FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

PowerRPDEVS Modellbibliothek für 9-wertige Logikschaltungen

nach IEEE Std. 1164-1993

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Christian Fiedler
Matrikelnummer 01363562

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 5. März 2019

Christian Fiedler	Wolfgang Kastner

PowerRPDEVS Model Library for 9-Value Logic Circuits

According to IEEE Std. 1164-1993

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Christian Fiedler
Registration Number 01363562

to the Faculty of Informatics

at the TU Wien

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

Vienna, 5th March, 2019

Christian Fiedler	Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Christian Fiedler
Mexikoplatz 2-3/4/6, A-1020 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, 5. März 2019

Christian Fiedler

Kurzfassung

Logikgatter, wie `AND`, `OR` und `XOR`, wenden eine Boolesche Operation auf ihre Eingangssignale an, um ein Ausgangssignal zu erzeugen. Die Simulation eines Schaltkreises bestehend aus Logikgattern, liefert daher das Ergebnis des zugehörigen Booleschen Ausdrucks. Das ist unkompliziert, solange der Schaltkreis keine Rückkopplungen enthält, wie dies zum Beispiel bei einem RS-Latch der Fall ist. Ein RS-Latch kann aus zwei `NOR`-Gattern aufgebaut werden, deren Ausgänge mit einem Eingang des jeweiligen anderen `NOR`-Gatters verbunden werden. Diese Modelle verhalten sich nicht immer entsprechend den Erwartungen des Benutzers.

PowerRPDEVS ist ein Simulator, der auf dem neuen Modellierungsformalismus *Revised Parallel Discrete Event System Specification* (RPDEVS) basiert. Der für RPDEVS definierte Simulationsalgorithmus verspricht transparenteres Modellverhalten im Falle von rückgekoppelten Systemen. Für diese Arbeit soll eine Logik-Bibliothek für PowerRPDEVS implementiert werden, die fundamentale Logikgatter (`AND`, `OR`, `XOR`) und andere hilfreiche Primitive enthalten soll, um die Möglichkeit zu erforschen, Latches, Flip-Flops und andere zustandsbehaftete Schaltkreise mit PowerRPDEVS zu simulieren. Insbesondere werden ein RS-Latch, ein RS-Flip-Flop, ein D-Flip-Flop und ein Shiftregister modelliert und simuliert.

Die Logik-Bibliothek wurde mit der 9-wertigen Logik aus IEEE Standard 1164-1993 implementiert, was ihre Fähigkeit, echte Hardware zu simulieren, verbessern sollte. Um das zu zeigen, wurde ein simpler Bus mit zwei Teilnehmern, die Schreibkonflikte auflösen, modelliert und simuliert.

Abstract

Logic gates like AND, OR and XOR perform Boolean operations on their inputs to produce an output. Simulating a circuit of logic gates yields the result of the corresponding Boolean expression. This is straightforward unless the circuit contains feedback loops, like for example in a static RS flip-flop. A static RS flip-flop can be constructed from two NOR gates whose outputs are connected to the inputs of the respective other NOR gate. Such models do not always behave according to the expectations of the modeler.

PowerRPDEVS is a simulator based on the new modeling formalism *Revised Parallel Discrete Event System Specification* (RPDEVS). The simulation algorithm defined for RPDEVS promises to be more transparent in its model behavior in the case of feedback systems. The purpose of this thesis is to implement a logic library for PowerRPDEVS containing fundamental logic gates (AND, OR, XOR) and other helpful primitives to explore the possibility of modeling flip-flops and other stateful circuits in PowerRPDEVS. In particular, models of a static RS flip-flop, a triggered RS flip-flop, a D flip-flop and a shift register are shown and simulated.

The logic library was implemented according to the 9-value logic of IEEE standard 1164-1993 which should improve its usability in simulating actual hardware. To show this, a model of a simple bus with two nodes that perform collision resolution is created and simulated.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**

2 Theoretical Background **3**
2.1 The Family of DEVS Formalism 3
2.2 IEEE 1164-1993 Standard for Multivalue Logic Systems 7

3 PowerRPDEVS **9**
3.1 Graphical User Interface 9
3.2 Engine 12

4 Multivalue Logic Library **17**
4.1 DEVSLogicMessage 17
4.2 Logic Gates Implementation 19
4.3 Library Elements 20

5 Example Logic Circuits **25**
5.1 Static RS Flip-Flop 25
5.2 Triggered RS Flip-Flop 31
5.3 D Flip-Flop 33
5.4 Shift Register 34
5.5 Serial Bus with Collision Resolution 36

6 Conclusion and Outlook **41**

A Code of Implementation **43**

List of Figures **63**

List of Tables **65**

Acronyms **67**

Bibliography **69**

CHAPTER 1

Introduction

Simulation of logic circuits in state-of-the-art general purpose simulators using their respective formalisms can be inconvenient and may yield unexpected results that do not match the intents of the modeler. This is what Junglas shows in his paper [Jun16], in which he built simple logic circuits based on discrete blocks of the modeling tools Simulink and Dymola / Modelica.

These logic circuits needed to be implemented with additional blocks (the *Memory* block in Simulink and the *Pre* block in Modelica) that introduce infinitesimal delays into the circuit. This is because the simulation algorithm apparently cannot deal with the algebraic loops that were introduced with the feedbacks. The algebraic loops can be avoided when the simulated logic gates are designed with an intrinsic delay which can also be observed in their physical counterparts.

The purpose of this thesis is to figure out how these issues affect PowerRPDEVS, a simulation software based on the Revised Parallel DEVS (RPDEVS) formalism [PHK17], which is a further developed version of the original Discrete Event System Specification (DEVS), first proposed by Zeigler [ZPK00], and Parallel DEVS (PDEVS) formalisms, later proposed by Chow [CZ94].

The main reason for revising these existing formalisms was that they did not allow modeling of Mealy behavior that can respond to an incoming event immediately without a state change. To imitate Mealy behavior in DEVS and PDEVS, it is necessary to change into a *transitory state* after the arrival of an incoming event. A transitory state is a state with zero lifetime and thus, is left again in the same instant of simulation time. Transitory states often lead to intransparent model behavior as discussed in [PHK17]. RPDEVS tries to avoid transitory states by allowing the modeler to model Mealy behavior directly.

For this thesis, a model library that contains fundamental logic gates (`AND`, `OR`, `XOR`, `NOT` gates) was implemented for PowerRPDEVS. It was decided to not limit this library to Boolean logic and implement a multivalue logic system. With this library, a number of logic circuits is modeled and simulated to investigate the behavior of algebraic loops in PowerRPDEVS.

Chapter two will first briefly introduce the formalisms DEVS and PDEVS and their mathematical background, then motivate RPDEVS and point out the major differences. In the following section in chapter two, the reasons for implementing the Institute of Electrical and Electronics Engineers (IEEE) standard 1164-1993 will be discussed which describes a 9-value multivalue logic system that is used in Very High Speed Integrated Circuit Hardware Description Language (VHDL).

Chapter three introduces PowerRPDEVS which is the proof-of-concept simulator and modeling software that is used for RPDEVS modeling. First, the user interface of the toolchain will be described. Then, a more detailed view is taken at the internal workings of PowerRPDEVS, its simulation algorithm and message passing mechanism.

The fourth chapter will at first discuss certain decisions that were made and considerations taken during the implementation phase of the library and then introduce the blocks that are implemented in this library and how they work in PowerRPDEVS.

In chapter five, the circuits from Junglas' paper [Jun16] are implemented and simulated in PowerRPDEVS and the results of the simulation and possible pitfalls in the modeling phase will be shown. It will also discuss the practical advantages of RPDEVS over traditional DEVS or PDEVS and also the possible differences specifically when simulating logic circuits between the RPDEVS algorithm and the algorithms used in Simulink and Modelica based on the simulation results.

A conclusion on the work is given afterwards and points will be mentioned that could be of interest in future research in this field.

CHAPTER 2

Theoretical Background

2.1 The Family of DEVS Formalism

2.1.1 DEVS and Parallel DEVS

DEVS is a modeling formalism first introduced by Zeigler in 1976 [ZPK00]. A DEVS model consists of so-called *atomics* (also *atomic DEVS*), which are systems with an internal state as well as input and output ports. Output ports can be connected to input ports with uni-directional connections. *Couplings* (also *coupled DEVS*) consist of connected atomics and couplings. Such a coupling can also have input and output ports that are connected to the inputs and outputs of its subcomponents. These components, atomics and couplings, are also referred to as *blocks* later on, when the distinction is not needed.

Mathematically, a DEVS atomic can be expressed with the following tuple:

$$< X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta > \tag{2.1}$$

$$\begin{aligned}
X &\;\dots\; \text{set of possible inputs} \\
Y &\;\dots\; \text{set of possible outputs} \\
S &\;\dots\; \text{set of possible (internal) states of the atomic} \\
\delta_{ext} &: Q \times X \to S \;\dots\; \text{external state transition function} \\
&\quad \text{where } Q = \{(s,e)|s \in S, e \in [0, ta(s)]\} \\
\delta_{int} &: S \to S \;\dots\; \text{internal state transition function} \\
\lambda &: S \to Y \;\dots\; \text{output function} \\
ta &: S \to \mathbb{R}_0^+ \cup \{\infty\} \;\dots\; \text{time advance function}
\end{aligned}$$

[PHRK16]

An atomic DEVS $A =< X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta >$ can receive internal and external events. When an *internal event* is received, the λ function is evaluated for the current internal state of A, the result is passed to the output ports of A and further to the atomics or couplings connected to them where it causes an external event. Thereafter, the δ_{int} function is evaluated for A and the resulting state becomes its new state. When an *external event* is received by A, its δ_{ext} function is evaluated which causes an immediate change of internal state. [ZPK00]

An atomic DEVS can be compared to a Moore-type finite state automaton (FSA) [Jos96]. The atomic's output function λ is only dependent on the current internal state of the atomic just like the output of a Moore-type FSA is bound to its state. The state transition functions $\delta_{int}, \delta_{ext}$ compute the new state but cannot change the output value. δ_{ext} is called everytime an external event arrives and thus, can incorporate the data of the external event into the new state. That implies that to change the output of the atomic, it has to change internal state first. This is also the way Moore-type FSA work, there are a few differences in detail though, e.g. the state space of an atomic DEVS could theoretically be infinite.

Models usually consist of multiple atomic and coupled DEVS that are connected. Together they again form a *coupled DEVS* which behaves like an atomic and thus could be replaced by a corresponding atomic. This property is called *closure under coupling.* [ZPK00]

Problems can arise when multiple DEVS in a model schedule internal events for the same time or when a DEVS receives an external event at the same time of a scheduled internal event. In these cases, the *select* function which is a property of the coupled DEVS chooses the order of the events. Correctly implementing the atomics and couplings of a model, so they can deal with simultaneous events and do not depend on the ordering imposed by the *select* function, can be difficult and tedious. If the blocks depend on this ordering, they are harder to reuse in other models. In further revisions of DEVS, it was tried to resolve this and other ambiguities. [ZPK00]

Chow and Zeigler published a revision of DEVS in 1994 called PDEVS [CZ94] which introduces the δ_{conf} transition function that prevents the serialization problems mentioned above regarding simultaneous internal and external transitions. Instead of evaluating simultaneous δ_{int} and δ_{ext} in a certain order, in PDEVS these collisions are resolved by evaluating the new δ_{conf} function.

With this change, the *select* function is not necessary anymore and simulation of multiple PDEVS components could run in parallel. With the new δ_{conf} function, an *atomic*

PDEVS can be described by the tuple:

$$< X, Y, S, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta > \tag{2.2}$$

$$\begin{aligned} \delta_{conf} : Q \times X \rightarrow S \ \ldots \ & \text{confluent state transition function} \\ & \text{where } Q = \{(s,e) | s \in S, e \in [0, ta(s)]\} \end{aligned}$$

To give an example of a DEVS model, a Boolean `NOT` gate with one input and one output shall be modeled as a DEVS atomic. The `NOT` gate will always output the negated input value (so `0` when the input is `1` and vice-versa) and react to inputs immediately.

$$\begin{aligned} \mathbb{B} &= \{\mathtt{0}, \mathtt{1}\} \\ NOT_{devs} &= \ < X = \mathbb{B}, Y = \mathbb{B}, S, \delta_{ext}, \delta_{int}, \lambda, ta > \end{aligned} \tag{2.3}$$

$$\begin{aligned} S &= \mathbb{B} \times \{0, \infty\} \ \ldots \ s = (s_1, \sigma) \in S \\ \delta_{ext}(s, e, x) &= \begin{cases} (x, 0) : \text{if } x \neq s_1 \\ s : \text{otherwise} \end{cases} \\ \lambda(s) &= \neg s_1 \\ \delta_{int}(s) &= (s_1, \infty) \\ ta(s) &= \sigma \end{aligned}$$

The model of the `NOT` gate (Equation 2.3) includes two transitory states (a state s is transitory if $ta(s) = 0$). Those states had to be introduced because λ is not evaluated when only an external event is triggered. An external event only leads to an evaluation of δ_{ext}. δ_{ext} will produce a state with $\sigma = 0$, so that an internal event is also triggered at the same point in simulation time. The internal event results in an evaluation of λ which sets the new output and δ_{int} which switches back to a stable state (σ is set to ∞).

When transitory states are present in a DEVS/PDEVS model, λ may be called multiple times for the same point in simulation time if the model switches to transitory states multiple times in a row. This can lead to endless loops in which simulation time is never increased and thus the simulation cannot complete.

2.1.2 Revised Parallel DEVS

RPDEVS (as introduced by Preyser et al. in [PHK17]) is based on PDEVS but applies major changes to the formalism:

1. The three functions δ_{ext}, δ_{int} and δ_{conf} of PDEVS are combined into one δ function. A case analysis whether the transition is external, internal or confluent, can be done optionally by the modeler within δ based on its additional parameters.

2. The λ function is also evaluated for blocks that receive purely external events.

3. The λ function depends on the inner state as well as on the external event. Earlier, atomics of the original formalisms DEVS and PDEVS were compared to Moore-type FSA – this change in the formalism together with the previous point now allows for atomics to behave like Mealy FSA (reacting to an input with an output without changing the internal state).

4. The δ function in the new formalism is evaluated after the λ evaluations have stopped. This is partly a design decision and partly a necessity due to the new λ function that has access to the data of the incoming external events. The DEVS model has time to stabilize throughout these multiple λ evaluations none of which may affect the internal state. When the results of λ remain the same throughout the model, it is ready for a state change, i.e. evaluation of δ.

The intention of the changes to λ was mainly to eliminate the necessity for *transitory states* (see subsection 2.1.1). These were needed in DEVS and PDEVS to model Mealy behavior, i.e. DEVS being able to produce output events instantanously in reaction to an input event.

The formal definition of the RPDEVS atomic conveys these changes:

$$\begin{aligned} &< X, Y, S, \delta, \lambda, ta > \\ \delta &: (Q \times X) \rightarrow S \ \ldots \text{ state transition function} \\ \lambda &: (Q \times X) \rightarrow Y \ \ldots \text{ output function} \\ &\text{where } Q = \{(s,e) | s \in S, e \in [0, ta(s)]\} \end{aligned} \tag{2.4}$$

[PHK17]

The `NOT` gate from section subsection 2.1.1 can now be modeled without transitory states (see Equation 2.5). The definition also became simpler because the δ functions were merged and because the model only needs one state now.

$$\begin{aligned}
NOT_{rpdevs} &= \; < X = \mathbb{B}, Y = \mathbb{B}, S, \delta, \lambda, ta > \\
S &= \{s_0\} \\
\delta(s, e, x) &= s_0 \\
\lambda(s, e, x) &= \neg x \\
ta(s) &= \infty
\end{aligned} \tag{2.5}$$

As with DEVS and PDEVS, couplings can be defined in RPDEVS. Like DEVS and PDEVS, also RPDEVS provides *closure under coupling.* [PHK17]

2.2 IEEE 1164-1993 Standard for Multivalue Logic Systems

The *PowerRPDEVS logic library* was implemented for this thesis. It works with nine logic values according to IEEE standard 1164-1993 [IEE93]. These are: `U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, `-`. The multivalue logic system (from now on called "IEEE 1164") specified in it was specifically designed for VHDL and hardware modeling. As such it is more powerful than simple two-value Boolean logic and – regarding the `AND`, `OR`, `XOR` and `NOT` functions – IEEE 1164 is equivalent to Boolean logic when the inputs are elements of $\{\texttt{0}, \texttt{1}\}$. Thus, it can be seen as a superset of Boolean logic.

2.2.1 Logic values

In table Table 2.1 the logic values of IEEE 1164-1993 are described.

2.2.2 Motivation for using IEEE 1164

The primary purpose of modeling tools and simulation is to describe particular aspects of a real-world process on the basis of a certain formalism. The modeler typically neglects some of the properties of this real-world system or simplifies the model in other ways (e.g. linearization).

The PowerRPDEVS logic library was designed to provide logic gates that match the Boolean operators `AND`, `OR`, `XOR` and other logic elements. Boolean logic operates only on two discrete values: *true* and *false*, or `0` and `1`.

The motivation for using IEEE 1164 instead of Boolean logic was that Boolean logic cannot deal with many real-world use-cases of electronic circuits natively, e.g. simulating a data bus on which two or more parties can send or receive data. If there are two sources that both send on one wire (the bus), the output is well-defined as long as both sources carry the same signal (`0` or `1`), however, if they carry complementary logic values

logic value	**description**
`U`	Uninitialized state. An input port was not connected or carries an undefined value. This value propagates through logic gates, so the error source can be found easily.
`X`	Forcing unknown. Strong driver with unknown logic value.
`0`	Forcing zero. Strong driver with low logic value (transistor connected to GND).
`1`	Forcing one. Strong driver with high logic value (transistor connected to VCC).
`Z`	High impedance. Disconnect wire.
`W`	Weak unknown. Weak driver with unknown logic value.
`L`	Weak zero. Weak driver with low logic value (pull down resistor).
`H`	Weak one. Weak driver with high logic value (pull up resistor).
`-`	Don't care. This value is only useful for optimizing VHDL synthesis.

Table 2.1: 9 logic values in IEEE 1164-1993

Boolean logic simply cannot resolve the output. While it is possible to work around this issue (see subsection 5.5.1), extending the domain of possible logic values helps to create models that more closely match the real world systems. With IEEE 1164, the *high impedance* value `Z` can be used by components to disconnect from the bus.

Still, this library should be useful even if the user only wants to use Boolean logic. If only elements of $\mathbb{B} = \{\texttt{0}, \texttt{1}\}$ are used as inputs, no signals are undefined and it is ensured that two complementary logic values are never connected to the same input port, the outputs will be elements of $\mathbb{B}$.

CHAPTER 3

PowerRPDEVS

PowerRPDEVS [POWb] is an open source software for simulating hybrid systems (systems with continuous and discrete components) based on the RPDEVS formalism, as explained in subsection 2.1.2. It was forked from PowerDEVS [POWa] which is a simulator that follows the DEVS formalism and its simulation algorithm was replaced by an implementation of the RPDEVS abstract simulator. As part of this replacement, the atomic block library of PowerDEVS became unusable and also had to be recreated.

The following sections will describe how the program is used to create atomics and models and simulate them. Afterwards, the engine of the simulator is described.

3.1 Graphical User Interface

3.1.1 Model Editor

The model editor (see Figure 3.1) is the primary user interface and the first window that appears when PowerRPDEVS is started. Here, models can be created by inserting and wiring RPDEVS atomic blocks from the libraries on the left. No programming is required if the desired model can be created by coupling already existing atomic blocks. This can be done graphically in the model editor. Creation of an atomic is described in subsection 3.1.3.

The blocks can be dragged from the left side into the model area on the right. Output ports can be connected to multiple input ports (all input ports receive the same message when the output port generates one) and multiple output ports can be connected to one input port (that one input port receives the messages from all the connected output ports). Especially in the latter case, care should be taken and the user should know how the used block deals with multiple inputs at the same port (the PowerRPDEVS logic library provides some blocks that merge these incoming messages, see section 4.3).

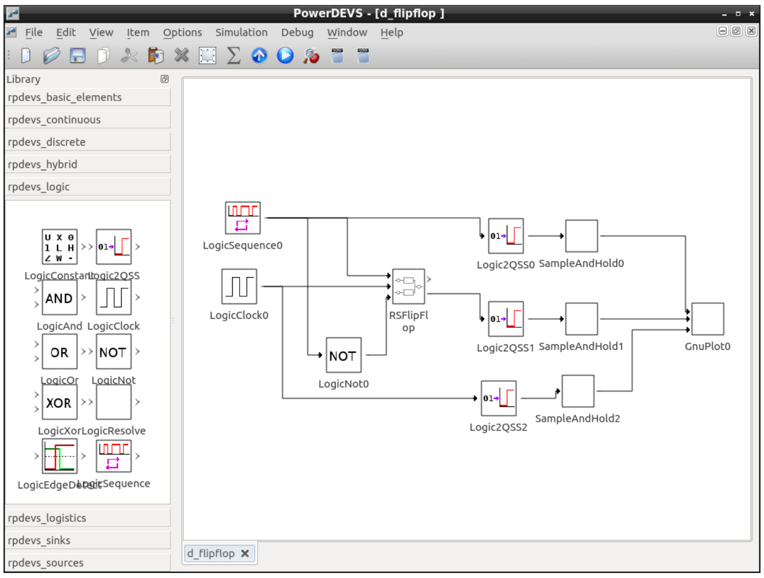


Figure 3.1: PowerRPDEVS Model Editor

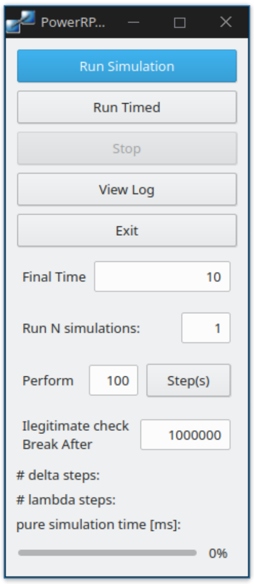
Couplings can be created by dragging the *coupled* block from the left side (which is inside the group *rpdevs_basic_elements*) into the model area. The coupling can then be opened by right-clicking it and then clicking the menu item *Open Coupled*. The model area becomes blank because the model editor navigated into the newly created coupling which is empty. Now the user can drag input and/or output ports from the left into the coupling. The user can also drag other atomic or coupled blocks into the coupling and wire them. When navigating back to the previous view by using the bar at the bottom of the model editor window, the coupling block shows the same number of input and output ports as were dragged into the coupling before.

3.1.2 Simulation of a model

When a simulation is started, PowerRPDEVS will compile all the involved atomic blocks, generate and compile C++ code that describes the model with its couplings and instantiates the atomic blocks' classes. This binary is then run to simulate the model.

By hitting the F5 function key or clicking the menu item Simulation→Simulate, the user can compile the model. When the compilation is finished, a dialog appears (see Figure 3.2) that allows to set additional constraints and then start the simulation.

Run Simulation and *Run Timed* both run the simulation but with *Run Timed* the engine will try to run the simulation in real time, i.e. the elapsed simulated time should always equal the elapsed wallclock time. The button *View Log* shows the simulation log that shows information about the execution – atomic blocks can write to the log which can be useful for debugging an atomic, there is also the *Recorder* block that can write the value



Final Time 10

Run N simulations: 1

Perform 100

Ilegitimate check Break After 1000000

delta steps:

lambda steps:

pure simulation time [ms]:

0%

Figure 3.2: PowerRPDEVS Simulation Dialog

of its input into the log.

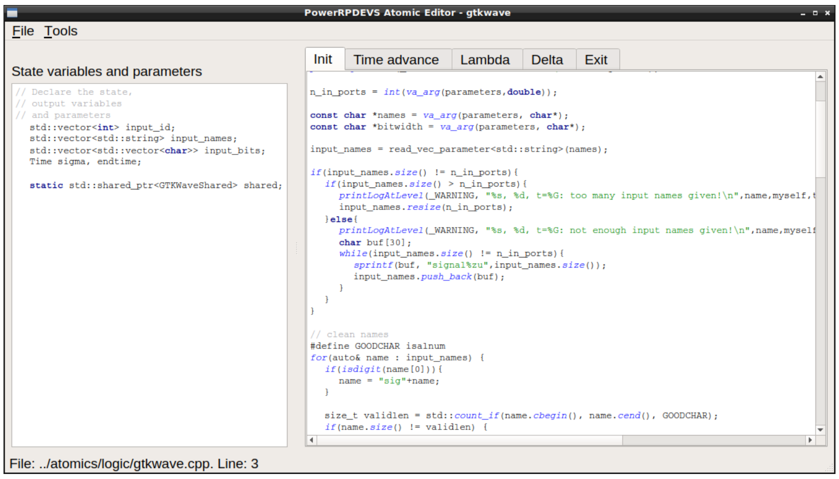
With the *Final Time* setting, the user can define for how long the simulation should run in seconds (simulated time). The number of simulations can be adjusted below. The user can also execute only a certain number of simulation steps.

3.1.3 Creating a new atomic

An atomic block is modeled as a C++ class that is derived from a base class called `Simulator`. It has member variables which represent the internal state of the atomic, an `init` method that is called once before the simulation starts, a `delta` method that represents the atomic's δ function, a `lambda` function that represents λ, and an `exit` function that is called after the simulation has ended.

The user can drag an atomic block from the left side into the model area, right-click it and click the *Edit Code* menu item. This opens the built-in *RPDEVS Atomic Editor* that allows to write C++ code to implement a new atomic RPDEVS. This editor imposes a layer of abstraction, because it does not show the whole C++ code of the atomic as one large file but groups the individual methods of the class that represents the atomic into a tabbed view. In the top bar, the user can switch between the functions *Init*, *Time Advance*, *Delta*, *Lambda* and *Exit*.

While *Time Advance*, *Delta* and *Lambda* directly correspond to the formalisms functions *ta*, δ and λ, there is no such correspondence for *Init* and *Exit*. *Init's* purpose is to set the initial state of the atomic and may be used to acquire system resources. It is called for all


PowerRPDEVS Atomic Editor - gtkwave
File Tools
State variables and parameters
Init
Time advance
Lambda
Delta
Exit

```
// Declare the state,
// output variables
// and parameters
   std::vector<int> input_id;
   std::vector<std::string> input_names;
   std::vector<std::vector<char>> input_bits;
   Time sigma, endtime;

   static std::shared_ptr<GTKWaveShared> shared;
```

```
n_in_ports = int(va_arg(parameters,double));

const char *names = va_arg(parameters, char*);
const char *bitwidth = va_arg(parameters, char*);

input_names = read_vec_parameter<std::string>(names);

if(input_names.size() != n_in_ports){
   if(input_names.size() > n_in_ports){
      printLogAtLevel(_WARNING, "%s, %d, t=%G: too many input names given!\n",name,myself,t
      input_names.resize(n_in_ports);
   }else{
      printLogAtLevel(_WARNING, "%s, %d, t=%G: not enough input names given!\n",name,myself
      char buf[30];
      while(input_names.size() != n_in_ports){
         sprintf(buf, "signal%zu",input_names.size());
         input_names.push_back(buf);
      }
   }
}

// clean names
#define GOODCHAR isalnum
for(auto& name : input_names) {
   if(isdigit(name[0])){
      name = "sig"+name;
   }

   size_t validlen = std::count_if(name.cbegin(), name.cend(), GOODCHAR);
   if(name.size() != validlen) {
```

File: ../atomics/logic/gtkwave.cpp. Line: 3

Figure 3.3: PowerRPDEVS Atomic Editor

blocks before the simulation starts. *Exit* serves no purpose for the formal simulation, but is required for cleanup on the software level (stopping programs, freeing system resources and memory). It is run after the simulation has completed.

These functions can be viewed in the large text editor box on the right-hand side by clicking them in the top bar (see Figure 3.3). On the left side there is an additional text editor that contains the atomic's internal state variables (member variables of the atomic's class) – all of them together form the internal state of the atomic as specified by the formalism.

3.2 Engine

The PowerRPDEVS engine is programmed in an object-oriented way in C++. As explained previously, to run the models they need to be compiled into a stand-alone executable binary file that performs the actual simulation. This binary is composed of the PowerRPDEVS engine, the model that is simulated and all the atomics that are part of it.

3.2.1 Message passing

The event-driven nature of RPDEVS is implemented as message passing between blocks. Atomic as well as coupling blocks have input and output ports. When an input port and an output port are connected, messages emitted at the otuput port will be forwarded to the input port.

For that purpose there is the abstract base class `DEVSMessage` in the engine. It is used to derive message types that will then be used by the various block libraries. The following is the source code of `DEVSMessage`:

```
class DEVSMessage {
public:
  int index;

  DEVSMessage() {
    index = 0;
  }
  DEVSMessage(int idx) {
    index = idx;
  }
  DEVSMessage(const DEVSMessage& msg) {
    index = msg.index;
  }
  DEVSMessage(std::string value_str) {
    index = 0;
    parseValueFromString(value_str);
  }

  virtual int getInt() {
    return(0);
  }

  virtual bool parseValueFromString(std::string value_str) {
    return(false);
  }

  virtual bool operator==(const DEVSMessage& msg) const {
    return(index==msg.index);
  }

  virtual bool operator!=(const DEVSMessage& msg) const {
    return(!((*this)==msg));
  }
  virtual DEVSMessage* getCopy() const {
    return(new DEVSMessage(*this));
  }

  virtual std::string toString() const {
    return(std::string("WARING: DEVSMessage-Instance!"));
  }
  virtual ~DEVSMessage() {};
};
```

The class contains no attributes except `index` which is used universally in derived classes for representing vectors. Every message has an associated index that represents its position in the vector.

Different message types are used for different purposes and atomics need a way to handle different messages or at least check if the message they receive has the correct type. To

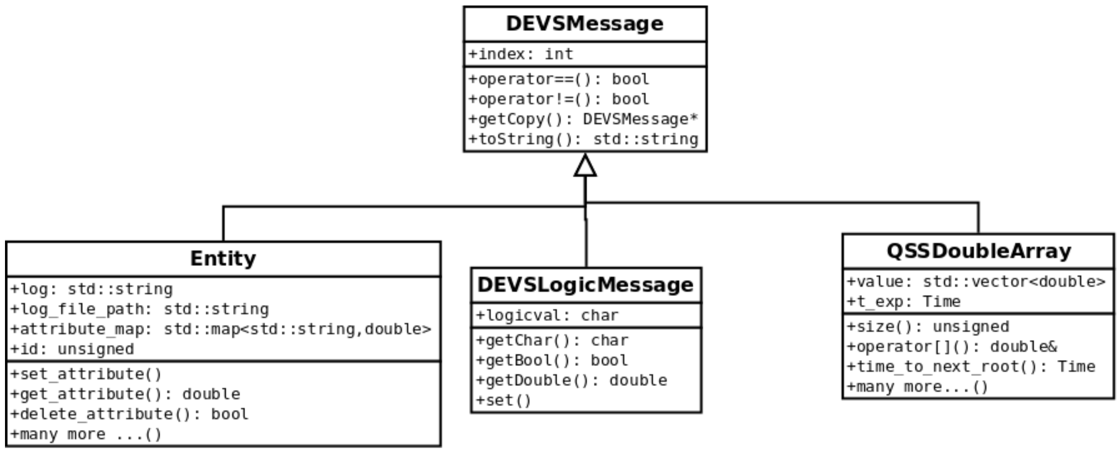


Figure 3.4: Message Passing class hierarchy

achieve this polymorphism is used. Input messages can be retrieved from the engine within the λ and δ functions as pointers to the base class `DEVSMessage*`. The function `pop_input` is used for that. A `dynamic_cast` may then be used to determine if the `DEVSMessage` is of a specific derived type, e.g. `QSSDoubleArray`:

```
DEVSMessage *msg;
while(pop_input(0, &msg)){
  auto q = dynamic_cast<QSSDoubleArray*>(msg);
  if(q != nullptr){
    // q is of type QSSDoubleArray
    // use q
  }
}
```

The life time of the received pointer is not guaranteed to be longer than the execution of the functions λ and δ, so if an atomic should store messages, it must copy the message object using the copy constructor or copy assignment operator `operator=`. It could also use the method `getCopy` and store the resulting pointer, but this is discouraged as this method is intended for the engine – moreover the block would have to do manual memory management. In any case, acquired memory or system ressources shall be freed when no longer needed or in the `exit` method.

To allow the λ iterations to stabilize, the engine will check if the output of λ of every atomic/coupling changed before copying the message from the output bag of one block to the input bag of another block. That is why the base class `DEVSMessage` implements `operator==` and `operator!=` and it requires the derived classes to implement `operator==` too (`operator!=` is implemented as the negative of `operator==` and shall not be overridden). The derived classes need two overloads of `operator==`, see section 4.1 for an explanation.

In Figure 3.4 the currently implemented message classes and the relations to each other are shown.

- **`QSSDoubleArray`** can represent continuous signals using Quantized State System (QSS) methods. It is used in most existing library components of PowerRPDEVS.
- **`DEVSLogicMessage`** can represent the logic values of IEEE 1164. It is only used in the PowerRPDEVS logic library currently.
- **`Entity`** represents an entity of logistics. It is used in the RPDEVS logistics module.

3.2.2 Simulation Algorithm

The simulation algorithm is formally defined by an *Abstract Simulator* in [PHKB19] and is implemented in the PowerRPDEVS engine.

The engine provides a `Simulator` class from which all atomic RPDEVS are derived. The functions `init`, `delta` (δ), `lambda` (λ), `ta` and `exit` have to be implemented by the atomic's class.

The `Coupling` class represents coupled RPDEVS and is also derived from `Simulator` so it provides the same functions that are required by the RPDEVS formalism, but the `Coupling` class implements them by calling the respective functions of its children (other couplings or atomics). `Coupling` also implements a large part of the engine as a model is essentially a coupling with nested couplings and atomics.

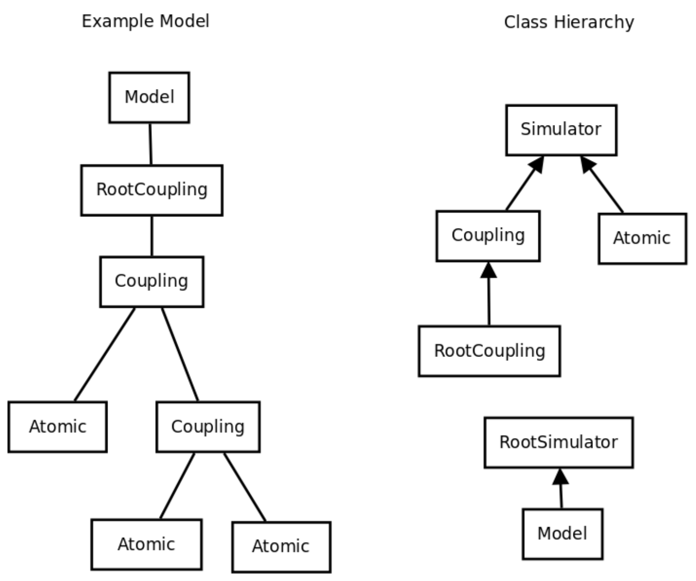


Figure 3.5: Example object-oriented model and class hierarchy

CHAPTER 4

Multivalue Logic Library

4.1 DEVSLogicMessage

The engine of PowerRPDEVS provides the class `DEVSMessage` that is used for the message passing (subsection 3.2.1). The developer is encouraged to derive from this class as it has virtual methods and a virtual destructor. The following `DEVSLogicMessage` class was derived from `DEVSMessage`:

```
class DEVSLogicMessage : public DEVSMessage
{
public:
  char logicval;
  // constructors:
  DEVSLogicMessage() : DEVSMessage()
  {
    logicval = 'U';
  }
  DEVSLogicMessage(char v, int _index = 0)
      : DEVSMessage(_index),
        logicval(v)
  {
  }
  DEVSLogicMessage(const DEVSLogicMessage &msg)
      : DEVSMessage(static_cast<DEVSMessage>(msg)),
        logicval(msg.logicval)
  {
  }

  char getChar() const
  {
    return logicval;
  }
  double getDouble() const
  {
    return getChar() == '1' ? 1 : 0;
  }

  bool getBool() const
  {
    return getChar() == '1' ? true : false;
  }
  void set(char c)
  {
    logicval = c;
  }

  virtual DEVSMessage *getCopy() const
  {
    return new DEVSLogicMessage(*this);
  }
  virtual std::string toString() const
  {
    return std::string(1, getChar());
  }

  virtual bool operator==(const DEVSLogicMessage &msg) const
  {
    if (index != msg.index)
    {
      return false;
    }
    return logicval == msg.logicval;
  }

  virtual bool operator==(const DEVSMessage &msg) const
  {
    if (typeid(msg) != typeid(*this))
      return false;
    return ((*this) == ((DEVSLogicMessage &)msg));
  }

  virtual ~DEVSLogicMessage() {}

  static bool valid(char v)
  {
    return logicfunction_value_valid(v);
  }

};
```

This class saves a logic value in a variable of data type `char` that can be one of the nine values of IEEE 1164 (subsection 2.2.1) and it is used for messages sent between blocks of the library. As `char` can hold more than nine values, a `valid` function is defined to verify that only valid logic values are used. This valid function calls `logicfunction_value_valid()` which is defined in `stdlogic1164.h` section A.2.

The `char` type is used because all nine logic values of IEEE 1164 have a single-character representation and it makes the code more readable to use them directly. Use of an equivalent *enum type* would be more tedious and the gain is not very high because C++'s type system does not enforce that *enum* variables only hold defined values. It might be beneficial to use a type that enforces that kind of type safety, but then more code for converting character strings to that type and vice-versa as well as implementing the IEEE 1164 logic functions would have to be added. This was not attempted for readability and simplicity purposes.

The *virtual* methods override the `DEVSMessage` base class functions that are used by the simulation algorithm. The `operator==` is implemented specifically for `DEVSLogicMessage` but an additional overload is provided that can check equality with `DEVSMessage` and that returns *false* if two DEVSMessages of different types are compared. The `getCopy` method is explained in subsection 3.2.1.

4.2 Logic Gates Implementation

Unary logic gates (e.g. `NOT`) as well as binary logic gates (e.g. `AND`, `XOR`) differ only in the function that is applied to the one or two inputs. So the implementation of unary and binary logic gates was written as templated classes `LogicGateUnary` and `LogicGateBinary` which both export the methods `init`, `ta`, `delta` and `lambda`. A function that maps input(s) to output can be given as template argument. That way the message handling that needs to be done in the λ and δ function was split from the actual functions. Thus, more unary and binary logic functions can be created as RPDEVS atomics by simply instantiating the above template classes and calling its methods inside the respective functions of the new atomic.

A shortened version of the `AND` atomic can be used as a reference to implement other binary functions as atomics, see the listing below. The complete implementation is provided in section A.4.

```
class logic_and: public Simulator {
        LogicGateBinary<logicfunction_and,logicfunction_resolution>
            ↪ gate;
public:
        logic_and(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void delta(double);
        void lambda(double);
        void exit();
};
```

```
11
12
13
14  void logic_and::init(double t,...) {
15    va_list parameters;
16    va_start(parameters,t);
17
18    n_in_ports = int(va_arg(parameters,double));
19
20    const char *init_logic_vec = va_arg(parameters, char*);
21    gate.init(DEVSLogicMessage::getLogicVector(init_logic_vec));
22
23    va_end(parameters);
24  }
25
26  double logic_and::ta(double t) {
27    return gate.ta();
28  }
29
30  void logic_and::delta(double t) {
31    gate.delta(*this,t);
32  }
33
34  void logic_and::lambda(double t) {
35    gate.lambda(*this,t);
36  }
37
38  void logic_and::exit() {
39  }
```

The instantiation of LogicGateBinary in line 2 takes the actual binary (meaning it has two inputs) logic function logicfunction_and and a resolution function logicfunction_resolution that resolves conflicts between multiple writers. Both are defined in the header file stdlogic1164.h which implements the IEEE standard 1164 [IEE93] and is shown in the appendix in full length in section A.2. The appendix also contains the implementation of LogicGateUnary and LogicGateBinary in section A.3.

4.3 Library Elements

In the following, it will be described how the atomics of the PowerRPDEVS logic library (see Figure 4.1) work.

As pointed out in subsection 3.2.1 the engine is already prepared to use vector values everywhere so most of these logic gates can transfer and work with multiple logic values (a logic vector) via one connection.

In order that these atomics can be used easily, most of them implement automatic *signal resolution* of inputs. That means, on ports where the atomic receives logic signals from more than one source, it will apply the resolution function defined in IEEE 1164 and it will only interpret the resulting value as input on those ports. That means if for example two wires are connected to a `NOT` gate and one carries a `0` and the other an `H` the resolution at the input port results in a `0` because the strong logic value wins against the weak. On this result of the resolution, the actual logic gate's function is applied. Thus, in the example the `NOT` gate would produce a `1` at its output.

When the input is a logic vector of multiple elements, resolution is applied to each element individually. This is also true for the logic functions like `AND`, `OR` etc.

In cases where automatic resolution is not provided but needed, the LogicResolve block shall be used. For blocks that have at least one input it is denoted below if they do not automatically resolve inputs.

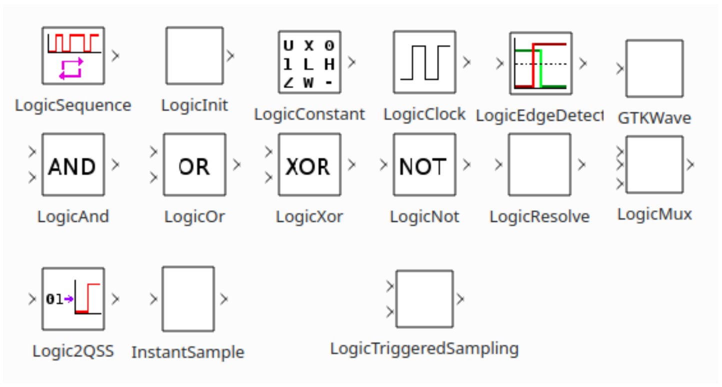


Figure 4.1: Multivalue Logic Library – all blocks

The blocks in Figure 4.1 are described in the order left to right, top to bottom in the following paragraphs.

LogicSequence LogicSequence is similar to LogicClock, it can switch between more than two states though. The user may specify a sequence of logic values that is repeatedly output. The period of one iteration over all logic values can be specified too. All output values of the sequence are held for an equal time slice of $\frac{\text{duration}}{\text{number of values in sequence}}$.

LogicInit The LogicInit block sets a user-defined logic value or a vector of them on its output when $t = 0$. It uses a transitory state to set the output to `Z` or a vector of `Z` with as many elements as the given vector in the next iteration of the simulation.

LogicConstant The LogicConstant block can be used as a source for a constant logic value or a vector of them. The user can specify the logic value/vector in the *Parameters* dialog of the block. This logic value/vector is saved in the instance of the LogicConstant block and is written to its output in the λ function when the simulation starts.

LogicClock LogicClock implements a clock. The period, duty cycle, high and low logic values can be be adjusted in the *Parameters* dialog. The block has a single output port which alternates between these two logic values. The high value is held for *duty cycle* $\times$ *period* and the low value is held for $(1 - \textit{duty cycle}) \times \textit{period}$.

LogicEdgeDetect The LogicEdgeDetect block will output `0` unless an edge in the input is detected. Then a `1` is output for a duration that can be specified in the *Parameters* dialog of the block.

It should be noted that there cannot be a detectable edge at the time $t = 0$ since a rising/falling edge is defined by a transition from a low logic value (`0` or `L`) to a high logic value (`1` or `H`) and vice versa. But, all inputs are initialized with `U` at $t < 0$, so at $t = 0$ the transition begins at the uninitialized value `U`.

GTKWave The GTKWave block can have arbitrarily many inputs and is used to view logic signals directly in the external program GTKWave. This program supports the IEEE 1164 logic values.

This atomic does not perform input value resolution.

LogicAnd, LogicOr and LogicXor The functions `AND`, `OR` and `XOR` are implemented with the respective tables obtained from the IEEE standard 1164 [IEE93]. These blocks allow an arbitrary number of input ports.

The implementation of LogicAnd is provided in section A.4.

LogicNot The `not` gate has a single input which is inverted according to the respective table in IEEE 1164 [IEE93] and then set as output of the block.

LogicResolve The resolve block allows to connect two or more wires, so the output of this block will be the result of the `resolved` function from IEEE 1164 (see [IEE93]). As wires in PowerRPDEVS are unidirectional the inputs are not affected.

LogicMux The LogicMux block is a multiplexer with $(1 + 2^n)$ inputs ($n \geq 1$). The first input is the selector which chooses one of the 2^n data inputs as active. The selector input has to be a logic vector of length n. This vector is converted to a number from 1 to 2^n and then the corresponding input of the atomic is chosen as active. Note that input 0 is the selector input. The active data input is forwarded to the output of the LogicMux block.

The conversion of the selector vector $S = (s_0, \ldots, s_{n-1})$ is done as follows:

$$\text{active port} = \begin{cases} 1 & \text{if } \exists s_i \in S : \text{to\_x01}(s_i) = \mathtt{X} \\ 1 + \sum_{i=0}^{n-1} 2^i \cdot \mathtt{to\_x01}(s_i) & \text{otherwise} \end{cases}$$

When the selector input cannot clearly be evaluated to `0` or `1` port 1 is chosen as active and an error is written to the log.

The function `to_x01` is defined in [IEE93] and its implementation is included in section A.2.

This atomic does not perform input value resolution.

Logic2QSS The Logic2QSS block converts incoming logic values to QSSDoubleArray. This is used for interfacing the logic library with the other libraries that are provided in PowerRPDEVS, e.g. tracking a logic value by plotting it using a GnuPlot block.

The conversion uses the `getDouble` method of the `DEVSLogicMessage` class and is done independently for every index.

This atomic does not perform input value resolution.

InstantSample The InstantSample block is designed for being connected between a Logic2QSS block and a GnuPlot block. It performs much better for logic values and optimizes its output for GnuPlot. It is only useful for this specific use case.

This atomic does not perform input value resolution.

LogicTriggeredSampling The LogicTriggeredSampling block is an edge-triggered sampling block. Its first input is a data line and the second input is a clock. Whenever an edge is detected at the clock the input signal is passed to the output.

There are two major configuration options:

- The user can configure which edges on the clock line lead to a change at the output (either rising edge, falling edge or both, where rising and falling are defined as in IEEE standard 1164-1993).
- The user can configure either a left limit or right limit sampling property which determines which value of the input line is forwarded when the input changes simultaneously with a clock edge.

The implementation of LogicTriggeredSampling is provided in section A.5.

CHAPTER 5

Example Logic Circuits

In this chapter, the results and challenges of reimplementing and testing the circuits from Peter Junglas' paper [Jun16] in PowerRPDEVS with the newly created PowerRPDEVS logic library will be described. Junglas implemented several flip-flops and a shift register in Simulink and Modelica and was facing issues.

Additionally, after the examples relating to Junglas' paper, another example circuit is presented that demonstrates the capabilities of the PowerRPDEVS logic library that are due to its multivalue logic.

5.1 Static RS Flip-Flop

The static RS flip-flop is a logic element with two inputs and two outputs which can save one bit of information. The two inputs are *Set* (S) and *Reset* (R). The outputs are conventionally called Q and $\overline{\text{Q}}$. Q represents the stored bit and $\overline{\text{Q}}$ its inverse, but they can also both hold the same logic value (particularly if S and R are `1`). In this case, the RS flip-flop is in an invalid state.

Setting the S input to `1` while R is `0` will set the stored information bit to `1` and setting S to `0` while R is `1` will clear the stored information bit (set it to `0`). When both inputs R and S are `0` the information bit is not changed.

5.1.1 Simulation in Simulink and Modelica

In his paper [Jun16], Junglas first implemented a static RS flip-flop. Thereby, he was forced to use a delaying block (`Memory` block in Simulink, `pre` block in Modelica) to break the algebraic loop that we are going to discuss in subsection 5.1.2. He presented two models of static RS flip-flops in Simulink (see Figure 5.1).

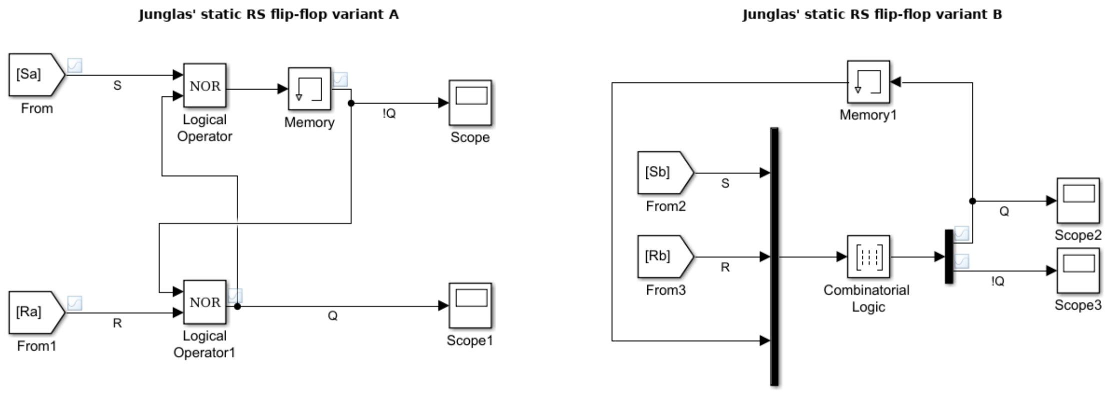


Figure 5.1: static RS flip-flop variants by Junglas

Variant A is a straight-forward approach using a *Memory* block and variant B feeds the output back to the input (with a delay) and uses a combinatorial function on the two inputs R and S and the old output Q to determine the new outputs Q and $\overline{\text{Q}}$. Variant B eliminates the delay in the forward path that was imposed by the Memory block, particularly the delay in variant A is asymmetric, i.e. the Q and $\overline{\text{Q}}$ do not change at the same time.

Junglas also described that variant A from above can be reproduced in Modelica with a *Pre* block taking the place of the *Memory* block. According to him, this simple version yields the expected results in Modelica, so there is no need to reproduce variant B.

5.1.2 Simulation in PowerRPDEVS

Due to the event based communication in PowerRPDEVS, signals that are actually continuous are modeled with discrete events that report every significant change in the signal. As the output value of a logic gate may depend on all input values, the logic gates fo PowerRPDEVS have to save the last received logic values for every port. Thus, they can react also to a change of only a few of their inputs. In such a case, the logic gate can use the values from its internal state for all inputs which did not receive a new signal value at a given time.

Moreover, as many of the atomics in the PowerRPDEVS logic library perform input signal resolution (see section 4.3), they have to save the latest logic value they received at every port for every source from whom they received it. There can be multiple sources for every port and if not all of them change at once, the old values from the internal state have to be used for the other sources when calling the resolution function.

When considering the logic gates of the logic library as operators of Boolean algebra we can describe logic circuits built in PowerRPDEVS using Boolean algebra. As the logic gates were designed with no intrinsic delay, a model without a feedback loop can be evaluated within the λ iteration phase. The result should match what is expected from the Boolean expression.

Once a feedback loop is added, this representation using Boolean algebra will contain an algebraic loop, see e.g. Equation 5.3, describing the static RS flip-flop in Figure 5.2. Explicit expressions for Q and $\overline{\text{Q}}$ cannot be found.

$$Q = \neg(R \vee \overline{Q}) \tag{5.1}$$
$$\overline{Q} = \neg(S \vee Q) \tag{5.2}$$
$$\Rightarrow Q = \neg(R \vee \neg(S \vee Q)) \tag{5.3}$$

A truth table can be obtained by considering Q and $\overline{Q}$ as independent and evaluating the first two equations in Equation 5.3 for S and R, see Table 5.1.

S	R	Q	$\overline{\text{Q}}$
1	1	0	0
0	0 | 1	0	1
0 | 1	0	1	0
no solution	no solution	1	1

Table 5.1: RS latch truth table obtained by Boolean algebra

This truth table gives a few indications, how the RS flip-flop might behave, but it does not really explain how state transitions work or what the output for $S = R = 0$ is, because Boolean algebra does not have a notion of time.

But, PowerRPDEVS uses λ iterations to iteratively calculate the outputs of the individual atomics. So, for the first iteration – when one or both of the `NOR` gates are evaluated – it has to use the old values for Q and $\overline{\text{Q}}$ for the calculation of the output. For the following iterations the value from the last iteration is used. Equation 5.4, Equation 5.5 and Equation 5.6 show the possible recurrence relations that result from these λ iterations. Which of these relations is actually solved depends on the particular implementation of the RPDEVS simulation algorithm and on whether the algorithm can run in parallel.

Solutions of these recurrences are given in Table 5.2 and Table 5.3 up until a point where the outputs stay the same for at least one iteration. At that point, the λ iteration would end. As Equation 5.4 and Equation 5.5 are symmetric, the solution for Equation 5.5 is omitted. In Table 5.3 the cases where the λ iteration cannot terminate are highlighted.

$$\begin{aligned} Q_n &= \neg(R \vee \overline{Q}_n) \\ \overline{Q}_n &= \neg(S \vee Q_{n-1}) \end{aligned} \tag{5.4}$$

$$\begin{aligned} Q_n &= \neg(R \vee \overline{Q}_{n-1}) \\ \overline{Q}_n &= \neg(S \vee Q_n) \end{aligned} \tag{5.5}$$

$$\begin{aligned} Q_n &= \neg(R \vee \overline{Q}_{n-1}) \\ \overline{Q}_n &= \neg(S \vee Q_{n-1}) \end{aligned} \tag{5.6}$$

$\mathbf{Q_{n-1}}$	**S**	**R**	$\mathbf{Q_n}$	$\overline{\mathbf{Q_n}}$	$\mathbf{Q_{n+1}}$	$\overline{\mathbf{Q_{n+1}}}$	$\mathbf{Q_{n+2}}$	$\overline{\mathbf{Q_{n+2}}}$
0	0	0	0	1	0	1	0	1
0	0	1	0	1	0	1	0	1
-	1	0	1	0	1	0	1	0
-	1	1	0	0	0	0	0	0
1	0	0	1	0	1	0	1	0
1	0	1	0	0	0	1	0	1

Table 5.2: Solutions of Equation 5.4.

$\mathbf{Q_{n-1}}$	$\overline{\mathbf{Q_{n-1}}}$	**S**	**R**	$\mathbf{Q_n}$	$\overline{\mathbf{Q_n}}$	$\mathbf{Q_{n+1}}$	$\overline{\mathbf{Q_{n+1}}}$	$\mathbf{Q_{n+2}}$	$\overline{\mathbf{Q_{n+2}}}$
0	0	0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	1	0	1
1	0	0	0	1	0	1	0	1	0
1	1	0	0	0	0	1	1	0	0
0	-	1	0	1	0	1	0	1	0
1	-	1	0	0	0	1	0	1	0
-	0	0	1	0	1	0	1	0	1
-	1	0	1	0	0	0	1	0	1
-	-	1	1	0	0	0	0	0	0

Table 5.3: Solutions of Equation 5.6.

Except for the highlighted cases, these tables show exactly the behavior we expect from an RS flip-flop, not taking into account issues of metastability and oscillation that occur in the hardware. The highlighted lines show a behavior similar to oscillation.

This means that using PowerRPDEVS and its logic library the simulation of most of the RS flip-flop's behavior is possible without having to add latency. Only when both R and S transition from `1` to `0` at once, simulation could lead to an infinite loop in the λ step and the simulator might abort simulation after a number of λ iterations. However, this is exactly the case in which a real flip-flop would also become unstable.

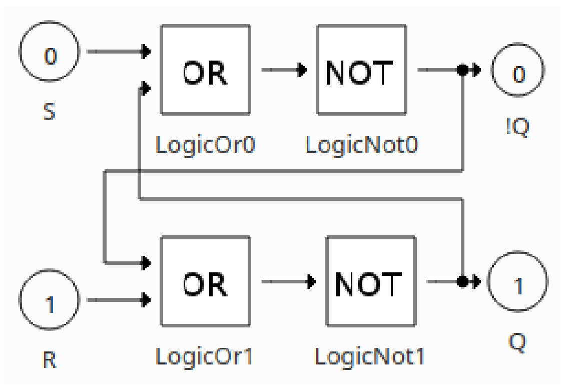


Figure 5.2: static RS flip-flop – model A

Thus, it was tried to model the static RS flip-flop in PowerRPDEVS without an effort of breaking the algebraic loop. Two models of the RS flip-flop were created: model A (Figure 5.2) is a coupling of two `OR` gates and two `NOT` gates and model B (Figure 5.3) is composed of two `NOR` gate couplings. The `NOR` gate couplings are composed of an `OR` and a `NOT` gate (see Figure 5.4). The two models differ slightly in behavior. While model A starts to oscillate when both inputs transition from `1` to `0` at the same time – the same behavior that was shown and highlighted in Table 5.3 –, model B does not.

The simulation results in Figure 5.5 show the results of model B. At the end of the simulation both inputs simultaneously transition from `1` to `0`. The results of model A look the same except the simulation is aborted at that point after a certain number of λ steps. The input sequences also enter the invalid state ($R = S = $ `1`) before that, but do not transition to `0` at the same time. Transitions with only one of S or R changing back to `0` are safe in both models.

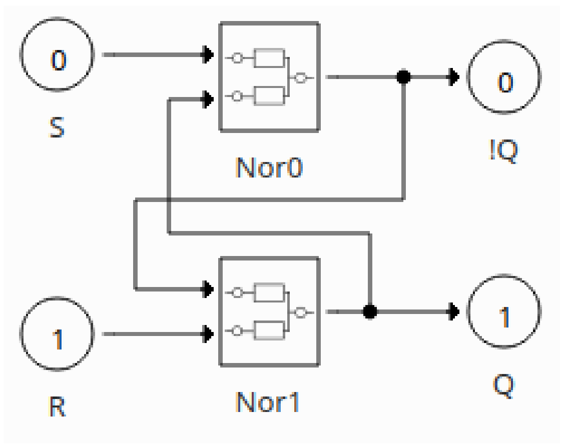


Figure 5.3: static RS flip-flop – model B

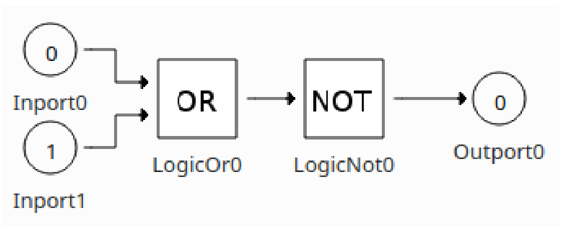


Figure 5.4: NOR gate coupling of model B

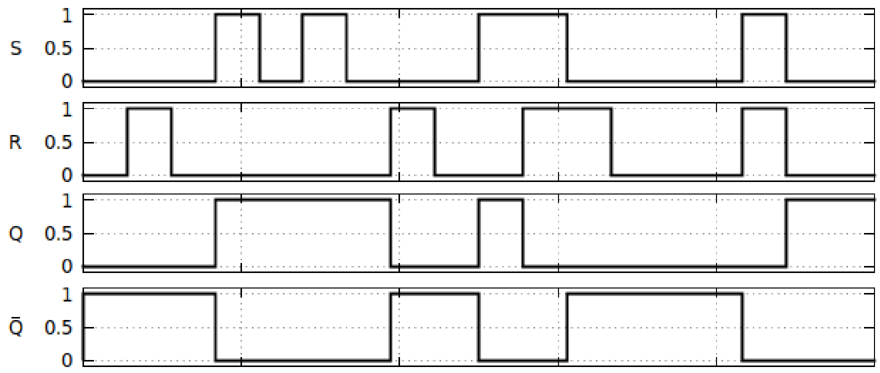


Figure 5.5: Simulation results of static RS flip-flop

The reason why model B does not oscillate is that the model is processed differently by the simulation algorithm. In PowerRPDEVS, although the model could be evaluated in parallel, the simulator evaluates the atomics' and couplings' λ and δ steps sequentially. In model B the two `NOR` couplings are the only blocks on the top level of the model and if both receive an input, they both have to call λ. As this happens sequentially, one `NOR` gate's λ is called and its output is immediately reused by the second `NOR` gate λ call. This leads to a behavior equivalent to the recurrences Equation 5.4 or Equation 5.5. We know that these cannot oscillate.

If the evaluation of both `NOR` gates is done in parallel – which could be the case in later versions of PowerRPDEVS – the behavior changes and none of the two `NOR` gates could then reuse an output of the other of the current λ iteration. This would lead to the same behavior as in model A. In model A, when both `OR` gates receive an input, first both `OR` gates produce their output, which activates the two `NOT` gates. Then, in the next λ iteration step, the two `NOT` gates calculate their output, reactivating the two `OR` gates for the next λ iteration step. This leads to the recurrence in Equation 5.6 which can oscillate.

The instability problem of course can be tackled by introducing a delay to break the algebraic loop, like Junglas did, but in practice it is probably a better idea to prevent the illegal input state for the static RS flip-flop completely with combinatoric circuitry that precedes the RS flip-flop itself, e.g. by using *Data* (D) and *Enable* (EN) as inputs and wiring them to the former S and R inputs as within a static D flip-flop:

$$S = D \land EN \tag{5.7}$$
$$R = \neg D \land EN \tag{5.8}$$
$$\tag{5.9}$$

5.2 Triggered RS Flip-Flop

A triggered RS flip-flop has an additional clock input (CLK). Whenever CLK triggers (when an edge is detected at the input), the values R and S determine the value of the stored bit as with the static RS flip-flop. The stored bit is not updated when there is no edge detected at the CLK input even if R or S change.

This flip-flop still has the problematic invalid state R = S = `1` but this input is now allowed intermittently as long as CLK does not trigger.

Junglas used a Trigger block in Simulink which he added to the subsystem of the static RS flip-flop. In Modelica, he wired the CLK input to a falling edge block and its output to `AND` gates that can deactivate the other inputs S and R (similar to the PowerRPDEVS model shown in Figure 5.7).

In this model, the falling block would detect falling edges on its input CLK and output a corresponding signal (e.g. `1` for edge detected and `0` for no edge detected). This means that when an edge is detected the falling block needs to hold the `1` for a certain time period whereas the falling edge at the input exists only for an instant of time.

An equivalent model was implemented in PowerRPDEVS (see Figure 5.7) with a custom *falling* block (see Figure 5.6) that behaves as described above and that will hold `1` at its output for an adjustable time period. Within this time period, the flip-flop accepts changes on its inputs S and R which allows for glitches as can be seen in the results in Figure 5.9 at $t = 2.5$.

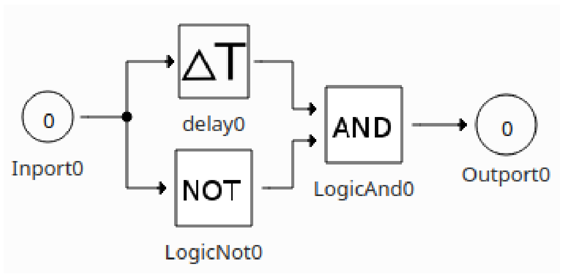


Figure 5.6: Coupling that detects falling edges

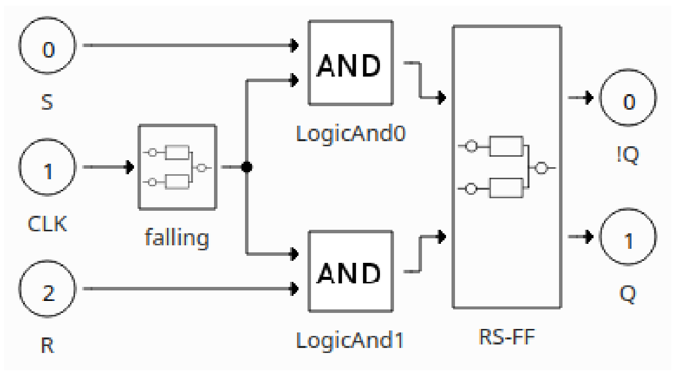


Figure 5.7: Triggered RS flip-flop in PowerRPDEVS – model A

In theory, the triggered RS flip-flop should adopt its inputs S and R only exactly when the falling edge occurs. It is easy to see from Figure 5.7 and Figure 5.6 that this behavior cannot be achieved by this simple model, as the output of the *falling* block will be high for a certain time period. This can also be seen in the simulation results in Figure 5.9 where input signal changes in this time frame produce glitches at the outputs of the RS flip-flop

It should be noted that this custom *falling* block behaves the same as the built-in LogicEdgeDetect block, so they yield the same results. The custom block was chosen to show how the same functionality can be implemented using a coupling.

To match the intended behavior the RS flip-flop was also implemented differently (see Figure 5.8) and for that the LogicTriggeredSampling block was created (see section 4.3).

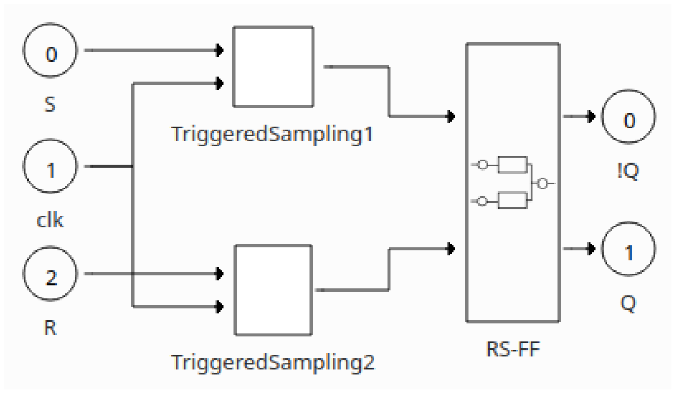


Figure 5.8: Triggered RS flip-flop in PowerRPDEVS – model B

In Figure 5.7 the two `AND` gates were used as switches together with the *falling* block. These blocks were replaced by two TriggeredSampling blocks that are wired to CLK on their second inputs and forward the signals on their first inputs when there is an edge on the second input. This way, an event is only triggered in the following static RS flip-flop when the clock input carries an edge.

The results in Figure 5.10 are as expected and do not show the glitches as in the results of model A (Figure 5.9).

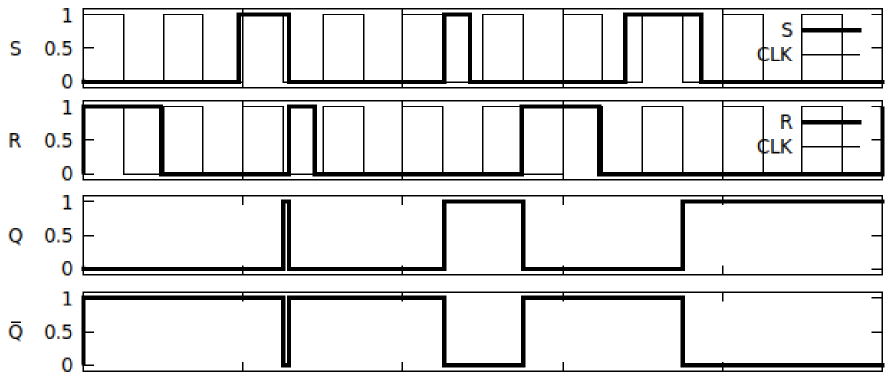


Figure 5.9: Simulation results of triggered RS flip-flop – model A

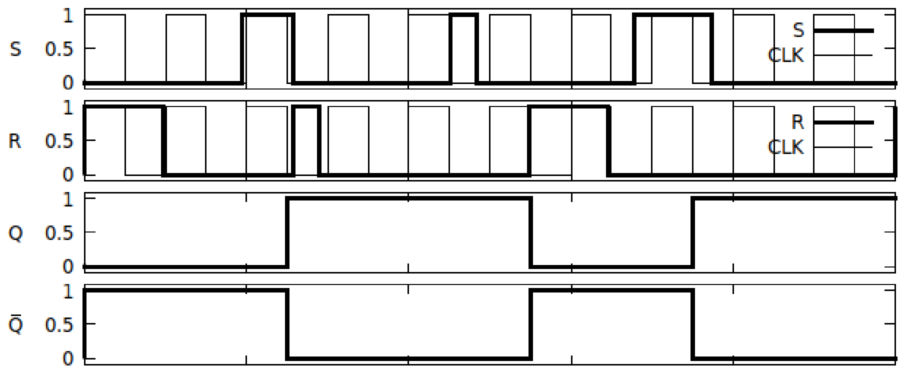


Figure 5.10: Simulation results of triggered RS flip-flop – model B

5.3 D Flip-Flop

The D flip-flop has two inputs: Data (D) and Clock (CLK). Whenever CLK triggers, the bit from the D line is stored in the flip-flop. It usually has only one output Q which holds the value of the stored information bit. This design avoids the invalid state of $R = S = 1$.

The D flip-flop that was implemented here is based on the triggered RS flip-flop model B from section 5.2. The D input of the D flip-flop model is on the one hand wired to the S input of the triggered RS flip-flop and on the other hand negated and wired to the R input of the triggered RS flip-flop. This matches the implementation of Junglas in [Jun16]. The model is shown in Figure 5.11. The *Tr-RS-FF* coupling is the before mentioned triggered RS flip-flop.

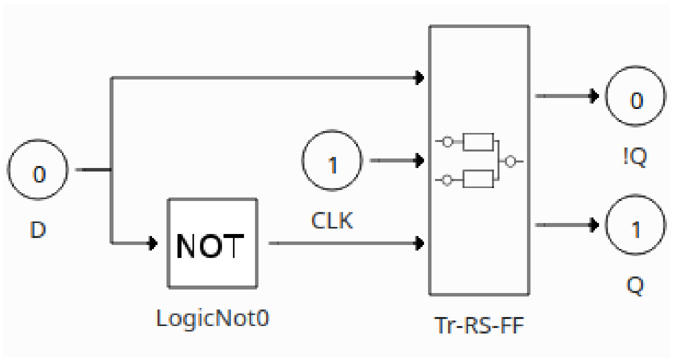


Figure 5.11: D flip-flop in PowerRPDEVS

The results in Figure 5.12 show the D and CLK inputs as well as the Q output ($\overline{\text{Q}}$ is omitted because the invalid state cannot occur and thus $\overline{\text{Q}} = \neg\text{Q}$). These results are as expected.

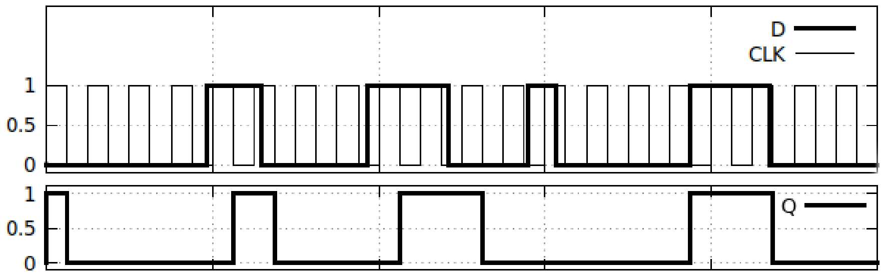


Figure 5.12: Simulation results of D flip-flop

It must be noted that, when its *limit* setting is set to *left*, the *LogicTriggeredSampling* library block already behaves like a D flip-flop with only a Q output. The simulation results are the same when a *LogicTriggeredSampling* block is used instead of the D flip-flop model shown in Figure 5.11. The reason is that the *LogicTriggeredSampling's* output only changes when its second input detects an edge and it always changes to the last received value from its first input excluding the value that it currently receives (when set to *left limit*).

5.4 Shift Register

A shift register can be created by connecting the output of a D flip-flop with the input of another one and forming an arbitrarily long chain, in which all the CLK inputs are connected to the same clock. Whenever CLK triggers, the bits stored in the flip-flops are propagated one flip-flop further through this chain. The first flip-flop saves its input which is the D input of the shift register. The value of the last flip-flop in the chain is lost in this process.

For this example, a shift register with three D flip-flops was implemented using three D flip-flops from before (see Figure 5.13).

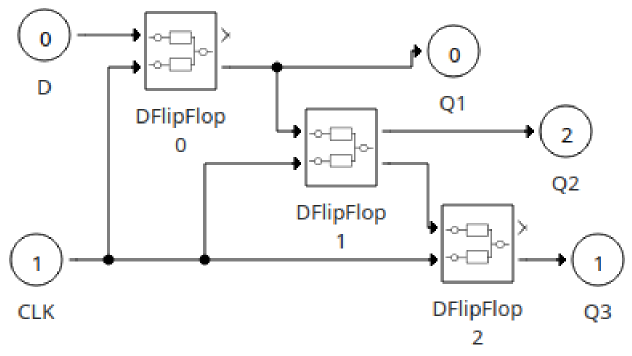


Figure 5.13: Shift register in PowerRPDEVS

The LogicTriggeredSampling atomics of *DFlipFlop0* are set to use the right limit (as before) and those of the other two flip-flops are set to left limit.

When all LogicTriggeredSampling are set to use the *right* limit the model does not show the behavior expected from a shift register. The input signal at the first D flip-flop would propagate through the whole circuit at once. It is important to note that this is correct behavior when no intrinsic delays are assumed (neither the wires nor the individual logic elements delay the signal). So, when modeled without delays the model does not resemble the expectation that the stored bits propagate by only one flip-flop at a time.

The LogicTriggeredSampling blocks in the second and third flip-flop in the shift register impose an infinitesimal delay on the data lines by propagating only the left limit of their inputs when they are triggered by the CLK input. The same effect can be achieved instead by placing generic Delay blocks between the D flip-flops to model a propagation delay of the previous flip-flop or a data line delay. The delay should be small compared to the clock's period. The LogicTriggeredSampling block was chosen because it works with arbitrary clock periods and to keep the modeling consistent with previous efforts of not introducing latency.

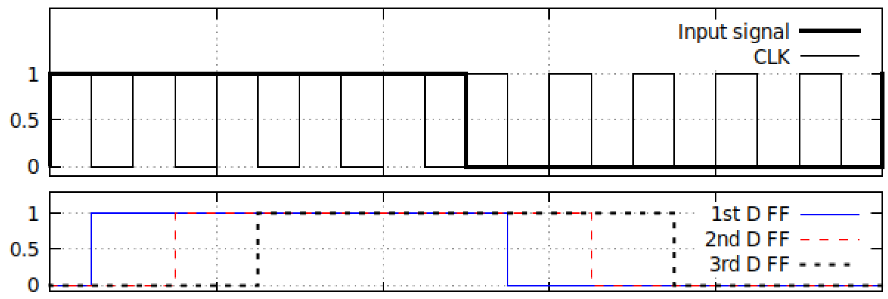


Figure 5.14: Simulation results of the shift register

The example shift register's D flip-flops were initialized with `0`'s and the input signal is at first `1` and later `0` as can be seen in Figure 5.14. It is also easy to see how the individual D flip-flop's adopt the `1` after one another and then reset to `0` in a first in – first out (FIFO) manner when the input signal changes.

5.5 Serial Bus with Collision Resolution

5.5.1 Buses in Boolean logic and 9-valued logic

To show some of the capabilites of the library regarding multivalue logic, an additional example is included here, implementing an asynchronous serial bus with collision resolution.

Buses are data lines that can be read and written by multiple clients. When two or more clients write to the bus at the same time, some mechanism needs to be in place to detect and/or resolve collisions in some way. Collisions are simultaneous writes to the data line of two or more clients.

Implementing a model of a bus using only Boolean logic is not straight-forward. A logic element that is connected to the bus must be able to indicate if it wants to write the bus or not. So the atomic/coupling that implements the bus could have two wires per client (one for the data D_i, and one for indicating if the data shall be written WR_i). The bus could then `AND` the individual D_i and WR_i and wire the results to an `OR` gate. But this is a bad abstraction for real bus systems that probably do not need an extra wire. Moreover, the case of multiple clients writing to the bus needs to be handled explicitly.

Using the PowerRPDEVS logic library implementing a bus is easier and more related to the physical implementation. Only one wire is needed to the bus because the connected nodes can signal that they do not want to write by setting their output to `Z`.

5.5.2 Example

The given example will be a simple imitation of a Controller Area Network (CAN) bus. Specifically, these are the attributes of the bus that shall be modeled:

1. The bus has a weak high and a strong low value. In practice, this means that simultaneous writes of ones and zeros will be resolved to zero. The data line of the bus is weak high (`H`) by default.
2. Nodes utilize Carrier-Sense Multiple Access/Collision Resolution (CSMA/CR) so they do not send when they perceive another node sending, and they listen to the bus while they write and disconnect if the value they read does not correspond to the value they have written.
3. Nodes need to use a fixed message format that begins with a strong value and then the ID of the node. This is used for *arbitration* as in CAN. Arbitration is the phase during message transmission when the node determines if it is allowed to write its message to the bus. Its ID might be overwritten by a lower ID, which would mean it has lost arbitration and needs to stop transmission. Only one node can win arbitration and write its whole message on the bus.

The implementation of the above points is described in the following. Unlike CAN, the example will only use one wire.

1. The bus is implemented with a LogicResolve block. The high value is `H` and the low value is `0` throughout the model. A LogicConstant with the value `H` is connected to the input of the Bus.
2. For Carrier-Sense Multiple Access (CSMA), the nodes need circuitry for detecting if another node is writing when they want to write, so they can wait until that other node is finished (this is not implemented in the example). For CSMA/CR they additionally need a collision detection circuit that is active during the arbitration phase and stops the nodes with lower priority from overwriting the node with the higher priority.
3. The message format used is described in Table 5.4.

`0`	8 bit ID	15 bit data	`H`

Table 5.4: Message format of the bus

In the example model, shown in Figure 5.15, there are two senders, Sender1 and Sender2 as well as a Bus which is a LogicResolve block. The senders output ports are wired to

Sender1 wants to send	`H0HH000H0H0H0H0H0H0HHHHHHH0H00`
Sender2 wants to send	`H0HH000000H0H0H0H0H0H0H0H0H0HH`

Table 5.5: Data in the bus example

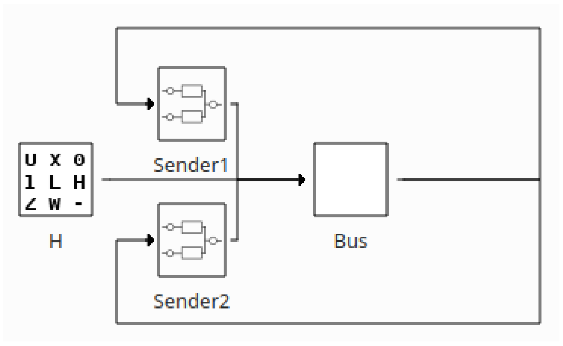


Figure 5.15: Serial bus model

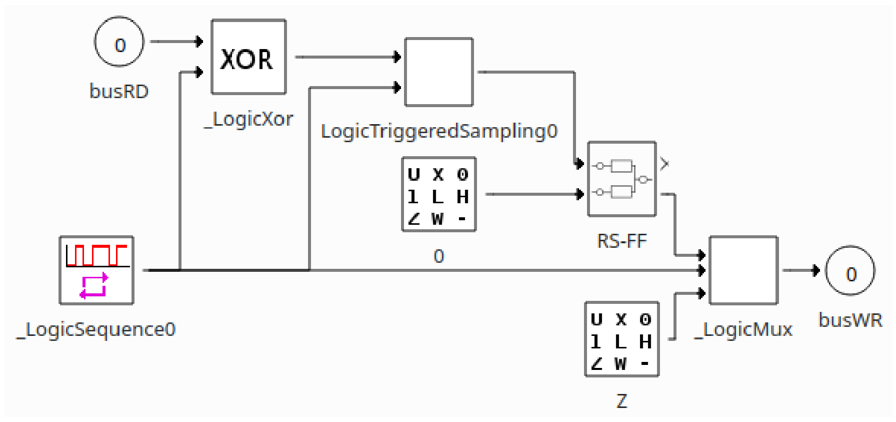


Figure 5.16: Sender coupling (Sender1 and Sender2 in Figure 5.15)

the input port of the LogicResolve block. The output port of the LogicResolve block is wired back to the sender nodes, as they also need to read from the bus.

Both senders are configured to start sending at the same time starting with an `H` followed by the message. The two messages have different ID's which will cause the node with lower priority to shut down during transmission wheras the node with higher priority continues to send without even recognizing that another node started a transmission.

Sender1 and Sender2 are identical, they only differ in their message parameter. On the one hand they need to have different IDs, so the collision resolution works, on the other hand the actual data in the message also differs. The coupling is shown in Figure 5.16. Table 5.5 shows the messages the two senders want to send.

In the coupling constituting a Sender, shown in Figure 5.16, the message transmission is implemented by a LogicSequence block. The rest of the circuit is for collision resolution. The `XOR` checks whether the written value corresponds to the value on the bus. If it does not, the flip-flop is set. Setting the flip-flop will toggle the MUX and write `Z` to the bus. In this proof-of-concept, the flip-flop is never reset, so the stopped node could never restart sending. Additional circuitry could detect the end of a message on the bus and reset the flip-flop to retry transmission.

The flip-flop needed to be decoupled by a LogicTriggeredSampling block, because initially, when the LogicSequence block changes its output and it is different from the value on the bus, the `XOR` will always produce a `1`. But at this point the logic values that will be applied to the bus were not resolved yet. So, only the last result of the `XOR` must be considered. This is done by setting the LogicTriggeredSampling atomic's *limit* setting to *left* and its *edge* setting to *both*, so every change in the output of LogicSequence triggers a new evaluation.

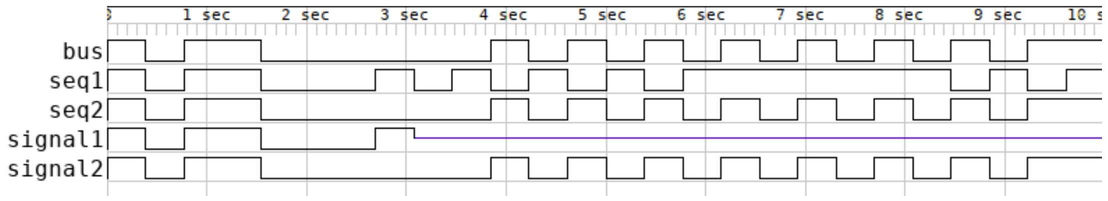


Figure 5.17: Result of simulation – Serial bus

In the results in Figure 5.17 we see the sequences that Sender1 and Sender2 want to send (`seq1` and `seq2`), the signals they are actually sending (`signal1` and `signal2`) and the signal that is on the bus (`bus`).

The results in Figure 5.17 are as expected. The ninth bit of the message, which is the last ID bit, is the first that differs for Sender1 and Sender2. Since the ID of Sender2 is lower than the ID of Sender1, Sender2 wins the arbitration. Sender1 recognizes this and shuts down its transmission. In the end the signal on the bus is identical to the message of Sender2, so every node connected to the bus can read Sender2's message directly as if Sender1 never had started sending.

CHAPTER 6

Conclusion and Outlook

We compared the DEVS and RPDEVS model of the simplest Boolean logic component, the `NOT` gate. The RPDEVS model turned out to be less complex than the corresponding DEVS model. This can be traced back to the capability of RPDEVS to model *true* Mealy behavior and to the consolidation of the different state transition functions.

Based on the work of Junglas, it was tested how the logic library that was implemented for this thesis performs when its elements are combined in feedback loops. Junglas' work-arounds for the particular issues that arose in Simulink or Modelica were pointed out. He had to introduce memory elements (delays) into the circuit and sometimes in a clever way to keep the timings of the model as expected. Such delays are actually there in physical implementations of the models, but might be unwanted in simulation.

The issues of the simulation with PowerRPDEVS and the roots of these issues were discussed. First and foremost, algebraic loops that form whenever there is a feedback loop of logic gates can cause infinite loops in the λ step of the simulation. The PowerRPDEVS simulator detects these algebraic loops and aborts the simulation after a certain number of λ iterations. The algebraic loop in the static RS flip-flop as implemented in this thesis can also oscillate in the λ step and the simulation is aborted. Therefore, such models might only be operated within certain specifications, e.g. the inputs of the static RS flip-flops may not transition from `1` to `0` simultaneously in one of the models presented. Also, because the PowerRPDEVS logic library elements have no propagation delay, the user must recognize where delays have to be included into the model for it to work properly. This was the case with the shift register which can only work with D flip-flops that delay their outputs.

Finally, an example was given demonstrating the multivalue capabilities of the PowerRPDEVS logic library that are owed to the implementation of the nine logic values of IEEE standard 1164-1993. A serial bus with collision resolution similar to the CAN bus was modeled with two senders starting transmission of their messages simultaneously.

By choosing appropriate logic values (one weak, the other strong) one can avoid signal resolutions resulting in the unknown logic value `X` and define the dominant signal on the bus. Further, a protocol had to be implemented, causing senders which do not read their own output on the bus to shut down.

In future work, the implementation of the PowerRPDEVS logic library could be made more flexible by merging the classes `LogicGateUnary` and `LogicGateBinary` and by restructuring the template arguments. The function that is applied by a logic gate class could be given to the constructor as some function object or maybe as a truth table instead of the current way of having it as template argument. This would allow to implement a configurable logic gate for which the user could enter an arbitrary truth table as a parameter. The implementation of further conversion blocks will be necessary to effectively use the other parts of the PowerRPDEVS library together with the logic library, e.g. a Schmitt trigger and analog-digital/digital-analog converters.

APPENDIX A

Code of Implementation

A.1 DEVSLogicMessage

A.1.1 DEVSLogicMessage.h

```
#ifndef DEVSLOGICMESSAGE_H
#define DEVSLOGICMESSAGE_H

#include <cctype>
#include <typeinfo>
#include <DEVSMessage.h>

#include "logic.h"

class DEVSLogicMessage : public DEVSMessage
{
public:
  char logicval;
  // constructors:
  DEVSLogicMessage() : DEVSMessage()
  {
    logicval = 'U';
  }
  DEVSLogicMessage(char v, int _index = 0)
      : DEVSMessage(_index),
        logicval(v)
  {
  }
  DEVSLogicMessage(const DEVSLogicMessage &msg)
      : DEVSMessage(static_cast<DEVSMessage>(msg)),
        logicval(msg.logicval)
  {
}
DEVSLogicMessage(std::string value_str) {
          index = 0;
          parseValueFromString(value_str);
      }

char getChar() const
{
  return logicval;
}
double getDouble() const
{
  return getChar() == '1' ? 1 : 0;
}

bool getBool() const
{
  return getChar() == '1' ? true : false;
}
void set(char c)
{
  logicval = c;
}

virtual int getInt() {
          switch(logicval)  {
              case '1':
              case 'H':
                  return(1);
                  break;
              default:
                  return(0);
                  break;
          }
      }

      virtual bool parseValueFromString(std::string value_str) {
          // string must either be a single character 'c' (the
              ↪ logic-value)
          // or it also gives the index: 'index:c', e.g.: '2:H', or
              ↪  '0:U'
          size_t pos=0;
          if(std::string::npos != (pos=value_str.find(':')) ) {
              index = std::stoi(value_str.substr(0,pos));
              pos++;
          } else {
              pos = 0;
          }
          pos = value_str.find_first_of("01LHUXZW-lhuxzw",pos);
```

```
            if(std::string::npos != pos) {
                logicval = toupper(value_str[pos]);
            }
            return(valid(logicval));
        }

  virtual DEVSMessage *getCopy() const
  {
    return new DEVSLogicMessage(*this);
  }
  virtual std::string toString() const
  {
    return std::string(1, getChar());
  }

  virtual bool operator==(const DEVSLogicMessage &msg) const
  {
    if (index != msg.index)
    {
      return false;
    }
    return logicval == msg.logicval;
  }

  virtual bool operator==(const DEVSMessage &msg) const
  {
    if (typeid(msg) != typeid(*this))
      return false;
    return ((*this) == ((DEVSLogicMessage &)msg));
  }

  virtual ~DEVSLogicMessage() {}

  static bool valid(char v)
  {
    return logicfunction_value_valid(v);
  }

  static std::map<int, char> getLogicVector(const char *s)
  {
    std::map<int, char> output;
    std::string vec(s);

    auto bra = find(vec.cbegin(), vec.cend(), '[');
    if (bra != vec.cend())
    {
      auto ket = find(vec.cbegin(), vec.cend(), ']');
      vec.erase(vec.cbegin(), bra + 1);
      vec.erase(ket);
    }
    auto it = find_if(vec.cbegin(), vec.cend(), isgraph);
    int i = 0;
    while (it != vec.cend())
    {
      output[i++] = *it;
      it = find_if(it + 1, vec.cend(), isgraph);
    }

    return output;
  }
};

#endif
```

A.2 Logic functions of IEEE 1164

A.2.1 stdlogic1164.h

```
#ifndef ASSERT
#error "don't include this file! include logic.h"
#endif

#define symbols std::string("UX01ZWLH-")

/* the following *_table's were taken from IEEE 1164
 * and translated to C arrays */

inline int logicfunction_index(char a)
{
  auto n = symbols.find(a);
  ASSERT(n >= 0 && n <= 8);
  return int(n);
}

constexpr bool logicfunction_value_valid(char a)
{
  return a == 'U' || a == 'X' || a == '0' ||
         a == '1' || a == 'Z' || a == 'W' ||
         a == 'L' || a == 'H' || a == '-' ;
}

/* a char that meets the following condition */
#define NOLOGICVAL ' '
static_assert(logicfunction_value_valid(NOLOGICVAL) == false, "");

inline char logicfunction_check_undefined_input(char a)
{
  constexpr char undefined_input = 'U';
  if(logicfunction_value_valid(a)){
    return a;
  }else{
    return undefined_input;
  }
}

inline char logicfunction_resolution(char a, char b, char def)
{
  static char resolution_table[9][9] = {
    { 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' },
    { 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' },
    { 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' },
    { 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' },
    { 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' },
    { 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' },
    { 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' },
    { 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' },
    { 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' }};
  ASSERT(logicfunction_value_valid(a) && logicfunction_value_valid(b)
      ↪ );
  return resolution_table[logicfunction_index(a)][logicfunction_index
      ↪ (b)];
}

inline char logicfunction_and(char a, char b)
{
  static char and_table[9][9] = {
    { 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' },
    { 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' },
    { '0', '0', '0', '0', '0', '0', '0', '0', '0' },
    { 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' },
    { 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' },
    { 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' },
    { '0', '0', '0', '0', '0', '0', '0', '0', '0' },
    { 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' },
    { 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' }};

  ASSERT(logicfunction_value_valid(a) && logicfunction_value_valid(b)
      ↪ );
  return and_table[logicfunction_index(a)][logicfunction_index(b)];
}

inline char logicfunction_or(char a, char b)
{
  static char or_table[9][9] = {
    { 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U' },
    { 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' },
    { 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' },
    { '1', '1', '1', '1', '1', '1', '1', '1', '1' },
    { 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' },
    { 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' },
    { 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' },
    { '1', '1', '1', '1', '1', '1', '1', '1', '1' },
    { 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' }};
  ASSERT(logicfunction_value_valid(a) && logicfunction_value_valid(b));
  return or_table[logicfunction_index(a)][logicfunction_index(b)];
}

inline char logicfunction_xor(char a, char b)
{
  static char xor_table[9][9] = {
    { 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' },
    { 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' },
    { 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' },
    { 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' },
    { 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' },
    { 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' },
    { 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' },
    { 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' },
    { 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' }};

  ASSERT(logicfunction_value_valid(a) && logicfunction_value_valid(b));
  return xor_table[logicfunction_index(a)][logicfunction_index(b)];
}

inline char logicfunction_not(char a)
{
  static char not_table[9] = {
    'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' };

  ASSERT(logicfunction_value_valid(a));
  return not_table[logicfunction_index(a)];
}

inline char logicfunction_nand(char a, char b)
{
  ASSERT(logicfunction_value_valid(a) && logicfunction_value_valid(b));
  return logicfunction_not(logicfunction_and(a,b));
}

inline char logicfunction_nor(char a, char b)
{
  ASSERT(logicfunction_value_valid(a) && logicfunction_value_valid(b));
  return logicfunction_not(logicfunction_or(a,b));
}

inline char logicfunction_nxor(char a, char b)
{
  ASSERT(logicfunction_value_valid(a) && logicfunction_value_valid(b)
     ↪ );
  return logicfunction_not(logicfunction_xor(a,b));
}

inline char logicfunction_to_ux01(char a)
{
  ASSERT(logicfunction_value_valid(a));
  switch(a){
  case '0':
  case 'L':
    return '0';
  case '1':
  case 'H':
    return '1';
  case 'U':
    return 'U';
  default:
    return 'X';
  }
}

inline char logicfunction_to_x01(char a)
{
  ASSERT(logicfunction_value_valid(a));
  switch(a){
  case '0':
  case 'L':
    return '0';
  case '1':
  case 'H':
    return '1';
  default:
    return 'X';
  }
}

inline bool logicfunction_rising(char _old, char _new)
{
  ASSERT(logicfunction_value_valid(_old) && logicfunction_value_valid
     ↪ (_new));
  if(logicfunction_to_x01(_old) == '0' && logicfunction_to_x01(_new)
     ↪ == '1')
    return true;
  else
    return false;
}

inline bool logicfunction_falling(char _old, char _new)
{
  ASSERT(logicfunction_value_valid(_old) && logicfunction_value_valid
     ↪ (_new));
  if(logicfunction_to_x01(_old) == '1' && logicfunction_to_x01(_new)
     ↪ == '0')
    return true;
  else
    return false;
}

inline char logicfunction_to_bit(char a, char def)
{
  char b = logicfunction_to_ux01(a);
  if(b == '0' || b == '1'){
    return b;
  }else{
    ASSERT(logicfunction_value_valid(def));
    return def;
  }
}
```

A.3 LogicGateUnary and LogicGateBinary

A.3.1 LogicGate.h

```
#pragma once
#include "simulator.h"
#include <cassert>
#include <numeric>
#include <stdexcept>
#include <vector>

static inline void debug_output(const char *name, int myself, double
   ↪ t, const char *text,
        const std::map<int, char> &outputs)
{
  std::string s(outputs.size(), ' ');
  for (const auto &a : outputs)
    if (DEVSLogicMessage::valid(a.second))
      s[a.first] = a.second;
  printLogAtLevel(_DEBUG_ROUGH, "%s, %d, t=%G: %s '%s' \n", name,
     ↪ myself, t, text, s.c_str());
}

```

```
static inline void override_output(const Simulator &sim, std::map<int, char> &outputs,
           const std::map<int, char> &init)
{
  // when the outputs are set replace all 'U's
  // with the initial values given (if any)

  // iterating over init, so to not mess with iterator validity in outputs
  for (const auto &init_pair : init) {
    std::map<int, char>::const_iterator val_it;
    if ((val_it = outputs.find(init_pair.first)) != outputs.cend() &&
        val_it->second == 'U') {
      outputs[init_pair.first] = init_pair.second;
    }
  }
}

static inline void write_output(Simulator &sim, const std::map<int, char> &outputs,
        const std::map<int, char> &old = std::map<int, char>())
{
  for (const auto &index_output : outputs) {
    DEVSLogicMessage C(index_output.second);
    C.index = index_output.first;
    try {
      // write if new output is different from old
      if (old.at(C.index) != C.getChar())
        sim.add_output(&C, 0);
    } catch (std::out_of_range &e) {
      // and write if no corresponding old output saved
      sim.add_output(&C, 0);
    }
  }
}

using LogicGateTernaryFunction = char (*)(char, char, char);
using LogicGateBinaryFunction = char (*)(char, char);
using LogicGateUnaryFunction = char (*)(char);

/* Class templates that implement the functionality of logic gates
 *
 * These logic gates don't have a state really, the variable last_inputs
 * is needed to comply with the RPDEVS formalism. Apart from the old input
 * values no data is stored.
 *
 *     class LogicGateResolver
 *     implements an abstraction layer that receives inputs from the
 *     simulator class and resolves them using a resolution function.
 *     It also saves old inputs. The resolved inputs can be accessed
    ↪ via
 *     resolve_lambda from lambda or resolve_delta from delta.
 *
 *     class LogicGateBinary
 *
 *     implements a logic gate that applies a binary function to an
 *     arbitrary number of logic inputs. As the order of these is not
 *     predetermined this binary function should be associative and
 *     commutative.
 *     The gate performs signal resolution on signals that were
    ↪ received
 *     on the same input and the same index using LogicGateResolver.
 *
 *     class LogicGateUnary
 *
 *     implements a logic gate that applies a unary function to a
 *     single logic vector input.
 *     The gate performs signal resolution on signals that were
    ↪ received
 *     on the same input and the same index using LogicGateResolver.
 *
 *
 * FIXME: there are hard-coded 'U's that should be removed either by
    ↪ adding
 *        a template parameter for undefined values or maybe a
    ↪ constructor
 *        with a respective parameter
 *
 * FIXME: there are hard-coded '0's that are used as default value
    ↪ for
 *        signal resolution.
 */

template <typename InputIterator, typename AccessFunc, typename
   ↪ ResolveFunc>
inline auto logicgate_resolve(InputIterator begin, InputIterator end,
   ↪ AccessFunc access,
              ResolveFunc resolve, char resolve_default = '0') -> char
{
  assert(begin != end);
  auto val = access(*begin);
  for (++begin; begin != end; ++begin) {
    val = resolve(val, access(*begin), resolve_default);
  }
  return val;
}
```

```

template <typename InputIterator, typename ResolveFunc>
inline auto logicgate_resolve(InputIterator begin, InputIterator end, ResolveFunc resolve,
            char resolve_default = '0') -> char
{
  return logicgate_resolve(begin, end, [](char x) -> char { return x; }, resolve,
         resolve_default);
}

template <LogicGateTernaryFunction Resolve>
class LogicGateResolver {
public:
  // outer map indices are DEVSMessage index member variable
  // vector indices are port indices
  // inner map indices are pairs of source and source port
  using saved_inputs_map =
      std::map<int, std::vector<std::map<std::pair<unsigned, unsigned>, char>>>;
  using map_type = saved_inputs_map::mapped_type::value_type;
  using map_pair = map_type::value_type;
  using resolved_inputs_map = std::map<int, std::vector<char>>;

protected:
  saved_inputs_map last_inputs;

  void read_input(Simulator &sim, saved_inputs_map &inputs, double t) const
  {
    if (!sim.input_bag_empty()) {
      DEVSMessage *in_msg;
      unsigned port;

      while (sim.pop_input(&in_msg, port)) {
        DEVSLogicMessage *lmsg = dynamic_cast<DEVSLogicMessage *>(in_msg);
        std::pair<unsigned, unsigned> src{sim.pop_src, sim.pop_src_port};
        if (lmsg == nullptr) {
          printLogAtLevel(_ERROR,
              "%s, %d, t=%G: input "
              "is not logic value \n",
              sim.name, sim.myself, t);
          continue;
        }
        inputs[lmsg->index].resize(sim.n_in_ports);
        inputs[lmsg->index][port][src] = lmsg->getChar();
      }
    }
  }
  resolved_inputs_map calculate_resolved_inputs(const
    ↪ saved_inputs_map &inputs,
                  const Simulator &sim, double t) const
  {
    resolved_inputs_map resolved_inputs;
    for (const auto &index_vec : inputs) {
      /* resolve all input signals so we have only one value
         per port save them in resolved_inputs so we can
         iterate over them easily */
      std::vector<char> resolved;
      int uninit = 0;
      for (auto &srcmap : index_vec.second) {
        char val = 'U';
        if (!srcmap.empty())
          val = logicgate_resolve(
              srcmap.cbegin(), srcmap.cend(),
              [](const map_pair &p) { return p.second; }, Resolve);
        else
          uninit++;
        resolved.push_back(val);
      }
      if (uninit != 0)
        printLogAtLevel(_ERROR,
            "%s, %d, t=%G: some ports of the logic "
            "gate are not initialized on index %d\n",
            sim.name, sim.myself, t, index_vec.first);
      resolved_inputs[index_vec.first] = std::move(resolved);
    }
    return resolved_inputs;
  }

public:
  resolved_inputs_map resolve_delta(Simulator &sim, double t)
  {
    read_input(sim, last_inputs, t);
    return calculate_resolved_inputs(last_inputs, sim, t);
  }

  resolved_inputs_map resolve_lambda(Simulator &sim, double t) const
  {
    auto inputs = last_inputs;
    read_input(sim, inputs, t);
    return calculate_resolved_inputs(inputs, sim, t);
  }

  resolved_inputs_map old_inputs(const Simulator &sim, double t)
    ↪ const
  {
    return calculate_resolved_inputs(saved_inputs_map{}, sim, t);
  }
};

/* RPDEVS implementation of a generic binary combinatorial logic gate
 * (i.e. AND-gate, NOR-gate, ...)
 *
 * F is the ternary function that the logic gate should perform
 * Resolve is the binary function that resolves two signals when
 * they arrive at the same port and index (the third parameter is
 * a default value which is used when resolution is not possible) */
template <LogicGateBinaryFunction F, LogicGateTernaryFunction Resolve
    ↪ >
class LogicGateBinary : public LogicGateResolver<Resolve> {
  std::map<int, char> last_outputs, init_outputs;
  using resolved_inputs_map = typename LogicGateResolver<Resolve>::
      ↪ resolved_inputs_map;

  void calculate_output(const resolved_inputs_map &inputs, std::map<
      ↪ int, char> &outputs) const
  {
    for (const auto &index_vec : inputs) {
      if (index_vec.second.empty()) {
        throw std::runtime_error("LogicGateBinary::calculate_output "
               "encountered an unexpected condition");
      }
      outputs[index_vec.first] =
          std::accumulate(index_vec.second.cbegin() + 1, index_vec.
              ↪ second.cend(),
              index_vec.second[0], F);
    }
  }

public:
  inline void init(const std::map<int, char> &output_init)
  {
    init_outputs = output_init;
  }

  inline double ta() const
  {
    return INF;
  }

  inline void delta(Simulator &sim, double t)
  {
    auto resolved_inputs = this->resolve_delta(sim, t);
    calculate_output(resolved_inputs, last_outputs);
    override_output(sim, last_outputs, init_outputs);
  }

  inline void lambda(Simulator &sim, double t) const
  {
    auto resolved_inputs = this->resolve_lambda(sim, t);
    std::map<int, char> tmp_outputs;
    calculate_output(resolved_inputs, tmp_outputs);
    override_output(sim, tmp_outputs, init_outputs);
    write_output(sim, tmp_outputs, last_outputs);
    debug_output(sim.name, sim.myself, t, "lambda output",
       ↪ tmp_outputs);
  }
};

/* RPDEVS implementation of a generic unary combinatorial logic gate
 * (i.e. AND-gate, NOR-gate, ...)
 *
 * F is the unary function that the logic gate should perform
 * Resolve is the ternary function that resolves two signals when
 * they arrive at the same port and index (the third parameter is
 * a default value which is used when resolution is not possible) */
template <LogicGateUnaryFunction F, LogicGateTernaryFunction Resolve>
class LogicGateUnary : public LogicGateResolver<Resolve> {
  std::map<int, char> last_outputs, init_outputs;
  using resolved_inputs_map = typename LogicGateResolver<Resolve>::
     ↪ resolved_inputs_map;

  void calculate_output(const resolved_inputs_map &inputs, std::map<
     ↪ int, char> &output) const
  {
    for (const auto &index_vec : inputs) {
      if (index_vec.second.empty()) {
        throw std::runtime_error("LogicGateUnary::calculate_output "
               "encountered an unexpected condition");
      }
      output[index_vec.first] = F(index_vec.second[0]);
    }
  }

public:
  inline void init(const std::map<int, char> &output_init)
  {
    init_outputs = output_init;
  }
  inline double ta() const
  {
    return INF;
  }
  inline void delta(Simulator &sim, double t)
  {
    auto resolved_inputs = this->resolve_delta(sim, t);
    calculate_output(resolved_inputs, last_outputs);
    override_output(sim, last_outputs, init_outputs);
  }

  inline void lambda(Simulator &sim, double t) const
  {
    auto resolved_inputs = this->resolve_lambda(sim, t);
    std::map<int, char> tmp_outputs;

    calculate_output(resolved_inputs, tmp_outputs);
    override_output(sim, tmp_outputs, init_outputs);
    write_output(sim, tmp_outputs, last_outputs);
    debug_output(sim.name, sim.myself, t, "lambda output",
       ↪ tmp_outputs);
  }
};
```

A.4 AND gate

A.4.1 logic_and.h

```
//CPP:rpdevs_logic/logic_and.cpp
#if !defined logic_and_h
#define logic_and_h

#include "simulator.h"
#include "stdarg.h"

#include "DEVSLogicMessage.h"
#include "LogicGate.h"


class logic_and: public Simulator {
// Declare the state,
// output variables
// and parameters

  LogicGateBinary<logicfunction_and,logicfunction_resolution> gate;
public:
  logic_and(const char *n): Simulator(n) {};
  void init(double, ...);
  double ta(double t);
  void delta(double);
  void lambda(double);
  void exit();
};
```

```
#endif
```

A.4.2 logic_and.cpp

```
#include "logic_and.h"

// custom code:
void logic_and::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from
    ↪ the editor.
va_list parameters;
va_start(parameters,t);
printLogAtLevel(_INFO, "%s, %d, t=%G: Init \n",name,myself,t);

n_in_ports = int(va_arg(parameters,double));

printLogAtLevel(_INFO, "%s, %d, t=%G: n_in_ports = %d \n",name,myself
    ↪ ,t,n_in_ports);

const char *init_logic_vec = va_arg(parameters, char*);
gate.init(DEVSLogicMessage::getLogicVector(init_logic_vec));

va_end(parameters);
}
double logic_and::ta(double t) {
//This function returns a double.
printLogAtLevel(_INFO, "%s, %d, t=%G: ta \n", name,myself,t);
/* return double value with the time to the next internal event here.
    ↪ */

return gate.ta();
}
void logic_and::delta(double t) {
printLogAtLevel(_INFO, "%s, %d, t=%G: delta \n", name,myself,t);

gate.delta(*this,t);

}
void logic_and::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)

printLogAtLevel(_INFO, "%s, %d, t=%G: lambda\n",name,myself,t);

gate.lambda(*this,t);

}
void logic_and::exit() {
//Code executed at the end of the simulation.
printLogAtLevel(_INFO, "%s, %d: exit \n", name,myself);

}
```

A.5 LogicTriggeredSampling atomic

A.5.1 logic_trigg_sampl.h

```
//CPP:rpdevs_logic/logic_trigg_sampl.cpp
#if !defined logic_trigg_sampl_h
#define logic_trigg_sampl_h

#include "simulator.h"
#include "stdarg.h"

#include "DEVSLogicMessage.h"
#include "LogicGate.h"

class logic_trigg_sampl: public Simulator {
// Declare the state,
// output variables
// and parameters

bool left_limit;
enum{ RISING, FALLING, BOTH } which_edge;
double sigma;
std::map<int,char> init_output;
LogicGateResolver<logicfunction_resolution> resolver;
using resolved_inputs_map = LogicGateResolver<
    ↪ logicfunction_resolution>::resolved_inputs_map;
resolved_inputs_map old_inputs;

public:
  logic_trigg_sampl(const char *n): Simulator(n) {};
  void init(double, ...);
  double ta(double t);
  void delta(double);
  void lambda(double);
  void exit();
};
#endif
```

A.5.2 logic_trigg_sampl.cpp

```
#include "logic_trigg_sampl.h"

```

```
// custom code:
void logic_trigg_sampl::init(double t, ...)
{
  // The 'parameters' variable contains the parameters transferred
     ↪ from the editor.
  va_list parameters;
  va_start(parameters, t);
  printLogAtLevel(_INFO, "%s, %d, t=%G: Init \n", name, myself, t);

  const char *limit = va_arg(parameters, char *);
  left_limit = false;
  if (strcmp(limit, "left") == 0)
    left_limit = true;

  const char *edge = va_arg(parameters, char *);
  which_edge = BOTH;
  if (strcmp(edge, "rising") == 0)
    which_edge = RISING;
  else if (strcmp(edge, "falling") == 0)
    which_edge = FALLING;

  const char *initvec = va_arg(parameters, char *);
  init_output = DEVSLogicMessage::getLogicVector(initvec);

  sigma = 0;

  // always end the init-function with:
  va_end(parameters);
}
double logic_trigg_sampl::ta(double t)
{
  // This function returns a double.
  printLogAtLevel(_INFO, "%s, %d, t=%G: ta \n", name, myself, t);
  /* return double value with the time to the next internal event
     ↪ here. */

  return (sigma);
}
void logic_trigg_sampl::delta(double t)
{
  printLogAtLevel(_INFO, "%s, %d, t=%G: delta \n", name, myself, t);

  sigma = INF;

  if (!input_bag_empty()) {
    old_inputs = resolver.resolve_delta(*this, t);
  }
}
void logic_trigg_sampl::lambda(double t)
```

```cpp
{
  printLogAtLevel(_INFO, "%s, %d, t=%G: lambda \n", name, myself, t);
  if (sigma == 0) {
    for (int i = 0; i < int(init_output.size()); i++) {
      DEVSLogicMessage C(init_output[i]);
      C.index = i;
      printLogAtLevel(_DEBUG_ROUGH, "%s, %d, t=%G: output(%d) = %c\n", name,
          myself, t, i, C.getChar());
      add_output(&C, 0);
    }
  }

  if (!input_bag_empty()) {
    auto resolved_inputs = resolver.resolve_lambda(*this, t);
    bool rising, falling;
    try {
      rising = logicfunction_rising(old_inputs.at(0).at(1),
                  resolved_inputs.at(0).at(1));
      falling = logicfunction_falling(old_inputs.at(0).at(1),
              resolved_inputs.at(0).at(1));
    } catch (std::out_of_range &e) {
      // no edge
      return;
    }

    printLogAtLevel(_DEBUG, "%s, %d, t=%G: rising=%d falling=%d\n", name, myself, t,
        rising, falling);

    if ((which_edge == RISING && rising) || (which_edge == FALLING && falling) ||
        (which_edge == BOTH && (rising || falling))) {
      printLogAtLevel(_DEBUG, "%s, %d, t=%G: triggered\n", name, myself, t,
          rising, falling);

      const resolved_inputs_map *output = &old_inputs;
      if (left_limit == false) {
        output = &resolved_inputs;
      }
      for (const auto &p : *output) {
        DEVSLogicMessage C(p.second[0]);
        C.index = p.first;
        add_output(&C, 0);
      }
    }
  }
}
```

```
void logic_trigg_sampl::exit()
{
  // Code executed at the end of the simulation.
  printLogAtLevel(_INFO, "%s, %d: exit \n", name, myself);
}
```

List of Figures

3.1 PowerRPDEVS Model Editor 10
3.2 PowerRPDEVS Simulation Dialog 11
3.3 PowerRPDEVS Atomic Editor 12
3.4 Message Passing class hierarchy 14
3.5 Example object-oriented model and class hierarchy 15

4.1 Multivalue Logic Library – all blocks 21

5.1 static RS flip-flop variants by Junglas 26
5.2 static RS flip-flop – model A 29
5.3 static RS flip-flop – model B 29
5.4 `NOR` gate coupling of model B 30
5.5 Simulation results of static RS flip-flop 30
5.6 Coupling that detects falling edges 31
5.7 Triggered RS flip-flop in PowerRPDEVS – model A 32
5.8 Triggered RS flip-flop in PowerRPDEVS – model B 32
5.9 Simulation results of triggered RS flip-flop – model A 33
5.10 Simulation results of triggered RS flip-flop – model B 33
5.11 D flip-flop in PowerRPDEVS 34
5.12 Simulation results of D flip-flop 34
5.13 Shift register in PowerRPDEVS 35
5.14 Simulation results of the shift register 36
5.15 Serial bus model 38
5.16 Sender coupling (Sender1 and Sender2 in Figure 5.15) 38
5.17 Result of simulation – Serial bus 39

List of Tables

2.1 9 logic values in IEEE 1164-1993 8

5.1 RS latch truth table obtained by Boolean algebra 27
5.2 Solutions of Equation 5.4. 28
5.3 Solutions of Equation 5.6. 28
5.4 Message format of the bus 37
5.5 Data in the bus example 38

Acronyms

CAN	Controller Area Network. 36, 39
CSMA	Carrier-Sense Multiple Access. 36
CSMA/CR	Carrier-Sense Multiple Access/Collision Resolution. 36
DEVS	Discrete Event System Specification. 1–6, 39
FIFO	first in – first out. 35
FSA	finite state automaton. 4, 6
IEEE	Institute of Electrical and Electronics Engineers. 2, 7, 8, 15, 19–22, 39, 57
PDEVS	Parallel DEVS. 1, 2, 4–6
QSS	Quantized State System. 14
RPDEVS	Revised Parallel DEVS. 1, 2, 5, 6, 9, 11, 15, 39
VHDL	Very High Speed Integrated Circuit Hardware Description Language. 2, 7

Bibliography

[CZ94] Alex Chung Hen Chow and Bernard P Zeigler. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In *Simulation Conference Proceedings, 1994. Winter*, pages 716–722. IEEE, 1994.

[IEE93] IEEE Design Automation Standards Committee and others. IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)(ANSI): IEEE Standard 1164-1993. *IEEE, New York*, 1993.

[Jos96] C Joslyn. The process theoretical approach to qualitative DEVS. 1996.

[Jun16] Peter Junglas. Pitfalls using discrete event blocks in simulink and modelica. ASIM, 2016.

[PHK17] Franz Preyser, Bernhard Heinzl, and Wolfgang Kastner. RPDEVS: Revising the Parallel Discrete Event System Specification. 12 2017.

[PHKB19] Franz Josef Preyser, Bernhard Heinzl, Wolfgang Kastner, and Felix Breitenecker. RPDEVS Abstract Simulator. In *Proc. of ASIM-Workshop Simulation technischer Systeme/Grundlagen und Methoden in Modellbildung und Simulation*, page 6, 02 2019.

[PHRK16] Franz Josef Preyser, Bernhard Heinzl, Philipp Raich, and Wolfgang Kastner. Towards Extending the Parallel-DEVS Formalism to Improve Component Modularity. In Thorsten Pawletta Dmitrij Tikhomirov, Heinz-Theo Mammen, editor, *Beiträge zum Work. der ASIM/GI-Fachgruppen STS und GMMS 2016*, pages 83–89, Lippstadt, 2016. ARGESIM Verlag Wien, Hochschule Hamm-Lippstadt 2016.

[POWa] PowerDEVS repository on sourceforge.org. `https://sourceforge.net/projects/powerdevs/`. Accessed: 5th March, 2019.

[POWb] PowerRPDEVS repository on sourceforge.org. `https://sourceforge.net/projects/powerrpdevs/`. Accessed: 5th March, 2019.

[ZPK00] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems.* Academic press, 2000.