WEB SIMULATOR CREATION TECHNOLOGY

Jiří Kofránek, Marek Mateják, Pavol Privitzer

Laboratory of Biocybernetics, Dept.of Pathophysiology, 1st Faculty of Medicine, Charles University, Prague, email: kofranek@gmail.com

Abstract

This document includes description of technologies that we use for the creation of web-based tutorials, educational and teaching simulators. When creating a web simulator, two types of problems must be taken into consideration. The first problem is the creation and identification of the mathematical model. This work is more of a research than development work, based on the creation of formalized mathematical description of the modeled reality. For the creation, tweaking/debugging and verification of simulation models, special software development tools are used. For a long time we have mostly been using Matlab/Simulink models, made by Mathworks, for the development process. Simulink belongs to block-oriented simulation languages, which enables the user to assemble computer models from each block with defined inputs and outputs, interconnect these blocks into computer networks and group computer networks into blocks with higher hierarchy. From the description of the block-oriented structure it is clear, how the values of each variable parameter is calculated in the model, that is, what is the algorithm for the relevant calculation process. Recently, we have been using a simulation environment based on the Modelica language. The most important innovation in Modelica is the option to describe each part of the model as a set of equatations and not as an algorithm used to solve these quotations. Models created in Modelica are well-arranged and better reflect the structure of the modeled reality. The other problem apparent during the creation of tutorial and educational simulators is the creation of the tutorial software itself. It is a very demanding development work, which requires the combination of ideas and experiences of teachers who create the script of the tutorial application, the creativity of art designers who create the multimedia components interconnected with the simulation model in the background, as well as the efforts of programmers who finally "sew up" the final masterpiece into its final shape. To automate the model debugging transfer from the simulation development environment (using Simulink or Modelica) into the development environment where the development application is programmed, specialized software tools (developed by us) are used. We have been creating tutorial simulators in ControlWeb development environments (originally designed for the creation of industrial control and measuring applications), in Microsoft .NET and Adobe Flash environments. Recently, we began using the Microsoft Silverlight platform, which enables distribution of simulators over the internet and may be executed directly into the internet browser environment (even on computers running various operating systems).

Keywords: Modeling, Simulation, Simulator

In place of an introduction – a web of physiological regulations

Thirty-six years ago the Annual Review of Physiology published an article (Gyuton et al., 1972) which at a glance was entirely different from the usual physiological articles of that time. It was introduced by a large diagram on an insertion. Full of lines and interconnected elements, the drawing vaguely resembled an electrical wiring diagram at first sight (Fig. 1). However, instead of vacuum tubes or other electrical components, it showed interconnected computation blocks (multipliers, dividers, summators, integrators, functional blocks) that symbolized mathematical operations performed on physiological variables (Fig. 2). Bundles of connecting wires between the blocks indicated the complex feedback interconnection of physiological variables at first glance. The blocks were arranged in eighteen groups that represented individual interconnected physiological subsystems. In the centre was a subsystem representing circulatory dynamics – linked through feedback links with other blocks: From the kidneys to tissue fluids and electrolytes to autonomic nervous control and hormonal control including ADH, angiotensin and aldosterone (Fig. 3).

In this entirely new manner, using graphically represented mathematical symbols, the authors described the physiological regulations of the circulatory system and its broader physiological relations and links with the other subsystems in the body – the kidneys, volumetric and electrolyte balance control,

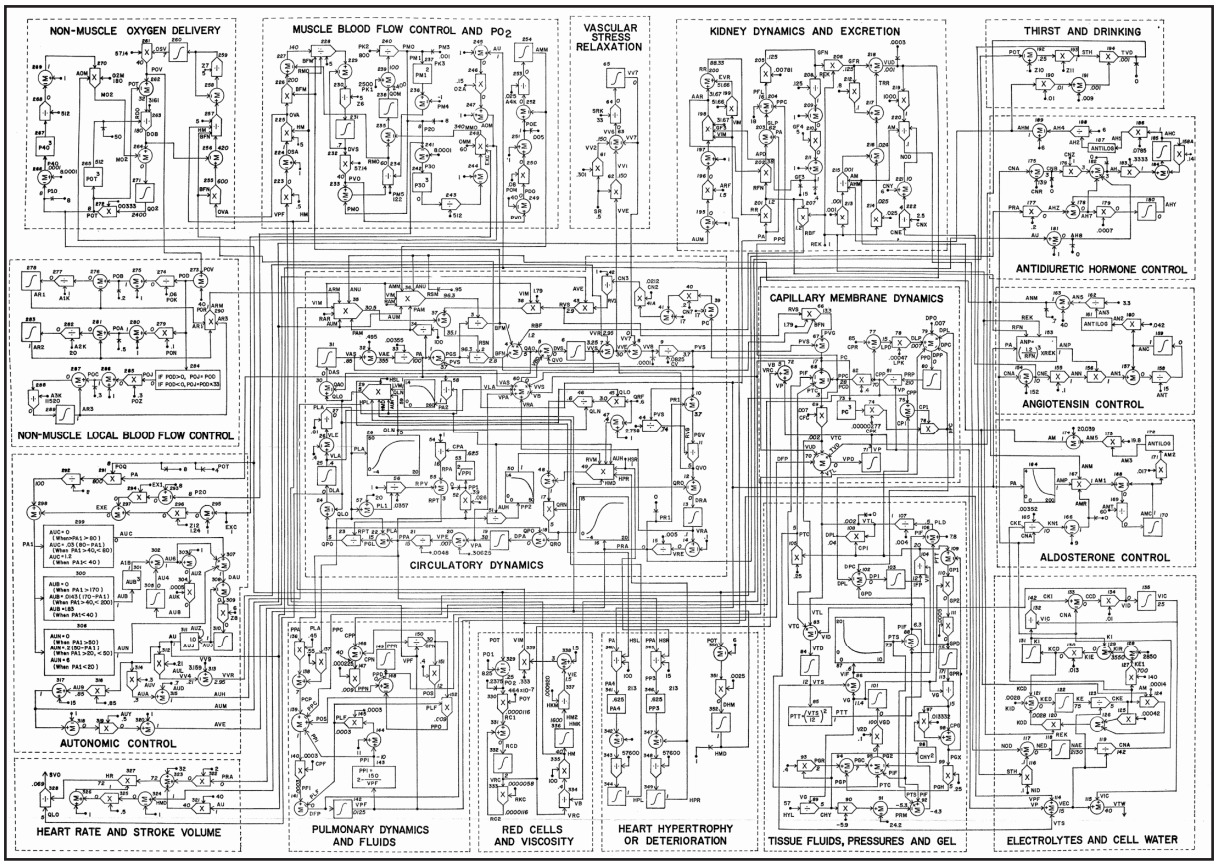


Figure 1: Guyton's blood circulation regulation diagram from 1972.

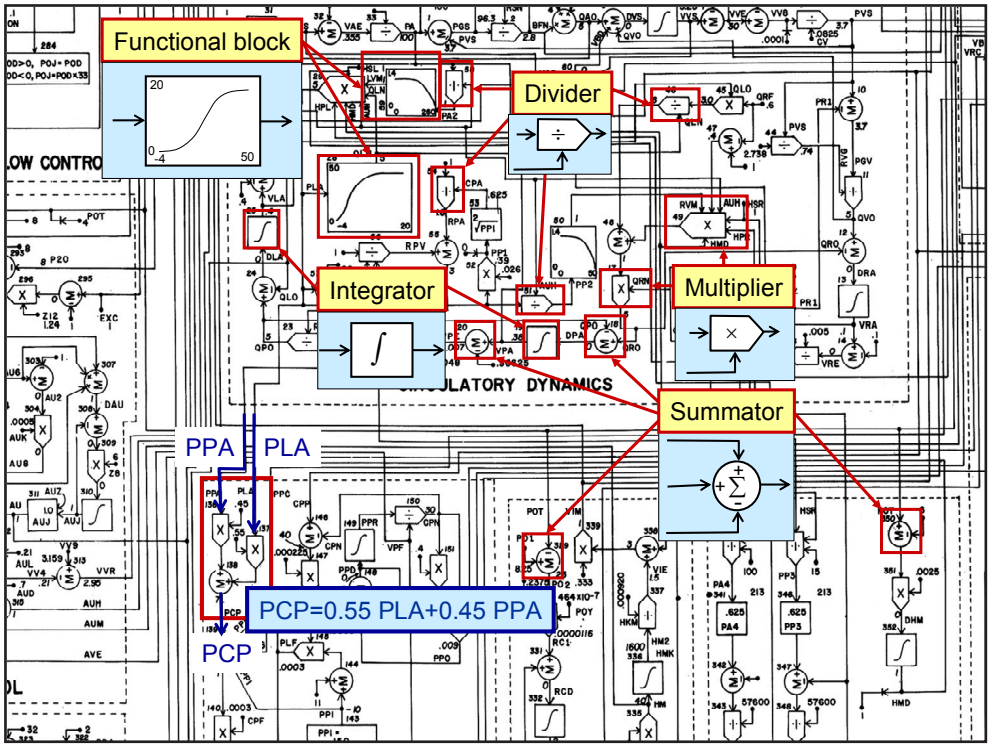


Figure 2: Individual elements in the block diagram of Guyton's model represent mathematical operations, the interconnection of elements represents equations in a graphically expressed mathematical model..

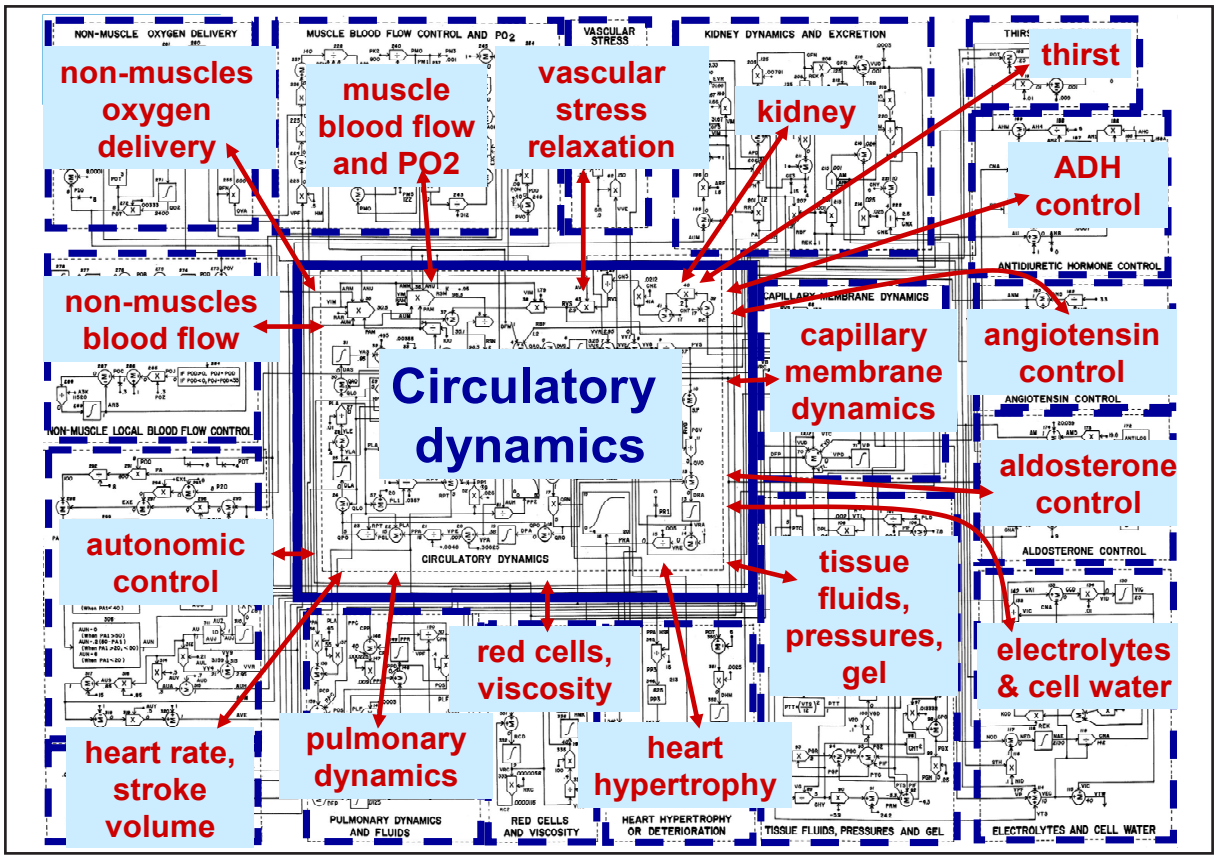


Figure 3: Individual interconnected subsystems in Guyton's model.

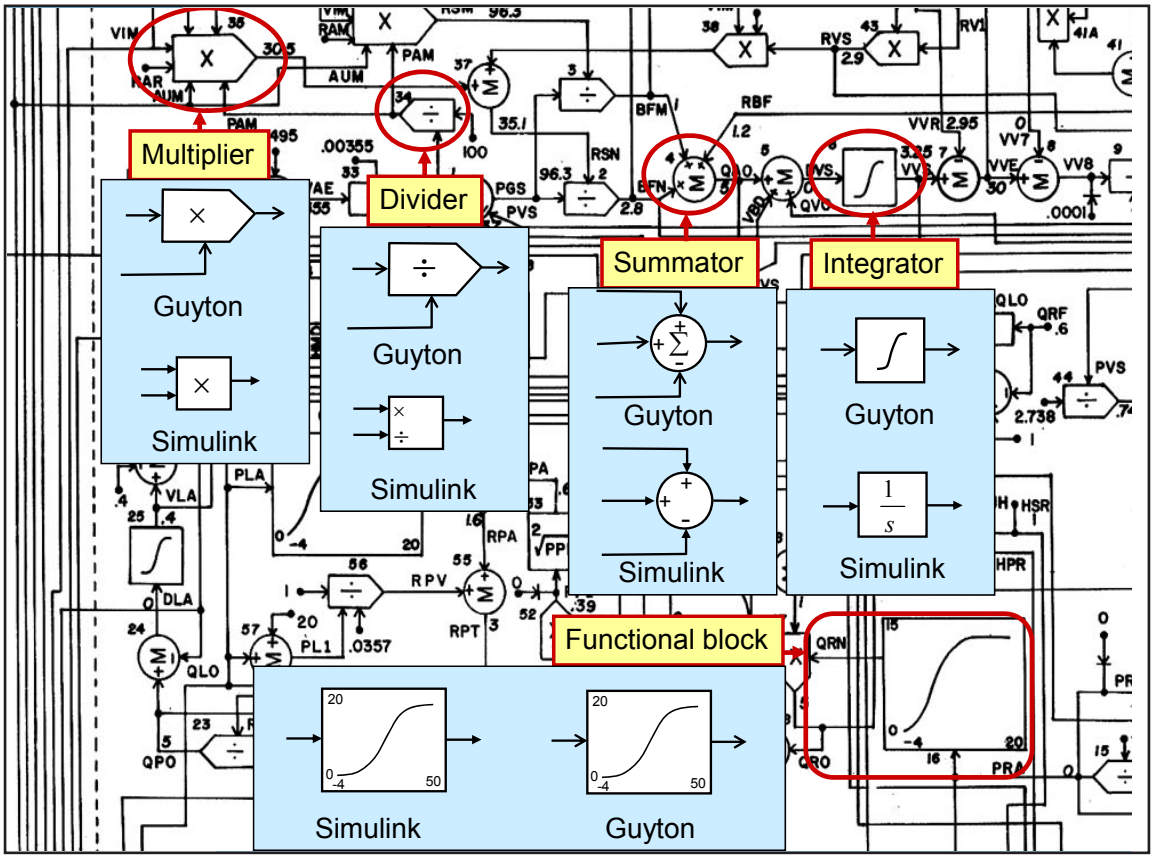


Figure 4: Appearance of blocks in Guyton's original graphical notation and in Simulink.

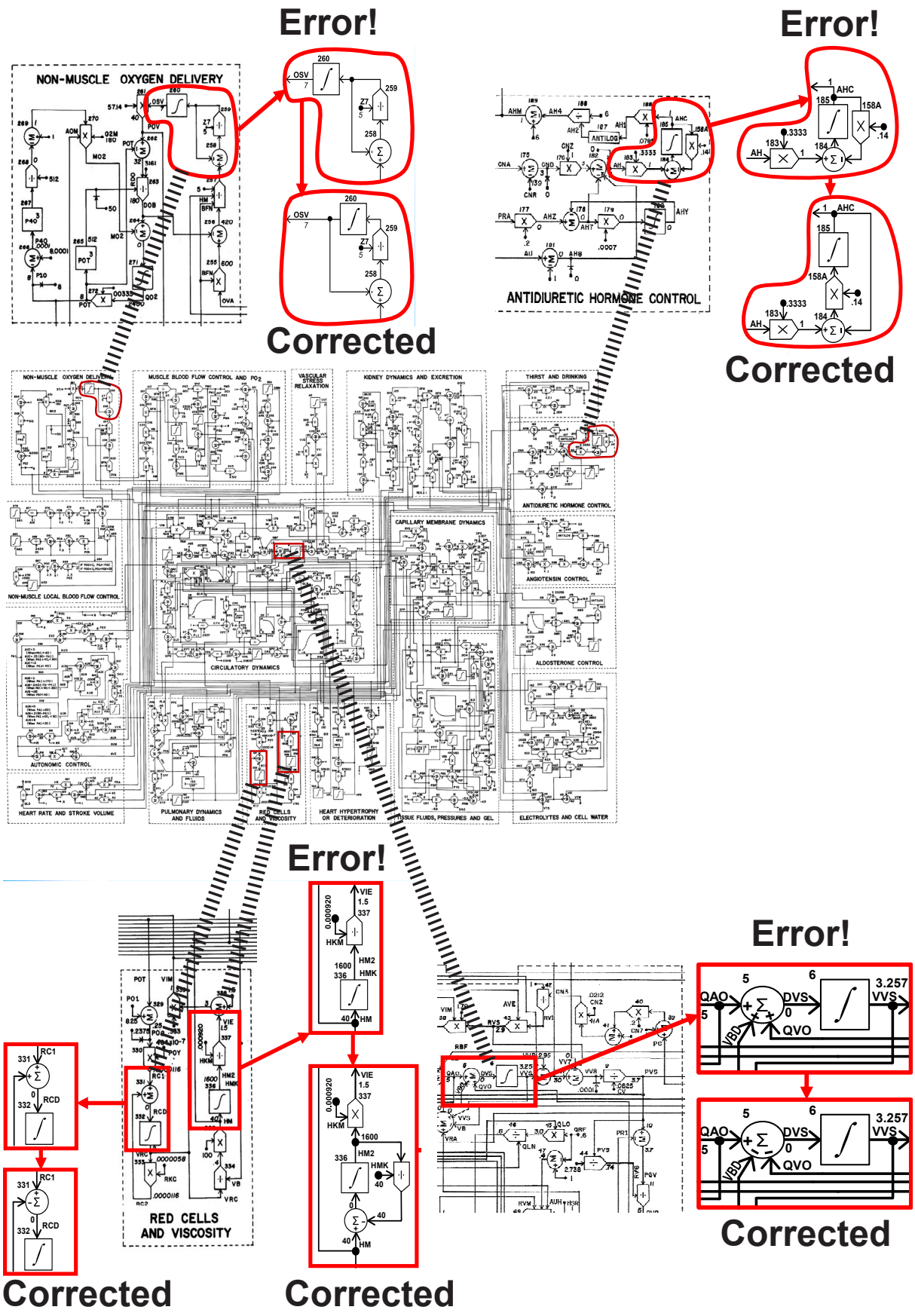


Figure 5: Correction of errors in Guyton's original diagram.

etc. Instead of an extensive set of mathematical equations, the article used a graphical representation of mathematical relations. This syntax allowed depicting relations between individual physiological variables graphically in the form of interconnected blocks representing mathematical operations. The whole diagram thus featured a formalized description of physiological relations in the circulatory system using a graphically represented mathematical model.

The actual description of the model in the article was mostly represented by the elementary (but fully illustrative) drawing. Comments on and reasons for the formulation of the mathematical relations were very brief, e.g.: "Blocks 266 through 270 calculate the effect of cell pO2, autonomic stimulation, and basic rate of oxygen consumption by the tissues on the actual rate of oxygen consumption by the tissues". This required exceptional concentration (and certain physiological and mathematical knowledge) from the reader to be able to understand the formalized relations between physiological variables.

A monograph (Guyton et al., 1973) published a year later, in 1973, explained a number of the adopted approaches in greater detail.

We do not see a mathematical representation of reality very often in biology and medicine. It should be noted that the process of formalization, i.e. the translation of a purely verbal description of a given network of relations into the formalized language of mathematics, is delayed in biological and medical sciences in comparison with engineering sciences, physics or chemistry. While the process of formalization in physics started as early as the seventeenth century, in medical and biological sciences it has been relatively delayed due to the complicacy and complexity of biological systems and has only advanced with cybernetics and computer technology. The methodological tool here is computer models built on a mathematical description of biological reality.

Formalized descriptions in physiology have been used since the late sixties (since the pioneering works of Grodins et al., 1967, describing respiration). Guyton's model was the first extensive mathematical description of the physiological functions of interconnected body subsystems and launched the field of physiological research that is sometimes described as "integrative physiology" today. Just as theoretical physics tries to describe physical reality and explain the results of experimental research using formal means, "integrative physiology" strives to create a formalized description of the interconnection of physiological controls based on experimental results and explain their function in the development of various diseases.

From this point of view, Guyton's model was a milestone, trying to adopt a systematic view of physiological controls to capture the dynamics of relations between the regulation of the circulation, kidneys, the respiration and the volume and ionic composition of body fluids by means of a graphically represented network.

Guyton's graphical notation of a formalized description of physiological relations provides a very clear representation of mathematical correlations – the blocks in network nodes represent graphical symbols for individual mathematical operations and the wires represent individual variables. Guyton's graphical notation was soon adopted by other authors – such as Ikeda et al. (1979) in Japan and Amosov et al. (1977) in the former USSR.

However, the graphical notation of the mathematical model using a network of interconnected blocks was only visualization when created – Guyton's model and later modifications (as well as the models of other authors that adopted Guyton's representative notation) were originally implemented in Fortran and later in C++.

Today's situation is different.

Today, there are specialized software simulation environments available for the development, debugging and verification of simulation models, which allow creating a model in graphical form and then testing its behaviour. One of these is the Matlab/Simulink development environment by Mathworks, which allows building a simulation model gradually from individual components – types of software

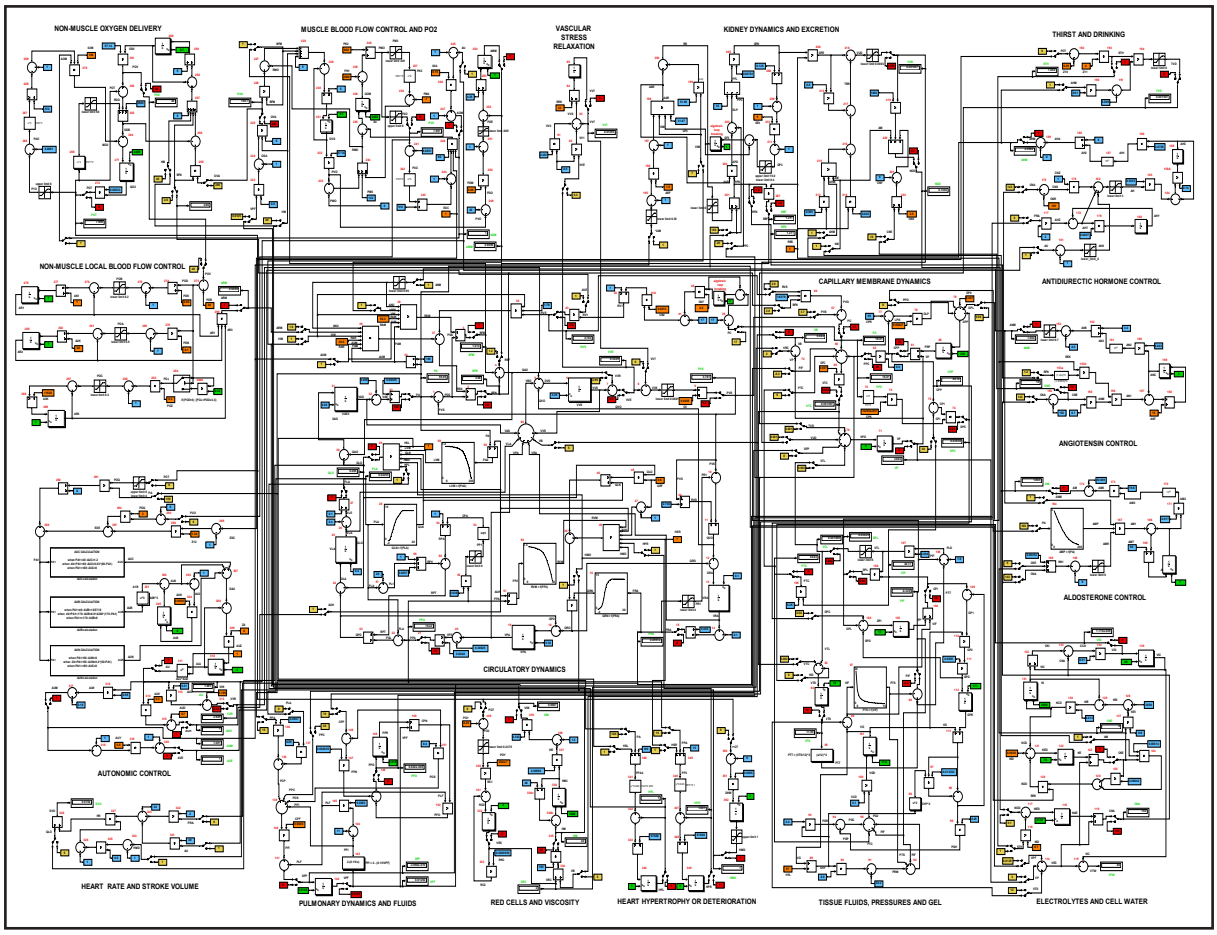


Figure 6: The implementation of Guyton's model in Simulink preserves the original arrangement of elements in Guyton's graphic diagram.

simulation elements that are interconnected using a computer mouse to form simulation networks. Simulink blocks are very similar to the elements used by Guyton for the formalized representation of physiological relations. The only difference is in their graphical form (see Fig. 4).

This similarity inspired us to use Simulink to revive Guyton's good, classic diagram and transform it into a working simulation model. When implementing the model in Simulink, we used switches that allow us to connect and disconnect individual subsystems and control loops while the model is running. We strove to keep the appearance of the Simulink model identical to the original graphic diagram – the arrangement, wire location, variable names and block numbers are the same.

The simulation visualization of the old diagram was not without difficulties – there are errors in the original graphic diagram of the model! It does not matter in the hand-drawn illustration but if we try to bring it to life in Simulink, the model as a whole collapses immediately. There weren't too many errors – switched signs, a divider instead of a multiplier, mixed-up interconnections between blocks, a missing decimal point in a constant, etc. However, there were enough to prevent the model from working. Some of the errors could be seen at first sight (even with no knowledge of physiology) – it is obvious from the diagram that the value of some variables in some integrators would quickly grow to infinity in operation (because of incorrectly drawn feedback) and the model would collapse. With a knowledge of physiology and system analysis, however, all of the errors could be identified with some work (Fig. 5). A detailed description of the errors and their corrections is in Kofránek, Rusz and Matoušek, 2007.

It is interesting that Guyton's diagram as a complex drawing was reprinted many times in various publications (recently see e.g. Hall, 2004, Van Vliet and Montani, 2005). However, nobody mentioned the errors or made an effort to correct them. This was understandable at the time the diagram was cre-

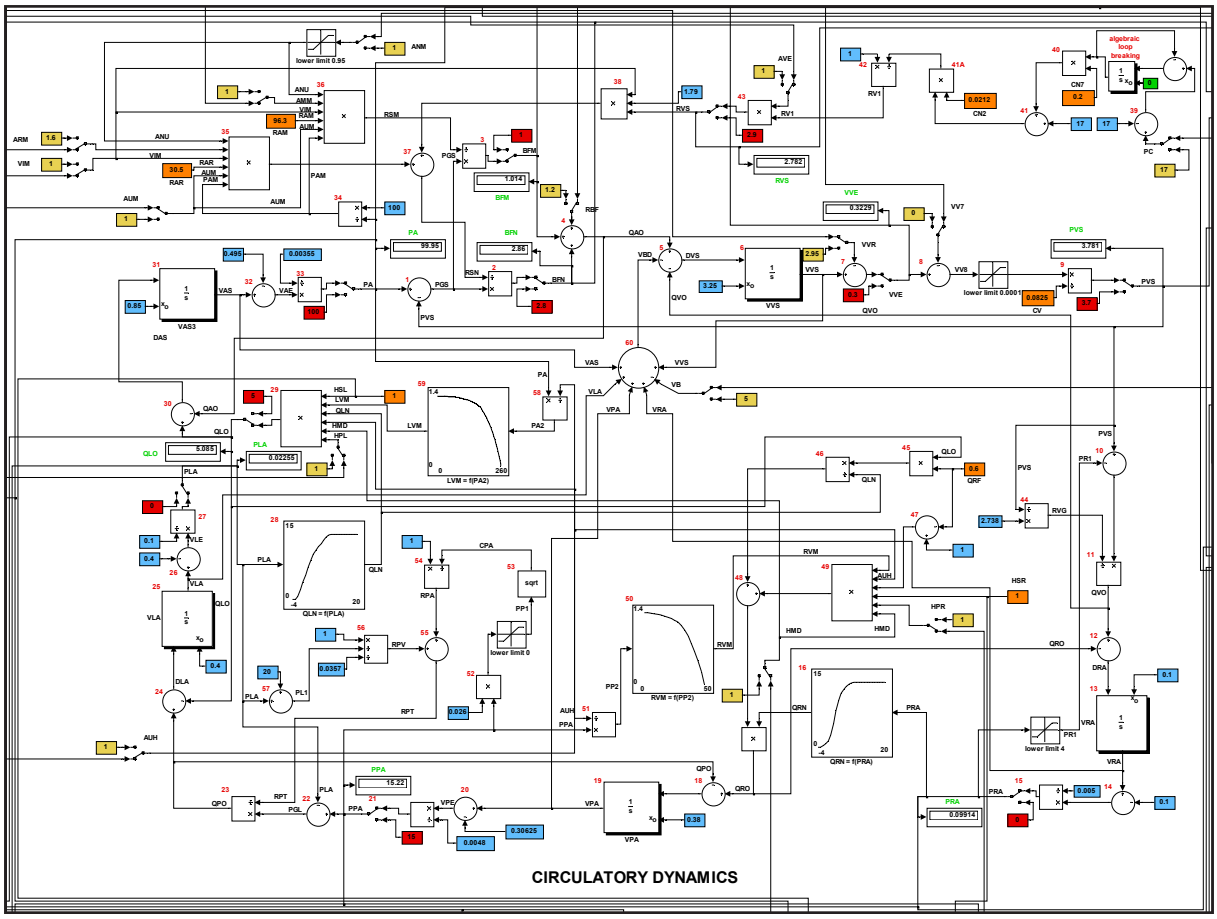


Figure 7: More detailed central structures of the Simulink implementation of Guyton's model, representing flows through aggregated parts of the circulatory system and the activity of the heart as a pump.

ated. Drawing software did not exist – the diagram was created as a complex drawing – and redrawing the complex diagram manually was not easy. It is also possible that the authors of the model did not wish to correct the errors – those who went to the trouble of analyzing the model could spot the drawing's "typos" and those who would just like to unthinkingly copy the diagram were out of luck. After all, the authors used to send out the source codes of Fortran programs for their model – so if anybody wished just to test the behaviour of the model, they did not have to program anything (at most, they had to routinely convert the program from Fortran to another programming language).

Our Simulink implementation of Guyton's (corrected) model (Figs. 6 and 7) is available for download at www.physiome.cz/guyton. Also available at that address is our Simulink implementation of a much more complex, later model from Guyton et al. There is also a very detailed description of all applied mathematical relations with an explanation.

Block-oriented simulation networks

Guyton's block diagram augured the rise of visual, block-oriented simulation languages. However, Guyton and his colleagues implemented the model in Fortran back in 1972 – Simulink version 1 was released eighteen years later (in 1990). Block-oriented simulation languages, of which Simulink is a typical example, allow assembling computer models from individual blocks with defined inputs and outputs. The blocks are grouped in libraries; when building a model, a computer mouse is used to create individual block instances, with inputs and outputs connected through wires that "conduct" information.

A Simulink network can be arranged hierarchically. Blocks can be grouped into subsystems that

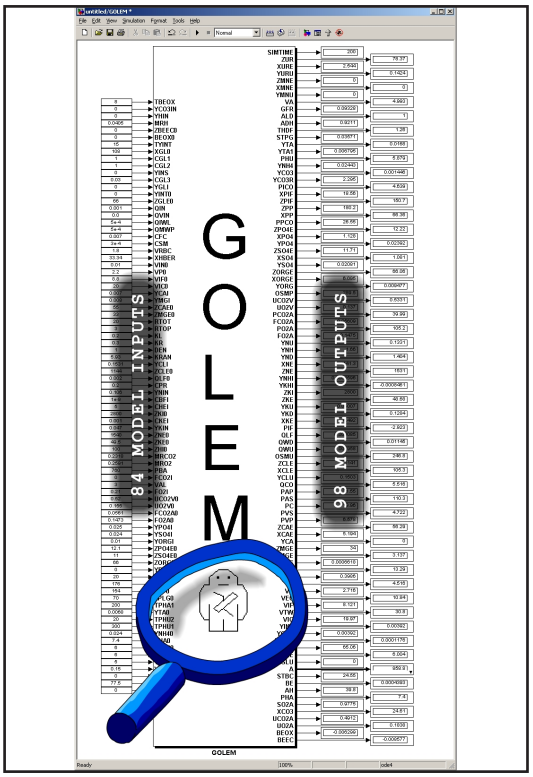


Figure 8: An example of a simulation chip (here representing the simulation model that is the basis for the GOLEM simulator (Kofránek et al., 2001). Its behaviour can be tested easily in the Simulink environment – input values (or value patterns) can be fed to the "input pins" and outputs or the time behaviour of outputs can be read out from the "output pins" by means of virtual displays or oscilloscopes. The next illustration shows the inside of this chip.

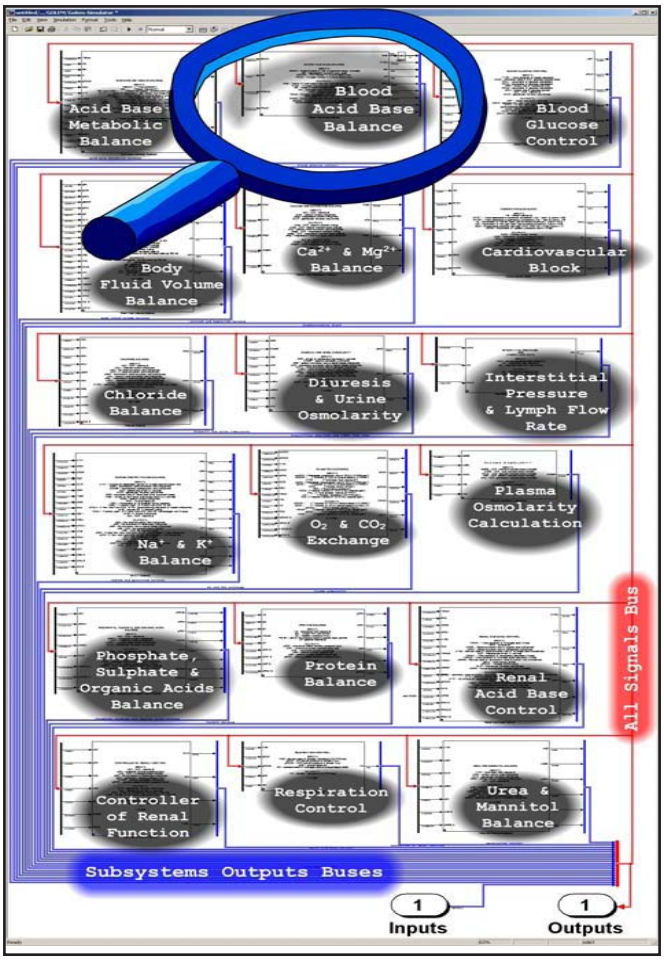


Figure 9: The "inside" of the simulation chip from the previous illustration. The structure resembles an electric network with interconnected integrated circuits, which represent simulation chips of a lower hierarchical level here. The next illustration shows the contents of the "Blood Acid Base Balance" chip.

communicate with their ambient environment through defined input and output "pins", making "simulation chips" of a sort. A simulation chip hides the simulation network structure from the user, much like an electronic chip hiding the interconnection of transistors and other electronic elements. Then the user can be concerned just with the behaviour of the chip and does not have to bother about the internal structure and calculation algorithm. The behaviour of a simulation chip can be tested by monitoring its outputs using virtual displays or virtual oscilloscopes connected to it. This is very useful especially for testing the behaviour of a model and expressing the mutual relations of variables.

The entire complex model can be then visualized as interconnected simulation chips and the structure of their interconnection clearly shows what effects are taken into account in the model, and how (Figs. 8–11).

This is very useful for interdisciplinary collaboration – especially in borderline fields such as biomedical system modelling (Kofránek et al., 2002). An experimental physiologist does not have to examine the details of mathematical relations hidden "inside" a simulation chip; however, from the mutual interconnection of simulation chips they will understand the model structure and will be able to check its behaviour in the appropriate simulation visualization environment.

Simulation chips can be stored in libraries and users can create their instances for use in their

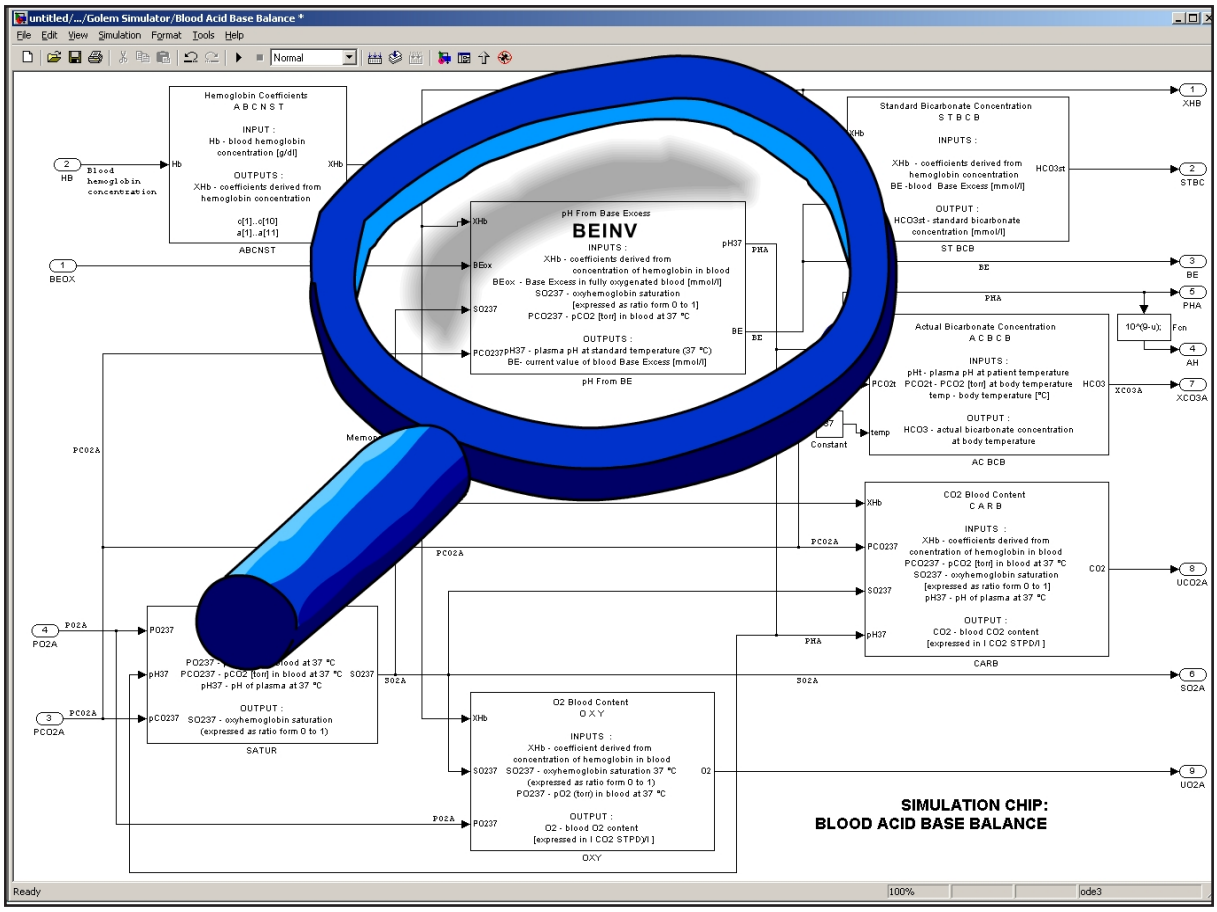


Figure 10: Simulation chips are arranged hierarchically. The illustration shows the "inside" of one of the simulation chips from Fig. 9. Since each simulation chip includes sufficiently detailed documentation of its inputs and outputs, the structure of relations inside a simulation chip (representing the physiological relations in a real organism) can be understood by physiologists. The next illustration shows the contents of the "BEINV" chip.

models (Fig. 12). For example, we created a Physiolibrary for modelling physiological regulations (http://www.physiome.cz/simchips).

Hierarchical, block-oriented simulation tools are thus used advantageously in the description of the complex regulation systems that we have in physiology. A formalized description of physiological systems is the subject matter of PHYSIOME, an international project that is a successor to the GENOME project. The output of the GENOME project was a detailed description of the human genome; the goal of the PHYSIOME project is a formalized description of physiological functions. It uses computer models as its methodological tool (Bassingthwaighte, 2000; Hunter et al., 2002).

Several block-oriented simulation tools developed under the PHYSIOME project have been used as a reference database for a formalized description of the structure of complex physiological models. These include JSIM (http://www.physiome.org/model/doku.php) and CEllML (http://www.cellml.org/).

Prof. Guyton's disciples and followers have expanded the original, extensive simulator of the circulatory system - Quantitative Circulatory Physiology (Abram et al., 2007) with an integrated connection of all important physiological systems. The latest result is the Quantitative Human Physiology simulator (Hester et al., 2008) represents today's most comprehensive and largest model of physiological functions. The model can be downloaded from http://physiology.umc.edu/themodelingworkshop/. The authors developed a special block-oriented simulation system to represent the complex model structure.

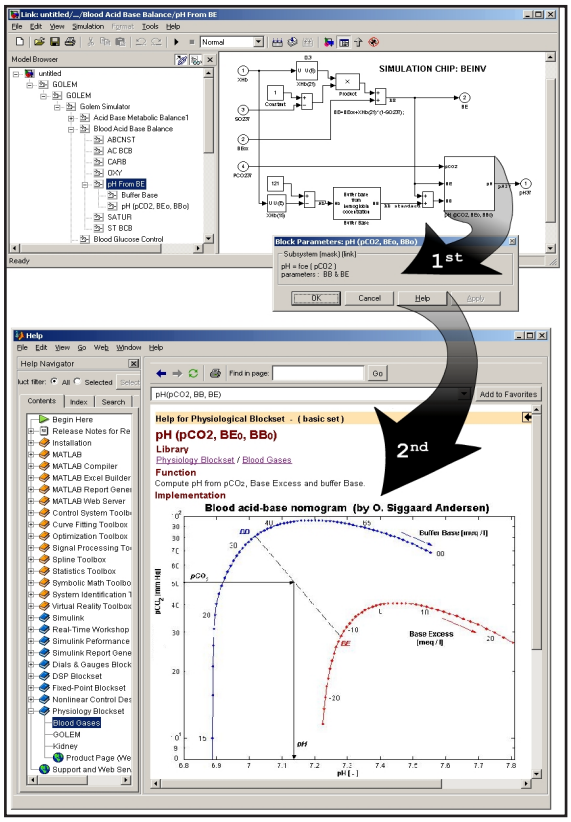


Figure 11: Structure of the BEINV simulation chip. Simulation chips at the lowest hierarchical level consist in interconnected elementary components of the Simulink development systems (adders, multipliers, integrators, etc.) and possibly simulation chips of further lower levels. These interconnected elements represent individual mathematical relations. Usually, each chip is dynamically linked with the appropriate documentation page including a factual description of the chip's function, including a description of the mathematical relations that constitute its basis.

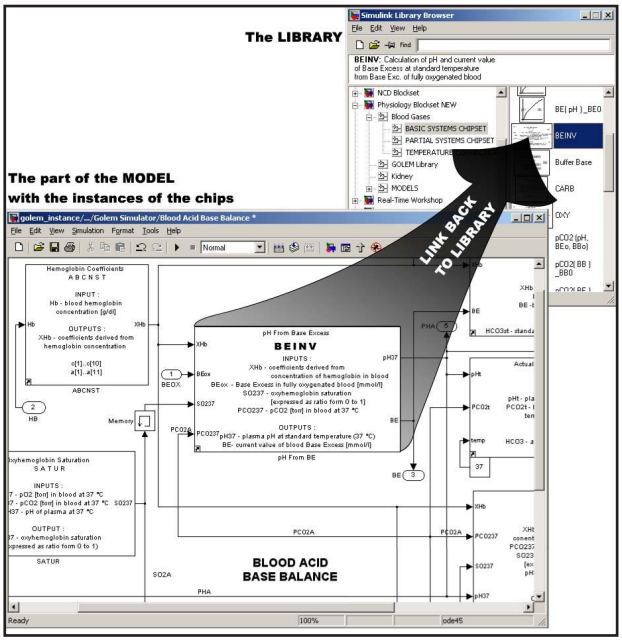


Figure 12: Simulation chips can be gathered in hierarchically arranged libraries in Simulink. It is then possible to use the mouse to "pick up" individual chips from the libraries (as from a tool palette), place them into the created application, interconnect them and create more complex models.

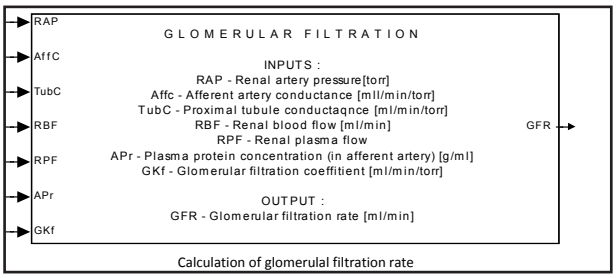


Figure 13 The "Glomerular Filtration" simulation chip calculates the glomerular filtration rate. The structure of calculation is hidden from the user. However, there must be no algebraic loops inside a chip.

Causal and acausal approaches

Block-oriented tools work with hierarchically interconnected blocks. The connections between blocks "conduct" signals that transmit the values of individual variables from the output of a block to the inputs of other blocks. The blocks process input information into output information.

The hierarchically arranged block-oriented description clearly shows how the values of individual variables are calculated in the model – i.e. what the calculation algorithm is.

However, the interconnection of blocks in a network of relations cannot be completely arbitrary. Interconnected elements **may not include any algebraic loops** – i.e. cyclic structures where an input value fed to the input of a calculation block depends (through several intermediate blocks) on the block's output value in the same time step.

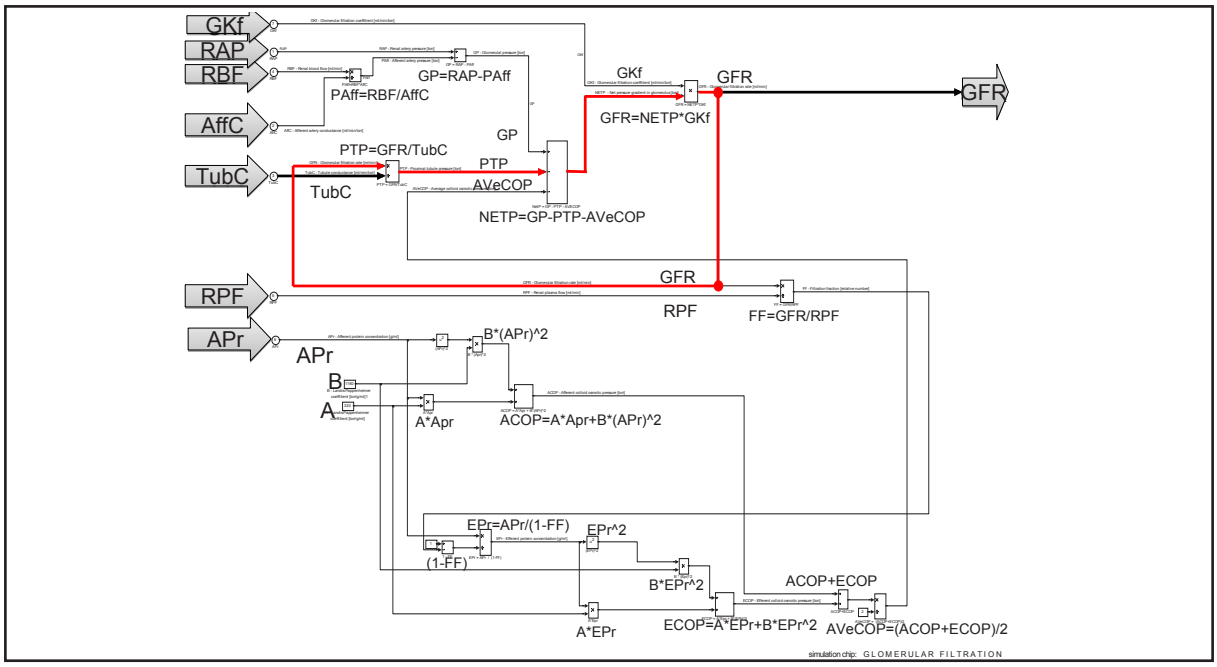


Figure 14: The interconnection of individual blocks inside the "Glomerular Filtration" simulation chip graphically represents individual mathematical relations for the calculation of the glomerular filtration rate. However, there is an algebraic loop. It is necessary to break the loop.

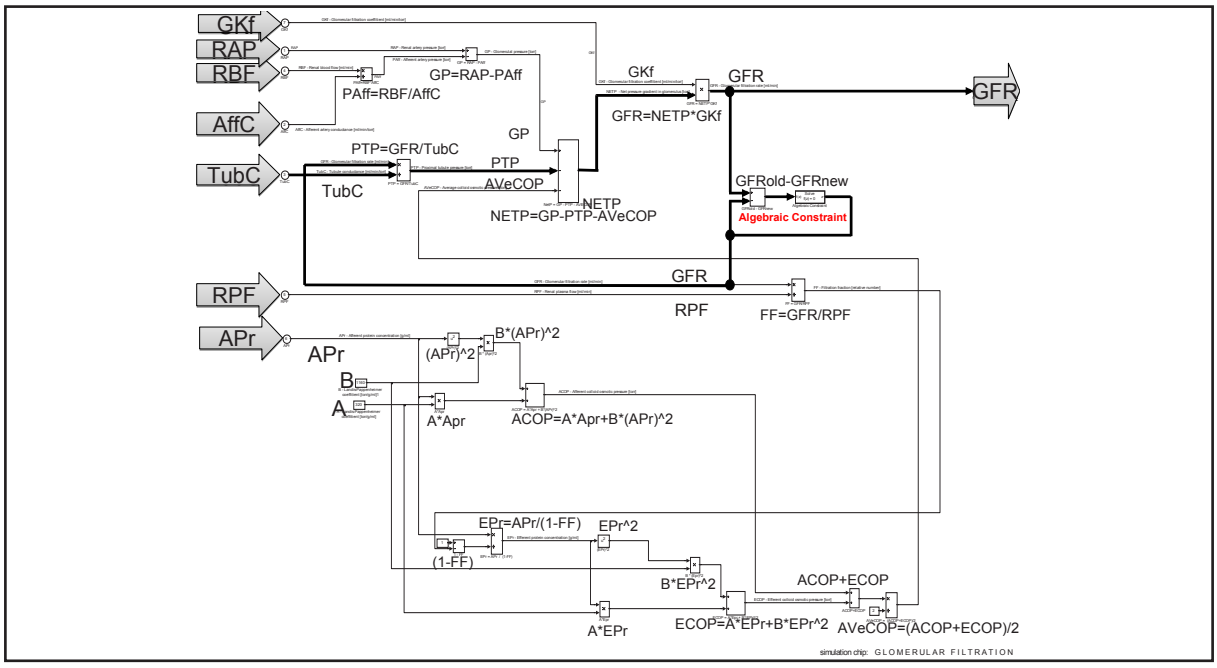


Figure 15: Breaking the algebraic loop in the calculation of the glomerular filtration rate. The interconnection of Simulink blocks reflects the calculation procedure rather than a graphical representation of mathematical relations.

For illustration, let us consider the small example of an algebraic loop in Simulink, a block-oriented language.

A model of the kidneys uses a simulation chip calculating the glomerular filtration rate. The individual inputs and outputs of that chip are shown in Fig. 13. The inside of the simulation chip consists in elementary blocks performing mathematical operations. The value of *GFR*, a variable representing

the glomerular filtration rate, is calculated from the value of *NETP*; to calculate *NETP* it is necessary to know the value of *PTP*, which is however calculated as the quotient of *GFR* and *TUBC* (Fig. 14). Our Simulink diagram contains an algebraic loop that must be broken. Therefore we solve an implicit equation in the blocks identified as "Algebraic Constraint" in Fig. 15 to calculate *GFR* in each integration step.

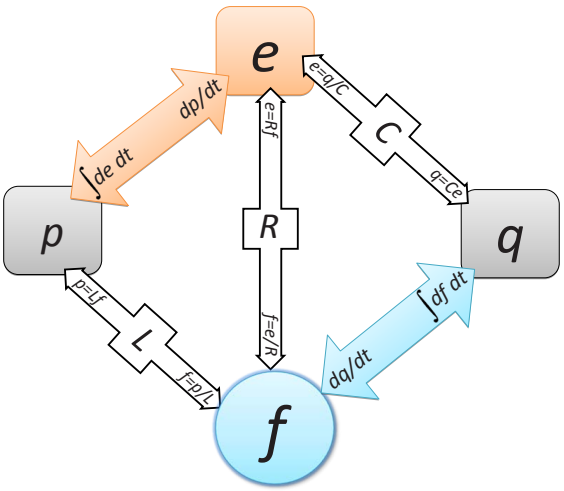
Therefore, a Simulink network does not constitute the graphical representation of mathematical relations in a model; rather, it is the graphical representation of a chain of transformations from input values to output values through Simulink elements where loops are not allowed.

If we focus on the representation of a structure of mathematical relations rather than the algorithm of calculations when building a model in Simulink, we can easily introduce algebraic loops into the model (however, the compiler will warn us about this). There are methods that can be used to get rid of algebraic loops (e.g. see Dabney and Harman , 2004) – however, they lead to transformations that make the model structure even more complex and the model more difficult to understand. The need to have a fixed direction of connection from inputs to outputs with no algebraic loops also makes model building more difficult.

The interconnection of blocks in Simulink thus ***reflects the calculation procedure rather than the actual structure of the modelled reality.*** We call this ***causal modelling.***

In complex systems, the ***physical reality of the modelled system becomes somewhat lost under the structure of calculation*** with this approach.

New, ***"acausal" tools*** have recently been developed for the creation of simulation models. The major innovation brought about by acausal modelling tools is the possibility to describe the individual parts of a model directly ***as a system of equations rather than an algorithm for solving the equations***. The notation of models is declarative (we describe the structure and mathematical relations, not the cal-



Generalized system properties:

- ***e*** *means generalized effort – corresponding to force in mechanics, voltage in electrical diagrams, pressure in hydraulics, etc.*
- ***f*** *is generalized flow – corresponding to velocity in mechanics, current in electrical diagrams, flow rate in hydraulics, temperature flow in thermodynamics, etc.*
- ***q*** *is generalized accumulation or deflection, representing the integral of the generalized flow. It corresponds e.g. to the stretching of a spring in mechanics, fluid volume in hydraulics, charge in electrical diagrams, accumulated heat in thermodynamics, etc.*
- ***p*** *is generalized momentum (inertance) – the integral of the generalized effort, representing kinetic energy; in hydraulics it represents the change of the flow rate proportional to the pressure difference (flow momentum), in electrical circuits it is the potential needed to change an electric current (induction), etc.*
- ***R***, ***C*** *and* ***L*** *represent constants of proportionality between the generalized system properties. They correspond e.g. to resistance, capacitance or weight.*

Figure 16: Relations between generalized system properties.

culation algorithm) – thus the notation is acausal.

Acausal modelling tools work with interconnected components that are instances of classes in which equations are directly defined.

The components (i.e. instances of classes with equations) can be interconnected by means of precisely defined interfaces – connectors; this defines a system of equations.

The latest version of Simulink provides certain options for using acausal tools as well. Mathworks, the producer of the Matlab/Simulink simulation tools, responded to the new trends by creating a special acausal Simulink library – Simscape – and related domain libraries such as SimElectronics, SimHydraulics, SimMechanics, etc.

A modern simulation language that is built directly on the acausal notation of models is ***Modelica*** (Fritzson, 2003). It was originally developed in Sweden and is now available both in an open-source version (developed under the auspices of Modelica Association, http://www.modelica.org/) and in two commercial implementations.

The first commercial implementation is made by Dynasim AB – which has been bought by Dassault Systemes, a multinational corporation (sold under the name of ***Dymola***, currently in version 7.3), and the other commercial implementation is made by MathCore (sold under the name of ***MathModelica***). Dynasim's Modelica has a good connection to the Matlab and Simulink simulation tools, while MathModelica can connect to the Mathematica environment made by Wolfram.

Modelica works with interconnected components that are instances of individual classes. Unlike the implementation of classes in other object-oriented languages (such as jw C# or Java), classes in Modelica have an additional special section in which ***equations*** are defined.

The equations do not mean assignment (i.e. storing the result of the calculation of an assigned command in a variable) but rather the definition of relations between variables (as is common in mathematics and physics). For example, the following notations of relations between variables expressing the resistance (R), flow (F) and pressure gradient (P) are equivalent:

$F=P/R$

$P/R=F$

$P=R*F$

$R*F=P$

$R=P/F$

$P/F=R$

Components (class instances) in Modelica can be interconnected by means of precisely defined interfaces – ***connectors***.

What is important is that the interconnection of components actually ***interconnects systems of equations in the individual components*** with one another. ***By interconnecting Modelica components, we do not define the calculation procedure but rather the modelled reality. The method of solving the equations is then "left to the machines".***

Generalized system properties

The representation of a model in an acausal simulation environment resembles the physical reality of the modelled world more than the standard interconnected block diagrams in causal modelling tools. This is associated with the generalized system properties of the real world (Fig. 16), where an important role is played by ***generalized effort*** (corresponding to force, pressure, voltage, etc. in the real world) and ***generalized flow*** (corresponding to current, flow rate, etc. in the real world). The integral of generalized

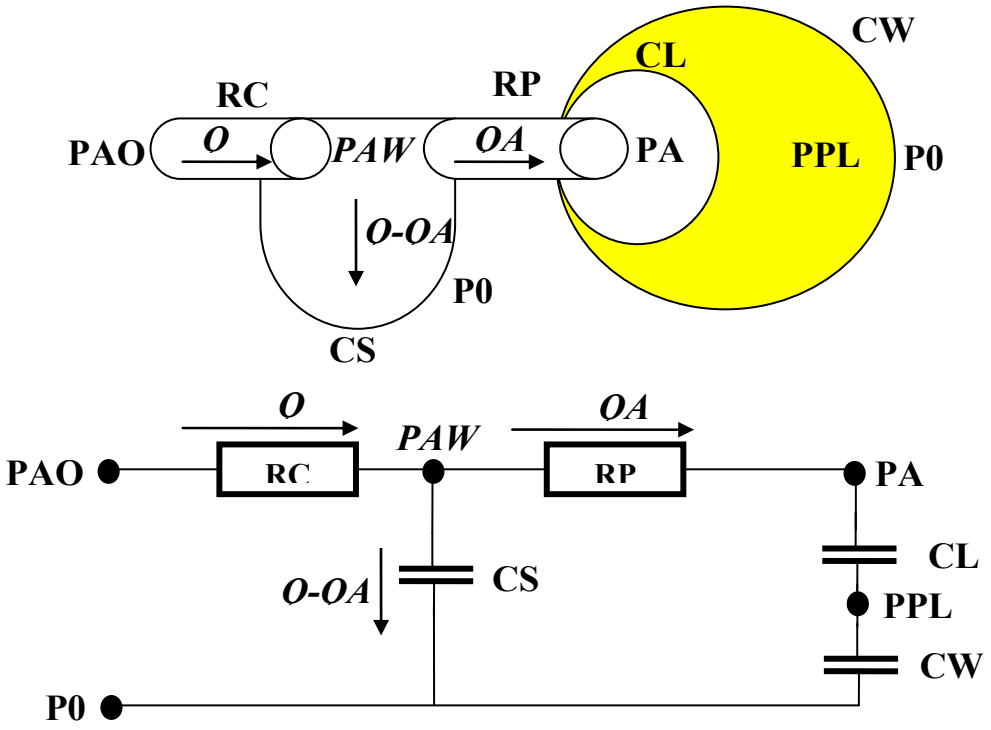


Figure 17: A simple pulmonary mechanics model (hydraulic and electrical analogy).

flow is ***generalized accumulation or deflection*** (in the real world, this can be e.g. an electrical charge but also the volume of a liquid or gas, stretching of a spring, accumulated heat, etc.). The integral of generalized effort is ***generalized momentum*** (this represents flow momentum in hydraulics, induction in electrical circuits, etc.).

Also related to the generalized system properties is the fact that descriptions of models of biological or physiological processes often use electrical or hydraulic analogies for reasons of clarity.

Let us illustrate the utilization of generalized system properties and the difference between modelling in block-oriented simulation tools and in Modelica with a physiological reality modelling example – a model of simple pulmonary ventilation mechanics.

Let us consider a ***simple pulmonary mechanics model*** that is schematically shown in Fig. 17. With a high level of simplification, the lungs can be seen as three bags interconnected through two tubes. The lungs are connected to the fan of the artificial pulmonary ventilation equipment, which periodically drives air into the lungs with the pressure *PAO*. *P0* is the pressure of the ambient atmosphere. Airflow *Q* runs through the upper respiratory tract that has the resistance *RC*. From the upper respiratory tract, air forces its way through the lower respiratory tract to the alveoli. The resistance of the lower respiratory tract is *RP*, the pressure in the central parts of the respiratory tract (at the boundary between the upper and lower respiratory tracts) is *PAW*, the pressure in the alveoli is *PA*.

Air expands the pulmonary alveoli, whose compliance is *CL* (as the total compliance of the lungs). The interpleural cavity is in between the lungs and the rib cage. The pressure in it is *PPL*. The chest has to expand as well during artificial pulmonary ventilation when air is forced into the lungs – the chest compliance is *CW*. The small portion of air that does not reach the alveoli expands the respiratory tract instead – its compliance is *CS*.

Now we can set up our equations. According to Ohm's law, it must be true that:

$$PAW - PA = RP\,QA$$
$$PAO - PAW = RC\,Q \tag{1}$$

The relation between the compliance, pressure gradient and volume (expressed as the integral of the flow rate) is expressed by these equations:

$$PA - PPL = \frac{1}{CL}\int QA\,dt$$
$$PPL - P0 = \frac{1}{CW}\int QA\,dt$$
$$PAW - P0 = \frac{1}{CS}\int (Q - QA)\,dt \tag{2}$$

According to the generalized Kirchhoff's law, the sum of all pressures (voltages) along a closed loop must be equal to zero, i.e. the following must hold true for the loop along the node PAW and along the node PA0:

$$(PAW - PA) + (PA - PPL) + (PPL - P0) + (P0 - PAW) = 0$$
$$(PAO - PAW) + (PAW - P0) + (P0 - PAO) = 0 \tag{3}$$

Substituting from the equations for Ohm's law and compliances, we get:

$$\begin{cases} RP\,QA + \left(\frac{1}{CL} + \frac{1}{CW}\right)\int QA\,dt - \frac{1}{CS}\int (Q - QA)\,dt = 0 \\ Q\,RC + \frac{1}{CS}\int (Q - QA)\,dt + (P0 - PAO) = 0 \end{cases} \tag{4}$$

Causal approach – implementation of the pulmonary ventilation mechanics model in Simulink

When building a model in Simulink, we have to define precisely the ***procedure of calculation*** from input variables to output variables. If we wish to calculate the reaction of the air flow to/from the lungs (***Q***) to the input – i.e. to the changes in pressure at the beginning of the respiratory tract (***PAO***) caused by the artificial pulmonary ventilation apparatus – the Simulink model will look like Fig. 18.

We can also simplify the Simulink model. First we obtain a differential equation (input variable ***PAO***, output ***Q***) from equations (4):

$$\frac{d^2PAO}{dt^2} + \frac{1}{RP\;CT}\frac{dPAO}{dt} = RC\frac{d^2Q}{dt^2} + \left(\frac{1}{CS} + \frac{RC}{RP\,CT}\right)\frac{dQ}{dt} + \frac{1}{RP\,CS}\left(\frac{1}{CL} + \frac{1}{CW}\right)Q \tag{5}$$

When we enter the numeric parameters of resistance (in units cm H_2O/L/sec) and compliance (in units L/cmH_2O) (Khoo, 2000):

$$RC = 1; \quad RP = 0,5; \quad CL = 0,2; \quad CW = 0,2; \quad CS = 0,005 \tag{6}$$

the equation (5) simplifies:

$$\frac{d^2PAO}{dt^2} + 420\frac{dPAO}{dt} = \frac{d^2Q}{dt^2} + 620\frac{dQ}{dt} + 4000\,Q \tag{7}$$

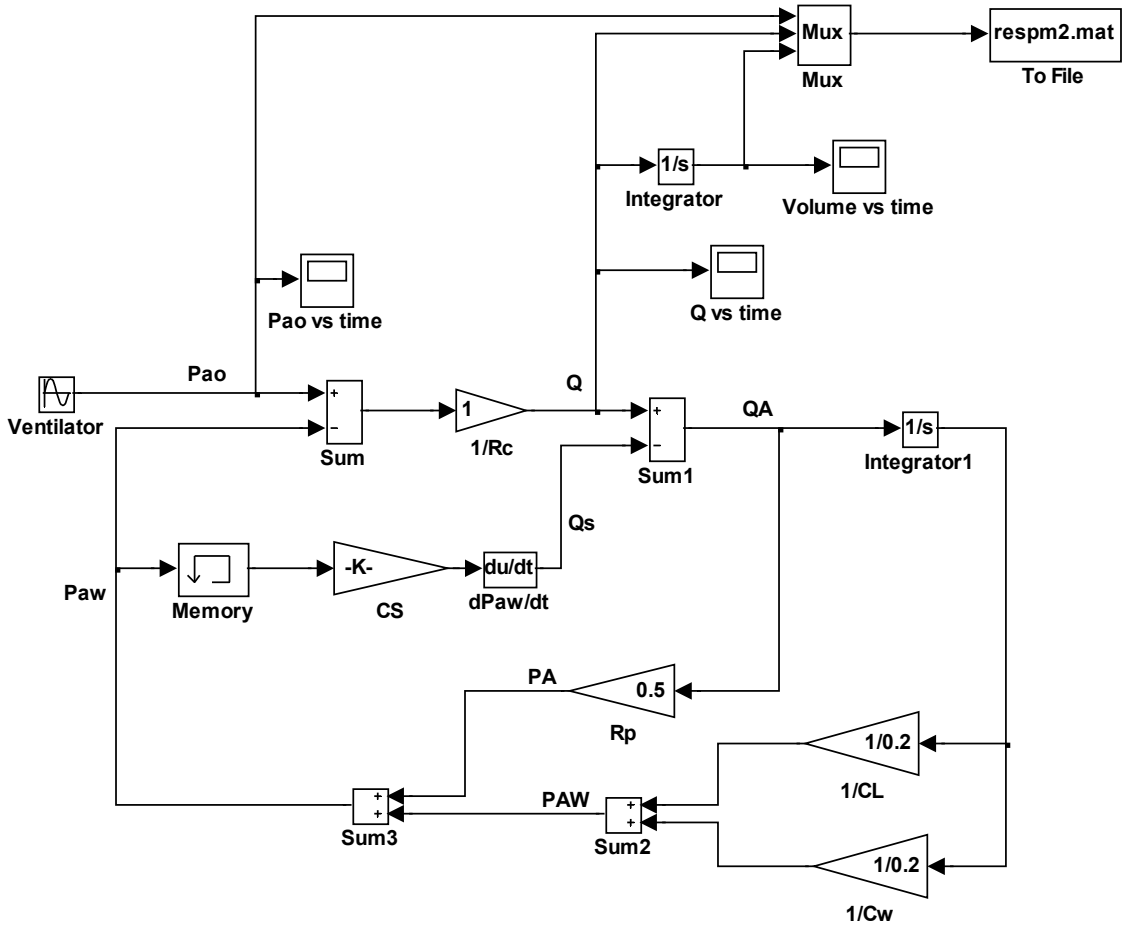


Figure 18: Simulink model implementation according to the equations (4).

In the Laplace transform of the equation (7), we get:

$$\frac{Q(s)}{PAO(s)} = \frac{s^2 + 420s}{s^2 + 620s + 4000} \tag{8}$$

This allows simplifying the Simulink model (Fig. 19):

However, when the values of the parameters change, the transform function (6) must be recalculated and the Simulink model will change.

Now we will make the model a little more complex by taking ***air inertia*** in the upper respiratory tract into account (Fig. 20).

In addition, we will now take into account the inertial element LC=0.01 cm H_2O s^2 L^{-1}:

$$LC = \frac{\Delta P}{\frac{dQ}{dt}} \tag{9}$$

where ΔP is the pressure gradient and dQ/dt is the flow acceleration, or:

$$\Delta P = LC \frac{dQ}{dt} \tag{10}$$

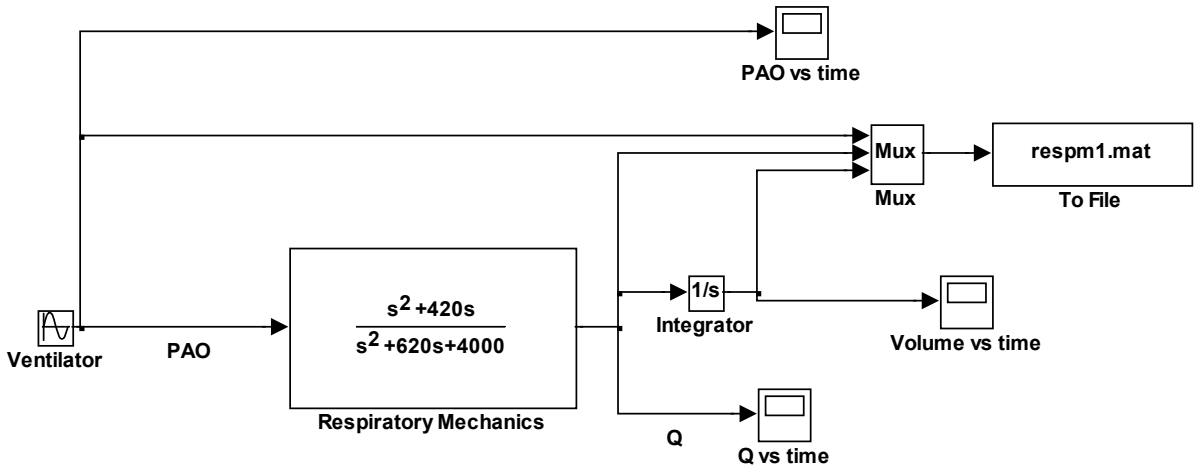


Figure 19 Simulink model implementation using the Laplace transform according to the equation (7).

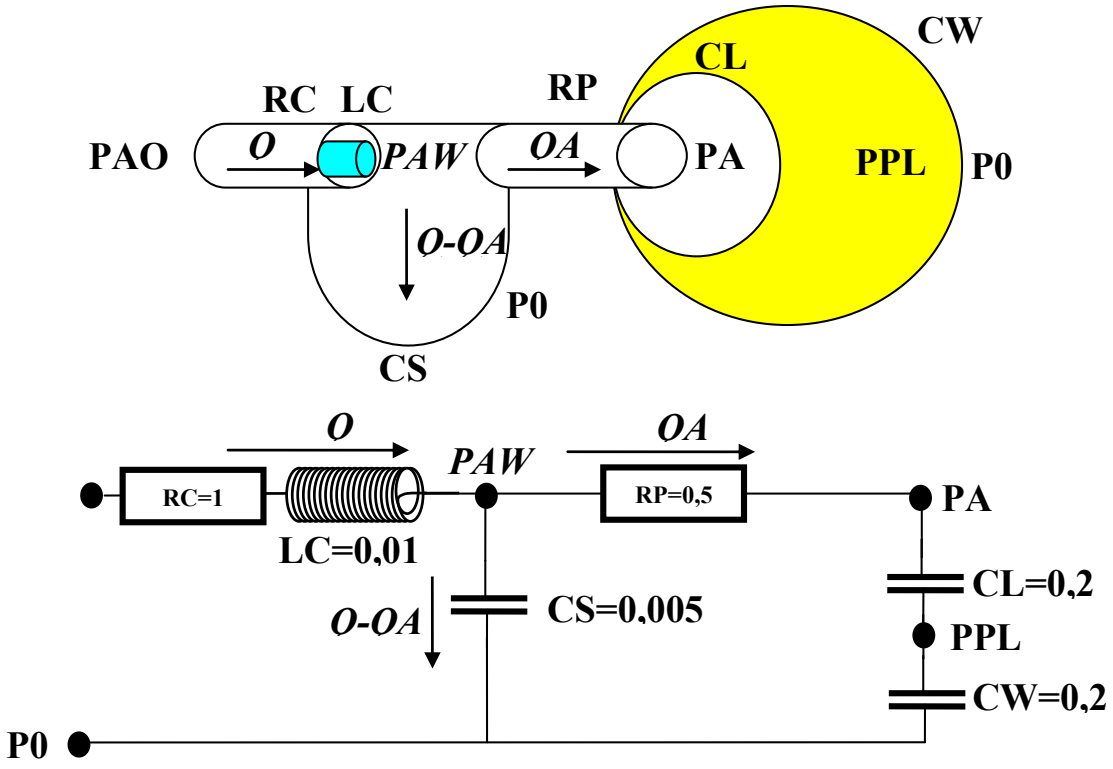


Figure 20: A simple pulmonary mechanics model taking inertia into account (hydraulic and electrical analogy).

Then we get this instead of the system of equations (4):

$$\begin{cases} RP\dfrac{dQA}{dt}+\left(\dfrac{1}{CL}+\dfrac{1}{CW}\right)QA-\dfrac{1}{CS}\left(Q-QA\right)=0 \\ RC\dfrac{dQ}{dt}+LC\dfrac{d^2Q}{dt^2}+\dfrac{1}{CS}\left(Q-QA\right)+\dfrac{dP0}{dt}-\dfrac{dPAO}{dt}=0 \end{cases} \tag{11}$$

Instead of the equation (7), we get:

$$\frac{d^2PAO}{dt^2} + 420\frac{dPAO}{dt} = 0{,}01\frac{d^3Q}{dt^3} + 5{,}2\frac{d^2Q}{dt^2} + 620\frac{dQ}{dt} + 4000Q \tag{12}$$

and in the Laplace transform, we get:

$$\frac{Q(s)}{PAO(s)} = \frac{s^2 + 420s}{0{,}01s^3 + 5{,}2s^2 + 620s + 4000} \tag{13}$$

This Simulink model will change (Fig. 21):

Since we always have to take into account the direction of the calculation in Simulink, the actual Simulink diagram is rather dissimilar to the physical reality of the described system. Even a small change in the model, such as the inclusion of the inertial element, requires careful calculation and a change in the model structure. The model will change significantly even if we consider spontaneous breathing instead of artificial pulmonary ventilation. The model input will be not the pressure *PAO* generated by the artificial pulmonary ventilation respirator but e.g. the compliance of the thoracic wall *CW* (a cyclic variation in the compliance can be used to model the function of the respiratory muscles).

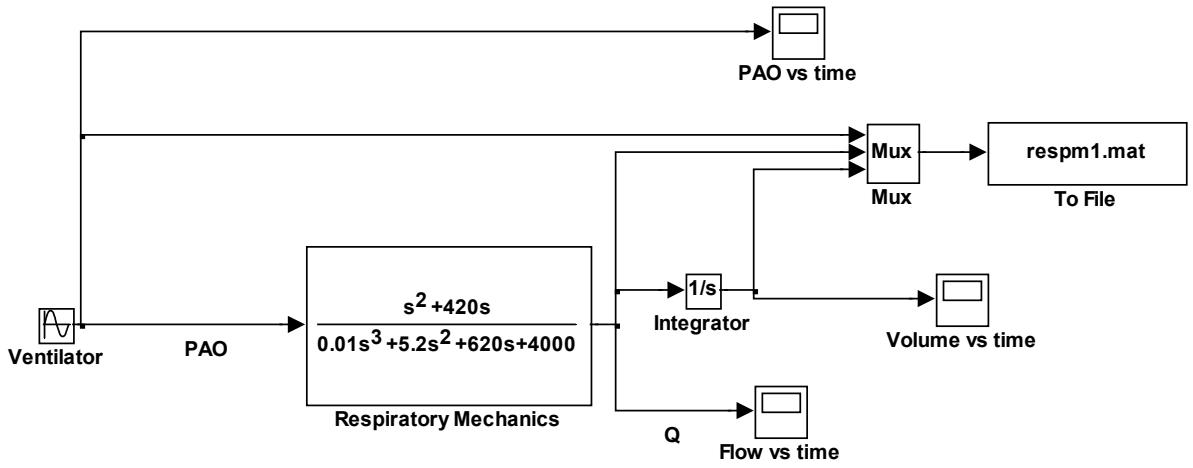


Figure 21: Simulink model implementation using the Laplace transform according to the equation (10).

Acausal approach – implementation of the pulmonary ventilation mechanics model in Modelica

Comparing the model structure in Figs. 17 and 20, formulated by means of generalized state variables, to the implementation of the model in Simulink (Figs. 18, 19, 21), we can see that the interconnected blocks in Simulink express the structure of the calculation procedure rather than the structure of the modelled reality.

In Modelica, this is different.

Acausal modelling tools, of which Modelica is a typical example, work with interconnected components that are instances of special classes in which ***equations*** are defined. When modelling in Modelica, the first task is to formally express the modelled reality by means of equations.

In our simple pulmonary mechanics model, we describe the resistances of the respiratory tract, expansible elastic bags, and we might take into account air flow inertia (see Figs. 17 and 20). The description of the air flow in the lungs belongs in the pneumatic domain. However, if we disregard the compressibility of gases, we can describe the model using the hydraulic domain. The same formal expression can be provided by an electrical analogy.

It is interesting that the individual fundamental elements have the same formal expression (Fig. 22) in different domains (electrical, hydraulic or pneumatic). This is due to the general system properties of the real world, where voltage or pressure correspond to generalized effort and electric current or medium flow correspond to generalized flow, as the case may be.

To build the pulmonary mechanics model in Modelica, we will need to define the equations of three elementary classes, whose instances we will use in the model. To express the resistance of the respiratory tract, we will use an instance of the Resistor class. The elastic respiratory tract, alveoli and chest will be described as elastic bags using an instance of the Capacitor class and the air flow inertia will be expressed using an instance of the Inductor class.

The fragment with an equation notation in the *"Resistor"* class, describing the relation between

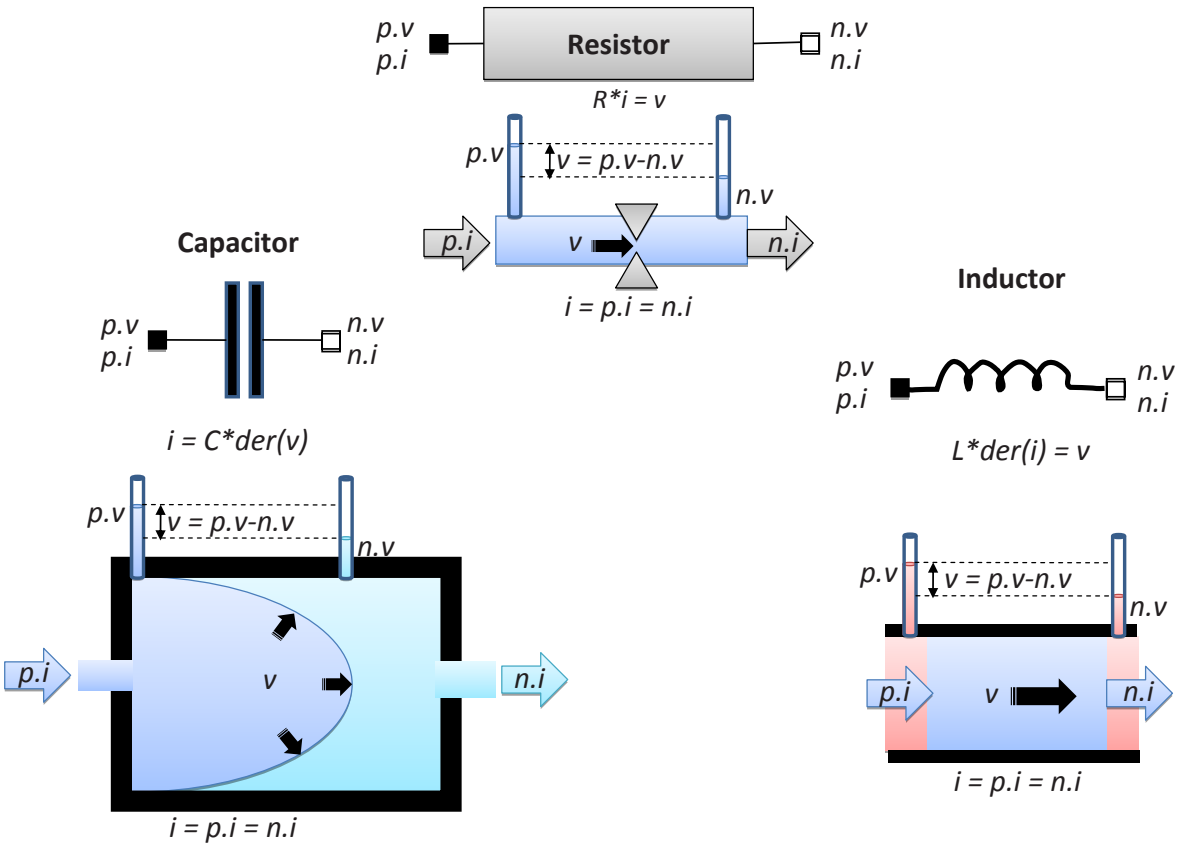


Figure 22: The hydraulic and electric elements are from different domains but have the same formal description. The analogy of voltage (v) in the hydraulic domain is pressure, the analogy of current (i) in the hydraulic domain is a stream of fluid (and a stream of gas in the pneumatic domain). Hydraulic resistance (R) follows Ohm's law in the same way as electric resistance (the voltage difference is just replaced with the pressure gradient and the current is replaced with the flow rate). The hydraulic analogy of a capacitor is an elastic bag expanded by the difference in pressures inside and outside the bag. The analogy of the electric capacity of a capacitor (C) is the compliance of the elastic bag wall. When we include inertia in a hydraulic system, the force that accelerates fluid flow is the pressure gradient. According to Newton's law, the acceleration of flow, i.e. the first derivative of flow der(i), is proportional to the pressure gradient (v) and inversely proportional to the weight of the selected fluid column, called inertance (L). In the electrical domain, inertance corresponds to coil inductance. Each element from the hydraulic or electric domain has two interconnecting connectors through which electric current or medium flow (p.i, n.i) flows in and out; as a rule, the running flow (i) never disappears in the element (i.e. i=p.i=n.i). Simultaneously, voltage or pressure (p.v, n.v) is connected to the connectors by interconnecting into a network, and a voltage gradient or a pressure gradient (v) builds up in the element.

variables expressing the resistance (*R*), pressure gradient (*v*) and flow (*i*) in Modelica, is simple, according to Ohm's law:

```
equation
  R*i = v;
end Resistor;
```

The *"Capacitor"* class is used to describe an elastic bag expanded by air flow at the input. The compliance (*C*) characterizes the level of "expansibility" of the bag wall due to the pressure difference (*v*) between the air pressure forcing air into the bag and the pressure outside the elastic bag. The flow rate of air coming to the bag (*i*) is then described by the following equation in the Modelica language (where *"der"* means derivative):

```
equation
  i = C*der(v);
end Capacitor;
```

The inertial element will be implemented in the model by means of the *"Inductor"* class. The force that accelerates air flow is the pressure gradient. According to Newton's law, the acceleration of flow, i.e. the first derivative of flow *der(i)*, is proportional to the pressure gradient (*v*) and inversely proportional to the weight of the selected gas column, called inertance (*L*). We can thus describe the relation between a change in the flow rate (*i*) and the pressure gradient (*v*) depending on inertance (*L*) using a simple equation in the *"Inductor"* class:

```
equation
  L*der(i) = v;
end Inductor;
```

Instances of the above-mentioned fundamental elements are interconnected in a network by means of connectors – two interconnecting connectors, labelled *"p"* and *"n"*, are defined for each of the elements. Voltage, or pressure for the hydraulic or pneumatic domain, is fed to each of them (*p.v*, *n.v*) when connected and an electric current or medium flow (*p.i*, *n.i*) can flow through the connectors.

Connectors are instances of special connector classes, in which the variables used for interconnection are defined. Components can be interconnected by means of connectors that are instances of the same connector classes (the "interconnection sockets" must be of the same type). In our case, connectors *"p"* and *"n"* are instances of the *"Pin"* connector class, which is able to interconnect voltages or pressures (*p.v*, *n.v*) and flows (*p.i*, *n.i*) with the environment. Values from the connectors are interconnected with the values of the variables (*i*) and (*v*) inside the individual fundamental elements.

As a rule, flow does not disappear anywhere in any of the above-mentioned fundamental elements – what flows into an element also flows from it (*i=p.i=n.i*), and the appropriate gradient is calculated from the difference in voltages or pressures (*v=p.v-n.v*).

Implementing this requirement is simple – since Modelica is an object-oriented language, all three of the above-mentioned classes of fundamental elements will have a common ancestor (*OnePor*t) from which they will inherit connectors *"p"* and *"n"* as well as the following equations:

```
equation
  v= p.v-n.v;
  0= p.i-n.i;
  i= p.i;
end OnePort;
```

The equations will thus connect the values of the pressures or voltages fed from the environment

to connectors *"p"* and *"n"* (*p.v*, *n.v*) with the pressure or voltage gradient (*v*) and express the same value of (electric or hydraulic) flow at both connectors (*p.i*, *n.i*) and inside the component (*i*).

Connector classes define the manner in which Modelica components communicate with one another. Figuratively speaking, by defining connector classes we define the types of "sockets". In connectors we define individual variables that the connector will use to interconnect a component with its environment.

It is defined for each variable in a connector whether it represents a flow (then the variable is identified with a *"flow"* attribute) or not (*"non-flow"* variables). This differentiation is important for the correct interpretation of the interconnection of individual components (instances of element classes) through the appropriate connectors (see Fig. 23). For flow variables, it is obvious that we must make sure the entity in question (whose flow the variable characterizes) neither disappears nor accumulates anywhere in the interconnection. Therefore, the sum of all interconnected variables with the "flow" attribute must be zero (as according to Kirchhoff's law in the electrical domain). For non-flow variables, an interconnection defines that their values must be the same for all interconnected connectors (according to Kirchhoff's first law). By interconnecting the instances of individual fundamental elements through connectors, we express the requirement of the zero algebraic sum of the values of interconnected flow variables and the requirement of the equality of the values of interconnected non-flow variables.

Each Modelica class can have a graphical representation – this is important especially for depicting the interconnection of instances where components are interconnected to create a clear graphical structure of a model. That is why we can also define an icon for each class in Modelica. The icon can be animated.

We can then create a model graphically in Modelica, by interconnecting the instances of individual elements that we select from a library with the mouse and setting the values of the appropriate parameters in a dialogue box.

For the implementation of our pulmonary ventilation model, we need to interconnect instances of the *"Resistor"*, *"Capacitor"* and *"Inductor"* elements.

However, we do not have to program the fundamental elements we need from the very beginning – Modelica includes extensive libraries from various physical domains (electric, hydraulic, mechanic, etc.) where such elements can be found.

In our specific case, we can take advantage of e.g. the visual components of electrical circuits for a quick solution – we will create the individual instances (*RC*, *RP*, *CL*, *CW* and *CS*), enter the appropriate values of parameters (*C* and *R*) and interconnect the components with a connector.

The result is shown in Fig. 24. Comparing the model structure implemented in Modelica with the original schematic drawing showing the model structure (Fig. 17),

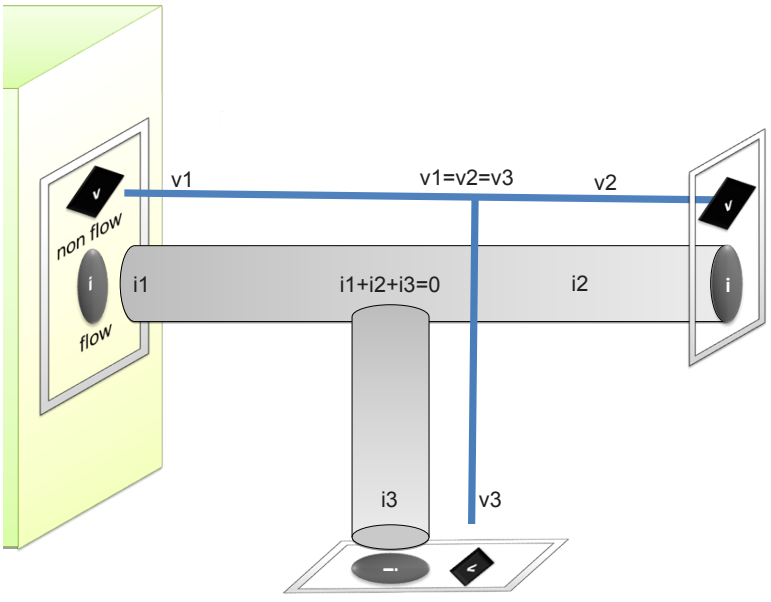


Figure 23: Interconnection of Modelica components by means of acausal connectors. The values of flow-type connector variables (here the variable i) will be set so that the algebraic sum of the values of all interconnected flows is zero. The values of other (non-flow) variables (here the value of v) will be set to the same value at all interconnected connectors.

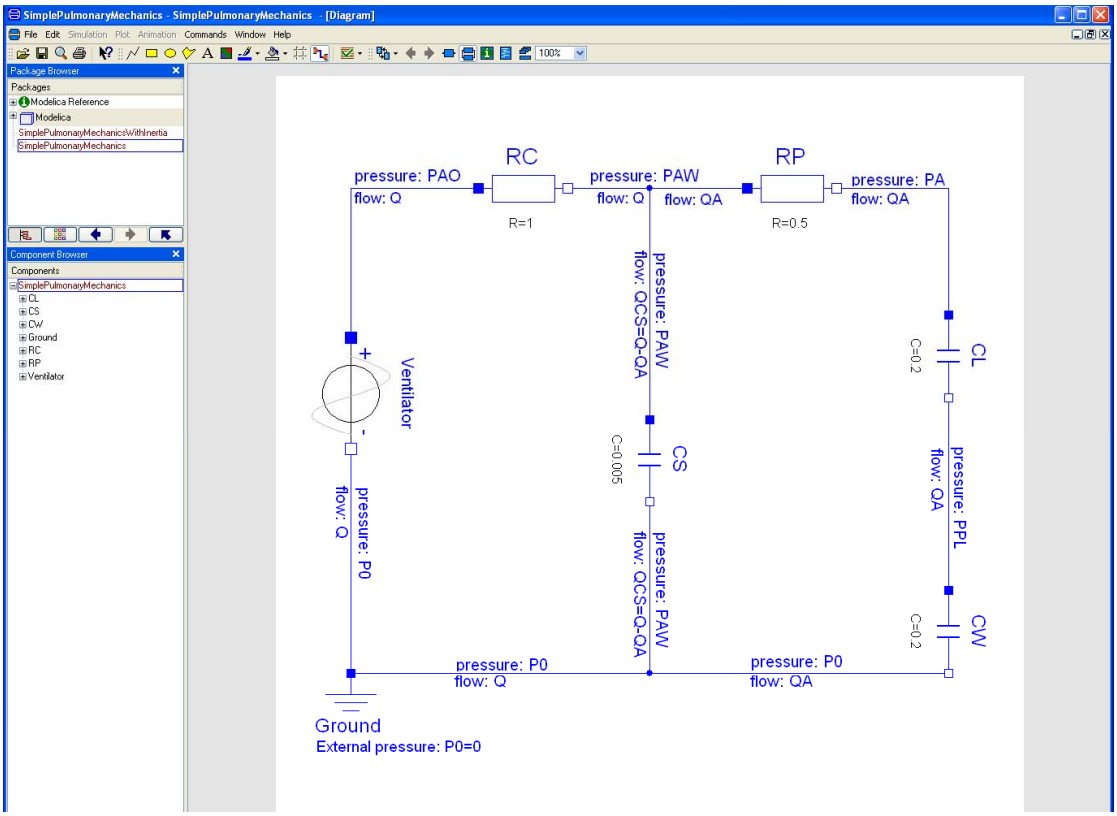


Figure 24: The implementation of the pulmonary mechanics model (according to Fig. 17) in Modelica resembles the modelled reality much more than the implementation in Simulink.

we can see that the Modelica solution is straightforward and (unlike the Simulink implementation – see Figs. 18 and 19) the structure of the model corresponds to the structure of the modelled reality.

Increasing the complexity of the model by including an inertial element does not cause any significant trouble – we just pick up the appropriate inertial component (*LC*) from the library with the mouse, set the value of its parameter (*L*) and interconnect it in the model. The structure of the model implemented in Modelica, shown in Fig. 25, corresponds to the structure of the modelled reality (see Fig. 20), while the structure of the Simulink implementation (Fig. 21) corresponds more to the method of solution for the model's equations.

The fundamental elements of the simulated reality can have very trivial notation of relations between the variables in question. A resistor, capacitor or coil from the electrical physical domain or their hydraulic analogies are illustrative examples of this.

A complex system for calculation will ensue from interconnecting the fundamental elements in networks – a system of equations will result from their mutual interconnections. Their numerical solution in causal simulation tools may not be trivial at all – e.g. more complex R-C-L models of circulation or respiration implemented in Simulink are very complex (see e.g. circulation models in our Simulink library, Physiolibrary – http://www.physiome.cz/simchips).

In Modelica, we do not have to bother with the method of solution for equations. Instead, more attention should be paid to the definition of equations in individual elements and interconnection of their instances (individual components).

In Modelica, the acausal tool itself will take care of the algorithm for solving the resulting system of equations and we can monitor the appropriate flows and pressures in various places in the simulated circuit when the simulation is launched.

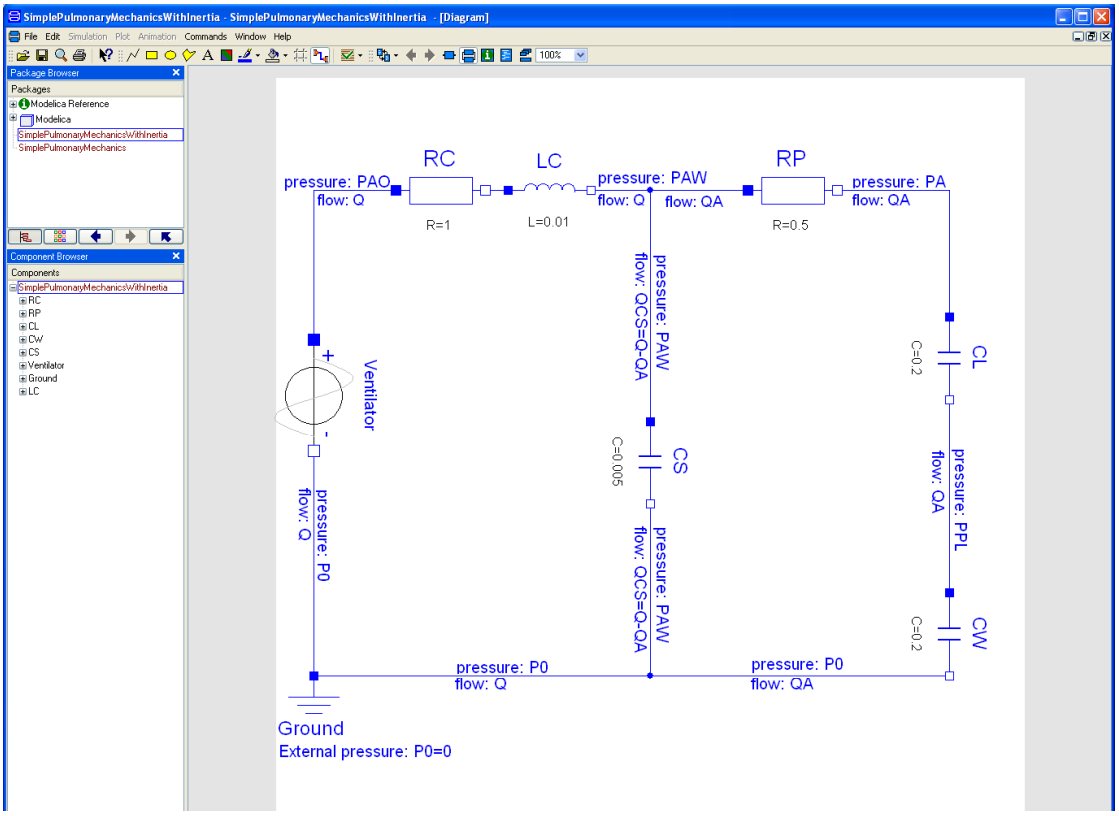


Figure 25: In Modelica, implementing the pulmonary mechanics model taking into account the inertial element (according to Fig. 20) merely requires adding the LC inertial component.

Causal and acausal connectors

The acausal connector interconnection of components is implemented by means of two types of variables: one representing a flow – for this, it holds true that the sum of flow values in all connected nodes is zero (because no medium accumulates in the area of branching into connected nodes); and one whose value remains the same in all connected nodes. It is advisable that each variable with the flow attribute is accompanied by a non-flow variable representing the generalized effort in relation to the flow variable in the connector interconnection.

Unlike Simulink components (which have defined component inputs and component outputs), we do not define what is an input and what is an output in an acausal interconnection. An acausal Modelica component does not calculate output values from input values. The interconnection of Modelica components by means of acausal connectors interconnects the equations in individual components into systems of equations.

In addition to acausal linking connectors, Modelica classes may include causal input connectors that are used to feed actual input variables from the environment, as well as causal output connectors that serve to send output variables to the environment.

In addition to equations, Modelica classes may also include a precisely defined algorithm for the calculation of output values from input values (a typical example is the modelling of functional dependencies).

Modelica components are thus interconnected using both acausal links and causal, directional inputs and outputs.

Causal connectors usually distribute signals – e.g. in a blood circulation model, signal causal inputs may contain signals used to set resistance values in components representing the resistance of the

circulatory system.

Consequently, a Modelica model is usually represented by a graphical set of components interconnected using both acausal and causal links. Components are instances of Modelica classes whose structure may also be represented as a network of interconnected instances.

An example of the definition and use of an acausal element – elastic compartment

Let us see a simple example of the definition and use of a Modelica class. When modelling the dynamics of blood vessels, we often need an elastic (inflatable) compartment.

Therefore we will define a class named *VascularElasticBloodCompartment* whose instances will be elastic, acausally interconnectable compartments that can be connected to the "distribution" of a fluid through an acausal connector – the fluid may flow to/from the compartment at a certain rate and under a certain pressure. We can assign a graphic icon to each class representing a model or connector in the programming environment. We can create an icon for our elastic compartment, too (Fig. 26).

This is not just a school example – we take this compartment into account in our Modelica implementation of an extensive model of physiological functions, "Quantitative Human Physiology" (Abram 2007, Coleman et al, 2008). Fig. 28 shows an example of the use of instances of the elastic compartment in our implementation of this extensive model.

We can imagine the elastic vascular compartment (Fig. 27) as an inflatable bag with one ***acausal interconnecting connector*** (let us name it e.g. *"ReferencePoint"*) that we will use to connect to the environment – this connector will provide us with two variables:

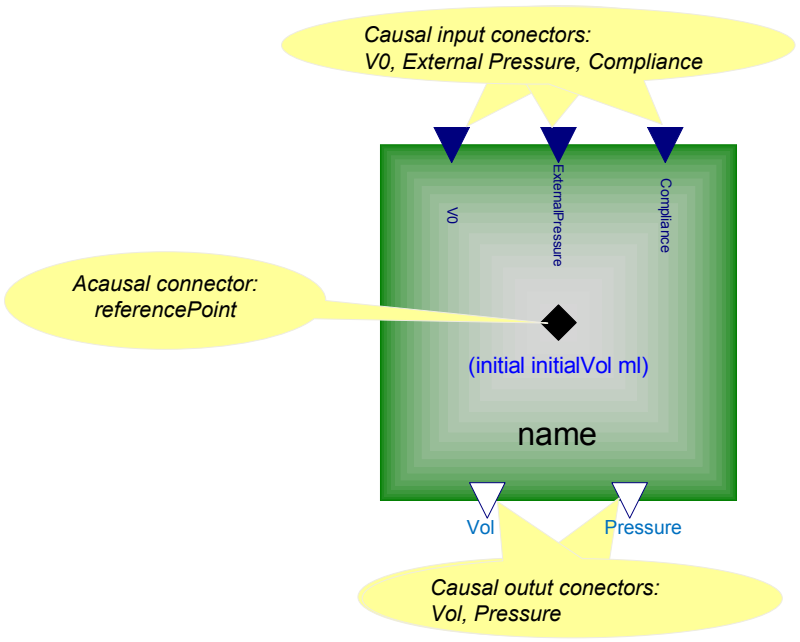


Figure 26: Modelica allows creating an icon for each created class representing a model or a connector, which will be used to interconnect instances of the class with other instances using graphic tools. The result is a model structure consisting in interconnected instances, very close to the modelled reality. Here we have created an icon for the elastic compartment, having one acausal connector (black diamond), three connectors for signal inputs and two connectors for signal outputs. Each instance of the elastic compartment will have this icon, displaying the actual value of the initial volume (specified as a parameter) instead of "initialVol" and the name of the instance instead of "name".

- flow *"ReferencePoint.q"*,
- pressure *"ReferencePoint.pressure"*.

If the connector is connected to the environment through a connector, the pressure value will truly be the same in all nodes connected to the compartment, and the flow will be distributed to all connected nodes so that its algebraic sum will be zero (nothing ever accumulates in the area of branching) – see the example of the component connection in Fig. 28.

Three signal (causal) inputs will enter the compartment from the outside:

- Basic charge "*V0*" – the value of the volume that must be reached before the pressure in the elastic compartment starts increasing. If the volume is less than zero, the pressure in the compartment will be zero.

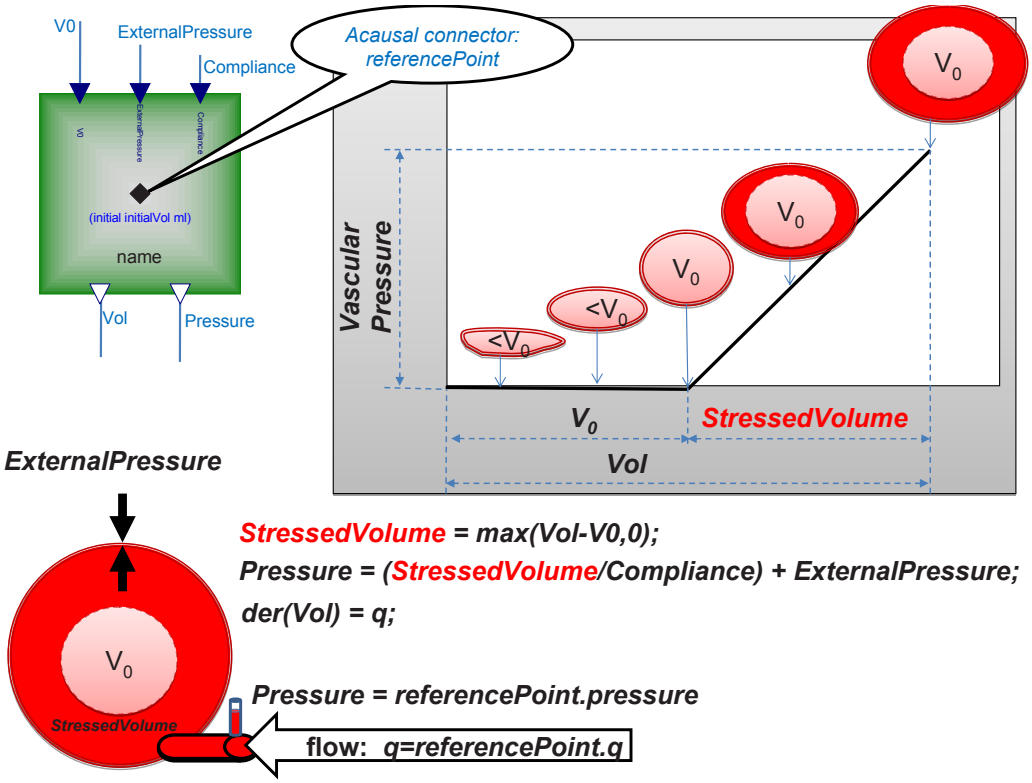


Figure 27: The concept of an elastic vascular compartment is based on the idea that when a blood vessel fills with blood, the pressure in the vessel is determined only by the external pressure on the vessel until a certain residual volume (V0) is achieved; the elastic and muscle fibres in the blood vessel will then start to tense and compress the volume of blood in the vessel with the VascularPressure pressure. If we label the volume of fluid in the blood vessel Vol, then the volume of blood stressing the vessel (StressedVolume) will determine the Pressure inside the vessel depending on its Compliance and on the external pressure on the vessel (ExternalPressure). The vascular compartment is connected to the system by means of the ReferencePoint connector, through which blood may flow into the compartment (at the rate of referencePoint.q) under pressure (referencePoint.pressure).

- Outer, external pressure "*ExternalPressure*" – the pressure of the ambient environment on the elastic compartment.
- "*Compliance*" of the elastic compartment – the pressure in the compartment will be inversely proportional to it if the compartment volume exceeds the basic charge.

Two (causal) signal outputs will go from the compartment to its environment:

- Information about the compartment's current volume, "*Vol*"
- Information about the pressure inside the compartment, "*Pressure*"

It is useful to design another parameter for the compartment (whose value will be read before the start of simulation), which would specify its initial charge:

Initial compartment volume, "*initialVol*"

We can also design an icon to display the elastic component in the programming environment.

The actual fragment of code describing the behaviour of the elastic compartment looks like this in Modelica:

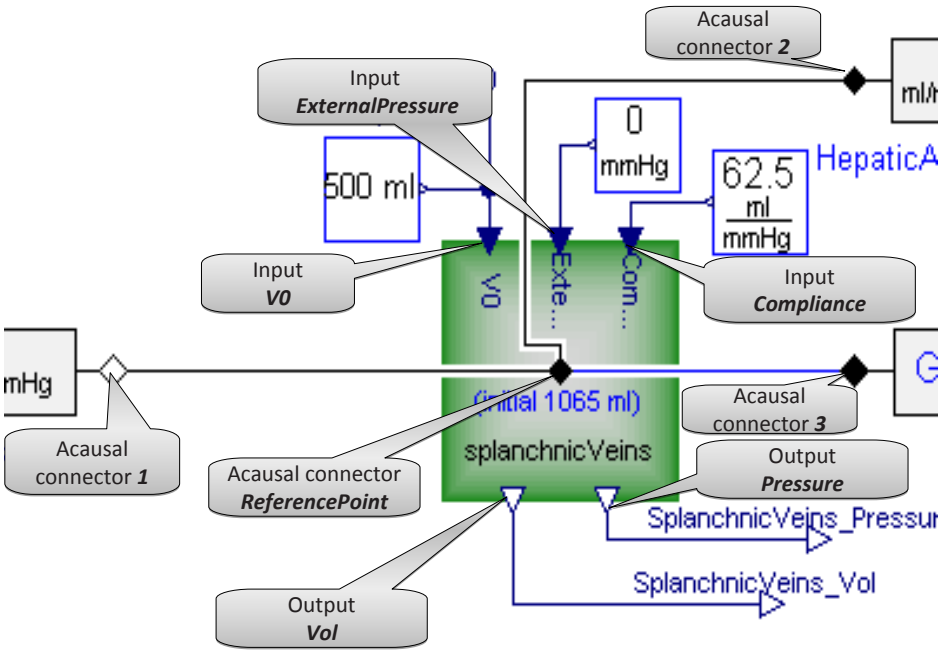


Figure 28: The instance "splanchnicVeins" of the elastic compartment "VascularElasticBloodCompartment". The acausal connection with the appropriate connectors at controllable resistors (labelled "connector1", "connector2" and "connector3" here) will interconnect the equations in the elastic compartment instance "splanchnicVeins" into a system of equations of all the interconnected elements. The pressure value will be the same at all interconnected connectors: splanchnicVeins.ReferencePoint.pressure = connector1.pressure = connector2.pressure = connector3.pressure.
The algebraic sum of all flows at the interconnected connectors must be zero: splanchnicVeins.ReferencePoint.q + connector1.q + connector2.q + connector3.q = 0.

```
model VascularElacticBloodCompartment extends QHP.Library.Interfaces.BaseModel;
  Real StressedVolume ( final quantity="Volume", final unit="ml");
  parameter Real initialVol( final quantity="Volume", final unit="ml")
                                  „initial compartment blood volume";
  ...
  initial equation
        Vol = initialVol;
  equation
        der(Vol) =  referencePoint.q;
        StressedVolume = max(Vol-V0,0);
        Pressure = (StressedVolume/Compliance) + ExternalPressure;
        referencePoint.pressure = Pressure;
end VascularElacticBloodCompartment;
```

The first line declares the model class; in addition, there is the declaration of a real variable, "*StressedVolume*", whose physical units will be checked. This is not just a question of code clarity and readability. The check of unit compatibility will enable us to avoid a very hard-to-find error, when we

exchange connectors in interconnections by mistake (if units are found to be incompatible, the check will not allow us to create the wrong interconnection at all).

Then there is the declaration of an "*InitialVol*" parameter, whose physical units will be checked as well. And then there is the equation section. The initialization of the compartment's initial volume, i.e. the variable "*Vol*", is declared first. The other lines in the equation section declare four equations. The first one is a differential equation – the derivative of the volume "*der(Vol)*" equals the inflow "*q*" from the connector "*referencePoint*".

The next equation declares that the value of the elastically stressed volume "*StressedVolume*" will be calculated as the difference between the compartment volume "*Vol*" and the value of its basic charge "*V0*" (which is an input); the equation also says that the value of the compartment volume may never drop down to negative values.

The third equation declares the relation between the "*Pressure*" in the compartment, the value of the "*StressedVolume*", the "*Compliance*" and the "*ExternalPressure*". We would like to repeat that these are not assignments but equations. The equation could also be written like this in Modelica:

Pressure - ExternalPressure= (StressedVolume/Compliance);

The last equation interconnects the value of "*Pressure*" in the compartment with the value of the pressure interconnected with its environment by the acausal connector through the "*referencePoint. pressure*".

The value of "Pressure" is also a signal output from the compartment – as a signal, it can be fed to other blocks – but it is a causal output (signal) variable and its value cannot be affected by what we connect it to. However, the situation is different with the interconnection from the acausal connector. When we interconnect an instance of the elastic compartment with other elements through the acausal connector, the four equations in the compartment will become part of the system of equations defined by the interconnection and the values of the variables in the elastic compartment instance will depend on the solution of the originated system of equations.

Hybrid models

Continuous dynamics expressed by a system of algebraic differential equations is often enough for the mathematical description of real-world models. However, we frequently need to represent discontinuous, discrete behaviour (which is often an approximation of quick continuous processes in physical systems) and continuous dynamic systems themselves are not enough for the description of real-world processes – examples include the opening and closing of valves in the hydraulic and pneumatic domain, the behaviour of diodes in the electrical domain or the switching on/off of genes, the creation and transmission of nerve impulses or the opening and closing of ion channels in the biological domain. Discrete event dynamic systems are frequent in the description of technical applications. Discrete hierarchical state automata are a very powerful tool for the formalized description of processes and their interactions (Harel, 1987).

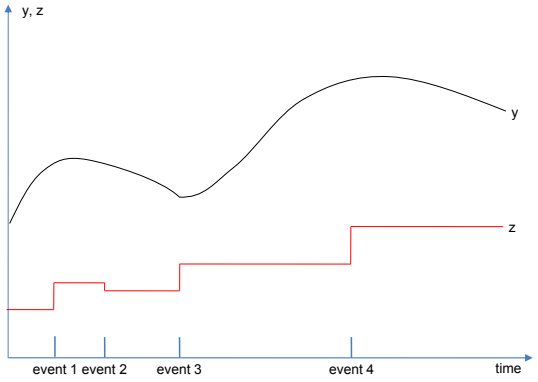


Figure 29: An example of the behaviour of real variables in hybrid systems. The continuous time variable "y" changes in time (its value does not have to be continuous – e.g. it may change discontinuously, perhaps in a jump, in response to an event). The real discrete-time variable "z" only changes its values at event instants.

When modelling large systems, it is often useful to combine discrete and continuous description to a lesser or greater extent. Such "hybrid" models can

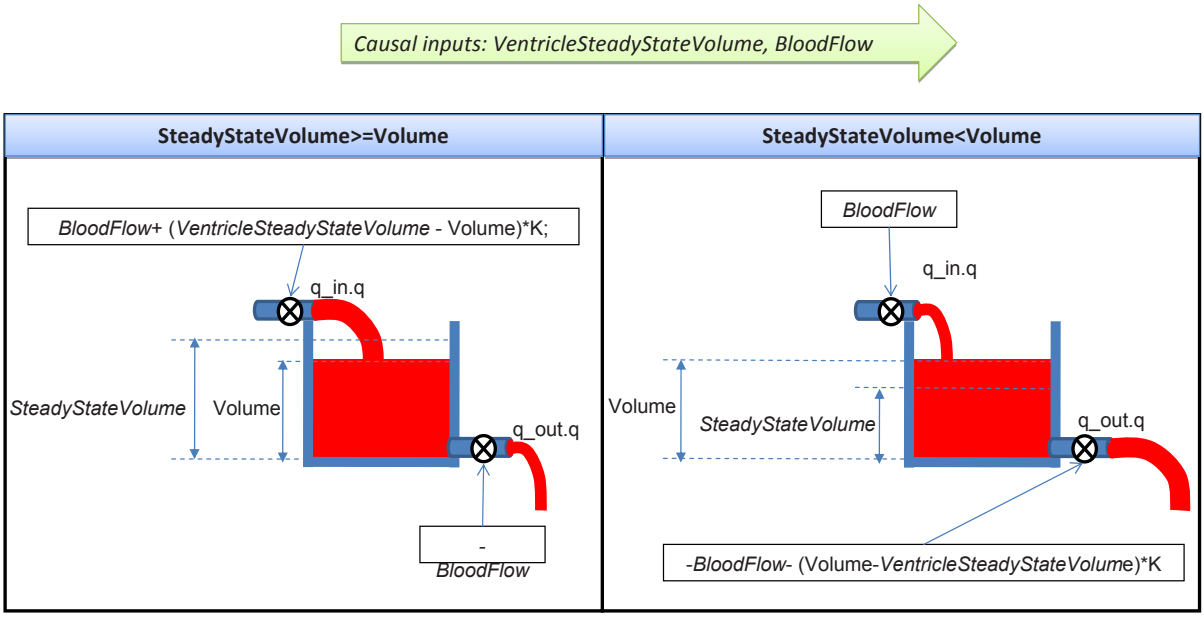
combine discrete and continuous time variables, and generate and react to various events (see Fig. 29).

Hybrid models are supported in modern development simulation environments. For example, a continuous dynamic system model in Simulink can be combined with hierarchical state automata created in a special modelling tool, Stateflow – the values of variables in Simulink can change the states of automata in Stateflow, and Stateflow can switch calculation blocks in Simulink by means of generated events, changing the calculation procedure.

However, acausal development tools can directly change the used equations (not just the method of solution). A small illustrative example can be the modelling of the average blood volume in a ventricle (see Fig. 30).

A ventricle is modelled as a continuous pump with a variable internal volume.

The ventricle model is connected to the circulation by means of acausal connectors "*q_in*" and "*q_out*". These connectors interconnect the flow of blood into ("*q_in*") and out of ("*q_out*") the ventricle. The change of blood volume in the ventricle will be determined by the algebraic sum of flows in both acausal connectors. In Modelica, this will be written as follows:




```
der(Volume)=q_in.q+q_out.q
if (SteadyStateVolume >=Volume) then
    q_in.q=BloodFlow+ (VentricleSteadyStateVolume - Volume)*K;
    q_out.q=-BloodFlow;
else
    q_in.q=BloodFlow;
    q_out.q=-(BloodFlow+ (Volume-VentricleSteadyStateVolume)*K);
end if;
```

Figure 30: The Modelica acausal modelling tool allows changing the system of equations in use dynamically. The illustration shows the hydraulic analogy of a ventricle volume model represented as a continuous pump with variable internal volume. Two acausal connectors (q_in) and (q_out) interconnect the component with its environment; the component receives the value of blood flow (BloodFlow) and the required value of blood volume in the ventricle (VentricleSteadyStateVolume) as its causal inputs. Equations calculate the volume of blood in the ventricle (Volume) and blood inflow and outflow values (q_in.q, q_out.q). The equations used will vary depending on whether or not the required volume of blood is greater than the current volume of blood in the ventricle (i.e. whether SteadyStateVolume >=Volume).

der(Volume)=q_in.q+q_out.q;

The model has two causal inputs – one is the current flow in the ventricle ("*BloodFlow*") and the other is the required volume of blood in the ventricle in the steady state ("*VentricleSteadyStateVolume*"). If that volume is greater than the current ventricle volume ("*Volume*"), then inflow to the ventricle will be set to a larger value than the outflow, proportionally to the difference between the required value and the actual value:

*q_in.q = BloodFlow + (VentricleSteadyStateVolume - Volume)*K;*

Outflow from the ventricle ("*q_out.q*") will be set to the value of "BloodFlow" with a negative sign, because it flows out of the compartment:

q_out.q=-BloodFlow;

Alternatively, when the required value of blood volume in the ventricle ("VentricleSteadyStateVolume") is less than the actual value ("*Volume*"), the inflow of blood will be set to "*BloodFlow*" and the outflow of blood will be set to a larger value than the inflow, proportionally to the difference between the actual value and the required value. The equation notation fragment in Modelica then looks like this:

```
model VentricleVolumeAndPumping;
....
equation
der(Volume)=q_in.q+q_out.q
    if (SteadyStateVolume >=Volume) then
        q_in.q=BloodFlow+(VentricleSteadyStateVolume - Volume)*K;
        q_out.q=-BloodFlow;
    else
        q_in.q=BloodFlow;
       q_out.q=
      (BloodFlow+(VolumeVentricleSteadyStateVolume)*K);
    end if;
end VentricleVolumeAndPumping;
```

Two equations are then switched over in the model's system of equations, depending on the values of the variables "*Volume*" and "*VentricleSteadyStateVolume*". At first sight, the notation looks like an assignment (as in standard programming languages) but they are equations. An equivalent notation may look like this:

```
model VentricleVolumeAndPumping;
....
equation
    delta = (VentricleSteadyStateVolume - Volume)*K;
    der(Volume) = delta;
    q_in.q + q_out.q = delta;
    if (delta<0) then
```

```
        q_in.q=BloodFlow;
    else
        q_in.q=BloodFlow+delta;
    end if;
end VentricleVolumeAndPumping;
```

Because they are equations, their order does not matter; nor does it matter whether the value of the variable "*delta*" in the third equation is on the right or on the left.

The actual notation of the equations used in Modelica is even more compact:

```
model VentricleVolumeAndPumping;
....
equation
    delta = (VentricleSteadyStateVolume - Volume)*K;
    der(Volume) = delta;
    q_in.q + q_out.q = delta;
    q_in.q = if (delta<0) then BloodFlow else BloodFlow+delta;
end VentricleVolumeAndPumping;
```

Modelica allows describing discrete and continuous systems acausally, providing many possibilities of combining models with discrete and continuous parts. Details can be found in Fritzon, 2003.

Combining acausal and causal (signal) connections in hierarchically arranged models

Modelica makes modelling large systems easier and more controllable and supports their hierarchical decomposition.

Modelica's object-oriented architecture supports the structuring of models into suitable parts having a coherent meaning so that they can be examined separately under certain conditions or re-used (whether in a different place in the same model or in another model), greatly enhancing the clarity of the created models. That is why we create large, reusable libraries of Modelica "simulation chips" in Modelica and each model is usually accompanied by an extensive, hierarchically arranged library of elements. Hierarchical components can be clicked to expand, which will reveal their internal structure.

An example of the hierarchical structure of a Modelica program is the "*VascularCompartments*" class (see Fig. 31), which implements a part of the blood circulation subsystem and makes use of an instance of the above-mentioned class "*VascularElasticBloodCompartment*". Blood flows through acausal connectors between elastic compartment instances, resistances of individual parts of the vascular system and two pumps modelling the activity of the right and left ventricles. The component also uses causal signal connections. An entire set of signal connections (coming from outside the component) is distributed e.g. by the "OrganBloodFlowSignals" bus. Input signal connections control the value of the peripheral resistors and the pumping functions of the right and left ventricles.

The structure of the model represents the structure of the modelled reality much better and much more clearly than models created in causal modelling environments. Just compare the Modelica model in Fig. 31 to the model shown in Fig. 6, implemented in Simulink. The two models represent roughly the same – the flow through an elastic vascular system and a heart pump (however, the Modelica model has more details). The Simulink model represents the calculation procedure rather than the structure of the modelled system. The advantage of acausal modelling tools is particularly evident in more complex

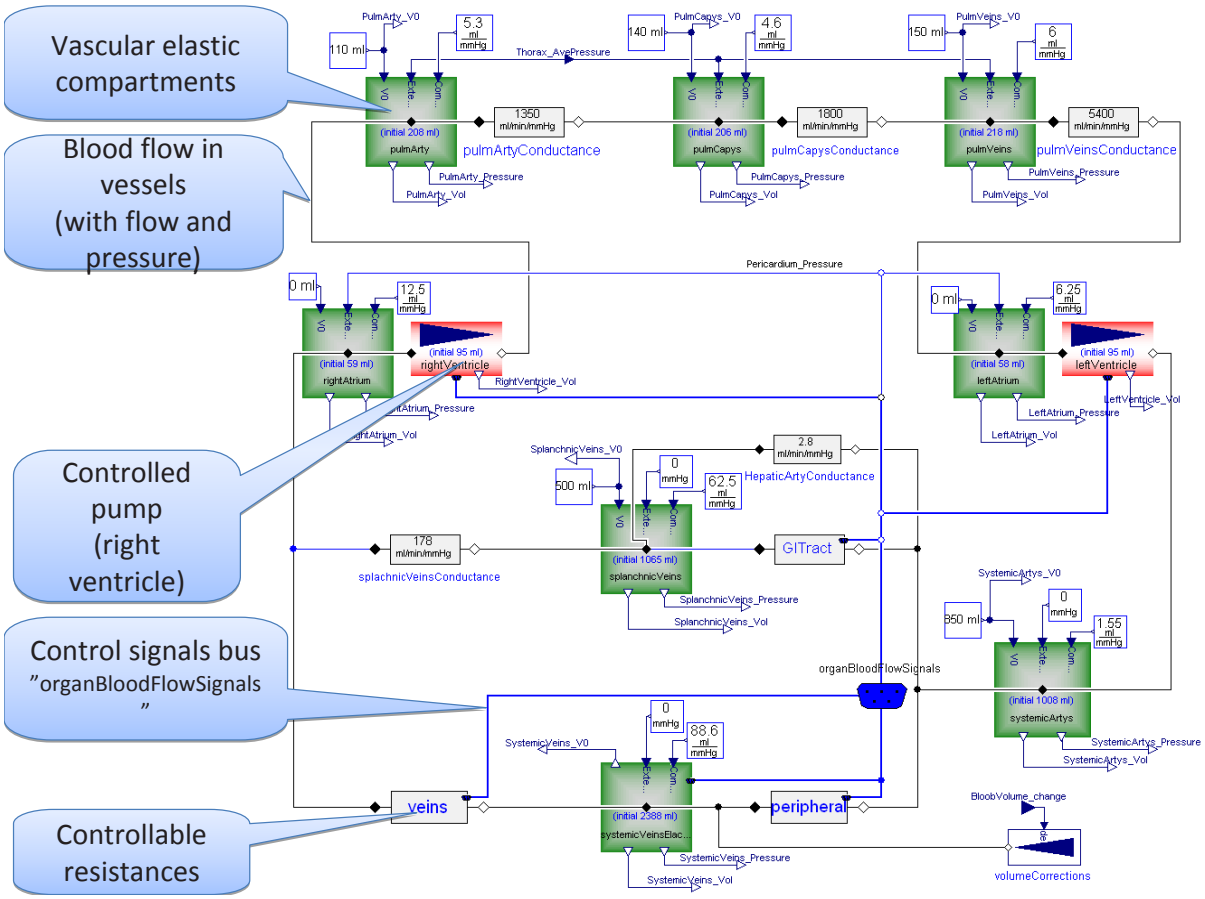


Figure 31: An example of a part of the blood circulation subsystem – an instance of the "Vascular-Compartments" class in Modelica (a part of the Modelica implementation of the large Quantitative Human Physiology model). The model combines acausal and causal (control, signal) connections. In this case, the interconnection by means of acausal connections models the distribution of blood flows and pressures among the individual interconnected components. The model is organized hierarchically; individual blocks can be clicked to expand and represent instances of classes in which equations are specified. The Modelica network thus represents the structure of the modelled system much better than networks in causal modelling tools, which rather represent the calculation procedure.

models, where the possibility of hierarchical model decomposition is crucial for success, as it is important for the interconnection of components to always express in an aggregated manner the cardinal relations at a given hierarchical level while details can be obtained by digging deeper into the structure of individual components, which will reveal the aggregated structure of the modelled reality at a lower hierarchical level.

For instance, the component representing the pump of the right ventricle is connected to the elastic compartment of the right atrium and the elastic compartment of the pulmonary arteries by means of two causal connectors (distributing the blood flow and blood pressure). Causal signal control inputs are connected to it from the "organBloodFlowSignals" bus. The "inside" of the component is shown in Fig. 32.

The heart is a pulsation pump that first draws blood from the atria into the ventricles during "diastole" – at the end of diastole, the volume of blood in the ventricle equals the end-diastolic volume (EDV). After the end of diastole, the valves between the atrium and the ventricle close and the ventricle starts contracting during "systole". The appropriate valves open and the right ventricle starts pumping

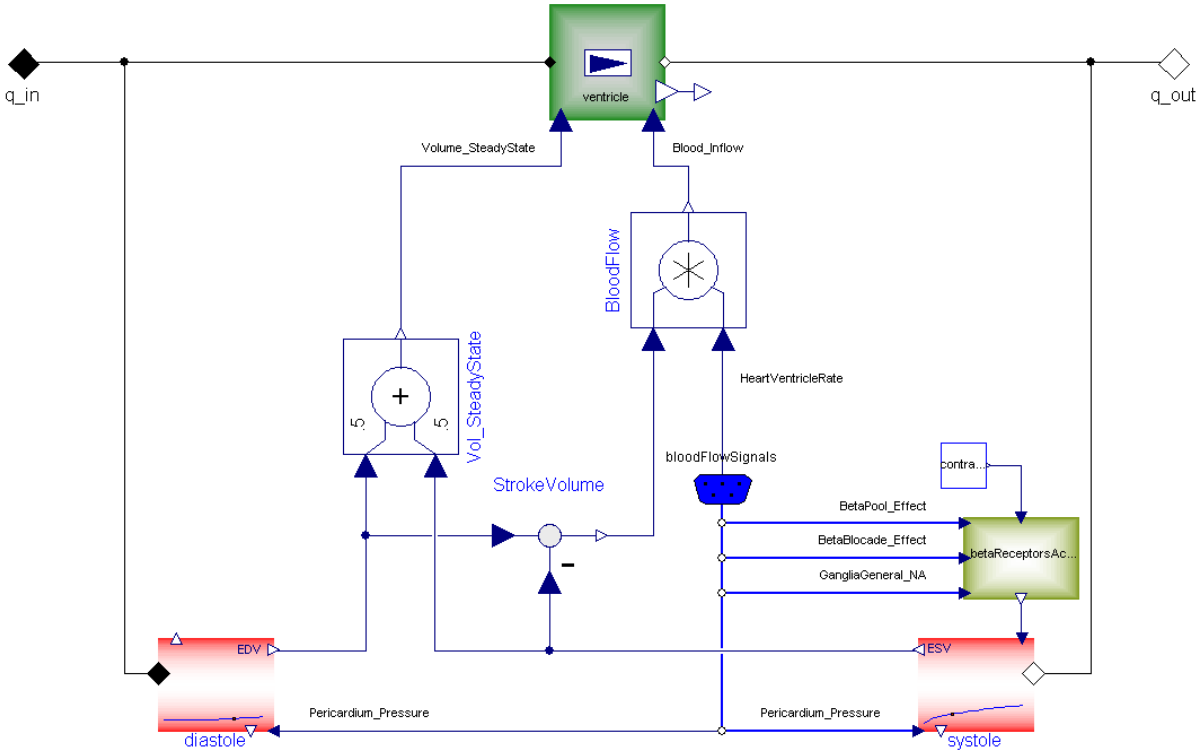


Figure 32: The "inside" of an instance of the right ventricle pump (component "rightVentricle") from Fig. 30. The ventricle is modelled as a continuous pump with variable internal volume.

blood out to the pulmonary artery (the left ventricle to the aorta). At the end of systole, the valves between the ventricle and the pulmonary artery in the right ventricle (and between the ventricle and the aorta in the left ventricle) close – the volume of blood in the ventricle at the end of systole is called end-systolic volume (*ESV*). The ventricle muscles relax, the pressure gradient between the atrium and the ventricle opens the atrioventricular valves and diastole begins again.

In the ventricle model, the end-diastolic volume (*EDV*) is calculated in a "diastole" component and the end-systolic volume in a "systole" component. Modelica allows not only designing of the graphic form of icons representing the individual components but also animating the icons (to improve clarity). In the given example, both components have animated curves during simulation, which represent the relation between pressure in the ventricle and the values of *ESV* and *EDV*. A dot on the curves represents the current value of *EDV* and current value of *ESV*. Blood pressure in the ventricle is derived from the value of the supply pressure in the atrium (this gets to the component from *"q_in"* by means of an acausal connector) and the value of the external pressure in the pericardium – this gets to the ventricle from the signal bus *"BloodFlowSignals"* by means of a causal connector, *"Pericardium_Pressure"*. In the "systole" component, blood pressure in the right atrium at the end of systole is derived from the value of the counter-pressure in the pulmonary artery (or pressure in the aorta in the left ventricle) – by means of an acausal connector, *"q_out"*, and the value of the external pressure in the pericardium (by means of the causal connector *"Pericardium_Pressure"*). During systole, the dependency of the *ESV* value on the end-systolic pressure is also affected by the stimulation (or blocking) of *"beta receptors"*, which results in changes in the contractile power of the heart muscle. A detailed description of equations that describe this dependency is contained in *"BetaReceptorsActivityFactor"*, a component whose output is the causal input for the *"systole"* component.

The ventricle model in Fig. 32 is not expressed as a pulsation pump but rather as a continuous

pump with variable internal volume. We do not model pumping "beat by beat" but by the average cardiac output per minute.

The systolic volume is calculated first (in the component *"StrokeVolume"*), as the difference between the end-diastolic (*EDV*) and end-systolic (*ESV*) volumes. The value of the blood flow per minute is calculated (in the *"BloodFlow"* multiplier) from the systolic volume multiplied by the heart rate. The value of the heart rate (*"HeartVentricleRate"*) comes from the outside, from the "bloodFlowSignals" bus.

The average volume of blood in the ventricle is estimated as the arithmetic mean (*"Vol_SteadyState"*) of the maximum heart charge in diastole (*EDV*) and heart volume at the end of systole (*ESV*). The ventricle is represented (by the *"ventricle"* component) as a continuous pump that has variable internal volume (the component is an instance of the model from Fig. 30). The pump is connected to the blood circulation by means of acausal connectors (*"q_in"* and *"q_out"*). It receives the calculated value of the cardiac output (Blood_inflow) and the required average value of the pump's internal volume ("Volume_SteadyState") by means of two causal connectors.

The model of the heart approximated as a continuous pump is sufficient (and sufficiently quick) for a number of applications in medical simulators. However, if we wish to model e.g. various valve defects, we have to use a more detailed model, describing the behaviour of the ventricle beat by beat (Fig. 33).

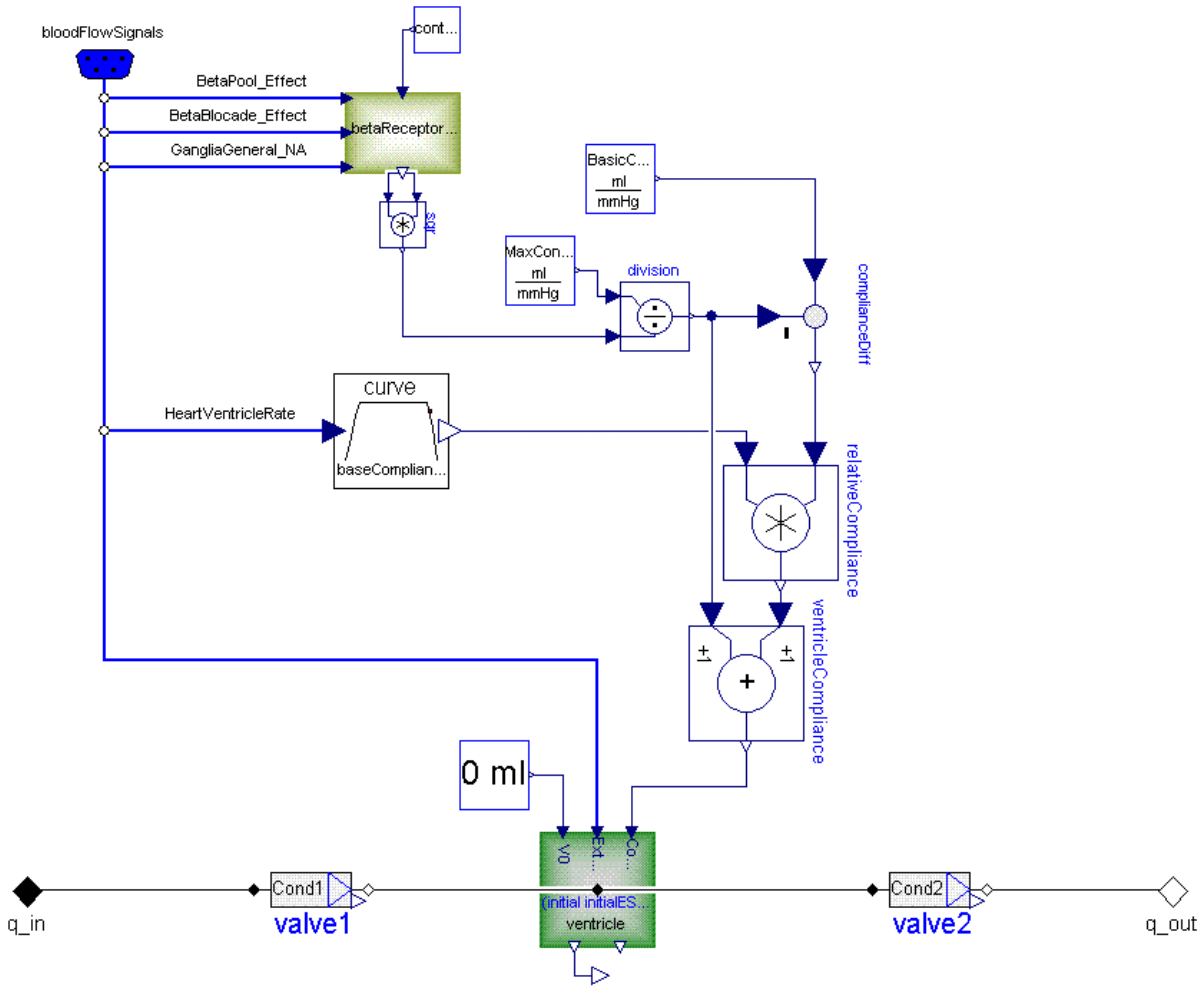


Figure 33: A ventricle model with valves, which generates a pulsating blood flow beat by beat. It has the same outer interface for interconnecting into the model of a higher hierarchical level as the pump model.

Replacing a simpler component with a more complex component does not have to mean reworking the entire model. Model notation in Modelica allows a very elegant exchange of components as different variants of classes with the same interface.

For example, it is possible to exchange the instances of the left and right ventricle models ("rightVentricle" and "leftVentricle" components) inside the blood circulation subsystem model (see Fig. 31): Instead of the continuous pump model of the ventricles (Fig. 32), we can insert instances of a more complex model into the diagram, generating blood flow beat by beat. We just have to cast the instances of the left and right ventricles.

The basis of the ventricle model with valves that generates a pulsating blood flow beat by beat (Fig. 33) is an elastic compartment ("ventricle"), which has a generated oscillating value of compliance (unlike the elastic compartment used in the blood vessels). The frequency of the oscillations is determined by the number of heartbeats per minute. The shape of a single compliance change period (the "curve" component) expresses the properties of the heart muscle. The amplitude is affected by the stimulation and blocking of the beta receptors. At last, the direction and rate of blood flow in the ventricle is derived automatically from the properties of the valve components ("valve1" and "valve2" components) and from the pressure gradients.

A simple valve model can be represented as an analogy of a series connection of an ideal diode with a resistor. An alternative (more complex) model of the valves will allow the modelling of various

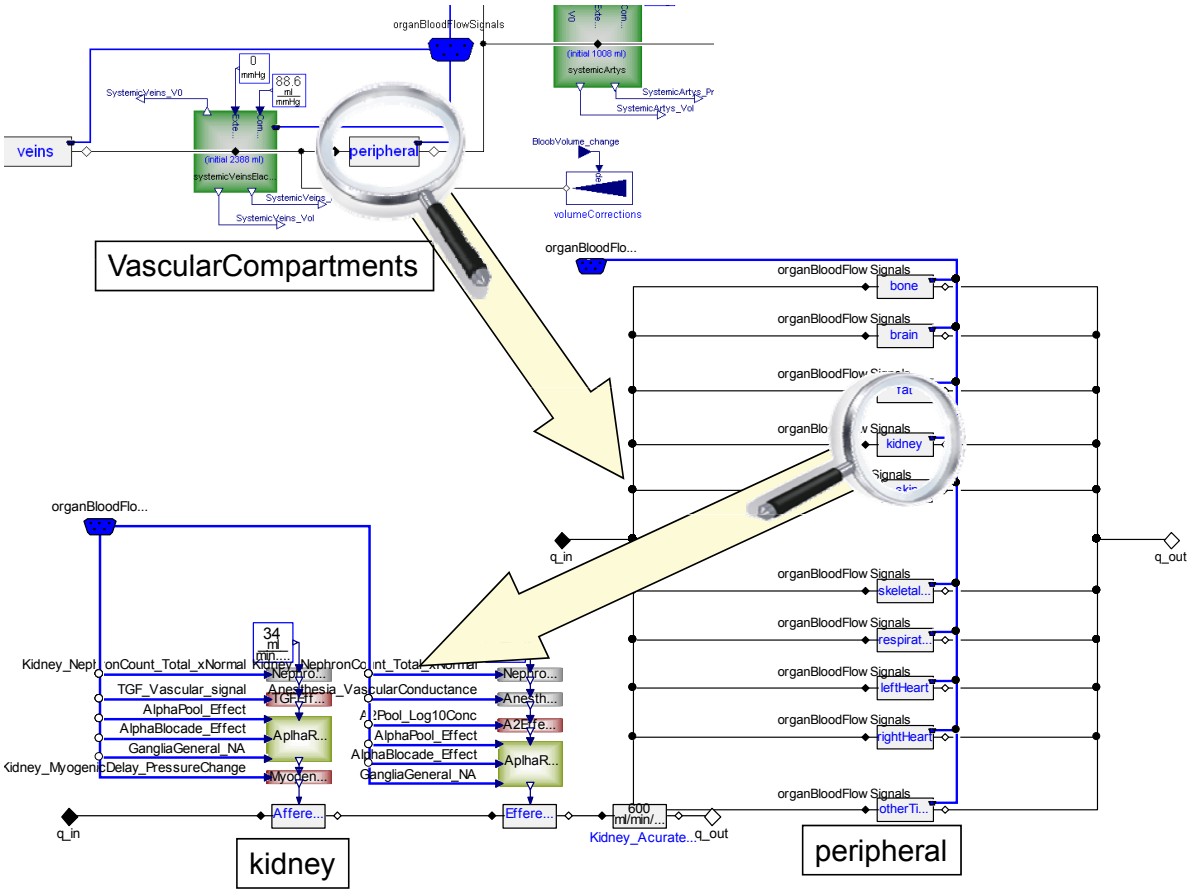


Figure 34: Hierarchical arrangement of models in Modelica. The "VascularCompartments" component (from Fig. 30) has one peripherally controlled resistor named "peripheral". When clicked, it will expand and show a number of controlled resistors connected in parallel. Clicking one of them – named "kidney" – will display complexly controlled resistors in the kidneys. The combination of acausal and (causal) signal connections and the wide range of graphical options for displaying the modelled relations allow the creation of hierarchically structured and "self-documenting" models.

valve defects.

By exchanging components of different complexity with the same interface, we can create model instances of different complexity as needed for their application use.

Modelica supports the possibility to exchange individual components by allowing defining interfaces with a variable number of control (input, causal) signals. Depending on the number of control signals, components may be more complex or, conversely, simplified, and their function may be tested when connected to a model of a higher hierarchical level. This major advantage can be used efficiently not only when debugging complex models but also when identifying a model from experimental data.

Making use of the hierarchy and component structure of models is very important in Modelica (see Fig. 34). For the model construction architecture, it is advisable to follow the rule stating that the structure of a component should always fit in a single screen. A complex tangle of connections is not the sign of a good design and calls for trouble.

The purpose of this chapter was not to describe the physiology of blood circulation. We just wanted to use the rather detailed description of the structure of some components to illustrate how acausal modelling tools allow the creation of richly hierarchically structured, easily modifiable, "self-documenting" models.

When modelling extensive systems, such as the models of interconnected physiological regulations as a basis for medical simulators, the acausal modelling environment of the Modelica language is a great help.

From Simulink to Modelica

It is simply amazing how fast the new Modelica simulation language adopted various commercial development environments. Only recently, just two commercial implementations of this language existed (D*ymola* from Dynasim and *MathModelica* from Mathcore), today (February 2010), the Modelica language is already used by simulation environments *LMS Imagine.Lab AmeSim* from the LMS company (http://www.lmsintl.com), *MapleSim* from Maplesoft (http://www.maplesoft.com/), *Mosilab* from the Frauenhofer company (http://www.fraunhofer.de) and *SimulationX* from ITI (http://www.iti.de).

Modelica is being used more and more in industrial applications. This modern simulation language is used by large corporations such as Siemens, ABB and EDF. Well-know automakers, such as AUDI, BMW, Daimler, Ford, Toyota, VW, use Modelica to design economic cars and air-conditioning units. The advancement of the development environments and technologies that use the Modelica language and the development of relevant application libraries is a part of the European research projects EUROSYSLIB, MODELISAR and OPENPROD financed by a total sum of 54 million € (see **http://www.modelica.org/**).

However, Modelica is still not used as much in biomedical applications.

The vast majority of biomedical simulation applications are still done in casual, block-oriented environments. These include referencing database development environments for biomedical models (such as the JSIM language http://physiome.org/model/doku.php or CEllML language http://www.cellml.org/).

A frequently used environment in biology and medicine is Matlab/Simulink – monographs dedicated to biomedicine models are usually equipped with additional software used in this environment, but so far without the use of new acasual or non-casual Simulink libraries, such as Wallish et al., 2008; Logan, Wolesensky, 2009; Oomnes et al., 2009.

However, already in 2006, Cellier and Nebot pointed out the benefits of Modelica, when used for clear implementation of physiological systems descriptions and interpretations. The classic McLeod's circulation system model was implemented by PHYSBE (PHYSiological Simulation Benchmark Ex-

periment) (McLeod, 1966; McLeod, 1967; McLeod, 1970). The difference is clearly seen, if we compare the Cellier model implementation (Cellier and Nebot, 2006) with the freely downloadable version of the PHYSBE model implementation in Simulink http://www.mathworks.com/products/demos/simulink/physbe/.

Haas and Burnhan, in their recently published monograph, pointed out the benefits and large potential of the Modelica language used for modeling medically adaptive regulatory systems (Haas, Burnham, 2008). The recent, Brudgád et al.. publication (2009) talks about work on the implementation of the SBLM –http://sbml.org/), in the Modelica language. This would enable us in the future, to simply run models, whose structure is described in the SBLM language, on development platforms, based on the Modelica language.

However, acasual models may be created in Simulink today as well, by using new acasual libraries (Simscape and others).

We have been using Matlab and Simulink for years to create and develop models of physiological systems (Kofránek et al., 2001, 2002, 2007) and have also been developing the relevant application Simulink library – the Physiolibrary http://physiome.cz/simchips). We have also developed the relevant software tools that simplify the transfer of models implemented in Simulink over to development environments (ControlWeb and Microsoft .NET), where we create our own tutorial and education simulators (Kofránek et al., 2008). Our development team gained priceless experience in previous years working with the Matlab/Simulink development environment made by the renown company MathWorks. On the other hand, we were also attracted by the new development environments using the Modelica language.

We were facing a decision whether to continue with the development process of physiological system models in Simulink (using new acasual libraries), or to make a radical decision and switch to the new Modelica language platform.

Our decision was affected by our efforts to implement a large model made by Guyton‘s coworkers and students (Hester et al., 2008). Their ***Quantitative Human Physiology (QHP)*** model is an extension of a tutorial simulator called the ***Quantitative Circulatory Physiology (QCP)*** (Abram et al., 2007).

The QHP model contains more than 4,000 variables and at the present time, it probably represents the largest and most extensive model of physiological regulations. It enables the user to simulate a wide range of pathological stages and statuses, including the effects of the relevant applied therapy.

Compared with the previous QCP simulator, whose mathematical background is hidden from the user in its source code written in C++, the QHP simulator uses a different approach. The QHP authors decided to separate the simulator implementation and description of the model quotations, in order to make the structure of the model more clear and apparent for the larger scientific community.

In 1985 the architect of this model, Thomas Coleman, had already created a special language used to write the model structure, as well as the element definitions into the simulator user interface. The language is based on modified XML notation. Model is then written by using XML files. A special convertor/decoder (DESolver) converts XML files into executable simulator code.

A detailed description of this language and DESolver converter, as well as the relevant educational tutorial, is freely accessible on the web page of the University of Mississippi (Fig. 35). The new QHP model is written in the XML language as well. Its structure with all details may be found at (http://physiology.umc.edu/themodelingworkshop), published as an open source.

Therefore, the user can modify this model as he wishes. However, the model description has been divided into more than 2,833 XML files in 772 directories, from which the special solver creates and executes the simulator (Fig. 36).

The entire structure of the model and following links and references are not easily identifiable. That is why the international research and development team in its SAPHIR project (System Approach

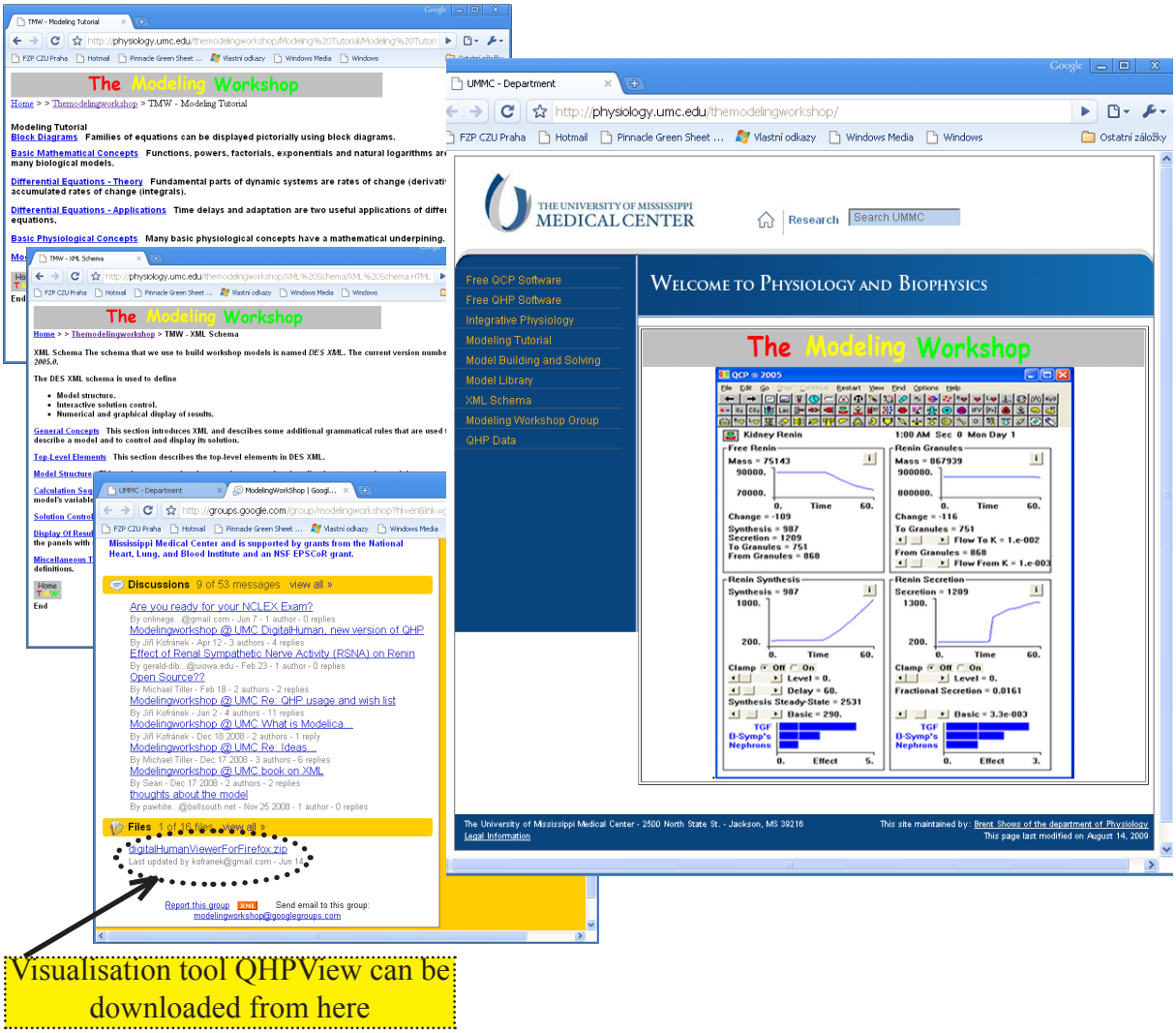


Figure 35: QHP language used to describe the simulator, the Quantitative Human Physiology (QHP/ Digital Human) is described in details on the webpage of the University of Mississippi Medical Center. Various smaller models in this language are available, as well as own source text and relevant QHP simulator translator. A registered discussion group is also available where information and experiences are shared. You can also download our tool (QHPView) here which is used to view mathematical relations in the model. These relations are scattered around and embedded in the source code, containing thousands of files.

for Physiological Integration of Renal, cardiac and respiratory control) decided to use the old Guyton models from 1972 (Guyton, Coleman, Grander, 1972) and the Ikeda model from 1979 (Ikeda, Marumo, Shirsataka, 1979) for the creation of its new and extensive model of physiological functions instead of the freely available QHP model. The source codes of the QHP model appeared unclear or hard-to-understand to those involved in this project (Thomas et al., 2008).

The extensive QHP model is still at the testing, modification and expansion stage. ***We have been able to agree on a long-term cooperation with*** the main architect of this simulator, Thomas Coleman, as well as with other co-authors from the University of Mississippi, focusing on the future development of this model.

We have been able to create a ***special software tool called QHPView*** (Fig. 37), which is able to create a clear and legible overview of mathematical relations and connections from thousands of source codes.

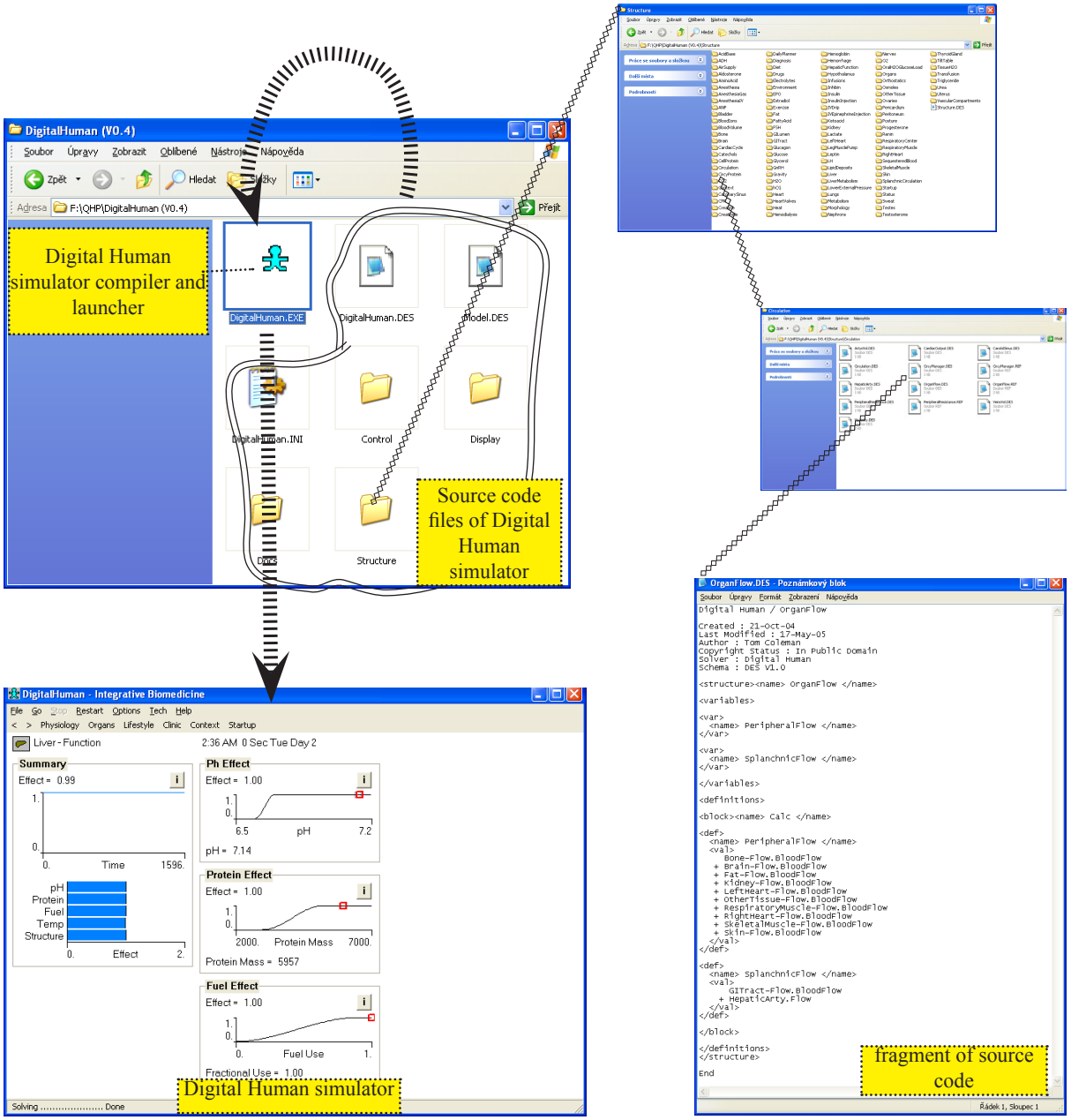


Figure 36: All necessary files of the QHP tutorial simulator (called the Digital Human by the authors in the last version). This simulator has been designed for the Windows operating system and does not require special installation. Only zip files must be unzipped into a selected folder. After you click the DigitalHuman.exe icon, the translator translates the source text embedded within thousands of directories and more than two thousand files and initiate its own simulator. Even though the source text of the simulator and the entire mathematical model on the background is offered as an open source (and in theory, the user may modify the model), the navigation through thousands of mathematical relations and viewing thousands of XML and interconnected files is rather difficult.

We are offering this tool as an open source on the QHP web page at (http://physiology.umc.edu/themodelingworkshop/).

First, we tried to implement the QHP model in the Simulink environment.

The model contains a wide range of relations that offer solutions for implicit quotations. That is why the implementation of this block-oriented model (outputs from one block are used as inputs for the

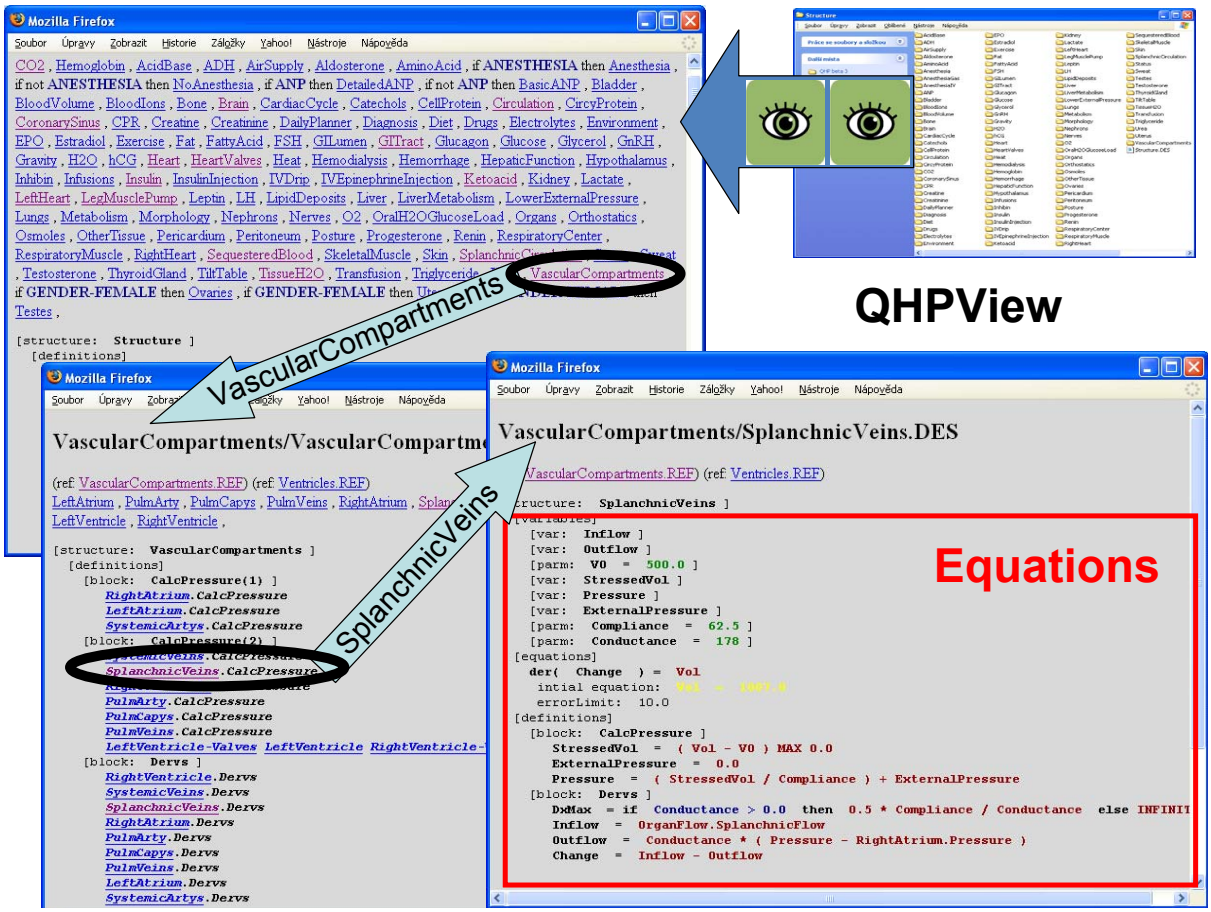


Figure 37: Visualization tool QHPView, created by us, simplifies viewing of the QHP/Digital Human simulator structure, containing more than two thousand XML files, scattered in thousands of directories, where quotations and links between them may not be apparent.

next blocks) is very difficult and as the implementation got more and more complex, the transparency of this model went down quickly. The use of new acasual libraries in this complex model proved to be problematic and the transparency of the model improved only a little bit.

Therefore, we decided to stop using the Simulink implementation and began to implement the Modelica language using the Dymola environment (http://www.3ds.com/products/catia/portfolio/dymola).

Very quickly we discovered that the ***implementation of a large and extensive model in Modelica is much more effective than using acasual libraries in Simulink.***

When we compared the Simulink and Modelica implementations we also discovered a significant difference. Mainly due to the fact that the new acasual libraries are only acasual superstructure of Simulink and not an objectively oriented modeling language based on quotation, as the Modelica language is.

Therefore, if we compare the development environments based on the simulation language Modelica with the Matlab/Simulink development environments made by Mathworks, we may say the following:

- contrary to Simulink, the model implemented in Modelica much better reflects the essentials and base of the modeled reality and the ***simulation modes are more clear, readable and less prone to errors;***

- ***the object architecture in*** Modelica enables the user to ***build and tweak models with an hierarchical structure gradually, while using reusable element libraries;***
- contrary to Simulink (which is the industrial standard for Mathworks), Modelica is a ***normalized programming language*** and therefore, it may contain ***various commercial and non-commercial developing environments competing between each other. This language is used*** for specific problem solutions originating in various application fields ***(for commercial and non-commercial specialized libraries);***
- In Modelica it is possible to ***combine casual (mostly signals) and acasual links non-invasively***; and unlike in Simulink, it is also possible, (within interconnected blocks) to create algebraic loops fairly easily – the assembler in Modelica contains symbolic manipulations on the background ***and therefore the disconnection of algebraic loops is the task for the development environment and not for the programmer.***

The above specified reasons led us to use, as the ***main implementation tool for the model creation, the Modelica*** language and we also gradually stopped using the Matlab/Simulink environment, (Kofránek, Mateják, Privitzer, 2008).

As far as the creation of application libraries and tools used in the Modelica development environments is concerned, we are involved in international cooperation.

The mutual efforts of 12 companies and 9 universities grouped together in the so-called Open Modelica Source Consortium, contributed to the development of the Open Modelica environment, distributable as an open source at (Open Modelica Source Consortium – see http://www.ida.liu.se/labs/pelab/modelica/OpenSourceModelicaConsortium.html). Our development team cooperates with Creative Connections s.r.o., which is a member of this consortium (see http://www.creativeconnections.cz/). We are currently developing a tool which would enable us to generate a source code/text in the C# language from a model created and debugged in Modelica.

QHP in a modeling coat

The implementation of the QHP model clearly shows the benefits of the model creation process when done in the Modelica language.

If we compare the complex structure of the QHP model by using the visualization option in QHPView (Fig. 37) with examples of implementations done in the simulation language Modelica, shown in previous pictures 26-34, we can see that the acasual implementation done in Modelica creates a transparent and legible model structure and therefore offers easier model modifications.

The QHP model implemented in Modelica is being currently ***modified and extended.***

Modifications and extensions of QHP were partially taken from our original model Golem (Kofránek et al., 2001) and further modified according to newest findings and experiences.

Our modifications are mainly extensions, which improve the usability of the model during the modeling of difficult breakdowns in acidobasic (acid-based), ionic, volume and osmotic homeostasis of inner environments, which is very important for urgent medicinal statuses.

Our modification of the QHP model is based mainly on the ***process of re-programming the acidobasicsubsystem balance***, which is based in the original QHP on the so-called Stewart acidobasic balance theory. Simply put, the so-called „modern approach“ of Stewart (Stewart, 1983) and his followers (e.g. Sirker et al., 2001; Fencl et al., 2000) explaining breakdowns in the acidobasic balance, uses mathematical relations calculating the concentration of hydrogen ions $[H^+]$ from partial pressure CO_2 in plasma (pCO_2), from total concentration (Buf_{tot}) of weak (partially dissociated) acids and from the difference between the concentration of fully dissociated cations and fully dissociated anions (SID - strong ion difference):

$[H^+]=Function\ (pCO_2,\ SID,\ Buf_{tot})$

The problem of this approach is that the precision of acidobasic calculations in the model depends on the precision of the *SID* calculation, that is the difference between the concentration of fully dissociated cations (that is mainly sodium and potassium) and fully dissociated anions (mostly chlorides). Imprecision that is created during the modeling of sodium, potassium and chlorides intake and excretion are transferred and reflected by the imprecision in the modeling process of the acidobasic status.

Even though Coleman et al. (2008), significantly improved the modeling of reception and excretion of sodium, potassium and chlorides in kidneys in his QHP model, if we model a long-term status (when nothing is happening with the virtual patient), the virtual patient (in the current model version) has a tendency to fall into slight and steady metabolic acidosis after one month of the simulated time.

Our evaluative approach towards the modeling and evaluation of breakdowns in acidobasic balance (Kofránek, 1980; Kofránek et al., 2007; Kofránek, 2009) is based on the modeling and evaluation of two flows – the creation and excretion of CO2 and the creation and excretion of strong acids, connected through the purification systems of each part of the bodily fluids. This approach, according to our opinion, better explains the physiological causality of acidobasic regulations, rather than direct modeling of acidobasic breakdowns through the balancing of accompanying electrolytes. Besides, the fidelity and truthfulness of the modeling process is getting better; mainly in mixed (acidobasic and electrolyte) breakdowns in inner environments.

Another important modification of the QHP, is the fact ***that the model was extended by adding the dependency of the potassium flow on the intake of glucose as a result of insulin,*** which enables us to model (besides other things), the influence of potassium solution infusions with insulin and glucoses, which are distributed in acute medicine for treating potassium depletions.

We have been using this „balancing and evaluation“ approach towards the modeling of acidobasic balance in our old „Golem“ simulator (Kofránek et al., 2001).

The extended QHP model serves as the base for the educational simulator ***„eGolem“,*** used in medical tutoring in clinical physiology of urgent statuses, which is being currently developed under the research project MSMT 2C067031. On the webpage of this project you may find the updated and current structure of our implementation of the QHP model (http://www.physiome.cz/eGolem/doc/QHP.html).

From a model to the simulator

A simulation model, implemented in the most sophisticated development environment, cannot be used as an education aid alone. It is the implementation of the formalized description of the modeled reality that enables testing of the behavior of the mathematic model during various input values and the search for model quotations and parameters, which within the established precision range, can ensure the sufficient compatibility of the behavior of the model with the modeled system (model identification).

Even after this goal is reached, there is still a long road ahead from the identified model to the educational or tutorial simulator (see Fig. 38).

The creation of a multimedia educational and tutorial simulator is very demanding developmental work requiring a combination of the ideas and experiences of all the teachers who create the script of the tutorial program, the creativity of art designers who create the interactive multimedia components, as well as the effort of programmers who create the necessary user interface and put together the final masterpiece and its final shape.

Thanks to advances in software technologies, we now have new tools available used not only for the creation of more effective simulation models but also tools that simplify the creation of own simulators equipped with attractive user interface graphics.

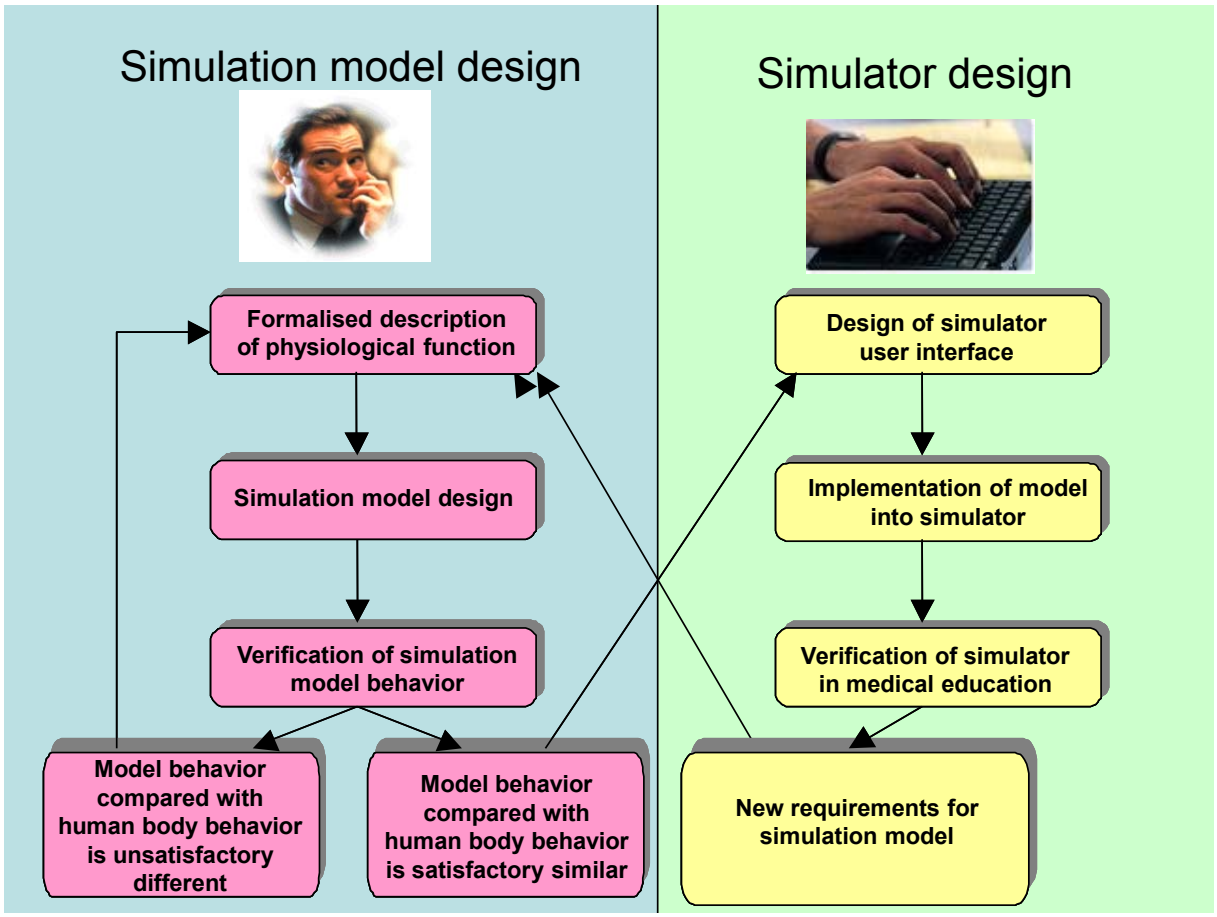


Figure 38: We have to deal with two types of problems when creating tutorial simulators: 1 ***The creation of the simulation model*** *– theoretical and development work, which is based on formalized presentation of reality, expressed through the mathematical model. The result should be the verified simulation model, which sufficiently represents the behavior of he modeled reality and within the required precision level.* ***2. The creation of the multimedia simulator*** *or rather the creation of the tutorial program, which uses simulation games – that is a practical application of theoretic results, which are connected and follows the research results. The base of the simulator is represented by created and verified models. It is a very demanding development work, which requires the combination of ideas and experiences of all teachers who create the script of the tutorial program, the creativity of art designers who create the interactive multimedia components, as well as the efforts of programmers who create the necessary user interface and put together the final masterpiece and its final shape.*

The skeleton of simulation application – the script

Simulation technologies have improved a great deal and have become more effective, while prices for equipment and software keep falling. The development of the Internet, virtual 3D environments such as the already mentioned software Second Life, or even the increasing range of medical simulators using robotized dummies, representing the patient, opened new options and potential for the medical education process. The use of a medical simulator during studies and education is becoming more and more popular and greatly utilized (mainly in the USA and Israel).

From the pedagogical point of view, an education process aided with a simulator is a very effective approach, but it is also a very demanding and challenging task for the teacher. Therefore, the most complex or sophisticated simulator equipped with the most attractive interface, is not guaranteed to be the most effective way of learning. Pedagogical efficiency depends on the ***teacher,*** who must have a

clear vision and know what is the most effective and most suitable way to use simulation model in the class.

The more complex the simulator, the clearer the picture the teacher must have in his mind, being sure of how the class should look and what types of simulation games he wants to use. This is also confirmed by our practical experiences with the use of complex simulators during classes (e.g. with Golem or QHP simulators). We have learned that a user interface offering many complex options or previews of hundreds of variables and parameters, takes away the concentration of students. Without clear pedagogical guidance pointing out what to look at or look for during the relevant simulation game while working with a complex simulator and without knowing how to interpret achieved results, the use of simulators is very ineffective.

However, from the pedagogical point of view, it is necessary to think in ***advance about how*** to use the simulator, before we actually begin with the creation. This is mostly true if we want to create multimedia and interactive educational programmes, available through the internet and using simulation games helping students understand and practice the subject better. The key for success is a ***good script.***

The first person the success depends on, is an experienced teacher, who must have a very clear picture in his mind as to how he wants to explain the problem to his students, how the multimedia application should work, where they may be used and how to use the simulation model to help him explain the subject better.

The skeleton (the backbone) of educational application is the ***script.*** The base is usually studying text – script, chapter in a textbook, etc. However, during the creation process of multimedia tutorial application, we have to imagine how the tutorial program will be displayed on the screen, how each screen should follow after another (the sequence), what the graphical design will be like, where interactive elements will be placed, where the audio input may be, how each animation will look, where the simulation model will be inserted, where the knowledge test will be inserted, how it will be evaluated and what reactions on the test results will be required.

The final shape of graphical elements is completed by a professional art designer. Therefore, a ***good communication between pedagogical expert/teacher – the script creator and the designer is necessary.*** The teacher doesn‘t need to draw perfectly, but he has to have a clear and well-planned image in his mind and be able to explain to the designer what he wants from him. The biggest roadblock from the beginning was the constant need to redraw already drawn animations, usually due to the pedagogue‘s/teacher‘s fault, as he did not have a clear vision or image as to how to create the multimedia element in the script. Therefore, it is well-worth paying attention to careful planning and preparation before the actual beginning of the project (Fig. 39).

During the creation process of the tutorial application script, we found it very useful to use a procedure that is applied during a regular animation film making process – that is to draw (best together with an artist) a pictorial script, a so-called „Story Board“ – an approximate sequence of each screen and then by using a regular text editor, write the relevant commentary (or a link reference pointing to the relevant part of the text) underneath.

Interactive multimedia programs are not scripts re-written and converted into a computer format. It is not a linear sequence of texts, audio sounds and moving pictures, as a typical animated movie is.

The significant feature of an educational and tutorial program is ***its interactivity*** and the possibility of branching and mutually interconnecting individual parts. To remake or transform text and picture scripts into a branching script linked with hypertext links of the accompanying interactive program is not easy at all.

One of the methodological problems that we had to solve during the creation of our scripts for tutorial application, was the problem how to display the structure of the tutorial program in the script, including interpretations, interaction with the user, program, branching, etc.

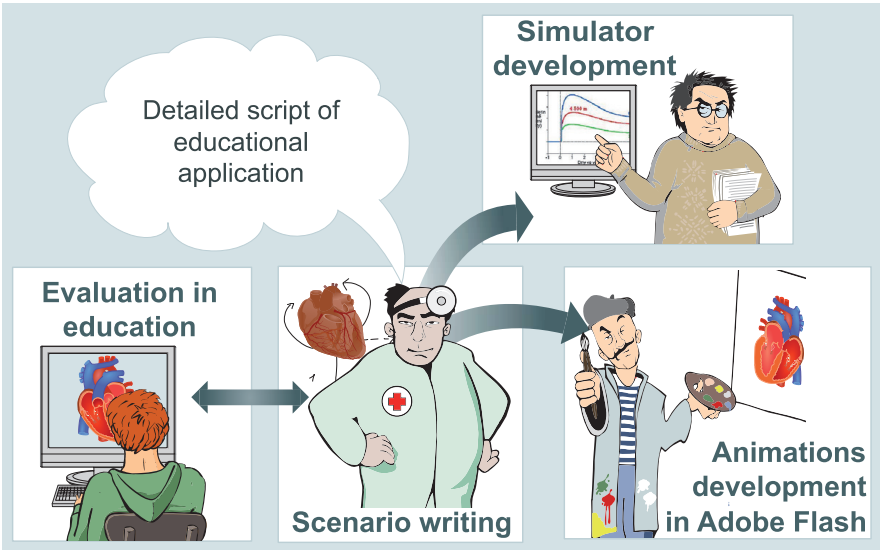


Figure 39: The pedagogue task – the role of the script creator in the development process of tutorial simulators is very important. The script author must together with the model creator precisely define how the model will be used in the tutorial application and how the user interface will look. They must prepare sufficient materials and specify the graphical requirements of all needed graphical elements and animations and present them before the art designer. Testing of the created simulator in class is very important as it usually opens new ideas and points out improvements and modifications.

The easiest way is to use a text or picture editor and by using regular block diagrams or structure diagrams, to describe the applicable branching, decision blocks, etc. with the relevant links on text pages and other relevant pictures stored in other files.

During the script writing, we also found using the abilities of modern text editors to create the required hypertext links helpful – which already gives the script some characteristics of its future interactivity.

We also tried using the ***Adobe Captivate*** tool to write the script for the tutorial application, see(http://www.adobe.com/cz/products/captivate/), which enabled us to create professional looking e-learning contents with advanced interactivity, without the need to have superb programming knowledge. We also found out that in order to write the script in the form of an interactively branched storyboard, this tool is too complicated. On the contrary. We learned that in order to share continuously created scripts between members of the development team, using simple tools is sufficient. To specify the requirements for the creation of the graphical elements given to designers, and for monitoring the results of their work and for monitoring other results, ***Microsoft OneNote*** was sufficient.

Modern interactive tutorial program is not an animated film converted into a computer form – the best advantage of interactivity offered by a computer, is the option to use ***simulator***, which through the use of simulation games enables the user to explain the problem in virtual reality. The script of the tutorial program must take this advantage into consideration. The author must answer these following questions – what type of simulation experiments should be offered with the simulation model, what the user interface of the simulation game will be like, and finally what are the requirements for the simulation model running in the background.

Therefore, it is necessary for the „scriptwriter“ to communicate with the model creator and to know the structure of the model, so he may propose possible modifications and be able to explain the specifications that the model should have and comply with.

A key factor is also pedagogical experience. Sometimes elements or issues that appear simple and easy-to-follow during the development of tutorial application, may become difficult and hard-to-follow issues once integrated into the pedagogical process. Besides that, during the use of simulation games in the education process, the modification of either the user interface or the simulation model on the application background is necessary.

It is also necessary to create script proposals in close cooperation with ***pedagogical experts and based on pedagogical experiences.*** Therefore, we found it very helpful to first try the simulation appli-

cation „on students“ in class and based on the results, create the explanation text, modify the simulator and propose the final form of the user interface for the production version of the relevant internet-based e-learning application.

Muscles of tutorial simulation applications – interactive multimedia components

To create the user interface of the tutorial simulator, it is rather impressive to present the simulator as a set of moving pictures, controlled by the simulation model. The controlled animations may graphically represent the meaning of numeric values – e.g. a schematic picture of a blood vessel may be extended or compressed, the heart may pulse quicker or slower, the lungs may breath deeper, the arm of a measuring apparatus may move and display a value of some variables of the model read from the simulation model running on the background, etc. On the other hand, we may enter various inputs into the simulation model (using various press buttons, buttons, levers and so on).

The graphic design is important as well, as it may determine how the tutorial and educational application will be accepted by potential users.To achieve a professional final look of the tutorial simulator, it is necessary to have an ***artist create the animations*** – the results are much better than if animations are created by a programmer, even with graphical talent. However, the art designer must be able to use interactive graphics tools. But art designers with these skills are very hard to find and artists who are able to work with these tools are in demand and are usually highly paid members of professional teams that produce computer games, web portals and multimedia commercials and applications, etc.

To accomplish this, we had to shift a ***great deal of our attention to training and schooling*** art designers in order to make sure that they know how to use these tools. Therefore, we began to work closely with the ***Václav Hollar College of Arts,*** where we opened an interactive graphics laboratory, as a detached workplace of the Charles University.

We spent a great amount of time trying to teach professional artists to work with interactive animation development tools, such as Adobe Flash, Microsoft Expression Blend, etc., or to work with 3D graphics development tools (e.g. Adobe Premiere), then to integrate it all into applications accessible through the internet (e.g. Adobe Flex) and finally teach them about the basis of program controlled interactive animations and about the creation of interactive web pages.

Our efforts were rewarded. Graphic artists stopped being shy in front of the computer and quickly understood that „a digital brush“ is just another creative tool, giving them a way to express themselves and that mastering a digital brush gives them an opportunity to succeed in the professional field.

We have also helped with the establishment of a „higher professional school“, where interactive graphics are taught in three-year courses. Workers of our Bio-cybernetics department and workers of the computer support system classes, also teach here, (**http://www.hollarka.cz/**).

On one side, the artist must work with the ***pedagogue, who creates the script of the tutorial application and also with the programmer*** who makes sure that interactive animations behave as required (e.g. he interconnects interactive animations with the simulation model). Therefore, even the art designer must have basic knowledge of programming, so he may communicate with the programmer effectively.

In the past, we have used the ***Golem simulator*** for the simulator interface, (Kofránek et al., 2001) and the ***ControlWeb software environment,*** originally designed for visualization and control of industrial processes (see http://www.mii.cz/cat?id=1&lang=409), which offers a wide range of preset elements – virtual instruments, which enabled us to create the user interface quickly and comfortably. However the quickly and comfortably created simulator looked more like a technological control panel than an interactive picture from a medical book (see Fig. 40).

When professional artists joined our development team, the potential of visualization improved significantly and we were no longer limited by the range of preset elements in tutorial programmes. We were able to create any shape of animated components, which we could connect with the simulation

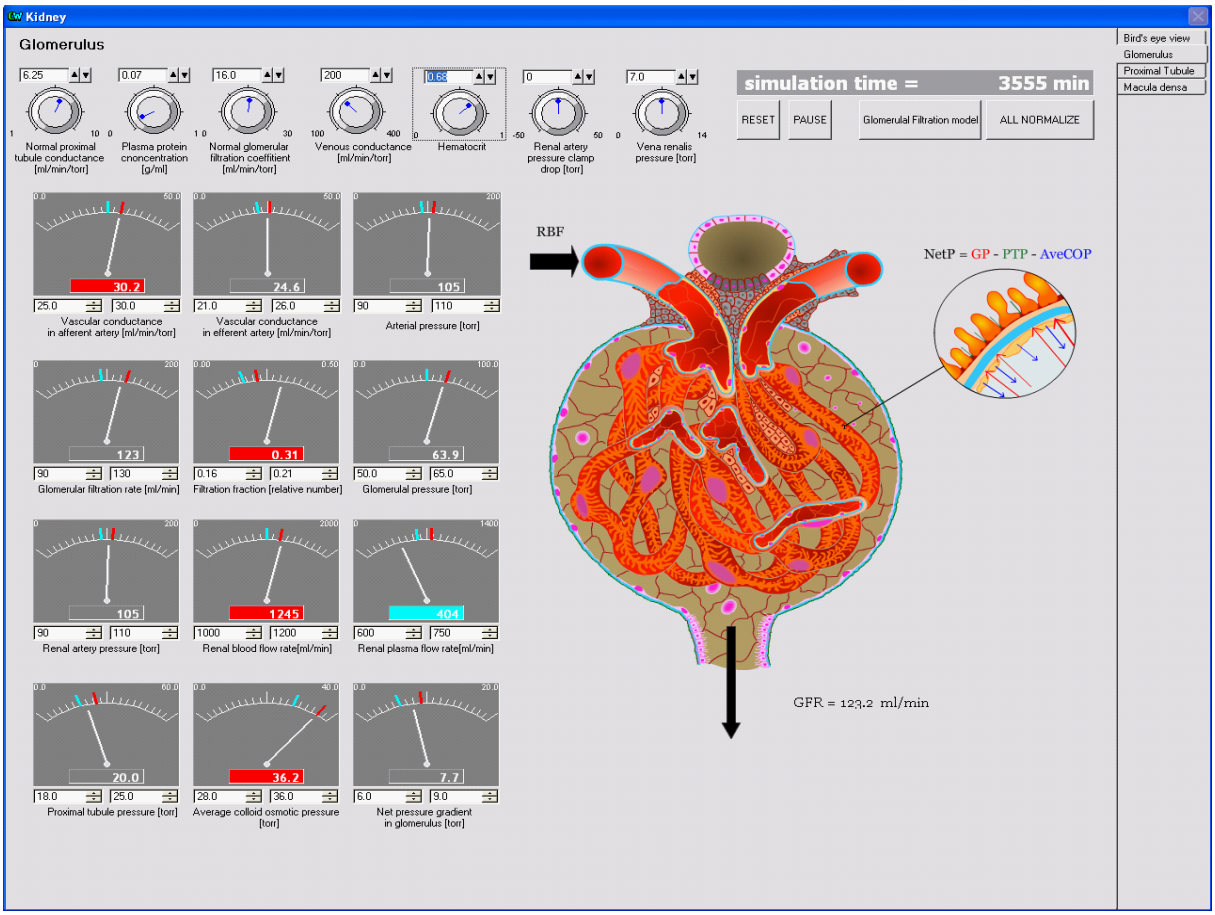


Figure 40: Visual interface of tutorial simulator demonstrating kidney functions, created in the ControlWeb environment. Outputs from the model are displayed on arm-gauges and at the same time they affect the shape of the inserted animated picture of the kidney glomeruli (artery diameters, arrow thickness and numeric value, etc.), created in Adobe Flash.

model and control them by the output or input values of the simulation model.

To create interactive multimedia components, connectable with the simulation model on the background, we use two tools (Fig. 41):

- The first one is ***Adobe Flash***, were we can create animated interactive components, which may be programmed (and may be connected with the simulation model in our applications). The created components may be easily played back in the internet browser (if the freely available Adobe Flash Player is installed) and on various operation system platforms. Also, less demanding numerical simulators, easily playable directly from the web page browser, may be created in Flash. We also used Flash components as the visual interface, communicating with the simulator core (through the ActiveX component) which were created in the ControlWeb and .NET environments.
- The second tool that we have started to use recreantly for the creation of simulator graphical components is the ***Microsoft Expression Blend*** development environment. This environment, while using the Microsoft .NET development environment, enables us to create applications that may be played or run directly in the internet browser, providing that the ***Microsoft Silverlight*** add-on is installed. The Microsoft Silverlight platform is capable of running numerically demanding applications with an interactive multimedia interface. This new Microsoft platform enables the user to distribute numerically demanding simulators over the internet, playable directly from the internet browser.

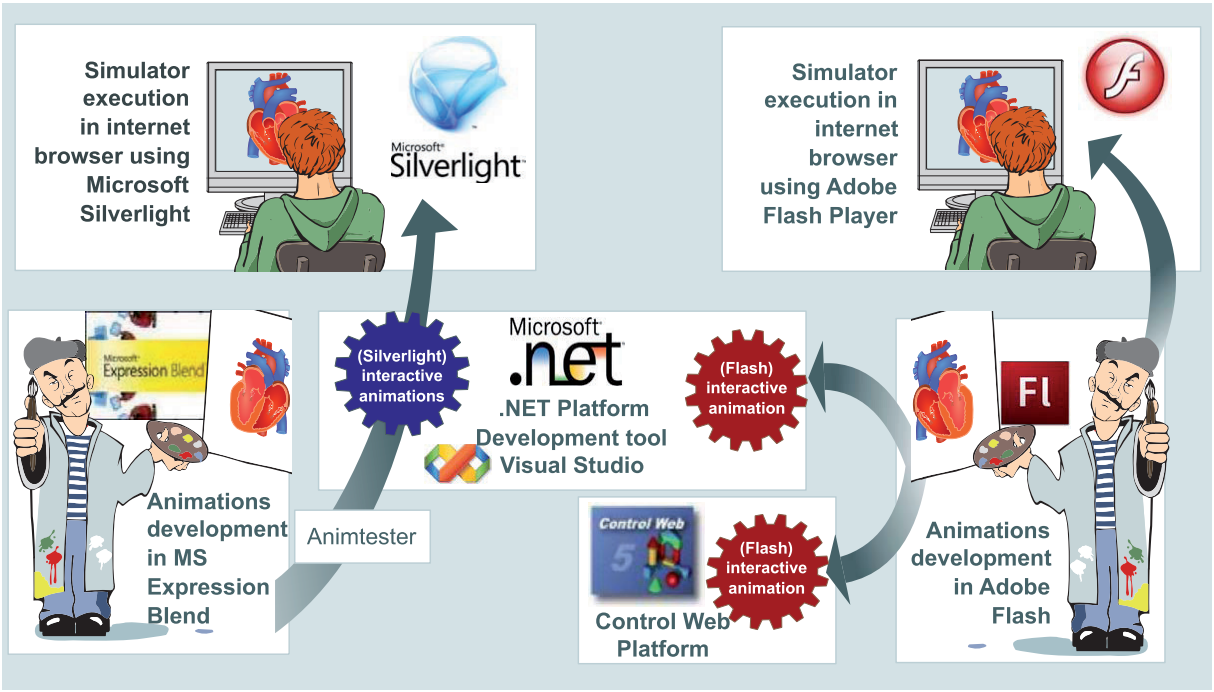


Figure 41: The responsibility of art designers is to create multimedia components and interactive animations connected with the simulation core of the tutorial simulators. Animation components created in Adobe Flash are used in numerically less demanding simulators, executable in internet browsers or as the user interface elements of simulators created in ControlWeb or Microsoft .NET. Recently, we have also been using Microsoft Expression Blend to create graphical components for multimedia simulators, executable through the Silverlight platform, directly from the internet browser. To simplify the creation of animations that will be controlled by the simulation model, we use our Animtester tool.

Adobe Flash Animation brush designed for artists

Flash has been developed over a long period of time. First, it was made by Macromedia and used only for the creation of animated pictures. At that time, we were creating tutorial animations in a different product, the Macromedia – Director (Fig. 42), which enabled us to control animations through scripts (Kofránek and Svačina, 2001).

During that time, the option to control animations through scripts became available for Flash as well, as the syntax got gradually richer and better. Starting with version 7 (sold and known as Macromedia Flash MX 2004), the Flash environment already contained an object control language (ActionScript) with syntax very similar to Java, which offers rather sophisticated and comfortable control of the visual interactive elements.

The huge success of Macromedia Flash is partly because its creators were able to successfully define an interface for artists ***and art designers*** (enabling them to create basic animation elements), as well as for ***programmers***, who by using the above-mentioned objective language can „vitalize“ and give interactivity to these components.

The basic component of the Flash application is a film/movie. Film may be divided into each scene, which may be played back in a programmed sequence or at random. Scenes are composed of sequences, which contain individual frames. Films may be linked – from each movie you may call a link from another movie and initiate playback of the movie. This is very useful in internet applications, when the first part of the animation is played and the other part of the movie is being downloaded in the background.

The creation of computer animations is based on the creation of classical animated films, where each frame is drawn on transparent foils and stocked above each other. Some frames may not be completely redrawn and they are only moved (e.g. the background is moved), while other frames must be redrawn completely or partially (e.g. just the moving figure, etc.) in order to create animation – moving pictures.

In Flash, (Fig. 43) each scene consists of several layers, which works similarly as the foils used in animated films.

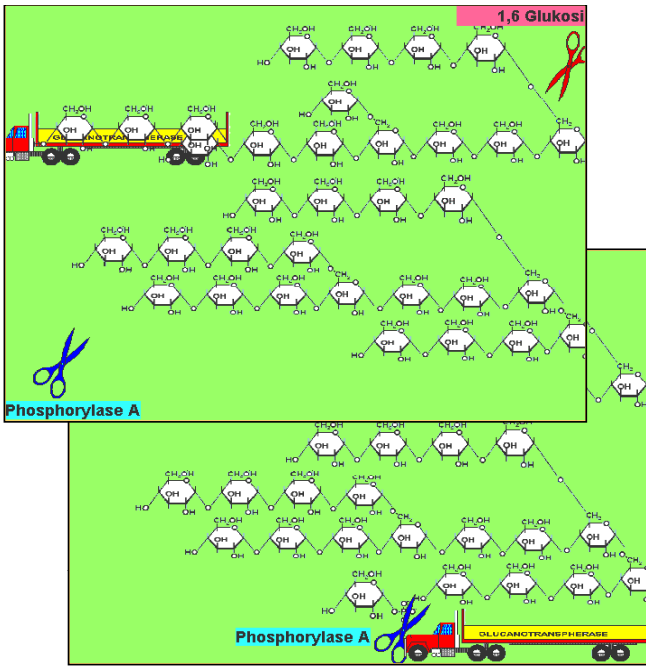


Figure 42: A multimedia tutorial program with interactive graphical animations created in Macromedia Director in 2000. At that time, Director offered stronger support for interactive programming than Flash. But Flash soon overcame this handicap and offered richer options of interactive controls than Director. Besides that, Flash offered a more intuitive user interface, resembling tools used for creation of animated films and became very popular among art designers. Therefore, we stopped using Director in 2000 and began using Flash for interactive animations.

Each frame or scene has several layers, where each picture element is stored. These visual elements may be drawn in each layer separately – Flash includes a powerful tool for the creation of vector pictures (vector pictures may also be imported from other external painting or drawing applications). Or the picture may be selected from a library and the required shape created. However, the sample shape may not only be represented by a still picture. You may select a clip from a movie, (MovieClip), which in reality is an instant classic of a previously created film. For example, if we want to create a picture of a plane, we can select a rotating propeller from a library and insert it into one of the layers, as one element of the picture. A special type of movie clips are buttons. The shape and graphics may be designed (when the cursor is moved over the button or when the button is pressed), as well as the button action, behavior and function. The movie clip may have a rather complex structural hierarchy – the film that creates the clip may contain instances from other clips. For example, a MovieClip of a car may contain movie clips of spinning wheels. Each instance of a MovieClip has its parameters (coordinates specifying its location on the screen, size, colors, transparency, etc.), which may be changed dynamically in the program. Besides that, the MovieClip class offers many methods that may be used (e.g. a method which detects the collision between two instances of a MovieClip, etc.).

During the creation of a MovieClip we can also program specific methods, which may be recalled from its instances. We may also program the complex behavior of visual components. It is rather easy to create special MovieClips as real components and then set their parameters and properties in a special component editor and recall their methods while the clip is running. This enables manufactures to create (and distribute and sell) various visual (and non-visual) components and helped with the introduction of Flash in the artistic community.

The graphical and the programing part of the movie is created in the development environment. Then it may be tested or translated into a sub-language (in .swf format), which may be interpreted by using the freely available and downloadable interpreter (the so-called Flash Player) and played back as an individually executable animation or it may be viewed in the internet browser (see Fig. 41).

Besides that, the created .swf file may be interpreted by using the special ActiveX component,

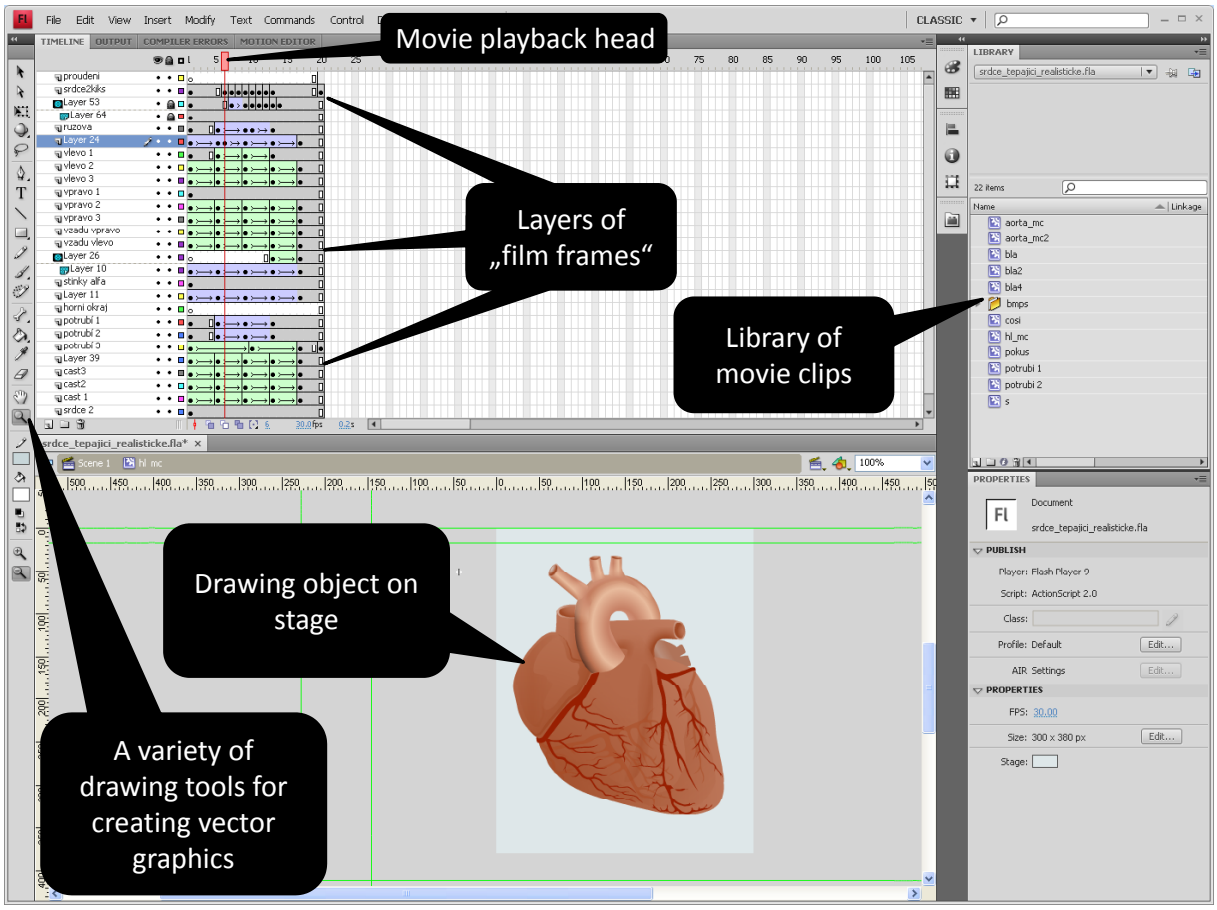


Figure 43: Adobe Flash environment offers tools for vector picture painting. Each layer of movie frames (as seen on the picture) may be also inserted with a sample of movie clip, selected from a library. The behavior of each visual and non-visual component may be programmed in a special programming window.

which may be integrated into another program – e.g. into an application created in ControlWeb or in Microsoft Visual Studio. The important thing is that this component may exchange messages with the application, enabling us to comfortably control the behavior of the interactive application through another application. The application may also receive messages from the interactive animation, describing or referring to the user action/intervention.

The huge success of Flash caused that the Adobe company bought Macromedia and Flash became one of the integral parts in portfolios containing computer graphic tools made by this manufacturer.

Today, Flash components may be used in so-called RIA formats (Rich Internet Application) – a new generation of multiplatform web applications with superb complex user interface design, created with ***Adobe Flex*** or as a desktop application created with ***Adobe Air***.

The speed of the .swf file interpreter (Flash Player) has improved and the ActionScript language can now be used to create the simulation core of tutorial simulators. The advantage of Flash tutorial and education applications (which may contain complex RIA applications, compiled in the Adobe Flex environment) is that these applications may be executed directly from the internet browser (providing that the relevant plug-in is installed) and run on all platforms.

We have created some tutorial simulators and multimedia interactive applications with simulation games in this environment (see Fig. 44).

In 2004, using Flash, we created (besides other applications) an interactive multimedia tutorial

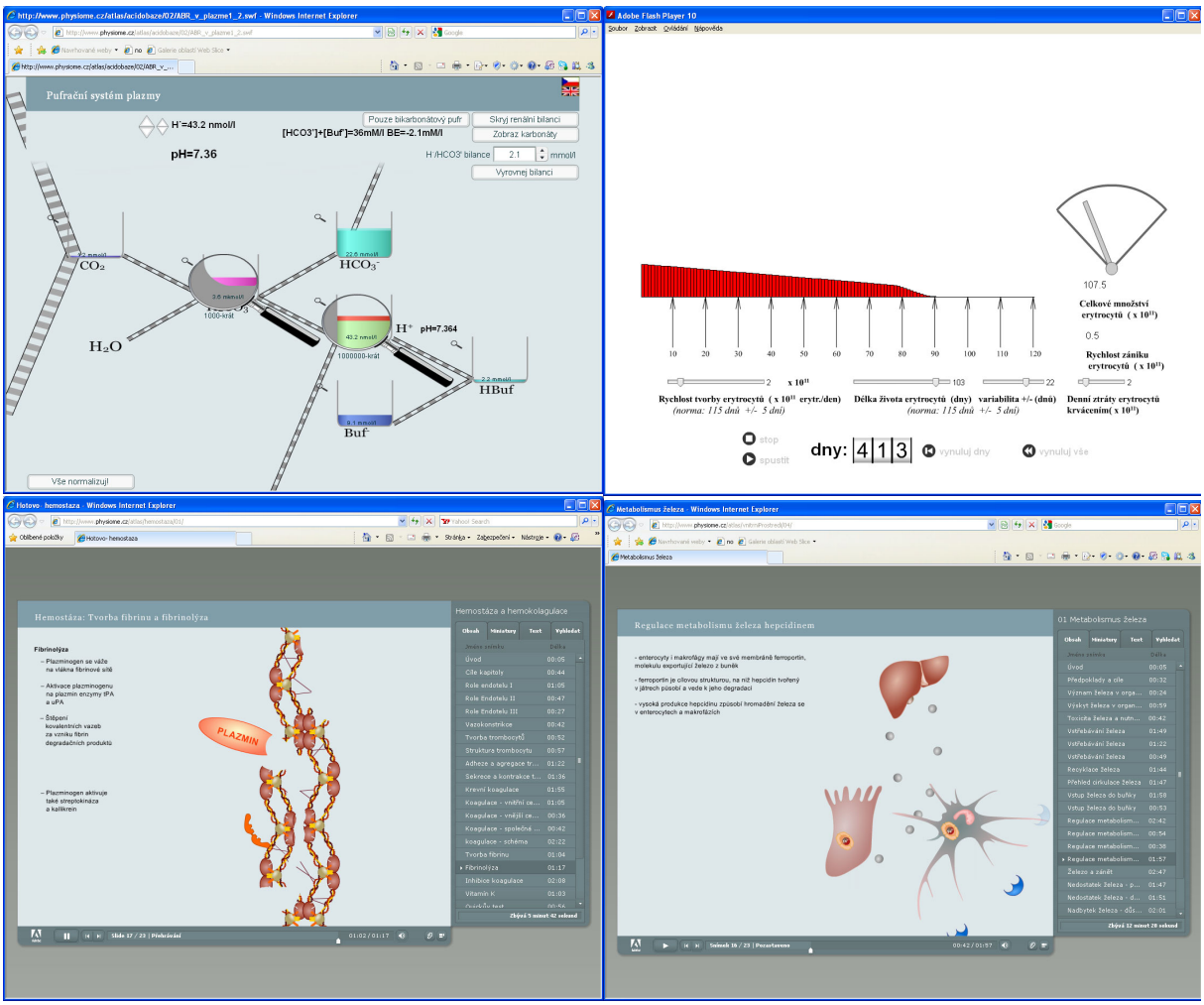


Figure 44: The Adobe Flash environment may be used for the implementation of simulators executable directly from the internet browser e.g. the model of acidobasic plasma balance or the anaemia model. Until now, Flash was also the main implementation environment for audio lessons and chapters in our internet Atlas of physiology and pathology. Lessons and lectures contain animations fully synchronized with sounds. Using a slider you may stop or move the lesson back and forth, while the animation is fully synchronized.

application called „Music as seen by physics and physiology. Tuning/tweaking Theory“ (http://physiome.cz/ladeni) for which we have received the „TECHFILM Laureate“ reward at the TechFilm festival, (Obdržálek, Kofránek, 2004). Using Flash, we have also created a twenty-minute film called „The Historical Meeting“ (Kofránek, 2006), dedicated to the history of relationship between the Czech Republic and Sweden during the Thirty-year war.

Our Atlas of physiology and pathology was also implemented using the Flash platform (see http://www.physiome.cz/atlas/index_en.html).

However, the Flash Player environment is still an environment based on the interpretation of .swf flash files. For numerically demanding calculation used in more complex simulators we encounter certain ***performance barriers.*** For more complex simulators, the Adobe Flash environment is (so far) insufficient.

More complex simulators are created in the ***.NET environment*** (and in the past they were created in ***ControlWeb***). Flash components are integrated into these simulators by using the ActiveX component. But to bridge between the two different „worlds“ of Adobe Flash and .NET, and to make sure that both operate in unison, hard programing work is involved.

Microsoft Expression Blend – a tool for the creation of „graphical puppets“ for simulators.

However, the Microsoft .NET platform may be used directly for graphical applications (without the need to interconnect Adobe Flash components). Thanks to ***WPF technology – Windows Presentation Foundation*** (Sells and Griffins, 2007) we can use the .NET platform to create complex graphic components containing animations, vector graphics, 3D elements, etc. Graphical elements may be created similarly as in Adobe Flash but with potentially better options for controlling their behavior, than in Adobe Flash.

Besides that, Microsoft swiftly reacted to the widely spread addition of Adobe Flash internet simulators and created its own ***Silverlight*** platform, which similarly as Flash Player, is capable of running complex applications combining text, vector and bitmap graphics, animations and videos from internet browser. The application runs as the primary application in the internet browser without the need to install it (the only required application to be installed is the Silverlight plug-in). Therefore, by installing

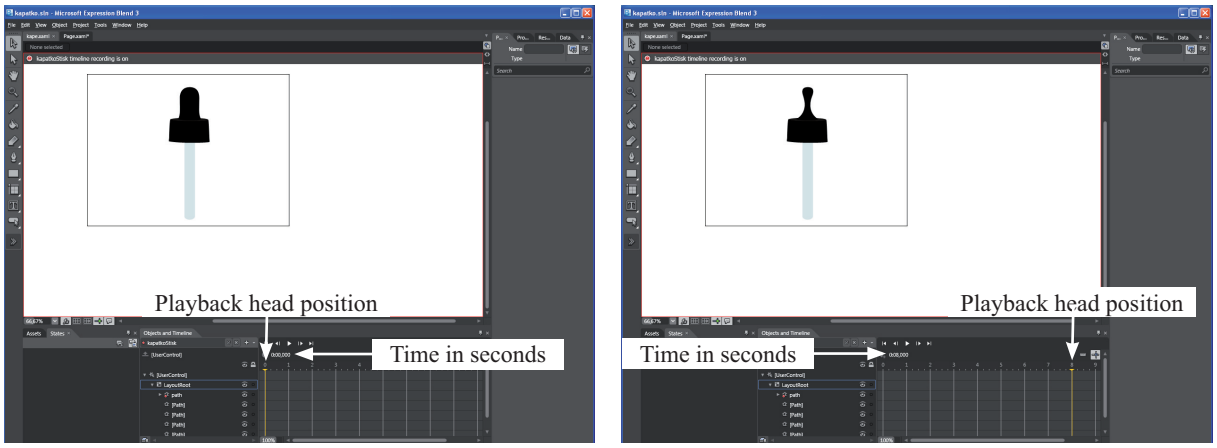


Figure 45: Key animation frames created in Microsoft Expression Blend are placed directly on the time axis. This makes the synchronization of animations with the audio track easier.

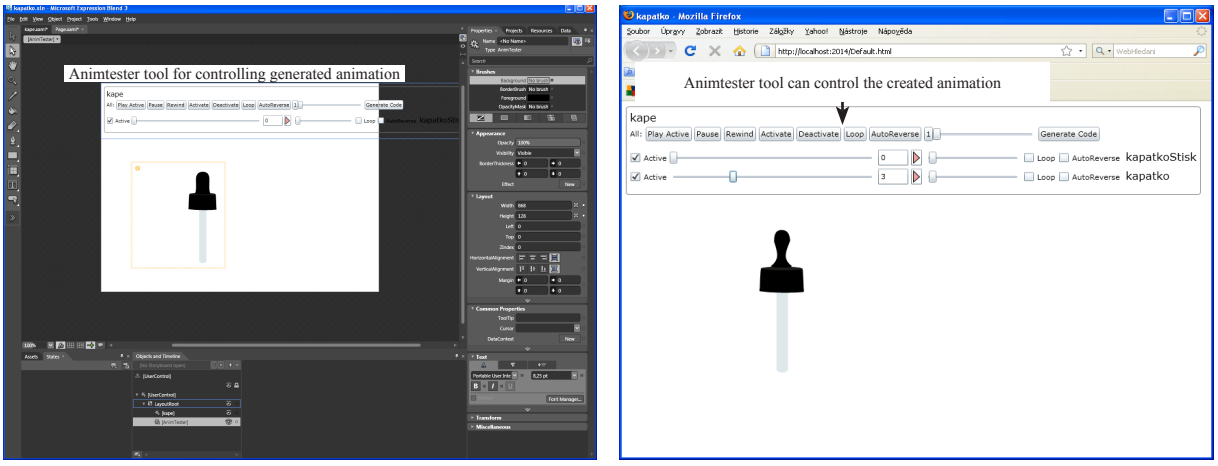


Figure 46: Animations created in Microsoft Expression Blend are actually becoming a property of the created graphical elements. By setting a value derived from a value on the time axis, you may control the shape of the displayed animated graphical element. Animtester enables the user to interconnect these properties with a visually displayable control element, which controls the animation in the running application and also to test the behavior of the animated graphical element. This tool is used for communication between the artist and the pedagogue – (different professional fields), where the pedagogue is the author of the tutorial application and the programmer interconnects the graphical elements with the simulation model within the tutorial application that is being created.

a small component, Silverlight enables the user to interactively control applications in the majority of web browsers (Internet Explorer, Firefox, Safari) and on various hardware and software platforms. Now Windows and Mac operational systems are directly supported for the most popular browsers and fully compatible open source implementation is under development for Linux OS. Applications created for this platform use a significant part of the .NET framework, which is a part of the plug-in (and therefore these applications may handle quite complex calculations).

Silverlight is a platform which is capable of distributing ***simulators that run directly in the internet browser*** via the internet (and even on ***computers with various operating systems*** – the only requirement is installing the relevant add-on (plug-in)).

The important parameter of ***Silverlight*** is that it ***includes native support of animations*** (Little, Beres, Hinkson, Rader, Croney, 2009). Therefore, animations are part of the application and it is not necessary to use an additional platform such as Adobe Flash for graphic layers.

The animation method is also more advanced. In the Adobe Flash environment, the animation is controlled by playing each frame in the preset speed (see Fig. 43 and Fig. 55). If all the frames are not rendered or loaded in time during the animation playback on a client computer, some frames are skipped over and the animation appears „jerky“. In Silverlight, the animation is driven directly by a time axis. That makes the playback smoother because the frame speed is progressively adjusted according to the resources available in the client computer – where the animation is played back.

Microsoft created a tool to create graphical elements and animations comfortably – the **Microsoft Expression Blend.** The graphical interface for ***Silverlight*** may also be created using this tool**.**

Microsoft Expression Blend offers an ***interface for art designers and programmers as well*** and works directly above the application created in Visual Studio .NET (Williams, 2008). That means that communication between ***the programmer and the art designer is much easier*** and design proposals don‘t need to be transferred over to the application project.

Key animation frames in Microsoft Expression Blend are not created by each frame (as in Ado-

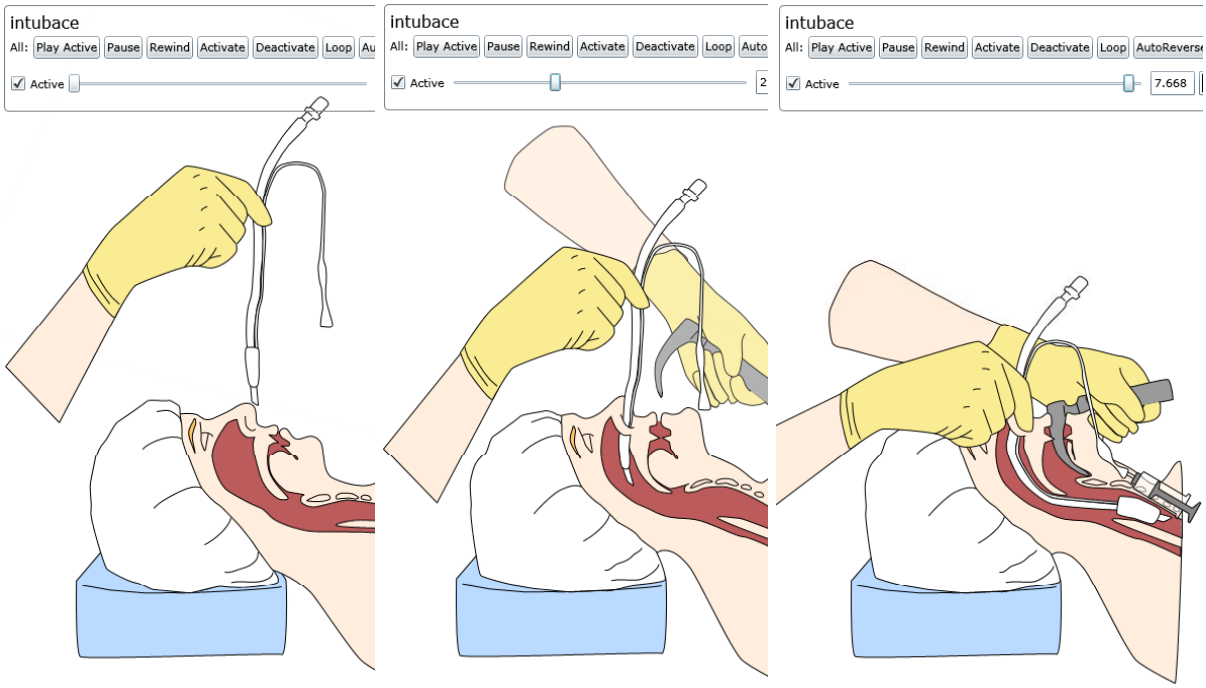


Figure 47: Patient intubation. An example of complex animation created in Expression Blend, while using Animtester.

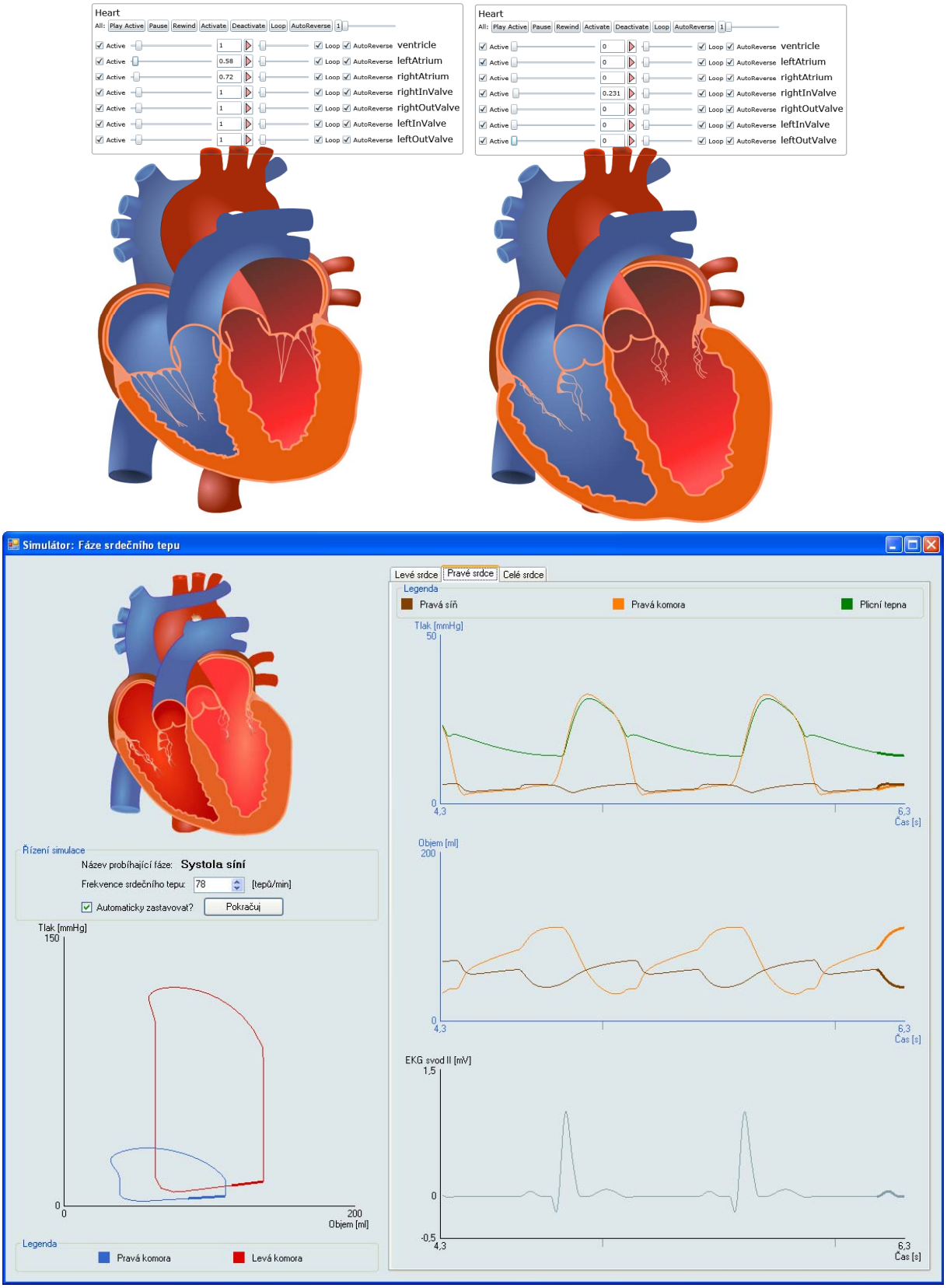


Figure 48: Animation of a beating heart. Outputs from the model affect the phases of the heart pulse, opening and closing of cardiac valves, etc. Auxiliary Animtester control elements are above the animation and enable the graphic designer to set and tweak each sub-animation. The graphical designer is completely shielded from the programming process. In the final simulator, the „control ropes and levers" are pulled by the simulation model, programmed on the background.

be Flash) but according to the time axis (Fig. 45). The drawn animations may then be expressed or described as the object property. The size of the animated project may be set according to value derived from the animation time axis (Fig. 46). Therefore, by entering the value of the created property we may control the shape of the animated graphical element.

The created and controllable animated graphical object may be used as a component for another, more complex animated object and it may be controlled or its shape may be changed by changing or by setting the values of the applicable property. This way we can create an animated puppet, whose final shape depends on the current setting of its component values.

To simplify the communication between the art designer and the programmer who implements his own simulator and also between the art designer and the author of the tutorial application, we have created a software tool called ***Animtester*** (Kofránek 2009). Using this software, graphic designers may tweak and create these puppets, without the need of additional programming work (see Fig. 47-48). Animester as a component is inserted into the Microsoft Expression Blend development tool and enables the user to generate properties of the graphical element from the created animations, controlled from outside, by using control elements (buttons and sliders). When the application is generated, the art designer and the author of the tutorial application (the pedagogue) may check and verify how the animated component will behave.

This enable the art designer to be shielded from the programming work details. And similarly, the author of the proposal of the tutorial application doesn‘t need to focus on the implementation details of the graphical proposal and may communicate with the art designer easier and thus reach his vision and goal quicker.

The task of the programmer implementing his own simulator is to interconnect the generated graphical object with the simulation model on the background. The animated „puppets“ created in this way and controlled via values that control their shape, may be ***directly connected to model outputs, without the need to add another programming „middle“/sub-layer for data propagation,*** as is necessary in ***Flash animations.***

The use of graphical options in the ***Silverlight platform*** replaces and compensates greatly for the original approach using animations based on the Adobe Flash platform. Therefore, during the creation of animations as a visual interface for simulators, we ***do not need the Adobe Flash platform, which may be fully replaced with the new animation tools from Microsoft.***

The brain of the tutorial and education application – the simulation model

Implementation of simulation models in the education and tutorial program is not a simple issue.

To create simulation models, we use special development tools, designed for tweaking, tuning and verification of simulation models (***Matlab/Simulink*** or acasual development using the ***Modelica*** language), which we discussed in previous chapters.

Debugged ***models*** must be ***converted*** from the development environment where they were created, debugged and verified ***into the environment where the tutorial simulator itself is being created.***

This may be done manually for simple models – as is often done in purely Flash tutorial simulators, where the development environment for the creation of simulators is Adobe Flash only.

However, for more complex models we created software tools which automate this work for us (see Fig. 49). In the Golem simulator, implemented in the ControlWeb environment, the model was represented as a controller of virtual measuring/control card (see Fig. 50). To automate the transfer or conversion of the simulation model from Matlab/Simulink we have created a generator, which creates the controller source text in the C language directly from the Simulink model. (Kofránek et al., 2002).

And to simplify the creation of simulators done in Visual Studio .NET (that is to eliminate „man-

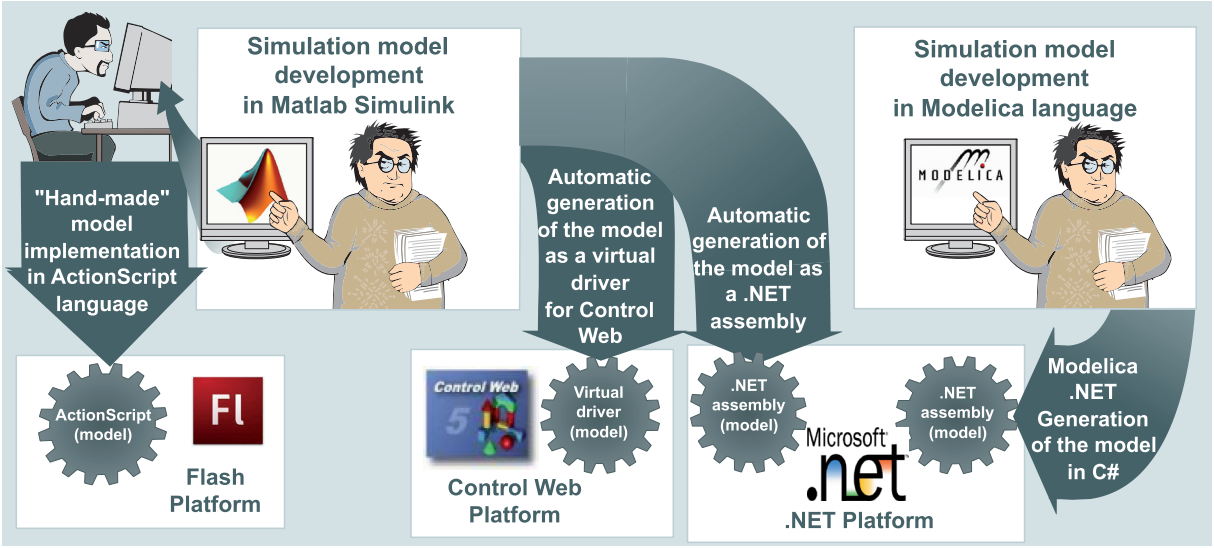


Figure 49: We have been using Matlab/Simulink for the development of simulation models but recently we began using the Modelica environment (e.g. Dymola or MathModelica). The identified simulation model may be manually re-written for the environment where the simulator will be created. This is possible only for simple simulators created in Adobe Flash or in Adobe Flex). For more complicated simulators created in .NET (or previously in ControlWeb) we have created special tools that automatically generate the simulation core from the Simulink model. At present, we are in the process of creating a tool for Modelica, which will be capable of generating the model in the C# language, which will enable us to create web simulators executable directly from the internet browser (using Silverlight).

ual“ programming of the debugged simulation model in Visual Studio .NET) we developed a special software tool (Kofránek et al., 2005; Stodulka et al., 2007), which automatically generates from the Simulink simulation model in a component form useable in the .NET environment.

The output in Modelica is the generated simulator program in C++. If we are ok with a simulator that needs to always be installed in the client computer then the program in C++ is sufficient. But, if we want to make use of the new options available in the .NET environment, which enables the user to create applications in Silverlight and run in the internet browser, then we have to create a tool which will

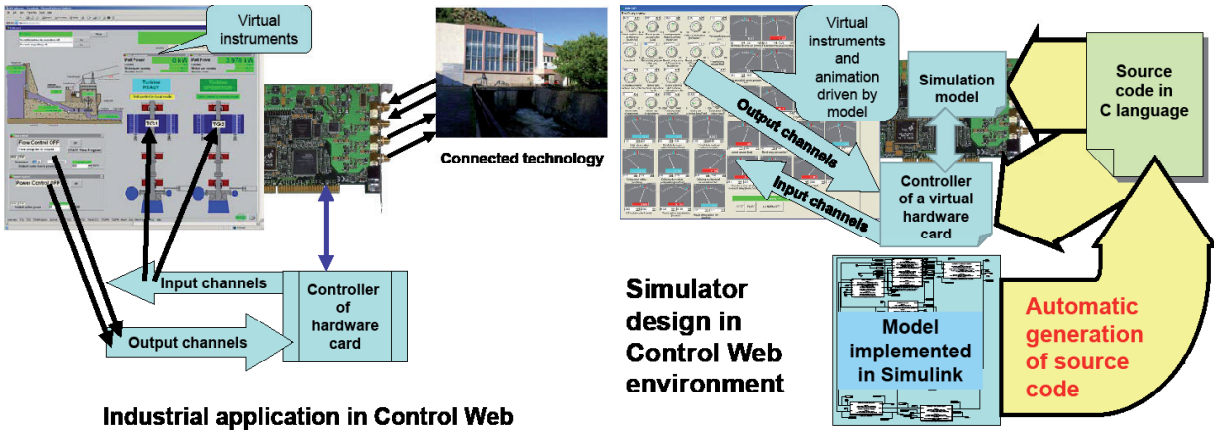


Figure 50: Simulator development in the Control Web environment, originally designed for visualization of control and measurement industrial applications. The simulation model is programmed as a software controller of a (non-existent) virtual card, and the application under development in Control Web communicates with the model as if it was a technological device.

generate the model source text in C# from Modelica, which is also our current goal at the international Open Modelica Source Consortium as mentioned earlier – (see http://www.ida.liu.se/labs/pelab/modelica/OpenSourceModelicaConsortium.html).

The body of the tutorial simulator – installed program or web application

The creation of a tutorial application is quite demanding programing work, based on the creation

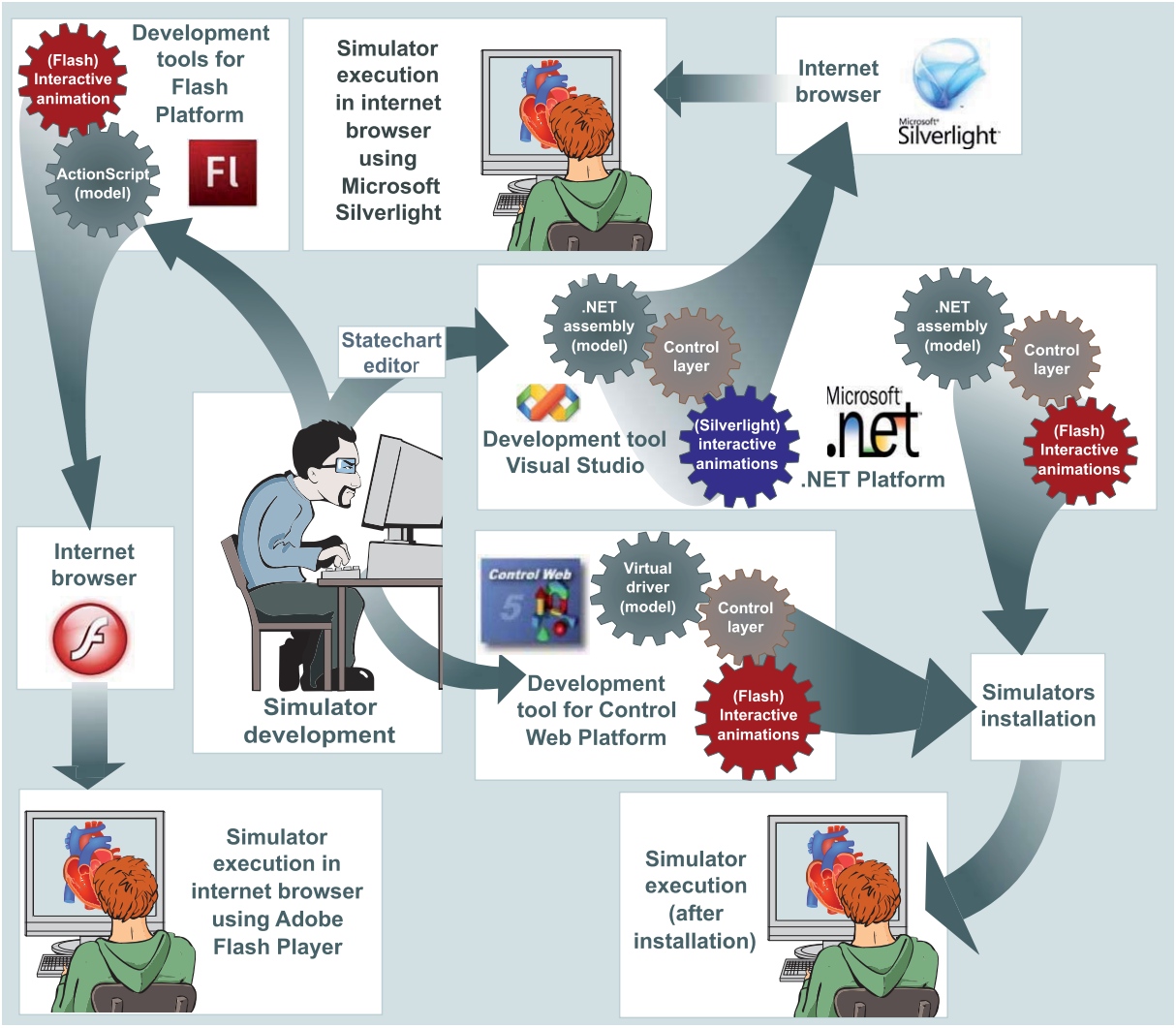


Figure 51: The programmer is responsible for the development of his own tutorial simulator. Based on the knowledge of the structure of the identified simulation model, we may use Adobe Flash or Adobe Flex to program the simulation core and the relevant user interface of the tutorial simulator, which uses Flash animations. The entire application may then be distributed via the internet and run/played in the internet browser, using Flash Player. However, more complex simulators are created in the .NET environment (in the past we were also creating simulators in ControlWeb). The simulator creation process assumes that the programmer will interconnect animations with the simulation component of the model (realized as . NET assembly). To make this programming process easier (and to create the interconnecting control layer) we have created a tool called Statechart Editor, which is based on the utilization of the hierarchy of status instruments. Simulators that use flash animations (interconnected with surroundings via ActiveX) require installation of the created application in the client computer. If we create animation in Microsoft Expression Blend, then we can create applications that are executable in the internet browser, with the Silverlight add-on. Using this method, we may create quite complex and numerically demanding tutorial simulators, distributed as web applications and executable in the internet browser, installed in the client computer.

of the simulation core of the developed application (unless this core was already automatically generated from the applicable simulation development tool) and its interconnection with graphical elements of the visual user interface (see Fig. 51).

For simple tutorial applications with simple models we can use Flash Player with the ActionScript language, which we use to program the simulator itself and the entire application may be run in the internet browser.

However, this is not enough for more complex applications.

In the past, we were creating simulators in the ***ControlWeb*** development environment, made by the Czech company Moravské přístroje (Kofránek et al., 2001; Kofránek et al 2002). The created application had to be installed in the client computer or (in case of web-distributed applications), at least the ControlWeb runtime environment had to be installed.

For the development of simulators today, we use the Microsoft .NET platform and for programming work, the Microsoft Visual Studio .NET programing environment, which offers great options for programming. We can also use the graphical component of the user interface created in Adobe Flash, which we can interconnect with (via ActiveX) the core of the simulator, which is represented by the simulation model and ***graphical components may then behave as puppets controlled by the simulation model.*** This method was used during the realization of our Atlas of physiology and pathology, dedicated to the basic dynamic properties of physiological regulatory systems – http://physiome.cz/atlas/sim/regulaceSys or the blood gas transfer tutorial simulator – http://physiome.cz/atlas/sim/BloodyMary.

The disadvantage of this approach is the necessity to install the program (offered through the internet interface) into the client computer. In such scenario the client needs to have the relevant installation rights applicable to the computer that he works on. However, this is not the case in computer classes, where computers are protected from the installation of unwanted software and the user must first ask the administrator for permission to install the educational program.

Therefore, it is desirable to be able ***to run and control even complex models directly from the web browser***. This is possible if the entire simulator can be executed in the ***Silverlight*** environment, that is, if the entire core has been created in the form of a control code designed to be used in the .NET environment (in .NET assembly), and the graphic components are created in the Microsoft Expression Blend environment.

Simulator structure – MVC architecture

If the architecture is more complex, the logic used to interconnect the visual user interface with the simulation model my be quite difficult. Therefore, it is better to insert a control layer between the visual element layer and the simulation model layer, which solves the communication logic between the user interface and the model and where the relevant context is stored. In literature (Collins, 1995; Leff, Rayfield, 2007) we may find the so-called MVC architecture (Model – View – Controller).

This setup is necessary mainly for more complex models and simulators, whose user interface is represented by many virtual instruments displayed on several interconnected screens. The advantages of this setup are seen mainly when applying model or user interface modifications (Fig. 52).

When designing the control layer, which connects the simulation model layer with the user interface, we found using interconnected status instruments very useful, which are able to memorize the relevant context of the model, as well as the context of the user interface.

For this reason, we have created a special software status tool called ***Statechart Editor,*** which enables us to visually design the interconnected status instruments, interactively test their behavior and automatically generate their source code for use in the Microsoft .NET environment. This tool makes programming the links connecting the simulation model with visual objects of the user interface in the tutorial simulator possible.

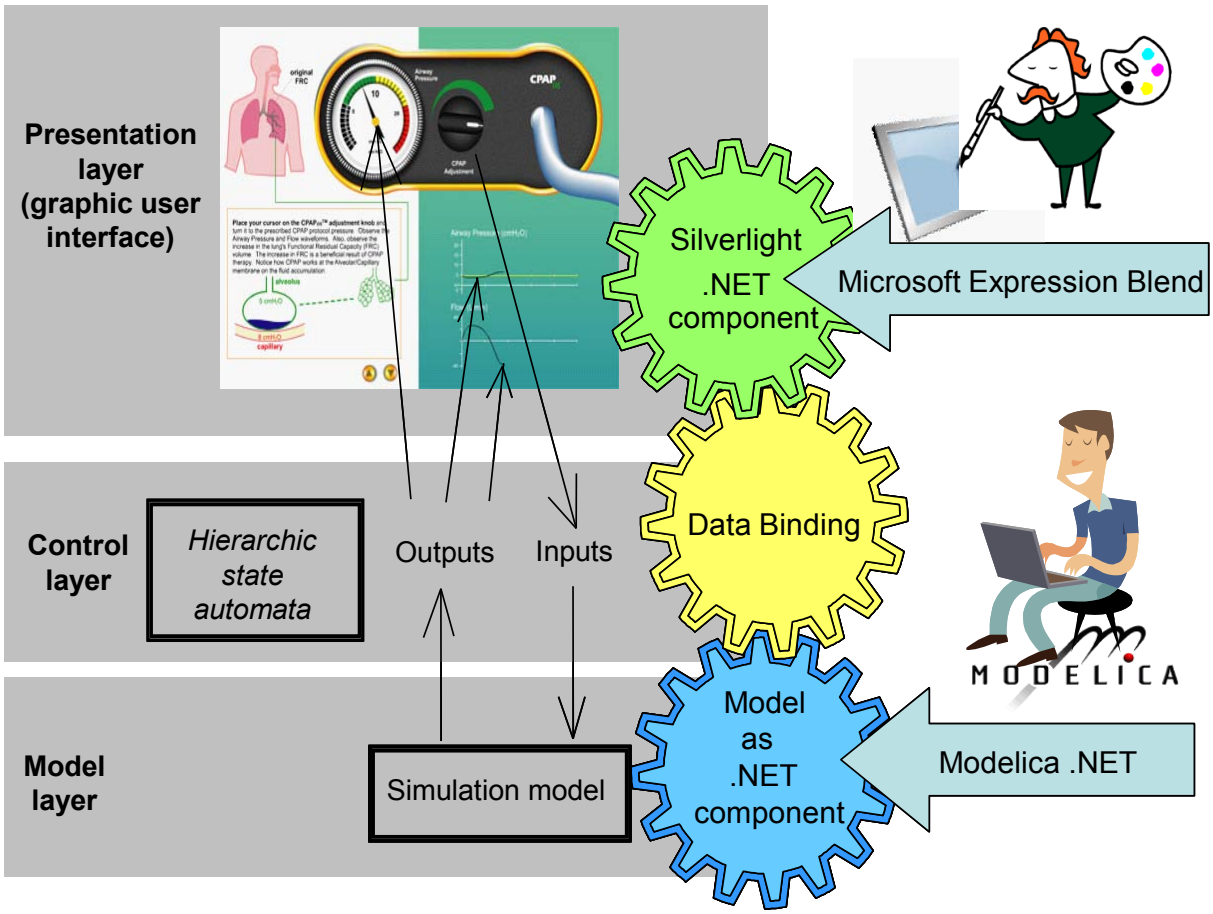


Figure 52: MVC architecture of the created simulator.

Interconnection of platforms used for the creation of models, simulators and animations

When creating simulators, we are forced to work with three types of different software tools.

- Software tools for the creation and debugging of mathematical models, which will be the base of the simulator – ***Matlab/Simulink*** and recently with acasual tools using the ***Modelica language.*** It is beneficial and effective to create simulation models in this environment, but it is problematic to operate simulators in this environment.
- A software ***tool for the development of own simulator*** – mostly the ***Microsoft Visual Studio .NET*** environment is used. In the past we were using the ***ControlWeb*** development environment, made by the Czech company Moravské přístroje, mainly because it offers excellent options for the quick creation of a simulator user interface – however, this interface has an overly technical character. Simple models are implemented in the ActionScript language and therefore it is sufficient to use the Adobe Flash environment, added with Adobe Flex.
- ***Software tools for creation of interactive multimedia graphics*** – user interface for simulators. Here, we have been using ***Adobe Flash*** (formerly Macromedia Flash) for a long time. Interactive animations, which may be programmed by using the special programming language ActionScript, may be created here. The important thing is that animations may communicate via the software (thanks to the above-mentioned programming option in the ActionScript language) with their surroundings, using ActiveX components. Today, we prefer the ***Microsoft Expression Blend*** development environment before Adobe, because we can create graphic components for the ***Silverlight platform***.

Because we use different development tools for simulation models and different tools for the simulator, we had to make sure that the results are flexibly transferred from one development environment into the other one – that is for example, automated model transfer from the Matlab/Simulink environment over to the Visual Studio Microsoft .NET (in the past over to ControlWeb).

These connection tools enabled us to develop and continuously update the mathematical model in the most suitable environment designed for mathematical model development and at the same time, to develop our own simulator in Visual Studio .NET (or in ControlWeb), without the need to „manually" reprogram the mathematical model.

Also, these tools made multi-disciplinary cooperation between members of the solution team easy – system analysts creating mathematical models and programmers implementing the simulator (see Fig. 53).

On the other hand, this method required working in three software environments and during each innovation we had to innovate the relevant connection tools.

From the user point of view, simulators are best distributed via the relevant web interface, which may be used also for the relevant interactive interpretation. Web based interpretation application may be easily interconnected with simple models implemented directly into ActionScript on a flash animation background (using this method we have created, for example, the tutorial application „Mechanical properties of skeleton muscle" – http://www.physiome.cz/atlas/sval/svalEN/SvalEN.html).

More complex models such as the complex model of blood gas transfer, (http://physiome.cz/atlas/sim/BloodyMary) required before the actual installation execution of the model on the client computer (and also the installation of .NET platform, which if not installed, is automatically downloaded from the Microsoft server).

The program installation process requires the user to have the relevant administration rights. Besides that, the model which runs as an independent application is connected indirectly with the web interface, where the multimedia interpretation is realized.

This problem solves our new technology of simulator creation, using the Silverlight platform (see Fig. 54). Graphical elements are created in the Microsoft Expression Blend environment.

But the simulation core needs to be realized as controlled code in the .NET environment – this should be ensured by our newly developed application Modelica .NET, which will be able to generate model code in C#.

To design the inner logic of the application we use hierarchical status instruments which are able to memorize the relevant context of the model, as well as the context of the user interface. The Statecharts editor environment, developed by us, enables the user to graphically design the instruments, to generate their code and tweak/debug them.

The benefit is that the graphical interactive elements and the simulation core are created on one platform – therefore the need to bridge between .NET and Adobe FLash, by using ActiveX components is eliminated.

The simulator must be easily combined with the interpretation chapter. The final application (the simulator and the interpretation chapter) may be realized as a web application executable directly in the web browser, without the need to install it in the client computer. It may run on various operation systems – the only requirement is to have the Silverlight plug-in installed in the relevant web browser.

Packaging of simulation games into multimedia interpretation

A simulator without an interpretation part requires an experienced pedagogue during use. Therefore, it is beneficial to combine simulators with explanatory lectures. Interactive programmes using virtual reality models offer great tutorial and educational benefits, because they combine tutorial text with

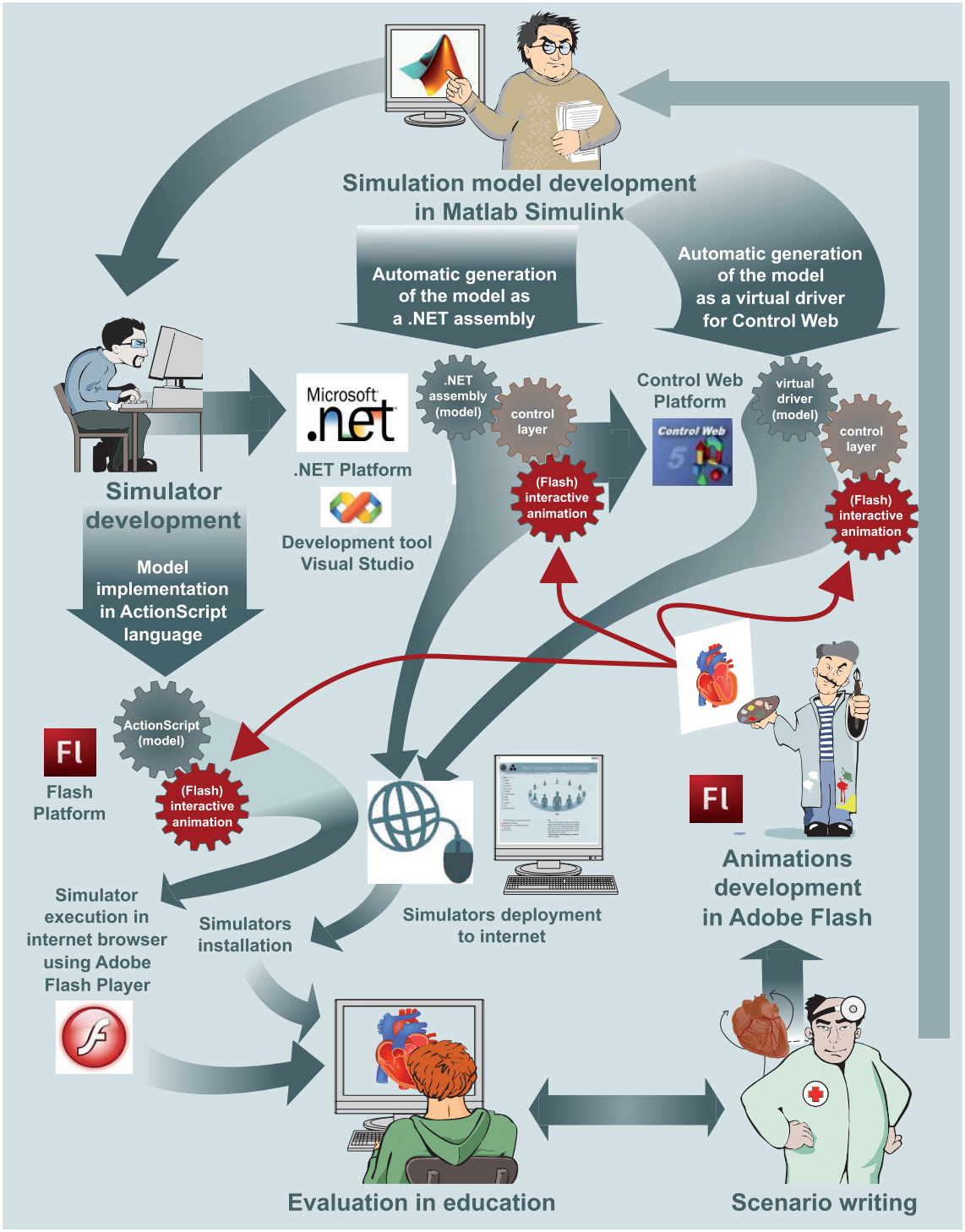


Figure 53: The original solution of creative interconnection of tools and applications, used for the creation of simulators and tutorial programmes using simulation games. The base of an e-learning program is a high-quality script, created by an experienced pedagogue. The creation of animated pictures is done by artists who create interactive animations in Adobe Flash. The core of simulators is the simulation model, created with special development tools, designed for the creation of simulation models. For a long time, we have been using Matlab/Simulink made by Mathworks for the development of models. The simulator development process is a demanding programming work. To make this task easier, we have developed special programmes that simplify the automatic transfer process of simulation models created in Matlab/Simulink over to ControlWeb or over to the Microsoft .NET environment.

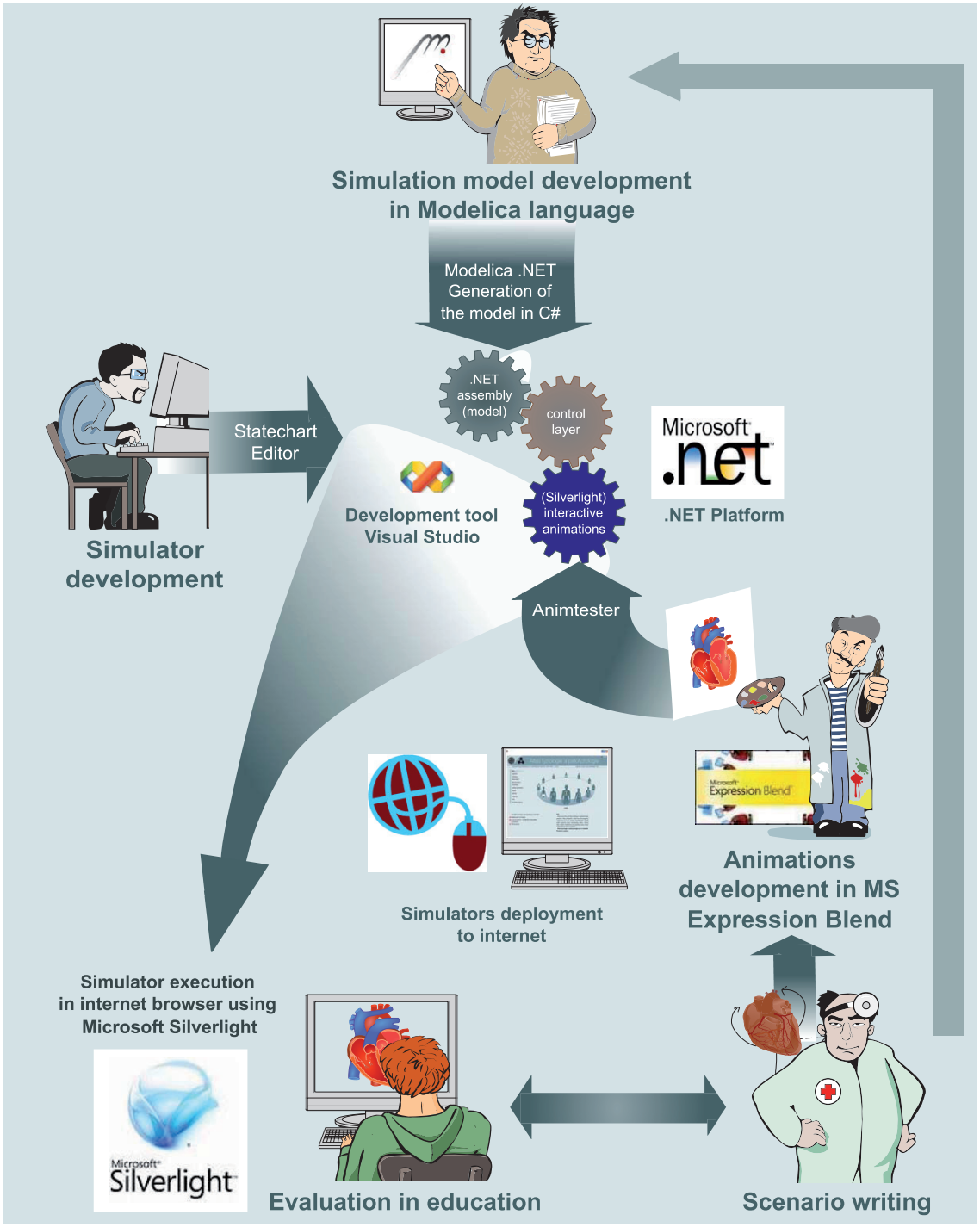


Figure 54: New solution of creative interconnection of tools and applications, used for the creation of simulators and tutorial programmes using simulation games. The base of an e-learning program is still a high-quality script, created by an experienced pedagogue. The creation of animated pictures is done by artists who create interactive animations in Expression Blend. To create and test animations that will be controlled by the simulation model, art designers use the Animtester software tool, developed by us. The core of simulators is the simulation model, created in the Modelica simulation language environment. Within the project Open Modelica Source Consortium, we are in the process of creating a tool which will be able to generate the source text from Modelica in the C# language. This will enables us to generate a component from .NET used in the final application on the Silverlight platform, which will enable us to distribute the simulator as a web application, running in the internet browser (even on computers with various operating systems).

animated pictures and with simulation games and therefore, better demonstrate and present the studied problem.

Our technology includes simulation games as a part of the e-learning multimedia tutorial lesson, based on a script created by an experienced teacher. The teacher or pedagogue compiles and proposes the wording of the text, as well as the shape of the pictures and animations.

Animations are designed by artists, closely cooperating with the pedagogue working in Adobe Flash and designing them for Flash Player, operating in the internet browser or (newer technology) the Microsoft Expression Blend platform in Silverlight.

The text is then read and synchronized with each animation and with simulation game references and links. Components are then compiled into study lessons.

However, to ***synchronize animation with audio*** in ***Adobe Flash*** is not that easy.

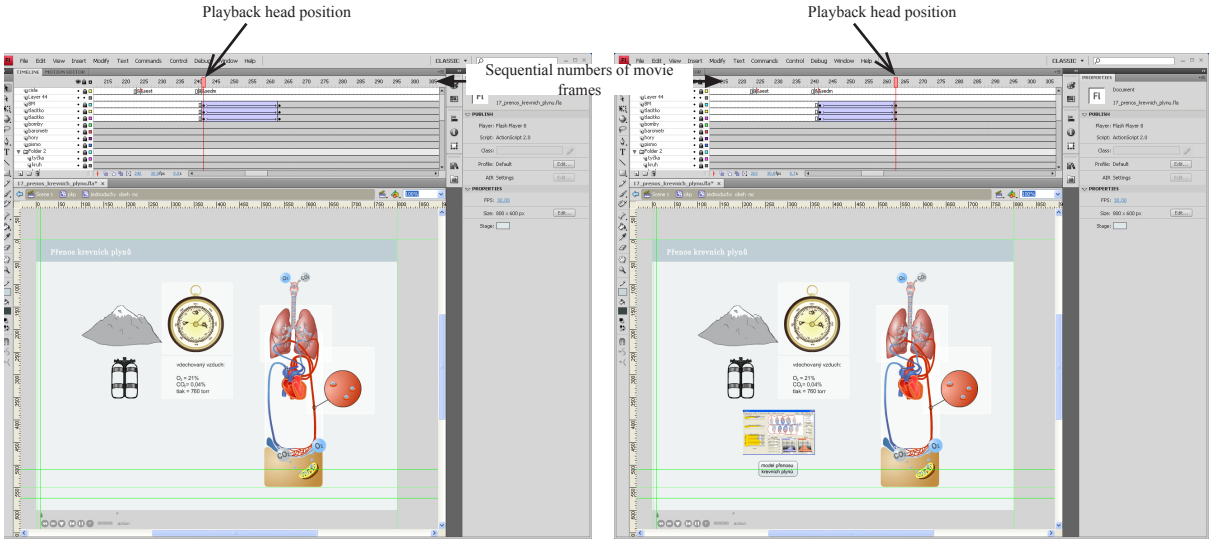


Figure 55: Animations created in Adobe Flash are created frame by frame as in a typical animated film. Providing that the playback speed is set, the timing visualization of individual animations depends on the time, when the playback head reaches the relevant scene/frame.

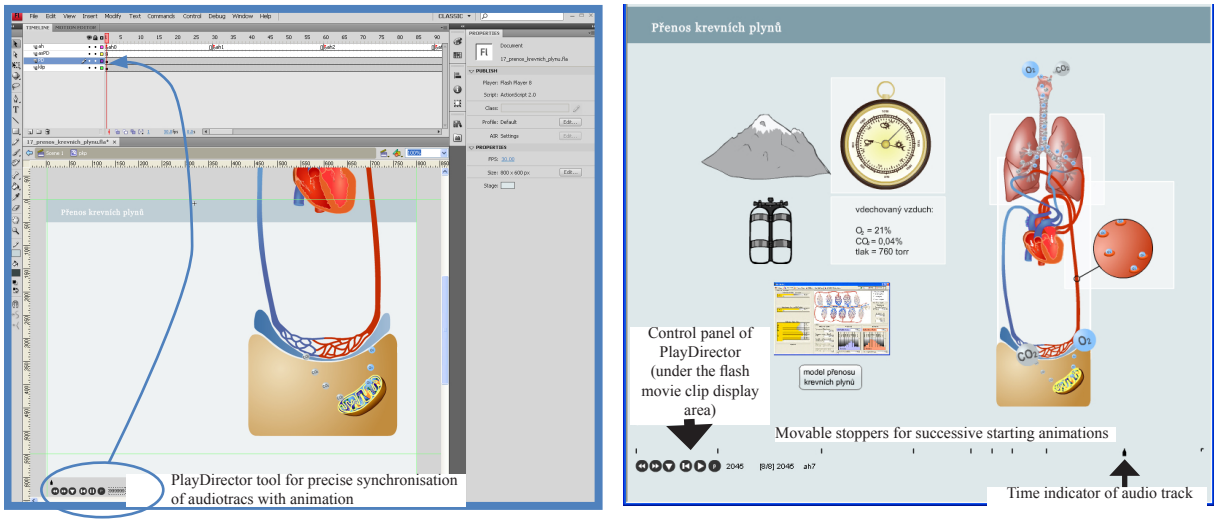


Figure 56: The PlayDirector tool is used for precise synchronization of audio tracks with animations. In the Adobe Flash environment, the tool is added to the created animation as the animation component. When played back in Flash Player, the tool enables the user to set required stoppers and thus set the precise time of the playback for each animation.

Animations in Adobe Flash are created as in regular animated film – by each frame (see Fig. 55) and therefore, the timing sequence of the visualization depends on the time, when the „playback head“ (at the preset playback speed), reaches the relevant scene/frame.

Audio track may be stored within the movie frame layer. When the synchronization process is done and animations with sound are played back, it is necessary to make sure (by using a command in ActionScript) that the played clip, playing the relevant animation, starts playing at the precise and correct time, when the playback head reaches its position.

To make the synchronization process easier, we have created a special tool called the ***PlayDirector*** – an element in the library, inserted into the created flash movie clip (Fig. 56). When the clip is played back in Flash Player, the audio tracks may be assigned interactive stoppers, according to which data are generated and used in the inserted script, which will make sure that the relevant animations will begin at the correct time.

To compile multimedia tutorial lessons created in Adobe Flash, we are using the Adobe Presenter development tool, supplied as software environment in the Adobe Connect server (Adobe Presenter may be purchased separately today – see http://www.adobe.com/products/presenter/).

However, the synchronization of animations with sound is much easier in the Silverlight environment. Because the animation approach in Silverlight is based on graphical property changes of the graphic objects within a certain time (unlike in Adobe Flash), the synchronization of animations with audio sounds is linear and therefore, the ***Microsoft Expression Blend*** environment is sufficient for this job and no other tools needs to be used or created.

Simulation games on the Web

One of the solid results of our efforts is our internet ***Atlas of physiology and pathology,*** created as a multimedia tutorial aid, which through the internet and through the use of the simulation model helps the teacher to explain the functions of individual physiological systems, causes and breakdowns - see http://physiome.cz/atlas.

Our Atlas is a part of the MEFANET network (MEdical FAculties NETwork), collecting electronic study textbooks and texts of medical universities in the Czech Republic and Slovakia (http://www.mefanet.cz/). Currently, an integration of English texts „Teaching Resources“ of the American physiological community into our Atlas is under way, see http://www.apsarchive.org/).

The Atlas combines interactive lectures and chapters and simulation games with models of physiological systems (see Fig. 57). During the creation of the model user interface, used as a base for simulation games, the atlas looked more like an atlas of animated pictures from the regular, printed Atlas of physiology (Silbernagl and Despopoulos, 2003) or the printed Atlas of pathophysiology (Silbernagl and Lang, 2005), rather than abstract regulation schemes used during biomedical classes.

However, contrary to the printed illustrations, pictures creating the multimedia user interface in simulators are ***„alive“ and interactive – changes in parameters or variables will change the picture look as well.*** Thanks to interactive illustrations we may create simulation games which, better than regular still pictures or simple animations, explains the dynamical relations in physiological systems and help students to understand the causes or reasons that are involved in the development of pathogeneses in various diseases.

The Atlas project is an open project – the results are freely accessible on the internet. It is created in a Czech and English version. Interactive animations and simulation models, including their source codes are available to all who are interested.

We would appreciate any cooperation and help from all who would like to be involved in the creation process of Atlas.

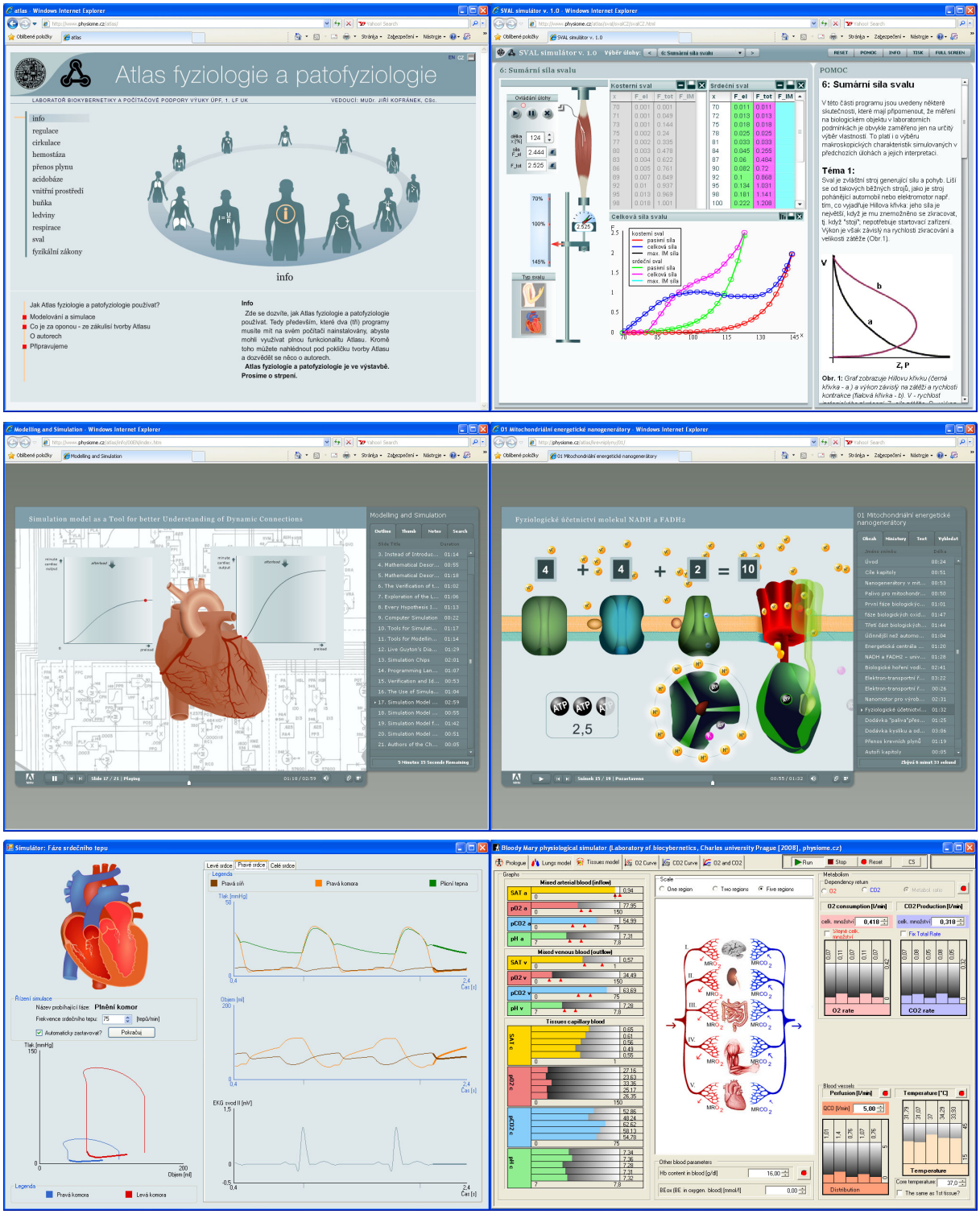


Figure 57: The Atlas of physiology and pathology combines interactive interpretations with sound, animations and simulation games. It has been created in Czech and (gradually) in an English version as well. It is freely available at: www.physiome.cz/atlas.

Conclusion – from enthusiasm to technology and multidisciplinary cooperation

Far gone is the time when a handful of enthusiasts, excited about the new possibilities of personal computers sold during the eighties, were making their own tutorial and education programmes. Computers are much more powerful today, the numeric and graphical possibilities of computers are enormous

in comparison with the computers of the eighties, a huge network of high-speed internet surrounds the entire planet and represents huge potential in the modern education process.

Also development tools and the methods of the software creation process are much more powerful and advanced. At the same time, users of these software applications must be much more experienced and educated.

The creation of high-quality software capable of utilizing the huge potential offered through the information and communication technologies of today, depends on the enthusiasm and hard work of individuals. It is a demanding and complicated development process, involving all kinds of professionals and experts:

- experienced teachers, whose script is the base for high-quality tutorial application
- system analysts working with professionals and experts on the relevant field and who are responsible for the creation of simulation models used in simulation games
- art designers who create the outer look and visual shape
- information specialists (programmers) who will „stitch up“ the application together into its final shape.

To make sure that the work of all kinds of specialist is efficient, it is necessary to have many interconnected development tools and methods which make the cooperation between all involved members easier and helps them overcome the differences and barriers in their fields. A great deal of hard work and effort needs to be put in to master these tools, but the final results are well worth it.

The tutorial software creation process is slowly becoming a blend and a combination of pedagogical experiences and the creativity of enthusiasts. It is mostly work for specialized teams using highly specialized development tools and it is beginning to look more and more like an engineering project.

References

[1] Abram, S.R., Hodnett, B.L., Summers, R.L., Coleman, T.G., Hester R.L., „Quantitative Circulatory Physiology: An Integrative Mathematical Model of Human Physiology for medical education.“ *Advannced Physiology Education*, 31 (2), pp.202 - 210, 2007.

[2] Amosov, N.M, Palec, B.L., Agapov, G.T., Ermakova, I.I., Ljabach, E.G., Packina, S.A., Soloviev, V.P.. *Theoretical research of physiological systems (in Russian)*. Naukova Dumaka, Kiev, 1977.

[3] Bassingthwaighte J. B., „Strategies for the Physiome Project“, *Annals of Biomedical Engeneering* 28, pp. 1043-1058, 2000.

[4] Brudgárd, J., Hedberg, D., Cascante, M., Gedersund, G., Gómez-Garrido, A., Maier, D., Nyman, E., Selivanov, V., Stralfors, P., „Creating a Bridge between Modelica and the Systems Biology Community“, *Proceedings 7th Modelica Conference, Como, Italy, Sep. 20-22, 2009*., pp. 473-479, The Modelica Association., Como, 2009. Available http://www.ep.liu.se/ecp/043/052/ecp09430016.pdf.

[5] Cellier, F. E., Nebot, A., „Object-oreiented modeling in the service of medicine“, *Proceedings of the 6tha Asia Conference, Bejing, China 2006.* 1, pp 33-40. International Academic Publishers, Bejing, 2006

[6] Coleman T.G, Hester, R., Summers, R. (2008, July): *Quantitative Human Physiology* [Online] Available http://physiology.umc.edu/themodelingworkshop/

[7] Collins, D. (1995). *Designing object-oriented user interfaces*. Benjamin Cummings (ISBN: 0-8053-5350-X), Redwood City, CA, 1995.

[8] Dabney J.B., Harman T.L., *Mastering Simulink*, Prentice Hall (ISBN: 0-13-142477-7), Houston, 2004,

[9] Grodins F.S., Buell J., Bart A.J., „Mathematical analysis and digital simulation of the respira-

tory control system.“, *J.Appl. Physiol.,* 22 (2), pp. 260-276, 1967.
[10] Guyton A.C., Coleman T.A., Grander H.J., „Circulation: Overall Regulation.“, *Ann. Rev. Physiol.*, 41, 13-41, 1972.
[11] Guyton A.C, Jones C.E and Coleman T.A., *Circulatory Physiology: Cardiac Output and Its Regulation.* WB Saunders Company, Philadelphia, 1973.
[12] Fencl, J., Jabor, A., Kazda, A., Figge, J., “Diagnosis of metabolic acid-base disturbances in critically ill patients“, *Am. J. Respir. Crit. Care.*, 162, pp. 2246-2251, 2000.
[13] Fritzson P., *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley-IEEE Press (ISBN 978-0-471-47163-4), Piscataway, NJ., 2003
[14] Haas, O. C., Burnham, K. J., „Systems Modeling and Control Applied to Medicine“, in O. C. Haas, K. J. Burnham, *Intelligent and Adaptive Systems in Medicine*, pp. 17-52, CRC Press, Boca Raton Fl , 2008.
[15] Hall J.E., „The pioneering use of system analysis to study cardiac output regulation“, *Am.J.Physiol.Regul.Integr. Comp.Physiol.* 287, pp. 1009-1001, 2004.
[16] Harel, D., „Statecharts: a visual formalism for complex systems“, *Science of Computer Programming* 8, pp. 231-274, 1987.
[17] Hester L. R., Coleman T., Summers, R., „A multilevel open source model of human physiology.“ *The FASEB Journal*, 22, p. 756, 2008
[18] Hunter P.J., Robins, P., Noble D., „The IUPS Physiome Project“, *Pflugers Archive-European Journal of Physiology*, 445, pp. 1–9, 2002.
[19] Ikeda N., Marumo F., Shirsataka M. A., „Model of overall regulation of body fluids“, *Ann. Biomed. Eng.* 7, pp. 135-166, 1979.
[20] Khoo M.C., *Physiological Control Systems: Analysis, Simulation, and Estimation*,.IEE Press (ISBN 0-7808-3408-6), New York 2000,
[21] Kofránek, J., „Modelling of blood acid base balance“, *Ph,D, Thesis,* Charles University, Faculty od General Medicine, Prague, 1980.
[22] Kofránek, J., Svačina, Š., „Multimedia simulators of glycoregulatory mechanisms as an interactive teaching tool,.“ in J. G. Anderson, M. Kapzer (Editor), *Simulation in the Health and Medical Sciences 2001*, pp. 165-170. Society for Computer Simulation International, Simulation Councils, San Diego, 2001.
[23] Kofránek J. Anh Vu L. D., Snášelová H., Kerekeš R. and Velan T., „GOLEM – Multimedia simulator for medical education“, in *MEDINFO 2001, Proceedings of the 10th World Congress on Medical Informatics. (London, UK, 2001)*, Patel, L., Rogers, R., Haux R. Eds., pp. 1042-1046, IOS Press, London, Available http://www.physiome.cz/references/medinfo2001.pdf.
[24] Kofránek, J., Andrlík, M., Kripner, T., Mašek, J., „Simulation chips for GOLEM – multimedia simulator of physiological functions“, in J. G. Anderson, M. Kapzer (Editor), *Simulation in the Health and Medical Sciences 2002*. pp. 159-163, Society for Computer Simulation International, Simulation Councils, San Diego, 2002. Available http://www.physiome.cz/references/simchips2002.pdf .
[25] Kofránek, J., Andrlík, M., Kripner, T., Stodulka, P., „From Art to Industry: Development of Biomedical Simulators“, *The IPSI BgD Transactions on Advanced Research , 1 #2(Special Issue on the Research with Elements of Multidisciplinary, Interdisciplinary, and Transdisciplinary: The Best Paper Selection for 2005)*, pp. 62-67. 2005. Available at http://www.physiome.cz/references/ipsi2005.pdf.
[26] Kofránek, J. (Director), Klučina, P. (Script), Kofránek, J. (Producer), Rejl, V. (Designer), Obdržálek, J. (Music), *Möten i historien, Historica encounter, Dějinné setkání,* Motion picture film. Bajt servis s.r.o. in cooperation with Charles University. Prague, 2006.
[27] Kofránek, J, Rusz, J., Matoušek S., „Guytons Diagram Brought to Life - from Graphic Chart to Simulation Model for Teaching Physiology“, in *Technical Computing Prague 2007. Full paper CD-ROM proceedings.* (P. Byron Ed.), pp. 1-13, Humusoft s.r.o. & Institute of Chemi-

cal Technology, Prague, 2007. Available (including source code): http://www.humusoft.cz/akce/matlab07/#k.

[28] Kofránek, J., Matoušek, S., & Andrlík, M., „Border flux ballance approach towards modelling acid-base chemistry and blood gases transport“, in V B. Zupanic, S. Karba, S. Blažič (Editor), Proceedings of the 6th *EUROSIM Congress on Modeling and Simulation, Full Papers (CD)* (TU-1-P7-4, pp. 1-9), University of Ljubljana, Ljubljana 2007. Available: http://www.physiome.cz/references/ljubljana2007.pdf.

[29] Kofránek, J., Mateják, M., Matoušek, S., Privitzer, P., Tribula, M., & Vacek, O., „School as a (multimedia simulation) play: use of multimedia applications in teaching of pathological physiology.“ *MEFANET 2008. CD ROM Proceedings,* (ISBN 978-80-7392-065-4), kofranek.pdf: pp. 1-26, Masarykova Univerzita, Brno, 2008. Available: http://www.physiome.cz/references/mefanet2008.pdf.

[30] Kofránek, J., Mateják, M. Privitzer, P., Tribula, M., „Causal or acausal modeling: labour for humans or labour for machines“, in V C. Moler, A. Procházka, R. Bartko, M. Folin, J. Houška, P. Byron (Editor), *Technical Computing Prague 2008, 16th Annual Conference Proceedings, CD ROM,* 058_kofranek.pdf: pp. 1-16. Humusoft s.r.o., Praha, 2008, Available: http://www.physiome.cz/references/tcp2008.pdf.

[31] Kofránek, J., „What is behind the curtain of a multimedia aducational games?“ in *EATIS 09 Contribution Proceedings. Euro American Conference on Telematics & Information Systems, 2009.* (ISBN 978-80-87205-07-5, 229-236), Wirelesscom sro., Prague, 2009. Available: http://www.physiome.cz/references/eatis2009.pdf.

[32] Kofránek, J., „Complex model of acid-base balance (in Czech).“, in *M. Zeithamlová (Editor), MEDSOFT 2009*, Praha: Agentura Action M., pp. 23-60. English translation of the paper is available at http://www.physiome.cz/references/medsoft2009acidbase.pdf, model is available at http://www.physiome.cz/acidbase.

[33] Leff, A., Rayfield, J. T., „Web-application development using the Model/View/Controller design patterns“, in *Fifth IEEE International Enterprise Distributed Object Computing Conference*, (ISBN: 0-7695-1345-X), IEE International, Seattle,118, 2007

[34] Little, J. A., Beres, J., Hinkson, G., Rader, D., Croney, J., *Silverlight 3 programer‘s reference (Wronx programer to programer)*. Wronx Wiley, Indianopolis, 2009.

[35] Logan, J. D., Wolesensky, J. D., *Mathematical methods in biology.* John Wiley & Sons,,Inc., Hoboken, NJ, 2009.

[36] McLeod, J., „PHYSBE: A ophysiological simulation benchmark experiment“, *Simulation*, 15: pp. 324-329, 1966.

[37] McLeod, J., „PHYSBE...a year later“, *Simulation*, 10, pp. 37-45, 1967.

[38] McLeod, J., „Toward uniform documentation-PHYSBE and CSMP“, *Simulation*, 14, pp. 215-220, 1970.

[39] Obdržálek, J., Kofránek, J., *Tuning theory, mutimedia educational program (in Czech)*, Charles University Prague, 1st Medical Faculty, Prague, 2004. Available: http://www.physiome.cz/ladeni.

[40] Oomnes, C., Breklemans, M., Baaijens, F., *Biomechanics: concepts and computation.* Cambridge University Press, Cambridge, 2009.

[41] Raymond, G. M., Butterworth E, Bassingthwaighte J. B., „JSIM: Free Software Package for Teaching Physiological Modeling and Research“, *Experimental Biology*, 280, pp. 102-107, 2003.

[42] Silbernagl, S., Despopoulos, A., *Taschenatlas der Physiologie.* Georg Thieme Verlag (ISBN 3-13-567706-0), Stuttgart ,2003

[43] Silbernagl, S., Lang, F., *Taschenatlas der Pathophysiologie.* Georg Thieme Verlag (ISBN 3-13-102192-6), Stuttgart: , 2005

[44] Sirker, A. A., Rhodes, A., Grounds, R. M.,“ Acid-base physiology: the ‚traditional‘ and ‚modern‘ approaches“, *Anesthesia*, 57, pp. 348-356, 2001.

[45] Stewart, P. A., „Modern quantitative acid-base chemistry“, *Can. J. Physiol. Pharmacol.*, 61, 1444-1461, 1983.

[46] Stodulka, P., Privitzer, P., Kofránek, J., Vacek, O., „Development of WEB accessible medical educational simulators“, in V B. Zupanic, R. Karba, S. Blažič (Editor), *Proceedings of the 6th EUROSIM Congress on Modeling and Simulation*, Vol. 2. Full Papers (CD). (MO-3-P4-2: pp. 1-6). University of Ljubljana, Ljubljana, 2007. Available: http://www.physiome.cz/references/eurosim2007stod.pdf.

[47] Wallish, P., Lusignan, M., Benayoun, M., Baker, T. I., Dickey, A. S., Hatsopoulos, N. G., *MATLAB for Neuroscientists: An Introduction to Scientific Computing in MATLAB.* Academic Press, Burlington, MA, 2008.

[48] Thomas, R. S., Baconnier, P., Fontecave, J., Francoise, J., Guillaud, F., Hannaert, P., Hernandéz, A., Le Rolle, V., Maziére, P, Tahi, F., White R. J., „SAPHIR: a physiome core model of body fluid homeostasis and blood pressure regulation“, *Philosophical Transactions of the Royal Society*, 366, pp. 3175-3197, 2008.

[49] Van Vliet, B.N., Montani J.P., „Circulation and fluid volume control“, in: *Integrative Physiology in the Proteomica and Post Genomics Age*, (ISBN 918-1-58829-315-2, 43-66), Humana Press, 2005.

Acknowledgement

Work on the development of medical simulators has been supported under the National Research Programme of Czech Republic, Project No. 2C06031and by Creative Connections s.r.o.