Production Chain in a Bakery - DEVS

Semester Project in

Modeling and Simulation (2014W)
by Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Felix Breitenecker
Institute of Analysis and Scientific Computing
Faculty of Mathematics and Geoinformation

Supervised by:

Franz Josef Preyser
Institute of Analysis and Scientific Computing
Faculty of Mathematics and Geoinformation

Submitted By:

Jawad Haider KAZMI (1328102),
Ishtiaq AHMAD (1229645) and
Ikram ULLAH (1329824)

March, 2015

Contents

Summary

The task of this project is to implement a simple DEVS-Model of a baking-oven with a DEVS-able simulation. The implemented model is described as follows;

At a certain point in time, portion of a paste with the attributes mass m and (up to now accumulated) costs K arrives at the Oven. If the oven is currently in the waiting - state (i.e. process-phase p=0), then the paste attributes stored in the oven state variables m and K and the process-phase p is incremented (i.e. p:=1). This is all done by the external transition function δ_{ext}. If process phase p=1, the paste is baking for the duration T_1. That is, the next event to be triggered is an internal event, which happens exactly T1 after the previous external event, where the paste arrived. This internal event leads to the execution of δ_{int}, which adds the costs K2 that arose from the baking phase to the accumulated costs K and increments the process phase p to p=2. p=2 means unloading-phase. The duration of the unloading phase is T_2. Therefore after T2 the next internal event is triggered which again results in the execution of δ_{int}. But this time , there is an output to be generated as well. So right before δ_{int} is executed, the output function lambda computes an "output-message". In our case, the output represents the baked bread, which again has the attributes mass m and costs K. the mass stays the same, so m will be set to the value stored as state variable m. Since the unloading phase also produce costs, the costs K will be set to the value stored as state variable K plus the costs K2 that arose in phase 2. After the output has been calculated, the internal transition function sets the state variables m and K to 0 also resets the process phase p to 0, meaning "waiting for the next paste to arrive."

Chapter 1

Introduction

1.1 DEVS-Model for Baking Oven

The task of this project is to implement a simple DEVS-Model of a baking-oven with a DEVS-able simulation. The implemented model is described as follows; At a certain point in time, portion of a paste with the attributes mass m and (up to now accumulated) cost K arrives at the Oven. If the oven is currently in the waiting - state (i.e. process-phase p=0), then the paste attributes are stored in the oven state variables m and K and the process-phase p is incremented (i.e. p:=1). This is all done by the external transition function δ_{ext} . If process phase p=1, the paste is baking for the duration T1. That is, the next event to be triggered is an internal event, which happened exactly T1 after the previous external event, where the paste arrived. This internal event leads to the execution of δ_{int} , which adds the costs K2 that arose from the baking phase to the accumulated costs K and increments the process phase p to p=2. p=2 means unloading-phase. The duration of the unloading phase is T2. Therefore after T2 the next internal event is triggered which again results in the execution of δ_{int}. But this time , there is an output to be generated as well. So right before δ_{int} is executed, the output function lambda computed an "output-message". In our case, the output represents the baked bread, which again has the attributes mass m and costs K. the mass stays the same, so m will be set to the value stored as state variable m. Since the unloading phase also produce costs, the costs K will be set to the value stored as state variable K plus the costs K2 that arose in phase 2. After the output has been calculated, the internal transition function sets the state variables m and K to 0 also resets the process phase p to 0, meaning "waiting for the next paste to arrive".

1.2 Paste arrive scenario

We have implemented a `Queue` to handle the situation when the process phase p is not zero and a paste arrives. If p is not zero and a paste is ready , it is added to the queue. After the execution of internal function and completion of phase 2 , queue is checked , if it is not empty then paste is picked from the queue on the basis of first in first out (FIFO).

Chapter 2

DEVS and `PowerDEVS`

2.1 DEVS Formalism

DEVS is an acronym for Discrete Event System . This is a formalism that was first introduced by Bernard Zeigler. It is used for the description of discrete event systems. A DEVS model has input events which are processed to produce output events(results). Output events are influenced by input events and initial conditions of the model.Figure 2.1 shows the basic structure of an atomic model. An Atomic DEVS model has the following formulation;

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

Where

X : set of input event values,
Y : set of possible output event values,
S : set of possible states values,
δ_{ext} : External state transition function,
δ_{int} : Internal state transition function,
ta : Time advance Function

2.2 The `PowerDEVS`

PowerDEVS is a general purpose software tool for DEVS modeling and simulation oriented to the simulation of hybrid systems. It allows defining atomic DEVS models in C++ language that can be then graphically coupled in hierarchical block diagrams to create more complex systems. The environment automatically translates the graphically coupled models into a C++ code which executes the simulation [1]. One of the features of PowerDEVS is the synchronization of simulation in real time operating systems with real time clock, which allows the design and implementation of synchronous and asynchronous digital controller. PowerDEVS is also an efficient tool for real time simulation of physical systems when combined with continuous system simulation library. The

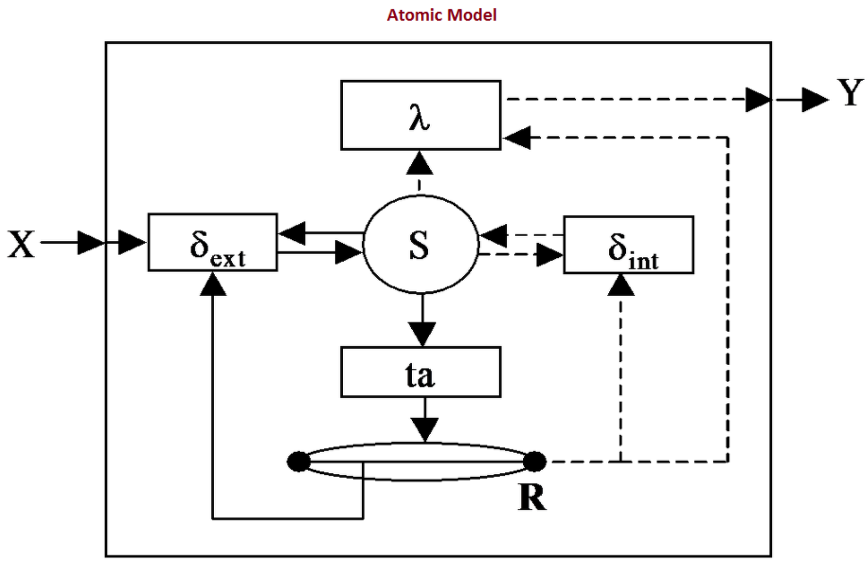

Figure 2.1: Atomic Model

interconnection of PowerDEVS with the numerical package Scilab is another attractive feature of PowerDEVS. The variables and functions in the workspace of Scilab can be used by PowerDEVS, and Scilab can process and analyze the result data sent from PowerDEVS.

PowerDEVS is composed of various independent programs:

2.2.1 The Model Editor

It is the main program of PowerDEVS which provides graphical interface for building and managing models and libraries, launching simulation, editing elementary blocks up to atomic model definitions and links to other application of PowerDEVS.

The Model Editor main window shown in Figure 2.2 is used to create and open models and libraries. At the left the list of libraries can be selected and blocks can be dragged from the libraries to the models. The selected library is active and the blocks are visible under the selected library. Models and libraries can be edited in a model window with the open and new model commands. The model window in Figure 2.3 shows a model consisting of with four sub models.

In the model window, the typical graphical editing facilities are provided so that blocks can be copied, re-sized, rotated, etc., while the connections between different ports can be directly drawn. Using the edit menu or right mouse button - edit, features of blocks, no matter whether coupled or atomic, can be edited. The block editing window shown in Figure 2.4 is used to configure the graphic appearance of the block and to choose the parameters of blocks. In the case

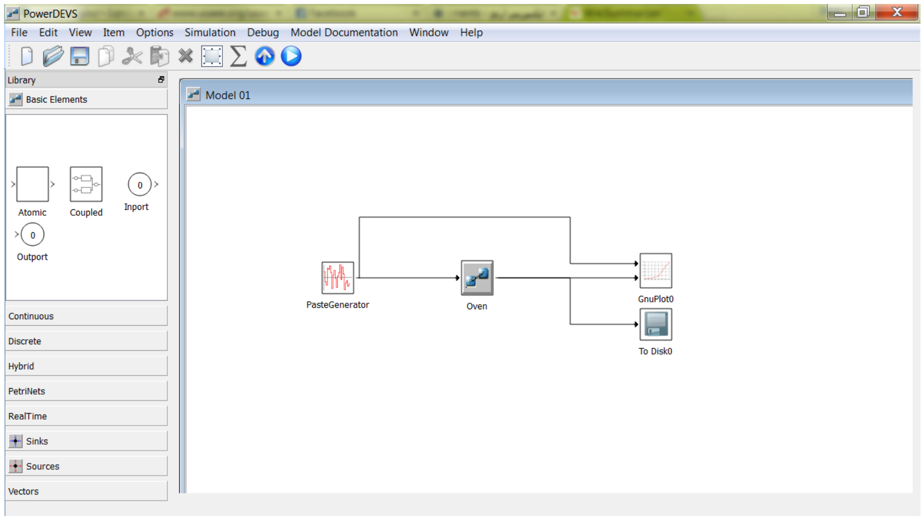

Figure 2.2: Model Editor main window

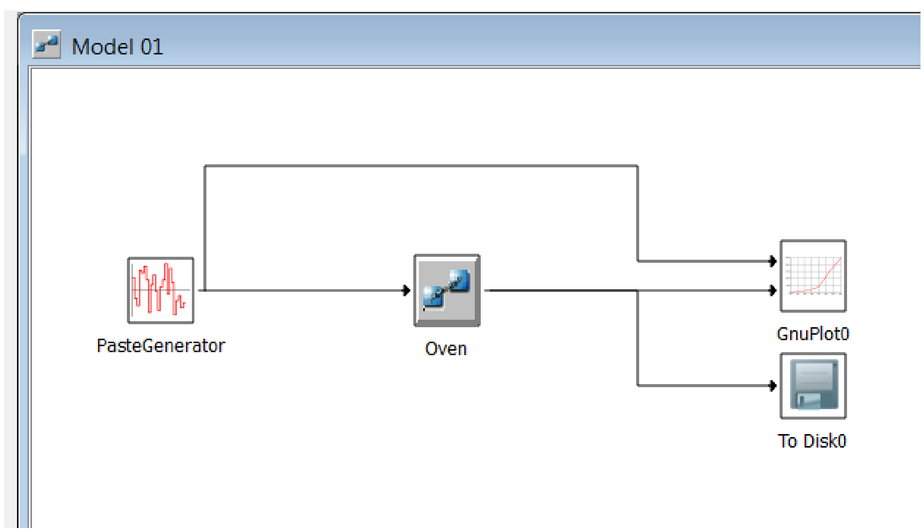

Figure 2.3: Model Window

of atomic models the associated code with the DEVS model definitions, can be selected in the block editing window. The values of the block parameters can be changed by double clicking on the block as shown in Figure 2.5. Thus the predefined blocks are taken from the libraries and the parameter values can be changed without editing them.

The Coupled models have no associated code and there are some other extra features which can be modified in the block editing window and the edit menu.

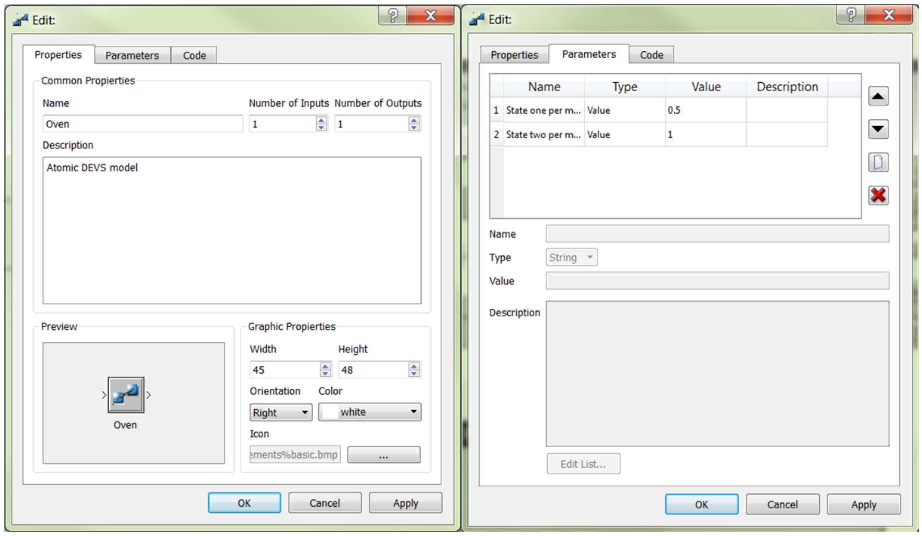

Figure 2.4: Block Edition Window

Various input and output ports of a coupled model are characterized by their names. The order of their appearance in the block can be changed in the editing window. Priorities can be established in editing menu of corresponding model to solve the simultaneous occurrence of events among blocks in the same submodel.

2.2.2 The Atomic Editor

The Atomic Editor facilitates the editing of C++ code corresponding to each DEVS model. The transition functions, output function, time advance, etc. for DEVS atomic models of elementary blocks are defined in it.

It can be invoked from the Editing Window to edit an existing code or to create a new one. It also can be run directly from Windows since it is a standalone application. The Atomic Editor main window is shown in Figure 2.6. In atomic editor the state variables, the output of the DEVS model and system parameters are defined. Then the C++ code for time advance, transition and output functions has to be placed in the corresponding windows and when the model is saved, the code is automatically completed and stored in the corresponding .cpp and .h files. The Atomic Editor was also designed to write a code which is very similar to the DEVS model definition. All the rest of the job related to simulation and implementation topics is automatically performed by the program.

2.2.3 Structure Generator

A PowerDEVS model is defined by a block diagram stored in a .pdm file which represents system structure and by the atomic models in .h and .cpp files which are generated by the atomic editor. Then, the simulation is run using Quick Simulation command in the Simulation menu of the Model Editor. The Quick Simulation command performs the simulation after converting the PowerDEVS model into a standalone program that executes the simulation. The final simulation time is only asked in this command and then the simulation is executed. The conversion of the model into a simulation program is done in two steps. The Structure Generator converts the model file (.pdm) which contains all the information about the model, into a coupled DEVS specification. The Structure Generator produces a .pds file which only contains the information about connections, location of atomic models and block parameters, etc., needed to build the simulation file. The coupling specification of the model is also converted into a formal DEVS coupling specification. As the Structure Generator is a standalone program it can be run directly from the Simulation menu or from the command line. If it is run in that way, it produces a report in .pds file showing the output. Figure 2.7 shows the out of structure generator.

2.2.4 The Preprocessor

The Preprocessor translates the model editor files into structure files which contain the coupling structure and the information to build up the simulation code, links the code of the different atomic models according to the corresponding structure file and compiles it to produce a standalone executable file which simulates the system. It basically translates the .pds file into a header .h file which binds the simulators and coordinators according to the coupling structure. The preprocessor also produces a make file which is then invoked to generate the program which executes the simulation. The Preprocessor can be invoked in a transparent way using the Quick Simulation command.

2.2.5 The Simulation Interface

The Simulation Interface (Figure 2.8) runs the stand alone executable files according to the structure of PowerDEVS as shown in Figure 2.9 and provides input mask to change the parameters of simulation like final time, number of simulations to perform, and the simulation mode (normal simulation, timed simulation, step-by-step simulation, etc.).

2.2.6 A running instance of Scilab

It acts as a workspace, where the simulation parameters can be read and results can be exported to. This instance is a modification of Scilab 4.1.2 to support this type of operations as shown in Figure 2.10.

All the applications of PowerDEVS except Model Editor were programmed in C++ with the graphical libraries QT. The Model Editor is the only application programmed in Visual Basic. PowerDEVS can runs under a real time operating system [3] synchronizing the events with a realtime clock with the capability of capturing interrupts at the atomic level. PowerDEVS also allows the direct implementation of asynchronous DEVSbased Quantized State Controllers [2] on a PC.

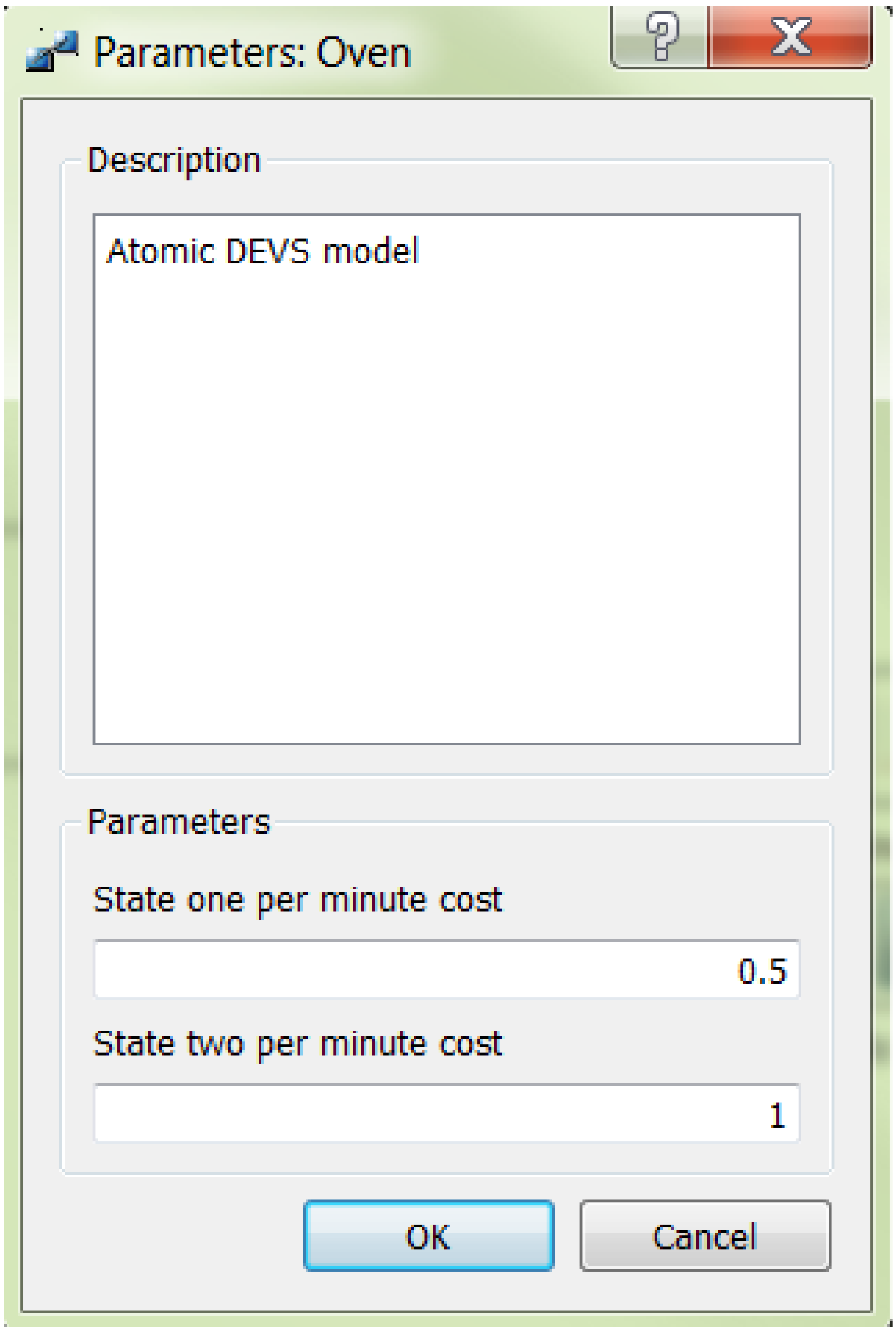

Figure 2.5: Parameter changing window

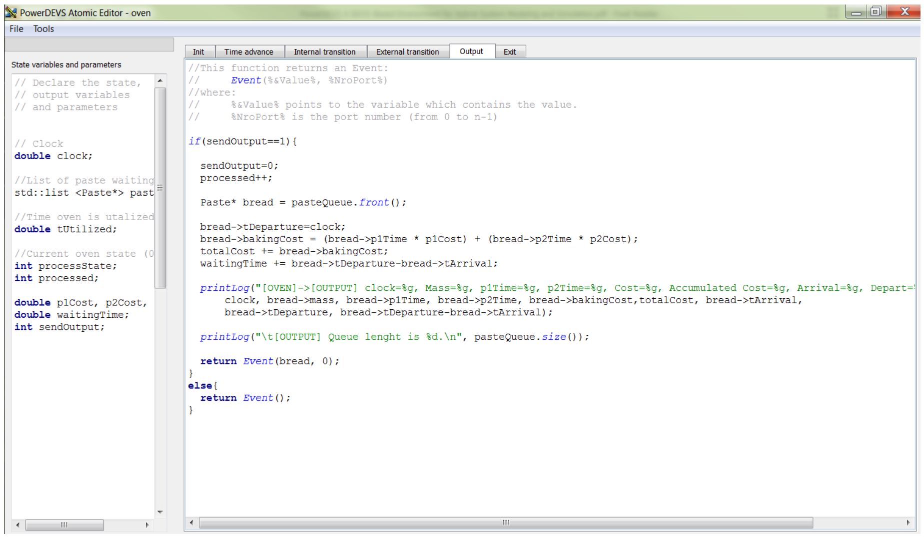

Figure 2.6: Atomic Editor main window

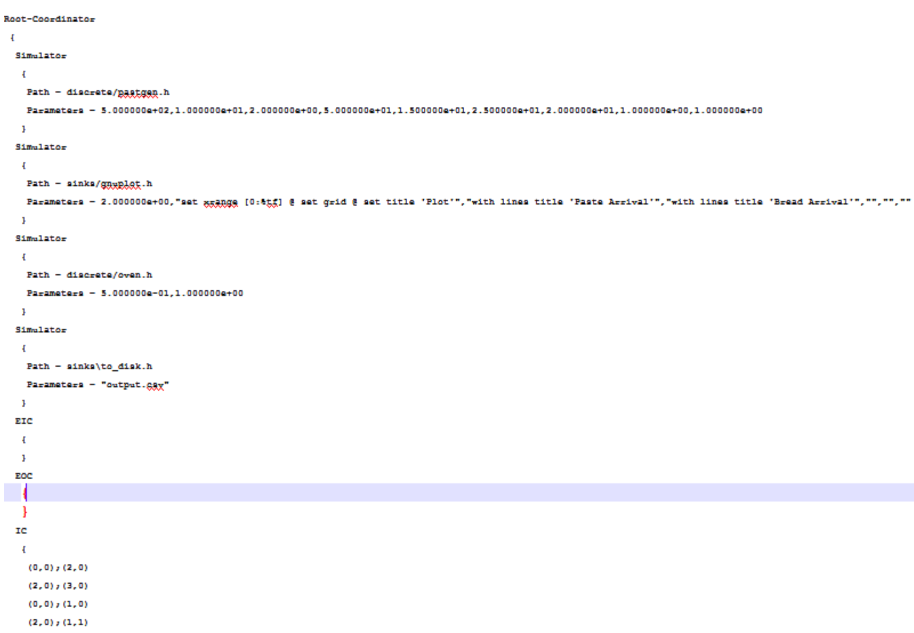

Figure 2.7: Structure Generator output

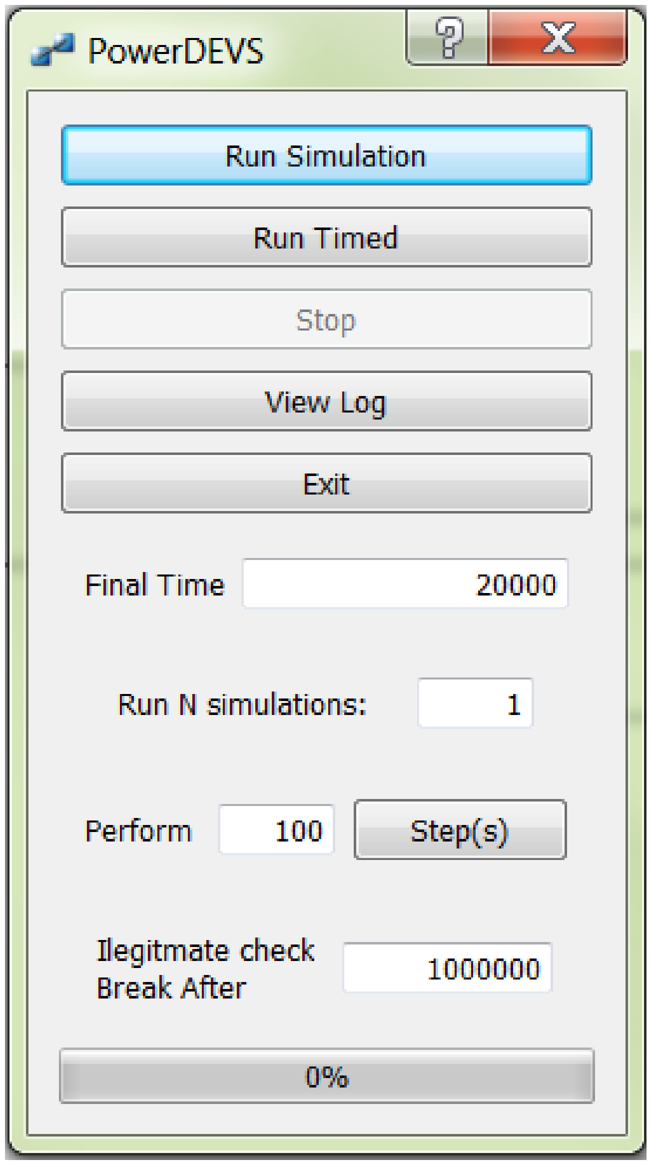

Figure 2.8: Simulation interface

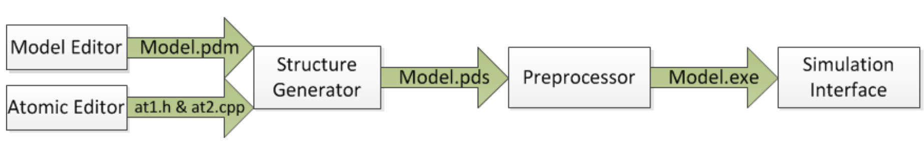

Figure 2.9: Simulation structure of PowerDEVS

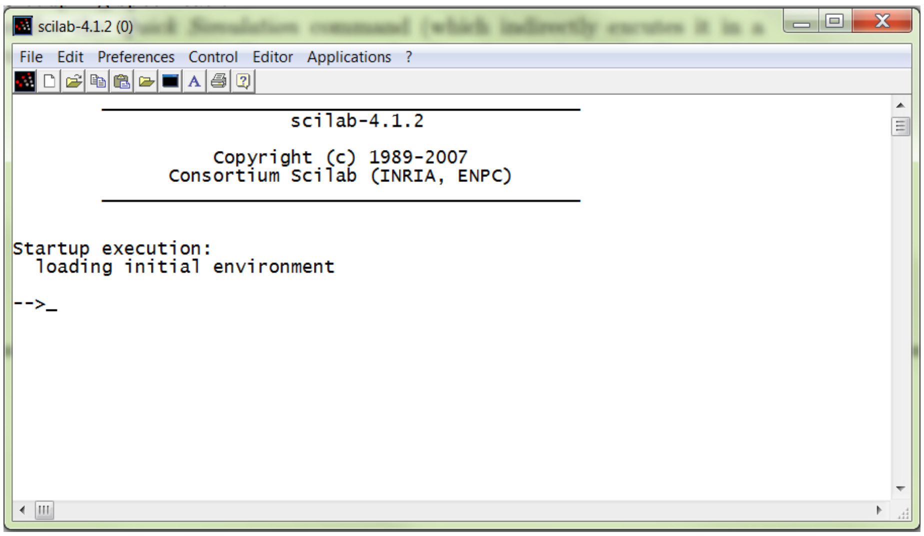

Figure 2.10: Scilab Instance

Chapter 3

Implementation

The DEVS model of the Baking Oven is shown in Fig. 3.1. The model shows that the implementation consists of four atomic models (`Paste Generator` , `Oven` , `Plot` and `Save to Disk`), two (`Paste Generator` and `Oven`) of which are programmed by us while the other two (`Plot` and `Save to Disk`) are used from the PowerDEVS's library. The `Paste Generator` generates paste with random mass and processing times. The `Paste` is then processed (baked) in the `Oven` and a `Bread` is produced.

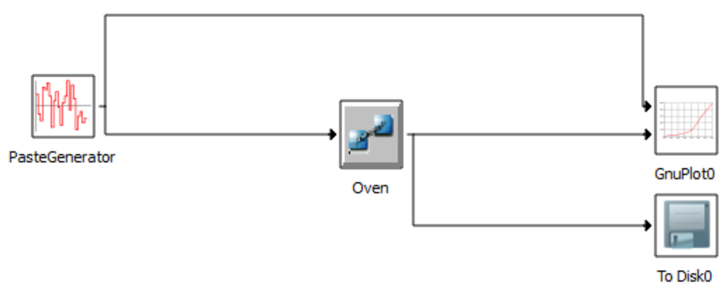

Figure 3.1: The Baking Oven Model in PowerDEVS

3.1 Paste structure

A structure is created to hold the parameters and to describe the paste. The definition of the structure is shown in the Listing 3.1.

```

struct Paste
{
    double mass;          // Mass of the paste
    double p1Time;        // Baking time required in P1
```

```
    double p2Time;        // Baking time required in P2
    double bakingCost;    // Baking cost

    double tArrival;      // Arrival time
    double tDeparture;    // Departure time (when baked as bread)
};
```

Listing 3.1: Paste structure

3.2 Paste Generator

In the `Paste Generator` parameters can be specified to produce random values. In PowerDEVS IDE, the parameters dialogue box for the `Paste Generator` is shown in the Fig. 3.2.

Listing 3.2 below, shows the state variables and objects used during the paste generation.

```

  double sigma;             // The TA value

  StochasticLib1 *stor;     // Lib for RN Generation
  Paste* paste;             // The paste structure object

  //Parameters
  double maxOutputs;        // Max pastes to be generated
  double massMax, massMin;  // Mass
  double p1Max, p1Min;      // P1 time
  double p2Max, p2Min;      // P2 time
  double genMax, genMin;    // Generation pause
```

Listing 3.2: State variables and objects

Listing 3.3 is the `Initialization` routine for the Paste Generator. The lines 1,2 are the call for passing the parameters from the IDE to a list. This list is later used to assign the configured values to the state variables (line 5–13). Next, StochasticLib1 [1] is initialized with a seed generated from the timer and a random number (15–16). The TA value state variable sigma is initialized to 0 (line 18). At the end, a a summary of the configured parameters is printed into the log file (lines 21–27).

```
1  va_list parameters;
2  va_start(parameters,t);
3
4  maxOutputs=va_arg(parameters,double);   //Max pastes
5
6  massMax=va_arg(parameters,double); //Max mass
7  massMin=va_arg(parameters,double); //Min mass
8  p1Max=va_arg(parameters,double);  //Max p1 time
9  p1Min=va_arg(parameters,double); //Min p1 time
10 p2Max=va_arg(parameters,double); //Max p2 time
11 p2Min=va_arg(parameters,double); //Min p2 time
12 genMax=va_arg(parameters,double); //Max ta for output
```

[1] http://gull.sourceforge.net/dev/classStochasticLib1.html

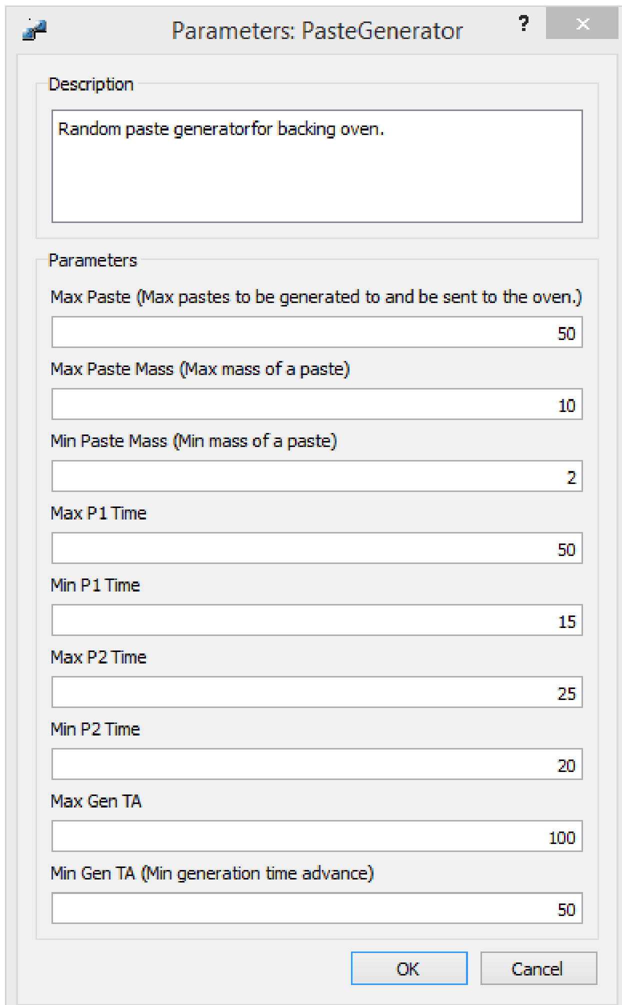

Figure 3.2: `Paste Generator` parameters configuration dialogue-box in PowerDEVS IDE

```
genMin=va_arg(parameters,double); //Min ta for output

int seed = (int)time(0)+rand(); //seed for random num lib
stor=new StochasticLib1(seed); //Initialized the lib

sigma=0;

//Print summary of the configured parameters
printLog("Paste Generator configured with:\n");
printLog("\tMax Pastes=%g\n", maxOutputs);
printLog("\tMax Mass=%g, Min Mass=%g\n", massMax, massMin);
printLog("\tMax P1=%g,   Min P1=%g\n", p1Max, p1Min);
printLog("\tMax P2=%g,   Min P2=%g\n", p2Max, p2Min);
printLog("\tMax Gen TA=%g,   Min Gen TA=%g\n", genMax, genMin);
printLog("\t---------------------------\n\n\n");
```

Listing 3.3: The `Initialization` for the `Paste Generator`

The time advance (TA) routine (Listing 3.4), generates and returns a random double value between `genMin` and `genMax` state variable untill the configured numbers of pastes (`maxOutputs`) have been generate. Otherwise it returns the maximum allowed values for a variable of type double (`std::numeric_limits<double>::max()`).

```
//Time Advance (TA)

//Check if the configured number of
//outputs have been generated.
if(maxOutputs>0) {
  return sigma;
}
else {
  return std::numeric_limits<double>::max();
}
```

Listing 3.4: The TA (Time Advance) function

The Internal Transition function (Listing 3.5) simply generates a random value between the configured parameter `genMin` and `genMax` and assigns it a state values, for next time advance.

```
//The Internal Transition
sigma=stor->IRandom(genMin,genMax);
```

Listing 3.5: The Internal Transition function for the `Paste Generator`

Since, `Paste Generator` has no input interface, it has nothing in the External Transition routine.

The Output function is shown in the Listing 3.6. A `Paste` is first created and than, its parameters are filled with random values under the configured range and then sent to the output port. Once, enough `Paste` have been generated an `empty` output is returned.

```
if(maxOutputs-->0)
{
 paste=new Paste();
 paste->mass=stor->IRandom(massMin,massMax);
 paste->p1Time=stor->IRandom(p1Min,p1Max);
```

```
paste->p2Time=stor->IRandom(p2Min,p2Max);

// Record in the Even Log
printLog("[PASTEGEN]␣Generated␣paste␣number␣\%d␣with␣mass=\%g,␣
    p1Time=\%g,␣p2Time=\%g\n",
       maxOutputs, paste->mass, paste->p1Time, paste->p2Time);

return Event(paste, 0);
}
else return Event();
```

Listing 3.6: The Output function for the `Paste Generator`

3.3 Oven

The `Oven` is modeled as a bakery oven having two baking stages. Each stage has an associated per time unit cost that can be configured in PowerDEVS using the parameters configuration dialog-box shown in the Figure 3.3.

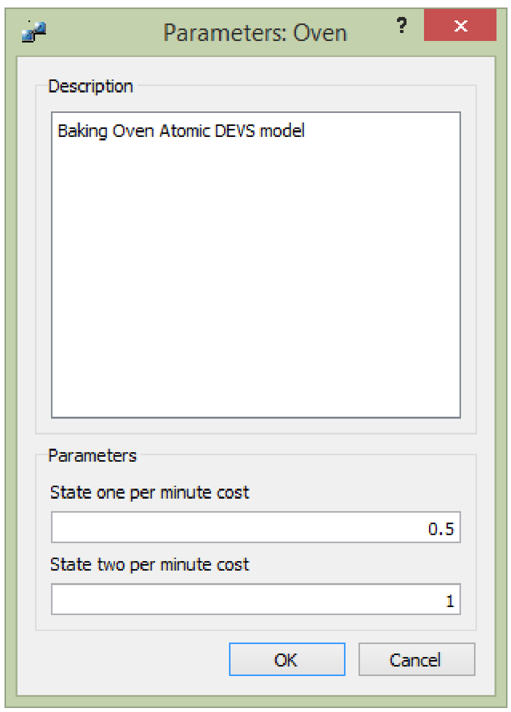

Figure 3.3: `Oven` configuration parameters dialog-box in PowerDEVS IDE

Listing 3.7, below shows the state variables and objects used by the `Oven` .

```
// Clock
double clock;
```

```

//List of waiting pastes
std::list <Paste*> pasteQueue;

Paste* bread;

//Current oven state (0, 1, 2)
int processState;

int processed;

double p1Cost, p2Cost, totalCost;
double waitingTime;
int sendOutput;
```

Listing 3.7: State variables and objects used by `Oven`

The code for the `Initialization` function of `Oven` is shown in Listing 3.8.

```
va_list parameters;
va_start(parameters,t);

p1Cost=va_arg(parameters,double);
p2Cost=va_arg(parameters,double);

clock=0;                          // Reset the clock
tUtilized=0;            // Reset

processState=0;
totalCost=0;
sendOutput=0;
processed=0;
waitingTime=0.0;


printLog("Oven␣Initilized␣clock=\%g,␣state=\%d\n", clock,
    processState, t);
printLog("\tConfigured␣with␣p1Cost=\%g,␣p2Cost=\%g\n
    ================================================\n\n", p1Cost,
    p2Cost);
```

Listing 3.8: `Initialization` function for `Oven`

```
double sigma;

switch(processState){
        case 0:         // We should never arrive here!
                if(pasteQueue.empty()){
                        printLog("[OVEN]->[TA]␣No␣paste␣to␣bake,␣
                            waiting␣for␣the␣paste␣INF␣(state=\%d)\n
                            ", processState);
                        sigma=std::numeric_limits<double>::max();
                }
                else sigma=pasteQueue.front()->p1Time;
                break;
        case 1:         // processState1
                sigma=pasteQueue.front()->p1Time;
                break;
```

```
        case 2:         // processState2
                sigma=pasteQueue.front()->p2Time;
                break;
        default:        // an invalid state
                sigma=std::numeric_limits<double>::max();
}

//sigma+=clock;
if(!t==0) printLog("[OVEN]->[TA] clock=\%g, state=\%d, sigma=\%g\n"
    , clock, processState,sigma);

return sigma;
```

Listing 3.9: `Time Advance` function for `Oven`

```
int dummy;

clock+=ta(0);           //Update clock

switch(processState){
        case 0:
                if(!pasteQueue.empty()) dummy=1;
                else dummy=0;
        case 1:
                dummy=2;
                break;
        case 2:
                sendOutput=1;
                bread = pasteQueue.front();
                //beingBakedPaste=pasteQueue.front();
                if(pasteQueue.empty()) { dummy=0; }
                else {
                        dummy=1;
                }
                pasteQueue.pop_front();
                break;
}
processState=dummy;

printLog("[OVEN]->[INT] clock=\%g, New state=\%d\n", clock,
    processState);
```

Listing 3.10: `Internal Transition` function for `Oven`

```
// update the clock
clock += e;

if(processState==0){
        processState=1;
}

pasteQueue.push_back((Paste*)x.value);
pasteQueue.back()-> tArrival = clock;

//beingBakedPaste =
printLog("[OVEN]->[EXT] at clock=\%g, New paste addedd.\n", clock);
```

```
printLog("\t[EXT]␣Queue␣lenght␣is␣\%d.\n", pasteQueue.size());
```

Listing 3.11: `External Transition` function for `Oven`

```
if(sendOutput==1){

        sendOutput=0;
        processed++;

        //Paste* bread = pasteQueue.front();

        bread->tDeparture=clock;
        bread->bakingCost = (bread->p1Time * p1Cost) + (bread->p2Time * p2Cost);
        totalCost += bread->bakingCost;
        waitingTime += bread->tDeparture-bread->tArrival;

        printLog("[OVEN]->[OUTPUT]␣clock=\%g,␣Mass=\%g,␣p1Time=\%g,␣p2Time=\%g,␣Cost=\%g,␣Accumulated␣Cost=\%g,␣tArrival=\%g,␣tDepart=\%g,␣tWait=\%g\n",
                        clock, bread->mass, bread->p1Time, bread->p2Time, bread->bakingCost,totalCost, bread->tArrival,
                        bread->tDeparture, bread->tDeparture-bread->tArrival);

        printLog("\t[OUTPUT]␣Queue␣lenght␣is␣\%d.\n", pasteQueue.size());

        return Event(bread, 0);
}
else{
        return Event();
}
```

Listing 3.12: `Output` function for `Oven`

```
//Code executed at the end of the simulation.
printLog("=================OVEN==========================\n");
printLog("\tClock=\%g", clock);
printLog("\tAccumulate␣Cost␣=␣\%g\n", totalCost);
printLog("\tAverage␣Waiting␣Time␣=␣\%g\n", waitingTime/processed);
printLog("\tPastes␣Processed␣=␣\%d\n", processed);
printLog("=========================================\n");
```

Listing 3.13: Exit function for `Oven`

Chapter 4

Results

4.1 Log file output

Below, is a sample instance of the log file printed as the output of the simulation.

Log file listing

```
Paste Generator configured with:
Max Pastes=2
Max Mass=10, Min Mass=2
Max P1=50,   Min P1=15
Max P2=25,   Min P2=20
Max Gen TA=100,   Min Gen TA=50
----------------------------


Oven Initialized clock=0, state=0
Configured with p1Cost=0.5, p2Cost=1
================================================

Simulation Initialized
[OVEN]->[TA] No paste to bake, waiting for the paste INF (state=0)
[PASTEGEN] Generated paste number 1 with mass=3, p1Time=35, p2Time=25
[OVEN]->[EXT] at clock=0, New paste addedd.
   [EXT] Queue length is 1.
[OVEN]->[INT] clock=35, New state=2
[OVEN]->[TA] clock=35, state=2, sigma=25
[OVEN]->[INT] clock=60, New state=1
[OVEN]->[TA] clock=60, state=1, sigma=35
[PASTEGEN] Generated paste number 0 with mass=2, p1Time=29, p2Time=23
[OVEN]->[EXT] at clock=79, New paste addedd.
   [EXT] Queue length is 1.
[OVEN]->[TA] clock=79, state=1, sigma=29
```

```
[OVEN]->[OUTPUT] clock=79, Mass=3, p1Time=35, p2Time=25, Cost=42.5,
>>>>>>Accumulated Cost=42.5, tArrival=0, tDepart=79, tWait=79
   [OUTPUT] Queue length is 1.
[OVEN]->[INT] clock=108, New state=2
[OVEN]->[TA] clock=108, state=2, sigma=23
[OVEN]->[INT] clock=131, New state=1
[OVEN]->[TA] clock=131, state=1, sigma=29
[OVEN]->[OUTPUT] clock=131, Mass=2, p1Time=29, p2Time=23, Cost=37.5,
>>>>>Accumulated Cost=80, tArrival=79, tDepart=131, tWait=52
   [OUTPUT] Queue length is 0.
[OVEN]->[INT] clock=160, New state=2
[OVEN]->[TA] clock=160, state=2, sigma=23
[OVEN]->[INT] clock=183, New state=0
[OVEN]->[TA] No paste to bake, waiting for the paste INF (state=0)
[OVEN]->[TA] clock=183, state=0, sigma=1.79769e+308
==================OVEN===========================
Clock=183 Accumulate Cost = 80
Average Waiting Time = 65.5
Pastes Processed = 2
=============================================
Simulation Ended (0.008 sec)

(>>>>>> indicates a continued line with the previous)
```

4.2 GNU Plot of outputs

Figure 4.1 below, shows the output of a sample instance of the simulation, plotted using the GNU Plot model in the `PowerDEVS` .

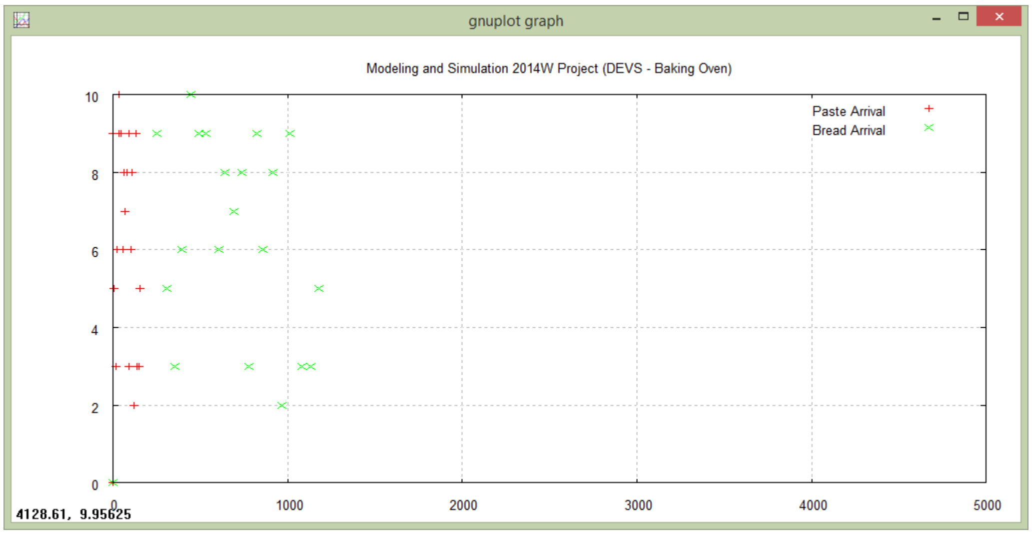

Figure 4.1: A sample run plotted using the GNU Plot model in `PowerDEVS` .

Bibliography

[1] F. Bergero and E. Kofman. PowerDEVS. A Tool for Hybrid System Modeling and Real Time Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 87(1–2):113–132, 2011.

[2] E. Kofman. Quantized-State Control. A Method for Discrete Event Control of Continuous Systems. *Latin American Applied Research*, 33(4):399–406, 2003.

[3] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. Rtai: Real time application interface. *Linux J.*, 2000(72es), April 2000.