XDEVS NO_STD: A RUST CRATE FOR REAL-TIME DEVS ON EMBEDDED SYSTEMS

Román Cárdenas[a b], Pedro Malagón[a b], Patricia Arroba[a b], and José L. Risco-Martín[c]

[a]Integrated Systems Laboratory, Universidad Politécnica de Madrid, Spain
{r.cardenas, pedro.malagon.marzo, p.arroba}@upm.es
[b]Center for Computational Simulation, Universidad Politécnica de Madrid, Spain
[c]Department of Computer Architecture and Automation, Universidad Complutense de Madrid, Spain
jlrisco@ucm.es

ABSTRACT

Modeling and Simulation (M&S) plays a crucial role in the design and analysis of complex systems, with the Discrete EVent System specification (DEVS) formalism being a widely adopted mathematical framework. This paper introduces xDEVS `no_std`, the first version of xDEVS written in the Rust programming language's `no_std` environment. Rust's features, including a data ownership mechanism, enable the development of high-performance, memory-safe simulations. xDEVS `no_std` focuses on Real-Time (RT) simulation for safety-critical embedded applications, leveraging Rust's abstractions to simplify code sharing and cross-compilation. The paper outlines the implementation design and Application Programming Interface (API), which facilitates the creation of both atomic and coupled DEVS models. The RT simulator integrates with hardware, handling external interrupts and enabling interactions with the embedded system. A use case on a RISC-V microcontroller demonstrates xDEVS `no_std`'s capabilities, illustrating how it can effectively orchestrate tasks of Cyber-Physical System (CPS) on embedded platforms.

Keywords: DEVS, embedded systems, real-time simulation.

1 INTRODUCTION AND RELATED WORK

Modeling and Simulation (M&S) have become indispensable in designing, developing, and analyzing complex systems [1]. Among the plethora of theoretical M&S methods, Discrete EVent System specification (DEVS) offers a robust mathematical framework that models systems as entities where state transitions are driven by discrete events, which can theoretically occur an infinite number of times over a continuum of time [2]. DEVS adopts a hierarchical structure to encapsulate the behavioral and structural characteristics of systems using the principles of set theory. The effectiveness of DEVS as a universal modeling formalism has been acknowledged, facilitating the completeness, verifiability, extensibility, and maintainability of the models [3]. The versatility of DEVS models is underscored by the array of computational frameworks that facilitate their simulation. Among them, the xDEVS framework [4] provides a cross-platform multi-programming language Application Programming Interface (API) that unifies existing DEVS tools with standardized wrappers while providing optimal performance for sequential and parallel simulation [5].

The characteristics of DEVS make it suitable for Hardware-In-The-Loop (HIL) simulation to validate a system under test [6]. Furthermore, DEVS has been extended to support Real-Time (RT) execution to aid during the design and implementation of Cyber-Physical Systems (CPSs) [7]. For example, the Cadmium simulation engine is a specialized RT kernel for bare-metal embedded systems [8]. The effectiveness of this approach has been demonstrated with different use cases, such as sensor fusion algorithms in smart

Proc. of the 2024 Annual Simulation Conference (ANNSIM'24), May 20-23, 2024, American University, DC, USA
C. Ruiz-Martin, B. Oakes and R. Cárdenas, eds.

buildings [9]. However, shipping DEVS models on embedded systems without an operating system often involves intricate workflows, such as programming in low-level languages and using complex cross-compilation procedures or remote debugging tools. Dynamic memory management, a common requirement in such tools, introduces uncertainty in predictability and can compromise memory safety. Such issues are generally avoided or even banned in safety-critical system standards for well-founded reasons [10]. Another aspect to consider is providing software that guides users in complying with the DEVS formalism while describing their models. In this context, Cadmium [11] is a C++ framework that follows a dynamic meta-programming approach to ensure that the model under study fits the DEVS formalism. This tool has been successful in standard models. However, complex models show long compilation and execution overheads due to their multiple model verification routines. Furthermore, while Cadmium checks compliance with DEVS, it cannot avoid possible memory safety errors in the model logic.

In this context, the Rust programming language [12] is an opportunity to address these issues. Rust is a high-level programming language that provides powerful abstractions without giving up performance. Its modern ecosystem simplifies code sharing and cross-compilation, and the Rust compiler enforces a data ownership mechanism to detect potential logic errors in compile time, guiding developers to create safer and higher-quality code. Rust allows you to define complex macros so that the compiler generates automatic code, analyzes software properties, and, in the event of an error, prints understandable messages to guide the developer in resolving it. Rust also presents a `no_std` environment [13] that opts out of those parts of Rust's standard library that rely on services usually provided by an operating system (e.g., file system or dynamic memory allocation). This helps us to develop multi-platform software that makes no assumptions about the system that will run the program. Rust `no_std` enables the development of more robust and predictable code that can be executed by a wider set of systems, from cloud servers to bare metal microcontrollers.

This paper presents xDEVS `no_std`, the first version of xDEVS that enables RT simulation of DEVS models on embedded systems. This simulator is written in Rust `no_std` and benefits from the memory safety checks and optimizations of the Rust compiler. To our knowledge, xDEVS `no_std` is the first DEVS simulation tool that opts out of using dynamic memory allocation, which makes it potentially suitable for safety-critical embedded applications. All design decisions of xDEVS `no_std` aim to help during the design, verification, and implementation of embedded systems for safety-critical applications with different technical requirements. The simulation algorithm leaves RT execution details open for different platforms, thus adapting to the requirements of each particular use case. For example, hard real-time systems can adapt the timing behavior of xDEVS `no_std` to ensure that the application timing constraints are met during execution. It also follows a robust typing approach that allows compile-time analysis of the model to verify that it complies with the DEVS specification, improving the correctness of the models under study without affecting run-time performance. The DEVS simulation acts as a lightweight RT kernel of the running embedded system. Functions to generate DEVS models are marked as constant (i.e., evaluated at compile time and always return the same outcome). In this way, the compiler can hard-code them, reducing the model creation overhead and enhancing the predictability of the application. Furthermore, the API provides the `component!` macro, an easy-to-use Rust procedural macro that simplifies the model description process and displays valuable information about potential model errors to guide users in tracking any possible problem. Rust Integrated Development Environments (IDEs) can use this information to outline where the model error is and to reduce the model description times. The xDEVS `no_std` crate is publicly available on `crates.io` [14].

The rest of the paper is organized as follows. Section 2 provides an overview of the software architecture of xDEVS `no_std`. We elaborate on the public API of the crate in Section 3, and Section 4 describes a use case to illustrate how to create a DEVS-based embedded application executed by a RISC-V microcontroller. Finally, we present some conclusions and future work in Section 5.

2 IMPLEMENTATION DESIGN

This section presents the implementation design of the xDEVS `no_std`. Figure 1 shows a Unified Modeling Language (UML) diagram of all the elements comprising this Rust crate. Gray blocks represent *private unsafe traits*. These traits are interfaces that xDEVS `no_std` automatically implements for all user-defined DEVS models after verifying their proper definition. By marking private traits as *unsafe*, the Rust compiler will not allow users to manually implement them. However, all the associated functions of these traits are safe and comply with the Rust compiler memory safety checks. Alternatively, white blocks denote *public structures and traits* that users manipulate to model and simulate their system under study.

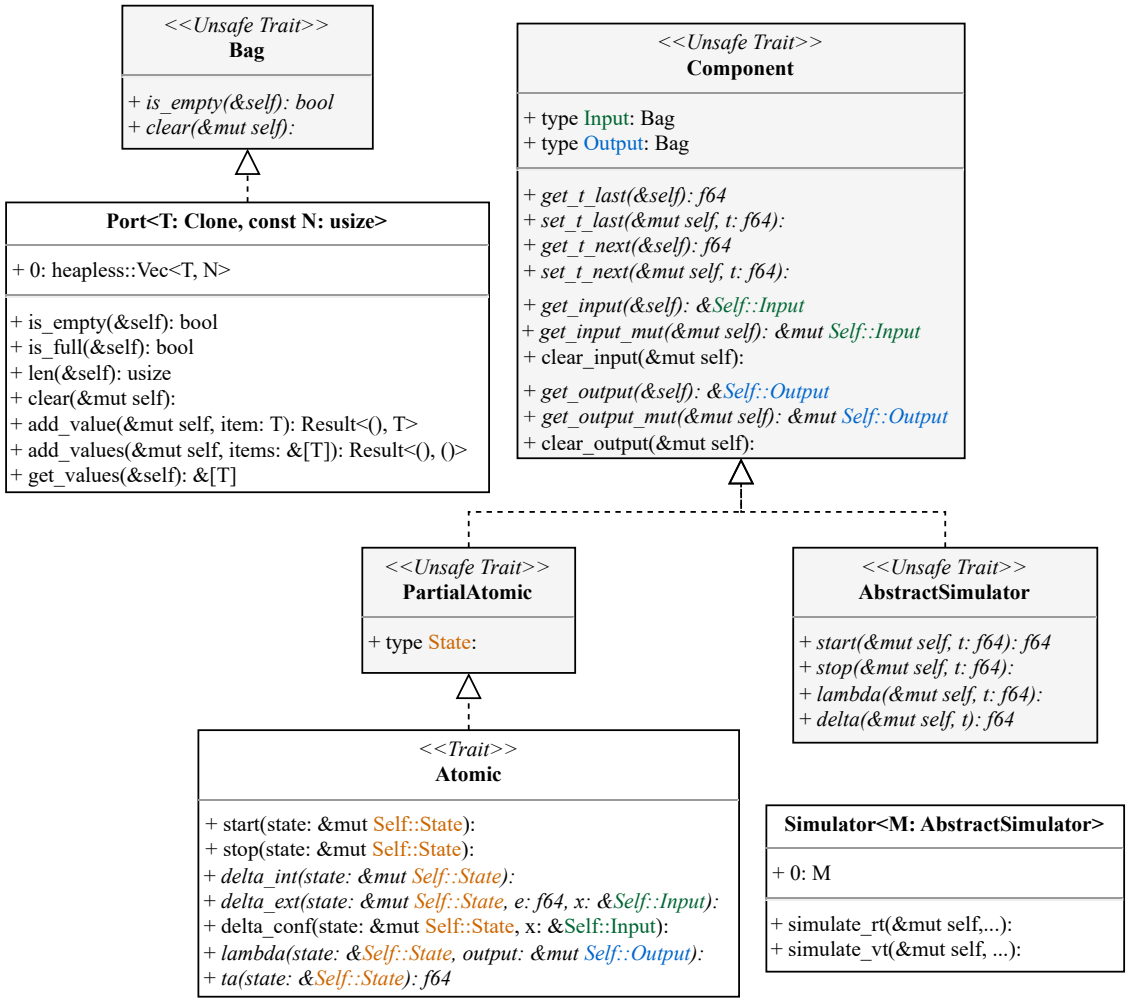


Figure 1: UML diagram of the xDEVS `no_std` crate.

The `unsafe trait Bag` defines the interface that any DEVS event bag must implement. In xDEVS `no_std`, the event bags must be data structures composed of one or more instances of `struct Port<T, const N>`. The generic type `T` determines the type of event data accepted by the port. Attempting to push an event of a different data type triggers a compilation error. On the other hand, the constant generic `N` specifies the maximum number of events that the port can contain simultaneously. As discussed in Section 1, Rust `no_std` does not provide any memory allocation mechanism, and while it is possible to define a custom allocator, this approach is often disregarded in safety-critical applications. To address this issue, `struct Port<T, N>` uses a vector implementation of the `heapless` crate [15] that does not use memory allocation.

DEVS models must implement the `unsafe trait Component`. The `type Input` and `type Output` associated types determine the data type used by the model to represent its input and output event set, respectively. Note that both associated types must implement the `unsafe trait Bag`. Otherwise, xDEVS `no_std` causes a compilation error. Atomic DEVS models must also implement the `unsafe trait PartialAtomic`, which determines the associated `type State` used to represent the model's state. Otherwise, the Rust compiler will not allow implementing the public `trait Atomic` to describe the desired behavior of the atomic model. Unlike C, Rust allows us to determine which references are mutable. Therefore, we can use this feature to enforce the correct usage of the DEVS specification at compilation time. For example, the `fn delta_ext` function of the `trait Atomic` expects a mutable reference of the model state and an immutable reference of its input bag. Alternatively, its `fn lambda` function cannot modify the state of the model, as it is an immutable reference. However, the output bag is mutable, allowing us to push output events. Attempting to break this contract would lead to a compilation error.

Given that xDEVS `no_std` does not use dynamic memory, high-level abstractions such as polymorphism are not possible. Therefore, unlike the other xDEVS simulation engines, every atomic and coupled model is an entirely different data structure that implements its version of the `unsafe trait AbstractSimulator`, and no code is shared among structures. In this way, the Rust compiler can check that all the described DEVS models comply with the proposed interface at no cost in execution time. Additionally, it is possible to apply better compilation optimizations to improve simulation performance. However, it will tend to produce larger binaries. Although memory size is a relatively common issue in embedded systems, we do not think this is a relevant issue for xDEVS `no_std`, as the complexity of potential embedded DEVS models would be significantly lower than other simulation tools designed to be executed on hosted systems. However, we plan to study this limitation in future work.

Structures that implement `unsafe trait AbstractSimulator` can be simulated using the public `struct Simulator<M>`. This structure provides two methods, `fn simulate_rt` and `fn simulate_vt`, to run RT and standard simulations, respectively. Section 3.2 provides more details on these methods. However, manually implementing these unsafe traits for every DEVS model is a cumbersome, repetitive, and error-prone process. The xDEVS `no_std` simulator provides the `component!` macro to aid during the model definition process. This procedural macro comprises a simple interface that analyzes the model before compiling the application and checks that it complies with the DEVS formalism. It then automatically implements all the unsafe traits according to the model specification. In addition, it displays valuable information on model errors to help users track any possible issue. Section 3.1 describes this macro.

3 APPLICATION PROGRAMMING INTERFACE

This section illustrates how to use the API of xDEVS `no_std`. We use the Generator Processor Transducer (GPT) model, a popular example in the DEVS community, to illustrate how it works. Figure 2 shows a schematic of the GPT coupled model. Every T_G s, the generator sends a job that needs to be processed via its G_{out} port. The processor receives these jobs via the P_{in} port and spends T_P s before sending the processed job via the P_{out} port. The transducer monitors the number of jobs generated and processed via the T_{gen} and T_{proc} ports, respectively. After T_T s, the observation window expires, and the transducer sends a message via the T_{out} port to tell the generator via its G_{in} port to stop generating new jobs.

3.1 DEVS Component Macro

In xDEVS `no_std`, all the DEVS models are declared using the `component!` macro. This macro simplifies the model description process by automatically implementing all the unsafe traits of xDEVS `no_std`. It also analyzes the model structure to ensure that it properly follows the DEVS formalism and triggers useful compilation error messages to guide users. The `component!` macro accepts the following parameters:

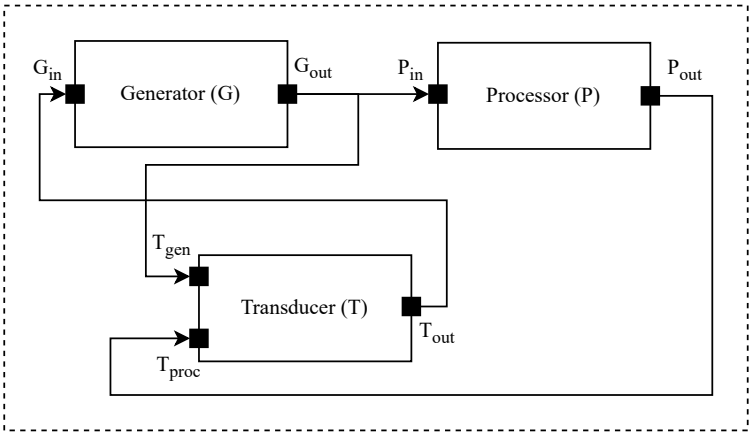


Figure 2: Generator Processor Transducer (GPT) model.

- `ident` (mandatory): it determines the name of the DEVS model to be created.
- `input` (optional): it describes the input ports of the model. If this field is not provided, it assumes that the model has no input port. Input ports are provided as a comma-separated list between curly brackets. Each port must follow the format `name<data[, size]>`, where `name` corresponds to the port name, `data` is the data type of the messages of the port, and `size` determines the port capacity. If `size` is not provided, it defaults to 1. Port names must be unique.
- `output` (optional): it describes the output ports of the model. It follows the same format as `input`.
- `state` (mandatory for atomic models): it determines which data type represents the model state.
- `components` (mandatory for coupled models): it enumerates the subcomponents of the coupled model. Components are provided as a comma-separated list between curly brackets. Each component must follow the format `name: T`, where `name` is the component name and `T` is its data type. At least one component must be provided. Component names must be unique.
- `couplings` (optional for coupled models): list of couplings provided as a comma-separated list between curly brackets. Couplings follow the format `[comp1.]port1 -> [comp2.]port2`, where `comp1` and `comp2` are component names, and `port1` and `port2` are the names of the coupled ports. If both component names are provided, it is an internal coupling. If `comp1` is missing, it is an external input coupling. Alternatively, if `comp2` is missing, the coupling is an external output coupling.

The `component!` macro automatically deduces whether the model under description is an atomic or a coupled model depending on the fields provided. Fields for atomic or coupled models are mutually exclusive, and the macro returns a compilation error if users mix them.

3.1.1 Atomic Models

For atomic models, we must provide the `state` field to the `component!` macro. For instance, if we want to define the transducer of the GPT model (see Figure 2), we must use the `component!` macro as follows:

```
xdevs::component!(
    ident = Transducer,
    input = { in_gen<usize, 2>, in_proc<usize, 1> },
    output = { out_stop<bool> },
    state = TransducerState
);
```

This macro creates three new data structures: `TransducerInput`, `TransducerOutput`, and `Transducer`. `TransducerInput` and `TransducerOutput` comprise the ports specified in the macro call and represent the input and output sets of the transducer model, respectively. The following code snippet shows the code automatically generated by the `component!` macro for the **`struct TransducerInput`**:

```
1  pub struct TransducerInput {
2    pub in_gen: xdevs::port::Port<usize, 2>,
3    pub in_proc: xdevs::port::Port<usize, 1>,
4  }
5  impl TransducerInput {
6    pub const fn new() -> Self {
7      Self{ in_gen: xdevs::port::Port::new(), in_proc: xdevs::port::Port::new() }
8    }
9  }
10 unsafe impl xdevs::aux::Bag for TransducerInput {
11   fn is_empty(&self) -> bool {
12     true && self.in_gen.is_empty() && self.in_proc.is_empty()
13 }
14   fn clear(&mut self) {
15     self.in_gen.clear(); self.in_proc.clear();
16   }
17 }
```

Lines 1 to 4 specify all the ports comprising the model input set. Additionally, lines 5 to 9 define a constant function to create new instances of this structure. Finally, lines 10 to 17 implement the **`unsafe trait`** `Bag` for the `TransducerInput` structure by propagating the method call to all its ports (note that, as shown in Figure 1, the **`struct Port`**`<T, N>` implements the **`unsafe trait`** `Bag`). After defining the `TransducerInput` and `TransducerOutput` structures, the macro generates the following code for the **`struct Transducer`**:

```
1  pub struct Transducer {
2    pub input: TransducerInput,
3    pub output: TransducerOutput,
4    pub t_last: f64, pub t_next: f64,
5    state: TransducerState,
6  }
7  impl Transducer {
8    pub const fn new(state: TransducerState) -> Self { /* Omitted for brevity */ }
9  }
10 unsafe impl xdevs::aux::Component for Transducer {
11   type Input = TransducerInput;
12   type Output = TransducerOutput;
13   ... /* Omitted for brevity */
14 }
15 unsafe impl xdevs::aux::PartialAtomic for Transducer {
16     type State = TransducerState;
17 }
18 unsafe impl xdevs::aux::AbstractSimulator for Transducer {
19   fn lambda(&mut self, t: f64) {
20     if t >= xdevs::aux::Component::get_t_next(self) {
21       <Self as xdevs::Atomic>::lambda(&self.state, &mut self.output);
22     }
23   }
24   ... /* Omitted for brevity */
25 }
```

As shown in lines 1 to 6, the `struct Transducer` comprises `input` and `output` fields of type `TransducerInput` and `TransducerOutput`, respectively. It also contains the `t_last` and `t_next` fields to keep track of the simulation time internally. Finally, it also has a `state` field of the data type defined in the macro call (in this example, `TransducerState`). Recall that the macro does not generate the state type and must be defined by the user before calling it. Lines 7 to 9 show that the `struct Transducer` also has a constant function to create the model. Next, lines 10 to 14 implement the `unsafe trait Component` for the model. Note that it sets the `Input` and `Output` associated types of the model to `TransducerInput` and `TransducerOutput`, respectively. If these structures did not implement the `unsafe trait Bag`, the compiler would trigger a compilation error at this point. Finally, as the `struct Transducer` corresponds to an atomic model, the `component!` macro implements the `unsafe trait PartialAtomic` to select its associated type `State` properly. It also implements the `unsafe trait AbstractSimulator` according to the DEVS abstract simulation algorithm for atomic models. Note how line 21 forces the `struct Transducer` to implement the `trait Atomic`. Otherwise, it will result in a compilation error. This trait specifies the actual behavior of the atomic model, and users must implement it according to their use case.

3.1.2 Coupled Models

For coupled DEVS models, the `component!` macro expects the `components` and `couplings` fields. If we wanted to generate the GPT model presented in Figure 2, the macro call would look as follows:

```
1  xdevs::component!(
2    ident = GPT,
3    components={generator: Generator, processor: Processor, transducer: Transducer,},
4    couplings = {
5      generator.out_job -> processor.in_job, processor.out_job -> transducer.in_proc,
6      generator.out_job -> transducer.in_gen, transducer.out_stop->generator.in_stop,
7    }
8  );
```

This model does not have input or output ports. Thus, the `GPTInput` and `GPTOutput` structures automatically generated by the macro are empty structures that implement the `unsafe trait Bag`. On the other hand, the `struct Generator` contains one field for each subcomponent, as shown in line 4 of the following snippet:

```
1  pub struct GPT {
2    pub input: GPTInput, pub output: GPTOutput,
3    pub t_last: f64, pub t_next: f64,
4    generator: Generator, processor: Processor, transducer: Transducer,
5  }
6  impl GPT {
7    pub const fn new(generator: Generator, processor: Processor,
8                     transducer: Transducer) -> Self { /* Omitted for brevity */ }
9  }
10 unsafe impl xdevs::aux::AbstractSimulator for GPT {
11   fn lambda(&mut self, t: f64) {
12     if t >= xdevs::aux::Component::get_t_next(self) {
13       xdevs::aux::AbstractSimulator::lambda(&mut self.generator, t);
14       xdevs::aux::AbstractSimulator::lambda(&mut self.processor, t);
15       xdevs::aux::AbstractSimulator::lambda(&mut self.transducer, t);
16     }
17   }
18   ... /* Omitted for brevity */
19 }
```

Again, the macro implements a constant constructor function for the coupled model to enhance compile-time optimizations (lines 6 to 9). As the `struct GPT` is not an atomic model, it does not implement the `unsafe trait PartialAtomic`, and the logic of its implementation of the `unsafe trait AbstractSimulator` (lines 10 to 19) corresponds to a coordinator in the DEVS abstract simulation algorithm. Note how, in lines 13 to 15, this implementation explicitly propagates the `fn AbstractSimulator::lambda` call to each subcomponent of the DEVS model. It also forces all these subcomponents to implement the `unsafe trait AbstractSimulator`. Otherwise, it will not compile.

3.2 Real-Time Simulator

The xDEVS `no_std` simulation engine is designed as a tool to develop CPSs that follow the DEVS specification. It acts as a lightweight RT kernel of the running embedded system. In this context, we must provide a DEVS simulation algorithm that i) translates simulation time to wall-clock time, ii) allows us to map external interrupts of the embedded system to input events of the model to alter the simulation, iii) allows us to map output events of the model to different tasks that the CPS may execute, and iv) adapts to different hardware constraints. The `struct Simulator` of xDEVS `no_std` provides the method `fn simulate_rt` to meet all these requirements. Figure 3 represents a workflow diagram of the simulation algorithm. The function takes five input parameters: the model under study (`model`), the initial and final simulation times (`t_start` and `t_stop`), a closure to wait for internal and external events (`wait_until`), and a closure to translate output events into other actions (`propagate_output`). Closures are anonymous functions that can capture values as private variables that persist in every call [12], hiding implementation-specific details under a common interface. Note that, when interacting with the model, the simulator can only use the `trait AbstractSimulator`, which provides a common interface for atomic and coupled models.

Initially, the simulation time `t` is set to `t_start`. The simulation time of the first internal event is given by the `fn start` method of the DEVS component. Next, the simulation algorithm enters the simulation loop. We refer to each iteration of this loop as a simulation step. Once the simulation time reaches `t_stop`, the algorithm exits this loop, executes the `fn stop` method of the component, and terminates the execution. In every simulation step, the algorithm first executes the `wait_until` closure. This closure expects two input arguments: the simulation time of the next expected internal event and a mutable reference to the input bag of the model. This closure translates the next expected simulation time into wall-clock time and pauses the execution of the simulation until this time. However, if an external event occurs, the closure may inject input events into the input bag and return ahead of schedule. In any case, it returns the new wall-clock time translated into simulation time, `t`. Note that `t` must be less than or equal to `t_next_internal`.

If `t` is less than `t_next_internal`, the algorithm checks if the model received any input event, and if so, it executes the model's state transition function, `fn delta`. Alternatively, if `t` is equal to `t_next_internal`, simulation time advanced until the next expected internal event. Therefore, the algorithm must execute the model's output function, `fn lambda`, before executing its state transition function. Note that the algorithm calls the `propagate_output` closure every time it executes `fn lambda`. This closure receives an immutable reference to the model's output bag, allowing it to execute arbitrary code outside of the simulation as a response to the output events.

4 USE CASE

This section illustrates how to use xDEVS `no_std` to develop CPSs on embedded systems. The DEVS model used to illustrate how this tool works is the Processor Transducer (PT) model, a modification of GPT that removes the generator atomic model. Figure 4 shows a schematic of the proposed model. This model receives new jobs to be processed through the PT_{in} port. Incoming jobs are sent to the processor and transducer models, which behave exactly as in the GPT model. Now, the stop message sent by the transducer

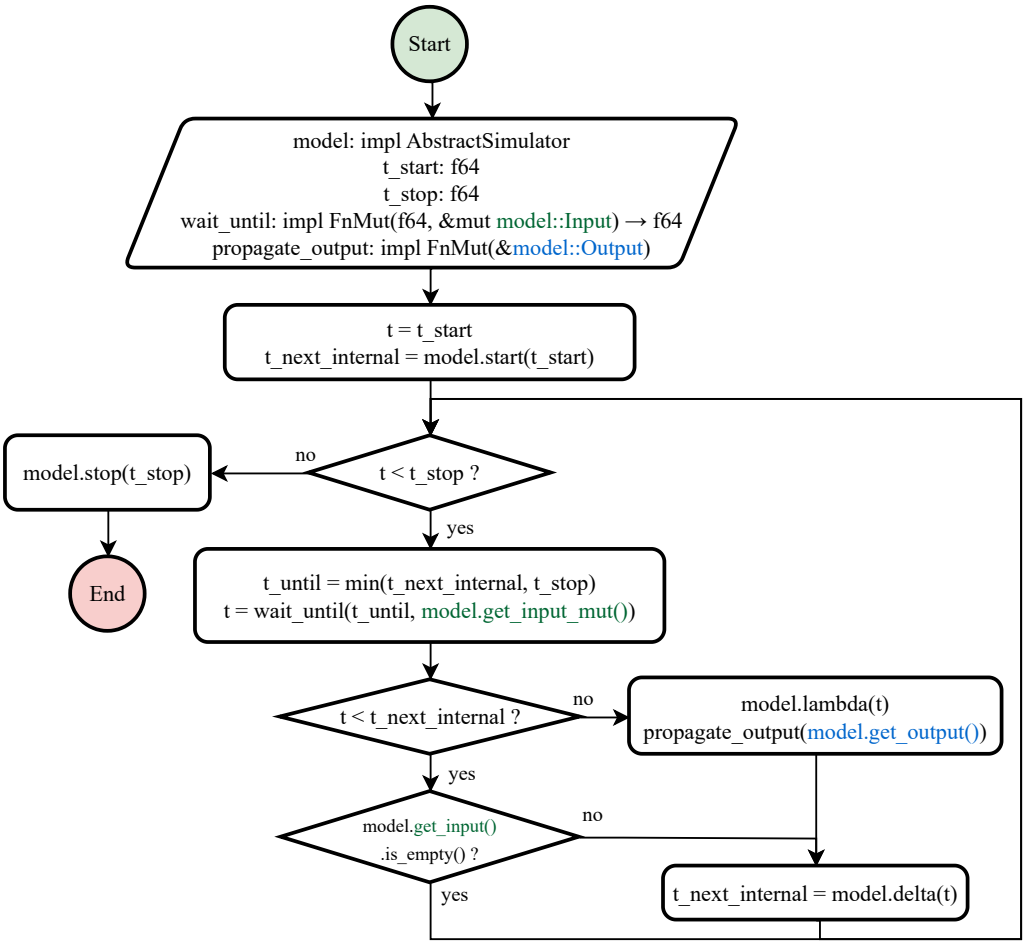


Figure 3: Real-time simulation workflow diagram.

is sent outside the model via the PT_{out} port. We use a button and an RGB Light Emitting Diode (LED) to illustrate the different ways xDEVS `no_std` can interact with the hardware of a CPS. Users can inject new jobs into the model by pressing the button. Event injection is handled by the `wait_until` closure. The state transition functions of the processor model control the red LED to indicate whether the processor is busy. Alternatively, the `propagate_output` closure will turn on the blue LED when the transducer model sends a stop message. Finally, the system will turn on the green LED once the simulation stops.

The platform in this use case is a SparkFun RED-V Things Plus, a development board that integrates a Freedom E310-G002 RISC-V microcontroller. The source code for this use case is publicly available in GitHub [16] together with a demonstration video and more examples. Figure 5 shows the hardware configuration. It comprises an RGB LED with its corresponding ballast resistors and a mechanical button. The LED is connected to digital pins 0 to 2, configured as non-inverted outputs. Alternatively, the button is connected to digital pin 9, configured as a pulled-up input pin.

Implementation details on the `wait_until` closure are hardware-dependent, and each platform must implement its logic. In this case, we followed an interrupt-driven approach that allows the embedded system to sleep between simulation steps. To do so, we use the board's Core Local Interruptor (CLINT) peripheral to schedule machine timer interrupts at the wall-clock time corresponding to the simulation time of the next expected internal event. Alternatively, each time the button is pressed, the Interrupt Service Routine (ISR)

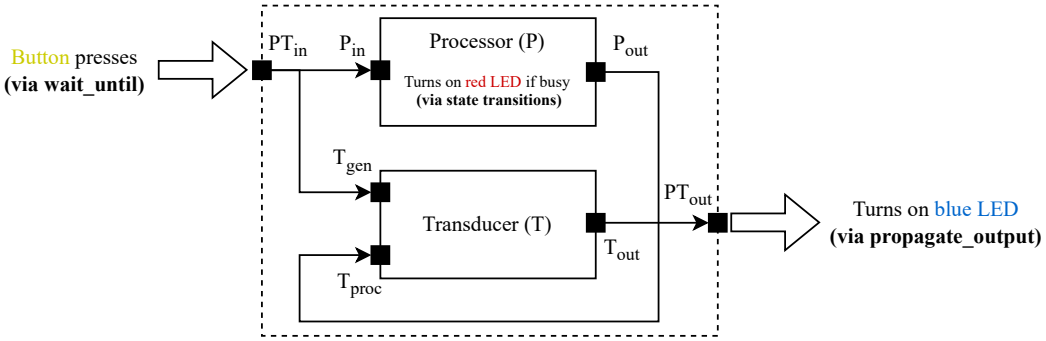
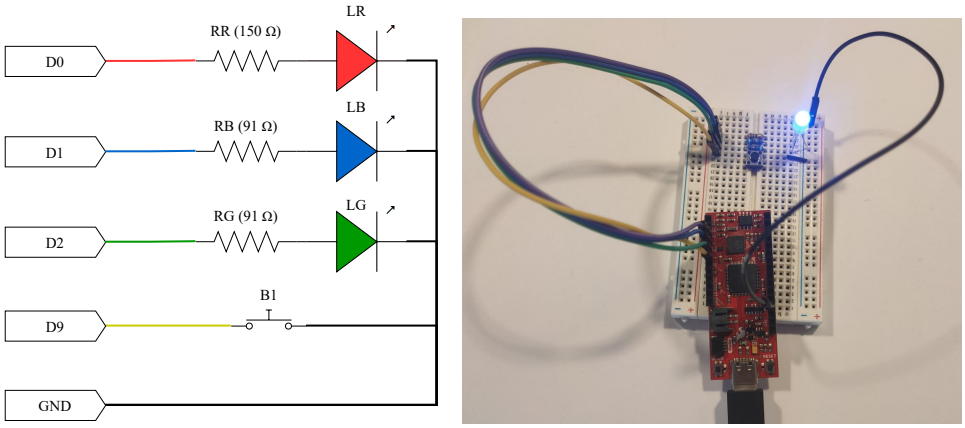


Figure 4: Schematic of the proposed Processor-Transducer (PT) model.



(a) Schematic of the circuit.

(b) Platform under test.

Figure 5: Hardware used for the use case.

assigned to this input sets a static atomic flag, `PRESSED`, to notify the `wait_until` closure. Figure 6 shows a workflow diagram of this closure proposed for this use case. First, it translates the simulation time of the next expected internal event to the corresponding tick of the CLINT's `mtime` register. This register is a monotonic clock that increases its value by one at a frequency of 32.768 kHz. Next, it checks if the button has been pressed or if CLINT reached the expected tick. If not, it configures the CLINT's `mtimecmp` register to cause a timer interrupt as soon as the `mtime` register reaches `next_tick` and executes the RISC-V's `wfi` instruction to wait for an interrupt to occur. When a button press is detected, `PRESSED` is atomically cleared and `wait_until` pushes a new message to the input port of the PT model. After receiving an interrupt, the closure disables CLINT interrupts and returns the simulation time corresponding to the current system tick.

In the proposed example, we set the time required by the processor to process a new job, T_P, to 2.1 s. Alternatively, the transducer monitoring window, T_T, is 10 s, and the simulation finishes after 15 s. Figure 7 shows a timeline of the RT simulation of the model under study. A video with the shown behavior is available on Archive. Button interrupts are represented as yellow crosses, while gray crosses correspond to CLINT timer interrupts. Input events injected by the `wait_until` closure are displayed as green dashed arrows, and output events handled by the `propagate_output` closure are blue dashed arrows.

Initially, the `wait_until` closure schedules a timer interrupt at the end of the transducer's observation window (i.e., $t = 10$ s). However, we press the button at $t = 4$ s. The `wait_until` closure wakes up earlier than expected and injects a new job into the model's input port. As a result, the processor turns on the red

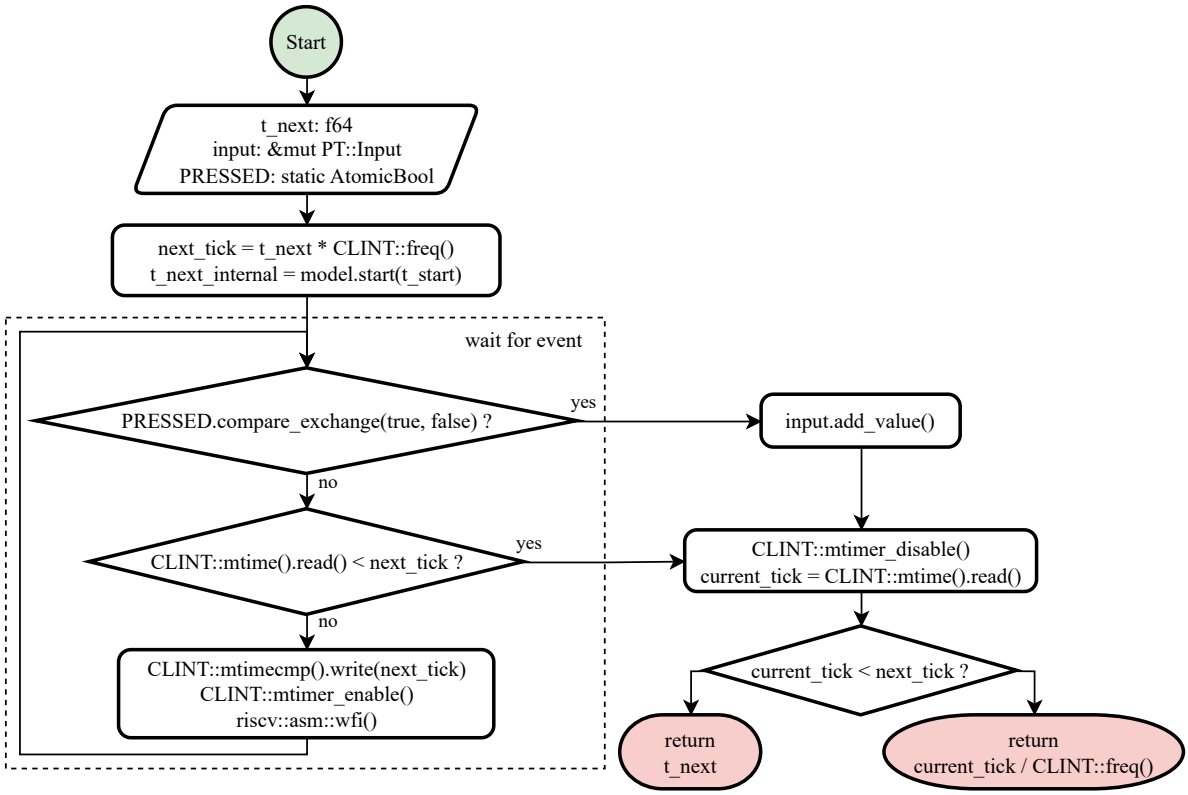


Figure 6: Workflow diagram of the `wait_until` closure for the proposed use case.

light LED to inform the user that it is busy processing this new job. While the processor is busy, it ignores any input job. At $t = 4 + T_P = 6.1\,\text{s}$, CLINT's timer causes an interrupt, and the simulation resumes. The processor is free again and turns off the red LED to notify that it now accepts incoming requests. In the example presented, we press the button again at $t = 9\,\text{s}$, keeping the processor busy (and the red LED turned on) until $t = 11.1\,\text{s}$. However, CLINT's timer raises an interrupt earlier at $t = 10\,\text{s}$ because the transducer's observation window finished. The transducer stop message is received by the `propagate_output` closure and consequently turns on the blue LED. As the processor is still busy, the red LED is still on, and the RGB LED remains magenta until the processor finishes processing the job and turns off the red LED. Finally, the LED remains blue until $t = 15\,\text{s}$, when simulation finishes. To inform users of its termination, green LED is turned on, showing a cyan color.

5 CONCLUSIONS AND FUTURE WORK

Integration of M&S methodologies in the design, implementation, and analysis of complex systems can significantly improve the quality and robustness of the resulting solution. Furthermore, approaches such as HIL simulation enable M&S tools to support the implementation, operation, and automation of modern embedded systems and CPSs. In this context, we present xDEVS `no_std`, a novel implementation of the xDEVS framework developed in the `no_std` environment of the Rust programming language.

xDEVS `no_std` offers a user-friendly, high-level API that automatically analyzes the model to ensure compliance with the DEVS formalism at compile time. It also benefits from Rust's data ownership model to facilitate the creation of safe, high-quality code. This tool does not rely on dynamic memory allocators to build models, leading to better predictability of the execution and making it suitable for safety-critical applications. Furthermore, it avoids making any assumptions about the hardware running the model, allowing

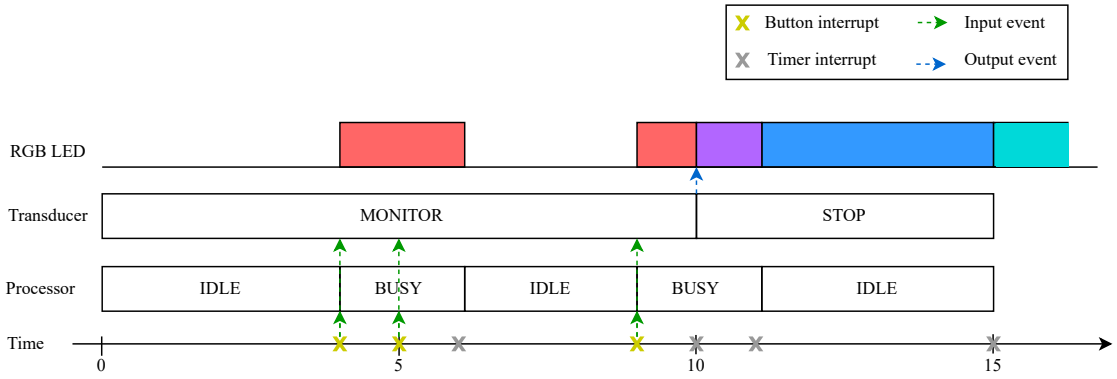


Figure 7: Timeline of the use case.

for the development of multi-platform models adaptable to different architectures and specific requirements. To demonstrate the potential utility of xDEVS `no_std` in developing modern embedded systems, we presented a use case involving a RISC-V microprocessor that uses the RT simulation algorithm of xDEVS `no_std`. This use case adopts an interrupt-driven methodology, effectively managing external interrupts by converting them into simulation input events. The system can also execute arbitrary code in response to simulation output events. In any case, the algorithm ensures that the wall-clock time is synchronized with the simulation time to deliver reliable results.

In future work, we will integrate xDEVS `no_std` with a Real-Time Operating System (RTOS). RTOSs are effective in orchestrating the concurrent execution of multiple tasks in embedded systems while ensuring that these tasks meet their timing deadlines. We believe that a collaboration between xDEVS `no_std` and an RTOS will offer substantial advantages in designing and testing embedded software in safety-critical applications. Additionally, we plan to investigate using abstract time bases to provide modelers with the flexibility to decide how simulation time is represented. This feature is particularly beneficial for platforms that lack dedicated hardware support for floating-point arithmetic. On the other hand, we want to extend xDEVS `no_std` to allow nested coupled models. Currently, coupled models must be owners of their subcomponents. This approach does not support nested coupled models, as the size of such structures cannot be known at compile time. We will overcome this limitation by allowing subcomponents to be mutable references. Unlike raw pointers, only one mutable reference to a given data structure can exist simultaneously. Finally, we will study the limitations that xDEVS `no_std` may face in terms of scalability as the complexity of the model increases. Although the DEVS models we expect to develop for embedded systems will not present complex topologies, it will be interesting to assess how a stack-only simulator performs compared to state-of-the-art dynamic memory-based solutions.

ACKNOWLEDGMENTS

This work has been supported by the Research Project SMART-BLOOMS (TED2021-130123B-I00), funded by MCIN/AEI/10.13039/501100011033 and the European Union NextGenerationEU/PRTR.

REFERENCES

[1] A. W. Wymore, *Model-Based Systems Engineering*, 1st ed. CRC Press, May 2018.

[2] B. P. Zeigler, A. Muzy, and E. Kofman, *Theory of modeling and simulation: discrete event and iterative system computational foundations*, 3rd ed. London, United Kingdom: Academic Press, an imprint of Elsevier, 2019, oCLC: 1049151782.

[3] H. Vangheluwe, "DEVS as a common denominator for multi-formalism hybrid systems modelling," in *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design*. Anchorage, AK, USA: IEEE, 2000, pp. 129–134.

[4] 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.

[5] R. Cárdenas, P. Arroba, and J. L. Risco-Martín, "A New Family of xDEVS Simulators for Enhanced Performance," in *2023 Annual Modeling and Simulation Conference*, May 2023, pp. 668–679.

[6] J. L. Risco-Martín, S. Mittal, J. C. Fabero, P. Malagón, and J. L. Ayala, "Real-time hardware/software co-design using DEVS-based transparent M&S framework," in *Proceedings of the Summer Computer Simulation Conference*, 2016, pp. 1–8.

[7] H. S. Sarjoughian and S. Gholami, "Action-level real-time DEVS modeling and simulation," *SIMULATION*, vol. 91, no. 10, pp. 869–887, 2015.

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

[9] J. Boi-Ukeme and G. Wainer, "A Framework for the Extension of DEVS With Sensor Fusion Capabilities," in *2020 Spring Simulation Conference (SpringSim)*, 2020, pp. 1–12.

[10] S. Andalam, P. S. Roop, A. Girault, and C. Traulsen, "A Predictable Framework for Safety-Critical Embedded Systems," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1600–1612, Jul. 2014.

[11] L. Belloli, D. Vicino, C. Ruiz-Martin, and G. Wainer, "Building DEVS Models with the Cadmium Tool," in *2019 Winter Simulation Conference (WSC)*, 2019, pp. 45–59.

[12] S. Klabnik and C. Nichols, *The Rust programming language*, 2nd ed. San Francisco: No Starch Press, 2023.

[13] J. Gjengset, *Rust for rustaceans*. San Francisco: No Starch Press, 2022.

[14] R. Cárdenas, "xdevs-no-std: no_std implementation of xDEVS for real-time simulation on embedded systems," Jan. 2024. [Online]. Available: https://crates.io/crates/xdevs-no_std

[15] J. A. Rivera, M. Lindner, and P. Lindgren, "Heapless: Dynamic Data Structures without Dynamic Heap Allocator for Rust," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. Porto: IEEE, Jul. 2018, pp. 87–94.

[16] R. Cárdenas, "Examples of xDEVS no_std on RISC-V," Jan. 2024. [Online]. Available: https://github.com/iscar-ucm/riscv-xdevs

AUTHOR BIOGRAPHIES

ROMÁN CÁRDENAS is a Teaching Assistant at Universidad Politécnica de Madrid (UPM), Spain. He obtained a Ph.D. in Electronic Systems Engineering in Cotutelle modality at UPM and Carleton University. His research interests include M&S in the IoT domain. His email address is r.cardenas@upm.es.

PEDRO MALAGÓN is an Associate Professor at Universidad Politécnica de Madrid (UPM), Spain, where he received his Ph.D. in Telecommunication Engineering. His research interests include security in embedded systems considering computer architecture. His email address is pedro.malagon.marzo@upm.es

PATRICIA ARROBA is an Associate Professor at Universidad Politécnica de Madrid (UPM), Spain, where she received her Ph.D. in Telecommunication Engineering. Her research interests include energy and thermal-aware M&S&O for Cloud and Edge 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. His research interests include computer-aided design and modeling, simulation, and optimization of complex systems. His email address is jlrisco@ucm.es.