Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

Development of QSS and PID Control Algorithm inside RT-DEVS Framework

Mengyao Wu

Department of System and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON K1S 5B6, Canada
mengyaowu@cmail.carleton.ca

Xinrui Zhang

Department of System and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON K1S 5B6, Canada
xinruizhang@cmail.carleton.ca

ABSTRACT

In this paper, we have implemented and tested QSS and PID atomic model for RT-DEVS framework by performing both simulation and on-target testing. RT-DEVS is built with DEVS formalism. It is a framework designed for bringing DEVS formalized design of real-time embedded system into microcontroller-based device such as robotics. QSS stands for Quantized State System. A QSS-based system generates events only when the system state has changed by a quantized value which is called quantum. Compared with discrete-time model, QSS-based model in its nature is discrete-event driven and it can largely reduce number of events generated. PID stands for proportional-integral-derivative. PID controller is a control loop device providing feedback by calculating correction based on proportional, integral, and derivative of difference between measured and desired system state variable. QSS and PID can be incorporated into Cyber-Physical System to reduce the number of messages passed through network while still have a stable control of the desired system state variable.

1 INTRODUCTION

For a physical system which has control and feedback loop physically separated with each other, it needs a communication channel to deliver the information from control side to the actuator side or vice versa. To sample the signal from one side and sends the information to another side with high accuracy, a very small, discrete-time step is required. However, if the communication channel is shared for many other physical components for exchanging state information, sampling of each signal at a fixed discrete-time step will lead to large amount of network traffic generated. This increases the chance of network congestion. A communication network with larger bandwidth can be deployed to combat congestion issue but this will lead to higher cost of the infrastructure.

A Quantized State System (QSS) method can be applied to reduce the number of events generated for reconstructing a continuous control or feedback signal. Instead of generating events at a fixed discrete-time step, a QSS-based system generates events only when the system state variable has changed by a quantized value which is called quantum. If proper order of mathematical model used, QSS can reduce number of events generated significantly while still providing adequate amount of information for the receiver to reconstruct the signal. At the same time, a proportional-integral-derivative(PID) controller is typically used to calculate the correction value the controller needs to produce to bring a system state variable to the desired value. The correction value is calculated by adding components together which are proportional, integral, and derivative of the error. The error is simply the difference between the measured and desired system state variable. Eventually, a QSS-PID enabled system can minimize state information exchange among physical components while still achieve a satisfying control of the target state variable. QSS-PID can be applied for many small to large physical systems such as vehicle Controller Area Network(CAN bus) and Cyber-Physical System (CPS).

In this paper, we implemented QSS atomic model with order 1 (QSS1) and order 2 (QSS2) modelling capability. We also implemented a PID atomic model. Both models are designed based on DEVS(Discrete Event Systems specifications) formalism within the RT-DEVS (Real-Time DEVS) framework. (Niyonkuru and Wainer 2016) introduces a RT-DEVS framework which executes DEVS models on hardware. Based on the framework in (Niyonkuru and Wainer 2016), we run the simulation within RT-DEVS with various test cases then port the software implementation of QSS in C++ to a NUCLEO-F446RE microcontroller for on-target testing. A potentiometer is used for adjusting the control signal for the brightness of the light. With QSS1 modelling, the number of events generated is less than that of a discrete-time sampling. With QSS2, much less events are generated while we can still achieve very smooth control of the brightness. P and D component of PID controller are implemented tested with RT-DEVS simulation.

2 Background

This section describes QSS and PID working principle.

2.1 Quantized State System (QSS)

QSS-based systems are continuous time systems since it only seeks for the time at which the system state has changed. This transforms the modelling of a continuous time signal from discrete-time to be discrete-event. When QSS is applied to network communications, it can reduce the number of messages passing through the network to reduce network congestion. When it is applied to real-time simulation, it avoids iterative solving and backtracking at discrete-time steps which are typically used in traditional numerical method.

QSS was first introduced by (Kofman and Junco 2001). (Kofman and Junco 2001) has shown that first order QSS can be exactly represented and simulated by a discrete event model within the framework DEVS formalism. (Kofman 2002) proposed a new second-order approximation called second-order quantized state systems (QSS2). According to (Kofman 2002), QSS2 preserves all the properties of QSS (QSS1) such as stability and convergence, allows the reducing the number of calculations with respect to QSS1. Based on QSS1 and QSS2, Kofman introduced QSS3 in (kofman 2006) which shows advantages in the integration of discontinuous systems.

In (kofman 2006) which introduces QSS3, the paper provides formal definition, properties, and comparison of QSS1, QSS2 and QSS3. It also provides examples of using QSS to control DC motor. To keep the originality of the formal descriptions for QSS families, it is better to use (kofman 2006) as the top choice of reference. However, in this paper we will explain QSS1 and QSS2 from another perspective tailored to our specific application. We are developing QSS atomic models as a library for the RT-DEVS framework. The QSS models can be added to control a light remotely with less events generated.

In our specific application, we have a physical system which has control and actuator devices physically separated. The control device should deliver a real-time continuous signal to the actuator. In the example of light control, a potentiometer is attached to control device while a LED light is attached to the actuator. The potentiometer adjusts a continuous analog signal indicates the desired brightness of the LED light. The continuous signal should be sent to the actuator of LED light via some communications channel. Instead of using discrete-time sampling of this signal, we are applying QSS to deliver this information.

2.1.1 QSS Event Generation on the Sender Side

A basic equation (1) can be used to answer the question at which time the events will be generated.

$$|y-\bar{y}| > \triangle Q. \tag{1}$$

In the equation (1), the symbol y represents the latest sampling of the input signal. This y signal is the source of signal that should be sent to the other side of the network. As a result, y can be deemed as the signal from the sender. $\bar{y}$ is the signal reconstructed or estimated by the receiver. When the difference, which is between the latest y sampling on the sender side and the estimated $\bar{y}$ value on the receiver side, is greater than the value denoted by ΔQ, an event will be generated. In this case, ΔQ is called a quantum which is chosen by the designer. A dynamic change of ΔQ is possible, but this is outside the scope of this paper.

2.1.2 QSS Signal Reconstruction on the Receiver Side

In equation (1), there are 3 variables y, $\bar{y}$ and ΔQ. y is known from latest sampling on the sender side while ΔQ is known by a predefined value. The only unknown is the value of $\bar{y}$. When an event is generated, there are some state information also sent to the receiver. The state information describes the latest state of the signal on the sender side. Upon receiving the latest state information, the receiver updates the previous state information with the latest one and continues to estimate (reconstruct) signal $\bar{y}$ with the latest state information. The receiver assumes that system state on the sender side remains the same until next event is generated and received.

2.1.3 QSS State Information

The state information should at least consist of latest sampling of y. In addition, it could also contain different order of derivatives of signal y which can be denoted by y', y" and so on. The sender keeps track of sampled y value and calculated different order of derivatives. Upon sender generating events, the sender sends y and all the derivatives to the receiver as state information.

The table 1 summarizes the state information for each QSS method to send. QSS1 only send latest signal sampling y value. This implies derivative of any order is zero hence the signal reconstructed is piece-wise constant. QSS2 sends both y and y'. As a result, the reconstructed signal is piece-wise linear.

Table 1: State Information used by different QSS algorithms

QSS Algorithm	State Information
QSS1	y
QSS2	y, y' = $y_k - y_{k-1}$ where k represents kth time step

To reconstruct the signal y on the receiver side, an integrator is used. In equation (2), K represents the kth time step. Δt is the time advancement for each time step. For QSS1, the y' is 0.

$$\bar{y_k} = \bar{y_{k-1}} + \Delta t * y'. \tag{2}$$

If the sender and receiver are synchronized, Δt is the fixed discrete-time step, equation (2) can be simplified as equation (3):

$$\bar{y_k} = \bar{y_{k-1}} + y'. \tag{3}$$

At any arbitrary discrete-time step, if no events generated, then y_k stays same as previous value for QSS1 while increases by latest y' received for QSS2. If it receives an event, receiver will update $\bar{y_k}$ to be the latest received y value. At this point, the whole process which contains QSS signal quantization, events generation, QSS signal reconstruction can be shown in the figure 1.

In the left figure of figure 1 , the blue dots represents the sampled y value at each fixed discrete-time step. A QSS1 quantizer is applied at the sender side. After the blue dot cross a quantum of 0.05, events are generated as red dot indicates. For each red dot, the corresponding time is when the event is generated and the y value of the red dot is the state information sent to the receiver. The right figure of figure 1 shows the comparison of receiver estimated $\bar{y}$ and original sampled y signal at the sender side. The estimated value is represented by the red dots. Upon the arrival of an event, the receiver updates the $\bar{y}$ with the received y value. After that, the estimated $\bar{y}$ value stays constant at each time step until the next event arrives. As a result, QSS1 receiver constructs the signal as piece-wise constant.

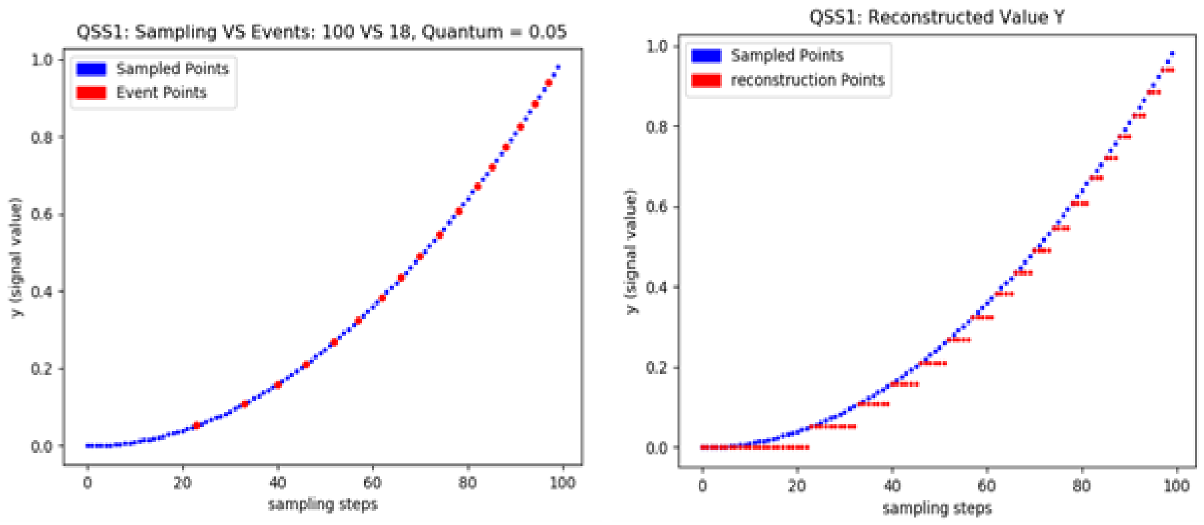


Figure 1: QSS1 algorithm event generation and reconstruction

In figure 2, with same representation and similar operation, QSS2 sender sends out one more state information y'. The receiver uses this y' information to update $\bar{y}$ as piece-wise linear. At the same time, the sender also knows the $\bar{y}$ at the receiver side since it also knows y' information. QSS2 sender calculates the difference between y and $\bar{y}$ along with current y' at each time step. If the difference is greater than ΔQ, then send y and y' as state information to QSS2 receiver.

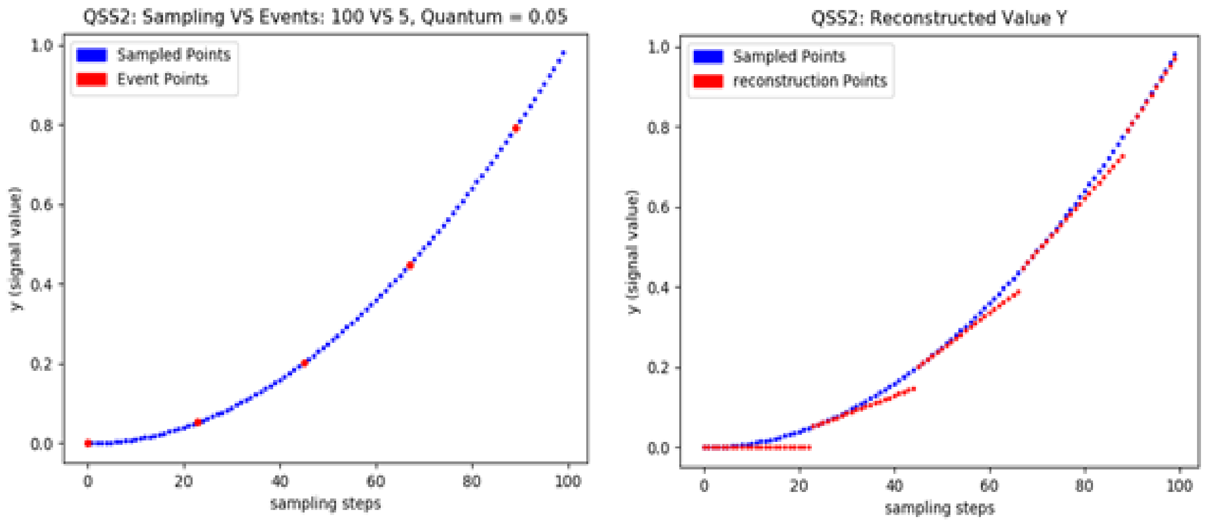


Figure 2: QSS2 algorithm event generation and reconstruction

Observed from figure 1 and figure 2, with same sampling steps, QSS1 generates 18 events while QSS2 only generates 5 events. At the same time, QSS2 produces a piece-wise linear reconstruction which models the original signal better.

2.2 Proportional-Integral-Derivative (PID) Controller

Proportional-Integral-Derivative (PID) control is the most common control algorithm used in industry. It is simple but effective so that it is universally accepted by industrial control.

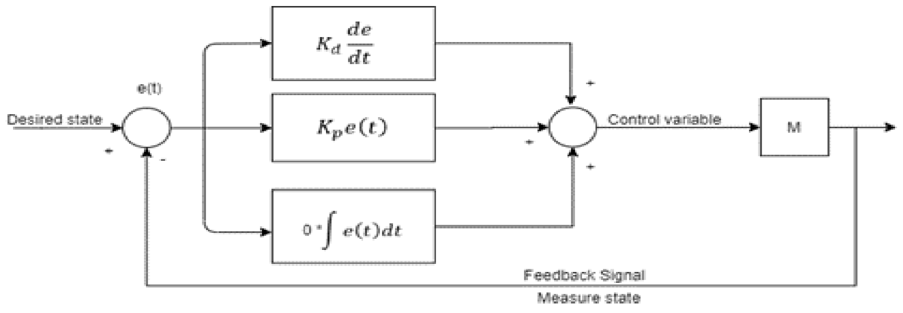


Figure 3: PID controller feedback loop

A typical scenario to apply a PID controller is when we want to adjust the measured state to the desired state effectively and efficiently. As shown in figure 3, the PID controller continuously calculates the difference between the measured state and the desired state, known as an error value e(t), then generates a control signal based on its proportional, integral, and derivative terms. The proportional term P is determined by multiplying the proportional gain (Kp) to the error value. In general, increasing the proportional gain will speed up the process of the control system response. However, if the proportional gain is too large, the applied control signal or the step will be large. Consequently, the measured state will oscillate around the desired state and not be able to reach the desired one. The derivative term D is calculated by multiplying the derivative gain (Kd) to the derivative of the error value. Similarly, the integral term I is generated by multiplying the integral gain (Ki) to the integral of the error value over time. Eventually, the control variable or the control signal is the sum of P, I, and D, three terms. Due to the scope of the project, the integral term is not considered by setting the integral gain (Ki) to 0.

Even though the PD controller has the calculated derivative term to prevent the measured state oscillating forever, there is still a need to adjust the proportional and derivative gain to make the system respond more elegantly. As shown in figure 4, if gains or the steps are too large for the system, the measured state will be bouncing around the desired one. However, if the step is too small then it takes a very long time to reach the desired state.

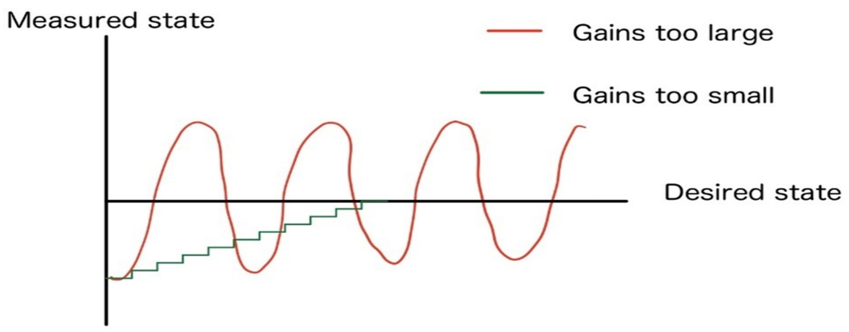


Figure 4: Effect of coefficient(step) of P component

3 Model Specification

This section describes the structure of the DEVS model. It first starts with an overview of the coupled model then provides detailed model specification for each submodule along with its atomic models.

The figure 5 provides an example to explain the process of embedding QSS-PID controller into an existing analog system. All the atomic models are represented by a rectangle. We only implemented atomic models highlighted in red color. They are qssSender, qssReceiver and pidController. The rest of the atomic models such as analogIn are already implemented as RT-DEVS library. The other components such as colorSensor without bounding rectangle represent the physical devices we attached to Nucleo board. The green led is a built-in device of Nucleo board and it has already been abstracted as an atomic model in the library.

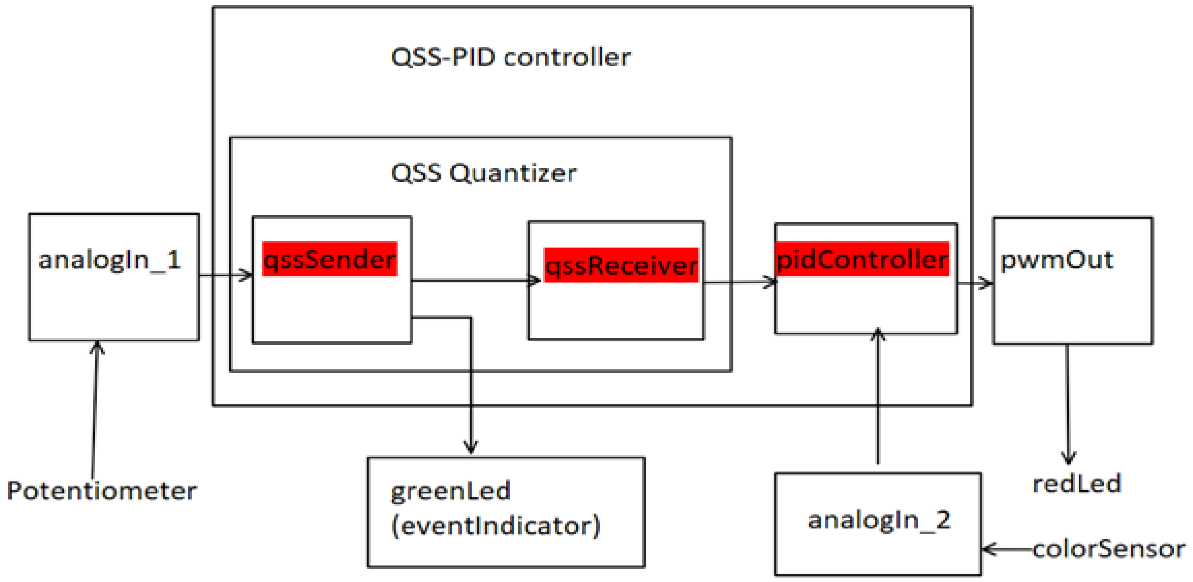


Figure 5: Overview of a QSS-PID based control system

The original system only has anlogIn1 directly connected to pwnOut. To deliver the continuous control signal, which is adjustment of potentiometer in this case, a small discrete-time step is required to sample the control signal. All the samplings will be sent through network and received by the pwmOut to control the redLed smoothly. As a result, large volume of traffic is generated if we want to have a smooth control of led brightness. To apply QSS-PID controller, analogIn_1 and pwnOut can be split up and QSS-PID controller can be inserted in between. The atomic model analogIn_1 provides sampling of signal at discrete-time step to qssSender. The qssSender quantizes the input signal based on predefined state change criteria and calculates the state information. Once a state change is determined, qssSender generates an event by sending out state information to qssReceiver. At the same time, qssSender also sends out a toggled signal to greenLed model so the green led will be toggled. This helps visualizing the events generated when we perform on-target testing. The qssReceiver receives the state information and use this new state information to calculate the estimated anlogIn_1 signal at each discrete-time step. The estimated analogIn_1 value will be sent to pidController at each discrete-time step. The qssReceiver keep doing the estimation until it receives next updated state information from qssSender. The pidController stores the estimated control signal from qssReceiver as desired value. Upon each sensor feedback update (through another analogIn_2 model), pidController calculates the correction value to be sent out to pwmOut model.

3.1 QSS Quantizer Atomic Models

This section describes the formal specification of QSS quantizer. Figure 6 shows the coupled QSS model which has qssSender and qssReceiver connected. Each of the 2 models will be explained in detailed.

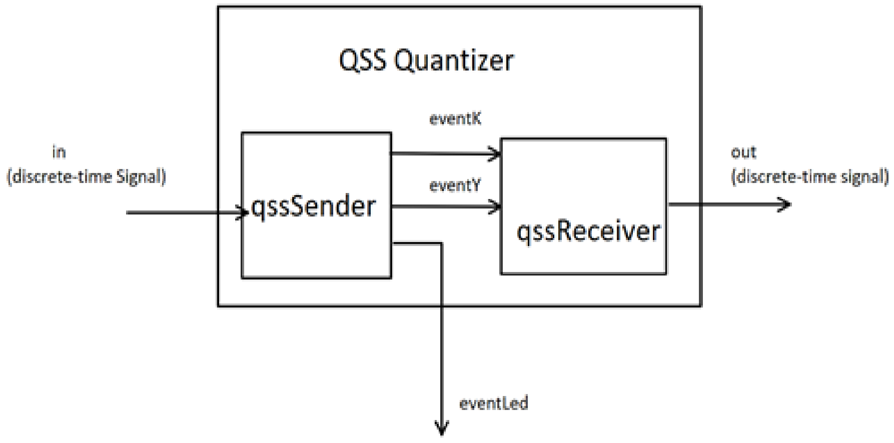


Figure 6: QSS quantizer formed by qssSender and qssReceiver

3.1.1 QSS Sender

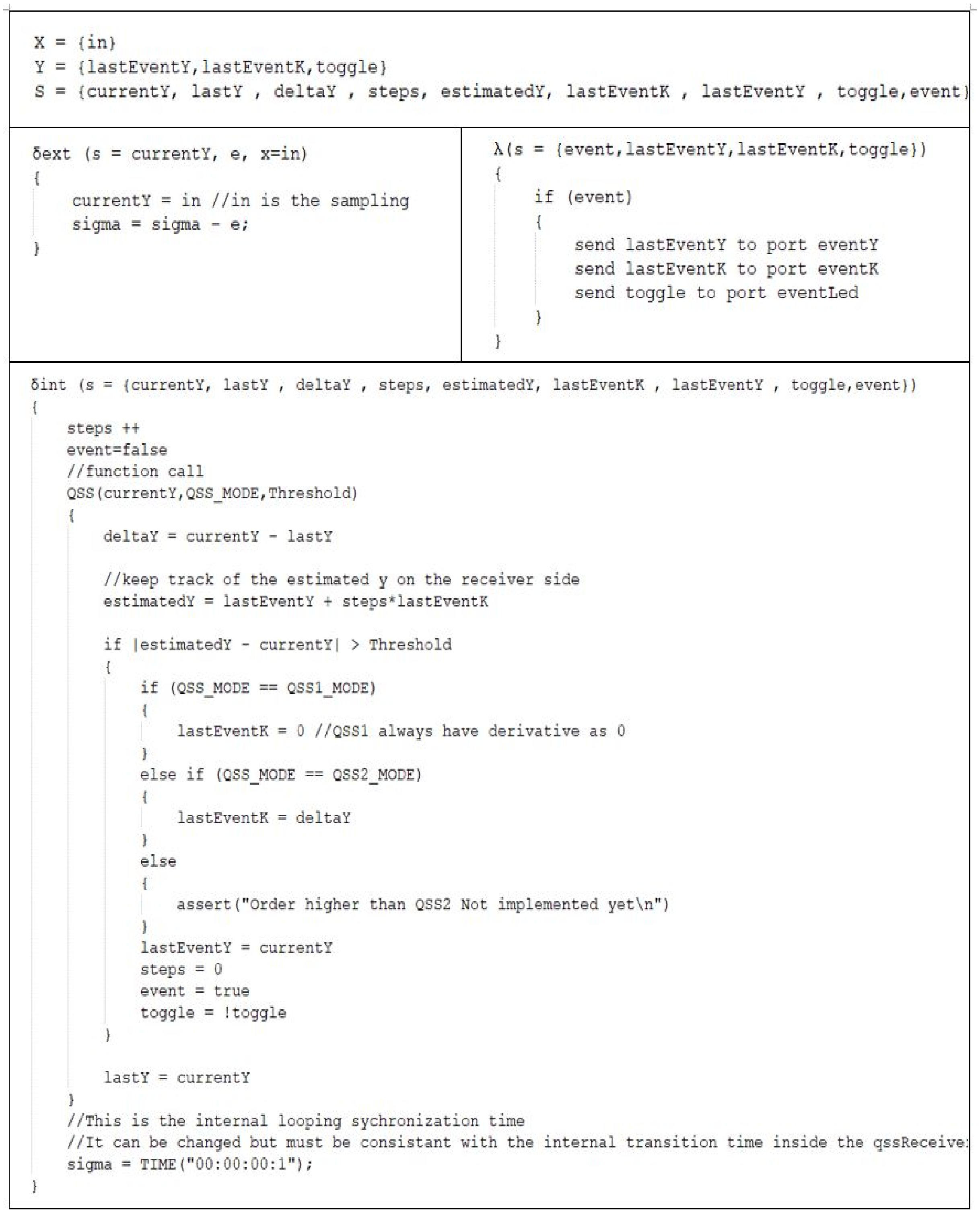
The formal specification of qssSender is provided in figure 7. The major activities performed by qssSender can be summarized as following steps:

1. External transition δext: receives signal sampling, update currentY
2. Internal transition δint: keep track of steps which indicates how many internal transitions has occurred since last event generation. Calculate estimatedY using last state information sent to the qssReceiver. Calculate the difference between estimatedY and currentY. The difference reflects how much estimatedY on the qssReceiver side has deviated from latest signal sampling. Check if the deviation is crossing a predefined threshold (quantum). If difference is above the threshold then an event is generated. Check which QSS method is chosen so corresponding derivative information is included as state information sent out. Reset steps since new event is generated and toggle the led. Prepare and update the state information to be sent out.
3. Output function λ: if an event flag is set by internal transition then send out latest state information
4. It must be mentioned that the time advancement set inside internal transition must be consistent with that on the qssReceiver.

3.1.2 QSS Receiver

The formal specification of qssReceiver is provided in figure 8. The major activities performed by qssReceiver can be summarized as following steps:

1. External transition δext: receives latest update of eventY and eventK. Note that if qssSender is using QSS1, eventK will be simply set to 0. If a new eventY value received (new event), then reset the steps for tracking how many times internal transition has occurred.
2. Internal transition δint: keep track of steps. Increase currentY (estimated) by currentK.
3. Output function λ: send out currentY to output port.
4. Note that time advance set in the internal transition must be consistent with that in the qssSender.



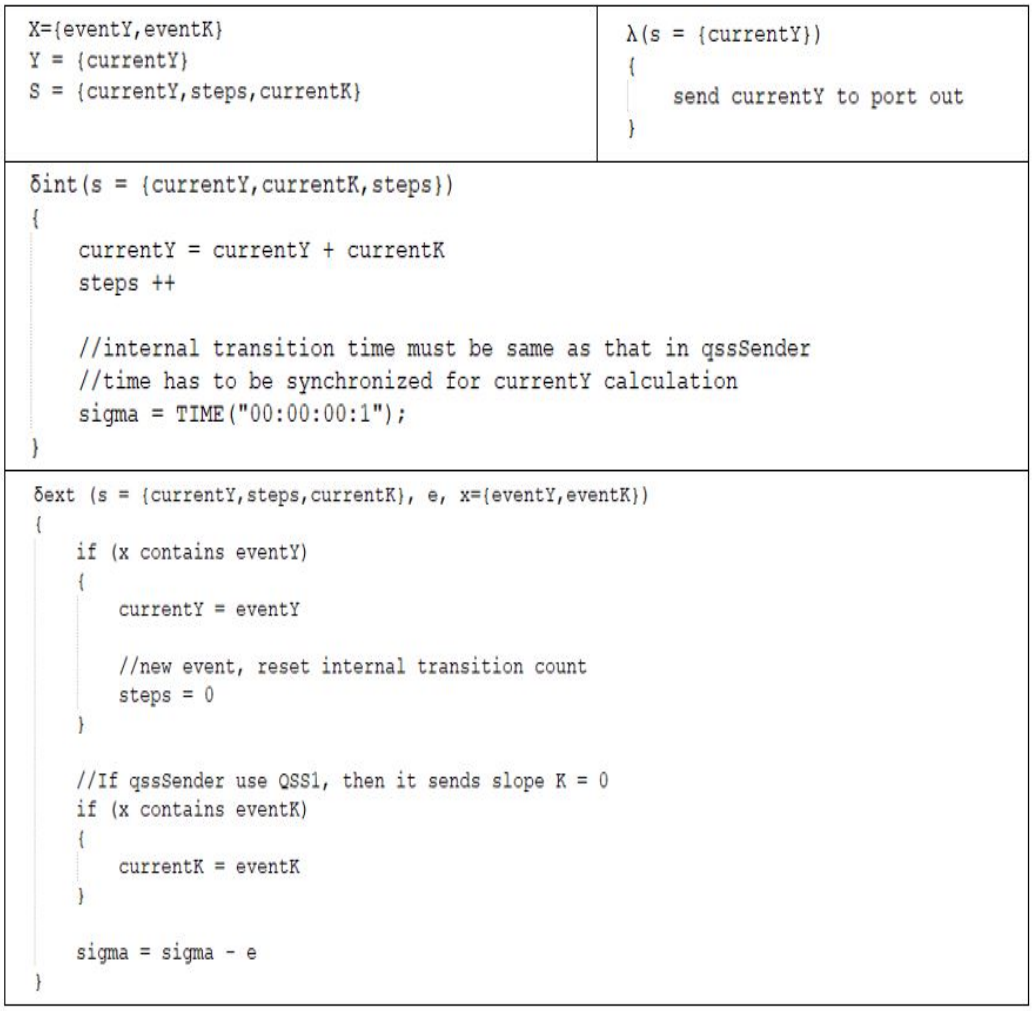
```
X = {in}
Y = {lastEventY,lastEventK,toggle}
S = {currentY, lastY , deltaY , steps, estimatedY, lastEventK , lastEventY , toggle,event}
```

```
δext (s = currentY, e, x=in)
{
    currentY = in //in is the sampling
    sigma = sigma - e;
}
```

```
λ(s = {event,lastEventY,lastEventK,toggle})
{
    if (event)
    {
        send lastEventY to port eventY
        send lastEventK to port eventK
        send toggle to port eventLed
    }
}
```

```
δint (s = {currentY, lastY , deltaY , steps, estimatedY, lastEventK , lastEventY , toggle,event})
{
    steps ++
    event=false
    //function call
    QSS(currentY,QSS_MODE,Threshold)
    {
        deltaY = currentY - lastY

        //keep track of the estimated y on the receiver side
        estimatedY = lastEventY + steps*lastEventK

        if |estimatedY - currentY| > Threshold
        {
            if (QSS_MODE == QSS1_MODE)
            {
                lastEventK = 0 //QSS1 always have derivative as 0
            }
            else if (QSS_MODE == QSS2_MODE)
            {
                lastEventK = deltaY
            }
            else
            {
                assert("Order higher than QSS2 Not implemented yet\n")
            }
            lastEventY = currentY
            steps = 0
            event = true
            toggle = !toggle
        }

        lastY = currentY
    }
    //This is the internal looping sychronization time
    //It can be changed but must be consistant with the internal transition time inside the qssReceive
    sigma = TIME("00:00:00:1");
}
```

Figure 7: Formal specification of qssSender



```
X={eventY,eventK}
Y = {currentY}
S = {currentY,steps,currentK}
```

```
λ(s = {currentY})
{
    send currentY to port out
}
```

```
δint(s = {currentY,currentK,steps})
{
    currentY = currentY + currentK
    steps ++

    //internal transition time must be same as that in qssSender
    //time has to be synchronized for currentY calculation
    sigma = TIME("00:00:00:1");
}
```

```
δext (s = {currentY,steps,currentK}, e, x={eventY,eventK})
{
    if (x contains eventY)
    {
        currentY = eventY

        //new event, reset internal transition count
        steps = 0
    }

    //If qssSender use QSS1, then it sends slope K = 0
    if (x contains eventK)
    {
        currentK = eventK
    }

    sigma = sigma - e
}
```

Figure 8: Formal specification of qssReceiver

3.2 PID Controller Atomic Model

This section describes the formal specification of PID controller. We only implemented P and D components.

As shown in figure 9, the pidController receives the desired value of the state variable as reference point. If it is connected to a QSS quantizer then the desired value is provided by qssReceiver. The pidController also receives measured value from the sensor. To adjust the coefficient of P and D components, analog input tuneP and tuneD are also provided. For every measured sensor value comes in, the pidController calculates the correction value and send it to out port.

The formal specification of pidController is provided in figure 10. The D component of the PID controller needs the time elapsed between current currentError and previousError. The internal transition has a fixed time advance which is used as local time reference for the pidController atomic model. The internal transition also increases steps by 1 for every call. Upon calculating D component, it is only necessary to keep track of steps since the time advance value can be incorporated with Kd which is coefficient of D component. The major activities performed by pidController are as followings:

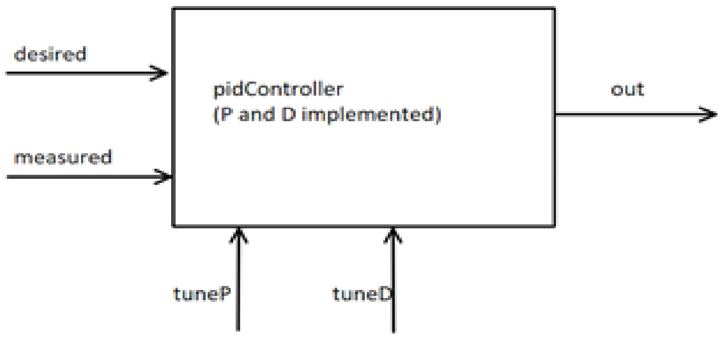


Figure 9: PID controller conceptual model

1. It receives desired value as input. This value can be provided by QSS quantizer.
2. It receives measured value from the sensor as feedback value of the state variable under control.
3. It calculates the currentError and store previousError, then calculate the correctionValue to be sent out.
4. It receives tuneP and tuneD as input to tune the P and D coefficient.
5. It uses internal transition to keep track of time steps have elapsed between 2 consecutive measurement input.

It is important to note that the internal transition should have smaller period (time advance) compared with the updating period of desired value and measured value. At the same time, desired value should be updated with longer period compared with measured value as PID controller typically needs some iterations to reach desired value.

4 Simulation and Testing

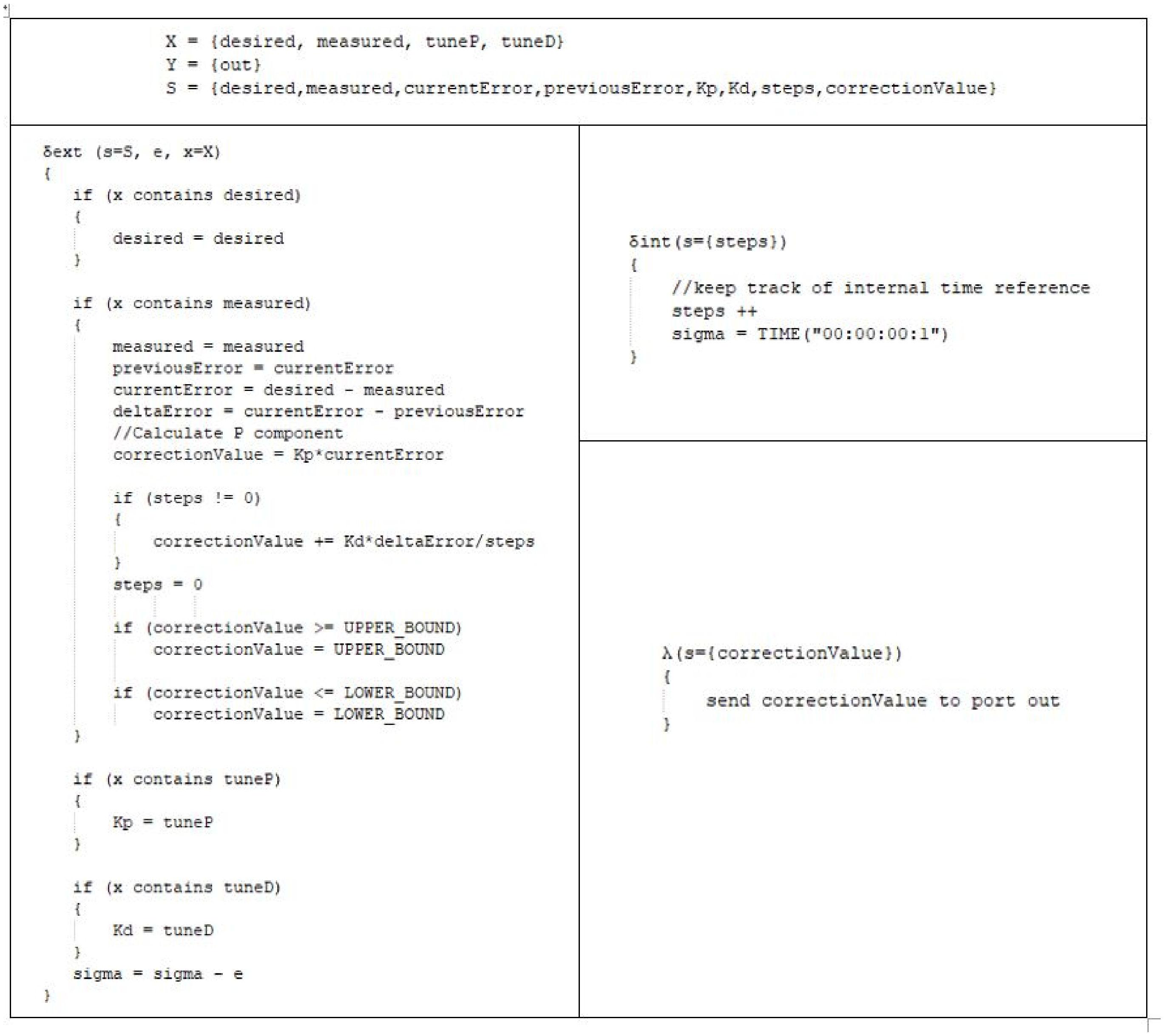
This section presents the simulation and testing of QSS and PID performed within the RT-DEVS framework. In addition, the QSS quantizer is also tested with hardware for controlling a LED light with QSS2 algorithm.

4.1 Simulation for QSS Quantizer

QSS quantizer consists of qssSender and qssReceiver. It is more efficient to test these 2 atomic models together as we can easily compare the input to qssSender and output from the qssReceiver. The code for assembling top model is shown in figure 11. We start with building a top model which connects qssSender and qssReceiver. Next, we create the required input/output files for simulating the input and output. The files created includes input files for simulating the sampled signal fed into qssSender for quantization, output files for checking the signal reconstructed by qssReceiver and events occurred (LED toggling file). The files which represent some physical components are connected to qssSender and qssReceiver. The atomic models are connected in the same way as shown in figure 6.

We start with QSS2 algorithm testing as shown in figure 12. A python program is developed to write sampled signal points into input file. The python program can also be used to parse the input/output file so the QSS quantizer testing result can be visualized. Next step is to compile the top model and execute it.

In figure 12, the input is a sinusoidal signal(with some noise added) sampled with 100 discrete-time steps represented by blue dots. The red dots represent the signal reconstructed by the qssReceiver. On the right side of the figure is the file representing the signals sent to the toggling LED from qssSender for visualizing the number of events generated. In this testing, a total of 20 events are generated by qssSender while there are 100 discrete-time steps fed into qssSender. The qssReceiver reconstructs the signal very well.



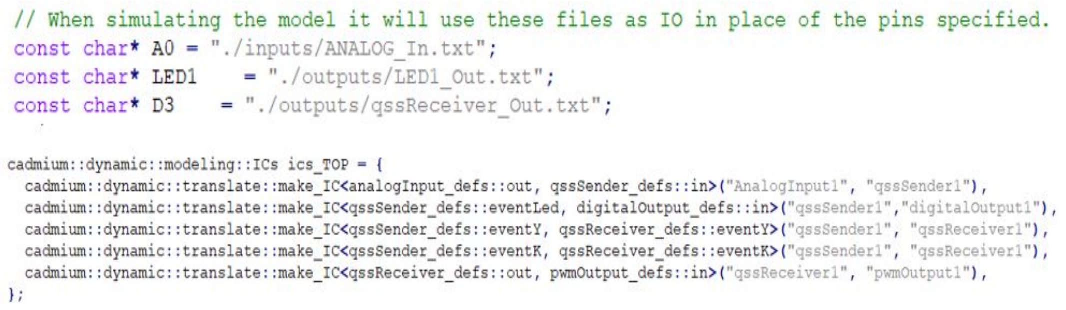
```
X = {desired, measured, tuneP, tuneD}
Y = {out}
S = {desired,measured,currentError,previousError,Kp,Kd,steps,correctionValue}
```

```
δext (s=S, e, x=X)
{
   if (x contains desired)
   {
       desired = desired
   }

   if (x contains measured)
   {
       measured = measured
       previousError = currentError
       currentError = desired - measured
       deltaError = currentError - previousError
       //Calculate P component
       correctionValue = Kp*currentError

       if (steps != 0)
       {
           correctionValue += Kd*deltaError/steps
       }
       steps = 0

       if (correctionValue >= UPPER_BOUND)
           correctionValue = UPPER_BOUND

       if (correctionValue <= LOWER_BOUND)
           correctionValue = LOWER_BOUND
   }

   if (x contains tuneP)
   {
       Kp = tuneP
   }

   if (x contains tuneD)
   {
       Kd = tuneD
   }
   sigma = sigma - e
}
```

```
δint(s={steps})
{
    //keep track of internal time reference
    steps ++
    sigma = TIME("00:00:00:1")
}
```

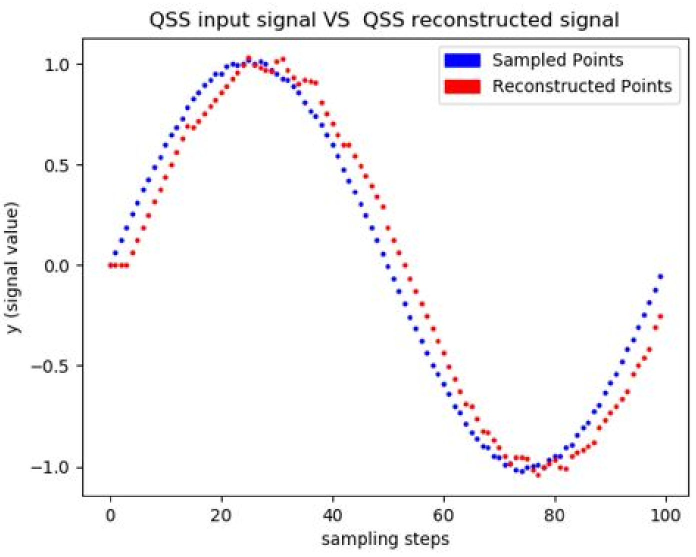
```
λ(s={correctionValue})
{
    send correctionValue to port out
}
```

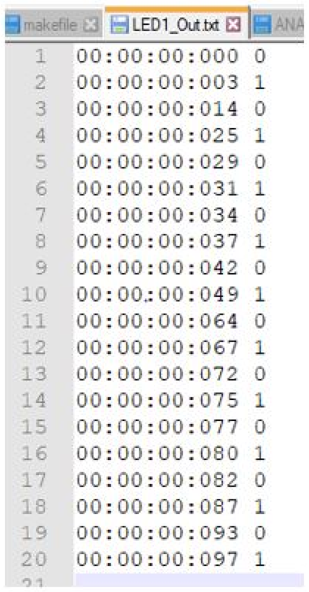
Figure 10: Formal specification of PID controller



```
// When simulating the model it will use these files as IO in place of the pins specified.
const char* A0 = "./inputs/ANALOG_In.txt";
const char* LED1    = "./outputs/LED1_Out.txt";
const char* D3    = "./outputs/qssReceiver_Out.txt";

cadmium::dynamic::modeling::ICs ics_TOP = {
  cadmium::dynamic::translate::make_IC<analogInput_defs::out, qssSender_defs::in>("AnalogInput1", "qssSender1"),
  cadmium::dynamic::translate::make_IC<qssSender_defs::eventLed, digitalOutput_defs::in>("qssSender1","digitalOutput1"),
  cadmium::dynamic::translate::make_IC<qssSender_defs::eventY, qssReceiver_defs::eventY>("qssSender1", "qssReceiver1"),
  cadmium::dynamic::translate::make_IC<qssSender_defs::eventK, qssReceiver_defs::eventK>("qssSender1", "qssReceiver1"),
  cadmium::dynamic::translate::make_IC<qssReceiver_defs::out, pwmOutput_defs::in>("qssReceiver1", "pwmOutput1"),
};
```

Figure 11: Top model of QSS quantizer simulation

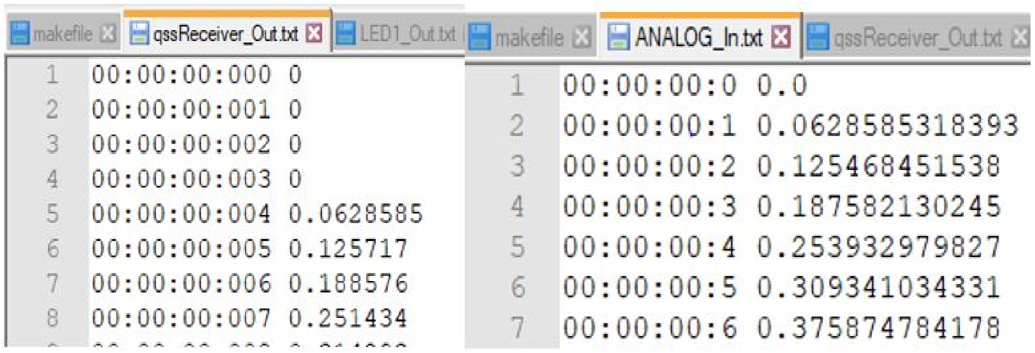




```
00:00:00:000 0
00:00:00:003 1
00:00:00:014 0
00:00:00:025 1
00:00:00:029 0
00:00:00:031 1
00:00:00:034 0
00:00:00:037 1
00:00:00:042 0
00:00:00:049 1
00:00:00:064 0
00:00:00:067 1
00:00:00:072 0
00:00:00:075 1
00:00:00:077 0
00:00:00:080 1
00:00:00:082 0
00:00:00:087 1
00:00:00:093 0
00:00:00:097 1
```

Figure 12: QSS2 simulation with sinusoidal signal as input

However, figure 13 shows that qssReceiver constructs the signal with 3 synchronization steps late. In the file qssReceiver_Out.txt, the first signal point is constructed at 00:00:00:004. This is caused by our synchronization mechanism inside the internal transition explained earlier. For real-world testing, this synchronization caused delay is negligible as the physical network delay is much larger than this one.

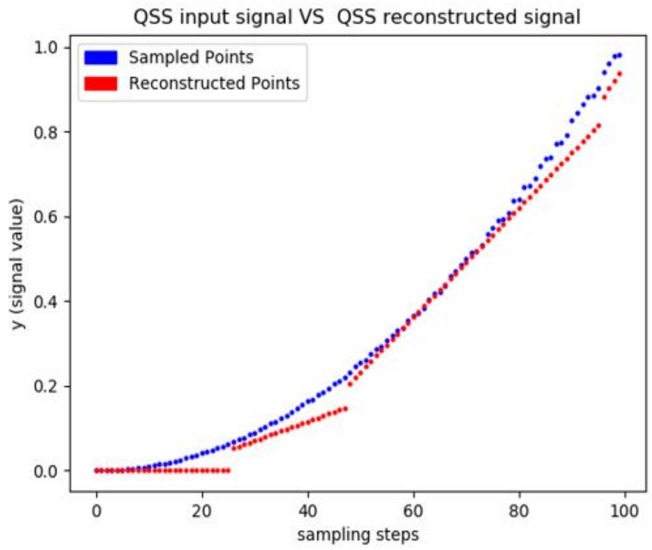


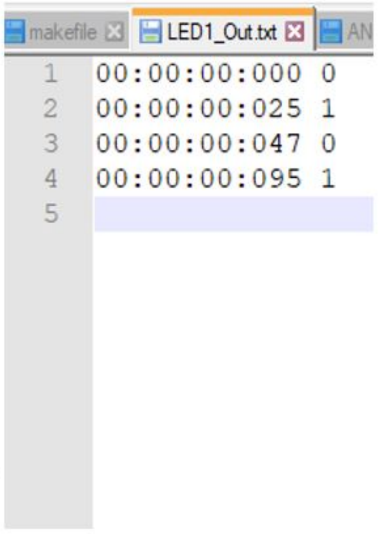
```
00:00:00:000 0
00:00:00:001 0
00:00:00:002 0
00:00:00:003 0
00:00:00:004 0.0628585
00:00:00:005 0.125717
00:00:00:006 0.188576
00:00:00:007 0.251434
```

```
00:00:00:0 0.0
00:00:00:1 0.0628585318393
00:00:00:2 0.12546845153 8
00:00:00:3 0.187582130245
00:00:00:4 0.253932979827
00:00:00:5 0.309341034331
00:00:00:6 0.375874784178
```

Figure 13: qssReceiver reconstructs the signal with delay

Figure 14 shows the sampling signal in the form of x^2. In this case, only 4 events are generated with QSS2 algorithm.





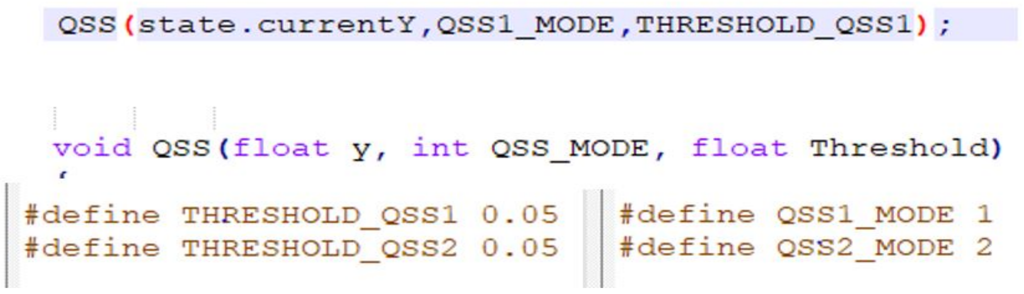
```
00:00:00:000 0
00:00:00:025 1
00:00:00:047 0
00:00:00:095 1
```

Figure 14: QSS2 simulation with x^2 signal as input

Next, qssSender.hpp is modified to use QSS1 algorithm as shown in figure 15. Note that we are using same quantum size of 0.05 for both QSS1 and QSS2.

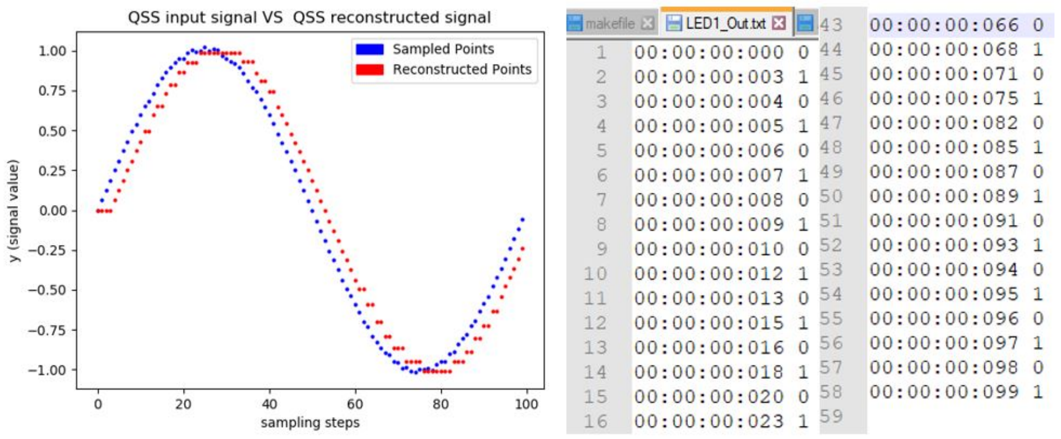
As shown in figure 16, the QSS1 algorithm generates 58 events from 100 discrete-time steps. Though it can reconstruct the signal with acceptable accuracy, the number of generated events is much bigger compared with 20 events using QSS2.

Figure 17 shows QSS1 modelling the input in the form of x^2. There are 17 events generated compared with only 4 events generated by QSS2.



```
QSS(state.currentY,QSS1_MODE,THRESHOLD_QSS1);

void QSS(float y, int QSS_MODE, float Threshold)
{
#define THRESHOLD_QSS1 0.05    #define QSS1_MODE 1
#define THRESHOLD_QSS2 0.05    #define QSS2_MODE 2
```

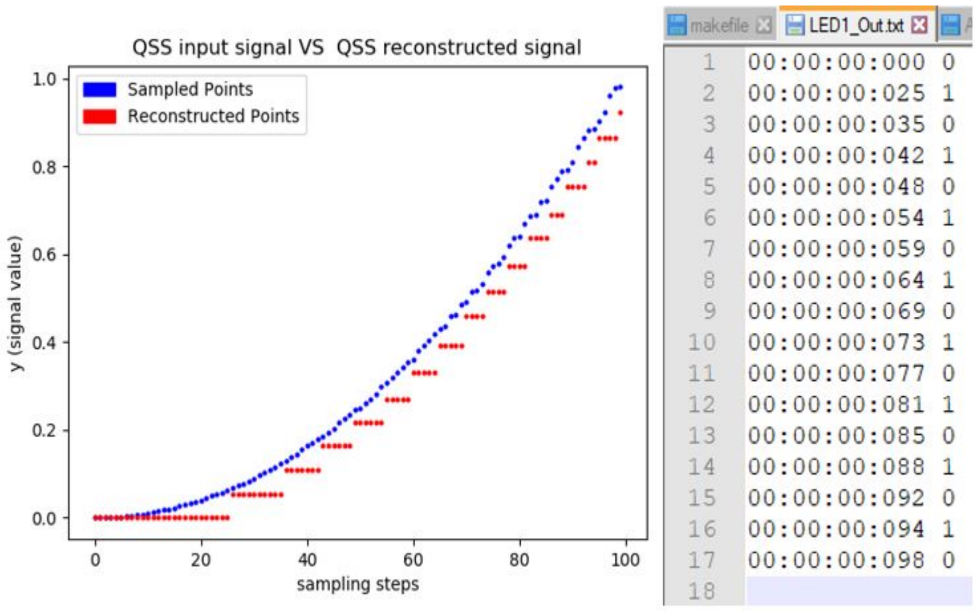
Figure 15: Code modification for using QSS1 algorithm



```
00:00:00:000 0
00:00:00:003 1
00:00:00:004 0
00:00:00:005 1
00:00:00:006 0
00:00:00:007 1
00:00:00:008 0
00:00:00:009 1
00:00:00:010 0
00:00:00:012 1
00:00:00:013 0
00:00:00:015 1
00:00:00:016 0
00:00:00:018 1
00:00:00:020 0
00:00:00:023 1
```

```
00:00:00:066 0
00:00:00:068 1
00:00:00:071 0
00:00:00:075 1
00:00:00:082 0
00:00:00:085 1
00:00:00:087 0
00:00:00:089 1
00:00:00:091 0
00:00:00:093 1
00:00:00:094 0
00:00:00:095 1
00:00:00:096 0
00:00:00:097 1
00:00:00:098 0
00:00:00:099 1
```

Figure 16: QSS1 simulation with sinusoidal signal as input



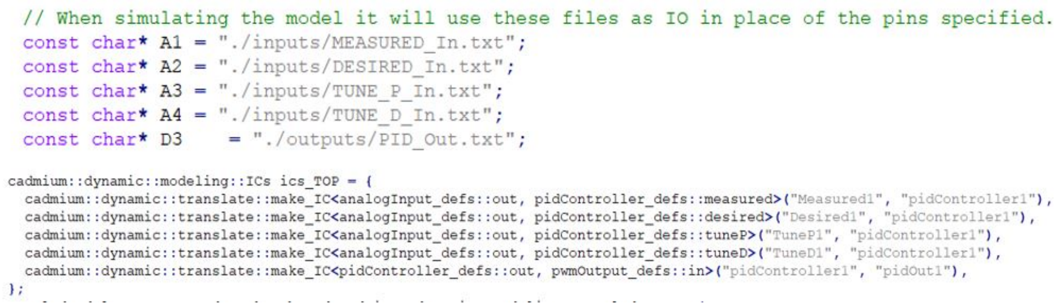
```
00:00:00:000 0
00:00:00:025 1
00:00:00:035 0
00:00:00:042 1
00:00:00:048 0
00:00:00:054 1
00:00:00:059 0
00:00:00:064 1
00:00:00:069 0
00:00:00:073 1
00:00:00:077 0
00:00:00:081 1
00:00:00:085 0
00:00:00:088 1
00:00:00:092 0
00:00:00:094 1
00:00:00:098 0
```

Figure 17: QSS1 simulation with x^2 signal as input

The testing done in the simulation proves that the atomic model qssSender and qssReceiver are working as expected.

4.2 Simulation for PID Controller

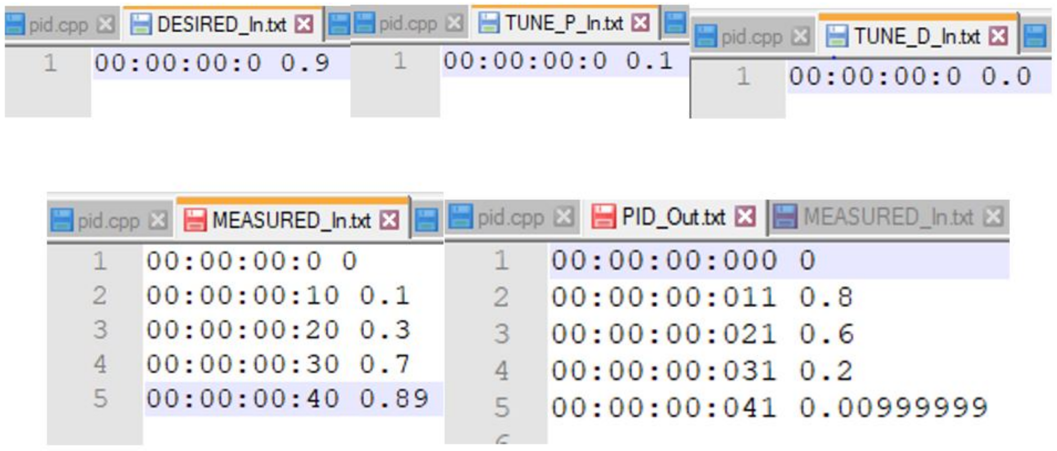
As shown in figure 18, we are using same approach as building QSS top model to build PID top model for simulation. The simulation of PID controller is more challenging than that of QSS quantizer as the PID controller does not have such input/output matching for comparison. We will focus on smaller scale of testing by using only few consecutive measured sensor feedback samplings with a single desired value.



```
// When simulating the model it will use these files as IO in place of the pins specified.
const char* A1 = "./inputs/MEASURED_In.txt";
const char* A2 = "./inputs/DESIRED_In.txt";
const char* A3 = "./inputs/TUNE_P_In.txt";
const char* A4 = "./inputs/TUNE_D_In.txt";
const char* D3    = "./outputs/PID_Out.txt";

cadmium::dynamic::modeling::ICs ics_TOP = {
  cadmium::dynamic::translate::make_IC<analogInput_defs::out, pidController_defs::measured>("Measured1", "pidController1"),
  cadmium::dynamic::translate::make_IC<analogInput_defs::out, pidController_defs::desired>("Desired1", "pidController1"),
  cadmium::dynamic::translate::make_IC<analogInput_defs::out, pidController_defs::tuneP>("TuneP1", "pidController1"),
  cadmium::dynamic::translate::make_IC<analogInput_defs::out, pidController_defs::tuneD>("TuneD1", "pidController1"),
  cadmium::dynamic::translate::make_IC<pidController_defs::out, pwmOutput_defs::in>("pidController1", "pidOut1"),
};
```

Figure 18: Top model for PID controller simulation

As shown in figure 19, we start with testing P component by setting tuneD as 0 in the input file TUNE_D_In.txt. The coefficient is 1 by setting tuneP as 0.1 multiplied by a factor of 10 inside the program. The desired value is set as 0.9 in the input file. Given the desired value 0.9, the correction value should be 0.8, 0.6, 0.2, 0. 001 based on measured value (sensor feedback). The model generates expected correction value as indicated in file PID_Out.txt. The redundant time step is removed from the output file and the original file have repeated value at every 00:00:00:001 discrete-time step. However, it is noticeable that there is 1 discrete-time step delay on the output side due to our synchronization mechanism. This delay is negligible if the updating period of measured value is much bigger.



```
00:00:00:0 0.9
```

```
00:00:00:0 0.1
```

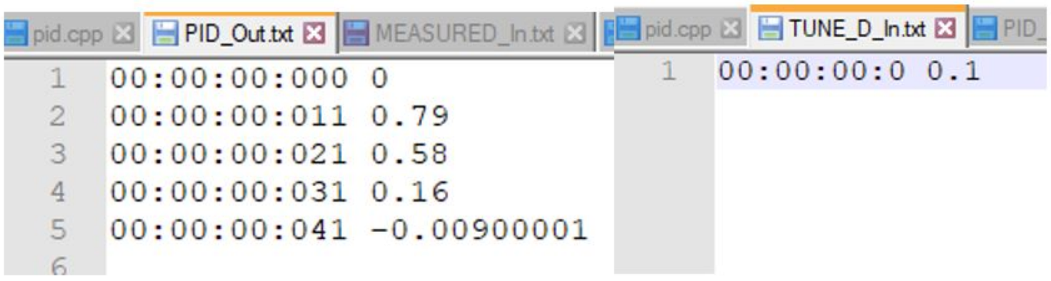
```
00:00:00:0 0.0
```

```
00:00:00:0 0
00:00:00:10 0.1
00:00:00:20 0.3
00:00:00:30 0.7
00:00:00:40 0.89
```

```
00:00:00:000 0
00:00:00:011 0.8
00:00:00:021 0.6
00:00:00:031 0.2
00:00:00:041 0.00999999
```

Figure 19: Input and output for PID controller without D component

As shown in figure 20, by setting tuneD as 0.1, we are setting coefficient of D component as 1. This will include the rate of change of error as a factor for calculating correction value output. With other setup exactly same, the error is decreasing at each time when measured value is updated. This should lead to D component to be negative. As a result, the correction value taken by P plus D component should be smaller than a single P component. The output file PID_Out.txt generated from simulation matches the expected behavior. When the measured value reaches desired value at last step, the PD component applies negative correction value to reduce the overshoot produced by a single P component.



```
00:00:00:000 0
00:00:00:011 0.79
00:00:00:021 0.58
00:00:00:031 0.16
00:00:00:041 -0.00900001
```

```
00:00:00:0 0.1
```

Figure 20: output for PID controller with D component

The testing done in the simulation for PID controller shows that P plus D component works as expected. However, PID controller should be tested with more test cases especially the P,D parameter tuning process. Current measured value is manually generated. Due to limited resources, more testing should be performed with a simulated PID feedback environment as future work. The simulated PID feedback environment should provide a virtual measured value update based on last correction value applied.

5 On-Target Testing of QSS Quantizer

This section briefly presents the on target-testing of QSS2 algorithm. After the simulation of QSS models proves the correctness of QSS algorithms, we build the top model and flash the executable into Nucleo board for on-target testing. The top model has exactly same structure as the one used for QSS simulation but built with different command. Potentiometer is rotated to adjust desired brightness of red LED light. One toggle of green LED indicates one event generated. It is observed that, over 16 seconds, QSS2 algorithm has achieved smooth control of the brightness of the red LED with only 24 events generated (green LED flashed 12 times). Figure 21 shows the 3 important hardware components which are green LED for events generation, red LED to be controlled and potentiometer for adjusting input signal. Demonstration video can be accessed at the YouTube link.

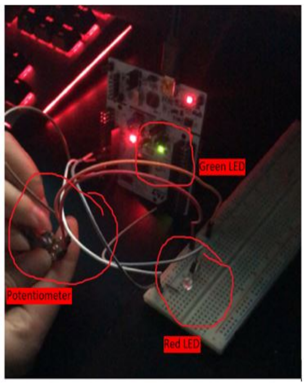


Figure 21: highlight of physical components for visualizing QSS2 algorithm

6 Conclusion

We have implemented QSS and PID atomic modes inside RT-DEVS framework. QSS model consists of 2 atomic models which are qssSender and qssReceiver. The QSS can be applied by splitting up an existing system, attach qssSender to the original sender and insert qssReceiver before the original receiver. The simulation shows that the QSS algorithm implementation is correct and it can effectively reduce events generation while still achieving satisfying control of a system state variable. The on-target testing performed with QSS2 controlled LED further verifies the correctness of algorithm implementation. The PID atomic model is verified with simulation but more comprehensive simulation is desired as future work. It is worth mentioning that some delay between input and output for both QSS and PID models is observed during simulation. This is introduced by our synchronization mechanism and it is negligible compared with real-world network delay.

REFERENCES

Niyonkuru, D., and G. Wainer. (2016) *A kernel for Embedded Systems Development and Simulation Using the Boost Library*. In Proceedings of the Symposium on Theory of Modelling and Simulation Society for Computer Simulation International, April 3rd-6th, Pasadena, California, 13.

Kofman, E. , and S. Junco . (2001). *Quantized state systems: A DEVS approach for continuous system simulation.*. Transactions of SCS 18 (3):123-132.

Kofman, E. (2002). *A Second Order Approximation for DEVS Simulation of Continuous Systems.*. Simulation, 78, 76–89.

Kofman, E. (2006). *A Third Order Discrete Event Simulation Method for Continuous System Simulation.*. Latin American Applied Research, 36(2):101–108.