FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Bond Graph Model Bibliothek für PowerRPDEVS

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Aleksander Grzymek
Matrikelnummer 01428243

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.Dr. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Franz Josef Preyser, BSc.

Wien, 23. November 2018

Aleksander Grzymek   Wolfgang Kastner

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Bond Graph Model Library for PowerRPDEVS

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Aleksander Grzymek
Registration Number 01428243

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof.Dr. Wolfgang Kastner
Assistance: Dipl.-Ing. Franz Josef Preyser, BSc.

Vienna, 23rd November, 2018

Aleksander Grzymek	Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Aleksander Grzymek
Weyringergasse 20/12, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. November 2018

Aleksander Grzymek

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich während des Studiums und der Anfertigung dieser Arbeit unterstüzt haben.

Zuerst möchte ich mich bei Prof. Wolfgang Kastner bedanken, für die Möglichkeit und die Begutachtung dieser Bachelorarbeit.

Ebenfalls möchte ich mich bei Dipl.Ing. Franz Preyser bedanken für das spannende Thema, die Unterstützung, zahlreiche Hinweise und inhaltliche Aufsicht.

Abschließend möchte ich mich bei meinen Eltern und meiner Familie bedanken für die Unterstützung vor und während dem Studium.

Acknowledgements

I would like to express my deep gratitude to Prof. Wolfgang Kastner for offering to me the possibility to write this bachelor thesis.

I would also like to express my very great appreciation to Dipl.Ing. Franz Preyser for the topic idea, constructive and valuable suggestions during the research and tireless corrections of my thesis.

Finally, I would like to thank my parents and further family for supporting me before and during the studies.

Kurzfassung

Revised Parallel DEVS (RPDEVS) ist ein Formalismus für ereignisorientierte Modellierung, der vor kurzem an der Technischen Universität Wien entwickelt wurde. Dabei handelt es sich um eine Modifikation der Parallel DEVS (PDEVS), welche wiederum aus dem ürsprünglichen klassischen Discrete Event System Specification (DEVS) Formalismus entstanden ist. Um Modelle in dem neuen Modellierungsformalismus RPDEVS simulieren zu können war es notwendig eine neue Simulationsengine zu entwickeln. Auf Basis eines klassischen DEVS-Simulators *PowerDEVS* wurde eine neue RPDEVS Engine programmiert. Der resultierende RPDEVS Simulator wird nun als PowerRPDEVS bezeichnet.

Aufgrund der abgeänderten Simulationsengine sind die ursprünglichen PowerDEVS Modell-Bibliotheken mit PowerRPDEVS nicht mehr kompatibel, das heißt es müssen neue Modell-Bibliotheken entwickelt werden. Das Ziel dieser Bachelorarbeit ist eine Bond Graph Bibliothek inklusive Kausalisierungsalgorithmus für PowerRPDEVS zu entwickeln. Dadurch soll es möglich sein, akausale Bond Graph Modelle im Simulator einzugeben. Die Bibliothek verbindet also ereignisdiskrete mit akausaler Modellierung.

Abstract

Revised Parallel DEVS (RPDEVS) is a discrete event modelling formalism that was recently developed at the TU Wien. It is a modification of the Parallel DEVS (PDEVS) which was derived from the classic Discrete Event System Specification (DEVS) formalism. For the new modelling formalism it was necessary to write a new simulation engine. Based on the classic DEVS simulator *PowerDEVS* a new RPDEVS engine was written. The resulting simulator is called PowerRPDEVS.

However, due to the different modelling formalisms, the original PowerDEVS model libraries cannot be re-used and therefore, new model libraries have to be developed. The goal of this bachelor thesis is to create and present a library for modelling and simulating bond graphs in the PowerRPDEVS environment together with a causalisation algorithm. Thereby it should be possible to enter acausal bond graph models in the simulator. Thus the library connects discrete event and acausal modelling.

Contents

CHAPTER 1

Introduction

1.1 Motivation

As described in [Pre15], the Discrete Event System Specification (DEVS) and also Parallel DEVS (PDEVS) formalisms do not support the modelling of 'true' mealy behaviour. Furthermore the way concurrent input messages are processed in DEVS and PDEVS forces the model developer to take into consideration all possible processing orders in order to make an atomic component independent from its connected components. To solve this problem, in [PHK18] Preyser et al. describe a derivative PDEVS formalism and call it Revised Parallel DEVS (RPDEVS). For the new formalism a simulator based on a DEVS simulator PowerDEVS[1] was developed and called PowerRPDEVS[2]. However the libraries from the PowerDEVS are not compatible with the RPDEVS formalism and it was necessary to write new ones.

There already are bond graph libraries for the DEVS formalism, like for example the one described by D'Abreu and Weiner in 2003 [DW03], however the bond graph models they used, were already causalised. Using the RPDEVS formalism, we can facilitate the new lambda-iteration (discussed in 2.1) to causalise the graph on the fly, determining the signals in an iterative way for a point in time.

1.2 State of the Art

Bond graph simulation in DEVS like formalisms is not a very popular topic and it was not reviewed for a long time. One of the first tries to develop such a library was presented in 2001 by A. Naamane, N. Giambiasi and A. Damiba [NGD01]. They have written a modelling tool for simulating bond graphs in Generalised Discrete Event

[1]Project available at sourceforge.com
[2]Project available at sourceforge.com

Systems (GDEVS) with the basic bond graph elements. In 2003, D'Abreu and Weiner [DW03] presented another simulation tool to represent the bond graphs within DEVS formalism. They used the CD++ tool kit and built their library based on QDEVS (Quantised Discrete Event System) formalism. In 2005 [DW05], the same authors presented another version of their bond graph library. This time the approximation is based on the Quantized State System (QSS) method (discussed in section 2.3).

Bond graphs (briefly introduced in 2.2) to be valid must be causalisable. This means, there must exist a solution to the set of mathematical equation representing the bond graph. The process of causalisation determines in which order the equations need to be solved. All the bond graph libraries previously mentioned run models which are already causalised. The library presented in this work causalises bonds on the fly.

As this thesis is based on the RPDEVS formalism, it will be described in chapter 2. Furthermore, chapter 2 contains a brief description of the bond graph framework and of QSS - the numeric integration method used in PowerRPDEVS.

The developed library is designed for the PowerRPDEVS environment, which will be described in detail in section 3.1. The detailed description of the bond graph library and its components will be provided in section 3.2.

In chapter 4, we present simulation examples of the working bond graph library for PowerRPDEVS.

CHAPTER 2

Theoretical Background

2.1 RPDEVS

RPDEVS is a discrete event system formalism for discrete-, hybrid- and continuous system modelling. RPDEVS is an extension to the PDEVS formalism, as PDEVS do not support 'true' mealy behaviour [PHK18].

First, we begin with the ancestors of the RPDEVS. According to [CZ94], the DEVS formalism was introduced in the early 70' and allows the modellers to hierarchically structure their models. The original DEVS model according to [ZKP00] is a tuple:

$$M =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$$

where
X: a set of external events,
S: a set of states,
Y: a set of output values,
δ_{int}: $S \rightarrow S$: the internal state function,
δ_{ext}: $Q \times X \rightarrow S$: the external transition function, where $Q = \{(s, e)|s \in S, 0 \leq e \leq ta(s)\}$ and e is the time elapsed since the last state transition,
$\lambda : S \rightarrow Y$: the output function,
$ta : S \rightarrow R^{+}_{0,\infty}$ the time advance function (where $R^{+}_{0,\infty}$ is the set of non negative real numbers).

A DEVS model updates its state via transition functions according to its current state and input. The model reacts to an input running δ_{ext}. Without any input, the model may have its internal transition which first leads to a call of the output function λ possibly producing output followed by an internal state transition δ_{int}. The state may only be changed in the state transitions δ_{int} and δ_{ext}.

PDEVS was introduced in [CZ94] as an extension to DEVS formalism to deal with the problems revealed in the original DEVS. The main problem were collisions - when δ_{ext} and δ_{int} should be run at the same time. As a solution, a confluent transition function δ_{con} was added to the formalism. However, in PDEVS it is still not possible to model 'true' mealy behaviour, that is immediately reacting to an input event with an output event without any state transition in between. In [PHRK16], an example of a model using a switch element was presented and shown the unintended behaviour of the PDEVS formalism. This led to a new extension to PDEVS, namely RPDEVS described in [PHK18]. RPDEVS defines only one transition function δ which combines all three transition functions from PDEVS. RPDEVS requires the input to stabilize before state transitions are run. Therefore the λ function is allowed to read the input and λ is run repetitively recalculating and resending the output until the input stabilizes (remains the same in two consecutive iterations). The λ function is also allowed to save its state, called lambda-state. Later, in the δ step the lambda-state maybe read and used for the transition. The models in RPDEVS are made out of *couplings* and *atomics* in a hierarchical structure. An atomic in RPDEVS is defined as a 6-tuple[PHK18]:

$$\langle X, S, Y, \delta, \lambda, ta \rangle,$$

where
X is the set of possible inputs,
S is the set of possible states, $S_\delta \subseteq S$, $S_\lambda \subseteq S$,
Y is the set of possible outputs,
$Q = \{(s,e) \,|s \in S, e \in [0, ta\,(s)]\}$,
$\delta : S_\delta \times S_\lambda \times [0,\infty) \times X \to S_\delta$*generic state transition function*,
$\lambda : S_\delta \times [0,\infty) \times X \to Y \times S_\lambda$ *output function*,
ta is the *time advance function.*

A coupled RPDEVS N is defined equally to the definition of PDEVS coupled N [PHK18] and therefore

$$\langle X_N^b, Y_N, D, \{M_d\}_{d\in D}, \{I_d\}_{d\in D_N}, \{Z_{i,d}\}_{i,d\in D_N} \rangle.$$

where
X_N^b is the set of possible inputs,
Y_N is the set of possible outputs,
D is the set of components,
$\{M_d\}$ is a component,
$\{I_d\}_{d\in D_N}$ is the influencees of d,
$\{Z_{i,d}\}_{i,d\in D_N}$ is an i-to-d output translation function,
For each d in D

$$M_d = \langle X_d, S_d, Y_d, \delta_d, \lambda_d, ta_d \rangle,$$

is a RPDEVS structure.

The figure 2.1 presents a simplified schema of RPDEVS simulation.

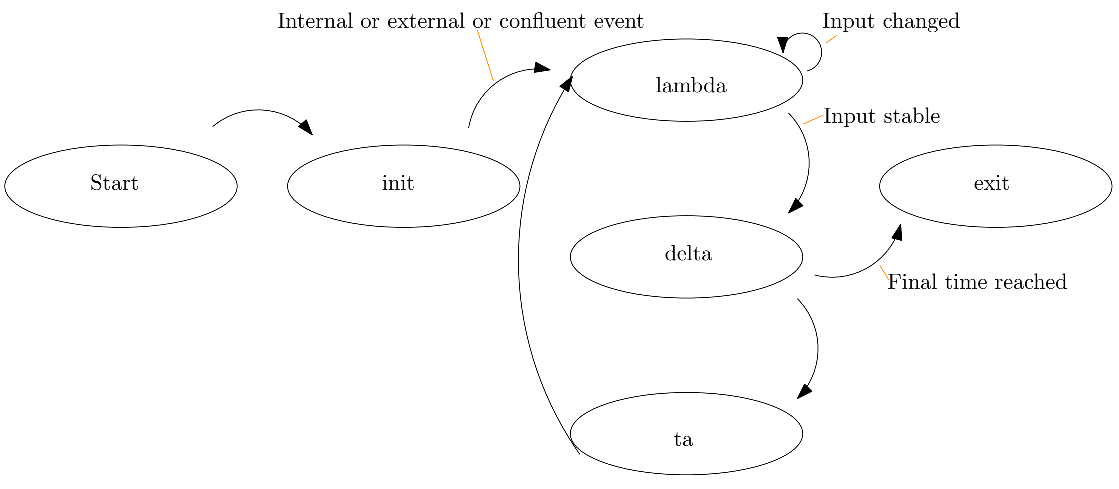


Figure 2.1: RPDEVS simulation sketch

2.2 Bond Graph

The bond graph framework generalizes the constraints of the physics laws of many domains (mechanical, electrical, hydraulic for example). In a bond graph, physical model is built out of components connected by (power) bonds, where the components can exchange power. The power flowing in a bond is the product of the effort (denoted e) and the flow (f). Every element of a bond graph can determine either the effort or the flow of a (power) bond. The bonds are directed and the half-arrow represent the direction of the power however the information between the elements are exchanged in both directions. An example of a bond graph is depicted in the figure 2.2b.

There are nine basic Bond Graph components, furthermore they are domain independent. We introduce them here briefly:

2.2.1 Source of Effort (S_e)

The source of effort just sends the effort on the bond. It has only one outgoing port. The element on the other side of the bond is supposed to provide the flow for the bond to be causalised. In case of PowerRPDEVS the effort sent by S_e may have any signal form like for example: constant signal, an impulse, a pulsing source or a sinusoidal signal source.

2.2.2 Source of Flow (S_f)

The source of flow works analogous to the source of effort. It also has one outgoing port. The S_f sends the flow on the bond and expects the component on the other side of the bond to compute the effort.

2.2.3 Resistor (R)

The resistor has one ingoing port. The resistor is a passive element and it responds to the given effort or flow with flow or effort respectively. The computation of the retuning signal is based on the incoming signal and the resistance parameter R. Following equations hold for the resistor:

$$f = \frac{e}{R}$$
$$e = fR$$

2.2.4 Capacitance (C)

The capacitance (sometimes also called *capacitor*) is an effort storage element with one input port. The capacity is defined by the parameter C. For the given flow the capacitor returns the effort according to the rule:

$$e_C(\tau) = \frac{1}{C}\int f_C(\tau)d\tau$$

2.2.5 Inertance (I)

The inertance component (sometimes also called *inductor*) is an flow storage element with one input port. The inductor has its inertance parameter L. It expects to receive the effort and returns the flow according to the rule:

$$f_I(\tau) = \frac{1}{L}\int e_I(\tau)d\tau$$

2.2.6 Transformer (Tr)

The transformer is an element with one input and one output port. It forwards the given effort or flow according to the rules described below. The values given to a transformer will be multiplied with the transformation factor r and forwarded at the other side of the transformer. Following rules hold for the transformer:

$$e_2 = e_1 r$$
$$f_2 = f_1 \frac{1}{r}$$

2.2.7 Gyrator (Gy)

The gyrator is an element with one input and one output port. It is similar to the transformer however it converts the effort to flow and the flow to effort with an conversion factor r and then returns it at the other side of the gyrator. The conversion factor of the gyrator - *gyrator ratio* r is analogous to the transformation factor. Following equations hold for the gyrator:

$$f_2 = \frac{e_1}{r}$$
$$e_2 = f_1 r$$

2.2.8 Junctions

Junctions are used to connect the elements described above. While thinking about electrical circuits we can say, the 1-junction represents the serial connection of the elements and the 0-junction represents the parallel connection of the elements. The junctions may have arbitrary many input and arbitrary many output ports. We denote the number of input ports with $j \in \mathbb{N}_0$ and the number of output ports with $k \in \mathbb{N}_0$.

One-Junction (1-Junction)

The one-junction has to fulfil the following equations regarding the efforts and flows (we number the bonds of a junction from 1 to $n{=}j+k$ and denote the effort on the bond $1 \leq p \leq n$ as e_p and analogously for the flow):

$$f_1 = f_2 = \ldots = f_n$$
$$\sum_{m=0}^{i} e_m - \sum_{m=0}^{j} e_m = 0$$

In other words the flow is the same for all the bonds connected to a one-junction and the sum of all incoming efforts must be equal the sum of outgoing efforts. Therefore only one neighbour element may provide the flow to a one-junction or $n-1$ elements provide the effort for the one-junction.

Zero-Junction (0-Junction)

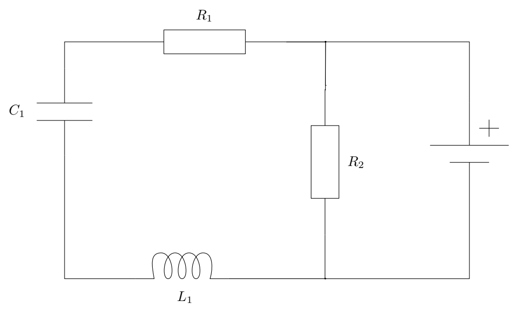
The zero-junction must fulfil rules analogous to the rules for the one-junction (we keep the variable notation):

$$e_1 = e_2 = \ldots = e_n$$
$$\sum_{m=0}^{i} f_m - \sum_{m=0}^{j} f_m = 0$$

In other words the effort is the same for all the bonds connected to a zero-junction and the sum of all incoming flows must be equal the sum of outgoing flows. Therefore only one neighbour element may provide the effort to a zero-junction or $n-1$ elements provide the flow for the zero-junction.

2.2.9 Bond Graph Modelling

Having a physical model, we want to model it as a bond graph. As shown in [ŠHČG12], there exists an iterative way to depict an electrical circuit as a bond graph. The figure 2.2a shows an example electrical circuit and the figure 2.2b shows its equivalent bond graph.



(a) A simple electrical circuit.

$$\begin{tikzcd} & R & R & \\ C & 1 \arrow[l, harpoon] \arrow[u, harpoon] \arrow[d, harpoon] & 0 \arrow[l, harpoon] \arrow[u, harpoon] & Se \arrow[l, harpoon] \\ & I & & \end{tikzcd}$$

(b) The equivalent bond graph.

Figure 2.2: A bond graph out of electrical circuit.

2.3 QSS

QSS, according to Preyser [Pre15], refers to the class of continuous signal description methods. These methods use numerical methods to describe continuous systems in a discrete event manner. To achieve this, the continuous signals are piecewise approximated by Taylor polynomials.

The order of the Taylor polynomial used for the approximation of the signal determines the order of the QSS method. Having QSS first order, we use a piecewise constant quantisation function. Having QSS second order we use the linear function to quantize the signal. In general the QSS method d uses Taylor polynomials of degree $d-1$ to discretize the continuous signal.

Model simulation often requires solving differential equations, which again requires computing integrals of functions. The integral of an Taylor polynomial $g(t)$ of degree d is a Taylor polynomial $G(t)$ of degree $d+1$. In QSS $G(t)$ will be again approximated by Taylor polynomials $h_i(t), i \in \mathbb{N}$ of degree d. The approximation will be updated whenever $|G(t) - h_i(t)| > q$, where q is the *quantum* parameter. The q parameter drives the precision of the numerical method.

The figure 2.3 presents $f(t) = G(t) - h_i(t)$. To know, when the approximation function has to be updated, we are looking for the point in time, when $f(t)$ exceeds q or $-q$. In the figure 2.3 it is the point t_0. From this point in time $G(t)$ will be approximated with some $h_{i+1}(t)$

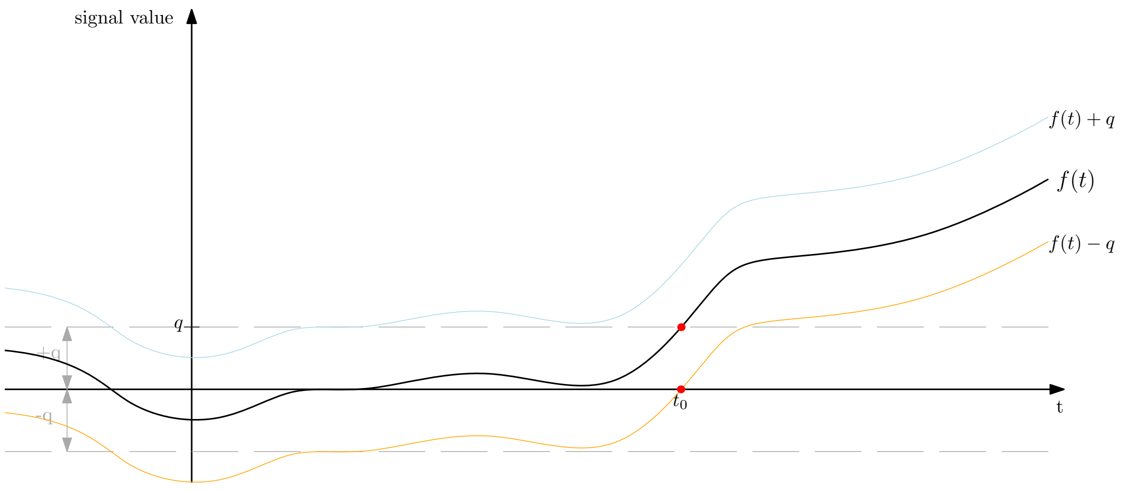


Figure 2.3: QSS: an example of signal difference $f(t)$

In PowerRPDEVS (described later in section 3.1) the maximum of QSS-order 4 is supported, as there exist algebraic methods to compute zero-crossings of a 4^{th} degree polynomial function.

In the paper Kofman and Junco [KJ01] mention, that the discrete event simulation may have some advantages over the discrete time simulation, considering the simulation speed and the possibility of distributed simulation. Also discrete time systems may be simulated by discrete event systems.

CHAPTER 3

PowerRPDEVS Bond Graph Library

3.1 PowerRPDEVS

PowerRPDEVS[1] is a simulation engine with a graphical user interface, supporting modelling within RPDEVS formalism. The project was started as a fork from the PowerDEVS project [2] however, the PowerDEVS engine only supports the DEVS formalism. Most of the code is written in the C++ programming language, using the Qt[3] framework for the graphical user interface.

3.1.1 Overview

PowerRPDEVS offers a comprehensive toolset for modelling and simulation. The main window of the program offers a sketch area to draw the graphs using drag-and-drop method, a set of tabs with graph components to choose, and two toolbars leading to other functionalities. The main window is depicted on figure 3.1.

[1]Project available at: https://sourceforge.net/projects/powerrpdevs/
[2]Available on sourceforge.com
[3]Available at https://www.qt.io/

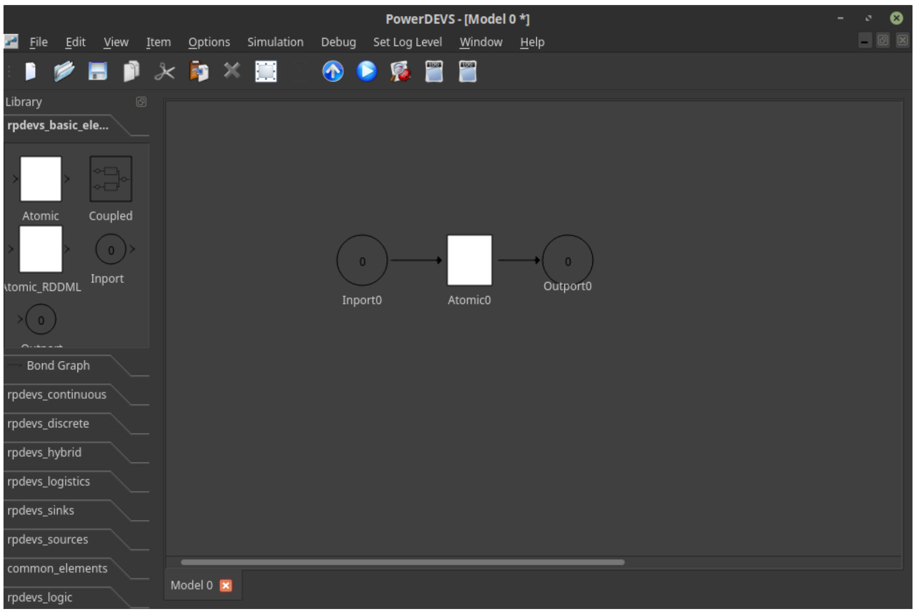


Figure 3.1: The main graph editor of PowerRPDEVS

The models in the PowerRPDEVS are built out of couplings which again consist of couplings or atomics. Every PowerRPDEVS atomic represents specific behaviour during the simulation and may communicate with other atomics or couplings through the input and output ports. An atomic may be for example a signal generator, periodically sending a signal of a specified value.

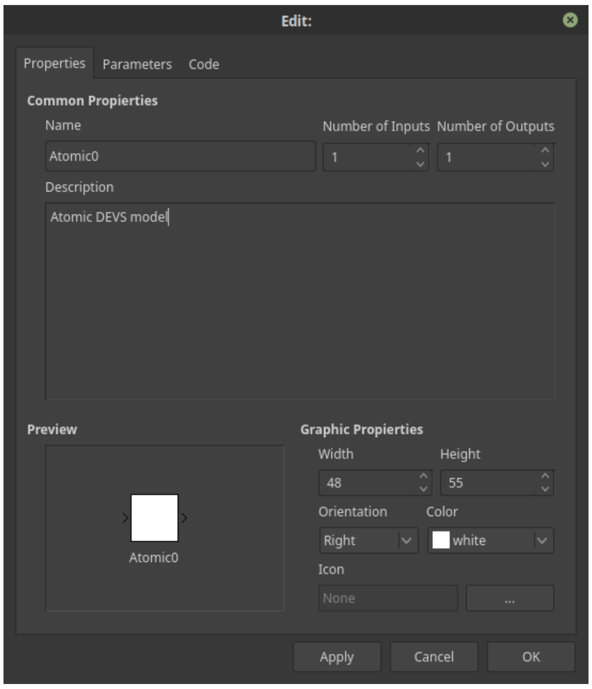
In the code, every PowerRPDEVS atomic is a C++ class extending the predefined *Simulator* class. Additionally, atomics may have their own parameters and functions. The RPDEVS formalism requires atomics to extend the Simulator class and therefore to have following methods implemented:

- init - where the atomic is prepared for the simulation. In most cases, the initial values for the atomic, such as parameters, will be read from the graphical user interface,
- ta - the time advance function which says how to advance the time for this specific atomic,
- delta - where the actual δ state transition function according to the RPDEVS formalism will be coded,
- lambda - where the λ output function from the RPDEVS formalism should be implemented,
- exit - defines actions taken at this atomic at the end of the simulation.

Code examples will be provided and discussed in detail later in section 3.2.

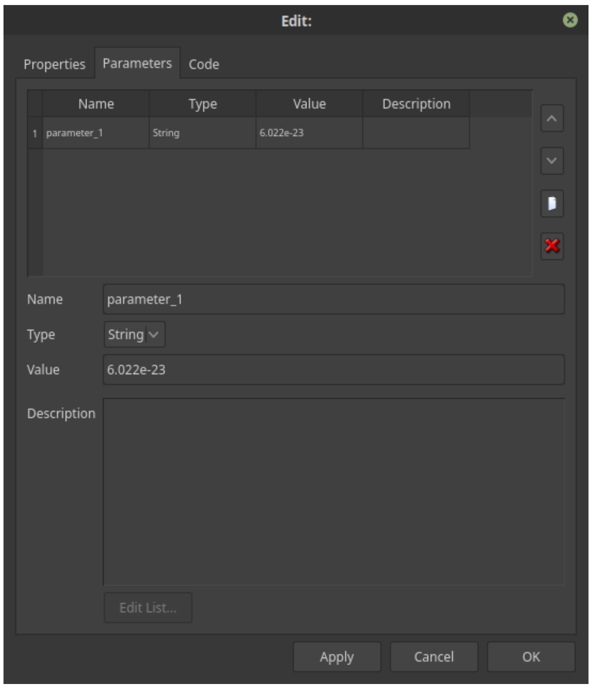
Atomics are organized into libraries (visible on the left side of the graphical user interface, in figure 3.1). The graphical interface of PowerRPDEVS provides tools to create, edit, load, remove and sort libraries. The libraries provided in the master branch of the project are designed to group the atomics by their purpose. For now, there are following libraries available: basic elements, continuous, discrete, hybrid, logistics, sinks, sources, common elements, logic, bond graph.

PowerRPDEVS provides an interface to fully manage atomics. With a right click on an atomic and the 'edit' command we get a window, where we can change every detail about the specific atomic such as: name, number of input ports, number of output ports, dimensions, background, background icon, description, parameters and code source (depicted in figures 3.2 and 3.3). Of course changes to the atomics code and parameters need to be done carefully, the components are ready to use for the user and it should only be necessary to set the ports and the parameters' values.



Name: Atomic0
Number of Inputs: 1
Number of Outputs: 1
Description: Atomic DEVS model
Width: 48
Height: 55
Orientation: Right
Color: white
Icon: None

Figure 3.2: The atomic editor of PowerRPDEVS

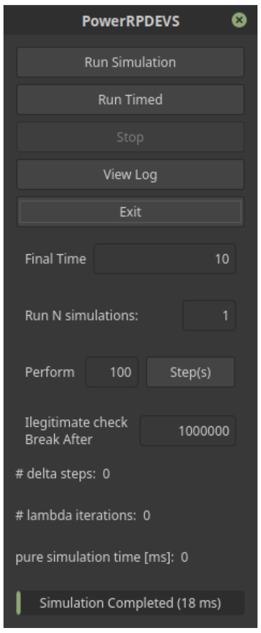


	Name	Type	Value	Description
1	parameter_1	String	6.022e-23	

Name: parameter_1
Type: String
Value: 6.022e-23
Description:

Figure 3.3: The atomic editor parameter tab

Apart from basic commands, almost every editor supports (open file, new file, etc.), there are simulation specific commands available in PowerRPDEVS. When clicking on 'Simulate' or hitting F5 the model consisting of couplings and atomics will be compiled in the background and prepared to run. After a successful compilation, a window to control the simulation like depicted in figure 3.4 shows up. It offers a possibility to run the complete simulation, let it run a defined number of steps, run it for a defined time, exit the simulation or show the log file.



Final Time 10

Run N simulations: 1

Perform 100 Step(s)

Ilegitimate check Break After 1000000

delta steps: 0

lambda iterations: 0

pure simulation time [ms]: 0

Simulation Completed (18 ms)

Figure 3.4: The PowerRPDEVS simulation settings

The log file contains information about the simulation run. It may be a valuable source of information while designing and implementing a new atomic or after a simulation stopped unexpectedly. What information is written to the log file is defined by the *log level* variable and the coded log expressions used in the single atomics. With a drop-down menu "Set Log Level" we can easily choose the log level for the next simulation (it requires recompilation of the code). There are six log levels:

- _ERROR - here only information about real errors should be written. For example, when the simulation aborts,
- _WARNING - at this level the information about possibly wrong behaviour of atomics should be given,
- _INFO - at this level the information about the normal progress of the simulation should be given,
- _DEBUG_ROUGH - here a brief information, useful for the debugging should be given,
- _DEBUG - should provide more information useful for debugging,
- _DEBUG_HARD - the highest logging level. It also provides the debug information from the simulation engine.

It should be taken into consideration that writing into a file is a quite time consuming operation (compared to numerical computations for example) and therefore, for faster simulations, the log level should be kept as low as possible.

In case when the log file is not a sufficient debugging help, a dedicated debugging tool may be used to analyse the simulation process step by step. At the moment, two debugging tools for Linux operating systems are supported: KDbg[4] and Nemiver [5].

Sometimes it is also necessary to recompile all the classes, also those where code was not directly changed. It may be the case, when we change an enumerate class which other classes use. If we do not recompile all classes, we will get wrong values for this enumeration. To deal with this problem, there is a special button which erases all the files compiled for the models and the simulation engine. All of them will be recompiled while starting a new simulation.

In case we start a simulation and errors occurred during the compilation process, an error dialog box pops up with a direct link to the compile.log file for further information about the cause of the failure.

Coupled elements may be opened revealing their inner structure and parameters may be set for them. It is not necessary to to change the source code to use these parameters, but they can be referenced in the graphical user interface. It is possible to set the atomic value as a reference to the coupling parameter in the form of "%parameter_name". It should be automatically recognized by the PowerRPDEVS and the value should be shown in a grey non-editable field.

During the simulation in the PowerRPDEVS, atomics are called in a specified order. First, atomics without any input port are triggered because they can only send information and therefore affect the output of connected blocks. Then the blocks with input and output ports are called and at the end, the atomics with input ports only are triggered. Additionally, it is possible to manage the priority order of the atomics within a coupling. This will not affect the outcome of the simulation however, it may speed up the computation as the input messages may stabilize faster.

3.1.2 Simulation Engine

The simulation engine is responsible for the correct simulation, according to the RPDEVS formalism. It creates the connections between the atomics or couplings and calls corresponding functions at the right moment. Everything begins with the init function of every coupling and every atomic within every coupling. Then the lambda function of every element is run, in the order as described earlier. Later lambda iterations, delta steps and *ta* are run alternating until the end of the simulation (see figure 2.1).

[4] Available at http://www.kdbg.org/

[5] Available at https://developer.gnome.org/nemiver/0.9/nemiver.html

To communicate with each other, the atomics and the couplings send messages through their output ports. These ports are connected to the input ports of some other atomics. Splitted connections (one-to-many) are also possible.

In the PowerRPDEVS, there is a special DEVSMessage base class. Only messages of this class or classes that are DEVSMessage derivatives may be sent in the simulation engine. One of these subclasses of DEVSMessage is *QSSDoubleArray*. This class has a container for double values to represent a QSS-signal. The numbers in this container are coefficients of the polynomial approximating the signal. QSSDoubleArray supports many mathematical operations on polynomials and also many mathematical operators are overloaded.

The base class DEVSMessage has a member variables `index` which is intended to enable vectorial communication. However, the `index` of a message can also be used to define different types of messages at a communication line. Every index has its special meaning defined internally for the atomics in a library. An experienced user may also mix atomics across libraries.

After a model in the graphical user interface is built, it may be compiled and run on the RPDEVS engine. PowerRPDEVS will automatically compile necessary files and create background code of the model to run. Having successfully compiled code, the user may start the simulation or adapt some simulation settings like the end time for example (depicted on figure 3.4).

3.2 Bond Graph Model Library

The bond graph library for PowerRPDEVS we recently created, is a collection of atomic components used to build a bond graph. It supports all the basic elements of bond graphs and there is also a sensor component added, to read the values of the effort and the flow exchanged on the bonds. The following elements were implemented as PowerRPDEVS atomics (in parenthesis we name their corresponding elements of electrical circuits for easier imagination):

Resistance R (resistor) - has only one input port, immediately responds to the messages in lambda function with the value computed according to its resistance parameter,

Capacitor C (capacitor) - has only one input port, sends the effort value and expects the flow value to compute its integral,

Inductance I (coil) - has only one input port, sends the flow value and expects the effort value to compute its integral,

Source of effort Se (constant voltage) - has only one output port, sends the effort value specified as parameter,

Source of flow Sf (constant current) - has only one output port, sends the flow value specified as parameter,

Sensor - has one input and one output port. It reads the values of the effort and the flow messages floating on a specified bond and forwards these values on the output port. There is a parameter to set, whether only effort, only flow or both values should be forwarded. The sensor needs to be connected to a connection between two bond elements (splitted connection),

Transformer Tr (transformer) - has one input and one output port. It represents the behaviour of a transformer. It has two parameters specifying the number of windings on the primary (n_1) and on the secondary side (n_2). These two values are necessary to compute the transformation ratio $r = \frac{n_1}{n_2}$.

Gyrator Gy (gyrator) - has one input and one output port. It represents the behaviour of a gyrator.

One Junction (1-junction) (serial connection of elements) - can have any number of input and any number of output ports. It represents the serial connection of elements.

Zero Junction (0-junction) (parallel connection of elements) - can have any number of input and any number of output ports. It represents the parallel connection of elements.

The detailed description of all components from bond graph library can be found in section 3.2.3.

3.2.1 Power Flow

As we already know from section 2.2, there is a power flow in every bond. One side of the bond determines the effort and the other one determines the flow. PowerRPDEVS supports directed messages only and bond graphs are undirected graphs, so to keep the graph design clean and easy for the user we automatically create additional *back connections* in the background while running the simulation. The back connections are like normal PowerRPDEVS connections (from a specified element on a specified port to another specified element on another specified port) however invisible for the user. While creating these connections the bond graph atomics are filling their *bond connection maps* with information. These maps contain information about neighbour atomics connected and the input and output ports used to communicate with them. We designed a C++ class to hold this bond connection information:

```
bond_connection::bond_connection() {
  id = -1;
  bond_element_type = bond_component_type::undefined;
  in_port = -1;
  out_port = -1;
  value = QSSDoubleArray(0.0);
}
```

```
std::map<int, bond_connection > bond_connection_map;
```

The index of the map is the input port, so every time we get a message at a certain bond, we know where to send the answer.

To create the back connections, we have written a distributed algorithm running partly in the init function and partly in the lambda function, for all elements where necessary.

The algorithm is represented with the following pseudo code 3.1:

The following drawing 3.5 shows the back connections (marked orange).

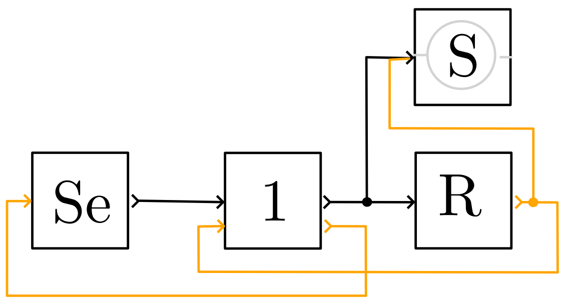


Figure 3.5: The back connections in the PowerRPDEVS bond graph (marked orange)

Algorithm 3.1: Add back connection

```
Input: Pointer to current bond graph component object $me$
Output: void
1 for every initial input port ip of me do
2     Get the RPDEVS atomic element $e$ and its output port $op$ where the
      connection to $ip$ comes from;
3     Add an output port $o$ at $me$;
4     Add an input port $i$ at $e$;
5     Connect $(me, o)$ to $(e, i)$;
6     for every additional connection starting at atomic $e$, output port $op$, going to
      atomic $f$, input port $ip$ do
7         Connect $(me, o)$ to $(f, ip)$;
8     end
9 end
```

3.2.2 Indices

As we already discussed in section 3.1.2 every message has an index (a non-negative integer) attached to it. In the bond graph library, these indices are facilitated to distinguish different types of messages. For better readability, we created an enumeration for the indices for the bond graph library (see code example below).

```cpp
enum class bond_index_type : int{
  effort, flow, value, ctype, Se, Sf, effort_equiv, flow_equiv,
  ressistance_equiv
};
```

The meanings of the particular indices is the following:

effort is used to send the effort values,

flow is used to send the flow values,

value is used to send an atomic specific value or property like for example the resistance of an resistor, which is a necessary information for a junction (if connected directly),

ctype is used to send messages with the element's type to its neighbours,

Se is used by junctions to send thesum of flow values of all flow-dominant components connected to them (if exists),

Sf is used by junctions to send the sum of effort values of all effort-dominant components connected to them (if exists), *effort_equiv* is used to send the equivalent effort value by junctions during causalisation (discussed later in section 3.2.3). phase when the junctions exchange their equivalent circuit information,

flow_equiv analogous to effort_equiv,

resistance_equiv analogous to effort_equiv.

3.2.3 Library Components

Now, we will describe the components of the PowerRPDEVS bond graph library in detail. Some parts of the code are similar for all the components so we will begin with a simple one and discuss only the differences to the other components. For all the elements, we will first discuss the init function, then the lambda function, and then the delta function.

The time advance function *ta* is common for all component of the bond graph library. This function always returns the σ value. σ is the time to the next δ step at the corresponding element. Additionally, debug information will be written to the log file according to the log level set.

```
double sensor::ta(double t) {
 printLogAtLevel(_DEBUG_ROUGH, "%s, %x, t=%G: ta \n", name, myself,
     t);
 return (sigma);
}
```

The exit function is also common for all the bond graph library components and it just writes an entry to the log file.

```
void sensor::exit() {
 printLogAtLevel(_DEBUG_ROUGH, "%s id=%x: exit \n", name, myself);
}
```

Sensor

The simplest atomic element of our bond graph library is the sensor. It expects effort and flow messages on its input port and forwards these messages according to the rule set. It may be only the effort, only the flow or both values. If the sensor gets a message on another index, a warning will be written to the log file.

In the init function, we read the parameters from the graphical user interface and initialize the state variables.

```
void sensor::init(double t,...) {
 va_list parameters;
 va_start(parameters, t);
 type = va_arg(parameters,char*);
```

```

  if (strcmp(type,"Effort")==0) {
    forward_flow = false;
    forward_effort = true;
  } else if(strcmp(type,"Flow")==0) {
    forward_flow = true;
    forward_effort = false;
  } else {
    forward_flow = true;
    forward_effort = true;
  }

  sigma = 0;
}
```

In the λ function we iterate through all input ports and collect the input messages that arrived. Having an effort or flow message, we forward it according to the variables initialized in the init function.

```
void sensor::lambda(double t) {

  if (!input_bag_empty()) {
    DEVSMessage *in_msg;

    unsigned int in_port = 0;
    while (pop_input(&in_msg, in_port)) {

      QSSDoubleArray *qm = ((QSSDoubleArray*)in_msg);

      switch(qm->index) {

        case static_cast<int>(bond_index_type::effort): {
          if(forward_effort) {
            add_output(qm, 0);
          }
          break;
        }
        case static_cast<int>(bond_index_type::flow): {
          if(forward_flow) {
            add_output(qm, 0);
          }
          break;
        }
        default: {
          break;
        }
      }
    }
  }
}
```

The sensor never changes its state and therefore the only thing we do in the δ function is to set sigma to infinity.

```
void sensor::delta(double t) {
  sigma = INF;
}
```

Resistor

The Resistor is a passive element and it responds immediately to effort or flow values arriving at the input. The resistor has its resistance parameter R, which is in this PowerRPDEVS bond graph library version only a static real number, however while designing the library we also considered a temperature driven resistors whose value would change in time. For this case, there are already parameter fields for the QSS-order and the quantum value prepared. Or we could also allow the resistor to have an input port and be controlled by another element.

The response to effort e or flow f is computed according to the following rules (Ohm's laws for electrical circuits): when we receive the effort, we compute the flow with the function:

$$f = \frac{e}{R}$$

when we get flow, the computed effort is

$$e = f \cdot R$$

In the init function, we read the parameters from the graphical user interface and as the resistor has one input port we need to run our back connection algorithm (see 3.1).

```
void resistor::init(double t, ...) {
  va_list parameters;
  va_start(parameters, t);

  std::istringstream arguments (va_arg(parameters,char*));
  arguments >> qss_order;
  arguments.clear();
  arguments.str(va_arg(parameters,char*));
  arguments.seekg(0);
  arguments >> quantum;
  arguments.clear();
  arguments.str(va_arg(parameters,char*));
  arguments.seekg(0);
  arguments >> resistance;

  // resistor has only one input and one output port, forming its one
     single bond interface:
  bond_connection_map[0]=bond_connection((Port)0);
  bond_connection_map[0].add_return_connection(this);

  causality = causality_type::not_yet_causalised;

  initially = true;
  sigma = 0;
}
```

In the lambda function, if we run it for the first time, we need to check the input ports. It may be the case, that some other element created input ports for us while running the back connection algorithm. If this is the case, we need to update the information about the bond connections. Then, we also send out value information (in this case the resistance parameter) because this information was not available to other elements before we finished our init function. This is done by the *send_introduction_and_value_message_at_all_output_ports* function. After this, the initialization phase for the resistor is finished.

```
if(initially) {
  // determine bond output ports to new, externally created input
     ports
  for(Port iport=n_in_ports_initial; iport < n_in_ports; iport++) {
    // search for connection with sink=myself and sinkPort = iport
    bond_connection_map[iport].in_port = iport;
    if(false ==
        bond_connection_map[iport].get_output_port_and_id(this)) {
      std::stringstream str_str;
      str_str << "ERROR in block "<< name << ",
          'bond_connection::get_output_port_and_id(Simulator *me)
          returned false for input port "<< iport << "! \n";
      throwException(str_str.str());
    }
  }
  // introduce to neighbours:
  send_introduction_and_value_message_at_all_output_ports(this,
  bond_component_type::R, QSSDoubleArray(resistance));
}
initially = false;
```

Next, we check the incoming messages. First we check whether neighbour components sent their component type. These are the messages with index ctype.

```
for(Port iport=0; iport < n_in_ports; iport++) {
  if (pop_input(iport, static_cast<int>(bond_index_type::ctype),
      &in_msg)) {
    QSSDoubleArray *qm = ((QSSDoubleArray*)in_msg);
    bond_connection_map[in_port].id = (int)((*qm)[0]);
    bond_connection_map[in_port].bond_element_type =
        static_cast<bond_component_type>((*qm)[1]);
  }
}
```

Later, we check for messages with effort, flow and value indices. We respond to the effort and flow messages according to the rules described above. Messages on other indices are ignored and a warning is written to the log file.

```
while (pop_input(&in_msg, in_port)) { // fetch all messages that
    arrived at port port_number. This is the only in port for source
    of effort

  QSSDoubleArray *qm = ((QSSDoubleArray*)in_msg); //convert the
      message to QSSDoubleArray because we caanot user pop_input on
      QSSDouble Array, but only on DEVSMessage

  switch(in_msg->index) {
    case static_cast<int>(bond_index_type::value): { // receive
        introduction message
      bond_connection_map[in_port].value = (*qm);
      break;
    }
    case static_cast<int>(bond_index_type::effort): {
      QSSDoubleArray out_msg = QSSDoubleArray((*qm));
      out_msg = out_msg/resistance;
      out_msg.index = static_cast<int>(bond_index_type::flow);

      add_output(&out_msg, bond_connection_map[in_port].out_port,
          static_cast<int>(bond_index_type::flow));
      break;
    }
    case static_cast<int>(bond_index_type::flow): {
      QSSDoubleArray out_msg = QSSDoubleArray((*qm));
      out_msg = out_msg*resistance;
      out_msg.index = static_cast<int>(bond_index_type::effort);
      add_output(&out_msg, bond_connection_map[in_port].out_port,
          static_cast<int>(bond_index_type::effort));
      break;
    }
    default: {
      break;
    }
  }
}
```

As the resistor has only a static resistance value for now, the state will never change and therefore the delta function just sets sigma to infinity (cf. above code example).

Capacitance

The capacitance element is also a passive element storing the effort of the power flowing into it. Capacitance in our library may have an initial effort (we may have a loaded capacitor in an electrical circuit) and the capacitance parameter C. Also there are QSS-order and quantum to control the QSS method used while running the simulation. Capacitance expects to receive the flow and computes its effort $e_C(t)$ according to the rule:

$$e_C(t) = e_C(t_0) + \frac{1}{C}\int_{t_0}^{t} f_C(\tau)d\tau \tag{3.1}$$

where:
t_0 is the current time,
$f_C(\tau)$ is the last received flow value.

Since, using QSS, the flow $f_C(\tau)$ is a polynomial, we can solve the integral algebraically, again resulting in a polynomial for $e_C(t)$ (however, with a degree that is the flow's degree incremented by one).

The init function of the capacitance is almost the same like for the resistor - it reads the parameters and creates back connections. However, there is an additional part where the capacitance sets its initial value and its effort.

```
initial_value = QSSDoubleArray(initial_val);
initial_value.t_exp = t;
effort =QSSDoubleArray(initial_value);
effort_output = effort;
```

The initial value is necessary to be saved in case a zero-junction has more than one capacitance or source of effort connected and needs to check whether they have all the same initial effort value, otherwise the simulation breaks with an error. The *effort_output* is a lambda state variable saving the last effort sent on the bond.

The lambda function of the capacitance is similar to the one of the resistor. It begins with the back connections check and, then introduces itself to other elements. Later, the messages are caught but only logged for debug purposes. It is more efficient to do the computation only once in the delta function because the lambda function may be run many times before the input bag stabilizes. A new part in the capacitance's lambda function is, the one where the effort message is sent.

```
if(t +(tn-t) == t) {
  effort_output_new = effort;
  effort_output_new.advance_time_to(t);
  effort_output_new.cutoff(qss_order);
  effort_output_new.index = static_cast<int>(bond_index_type::effort);
  add_output(&effort_output_new, 0);
}
```

This is always done when $tn = t$. In the code, we see $t + (tn - t) == t$. This is due to the arithmetic rounding errors that can occur in floating point arithmetic. The value of the effort at current time is computed, then the coefficients of the QSS polynomial that exceed the QSS order selected with the corresponding parameter are cut off, before it is sent at index effort.

The delta function is the only one, in which the state might be changed. Every time the delta function of the capacitance gets a new flow message, the effort has to be recomputed and saved as current state. Having a flow message we compute the effort according to equation 3.1. The QSS method is part of the simulation engine implemented with the class QSSDoubleArray. QSSDoubleArray provide the method `integrate` for integrating polynomials.

```
void capacitor::delta(double t) {
  initially = false;
  effort_output = effort_output_new;

  Port in_port;
  DEVSMessage *in_msg;

  while(pop_input(&in_msg, in_port)) {
    QSSDoubleArray *in_qss = (QSSDoubleArray*)in_msg;
    switch(in_qss->index) {

      case static_cast<int>(bond_index_type::flow): {
        flow = QSSDoubleArray(*in_qss);
        effort = effort.advance_time_to(t);

        double effort_now = effort.value_at(t);
        effort = flow;
        effort.integrate();
        effort = effort*(double)(1.0/capacity);
        effort = effort + effort_now;
        effort.t_exp = t;

        break;
      }
      default: {
        break;
      }
    }
  }

  ...
}
```

With the new value of the effort, we need to check, when the difference between the last effort value we sent and the newly computed one will exceed the quantum q. First, we compute the difference $f(t)$. Then, we compute $f(t) - q$ and look for the next zero crossing t_{0_1} using a function already provided by the QSSDoubleArray class - *time_to_next_root(t)*. Then, we add $2q$ to $f(t) - q$ to get $f(t) + q$. We look for the next zero crossing t_{0_2} of this function and then we take the minimum of t_{0_1} and t_{0_2} as the σ time when we need to send an updated state of the capacitance.

```
void capacitor::delta(double t) {

...

  if(effort.t_exp == t || effort_output.t_exp==t) {
    sigma = INF;
    QSSDoubleArray difference = QSSDoubleArray(effort -
        effort_output);
    difference = difference - quantum;
    sigma = difference.time_to_next_root(t);
    difference[0] += 2*quantum;
    Time t2 = difference.time_to_next_root(t);
    if(t2 < sigma) {
      sigma = t2;
    }
}
```

Inductance

The inductance element in our bond graph library is analogous to the capacitor element with effort and flow swapped. So the inductance has the parameters: initial flow, inductivity L, QSS-order, and quantum. It computes the flow for a given effort according to the rule:

$$f_I(t) = f_I(t_0) + \frac{1}{L}\int_{t_0}^{t} e_I(\tau)d\tau \tag{3.2}$$

where:
t_0 is the current time,
$e_I(\tau)$ is the last received effort value.

The code is also analogous.

Tranformer

The transformer is an element with one input and one output port. The values given to a transformer will be multiplied with the transformation factor r and forwarded at the other side of the transformer. Like in an electrical transformer, the factor r is computed according to the primary parameter n_1 and secondary parameter n_2. When speaking of an electrical transformer, the n_1 would be the number of turns in the primary winding and the n_2 would be the number of turns in the secondary winding. The factor r is the fraction $r = \frac{n_1}{n_2}$. Additionally, the transformer has also an efficiency parameter h (like in an electrical transformer there may be some power loss).

Like the resistor, the transformer responds to the messages immediately in the lambda function and the delta function only writes a log entry. There are five types of messages a transformer may respond to:

- effort message - the transformer sends to the other side an effort message e_1 with the transformed effort according to the rule: $e_1 = e_0 \cdot r \cdot h$
- flow message - the transformer sends to the other side a flow message f_1 with the transformed flow according to the rule: $f_1 = f_0 \cdot \frac{1}{r} \cdot h$
- equivalent effort message - the transformer sends to the other side an equivalent effort message with the transformed equivalent effort according to the rule: $e_{e1} = e_{e0} \cdot r \cdot h$
- equivalent flow message - the transformer sends to the other side an equivalent flow message with the transformed equivalent flow according to the rule: $f_{e1} = f_{e0} \cdot \frac{1}{r} \cdot h$
- equivalent resistance message - the transformer sends to the other side an equivalent resistance message with the transformed equivalent resistance according to the rule: $R_{e1} = R_{e0} \cdot r^2 \cdot h$

The code for the transformer is similar to the code for the resistor. The main difference is in the lambda function due to different functionality. The following code snippet from the transformer's lambda function represents the message handling of the transformer:

```
unsigned int in_port = 0;
while (pop_input(&in_msg, in_port)) {

 QSSDoubleArray *qm = ((QSSDoubleArray*)in_msg); /

 switch(in_msg->index) {
  case static_cast<int>(bond_index_type::value): {
   bond_connection_map[in_port].value = (*qm);
   break;
  }
  case static_cast<int>(bond_index_type::effort): {
   QSSDoubleArray out_msg;
   out_msg = QSSDoubleArray((*qm)*r*efficiency);
   out_msg.index = static_cast<int>(bond_index_type::effort);
   if(bond_connection_map.count( in_port )) {
    if(n_out_ports > in_port) {
          add_output(&out_msg, in_port,
              static_cast<int>(bond_index_type::effort));
        }
      }
      break;
    }
    case static_cast<int>(bond_index_type::flow): {
      QSSDoubleArray out_msg;
      out_msg = QSSDoubleArray(((*qm)/r)*efficiency);
      out_msg.index = static_cast<int>(bond_index_type::flow);
      if(bond_connection_map.count( in_port )) {
        if(n_out_ports > in_port) {

          add_output(&out_msg, in_port,
              static_cast<int>(bond_index_type::flow));
        }
      }
      break;
    }
    case static_cast<int>(bond_index_type::effort_equiv): {
      QSSDoubleArray out_msg;
      out_msg = QSSDoubleArray((*qm)*r*efficiency);
      out_msg.index = static_cast<int>(bond_index_type::effort_equiv);
      printLogAtLevel(_DEBUG_HARD, "%s t=%G, created response %s: \n",
         name, t, out_msg.toString().c_str());
      if(bond_connection_map.count( in_port )) {
        if(n_out_ports > in_port) {
          add_output(&out_msg, in_port,
              static_cast<int>(bond_index_type::effort_equiv));
        }
      }
      break;
    }
    case static_cast<int>(bond_index_type::flow_equiv): {
      QSSDoubleArray out_msg;
      out_msg = QSSDoubleArray(((*qm)/r)*efficiency);
      out_msg.index = static_cast<int>(bond_index_type::flow_equiv);
      if(bond_connection_map.count( in_port )) {
        if(n_out_ports > in_port) {
          add_output(&out_msg, in_port,
              static_cast<int>(bond_index_type::flow_equiv));
        }
      }
      break;
    }
    case static_cast<int>(bond_index_type::ressistance_equiv): {
      QSSDoubleArray out_msg;
      out_msg = QSSDoubleArray(((*qm)*r*r)*efficiency);
      out_msg.index =
        static_cast<int>(bond_index_type::ressistance_equiv);
      if(bond_connection_map.count( in_port )) {
        if(n_out_ports > in_port) {
          add_output(&out_msg, in_port,
              static_cast<int>(bond_index_type::ressistance_equiv));
        }
      }
      break;
    }
    default: {
      break;
    }
  }
}
```

Gyrator

The gyrator is an element with one input and one output port. It is similar to the transformer however it converts the effort to flow and the flow to effort with a conversion factor r and then sends it at the other side of the gyrator. The conversion factor of the gyrator - *gyrator ratio* r is analogous to the transformation factor of the transformer and is computed according to the primary parameter n_1 and secondary parameter n_2. The factor r is the fraction $r = \frac{n_1}{n_2}$.

Like the transformer, the gyrator responds to the messages immediately in the lambda function and the delta function only writes a log entry. There are five types of messages a gyrator may respond to:

- effort message - the gyrator sends to the other side a flow message f_1 calculated from the effort according to the rule: $f_1 = \frac{e_0}{r}$
- flow message - the gyrator sends to the other side an effort message e_1 calculated from the flow according to the rule: $e_1 = f_0 \cdot r$
- equivalent effort message - the gyrator sends to the other side an equivalent flow message with the conversion result: $f_{e1} = \frac{e_{e0}}{r}$
- equivalent flow message - the gyrator sends to the other side an equivalent effort message with the conversion result: $e_{e1} = f_{e0} \cdot r$
- equivalent resistance message - the gyrator sends to the other side an equivalent resistance message with the converted equivalent resistance according to the rule: $R_{e1} = \frac{r^2}{R_{e0}}$

The code for the gyrator is similar to the code for the transformer. The main difference is in the lambda function due to different functionality. The following code snippet from the gyrators' lambda function represents the message handling of the gyrator:

```
unsigned int in_port = 0;
while (pop_input(&in_msg, in_port)) {

  QSSDoubleArray *qm = ((QSSDoubleArray*)in_msg);

  //the flow does not matter for source of effort

  switch(in_msg->index) {
    case static_cast<int>(bond_index_type::value): {
      bond_connection_map[in_port].value = (*qm);
      break;
    }
    case static_cast<int>(bond_index_type::effort): {
      QSSDoubleArray out_msg;
      out_msg = QSSDoubleArray((*qm)/r*efficiency);
      out_msg.index = static_cast<int>(bond_index_type::flow);
      if(bond_connection_map.count( in_port )) {
        if(n_out_ports > in_port) {
          add_output(&out_msg, in_port,
              static_cast<int>(bond_index_type::flow));
        }
      }
      break;
    }
    case static_cast<int>(bond_index_type::flow): {
      QSSDoubleArray out_msg;
      out_msg = QSSDoubleArray(((*qm)*r)*efficiency);
      out_msg.index = static_cast<int>(bond_index_type::effort);
      if(bond_connection_map.count( in_port )) {
        if(n_out_ports > in_port) {
          add_output(&out_msg, in_port,
              static_cast<int>(bond_index_type::effort));
        }
      }
      break;
    }
    case static_cast<int>(bond_index_type::effort_equiv): {
      QSSDoubleArray out_msg;
      out_msg = QSSDoubleArray((*qm)/r*efficiency);
      out_msg.index = static_cast<int>(bond_index_type::flow_equiv);
      if(bond_connection_map.count( in_port )) {
        if(n_out_ports > in_port) {
          add_output(&out_msg, in_port,
              static_cast<int>(bond_index_type::flow_equiv));
        }
      }
      break;
    }
    case static_cast<int>(bond_index_type::flow_equiv): {
      QSSDoubleArray out_msg;
      out_msg = QSSDoubleArray(((*qm)*r)*efficiency);
      out_msg.index = static_cast<int>(bond_index_type::effort_equiv);
      if(bond_connection_map.count( in_port )) {
        if(n_out_ports > in_port) {
          add_output(&out_msg, in_port,
              static_cast<int>(bond_index_type::effort_equiv));
        }
      }
      break;
    }
    case static_cast<int>(bond_index_type::ressistance_equiv): {
      QSSDoubleArray out_msg;
      QSSDoubleArray r2 = QSSDoubleArray(r*r);
      out_msg = QSSDoubleArray((r2/(*qm))*efficiency);
      out_msg.index =
          static_cast<int>(bond_index_type::ressistance_equiv);
      if(bond_connection_map.count( in_port )) {
        if(n_out_ports > in_port) {
          add_output(&out_msg, in_port,
              static_cast<int>(bond_index_type::ressistance_equiv));
        }
      }
      break;
    }
    default: {
      break;
    }
  }
}
```

Source of effort

The source of effort is a simple element. It only sends the effort every time the lambda function is called. In the init function, the effort parameter is read and saved. The delta function sets the initialization variable to false and sigma to infinity, because the value of the source of effort is constant. However, we have an idea to expand this element in the future and allow it to have an input port and therefore to drive the effort with an input signal.

Source of flow

Source of flow behaves analogous to the source of effort and the code is also analogous using flow instead of effort.

1-Junction

The junctions are the most complicated components in the bond graph library. They may have an arbitrary number of input ports and also an arbitrary number of output ports. When compared to the electrical circuit, junctions define how the components are connected with each other.

The 1-Junction represents the serial connection of the elements. We know the flow in the serial connected elements is the same for all of them and the sum of incoming and outgoing efforts must be zero. So having $n \in \mathbb{N}$ bonds where $j \in \mathbb{N}_0$ are incoming and $k \in \mathbb{N}_0$ are outgoing, the one junction must preserve:

$$f_1 = f_2 = f_3 = \cdots = f_n \tag{3.3}$$

and

$$\sum_{i=1}^{n} e_i = 0 \tag{3.4}$$

or

$$\sum_{i=1}^{j} e_i = -\sum_{i=1}^{k} e_i \tag{3.5}$$

Causalisation As stated before, we causalise the bond graph on the fly and the main part of this process is done by the junctions. We have elements defining effort (source of effort, capacitor), we have elements specifying flow (source of flow, inductor), but we also have elements without specified causality like the resistor, junctions, transformer and gyrator. Now, we describe the causalisation process at the 1-junction.

A component x, which delivers flow to the 1-junction automatically causalises the junction and we can forward the flow to all other elements connected to the 1-junction expecting them to send their effort. Then, the sum of efforts of all of the elements but x is computed and the remaining effort is sent to the x component, so it can update its flow.

```
if(this->causality == causality_type::flow || send_flow) {
 //send flow to every node but the source node;
 for (std::map<int, bond_connection >::iterator
    it=bond_connection_map.begin(); it!=bond_connection_map.end();
    ++it) {
  bond_connection bc = (bond_connection )(it->second);
  if( bc.out_port != -1 && flow_in_port != bc.in_port) {
   QSSDoubleArray flow_ret = QSSDoubleArray(flow);
   flow_ret.index = static_cast<int> (bond_index_type::flow);
   add_output(&flow_ret, bc.out_port);
  }
 }
}
```

In case a 1-junction has more than one flow determining element with different values, an exception is raised, and the simulation breaks.

```
int sources_of_flow =
   bond_neighbour_type[bond_component_type::Sf].size();
int inductions = bond_neighbour_type[bond_component_type::L].size();

if(sources_of_flow > 1) {
 MsgException illegitimateModel("There are too many sources of flow
    connected to one-junciton. There may only be one source of flow
    connected to one-junction");
 throw illegitimateModel;

 break;
} else if(inductions > 0 && sources_of_flow > 0 ) {
 QSSDoubleArray sf_flow =
 bond_connection_map[
    bond_neighbour_type[bond_component_type::Sf][0]].value;
 int f = 0;
 for(; f<bond_neighbour_type[bond_component_type::L].size(); ++f) {
  bond_connection bc_tmp =
  bond_connection_map[bond_neighbour_type[bond_component_type::L][f]];
  inductor* ind = (inductor*)this->father->D[bc_tmp.id];
  if(!(ind->initial_value == sf_flow)) {
   MsgException illegitimateModel("Initial flow of Inductor does
      not match the initial value of source of effort. See log for
      more information.\n");
   throw illegitimateModel;
  }
 }
 break;

} else if(inductions > 1) {
 QSSDoubleArray c0_effort =
    ((inductor*)this->father->D[bond_connection_map[
    bond_neighbour_type[
    bond_component_type::L][0]].id])->initial_value;
 int f = 1;
 for(; f<bond_neighbour_type[ bond_component_type::L].size(); ++f) {
  bond_connection bc_tmp =
  bond_connection_map[bond_neighbour_type[bond_component_type::L][f]];
  inductor* ind = (inductor*)this->father->D[bc_tmp.id];
  if(!(ind->initial_value == c0_effort)) {
   MsgException illegitimateModel("Initial flow of Inductors does
      not match. See log for more information.\n");
   throw illegitimateModel;
  }
 }
 break;
}
```

Unfortunately, it is not always the case that we have the flow provided. Sometimes all we know are just a few effort values. Assuming we are speaking of an causalisable bond graph, the 1-junction needs to find out other values and causalise itself. A simple case of a source of effort and two resistors is depicted in the figure 3.6. Now we only know the effort coming from the source of effort and no component is determining the flow. We have two uncausalised resistors. To resolve this problem, the 1-junction needs knowledge about the neighbourhood. In every lambda iteration it may receive messages with information about other components connected to it. The information will be saved in neighbourhood vectors and also the value of resistance sum will be computed:

$$R_{sum} = \sum_{i=1}^{r} R_i \tag{3.6}$$

Knowing this resistance sum, the 1-junction can compute the flow and send it to all of the components connected. This part of code computes the flow and sends the messages:

```
else if(send_computed_flow) {
 QSSDoubleArray flow_back = (effort_sum ) / ressistance_sum;
 flow_back.index = static_cast<int> (bond_index_type::flow);
 flow_back.t_exp = t;
 equiv_flow = QSSDoubleArray(flow_back);
 for (std::map<int, bond_connection >::iterator
     it=bond_connection_map.begin(); it!=bond_connection_map.end();
     ++it) {
   bond_connection bc = (bond_connection )(it->second);
   QSSDoubleArray flow_send = QSSDoubleArray(flow_back);
   add_output(&flow_send, bc.out_port );
 }
}
```

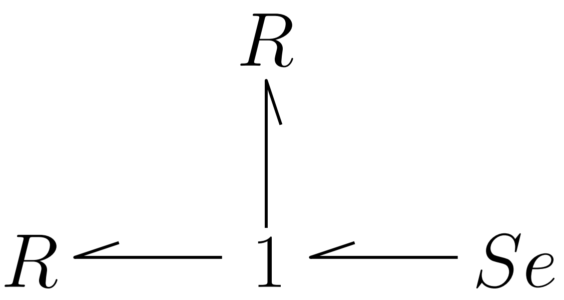


Figure 3.6: A simple bond graph with serial connection

Having only one resistor in this example, we would just compute the effort it should get, because there is only one element with unknown effort. Then the resistor will return us the flow. In the code we can see, that effort is sent to the only component with unknown effort.

```
if(send_effort) {
  QSSDoubleArray effort_back = QSSDoubleArray(effort_sum);
  effort_back.index = static_cast<int> (bond_index_type::effort);
  add_output(&effort_back, bond_connection_map[
      unknown_effort_elements.begin()->first].out_port );
}
```

The *effort_sum* variable is updated with every effort message arriving unless it is an effort computed by a resistor which got the computed flow earlier. The update is depicted in the following code:

```
if(effort_value_map.find(in_port) != effort_value_map.end()) {
  effort_sum = (effort_sum - (effort_value_map[in_port]) *
      effort_factor) ;
  effort_sum = effort_sum + ((QSSDoubleArray)(*qm) * effort_factor);
  effort_value_map[in_port] = QSSDoubleArray(*qm);
  if(known_effort_elements.size() == 0 && effort_from_junction ==
      false) {
    effort_from_junction = true;
  }

  for(int it2 = 0; it2 < known_equivalent_effort_elements.size();
      it2++) {
    if(known_equivalent_effort_elements[it2] == in_port) {
     missing_efforts = missing_efforts-1;
     unknown_effort_elements.erase(in_port);
     known_effort_elements.push_back(in_port);
     known_equivalent_effort_elements.erase(
         known_equivalent_effort_elements.begin()+it2);
    }
  }

} else {
  effort_sum = effort_sum + ((QSSDoubleArray)(*qm) * effort_factor);
  effort_value_map.insert(std::make_pair(in_port,
      QSSDoubleArray(*qm)));
  missing_efforts = missing_efforts-1;
  unknown_effort_elements.erase(in_port);
  known_effort_elements.push_back(in_port);
}
```

The 1-junctions also calculates the effective capacity C_{sum} and the effective inductance L_{sum}:

$$C_{sum} = (\sum_{i=1}^{c} \frac{1}{C_i})^{-1} \tag{3.7}$$

$$L_{sum} = \sum_{i=1}^{l} L_i \tag{3.8}$$

So we know, we can causalise a 1-junction with unknown values of effort/flow for resistors only, but when having a 1-junction connected to a 0-junction or transformer or gyrator, we do not have the information about the resistance. In this case, the junctions need to exchange the information about their equivalent values. Transformer and gyrator will only forward the appropriate value to the next junction as they should only be connected with junctions.

The 1-junction sends its equivalent flow and its equivalent resistance to the 0-junction and the 0-junction send its equivalent effort and its equivalent resistance to the 1-junction. The following code shows how a 1-junction is sending the equivalent values:

```
else if(send_equiv_flow) {
 if(known_equivalent_effort_elements.size() ==
     unknown_effort_elements.size() &&
     bond_neighbour_type[bond_component_type::Sf].size() +
     bond_neighbour_type[bond_component_type::L].size() == 0) {
   QSSDoubleArray flow_back = (effort_sum ) / equiv_ressistance;
   flow_back.index = static_cast<int> (bond_index_type::flow);
   flow_back.t_exp = t;
   equiv_flow = QSSDoubleArray(flow_back);

   //Send effort equivalent to all
   for (std::map<int, bond_connection >::iterator
       it=bond_connection_map.begin(); it!=bond_connection_map.end();
       ++it) {
     bond_connection bc = (bond_connection )(it->second);

     if(known_equivalent_effort_elements.size() > 0) {
       if(bond_connection_map[known_equivalent_effort_elements[0]].id
           == bc.id) {
         continue;
       }
     }

     QSSDoubleArray flow_send = QSSDoubleArray(flow_back);
     add_output(&flow_send, bc.out_port );
   }
  }
}
```

```cpp
if(send_equiv_ressistance) {
  for(unsigned int f = 0; f <
      bond_neighbour_type[bond_component_type::ZeroJunction].size();
      ++f) {
    bond_connection bc_tmp = bond_connection_map[
        bond_neighbour_type[bond_component_type::ZeroJunction][f]];
    QSSDoubleArray resistance_equiv = ressistance_sum;
    resistance_equiv.index = static_cast<int>
        (bond_index_type::ressistance_equiv);

    add_output(&resistance_equiv, bc_tmp.out_port );
  }
}
```

Zero-Junction

The 0-junction interpreted in an electrical circuit would be the parallel connection of the components. The 0-junction is analogous to the 1-junction, however the rules for the computations change.

For the 0-junction, the effort must be equal for all connected elements and the sum of all flows (incoming and outgoing) must be zero. So having $n \in \mathbb{N}$ bonds, where $j \in \mathbb{N}_0$ are incoming and $k \in \mathbb{N}_0$ are outgoing, the 0-junction must preserve:

$$e_1 = e_2 = e_3 = \cdots = e_n \tag{3.9}$$

and

$$\sum_{i=1}^{n} f_i = 0 \tag{3.10}$$

or

$$\sum_{i=1}^{j} f_i = -\sum_{i=1}^{k} f_i \tag{3.11}$$

To be causalised, the zero-junction needs exactly one component sending the effort or $n-1$ elements sending the flow.

The rules to effective resistance R_{sum}, the effective capacitance C_{sum}, and the effective inductance L_{sum} are also different compared to the 1-junction:

$$R_{sum} = (\sum_{i=1}^{r} \frac{1}{R_i})^{-1} \tag{3.12}$$

$$C_{sum} = \sum_{i=1}^{c} C_i \tag{3.13}$$

$$L_{sum} = (\sum_{i=1}^{l} \frac{1}{L_i})^{-1} \tag{3.14}$$

The code for the 0-junction is analogous to the code of the 1-junction.

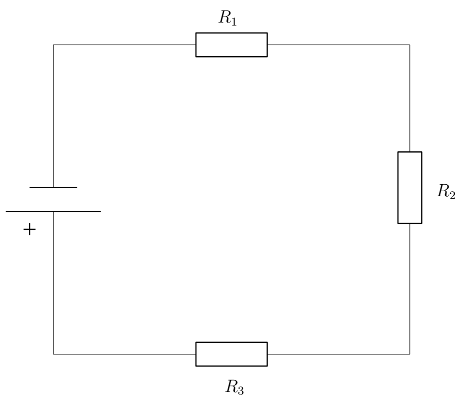
CHAPTER 4

Simulation Examples

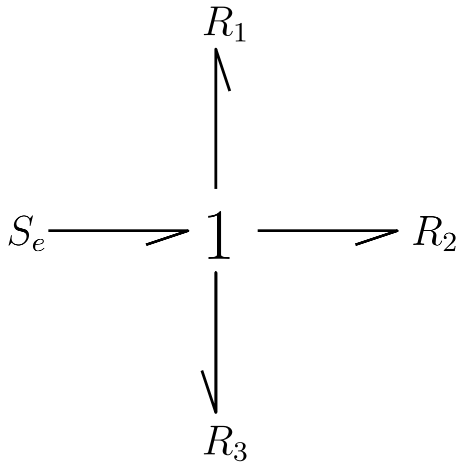
In this chapter, we present simulation examples using our bond graph library in PowerRPDEVS. The examples are electrical circuits, mechanical systems and models with parts from electrical, mechanical and hydraulic domains. We will present a bond graph, then its implementation with our library, followed by graphs showing the simulation results.

4.1 Voltage Divider

The first example is based on an electrical circuit of a voltage divider. Our model consists of a source of effort connected to three resistors in series. The electrical circuit is depicted in figure 4.1a. Figure 4.1b presents the corresponding bond graph.



(a) Electrical circuit.



(b) Bond graph.

Figure 4.1: Voltage divider

$$\begin{cases} e_0 + e_1 + e_2 + e_3 = 0 \\ e_1 = R_1 \cdot f_1 \\ e_2 = R_2 \cdot f_2 \\ e_3 = R_3 \cdot f_3 \\ f_0 = f_1 = f_2 = f_3 \end{cases}$$

$$\implies e_0 = -R_1 \cdot f_0 - R_2 \cdot f_0 - R_3 \cdot f_0 = -f_0 \cdot (R_1 + R_2 + R_3)$$
$$\iff f_0 = -\frac{e_0}{R_1 + R_2 + R_3}$$

$$f_0 = \frac{e_0}{R_{sum}}$$

In this example the causalisation algorithm is important. As we can see, no element in this bond graph is predestinated to provide the flow. We have the effort and three non-causalised resistors. In this case the 1-junction computes the equivalent resistance of the connected elements and is able to compute the resulting flow. Later the computed flow is sent to the resistors.

The PowerRPDEVS bond graph model for this example is depicted in figure 4.2. The sensors were added to measure the effort and flow of the resistors. The source of effort sends constant effort of 48V(Volt). The resistances are: $R_0 = 6\Omega$, $R_1 = 4\Omega$, $R_2 = 2\Omega$. Solving the voltage divider we should get the voltages respectively: $V_0 = 24$V, $V_1 = 16$V, $V_0 = 8$V. The current in this circuit is same for all elements $I = 2$A(Ampere).

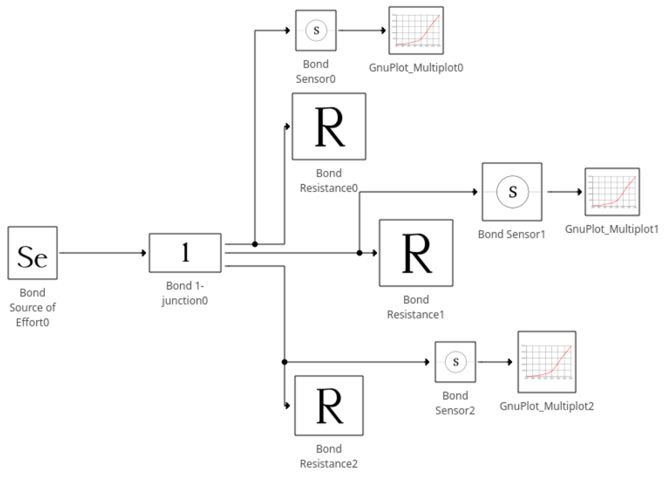


Figure 4.2: PowerRPDEVS bond graph of voltage divider.

The simulation returns the expected values. Figures 4.3, 4.4, 4.5 depict the effort and flow values of the resistors R_1, R_2, R_3 respectively.

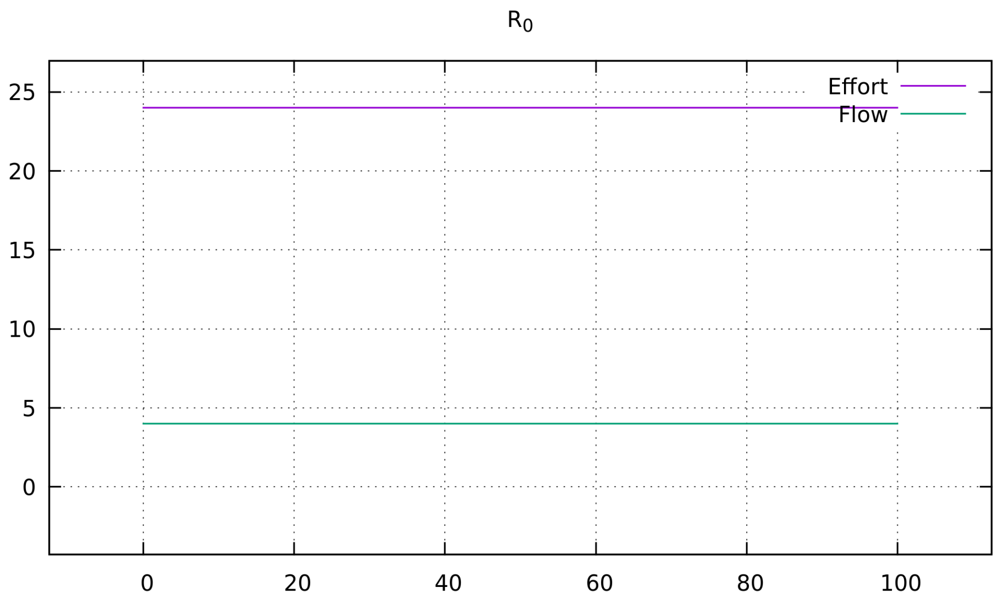


Figure 4.3: Effort and flow at the resistor R_0.

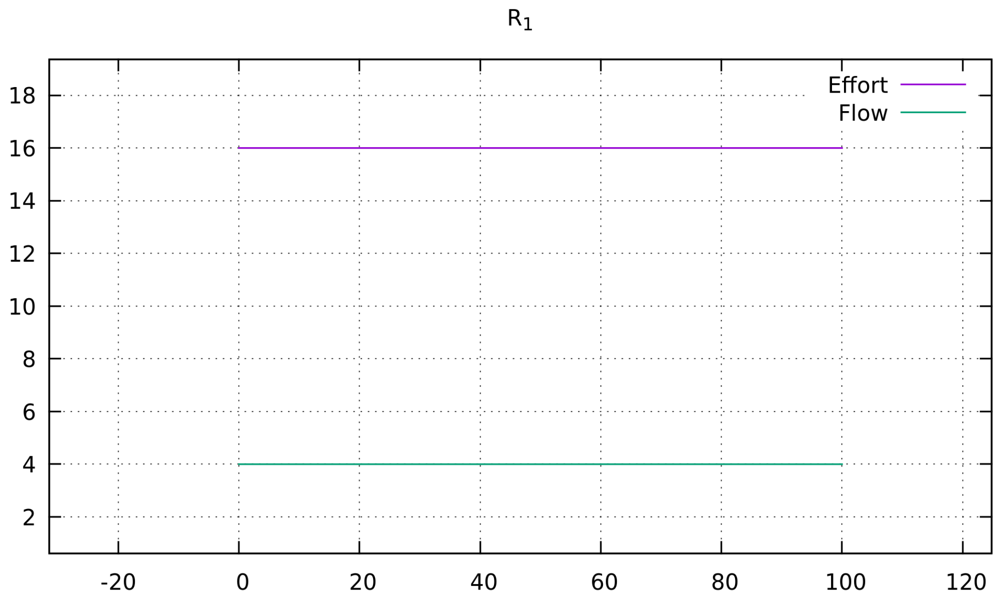


Figure 4.4: Effort and flow at the resistor R_1.

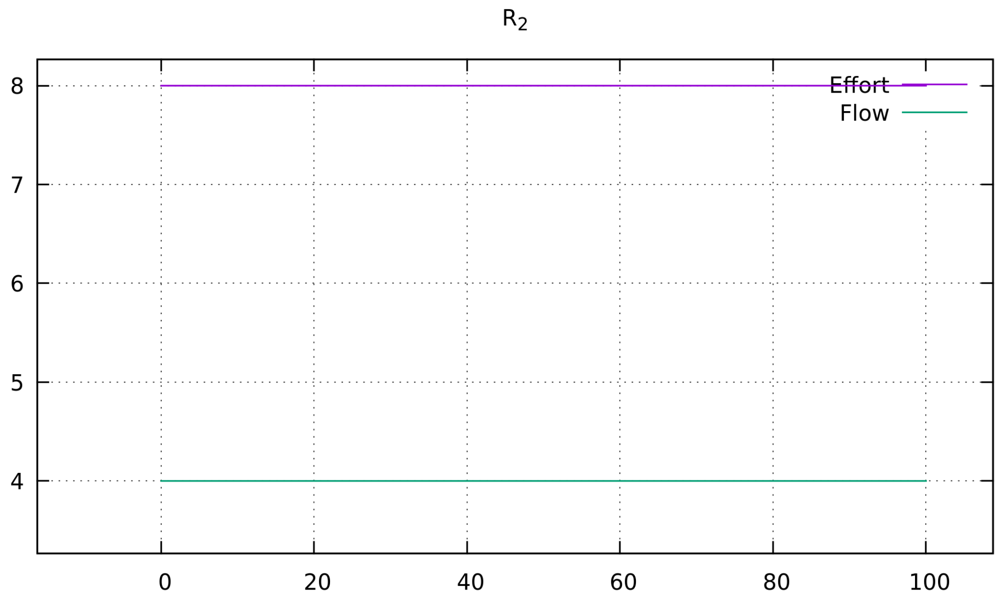
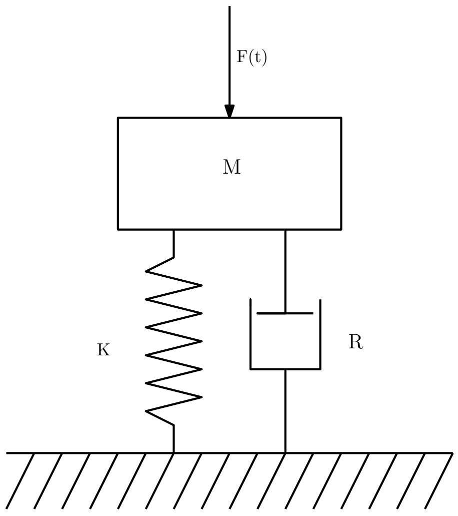


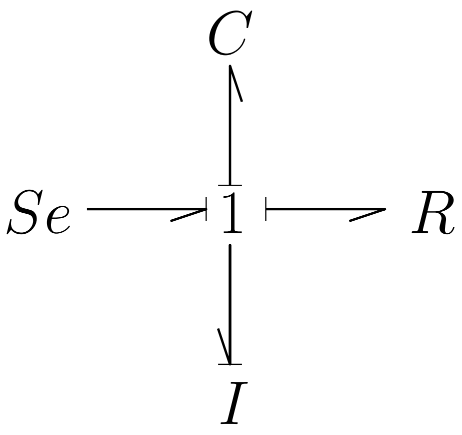
Figure 4.5: Effort and flow at the resistor R_2.

4.2 Mass-Spring-Damper

This example is a mechanical system of a mass-spring-damper based on [Sam03]. The physical model is depicted in the figure 4.6a. In the figure 4.6b, we can see the bond graph of the mass-spring-damper. In this model the source of effort is the force that acts on the object, the capacitor represents a spring, the inductor represents the mass of the object on the resistor represents the damper.



(a) The physical model.



(b) The corresponding bond graph.

Figure 4.6: A mass-spring-dampener model [Sam03].

Figure 4.7 shows mass-spring-damper bond graph built as a model in PowerRPDEVS.

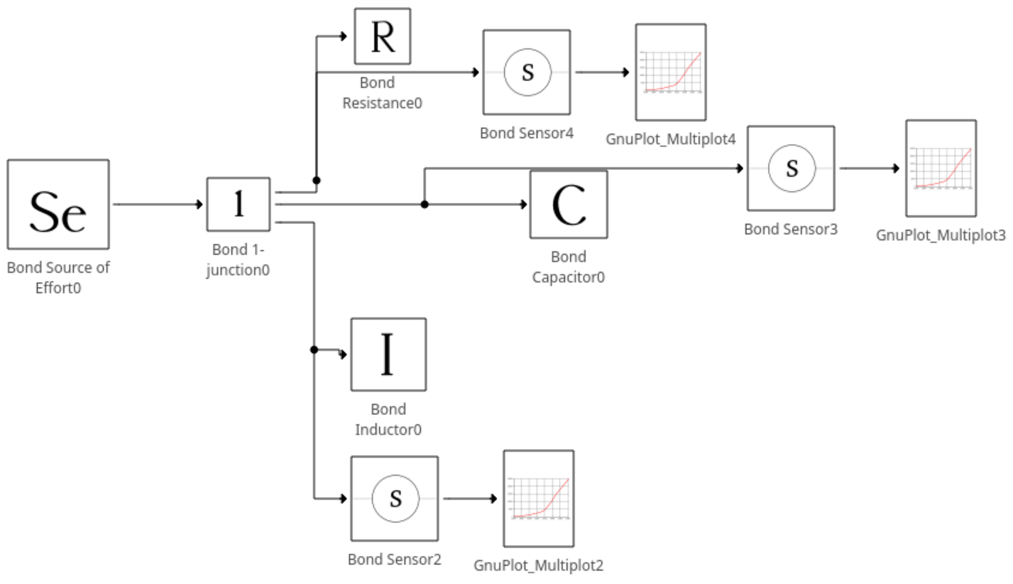


Figure 4.7: The mass-spring-damper bond graph modelled in PowerRPDEVS

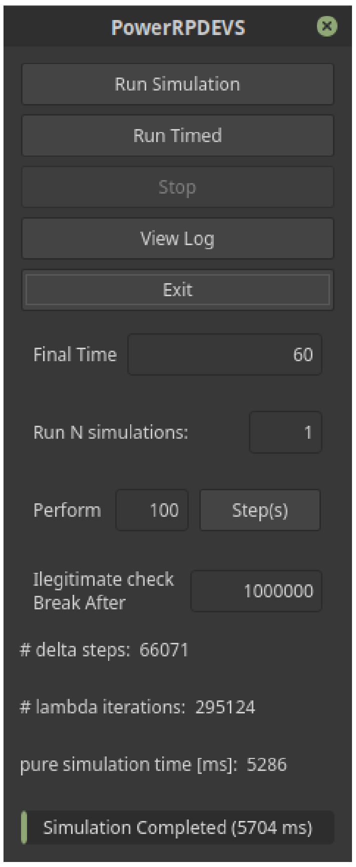
We simulated this bond graph two times with the same parameters but using two different QSS-orders. The parameters of the components are the following:

- spring C: stiffness: $0.2\frac{N}{m}(\frac{Newton}{meter})$ and initial effort of: 24N,
- source of force Se: Force: 10N,
- resistor R: resistance: $0.1\frac{Ns}{m}$,
- mass I: inductivity: 2kg and initial flow: $0\frac{m}{s}$.

We simulated this bond graph for 60s.

4.2.1 QSS2 Method

First time we used the QSS method second order and the quantum of 10^{-5}. The simulation took 5704ms having 66071 delta-steps and 295124 lambda-iterations. Figure 4.8 presents the simulation window after the simulation.



Final Time 60

Run N simulations: 1

Perform 100

Ilegitimate check Break After 1000000

delta steps: 66071

lambda iterations: 295124

pure simulation time [ms]: 5286

Simulation Completed (5704 ms)

Figure 4.8: The PowerRPDEVS simulation window after simulation

In the PowerRPDEVS, bond graph above we also attached sensors to the bonds. The sensors receive every message flowing on the bond and forward effort and flow messages to the next element *GNUPlot Multiplot*. The GNUPlotMultiplot component is taken from the *sinks* library (also available in the PowerRPDEVS) and it generates scripts for gnuplot [1] and runs gnuplot with the scripts as input. The following figures present graphically the results of the simulation. In this example the effort is given in N (Newtons), the flow is given in $\frac{m}{s}$ and the time is given in s.

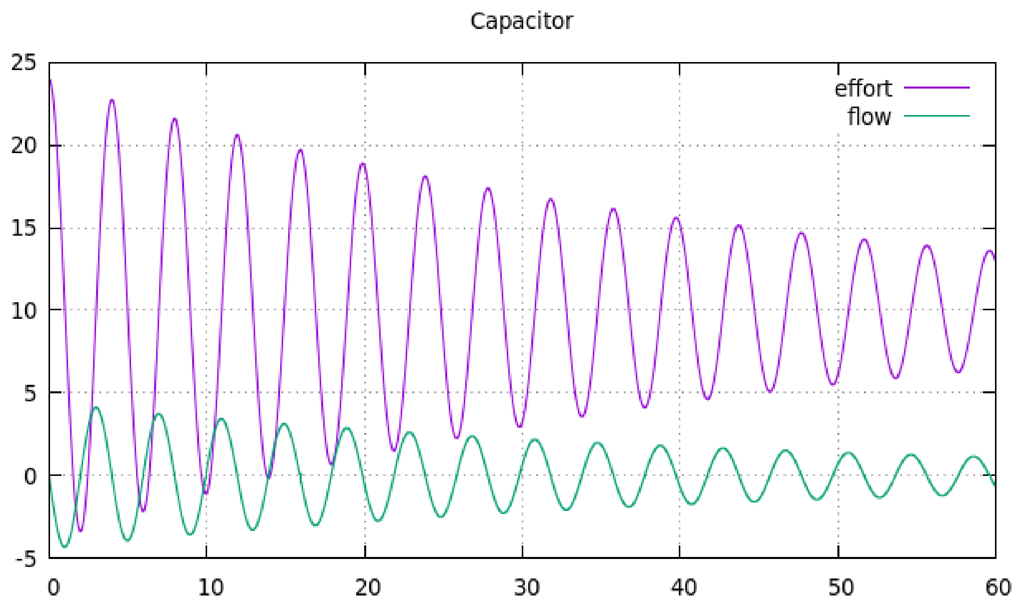


Figure 4.9: Effort and flow at the capacitor

In the figure 4.9, we can see the effort begins with 24N as we set it to have this initial value. Later the effort is oscillating around 10N which is the effort given from the source of effort. The flow begins at $0\frac{m}{s}$, as there is no flow in the bond at the beginning.

[1] Avialable from Ubuntu's apt repository

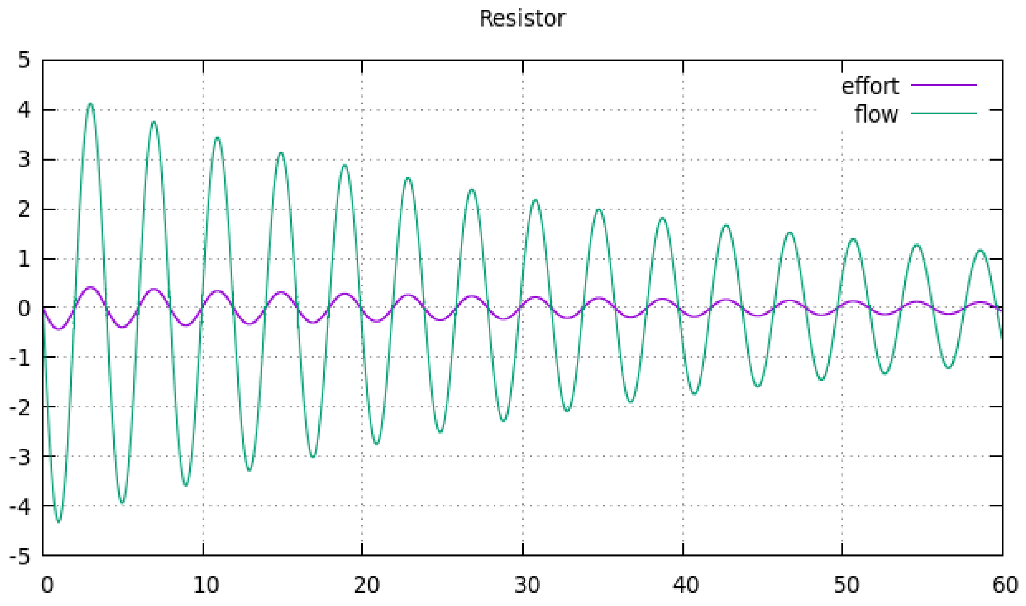


Figure 4.10: Effort and flow at the resistor

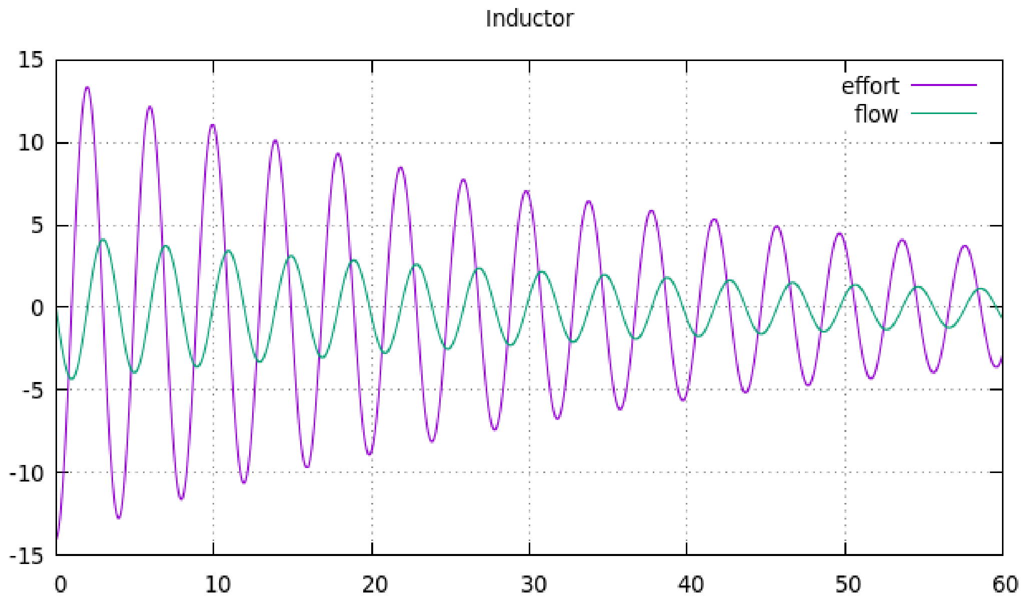
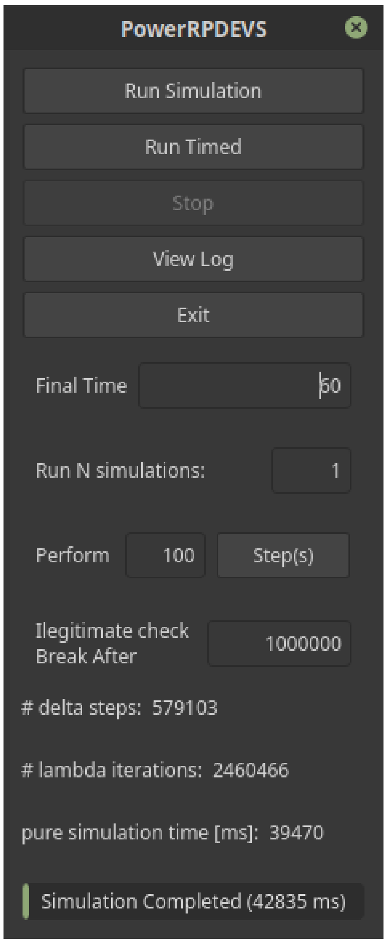


Figure 4.11: Effort and flow at the inductor

From the figures we can state that the object was swinging at the start of the simulation, with a trend to stabilize with the time progress.

4.2.2 QSS1 Method

For the second simulation, we took the QSS method of first order and a quantum parameter of 10^{-3}. This simulation took a lot more time (on the same computer without any load). It ended at 42835ms after running 579103 delta-steps and 2460466 lambda iterations. We can see it depicted in the figure 4.12



Final Time 60

Run N simulations: 1

Perform 100

Ilegitimate check Break After 1000000

delta steps: 579103

lambda iterations: 2460466

pure simulation time [ms]: 39470

Simulation Completed (42835 ms)

Figure 4.12: The PowerRPDEVS simulation window after simulation

The following graphs present the result of the simulation (the units as before):

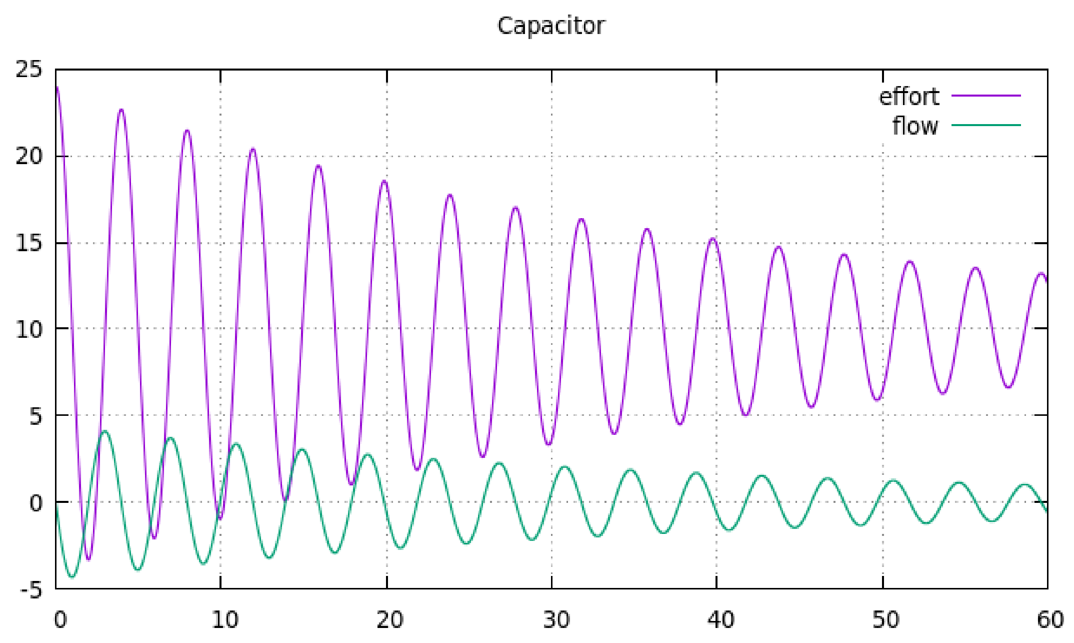


Figure 4.13: Effort and flow at the capacitor

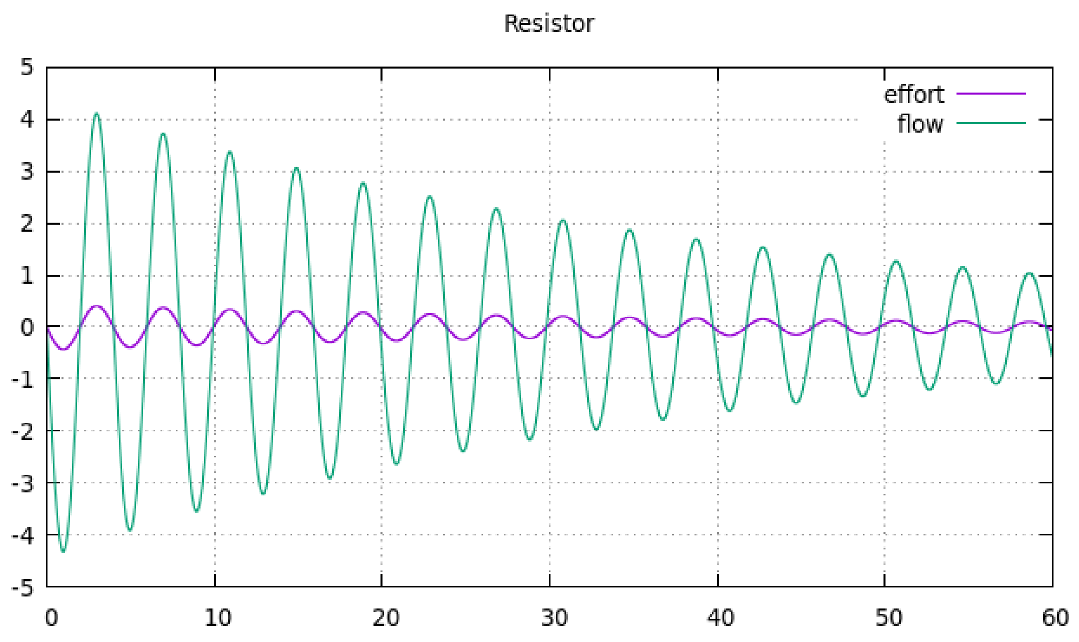


Figure 4.14: Effort and flow at the resistor

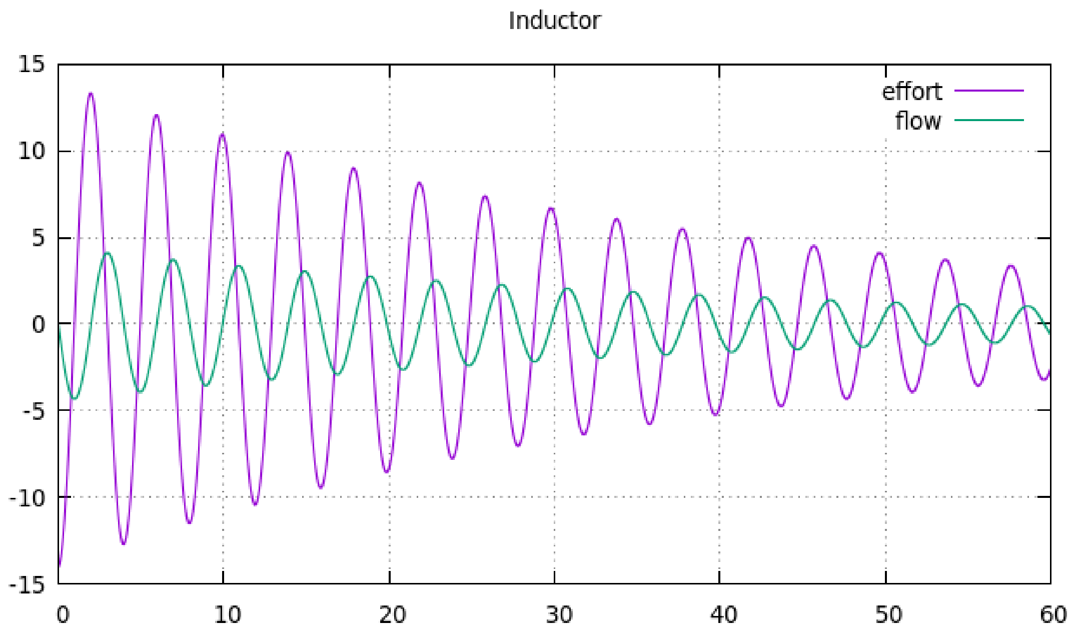


Figure 4.15: Effort and flow at the inductor

As we can see, the values are quite the same, the difference is, how often the value will be updated. It also depends on the quantum parameter however in this example the QSS method of order 2 is much faster than the QSS method of order 1.

4.3 Electric Motor Running a Hydraulic Pump with Mechanical Shaft

The following example, adapted from [Sam03], shows how we can model different physical domains together in one bond graph. We could imagine an electric engine powering a shaft which powers a water pump pumping water. In this example, we have the electrical power, the mechanical power and the hydraulic power combined. This can be modelled as one bond graph. We can see the model and the corresponding bond graph respectively in the figures 4.16 and 4.17.

This model uses almost all components (apart from the source of effort) from our PowerRPDEVS bond graph library. The figure 4.18 presents the bond graph modelled in PowerRPDEVS.

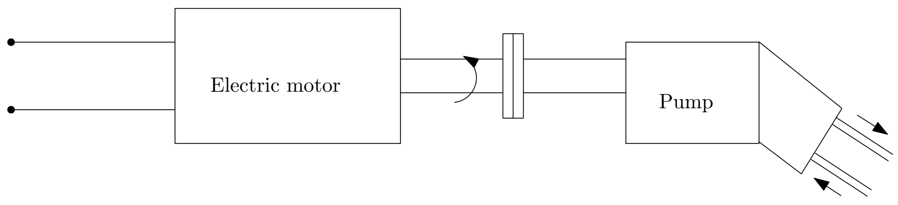


Figure 4.16: Electric motor running a hydraulic pump with mechanical shaft [Sam03].

$$\begin{tikzcd}
 & & & I_1 & & C_0 & I_0 \\
S_f \arrow[r] & 1 \arrow[r, "V", "i"'] & Gy \arrow[r, "\tau", "\omega"'] & 1 \arrow[u] \arrow[d] \arrow[r, "\tau", "\omega"'] & Tf \arrow[r, "P_1", "V_1"'] & 0 \arrow[u] \arrow[d] \arrow[r, "P_2", "V_2"'] & 1 \arrow[u] \arrow[d] \\
 & & & C_1 & & R_0 & R_1
\end{tikzcd}$$

Figure 4.17: Electric motor - hydraulic pump bond graph [Sam03].

The source of flow models the electrical engine. Then with the gyrator the electrical energy is converted to mechanical rotational energy. The inertial element I_1 represents the rotational energy storage in the mechanical shaft. The capacitance C_1 represents the torsion spring property of the rotating shaft. Then, the rotational energy is transformed to hydraulic energy with a transformer. The inertial element I_0 represents the moving parts of the pump, the capacitance element C_0 represents the compressibility of the fluids in the pump (energy storage). The resistance elements are added, because a non ideal pump will leak some energy on the moving parts.

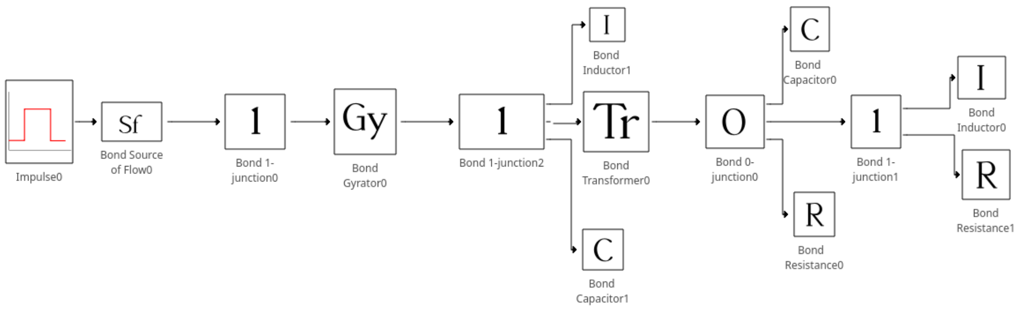


Figure 4.18: Electric motor - hydraulic pump bond graph modelled in PowerRPDEVS.

We run the simulation with the QSS method of second order, a quantum of 10^{-8}, and the following parameters (using the International System of Units (SI)): $S_f = 60$ $I_0 = 80$,

$C_0 = 200$, $C_1 = 150$, $R_0 = 10000$, $I_1 = 100$, $R_1 = 10000$, for the transistor $n_1 = 7$ and $n_2 = 6$, and for the gyrator $n_1 = 2$ and $n_2 = 7$.

In the simulation, the electrical engine is powered for 600s and then turned off. We can see this behaviour in the figure 4.19.

The following figures present the effort and flow at chosen bonds (units are also in SI):

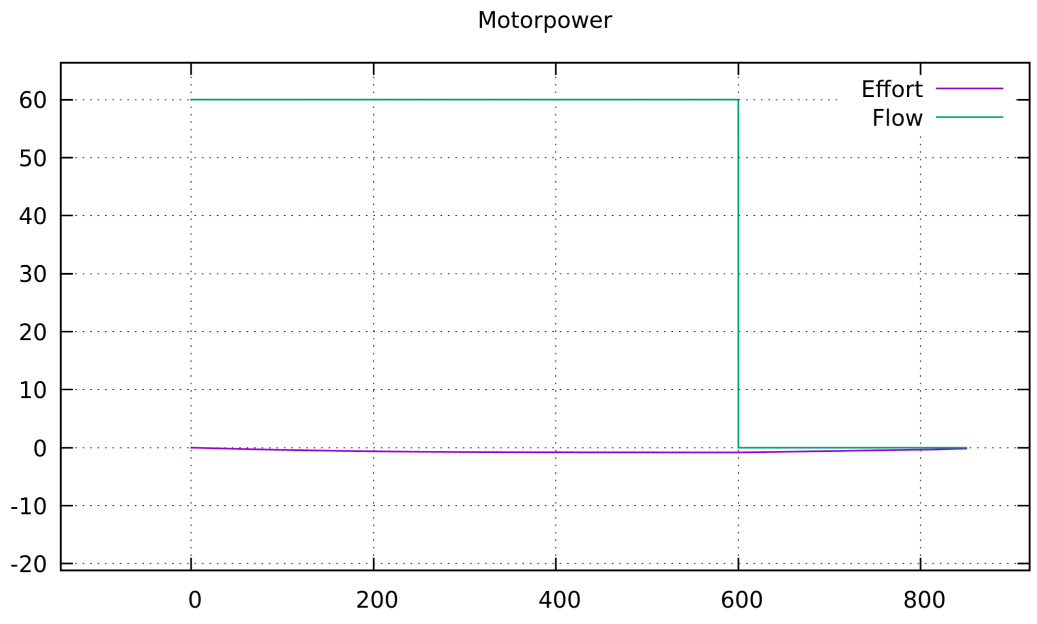


Figure 4.19: Effort and flow at the electrical motor.

At the moment, when the electrical engine is turned off, the shaft still rotates and the pressure rises, however the volume of the water flow decreases. The shaft slows down and as the water pressure works against the shaft, at the time about 820, the shaft starts rotating in the opposite direction (green line).

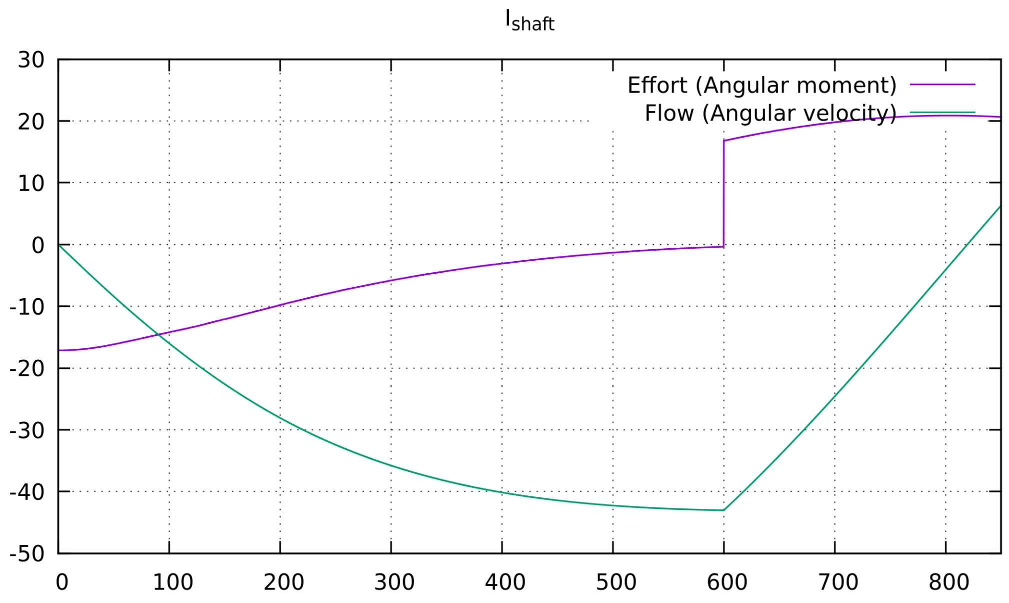


Figure 4.20: Effort and flow at the I_0.

The leakage of the pump is minimal. Therefore the pressure on the resistors is high, but the water volume flowing is low, oscilating about 0 (figure 4.21).

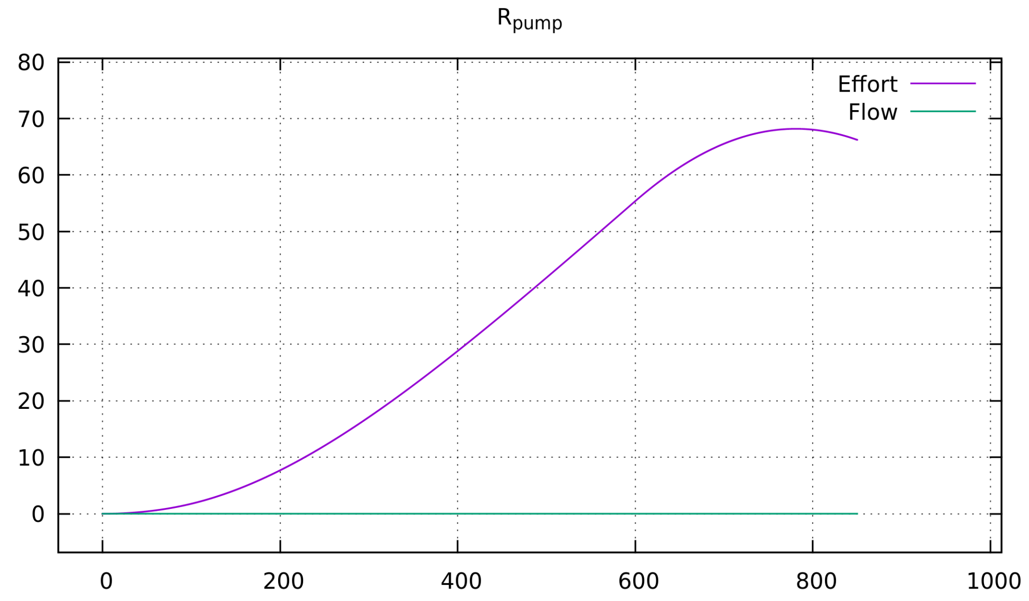


Figure 4.21: Effort and flow at the R_1.

In the figure 4.22 we can see, that the pump pushes the water back at the time of about 820.

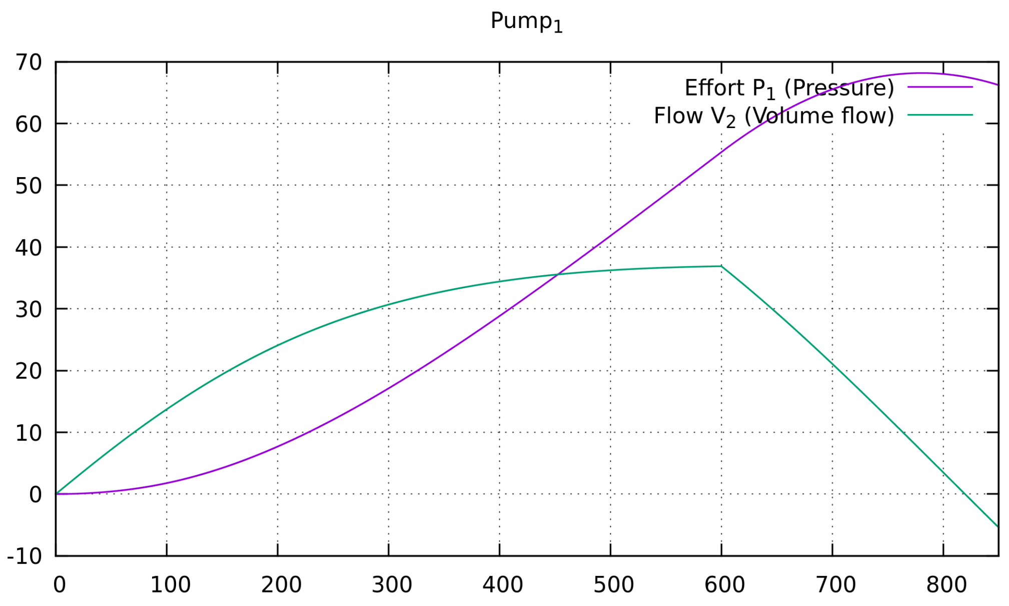


Figure 4.22: Effort and flow at the first part of the pump.

CHAPTER 5

Conclusion and Outlook at the Future Work

In this work, we have presented a new bond graph library for PowerRPDEVS. The bond graph library implements all the basic bond graph elements. Using the RPDEVS formalism, we also introduced a new distributed algorithm to causalise bond graphs. This algorithm also works for bond graphs which, in principle can be causalised, but which do not have a unique solution for the causalisation (e.g. voltage divider).

During our work on this first version of bond graph library for PowerRPDEVS, we thought about some improvements and features we could implement in the future.

A further extension could be a switch element to switch parts of the bond graph active during simulation.

Another suggestion we have in mind is a resistor with non-constant resistance parameter (like for example temperature driven resistor) or with a non-bond resistance input port to set the resistance from outside. This however also influences the junctions which exchange their equivalent resistance during the causalisation phase.

The bond graph library for PowerRPDEVS may also be extended with new elements. In a work from Borutzky [Bor15], we can find examples for other bond graph components useful for the modelling of hybrid systems. Borutzky implemented the well known bouncing ball model as bond graph, using a *modulated transformer* and a *modulated source* element. Furthermore, he proposes *switched power junctions*.

List of Figures

List of Tables

List of Algorithms

Bibliography

[Bor15] Wolfgang Borutzky. Bond graph model-based fault diagnosis of hybrid systems. pages 21–48, 2015.

[CZ94] Alex Chung Hen Chow and Bernard P Zeigler. Parallel devs: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation*, pages 716–722. Society for Computer Simulation International, 1994.

[DW03] Mariana C D'Abreu and Gabriel A Wainer. Hybrid dynamic systems: models for continous and hybrid system simulation. In *Proceedings of the 35th conference on Winter simulation: driving innovation*, pages 641–649. Winter Simulation Conference, 2003.

[DW05] Mariana C D'Abreu and Gabriel A Wainer. M/cd++: modeling continuous systems using modelica and devs. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 229–236. IEEE, 2005.

[KJ01] Ernesto Kofman and Sergio Junco. Quantized-state systems: a devs approach for continuous system simulation. *Transactions of The Society for Modeling and Simulation International*, 18(3):123–132, 2001.

[NGD01] Aziz Naamane, Norbert Giambiasi, and A Damiba. Generalized discrete event simulation of bond graph. *Simulation*, 77(1-2):4–22, 2001.

[PHK18] Franz Preyser, Bernhard Heinzl, and Wolfgang Kastner. Rpdevs: Revising the parallel discrete event system specification. *IFAC-PapersOnLine*, 51(2):242–247, 2018.

[PHRK16] Franz Preyser, Bernhard Heinzl, Philipp Raich, and Wolfgang Kastner. Towards extending the parallel-devs formalism to improve component modularity. In *Tagungsband des Workshops der ASIM/GI-Fachgruppen STS und GMMS 2016*, pages 83–89, 2016.

[Pre15] Franz Josef Preyser. An approach to develop a user friendly way of implementing dev&dess models in powerdevs. *Master's Thesis, TU Wien, Wien*, 2015.

[Sam03] A. Samantaray. About bond graph–the system modeling world. `http://groups.csail.mit.edu/drl/wiki/index.php?title=File:Samantaray_2001_www.bondgraphs.com_about.pdf`, 2003. Accessed: 2018-11-23.

[ŠHČG12] Patrik Šarga, Darina Hroncová, Miloslav Čurilla, and Alexander Gmiterko. Simulation of electrical system using bond graphs and matlab/simulink. *Procedia Engineering*, 48:656–664, 2012.

[ZKP00] Bernard P Zeigler, Tag Gon Kim, and Herbert Praehofer. Theory of modeling and simulation. pages 3–9;75–91, 2000.