# Causality of System Dynamics Diagrams

Peter Junglas

PHWT Vechta Schlesier Str. 13, 49356 Diepholz, Deutschland; *peter@peter-junglas.de*

Simulation Notes Europe SNE 26(3), 2016, 147-154
DOI: 10.11128/sne.26.tn.10343
Received: September 3, 2015; Revised: March 20, 2016;
Accepted: April 15, 2016;

**Abstract.** System dynamics diagrams are generally regarded as a very simple modeling tool that can be implemented easily with standard techniques. But a few examples will show that this can be more complicated than expected: The causality – i.e. the assignment of block connections to inputs or outputs – can depend on the state of the complete system. How this affects the design of system dynamics libraries will be shown for the different modeling approaches used in Modelica and Simulink.

## Introduction

System dynamics diagrams are a modeling method that is used mainly for non-technical subjects like economy or ecology [1]. Commercial tools for modeling and simulation are readily available (e.g. Stella from ieee systems [2]), there even exists a free Modelica implementation [3].

In view of the basically very simple structure of the diagrams one should assume that they can be easily implemented, e.g. using Simulink to create corresponding blocks. Trying this one finds that the signal flow method can be inconvenient to reproduce certain details, because every connection of a block has to be defined beforehand as input or output, i.e. the causality of each connection is fixed. However in some examples the causality changes dynamically according to the current state of the system. For the implementation of this behaviour a modeling approach like "Physical Modeling" [4] seems to be better suited, because here the causality is dynamical in general and can only be determined in the context of the complete system.

In the following several examples are going to illustrate the basic problems, and implementations in Modelica and Simulink will show, how they can be solved. While the Simulink library is rudimentary and only serves as a proof of concept, the Modelica version is quite complete and can be useful on its own to allow for the simulation of system dynamics diagrams with physical modeling programs. It has been developped originally for a textbook [5] and is available under an open source licence from the author's homepage [6]. The already existing Modelica implementation by Cellier et al. [3] does not address the problems mentioned here, because it concentrates on the simulation of the famous world models, which are free of causality problems.

## 1 Basic System Dynamics Diagrams

System dynamics diagrams consist of three different types of basic building blocks: **Reservoirs** are storage elements representing state variables, which change their values – often called levels – through ingoing or outgoing flows. **Flows** work like valves and define the value of the flows, they connect reservoirs with external sources or sinks or other reservoirs. They can use the values of auxiliary variables that are computed with **converters**. Fig. 1 shows a simple diagram containing these blocks.

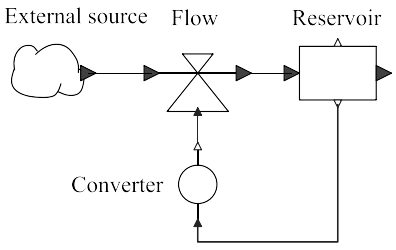


**Figure 1:** Basic blocks in system dynamics diagrams.

As an example we consider a simple model of population growth: The size of the population $N$ changes according to the number of births $B$ and deaths $D$ per time. $B$ is simply given by a constant rate $b$, whereas $D$

is limited by a fixed capacity $N_c$:

$$B = bN$$
$$D = dN \quad \text{with} \quad d = \frac{d_0}{1 - N/N_c}$$

In the complete model (cf. fig. 2) the parameters $b$, $d_0$ und $N_c$ are defined in converters, an additional converter uses them to compute the rate $d$. The flows multiply their two inputs to get the flow values $B$ and $D$. The diagram only shows the basic relations between the variables, the concrete formulas are hidden inside the blocks.

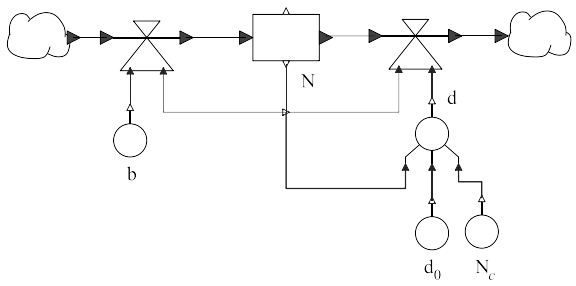


**Figure 2:** Model `population`.

The causality of the components is straightforward: Converters can have an arbitrary number of inputs, but only one output that is connected to other converters or the correspondig inputs of a flow. A flow block uses these inputs to compute the value of the flow and passes it as an output value to the connected reservoirs. The arrows denoting the flows in the diagram seem to contradict this view, but they only denote the (positive) direction of a flow, not the logical flow of the signal, which always proceeds from a flow to a reservoir. Finally a reservoir subtracts input and output flow and computes the value of its state variable by simple integration. This value is provided via explicit outputs, which can be used in converters or flows. This basic idea has been used in the Modelica library described in [3] and can be easily implemented in Simulink.

# 2 Models with Variable Causality

## 2.1 Stock with saturation

A reservoir has two optional parameters that define minimal and maximal level values. In the example model `sink` (cf. fig. 3) the first reservoir $S_1$ has a minimal value of 0 und a start value of 4, the outgoing flow is set to 0.5 by a constant converter. The result of the simulation can be seen in fig. 4: According to the constant flow the level of $S_1$ decreases linearly with time, until at t = 8 the minimal value is reached, and stays constant thereafter. The subsequent reservoir $S_2$ has the corresponding behaviour, especially it stays constant after t = 8.

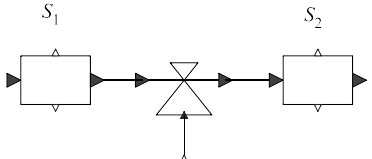


**Figure 3:** Model `sink`.

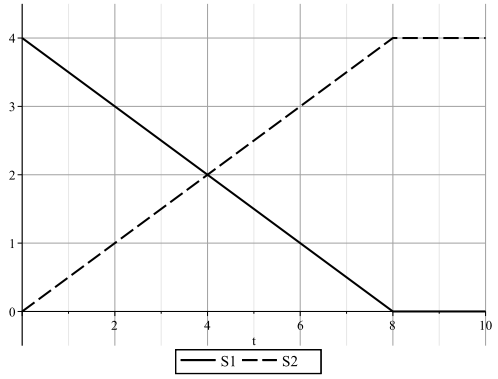


**Figure 4:** Simulation result of `sink`.

At first sight it seems that the model can be implemented easily by adding a saturation into the reservoir $S_1$. This doesn't affect the behaviour of $S_2$ though, its level would rise steadily after t = 8 according to the given flow value. Instead one has to guarantee that the outflow at $S_1$ and the corresponding inflow at $S_2$ change from 0.5 to 0 according to the level of $S_1$. This is a typical causality problem: Before t = 8 the size of the flow is defined by the flow block, afterwards it is reduced to 0 by the reservoir $S_1$. And the situation gets even more complicated, if $S_2$ has an upper limit smaller than 4: Now it is $S_2$ that has to change the flow value.

## 2.2 Simple conveyor belt model

The conveyor block models a simple conveyor belt, its input values appear at the output after a given delay.

It is a discrete element with a fixed sample time. In the test model `conveyor` (cf. fig. 5) a time varying input is transported by a conveyor and accumulated in a reservoir. Fig. 6 shows the result of the simulation, which comes as no surprise.

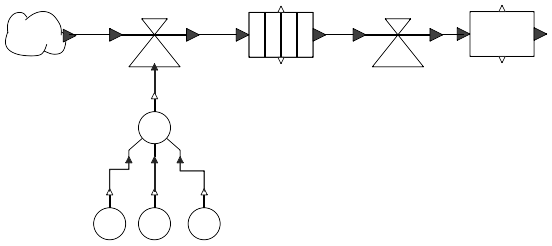

**Figure 5:** Model `conveyor`.

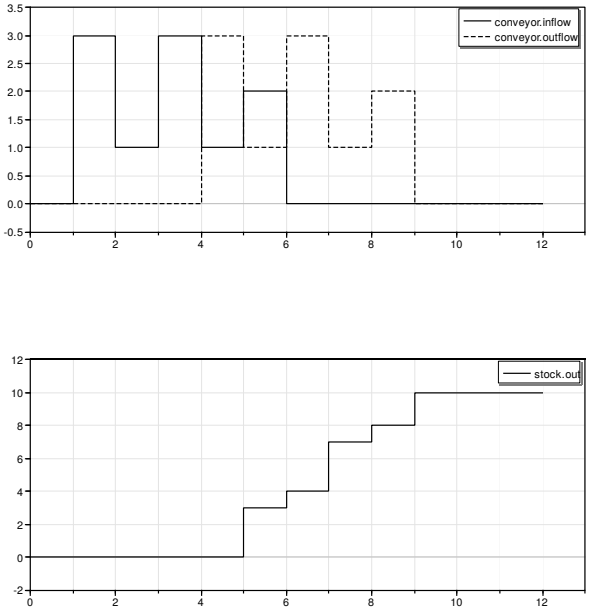


**Figure 6:** Simulation result of `conveyor`.

The interesting point here is the causality of the outflow: It is completely defined by the conveyor block and simply transported to the following reservoir. The value given by the flow element is disregarded completely. In this example the causality is not changing dynamically, but it has the "wrong" direction – at least compared to the standard situation defined in section 1.

## 2.3 Modeling a simple manufacturing machine with `Oven`

A particularly clear-cut example is the `Oven`, a discrete model for a generic manufacturing machine. It has the three parameters `initialLoad`, `capacity` and `cookingTime` and behaves like a baking tray: Initially it is loaded according to the input flow, until its capacity is reached. Subsequently the cooking time starts, at the end of which the complete content is forwarded to the output flow. The model `oven1` (cf. fig. 7) shows the basic behaviour of the component, using the parameter values `capacity = 3` and `cookingTime = 2`. The input flow has the constant value 2, the output flow the value 1.

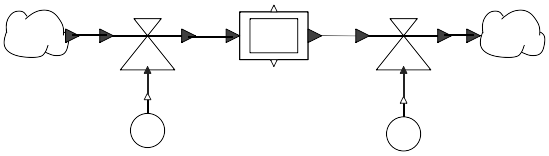

**Figure 7:** Model `oven1`.

The plot in fig. 8 shows the input and output flow in the upper part and the load of the `oven` in the lower part. At time t = 2 two incoming parts are stored in the `oven`, in the next step only one. At t = 3 the capacity is reached, the processing begins. Afterwards at t = 5 the complete content of three elements is released, while at the same time the next two parts arrive at the input. The concrete timing behaviour, especially the overlapping of output and input, is a matter of definition and is modelled here after the corresponding blocks in the Stella environment.

The size of the input flow depends in a complicated way on the preceding flow element and the state of the `oven`: During the loading phase the value is determined by the flow, until the capacity is reached. The input then is given as the minimum of the input flow and the remaining space in the `oven`. During the processing time the `oven` sets the input to zero. The output flow is defined by the `oven` alone: During loading and processing it is zero and rises to the full value of `capacity` only during a short discharging phase. As in the `conveyor` example the value of the successive flow element is ignored completely.

The situation gets even more complicated if one extends the model `oven1` by reservoirs $S_1$ and $S_2$ at the input and output of the `oven`: When $S_1$ is going to run empty – and has a minimal value of zero –, its output

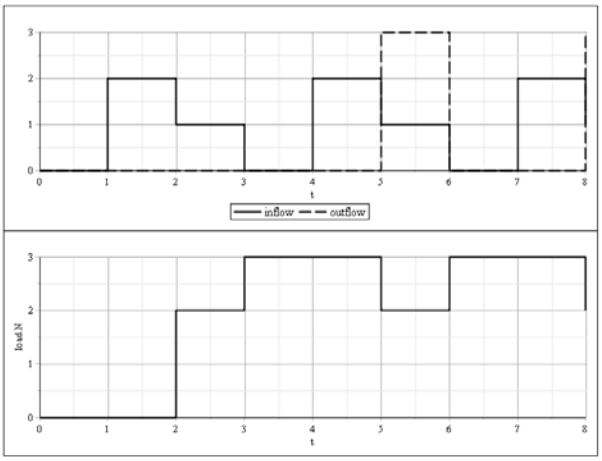


**Figure 8:** Simulation result of `oven1`.

flow is calculated by its last level, the (maximal) size defined by the flow element and the current state of the `oven`. Fig. 9 shows a typical simulation result for such a case.

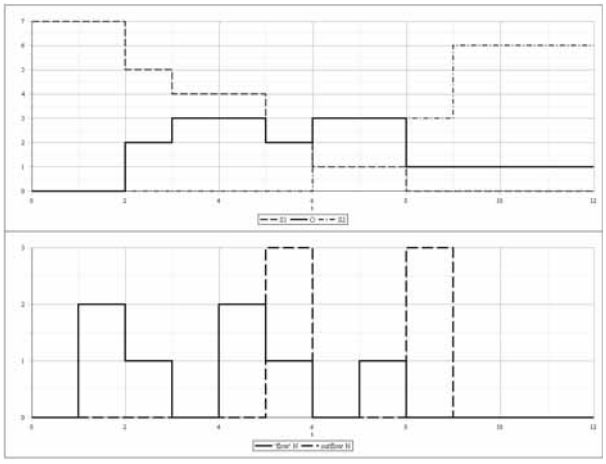

**Figure 9:** Simulation result of `oven2`.

And if the output reservoir $S_2$ has an upper saturation limit, the simulation may fail: The `oven` wants to get rid of its content, but the output storage has no room for it. Obviously this is a situation to avoid not only in the simulation.

# 3 Implementation of a System Dynamics Library in Modelica

## 3.1 Design of Modelica libraries

Physical modeling environments based on Modelica use a completely different approach to the problem of causality [4]: It is not necessary to determine, which quantities can be defined as input variables and used to compute the values of output variables. Instead one only specifies the relevant equations for a component, without solving each of them for a variable. Additional equations are generated automatically using the connections between the components. For this purpose one distinguishes two types of quantities: Flow variables add together to zero at connection points – they are often the time derivatives of conserved quantities – , whereas potential variables meeting at a connection have the same value.

How to create a simulation program out of the resulting system of equations – generally a DAE system –, is a difficult problem, but has been solved in many cases of practical interest [7]. The corresponding algorithms are implemented in Modelica based simulation programs like Dymola or MapleSim.

The high flexibility that has been reached in Modelica, comes with a price especially for library builders: Since the description of the blocks does not state explicitely where a quantity is computed, it is not guaranteed automatically, that an arbitrary combination of blocks and connections leads to a closed system, having the same number of equations and variables. A remedy for this problem has been presented in [8]: One defines an equal number of flow and potential variables at each connection point (**connector**) and provides each block with as many equations as it has external flow variables. In addition one can augment a block with "common" signal lines that are explicitely designated as input or output. In this case one needs an extra equation in the block for each output variable. Models that adhere to these conditions are called "balanced". Provably they contain an equal number of variables and equations.

## 3.2 Conception of the connector

These considerations will now be applied to the construction of a system dynamics library in Modelica. Starting point is the definition of a suitable connector `MassPort` that contains the size of the flow as

Modelica flow variable dm, corresponding to its integral, the state variable m. The following two questions have to be solved now:

1. Which quantity can be used as potential variable that accompanies dm?
2. How can the equations be distributed between reservoirs and flows?

Motivated by the basic systematics of system dynamics models that has been introduced in section 1 the integration of the flow shall be done inside a reservoir, which contains the following equation:

```
der(m) = inflow.dm +
outflow.dm;
```

This formulation employs the usual sign convention in Modelica: A flow variable is positiv, when it flows into the block.

Basically a flow block now computes the value of dm using its input quantities. But as section 2 has made clear, it has to take into account the levels of the adjacent reservoirs. Therefore reservoirs have to send the necessary information as a real value data that adopts the role of the potential variable. Considering the examples above there are three different possibilities, how the value of data can be used:

1. not at all, the reservoir simply accepts any value given by the flow,
2. the flow is set to data,
3. the flow is limited by data.

The first case complies with the "standard" causality. The second case corresponds to the situation at the outflow of the conveyor or oven, the third to a reservoir with saturation or the loading of the oven.

This behaviour could be implemented by setting data = 0 in the first case and using the sign of data to distinguish between the other two cases. But this idea has two drawbacks: There is no easy way to implement additional uses of data that could come up in future extensions, and the test for zero with real variables is a bad idea anyway. For that reason the connector will be extended by an integer variable info, which is used to indicate the corresponding case. This variable has a fixed causality: It is always computed by a reservoir and used inside a flow block. Therefore we need two variants of the connector, where info is designated as output or input respectively:

```
connector MassPortR
  "mass port of reservoirs"
  flow Real dm;
  Real data;
  output Integer info;
end MassPortR;

connector MassPortF
  "mass port of flows"
  flow Real dm;
  Real data;
  input Integer info;
end MassPortF;
```

### 3.3 Structure of the system dynamics library

After these preliminary considerations the further construction of a system dynamics library is straightforward. It consists of the following four subpackages:

- Interfaces
- Reservoirs
- Converters
- Flows

As usual the definition of the connectors, base classes and auxiliary functions are collected in Interfaces. In particular it contains the function constrainedRate that combines the external input value of a flow with the data and info variables of its two MassPorts to compute the actual flow value. This calculation is included in the base class GenericFlow and inherited by all flow components.

The subpackage Reservoirs contains the standard elements Stock and SaturatedStock together with CloudSource and CloudSink, which represent external sources or sinks. Discrete components are the by now well-known Oven and Conveyor supplemented by discrete versions StockD and SaturatedStockD.

All components in the Flows subpackage have two MassPorts to connect to surrounding reservoirs. The basic Flow has a signal input that defines the flow value in standard situations. Additionally the library provides variants with several inputs to implement commonly used simple equations and some elements for discrete simulations.

The elements in Converters exclusively have signal connections, they are defined in Modelica as

`block`, which means that they have a fixed causality. Programs specialised to system dynamics modeling usually have only one `converter` and one `flow` block. The actual relations can be defined as parameter values, the number of inputs is adapted automatically. Unfortunately this feature cannot be implemented using Modelica. For this reason the subpackages contain several components that implement the most common relations. A special feature are the two blocks `SwitchedConverter` and `TimeSwitchedConverter`, which switch between two inputs according to a control input or the simulation time, and the `GraphConverter`, which implements a function by linear interpolation between table values that are read from a file. Blocks containing arbitrary relations can be created easily by inheriting from a proper base block and adding a few lines of Modelica code.

# 4 System Dynamics Diagrams in Simulink

In the signal flow method connections have a fixed causality, which can not change dynamically according to the state of the system. Of course this doesn't imply that one cannot implement models like the examples above, but one has to take care of the causality problems explicitely. In the following some example models in Simulink will show how this can be achieved. A simpler, but less systematic implementation is described in [5].

## 4.1 Modeling of continuous blocks

As has been described already in section 1 the "standard" situation has a fixed causality. Therefore it is easy to construct corresponding blocks for reservoirs, flows and converters in Simulink and create models like the `population` example (cf. fig. 10). No external cloud blocks have been included, since they don't represent any equations anyhow.

Unfortunately it is not possible to make the model look more like a system dynamics diagram due to a fundamental restriction of Simulink blocks: All input signals are attached to one side of a block, all output signals to the opposite side. The "upwards" orientation of the flows makes the distribution of lines a bit more pleasant, but in larger examples it is hard to avoid a hay-wire circuitry. The situation gets even worse by the additional lines that are necessary to cope with the dynamical causality in the following examples.

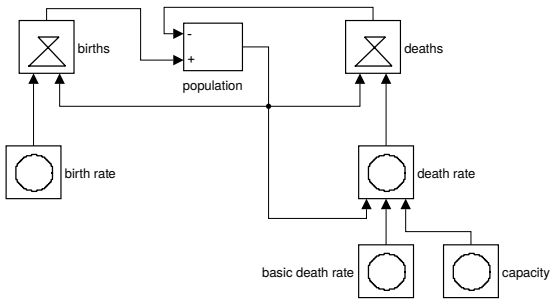


**Figure 10:** Model `population` in Simulink.

For the implementation of a reservoir with saturation we can resort to the ideas used in Modelica before: A reservoir gets two additional outputs to signal the following flow block, when it is empty, and the preceeding flow block, when it is full. A flow block has two corresponding inputs that are connected to the surrounding reservoirs.

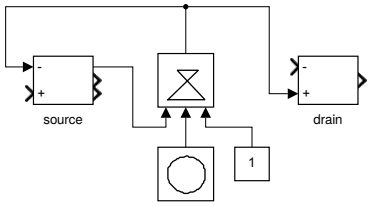


**Figure 11:** Model `sink` in Simulink.

Fig. 11 shows how this idea is used in the model `sink`. For simplicity the drain is realised as a simple stock without saturation. The corresponding flow input has the constant value 1 to indicate that the flow can be delivered, otherwise it would be 0. This makes the implementation of the flow block very simple: It just multiplies its three inputs.

## 4.2 Modeling of discrete blocks

In a discrete model a reservoir with saturation behaves differently than in the continuous case: Due to the fixed time step the inflow can be limited by the space available or the outflow by the current level. Therefore the two outputs that correspond to the data value of the Modelica connector will provide the maximal and minimal values possible instead of simply 1 or 0 as in the continuous case.

The discrete version of the flow block has to be changed accordingly: Instead of just multiplying its

three inputs, it now takes their minimal value. If a connected reservoir has no saturation the unconnected data input of the flow needs a constant value of `Inf` (i.e. infinity) instead of 1. With these modifications a Simulink version of the `oven2` example with a limited reservoir at the input looks like fig. 12.

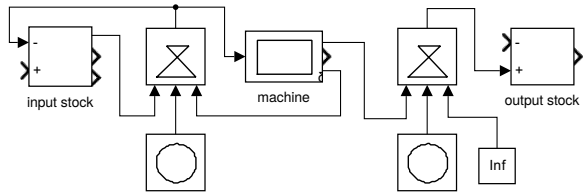


**Figure 12:** Modell `oven2` in Simulink.

But there is another feature still missing: The oven defines the output flow, irrespectively of the value proposed by the following flow block. One could implement this behaviour by adding another signal from the reservoir to the flow mimicking the info value in Modelica, but this would clutter the diagram with even more lines. Instead the flow block gets a boolean parameter `useFlow` that is set to `false` manually, if the flow is preceded by an `oven` or a `conveyor`. Fig. 13 shows the complete implementation of the discrete flow block. Now the model `oven2` reproduces the results of the equivalent Modelica example exactly.

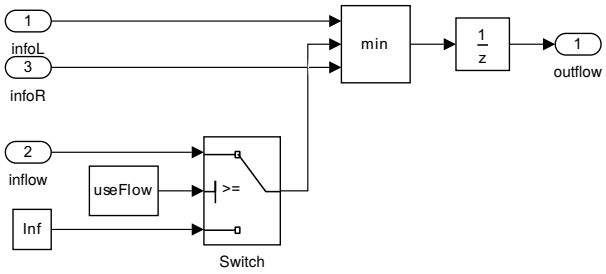


**Figure 13:** Implementation of the discrete `flow` block in Simulink.

The implementation of the `oven` component itself is cumbersome but straightforward, it uses three `UnitDelay` blocks representing internal variables and mimics the Modelica code completely. Alternatively one could use an S-function to program the `oven` directly in Matlab.

In contrast the `conveyor` is completely trivial, it just consists of an `Integer Delay` block. But it shows an interesting difference to its Modelica counterpart: To achieve a three step delay one sets the internal parameter to three (obviously), but in the Modelica case, which uses an array together with the `pre` operator and a `while sample()` construct, one has to set the parameter to four to get the same result. Apparently the detailed timing of an event is handled differently in Modelica and Simulink.

Though all example models have been successfully implemented in Simulink, the results lack the simplicity and flexibility of the Modelica version. This is only partly due to the dynamic causality, but mainly – and trivially – to the restrictive placement rules for connections on Simulink blocks.

## 5 Conclusions

The development of a flexible system dynamics library is much easier using the dynamic causality of physical modeling environments. Nevertheless it is possible to mimic it completely in Simulink using a larger number of signal lines between the blocks. Reversing the argument one could define the flow and potential variables in the `MassPort` connectors with a fixed causality, since `dm` is always computed in a flow, `data` in a reservoir. This shows that the idea of "dynamical causality" in system dynamics diagrams is mainly a matter of convenience and depends on the definition of "one connection".

In modeling courses a presentation of the ideas behind the two different implementations will clarify the notion of causality and broaden the modeling skills of the students. An interesting point here, which will need further clarification, is the different behaviour of the `pre` operator in Modelica and the 1/z block in Simulink.

Compared to dedicated system dynamics environments users of the Modelica library have to cope with limitations of their tools. A main point is the missing of the feature to input formulas directly as parameters. Even if components for the most common relations are provided and more can be created by a few lines of Modelica, the typical user of system dynamics software has little intention to write explicit code. In any case the Modelica library presented here is a simple and cheap replacement for specialised tools - at least for teaching purposes.

## References


[1] Hannon B, Ruth M. *Dynamic Modeling.* Springer, New York, 2nd edition, 2001.

[2] Richmond B, Peterson S, Vescuso P. *An Academic User's Guide to STELLA.* High Performance Systems, Inc., Lyme, N.H., 1987.

[3] Cellier FE. World3 in Modelica: Creating System Dynamics Models in the Modelica Framework. In *Proceedings of the 6th International Modelica Conference*, Bielefeld, Germany,2008; p. 393 – 400.

[4] Fritzson PA. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3.* Wiley & Sons, New York, 2015.

[5] Junglas P. *Praxis der Simulationstechnik.* Europa-Lehrmittel, Haan-Gruiten, 2014.

[6] Junglas P. *System dynamics library in Modelica* [Online]. [cited 2015 July 15]; Available from: http://www.peter-junglas.de/fh/simulation/systemdynamics.html

[7] Cellier FE, Kofman E. *Continuous System Simulation.* Springer, New York, 2010.

[8] Olsson H, Otter M, Mattsson SE, Elmqvist H. Balanced Models in Modelica 3.0 for Increased Model Quality. In *Proceedings of the 6th International Modelica Conference*, Bielefeld, Germany, 2008; p. 21 –33.