuli (i.e. $\vec{c} = \vec{a} + \vec{b}$) we get the response $\overrightarrow{r_c}$ that is the superposition of the two original responses (i.e. $\overrightarrow{r_c} = \overrightarrow{r_a} + \overrightarrow{r_b}$). The preservation of the scalar multiplication means that when we multiply a stimulus by c then the corresponding response is also multiplied by c.

In a ***nonlinear system*** $\vec{\mathrm{f}}$ or $\vec{g}$ or both are not linear. An example of nonlinear system follows. The systems in previous examples are linear.

Example 3. *Flow system*

Suppose we have a tank with some fluid (Figure 5). The fluid can flow into the tank and out of the tank. The flow in is defined by the function $fl_i(t)$ and the flow out by the function $fl_o(t)$. Both flows can be regulated by corresponding input and output valves. The capacity of the tank is represented by its maximum fluid level Cp. The function $x_1(t)$ is an actual level of fluid in the tank and its value is from the interval $< 0, Cp >$.

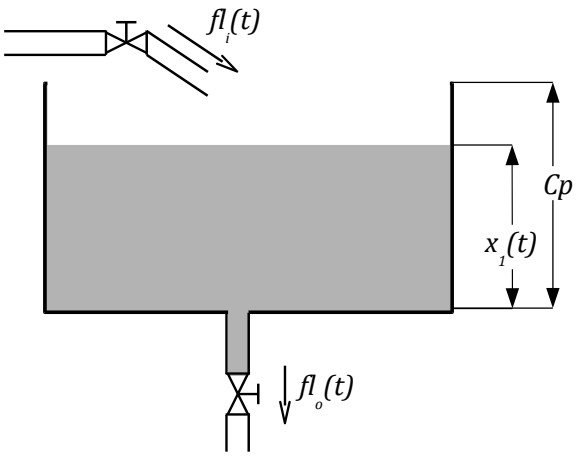


Figure 5. A simple flow system.

When building a model of this system we can identify the flow in and out with input variables $u_1(t)$ and $u_2(t)$. The state variable will be the actual flow and we will keep its original name - $x_1(t)$. The output of the model will be identical to its state. This model allows us to control both flows and observe an actual level of fluid in the tank.

The state space model of the system will then be as follows:

$$u_1(t) = fl_i(t), \qquad u_2(t) = fl_o(t) \tag{37}$$

$$\dot{x_1}(t) = \begin{cases} 0 & (x_1(t) = 0 \ \wedge fl_i(t) \leq fl_o(t)) \ \vee \\ & \quad \vee \ (x_1(t) = Cp \ \wedge fl_i(t) \geq fl_o(t)) \\ \mathrm{fl_i}(t) - fl_o(t) & otherwise \end{cases} \tag{38}$$

$$x_1(0) = 0 \tag{39}$$

$$y_1(t) = x_1(t) \tag{40}$$

The input variables are defined by (37), the state equation is (38) and the output equation is (40). The initial condition (39) states that the tank is empty at the beginning. For the sake of simplicity we decided to define the flow in and out as a change of the fluid level in the tank.

What we can see here is that the definition of the state equation (38) is not that "smooth" as in the previous examples. There are two cases, in the first one $\dot{x_1}(t) = 0$ and in the second one $\dot{x_1}(t) = \mathrm{fl_i}(t) - fl_o(t)$. The first case means that there will not be any decrease of the fluid level when the tank is empty, despite the output valve being open. And, similarly, there will be no increase in the fluid level after Cp is reached, despite the input valve being open. The constraints for $x_1(t) = 0$ and $x_1(t) = Cp$ cause discontinuities in $\dot{x_1}(t)$ and also non-linearity of the system.

To illustrate the nature of non-linearity of this system in more detail, assume that the fluid level $x_1(t)$ is measured in millimetres and the volume and dimensions of the tank are such that the increase or decrease of the fluid volume by $1\ l$ will change the fluid level by $1\ mm$ and $Cp = 250\ mm$ (in fact $250\ l$). So, we can define input and output flow in millimetres per second (mm/sec). Also assume that for each t, $t \in < t_0, t_1 >$, $t_0 = 0\ sec$, $t_1 = 75\ sec$ it holds that

$$u_1^1(t) = fl_i^1(t) = 5\ mm/sec \tag{41}$$

$$u_2^1(t) = fl_o^1(t) = 0\ mm/sec$$

so 5 litres of the fluid will flow into the tank each second, increasing the level by $5\,mm$ until $t_a = 50\ sec$. At t_a the tank will be full and will remain full until t_1. Now observe the same system with values of both input variables doubled. That is with

$$\begin{aligned} u_1^2(t) &= fl_i^1(t) = 2 \cdot 5 = 10\ \ mm/sec \\ u_2^2(t) &= fl_o^1(t) = 2 \cdot 0 = 0\ mm/sec \end{aligned} \qquad (42)$$

If the system will be linear, the speed of the fluid level increase will always be double than in the first case. However, this is not true in the interval $< t_b, t_a)$, where $t_b = 25\ sec$, because the maximum level is already reached at t_b and the level will not increase from this moment. It is true again in $< t_a, t_2 >$ as here the fluid level is fixed (Cp) in both cases. The sample pats for both cases are shown in Figure 6. The state variable (function) $x_1^1(t)$ is for the first case (41) and $x_1^2(t)$ for the second case (42). The situation will be similar if we multiply by other scalars or add different values of input parameters.

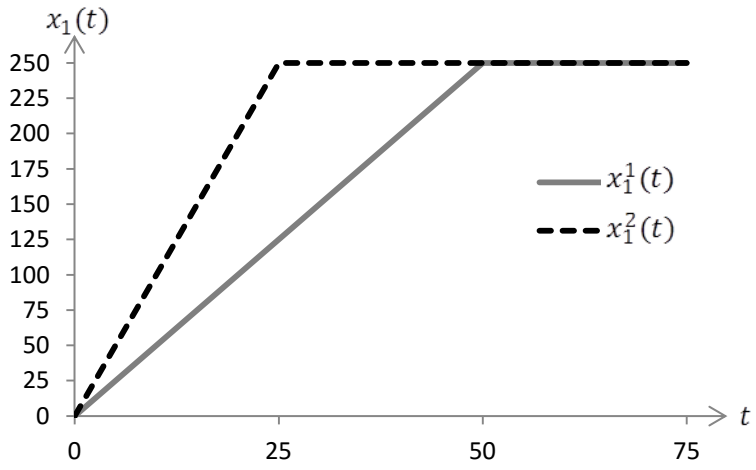


Figure 6. Sample paths for the flow system.

□

II.3.4 Continuous-State vs. Discrete-State

In section II.2 we defined the state space X of a system as a set of all possible values its state $\vec{x}$ may take. And according to the nature of the state space we can divide systems (models) to *continuous-state* and *discrete-state*.

A ***continuous-state model*** is a model where state variables are ***continuous variables***, i.e. they can generally take on any real (or complex) value. A consequence of this is that even if values of state variables are from some bounded non-empty interval, there are infinitely many of them and they form an *uncountable set*. An *uncountable set* is a set with the cardinality higher than the cardinality of the set of natural numbers. State equations in these models are usually differential. Continuous-state dynamic systems has been presented in Example 2 and Example 3.

In a ***discrete-state model*** domains of state variables are discrete sets. They can be finite, for example a set of basic colours or infinite but countable (also called *countably infinite*) such as the set of natural numbers or the set of integers. The state in these systems doesn't change continuously but suddenly, from one discrete value to another and the mechanism of state transition can be usually described in the form of logical statements such as "if some event e occurs and the system is in the state x, then change the state to x'". This makes them simpler to visualise but the mathematical apparatus needed to formally express the state equations may be considerably more complex as in the case of the continuous-state models (Cassandras & Lafortune, 2008).

There can also be so-called ***hybrid systems*** where some state variables are continuous and some are discrete.

A distinction between continuous and discrete state variables is illustrated in Example 4 while a complete discrete-state model is presented in Example 5.

Example 4. *Fluid tank vs. fluid storage*

In the flow system from Example 3 the state variable $x_1(t)$ is continuous. If the capacity of the system is $250\ l$ and the maximal fluid level is $Cp = 250\ mm$ then $x_1(t)$ can take any real value from the closed bounded interval $< 0, 250 >$. This means infinitely many states. And uncountable number of states, too, as we can find infinitely many real numbers between each two distinct real numbers, no matter how close they are one to each other.

Now, imagine a fluid storage system of the same capacity, $250\ l$, but in this case the fluid is stored in sealed containers, each containing $25\ l$ of the fluid. Instead of manipulating valves here one can add or remove a container to or from the storage area. And the state variable can only take values from the set $\{0,25,50,75,100,125,150,175,200,225,250\}$.

□

Example 5. *Doctor's waiting room*

A waiting room, i.e. one that is a part of a doctor's office, is another example of a natural discrete-state system. The state variable is a number of patients currently present in the room and for input variables we can choose functions that determine how patients arrive and leave the room (Figure 7).

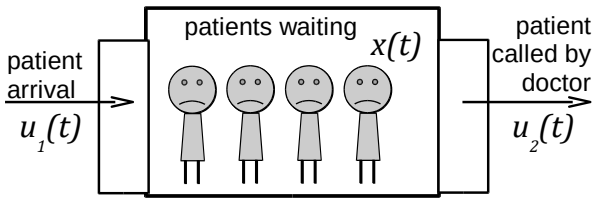


Figure 7. A system describing arrival and departure of patients to and from a doctor's waiting room

For the sake of simplicity we assume that only one patient can arrive or leave at once and no arrival and departure occur simultaneously. We also assume that the capacity of the waiting room is not limited. Then the input function for the arrival can be defined as in (43) and for the departure as in (44).

$$u_1(t) = \begin{cases} 1 & if\ a\ patient\ arrives\ at\ time\ t \\ 0 & otherwise \end{cases} \quad (43)$$

$$u_2(t) = \begin{cases} 1 & if\ a\ patient\ is\ called\ at\ time\ t \\ 0 & otherwise \end{cases} \quad (44)$$

Considering the assumptions we made and the definitions of the input functions we can have three different situations:

1. A patient arrives to the waiting room: $u_1(t) = 1$ and $u_2(t) = 0$. The value of the state variable is incremented by one.
2. A patient leaves the waiting room, because he is called by the doctor: $u_1(t) = 0$ and $u_2(t) = 1$. The value of the state variable is decremented by one.
3. No one arrives to or leaves the room: $u_1(t) = 0$ and $u_2(t) = 0$. The value of the state variable is unchanged.

In this model the state is changing suddenly from one discrete value to another. Because of these discontinuities we cannot use differential equations to define state equation of the model. Instead of it we write the equation in the form (45), which defines how the value of the state variable in the next time moment $t + 1$ depends on the values in the given time moment t and values of inputs at t.

$$x(t+1) = \begin{cases} x(t) + 1 & u_1(t) = 1 \ \wedge \ u_2(t) = 0 \\ x(t) - 1 & u_1(t) = 0 \ \wedge \ u_2(t) = 1 \wedge x(t) > 0 \\ x(t) & otherwise \end{cases} \quad (45)$$

A sample path for this system can be seen in Figure 8. Here we can see that the state remains unchanged for the most of the time. It changes only at time moments when an event of patient arrival or departure occurs.

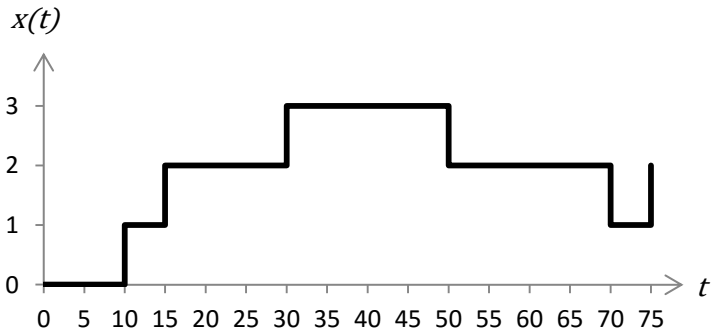


Figure 8. Sample path for the Doctor's waiting room.

□

II.3.5 Continuous-time vs. Discrete-time

This is similar to the previous criterion, but now it is about the time. ***Continuous-time systems*** are systems where the time is a continuous variable, so all input, state and output variables are defined for all possible values of time (from given interval). Thinking about the time as a continuous variable

is natural. In fact, each real system is a continuous-time system: its inputs and outputs can be observed at any time instant (in which the given system exists). It is when we create and simulate models that we come to the discrete-time systems. Continuous-time systems are usually described by differential equations and all systems in the examples above, except Example 5, are of this type.

In ***discrete-time systems*** the time is a ***discrete variable***. So, if the time is from some interval (e.g. from *2* to *4*) then it has finite number of values (e.g. *2*, *2.5*, *3*, *3.5*, *4*). And input, state and output variables of such system are defined at these time instants only. We said that it is natural to consider time a continuous variable, so the question is why we deal with systems with discrete time. But, in fact, there are several cases where such systems are appropriate:

1. *Output and state variables can change only at exactly defined time instants.* This is, for example, the case of digital circuits, which operate according to some internal clock. So, any change in them can only happen when these clock ticks. Between the ticks the state and output remain unchanged, so there is no reason to define corresponding variables for any time moment.
2. *A model of a system is based on a finite set of data, recorded at certain time moments (usually at regular intervals).* In some cases these data can be approximated to continuous functions. If not, we consider that they change only at the instants when they were measured. We can say that a system is discrete-time because of our inability to handle it as a continuous-time system.

It should be noted that continuous-time systems usually become de-facto discrete-time systems when simulated on digital computers. This is because in most cases we rely on numerical methods for differential equations calculation and these methods compute values for discrete time instants only. Because of this it is reasonable to think about using a discrete-time model instead of a continuous-time model for given modelling and simulation task.

Time values in discrete-time systems form a sequence

$$t_0, t_1, \dots, t_k, \dots, \qquad (46)$$

where

$$\forall i (i \geq 0): t_i < t_{i+1}.$$

The distance between two subsequent values is constant and is usually called *sampling interval* (T):

$$\forall i (i \geq 0): t_{i+1} = t_i + T.$$

Instead of differential equations a state space model of a discrete-time system uses so-called ***difference equations***, where the real variable t is replaced by an integer variable k. The variable k defines the index of given value in the sequence (46), e.g. $k = 3$ for t_3. We can also say that k is the number of intervals of the length T that passed since the initial time point t_0. The exact form of the state space model is as follows:

$$\vec{x}(k+1) = \vec{f}(\vec{x}(k), \vec{u}(k), k) \qquad (47)$$
$$\vec{x}(0) = x_0 \qquad (48)$$

$$\vec{y}(k) = \vec{g}(\vec{x}(k), \vec{u}(k), k) \qquad (49)$$

An example of a discrete-time sample path can be seen in Figure 9. It represents the state variable $x_1^1(t)$ from Figure 6, but here the time line is a sequence from 0 to 70 with a sampling interval $T = 10$.

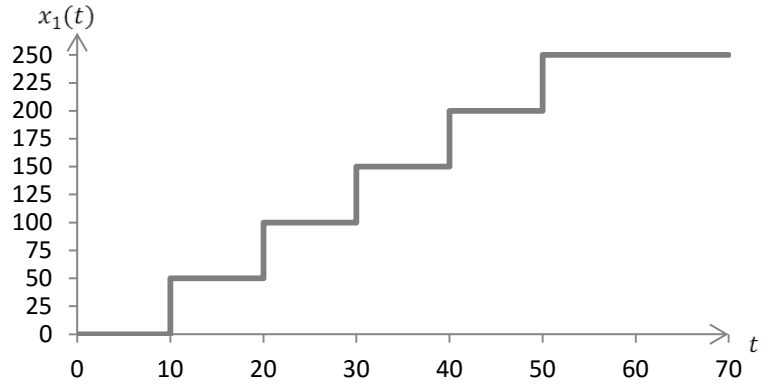


Figure 9. Discrete-time sample path for the flow system.

II.3.6 Time-driven vs. Event-driven and Discrete Event Systems

We distinguish between these two types of systems with respect to what (we think) is causing a system to change its state.

In ***time-driven systems*** the driving force is the time itself, the state continuously changes as time changes. Continuous-state systems, both continuous-time and discrete-time, belong here, because there are infinitely many values for state variables and we can observe different values no matter how small is the interval between observations. All dynamic systems introduced in the examples above are time-driven.

In ***event-driven systems*** the cause of a state change is an occurrence of some asynchronously generated discrete *event* and the state changes instantaneously, at the moment of the event occurrence, from one value to another. So, all event-driven systems are discrete-state, too. The term *asynchronously generated* means that these events can occur at any time moment, not only at ticks of a clock. Cassandras and Lafortune (2008) draw a nice analogy between event-driven state transitions and interrupts in computer systems: "While many of the functions in a computer are synchronized by a clock, and are therefore time-driven, operating systems are designed to also respond to asynchronous calls that may occur at any time. For instance, an external user request or a timeout message may take place as a result of specific events, but completely independent of the computer clock."

The doctor's waiting room from Example 5 can be regarded as an event-driven system, because the state (the number of people in the waiting room) changes only when one of the two events, the patient arrival and the patient call, occurs. But in Example 5 we defined it as a time-driven system as we assumed that these events occur only at clock ticks. If we want to describe it as event-driven system we need a formalism different from the difference equation (45). We need to relate state

changes to events, not to time, to express relations like “if a patient arrives, the number of patients in the waiting room is increased by one”. An appropriate language to describe event-driven systems are *Petri nets*, which utilisation for modelling and simulation of these systems is treated is chapter VI, including several examples.

For the first time in our taxonomy we have some form of exclusivity here: all continuous systems are time driven, while event-driven systems are a subclass of discrete-time systems. This is not the case of the previous criteria: continuous-state systems can be continuous-time or discrete-time and the same is true for discrete-state systems.

In the previous section we mentioned that simulation models of continuous-time systems are in fact discrete-time systems. And we have a similar situation here: To simulate an event-driven system on a digital computer we have to find a clock precise enough to make every event occurring at some tick of the clock. For example, in the case of timed Coloured Petri nets (section VI.5) this clock is represented by so-called *simulated time* and events occur only at its time instants.

An important class of event-driven systems are ***discrete-event systems*** (*DES*), which are discrete-state event-driven systems. They are in detail treated in chapters IV to VI of this book.

II.3.7 Stochastic vs. Deterministic

A system is considered ***stochastic*** if one or more of its output variables are ***random variables***. A *random variable* is a variable, which value cannot be specified exactly, only a set of its possible values and a probability of each of these values are known. The mathematical function, which describes these possible values and their probabilities, is called a *probability distribution*. A random variable can be discrete or continuous. Systems with no random output variables are called ***deterministic***.

All the systems presented in the examples above can be made stochastic by making their input variables random. Concrete examples of stochastic DES can be found in sections VI.6 and VI.7.

III Logic Simulation

The *logic simulation* simulates operation of logic systems. As logic systems are also called logic circuits or digital circuits there are also alternate names for the logic simulation: digital simulation and digital circuit simulation. Here signals are usually represented by discrete bands of analog levels. A short overview of logic simulation, presented in this chapter, is based on (Wang et.al, 2009) and (Neuschl et al, 1988), where an interested reader can find more information on the topic.

The logic simulation is typically used in design and development of new circuits to validate and verify the design at different levels of abstraction and in diagnostics to check completeness of tests. The completeness is checked by running tests on simulated circuits with and without known errors.

Level	Systems	Elements	Signal units
Electronic System Level (ESL)	Computer systems	CPU, memory, I/O devices, channels,...	word blocks
Register transfer level (RTL)	CPU, ALU, memory,…	register, coder, decoder	words *(sequences of bites)*
Gate level (GL) (Logic circuits level)	register, coder, counter	gates (and, or,…), flip-flops(D, JK,…)	bites
Electronic circuit level (Transistor level)	gates, flip-flops	transistor, diode, capacitor, …	voltage levels
Physical level	transistor, diode, resistor	diffusion areas, contacts	-(physical dimensions)

Table 1. Levels of logic circuits design.

Logic systems can be modelled at several *levels of abstraction* (Table 1). From these levels only a simulation at RTL and GL are usually considered as the logic simulation. At the *transistor level* circuits are modelled as continuous-state systems, where input, output and state equations are defined using electrical laws, such as Ohm's or Kirchhoff's laws. And the *Electronic System Level* provides the least detailed view of the circuits, so usually the discrete-event models (e.g. queuing systems) are used here. They are often stochastic, because of subsystems described by random distributions. For example, a memory unit can be described as a timed automaton or a schematic at RTL or GL. At ESL it will be an element, characterized by its capacity and random distributions defining time needed to write and read the memory. This randomness is not because there are no corresponding deterministic models but because there is no “room” for such details at ESL.

The distinction between *RTL* and the *gate level* is not very clear. We often have mixed designs, where one element is a gate and another one is an RTL component such as a memory register or a multiplexer. This means that a simulation at these two levels at once has to be possible, too. Models

of elements can be *mathematical*, especially at the gate level, where Boolean algebra and finite automata are used, or *behavioural*, often described in an appropriate language, such as *VHDL* (Very-high-speed integrated circuits Hardware Description Language). Logic systems *modelled* at these two levels are discrete-state and can be time-driven or event-driven.

III.1 Modelling of Digital Circuits

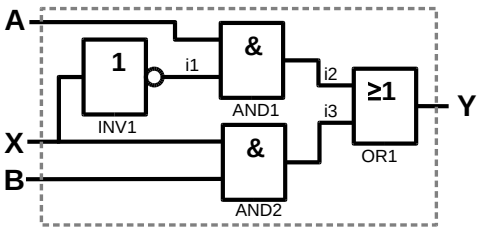
To model a logic circuit we need to pick up an appropriate modelling language and decide how to represent signals (variables) and delays. When using simulation to evaluate diagnostic tests we also have to consider means by which faults will be represented.

III.1.1 Modelling languages

In general, the languages used to model digital circuits can be divided into two groups:

1. *Languages describing only structure.* These languages use fixed library of elements (e.g. gates, flip-flops, registers, etc.). Some of them also offer parametrisation, which allows to, for example, choose number of inputs or delays of the elements.
2. *Languages describing structure and behaviour.* Here new elements can be specified by defining their inputs, outputs and behaviour in given language. Languages can be general-purpose programming languages or hardware description languages, such as VHDL or Verilog.

Languages from both groups can also support *hierarchical design*, where new elements are defined as a composition of existing ones. When a language from the first group has this feature, its possibilities become very close or even equal to the languages from the second group. Several models of the same circuit are illustrated in Figure 10. The schematic in Figure 10 a) can be regarded a definition of a new element (therefore the grey dashed rectangle), provided the hierarchical design is supported.



a)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux_2to1 is
 Port ( A : in   STD_LOGIC;
        B : in   STD_LOGIC;
        X : in   STD_LOGIC;
        Y : out  STD_LOGIC );
end mux_2to1;

architecture Behavioral of mux_2to1 is
begin
    Y <= A when (X = '0') else B;
end Behavioral;
```

b)

$$Y = (A \cdot \bar{X}) + (\mathrm{B} \cdot \mathrm{X})$$

c)

Figure 10. Three models of a 2-to-1 1 bit multiplexer: A schematic in a language describing only structure (a), a VHDL code (b) and a mathematical input-output model in the form of a Boolean function (c)

III.1.2 Signal Models

In logic simulation, *variables* are usually called *signals*. For example, in the schematic in Figure 10 a) we have three input signals, or variables, *A*, *B* and *X*, one output variable (signal) *Y* and three "internal" signals *i1*, *i2* and *i3*. However, the internal signals are not an equivalent of state variables. This is a combinational circuit, so an input-output model is satisfactory as it is also clear from the models in Figure 10 b) and c). The internal signals are only named connections between the elements (gates) in the schematic. Circuits modelled and simulated at RTL or GL are discrete-state systems and signal values are from some finite set. Each set of values is regarded a *signal model*. At GL the most common sets, or signal models, are:

- *{L, H}.* Only two values are used. The value *L* (*low*) is an equivalent of logical *0* in the Boolean algebra and the value *H* (*high*) of the logical *1*. They represent corresponding voltage levels, for example L=0V, H=5V.
- *{L, H, u}*. Most of the digital circuits contain some sort of memory elements (i.e. flip-flops) that can store some value. This value is computed from the previous inputs of given element and when the circuit is powered up it is usually unknown. The unknown value is designated as *u* or *X*. It means that the value is *L* or *H*, but we don't know which one it is. The value *u* can be also used for some kinds of hazards (oscillations).
- *{L, H, Z}, {L, H, u, Z}.* These models are used for circuits with so-called *tri-state gates*, which output can have one of the three values: L, H or *high impedance*, denoted as *Z.* If the output of a tri-state gate is *Z* then it doesn't influence the rest of the circuit. This means that outputs of more than one tri-state gates can share a common wire provided that all of them but one are Z at any time instant.
- *{L, H, R, F}, {L, H, u, R, F}*. So far, we assumed that an output of a gate changes instantly from one value to another. The additional values *R* and *F* allow us to model that this change takes some time. The value *R* (*rise*) means that the value of the signal is changing from *L* to *H* and the value *F* (*fall*) that it is changing from *H* to *L*.

There are also other models with values representing various kinds of hazards. And in the case of MOS digital circuits simulation we also distinguish between strong and weak L and H values. At RTL the signal values are usually sequences (arrays), which members are values listed above.

III.1.3 Timing Models

To make the logic simulation realistic, we need to take into account the time needed to change the output of a gate or to transfer a signal through a wire. If we do not take timing into account, we have the *zero-delay timing model* and the models are static.

In timed models we consider three types of delays:

- *Transport delay.* It represents the time a gate needs to change its output after its inputs are changed. Various models can be used here and they are described in section III.1.3.1.
- *Inertial delay.* It is the minimum input pulse duration necessary for the output to switch states. This means that pulses shorter than this delay cannot pass through an element, so the inertial delay models the limited bandwidth of logic gates. (Wang et.al, 2009)
- *Wire delay* is the time a signal needs to get from an output of one element to an input of another element. It can be different for every such connection.

Functional elements, such as flip-flops, have more complicated behaviours than simple logic gates and require more sophisticated timing models. Besides the input-to-output transport delay, the flip-flop timing model usually contains timing constraints, such as setup/hold times and inertial delays for each input. (Wang et.al, 2009)

III.1.3.1 Transport Delay Models

To model the transport delay the following three models are usually used (Wang et.al, 2009):

- *Nominal delay model*, where the same delay value is used when the output is rising (i.e. going from *L* to *H*) and falling (i.e. going from *H* to *L*).
- *Rise/fall delay model*. Here different values are used for fall and rise.
- *Min–max delay model.* This model is used when the transport delay cannot be specified exactly. Instead, an interval *<min, max>* is defined and the actual delay can be any value from the interval.

III.1.4 Fault Models

Fault models are used to simulate faults in circuits when the simulation is used to evaluate diagnostic tests. The test passes the evaluation if it successfully reveals these emulated faults. At the gate level, we use short circuits between signal wires and *single stuck line fault model*. The latter can be

- *stuck at 0*, where the given signal is always 0 and
- *stuck at 1*, where the given signal is always 1.

We assume that only one input on one gate will be faulty at a time, which works well for the TTL logic but only moderately well for the CMOS logic. At RTL a fault represents a group of faults from lower level, i.e. memory faults, control faults, or decoder faults.

III.2 Logic Simulation Methods

There are two commonly used methods for performing logic simulations (Wang et.al, 2009): *compiled-code simulation* and *event-driven simulation*.

The ***compiled-code simulation*** means that the digital circuit model is translated into a computer program, a series of machine instructions that emulate individual gates and the interconnections between them. This method is incapable of timing modelling and uses zero-delay model only. Therefore it can be used only for evaluation of logic function correctness of given circuit (this kind of simulation is also called *cycle-based simulation*). It is the most effective for the *{L, H}* signal model. Its advantage is direct usability of machine instructions for Boolean operations and bit-wise logic operations, which speeds up the simulation.

The ***event-driven simulation*** is highly efficient as it evaluates gates only when necessary. An *event* is a switching (i.e. a change) of a signal value. An event-driven simulator monitors occurrences of events to determine which logic elements to evaluate. It can handle any delay model by means of a mechanism called *event scheduler*.

Event scheduler is a process responsible for registering events and executing them when their time comes. One of the possible implementations of the scheduler is by means of a *priority queue*. The ***priority queue*** in this case is a data structure, which consists of several queues, each for different priority. The individual queues are classical FIFO queues, i.e. the first element inserted into the queue

will be the first one removed from it. When a new element *a* with priority *p* is to be inserted into the priority queue then the queue for the priority *p* is selected and *a* is inserted into its rear. When there is a request to remove an element then a nonempty queue with highest priority is selected and the element is removed from its front. In the event scheduler these priorities are time instants when events should occur and members of the queue are pairs (a, v'_a), where a is a signal and v'_a is a new value that is assigned to a by the event (*Figure 11*). The lowest time value is the highest priority. Use of the priority queue event scheduler is illustrated by Example 6.

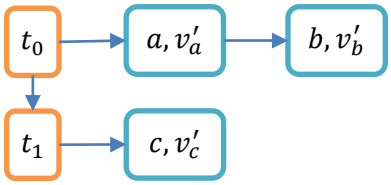


Figure 11. An example of a priority queue with two queues used for an event scheduler

Example 6. *Event-driven simulation of multiplexer*

In this example we show how the priority queue can be used for an event-driven simulation of the 2-to-1 multiplexer from Figure 10 a). We will use the nominal delay timing model with transport delays as in Table 2. We assume that the input signal *A* is always *H*, *B* is always *L* and *X* is initially *H* and at *time=60* changes to *L* (Figure 12). The diagram in Figure 12 also shows how other signals of the multiplexer change with respect to the inputs and transport delays (for the output and internal signals we assume the value *u*, indicated by a dashed line).

Element	INV1	AND1	AND2	OR1
Transport delay	10	15	15	15

Table 2. Transport delays of gates of multiplexer.

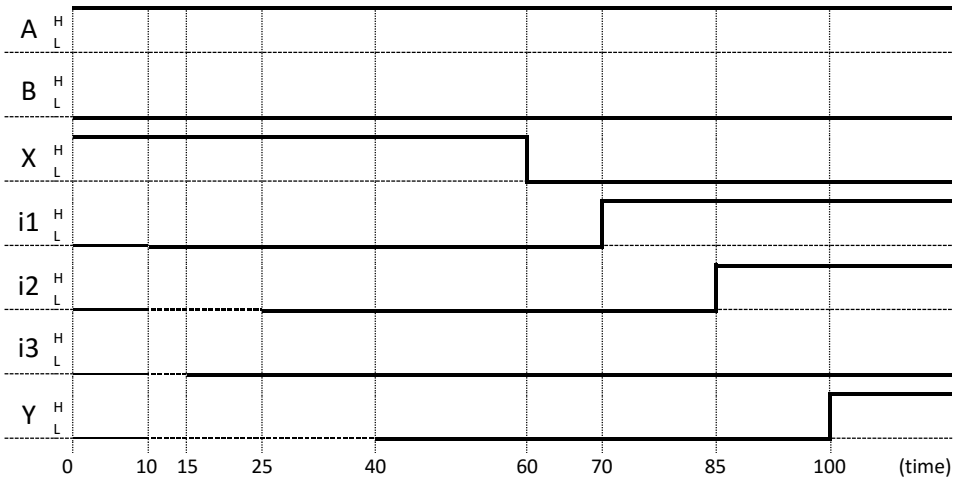


Figure 12. Signals diagram of the multiplexer

We start the simulation by inserting all the events we know at the beginning into the priority queue. After this the queue will look like in Figure 13.

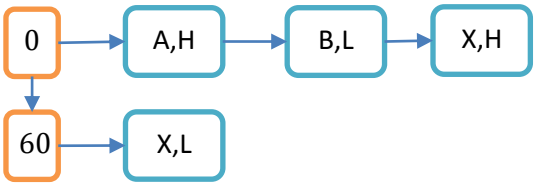


Figure 13. Priority queue before start of the simulation

Then we remove and process the events from the priority queue one by one. Processing the event means removing it from the queue and evaluating how and when the change of a signal the event represents changes other signals in the system. These changes are inserted as new events into the priority queue. To make the simulation more effective this is only performed for events that really change given signals. Development of the priority queue during the simulation is shown in Figure 14.

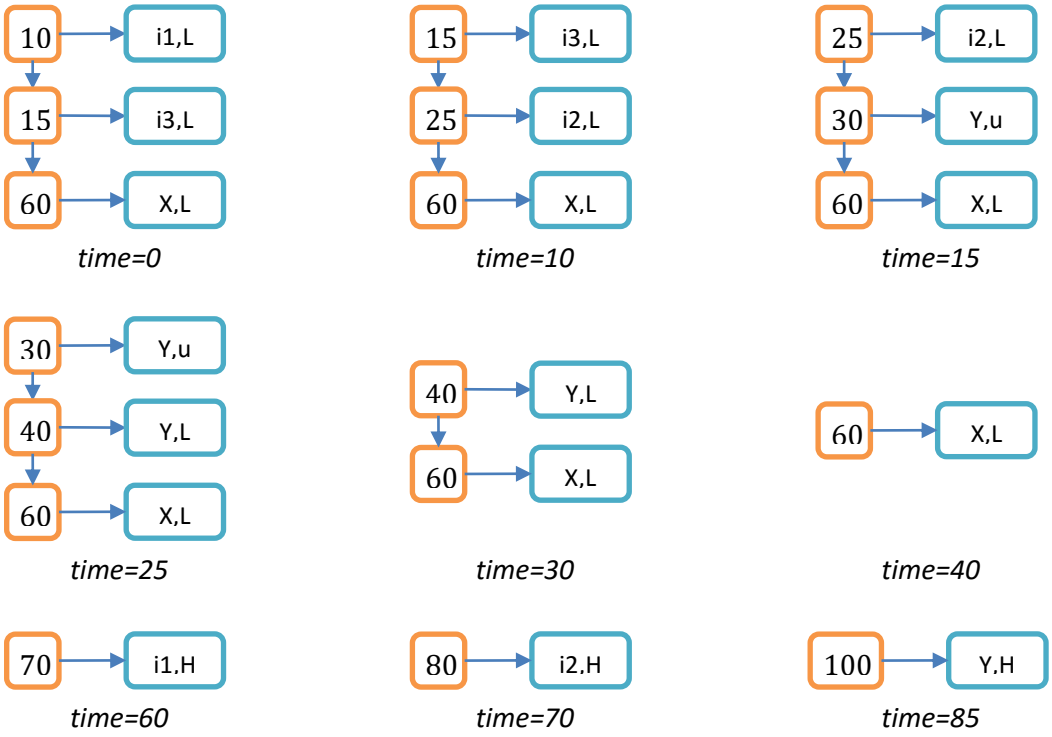


Figure 14. Priority queue after processing all events for given time instant

At *time=100* the priority queue is emptied because no new events are generated and the simulation ends. We can see that the occurrence of the event *(Y,u)* doesn't produce any new event. This is because the value of the signal *Y* remains the same (*u*). Another reason, which is a sufficient one, too, is that the signal *Y* is not an input of any element (the same is true for the event *(Y,H)* at *time=100*).

□

IV Pseudo-random Numbers

As we mentioned earlier, the discrete-event systems are often stochastic, so in their simulation we use *random*, or more exactly *pseudo-random*, *numbers*. Random numbers are very useful in various areas related to computers and information technologies. For example, in numerical analysis we can use them to solve complicated integrals by so-called Monte Carlo methods. In cryptography they are used for several purposes, such as for generation of keys that encrypt and decrypt data. Their place in software development is in testing, to generate testing data. In simulation we use them as a source of randomness in a model. This chapter can be regarded as an excerpt from (Perros, 2009), where the reader can find more details about pseudo-random numbers and their generators.

It makes no sense to talk about an individual random number, without any relation to other numbers. Instead we talk about random numbers in the context of some ***sequence of random numbers***. The word "random" here means that there seems to be no relation between members of the sequence and these members follow some theoretical or empirical probability distribution. According to a method of their generation random numbers can be of two kinds – *true random numbers* and *pseudo-random numbers*.

The ***true random numbers*** have to be generated by a completely unpredictable and non-reproducible source. Usually, physical phenomena are used as generators. For example a radioactive source, a thermal noise from a resistor or a semi-conductor diode or a human computer interaction processes, such as mouse or keyboard use. They are very useful in cryptography.

The ***pseudo-random numbers*** are numbers generated by some algorithm, i.e. in a deterministic way. This means that given the same starting value, called ***seed***, the algorithm will always generate the same sequence. This determinism is not good for cryptography as an attacker can find out the way in which numbers are generated and reproduce it. However it is desirable for software testing or simulation, because we can easily replicate experiments. The word "pseudo" means that these numbers are not really random but seem to be random. The ***pseudo-random number generators***, i.e. algorithms that generate these numbers, usually work in such a way that they give us next member of a sequence of pseudo-random numbers and these numbers are *uniformly distributed*. ***Uniformly distributed*** means that all possible values can occur with the same probability.

The term ***pseudo-random number*** is typically reserved for random numbers that follow uniform distribution on the interval $< 0,1 >$. All numbers that follow another distribution or are from another interval are called ***random variates*** or ***stochastic variates***.

IV.1 Probability Distributions

In the previous section we said that members of a sequence of random numbers follow some theoretical or empirical probability distribution. At this place we recall some basic facts from the probability theory about what defines these distributions.

The probability that a value of some real-valued random variable X is smaller than some real number x is denoted as

$$\Pr(X \leq x).$$

There are two functions that characterize a probability distribution – *the cumulative distribution function* (*cdf*) and the *probability density function* (*pdf*).

The ***cumulative distribution function*** (***cdf***), denoted $F_X(x)$ or just $F(x)$, is defined as

$$F_X(x) = \Pr(X \leq x).$$

Then the probability that a value of X will be from some interval $(a, b >$ is

$$\Pr(a < X \leq b) = F_X(b) - F_X(a)$$

Probability of an exact value is always zero, i.e. $\Pr(X = a) = 0$.

The ***probability density function*** (***pdf***), denoted $f_X(x)$ or just $f(x)$, describes the relative likelihood for this random variable to have a given value. In this case the probability that a value of X will be from some interval $(a, b >$ can be computed as

$$\Pr(a < X \leq b) = \int_a^b f(u)\, du$$

So, the relationship between *pdf* and *cdf* is

$$F(x) = \int_{-\infty}^{x} f(u)\, du$$

IV.2 Generators

The ***pseudo-random number generators*** are algorithms that generate sequences of pseudo-random numbers. They usually work on demand, i.e. they generate next number of a sequence when asked. The first number or first numbers of the sequence must be given and they are called ***seed***. The generators are deterministic, so for the same seed the generator will always generate the same sequence. Numbers generated by them have to be

- uniformly distributed,
- statistically independent,
- reproducible and
- non-repeating for any desired length.

One of very important properties of generators is a ***period***, which is a number of successively generated pseudo-random numbers after which the sequence starts repeating itself.

IV.2.1 Middle-square Method

This is the oldest generator, invented by the computer science pioneer John von Neumann. It is very simple and consists of two steps:

1. Take the square of previously generated number.
2. Extract the middle digits.

It is not recommended for practical use, because it is slow and has a very short period.

IV.2.2 Congruential Methods

These methods are very popular, because they are simple, fast and produce statistically acceptable numbers, at least for the simulation. They follow the general formula, where the $i+1$th member of the sequence (i.e. x_{i+1}) is computed from the previous members as

$$x_{i+1} = \big(f(x_i, x_{i-1}, \dots)\big) mod\ m$$

And individual methods differ in how many previous members of the sequence they use to compute an actual member x_{i+1} and in how the function f is defined. We say that a congruential generator *has a full period* if its period is equal to m.

Examples of congruential generators are quadratic congruential generator

$$x_{i+1} = \big(a_1 x_i^2 + a_2 x_{i-1} + c\big) mod\ m$$

and linear congruential generator

$$x_{i+1} = (a x_i + c) mod\ m$$

The linear generator generates numbers between 0 and $m-1$. If we want to generate numbers from $< 0{,}1 >$, we divide each generated number by $m-1$ (this is also true for other types of generators). The period of the linear generator is full when

- m and c have no common divisor,
- $a-1$ is divisible by all prime factors of m and
- $a-1$ is a multiple of 4 if m is a multiple of 4.

When implementing the linear generator an optimisation in the form of setting m to the size of used register is often used. Then modulo operation is automatically performed by overflow of the register.

Other examples of generators from this family are *Tausworthe generators*. They are additive congruential generators with $m = 2$.

IV.2.3 Composite Generators

Composite generators are generators that combine two (or more) separate generators, usually congruential. They can have good statistical properties, even if the generators used are bad. A generator composed from two generators G1 and G2 can proceed as follows:

1. Generate a sequence $x_1 \dots x_k$ using generator G1
2. Generate an integer r, $r \in 1, \dots, k$ using generator G2
3. Return x_r
4. Generate a new random number using G1 and replace x_r in the sequence with it.
5. Go to step 2.

IV.2.4 Lagged Fibonacci Generator

The Lagged Fibonacci Generators or *LGF*, are based on the Fibonacci sequence

$$x_n = x_{n-1} + x_{n-2}, x_0 = 0, x_1 = 1$$

They have a general form

$$x_n = (x_{n-j}\ op\ x_{n-k})\ mod\ m$$

where op is an algebraic operation $(+, -, *, \ldots)$ and $0 < j < k$.

Their advantages are very good statistical properties, efficiency only a bit lower than of the congruential generators and parallelization possibilities. The biggest disadvantage is high sensitivity on the seed, so the seed have to be carefully selected to obtain good statistical properties.

Commonly used choice for op is addition. The parameter m is usually computed as $m = 2^M$ and good choices for j, k and M are

- $j = 5, k = 17, M = 31$ and
- $j = 24, k = 55, M = 31$.

IV.2.5 Mercenne Twister

This generator is a variation on a two-tap generalised feedback shift register, which in fact is a LFG with xor as op. Its period is a Mersenne prime. A Mersenne prime is a prime number of the form $M_n = 2^n - 1$.

Mercenne Twister generates a sequence of bits, which is grouped into blocks (32-bit) and each block blocks is considered to be a random number. It has very good statistical properties and its maximum period is very high - $2^{19937} - 1$. Its disadvantages are complex implementation and sensitivity to poor initialisation. Complete description of this algorithm can be found in (Perros, 2009).

IV.3 Statistical Tests for Generators

They are used to check the output of a pseudo-random number generator statistically and belong to the statistical hypothesis testing. One of the most fundamental test is the *frequency test*. It is considered fundamental, because if a generator fails it, it will probably fail other tests, too. It checks whether there is approximately the same number of occurrences of each digit in the generated sequence. The *serial test* is similar to the frequency test but for pairs of digits. The *Autocorrelation test* is based on the fact that if the sequence s of n bits, created by a generator, is random, then it is different from another bit string obtained by shifting the bits of s by d positions. The *Runs test* is used to test the assumption that the pseudo-random numbers are independent of each other (mutually independent). It checks whether counts of ascending and descending runs follow a certain distribution. It belongs to the diehard tests. Finally, the *Chi-squared test for goodness of fit* checks whether a sequence of pseudo-random numbers in $< 0,1 >$ is uniformity distributed.

IV.4 Generation of Random Variates

The generators mentioned above produce uniformly distributed numbers from the interval $< 0,1 >$. To get random variates that follow other distributions we need to transform them. There are several methods for transformation and in this section we describe two of them: *inverse transform sampling* and *rejection method*.

IV.4.1 Inverse Transform Sampling

This method uses inversed cumulative density function (cdf) of the target distribution (i.e. the distribution we want the random variates to follow). This limits its usability to distributions which cdf can be analytically inverted.

If $r, r \in < 0,1 >$ is a generated pseudo-random number then a random variate x following a random distribution with cdf F can be computed as

$$x = F^{-1}(r).$$

Examples of the sampling are in Table 3.

sampling from	Uniform distribution from interval $< a, b >$.	Exponential distribution with rate parameter λ.
cdf	$F(x) = \begin{cases} 0 & for\ x < a \\ \frac{x-a}{b-a} & for\ a \leq x < b \\ 1 & for\ x > b \end{cases}$	$F(x) = 1 - e^{-\lambda x}$
random variate	$x = a + (b - a)r$	$x = -\frac{1}{\lambda}\ln r$

Table 3. Transport delays of gates of multiplexer.

IV.4.2 Rejection Method

The rejection method can be used when x has a finite range (i.e $x \in < a, b >$) and the probability density function (pdf) $f(x)$ of the target distribution is bounded on $< a, b >$). The method uses a *pdf with normalized range*, which is the function

$$f_{norm}(x) = c \cdot f(x),$$

where c is a constant such that $\forall x(a \leq x \leq b)$: $cf(x) \leq 1$. Then the transformation itself proceeds as follows:

1. Generate a pair of random numbers (r_1, r_2), $r_1 \in < 0,1 >$, $r_2 \in < 0,1 >$.
2. If $r_2 \leq f_{norm}(a + (b - a)r_1)$ go to step 3, otherwise go to step 1.
3. Return $x = a + (b - a)r_1$ as the random variates.

V Queuing Systems

Queuing systems are the subject of so-called *queuing theory* that is dedicated to the mathematical study of systems consisting of queues (i.e. waiting lines) and servers. There can be *single queue queuing systems* (Figure 15) with only one queue and one or more servers with the same parameters (i.e. with one service providing facility (Cassandras & Lafortune, 2008)) or *queuing networks* (Figure 16) with multiple queues and servers connected together.

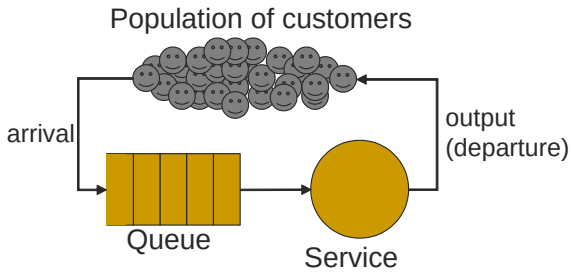


Figure 15. Single queue queuing system

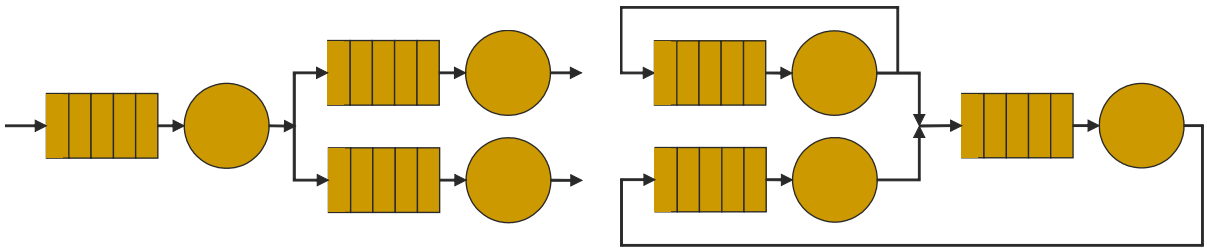


Figure 16. Example of open (a) and closed (b) queuing network

V.1 Single Queue Queuing System

As its name suggests, a *single queue queuing systems* (Figure 15) consists of one *queue* and one *service providing facility*, which we simply call a *service*. There is also a *population of customers*, who arrive to the system, wait in the *queue* to be served by the *service* and leave the system after being served.

The population of customers can be *finite* or *infinite*. The ***finite population*** means that the system will always serve the same limited number of customers, but not necessarily exactly the same customers. In the model it will look like the same customer enters the queue again after being served (
Figure 17 a)) but in the real system they can be different individuals. This is because in models of queuing systems we usually do not deal with identity of customers, they are simply undistinguishable. A queuing system with finite population is ***closed***. In the case of ***infinite population*** the number of customers that can possibly enter the system is unlimited and such systems are called ***open*** (

Figure 17 b)). Of course, by customers we don't only mean people. They can be tasks, jobs, processes, products, etc.

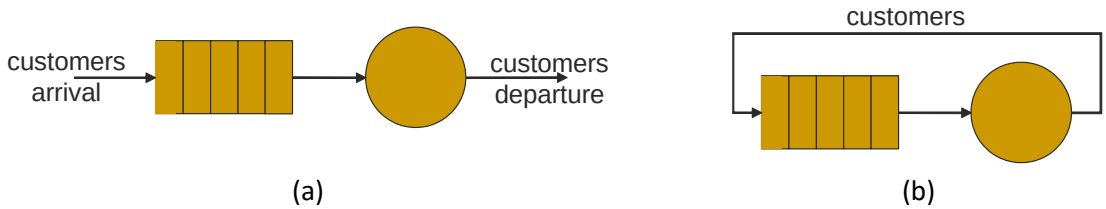


Figure 17. Open (a) and closed (b) single queue queuing system

The way in which customers enter given queuing system is mostly random and is defined by the ***arrival pattern***, which specifies a random distribution of intervals between two adjacent arrivals of customers. Departure of customers is not treated in any specific way in most cases.

V.1.1 Queue

The queue is the part where customers wait to be served by the service. There are two important parameters of the queue: the maximum size of the queue, or *queue capacity*, and *queuing discipline*.

The ***queue capacity*** is the maximum number of customers that may wait in the queue. In some theoretical models we assume an unlimited capacity.

The ***queuing discipline*** defines the policy of adding and removing customers to and from the queue:

- *FIFO* (First In is First Out). The classical queue, where the first customer that enters it is the first to be served.
- *LIFO* (Last In is First Out). The queue in this case works as a stack.
- *SIRO* (Serve In Random Order). There is no particular order of customers here, they are picked up randomly.
- *Priority Queue*. This can be built from several ordinary queues, each for one priority. It has been already described in section III.2.

V.1.2 Service

The ***service*** represents an activity the customers are waiting for. To serve a customer takes some time, so one of service parameters will be its duration. As in the case of arrival it is mostly random and is defined by the ***service pattern*** as a random distribution of service duration.

Second important parameter is the ***number of servers***, which defines how many customers can be served at once. On the basis of this parameter we distinguish

- *single-channel systems* with one server and
- *multi-channel systems* with more than one server.

The maximum number of customers being served simultaneously together with the queue capacity define the ***system capacity*** of given queuing system.

V.1.3 Kendall's Notation

In queuing theory there is a standard way to describe queuing systems, created by D. G. Kendall and therefore called *Kendall's classification* or *Kendall's notation*. The original notation, proposed by Kendall in 1953 consisted of three factors (*A/B/s*) but has been later extended to six, *A/B/s/q/c/p*, where

- *A* is the arrival pattern,
- *B* is the service pattern,
- *s* is the number of servers,
- *q* is the queuing discipline,
- *c* is the system capacity or queue capacity,
- *p* is the population size, i.e. number of possible customers.

There are some standard values for patterns *A* and *B*, for example

- *M* - Poisson arrival distribution (exponential inter-arrival distribution) or an exponential service time distribution,
- *Em* - Erlang distribution,
- *D* - deterministic or constant value (also called degenerate distribution),
- *G* - general distribution with a known mean and variance.

For example, a queuing system denoted as *M/M/1* has Poisson arrival distribution, exponential service time distribution, one server, a queue with FIFO or unspecified discipline and unlimited capacity and is open.

V.2 Performance Measures

When we design a model of a real system as a queuing system, we are usually interested in its performance with respect to use of some shared resource, represented by services. The standard performance measures are:

- average waiting time,
- expected number of waiting customers,
- expected number of customers receiving service,
- probability of an empty system,
- probability of a full system,
- probability of having an available server and
- probability of having to wait a certain time to be served.

For some types of queuing systems, such as M/M/1, these measures can be obtained analytically using the queuing theory. In other cases we have to use simulation.

VI Discrete Event Simulation with Coloured Petri Nets

In this chapter we show how the *Petri nets* (*PN*) formalism can be used for modelling and simulation of discrete – event systems (i.e. discrete-state event-driven systems). PN are a formal language that is able to naturally express behaviour of non-deterministic, parallel and concurrent systems. One of their biggest advantages is an easy to understand graphical notation. They also offer analytical methods, which, for example, allow to derive invariant properties from the structure of the net. The original PN formalism has been introduced in the PhD thesis (Petri, 1962) by Carl Adam Petri. As the name of the thesis suggests, his intention was to design formalism able to describe systems consisting of cooperating (communicating) finite automata. Since its introduction in 1962 the PN formalism has been modified and extended in many ways by various researchers, resulting in many types of PN with different expressional and modelling power. A good introduction to various types of PN can be found in (Reisig & Rozenberg, 1998a-b). Basic types of Petri nets are also well-covered in (Reisig, 2013).

The *Place-Transition Nets* (P/T nets) are considered a basic type of Petri nets nowadays and they are very close to the C.A. Petri's original concept of communicating sequential automata. However, P/T nets are not very useable for specification of simulation models of discrete-event systems. This is primary because they lack any means for (convenient) representation of data and do not incorporate timing. Therefore, PN types with greater expressional and modelling power are required. For this book we have chosen *Coloured Petri nets* (*CPN*) since they are well covered in literature (Jensen, 1994), (Jensen, 1997a-c), (Jensen & Kristensen, 2009), have very good support in the form of the *CPN Tools* software tool (http://cpntools.org/) and are suitable for simulation-based analysis (Jensen et.al, 2007), (Jensen & Kristensen, 2009). Definitions of CPN, used in this chapter, have been taken from the aforementioned books and papers by Kurt Jensen. The Petri net models shown here have been created using the *CPN Tools* software.

VI.1 Basic Concepts

We start our discussion on *Petri nets* (*PN*) with an informal introduction to their basic concepts, namely how they look and behave and how they allow us to describe nondeterministic and concurrent behaviour. In general, a Petri net has a form of an oriented bipartite graph. The two types of vertices are *places* and *transitions*. *Places* have the shape of circles or ellipses and represent state of the net. *Transitions* have the shape of bold lines or rectangles and represent actions or events that can change state of given PN. Places and transitions are connected by directed arcs. Arcs can connect only vertices of different kind, so we will never have an arc between two places or two transitions. Places hold objects, called *tokens* and by the nature of these tokens we distinguish two basic classes of Petri nets:

1. *Low-level Petri nets*. Here, tokens are all the same, we cannot distinguish between them. P/T nets belong to this class. Undistinguishable tokens are usually shown as black dots inside a place they occupy. In CPN we can also have these types of tokens, in current (2015) version of CPN Tools they are denoted "()".
2. *High-level Petri nets*. Tokens can have different values. Usually, we also specify a type of given token and often it is required that one place only holds tokens of one type. CPN

belong to this class. The word "coloured" in their name refers to the fact that tokens have different colours and are not just undistinguishable black dots.

Number and values of tokens in some place p of a Petri net is called *marking of* p and is denoted $M(p)$. If we fix ordering of places of the net, say to $p_1, p_2, \dots, p_n$, we can write the marking of the whole net as a vector $M = (M(p_1), M(p_2), \dots, M(p_n))$. Markings represent states of PN. We distinguish between different markings by using lower and upper indices, for example M', ${M_1}'$, M_2. M_0 is usually reserved for the *initial marking*, that is the marking, which the net has when it begins its computation. *Computation* of PN is a sequence of *firings* or *executions* of its transitions. A transition *t* can be *fired* (*executed*) if there are enough tokens of required values in *pre-places* of t. When t fires, it removes tokens from its pre-places and adds tokens to its *post-places*. This means that the net *reaches* a new marking. *Pre-places* of t are places from which there is an arc to t and *post-places* of t are places to which there is an arc from t. Values and number of tokens that are required for firing and are removed and added by firing of t are defined by *arc expressions*, associated to arcs from and to t. Computations of PN can be written in the form of *occurrence sequences*, which record transitions fired and markings reached. How various basic types of behaviours can be modelled by Petri nets and how exactly enabling and firing of transitions proceed is shown in following examples. While we use CPN here, for the sake of simplicity we restrict ourselves to what is also possible in simpler types, such as P/T nets. That is, we use only undistinguishable tokens.

Example 7. *Nondeterminism*

In Figure 18 we can see a simple Petri net that describes a process with two branches. Only one of these branches can be executed and it is chosen non-deterministically which one it will be. The net has 7 places, `p1`,`p2`,…,`p7`, and 7 transitions. The first transition, `act1`, represents some action that occurs at the beginning of the computation of the net. Transitions `choice1`, `altAct2_1` and `altAct3_1` belong to the first branch and `choice2`, `altAct2_2` and `altAct3_2` to the second branch.

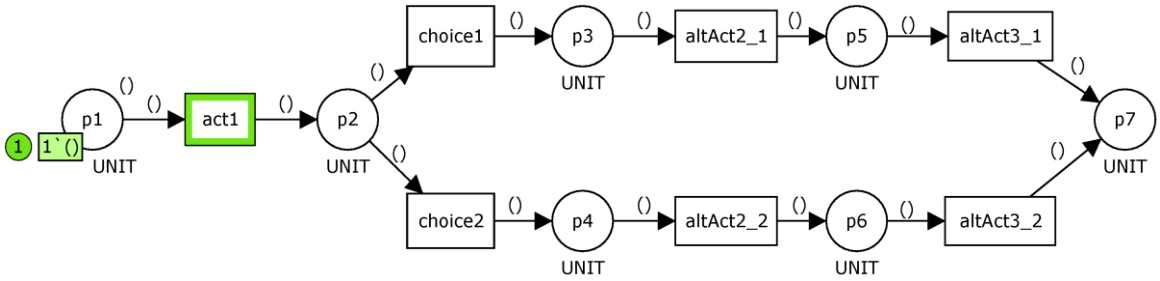


Figure 18. Petri net with two non-deterministic branches of process execution in its initial marking M_0

Each place of the net can hold only undistinguishable tokens, i.e. tokens of the type `UNIT`, which has only one value - "`()`". Brackets are used for this value, because they resemble a circle. Types in CPN are also called *colour sets* and they are usually written below the places. Initial marking of the places of the net is defined by *initialisation expressions*. We have such an expression in the

outer upper right corner of p1 and it says "()". This means that there is one token (of value "()") in p1 in the initial marking M_0. Formally, we write this as $M_0(p1) = 1`()$ or $M_0(p1) = ()$. There are no initialisation expressions for other places, so there are no tokens in them in M_0. If we fix ordering of places to p1,p2,…,p7 we can write

$$M_0 = (1`(), \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \quad (50)$$

where $\emptyset$ means "no tokens". Figure 18 shows the net in this initial marking. The number of tokens in $p1$ is shown in a green circle, followed by a green rectangle with an expression describing the marking in detail. Here it is "1`()", meaning "one token of the value ()". The transition act1 is enabled in $M_0$ as there is enough tokens in its pre-place p1. Enough means one token since the arc expression of the arc from $p1$ to $act1$ is "()", which is the same as "1`()". Enabling of act1 is highlighted by a green frame in Figure 18. Firing of act1 in M_0 changes the marking to M_1,

$$M_1 = (\emptyset, 1`(), \emptyset, \emptyset, \emptyset, \emptyset, \emptyset), \quad (51)$$

and enables transitions $choice1$ and $choice2$ (Figure 19). Formally we write

$$M_0\,[\text{act1}\rangle\, M_1\,. \quad (52)$$

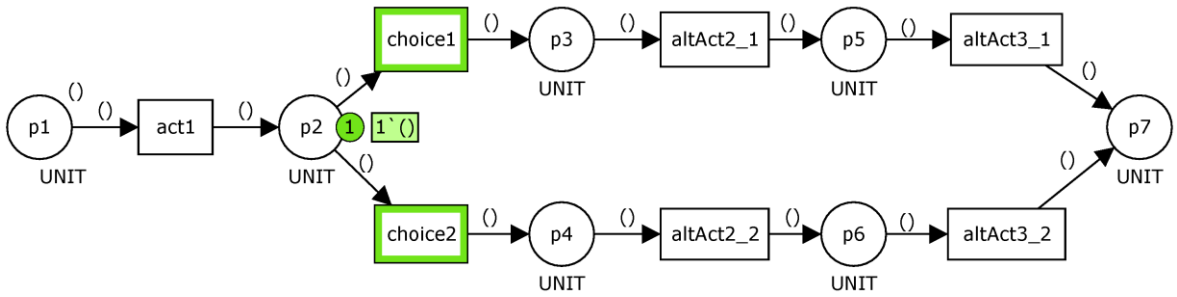


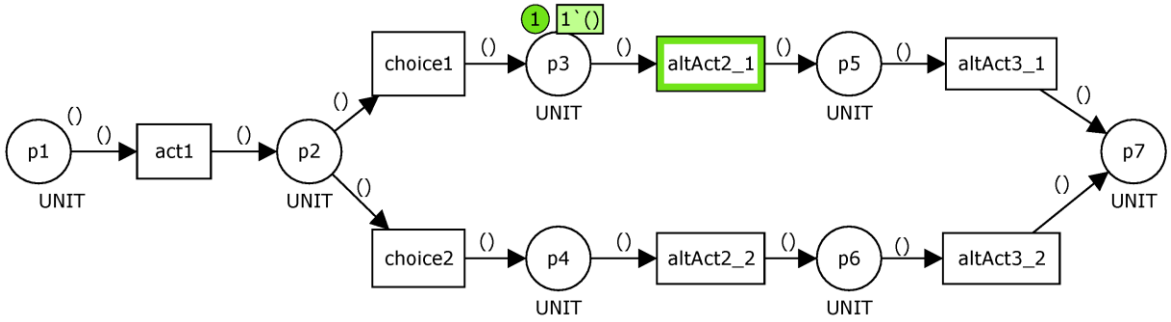
Figure 19. Net from Figure 18 in the marking M_1

As we can see, firing of act1 removed one token from p1 (because the arc from p1 to act1 has the arc expression "()") and added one token to p1 (because the arc from act1 to p2 has the arc expression "()", too). In M_1 both choice1 and choice2 are enabled, but only one of them can be fired. This is because there is only one token in p2 and when one of these transitions fires the token is consumed and there is no one left for the other transition.

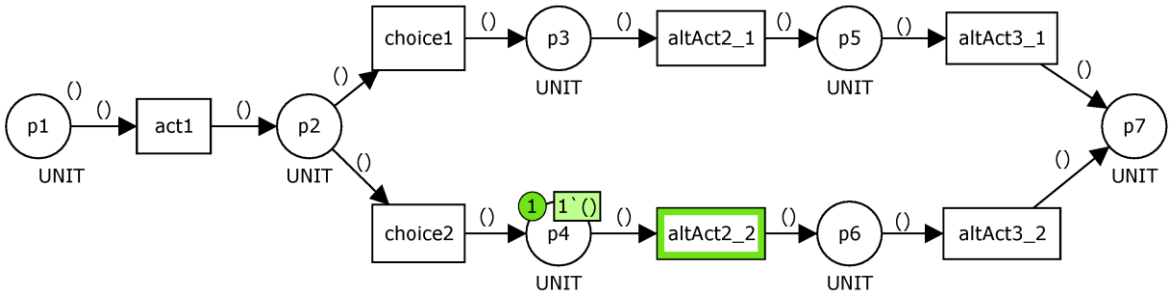
One may ask how do we determine which transition will be fired in M_1. The answer is by no means, one of the enabled transitions is chosen *non-deterministically* and fired. The firing of choice1 results in M_2 (53) and of choice2 in M_4 (54).

$$M_2 = (\emptyset, \emptyset, 1`(), \emptyset, \emptyset, \emptyset, \emptyset) \quad (53)$$
$$M_4 = (\emptyset, \emptyset, \emptyset, 1`(), \emptyset, \emptyset, \emptyset) \quad (54)$$

The net in these markings is also shown in Figure 20.



(a)



(b)

Figure 20. Net from Figure 18 in the marking M_2 (a) and M_4 (b)

Both choices eventually lead to the same marking M_6 (55), shown in *Figure 21*. There is no enabled transition in M_6 and we call such markings *deadlocks*. However, in this case a designation *final marking* is more appropriate.

$$M_6 = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, 1`()) \qquad (55)$$

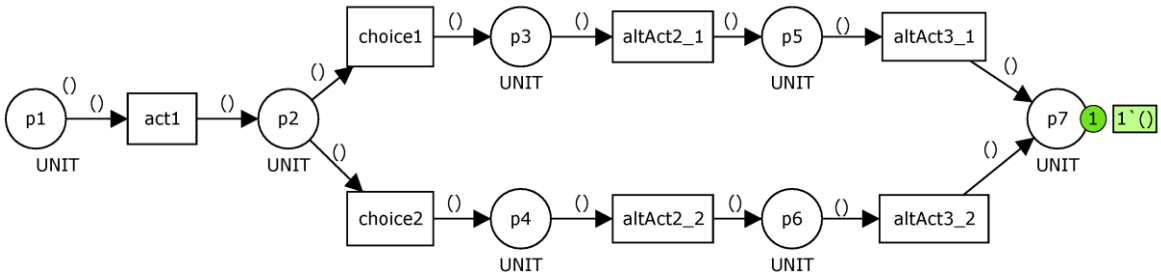


Figure 21. Net from Figure 18 in the marking M_6

To conclude, there are only two possible *occurrence sequences* in this net: (56), which chooses the first branch and (57), which chooses the second one.

$$M_0\,[\text{act1}>\,M_1\,[\text{choice1}>\,M_2\,[\text{altAct2\_1}>\,M_3\,[\text{altAct3\_1}>\,M_6 \qquad (56)$$

$$M_0\,[\text{act1}>\,M_1\,[\text{choice2}>\,M_4\,[\text{altAct2\_1}>\,M_5\,[\text{altAct3\_1}>\,M_6 \qquad (57)$$

Markings M_0, M_1, M_2, M_4, M_6 are as in (50), (51) and (53) to (55) and M_3, M_5 are

$$M_3 = (\emptyset, \emptyset, \emptyset, \emptyset, 1`(), \emptyset, \emptyset),\ M_5 = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, 1`(), \emptyset).$$

□

Example 8. *Parallelism*

The net in Figure 22 is very similar to the one from the previous example. Again, it is about a process that is split into two branches. What is different is that here these two branches, represented by transitions `prlAct1` and `prlAct2` and adjacent places, can be executed in parallel.

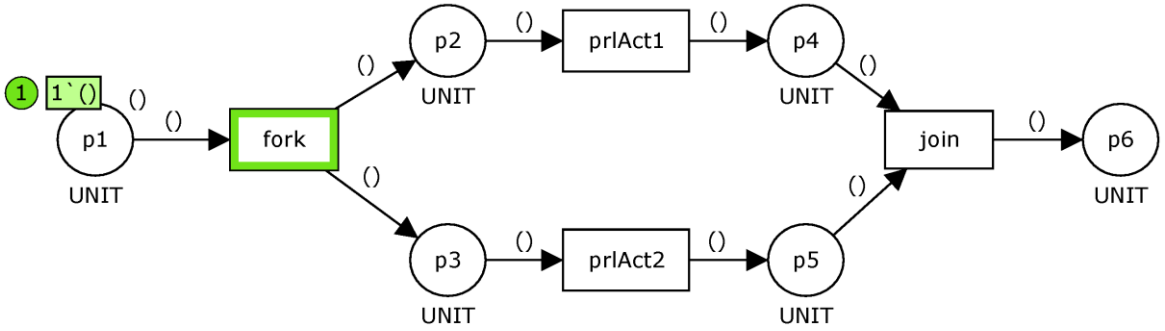


Figure 22. Petri net, representing a process with two parallel branches of process execution, shown in its initial marking $M_0 = (1`(), \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

This difference is caused by the fact that firing of `fork` removes one token from `p1` and adds one token to `p2` *and* one to `p3`, so in $M_1 = (\emptyset, 1`(), 1`(), \emptyset, \emptyset, \emptyset)$ we can fire both `prlAct1` *and* `prlAct2` (Figure 23).

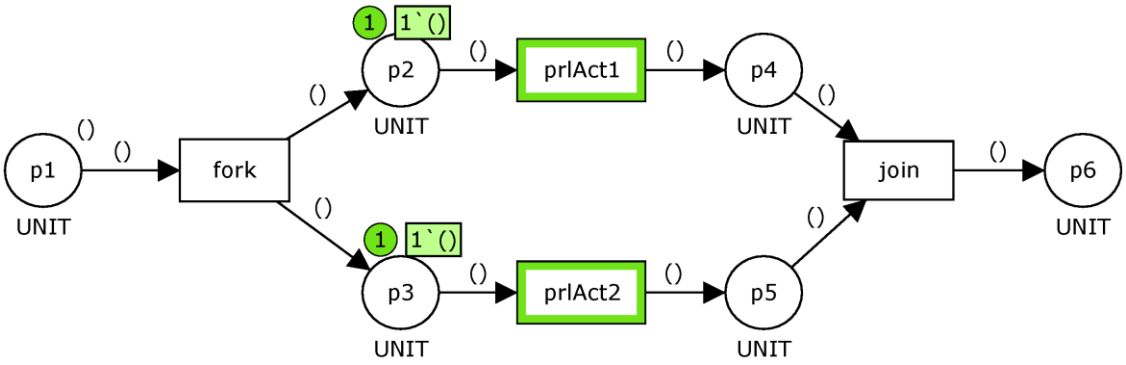


Figure 23. Petri net from Figure 22 in marking $M_1 = (\emptyset, 1`(), 1`(), \emptyset, \emptyset, \emptyset)$

There are three possible occurrence sequences we can fire *in* M_1: `prlAct1` prior to `prlAct2` (58) or vice versa (59) or both transitions *at once* in one *step* (60).

$$M_0[\text{fork}> M_1\ [\text{prlAct1}> M_2\ [\text{prlAct2}> M_4\ [\text{join}> M_5 \qquad (58)$$

$$M_0[\text{fork}> M_1\ [\text{prlAct2}> M_3\ [\text{prlAct1}> M_4\ [\text{join}> M_5 \quad (59)$$

$$M_0[\text{fork}> M_1\ [\text{prlAct1,prlAct2}> M_4\ [\text{join}> M_5 \quad (60)$$

In (58) to (60) new markings are $M_2 = (\emptyset, \emptyset, 1`(), 1`(), \emptyset, \emptyset)$, $M_3 = (\emptyset, 1`(), \emptyset, \emptyset, 1`(), \emptyset)$ *and* $M_4 = (\emptyset, \emptyset, \emptyset, 1`(), 1`(), \emptyset)$. As you can see, in this net we have markings with more than one place possessing tokens, namely M_1 to M_4.

□

The previous two examples demonstrated how to model non-deterministic and parallel behaviour by Petri nets. They have two things in common. First, they don't have any concrete meaning. And second, they represent processes, which proceed from some initial state (marking) to some final state. In the next example we have a net with *cyclic behaviour*. That is, a net, which is required to run forever. Many real systems, such as network protocols or processes inside operating systems, operate in this way and Petri nets are mostly used to describe these kinds of systems. This is why a marking in which no transition can be fired is usually called *deadlock* (negative meaning) and not *final state* (positive meaning). The next two examples describe systems with cyclic behaviour and they also show how to model *synchronisation* between *concurrent* processes.

Example 9. *Mutual exclusion*

The net in Figure 24 represents a simple solution to a problem known from operating systems domain as *mutual exclusion*. The problem is formulated as follows: We have two processes, say $p1$ and $p2$. Each of these processes can operate inside or outside a critical area. The critical area is some shared resource (e.g. a memory), which requires that only one process can access it at once. So, when $p1$ is inside the area, $p2$ has to be outside and vice versa.

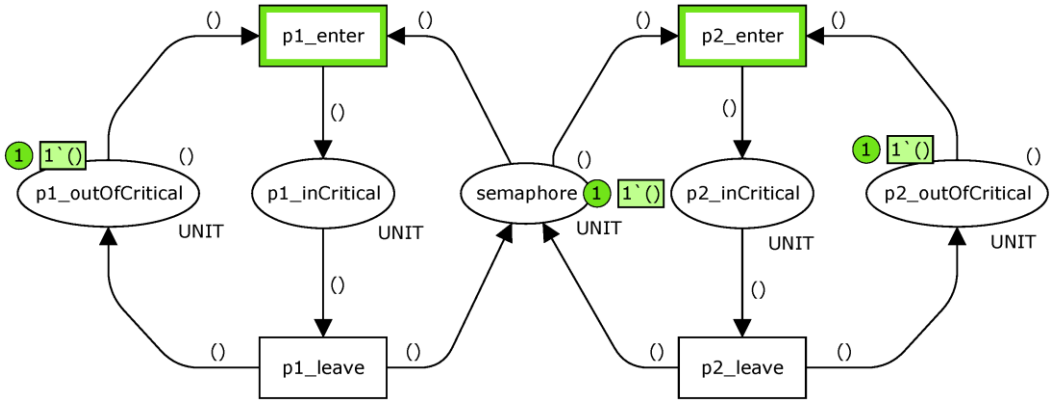


Figure 24. Petri net representing mutual exclusion of two processes in its initial marking M_0

The part of the net left to the place `semaphore` belongs to the first process, $p1$. Token in `p1_outOfCritical` means that $p1$ operates outside of the critical area, token in `p1_inCritical` means it operates inside. The firing of `p1_enter` represents the event of $p1$ entering the critical area, the firing of `p1_leave` the event of leaving the area. The part right to the place `semaphore` belongs to $p2$ and operates in similar way. The place `semaphore` and

adjacent arcs provide interlocking mechanism that prevents $p1$ and $p2$ to enter the area at once. Processes $p1$ and $p2$ are *concurrent*, because they run simultaneously and compete for access to shared resources.

The net has only three reachable markings. If we fix place ordering to `p1_outOfCritical`, `p1_inCritical`, `semaphore`, `p2_inCritical`, `p2_outOfCritical`, the markings are

- $M_0 = (1`(), \emptyset, 1`(), \emptyset, 1`())$ – initial marking, both processes out of the critical area,
- $M_1 = (\emptyset, 1`(), \emptyset, \emptyset, 1`())$ – $p1$ inside the critical area and
- $M_2 = (1`(), \emptyset, \emptyset, 1`(), \emptyset)$ – $p2$ inside the critical area.

Since there is no deadlock, there are infinitely many occurrence sequences, but they will always be composed of these four sequences:

- M_0 [p1_enter> M_1 - $p1$ enters the critical area,
- M_1 [p1_leave> M_0 - $p1$ leaves the critical area,
- M_0 [p2_enter> M_2 - $p2$ enters the critical area and
- M_2 [p2_leave> M_0 - $p2$ leaves the critical area.

For example, a more complicated occurrence sequence can be:

$$M_0[\text{p1\_enter}\rangle M_1\,[\text{p1\_leave}\rangle M_0\,[\text{p2\_enter}\rangle M_2\,[\text{p2\_leave}\rangle M_0[\text{p2\_enter}\rangle M_2$$

□

In the examples above we explored already created PN models. The next one shows how to design a PN model of more complicated process step by step.

Example 10. *Manufacturing process*

Suppose that we have to design a Petri net model of a manufacturing process for a manufacturing site, consisting of three parts:

- *A parts stack.* The stack holds sets of parts; each set is used to assembly one product. The stack has a capacity of 100 sets and can be refilled by packages of 20 sets. At the beginning the stack is empty.
- *An assembly line with one assembly station.* The line takes a set of parts from the parts stack, moves it to the assembly station, the station assemble a product from the set of parts and the assembled product is moved to the end of the line. After the product is removed a new set of parts can be used to make a new product. At the beginning the line is empty and ready to take a new set of parts.
- *A crane* that moves finished products from the end of the line to a *storage box*. The crane has north/south orientation with the end of the line on the north end and the box on the south end. At the beginning the crane is on the south end and is not holding anything and the box is empty.

We will start with designing a Petri net that represents the assembly line part of the manufacturing process. The net models it as a sequential cyclical process and is shown in Figure 25.

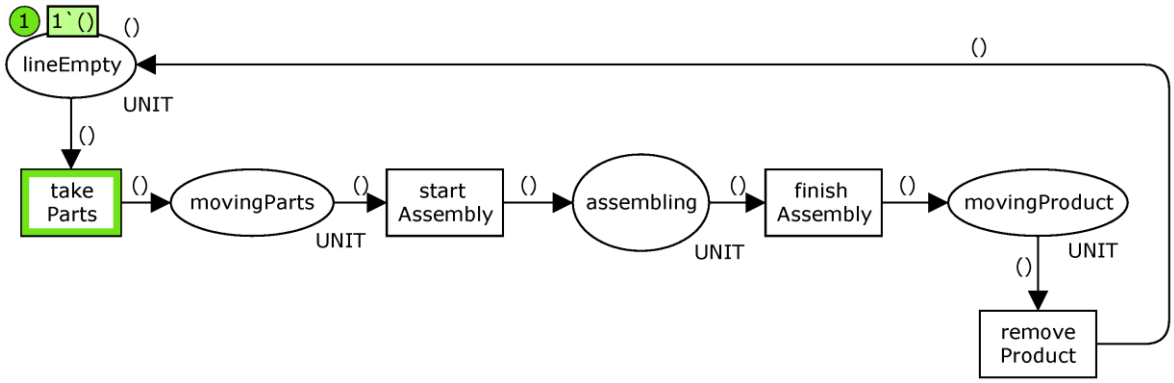


Figure 25. Assembly line part as a Petri net in its initial marking

Places and transitions of the net have the following meaning:

- `lineEmpty` – the assembly line is empty (if there is a token in this place).
- `movingParts` – the line is moving parts from its beginning to the assembly station.
- `assembling` – a product is being assembled from the parts at the assembly station.
- `movingProduct` – the line is moving the assembled product from the station to the end of the line.

- `takeParts` – the line takes a set of product parts and starts moving them towards the assembly station (when this transition fires).
- `startAssembly` – the line stops moving as the parts set reaches the station and starts the product assembly.
- `finishAssembly` – the line finishes the product assembly and starts moving the product towards its end.
- `removeProduct` – the line stops moving as the product reaches its end and the product is removed.

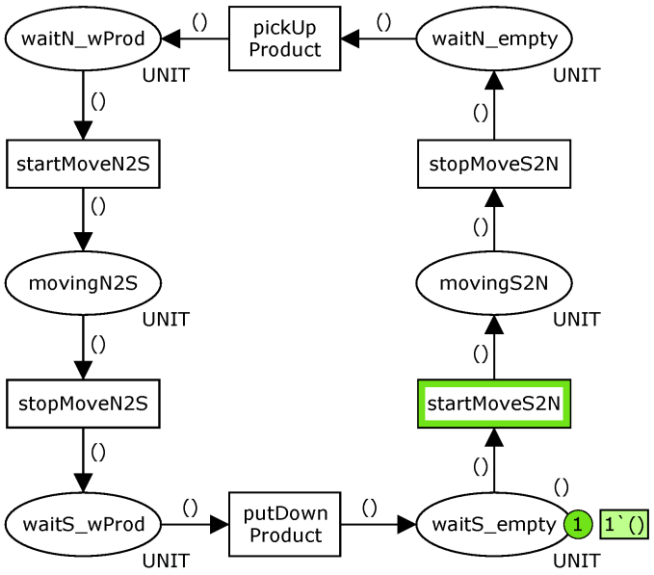


Figure 26. Crane part as a Petri net in its initial marking

In a similar way we design a net for the crane (Figure 26). Here the meaning of places and transitions is:

- `waitS_empty` – the crane is waiting above the storage box (i.e. in its south position) without holding anything (if there is a token in this place).
- `movingS2N` – the crane is moving from the box to the end of the production line (i.e. to its north position) without holding anything.
- `waitN_empty` – the crane is waiting above the end of the production line without holding anything.
- `waitN_wProd` – the crane is waiting above the end of the production line holding an assembled product.
- `movingN2S` – the crane is moving from the end of the production line to the box with the product.
- `waitS_wProd` – the crane is waiting above the storage box holding the product.

- `startMoveS2N` – the crane starts moving from south to north (when this transition fires).
- `stopMoveS2N` – the crane stops moving from south to north after it reaches the end of the line.
- `pickUpProduct` – the crane picks up an assembled product from the end of the line.
- `startMoveN2S` – the crane starts moving from north to south.
- `stopMoveN2S` – the crane stops moving from north to south after it reaches the box.
- `putDownProduct` – the crane drops the product to the box.

Now we have two nets and we need to put them together into one net. From the specification of the process it is clear that to empty the line we have to pick up the finished product by the crane. So, we combine the nets by merging the `removeProduct` transition from the first net and the

`pickUpProduct` transition from the second net. We name the resulting transition `pickUpProduct`. The next step is to add places and transitions for the parts stack. Optionally, we can also add a place which will hold finished products and thus represent the storage box.

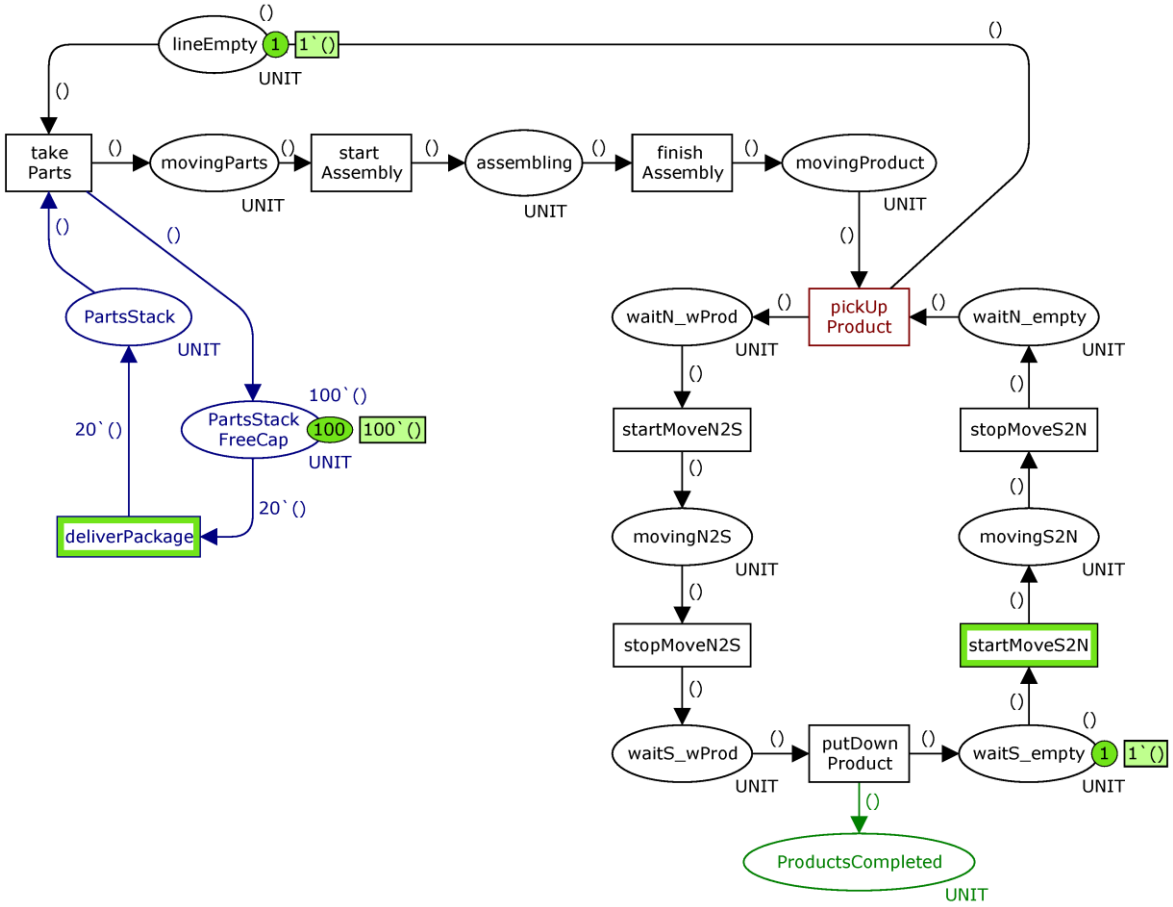


Figure 27. Petri net of the whole manufacturing process.

The final PN model of the manufacturing process can be seen in Figure 27. The merged transition `pickUpProduct` is rendered in brown and the place `ProductsCompleted` for the storage box is in green. The blue part models the parts stack together with package delivery activity. The place `PartsStack` holds tokens representing parts sets, one token for each set. The number of tokens in `PartsStackFreeCap` is equal to the place left in the parts stack, so its initial marking is 100 tokens. The transition `deliverPackage` represents delivery of a package with 20 parts sets.

Places `PartsStack`, `PartsStackFreeCap` and `ProductsCompleted` are different from other places in this net and the nets from previous examples, because they usually hold more than one tokens. So, they cannot be regarded as states (i.e. the system or its part is in a state represented by given place if the is a token in the place). Rather than that they are counters or can be seen as natural number variables.

□

In section II.3.6 we mentioned that the doctor's waiting room system from Example 5 can be modelled as an event-driven system and that PN are a suitable language for this. So, in the final example of this section we show how such Petri net looks like.

Example 11. Petri net for doctor's waiting room

Recall that the doctor's waiting room system from Example 5 consisted of a room holding patients waiting to be called by a doctor. Number of patients in the room was a (discrete) state variable and there were two kinds of events, which occurrence was able to change the state:

1. A patient arrives to the waiting room, which increases the number of patients by one.
2. A patient leaves the waiting room, which decreases the number of patients by one.

A Petri net model of this system will consist of one place (`patients`), which marking will be equal to the value of the state variable and two transitions, one (`pArrival`) for the first event and one (`pCall`) for the second. The graph of the net is shown in Figure 28.

Figure 28. Petri net of the doctor's waiting room system in its initial marking

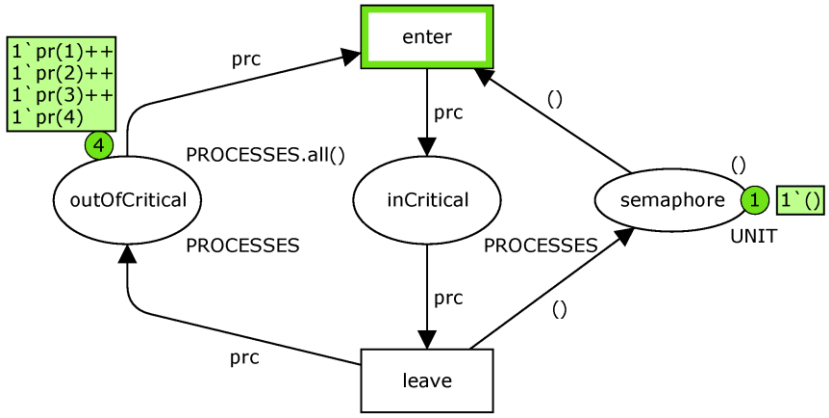
□

VI.2 Tokens with Values

As we can see from the examples presented so far, the use of undistinguishable tokens can lead to really big models, even in the case of systems of an average size. For example, to extend the MuTex PN from Example 9 by an additional process we have to add two transitions (e.g. `p3_enter`, `p3_leave`) and two places (e.g. `p3_outOfCritical`, `p3_inCritical`). This is because the only way to distinguish between processes is to use separate places and transitions for each of them. However, if we use *distinguishable*, coloured, *tokens*, then we need only one place for the processes inside the critical area and one place for the processes outside the critical area. Of course, there is a cost of using distinguishable tokens: we have to define different types (colour sets) for places, constants and variables and use more complicated arc expressions. Sometimes it is also needed to use so-called *guards* for transitions, which are additional conditions for transition firing. A significant part of CPN is *declarations*, which defines colour sets, constants, variables and functions used in given net. In CPN Tools all declarations, expressions and predicates (e.g. transition guards) are written in *CPN ML*, which is a modification of a general-purpose functional programming language *Standard ML* (*SML*). How exactly a high-level version of the mutual exclusion PN can look like is shown in Example 12. This example and Example 13 also introduce and explain some related terms, namely *multiset*, *binding*, *binding element* and *step*.

Example 12. *MuTex as CPN*

The graph of CPN for mutual exclusion of four processes is shown in Figure 29 a). Its structure is similar to a structure of a low-level PN for MuTex (as in Example 9) if only one process is involved. This time tokens of different values (colours) are used to represent individual processes. The tokens are from a colour set called `PROCESSES`, which is defined in the declarations of the CPN (Figure 29 b). The `PROCESSES` is an index colour set (this is specified by the keyword `index`), that is a colour set which individual members are identified by a common name (`pr` here) and an index value from given range. In the `PROCESSES` the range is from `1` to `prNo`, where `prNo` is an integer constant set to `4`. In the declarations we can also find a declaration of the colour set `UNIT`, which is equal to the build-in CPN ML type `unit` and a variable `prc` of the type `PROCESSES`. The variable is used in arc expressions.




```
colset UNIT =
unit;

colset PROCESSES
  = index pr with
1..prNo;

val prNo=4;

var prc:
PROCESSES;
```

a) *b)*

Figure 29. CPN for mutual exclusion of four processes in M_0: graph (a) and declarations (b).

The initial marking of the net is

$$M_0 = (1\text{`pr}(1)\text{++}1\text{`pr}(2)\text{++}1\text{`pr}(3)\text{++}1\text{`pr}(4), \emptyset, 1\text{`}())$$

provided that the order of places is `outOfCritical`, `inCritical` and `semaphore`. The expression $1\text{`pr}(1)\text{++}1\text{`pr}(2)\text{++}1\text{`pr}(3)\text{++}1\text{`pr}(4)$ defines a *multiset* with one occurrence of `pr(1)` , one occurrence of `pr(2)`, one occurrence of `pr(3)` and one occurrence of `pr(4)`. The "++" operator can be read as "and" or can be seen as a (multi)set union. A *multiset* is like a set with multiple occurrences of the same members allowed. Formal definition of multiset can be found in the next section. So, in M_0 the place `outOfCritical` holds tokens for all processes and to put these tokens into the place, the initialisation expression `PROCESSES.all()` is used. The expression means "create a multiset that contains one occurrence of each member of the colour set `PROCESSES`." The one undistinguishable token in `semaphore` means that a process can enter the critical area.

A process enters the critical area when the transition `enter` is fired and leaves it when `leave` is fired. But firing of these transitions is not that simple as in the previous nets. While the value of token removed or added from or to the `semaphore` is always the same (`()`), in the case of first two places it varies. The corresponding arc expressions consist of the variable `prc`, so the value of the token can be any value from `PROCESSES`.

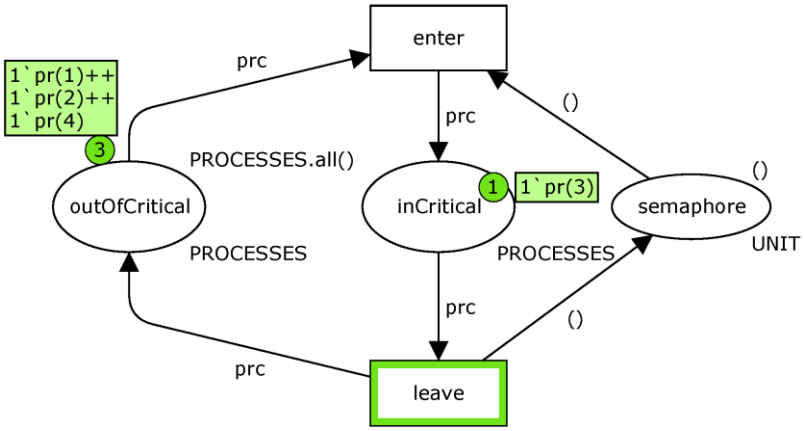


Figure 30. The net from Figure 29 in the next marking M_1.

When variables are present in the arc expressions, then there are usually various ways of firing corresponding transitions. So, it is not enough to say that a transition has been fired; we also have to add what values were assigned to the variables. In CPN terminology these assignments are called *bindings*. They are written in sharp brackets, for example $< prc = pr(1) >$. A firing is then defined by a pair, consisting of a transition and a binding. The pair is called *binding element* and is written is round brackets, for example $(leave, < prc = pr(1) >)$. In occurrence sequences we can also use binding elements, like in the next one, which leads us from M_0 to M_1(Figure 30) and back to M_0:

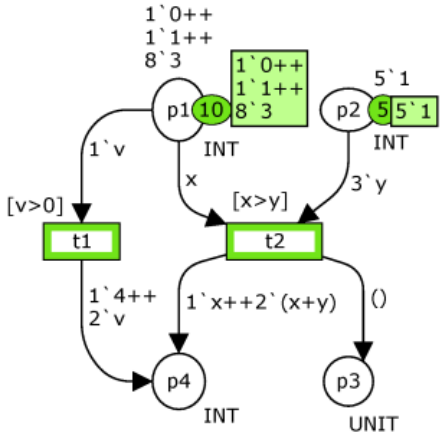
$$M_0\ [(enter, <\text{prc=pr(3)}>) >\ M_1\ [(\text{leave}, <\text{prc=pr(3)}>) > M_0$$

□

It should be noted that provided all colour sets are finite, nets with undistinguishable tokens only (i.e. P/T nets) have the same expressional power as nets with coloured tokens (CPN). The "same expressional power" means that they can describe the same class of systems. Every CPN with finite colour sets can be *unfolded* to an equivalent P/T net, where we will have one place for each combination of token values that can appear in it and one transition for each binding. CPN have the same expressional power as P/T nets, but they have greater modelling power, which means that it is easier to create, modify and understand CPN models than P/T net models. For example, to modify the MuTex net from Example 9 to capture mutual exclusion of 100 processes, we need to add 196 places, 196 transitions and 588 arcs. To do the same thing with the CPN from Example 12 we just need to change the value of `prNo` to `100`. Disadvantages of CPN are a need to use more complicated language and much more complicated formal analysis methods.

The second example shows two small CPN, where most of tokens have integer values, arc expressions are more sophisticated and transitions have guards.

Example 13. *Arc expressions and guards.*

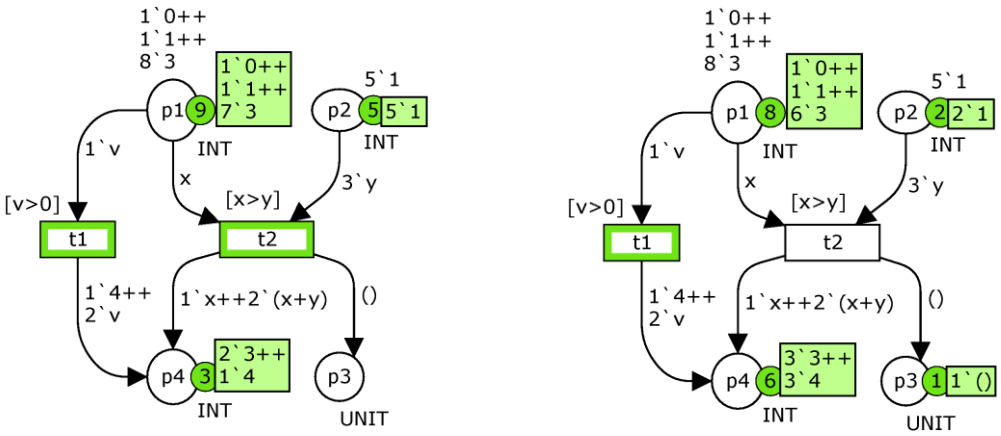



```
colset UNIT = unit;

colset INT = int;

var v,x,y : INT;
```

a) *b)*

Figure 31. A CPN in M_0: graph (a) and declarations (b).

The first net, consisting of two transitions, `t1` and `t2`, and four places, is shown in Figure 31. Place `p3` holds undistinguishable tokens (i.e. of colour set `UNIT`), tokens in all other places have integer values (colour set `INT`). The transition `t1` can fire if and only if there is at least one token with a value greater than `0` in `p1`. The constraint on the token value is expressed by the guard `[v>0]`. Firing of `t1` removes one token of chosen value from `p1` and add two tokens of the same value and one token of value `4` to the place `p4`. In the initial marking $M_0 = (1`0{+}{+}1`1{+}{+}8`3, 5`1, \emptyset, \emptyset)$ (ordering of places is from `p1` to `p4`) the transition `t1` can fire with binding $< v = 1 >$ or $< v = 3 >$. The expression $< v = 0 >$ is not even considered a binding of `t1`, because it doesn't satisfy the guard `[v>0]`. The result of firing (t1,<v=3>) is $M_1 = (1`0{+}{+}1`1{+}{+}7`3, 5`1, \emptyset, 2`3{+}{+}1`4)$, shown in Figure 32 a).



a) *b)*

Figure 32. The net from Figure 31 in the next marking M_1 (a) and the next next marking M_2 (b).

There is only one binding for a firing of `t2` in M_0 or M_1: <x=3,y=1>. And firing of the binding element (t2,<x=3,y=1>) in M_1 results in M_2 (Figure 32 b). The firing of `t2` adds three tokens to `p4`: one of the value of `x` and two of the value equal to the sum of `x` and `y`. It also adds one token to `p3`. The complete occurrence sequence, shown in Figure 31 and Figure 32 is

$$M_0[(t1,<v=3>)> M_1\ [(t2,<x=3,y=1>)> M_2$$

The CPN theory allows us to fire more than one binding element at once, in so-called *steps*. These steps are in fact multisets over binding elements and are written in the same style as markings of places. For example, a step Y_1,

$$Y_1 = 1`(t1,<v=1>)++2`(t1,<v=3>)++1`(t2,<x=3,y=1>))$$

is enabled in M_0 and its firing in M_0 leads to M_3 (we write $M_0\ [Y_1>M_3$), where

$$M_3 = (1`0++5`3, 2`1, 1`(), 2`1++5`3++5`4)$$

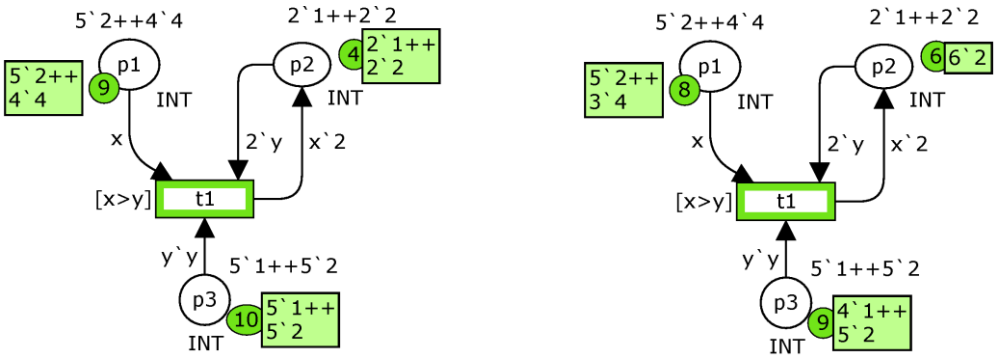


Figure 33. A graph of a CPN in M_0 (a) and M_1 (b).

The second net (Figure 33) shows that values of tokens can define a number of other added and removed tokens. The net is shown before (M_0) and after (M_1) firing the step Y,

$$Y = 1`(t1,<x=4,y=1>))$$

□

VI.3 Undistinguishable Tokens in CPN Tools

In the examples above we used the `UNIT` colour set to represent the traditional undistinguishable tokens. But we can use any unique value to represent this kind of tokens, using `UNIT` is just the recommended way. In fact, `UNIT` was introduced just few years ago, before it a colour set named `E` holding a value `e` had been the standard. We can still find the `E` colour set in some examples shipped with CPN Tools, such as the distributed database. Even the handling of `UNIT` tokens has changed recently. From the version 4.0 of CPN Tools we can use a simplified notation in expressions: when only one token is used we just write "`1`" or nothing (in arc

expressions), when more than one token is used, we write number of tokens only (i.e. "5" instead of "5`()"). The simplified way was introduced to make inscription of the undistinguishable tokens similar to low-level Petri nets. All three ways of handling undistinguishable tokens are illustrated in Figure 34.

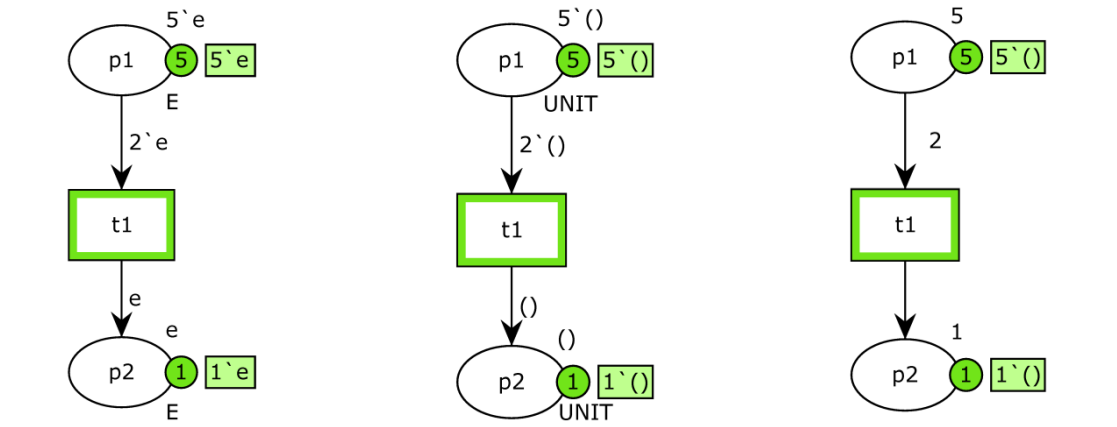


Figure 34. *A CPN fragment with undistinguishable tokens: a version using the E colour set (a), a version with the* `UNIT` *colour set (b) and a version with the simplified notation for* `UNIT` *(c).*

Unless stated otherwise we use the notation as in Figure 34 b) in this chapter.

VI.4 Definition of CPN Structure and Behaviour

Above we introduced and explained terms related to Coloured Petri nets informally, using examples. In this section we provide basic formal definitions of CPN structure and behaviour, which correspond to the "official" definitions that can be found in (Jensen, 1994), (Jensen, 1997a-c) or (Jensen & Kristensen, 2009). The definitions make use of the following symbols and notations:

- $\mathbb{N}$ – the set of natural numbers including zero,
- $\mathbb{B}$ – the boolean type, $\mathbb{B} = \{true, false\}$,
- $Type(v)$ – the type of a variable v,
- $Type(expr)$ – the type of an expression $expr$,
- $Type(A)$ – the type of variables in a set A, $Type(A) = \{Type(v) | v \in A\}$
- $Var(expr)$ – the set of variables in an expression $expr$,
- $expr<b>$ –the value obtained by evaluating an expression $expr$ in a binding b. It is required that $Var(expr) \subseteq Var(b)$.

We start with the definition of the multisets (Definition 1), which, as we have already seen, can be used to define markings, arc expressions and steps.

Definition 1. *Multiset*

A multiset m over a non-empty set S, *is a function* $m: S \to \mathbb{N}$, *represented as a formal sum (61).*

$$\sum_{s \in S} m(s)`s \qquad (61)$$

By S_{MS} *we denote the* set of all multi-sets *over* S. *The non-negative integers* $\{ m(S) | s \in S \}$ *are the* coefficients *of the multi-set.*

We say, that s belongs to m $(s \in m)$ *if and only if* $m(s) > 0$. *An* empty multiset is denoted as $\emptyset$.

□

The classical sets can be seen as a class of multisets, where all coefficients are equal to one. Several operations can be defined for multisets. They are given in Definition 2.

Definition 2. *Operations for multisets.*

Let $m, m_1, m_2 \in S_{MS}$ *and* $n \in \mathbb{N}$. *Then standard multiset operations and predicates are defined as follows:*

1. Addition. $m_1 + m_2 = \sum_{s \in S}(m_1(s) + m_2(s))`s$.
2. Scalar multiplication. $n * m = \sum_{s \in S}(n * m(s))`s$.
3. Size. $|m| = \sum_{s \in S} m(s)$.
4. Comparison. $m_1 = m_2 \equiv \forall s \in S: m_1(s) = m_2(s)$.
   $m_1 \neq m_2 \equiv \exists s \in S: m_1(s) \neq m_2(s)$
   $m_1 \leq m_2 \equiv \forall s \in S: m_1(s) \leq m_2(s)$
   ($\geq, <, >$ are defined in the similar way.)
5. Subtraction (defined only for $m_1 \geq m_2$). $m_1 - m_2 = \sum_{s \in S}(m_1(s) - m_2(s))`s$.

□

A multiset m with the size $|m| = \infty$ is *infinite,* m with the size $|m| < \infty$ is *finite*. Multisets and their operations are illustrated by the next example.

Example 14. *Multisets and operations for multisets.*

Suppose that we have a set R,

$$R = \{2,4,5,8,9\}.$$

Then multisets from R_{MS} can be, for example, $r_1$and r_2:

$$r_1 = 1`2 + 2`4 + 8`9, \qquad r_2 = 3`2 + 2`4 + 2`5 + 3`8 + 11`9 \qquad (62)$$

The multiset $r_1$is a mapping $r_1: R \to \mathbb{N}$, where

$$r_1(2) = 1, r_1(4) = 2,\ r_1(5) = 0, r_1(8) = 0, r_1(9) = 8$$

Similarly, for r_2 we have

$$r_2(2) = 3, r_2(4) = 2,\ r_2(5) = 2, r_2(8) = 3, r_2(9) = 11.$$

The size of r_1 is

$$|r_1| = 1 + 2 + 8 = 11,$$

for r_2 we have $|r_2| = 21$. Both multisets are finite and $r_1 \leq r_2$. We can apply all three remaining operations to r_1 and r_2:

$$r_1 + r_2 = 4`2 + 4`4 + 2`5 + 3`8 + 19`9$$
$$5 * r_1 = 5`2 + 10`4 + 40`9$$
$$r_2 - r_1 = 2`2 + 2`5 + 3`8 + 3`9$$

□

We already encountered multisets in Example 12 and Example 13. But they differ from (62) in using the "++" operator instead of "+". In fact, it is the same operator (addition), but in CPN Tools we use "++", because "+" is reserved for the "normal", numerical, addition. After introducing multisets we are ready to define (the structure of) CPN.

Definition 3. *Coloured Petri net.*

A Coloured Petri net *is a 9-tuple*

$$CPN = \{\Sigma, P, T, A, N, C, G, E, I\},$$

where

- Σ *is a finite* set of *non-empty* types (colour sets),
- P *is a finite* set of places*,*
- T *is a finite* set of transitions *such that* $P \cap T = \emptyset$*,*
- A *is a finite* set of arcs *such that* $P \cap A = T \cap A = \emptyset$*,*
- N *is a node function,* $N: A \to P \times T \cup T \times P$*,*
- C *is a colour function,* $C: P \to \Sigma$*,*
- G *is a guard function,* $G: T \to GExpr$*, where* $GExpr$ *is a set of expressions such that*
$$\forall t \in T: \left(Type\big(G(t)\big) = \mathbb{B} \ \wedge\ Type\left(Var\big(G(t)\big)\right) \subseteq \Sigma\right),$$
- E *is an arc expression function,* $E: A \to AExpr$*, where* $AExpr$ *is a set of expressions such that*
$$\forall a \in A: \left(Type\big(E(a)\big) = C(p)_{MS}\ \wedge\ Type\left(Var\big(E(a)\big)\right) \subseteq \Sigma\right),$$
where p *is the place in* $N(a)$*.*
- I *is an initialisation function,* $I: P \to IExpr$*, where* $IExpr$ *is a set of closed expressions such that:*
$$\forall p \in P: Type\big(I(p)\big) = C(p)_{MS}.$$

□

A *closed expression* is an expression with no variables. The meaning of the sets and functions in the CPN definition is explained in the next example, which presents specifications of the nets from Example 12 and Example 13.

Example 15. *CPN definition.*

The MuTex CPN from Figure 29 a) can be specified according to Definition 3 as follows:

$$MuTex = \{\Sigma_{MT}, P_{MT}, T_{MT}, A_{MT}, N_{MT}, C_{MT}, G_{MT}, E_{MT}, I_{MT}\}, \quad (63)$$

$$\Sigma_{MT} = \{PROCESSES, UNIT\} \quad (64)$$

$$P_{MT} = \{outOfCritical, inCritical, semaphore\} \quad (65)$$

$$T_{MT} = \{enter, leave\} \quad (66)$$

$$A_{MT} = \{a_1, a_2, a_3, a_4, a_5, a_6\} \quad (67)$$

$$N_{MT} = \{(a_1, (outOfCritical, enter)), (a_2, (enter, inCritical)), (a_3, (inCritical, leave)), (a_4, (leave, outOfCritical)), (a_5, (leave, semaphore)), (a_6, (semaphore, enter))\} \quad (68)$$

$$C_{MT} = \{(outOfCritical, PROCESSES), (inCritical, PROCESSES), (semaphore, UNIT)\} \quad (69)$$

$$G_{MT} = \{(enter, true), (leave, true)\} \quad (70)$$

$$E_{MT} = \{(a_1, prc), (a_2, prc), (a_3, prc), (a_4, prc), (a_5, ()), (a_6, ())\} \quad (71)$$

$$I_{MT} = \{(outOfCritical, PROCESSES.all()), (inCritical, \emptyset), (semaphore, 1`())\} \quad (72)$$

The text "MT" is added to the underscore of names of all sets and functions to distinguish between the general definition and the definition of MuTex CPN. The functions ((68) to (72)) are defined as sets of pairs, so, for example, (70) means that $G_{MT}(enter) = true$ and $G_{MT}(leave) = true$).

The first three parts are really easy to identify in the graph of the net: Σ_{MT} (64) lists the two colour sets (types) used in the net, P_{MT} (65) contains the places and T_{MT} (66) the transitions of the net.

The arcs listed in A_{MT} (67) don't appear under these names in the graph, but the node function N_{MT} clearly identifies them. For example, the third member of N_{MT} (68) is $(a_3(inCritical, leave))$, so a_3 is the arc from the place `inCritical` to the transition `leave`.

The colour function C_{MT} (69) assigns colour sets to places and the guard function G_{MT} (70) guards to transitions. The transitions in the MuTex CPN have no guards, so all values in G_{MT} are just $true$. The arc expression function E_{MT} (71) assigns arc expressions to the arcs of the net and, finally, the initialisation function I_{MT} (72) assigns expressions that define initial marking to the places.

If we compare the specification in (63) - (72) to Figure 29, we will find one significant difference: There are no declarations in the specification. The variables and colour sets are

named, but they are not defined. So, to make the textual specification complete, we should add the declarations from Figure 29 b) to it. For the net from Figure 31 we get the following specification:

$$
\begin{aligned}
&MuTex = \{\Sigma_{EX}, P_{EX}, T_{EX}, A_{EX}, N_{EX}, C_{EX}, G_{EX}, E_{EX}, I_{EX}\},\\
&\Sigma_{EX} = \{INT, UNIT\}\\
&P_{EX} = \{p1, p2, p3, p4\}\\
&T_{EX} = \{t1, t2\}\\
&A_{EX} = \{a_1, a_2, a_3, a_4, a_5, a_6\}\\
&N_{EX} = \{(a_1, (p1, t1)), (a_2, (p1, t2)), (a_3, (p2, t2)), (a_4, (t1, p4)),\\
&\qquad (a_5, (t2, p4)), (a_6, (t2, p3))\}\\
&C_{EX} = \{(p1, INT), (p2, INT), (p3, UNIT), (p4, INT)\}\\
&G_{EX} = \{(t1, v > 0), (t2, x > y)\}\\
&E_{EX} = \{(a_1, 1`v), (a_2, x), (a_3, 3`y), (a_4, 1`4 + 2`v),\\
&\qquad (a_5, 1`x + 2`(x + y)), (a_6, ())\}\\
&I_{EX} = \{(outOfCritical, PROCESSES.all()),\\
&\qquad (inCritical, \emptyset), (semaphore, 1`())\}
\end{aligned}
$$

To illustrate other notations: $Var(\mathrm{E_{EX}}(a_5)) = \{x, y\}$ and $Type(\mathrm{E_{EX}}(a_4)) = INT_{MS}$.

□

Before defining the behaviour of CPN we, similarly to (Jensen, 1994), need to introduce some additional notation for all $t \in T$and all pairs of nodes $(x_1, x_2) \in (P \times T \cup T \times P)$:

- $A(t)$ – a set of arcs adjacent to a transition t, $A(t) = \{a \in A | N(a) \in P \times \{t\} \cup \{t\} \times P\}$,
- $Var(t)$ – a set of variables that occur in the guard of t, $t \in T$, and in the expressions of arcs adjacent to t, $Var(t) = \{v | v \in Var(G(t)) \vee \exists a \in A(t): v \in Var(E(a))\}$,
- $A(x_1, x_2)$ – a set of arcs from a node (i.e. a transition or a place) x_1 to x_2, $A(x_1, x_2) = \{a \in A | N(a) = (x_1, x_2)\}$ and
- $E(x_1, x_2)$ – an expression, representing a multiset of tokens, that is a sum of all expressions on the arcs from x_1 to x_2, $E(x_1, x_2) = \sum_{a \in A(x_1, x_2)} E(a)$.

The first behaviour-related definition is about binding.

Definition 4. *Binding.*

A binding of a transition t *is a function* b: $Var(t) \rightarrow \bigcup_{v \in Var(t)} Type(v)$ *for which it holds that:*

1. $\forall v \in Var(t): b(v) \in Type(v)$
2. $G(t)<b>=true$

By $B(t)$ *we denote the set of all bindings for* t.

□

The first condition of Definition 4 tells us that a value assigned to each variable have to be from the type of the variable and the second condition that the guard of the corresponding transition

has to be true for the values of $Var(t)$. The bindings are usually written in sharp brackets, as we have already seen in Example 12 and Example 13. The next two definitions formalize the meaning of token elements, markings, binding elements and steps.

Definition 5. *Token element and marking.*

A token element *is a pair* (p, c)*, where* $p \in P$ *and* $c \in C(p)$*. The* set of all token elements *is denoted by* TE.

A marking *is a multi-set over* TE*. The sets of all markings is denoted by* $\mathbb{M}$*. The* initial marking M_0 *is the marking obtained by evaluating the initialisation expressions:*

$$\forall (p, c) \in TE: M_0(p, c) = (I(p))(c)$$

□

An example of a marking written according to Definition 5 is (73), where token elements are $(p1, 0)$, $(p1, 1)$ and so on.

$$M_0 = 1`(p1,0)++1`(p1,1)++8`(p1,3)++5`(p2,1) \qquad (\ 73\ )$$

The multiset (73) is in fact the initial marking of the first net from Example 13, depicted in Figure 31. But the form in which M_0 is written in Example 13 (and other examples) differs from (73). We can say that the form used in the examples is typical for Petri nets, while the form from Definition 5 is not. Both forms are useable for CPN and formally the "typical" form is defined by a function $M^*: P \to \Sigma_{\mathrm{MS}}$,such that

$$\forall p \in P: \left(M^*(p) \in C(p)_{MS} \wedge \left(\forall c \in C(p): \left(M^*(p)\right)(c) = M(p, c)\right)\right)$$

Then the second form, according to (73), is

$$M_0^*(p1) = 1`0++1`1++8`3, M_0^*(p2) = 5`1, M_0^*(p3) = M_0^*(p4) = \emptyset$$

and $\left(M_0^*(p1)\right)(0) = M(p1,0) = 1$, $\left(M_0^*(p1)\right)(1) = M(p1,1) = 1$, $\left(M_0^*(p1)\right)(3) = M(p1,3) = 8$, $\left(M_0^*(p2)\right)(1) = M(p2,1) = 5$ (in other cases the value is 0). To simplify the notation both M^* and M are denoted as M (Jensen, 1994) and we do the same here.

Definition 6. *Binding element and step.*

A binding element *is a pair* (t, b)*, where* $t \in T$ *and* $b \in B(t)$. The set of all binding elements *is denoted by* BE.

A step *is a non-empty and finite multi-set over* BE.

□

Inscription of binding elements and steps has been already demonstrated in Example 12 and Example 13. Now we have everything necessary for the definition of CPN behaviour, namely for defining enabling and firing of steps.

Definition 7. *Enabling and firing of steps. Direct reachability.*

A step Y is enabled *in a marking* M *(denoted* $M[Y>$ *) if and only if the following property holds:*

$$\forall p \in P: \sum_{(t,b)\in Y} E(p,t)<b> \leq M(p)$$

We also say that (t,b) *(denoted* $M[\,(\mathrm{t,b})>$ *)* is enabled *and* t is enabled *(denoted* $M[t>$ *). In addition, we say that* the elements of Y are concurrently enabled *(provided* $|Y| \geq 0$*).*

When a step Y *is enabled in a marking* M_1 *it may* fire *(*occur*), changing* M_1 *to another marking* M_2*, defined for all* $p \in P$ *as:*

$$\forall p \in P: M_2(p) = \left(M_1(p) - \sum_{(t,b)\in Y} E(p,t)<b> \right) + \sum_{(t,b)\in Y} E(t,p)<b>$$

We say that $M_2$is directly reachable from M_1 *and write it as* $M_1\,[Y> M_2$.

□

Firings and steps have been already presented in Example 7 to Example 13, here we demonstrate the computation of a new marking according to Definition 7 on the firing $M_1\,[(\mathrm{t2,<x=3,y=1>})> M_2$ from the first net in Example 13. Recall that the firing changed the marking

$$M_1 = (1`0+1`1+7`3, 5`1, \emptyset, 2`3+1`4)$$

to

$$M_2 = (1`0+1`1+6`3, 2`1, 1`(), 3`3+3`4).$$

As an example we take $\mathrm{p2}$, where

$$M_2(p2) = \ (M_1(p2) - 3`1) + \emptyset = \ (5`1 - 3`1) = 2`1$$

and $\mathrm{p4}$, where

$$M_2(p4) = \ (M_1(p4) - \emptyset) + (1`3 + 2`4) = (2`3+1`4) + \ (1`3 + 2`4) = 3`3+3`4.$$

The final definition of this section extends the notion of occurrence and reachability from Definition 7.

Definition 8. *Occurrence sequence. Reachability.*

A finite occurrence sequence *is a sequence of markings and steps*

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

such that $n \in \mathbb{N}$ *and*

$$\forall i \in \{1, \dots, n\}: M_i[\, Y_i > M_{i+1}$$

M_1 *is the* start marking*,* M_{n+1}*is the* end marking *and* n *is the* length *of the sequence.*

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

$$M_1[\, Y_1 > M_2[\, Y_2 > \, \dots,$$

where $\forall i \in \{1, \dots\}: M_i[\, Y_i > M_{i+1}$.

A marking M_b *is* reachable *from a marking* M_a *if and only if there exists a finite occurrence sequence starting in* M_a *and ending in* M_b*. The set of markings which are reachable from* M_a *is denoted by* $[M_a>$*. A marking is* reachable *if and only if it belongs to* $[M_0>$*.*

□

Occurrence sequences of zero length are allowed, so $\forall M \in [M_0>: M \in [M>$. It should be noted that a number of properties have been defined for CPN, for example:

- *Boundedness:* Let $p \in P$, $n \in \mathbb{N}$ and $m \in C(p)_{MS}$ be given. Then n is an *integer bound* for p if and only if $\forall M \in [M_0>: |M(p)| \leq n$ and m is a *multi-set bound* for p if and only if $\forall M \in [M_0>: M(p) \leq m$.
- *Home properties:* Let a marking $M \in \mathbb{M}$ and a set of markings $X \subseteq \mathbb{M}$ be given. Then M is *a home marking* if and only if $\forall M' \in [M_0>: M \in [M'>$ and X is a *home space* if and only if $\forall M' \in [M_0>: X \cap [M'> \neq \emptyset$.
- Liveness properties: Let a marking $M \in \mathbb{M}$ and a set of binding elements $X \subseteq BE$ be given. Then M is *dead* (a *deadlock marking*) if and only if no binding element is enabled in M (i.e. $\forall x \in BE: \neg M[x>$) and X is *dead in* M if and only if no element of X can become enabled (i.e. $\forall M' \in [M_0> \forall x \in X: \neg M'[x>$. X is *live* if and only if there is no reachable marking in which X is dead.

Whether a net has these properties can be verified using analytical methods such as state space or invariant analysis.

VI.5 Timed CPN

CPN as we know them so far can be used to specify functionality of systems, but without any explicit representation of time. This is satisfactory, if our goal is to validate or verify basic, functional properties, like the ones listed at the end of the previous section. But for dynamic simulation models we need some representation of time. For example, we need to describe somehow that an execution of an event, represented by a firing of some transition, takes some time. Fortunately, CPN theory and CPN Tools allow to incorporate time into models. Nets, which do this are called *Timed CPN*. To distinguish between the time in the real world and the time in CPN models, we call the latter one *simulated time*. The *simulated time* is a symbolic representation of time during simulation and has a form of a natural number. It has the value 0 at the beginning of the simulation and its advance depends on the simulated model. Timed CPN differs from "ordinary" CPN in the following features:

1. A value called *timestamp* can be associated with tokens. When a token has timestamp, it is written after the ordinary value of the token. The value and the timestamp are separated by "`@`" symbol. For example "`2@5`" is a token with value `2` and timestamp `5`. The timestamp of a token represents the time when the token has been created.
2. *Colour sets* can be *timed*. A timed colour set is an ordinary colour set, where each value includes a timestamp. Tokens, which include timestamps are always members of some timed colour sets. The keyword `timed` is used to declare that a colour set is timed. So, the value "`2@5`" can be a member of timed integer colour set, declared as
   "`colset INTtm = int timed;`".
3. Arc expressions may include so-called *delay expressions*, which define timestamps of newly created tokens. These expression have the form "`@+expr`", where `expr` is an arithmetic expression of the integer type.
4. A delay expression can be also associated with a transition, as so-called transition *time inscription*. It has the same effect as if the expression is added to each outgoing arc of the transition.
5. Delay expressions can also be a part of the initialization expressions of places, whose types are timed colour sets. Here they define timestamps of tokens in M_0.

Enabling and firing in timed CPN is elegantly defined by introducing the notion of *token availability*.

Definition 9. *Token availability.*
A token $v@s$ is available at simulated time $tsim$ if and only if $tsim \geq s$.

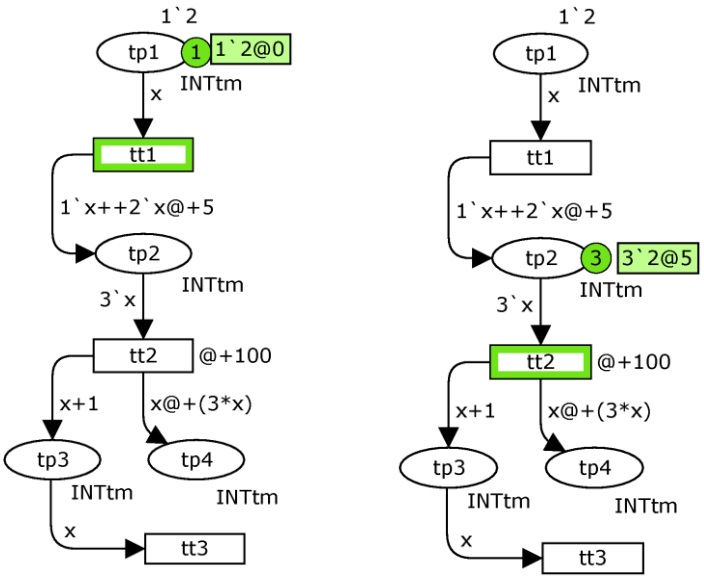
This means that all what was said in Definition 8 and Definition 9 remains valid, but only available tokens are taken into account. And tokens generated by firings have timestamps computed by corresponding delay expressions. All tokens without timestamps are regarded as tokens with timestamps equal to 0, so they are always available. *Simulation of timed CPN* then proceeds according to the following algorithm:

Step 1. Set the value $tsim$ of simulated time to 0.
Step 2. If there is no enabled transition, go to Step 4.
Step 3. Pick up one enabled transition and fire it. Go to Step 2.
Step 4. Set the value $tsim$ to the lowest value $tsim'$ such that there is an enabled transition at $tsim'$. Go to Step 2. If such $tsim'$ cannot be found, end simulation.

Of course, a user can interrupt the simulation anytime, specify a number of firings in given simulation run or specify other conditions of simulation run termination. Specification and simulation of timed CPN is illustrated by Example 16.

Example 16. *Timed CPN.*

```
colset
  INTtm = int timed;

var
  x : INT;
```



tsim=0 *tsim=5*

a) *b)* *c)*

Figure 35. A timed CPN in $M_0$and M_1: declarations (a), graph in M_0 (b) and in M_1 (c).

A simple timed CPN in its initial marking is shown in Figure 35 b). All places hold tokens of colour set `INTtm` (Figure 35 a), which means that their values consist of integer values and timestamps. In M_0 only the place `tp1` holds a token. Because the initialisation expression $I(tp1) = 1`2$ doesn't contain a delay expression the token in `tp1` has the timestamp equal to $0$. The transition `tt1` is enabled in $M_0$ in the simulated time $tsim = 0$ as the required token in `tp1` is already available. After (tt1,<x=2>) is fired the net appears in the marking M_1 (Figure 35 c) and the simulated time advances to 5, because it is the nearest time point where any transition is enabled again. The firing of (tt1,<x=2>) produces three tokens of the value 2 and timestamp 5 ($3`2@5$), despite the fact that from the expression associated with the arc from tt1 to tp2 one can get an impression that it will be $1`2@0++2`2@5$. This is because when we have an arc expression with more than one token values and only one delay expression then the delay expression is applied to all tokens. We can produce tokens with different timestamps for the same place, but then each token value in given arc expression after the first delay expression have to have its own delay expression. So, for example

$$1`x \;++\; 2`x@ + 5 \;++\; 4`x \;++\; 3`(x + 1)@ + 10$$

is illegal, because there is no delay expression after $4`x$ (no delay expression after $1`x$ is OK). The correct version will be

$$1`x \;++\; 2`x@ + 5 \;++\; 4`x@ + 10 \;++\; 3`(x + 1)@ + 10,$$

which produces the following multiset of tokens:

$$3`2@5 \;++\; 4`2@10 \;++\; 3`3@10.$$

The transition enabled in M_1 is `tt2`. Delay expressions are associated with both `tt2` and the arc from `tt2` to `tp4`, so when (tt2,<x=2>) fires in M_1 and $tsim = 5$, the new token in `tp3` will have the timestamp $5 + 100 = 105$ and the new one in `tp4` will have the timestamp $5 + 100 + 3 * 2 = 111$. After firing (tt2,<x=2>) $tsim$ advances to 105 and marking changes to M_2 (Figure 36 a). In M_2 (tt3,<x=3>) can be fired and after the firing the marking changes to M_3 (Figure 36 b). The simulation of the net ends here as there will be no enabled transitions in the future.

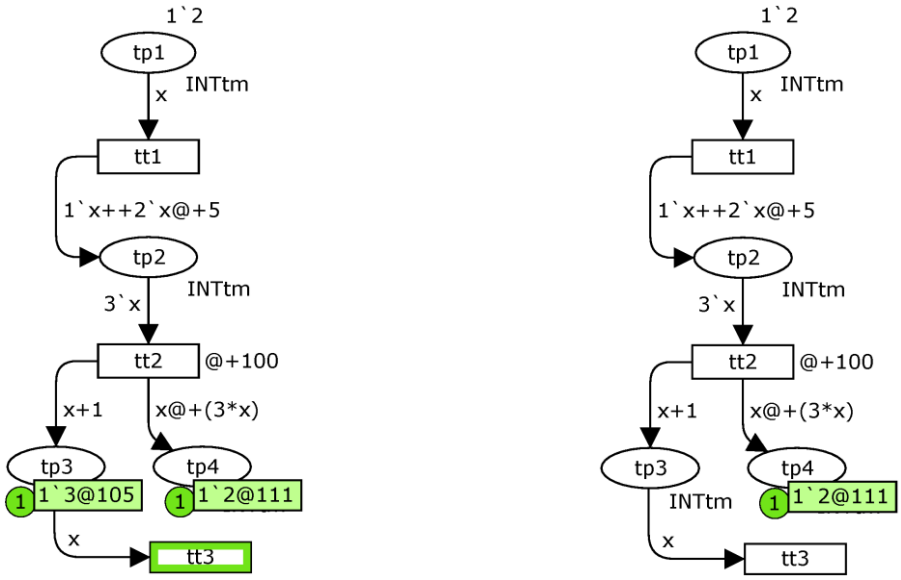


Figure 36. The timed CPN from Figure 35 in M_2 (a) and M_3 (b).

□

What was said about arc expressions is true for output arcs (i.e. arcs from transitions to places). But delay expressions can also be used on input arcs (i.e. to transitions). Here they are called *pre-empting time stamps* and allow to use tokens before they become "officially" available in given net. For example, take the net fragment in Figure 37. Here `t1` can be fired in $tsim = 5$ despite the fact that tokens in `p1` have the timestamp 10. This is thank to the expression "@+5".

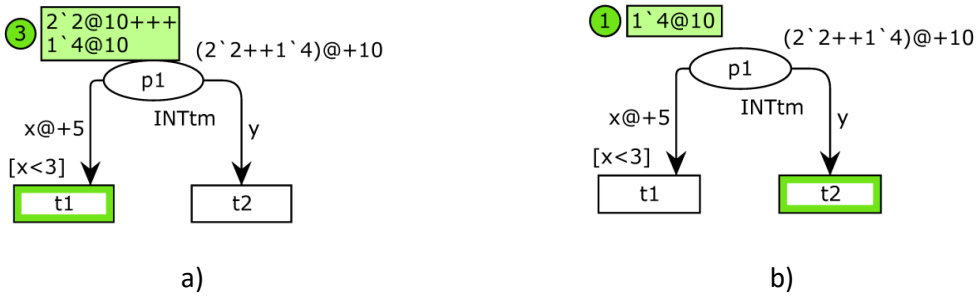


Figure 37. A CPN fragment with pre-empting time stamps in tsim=5 (a) and tsim=10 (b).

The colour set used in Figure 37 is declared as `colset INTtm = int timed;`.

VI.6 Queuing System as Timed CPN

In section V queuing systems were introduced as a particular kind of discrete systems. All components of queuing systems can be also modelled by high-level PN, such as timed CPN. This is demonstrated by Example 17, which presents a CPN model of simple single queue queuing system with a limited capacity FIFO queue and one server. It is inspired by one of the examples shipped with the CPN Tools.

Example 17. *Simple queuing system as timed CPN.*

The graph of timed CPN that represents a simple queuing system with one server serving customers is shown in Figure 38 and its declarations in Figure 39. In the Kendall's classification it can be designated as *M/G/1/FIFO/51*, because

- The time between two subsequent job arrivals (i.e. the interarrival time) has an exponential distribution (*M*), here with the rate (or intensity) $\lambda = \frac{1}{5}$ or mean $\frac{1}{\lambda} = 5$.
- The time to process a job has some general random distribution with known mean and variance (*G*). In this case it is the normal distribution with mean $\mu = 6$ and variance $\sigma = 3$.
- There is one server (*1*).
- The queue is a classical *FIFO* queue.
- The queue capacity is 50 and the server can serve one customer at once, so the system capacity is *51*.
- The population size is not specified because it is unlimited.

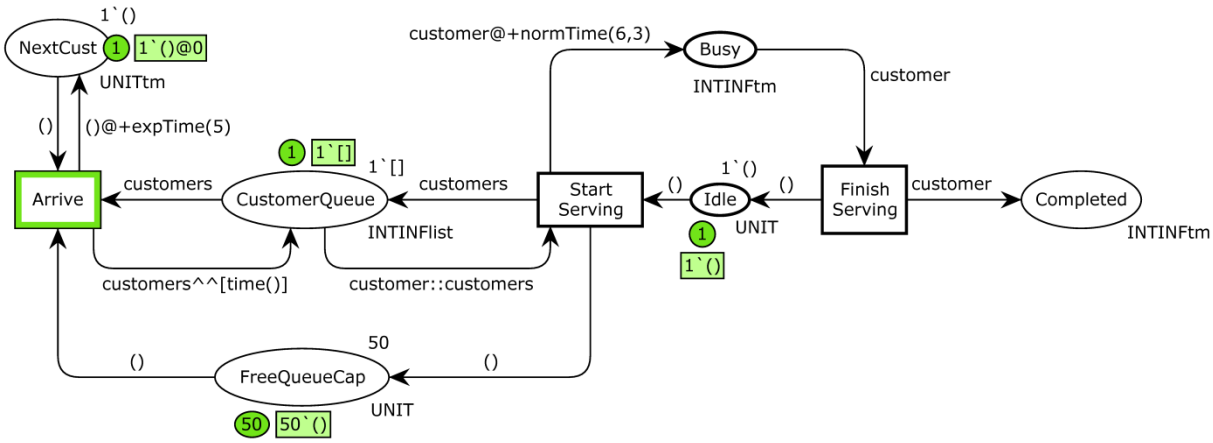


Figure 38. Graph of a CPN representing simple queuing system in M_0.

The *customers' arrival* is modelled by the place `NextCust` and the transition `Arrive`. In the initial marking the place `NextCust` hosts one token with the timestamp 0. If there is an available token in `NextCust` and at least one token in `FreeQueueCap`, `Arrive` can be fired. Firing of `Arrive` represents arrival of a new customer. We can see that the arrival pattern is

implemented as a delay expression on the arc from `Arrive` to `NextCust`, which computes the timestamp of a newly created token in `NextCust` as current time plus and integer value that is (approximately) exponentially distributed with a mean equal to 5. The value is computed by the function `expTime`, which declaration is in Figure 39. The function uses CPN ML build-in function `exponential` and surrounding code is about conversion from integer to real and back. The function `normTime`, used to generate random numbers from a Gaussian distribution, is defined in similar way.

```
colset UNIT = unit;
colset INTINF = intinf;
colset UNITtm = unit timed;
colset INTINFtm = intinf timed;
colset INTINFlist = list INTINF;

var customers: INTlist;
var customer: INT;

fun expTime (mean: int) = let
   val realMean = Real.fromInt mean
   val rv = exponential((1.0/realMean))
in floor (rv+0.5)
end;

fun normTime (mean: int, variance: int) = let
   val realMean = Real.fromInt mean
   val realVar =  Real.fromInt variance
   val rv = normal(realMean,realVar)
in floor (rv+0.5)
end;
```

Figure 39. Declarations of the simple queuing system CPN from Figure 38.

The *queue* is composed of places `CustomerQueue` and `FreeQueueCap` (and adjacent arcs). The place `CustomerQueue` holds only one token in every marking. This token is a list of values from the colour set `INTINF` and represents the customers waiting in the queue. The colour set `INTINF` is an integer type without limits. The function `time()` returns an actual simulated time (as a value from `INTINF`), the operator "`^^`" is concatenation of lists and "`::`" in "`h::s`" is a list composed of a head `h` (a value) and a tail `s` (a list). Firing of `Arrive` adds a value to the end of the list in `CustomerQueue` and firing of `Start_Serving` removes a value from the head (front) of the list. To illustrate working of these operators consider an occurrence sequence

$$M_a\ [(Arrive, \langle \text{customers}=[19,20] \rangle) >\ M_b,$$

$$M_b\ [(Start\_Serving, \langle \text{customer}=19, \text{customers}=[20,27] \rangle) >\ M_c$$

The firing of `Arrive` occurs in M_a,

$$M_a(CustomerQueue) = [19,20]$$

at $tsim = 27$ and results in M_b,

$$M_b(CustomerQueue) = [19,20]\text{^^}[27] = [19,20,27].$$

The subsequent firing of `Start_Serving` removes the first token from `CustomerQueue` and we will have

$$M_c(CustomerQueue) = [20,27]$$

because

$$[19,20,27] = 19 :: [20,27].$$

The place `FreeQueueCap` and adjacent arcs are used to limit the capacity of the queue to 50, the number of tokens in this place is equal to the number of free slots in the queue. The consequence of this solution is that the interarrival time follows the defined distribution only when the queue is not full.

The part of the net consisting of places `Idle` and `Busy` and transitions `StartServing` and `Finish_Serving` is a *server* providing the *service* of this system to customers. A token in `Idle` means that the server is unoccupied, a token in `Busy` that a customer is being served. The delay expression on the arc from `Start_Serving` to `Busy` defines the service pattern: a time needed to serve a customer is an integer number drawn from the normal distribution with mean 6 and variance 3.

This queue system also contains a model for the *departure* – the place `Completed`. It collects tokens representing customers served by the system. The purpose of integer values stored in tokens is to provide information about the time when corresponding customers entered the system. If we take a token from `Completed` and subtract its value from its timestamp we get a total time given customer spent in the system - by waiting in the queue and being served.

□

VI.7 Utilities for Simulation Studies in CPN Tools

We already know how to create a timed CPN model of a discrete-event system and now it is the right time to show how to perform actual simulation studies using these models. First, we introduce simulation-specific features of CPN Tools, namely monitoring functions and replications. Then, in the next section, we present a complete simulation study of a system based on the net from Example 10.

VI.7.1 Monitors

Usually the purpose of simulation experiments is to collect some data about performance of simulated systems (models). To be able to collect these data during a CPN simulation CPN Tools allows to define special collections of functions, called *monitors*. Monitors may also have another purpose, such as to terminate a simulation. Four categories of monitors are currently available in CPN Tools:

- *Breakpoint monitors.* They are used to stop a simulation. A breakpoint monitor defines a condition and the corresponding simulation stops when it becomes true.
- *Data collector monitors* are the most essential category for simulation studies as they extract numerical data from given net during simulation and save them to text files with a predefined format. Simulation results are then calculated on the basis of these data. The data can be, for example, values assigned to variables during firings, a number or values of tokens in places or a number of firings of a transition.
  A data collector produces two outputs. The first is a text file with the "`.log`" extension and a name identical to the name of the monitor. The text file is created only if the "Logging" option is checked for the monitor in CPN Tools and contains all values collected by the monitor. For each value it also records in which simulation step and which simulated time it was obtained. The second file is a simulation report in HTML format. The report contains average, minimum and maximum from the values collected by the monitor in given simulation run.
- *Write-in-file monitors*. As the name suggests, they can write a text to files. What is written depends on modellers. They are useful for data extraction in cases when the pre-defined file format provided by the data collector monitors is inappropriate. The file created has a name identical to the name of the monitor. File extension is not fixed and can be specified in CPN Tools.
- *User-defined monitors* are generic monitors without any specific purpose, so they can be used when a functionality not provided by other types of monitors is required. An example of such functionality is a communication with an external application by means of TCP/IP.

Each monitor can be associated with a subset of nodes (i.e. places and transitions) of given CPN. This gives a monitor an access to tokens in associated places and to values used in firings of associated transitions. The time of monitor execution depends on its association to transitions:

- If a monitor is associated with one or more transitions then it is executed after one of these transitions is fired.
- If a monitor is associated with zero transitions then it is executed after each simulation step (i.e. after each transition firing).

Each monitor consists of several functions, namely:

- *Initialization function (*`init`*).* It initializes a monitor before a simulation (simulation run) starts. For example, in a write-in-file monitor it can write a header to a file. This function has access to the places associated with the monitor but not to the associated transitions.
- *Predicate function (*`pred`*).* This function defines a condition for the execution of the next two functions. It is called after simulation steps and when it returns `true` then the observation and action functions of the monitor are executed. If it returns `false`, these functions are not executed.

- *Observation function (*`obs`*).* Its purpose is to extract data from the associated nodes of the net and it can return any type of value.
- *Action function (*`action`*).* This function is executed after the observation function and can process the value returned by it. It doesn't have an access to the associated nodes.
- *Stop function (*`stop`*).* It is executed when a simulation ends (i.e. when simulation stop criteria are met). For example, in a write-in-file monitor it can write a footer to a file. As in the case of the initialization function, it can access associated places only.

What functions can be defined by the modeller depends on the monitor category. For the breakpoint monitor category only the predicate function is available. For data collector and write-in-file monitors one can define all but the action functions. In user-defined monitors all five functions are available.

Above we used the term *simulation* or *simulation run* several times. But what exactly the simulation (run) of a net in CPN Tools is? We can define it as an occurrence sequence that starts in given marking (usually the initial marking of the net) and ends in a marking in which at least one of the following criteria is met:

- There are no more enabled transitions.
- Specified number of steps (firings) has been executed. The number of transitions firing can be defined for some simulation modes in CPN Tools, namely "Fast forward" and "Play".
- The simulation is stopped by a user, i.e. by using so-called "Stop tool".
- Predicate function of some breakpoint monitor returns true.

An exact procedure of monitor creation can be found in the CPN Tools documentation at http://cpntools.org/, but in general we recommend following these steps:

1. Select an appropriate monitor category from the "Monitoring" palette (toolbar) and click on one of the nodes you want to be associated with the monitor. The template code of the monitor is generated automatically.
2. Rename the monitor.
3. If needed, go to the "Nodes ordered by pages" part of the monitor definition and add surrounding places and transitions using the corresponding context menu.
4. If any places and transitions have been added in step 3, generate the monitor template code again (using context menu).
5. Add your own code to the generated functions.

How the monitors can look like is illustrated by Example 18, which adds monitors to the net from Example 17. It also shows how multiple simulations can be run automatically in CPN Tools.

Example 18. Monitors for the queuing system.

The monitors in this example allow collecting data from the simple queue system CPN about the total time spent in the system by a customer, queue waiting times and a queue length. CPN Tools can generate the template code of monitors, so the modeller usually adds only a code defining

the specific functionality of given monitor. There are four monitors, one from each category. The code of the monitors is presented in Figure 40 to Figure 43, where headers of functions are written in bold and the code added or modified manually is in italic.

```
fun pred(bindelem,Top'Completed_1_mark : INTINFtm tms) =
let
  fun  predBindElem(Top'Finish_Serving (1,{customer})) =
        (size Top'Completed_1_mark >= 300)
     | predBindElem _ = false
in
  predBindElem bindelem
end
```

Figure 40. Breakpoint monitor Served300Customers for the CPN from Example 17.

The first one is a breakpoint monitor named `Served300Customers`, which stops simulation after 300 customers have been served. Its code is shown in Figure 40. The monitor is associated with the place `Completed` and transition `Finish_Serving`, because the number of served customers is increased only when `Finish_Serving` is fired and it is equal to the number of tokens in the place `Completed`. The monitor contains only the predicate function `pred`, which has an access to the variables used in firing of `Finish_Serving` (i.e. `customer`) and marking

```
fun init() = NONE

fun pred(bindelem) =
let
  fun  predBindElem (Top'Finish_Serving (1,{customer})) = true
     | predBindElem _ = false
in
  predBindElem bindelem
end

fun obs(bindelem) =
let
  fun  obsBindElem (Top'Finish_Serving (1,{customer})) =
        time()-customer
     | obsBindElem _ = ~1
in
  obsBindElem bindelem
end

fun stop() = NONE
```

Figure 41. Data collector monitor TimeSpentInSystem for the CPN from Example 17.

of `Completed` (represented by the input variable `Top'Completed_1_mark`). The only thing a user has to add to the generated code is the condition in the local function `predBindElem`.

The function predBindElem has two forms (or *rules* in the SML terminology). The first one is with formal parameters (or an *argument pattern* in the SML terminology) "Top'Finish_Serving(1, {customer})". This rule is used when Finish_Serving is fired with correct binding and checks whether the system already served 300 customers. A monitor with more than one transitions associated will have one rule for each of them. The second rule has the argument pattern "_", which means "any argument". Because only one rule is executed when the function is called and it is the first rule whose argument pattern matches actual parameters of the function, the second rule is executed only when Finish_Serving is fired with some invalid binding. All monitor functions that have access to transitions are structured similarly.

The data collector monitor TimeSpentInSystem (Figure 41) is associated only with the transition Finish_Serving. It has three functions but only the observation function obs contains code added by the modeller. The code subtracts value of the variable customer from an actual simulated time, obtained by the function time(). The value of customer contains arrival time of a customer whose serving has been finished by the corresponding firing of Finish_Serving.

```
fun init(Top'CustomerQueue_1_mark : INTINFlist ms) =
maxQueueLength:=0

fun pred(bindelem,Top'CustomerQueue_1_mark : INTINFlist ms) =
let
  fun  predBindElem (Top'Arrive (1,{customers})) = true
     | predBindElem _ = false
in
  predBindElem bindelem
end

fun obs(bindelem,Top'CustomerQueue_1_mark : INTINFlist ms) =
let
  fun  obsBindElem (Top'Arrive (1,{customers})) =
        (length customers) + 1
     | obsBindElem _ = 0
in
  obsBindElem bindelem
end

fun action(observedval) =
  (if (!maxQueueLength)<observedval then
       maxQueueLength:=observedval else ())

fun stop(Top'CustomerQueue_1_mark : INTINFlist ms) = ()
```

Figure 42. User defined monitor UpdateMaxQueueLength for the CPN from Example 17.

The third monitor is a user defined monitor UpdateMaxQueueLength (Figure 42). Its purpose is to determine the maximal number of customers waiting in the queue. To do this it checks the size of the list, stored in the token in the place CustomerQueue, every time the transition

Arrive is fired. Naturally, it is associated with these two nodes. The maximal number is stored in a reference (global) variable maxQueueLength, declared as

```
globref maxQueueLength=0;
```

The init function of UpdateMaxQueueLength sets maxQueueLength to 0, obs extracts an actual queue length and action updates maxQueueLength if necessary. The input parameter observedval contains a value returned by obs. The operator "!" is used to obtain the value of given variable.

```
fun init(Top'CustomerQueue_1_mark : INTINFlist ms) =
  "<simulation>\n"

fun pred(bindelem,Top'CustomerQueue_1_mark : INTINFlist ms) =
let
  fun    predBindElem  (Top'Start_Serving  (1,{customer,customers}))  =
true
     | predBindElem _ = false
in
  predBindElem bindelem
end

fun obs (bindelem,Top'CustomerQueue_1_mark : INTINFlist ms) =
let
  fun  obsBindElem (Top'Start_Serving (1,customer,customers})) =
        " <simrec>\n"^
        "  <qlength>"^Int.toString((length customers)+1)^
        "</qlength>\n"^
        "  <wtime>"^IntInf.toString(time()-customer)^"</wtime>\n"^
        " </simrec>\n"
     | obsBindElem _ = ""
in
  obsBindElem bindelem
end

fun stop(Top'CustomerQueue_1_mark : INTINFlist ms) =
        " <maxqlength>"^Int.toString(!maxQueueLength)^
        "</maxqlength>\n"^
        "</simulation>\n"
```

Figure 43. Write-in-file monitor WaitingTimeAndQLength for the CPN from Example 17.

The last monitor we would like to show here is a write-in-file monitor WaitingTimeAndQLength (Figure 43), associated with the place CustomerQueue and transition Start_Serving. The monitor extracts actual queue length and queue waiting time of a customer just removed from the queue by a firing of Start_Serving and writes them together with the maximal number of customers waiting in the queue to an xml file. The queue length and waiting time are extracted and formatted in XML by the obs fuction, init fuction adds a header and stop function a footer with the maximal queue length (from the variable maxQueueLength). Figure 44 shows an XML file generated by WaitingTimeAndQLength

in a simulation run where 3 customers were served. The value in the `maxqlength` element is 2 because when the simulation stopped two new customers had been waiting in the queue.

```
<simulation>
 <simrec>
  <qlength>1</qlength>
  <wtime>0</wtime>
 </simrec>
 <simrec>
  <qlength>1</qlength>
  <wtime>0</wtime>
 </simrec>
 <simrec>
  <qlength>1</qlength>
  <wtime>4</wtime>
 </simrec>
 <maxqlength>2</maxqlength>
</simulation>
```

Figure 44. An example of the XML file generated by the monitor WaitingTimeAndQLength.

In CPN Tools, nets can be hierarchical, divided into multiple parts, called pages. Even if the net consists of only one part (as in all examples in this book), it has to be placed on a named page. In this example the name of the page is `Top`, which is why the text "`Top'`" occurs often in the code of the monitors.

□

VI.7.2 Simulations and Replications

To execute multiple simulations (simulation runs) automatically we can use so-called *simulation replications* in CPN Tools. For example, to execute 10 simulation runs of the net from Example 18 we add a text field (using the "Text" tool from the "Auxiliary" palette in CPN Tools) to the graph of the net and write

```
CPN'Replications.nreplications 10
```

Then we right click on the text and choose "Evaluate ML" from the corresponding context menu. After this, 10 simulations are performed one by one and data collected in each simulation are stored in a separate folder.

VI.8 Simulation Study: Small Manufactory

This section presents a fictional but complete simulation study, which follows the steps defined in section I.3. The study analyses a workflow in a small manufactory using a simulation model similar to the one in Example 10. The whole section can be considered as one example, divided into subsections with respect to the simulation study steps.

VI.8.1 Problem Formulation

Suppose that we have a manufactory assembling products from delivered sets of parts. The basic manufacturing process, or workflow, can be described as follows:

1. The sets of parts are *delivered in packages*, each contains 50 sets. They can be delivered every 60 minutes but the period between deliveries is usually longer because the next delivery can arrive only after the previous one has been processed (i.e. unpacked and put into a queue).
2. Delivered sets are then put into an *input queue* with FIFO discipline. The queue has the capacity of 50 sets. The time needed to unpack one set of parts to the input queue is approximately 50 seconds (provided that there is enough space in the queue). A delivery can be unpacked only after the unpacking of the previous one has been finished.
3. The input queue is connected to an *assembly line*, which consists of two *belt conveyors* and an *assembly station*. The first conveyor moves a set of parts from the input queue to the assembly station then the assembly station constructs a product from the set and, finally, the product is moved by the second conveyor to the end of the line.
   According to the official documentation of the assembly line the time needed to move a set by the first conveyor is 25 seconds, the time to assembly a product is 67 seconds and the time for the transfer by the second conveyor is 20 seconds.
   The sets of parts are removed from the input queue automatically. They are removed one by one; the removal of one set takes about 6 seconds. For safety reasons a set can be removed (and put on the first conveyor) only if the whole assembly line is empty.
4. A finished product is removed from the end of the line by a *crane*, which moves it to a packaging station.
   The crane can handle only one product at once and, according to its documentation, the movement from the packaging station to the end of the line takes 46 seconds. The movement in the opposite direction is slower, because the crane holds a product, and takes 63 seconds. The time needed to pick up the product is 5 seconds and the time for releasing it is 6 seconds.
5. The *packaging station* consists of an *output queue* and a room where the actual packaging occurs. The output queue has the capacity of 50 products and obeys the FIFO discipline. At one end the products are inserted into the queue by the crane and at the other end removed by employees, who do the packaging. The time needed to pack one product by two employees is approximately two minutes (120 seconds).
6. The finished products are then moved to a warehouse. Capacity of the warehouse and means of transport from the packaging station to the warehouse are such that they will sustain 300% increase of production.

The simulation study should examine the ways in which the manufacturing process can be optimized.

VI.8.2 Setting of Objectives and Overall Project Plan

The current configuration of the manufactory offers several possibilities for optimization. In principal, the optimization can be about speeding up the manufacturing process or reducing costs. The study should explore these possibilities by finding answers to the following questions:

1. *How much will putting more products on the line speed up the process?* The current situation is such that the rule "only one set of parts/product on the assembly line" makes the manufacturing process safe but also slows it down. Two options are considered:
    a. The next set of parts is removed from the input queue when the assembly of the previous one is completed. In this way we can save the time needed to move a product by the second conveyor.
    b. The next set of parts is removed from the input queue before finishing the assembly of the previous one. This option is more risky, but can save a part of the time need for the assembly itself.

    Both options should be evaluated by simulation experiments to decide which one is worth the risk.

    The experiments should also reveal *whether it will be needed to put some sort of stack at the end of the line*: now there is no stack and this can be a problem, because the crane occasionally fails to pick up a product.
2. There are new, faster, assembly stations on the market. *How will the manufactory perform with these new stations?*
3. *Are the current delivery intervals and queue sizes optimal or should they be modified?*

Considering the nature of the manufacturing process and the questions (objectives), we can conclude that performing a simulation study is an appropriate way to answer them. This is in particular because:

- *There is no suitable analytical solution.* There are several random processes and the whole process is not simple enough to be solved by the queuing theory.
- *It is feasible to construct a simulation model with enough detail to be a satisfactory representation of real situation.* Preliminary measurements revealed that the random processes approximately follow existing theoretical random distributions, so it seems that the whole system can be modelled as a stochastic timed discrete-event model.
- *It is impossible to perform the required experiments with the real manufactory*. The manufactory management will not allow putting more sets/products on the assembly line because of the safety reasons. The second question is about new equipment and even borrowing it and putting into the line is too expensive. The third question can possibly be answered for the current configuration but not for the proposed ones.

The simulation study will proceed according to the following *project plan*:

1. *Acquisition of real data about duration of individual phases of the manufacturing process and related information.* As usual, the real life situation differs from estimated values and values given by corresponding documentation. And there are also occasional events such as a jammed conveyor or a failed attempt to pick up a product by the crane. Because of this we cannot use the values specified in section VI.8.1 directly. We have to obtain data from the actual manufacturing process. For this we can use historical data, recorded by the manufactory staff and/or perform new measurements.
2. *Analysis of the acquired data* in order to determine whether they fit or not into some existing random distributions.

3. *Selection of a proper type of simulation model* on the basis of the acquired data and their analysis. In this case it will be found out that a discrete-event model is appropriate and it will be realized as a timed coloured Petri net.
4. *Creation of the simulation model.* A timed CPN model of the manufacturing process is created. The model should be designed in such a way that it will be easy to modify it according to the simulation study objectives.
   The timed CPN can be considered a language for specification of mathematical models of discrete event systems. These models have a benefit of being executable, so they can be simulated without a translation to another form. Therefore this step replaces the steps "model conceptualization", "model translation" and "verification" from section I.3. To be more concrete, this new step is a replacement of the model conceptualization, the model translation is performed automatically by CPN Tools and therefore the verification is not necessary.
5. *Validation of the simulation model with respect to the acquired data*. We perform simulations with the model and compare simulation results with the acquired data. If they match we can consider the model validated.
6. *Modification of the model according to required optimisation.* We modify the model to reflect desired optimisation.
7. *Evaluation of changes by simulation experiments*. We run simulations of the modified model(s) and by analysis of their results evaluate given optimization.
8. *Repetition of previous two steps.* We repeat previous two steps until satisfactory optimisation will be reached.
9. *Formulation of suggestions of the basis of the study results.* We summarize the study results and prepare suggestions for the manufactory management about what to change in the workflow.

VI.8.3 Data Collection and Analysis

Let us assume that we had collected sufficient data about the manufacturing process and from these data we concluded that the real performance of individual phases of the process only approximately follows the data given in section VI.8.1. After detailed analysis of the data we found out that they are close enough to existing probability distributions, so we can use these distributions to describe duration of the phases. Most of them fit normal distributions, but there are some exceptions.

How these distributions are derived from the collected data we show on the duration of the product assembly: According to the official documentation this should take 67 seconds, but measurements and historical data showed varying results. Assume that we collected 249792 values, ranging from 59 to 79 and with the average of 68.405 seconds. A tool that is very useful in finding out whether a set of values fits some theoretical existing probability distribution is a histogram, i.e. a graph showing count (frequency) of each value in the dataset. The count can be seen in the second row (*Count*) of Table 4 and the histogram in Figure 45. The shape of the histogram looks promising; it is close to a normal distribution with mean 68 and variance around 8.

Value	59	60	61	62	63	64	65	66	67	68	69
Count	115	397	1175	3027	6368	5304	19692	22276	34436	37637	35091
Prob.	0.0005	0.0016	0.0047	0.0121	0.0255	0.0212	0.0788	0.0892	0.1379	0.1507	0.1405

Value	70	71	72	73	74	75	76	77	78	79
Count	28190	20709	14805	15943	2846	1187	426	122	35	11
Prob.	0.1129	0.0829	0.0593	0.0638	0.0114	0.0048	0.0017	0.0005	0.0001	0.0000

Table 4. Frequency and probability of individual values from collected dataset of assembly durations.

To investigate this further we derive the *probability density function* (pdf) of the dataset. This can be done using the formula (74)

$$pdf(val) = count(val)/totalCount, \qquad (74)$$

where $count(val)$ is the frequency for a value val (e.g. 5304 for 64) and $totalCount$ is the size of the dataset (e.g. 249792). Values of pdf for our dataset are shown in the third row (*Prob.*) of Table 4.

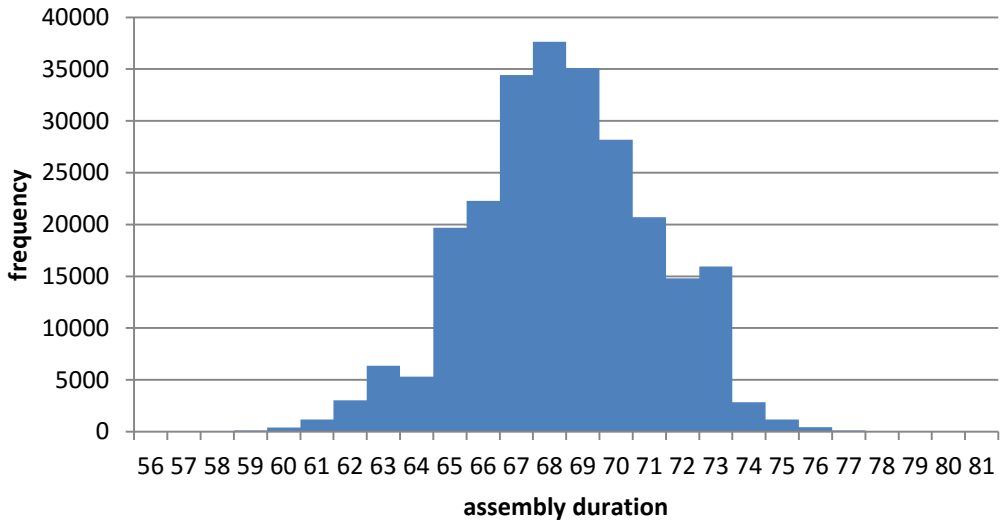


Figure 45. Histogram for collected dataset of assembly durations.

The shape of the pdf can be seen in Figure 46 (blue curve). After comparing the curve to known distributions we conclude that it is closest to the normal distribution with mean 68 and variance 7 (red curve in Figure 46).

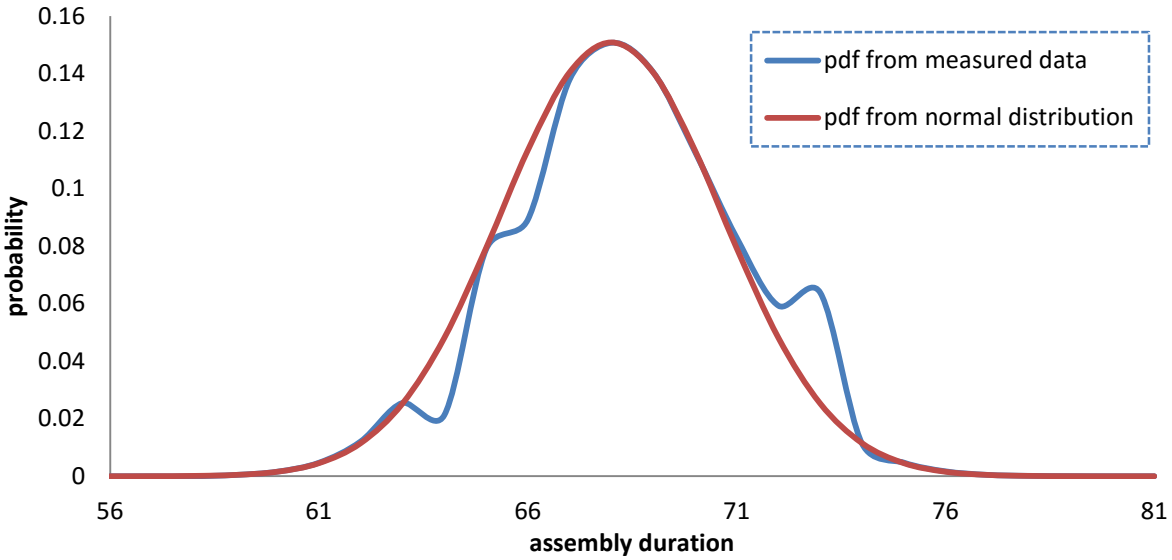


Figure 46. Probability density function of the dataset (blue curve) and normal distribution with mean 68 and variance 7.

Using this process we obtain distributions for all delays (durations). Let us assume that they are (measured in seconds):

- Normal distribution with mean 3550 and variance 765 for *package delivery*. Although it never happened during our measurements, employees of the manufactory told us that approximately in one of 6000 cases the delivery fails, so the time needed is doubled. As they were not able to provide any further details about these failures, we will assume that they are uniformly distributed.
- The *time needed to unpack one set of parts* and put it into the input queue can be best described as consisting of two parts. The first one is fixed, to 46 seconds, and the second one is characterised by the exponential distribution with mean 3.
- Discrete uniform distribution from 5 to 6 for the *removal from the input queue*.
- Normal distribution with mean 25 and variance 4 for the *transportation by the first conveyor* and with mean 20 and variance 2 for the *transportation by the second one*.
- As we already know, normal distribution with mean 68 and variance 7 for the *duration of the product assembly.*
- The value 5 plus a value from the exponential distribution with mean 1 for the *pickup of the product by the crane*. The second value reflects occasional problems with the pickup.
- Normal distribution with mean 64 and variance 3 for the *movement of the crane from the end of the line to the packaging station*, with mean 46 and variance 2 for the *movement in the opposite direction* and with mean 6 and variance 1 for *releasing the product*.
- Normal distribution with mean 123 and variance 17 for the *product packaging*.

An interested reader can find more information about how to choose probability distributions for simulation models in (Law, 2012).

VI.8.4 Model Creation

Now we are ready to pick up a proper type of model for our simulation study. Because all key processes of the system (approximately) follow existing theoretical random distributions, the

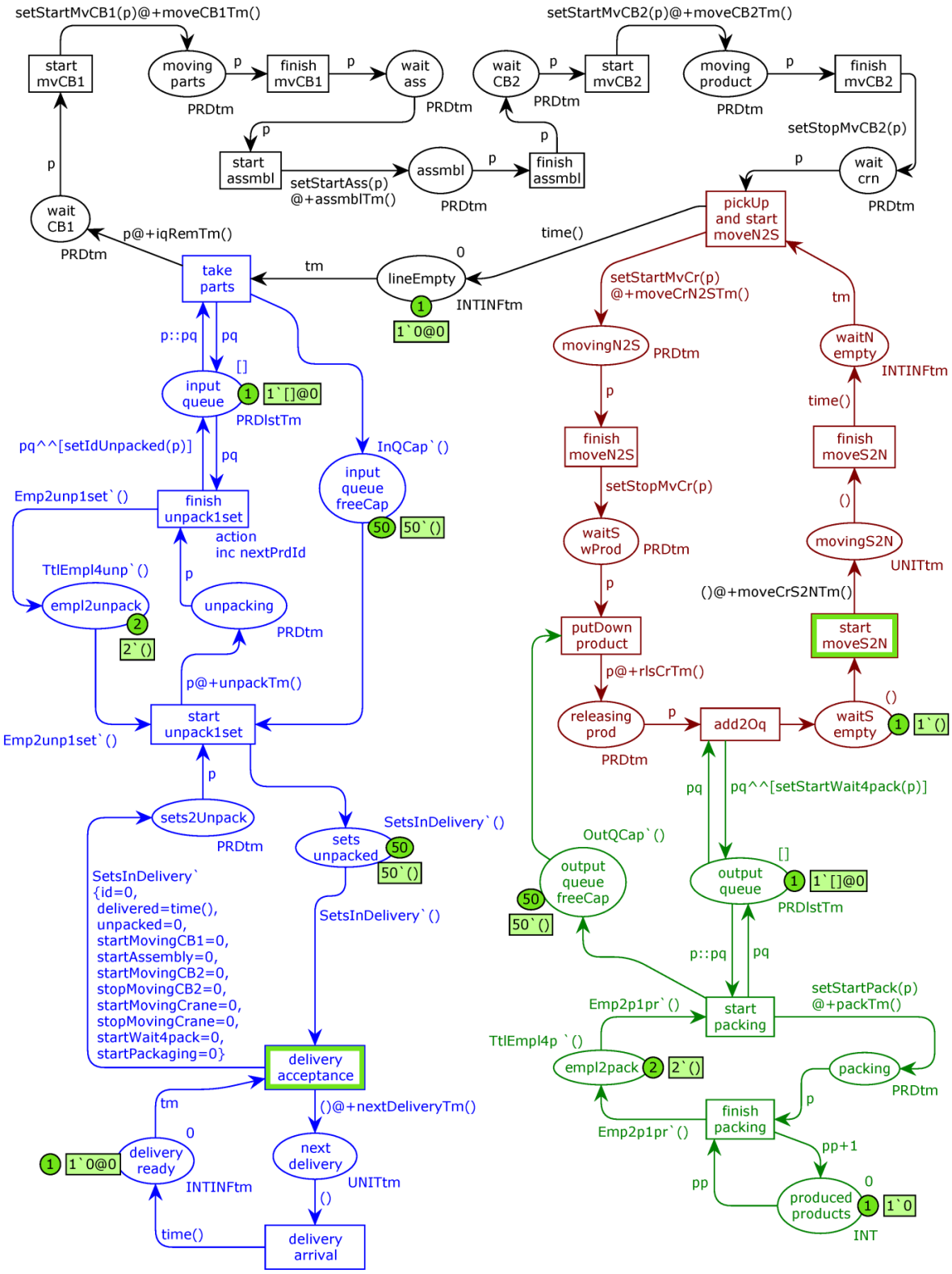


Figure 47. Graph of the manufactory workflow CPN model.

whole system can be modelled as a stochastic timed discrete-event model. We will specify the model as a timed CPN and use CPN Tools software for simulation. Transitions of the model will represent key events of the manufacturing process (workflow) and tokens will carry data necessary to record durations of individual phases of the process. These durations will be recorded by monitors of the model. One time unit in the model will be equal to one second.

The graph of the model is shown in Figure 47. It is divided in four parts, distinguished by different colours:

- The blue part models the package delivery, unpacking of the packages and putting sets of parts to the input queue and the first conveyor (steps 1 and 2 of the description in section VI.8.1).
- The black part models the assembly line (step 3 from section VI.8.1).
- The brown part models the crane (step 4).
- The green part models the output queue and product packaging (step 5).

As the capability of the warehouse and delivery to it will be sufficient even after the optimization, this part of the process (step 6 from section VI.8.1) is not modelled. The arcs without arc expressions visible have arc expressions "`()`" and places without colour set visible have "`UNIT`" as the colour set.

The CPN model uses colour sets `UNIT`, `UNITtm` and `INTINFtm`, defined as in Figure 39 and new colour sets `PRD`, `PRDtm` and `PRDlstTm` (Figure 48).

```
colset PRD = record
  id:INT *
  delivered:INTINF *
  unpacked:INTINF *
  startMovingCB1:INTINF *
  startAssembly:INTINF *
  startMovingCB2:INTINF *
  stopMovingCB2:INTINF *
  startMovingCrane:INTINF *
  stopMovingCrane:INTINF *
  startWait4pack:INTINF *
  startPackaging:INTINF;

colset PRDtm = PRD timed;
colset PRDlstTm = list PRD timed;
```

Figure 48. Declarations of new colour sets of the manufactory workflow CPN model from Figure 47.

The record colour set `PRD` is used for tokens that carry information about time of important events during the product (set of parts) lifetime. These data are then used by data collector monitors of the net. The set `PRDtm` is a timed version of `PRD` and `PRDlstTm` is a list of records of `PRD` type, again timed.

The net also uses a couple of constants (
Figure 49) and variables (Figure 50).

```
val SetsInDelivery = 50;
val InQCap = 50;
val Emp2unp1set=2;
val TtlEmpl4unp = 2;
val OutQCap = 50;
val TtlEmpl4p = 2;
val Emp2p1pr=2;
```

Figure 49. Declarations of constants of the CPN model from Figure 47.

The constant `SetsInDelivery` defines the number of sets of parts in one delivery, `InQCap` the capacity of the input queue, `Emp2unp1set` the number of employees needed to unpack one set of parts and put them to the input queue and `TtlEmpl4unp` the total number of employees assigned to the task of unpacking the sets of parts and putting them to the input queue. `OutQCap` is the capacity of the output queue, `TtlEmpl4p` is the total number of employees assigned to the task of taking the finished products from the output queue and subsequent packaging and `Emp2p1pr` is the number of employees needed to take a product from the output queue and pack it.

```
var p:PRDtm;
var pq:PRDlstTm;
var tm:INTINFtm;
var pp:INT;
globref nextPrdId=0;
```

Figure 50. Declarations of variables of the CPN model from Figure 47.

The variable `p` is the most frequent in the model; it is used on every arc where a token representing a set of parts or a product is processed, `pq` is used for lists (queues) of products or sets of parts. The last one, `nextPrdId`, differs from the rest in being a global variable. This means that its value is not reset before each transition firing. It is used to assign id to a set of parts or a product and its value is updated by the command

```
action inc nextPrdId,
```

associated with the `finish_unpack1set` transition.

To make the model more readable, most expressions that need to be computed are specified in the form of CPN ML functions with self-explanatory names. These functions can be divided into two groups:

1. *Delay functions*, which compute durations of corresponding process phases using the probability distributions defined in section VI.8.3. To compute durations they use build-in probability functions from CPN ML and functions `normTime` and `expTime` (Figure 39).

2. *Token update functions*, which update fields in a record of `PRD` type. Usually only one field is updated with the value of current simulation time. Other fields are copied.

```
fun nextDeliveryTm()=
  let
    val failRate = (if discrete(1,6000)=1 then 2 else 1)
  in
    failRate*normTime(3550,765)
  end
```

Figure 51. Declaration of the CPN ML function nextDeliveryTm.

The delay functions are as follows:

- `nextDeliveryTm()` – returns the time after which a new delivery will arrive. This function combines the discrete uniform distribution (function `discrete` from CPN ML) for delivery failure with normal distribution (function `normTime`). Its code is as listed in Figure 51.
- `unpackTm()` – returns time needed to unpack one set of products and put it into the input queue. It is defined as `fun unpackTm() = 46+expTime(3).`
- `iqRemTm()`– returns the time needed to take a set from the queue and put it to the first conveyor belt of the assembly line. It is defined as
  `fun iqRemTm() = discrete(5,6).`
- `moveCB1Tm()` – returns the time it takes to move a set of parts by the first conveyor. It is defined as `fun moveCB1Tm() = normTime(25,4).`
- `assmblTm()`– returns the time it takes to assembly one product from one set of parts. It is defined as `fun assmblTm() = normTime(68,7).`
- `moveCB2Tm()`– returns the time it takes to move a product by the second conveyor. It is defined as `fun moveCB2Tm() = normTime(20,2).`
- `moveCrN2STm()` – computes the time needed to move the crane with a product from north to south. The value returned by this function also includes the time needed to pick up the product and takes into account the fact that sometimes the attempt to pick up a product is not successful. Its code is
  `fun moveCrN2STm()  = 5+expTime(1)+normTime(64,3).`
- `rlsCrTm()` – returns the time needed to release a product by the crane at the packaging station. Its code is `fun rlsCrTm() = normTime(6,1).`
- `moveCrS2NTm()` – computes the time needed to move the empty crane from south to north. Its code is `fun moveCrS2NTm()  = normTime(46,2).`
- `packTm()` – returns time needed to take one finished product from the output queue and pack it. It is defined as `fun packTm() = normTime(123,17).`

There are eight token update functions in the model:

- `setIdUnpacked(pd)`- assigns a new id to a set of products (and corresponding product after assembly) `pd` and records the time when it was unpacked into the variable unpacked. The code of the function can be found in Figure 52.
- `setStartMvCB1(pd)` - records the time when the corresponding set of parts `pd` begun its movement on the first conveyor belt (i.e. updates the field `startMovingCB1` with `time()`).
- `setStartAss(pd)` - records the time when the assembly of the corresponding product `pd` started (i.e. updates the field `startAssembly` with `time()`).
- `setStartMvCB2(pd)` - records the time when `pd` begun its movement on the second conveyor belt (i.e. updates the field `startMovingCB2` with `time()`).
- `setStopMvCB2(pd)` - records the time when `pd` finished its movement on the second conveyor belt (i.e. updates the field `stopMovingCB2` with `time()`).
- `setStartMvCr(pd)` - records the time when `pd` was picked up by the crane (field `startMovingCrane`).
- `setStopMvCr(pd)` - records the time when the crane had stopped moving with the corresponding product `pd` but before it dropped it (field `stopMovingCrane`).
- `setStartWait4pack(pd)` - records the time when `pd` was put into the output queue (field `startWait4pack`).
- `setStartPack(pd)` - records the time when packaging of the corresponding product `pd` started (field `startPackaging`).

```
fun setIdUnpacked(pd:PRD)=
{  id= !nextPrdId,
   delivered=(#delivered pd),
   unpacked=time(),
   startMovingCB1=(#startMovingCB1 pd),
   startAssembly=(#startAssembly pd),
   startMovingCB2=(#startMovingCB2 pd),
   stopMovingCB2=(#stopMovingCB2 pd),
   startMovingCrane=(#startMovingCrane pd),
   stopMovingCrane=(#stopMovingCrane pd),
   startWait4pack=(#startWait4pack pd),
   startPackaging=(#startPackaging pd)
}
```

Figure 52. Declaration of the CPN ML function setIdUnpacked.

As it can be seen in Figure 47, the initial marking of the net represents a situation where the manufactory is ready to accept the first delivery (a token with the value `0` and timestamp `0` in the place `delivery_ready`), both queues are empty (fifty tokens in the place `input_queue_freeCap` and in `output_queue_freeCap`), two employees are ready to unpack sets of parts (two tokens in the place `empl2unpack`) and another two to pack finished products (two tokens in `empl2pack`), the crane is waiting empty at the packaging station (a token in `waitS_empty`) and the assembly line is empty, too (a token in `lineEmpty`). Two transitions are enabled: `delivery_acceptance` and `start_moveS2N`.

A firing of delivery_acceptance means that a newly arrived delivery of sets of parts is accepted for unpacking. This can happen only when there are no sets to be unpacked from the previous delivery. This is why the transition delivery_acceptance is connected to the place sets_unpacked, which holds a token for each unpacked set from the previous delivery. The firing of delivery_acceptance adds a token for each set to the place sets2Unpack and one token with a timestamp equal to the time of the next delivery arrival to next_delivery. This is because the firing also means a start of unpacking and ordering of a new delivery. Arrival of the next delivery is represented by a firing of delivery_arrival. This creates a new token in delivery_ready. The token stores the time of its creation and this value is used to compute difference between the time of arrival and acceptance.

The tokens in sets2Unpack have the time of their creation, i.e. the time of the delivery of given set, stored in the field delivered. An unpacking of a set starts with a firing of start_unpack1set and ends with a firing of finish_unpack1set, which adds a (token representing a) set of parts to the input queue, modelled by the token in the place input_queue. This token is a list and is handled in exactly the same way as the one in the place CustomerQueue in the net from Example 17. Tokens in empl2unpack stand for employees that can be used for the unpacking and input_queue_freeCap holds a token for every free place in the input queue. A token in unpacking is a set of parts being unpacked.

By a firing of take_parts a set of parts is taken from the input queue and put on the first conveyor. Of course, this can happen only when the assembly line is empty (i.e. there is a token in lineEmpty). After this, the set waits (as a token in wait_CB1) to be moved by the first conveyor. The movement starts with firing start_mvCB1 and ends by firing finish_mvCB1. The set during the movement is represented by a token in moving_parts. After the conveyor the set waits (in wait_ass) to be assembled. The assembly starts by a firing of start_assmbl, ends by a firing of finish_assmbl and a token in assmbl is a product being assembled. The second conveyor is modelled similarly to the first one (places wait_CB2 and moving_product and transitions start_mvCB2 and finish_mvCB2). The second conveyor delivers the product to the end of the assembly line, where it waits (in wait_crn) to be taken by the crane.

After waiting, the product is picked up by the crane (a firing of pickup_and_start_moveN2S), which immediately begins to move it to the packaging station. The firing of this transition also means emptying of the assembly line and the time of the firing is recorded into a newly created token in lineEmpty. The product during the movement of the crane is represented by a token in movingN2S and the movement is finished by a firing of finish_moveN2S. Then the product waits in waitS_wProd to be released to the output queue by a firing of putDown_product. This can only happen when the output queue is not full (i.e. there is at least one token in output_queue_freeCap). The released product is represented by a token in releasing_prod and its insertion to the output queue by a firing of add2Oq. A movement of the crane back from the packaging station to the assembly line is modelled by transitions start_moveS2N and finish_moveS2N and places movingS2N and waitN_empty.

Finally, the product is packaged at the packaging station. The output queue is a token in the place `output_queue`, which is treated in a way similar to the one in `input_queue`. The packaging process starts with a firing of `start_packing` and ends with a firing of `finish_packing`. A product during the process is modelled by a token in `packing` and tokens in `empl2pack` stand for employees that do the packing. For each packed product the counter, represented by a token in `produced_products`, is increased by one.

VI.8.5 Monitors for the Simulation Model

To perform validation and simulation experiments we need to add monitors to our model. All but one are data collector monitors and most of them measure duration of individual phases of the manufacturing process. They are described in section VI.8.5.1. The last one is a breakpoint monitor and it is discussed in section VI.8.5.2.

VI.8.5.1 Data Collector Monitors

These monitors can be divided into four groups, identified by prefixes of their names:

1. *Queue length measurement* (prefixes `ql1` and `ql2`). Their primary purpose is to acquire data needed to answer the 3rd question formulated in section VI.8.1.
2. *Overall time measurement* (prefixes `ta1` and `ta2`). They measure duration of two basic phases of the manufacturing process: waiting before unpacking and the manufacturing process itself –from unpacking of a set of parts to packing of the finished product. They are important for all three questions from section VI.8.1, the first of them especially for the 3rd question.
3. *Detailed time measurement* (prefixes `tp1` to `tp6`). These monitors focus on some phases of and delays during the manufacturing process. They are especially important for answering the 1st and the 2nd question from section VI.8.1.
4. *Token value measurement* (prefix `v1`). This group contains only one monitor, which records values of tokens from the place `produced_products`. It is important with respect to the 1st and the 2nd question.

Names and purpose (i.e. what value they record) of the monitors are listed in Table 5 and more details about them can be found in Table 6. The third column of Table 6 contains a new code that replaces "`0`" in the generated observer (the function `obs`) of the monitor. For example, if the generated code is

```
fun obs (bindelem) = let
  fun obsBindElem (Top'finish_mvCB2 (1, {p})) = 0
    | obsBindElem _ = ~1
in obsBindElem bindelem end
```

and the corresponding cell in the third column contains "`time() - #startMovingCB1 p`" then the code of the observer is

```
fun obs (bindelem) = let
  fun obsBindElem (Top'finish_mvCB2 (1, {p})) =
                          time() - #startMovingCB1 p
    | obsBindElem _ = ~1
```

```
in obsBindElem bindelem end
```

Monitor	*Recorded value*
`ql1_inQueueAfterIns`	Number of items in the input queue after a set of parts is added to it.
`ql2_outQueueAfterIns`	Number of items in the output queue after a finished product is inserted into it.
`ta1_DelArr2StartUnp`	The time a new delivery of sets of parts has to wait to be accepted for unpacking after it arrives.
`ta2_StartUnp2EndPck`	The total time needed to produce one product, i.e. the time from the moment when given set of parts starts to be unpacked to the moment when packing of the product assembled from the set is finished.
`tp1_LineWt4Parts`	The time the empty assembly line has to wait for a set of parts. This value will be greater than 0 if the assembly line is empty and there is no set of parts waiting in the input queue.
`tp2_PartsWt4Line`	The time a set of parts has to wait in the input queue for being put on the assembly line.
`tp3_OnAssLine`	Total time a set of parts and corresponding product spend at the assembly line – from being loaded on the first conveyor, through the assembly to reaching the end of the second conveyor.
`tp4_CrWt4Prd`	The time the crane has to wait for a product to arrive at the end of the assembly line (i.e. the end of the second conveyor).
`tp5_PrdWt4Cr`	The time a product has to wait for the crane at the end of the assembly line.
`tp6_Wt4CrRls`	The time the crane has to wait with a product at the packaging station before releasing it. Only the full output queue can cause this value to be greater than 0.
`v1_products`	Number of products produced from beginning of given simulation run to the moment when packing of a new product is finished.

Table 5. Data collector monitors of the net from Figure 47.

Monitor	*Associated transition*	*New code in obs*
ql1_inQueueAfterIns	finish_unpack1set	(length pq)+1
ql2_outQueueAfterIns	add2Oq	(length pq)+1
ta1_DelArr2StartUnp	delivery_acceptance	time()-tm
ta2_StartUnp2EndPck	finish_packing	time() - #delivered p
tp1_LineWt4Parts	take_parts	time()-tm
tp2_PartsWt4Line	take_parts	time() - #unpacked p
tp3_OnAssLine	finish_mvCB2	time() - #startMovingCB1 p
tp4_CrWt4Prd	pickUp_and_start_moveN2S	time()-tm
tp5_PrdWt4Cr	pickUp_and_start_moveN2S	time() - #stopMovingCB2 p
tp6_Wt4CrRls	putDown_product	time() - #stopMovingCrane p
v1_products	finish_packing	pp+1

Table 6. Details of data collector monitors of the net from Figure 47.

VI.8.5.2 Breakpoint Monitor

The purpose of the one breakpoint monitor, named `endSimAfterOneShift`, is to limit duration of each simulation run. This duration will be equal to one shift in the manufactory, which is 8 hours (28800 seconds) long. To make this length easily adjustable we add a constant `shiftDuration`, declared as

```
val shiftDuration:INTINF  = 28800;
```

The monitor `endSimAfterOneShift` can be associated with any place of the net, because it only checks whether the simulated time reached the value defined by `shiftDuration`. However, the monitor should not be associated with any transition as we want it to be run after each simulation step. As in the case of the monitor `Served300Customers` from Example 18, only the code of the predicate function has to be modified:

```
fun pred(Top'next_delivery_1_mark:UNITtm tms)=
                time()>=shiftDuration.
```

VI.8.6 Validation

After completion of monitors the model is ready for simulation. First simulation runs are used to *validate* the model (step 5 of the project plan): we simulate the model with all parameters set as described in the previous section and compare it to the data collected before. Table 7 lists results of 30 simulation runs, where each simulation run equals to one 8 hours shift (thanks to the breakpoint monitor). For all measures except the last one the value in the column "*Minimum*" is the minimum of minima of individual simulation runs, in "*Maximum*" the value is maximum of maxima and in the last one average of averages. In the case of the last measure all three values are computed from maxima of individual simulation runs. This is because the number of products manufactured during one shift is equal to the maximal value recorded by monitor `v1_products` during one simulation run. For identifiers of the measures we use prefixes of the monitors used to obtain their values.

Measure (id)	*Minimum*	*Maximum*	*Average*
Number of items in the input queue (*ql1*)	1	50	41,3
Number of items in the output queue (*ql2*)	1	3	1,1
New delivery waiting time for unpacking (*ta1*)	0	2649	1113,29
Total production time per product (*ta2*)	351	12762	7131,24
Assembly line waiting time for parts (*tp1*)	0	55	0,2
Parts waiting time for assembly line (*tp2*)	0	6134	4781,21
Time on the assembly line per product (*tp3*)	100	128	112,97
Crane waiting time for a product pickup (*tp4*)	0	130	1,08
Product waiting time for the crane (*tp5*)	0	20	4,08
Crane waiting time for a product release (*tp6*)	0	0	0
Number of products produced per shift (*v1*)	231	233	231,67

Table 7. Data gathered from 30 simulation runs of the net from Figure 47.

Let us assume that these simulation results match the collected data, so the model is validated and we can proceed to simulation experiments. If not, the model has to be adjusted and simulated again until the match is achieved.

VI.8.7 Simulation Experiments and Analysis of Results

The model is validated, so we can perform simulation experiments that will provide answers to the three questions stated in section VI.8.2. To do this, we modify the simulation model accordingly (step 6 of the project plan), run simulations and analyse their results (step 7) and repeat these two steps if necessary (step 8).

VI.8.7.1 Experiment 1: More Products on the Line

The first experiment should answer the question 1, namely whether allowing two products on the assembly line will speed up the whole manufacturing process significantly. In section VI.8.2 we specified two options for this speed-up:

a. The next set of parts is put on the line when the assembly of the previous one is completed.
b. The next set of parts is put on the line when the assembly of the previous one starts.

Both options require changes in the part of the simulation model that represents the assembly line. Changed parts are shown in Figure 53 and Figure 54, where new nodes (2 places) and arcs are rendered in red. The place `lineEmpty` is renamed to `lineReady4NxtSet` as the original name no more reflects the reality.

We also added a new monitor, `ms1_waiting4CB2`, which records number of tokens in the place `wait_CB2` after each firing of its adjacent transitions (`finish_assmbl` and `start_mvCB2`). This is because we anticipated accumulation of tokens (assembles products) in this place. The new monitor is so-called *marking size monitor*. It is a special case of the data collector monitor and in CPN Tools it can be created by selecting "Mark Size" on the "Monitoring" palette and clicking on the corresponding place. No code modification is necessary.

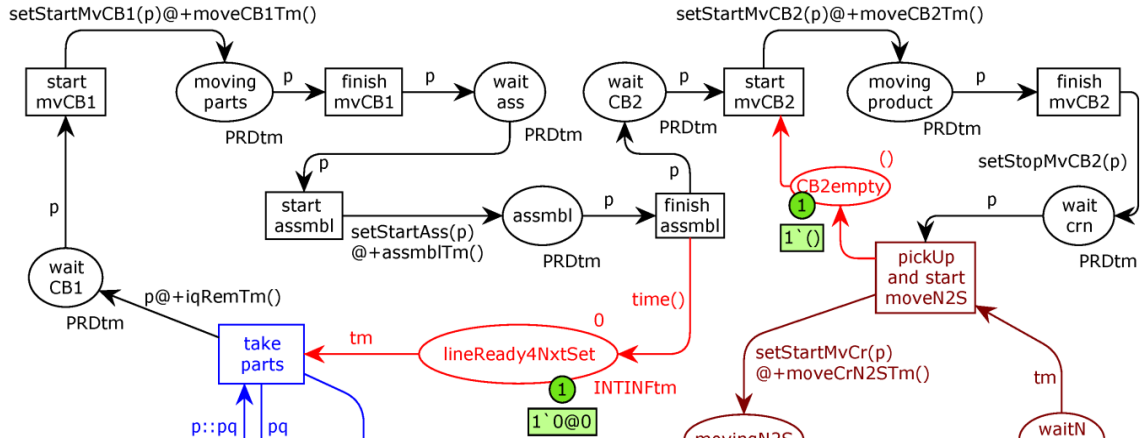


Figure 53. A part of the graph of the CPN model for Experiment 1, option a, which differs from the model in Figure 47.

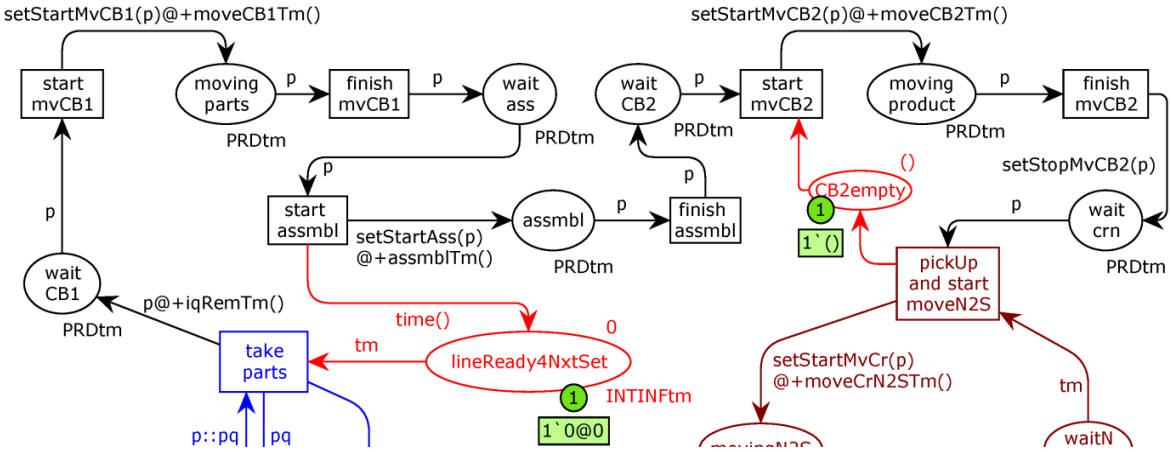


Figure 54. A part of the graph of the CPN model for Experiment 1, option b, which differs from the model in Figure 47.

Results of simulation experiments with both modified models can be seen in Table 8. Here columns 2 to 4 are the results from the first model, columns 5 to 7 from the second one. Minima, maxima and averages are computed in the exactly same way as in Table 7. Now we analyse the results by comparing them to the ones in Table 7. We can see that allowing two products on the line doesn't bring any advantage at all. The number of products produced per shift (measure *v1*) is almost exactly the same: the average is slightly higher (231,76 or 231,73 to 231,67), but the maximal number of products remains the same (233). After more detailed analysis it becomes clear that the crane is too slow now and products are accumulating before the second conveyor (measure *ms1*). And we should also take into account that in reality the process can be even slower, because in this experiment we assume that there are no additional delays caused by the accumulation. So, the conclusion is that without speeding up the crane it makes no sense to allowing more products on the assembly line.

VI.8.7.2 Experiment 2: Two Products on the Line and Faster Crane and Packing

The second experiment should start where the first one finished and evaluate an effect of a faster crane. However, a quick look on the packing duration reveals that speeding up the crane will not be enough: The mean of the packing duration is 123, so if we assume that to pack one product we need exactly 123 seconds then we will be able to pack at most 234 products during one shift (28800 seconds). Therefore the packaging station will become the bottleneck of the system after speeding up the crane.

Let us assume that after consulting the manufactory management the following maximal speedup has been estimated:

- duration of crane movement from the end of the line to the packaging station to mean 38 and variance 2 (from 64 and 3),
- duration of crane movement in the opposite direction to mean 35 and variance 2 (from 46 and 2) and
- duration of packing one product to mean 90 and variance 11 (from 123 and 17).

Measure (id)	*Option a.*			*Option b.*		
	Min.	*Max.*	*Average*	*Min.*	*Max.*	*Average*
Number of products waiting for the 2nd conveyor (*ms1*)	0	57	27,07	0	177	90,61
Input queue size (*ql1*)	1	50	39,62	1	1	1
Output queue size (*ql2*)	1	4	1,53	1	4	1,55
New delivery w. t. for unp. (*ta1*)	0	1470	468,90	0	0	0
Total prod. t. per prod. (*ta2*)	347	24180	7598,47	355	28426	6191,94
Ass. line w. t. for parts (*tp1*)	0	60	0,17	8	1246	40,1
Parts w. t. for ass. line (*tp2*)	0	4945	3679,08	0	0	0
T. on ass. line per prod. (*tp3*)	104	21021	2337,39	102	27316	4621,83
Crane w. t. for prod. p.up (*tp4*)	0	135	0,52	0	128	0,51
Prod. w. t. for the crane (*tp5*)	0	116	100,8	0	113	101,4
Crane w. t. for prod. rel. (*tp6*)	0	0	0	0	0	0
Products produced per shift (*v1*)	231	233	231,76	231	233	231,73

Table 8. Data gathered from experiment 1 (30 simulation runs of the nets for options a and b).

As both options led to similar results in the previous experiment, this one was performed only for the safer one, the option a. Results can be seen in the last three columns of Table 9, its columns 2 to 4 repeat results from Table 8 for better comparison. The most significant changes are in bold.

The results show significant increase in production, from 231 to 290. Also important is that at most one product waits for the second conveyor (measure *ms1*), so we do not need to add any form of stack or queue before the conveyor. From measures *tp4* and *tp5* we see that the crane usually has to wait for a product while a product almost never waits for the crane. This means that the crane, and possibly also the packaging station, are little bit faster than necessary now. After some more experiments (which we will not describe in detail here) we found out that with

- duration of crane movement from the end of the line to the station set to mean 45 and variance 2,

- duration of crane movement in the opposite direction set to mean 40 and variance 2 and
- duration of packing one product set to mean 99 and variance 11

we achieve almost the same number of products per shift, 288 products on average (minimum is 287 and maximum is 289).

Measure (id)	*Opt. a. (exp.1)*			*Opt. a. with speedup. (exp.2)*		
	Min.	*Max.*	*Average*	*Min.*	*Max.*	*Average*
No. of prd. wait. for CB2 (*ms1*)	0	57	27,07	0	**1**	**0**
Input queue size (*ql1*)	1	50	39,62	1	50	39,33
Output queue size (*ql2*)	1	4	1,53	1	1	1
New delivery w. t. for unp. (*ta1*)	0	1470	468,90	0	1462	450,87
Total prod. t. per prod. (*ta2*)	347	24180	7598,47	292	**10223**	**5567,04**
Ass. line w. t. for parts (*tp1*)	0	60	0,17	0	54	0,16
Parts w. t. for ass. line (*tp2*)	0	4945	3679,08	0	4951	3643,84
T. on ass. line per prod. (*tp3*)	104	21021	2337,39	99	**126**	**113**
Crane w. t. for prod. p.up (*tp4*)	0	135	0,52	0	141	**13,93**
Prod. w. t. for the crane (*tp5*)	0	116	100,8	0	**2**	**0**
Crane w. t. for prod. rel. (*tp6*)	0	0	0	0	0	0
Products produced per shift (*v1*)	231	233	231,76	**288**	**291**	**290**

Table 9. Data gathered from experiment 2 (30 simulation runs of the net for option a) compared to experiment 1.

VI.8.7.3 Experiment 3: Faster Assembly Station

To answer the second question from section VI.8.2, wherever a replacement of the assembly station with a faster one will lead to more products manufactured, we take the original model and modify values (`68` and `7`) in the `assmblTm` according to the parameters of the new station. Then we run simulations, which will lead to the results similar to the experiment 1. That is, the number of products produced per shift will remain the same because of the slow crane and packaging station. Only after changes similar to the ones in the experiment 2 the utilisation of the new assembly station will lead to increased production. For example, with the new station two times faster than the old one, the manufactory will produce about 318 products per shift.

VI.8.7.4 Experiment 4: Evaluation of Queue Sizes and Delivery Intervals

As the last question from section VI.8.2 is all about the original manufacturing process, it seems that answering it doesn't require any modifications of the original simulation model and we can work with the results from Table 7. From these results we see that:

- The input queue holds 39,62 sets on average (measure *ql1*) and detailed analysis of the data collected by the monitor `ql1_inQueueAfterIns` shows that it becomes fully occupied after approximately 2 hours from the start of the shift.
- The output queue is rarely used, usually there is only one product and the maximal number of products it holds is 3 (measure *ql2*).

It seems that the output queue can be completely eliminated and to verify this we reduce its capacity to 1 (i.e. we set the constant `OutQCap` to 1) and run simulations. Results will confirm our assumption, the number of products produced per shift will be 231,5 on average, the minimal number will remain 231 and the maximal number will be 232.

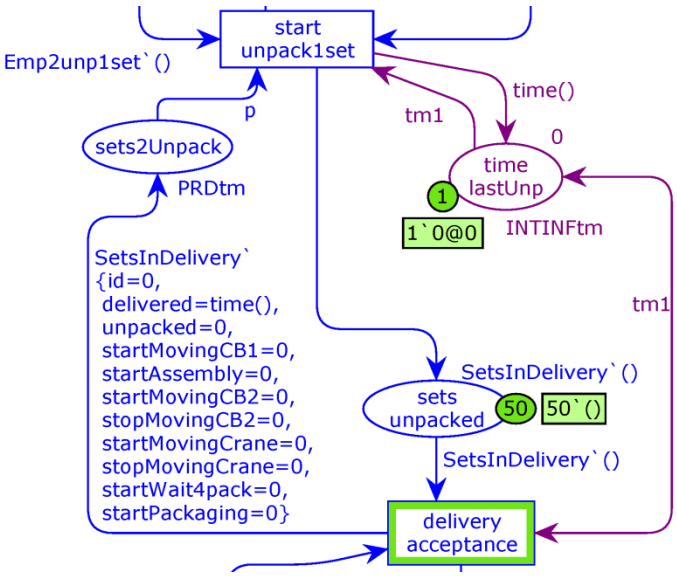


Figure 55. A part of the graph of the CPN model for Experiment 4, which differs from the model in Figure 47.

To evaluate the delivery intervals we can use data gathered by the monitor `ta1_DelArr2StartUnp` (measure *ta1* in Table 7). But they reveal only half of the information we need. They tell us when and for how long a newly arrived delivery has to wait to be accepted for unpacking, in other words they show us when the delivery intervals are too short. However, we do not have a monitor that records when this interval is too long, i.e. when the manufactory waits for a new delivery. To be able to create such a monitor we need to add a new place `time_lastUnp` to the blue part of the model (Figure 55, new parts are purple). This place will hold the time of the last unpacking. The monitor itself, named `ta1a_UnpWt4Del`, will be associated with the transition `delivery_acceptance`. The new code in its observer will be

`time()-tm1`. The variable `tm1` is a new variable of the type `INTINFtm`. There are 6 deliveries during one shift and the data collected for them can be seen in Table 10. Results from other simulation runs are very similar.

Delivery no.	1	2	3	4	5	6
Time of delivery	0	3574	7140	12198	18324	24468
Delivery waits for unpacking (ta1_DelArr2StartUnp)	0	0	0	1470	2594	2582
Manufactory waits for delivery (`ta1a_UnpWt4Del`)	0	1158	1093	0	0	0

Table 10. Data about delivery intervals from one simulation run.

From this analysis we can draw the following recommendation for the manufactory management:

- The input queue can remain as it is.
- The output queue can be completely eliminated.
- The second and third delivery should be ordered about 10 minutes sooner.
- The fourth delivery can be ordered about 23 minutes later and the last two deliveries about 40 minutes later.

VI.8.7.5 Further experiments

The experiments presented here are not the only ones that can be performed with the simulation model. Another experiment can estimate how much the number of employees needed for packing and unpacking can be reduced without affecting overall effectivity of the process. For example, we can unite places empl2unpack and empl2pack and observe whether the both unpacking and packing can be performed by common, reduced, group of employees.

VI.8.8 Documentation

To consider the simulation study completed it has to be well-documented. The documentation should be based on progress reports that are prepared several times (regularly or when some partial goal is achieved) and has to include a detailed description of the simulation model. We can say that the documentation of our study should contain most of what was written in the previous sections. Its outline can be as follows:

1. *Requirements.* Specifies goals of the study, i.e. what a customer expects to be accomplished by the study. It should also describe the system that is the subject of the study. In our case the requirements are the three questions from section VI.8.2 and the study subject is described in section VI.8.1.

2. *Input data analysis.* Analysis of the data collected about the real system. This should result in selecting an appropriate type of simulation model and representation of processes. Section VI.8.3 contains such an analysis. All the collected data should be stored in some form (e.g. a database), too.
3. *Simulation model description.* Description of all the parts of the simulation model in such a detail that it allows any qualified person to understand, use and modify the model. For a CPN model this should contain a graph of the model, meaning of places and transitions and specification of used colour sets, constants, variables, functions and monitors. Here it is covered by sections VI.8.4 and VI.8.5.
4. *Simulation experiments report.* For each experiment this part records changes in the simulation model we had to make in order to perform the experiment, simulation results and their analysis and conclusion. The conclusion usually contains answers to questions that led to the experiment or recommendations for the customer (see section VI.8.7).
5. *Conclusion.* Overall conclusion of the study, based on the conclusions of individual experiments. It is a basis for an implementation of the study results and is covered by the next section.

VI.8.9 Conclusion and Implementation

In conclusion, our study recommends the following changes to be implemented in the manufactory.

1. *Elimination of the output queue.* As the experiment 4 shown, this queue is seldom used and its removal has a minimal effect on the production (only one product less is produced per shift).
2. *Two products on the assembly line but only with faster crane and packaging station.* According to the experiments 1 and 2, allowing two "products" (more exactly one set of parts and one product) on the assembly line at once can lead to increased production only when the speed of the crane and packaging station are increased.
3. *Faster assembly station, but only with faster crane and packaging station.* Replacement of the assembly station will help only with a faster crane and packaging (experiment 3).

Which of these recommendations will be implemented depends on the manufactory management, which has to consider a cost of and expected profit from the implementation.

References

Banks et al. (1998). Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice, John Wiley & Sons, New York, NY.

Banks, J.; Carson, J.S.; Nelson, B.L. & Nicol, D.M. (2001). Discrete-Event System Simulation, Third Edition, Prentice-Hall, Upper Saddle River, NJ.

Cassandras, Ch. G. & Lafortune, S. (2008). Introduction to Discrete Event Systems, second edition, Springer.

Cellier, F. E. & Greifeneder, J. (1991). Continuous System Modeling, , Springer-Verlag, New York, NY.

Cellier, F.E. & Kofman, E. (2006). Continuous System Simulation, Springer-Verlag, New York, NY.

Jensen, K. (1994). An Introduction to the Theoretical Aspects of Coloured Petri Nets. A Decade of Concurrency, Lecture Notes in Computer Science vol.803. Springer-Verlag 1994, ss. 230-272, www.daimi.au.dk/PB/476/PB-476.pdf

Jensen, K. (1997a). Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts., Springer-Verlag, ISBN 3-540-60943-1

Jensen, K. (1997b). Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods., Springer-Verlag, ISBN 3-540-58276-2

Jensen, K. (1997c). Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use., Springer-Verlag, ISBN 3-540-62867-3

Jensen, K. & Kristensen, L.M. & Wells, L. (2007). Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. International Journal on Software Tools for Technology Transfer, vol. 9, no. 3-4 pp. 213-254

Jensen, K. & Kristensen, L.M. (2009). Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer-Verlag Berlin Heidelberg, doi:10.1007/b95112.

Křivý, I. & Kindler, E. (2001). Simulation and Modelling, University of Ostrava (in Czech), http://prf.osu.cz/kip/dokumenty/Msm.pdf

Law, A.M. (2012) A tutorial on how to select simulation input probability distributions. *In Proceedings of the 2012 Winter Simulation Conference (WSC)*, IEEE, ISBN 978-1-4673-4779-2

Neuschl, Š.; Blatný, J.; Šafařík, J. & Zendulka, J. (1988). Modelling and Simulation. ALFA Bratislava (in Slovak).

Petri, C.A. (1962). Kommunikation mit Automaten, PhD thesis, Institut für Instrimentelle Mathematik, University Bonn

Perros, H. (2009). Computer Simulation Techniques - The Definitive Introduction, http://www4.ncsu.edu/~hp/files/simulation.pdf

Reisig, W. & Rozenberg, G. (Eds.) (1998a) Lectures on Petri Nets I: Basic Models, LNSC vol. 1491, Springer, ISBN: 978-3-540-65306-6

Reisig, W. & Rozenberg, G. (Eds.) (1998b) Lectures on Petri Nets II: Applications, LNSC vol. 1492, Springer, ISBN 978-3-540-65307-3

Reisig, W. (2013). Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies. Springer, ISBN 978-3-642-33278-4

Wang, L.T; Changc, Y.W. & Cheng, K.T. (2009). Electronic Design Automation: Synthesis, Verification, and Test, Elsevier, ISBN: 978-0-12-374364-0

Táto publikácia vznikla za finančnej podpory z **Európskeho sociálneho fondu** v rámci Operačného programu **VZDELÁVANIE.**

Prioritná os 1 Reforma vzdelávania a odbornej prípravy

Opatrenie 1.2 Vysoké školy a výskum a vývoj ako motory rozvoja vedomostnej spoločnosti.

Názov projektu: **Balík prvkov pre skvalitnenie a inováciu vzdelávania na TUKE**

ITMS 2611020070

NÁZOV: Modelovanie a simulácia
AUTOR: Ing. Štefan Korečko, PhD
RECENZENTI: doc. Ing. Branislav Sobota, PhD.
doc. Ing. Milan Šujanský, PhD.
VYDAVATEĽ: Technická univerzita v Košiciach
ROK: 2015
ROZSAH: 96 strán
NÁKLAD: 100 ks
VYDANIE: prvé
ISBN: 978-80-553-2014-4