Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

HANDLING ASYNCHRONOUS INPUTS IN DEVS BASED REAL-TIME KERNELS

Sasisekhar Govind[1] and Gabriel Wainer[1]

[1]Dept. of Systems and Computer Eng., Carleton University, Ottawa, ON, CANADA

ABSTRACT

Real-time systems are complex to design and implement. Various modelling and simulation techniques are employed to make this task more structured and efficient. However, there is often a disconnect between modeling for simulation and development for deployment. In this paper we discuss a technique to bridge this gap between simulation and deployment, specifically dealing with a framework to handle asynchronous inputs into a system developed using the Discrete Event System Specification. Further, this paper presents a case study that demonstrates the effectiveness of the framework, and the congruence between simulation and deployment of a real-time system is determined.

1 INTRODUCTION

Real-Time (RT) Systems are essential across various sectors such as aerospace, healthcare, and automotive industries, where precise timing and quick to environmental stimuli are crucial. These systems ensure that operations within critical infrastructures run seamlessly and efficiently, handling tasks from monitoring heart rates in medical devices to controlling flight systems in aircraft (Kopetz et al. 2022).

In this research we explore the complexities associated with RT Systems, which are defined by stringent timing constraints and their need to promptly address external events. Developing these systems involves significant challenges, as developers are required to efficiently manage limited resources while ensuring the system operates stably and without errors. To mitigate these challenges, Modeling and Simulation (M&S) techniques are employed as critical tools in formalizing and structuring the development process. Simulation enables developers to attain a comprehensive understanding of the system's behavior before deployment, thereby significantly reducing the likelihood of operational failures. Also, it provides a means to monitor and optimize resource consumption, which is vital for systems constrained by limited capacities. Such an M&S driven engineering methodology is proposed in (Gianni et al. 2014).

One effective way to model RT Systems is by representing them within a finite set of states governed by specific rules for transitioning between these states. A robust methodology that accurately captures these states and transitions is essential for effectively modeling RT Systems. The Discrete Event System Specification (DEVS) formalism is particularly well-suited for this task. DEVS offers a structured approach to model the event-driven systems that characterize many RT Systems, enabling both the design and direct deployment of models (Moallemi et al. 2013).

Further, the application of DEVS goes beyond mere system design; one can also directly deploy models written in DEVS. Various RT Kernels like RT-Cadmium have been developed specifically for executing DEVS models. The kernel, based on the Abstract simulator, orchestrates the execution and simulation of the DEVS models (Wainer et al. 2019).

However, integrating RT Kernels creates significant overheads due to the additional abstraction layer they employ. The extra kernel layer not only consumes a portion of expensive memory but can also lead to timing penalties when external inputs traverse through these abstracted layers. While recent advancements have led to more lightweight and robust kernels (Govind et al. 2023), the efficient management of asynchronous inputs remains a formidable challenge in RT Systems.

The primary goal of this paper is to review existing methodologies in the domain and propose an improved framework for handling interrupts in RT Systems. It aims to improve the integration of interrupts, ensuring they do not disrupt the standard execution flow of the Abstract Simulator.

The paper introduces a novel *Interrupt Component* designed to seamlessly integrate real-world signals into the DEVS modeling framework. This component acts as a plugin to the Abstract Simulator, enabling the transformation of physical signals into a format compatible with the DEVS domain. This integration bridges the gap between real-time system execution and simulation analysis, offering a novel approach to manage the dynamic aspects of RT Systems more effectively.

This paper is divided into 5 sections excluding the Appendix, Biography and Referenced. The first section introduced you to the goals of this paper. Section 2 provides insight on topics required to understand the content of the paper. Section 2 also highlights the previous work in this field. Section 3 goes into the methodology and implementation of the interrupt logic. Section 4 provides a case study and section 5 concludes the paper and provides possible future extensions of the work presented in this paper.

2 RELATED WORKS

2.1 Discrete Event Systems Specification

Discrete Event Systems Specification (DEVS) is a modelling formalism devised for analyzing the performance of continuous time, discrete event, dynamic system (Ziegler et al. 2000). DEVS is composed of two variants of models, Atomic and Coupled. The Atomic model presents the base behavior of a small part of the entire system while a Coupled model composed of other Atomic and/or coupled models that describe the system behavior at a higher level. A description of the classic DEVS models can be found in the Appendix.

There are multiple extensions of the DEVS specification. This research is concerned with the Parallel-DEVS (P-DEVS) formalism. The P-DEVS formalism (Chow et al. 1994) aims to parallelize the execution of DEVS models. The P-DEVS formalism was developed to ensure that parallelism is achieved, while ensuring that collisions are handled, and that closure and hierarchical consistency is maintained. The P-DEVS formalism forgoes the Select function of a Coupled model and allows the modeler to describe behavior upon collision by introducing the Confluent Transition (δ_{con}). δ_{con} allows the modeler to define the model‘s execution behavior when the internal transition function and the external transition function occur at the same time. The complete formal specification of the P-DEVS formalism can be found in the Appendix.

2.2 Abstract Simulator

The Abstract simulator (Chow, Ziegler, and Kim 1994) simulation engine was developed to demonstrate the soundness of the P-DEVS formalism. The abstract simulator is specialized into two different simulation engines, the *simulator,* and the *coordinator*. The *simulator* handles the simulation of atomic models, and the *coordinator* handles the simulation of coupled models. There is always a *root coordinator* that acts as the ‘master’, orchestrating the entire simulation.

The simulation engines execute the appropriate transition function in the appropriate models by means of message passing. The simulation engines use five messages, (@, t), (*, t) and (done, t) for synchronization and (y, t) and (q, t) for data transmission. In the interest of brevity, this is a grossly simplified overview of the abstract simulator. The *simulator* simulation engine processes the $\lambda(s)$ function of the associated atomic model when it receives the (@, t) message. Once the output is generated, the *simulator* transmits the output through (y, t) and synchronizes using the (done, t) message. Upon receiving the (q, t) message, the *simulator* adds the input data to the input bag of the atomic model, and returns the (done, t) message for synchronization. Upon receiving the transition message (*, t), the *simulator* executes the $\delta_{int}(s)$ if the atomic model bag contains no inputs or executes the associated $\delta_{ext}(s, e, bag)$ if the bag is not empty. If the $\delta_{int}(s)$ and $\delta_{ext}(s, e, bag)$ collide, $\delta_{con}(s, bag)$ of the associated atomic model is executed to

resolve the conflict. Once the transition is complete, the (done, t_N) message is returned where t_N is the time advance (sometimes referred to as σ) of the atomic model.

The *coordinator* simulation engine is associated with the coupled model and uses the same five messages for synchronization and message passing. The *coordinator* is responsible for ensuring synchronization between the parent *coordinator* (which could be the *root coordinator*) and all its child *simulators* and *coordinators*. Upon receiving a (y, t) from a child *simulator*, converts it into an (q, t) message and transmits it to the receivers (could be one of its children or the parent *coordinator*). Upon receiving the (@, t) and (*, t) from the parent, the *coordinator* distributes it to the appropriate children. And, when the *coordinator* receives the (done, t_N) messages from its children, the *coordinator* chooses the smallest t_N amongst all the t_Ns it receives and transmits a (done, t_N) to its parent.

2.3 Cadmium Modelling Framework

Cadmium is a tool for DEVS modeling and simulation. It is a header only C++ library that implements the Abstract Simulator. Developed in the Advanced Real-time Simulations Laboratory, Cadmium supports the simulation of classic DEVS, P-DEVS, Cell-DEVS, Asymmetric Cell-DEVS (Cardenas et al. 2022) and the support to implement DEVS models on an embedded platform.

Efforts of students in the Lab (Earle et al. 2020) have helped improve Cadmium as a real time kernel. Further advancements allowed Cadmium to be implemented on a wide range of embedded platforms (Govind et al. 2023) making it a suitable simulator for demonstrating the implementation of the interrupt handling framework.

Recent developments in this field, aligning with our current work, are demonstrated in the paper (Sebastian et al. 2024). The authors have implemented a similar framework within their simulation environment, highlighting the relevance of the approach presented in this paper.

3 METHODOLOGY

As discussed earlier, the PDEVS abstract simulator offers a rigorous framework for event scheduling and inter-model message passing. The algorithm ensures that the appropriate (*imminent*) events are scheduled according to the *time advance* parameter of each model and passes the appropriate messages to trigger the δ_{ext} of the corresponding models within the simulation environment. This behavior is pivotal in the development of an RT execution engine capable of executing DEVS model on an embedded system.

The authors in (Earle et al. 2020) provide a detailed analysis of the implementation of a DEVS-RT kernel that adheres to the DEVS formalism based on the Abstract Simulation algorithm. The authors provide an implementation methodology for integrating the RT execution kernel into the Cadmium simulator. The paper provides meticulous details on the design requirements of the RT kernel and the reasoning behind each of the design choices. Among the various critical elements of the implementation, the RT Clock is deemed one of the most fundamental blocks as it facilitates the synchronization of simulation and real time.

Figure 1 shows the fundamental RT Clock algorithm. The algorithm collects outputs and advances the simulation. Once the simulation has been advanced and the simulator is aware of the minimum time to the next event, the RT Clock puts the device to sleep till the time of the next event. If the simulation has ended, the simulator exits.

This basic implementation of the RT Clock provides the foundation for the work showed by our work. In our work, we move to the newer Cadmium version, Cadmium V2 (Cardenas et al. 2022) which uses the same abstract simulation algorithm to simulate and execute models. Further, the ESP32 embedded platform (mentioned in previous section) was chosen as the basis of implementation due to its wide support and use in the industry.

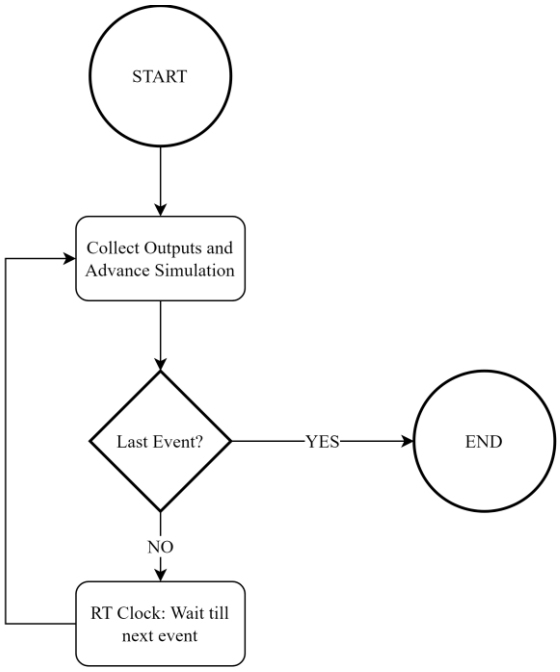

Figure 1: RT clock.

In CadmiumV2, the implementation of the RT Clock leverages the C++ standard library *std::chrono* for its time-related functions. However, it has been observed that the use of *std::chrono::steady_clock* introduces anomalies, resulting in delays during execution that are deemed inadequate. To address these shortcomings, we proposed a solution within the context of RT-Cadmium, involving the adaptation of the RT Clock method to the native platform's capabilities (Govind et al. 2023). In line with this approach, and to attain enhanced performance, we integrated the RT Clock using a clock library specific to the ESP32.

The precise details of the RT Clock's implementation on the ESP32 are documented, available in the Appendix. At its core, the RT Clock class is designed around a *wait_until(timeNext)* method. This method enables the microcontroller to enter an idle/sleep state until the scheduled time for the next event arrives. This method serves as a critical component in the framework of the CadmiumV2 execution engine, particularly in facilitating the management of asynchronous inputs and interrupts. This implementation strategy advances the simulation engine's capability to achieve real-time performance.

One challenge is ensuring that the implementation adheres to the DEVS modelling paradigm (Earle et al. 2020). The integrity of the models should be preserved in such a manner that the same model employed for simulation purposes functions identically when deployed on the microcontroller. This approach highlights the principle that modelers should not be compelled to adapt or 'hack' their models to bridge the gap between simulation and real-time execution, thereby facilitating a straightforward and efficient transition from conceptual modeling to practical application.

To do so, we designed an *Asynchronous Event Observer* to monitor system interrupts. Upon detecting an interrupt, the observer triggers an external transition within the system. While effective, this method necessitates modifications to the Abstract Simulation Algorithm. Our objective is to develop a strategy that accommodates interrupts without necessitating alterations to the abstract simulator, thereby preserving its integrity.

To achieve this, we have defined an *Interrupt Component* (*IC*). The *IC* is defined as follows:

$$IC = \langle X^b, Y^b, \{T_{i,j}\} \rangle$$

Where,

- X^b : Input bag.

- Y^b : Output Bag.
- $T_{i,j}$: i-to-j output transformation function.

When an interrupt input arrives at an input X_i, that has to be routed to output Y_j; the $T_{i,j}$ function is called to transform the input data (not a DEVS message) into an output format (is a DEVS message). Once Y_j has been populated, the *IC* transmits the (y, t) and (done, t) messages to the *root coordinator* of the model being executed. Essentially, one could think of the *IC* as a domain transformation function that transforms data/ signals from the real-world 'domain' to the DEVS 'domain', much like how the *Fourier* transform converts signals from the time domain into the frequency domain. Further, extending on the analogy, $T_{i,j}$ acts like the transformation kernel $e^{-j\omega t}$ that contains components of both domains and helps in the actual transformation of the domains. An overview of a practical implementation is given below.

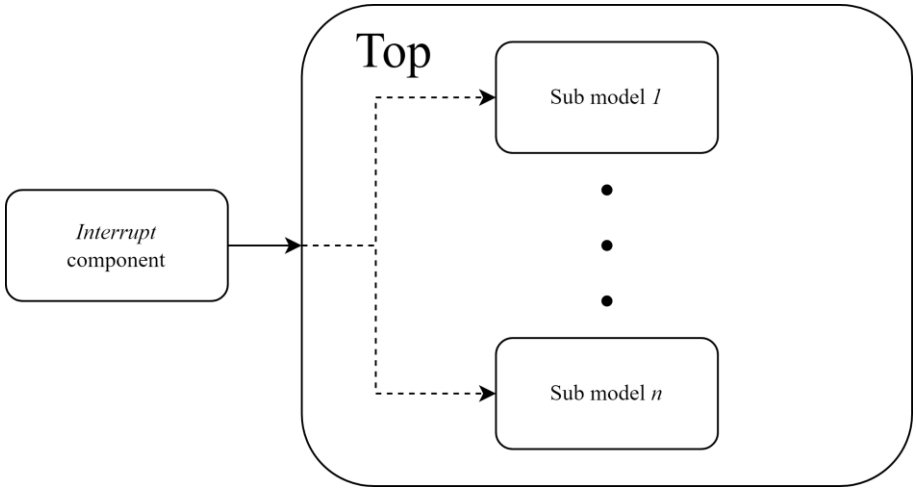

Figure 2: Interrupt Component (IC).

Figure 2 describes an overview of this idea. The figure shows the *IC* connected to the Top coupled model. The top coupled model has *n* Sub models, either atomic or coupled. The *IC* receives interrupts from the external environment and propagates the data to the Top coupled model (through (y, t) messages). The execution of the Top coupled model is handled by the *root coordinator.* And because of the hierarchical nature of DEVS, the *root coordinator* (and all *coordinators*) is not concerned with the execution and scheduling of its parents; it only schedules the execution of its children. However, the *coordinator* responds to (@, t), (*, t), (y, t), (q, t) and (done, t) messages from its parents. This behavior of the abstract simulator is used to implement the *IC* as shown in Figure 2. As far as the *root coordinator* is concerned, it receives the (y, t) messages from a 'parent' *coordinator* and handles it per the Abstract Simulation algorithm. Hence, the soundness of the P-DEVS formalism as we follow the original algorithm.

Since the *IC* has to detect interrupts and send (y, t) messages at any time (maybe non-deterministic), it was decided, that the RT Clock class (specifically the *wait_until()* method) is the optimal location for its implementation. When an interrupt arrives, the *IC* injects the input (y, t) into the *root coordinator*. In the definition of the Top coupled model, an 'in' port is defined as an input port (in $\in X$) and the appropriate *influencees* are defined in the set *EIC*. The *root coordinator* receives the (y, t) messages and routes it according to the definitions in *EIC.*

This generic approach allows any modeler to implement interrupts in any simulator that follows the abstract simulation algorithm. In our work, we have implemented the interrupt handler in the Cadmium V2 simulator (Figure 3).

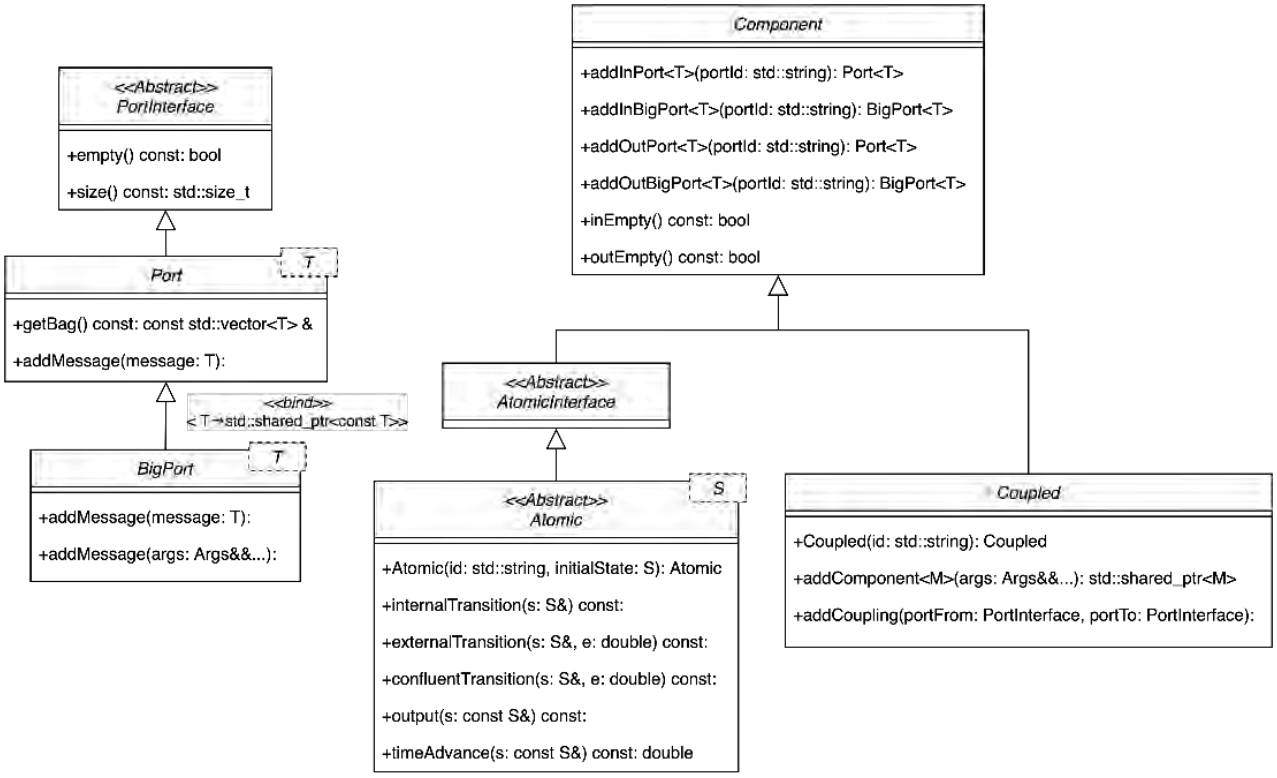

Figure 3: UML class diagram of the Cadmium V2 simulator (Cardenas et. al. 2022).

The figure shows the classes architectures to be followed to create atomic and coupled models to be simulated in Cadmium V2. The figure shows seven classes, *PortInterface, Port, BigPort, Component, AtomicInterface, Atomic* and *Coupled.* The classes with *Interface* in their names are the classes the simulator uses to interact with the atomic, coupled and ports defined by the user.

As can be observed, the atomic and coupled classes inherit traits from the Component class. Hence, the *IC* can be made from the Component class of the simulator. Hence, the *wait_until()* method of the RT Clock class is implemented to create an *IC* with an output port. A pseudo code for the implementation of the *wait_until()* method is show below:

```
function wait_until(timeNext) {
   while(time.now() < timeNext){
      if(Handler.interrupt_recvd()){ //Handler is an instance of IC in RT Clock

         top->propagate(Handler.T());
         break;
      }
   }
   return minimum(timeNext, time.now());
}
```

In the *wait_unti()* method, while waiting for the time specified by the *timeNext* input parameter, the *Handler.interrupt_recvd()* function returns a Boolean 'true'. The transformation function of *Handler* (which is an instance of *IC*) is called, and the returned data is propagated to the input port of the Top coupled model. The *wait_until()* method returns the current time to set the virtual time last of the simulator. If the system has not been interrupted, current time is the same as *timeNext* and hence that is returned, if however, *wait_until()* returns early due to an interrupt, the current time is to be set as the virtual time last of the

simulator. The detailed implementation of the interrupt mechanism and the *IC* class can be found in the Appendix.

When modelling, the modeler ties the input port of the top coupled model to the input port of the model that expects input from the external environment. When simulating, this port can be connected to a generator to simulate interrupt inputs and when executing, the *IC* comes into play. An example scenario with a generic Top coupled model is shown in Figure 4.

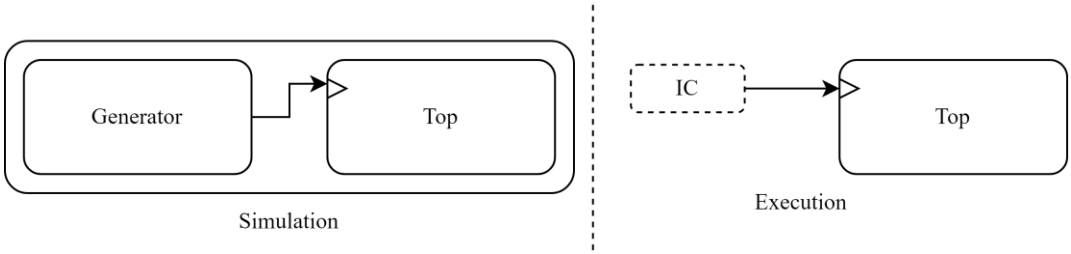

Figure 4: Simulation and execution of the Top model.

Figure 4 shows the simulation and execution of the top coupled model. On the left-hand side, the simulation environment is shown with the Generator, acting as a stub, generating test inputs for the Top coupled model. On the right-hand side, the execution environment is shown with the Generator being replaced by the *IC*. The Generator can be modeled with a Random Variable that mimics the arrival rate of the modeler's actual input.

Assuming that *f(t, λ)* is the Probability Distribution Function with inter-arrival time *t* and arrival rate λ, and *F(t, λ)* is the Cumulative Distribution Function of *f(t, λ)*, you can model the Generator as follows:

$$\text{Generator} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

Where,

- $X \in \{\phi\}$
- $Y \in X_{in}(\text{Top})$
- $S = \{\text{data}, R\}$
- $\delta_{int}(s) = \{$ data = *sample data of type* $X_{in}(Top)$, R = *uniform random value* $\}$
- $\delta_{ext}(s, e, x) = \phi$
- $\lambda(s) = \{\text{send data}\}$
- $ta = F^{-1}(R)$

If we assume that the input is modelled with a *Poisson Distribution*, the inter arrival rate would be *Exponentially Distributed*. In this scenario, $F(t, \lambda) = 1 - e^{-t\lambda}$ and,

$$ta = -\lambda^{-1} ln(R) \quad (1)$$

Where R is a uniformly distributed random number.

4 CASE STUDY: VIDEO SURVEILLANCE SYSTEM

This section discusses how to use the methods presented above with a simple case study of a surveillance system. The system comprises two separate computing units: an ESP32 microcontroller dedicated to motion detection and alert, and a Raspberry Pi single-board computer responsible for video capture and transmission.

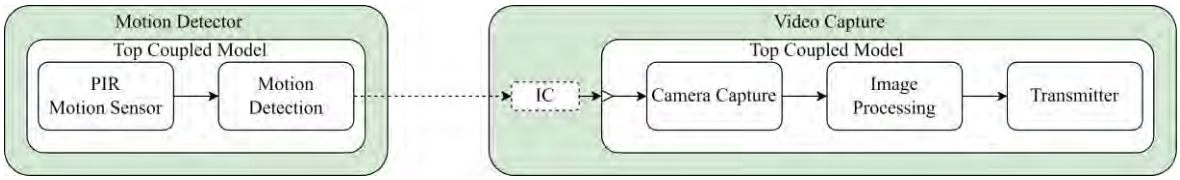

Figure 5: Surveillance system.

Figure 5 shows a block diagram of the surveillance system. The box labelled Motion Detector contains a *Top Coupled Model* that encompasses a *Passive Infrared (PIR) Motion Sensor* atomic and a *Motion Detection* atomic. On the right, the box labelled Video Capture contains the *IC* and a *Top Coupled Model* which encompasses the *Camera Capture* atomic, *Image Processing* atomic, and the *Transmitter* atomic. In the ESP32, the Passive Infrared (PIR) Motion Sensor atomic model and the Motion Detection atomic model detect motion and generates a Boolean signal, which is then transmitted (and in this case is connected to the Video Capture model). Subsequently, the Video Capture model, through the *IC* model, processes the incoming signal to activate or deactivate video capture, processing, and transmission via its *Camera Capture* atomic, *Image Processing* atomic, and *Transmitter* atomic models, respectively. Not shown in the block diagram is a receiver that receives the frames and displays a video output.

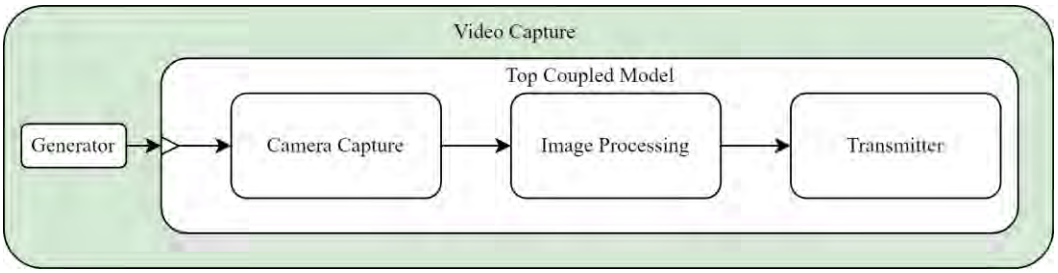

Figure 6: Simulation of asynchronous inputs.

Figure 6 shows the simulation of setup of the Video Capture Model. The *IC* model is replaced by the Generator model, but all other models remain the same. No change is to be made to the model for execution and simulation was modeled using the exponential distribution per Equation $ta = -\lambda^{-1}\, ln(R)$ (1.

Table 1: Simulation logs of the 'Video Capture' model.

Time	**Model ID**	**Model Name**	**Port Name**	**Data**
10.099	3	cam_capture	out	[640 x 480]
10.099	2	im_proc	out	40564
10.099	1	udp_send		Sent: 40564
10.75	*3*	*cam_capture*		*Active: 0*
10.75	*4*	*generator*	*out*	*false*
17.78	*3*	*cam_capture*		*Active: 1*
17.78	*4*	*generator*	*out*	*true*
17.78	3	cam_capture	out	[640 x 480]
17.78	2	im_proc	out	41412
17.78	1	udp_send		Sent: 41412

Table 1 shows a section of the simulation logs generated by the Video Capture model. The logs have five columns, *Time*, *Model ID*, *Model Name*, *Port Name*, *Data*. In the interest of brevity, only the logs of λ(s) and some δ_{ext}(s, e, x) events are shown. The logs show the *Time* of event, *Model ID* and *Model Name* of the model that created the event, the *Port* where the output was generated and the *Data* at the port.

Observing the table, we see that the 10.099th time point a frame is captured, processed and transmitted. When the time point moves towards 10.75s, the Generator produces an output '*false'* which stops the transmission. The transmission restarts when the Generator produces an output *true* at the 17.78th time point.

This is the expected behavior of the system. Once the model was verified, the system was implemented in the embedded platform.

Table 2: Execution logs of the 'Video Capture' unit.

Time	**Model ID**	**Model Name**	**Port Name**	**Data**
1010.033	3	cam_capture	out	[640 x 480]
1010.033	2	im_proc	out	40537
1010.033	1	udp_send		Sent: 40537
1010.066	3	cam_capture	out	[640 x 480]
1010.066	2	im_proc	out	40504
1010.066	1	udp_send		Sent: 40504
1010.066	***3***	***cam_capture***		***Active: 0***
1020.07	***3***	***cam_capture***		***Active: 1***
1020.07	3	cam_capture	out	[640 x 480]
1020.07	2	im_proc	out	40512
1020.07	1	udp_send		Sent: 40512

Table 1 shows a section of the simulation logs generated by the Video Capture model. The logs have five columns, *Time*, *Model ID*, *Model Name*, *Port Name*, *Data*. In the interest of brevity, only the logs of λ(s) and some δ_{ext}(s, e, x) events are shown. The logs show the *Time* of event, *Model ID* and *Model Name* of the model that created the event, the *Port* where the output was generated and the *Data* at the port.

Observing the table, we see that the 10.099th time point a frame is captured, processed and transmitted. When the time point moves towards 10.75s, the Generator produces an output '*false'* which stops the transmission. The transmission restarts when the Generator produces an output *true* at the 17.78th time point.

This is the expected behavior of the system. Once the model was verified, the system was implemented in the embedded platform.

Table 2 shows a small section of the execution logs generated by the Video Capture model. Like Table 1, in the interest of brevity, only the logs of λ(s) and some δ_{ext}(s, e, x) events are shown.

From the table, we can observe that the *1010.033*s time point, the *cam_capture* (corresponding to the Camera Capture atomic in Figure 5) captures an image of resolution 640x480. This image goes into the *im_proc* (corresponding to the Image Processing atomic in Figure 5) processes the image and produces an output of 40537 bytes. These bytes are sent to the *udp_send* (corresponding to the Transmitter in Figure 5) that transmits the processed frame to the server. These steps are followed every time a frame is transmitted and can be seen in the table at the *1010.066*s and *1020.07*s time points. Further, the 33ms difference between *1010.033*s time point and *1010.066*s time point shows that the frame rate of transmission is 1/33ms ~ 30fps.

The two bold entries in the table at the 1010.066th and the 1020.07th time point corresponds to the arrival of a signal from the ESP32. At the *1010.066*th second mark, a deactivate signal was sent that deactivated the camera until the *1020.07*th second mark when an activate signal was sent. This would mean that the ESP32 stopped detecting motion at the 1010.066th time mark and detected motion about 10 seconds later.

Capturing frames from a camera at 30fps is an extremely CPU intensive task especially on the Raspberry Pi 4 with relatively constrained computing resources. For comparison, when polling was implemented to test against the *IC*, it failed to detect the pulses from the ESP32. This portrays the effectiveness of the *IC* at accepting asynchronous inputs.

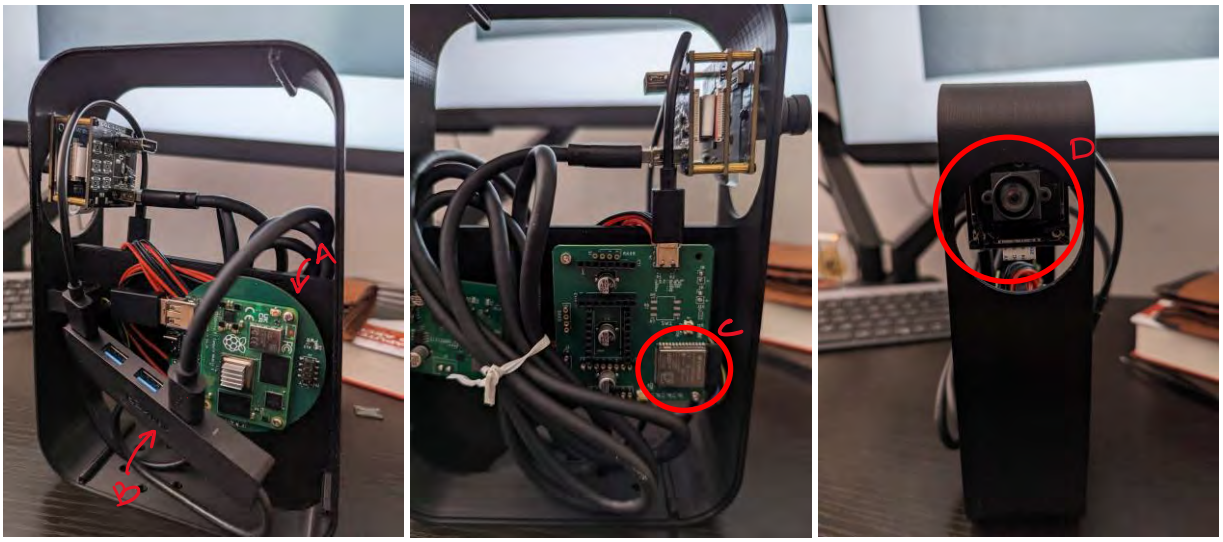

Figure 7: (left to right) a) Side (left) view of the surveillance system b) side (right) view of the surveillance system c) Front view of the surveillance system.

The three pictures in Figure 7 show the real-world model of the surveillance system. Figure 7 a) and b) show the two sides of the model. In Figure 7 a), two parts labelled 'A' and 'B' can be seen. 'A' is the Raspberry Pi Compute Module (CM4) on a custom Printed Circuit Board (PCB) and 'B' is a device (USB hub) that allows one to split a single USB port into multiple USB ports. As can be seen, there are two devices connected to 'B'. Figure 7 b) shows a PCB with a component labelled 'C'. 'C' is the ESP32-WROOM32S3 microcontroller with appropriate circuitry to support a motion sensor and other components. A cable can be seen protruding from that board, that cable connects to 'B'. Figure 7 c) shows the front view of the Surveillance system, where the Camera can be seen. This camera also connects to 'B'.

The Video Capture model runs on 'A'. The Camera Capture atomic receives interrupts from 'C' through the hub 'B'. The Transmitter atomic transmits each video frame to a User Datagram Protocol (UDP) server, where the stream can then be fetched from.

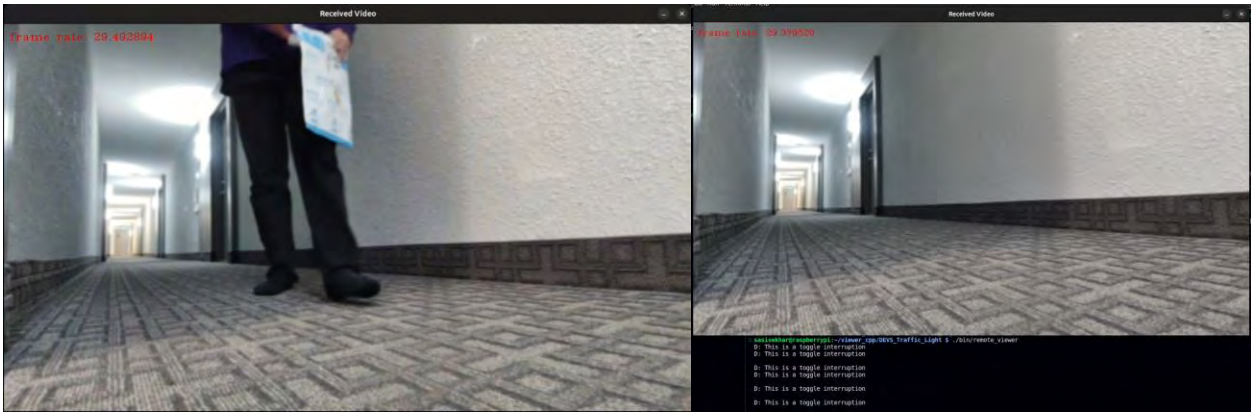

Figure 8: (from left to right) a) activated camera b) deactivated camera.

Figure 8 shows snapshots of the surveillance footage. Figure 8 a) shows a snapshot of the video stram from the camera when active, as can be observed, a courier package delivery person holding a package can be seen in the picture. Figure 8 b) shows a snapshot of the stream and the message logs from the *Camera Capture* atomic model.

As can be observed, Figure 8 a) shows that the camera has been activated at the detection of motion. The frame rate can be observed to be approximately 30 frames per second (fps). Figure 8 b) shows the screen capture along with the message logs from the *Camera Capture* atomic. It can be seen that the toggle message was sent 6 times which implies that the surveillance system captured 3 instances of motion (each pair is activation and deactivation of the camera)

We can observe that, even when the camera is being polled at 30fps (i.e. $\sigma = 0.033$s), the interrupts are being captured every time the motion detector detects motion. This portrays the performance improvement asynchronous input capture brings over polling, in that, even when the CPU is at a high utilization percentage, interrupts enable the scheduler to react to inputs without having to spend CPU clock cycles checking for the state of the input port.

5 CONCLUSION

This paper has presented a detailed exploration and implementation of a framework designed to enhance the handling of asynchronous inputs in RT Systems, utilizing the DEVS and a novel *Interrupt Component* (IC). Through the development and integration of the IC within the DEVS framework, this study has successfully demonstrated a method to bridge the gap between simulation and real-time execution, thereby improving the operational efficiency and reliability of RT Systems.

The proposed framework was rigorously tested through a case study involving a video surveillance system, which highlighted the practical benefits of integrating the IC to handle real-time inputs effectively. The implementation on an ESP32 platform showcased the framework's ability to process inputs dynamically and respond to environmental changes without the need for constant polling, thus reducing CPU load and enhancing system responsiveness. The results from the case study clearly indicate that the asynchronous event handling mechanism can significantly improve the performance of RT Systems, particularly in scenarios requiring high-frequency input monitoring and processing.

Moreover, the adoption of the IC in the DEVS modeling environment has ensured that the models remain true to their theoretical specifications when deployed. This congruence between simulation and deployment underscores the robustness of the DEVS formalism and the effectiveness of the enhanced modeling techniques presented in this paper.

In the future, we plan on employing the IC and integrating it with the DEVS-Inter Atomic Communication (DEVS-IAC) protocol (Govind et al. 2024) to develop truly hard real-time distributed system, thus enabling the protocol to achieve optimal bandwidth. The integration of the IC and DEVS-IAC into existing and future RT Systems could lead to significant improvements in various industrial applications, ensuring that such systems are more adaptable, reliable, and efficient in their operation.

A IMPLEMENTATION DETAILS

The appendix of this paper can be found here. The link takes you to a GitHub Repository (Repo) where a ReadMe file contains all the formal definitions and implementation explanations pointed to in this paper. The Repo also contains the C++ code that implements the *IC* in Cadmium V2.

REFERENCES

Kopetz, H, W. Steiner, Authors. 2022. *Real-time systems: design principles for distributed embedded applications*. [13 ed.]. Springer Nature

Gianni, D., A. D'Ambrogio, A. Tolk, Editors. 2014. *Modeling and simulation-based systems engineering handbook*. New York: CRC

Moallemi, M., G. A. Wainer. 2013. "'Modelling and Simulation-Driven Development of Embedded Real-Time Systems". *Simulation Modelling Practice and Theory* 38:115-131

Wainer, G. A., J. Boi-Ukeme, 2019, "Applying Modelling and Simulation for Development of Embedded Systems." In *SummerSim -SCSC*. July 22nd-24th, Berlin, Germany.

Govind, S., J.S.R. Alex, G. Wainer. 2023. "Adopting the DEVS Kernel 'RT-Cadmium' to the ESP32 Embedded Platform". B. Tech Thesis, VIT University. Last accessed 9th April: arXiv:2304.07961

Gon, K. T., B. P. Zeigler, and H. Praehofer. 2000. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. [2 ed.]. Academic Press

Chow, A.C., B. P. Ziegler. "Parallel DEVS: A Parallel, Hierarchical, Modular Modelling Formalism". In 1994 *Winter Simulation Conference (WSC)*, 716-722, https://doi.org/10.1109/WSC.1994.717419

Chow, A.C., B. P. Ziegler, D. H. Kim. 1994. "Abstract Simulator for the Parallel DEVS Formalism". In *Fifth Annual Conference on AI, and Planning in High Autonomy Systems*. December 7th-9th, Gainesville, Florida, USA, 157-163

Cardenas, R., G. Wainer. 2022. "Asymmetric Cell-DEVS Models with the Cadmium Simulator". *Simulation Modelling Practice and Theory* 121:102649

Earle, B., K. Bjornson, C. Ruiz-Martin, G. Wainer. 2020. "Development of a Real-Time DEVS Kernel: RT-Cadmium". In *Spring Simulation Conference (SpringSim)*. May 19th -21st, Fairfax, VA, USA, 1-12

Sebastian, F. O., R. Cardenas, P. Arroba, J. L. Risco Martin. 2024 "A Novel Real Time DEVS Simulation Architecture with Hardware-in-the-Loop Capabilities". Proceedings of *Annual Simulation Conference (ANNSIM)*. May 20th-23rd, Washington DC, USA

Govind, S. , G. Wainer. 2024 "DEVS Based Robust Communication Protocol for Inter-Simulation Communication in Cadmium". Accepted in *Annual Simulation Conference (ANNSIM)*. May 20th -23rd , Washington DC, USA

AUTHOR BIOGRAPHIES

SASISEKHAR GOVIND is a Ph.D. student at Carleton University, under the supervision of Dr. Gabriel Wainer. He holds a bachelor's degree in Electronics and Communications Engineering from VIT, India. His research interests lie in embedded systems and distributed simulations. His email address is sasisekharmangalamgo@cmail.carleton.ca

GABRIEL WAINER is a Professor at the Department of Systems and Computer Engineering at Carleton University. He received his M.Sc. (1993) from the University of Buenos Aires, Argentina, and his Ph.D. (1998, highest honors) from UBA/ Université Aix-Marseille-III, France. He is a fellow of SCS. His email address is gwainer@sce.carleton.ca