PID CONTROL USING QUANTIZED STATE SYSTEMS IN RT-DEVS FRAMEWORKS

Mahya Shahmohammadimehrjardi
Gabriel Wainer
Mengyao Wu
Xinrui Zhang
Department of System and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON, CANADA
mahyashahmohammadim@cmail.carleton.ca, gwainer@sce.carleton.ca,
mengyaowu@cmail.carleton.ca, xinruizhang@cmail.carleton.ca

ABSTRACT

RT-DEVS is a framework for integrating DEVS structured real-time embedded system design into microcontroller-based devices. In a Quantized State System (QSS), events are only produced when the system state has changed by a predefined value. PID controllers use a control loop to generate feedback by computing a correction based on the **P**roportional, **I**ntegral, and **D**erivative of the difference between the measured and intended values of a system state variable. Both the piecewise constant (QSS1) and piecewise linear (QSS2) quantized state integration schemes are implemented and tested in this work. Then a QSS-based PID controller using Real-Time DEVS is defined and built and evaluated through simulation and on-target testing, showing how we can use this method for PID control, reducing the volume of messages while maintaining reliable control.

Keywords: DEVS, RT-DEVS, QSS, QSS-PID Controller.

1 INTRODUCTION

Human life has been greatly impacted by automatic machines and robots, and scientists predict that this impact will only increase in the future. Many of these machines may be categorized as Cyber Physical Systems (CPS), defined as an orchestration of embedded computers and physical systems in which computers control physical processes, which affect computations (Lee 2015). A CPS consists of a physical part, a cyber part, controllers, actuators, and other components, such as communication devices. In these systems the passage of time is a central feature of system behavior (Cho et al. 2020) and timing constrains must be satisfied by implementing Real Time (RT) control. However, there are challenges with RT control implementation: their development is time consuming, error prone, and expensive (Boi-Ukeme et al. 2020). To overcome these challenges, formal methods for RT analysis have shown promising results, however, do not scale well due to explosion of states and complexities in the analysis. Furthermore, in formal methods, the focus is on software specifications, and as a result, the interactions between the formal model representing the CPS becomes not simple to define and study.

Modeling and Simulation (M&S) has been used as an alternative to deal with these issues, but when building CPS, initial models and simulations are often abandoned when the software needs to be deployed on the target hardware. Our research proposes building Discrete-Event Modeling of Embedded Systems,

and the DEVS formalism (Discrete Event System Specification) is the tool of choice. DEVS is a formal M&S theory that is well tailored to define and study RT software as well as the physical environment they control (Zeigler 2000). DEVS is an increasingly accepted framework that provides an abstract and intuitive way of modeling, independent of underlying simulators, hardware, and middleware, and allows us to use the same models consistently throughout the development cycle (making original models part of the final product).

In CPS with control and feedback loops, we need to combine continuous models with discrete-event supervisory control. For instance, in traditional Proportional-Integral-Derivative (PID) controllers, we calculate the correction value needed to bring a system state variable to a desired value. The correction value is calculated by adding components together that are Proportional, Integral, and Derivative of the error (the difference between a measured and desired system state variable). Simulating these systems needs continuous model simulation, which needs to define a sample timestep (and to provide high accuracy, a very small, discrete-time step is required, increasing computational costs). Instead, Quantized State System (QSS) provides a way that can be applied to reduce the number of events generated for reconstructing a continuous control or feedback signal (Kofman 2002, 2006). 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. QSS can reduce the number of events generated significantly while still providing adequate amount of information to reconstruct the signal.

Combining DEVS with QSS allows us to discretize continuous systems efficiently, as QSS can be exactly represented and simulated by a discrete event model (in particular within the framework of the DEVS approach). Here we show how to combine these methods to build a QSS-PID controller in DEVS, which minimizes state information exchange among physical components while still achieve a satisfying control of the target state variable. We discuss both the piecewise constant (QSS1) and piecewise linear (QSS2) quantized state integration schemes and its implementations and provide a case study to verify and validate the performance of applying QSS in a hybrid system.

The paper is organized as follows. Section 2 gives an introduction on DEVS formalism, RT-DEVS, QSS and some related works. Development of a QSS-PID controller in RT-DEVS framework is described in detail in section 3. Case study and simulation results are investigated in section 4 And lastly, conclusion and new opportunities for research are discussed in section 5.

2 BACKGROUND AND RELATED WORK

Discrete-event systems specification (DEVS) is one of the formal modelling approaches in M&S theory (Zeigler 2000). According to DEVS, a system of interest is seen as a source of behavioral data for the study within a given experimental frame. Using a set of instructions, rules, or mathematical equations, the model tries to replicate the behavior of the system of interest under the experimental conditions. A real system modeled using DEVS can be described as a composition of *atomic* and *coupled* components.

DEVS has been extended as RT-DEVS to model real time systems where predictability of the execution time is essential, like in avionic control systems or braking systems in autonomous cars (Hong et al. 1997) RT-DEVS provides the possibility of executing real time models by adding a time interval function (time-windows). The time windows restrict the simulation time to be completed within a real time (Boi-Ukeme et al. 2020). RT-DEVS was first introduced by Zeigler and Kim (1993), for RT event-based control. In Hong et al. (1997), DEVS for RT that models and simulates a multi-tasking parallel operating system was defined and used for performance degradation estimation and overall system throughput. In Sarjoughian et al. (2013), ALRT-DEVS is used for hard RT simulation which, offering abstract communication for the interaction of software systems. Other research has focused on incorporating real-time functionalities in a DEVS environment. Cho and Kim (2001), introduces a scheduling policy and a method for evaluating its viability in RT-DEVS. A real-time simulator known as E-CD++ was presented in Niyonkuru and Wainer (2016), as an embedded version of CD++ tool. A related research effort, presented in Hu et al. (2001),

shows how to use DEVS on a TINI Chip, with minimal memory and computing power. In Boukerche et al. (2007), RT-DEVS was deployed in a Run-Time Infrastructure (RTI) for designing a RT-RTI (Real Time RTI). A frame-based RT event scheduling algorithm with RT-DEVS was presented in Zhang et al. (2018). This real-time event scheduling module was then assessed for adequacy and efficiency.

QSS was first introduced by Kofman and Junco (2001). 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 applied to RT simulation, it avoids iterative solving and backtracking at discrete-time steps which are typically used in traditional numerical method. First order QSS can be exactly represented and simulated by DEVS (Kofman and Junco 2001). A second-order approximation (Kofman 2002) called second order QSS (QSS2) preserves the properties of QSS (stability and convergence) while reducing the number of calculations with respect to QSS1. QSS3 (Kofman 2006) shows advantages in the integration of discontinuous systems.

If first order quantizers are applied to the system instead of zero order quantizers (which are used in QSS1), the system will become second order Quantized State System (QSS2*)* (Kofman 2002). In QSS2, the outputs are generated when the difference between the input and current output reaches a threshold (Δq), the output is set to a slope starting at the new input value. Amongst several ways of deciding on the slopes of the outputs, two common ways were proposed. One is to obtain the derivative of the first input trajectory. Another is to obtain the slope from the previous output (subtracting the latest input from the last output from the first point of the same output and dividing it over the time difference). The second method was used in this work since only requires the current value and the previous value to calculate the slope. By considering a QSS1 quantizer being applied to a real time system, an event is generated when an input goes to the QSS and the difference between current input and the sampling have passed the threshold. The latest sampling value is required for reconstructing the signal and it needs to be transmitted as the state information. In a QSS2 quantizer, in addition to the latest sampling value, information about the slope of the trajectory needs to be transmitted and the slope can be extracted from the two previous consecutive input points ($y_k - y_{k-1}$).

The most basic type of QSS is QSS1, which defines a quantization function with hysteresis to create piecewise constant trajectories from piecewise linear input trajectories (Fernández and Kofman 2014). A quantizer equipped with quantization function with hysteresis is known as a zero order quantizer. A system can be called QSS1 if we include these zero order quantizers at the output of the continuous system integrators. A typical quantization function with hysteresis has a set of real numbers as quantized values Q = $\{d_0, d_1, \ldots, d_r\}$, where $d_{i-1} < d_i$ with $1 \leq i \leq r$. According to this quantization function, the output constant trajectory would be updated to the new quantized value $q(t) = d_{i+1}$ if the input trajectories x(t) has positive slope and reach the subsequent quantized value d_{i+1} (within upper saturation value). The output constant trajectory would then be updated to the prior quantized value, $q(t) = d_{i-1}$, if the input trajectory x(t) has a negative slope and exits the hysteresis window range $\pm\varepsilon$. Otherwise, $q(t)$ remains the same d_i (to avoid the chattering effect, ε is considered for down falling trajectories). To understand the QSS algorithm applied to the system, let *y* represent the latest sampling of the input signal. Then, y is the source signal and, $\bar{y}$ will be the reconstructed or estimated signal. When the difference between the latest y sampling and the estimated $\bar{\boldsymbol{y}}$ value (unknown value) is greater than the quantum size (Δq), an event will be generated. The quantum size is chosen by the designer.

In QSS1, the state information contains latest sampling of y. In higher order algorithms, in addition to y, the state information also contains different order of derivatives of signal y which can be denoted by y', y'' and so on. We keep track of sampled y value and calculated different order of derivatives. Upon generating events, *y* and all the derivatives are transmitted as state information.

Table I: State Information used by different QSS algorithms.

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

Table I summarizes the state information required for QSS1 and QSS2 methods. QSS1 only sends the latest signal sampling y value. This implies the derivative of any order is zero hence the signal reconstructed is piece-wise constant. QSS2 sends both y and y' so that the reconstructed signal is piece-wise linear. To reconstruct the y signal an integrator is used. In equation 3, k represents the k^{th} time step and $\boldsymbol{\Delta t}$ is the time advancement for each time step. For QSS1, the y' is 0.

$$\overline{\boldsymbol{y_k}} = \overline{\boldsymbol{y_{k-1}}} + \boldsymbol{\Delta t} * \acute{\boldsymbol{y}} \tag{3}$$

If $\boldsymbol{\Delta t}$ is a fixed discrete-time step, equation (3) can be simplified:

$$\overline{\boldsymbol{y_k}} = \overline{\boldsymbol{y_{k-1}}} + \acute{\boldsymbol{y}} \tag{4}$$

At any arbitrary discrete-time step, if no events are generated, $\boldsymbol{y_k}$ stays the same as the previous value for QSS1 while increases by latest y' received for QSS2. If it receives an event, it will update $\overline{\boldsymbol{y_k}}$ to be the latest y value.

QSS signal quantization and reconstruction is shown in figure 1. The blue dots represent the sampled value y at each fixed discrete-time step. After the blue dot crosses a quantum of 0.05, events are generated as red dots. 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. The right part shows a comparison of estimated $\overline{\boldsymbol{y}}$ and original sampled y signal. Upon the arrival of an event, it updates $\overline{\boldsymbol{y}}$ with the received y value. After that, the estimated $\overline{\boldsymbol{y}}$ value stays constant until the next event. As a result, QSS1 constructs the signal as piece-wise constant.

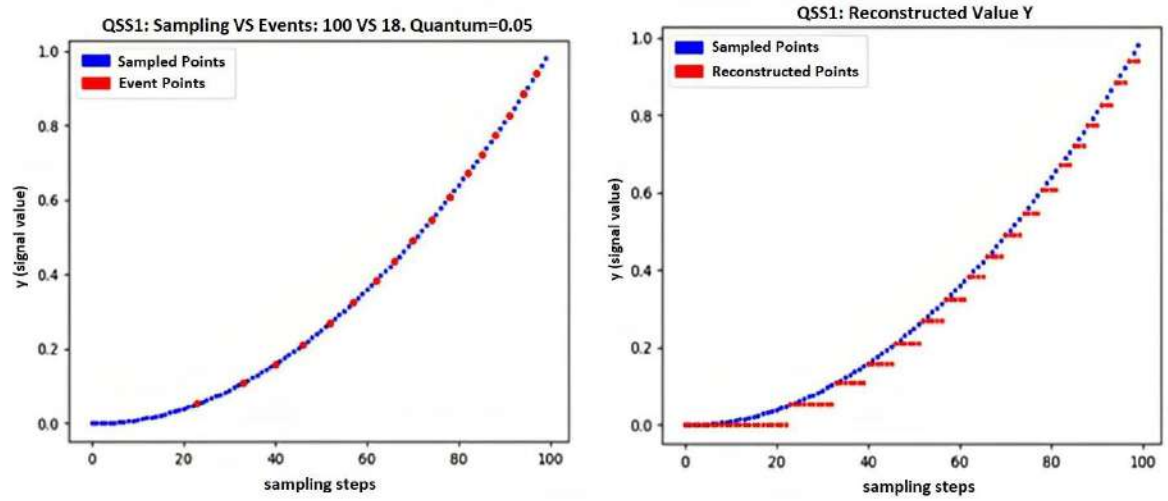

Figure 1: QSS1 algorithm event generation and reconstruction.

In figure 2, using the same representation and operation, QSS2 sends out one more state information y'. This is used to update $\bar{y}$ as piece-wise linear. At the same time, it also knows the $\overline{\boldsymbol{y}}$ since it also knows y'. QSS2 calculates the difference between y and $\overline{\boldsymbol{y}}$ along with current y' at each time step. If the difference is greater than $\boldsymbol{\Delta q}$, then send y and y' as state information.

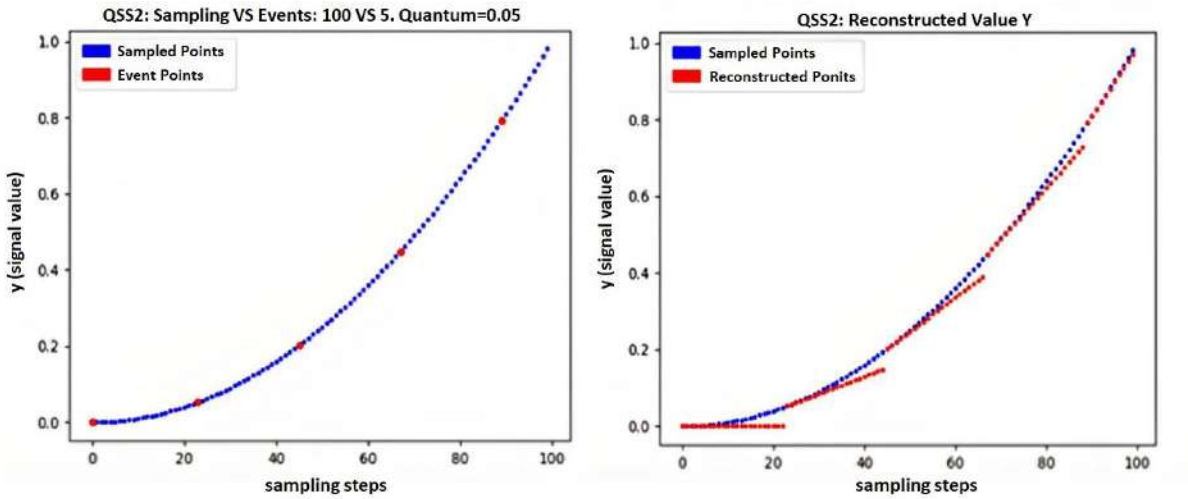

Figure 2: QSS2 algorithm event generation and reconstruction.

Observed from figures 1 and 2, with the 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.

We are interested in investigating these methods applied to the Proportional-Integral-Derivative (PID) control, as this is the most common control algorithm used in industry. It is simple but effective so that it is universally accepted by industrial control. 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 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 proportional, integral, and derivative terms.

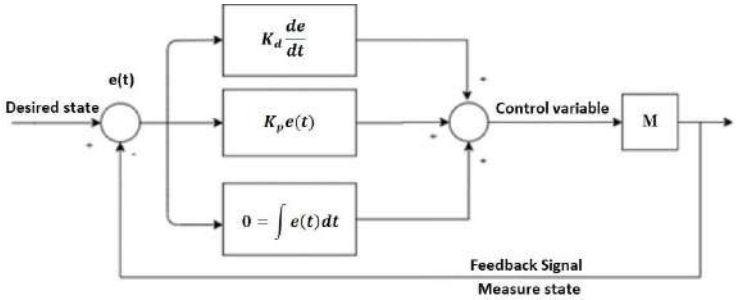

Figure 3: PID controller feedback loop.

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.

Although the PID 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 oscillating. However, if the step is too small then it takes a long time to reach the desired state.

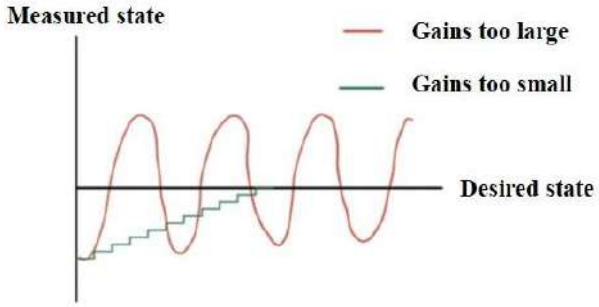

Figure 4: Effect of coefficient (step) of P component.

3 QSS-PID CONTROLLER IN DEVS

We built a QSS-PID controller, which consists of a PID controller atomic model and a QSS quantizer coupled model. The output of the quantizer is the input of the PID controller and the output of the PID controller atomic model is the output of the whole QSS-PID controller that controls the physical target. This QSS-PID controller can be implemented in between of the actuator and the physical target in a CPS.

Figure 5 shows the coupled QSS model which includes a qssSender and qssReceiver. In this figure, Y represents the latest sampling, K represents the slope information and eventLed is sent as an output t for visualizing when an event is generated.

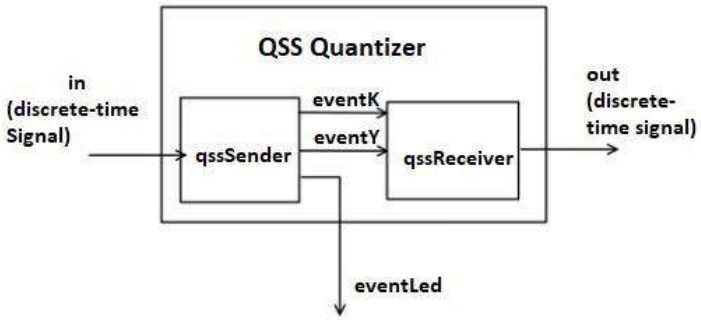

Figure 5: QSS quantizer formed by qssSender and qssReceiver.

The definition of qssSender is provided in figure 6. In qssSender atomic model, the external transition δ_{ext} receives signal sampling and update currentY. Internal transition δ_{int}, keeps track of steps which indicates how many internal transitions has occurred since last event generation and calculate estimatedY using last state information sent to the qssReceiver. It also, calculate the difference between estimatedY and currentY; the difference reflects how much estimatedY on the qssReceiver side has deviated from latest signal sampling. Internal transition δ_{int} should also check if the deviation is crossing a predefined threshold (quantum). If difference is above the threshold then an event is generated. If QSS2 method is chosen so corresponding derivative information is included as state information sent out (eventK). Then prepares and updates the state information to be sent out. After sending the state information, it resets the steps and since new event is generated it toggles the led as well (eventLed). The output function λ send out latest state information if an event flag is set by internal transition. It must be mentioned that the time advancement set inside internal transition must be consistent with that on the qssReceiver.

```
X= {in}
Y= {lastEventY, lastEventK, toggle}
S= {currentY, lastY, dataY, steps, estimated, 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 lastEvenetY to port eventY
        send lastEventK to port eventK
        send toggle to port eventLed
    }
}
```

```
δint (s = {currentY, lastY, deltaY, steps, estimated, 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
        estimated = lastEventY + steps*lastEventK
        if  | estimatedY – currentY | > Threshold
        {
            if  (QSS_MODE == QSS1_MODE)
            {
                lastEventK = 0 //QSS1 always have derivative as zero
            }
            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 synchronization time
    //It can be changed but must be consistent with the internal transition time inside the qssReceiver
    Sigma = TIME ("00:00:00:1")
}
```

Figure 6: Definition of qssSender.

The definition of qssReceiver is similar; the external transition δ_{ext} receives latest update of eventY and eventK. (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. Internal transition δ_{int} keeps track of steps and increase currentY (estimated) by currentK. The output function λ sends out currentY to output port.

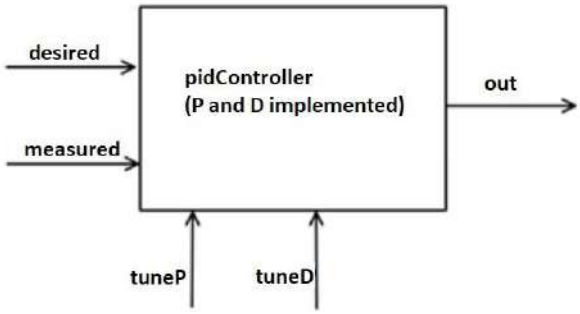

Figure 7: PID controller conceptual model.

As shown in figure 7, the pidController receives the desired value as reference point (and the idea is to have it connected to a QSS quantizer so then the desired value will be a quantized input). The pidController model also receives measured value from the sensor. We adjust the coefficients of P and D components, through the analog input ports tuneP and tuneD. For every measured sensor value comes in, the pidController calculates the correction value and send it to the out port. The D component of the PID controller requires information of the time elapsed between current currentError and previousError since it needs to calculate the derivative. 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 definition of pidcontroller is provided in figure 8.

```
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
```

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

```
λ (s={correctionValue})

{

      send correctionValue to port out

}
```

```
    }
    sigma = sigma - e
}
```

Figure 8: Formal specification of PID controller.

It is important to note that the internal transition should have smaller 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 A LIGHT INTENSITY CONTROLLER

In this section we show the definition of a light intensity controller is proposed here as a CPS case study to validate the performance of the QSS-PID controller. In this system, a potentiometer is attached to the controller while a LED light is attached to the actuator. The potentiometer adjusts a continuous analog signal that indicates the desired brightness of the LED light. The continuous signal should be sent to the actuator of LED light via some communications channel in the original system. Instead of using discrete-time sampling of this signal, QSS is applied to deliver this information. 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.

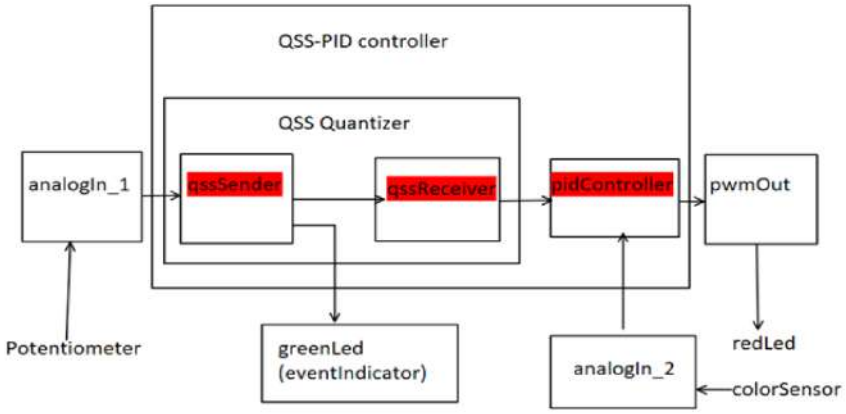

Figure 9: Overview of a QSS-PID based control system.

Figure 9 provides the DEVS model of this light controller using the QSS-PID controller. The QSS-PID controller models were implemented both in simulated mode and in the RT-Cadmium framework (Earle et al. 2020). We used a NUCLEO-H743ZI development board for the deployment of the DCIF. This is an STM32H7 ARM Cortex-M7-based microcontroller in the LQFP144 package. 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. Therefore, if we wish to control LED brightness smoothly, analogIn1 and pwnOut can be divided in order to apply the QSS-PID controller. The analogIn1 atomic model provides sampling of signal at discrete-time step to qssSender. In order to determine the state information, the qssSender quantizes the input signal using predetermined state change criteria. 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 would toggle. 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.

The simulation of qssreceiver reconstructed signals with sinusoidal and exponential input signals sampled with discrete-time steps is shown and compared using both QSS1 and QSS2 algorithms at the qssSender. Since the QSS quantizer consists of qssSender and qssReceiver, it is more efficient to test these two atomic models together as we can easily compare the input to qssSender and output from the qssReceiver. The top model connects the qssSender and qssReceiver. Sampled signal is fed to the qssSender by the input files and the output files are for checking the reconstructed signal by qssReceiver. There are also files representing physical components connected to qssSender and qssReceiver such as potentiometer.

In figures 10 and 11, sinusoidal and exponential signals (with some noise added) sampled with 100 discrete-time steps are fed into the QSS1 and QSS2 quantizer subsequently. Sampled input signals are represented by blue dots and the red dots represent the signal reconstructed by the qssReceiver. It was observed with 100 discrete-time steps sinusoidal input fed into qssSender, a total of 20 events are generated by qssSender which is 80% event reduction. With exponential input signal only 4 events were generated which is 96% reduction.

Simulation results indicates qssReceiver reconstructs the signal with 3 synchronization steps late which is caused by the synchronization mechanism inside the internal transition function. Even though, this synchronization delay is negligible compared to the physical network delay in real-world testing, it should be considered and get resolved for hard real time systems.

As shown in figure 10, with a sinusoidal input, the QSS1 algorithm generates 58 events from 100 discrete-time steps. The number of generated events is still substantially more than the 20 events produced by the QSS2 method, even though the signal reconstructed using the QSS1 algorithm was sufficiently accurate. Additionally, 17 events were generated using exponential input and the QSS1 algorithm, which is a relatively high amount when compared to the 4 events produced by the QSS2 method.

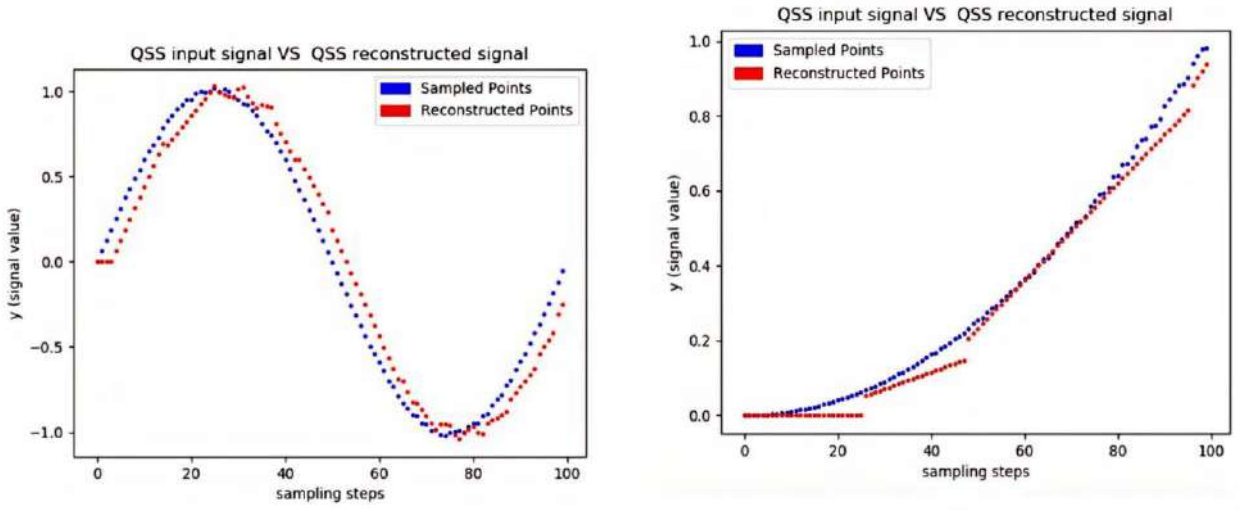

Figure 10: QSS2 simulation with sinusoidal and exponential inputs.

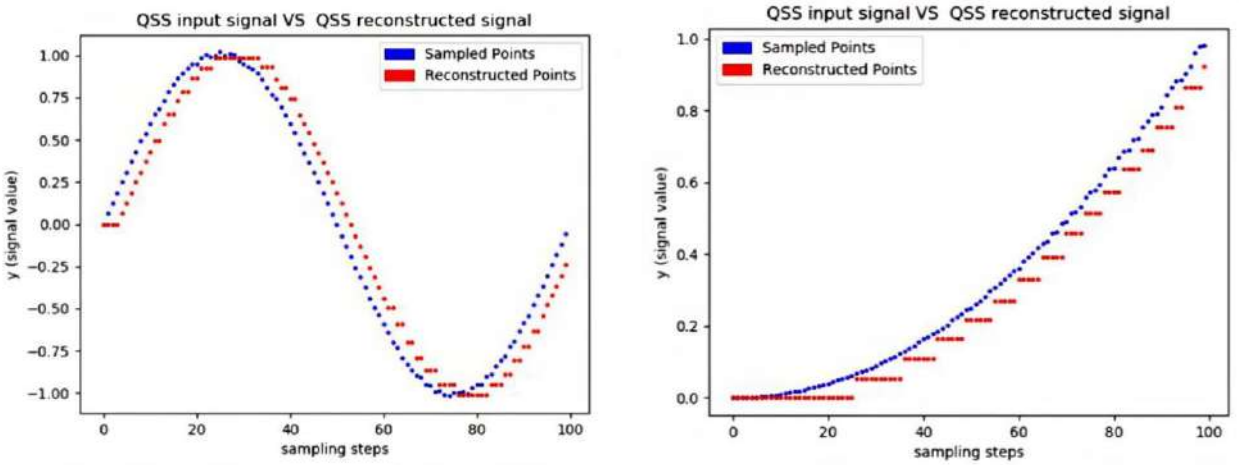

Figure 11: QSS1 simulation with sinusoidal and exponential inputs.

To test the PID controller, we feed the PID controller with an input value as the desired value. Based on the desired value and the measured value (sensor feedback), the correction value was calculated by the controller. The correction values generated by the PID controller were 0.8, 0.6, 0.2, 0. 001 and the measured values corresponding to these correction values were 0.1, 0.3, 0.7, 0.89. There is a 8.9 ms time difference between the time correction value was generated and the time the output value was measured. This delay is negligible compared to the required time for the system to get the output to the desired value which is 40 ms.

For testing the D component, tuneD was set to 1 and tuneP as 0. And the desired value was set to 0.9 again. Correction values were 0.79, 0.58, 0.16 and -0.009. It is observed that the error is decreasing at each time when measured value is updated which leads to negative value for D component. As a result, it can be concluded that the correction value taken by P plus D component should be smaller than a single P component. 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.

Simulation results indicated that P plus D component works as expected. However, PID controller should be tested with more test cases especially with the P,D parameter tuning process. In the performed simulation explained above, measured value was manually generated due to limited resources, and 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.

Results from the simulation show that QSS1 produced more events than QSS2 when using the same number of sample steps. Additionally, the QSS2 quantizer is a better quantizer in terms of signal reconstruction due to the piece-wise linear reconstruction it provides, which better models the original signal. On target testing for the QSS2 algorithm was used because the simulation results indicated a better outcome when employing the QSS2 algorithm. The top model of the QSS-PID controller was built according to the PID and QSS quantizer atomic models. The executable was flashed into Nucleo board for on-target testing afterwards. Potentiometer rotation is used as the input to adjust the brightness of the red LED light. Green LED is used to capture the events generated by the qssSender (one toggle of green LED indicates one event generated).

It is observed that, over 16 seconds, QSS2 algorithm has achieved smooth brightness control of the red LED with only 24 events generated (green LED flashed 12 times). Figure 12 shows three important hardware components used in the experiment; green LED for events generation, red LED to be controlled and potentiometer for adjusting input signals.

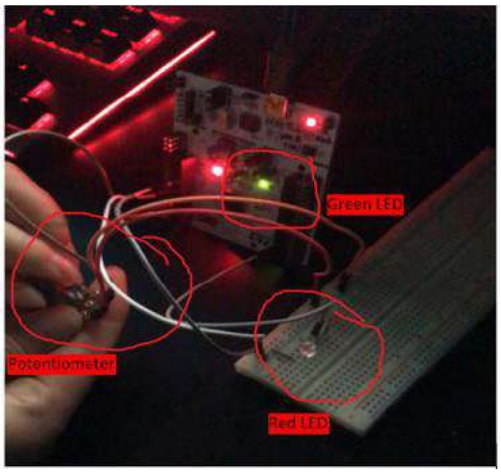

Figure 12: Highlight of physical components for visualizing QSS2 algorithm.

5 CONCLUSION

We have implemented and tested QSS1, QSS2 and PID atomic modes inside RT-DEVS framework. QSS model consists of 2 atomic models which are qssSender and qssReceiver. The simulation results show 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; for instance, the delay can be reduced and integration part (I component) can be considered and investigated. DEVS approach of the QSS3 proposed in Kofman (2006), can be applied into the RT-DEVS framework. This formal specification can then be used to update the qssSender and qssReceivers in the proposed QSS-PID controller. Furthermore, the results can be investigated and compared with QSS1 and QSS2 results.
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

Boi-Ukeme, J., C. Ruiz-Martin, G. Wainer. 2020. “Real-Time Fault Detection and Diagnosis of CPS Faults in DEVS,” *Intl. Conf. on Dependability in Sensor, Cloud and Big Data Systems (DependSys)*. pp. 57-64.

Boukerche, A., A. Shadid and M. Zhang. 2007. "A Formal Approach to RT-RTI Design Using Real Time DEVS". *International Workshop on Haptic, Audio and Visual Environments and Games*, pp. 84-89.

Cho B. M., S. Jang, and K. J. Park. 2020. “Channel-aware congestion control in vehicular cyber-physical systems”. *IEEE Access*, vol. 8, pp. 73193–73203.

Cho, S. M., and T. G. Kim. 2001. “Analysis of feasibility for Real time simulation of rt-DEVS Models” *2001 IEEE International Conference on Systems, Man and Cybernetics.* vol. 5, pp. 3069-3074.

Earle, E., K. Bjornson, C. Ruiz-Martin and G. Wainer. 2020. "Development of A Real-Time Devs Kernel: RT-Cadmium," *2020 Spring Simulation Conference (SpringSim)*, pp. 1-12.

Fernández, J., and E. Kofman. 2014. “A stand-alone quantized state system solver for continuous system simulation,” *SIMULATION*, vol. 90, no. 7, pp. 782–799.

Hong, J. S., H. Song, T. G. Kim, K. H. Park. 1997. “A Real-Time Discrete Event System Specification Formalism for Seamless RT Software Development”. *Discrete Event Dynamic Systems* **7**, pp. 355–375.

Hu, X., B. P. Zeigler and J. Couretas. 2001. "DEVS-on-a-chip: implementing DEVS in real-time Java on a tiny Internet interface for scalable factory automation". *2001 IEEE International Conference on Systems, Man and Cybernetics,* vol.5, pp. 3051-3056.

Kofman, E. 2002. "A second-order approximation for DEVS simulation of Continuous Systems". *SIMULATION*, vol. 78, no. 2, pp. 76–89.

Kofman, E. 2006. "A third order discrete event method for continuous system simulation", *Latin American Applied Research,* vol. 36, no. 2.

Lee, E. 2015. "The Past, Present and Future of Cyber-Physical Systems: A Focus on Models". *Sensors*, vol. 15, no. 3, pp. 4837–4869.

Niyonkuru, D., and G. Wainer. 2016. "A kernel for embedded systems development and simulation using the boost library". *TMS/DEVS Symposium on Theory of Modeling & Simulation*, pp. 1-8.

Sarjoughian, H. S., S. Gholami, T. Jackson. 2013 *"*Interacting real-time simulation models and reactive computational-physical systems*",2013 Winter Simulation Conference (WSC)*, pp. 1120-1131.

Wainer, G., R. Goldstein, A. Khan. 2018. "Introduction to the DEVS formalism and its application for modeling and simulating CPS". *2018 Winter Simulation Conference*, pp. 177-191.

Zeigler B. P., and J. Kim. 1993. "Extending the DEVS-scheme knowledge-based simulation environment for real-time event-based control," *IEEE Trans on Robotics and Automation*, 9(3), 351–356.

Zhang, X., G. Li, et al. 2018. "Design and Implementation of Real-Time DEVS Event Scheduling Algorithm". *International Conf. on Mechanical, Control and Computer Engineering*, pp. 163-168.

AUTHOR BIOGRAGHIES

MAHYA SHAHMOHAMMADIMEHRJARDI is a Ph.D. student in System and Computer Engineering at Carleton University, Ottawa, ON, Canada. Her research interests include modeling real time embedded systems, biomedical image processing and noninvasive sensors. She received her M.Sc. from California State University, Northridge in Electrical Engineering in 2020. Contact her at mahyashahmohammadim@carleton.ca.

GABRIEL A. WAINER is a full-time Professor in the Systems and Computer Engineering Department at Carleton University, Ottawa, ON, Canada. He is the head of the Advanced Real-Time Simulation lab, located at Carleton University's Centre for advanced Simulation and Visualization (V-Sim). He is known for his research in discrete-event simulation, in the Cell-DEVS specification, a variant of Discrete Event System Specification (DEVS) and real time systems. Contact him at gwainer@sce.carleton.ca.

MENGYAO WU received the B.Eng. degree from Carleton University in 2020, where he is currently pursuing the Ph.D. degree in Electrical and Computer Engineering. His research interests include artificial intelligence and networking for connected/autonomous vehicles. Contact him at mengyaowu@carleton.ca.

XINRUI ZHANG is a PhD student in Electrical and Computer Engineering at Carleton University, Ottawa, ON, Canada. Her research interests include secure machine learning operations (SecMLOps), security patterns, and security engineering for machine learning-based systems. Zhang received her B.Eng. from Carleton University in 2020. Contact her at xinrui.zhang@carleton.ca.