A NOVEL REAL-TIME DEVS SIMULATION ARCHITECTURE WITH HARDWARE-IN-THE-LOOP CAPABILITIES

Óscar Fernández-Sebastián[a], Román Cárdenas[b c], Patricia Arroba[b c], and José L. Risco-Martín[a]

[a]Department of Computer Architecture and Automation, Universidad Complutense de Madrid, Spain
{osfern06, jlrisco}@ucm.es
[b]Department of Electronic Engineering, Universidad Politécnica de Madrid, Spain
{r.cardenas, p.arroba}@upm.es
[c]Center for Computational Simulation, Universidad Politécnica de Madrid, Spain

ABSTRACT

Advanced Modelling and Simulation (M&S) techniques, including Digital Twins (DTs) and Hardware-In-the-Loop (HIL) simulation, are essential for the design and operation of today's complex systems. Despite these advances, there remains a gap between the simulation methods used in design and those used in implementation. This paper introduces a Real-Time (RT) simulation framework that connects models using the Discrete-Event System Specification (DEVS) with external systems, improving implementation and operation. The Python xDEVS tool has been enhanced to introduce a new suite of real-time simulation components and a DEVS-based communication interface, facilitating adaptable communication protocols. The framework demonstrates its capability to integrate both DEVS and non-DEVS systems and establish effective communication through various experiments, ranging from RT simulation of DEVS models to complex multi-component systems. This tool has significant potential in supporting real and incremental system implementations through HIL techniques, making a valuable contribution to complex system design and operation.

Keywords: DEVS, Real-Time, Hardware-In-the-Loop.

1 INTRODUCTION AND RELATED WORK

Modeling and Simulation (M&S) techniques are essential for the design and implementation of complex systems within the emergent domain of Industry 4.0 [1]. These methodologies facilitate representing the behavior of systems under development, enabling the identification of bottlenecks or design flaws, thus reducing associated costs and development duration. M&S is a multidisciplinary field, finding applications across various areas ranging from train traffic management [2] to queuing theory analysis [3] and polymer research [4].

M&S techniques can be classified based on their temporal modeling approach [5]. In Discrete Event Simulations (DESs), events occur chronologically in discrete instants of time and result in a change in system state [6]. Some of the most popular DES approaches are Petri nets [7], Markov chains [8], or the Discrete-Event System Especification (DEVS) formalism [5]. DEVS is notable for its modular and hierarchical specification, which allows the system to be decomposed into small independent modules. Moreover, the mathematical basis of this methodology facilitates the formal verification of the different parts that comprise the system, providing a rigorous definition for discrete-event modeling and simulation. In DEVS, there are two ways to describe how a system behaves: **atomic models**, which represent a system's autonomous behavior as a series of state transitions and responses to outside events, and **coupled models**, which depict a system as the interconnection of other DEVS components, either atomic or coupled.

Most of the DEVS-compliant simulation engines are based on the abstract simulator algorithm [9]. This approach uses independent coordinating simulation engines to synchronize parallel activity along asynchronous processors. This algorithm has two types of simulation engines: **simulators**, which govern the behavior of their respective atomic models, and **coordinators**, which manage message propagation and synchronize their subordinate simulators and coordinators. Both entities adhere to the same interface, simplifying model encapsulation and reinforcing the hierarchical concept of DEVS.

Nowadays, simulating a system within a virtual time environment is largely straightforward. However, with increasing complexity and demands on the system, simulated environments may lack realism. Consequently, alternative simulation techniques such as Digital Twins (DTs) [10] and Hardware-in-the-loop (HIL) [11] have emerged in recent years. DTs refers to the virtual copy that aims to replicate the behavior of a physical entity [10], meanwhile HIL is the introduction of hardware components into the simulation loop rather than testing on pure mathematical models [11]. Following these techniques, the Real-Time (RT) simulation has become more relevant. A RT simulation can be understood as a process that must match the wall-clock time (i.e., if a certain physical process takes two seconds, the same process must take two seconds in the simulation). Combining DTs, HIL, and RT simulations allows for more complex and realistic scenarios. These RT simulations have benefits such as achieving an objective representation, testing dynamic interaction among systems, or making a deeper assessment of safety and risks [12]. Although DEVS is being used for different projects, few of them focus on the RT paradigm due to a lack of standardization or proposed methodologies. An excellent example in the literature details the enhancement of Cadmium, an existing DEVS simulation framework, to incorporate RT simulation [13]. The utility of the adapted tool has been evidenced through several case studies, including a line-following robot and a sophisticated sensor network. This adaptation involved developing an additional real-time clock to align software time management with actual time delays. At the cost of modifying the DEVS models, they support this new modality.

This work aims to provide a new methodology to deal with RT simulation for DEVS. Although there are examples in the literature of RT in DEVS, none of them adhere to the formalism and, as a consequence, the atomic and coupled models, as well as the simulation protocols are changed. On the contrary, our methodology keeps the integrity of pure DEVS models, and it is by processing the external events how the RT is achieved. This results in a more flexible solution to the actual paradigm. This solution has more outlets, as it can be applied to any model already defined without having to modify it. In addition, our methodology has the potential to serve as a basis for HIL and DT simulation techniques. The methodology introduces new components responsible for managing interactions between the simulation environment and the physical world. In addition, the system can detect timing deviations and correct them if they are small enough. Finally, some examples illustrate how this new methodology can be used and applied. From the system under study, the RT simulation is analyzed, assessing how much time differs from the desired one to the simulated one. Later, a complex system integrates the communication between DEVS and non-DEVS models, illustrating the use of HIL techniques. The work is based on the Python version of the xDEVS simulator [14]. Unless otherwise stated, when referring to xDEVS, it implies the context of the Python implementation.

The remainder of this paper is organized as follows. Section 2 and Section 3 provide details about the proposed RT DEVS approach. Section 4 illustrates how the proposed RT simulator can be used to run RT simulations and connect to external entities using different communication techniques. Finally, Section 5 draws the main conclusions and discusses future work.

2 SYSTEM OVERVIEW

In our proposed approach for RT simulation of DEVS models, we introduce three novel components: the RT coordinator, the RT manager, and the Input/Output (I/O) handlers. The **RT coordinator** is based on the abstract simulator algorithm as described by Chow in 1994 [9] and is currently implemented in xDEVS. We have extended its capabilities to efficiently manage external events by enabling interaction between the DEVS model and the RT manager. The **RT manager** is a new component whose primary function is to manage the incoming and outgoing events of the system. It directly interacts with the different handlers, as it re-routes events to and from them. In addition to handling the external events between the coordinator and the handlers, the RT manager provides the RT behavior to the simulation. Finally, the **I/O handlers** are responsible for the communication interfaces of the system. Input handlers inject events into the system, and output handlers process output events to re-route them outside the simulation. We establish an abstract interface to characterize these components, allowing the simulation to communicate with any type of technology or protocol such as Transport Control Protocol (TCP) or Message Queuing Telemetry Transport (MQTT).

Figure 1 illustrates the architecture of the proposed system. We categorize system components based on the temporal domains within which they operate. The DEVS model and the RT coordinator work in a virtual time framework. This is the temporal dimension governed by the simulation's internal logic. On the other hand, the RT manager and the I/O handlers operate according to wall-clock time. The RT manager is responsible for translating both time domains and ensuring that there is no time drift.

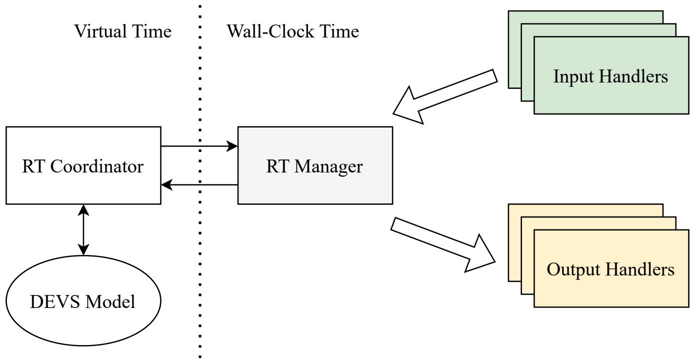

Figure 1: Overview of the proposed approach for RT simulation on DEVS.

Figure 2 presents an overview of an RT simulation. It exemplifies the general behavior between the I/O handlers, the RT manager, and the RT coordinator during an execution. Further details of each component are given in Section 3. The RT coordinator first executes the internal transition function of the model, δ_{int}. This function is in charge of transitioning to the next state of the model if no input is received. The next internal transition function is expected at $t = t_1$. Thus, the coordinator asks its RT manager to wait until that time by calling its $sleep(t_1)$ method. In summary, this method will block the system by waiting for external events. The waiting time interval is colored purple. In the first case, no input events are received. Therefore, the RT manager returns the tuple $(t_1, \emptyset)$ to inform the RT coordinator that we have reached t_1 without any external interruption. Thus, the RT coordinator will execute the output function (λ) and internal transition functions. Next, it will eject, if any, the outgoing events. To do so, the RT manager forwards these events to all the output handlers. Figure 2 represents this process with orange arrows. In the next purple period, the RT manager receives an event from one input handler (green arrow) before the next timeout, t_2. Therefore, the RT manager waits for a window period to aggregate input events. The window period is blue in Figure 2. At $t = t'_2 < t_2$, the time window expires, and the RT manager returns the tuple $(t'_2, input \neq \emptyset)$ to notify the RT coordinator that an external event occurred. Consequently, the RT coordinator only executes the external transition function (δ_{ext}) and ejects the leaving events, if any. Finally, the RT manager receives new input events, but the time window collides with the next expected internal transition, t_3. Therefore, the incoming events that have arrived until that time are returned in the tuple $(t_3, input \neq \emptyset)$ In this case, the RT coordinator executes the output and confluent transition functions and ejects the leaving events.

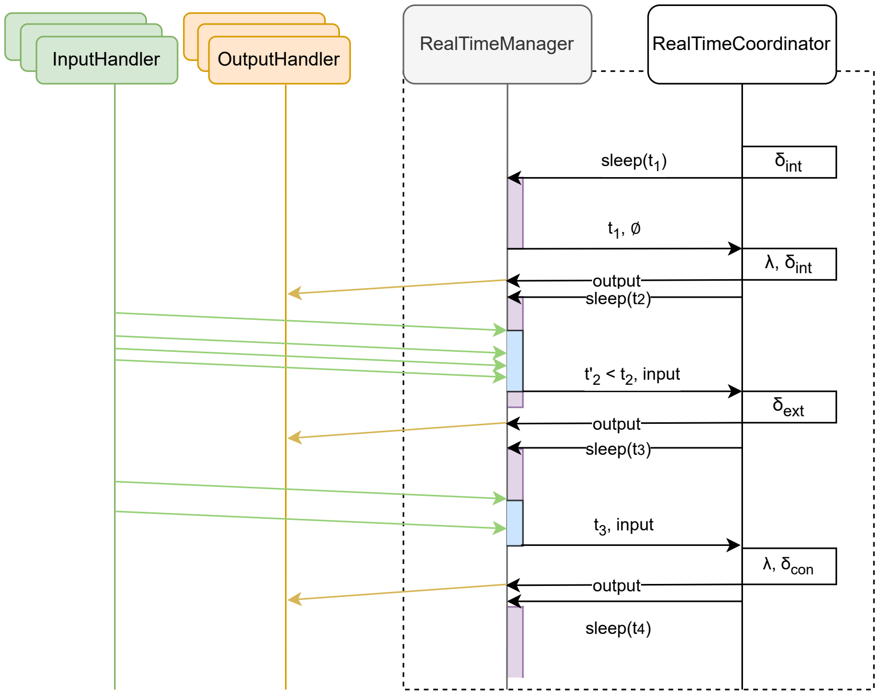

Figure 2: Sequence diagram of the `RealTimeCoordinator.simulate()` method.

3 REAL-TIME COMPONENTS

This section delves deeper into each of the main components that the methodology proposes to achieve a RT simulation and the handling of external events.

3.1 Real-Time Coordinator

The RT coordinator is constructed from the already existing `class Coordinator` in xDEVS. The xDEVS framework implements the abstract simulator algorithm [9]. This algorithm is made up of cooperating simulation engines (simulators and coordinators) that synchronize DEVS models during executions to achieve a simulation implementation. However, we customized the xDEVS coordinator to our use case to allow RT simulation. Consequently, the new `class RealTimeCoordinator` is created. The two core differences are that, first, unlike its parent class, it expects a reference to the RT manager in the constructor function. This allows both components to communicate and exchange events. Second, the new `RealTimeCoordinator.simulate()` method replicates the functionality of `Coordinator.simulate_time()`, but for our RT context.

Listing 1 illustrates a simplified pseudo-code of the new `RealTimeCoordinator.simulate()` method. Broadly speaking, the method will execute the well-known DEVS functions, such as the output function (λ) or the state transition functions, which depend on the received inputs to the model (i.e., δ_{int} for no inputs, δ_{ext} for any input, and δ_{con} if both functions collide). And it will also manage the I/O of external events to the system through the RT manager.

```
def simulate(time_interv: float):
    intialize();
    while clock_time < time_interv:
        t, events = manager.sleep(t_next)
        Inject incoming events
        clock_time = t
        if clock_time == t_next:
            Execute output function
            Manager propagates outputs
        Execute delta function
        Manager ejects outgoing events
        Clean ports
    exit();
```

Listing 1: Pseudo-code of `RealTimeCoordinator.simulate()` method.

Firstly, the parameter `float time_interv` determines the simulation time to be executed in seconds. As soon as the simulation clock time is greater than this value, the simulation is finished. Otherwise, in each iteration, the coordinator asks its RT manager to wait until the next simulation time, `t_next`. During this waiting period, the manager is waiting for input events from external systems. Eventually, the RT manager will return a tuple with the new simulation time (`t`) after waiting and a set of input events (`events`). The set of events (if any) is then injected into the system through the models' ports. However, depending on the state of the input set, we can have different scenarios. If the event set is empty, there has been no external event and no event will be injected into the system. Therefore, `t = t_next` and the imminent transition is due to an internal state transition. Thus, the coordinator must execute the output function before the internal transition function. On the other hand, if `t = t_next` but the set of events is not empty, there has been a state transition collision, so the confluent transition function executes after the output. Finally, if `t < t_next`, the input set is not empty, indicating that an external event happened. As no internal state transition was expected, the output function is not executed before the external state transition function.

To summarize the code shown in the listing, the RT coordinator has a behavior similar to that implemented in the xDEVS framework, where it is responsible for performing the main functions of a DEVS model to accomplish the simulation of a system. However, in the RT context and following our methodology, its algorithm is changed. The most significant change is based on the use of the RT manager, which now allows the injection or ejection of events.

3.2 Real-Time Manager

The RT manager is responsible for interfacing with the input and output handlers to inject input events from and eject output events to external sources, respectively. Additionally, it provides RT behavior to the simulation by waiting for incoming events in the system. To achieve these features, we created the **`class RealTimeManager`**. The main attributes of this class can be divided into two groups. Firstly, we introduce the simulation configuration group:

- `float max_jitter`: maximum delay jitter the system can assume. By default, it is set to **`None`** (i.e., no maximum delay jitter). If the execution of output and state transition functions takes longer than `max_jitter`, the simulation terminates earlier.
- `float time_scale`: this parameter allows the user to scale the simulation time to make it faster or slower. By default, it is set to 1 (i.e., no time scale).

- `float event_window`: time interval to wait after receiving one input event to group subsequent events and avoid multiple external interruptions. By default, it is set to 0 (i.e., events are forwarded immediately without event packing).

Secondly, we present the attributes to keep track of the events and the wall-clock time while simulating:

- `SimpleQueue input_queue`: a Multiple Producer, Single Consumer (MPSC) queue where input handlers push incoming events. The RT manager receives them via this queue.
- `float last_r_time`: the last wall-clock time at which the previous call to the method `RealTimeManager.sleep()` finished.
- `float last_v_time`: the last virtual time at which the previous call to the method `RealTimeManager.sleep()` finished.

The behavior of this class is divided into two stages. The first stage occurs during the `RealTimeManager.initialize()` method, where it sets the initial virtual and wall-clock times and creates and activates all the I/O handlers. Each handler runs on an independent thread in daemon mode (i.e., if the main program exits, handler threads stop abruptly without completing their tasks). The second stage consists of its `RealTimeManager.sleep()` method, which is responsible for implementing the RT feature while waiting for external input events. Figure 3 illustrates a workflow diagram of this method.

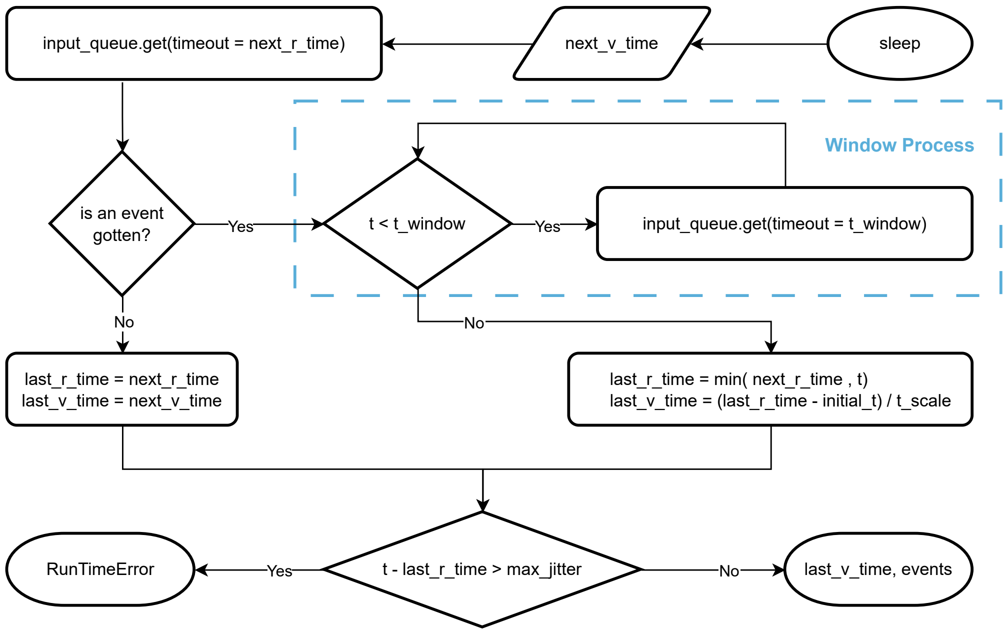
input_queue.get(timeout = next_r_time)
next_v_time
sleep
Window Process
is an event gotten?
Yes
t < t_window
Yes
input_queue.get(timeout = t_window)
No
No
last_r_time = next_r_time
last_v_time = next_v_time
last_r_time = min(next_r_time , t)
last_v_time = (last_r_time - initial_t) / t_scale
RunTimeError
Yes
t - last_r_time > max_jitter
No
last_v_time, events

Figure 3: Workflow diagram of the `RealTimeManager.sleep()` method.

As mentioned in Section 3.1, this method receives the simulation time at which a model internal event is expected to happen, `float next_v_time`. This simulation time is translated into wall-clock time, `float next_r_time`. The value of `next_r_time` is computed as follows:

$$next\_r\_time = last\_r\_time + (next\_v\_time - last\_v\_time) * time\_scale. \tag{1}$$

Next, the RT manager waits for incoming events from the input handlers. By calling the `input_queue.get(next_r_time)`, the whole system is blocked by the queue until an event arrives or until the wall-clock time reaches `next_r_time`. This blocking achieves the RT behavior of the system.

If a message arrives, the RT manager waits for `event_window` additional seconds to group input events before continuing. The next queue deadline, `t_window`, is computed as follows:

$$t\_window = min(t + event\_window, next\_r\_time), \tag{2}$$

where `t` is the wall-clock time of the instant at which the first event was received. Notice that using the `min` function ensures that the next internal event of the DEVS model is not ignored. Figure 3 outlines this window process with a blue dashed box. Regardless of whether events are received, the variables `last_r_time` and `last_v_time` are updated for the next iteration. In the case of no events, the update is straightforward as the finish time is expected to be the next time computed at the beginning. On the other hand, if an event or events occur, the finish time must be less than or equal to `next_r_time`. Thus, `last_r_time` is set to the minimum value between `next_r_time` and the current wall-clock time. In this way, we ensure again that the algorithm does not miss the next model internal event. The `last_v_time` is `last_r_time` converted to the simulation time scale:

$$last\_v\_time = \frac{last\_r\_time - initial\_wall\_clock\_time}{time\_scale}. \tag{3}$$

Finally, the RT manager checks that it did not exceed the maximum jitter allowed, `max_jitter`. If the maximum jitter is exceeded, a `RunTimeError` **`Exception`** is raised, and the system simulation is halted. Otherwise, `last_v_time` and the list of events received from the `input_queue` are returned to the RT coordinator.

Next, we elaborate on the communication between the I/O handlers and the RT manager through queues. Input handlers inject new incoming messages into a shared MPSC queue in the form of events. The manager subsequently receives these events, as detailed in Figure 3. In contrast, the RT manager injects all the leaving events into each of the output handlers' queues, which are then responsible for reacting to the events accordingly.

3.3 Input/Output Handlers

The motivation behind the I/O handlers is to be able to adapt external events as input messages and forward output messages out of the simulated system. It is important to remember that the I/O handlers share a composition relation with the RT manager, meaning that the handlers can not exist without a manager. The interface for injecting or ejecting messages from or to the system must be standardized. To this end, it is required to follow a design methodology that will provide a common framework to implement all the desired manners to inject or eject messages for our system. For that reason, the factory design pattern [15] becomes significant. In this pattern, the common framework of the I/O handlers are an abstract parent class with its abstract methods. These methods, although common to any input or output handler, are implementation specific, giving greater flexibility to the system. Thus, each parent class acts as an Application Programming Interface (API) to communicate with other parts of the system. However, an abstract class can not be instantiated. It requires a sub-class that inherits from it and implements the abstract methods. Therefore, it is the factory the one in charge of creating different I/O handler specifications, hiding the complexity of supporting multiple implementations and enhancing the proposed tool's flexibility, scalability, and maintainability. Finally, the power of the factory pattern ends when adding I/O handlers. The methods `RealTimeManager.add_I/O_handler()` make this task straightforward. By providing a specific identifier, the methods can identify which type of handler the user wants to implement in the system. This makes it easier for the user to configure the scenarios once the handlers have been designed.

3.3.1 Input Handlers

The abstract **class InputHandler** describes how to receive messages from external sources and route them into the system. The Unified Modeling Language (UML) class diagram in Figure 4 illustrates the different input handlers currently available in xDEVS. Namely, the system supports injecting events via Comma Separated Value (CSV) files, TCP sockets, and MQTT messages.

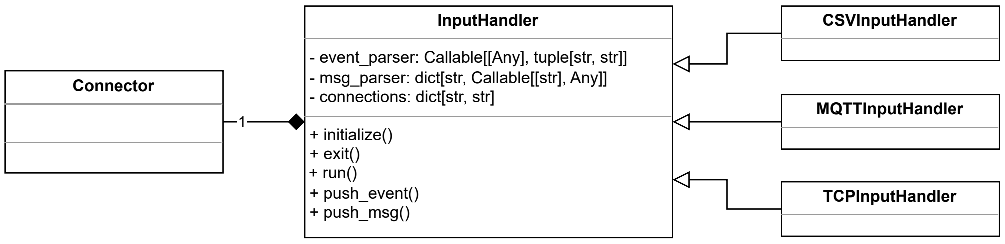

Figure 4: UML diagram of an InputHandlers structure.

The main attributes of the **class InputHandler** are the following:

- `Callable[[Any], tuple[str, str]] event_parser`: a function that transforms incoming events of any type into tuples (port, message). Both elements of the tuple must be strings.
- `dict[str, Callable[[str], Any]] msg_parsers`: a dictionary whose keys are port names, and values are functions that create an object of the corresponding port type from a string.
- `dict[str, str] connections`: a dictionary to support communication with other independent systems. This attribute is used by the input handler's `Connector`, which links ports of independent systems. Keys represent the ports of the external system we want to establish communication with, and values are the ports of the model from which the connection is established.

3.3.2 Output Handlers

The **class OutputHandler** is responsible for transmitting the system's internal messages to external sources. In Figure 5, an UML class diagram illustrates the different output handlers currently available in xDEVS. In line with the input handlers implementations, the system supports saving messages in CSV files, transmitting them via TCP sockets, and publishing them on an MQTT topic.

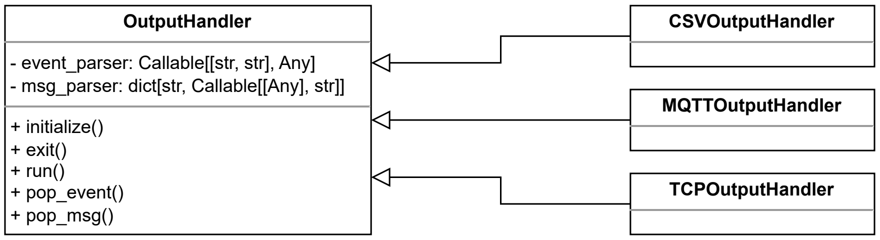

Figure 5: UML class diagram of an`OutputHandler` structure.

The main attributes of the **class OutputHandler** are the following:

- `Callable[[str, str], Any] event_parser`: event parser function. It transforms incoming tuples (port, message) into events. Note that both are represented as strings.

- `dict[str, Callable[[Any], str]] msg_parser`: message parser function. Keys are port names, and values are functions that take any object of the corresponding port type and convert it into a string. If a parser is not defined, the output handler assumes that the port type is `str` and forwards the message as is.

Furthermore, the `OutputHandler` has a `queue.SimpleQueue queue`, where all the desired events to be ejected are put. In contrast with the input handlers, each output handler has its queue.

On the other hand, the output handler's `run()` method is an `@abstractmethod`, so every plugin has to implement it. As the outgoing messages have to follow some kind of syntax according to the protocol implemented, a format parser takes place. This translation between xDEVS language to the implementation specification is achieved with the following methods. First, **`pop_message() -> tuple[str, str]`** waits until it receives an outgoing message in the `queue` and returns two objects that identify the output port and the message as strings. It uses the `msg_parser` attribute to do so. Second, **`pop_event() -> Any`** waits until `pop_message()` returns a port and a message in the string format. Using the attribute `msg_parser`, it parses them into an event for the specific implementation and returns it. This event will be the one ejecting the system.

4 EXPERIMENTAL RESULTS

The baseline system, illustrated in Figure 6, serves as the foundation for the experiments. This use case scenario models the operation of a store cashier. In this system, clients (modeled as "ClientGenerator") join the line at a store's queue (modeled as "StoreQueue"). The queue operates on a First Input First Output (FIFO) principle, matching each client with an available employee, analogous to customers waiting in line to check out their purchases. Each employee (modeled as "Employee") signals their availability by indicating whether they are free or occupied following an interaction with a client, thereby emulating the process of checking out purchases.

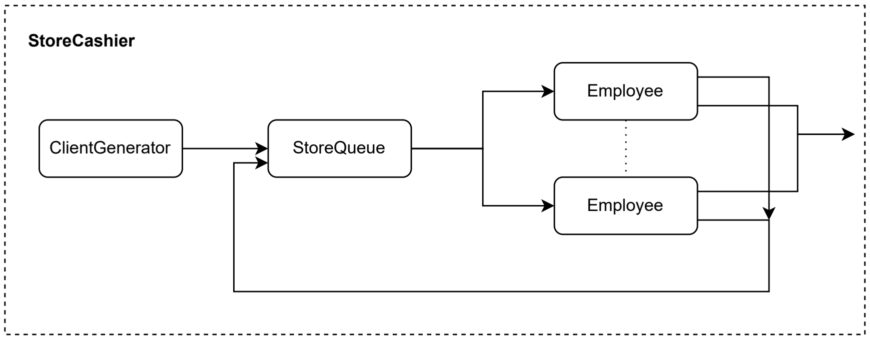

Figure 6: DEVS scenario under study.

To evaluate the new RT simulation algorithm, the system is first fully simulated in RT as a single, complete, coupled DEVS model. Subsequently, the system is partitioned into independent subsystems emulating a HIL methodology, incorporating I/O handlers and various communication protocols.

Table 1 lists the parameters used to configure the system and its simulation. By setting the values for *max_-jitter* and *event_window*, the system is configured to tolerate time delays up to a maximum of 0.2 seconds, and events are processed collectively if they occur within 0.5 seconds. A *time_scale* value of 1 indicates that the simulation time matches the wall-clock time. Regarding system parameters, the time spent by clients and employees in the queue and interacting with purchases is modeled using a Gaussian distribution.

Table 1: Parameters of a basic DEVS simulation.

Configuration	Parameter	Code name	Value	Units
Simulation	Duration of the simulation	*sim_time*	52	s
	Maximum delay of the system	max_jitter	0.2	s
	Period for acceptance of subsequent events	event_window	0.5	s
	Simulation time scale	time_scale	1	
System	Amount of employees	*n_employees*	3	
	Clients arrivals period	*mean_generator*	3	s
	Period of each employee	*mean_employees*	5	s
	Standard deviation of client arrivals	*stddev_clients*	0.5	s
	Standard deviation of employees	*stddev_employees*	0.8	s

In the first experiment, as we are conducting a RT simulation, we expect to match the desired simulation time to the actual recorded simulation time. To this end, Table 2 presents the results of five experiments carried out to assess the precision of the simulation system when performing RT simulations. Each experiment has a different simulation duration, as detailed in the first column. The mean actual duration is then determined from five simulation runs at each scheduled time. This table also shows a confidence interval of the 95% considering that the samples follow a t-distribution.

Table 2: Real-time simulation results and error.

Desired Time (s)	Mean actual time (s)	Error (%)
13	13.0078 ± 0.0058	0.0600
31	31.0118 ± 0.0161	0.0380
52	52.0073 ± 0.0036	0.0140
121	121.0054 ± 0.0020	0.0045
185	185.0062 ± 0.0034	0.0034

The relative error is calculated as follows:

$$e(\%) = \frac{RealSimulatedTime - DesiredSimulatedTime}{DesiredSimulatedTime} \cdot 100, \tag{4}$$

and expresses the difference between the actual and expected duration as a percentage. The data reveal that the variance between the actual and scheduled duration across the experiments is negligible and that the error decreases as the target simulation duration increases. These results are positive because the aim of the first experiment is to guarantee a precise simulation in wall-clock terms. In addition, the final time of each test is the result of the correct functioning of each component, which also progresses in RT. Therefore, it supports the conclusion that employing the proposed RT simulation algorithm achieves the expected behavior.

After validating the proposed RT simulation approach, we now increase the complexity of both the simulation and the system. First, we divide the system into separate DEVS and non-DEVS subsystems. These subsystems interact with each other via I/O handlers. The new system under study is depicted in Figure 7, featuring three autonomous subsystems. **StoreQueue** and **Employees** are now two independent DEVS models. Alternatively, the *ClientGenerator* is now a non-DEVS process that behaves as a customer generator executing a **TCP script**. However, with the presence of three distinct systems, communication now occurs over TCP and MQTT protocols (as seen in Figure 7), rather than through regular DEVS couplings. Additionally, a CSV file documents the activity logs of the employees. Despite the fact that this experiment is only software-based, it does not limit the possibilities of a HIL simulation. It not only incorporates well-known hardware protocols such as MQTT, but also validates communication among independent systems. This scenario demonstrates the feasibility of incorporating HIL simulations with the proposed methodology, where the system can be partitioned while maintaining its original functionality.

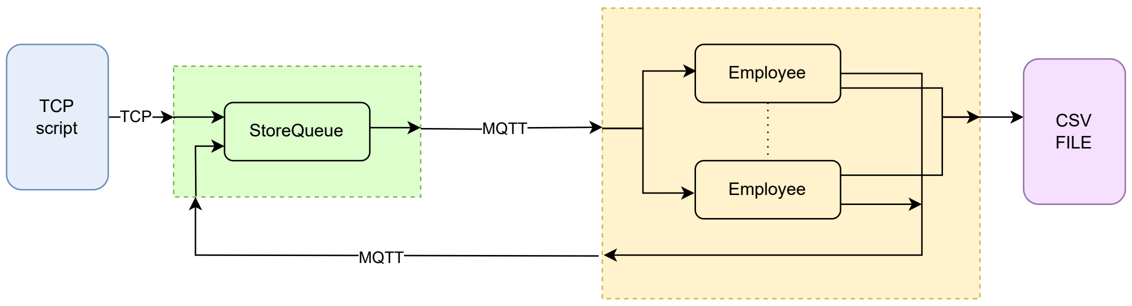

Figure 7: DEVS scenario with non-DEVS models and mixed protocols.

Figure 8 shows a sequence diagram that illustrates the interactions within the system. The result of simulating the new scenario is positive; however, providing the result of each independent system is out of the scope of this paper. For that reason, the interactions shown in Figure 8 summarized the result of simulating the new system. Observe how communication is initiated using TCP or MQTT protocols, as shown in the lower part of the diagram. Before message transfer, each DEVS subsystem must initialize its corresponding I/O handlers. Should any issues arise with these handlers, the system reports the error. Once the handlers are activated successfully, message exchanges begin. The TCP script directs clients to the queue. When these clients are received as TCP events, the queue tracks the availability of the employees. When a client is present and an employee is free, the queue pairs them and notifies the Employee system with an MQTT message. Upon receiving this message, the employee dedicates their time to the client and upon completion, the transaction is logged in the CSV file.

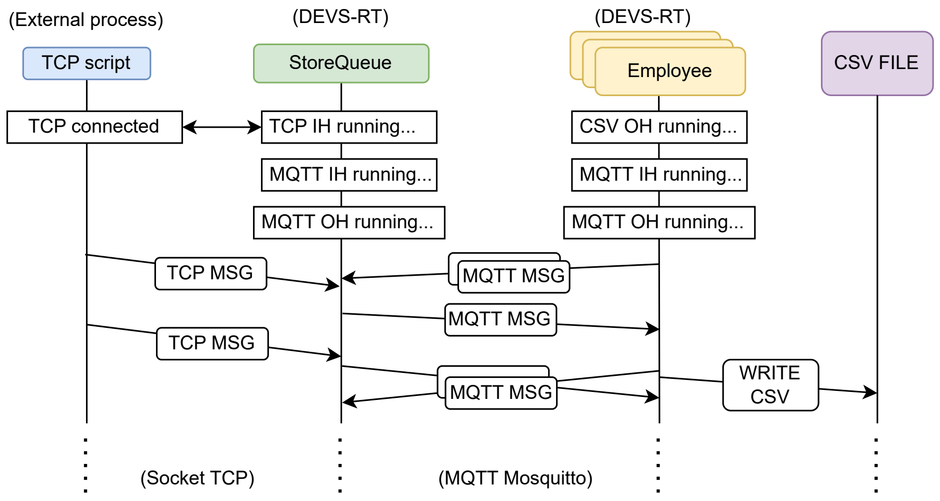

Figure 8: Sequence diagram of a complex system.

To replicate the simulations carried out in this section, the proposed scenarios and instructions to follow can be found in GitHub.

5 CONCLUSION

M&S are indispensable tools in the design and development of complex systems, applicable across various fields and levels of expertise. M&S not only enables the identification and rectification of potential problems early in the design process but also enhances the overall performance of the system by allowing iterative improvements before committing to large-scale production. Despite the widespread utility of simulation tools, a common challenge is the lack of a unified methodology for implementation, which can complicate the integration of disparate systems and hinder collaborative efforts. This paper addresses this challenge by presenting an RT simulation engine that leverages the DEVS formalism to facilitate the integration of DEVS and non-DEVS systems. The modular nature of DEVS also supports the incorporation of HIL techniques during system implementation.

The DEVS simulation engine developed in this work builds on the existing xDEVS framework, improving it with RT simulation capabilities and enabling inter-system communication through various protocols. We have defined and implemented an RT algorithm that allows for the simulation of DEVS models without altering their original specification. The RT behavior is achieved via RT coordinators and managers. We also developed a framework for establishing communication between systems, using RT managers and I/O handlers to support multiple communication protocols. This framework offers a flexible solution for system-specific communication needs. Furthermore, we implemented an interface for connecting DEVS and non-DEVS models. I/O handlers facilitate the translation of messages between systems, ensuring seamless communication regardless of the underlying technology. Finally, the feasibility of performing HIL simulations has been demonstrated. The article presents a gradual increase in system complexity, from RT simulation to a complex system that integrates multiple communication protocols (TCP and MQTT), CSV files for data logging, and the interplay between DEVS and non-DEVS modeled systems.

This work paves the way for more accurate and flexible RT simulations of complex systems, as well as the integration of various components of the system. Future improvements could focus on enhancing synchronization among independent systems, introducing the use of digital twins for more accurate modeling of systems, and considering alternative programming languages better suited for embedded systems, such as Rust. Nevertheless, this work provides a solid foundation for the development of robust and adaptable RT simulation tools. The Python code implemented for our simulation tool, as well as instructions to run the proposed use cases, are publicly available at GitHub [16].

ACKNOWLEDGMENTS

This work has been supported by the Research Projects IA-GES-BLOOM-CM (Y2020/TCS-6420) of the Synergic program of the Comunidad Autónoma de Madrid and SMART-BLOOMS (TED2021-130123B-I00) funded by MCIN/AEI/10.13039/501100011033 and the European Union NextGenerationEU/PRTR.

REFERENCES

[1] Antonio Cimino, Maria Grazia Gnomi, Francesco Longo, Gabriele Barone, Maddalena Fedele, and Domenico Le Piane, "Modeling & Simulation as Industry 4.0 enabling technology to support manufacturing process design: a real industrial application," *Procedia Computer Science*, vol. 217, pp. 1877–1886, Jan. 2023.

[2] N. Aripov, S. Kamaletdinov, N. Tokhirov, S. Khudayberganov, A. Bashirova, and M. Djalalovna, "Simulation modeling of train traffic based on GIS technologies," in *AIP Conference Proceedings*, Tashkent, Uzbekistan, 2023, p. 060025.

[3] N. Madadi, A. H. Roudsari, K. Y. Wong, and M. R. Galankashi, "Modeling and Simulation of a Bank Queuing System," in *Modelling and Simulation 2013 Fifth International Conference on Computational Intelligence*, Sep. 2013, pp. 209–215, iSSN: 2166-8531.

[4] T. E. Gartner and A. Jayaraman, "Modeling and Simulations of Polymers: A Roadmap," *Macromolecules*, vol. 52, no. 3, pp. 755–786, Feb. 2019.

[5] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*, 2nd ed. San Diego: Academic Press, 2000.

[6] J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys*, vol. 18, no. 1, pp. 39–65, Mar. 1986.

[7] J. L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223–252, Sep. 1977.

[8] P. Glynn, "A GSMP formalism for discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 14–23, Jan. 1989.

[9] A. Chow, B. Zeigler, and Doo Hwan Kim, "Abstract simulator for the parallel DEVS formalism," in *Fifth Annual Conference on AI, and Planning in High Autonomy Systems*. Gainesville, FL, USA: IEEE Comput. Soc. Press, 1994, pp. 157–163.

[10] M. Singh, E. Fuenmayor, E. P. Hinchy, Y. Qiao, N. Murray, and D. Devine, "Digital Twin: Origin to Future," *Applied System Innovation*, vol. 4, no. 2, p. 36, Jun. 2021.

[11] M. Bacic, "On hardware-in-the-loop simulation," in *Proceedings of the 44th IEEE Conference on Decision and Control*, Dec. 2005, pp. 3194–3198, iSSN: 0191-2216.

[12] P. Forsyth, T. Maguire, and R. Kuffel, "Real time digital simulation for control and protection system testing," in *2004 IEEE 35th Annual Power Electronics Specialists Conference (IEEE Cat. No.04CH37551)*, vol. 1, Jun. 2004, pp. 329–335 Vol.1, iSSN: 0275-9306.

[13] B. Earle, K. Bjornson, C. Ruiz-Martin, and G. Wainer, "Development of a real-time DEVS kernel: RT-Cadmium," in *Proceedings of the 2020 Spring Simulation Conference*, ser. SpringSim '20. San Diego, CA, USA: Society for Computer Simulation International, May 2020, pp. 1–12.

[14] J. L. Risco-Martín, S. Mittal, K. Henares, R. Cardenas, and P. Arroba, "xDEVS: A toolkit for interoperable modeling and simulation of formal discrete event systems," *Software: Practice and Experience*, vol. 53, no. 3, pp. 748–789, Mar. 2023.

[15] C. Giridhar, *Learning Python Design Patterns*. Packt Publishing Ltd, Feb. 2016, google-Books-ID: 161KDAAAQBAJ.

[16] R. Cárdenas, Óscar Fernández-Sebastián, and K. Henares, "xdevs.py: Version of the xDEVS simulator for Python projects," 2023. [Online]. Available: https://github.com/iscar-ucm/xdevs.py

AUTHOR BIOGRAPHIES

ÓSCAR FERNÁNDEZ-SEBASTIÁN graduated in Telecommunications Engineering from the Universidad Politécnica de Madrid (UPM), Spain, in 2023, where he is pursuing a Master of Engineering in Telecommunications. He is currently working as a research assistant at the Department of Computer Architecture and Automation, Universidad Complutense de Madrid (UCM), Spain. His research interests include real-time simulation environments. His email address is osfern06@ucm.es

ROMÁN CÁRDENAS is a Teaching Assistant in the Department of Electronic Systems at Universidad Politécnica de Madrid (UPM), Spain. He obtained a Ph.D. in Electronic Systems Engineering and a Ph.D. in Electrical and Computer Engineering in Cotutelle modality at UPM and Carleton University (CU), Canada. His research interests include modeling and simulation with applications in the IoT domain. His email address is r.cardenas@upm.es.

PATRICIA ARROBA is an Associate Professor in the Department of Electronic Systems at Universidad Politécnica de Madrid (UPM), Spain, where she received her Ph.D. in Telecommunication Engineering. Her research interests include energy and thermal-aware modeling, simulation, and optimization for Cloud and Edge Computing infrastructures. She can be reached at p.arroba@upm.es.

JOSÉ L. RISCO-MARTÍN received his Ph.D. from Universidad Complutense de Madrid (UCM), Spain, where he currently is a Full Professor in the Department of Computer Architecture and Automation. His research interests include computer-aided design and modeling, simulation, and optimization of complex systems. His email address is jlrisco@ucm.es.