A NEW FAMILY OF XDEVS SIMULATORS FOR ENHANCED PERFORMANCE

Román Cárdenas
Patricia Arroba

Laboratorio de Sistemas Integrados (LSI)
CCS–Center for Computational Simulation
Universidad Politécnica de Madrid
ETSI Telecomunicación, Avenida Complutense, 30
Madrid 28040, SPAIN
{r.cardenas,p.arroba}@upm.es

José L. Risco-Martín

Dpt. of Computer Architecture and Automation
Universidad Complutense de Madrid
C/ Profesor José García Santesmases, 9
Madrid 28040, SPAIN
jlrisco@ucm.es

ABSTRACT

Modeling and Simulation (M&S) techniques are extensively used for analyzing complex systems. Among different approaches, the Discrete EVent System specification (DEVS) mathematical formalism presents a robust methodology for defining complex models. The xDEVS framework is a cross-platform tool that provides a uniform Application Programming Interface (API) to simulate DEVS-based models using different programming languages like C++, Java, or Python. However, executing DEVS models on embedded devices equipped with small processors usually requires lower-level programming languages. This paper introduces xDEVS/Rust and xDEVS/C, two new xDEVS interfaces written in Rust and C, respectively. These new implementations bring xDEVS closer to supporting the execution of DEVS models in embedded systems. Also, they feature remarkably high performance, making them a valuable option for simulating complex systems in traditional platforms. We illustrate how competitive these new implementations are by comparing them with some of the fastest state-of-the-art simulators.

Keywords: C, DEVS, M&S Tools, Performance, Rust.

1 INTRODUCTION

Modeling and Simulation (M&S) has been extensively utilized to design and analyze complex systems. Different M&S theoretical methods have been defined for this purpose, such as Petri Nets, Timed Automata, and the Discrete EVent System specification (DEVS) formalism (Zeigler, Muzy, and Kofman 2018). The DEVS formalism is a comprehensive method for modeling complex systems' structural, behavioral, and information aspects using mathematical set theory. DEVS also gives some benefits for analyzing and creating complex models: completeness, verifiability, extensibility, and maintainability. Systems described using the DEVS theory can be easily implemented in a computational environment across different platforms.

There are numerous libraries and tools within the M&S community that facilitate the implementation of DEVS models. Some examples are aDEVS (Nutaro 2023), Cadmium (Cárdenas and Wainer 2022), MS4Me (Seo et al. 2013), PyPDEVS (Bolduc and Vangheluwe 2002), or VLE (Quesnel, Duboz, and Ramat 2009). Among them, xDEVS is a cross-platform, object-oriented, M&S framework designed to bring some unification to the existing DEVS M&S libraries through the use of wrappers and provide optimal performance and deployment of models in parallel and distributed computing architectures (Risco-Martín

et al. 2022a, Risco-Martín et al. 2022b). xDEVS incorporates different DEVS Application Programming Interfaces (APIs) with equivalent interfaces: the original Java implementation (xDEVS/Java), a C++ implementation (xDEVS/C++), a C# implementation (DEVS/C#), a Go implementation (DEVS/Go), and a Python implementation (xDEVS/Python). However, xDEVS and DEVS APIs, in general, still need facilities for the deployment of models inside embedded systems, which highly facilitates the development of modern Cyber-Physical Systems (CPSs) or the implementation of the Digital Twin (DT) paradigm.

There exist some approaches that facilitate the deployment of Hardware-In-The-Loop (HIL)-driven DEVS M&S methods (Moallemi and Wainer 2010, Risco-Martín et al. 2016). However, incorporating programming tools that facilitate the development, like compilers, linkers, debuggers, etc., is challenging, since embedded systems are usually based on small processors with limited programming support. Here, the C programming language remains undisputed, with a plethora of Integrated Development Environments (IDEs) for embedded systems. Additionally, the Rust programming language is gaining momentum. Both languages also offer high-performance scores. Consequently, we have incorporated two new variants to the xDEVS API: xDEVS/C and xDEVS/Rust. This paper introduces xDEVS/Rust and xDEVS/C, the two new xDEVS APIs written in Rust and C, respectively. A performance analysis is also designed to show the benefits of using these two programming languages. Both APIs are available at GitHub (Risco-Martín et al. 2014). The paper is organized as follows. Section 2 provides a detailed explanation of the xDEVS/Rust architecture. Section 3 summarizes the same architecture for the xDEVS/C API. In Section 4, a set of experiments are conducted to demonstrate the good performance of these two APIs. Finally, we present some conclusions and future work in Section 5.

2 XDEVS RUST

Rust is a modern high-level programming language that provides powerful abstraction mechanisms to ease the development process without giving up performance. In addition, it provides a modern ecosystem that simplifies code sharing, dependency management, and code compilation processes. The Rust compiler enforces a data ownership mechanism to detect potential logic errors in compile time. This particularity guides developers to create safer, higher-quality code. However, Rust also introduces new challenges that make it difficult to adapt the approach used in other xDEVS tools. For instance, it does not support Object-Oriented Programming (OOP). While there are mechanisms to emulate OOP patterns, composition is preferred over inheritance. Furthermore, the data ownership scheme implies a significant change to the code structure compared to other programming languages. Figure 1 shows a simplified Unified Modeling Language (UML) diagram of xDEVS Rust crate. In Rust, software packages or libraries are referred to as crates.

Event propagation follows the message-passing over ports approach. The private structure `Bag<T>` is a container of messages of type `T`. In this way, bags can only contain messages of a given data type defined at compile time. The `Bag<T>` structure implements the `Port` trait. Rust traits are similar to interfaces. They define a set of methods and associated functions that any trait implementer must include. The `Port` trait defines the expected behavior for any port (e.g., checking if the bag is empty or clearing the bag). The `InPort<T>` and `OutPort<T>` public structures are wrappers of a `Bag<T>` that represent input and output ports, respectively. The prior only allows reading messages in the bag, while the latter allows modelers to add new messages to the underlying bag.

The xDEVS crate follows a composition approach: any DEVS model must contain a `Component` structure. This structure contains all the input and output ports of the model. Ports are sorted in a vector to provide a fast iteration during simulation. When adding a new port, we must provide a name. The `Component` structure ensures that port names are unique. The ownership policy of Rust makes it difficult to implement a classic model-to-simulator transformation. Instead, the `Component` structure contains private simulation-related fields necessary to simulate the model (e.g., simulation times of the previous and next state transi-

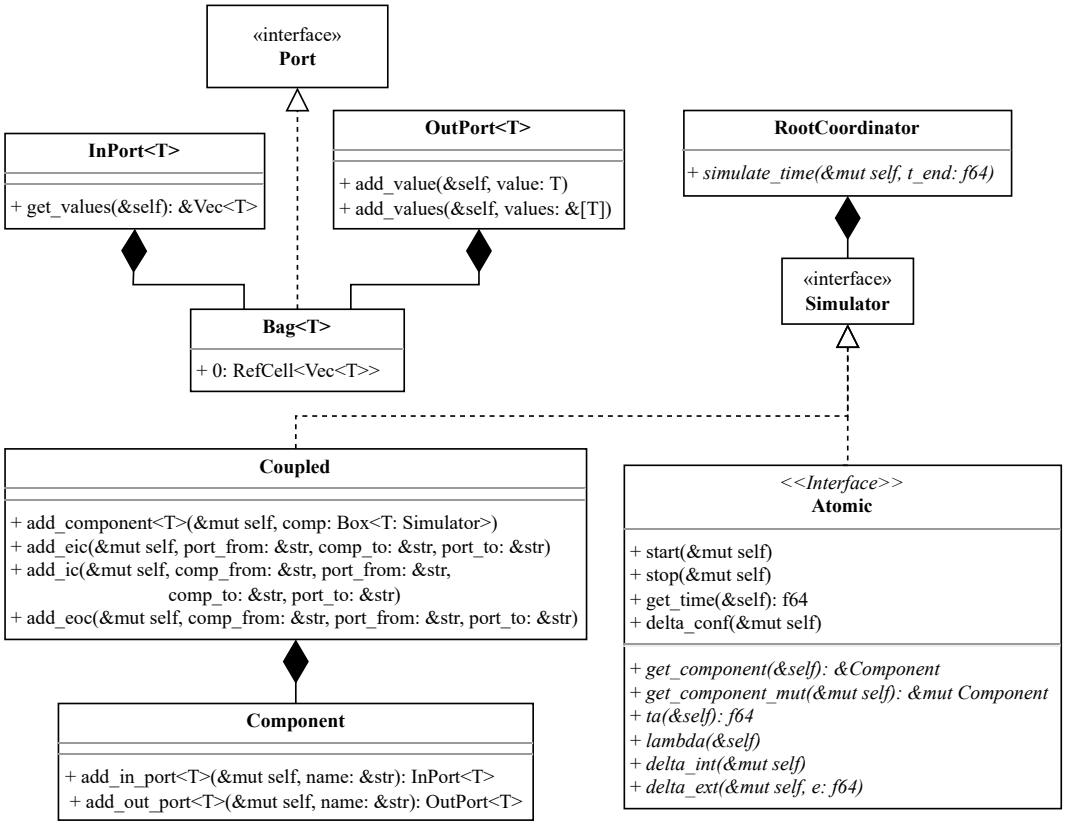


Figure 1: UML diagram of the xDEVS Rust crate.

tions). To enforce a separation of concerns between modeling and simulation, this additional logic is private, and modelers cannot access it.

The xDEVS crate includes the `Atomic` public trait. This trait includes the functions associated to any atomic model (i.e., output function, transition functions, and time advance functions). Additionally, it must implement *getter* methods to allow other parts of the xDEVS crate to access the component contained by the atomic model. Next, we illustrate how to implement the *Generator* atomic model for the Generator Processor Transducer (GPT) model (Zeigler, Muzy, and Kofman 2018):

```rust
use xdevs::modeling::*;
struct Generator {
    component: Component,
    input: InPort<bool>, output: OutPort<usize>,
    period: f64, count: usize, sigma: f64,
}
impl Generator {
    fn new(name: &str, period: f64) -> Self {
        let mut component = Component::new(name);
        let input = component.add_in_port::<bool>("input");
        let output = component.add_out_port::<usize>("output");
        Self { component, input, output, period, count: 0, sigma: 0. }
    }
}
impl Atomic for Generator {
    fn get_component(&self) -> &Component { &self.component }
```

```
17      fn get_component_mut(&mut self) -> &mut Component { &mut self.component }
18      fn lambda(&self) { self.output.add_value(self.count); }
19      fn delta_int(&mut self) {
20          self.count += 1;
21          self.sigma = self.period;
22      }
23      fn delta_ext(&mut self, e: f64) {
24          self.sigma -= e;
25          if self.input.get_values().at(0).unwrap_or(false) {
26              self.sigma = f64::INFINITY;
27          }
28      }
29      fn ta(&self) -> f64 { self.sigma }
30  }
```

The `Generator` atomic model is a structure containing a `Component` and all the necessary input and output ports (lines 3 and 4). Then, we add any additional state parameter required by the model (line 5). In this model, `period` corresponds to the time to wait before creating a new job. The model keeps track of the number of jobs created so far in `count`. Finally, `sigma` is a helper field to simplify the definition of the time advance function. In Rust, complex structures usually implement a `new` associated function to create new elements (lines 8 to 13). When creating a new port, we must provide the desired message type and a unique identifier. For instance, the `input` port is an input port of messages of type `bool` which unique identifier is `"input"` (line 10). The `Generator` model implements the `Atomic` trait in lines 15 to 30. First, we define the *getter* methods that return a reference to the inner `Component` structure. Then, we must implement all the methods associated with atomic models. Note that we do not have to implement the confluent transition function. By default, it is set to $\delta_{conf}(s) = \delta_{ext}(\delta_{int}(s), 0)$.

The xDEVS crate includes the `Coupled` structure for developing coupled models. This structure already contains a `Component` structure. Thus, we can create new ports to the coupled model in the same way as in atomic models. Coupled models allow modelers to add subcomponents and couplings between ports. The `Coupled` structure ensures that its submodels have a unique identifier. Additionally, it forbids duplicate couplings and couplings between ports of different message types. To achieve this behavior, couplings are defined using the name of the components and ports involved. With this methodology, coupled models make sure that i) the submodels with the specified name exist, ii) that the ports exist, iii) that the coupling between these ports has not been defined before, and iv) that the message type of the ports matches. Thus, the Rust version of xDEVS is the first simulator of the xDEVS family able to do all these checks. Next, we illustrate how the GPT coupled model is defined in the xDEVS crate:

```
use xdevs::modeling::*;
struct GPT(Coupled);
impl GPT {
    fn new(name: &str, period: f64, proc_time: f64, observation: f64) -> Self {
        let mut gpt = Coupled::new(name);

        let generator = Generator::new("generator", period);
        let processor = Processor::new("processor", proc_time);
        let transducer = Transducer::new("transducer", observation);
        gpt.add_component(Box::new(generator));
        gpt.add_component(Box::new(processor));
        gpt.add_component(Box::new(transducer));

        gpt.add_ic("generator", "output", "processor", "input");
        gpt.add_ic("generator", "output", "transducer", "input_g");
```

```
16            gpt.add_ic("processor", "output", "transducer", "input_p");
17            gpt.add_ic("transducer", "output", "generator", "input");
18
19            Self(gpt)
20        }
21    }
```

As the GPT model does not have any input nor output port, `GPT` is just a tuple encapsulating a `Coupled` structure (line 2). When creating a new `GPT` model (lines 4 to 20), we first create a new `Coupled` structure. Then, we create all the submodels (lines 7 to 9) and add them to the coupled model using the `add_component` method (lines 10 to 12). Note that models must be first encapsulated into a `Box`. In Rust, `Box` structures are smart pointers that own the value they point to. Finally, we define all the couplings in the model. In this case, we only have internal couplings. Thus, we use the `add_ic` method, specifying the source and output port and the destination and input port. If we want to add external input couplings or external output couplings, we must use the `add_eic` and `add_eoc` methods.

In contrast with other approaches, xDEVS Rust does not include supplementary structures to control the simulation workflow of each component. Instead, DEVS components already include all the necessary elements to run a simulation. `Coupled` structures and any structure implementing the `Atomic` trait automatically implement the `Simulator` trait. This trait is equivalent to the abstract simulator concept, as it defines all the methods and associated functions required to run a simulation. To run a simulation over a model, we must create a `RootCoordinator` structure. `RootCoordinator<M>` wraps a structure of type `M`. `M` must implement the `Simulator` trait. Next, we illustrate how to simulate the `GPT` model:

```
1  use xdevs::simulation::RootCoordinator;
2  fn main() {
3      let gpt = GPT::new("gpt", period, proc_time, observation);
4      let mut simulator = RootCoordinator::new(coupled.0);
5      simulator.simulate_time(f64::INFINITY);
6  }
```

First, we create a `RootCoordinator` in charge of simulating the GPT model. Finally, we start the simulation by executing the `simulate_time` method. We must specify for how long we want the simulation to run. In this example, we want to simulate the model until all its elements passivate.

3 XDEVS C

The C version of xDEVS uses the Portable Operating System Interface (POSIX) and follows the general xDEVS specification published by Risco-Martín et al. (2022b). Figure 2 shows a simplified UML diagram of the xDEVS/C architecture. This architecture is a structured programming version of the OOP xDEVS original design. In xDEVS/C, the top abstract classes such as `Component` or `AbstractSimulator` do not exist because of the absence of polymorphism in C. In the xDEVS/C modeling layer, we find three main structures: `atomic`, `coupled`, and `coupling`, equivalent to the `Atomic`, `Coupled`, and `Coupling` xDEVS general OOP interfaces, respectively. In xDEVS/C, ports are directly mapped to integers, and messages are encapsulated as a list per port of values through the `input` and `output` attributes. The atomic structure has a `state`, which includes the DEVS sigma and phase elements plus other data introduced by the modeler for each specific atomic model. To allow us to define atomic models, the `atomic` C structure stores pointers to the DEVS standard atomic functions that must be implemented for each new atomic model. Other functionalities, like adding components or coupling relations to a coupled model, are implemented as general functions like `list_push_back` (the list interface is not included in the diagram) and `add_coupling`, respectively. Similarly, the simulation layer is composed of two structures: `simulator`

and `coordinator`, with equivalent attributes to the `Coordinator` and `Simulator` classes of the xDEVS general simulation layer defined by Risco-Martín et al. (2022b). The function members of both the `simulator` and `coordinator` instances are driven by general functions with `simulator_` and `coordinator_` prefixes, respectively. As in the modeling layer, the set of simulators inside a coordinator is managed by general lists.

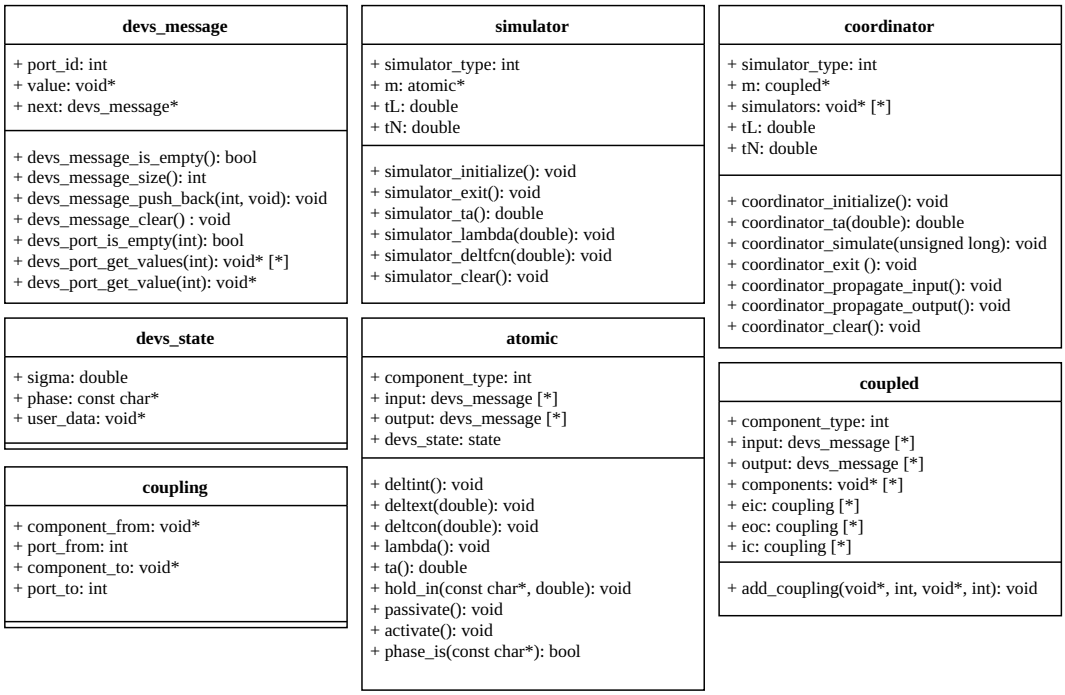


Figure 2: UML diagram of the xDEVS/C implementation.

The following code snippet shows how to implement the *Generator* atomic model for the *GPT* example. The initialization function is called at the beginning of the simulation. After that, the DEVS standard output and transition functions are defined. As can be seen, the output function sends messages through a specific output port, and the transition functions straightforwardly operate over the current state.

```
#include "generator.h"
void generator_init(atomic *self) {
  generator_state *s = (generator_state *)self->state.user_data;
  s->job_next_id = 1;
  hold_in(self, "active", s->period);
}
void generator_lambda(atomic *self) {
  job *j = (job *)malloc(sizeof(job));
  generator_state *s = self->state.user_data;
  j->id = s->job_next_id;
  devs_message_push_back(&(self->output), GENERATOR_OUT, j);
}
void generator_deltint(atomic *self) {
  generator_state *s = self->state.user_data;
  s->job_next_id++;
  hold_in(self, "active", s->period);
}
void generator_deltext(atomic *self, const double e) {
```

```
  resume(self, e);
  passivate(self);
}
```

Next, we show how to build the coupled *GPT* model and run the simulation:

```
#include "../../core/devs.h"
int main(int argc, char *argv[]) {
  coupled* gpt = coupled_new();

  atomic *g = generator_new(1.0);
  atomic *p = processor_new(3.0);
  atomic *t = transducer_new(100.0);
  list_push_back(&(gpt->components), g);
  list_push_back(&(gpt->components), p);
  list_push_back(&(gpt->components), t);

  add_coupling(gpt, g, GENERATOR_OUT, p, PROCESSOR_IN);
  add_coupling(gpt, g, GENERATOR_OUT, t, TRANSDUCER_ARRIVED);
  add_coupling(gpt, p, PROCESSOR_OUT, t, TRANSDUCER_SOLVED);
  add_coupling(gpt, t, TRANSDUCER_OUT, g, GENERATOR_IN);

  coordinator *c = coordinator_new(gpt);
  coordinator_initialize(c);
  coordinator_simulate(c, 1000);
  coordinator_exit(c);
  coordinator_delete(c);

  return 0;
}
```

As can be seen, the procedure is quite similar to the xDEVS OOP APIs. The coupled model does not need particular functionality; thus, atomic models are just created and interconnected. After that, the coordinator is defined with the root-coupled model, and the simulation is launched for 1000 DEVS cycles. xDEVS/C takes exceptional care of deleting all the dynamic memory allocated in the simulation process. The user is also in charge of releasing events from memory when they are no longer needed.

4 PERFORMANCE EVALUATION

Here we compare the performance of the new versions of xDEVS with other state-of-the-art simulators. We compared the simulation speed of the C and Rust implementations of xDEVS with the Java and C++ implementations of xDEVS. These engines are the fastest state-of-the-art implementations of the xDEVS family. Additionally, we included in the comparison the aDEVS simulation tool (Nutaro 2023). The aDEVS engine, written in C++, has been proven to be one of the fastest DEVS-compliant simulators currently available (Cárdenas et al. 2022). To measure the performance of the simulation engines under study, we use the DEVStone benchmarking technique (Glinsky and Wainer 2005). DEVStone presents multiple synthetic models with configurable degrees of complexity. Essentially, it defines a set of coupled models nested recursively. DEVStone presents five complexity configuration parameters: i) *type*, which determines the topology of each layer, ii) *width*, which specifies the atomic models in each layer, iii) *depth*, which corresponds to the number of layers, iv) *internal transition delay*, and v) *external transition delay*. The internal and external transition delays specify the Central Processing Unit (CPU) time that atomic models must spend running computation-intensive operations when their internal and external transition functions are triggered, respec-

tively. LI models present the simplest topology. HI models add internal couplings to the topology. In HO models, the number of ports and input and output couplings increases. Finally, HOmod models present a two-dimensional topology in which the number of atomic models and couplings escalates exponentially with the width and depth of the model.

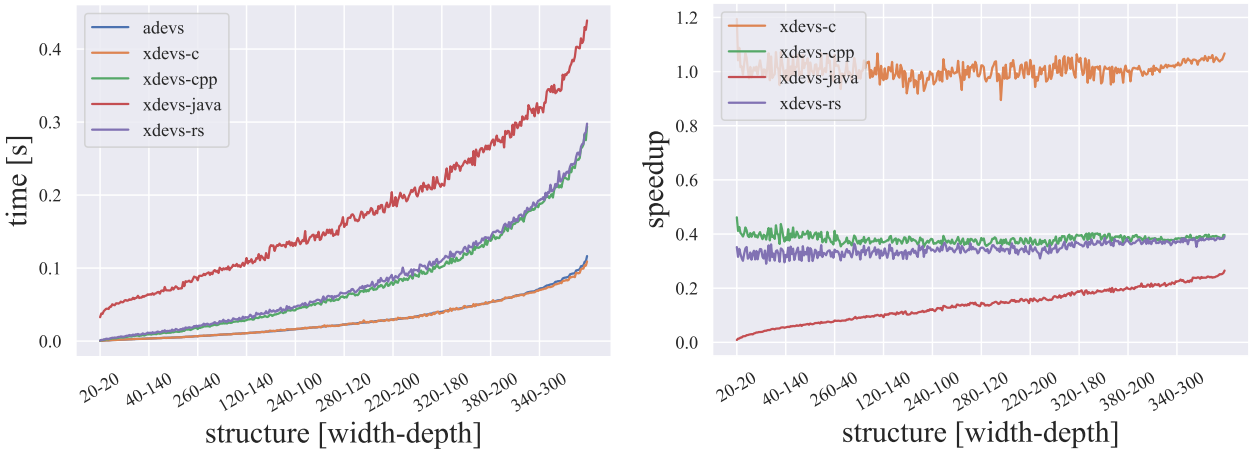
The internal and external transition delays were set to 0. This way, we constrain execution time to create the model and simulate the benchmark. For the LI, HI, ad HO model types, we ran experiments whose width and depth ranged from 20 to 400 at steps of 20. On the other hand, we included HOmod structures with widths and depths ranging from 5 to 50 at step 5. In total, we explored 1,164 different scenarios. We ran each scenario 30 times to assume that the sample mean is normally distributed. The obtained results are shown with a confidence interval of 95%. We simulated the experiments on a MacBook Pro (Retina, 15-inch, Mid 2015) with MacOS 1.6.2, a 2.5 GHz Quad-Core Intel Core i7 CPU, and a 16 GB 1600 MHz DDR3 memory. The DEVStone models were compiled using the Apple clang version 14.0.0 toolchain for C and C++ engines, rustc 1.66.0 for Rust xDEVS, and OpenJDK 19.0.1 for Java. C, C++, and Rust binaries were compiled in release mode.

Table 1 shows a sample of corner cases of the experiments. For each target and model type, we show the deepest, widest, and most complex models. Additionally, we outline the first and second fastest simulation engines for every DEVStone model in dark and light blue, respectively. The aDEVS simulator showed the best results for simpler models. However, as the complexity of the DEVStone models under study increases, other engines manage to outperform aDEVS. In contrast, xDEVS Java presented a high simulation setup overhead. Thus, it obtained the worst performance results for the simplest models. Nevertheless, this overhead becomes increasingly negligible as the model complexity increases. For the most complex configurations, xDEVS C++ is considerably slower than the other engines.

Table 1: Mean simulation time of the engines under study.

DEVStone Model			**Simulation time (s)**				
Type	**Width**	**Depth**	**aDEVS**	**xDEVS C**	**xDEVS C++**	**xDEVS Java**	**xDEVS Rust**
LI	20	400	0.005 ± 0.000	0.005 ± 0.000	0.013 ± 0.000	0.075 ± 0.002	0.017 ± 0.000
	400	20	0.005 ± 0.000	0.005 ± 0.000	0.012 ± 0.000	0.071 ± 0.001	0.014 ± 0.000
	400	400	0.116 ± 0.002	0.109 ± 0.001	0.294 ± 0.005	0.439 ± 0.006	0.298 ± 0.001
HI	20	400	0.018 ± 0.000	0.021 ± 0.000	0.088 ± 0.001	0.143 ± 0.003	0.032 ± 0.000
	400	20	0.263 ± 0.001	0.327 ± 0.002	1.494 ± 0.007	0.542 ± 0.010	0.194 ± 0.001
	400	400	19.305 ± 0.127	8.408 ± 0.036	40.086 ± 0.605	12.745 ± 0.140	6.349 ± 0.013
HO	20	400	0.020 ± 0.000	0.028 ± 0.000	0.112 ± 0.001	0.157 ± 0.007	0.039 ± 0.000
	400	20	0.345 ± 0.006	0.463 ± 0.003	1.857 ± 0.010	0.594 ± 0.006	0.229 ± 0.004
	400	400	20.884 ± 0.148	10.831 ± 0.031	48.327 ± 0.204	13.502 ± 0.084	6.798 ± 0.030
HOmod	5	50	0.010 ± 0.000	0.031 ± 0.000	0.102 ± 0.000	0.118 ± 0.004	0.013 ± 0.000
	50	5	0.129 ± 0.000	0.241 ± 0.001	0.706 ± 0.002	0.342 ± 0.004	0.078 ± 0.000
	50	50	60.630 ± 0.350	50.126 ± 0.121	134.316 ± 1.027	33.372 ± 0.171	15.340 ± 0.069

Figure 3 shows the obtained results for all the LI models in the engines under study. The abscissae arrange the simulated models in increasing mean execution time for the aDEVS simulator. As shown in Figure 3a, the aDEVS and the C version of xDEVS present the best performance, followed by the C++ and Rust versions of xDEVS. Figure 3b represents the speedup of the xDEVS engines over aDEVS. Speedups over 1 imply that the engine is faster than aDEVS. The C version of xDEVS showed a speedup near 1. On the other hand, the speedup of the C++ and Rust versions of xDEVS is close to 0.4. Note that even though the Java simulator shows the worst results, its speedup curve shows a constant improvement as the model complexity increases. Furthermore, the time scale is in the order of milliseconds for LI models. For example, the time



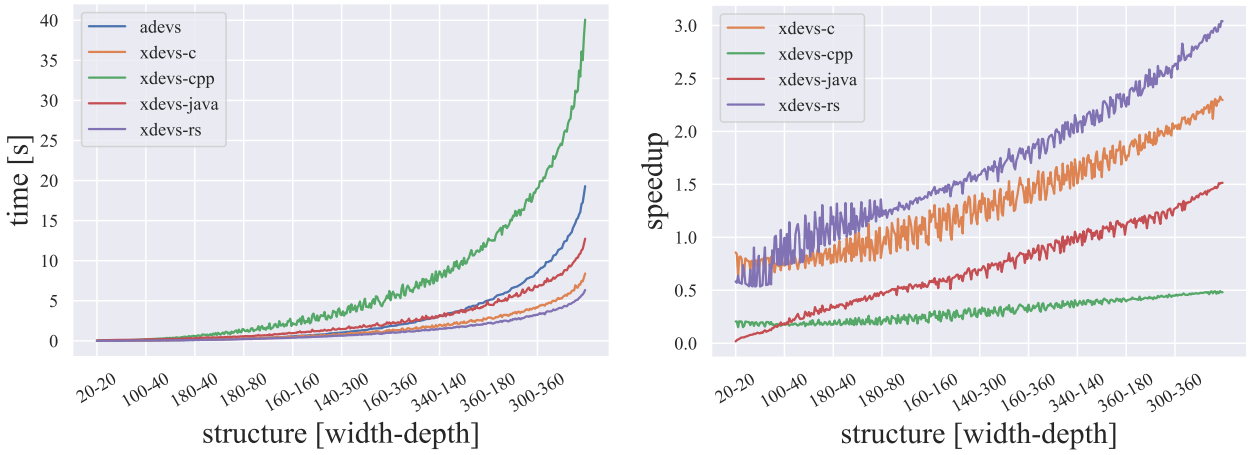
(a) Execution time.

(b) Speedup over aDEVS.

Figure 3: Simulation results for LI models.

difference between xDEVS Java and xDEVS C for the LI model with a width and depth of 400 is 330 ms. While xDEVS C is 4 times faster, this difference may be considered negligible.

As the model topology increases, we experience a significant change in the performance of the engines under study. Figure 4 displays the obtained results for HI models. Now, for the most complex HI model, the difference between the fastest engine (i.e., xDEVS Rust) and the slowest engine (i.e., xDEVS C++) is 33.737 s. This difference is significant enough to be considered when choosing one or another engine. As shown in Figure 4b, aDEVS is still the fastest engine for the simplest configurations. However, the new C and Rust versions of xDEVS outperform it soon. Furthermore, xDEVS Java can outperform aDEVS for the most complex models. The best speedups obtained by the C, Java, and Rust versions of xDEVS are 2.30, 1.51, and 3.04, respectively.

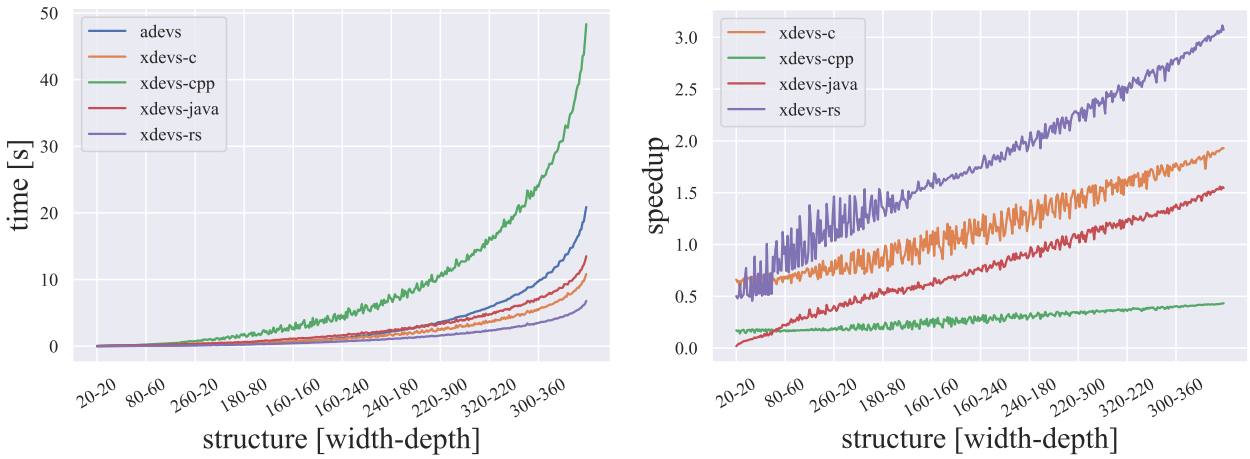


(a) Execution time.

(b) Speedup over aDEVS.

Figure 4: Simulation results for HI models.

This trend repeats for HO models. Figure 5 displays the execution time of the simulation engines under study for DEVStone models of the HO type. Again, the Rust and C versions of xDEVS outperform aDEVS as soon as the model complexity increases. However, the speedup obtained by the C implementation shows



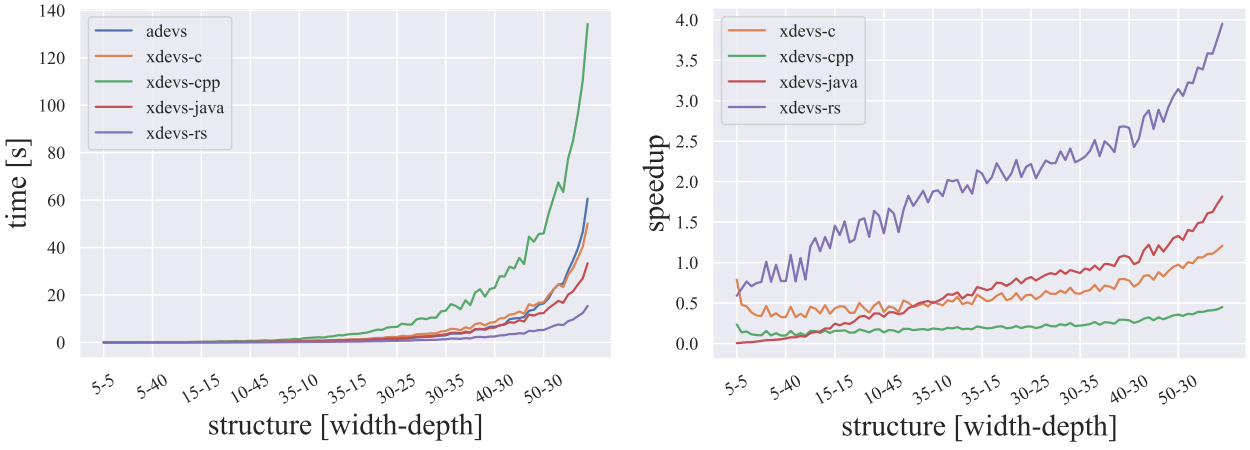
(a) Execution time.

(b) Speedup over aDEVS.

Figure 5: Simulation results for HO models.

a small degradation compared to HI models. The speedup of the other versions of xDEVS is similar to the one obtained in HI models. Note that the performance of xDEVS Java is now very close to xDEVS C. The results obtained by xDEVS Java and Rust are very similar to the ones obtained for HI models. However, xDEVS C presents a slight downgrade compared to HI models, with speedups ranging from 0.63 to 2.33.

Figure 6 shows the execution results for HOmod models. The topology of the HOmod model differs significantly from the rest of the DEVStone model types. Thus, the execution time required by the engines is significantly higher compared to the other topologies. For instance, xDEVS C++ took an average time of 134.316 s to execute HOmod models with a width and depth of 50, while the same engine only required 48.327 s to simulate an HO model with width and depth of 400. The xDEVS Rust simulation engine showed its best performance upgrade for this model, achieving a maximum speedup of 3.95. On the other hand, xDEVS C struggled to follow aDEVS, and only achieved a speedup greater than 1 for the most complex models. xDEVS Java presents good results, with a maximum speedup of 1.82 for the most complex model.



(a) Execution time.

(b) Speedup over aDEVS.

Figure 6: Simulation results for HOmod models.

We illustrate the percentage of execution time spent on every DEVStone type for the engines under study in Figure 7. In general, aDEVS, xDEVS C, and xDEVS C++ seem to be the simulation engines that are more sensitive to the model complexity, as they spend more than 60% of the overall time simulation the HO and HOmod models. This is especially notable for xDEVS C, which spent 24% of the total time executing the HOmod models. In contrast, xDEVS Java and xDEVS Rust required less than 60% to execute the same model set. On the other hand, these two engines are the only ones that present a significant percentage of execution time for LI models. This information, together with the speedup trends shown in Figure 3b, suggest that these simulators present a higher simulation setup overhead compared to the other engines. This overhead becomes negligible as the simulation time increases.

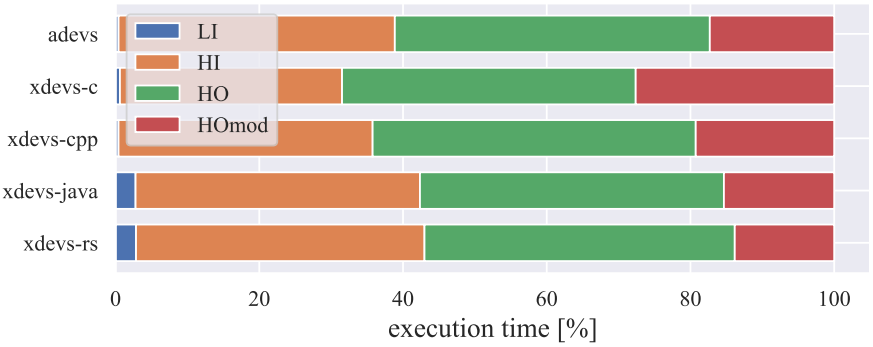


Figure 7: Execution time percentage per DEVStone model type.

5 CONCLUSIONS AND FUTURE WORK

Applying M&S techniques can significantly help during the design and study of complex systems. Furthermore, approaches such as HIL modeling or DTs development integrate M&S tools for the implementation, operation, and automation of modern CPSs. Here we presented xDEVS/C and xDEVS/Rust, two new implementations of the xDEVS framework written in programming languages that are suitable for developing embedded devices. These tools show a significant performance upgrade compared to other xDEVS versions. To illustrate their benefits, we compared the execution time of 1,164 different DEVStone synthetic models for these new implementations and other simulators including aDEVS, one of the fastest simulators up to date. Results show that xDEVS/Rust and xDEVS/C outperform state-of-the-art alternatives for complex models. Moreover, their performance speedup increases linearly with the model complexity in most cases. As future work, we plan to develop Real-Time (RT) applications for embedded devices using the presented frameworks. Additionally, we are currently working on a lightweight communication interface for interconnecting xDEVS RT applications regardless of their underlying programming language or platform.

ACKNOWLEDGEMENTS

This work has been partially supported by the Spanish Ministry of Science and Innovation, under grants PID2019-110866RB-I00/AEI/10.13039/501100011033 (managed by the Spanish State Research Agency), and TED2021-130123B-I00 (under the Ecological and Digital Transition Program).

REFERENCES

Bolduc, J.-S., and H. Vangheluwe. 2002. “A modeling and simulation package for classic hierarchical DEVS”. *MSDL, School of Computer McGill University, Tech. Rep.*

Cárdenas, R., K. Henares, P. Arroba, J. L. Risco-Martín, and G. A. Wainer. 2022. "The DEVStone Metric: Performance Analysis of DEVS Simulation Engines". *ACM Trans. Model. Comput. Simul.* vol. 32 (3), pp. 21:1–21:20.

Cárdenas, R., and G. Wainer. 2022, December. "Asymmetric Cell-DEVS models with the Cadmium simulator". *Sim. Modelling Practice and Theory* vol. 121, pp. 102649.

Glinsky, E., and G. Wainer. 2005. "DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments". In *Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pp. 265–272. Montreal, Quebec, Canada, IEEE, IEEE.

Moallemi, M., and G. Wainer. 2010. "Designing an Interface for Real-Time and Embedded DEVS". In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10. San Diego, CA, USA, Society for Computer Simulation International.

Nutaro, J. 2023. "aDEVS (a Discrete EVent System Simulator)". https://web.ornl.gov/~nutarojj/adevs/.

Quesnel, G., R. Duboz, and E. Ramat. 2009, April. "The Virtual Laboratory Environment – An operational framework for multi-modelling, simulation and analysis of complex dynamical systems". *Sim. Modelling Practice and Theory* vol. 17, pp. 641–653.

Risco-Martín, J. L., K. Henares, S. Mittal, L. F. Almendras, and K. Olcoz. 2022a. "A Unified Cloud-Enabled Discrete Event Parallel and Distributed Simulation Architecture". *Sim. Modelling Practice and Theory*.

Risco-Martín, J. L., S. Mittal, J. C. Fabero, P. Malagón, and J. L. Ayala. 2016. "Real-time Hardware/Software Co-Design Using DEVS-based Transparent M&S Framework". In *Proceedings of the 2016 Summer Simulation Multiconference (SummerSim 2016)*.

Risco-Martín, J. L., S. Mittal, K. Henares, R. Cardenas, and P. Arroba. 2022b. "xDEVS: A toolkit for interoperable modeling and simulation of formal discrete event systems". *Softw: Pract and Exper*.

Risco-Martín, J. L., S. Mittal, K. Henares, and R. Cárdenas. 2014. "xDEVS: A cross-platform discrete event system simulator". https://github.com/iscar-ucm/xdevs.

Seo, C., B. P. Zeigler, R. Coop, and D. Kim. 2013. "DEVS modeling and simulation methodology with MS4 Me software tool.". In *SpringSim (TMS-DEVS)*, pp. 33. SpringSim (TMS-DEVS).

Zeigler, B. P., A. Muzy, and E. Kofman. 2018. *Theory of modeling and simulation: discrete event & iterative system computational foundations*. Academic press.

AUTHOR BIOGRAPHIES

ROMÁN CÁRDENAS received the M.Sc. Degree in Telecommunication Engineering in 2019 from Universidad Politécnica de Madrid (UPM), Spain, where he pursues a Ph.D. in Electronic Systems Engineering in Cotutelle with Carleton University (CU). 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 Assistant Professor at Universidad Politécnica de Madrid (UPM), Spain. She received her Ph.D. Degree in Telecommunication Engineering from UPM in 2017. Her research interests include energy and thermal-aware modeling and optimization of data centers. 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.