Received: 3 November 2021 | Revised: 22 September 2022 | Accepted: 18 October 2022

DOI: 10.1002/spe.3168

RESEARCH ARTICLE

xDEVS: A toolkit for interoperable modeling and simulation of formal discrete event systems

José L. Risco-Martín[1,2] | **Saurabh Mittal**[3] | **Kevin Henares**[1] | **Román Cardenas**[4,5] | **Patricia Arroba**[2,4]

[1]Department of Computer Science and Automation, Complutense University, Madrid, Spain

[2]Center for Computational Simulation, Universidad Politécnica de Madrid, Madrid, Spain

[3]The MITRE Corporation, McLean, Virginia, USA

[4]Integrated Systems Laboratory, Universidad Politécnica de Madrid, Madrid, Spain

[5]Advanced Real-Time Simulation Laboratory, Carleton University, Ontario, Canada

Correspondence
José L. Risco-Martín, Department of Computer Architecture and Automation, Complutense University, Madrid Spain.
Email: jlrisco@ucm.es

Abstract

Employing Modeling and Simulation (M&S) extensively to analyze and develop complex systems is the norm today. The use of robust M&S formalisms and rigorous methodologies is essential to deal with complexity. Among them, the Discrete Event System Specification (DEVS) provides a solid framework for modeling structural, behavior and information aspects of any complex system. This gives several advantages to analyze and design complex systems: completeness, verifiability, extensibility, and maintainability. DEVS formalism has been implemented in many programming languages and executable on multiple platforms. In this paper, we describe the features of an M&S framework called xDEVS that builds upon the prevalent DEVS Application Programming Interface (API) for both modeling and simulation layers, promoting interoperability between the existing platform-specific (C++, Java, Python) DEVS implementations. Additionally, the framework can simulate the same model using sequential, parallel, or distributed architectures. The M&S engine has been reinforced with several strategies to improve performance, as well as tools to perform model analysis and verification. Finally, xDEVS also facilitates systems engineers to apply the vision of model-based systems engineering (MBSE), model-driven engineering (MDE), and model-driven systems engineering (MDSE) paradigms. We highlight the features of the proposed xDEVS framework with multiple examples and case studies illustrating the rigor and diversity of application domains it can support.

KEYWORDS

DEVS formalism, discrete events, modeling and simulation tools, parallel simulation, simulation performance

Abbreviations: API, Application Programming Interface; DEVS, Discrete Event System Specification; DEVSML, DEVS Modeling Language; DSL, Domain Specific Language; DUNIP DEVS, Unified Process.; M&S, Modeling and Simulation; MBSE, Model-Based Systems Engineering; MDE, Model-Driven Engineering; MDSE, Model-Driven Systems Engineering; PSM, Platform Specific Model; V&V, Verification & Validation .

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2022 The Authors. *Software: Practice and Experience* published by John Wiley & Sons Ltd.

Check for updates

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

1 | INTRODUCTION

Over the last few decades, Modeling and Simulation (M&S) techniques have been used to analyze and develop complex systems in various application fields. Several approaches have been developed, including formalisms as Petri Nets, Timed Automata,[1] and Discrete Event System Specification (DEVS) formalism. DEVS is a general formalism for discrete event system modeling based on mathematical Set theory.[2] The DEVS formalism provides a framework for modeling structural, behavioral and information aspects of any complex system. This gives several advantages to analyze and design complex systems: completeness, verifiability, extensibility, and maintainability. Once a system is described in terms of the DEVS theory, it can be easily implemented in a computational environment across multiple platforms. The Classic DEVS formalism was first published in 1976, followed by other variants like Discrete Event and Differential Equation System Specification (DEV&DESS, 1990), Dynamic Structure DEVS (DynDEVS, 1995), Parallel DEVS (PDEVS, 1996), Generalized DEVS (GDEVS, 2001), or Hybrid DEVS (XDEVS, 2022).[2,3] Currently, PDEVS is the prevalent DEVS, implemented in numerous libraries (e.g., C++, Java, Python). In the following, unless it is explicitly noted, the use of DEVS implies PDEVS.

The DEVS formalism was invented to bring coherence and unify the field of discrete-event M&S and Systems theory, using a rigorous mathematics and a systems theoretical framework. Despite this underlying coherence and unity, DEVS remains strongly aligned with the associated platform-dependent simulation engine, implemented using programming languages like C++, Python, Java, and so forth. There have been efforts to develop DEVS Domain Specific Language (DSLs) such as DEVS Modeling Language (DEVSML) that offer promise in bringing focus back to modular complex systems engineering and illustrate how a DEVS platform independent model could be made executable with different platform specific model implementations.[4,5] DEVSML is by far the only DEVS Domain Specific Language (DSL) (implemented in DEVSML Studio)[6] that integrates more than one DEVS implementations. There exist many libraries and tools within the M&S community that express DEVS models such as aDEVS,[7] Cadmium,[8,9] PyPDEVS,[10] DEVS-JAVA,[11] MS4Me,[12] VLE[13] and so forth. However, this variety presents a serious difficulty in sharing models, especially for model reuse and model composition for large complex systems M&S efforts involving paradigms like Model-Based Systems Engineering (MBSE), Model-Driven Engineering (MDE), and Model-Driven Systems Engineering (MDSE).[4,14] As a result, a widely accepted framework is more necessary than ever. Not only to guarantee a universal specification of models and simulators in a computational environment but also to allow the system designer to address fundamental issues such as verification, validation, scalability, reliability and so forth, across a family of models developed using DEVS paradigm.

To tackle this issue, we formally introduce **xDEVS**, a cross-platform, object-oriented, M&S framework developed at the Department of Computer Architecture and Automation at the Complutense University of Madrid. The motivation behind xDEVS framework is to unify the existing DEVS M&S frameworks, and provide optimal performance and deployment of models in parallel and distributed computing architectures. Although xDEVS started as a Java implementation in 2014 and has been integrated with DEVSML Studio, we have progressively released several upgrades, which brings us to the following general contributions withing this article:

1. Consolidation of all the recent advances like parallel and distributed simulation, integration of a platform-independent specification language, called DEVSML,[5] flattening models and so forth.
2. Incorporation of three different implementations with equivalent DEVS interfaces, the original Java implementation (xDEVS/Java), a C++ implementation (xDEVS/C++), and a Python implementation (xDEVS/Python); Additionally, xDEVS incorporates wrappers for integrating DEVS models implemented with other simulation engines, which facilitates DEVS-to-DEVS and DEVS-to-non-DEVS interoperability;
3. Addition of a constraints definition syntax that allows checking arithmetic relations among the port outputs of a DEVS model, and a unit testing tool that allows obtaining state trajectories easily, facilitating model verification.
4. Utilities to facilitate the deployment of MBSE, MDE, and MDSE approaches for building executable architectures.

As technical contributions, we aim to offer a common Application Programming Interface (API) for developing DEVS models in several of the most popular programming languages and provide a renewed push to DEVS Standardization efforts within the DEVS and M&S as a Service (MSaaS) communities. The development of common API is implemented through various wrappers. The incorporation of wrappers allow xDEVS to interact with atomic components defined

TABLE 1 Comparison between some characteristics of several well-known DEVS-based simulation engine Application Programming Interface (APIs)

	aDEVS	Cadmium	PyPDEVS	DEVSJAVA	MS4Me	VLE	xDEVS
Parallel DEVS formalism	✓	✓	✓	✓	✓	✓	✓
DEVS wrappers	✓	✓	✓				✓
Model flattening		✓	✓		✓		✓
Sequential simulation	✓	✓	✓	✓	✓	✓	✓
Real-time simulation		✓	✓	✓	✓		✓
Parallel simulation			✓[a]		✓	✓[a]	✓
Distributed simulation			✓[a]		✓[a]	✓[a]	✓
Profiling of the simulation							✓
Unit testing/debugging					✓	✓	✓

[a]In these cases, the original model requires some extra modifications.

for other simulators (developed in C++, Java, or Python). Further, the xDEVS distributed architecture allows us to communicate seamlessly within the three xDEVS platform-specific simulation engines: xDEVS/C++, xDEVS/Java, and xDEVS/Python. This provides full interoperability between different platform-specific DEVS M&S engines. This interoperability effort alleviates the heterogeneity of DEVS implementations and provides a way to compose and execute large scale models developed in different platform-specific DEVS-based implementations. Around this integral DEVS M&S engine, we provide additional utilities to improve performance and alleviate the maintainability and scalability for creating complex models. Table 1 summarizes the main contributions related to API functionalities. The feature list in the first column is a selective list focusing on the advanced simulation and API capabilities rather than user-centric capabilities like Graphical User Interfaces and so forth.

xDEVS is available under the GNU General Public License v3.0 license and provides support for specifying and running classic, parallel, real-time, and distributed DEVS simulations.

To show the complete xDEVS M&S framework, the paper is organized as follows. Section 2 offers details about the DEVS formalism. In Section 3, a detailed explanation of the architecture of the xDEVS M&S framework is provided, as well as all the features that facilitates the conception, deployment, verification and validation of discrete event system models. Section 4 describes the benefits of xDEVS in supporting the development of a standardized M&S framework. Section 5 shows a performance comparison of the different xDEVS platform-specific implementations. Several case studies using the xDEVS framework from diverse application domains are presented in Section 6. Finally, we present some discussion in Section 7 and conclusions in Section 8.

2 | THE DEVS FORMALISM

DEVS is a general formalism for specifying discrete event complex dynamical systems and is based on mathematical set theory. As stated earlier, we will be using Parallel DEVS implementation when we refer DEVS in this paper (see Reference 2, section 4.3). DEVS enables the behavior representation of a system by three mathematical sets and five characteristic functions: input set (X), output set (Y), state set (S), external transition function (δ_{ext}), internal transition function (δ_{int}), confluent function (δ_{con}), output function (λ), and time advance function (ta). It enables the structure representation of a system by explicit coupling and containment relationships. There are two types of DEVS models: atomic and coupled. The behavioral aspect of DEVS formalism is handled by the Atomic model and the structural aspect of a DEVS formalism is handled by the Coupled models. The overall complex system's behavior emerges when a coupled model is simulated.

Atomic models are directly expressed through the three mathematical sets and five characteristic functions as specified above. Atomic DEVS models process input events based on their model's current state and qualifying conditions, generates output events and transition to the next state. An atomic model is defined by Equation 1:

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

$$A = \langle X, Y, S, \delta_{\text{ext}}, \delta_{\text{int}}, \delta_{\text{con}}, \lambda, ta \rangle, \tag{1}$$

where:

- X is the set of input events, described in terms of pairs port-value: $\{p \in IPorts, v \in X_p\}$ is the set of input ports and values.
- Y is the set of outputs, also described in terms of pairs port-value: $\{p \in OPorts, v \in Y_p\}$ is the set of output ports and values.
- S is the set of sequential states.
- $\delta_{\text{ext}} : Q \times X^b \rightarrow S$ is the external transition function. It is automatically executed when an external event arrives to one of the input ports, changing the current state if needed.

  - $Q = (s, e) | s \in S, 0 \leq e \leq ta(s)$ is the total state set, where e is the time elapsed since the last transition.
  - X^b is the set of bags over elements in X.

- $\delta_{\text{int}} : S \rightarrow S$ is the internal transition function. It is executed right after the output (λ) function and is used to change the state S.
- $\delta_{\text{con}} : Q \times X^b \rightarrow S$ is the confluent function, subject to $\delta_{\text{con}}(s, ta(s), \emptyset) = \delta_{\text{int}}(s)$. This transition decides the next state in cases of collision between external and internal events, that is, an external event is received and elapsed time equals time-advance. Typically, $\delta_{\text{con}}(s, ta(s), x) = \delta_{\text{ext}}(\delta_{\text{int}}(s), 0, x)$.
- $\lambda : S \rightarrow Y^b$ is the output function. Y^b is the set of bags over elements in Y. When the time elapsed since the last output function is equal to $ta(s)$, then λ is automatically executed.
- $ta(s) : S \rightarrow \Re_0^+ \cup \infty$ is the time advance function.

Coupled models are the aggregation/composition of two or more atomic and/or coupled models connected by explicit couplings. The formal definition of a coupled model is described in Equation (2):

$$M = \langle X, Y, C, EIC, EOC, IC \rangle, \tag{2}$$

where:

- X is the set of inputs described in terms of pairs port-value: $\{p \in IPorts, v \in X_p\}$.
- Y is the set of outputs, also described in terms of pairs port-value: $\{p \in OPorts, v \in Y_p\}$.
- C is a set of DEVS component models (atomic or coupled). Note that C makes this definition recursive. The original DEVS formalism uses two sets $\{D, M_{d \in D}\}$ to include atomic and coupled components. We use this equivalent notation because it directly corresponds to the software implementation detailed below.
- EIC is the external input coupling relation, from external inputs of M to component inputs of C.
- EOC is the external output coupling relation, from component outputs of C to external outputs of M.
- IC is the internal coupling relation, from component outputs of $c_i \in C$ to component outputs of $c_j \in C$, provided that $i \neq j$.

Given the recursive definition of M, a coupled model can itself be a part of a component in a larger coupled model system giving rise to a hierarchical DEVS model construction, as Figure 1 shows.

3 | XDEVS ARCHITECTURE

xDEVS has its basis in defining a universal DEVS API for both modeling and the simulation levels. The API is realized in three widely used object-oriented programming languages: C++, Java, and Python. The repository is made available through an API project at Reference 15, where the project has three principal branches (named C++, Java, and Python).

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 1 Hierarchical DEVS model with atomic and coupled components and associated couplings

This framework allows the specification and execution of DEVS models. In the following sections, we describe the xDEVS modeling and simulation application programming interface (APIs) and describe its main features, showing some useful examples.

3.1 | Application programming interface

xDEVS, based on the DEVS formalism, has a clear separation between the modeling and simulation layers. A class diagram showing the relationship between these modeling and simulation layers is shown in Figure 2.

DEVS models in xDEVS are created using two main components. `Atomic` components define the behavior of the system. `Coupled` components contains other `Atomic` and `Coupled` components, creating a model hierarchy. Both of them have `Ports`, that represent input/output information points. To link two components of the model a `Coupling` can be created, selecting the source and destination `Ports`. The information of `Couplings` is contained in the `Coupled` elements that wrap the ports to be linked.

The simulation layer is based on the concept of the Abstract Simulator. Following this concept, we divide the simulation entities in `Simulators` and `Coordinators` both inheriting from the `Abstract Simulator`. A `Simulator` component instance maps to an `Atomic` component. Each `Coordinator` component instance synchronizes with the corresponding children `Simulators` and `Coordinators`, to mirror the hierarchical Coupled model. This results in an equivalent hierarchy to the one described for the modeling layer.

3.1.1 | Modeling layer

This section provides some implementation examples in the different xDEVS branches. It presents some basic components implementations, both atomic and coupled, and details the salient features of various Application Programming Interface (APIs).

Component

This is the base component class that creates a black-box with a name, inports and outports. The API is shown* in Listing 1.

```
interface IComponent {
   void initialize();
   void exit();
   void addInPort(Port port);
   void addOutPort(Port port);
}
```

Listing 1: xDEVS Component API

*For simplicity, getters and setters are omitted.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

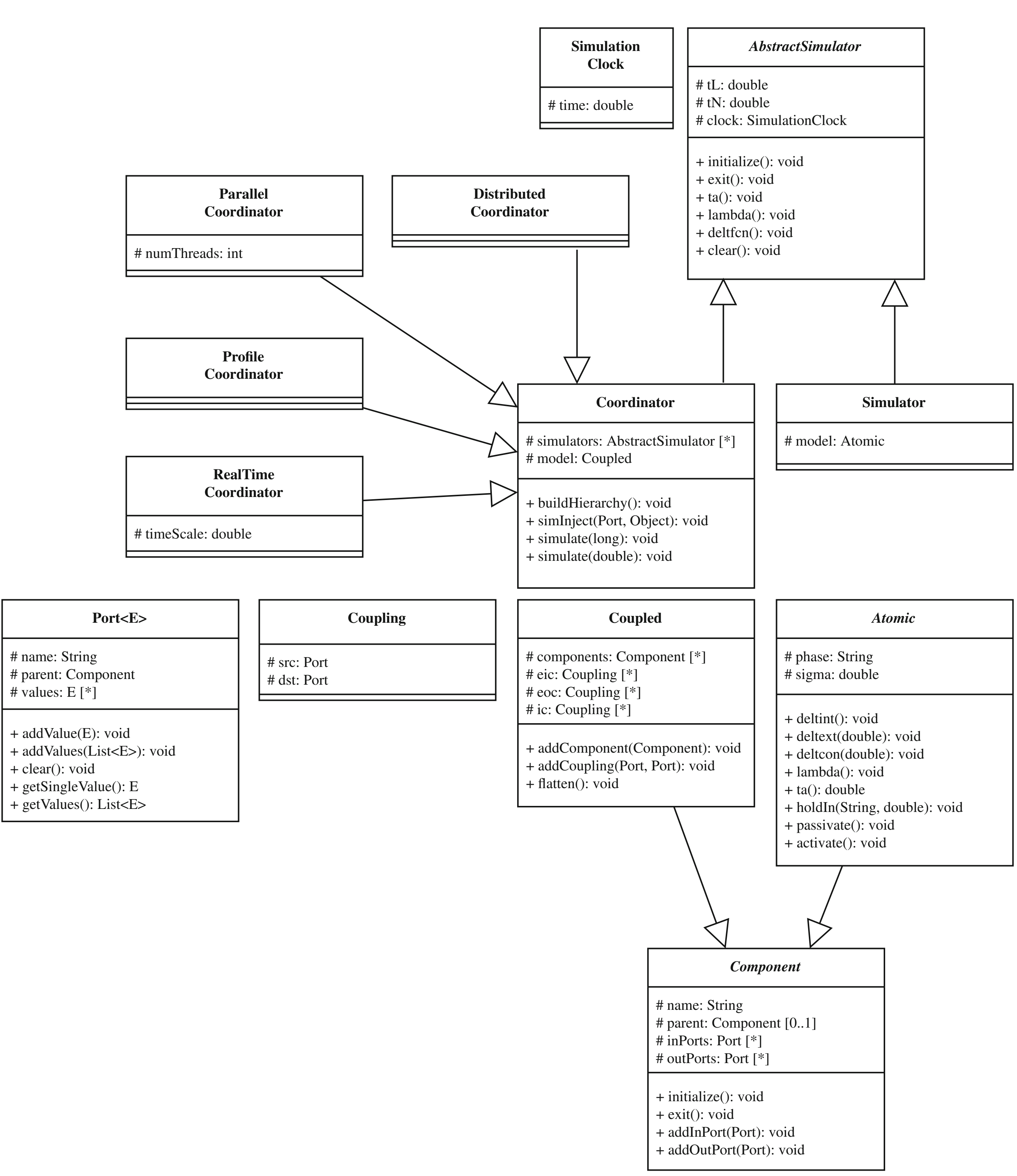


FIGURE 2 Class diagram of the xDEVS architecture

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The main member functions included in this class are the following:

- Initialization function (*initialize*): this is executed when the component is initialized, right before the simulation begins.
- Finalization function (*exit*): this is executed when the simulation is finished, to release computing resources.
- Finally, *addInPort* and *addOutPort* are auxiliary functions to add input and output ports, respectively.

Atomic components

Recall that the behavior of the DEVS models is encapsulated in the atomic components. Each atomic model has a *phase*, which is a semantic label capturing the current *state* of the component, and a *sigma*, which is the duration of time remaining in the current phase. The atomic components base their operation on events. An event can be a time-based event, a variable update input or an external input event. The response to each of these events is defined by implementing specific abstract methods of the `Atomic` class. The main events controlled by these constructs are the following:

- External event (*deltext*): it is activated when one or more messages arrive at any of the input ports of an atomic component.
- Internal event (*deltint*): it is triggered after the lifetime of the present phase has been consumed (specified by *sigma*).
- Confluent event (*deltcon*): it is triggered when both the internal and external events are scheduled for a specific simulation time instant. The corresponding method is already implemented with the most common expected behavior (execute the internal event method first, and then the external event method). Hence, it only has to be implemented when an alternative behavior is expected, for example, selection of external input first then the internal event or ignoring an external event altogether.
- Output function (*lambda*): it is activated before every internal and confluent state transition. All the output values have to be sent through the output ports in this function.
- Time advance function (*ta*): it is executed internally when the model prescribes a time-to-live for every phase. It returns the sigma of a phase. Experienced DEVS developers mostly do not implement this as they tend to set the sigma based on specific events in the *deltint*, *deltext*, or *deltcon* functions.

In addition, there are some utility functions that are non-essential for DEVS formalism but provide ease of use of basic DEVS functionality:

- *holdIn(String phase, double sigma)*: This method holds the model in the provided *phase* name for *sigma* duration.
- *passivate()*: This methods holds the phase as *passive* for *sigma = infinity* duration.
- *activate()*: This method holds the phase as *active* for *sigma = 0* duration.

The xDEVS modeling API is shown in Listing 2.

```
interface IAtomic extends IComponent {
   // DEVS methods:
   void deltint();
   void deltext(double e);
   void deltcon(double e);
   void lambda();
   double ta();

   // Utility methods
   void holdIn(String phase, double sigma);
   void passivate();
   void activate();
}
```

Listing 2: xDEVS Component API

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Coupled components

Coupled components encapsulate other atomic and coupled components and specify the couplings among them. This grouping enables development of a hierarchical system. The API for Coupled components deals with developing the structure of the hierarchical system. It is shown in Listing 3.

```
interface ICoupled extends IComponent {
   void addComponent(IComponent component);
   void addCoupling(Port outport, Port inport);
   void flatten();
}
```

Listing 3: xDEVS Modeling Coupled API

The modeling API for atomic and coupled components described above is implemented in three languages (Java, C++, and Python) in the current xDEVS distribution and an implementation example is available in Appendix A.

Realizing the DEVS formalism with the xDEVS API

Given the object-oriented nature of the xDEVS API, there are some *structural* constructs that are adopted to implement the DEVS formalism within the Object-oriented paradigm. The class diagram depicted in Figure 2:

- The atomic state set S is defined in the form of *phase* names and object attributes. The phase names populate the state set but the quantification of state is done through manipulation of the object attribute variables. As a result, the internal transition function for instance, does not need arguments: $\delta_{\text{int}}()$, and not $\delta_{\text{int}}(S)$ as in the DEVS formalism. The object attributes are updated in $\delta_{\text{int}}()$ or $\delta_{\text{ext}}()$ functions per modeler's design, in addition to tracking the phase.
- Input and output events are directly stored at ports. Each `Component` object, Coupled or Atomic, has a set of input and output ports as class attributes, and each port has a linked list of values . The set of values stored at all the input ports represent the X^b DEVS bag, whereas the set of values at the output ports represent the Y^b DEVS bag that contains values without any order of arrival. Thus, to send a value through the output function, the user only has to add a message value to the corresponding output port; consequently, as the state is intrinsic to the atomic class, the output function does not need to receive arguments nor return values, it is $\lambda()$ instead of $Y^b \leftarrow \lambda(S)$. Equivalently, the external transition function does not have to receive the set of inputs, or even receive/return the state, according to the previous point. This is accomplished with reading each input port and check whether there are events stored at them or not, i.e., xDEVS uses $\delta_{\text{ext}}(e)$ instead of $S \leftarrow \delta_{\text{ext}}(S, e, X^b)$. This is a very important aspect of DEVS formalism that encapsulates complex state of a modular atomic component as a black box whose behavior is defined strictly through I/O message sets and the time intervals that are needed to transform input message temporal behavior into output message temporal behavior. Refer to Zeigler et al.[2] for a more elaborate description.
- As can be seen in Figure 2, ports include a reference back to their parent components (coupled or atomic). This way, when defining a coupling relation, only the ports are needed to build the connection (see the `addCoupling` function in Listing 3. This is more a design feature rather than a formalism specification and assumes implementation of unique port names. Some DEVS implementations (e.g., DEVSJAVA) include the associated DEVS component with the port name.
- There exist two non-DEVS functions: `initialize` and `exit`. These functions are executed right before the simulation begins and right after the simulation ends, respectively. Both have been designed to perform some computational tasks that are outside the DEVS formalism, like opening files before the simulation, closing files after the simulation, creation and destruction of threads or containers and so forth.

3.1.2 | Simulation layer

This layer defines all the simulation entities necessary to simulate a DEVS model composed of atomic and coupled components. Although the modeling and simulation layers communicate with each other at simulation execution, it is worth

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

noting that they are completely independent and a categorical feature of any DEVS-compliant architecture. In this way, the simulation entities only keep references of the models for exchanging the appropriate temporal events and propagate the outputs of the components through the couplings of the model. This categorical separation provides horizontal scaling of modeling and simulation layers.

The simulation layer is based on the Abstract Simulator defined by Zeigler and Chow.[16] It contains two types of entities: *Simulator* and *Coordinator*. A `Simulator` is attached to an atomic model and is in charge of controlling the atomic model's behavior through the modeling API. The coordinators are attached to coupled models and control the management of all their component child simulators (and coordinators in a hierarchical coupled model).

The Abstract Simulator API is shown in Listing 4. This corresponds to the DEVS atomic model and is responsible for atomic model's dynamic behavior.

```
interface ISimulator {
   void initialize();
   void exit();
   void ta();
   void lambda();
   void deltfcn();
   void clear();
}
```

Listing 4: Abstract DEVS Simulator API

Accordingly, the Coordinator API deals with executing a DEVS coupled model over time. The Coordinator API holds a reference to a hierarchical coupled model and their associated simulators or any coordinators per the hierarchy. The API is shown in Listing 6.

```
interface ICoordinator extends ISimulator {
   void buildHierarchy();
   void simInject(Port port, Object value);
   void simulate(long iterations);
   void simulate(double time);
}
```

Listing 5: DEVS Coordinator API

Based on different use cases for simulation deployment, xDEVS includes different types of coordinators:

- Coordinator: the default coordinator, runs sequential simulations in virtual time.
- Parallel coordinator (*CoordinatorParallel* in Figure 2): This coordinator behaves as the sequential coordinator but uses multiple concurrent threads and is apt for multi-core machines, which are ubiquitous today. This coordinator distributes the component simulators to different cores on the same local computer for concurrent execution.
- Real-time coordinator (*RTCentralCoordinator* in Figure 2): This coordinator synchronizes with the operating system clock to run the simulators in wall-clock time.
- Distributed coordinator (*CoordinatorDistributed* in Figure 2): this coordinator deploys component simulators on machines separated over a physical network, across Local Area Network (LAN) or Wide Area Network (WAN) such as World Wide Web.
- Profile coordinator (*CoordinatorProfile* in Figure 2): This coordinator can be registered during a simulation session and saves all the metrics in an external file. Metrics include the number of events triggered, number of times each DEVS function is executed and time consumed.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 Features included in the xDEVS simulation engine

	xDEVS/C++	xDEVS/Java	xDEVS/Python
Parallel DEVS formalism	✓	✓	✓
DEVS wrappers	✓	✓	✓
Model flattening	✓	✓	✓
Sequential simulation	✓	✓	✓
Real-time simulation	✓	✓	✓
Distributed simulation	✓	✓	✓
Parallel simulation	✓	✓	
Profiling of the simulation	✓	✓	✓
Memory-shared ports[18]			✓
Constraints definition[19]	✓		✓
Unit testing[20]		✓	

A typical execution of a DEVS coupled model by the Coordinator API as implemented in Java, C++ and Python is provided in Appendix B.

3.2 Main features

The latest xDEVS distribution implements the above application programming interface (APIs) in Java, C++ and Python. Table 2 shows all the available features. When a new feature is envisioned and scheduled for development, usually triggered by different projects requirements, it is first implemented and tested in a particular language and then progressively translated to all the xDEVS branches. The API for enhanced utility is accordingly updated.

As stated earlier, the simulation engine follows the parallel DEVS formalism. Each branch implements wrappers for different non-xDEVS frameworks. This feature allows us to perform a heterogeneous composition of models inside the same programming language, for example, with aDEVS, DEVSJAVA, or PyPDEVS. To improve performance, and to easily deploy distributed simulations, models are flattened by default.

The three branches offer sequential, real-time, and distributed simulation executions. Distributed simulations allow us to compose simulations combining models of the different branches using the Wrapper design pattern and deploy them over a computer network. Consequently, the DEVS-to-DEVS interoperability paradigm is fully addressed with respect to bridging the simulation libraries of aDEVS, DEVSJAVA and PyPDEVS. The parallel simulation is not currently possible in xDEVS/Python. Additionally, xDEVS allows the profiling of simulations,[17] registering the wall-clock time used by all the DEVS characteristic functions. xDEVS/Python also implements a memory-shared mechanism to improve performance in propagating messages, which is a solution to a well-recognized DEVS simulation engine (DSE) performance bottleneck.

To deal with error, the model and the simulator must comply with a Verification & Validation (V&V) process. The verification part consists of checking if a simulator is in error, while the validation phase consists of checking if a model is in error.[2] Verification must establish that the *simulation relation*[2] holds between a simulator and a model. There are two general approaches to do verification: (i) formal proofs or (ii) extensive testing. When using simulation engines, the most common verification method is extensive testing. On the other hand, validation must test a model for validity through *modeling relation*.[2] To fulfill this process, input trajectories are typically generated on both the source system and the model.[†] Then, the corresponding output trajectories should be equal, according to some terms of equivalence. To facilitate these two processes, xDEVS includes two utilities: a Unit testing API and a Constraints checker.

[†]In DEVS, this is typically performed following the experimental frame scheme.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Finally, xDEVS also provides support for MBSE development through a graphical tool called DEVSML Studio. xDEVS allows the user to execute the model with the advancement of software engineering practices.

In the following, we describe all these features.

3.2.1 | Wrappers

Adding compatibility between different DEVS simulation engines, and more specifically, between DEVS models developed in the same programming language is the first step to designing a DEVS M&S interoperable framework. Thus, xDEVS incorporates adapters to interact with other simulation engines.

xDEVS version 1.0.0 includes wrappers for aDEVS 3.3 (in xDEVS/C++), DEVSJAVA 3 (in xDEVS/Java) and PyPDEVS 2.4.1 (in xDEVS/Python). The design of the other adapters follows the same pattern. As can be seen in Appendix C, adding more wrappers is a straightforward process. The target model is added as an attribute, and each DEVS function elaborates a simple conversion. Using these adapters, xDEVS models can be easily combined with external DEVS libraries, especially in the C++, Java and Python, as shown in Section 4.

Another important wrapper is `Coupled2Atomic`, which represents an abstraction of a coupled model with an atomic model. Due to closure under coupling property of the DEVS formalism (borrowed from Systems theoretical closure under composition principle), we have an abstraction mechanism by which a coupled model can be executed like an atomic model. In traditional DEVS hierarchical modeling, a coupled model is merely a container and has corresponding coupled-simulators (see Figure 3A). Using the closure under coupling property, it can be transformed into an atomic model with the lowest level atomic simulator (see Figure 3B). This has been accomplished by implementing this adapter, originally proposed in Reference 21. The `Coupled2Atomic` wrapper takes special relevance in distributed simulations: when a model is split, and each part is simulated in different machines, a whole part can be a coupled model. Without this wrapper, partitioning a hierarchical model for distributed deployment is a challenge. This wrapper allows simulating the whole coupled model hierarchy as a singly atomic model deployed on a specific machine.

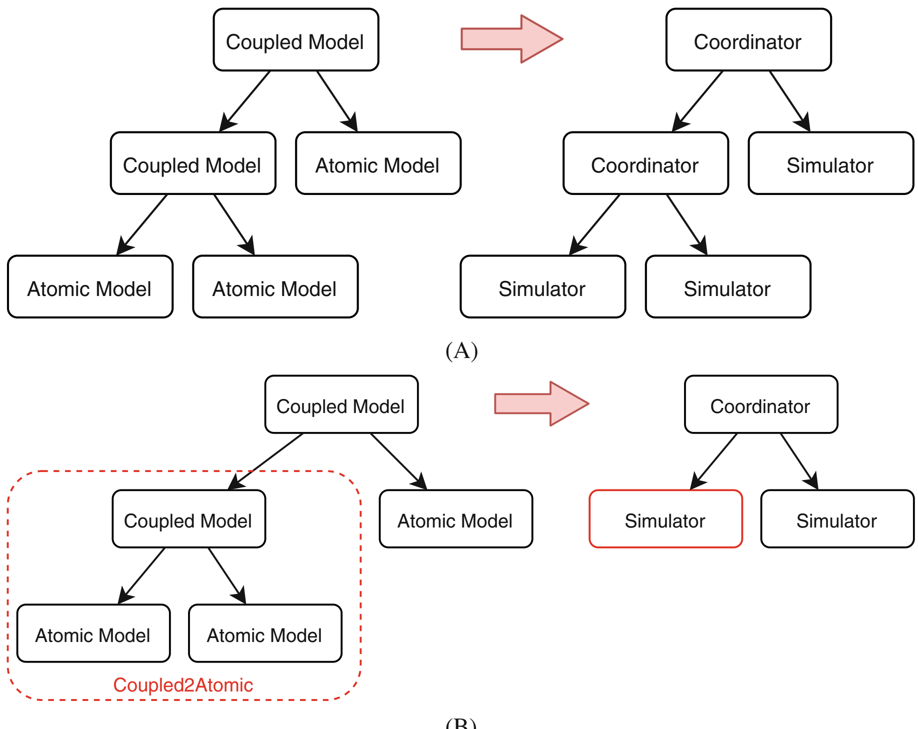


FIGURE 3 Hierarchical simulator assignment with Coupled2Atomic adapter. (A) Hierarchical simulator assignment for a hierarchical model, (B) hierarchy of simulators and coordinators

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Finally, the mechanism of building wrappers is also a powerful tool to provide interoperability between DEVS and non-DEVS models. For example combining xDEVS/Java models with MATLAB models, was demonstrated in Reference 22. We do not address this issue formally because there is no standard mechanism to communicate a DEVS wrapper with a non-DEVS model. The structure of the wrapper completely depends on the implementation of the non-DEVS model.

3.2.2 | Models flattening

Model flattening is often used to simplify the models for simulation efficiency and reduce the overheads introduced by message passing between coordinators in complex models with deep hierarchies. This technique takes advantage of closure under coupling[23] to generate an equivalent single level model from the original model. Hence, the intermediate coupled models are eliminated, and the message passing happens directly between all the atomic models. This algorithm has been incorporated in all the xDEVS branches, being able to perform this transformation when specifying the simulation root coordinator.

3.2.3 | Sequential, real-time, parallel, and distributed simulations

xDEVS allows model simulation using a traditional and sequential process (`Coordinator` class), a real-time simulation (`RTCentralCoordinator` class), multi-threading simulation (`CoordinatorParallel` class) and distributed simulation (`CoordinatorDistributed` class). Due to the categorical separation of the modeling and simulator Application Programming Interface (APIs), the deployment of a particular simulator on a specific computational architecture is transparent to the underlying model implementation, that is, the model behavior and its specification is unaffected by the underlying computational execution of the simulation engine. This is one of the major advantages of DEVS formalism that provides mechanisms for simulation engine verification independent of the model specifications. In the sequential, real-time, and parallel approaches, the Coordinator class is executed in a local main thread on the local machine. In the distributed simulation, coordinators and simulators must be launched as independent processes that communicate through sockets. An example on invoking these different coordinators for the same model is shown in Appendix D in sufficient detail.

Technical details on how to deploy a distributed simulation are quite complex, out of the scope of this paper, since it requires the deployment of virtual machines, containers, or any distributed infrastructures using complex scripts. However, for testing purposes, a distributed simulation can be launched into a single computer, running one distributed coordinator with the configuration file, and two simulators in different terminals with the configuration file as well, and test a distributed simulation in a single machine, like the following code excerpt shows (Listing 6):

```
$ # Terminal 1
$ java -cp xdevs.core.simulation.distributed.CoordinatorDistributed efp.xml
$ # Terminal 2
$ java -cp xdevs.core.simulation.distributed.SimulatorDistributed efp.xml ef
$ # Terminal 3
$ java -cp xdevs.core.simulation.distributed.SimulatorDistributed efp.xml processor
```

Listing 6: Execution of a distributed simulation in a single computer

We have demonstrated a containerized deployment of distributed xDEVS in our earlier work[5] and cloud deployment of parallel and distributed xDEVS in our most recent work.[24]

3.2.4 | Profiling of the simulation

xDEVS provides a special coordinator, named `CoordinatorProfile`. This coordinator measures the number of calls and the wall-clock time taken by each invoked function in the simulation layer. Following the scheme given in Figure A1B,

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

a set of `CoordinatorProfile` and `SimulationProfile` objects are created. `CoordinatorProfile` registers number of calls and time consumed by functions like `initialize`, `exit`, `deltfcn`, `lambda`, `propagateOutput` and, while the `SimulatorProfile` is in charge of similar functions, with the exception of those related to the propagation of events, which is performed by the coordinator. Using these data, a modeler can instrument the time consumed by each coupled and atomic model, since at the end, a majority of the functions in the simulation layer directly call to the DEVS functions in the modeling layer. One example is `lambda`, which in a coordinator class calls to all its children's `lambda` functions, and these children, in case they are a `Simulator`, directly call to the output function of the DEVS model. In its current version, xDEVS does not perform explicit profiling of the model to avoid excessive intrusion and overhead in performance.

The root coordinator accumulates data that can be stored in a .csv file. This file can be filtered to analyze the computational load of the DEVS model. An example is shown in Appendix E. The report shows the class, the number of atomic models participating in the simulation, the wall-clock time performed by the transition function, the output function, and the sum of both time values. Finally, the last column shows the time percentage consumed by each atomic class. This way, the modeler can quickly analyze the model's bottleneck.

3.2.5 | Memory-shared ports

The Python version of xDEVS implements a modification of the PDEVS abstract simulator algorithm originally proposed by Chow and Zeigler.[16] This alternative xDEVS simulation algorithm, called the chained simulator, aims to improve the performance of sequential and parallel simulations by using a function-oriented approach instead of message-passing.[18] This algorithm exploits shared memory patterns to avoid unnecessary message propagation. Appendix F provides an example implementation of the algorithm. The speedup obtained by the chained simulator depends on the structure and characteristics of the model under study. Generally, models with a high number of couplings and models that generate more than one event per port simultaneously can benefit the most from this algorithm. The chained algorithm has been proven to reduce up to 40% of synchronization overhead.[18]

3.2.6 | Unit testing

V&V techniques, widely used in the software industry, have limited presence in the M&S field. Ad-hoc techniques are generally used to verify simulations, and most of the simulation engines lack complete and robust tools for validating their models. Bringing these software techniques would help to automate the V&V of simulation models in a straightforward way. As part of this process, we have integrated a unit testing framework in xDEVS[20] allowing modelers to benefit from the verification procedures. This framework can easily inject test cases as input data into the models, capture the states and outputs of their internal components, and compare them against the expected behavior.

For the specification of the test cases, we have defined an XML-based format that allows specifying the information regarding input generation and behavior verification in a structured manner. Figure 4 shows the structure of these test case files. They contain two main sections: *Generators* and *States*. In the *Generators* section, we define the generator modules injecting inputs into the system. Given the object-oriented paradigm used by xDEVS simulation engines, these generators are defined as classes in the project structure and are dynamically instantiated in the testing procedure. Additionally, since this unit testing framework was introduced in the Java xDEVS simulator, each *Generator* element specifies the classpath of its generator module and the input port where the produced values have to be injected. Notably, several generators can be defined, even in different levels of the hierarchical design.

The *States* section includes information about the variables and outputs at a given simulation time. Each *State* can incorporate port outputs and state variables. *Port* elements must include as the name attribute, the complete path of the port to be instrumented. This includes both the path of the module containing the port and the port name, in a fully qualified syntax: *component1.component2.componentN.portName*. As seen in Figure 4, it is also possible to inspect the values of Atomic modules. It is worth mentioning that these variables can be checked even if they are private in the class design, and internal object attributes can also be inspected following the syntax: *object1.object2.attribute_name*. Through this framework, the modeler can add a verification layer to their simulation models, checking their correctness and behavior. Moreover, the test case files are easily readable and can complement the project documentation. This unit

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
<UnitTest>
    <Generators>
        <Generator name="generator_name" type="path.to.the.generator_class" port="oOut"
            connectTo="path.to.other.module_port" />
        <!-- ... -->
    </Generators>

    <States>
        <State time="[TimeState1]">
            <Port name="coupled1.out_port1">
                <[OutputType] attr1="val1" attr2="val2" />
                <[OutputType] attr1="val3" attr2="val4" />
                <!-- ... -->
                <[OutputType] attr1="val5" attr2="val6" />
            </Port>
            <Port name="coupled1.atomic1.out_port1"> <!-- ... --> </Port>
            <!-- ... -->

            <Atomic name="coupled1.atomic2" phase="active" sigma="200" />
            <Atomic name="coupled1.coupled2.atomic3" simple_attr="val1"
              obj_attr.simple_attr="val2"/>
        </State>

        <State time="[TimeState2]"> <!-- ... --> </State>
        <!-- ... -->
        <State time="[TimeStateN]"> <!-- ... --> </State>
    </States>
</UnitTest>
```

FIGURE 4 XML-based syntax to specify test cases. It allows checking port outputs and internal attributes in all the components of the DEVS simulation.

testing framework also includes auxiliary classes that allow fulfilling the testing process directly using programming code, facilitating the adaptation of the methodology for alternative test case formats or requirements.

3.2.7 | Constraints definition

As part of the efforts made in xDEVS to enable mechanisms to perform V&V, a constraint-based methodology is implemented in the simulation layer[19] that allows the verification of DEVS-based models. Introduced in the C++ xDEVS branch, it allows specification of custom JSON-based syntax constraints based on model component's outcomes. Due to its placement in the DEVS simulation layer, it is completely decoupled from the model and the simulation engine syntax. These constraints are specified following a mathematical approach in terms of arithmetic and logical equations. Their terms can refer to simple and complex data structures, as long as the used operators are defined in such data types. Figure 5 shows the expected structure of a constraints specification file, containing two main sections, namely *vars* and *constraints*.

The *vars* section is optional and includes a collection of variables, each of one representing the values contained in a port or the arithmetic combination of the values of several ports. Each variable is expressed as a pair of `<variable_name>: <arithmetic_expression>`. The variable name must consist of uppercase and lowercase letters, numbers, and underscores. Arithmetic expressions are defined using the ports as terms, and the "+", "-", "*", and "/" operators. Ports are referenced following the full DEVS path in the following format: $coupled_1.coupled_2. \ldots\ coupled_N.atomic_a.port_p$, where $coupled_i$, $atomic_a$, and $port_p$, are coupled, atomic and port identifiers specified in the definition of the DEVS model structure.

The *constraints* section is mandatory and specifies the set of constraints checked in simulation time, expressed as logical expressions. Each constraint has three main attributes: (i) constraint name, specified with the same restrictions as the variable names for any programming language, (ii) the logical expression, and (iii) the severity level. The terms

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
{
    "vars": {
        <variable_name1>: <arithmetic_expr1>,
        <variable_name2>: <arithmetic_expr2>,
        ...
        <variable_nameN>: <arithmetic_exprN>
    },
    "constraints": {
        "constraint_name1": {"expr": <logic_expr1>, "level": <"info"/"error">},
        "constraint_name2": {"expr": <logic_expr2>, "level": <"info"/"error">},
        ...
        "constraint_nameN": {"expr": <logic_exprN>, "level": <"info"/"error">}
    }
}
```

FIGURE 5 Example of a JSON-based constraints file

of these expressions can correspond either to model's ports or to the previously defined variables. It can also contain arithmetic and logical operators. Also, some auxiliary functions can be used to deal with arrays, such as sum, len, min, and max. The severity level of a constraint can be set to *'info'* for generating warnings when the assertion indicated by the logical expression is not accomplished, or be set to *'error'* to point critical constraints. These constraints terminate the simulation execution when their related expression is not fulfilled.

The constraints are always evaluated at the end of each simulation cycle, before the cleaning of the ports, provided that the implied output ports are not empty. The chosen implementation architecture allows checking mathematical properties over multiple system components, even if they have been defined at different levels of the DEVS hierarchical structure. Moreover, as it is conceived as part of the DEVS simulation layer, the verification process is completely orthogonal to the model design.

The execution time overhead introduced by the Constraints checker is minimal. It is linear with the number of constraints, that is, its complexity is $O(N)$ using the Big O notation and being N the number of constraints. On the other hand, the Unit testing tool complexity mostly depends on the intensity of the input trajectories introduced by the user. In any case, these two tools do not introduce scalability issues as the model becomes more complex.

3.2.8 | DEVSML Studio

As mentioned above, xDEVS provides support to MBSE through DEVSML Studio (Figure 6).[6] DEVSML Studio offers a fully functional textual/visual programming language. This software package, still in development, provides an integrated development environment based on DEVSML, where various languages can be used at the modeling layer leveraging the modeling API interface. xDEVS is made available as the Platform Specific Model (PSM) of the more abstract DEVSML.

The execution of model through a simulator is a complete discipline in itself and till date executable UML/SysML are incomplete as they lack theoretical systems foundation to begin with. MBSE with the help of metamodeling, the use of domain specific languages and with a profound background on theory on modeling and simulation can bring together MBSE and MDE as a MDSE in a definitive manner and extend it for application in a netcentric environment. This process can be integrated into xDEVS through DEVSML Studio with the use of DEVSML and the DEVS Unified Process (DUNIP). DUNIP and DEVSML are a methodology and a language for conceptual modeling and engineering of complex systems that integrates in a single view the functional, structural and procedural aspects of the modeled system using formal yet intuitive UML graphics that are dynamically generated. For more information about DEVSML and DUNIP, the reader is referred to References 4,5,25.

DEVSML Studio main features are:

- It is based on Eclipse PDE with Xtext Extended Backus Naur Form (EBNF) grammar underneath as DEVSML metamodel.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

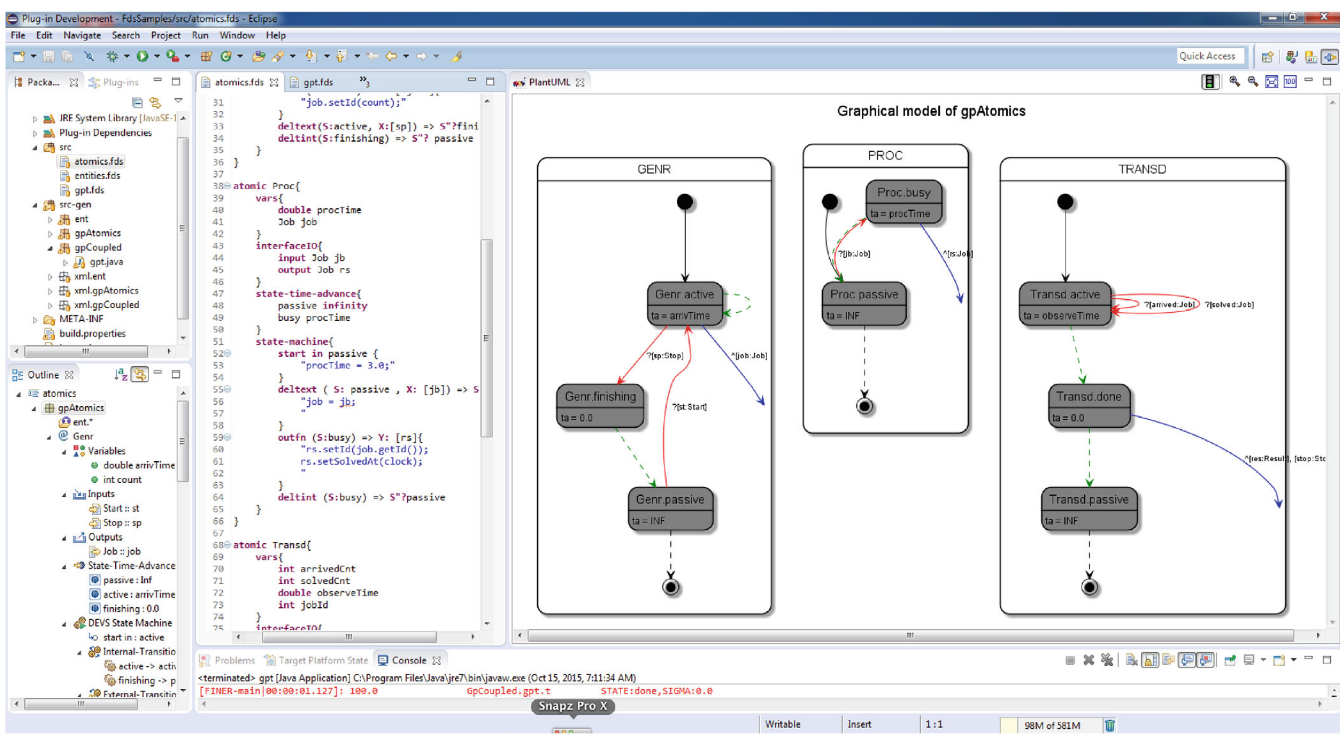


FIGURE 6 Snapshot of DEVSML Studio showing DEVS Behavior State Machines graphical diagrams rendered automatedly from textual DEVSML description using MDE principles

- It provides textual templates for atomic and coupled models, rich with code-completion and DEVS model validation.
- It provides a visualization plugin for rapid visual inspection of both the atomic and coupled DEVS. The visualization plugin is based on open-source PlantUML plugin.
- It provides auto-generated compiled Java code for ready execution of DEVSML.
- It integrates EclEmma Code Coverage plugin for JVM executable platform-specific code.
- Code-snippets are provided as string and when a model runtime is configured for a DEVSML project, the PSM shows errors in the generated platform-specific code.
- It shows the hierarchical structure of a DEVS file in the outline view.

Figure 6 shows how DEVSML Studio can be used to develop the traditional GPT DEVS academic example, combining some of the features enumerated above.

4 | XDEVS STANDARDIZATION AND INTEROPERABILITY

The DEVS theoretical foundations guide implementations in different programming languages across different hardware platforms, as can be seen in Section 3.1. This forms the basis of developing an M&S Standard that would align various implementations. This section discusses the fundamentals on achieving a DEVS M&S Standard with the help of the xDEVS M&S API.

One of the main features to highlight is the direct consequence of separating the model from the simulator: this results in multiple ways to write a simulator for the same model, albeit, guided by the constraints of the simulation protocol that defines the interactions between a DEVS-compliant model and a DEVS-compliant simulator. For instance, there are virtual-time simulators (where the simulator can skip from one event time to the next without traversing the intervening time interval) and real-time simulators (where time is interpreted as wall clock readings, so the simulator must wait for the time to its next scheduled event to expire before handling the event). In addition to the different combinations of model-type/simulation-software, an M&S standard facilitates the selection of a specific simulator for a corresponding

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 7 Conceptual architecture of the DEVS Standard (a)

model component/architecture (e.g., sequential vs. parallel/distributed, and within the latter, conservative vs. optimistic time advance for virtual-time as well as centralized vs. noncentralized time control in real-time execution).

The standard will facilitate simulator execution over different platforms, such as Windows versus UNIX; different programming languages such as C++ versus Java; and further guides inclusion of different networking protocols like sockets vs the message-passing interface for exchanging messages between participating simulators. A standard M&S API will, therefore, harmonize both the model and simulator interfaces across various implementations. For example, the same model may be simulated in virtual-time and real-time, and can be executed in a distributed and a local, non-distributed fashion.

In summary, the proliferation of DEVS-based M&S engines would benefit from such standardized DEVS API, facilitating model's extended reuse and interoperability at-large, especially in distributed simulation use cases. The DEVS formalism specifies the abstract simulation protocol that accurately simulates DEVS atomic and coupled models. Interpreted in a distributed simulation environment, the DEVS abstract simulator gives rise to a universal abstract simulation protocol with specific mechanisms for communicating with other simulators, that is, the federates. It also specifies how federates interact in an iterative cycle that controls how time advances when federates exchange messages and perform internal state updating. A significant feature compared to simulation based on the HLA standard is if the federates are DEVS-compliant, and the federation runtime correctly simulates the DEVS-compliant models, then closure under coupling guarantees a well-defined resultant behavior from concurrent interactions.

We present Figure 7 as the initial conceptual architecture of the standard. This architecture is divided into four layers: the simulation protocol (parallel, distributed, etc.), the simulation layer, the modeling layer, and the transformations layer. This is a simpler form of the DEVSML 3.0 Stack.[5] A DEVS simulation engine (DSE) is labeled in Figure 7 for instance as DSE1/A, being DSE1 the simulation engine 1, which uses the programming language A. xDEVS has been explicitly labeled because it will be used to interoperate between different DSEs. The programming language of xDEVS has been denoted as "*", since xDEVS supports several programming languages (C++, Java, and Python). It is worthwhile to mention that the in a large number of simulation applications, models are always coupled to their corresponding simulation engines. There are no mechanisms to decouple models and simulators: (a) through the use of wrappers or (b) using a standard modeling language and transformations. In the following, we describe this concept that separates the model specification layer with the simulator implementation layer.

4.1 | Platform independent models

Standardizing DEVS model representation as a Platform Independent Model (PIM) allows a model to run on any DSE. An example of a DEVS PIM is implemented in DEVSML Studio.[6] This is powerful because a model can be retrieved not only from a persistent storage but can integrate DEVS with other domain specific languages (DSLs). It provides PIMs and DSLs to interoperate with DEVS semantics once the transformations are developed. Different groups have used PIMs as a mechanism for interchanging model information. Once a PIM specification has been defined, several mechanisms to verify the correctness of the model can be elaborated.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Figure 7 shows a DEVS PIM standard definition framework. Once a DEVS PIM definition is broadly accepted, its model can be transformed to any other DSE, implementing the corresponding transformation. Several DEVS PIM proposals can be found in the literature. We highly recommend DEVSML 3.0,[5] due to its simplicity as a DEVS-compliant DSL and its code-generation capabilities to generate an executable xDEVS/Java application.

4.2 | DSE1/A to DSE2/A interoperability for the same programming language

As stated in Section 2, there are many libraries for expressing DEVS models. All have efficient implementations for executing the DEVS protocol. To find an interoperable DEVS M&S framework using xDEVS, we must design compatibility between different DSEs and xDEVS, that is, a model implemented using either DSE/A or DSE/B should be able to be simulated in xDEVS/A or xDEVS/B, respectively. In addition, the root coupled model should be able to be simulated in xDEVS. As explained in Section 3, xDEVS incorporates some adapters of other DSEs (e.g., aDEVS, DEVSJAVA, PyPDEVS), and other adapters from the remaining DEVS engines in the community can be easily implemented. With the current implementation, xDEVS allows interoperability among different DEVS simulation engines. As a result, we can combine different DSEs with wrappers, as long as they are implemented in the same programming language (C++, Java, or Python). It is worth mentioning that coupled models do not represent an obstacle since they can be easily converted to atomic models using xDEVS using the flattening feature.

4.3 | DSE to non-DEVS interoperability

The best way of including a non-DEVS model into DEVS simulations is with the use of the same wrapper design pattern. However, as we mentioned in Section 3, building an adapter or wrapper for a non-DEVS model requires ad-hoc solutions, but is still possible. This relies on the aspect that the DEVS modeler needs to be familiar with the non-DEVS model. For instance, in Reference 4 authors define a barrel-filler model combining DEVS models with MATLAB models through a wrapper.

4.4 | DSE/A to DSE/B interoperability between different programming languages

A model and its simulator are considered as two distinct elements in the DEVS theory. The simulation protocol describes how a DEVS model should be simulated, whether in a standalone fashion or in a federated manner. Such a protocol is implemented by an algorithm in the simulation layer that executes the model. To reach DEVS interoperability between DSEs implemented in different programming languages, we must address it at the simulation layer (as it is transparent to the modeling layer.[21] Thus, in xDEVS we focus on implementing a simulation architecture that allows the simulation of any DSE/* model. This is done by taking advantage of the xDEVS distributed simulation architecture. Since all the messages are passed in an xDEVS standard format (using sockets and serialized to Javascript Object notation (JSON)), xDEVS/Java simulators can be combined with xDEVS/C++ or xDEVS/Python simulators (as any DEVS-compliant simulation implements the same abstract DEVS simulation protocol).

Figure 8 shows a simulation of a generator-processor transducer (GPT) DEVS model.[2] The example is described in Appendix A. In this example, the root GPT coupled model has been defined using the configuration file described in Appendix A (see Figure D1) and an xDEVS/Java coordinator. The Generator model is implemented using another DEVS simulation engine implementated in the C++ programming language (called DSE1/C++). The Processor model is developed using a third DSE implemented using Python programming language (called DSE2/Python). Finally, the Transducer atomic model is implemented in xDEVS/Java. A non-DEVS model can be incorporated as long as the wrapper is implemented. The simulation does not have to be distributed whether the localhost is used in all the models. As can be seen, when all the model adapters are implemented, the simulation of heterogeneous models as the one presented in Figure 8 is thereby, achieved.

To summarize, using xDEVS, the conceptual architecture of the standard is presented in Figure 9, where both the modeling and simulation layers are completely unified at the API level for different DSEs through xDEVS wrappers. We presented several ways to implement a DEVS standard protocol using the xDEVS simulation engine. In the modeling layer, we can provide interoperability between different DEVS M&S platforms using the well-known wrapper pattern.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

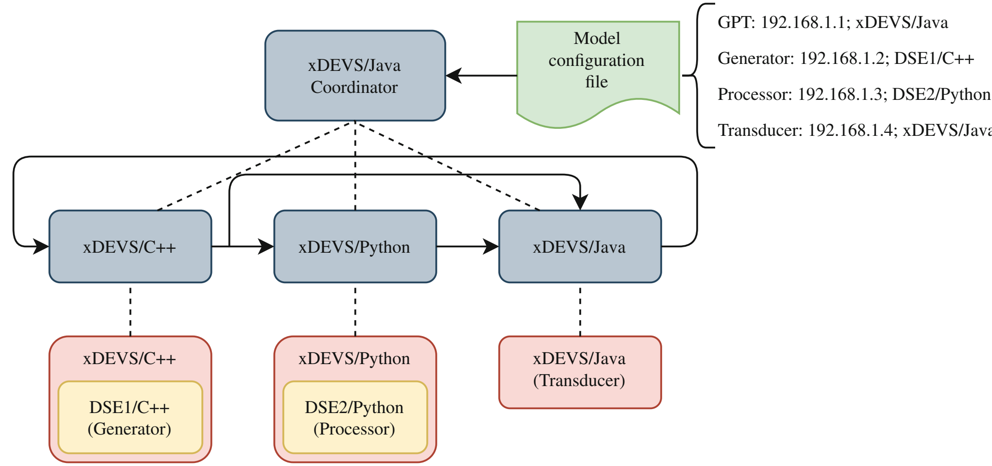


FIGURE 8 Federation of DEVS simulators with heterogeneous DSEs and the xDEVS common distributed interface

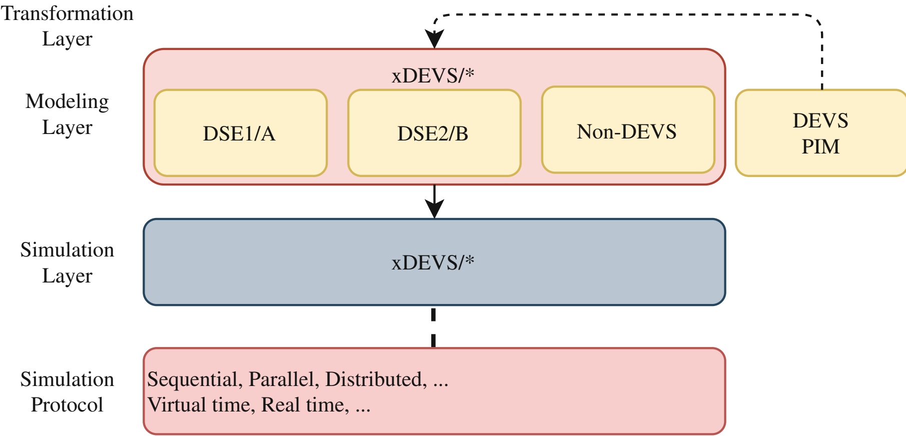


FIGURE 9 An Implementation of the conceptual architecture of DEVS Standard using xDEVS

The DEVS standard can be easily implemented in the simulation layer using the xDEVS distributed simulation architecture to provide a unified simulation layer. As we have seen, the simulation process using the xDEVS M&S API is straightforward since this architecture already provides the unified translation of messages, or in another way, the implementation of a standard message format.

The DEVS formalism has been in existence for more than four decades and has been implemented in almost all programming languages. Recently, it has been extended to incorporate various PIMs and DSLs. The DEVS Community has thought over these issues over the years and it is only recently that the work on DEVSML Stack and the current xDEVS engine provides an integrative view of bringing various DEVS implementations together. Undoubtedly, the DEVS theory already had the conceptual constructs such as DEVS simulation protocol and Levels of Systems specification that facilitated the development of such a standard but the missing link was the incorporation of the latest in composable software engineering and metamodeling concepts that surfaced in the last decade. Together, with xDEVS meta-architecture, application programming interface (APIs) and implementations in Java/C++/Python, a Standard is thus realized.

5 | PERFORMANCE ANALYSIS

The Java branch of xDEVS, as the first implementation of this framework, has already been compared with other state of the art DEVS simulators, showing robust performance.[26] In this section, we analyze the performance of the entire framework, including the C++ and Python implementation through the set of DEVStone benchmarks.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5.1 | DEVStone

DEVStone[27] is a synthetic benchmark devoted to automate the evaluation of DEVS-based simulation approaches. It allows the generation of different types of models, each of them specialized in measuring specific aspects of simulation. This benchmark has become popular over the years, and has been used by plenty of authors of the state of the art to evaluate and compare the performance of different DEVS simulators.[26,28,29] DEVStone describes several synthetic models that can be configured to vary their size and complexity. For this, they present a recursive structure with configurable depth where all the levels contains the equivalent components and interconnections. The customization of the models is done through the use of four parameters: (i) *width*, that affects to the number of components per layer, (ii) *depth*, that specifies the number of nested coupled models, (iii) *internal transition delay*, and (iv) *external transition delay*. These two types of delays execute CPU-intensive operations a fixed amount of time in the internal and external events of the atomic components.

DEVStone describes four types of models (depicted in Figure 10):

- **LI** model: it is the simplest model, with a low level of interconnections in their coupled models.
- **HI** model: similar to the LI model, but increases the number of internal couplings.
- **HO** model: variation of the HI models where all the atomic components in each coupled module are connected to the coupled output port.
- **HOmod** model: it reproduces an exponential level of coupling and outputs model.

5.2 | xDEVS engines comparative analysis

In Figure 11 we can see a plot matrix representing a comparison of aDEVS vs. xDEVS DEVStone simulation times. aDEVS stands for *a Discrete EVent System simulator, a C++ library for constructing discrete-event simulations based on the Parallel DEVS and Dynamic DEVS formalisms, developed at Oak Ridge National Laboratory (ORNL) by Jim Nutaro. We selected aDEVS because it is the fastest DEVS simulator, and is typically used as a baseline for performance analysis*. In Figure 11, each row corresponds to one of the DEVS implementations (aDEVS, xDEVS/C++, xDEVS/JAVA, and xDEVS/Python), and each column represents a specific DEVStone model class (LI, HI and HO). We have not included HOmod because the large amount of memory demanded by this DEVStone model class forces us to use a low range of HOmod instances, and as a consequence HOmod does not allow us to perform a fair comparison (see e.g., Reference 26). For each specific pair of these implementations and DEVStone models, a plot contrasting the simulation times is shown. The X and Y axes represent the DEVStone width and depth parameters used in the simulations. The color corresponds to the execution time of each DEVS simulator for the given DEVStone model instance. This execution time includes the loading of the model, the initialization of the coordinator and the simulation time.

Taking into account the heat maps illustrated in Figure 11, aDEVS is still the fastest simulation engine. However, xDEVS/C++ offers equivalent execution times, reaching better peak performance values as the DEVStone instances are made more complex. In fact, for HO, xDEVS/C++ shows a better peak performance. Appendix G shows that, in average, aDEVS is finally better than xDEVS/C++, but xDEVS/C++ is an excellent alternative, given the wide range of features enumerated in Table 2. The same happens with xDEVS/Java. This engine is slower than xDEVS/C++ and aDEVS, mainly produced by the extra resources needed by the Java Virtual Machine. However, as happens with xDEVS/C++, the performance with respect to aDEVS is improved as the DEVStone model gains in complexity, reaching a similar peak performance in DEVStone large HO models (see Table G2 for more details). Parallel and distributed simulations with xDEVS/JAVA have an straightforward deployment, in comparison to xDEVS/C++. Thus, we may consider xDEVS/JAVA as the right choice when the model complexity is high and parallelization is mandatory. Finally, the Python version of xDEVS is the slowest one. Some models were impossible to load and were assigned the worst execution overall time. This was expected as Python is an interpreted language. However, since Python libraries have excellent toolboxes for data analysis and management, we consider that xDEVS/Python can be the best option for data scientists.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

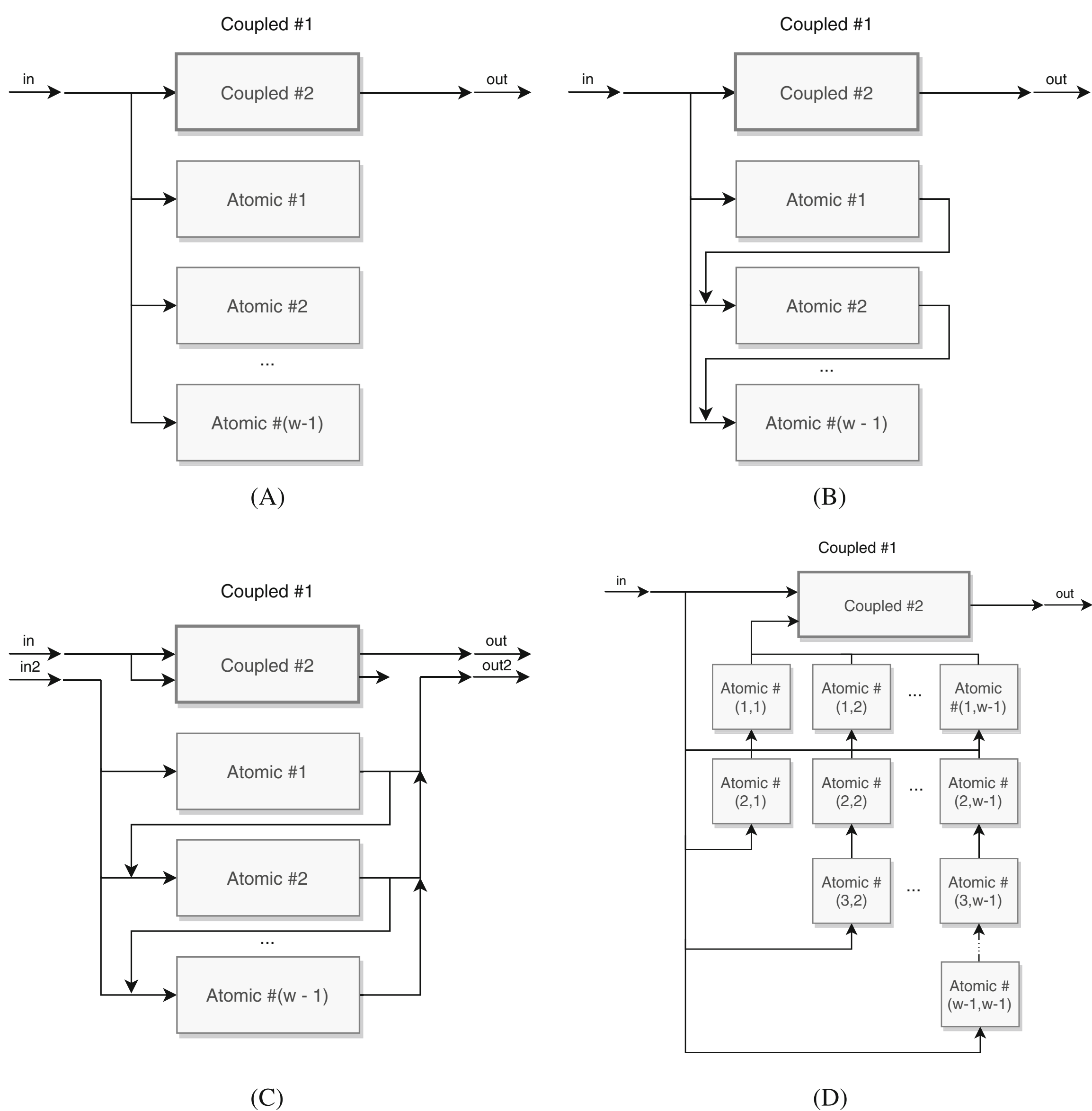


FIGURE 10 DEVStone models. (A) Low level of Interconnections model (LI), (B) high Input couplings model (HI), (C) HI model with numerous outputs model (HO), (D) exponential level of coupling and outputs model (HOmod)

Overall, we may conclude that aDEVS or xDEVS/C++ simulation engines are the appropriate choice for small models, whereas xDEVS/C++ and xDEVS/JAVA are adequate for complex models where we want to reach a competitive performance, that can also be enhanced with parallelization or distribution. Given the varied set of features offered by xDEVS, this makes our proposed simulation toolkit an excellent alternative for interoperable modeling and simulation of formal discrete event systems.

6 | CASE STUDIES

Since the creation of a basic Java implementation of xDEVS in 2012, xDEVS/Java, and the final M&S architecture presented in this work, the framework has been used in various research works, in fields as diverse as Cloud/IoT,[30,31] medicine,[32,33] Smart Grid,[34] military applications[22] or netcentric system of systems engineering.[4] Several recent case studies exist that highlight the impact of these features to bring the benefit of integrated M&S in complex systems engineering life cycle. In fact, it is the pursuit of having a cohesive framework which led to development of various unifying features

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

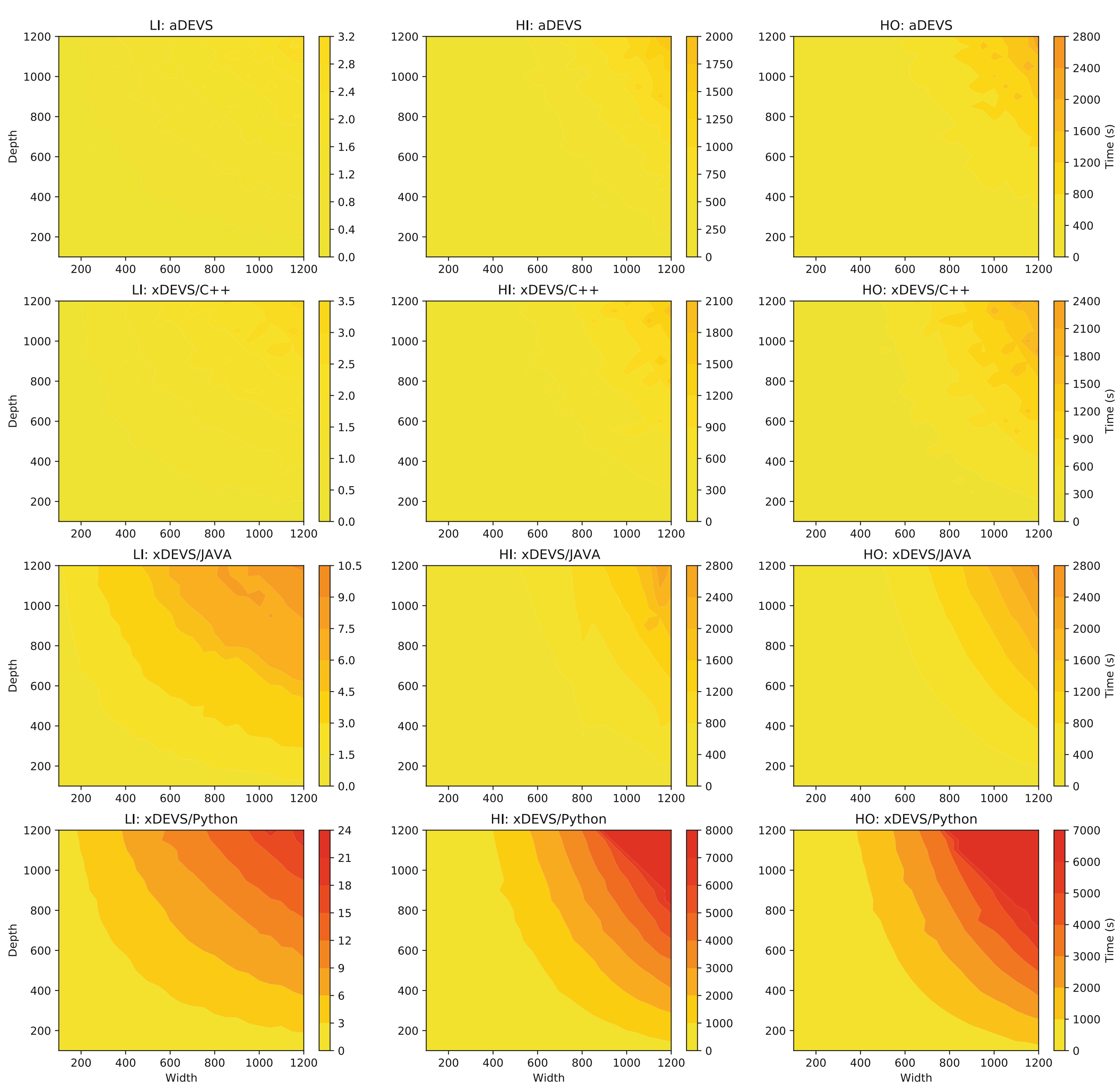


FIGURE 11 DEVStone simulation times comparison between the different xDEVS implementations

(described in Section 3) as demanded by these case studies. In this section, we provide a few such notable real world examples of complex systems demonstrating the applicability of xDEVS M&S framework. At the end of each example, we provide a brief discussion enumerating the main xDEVS characteristics that were developed and eventually used for that particular case.

6.1 | Migraine prediction system

In this model,[32] Pagan et al. use the xDEVS/Java to adapt a methodology for predictive modeling of symptomatic crises in chronic diseases presented in their research[35] to address the prediction of pain episodes in the migraine disease. The resulting system allows capturing different hemodynamic variables and produce alarms to warn the migraines

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

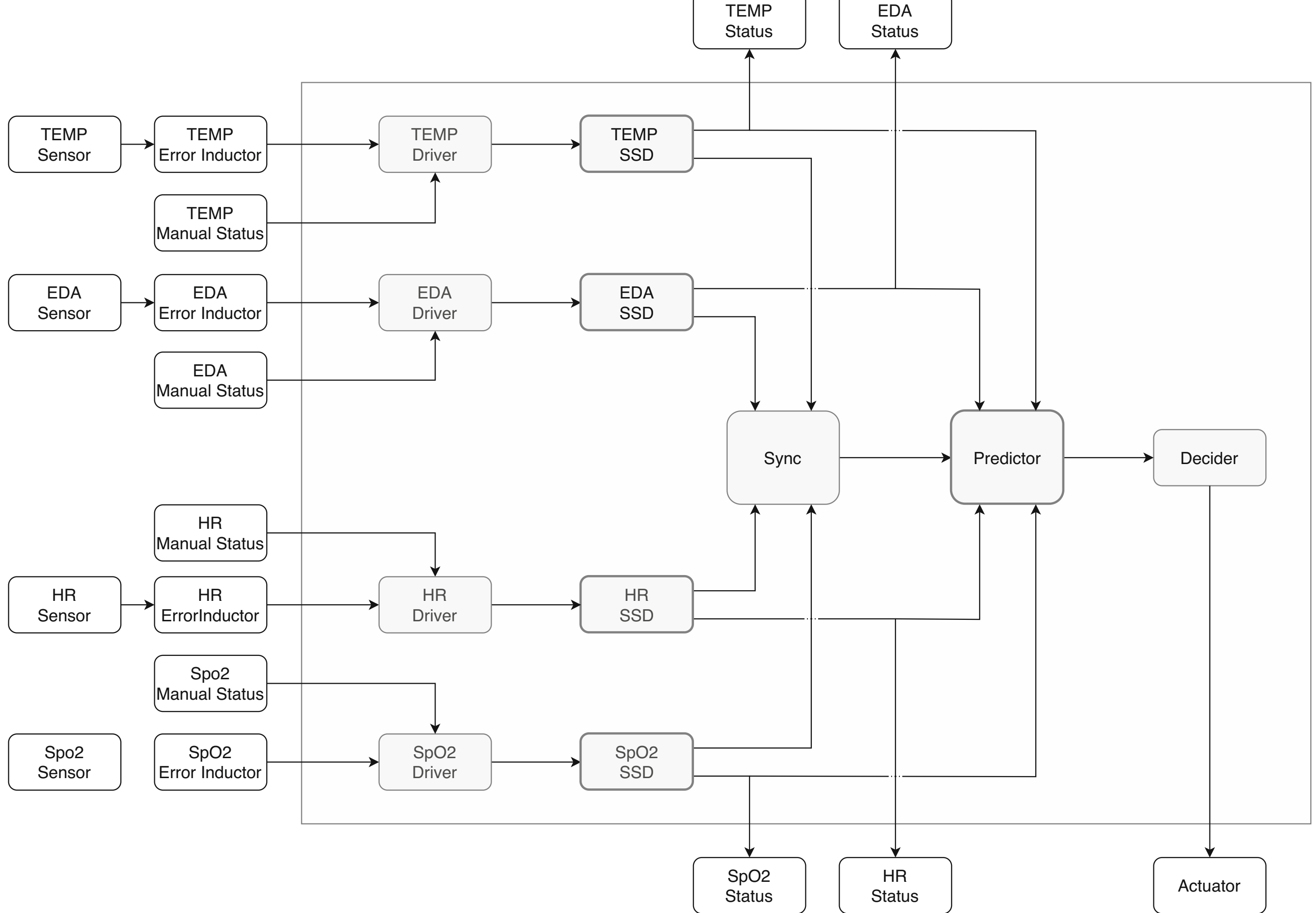


FIGURE 12 Root view of the migraine prediction system DEVS model

patients of the proximity of a pain episode as early as 45 min before the onset of pain. The methodology incorporates training of different sets of models, that are activated based on the available signals (being able to generate predictions if the predictive model has three or four available signals). Its root component is shown in Figure 12. It has eight inputs, four of which from the Autonomous Nervous System related signals, captured through in-body sensors. These inputs are provided by the `TEMP` (body temperature), `EDA` (electrodermal activity), `HR` (heart rate) and `SpO2` (oxygen saturation) and go through `ErrorInductor` components before entering the prediction system. The corresponding error inductors adapt the signals to reproduce several issues that affect sensors in real life (noise, saturation, and disconnections). The four `Manual Status` inputs are used to notify the restoration of damaged sensors and reset its operation. As outputs, it has the alarm signal itself and four LEDs indicating if there is some kind of error in the input signals.

Each pair of sensor and the reset component go to a `Driver`. These components are intended to put timestamps to the captured data, based on a shared clock. The resulting data is directed to Sensor Status Detector (`SSD`) coupled components. Inside them, there are several atomic components in charge of abnormal behaviors detection. Also, when a failure in a system is detected, a signal to the `Predictor` component is raised and an internal Gaussian-based signal-reparation atomic component is activated. This components generate a predicted signal based on values buffered before the sensor failure. To assure its reliability, this signal is generated only for a limited time. Its worth to note that as the signal-generation behavior is encapsulated in an atomic component, it could be easily interchangeable by other methodologies. In Figure 13, we can see how the `SSD` activates the generation of a predicted signal when an error occurs and keeps the original signal when there is no error detected.

The outputs of all the `SSD` goes to the `Sync` component, that synchronizes and buffers the data for simultaneously supplying the values for the four variables. Its output is received by the `Predictor`. This coupled component selects the

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

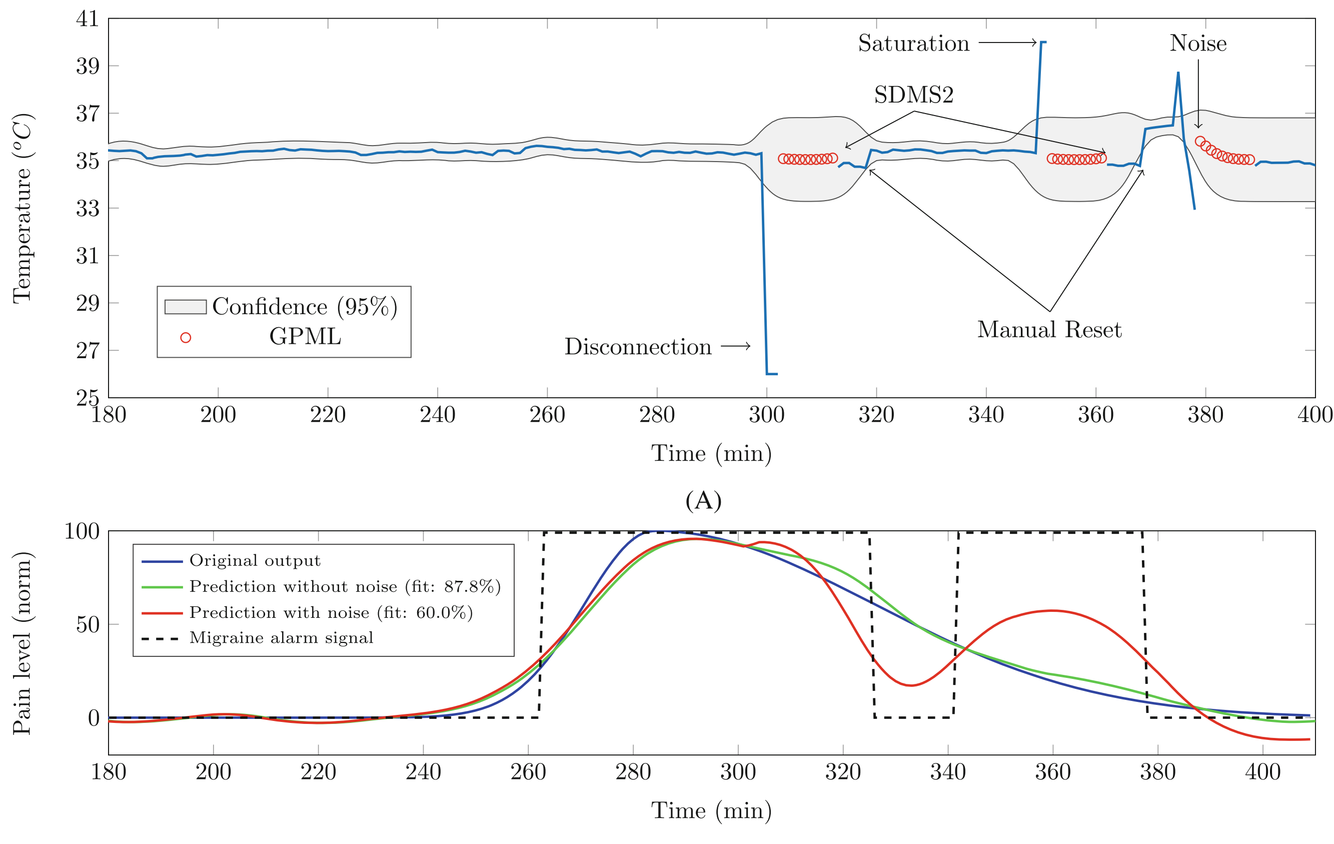


FIGURE 13 Response of the migraine prediction system against failures. (A) System events against failures in the temperature signal, (b) system's output and alarm event

suitable set of models based on the available signals and generate a value predicting the probability that a pain episode is approaching. When this probability exceeds a predefined threshold, the `Decider` raise an alarm to warn the patient. In Figure 13 we can see the response of the system against failures in the sensor. Figure 13A shows a temperature signal that is affected by several types of errors (disconnection, saturation, and noise). When any of them are detected, a Gaussian Process for Machine Learning (`GPML`) module is activated, generating predictions for a limited time until the user presses the `Manual Status` button. When the `GPML` cannot generate more reliable predictions, the signal is discarded until recovery, changing the set of models used for the generation of the alarm. Figure 13B shows the response of the system facing different conditions. The blue line represents the actual pain level registered by the patient, while the green and red lines compare the output of the system with and without the presence of noise. Finally, the dashed line indicates when the alarm goes off based on that predictions.

6.1.1 | Impact of xDEVS architecture and features

The main impact of the xDEVS framework was at the V&V stage in the Migraine Prediction System engineering lifecycle. This project developed a conceptual model, simulated with a local coordinator, which was then upgraded to distributed, real-time coordinator for its design and transition to a real-world system, without changing the model (due to the categorical DEVS separation of modeling and simulation layers). The final objective was to build a hardware device to perform the functionality described above. The developed DEVS model was faithfully reproduced and deployed into the real device demonstrating the entire MBSE workflow. To verify the model, both the timing constraints and the numerical results were carefully analyzed. The modeled DEVS transition delays were equivalent to those specified for the final hardware device, to assure the fulfillment of hardware timing constraints. In addition, the data was validated against actual results. Real time xDEVS coordinators were used to mirror the behavior of the device in a real scenario. Some xDEVS wrappers

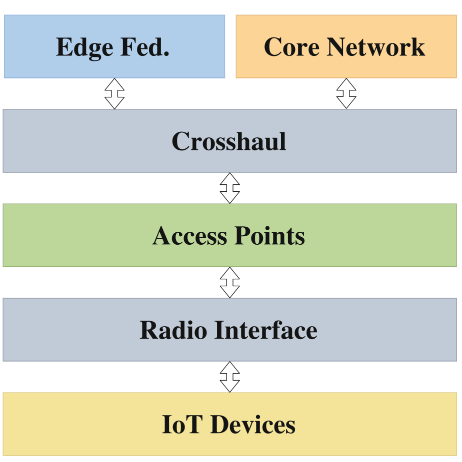


FIGURE 14 Mercury computing model

were also adapted to communicate the xDEVS model with numerical tools like MATLAB (R). At the end, the model was synthesized in an FPGA, obtaining the expected results. More details were reported in Reference 36.

6.2 | Mercury: A framework for modeling data stream-oriented Internet of Things (IoT) applications

Mercury[31] is a modeling, simulation, and optimization framework to analyze various dimensions and the dynamic operation of real-time Fog computing scenarios, developed with the xDEVS/Python implementation. It allows specifying 2D Mobility scenarios and includes a 5G-based model. Figure 14 presents a general view of the computing model of the Mercury framework. It is divided into six layers:

- *IoT Devices* layer: This includes the user equipment (UE) devices of the scenario. This devices are often data stream-oriented that delegate some processing tasks to the cloud due to their computational and energetic limitations. In Mercury, each UE may implement one or more applications that generate data streams to be processed.
- *Edge Federation* layer: This is composed of a set of edge data centers (EDCs), communicated through a specific radio access network (RAN). These include the actual processing units, where the UE tasks are computed. It follows a Functions-as-a-Service (FaaS) concept, where the resources are not reserved to a particular application. Instead, the infrastructure starts a process only when a client makes a service request.
- *Access Points* (AP) layer: This specifies the particular AP where the UE can connect to communicate to the EDCs.
- *Radio Interface* layer: This includes the resources that connect the UEs with the different APs of the scenario.
- *Core Network* layer: This assumes the role of an internet service provider (ISP), enforcing access control policies in the RAN and configuring the *Crosshaul* layer.
- *Crosshaul* layer: This interconnects the EDCs, APs and core networks.

The framework also includes utilities to ease the process of selecting the APs and EDCs' optimal location and generating useful output plots to study the results of the simulations. The objective here is to determine the optimal locations of these elements to (a) provide a good quality of service in an advanced driver assistance system (ADAS), where cars are continuously sending data to the EDCs to analyze the environment and driver condition, trying to avoid accidents, and (b) reduce the energy consumed by these EDCs, and in consequence reduce the ecological footprint and costs. In Figure 15, we can see some examples analysis of EDCs location.

Figure 15A depicts the result of executing an automatic allocation of EDCs and APs based on the density of movement of geolocalized UE (cars in this case). Here, the EDCs are represented with small squares (in orange, green and blue colors) and the their respective APs are represented with stars. Also, the areas covered by each one of the EDCs are

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

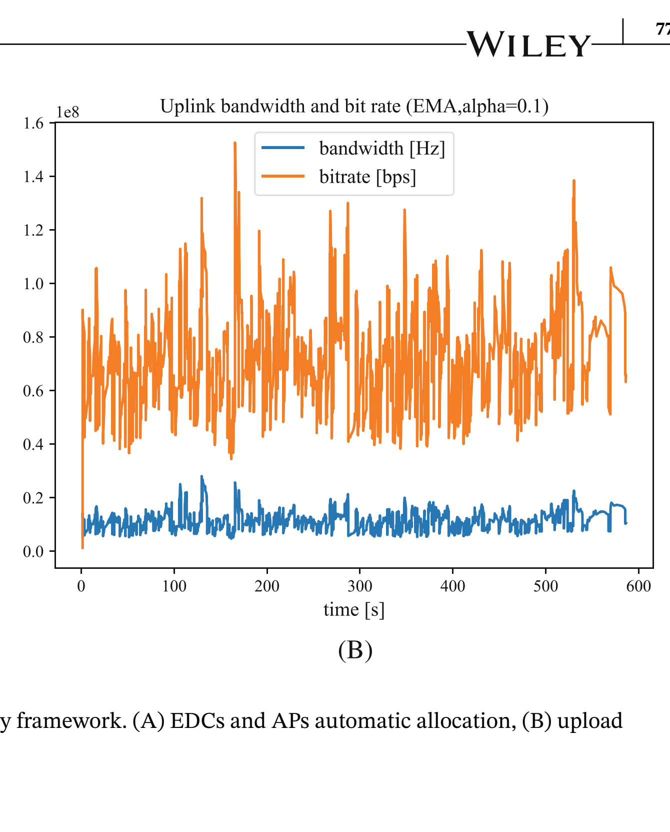


FIGURE 15 Examples of output plots generated by the mercury framework. (A) EDCs and APs automatic allocation, (B) upload bandwith and bit rate

shown. Figure 15A shows the evolution of the uplink and bit rate for a specific scenario. Similar plots are generated for the downlink bandwith and bit rate. Apart from the ones seen here, additional plots are available, showing the uplink spectral efficiency, EDC's power consumption comparisons and UE's perceived delay comparison.

6.2.1 | Impact of xDEVS architecture and features

This use case was developed using the xDEVS/Python. The IoT ADAS model is a complex model, needing more than 20 min to simulate 10 s of the real world. As a consequence, continuous profiling, one of the xDEVS features, was essential to locate the bottlenecks and improve execution speed. To address the performance issue, two actions were taken: (1) Some principles of multi-resolution modeling allowed us to have several equivalent versions of the same model, replacing the hierarchical atomic models acting as bottleneck with others, with simpler in structure (using the closure-under-coupling DEVS property to encapsulate some complex models, formed by many atomics, into one single atomic model), and with a coarse-grain time resolution. This allowed the mapping of wall-clock simulation times to the real-world time resolution. More details were reported in Reference 37, and (2) the architecture of the xDEVS/Python simulation engine was improved, incorporating the memory-shared ports technique listed in Table 2, obtaining an even higher speed-up (see Appendix F, for more details). This use case leverage advanced instrumentation, profiling, model flattening and memory optimization techniques for scalability.

6.3 | Unmanned aerial vehicles (UAVs) in hostile environments

This simulation,[22] developed using the xDEVS/Java implementation, presents a scenario with military conflicts. In this research, offensive UAVs send missiles to objective (target) points, and Air Defense Units (ADUs) try to block the UAV attacks. For each launched missile, the position, velocity, and Euler angles are tracked. In the model, UAV modules share targets information with the ADUs to check if a radar has detected the corresponding target or a missile has hit upon it. ADUs send information to UAVs regarding the state of the missiles (for example, if they have been exploded). The root model of this simulation is depicted in Figure 16. As it can be seen, there are two types of Radar modules inside the ADUs, one for detecting the missiles and a second one to track UAVs' positions. Although the UAVs follow a precomputed path initially, they can modify their path when some unknown threat appears. Details about the behavior design of UAVs and Missiles are provided in Appendix H.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 16 DEVS root coupled model for UAVs simulation

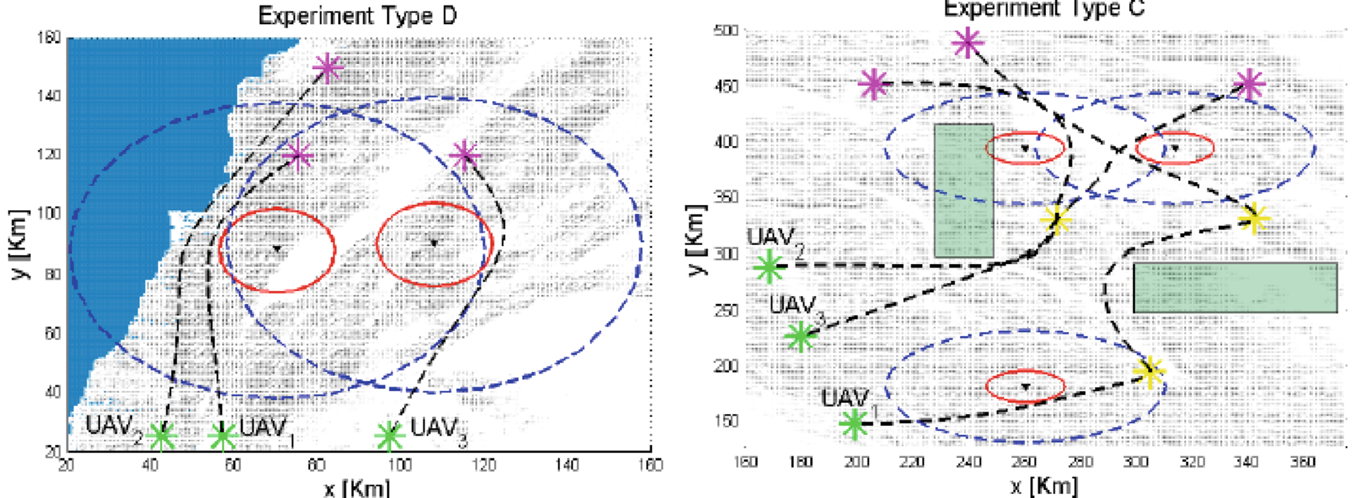


FIGURE 17 Eagle eye view of UAVs simulations scenarios

Several scenarios were modeled. The one depicted in Figure 16 leverages the distributed capabilities of xDEVS/Java. The UAVs and the ADUs were simulated in different workstations, sending intermediate communications over the network (shown with dashed lines). Some sample scenarios are shown in Figure 17. Blue dashed lines encloses the area covered by ADUs, risky to fly over them, but not critical. Red solid lines correspond to critical areas covered by tracking radars. The trajectories of the UAVs are represented with the black dashed lines. Waypoints are depicted with asterisks, and prohibited zones with rectangles.

This case study also used the interoperability standard included in the DEVS M&S framework to execute this military scenario in Java and .NET DEVS-based simulators. The DEVS/SOA standard[21] embodies service oriented architecture (SOA) utilizing Web Service Description Language (WSDL) standard to describe the simulator and coordinator interfaces and SOAP standard to support communication operations between them. This allows the execution of models without local access to modeling components. The detailed execution is described in the text by Mittal and Martin.[4]

6.3.1 | Impact of xDEVS Architecture and features

This use case was one of the main applications of the xDEVS Application Programming Interface (APIs) distributed simulation application, first reported as a chapter in Reference 4. As mentioned above, the distributed simulation was initially

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

supported by the SOAP standard, using the concept of Simulation as a Service (SaaS), with memoryless atomic models and the server-side storing the state of the atomic models. This was essential to compose a fully operative distributed simulation with persistent state capability. After the evolution of Cloud computing and availability of Infrastructure as a Service (IaaS) from providers like Amazon, Google, or Microsoft, the xDEVS distributed architecture evolved into a much simple architecture using traditional sockets and automation in containerized simulation deployment using Docker. The current design described in Appendix D facilitates the distributed deployment of simulations using cloud services. The performance analysis in a cloud environment will be reported in our future work. Additionally, the complexity of these UAV-based scenarios forced us to leverage the xDEVS profiling framework (see Appendix E) and debugging mechanisms, to mitigate model bottlenecks, establish constraints and conduct unit testing.

6.4 | Synopsis

This section described the three most significant case studies that were developed using xDEVS infrastructure over a decade. Each case study highlighted the impact of xDEVS architecture and the xDEVS feature set that was used in the project. Actually, it was the case study itself that put a forcing function to advance the xDEVS feature set! The current feature set described in Section 3 provides a robust capability set for an M&S framework that can contribute to digital twin engineering in a cloud-based IoT environments that can support interoperability between heterogeneous components. The full set of current xDEVS features are enumerated in Table 2 and provide robust testing and evaluation of model execution to optimize the model design, which eventually will be traced back to the physical system design requirements.

7 | DISCUSSION

Modeling and simulation are distinct activities. Modeling facilitates understanding of phenomena (both natural and artificial) and helps develop an understanding (both personal and shared). This understanding when coupled with traditional systems and software engineering practices gave way to the development of MBSE in its current state. Simulation subsumes modeling, that is, simulation is operational only when there exists a model to execute on a platform (e.g., mental, collaborative, computational). This execution affords experimentation with the model and provides opportunities to experience the "model" in various settings (e.g., Live, Virtual and Constructive environments. MBSE without simulation, henceforth, involves effort spent in the development of only the model. The model may or may not be executable. It is certainly not simulatable.

In System of Systems (SoS) engineering or complex systems engineering (CSE) settings, due to a large number of stakeholders, this activity takes on a whole new meaning where developing a shared understanding is an achievement in itself. IT-enabled modeling environments and tools commercially available (e.g., IBM Rhapsody, NoMagic Cameo) provide the needed centralized repository and model editing environments to facilitate model development. The prime objective of this activity is to bring the stakeholders on the same page. In this regard, MBSE can exploit the immersive powers of storytelling to convey an evolving system design and concept of operations to technically unsophisticated stakeholders.

Between the MBSE without simulation and MBSE with simulation is the realm of executable models. Formal methods are applied in this model, which lead to software implementation. This enables testing and verification of systems under investigation during the model runtime, that is, the dynamic behavior in a simulation execution. While they are not supported by experimentation infrastructure, indeed they do allow experience with the system under study. MBSE with simulation affords experimentation and experience with the model. Simulation engineering requires an advanced computer science theory, methods and techniques to provide a computational substrate for the model to execute. When simulation engineering is coupled with systems theory and software engineering to develop the computational platform, we get a composable M&S platform. Application of DEVS theory and Software/Systems Engineering principle to develop xDEVS framework is such an example. In SoS, CSE and so forth. settings, the computational platform becomes an explicit engineering exercise as new domains are brought in the simulation environment. The prime objective of this activity is to experiment with the model and gain experience in understanding model's behavior. Combining M&S with MBSE can enhance the ability of models in virtual worlds to foster discovery of previously unknown interactions and dependencies among system elements and between the system and the environment.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

MBSE, even with simulation, is inadequate to support complex systems engineering. Complexity Science principles incorporating concepts like nonlinearity, emergent behavior, network connectivity and so forth. is being brought in to augment MBSE practices with DEVS[4,14,38] and all the case studies described in Section 6 demonstrate a comprehensive methodology for their application to next generation complex systems such as Internet of Things (IoT), Cyber Physical Systems (CPS) and Military wargaming.

8 | CONCLUSIONS

xDEVS is a modeling and simulation framework that allows the specification and execution of models that conform to the DEVS formalism. This formal definition improves the quality of the models that describe discrete event dynamic systems, and reduces the ambiguity in specification, development times and release cycles. This facilitates model reusability and sharing, leading to building complex hierarchical, modular and interoperable models.

xDEVS provides a single API specification and three different implementations with equivalent syntax, allowing the definition of models that use three of the most well-known and popular object-oriented programming languages (C++, Java, and Python). Moreover, it includes several verification tools and additional features that improve both the modeling and simulation experience.

We described the xDEVS API and architecture in detail with some representative code examples and case studies. As each xDEVS implementation adheres to the abstract simulator concept, the framework includes several interchangeable Coordinator definitions based on the same base Coordinator interface. This makes the process of changing between sequential, parallel, and real-time simulation execution, without the need for changing the model. Indeed, this is the hallmark of the DEVS theory that the same model is can be tested and evaluated in sequential, parallel, real-time, faster than real-time or distributed or any combination of such coordinators.

On the performance side, the experiments conducted on different xDEVS engines have shown that aDEVS is still the fastest DEVS simulation engine. However, xDEVS/C++ offered equivalent execution times, reaching better peak performance values as the DEVStone instances became more complex. Similarly, xDEVS/Java produced comparable peak performance values in DEVStone large HO models. Given the broad set of xDEVS features, this makes our proposed simulation toolkit an excellent alternative for interoperable modeling and simulation of formal discrete event systems.

Over the years, xDEVS has been used to simulate a great variety of applications in disparate domains. A sample of these case studies was briefly discussed that provide the evidence about the flexibility and robustness of both the formalism and its implementation, and how this M&S framework improves the development workflows by allowing the construction of safe, fully instrumented and profiled complex system models. We discussed three case studies that have been real-world projects that leverage the advanced xDEVS API, feature set and infrastructure. All the three projects have been reported in literature and are in continuous development.

MBSE in its current state is very much tied to traditional systems engineering and needs to be expanded to incorporate complex systems engineering practices, latest software modeling and engineering practices, and formal and rigorous M&S infrastructure engineering. It seems clear that we need to get a better handle on the whole SoS life-cycle with a more deliberate combined MBSE/DEVS approach. xDEVS integration with DEVSML Studio and the DEVS Unified Process[4] extends its feature set with the broader system of systems engineering discipline. xDEVS is available at a public repository[15] under the GNU GPL license and contains an example models set that can be used to facilitate the learning of model definition.

This paper summarizes the decade long work on xDEVS that began around 2011 (during our first book[4]) and reports its current feature set, its impact on significant case studies, toolsets that were developed along the way and its potential impact to multiple communities, especially, the DEVS Community and the MBSE Community by demonstrating the applicability of DEVS constructs to engineer hardware/software M&S environments that are open to fundamental V&V methods and large scale model composability.

AUTHOR CONTRIBUTIONS

Conceptualization, methodology, writing-original draft preparation, **José L. Risco-Martín** and **Kevin Henares**; validation, **Román Cardenas and Patricia Arroba;** formal analysis, **José L. Risco-Martín and Saurabh Mittal;** investigation, **Saurabh Mittal and Román Cardenas;** resources, funding and acquisition, project administration,

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

José L. Risco-Martín and Patricia Arroba; review and editing, **José L. Risco-Martín, Saurabh Mittal, Román Cardenas, and Kevin Henares;** supervision, **Saurabh Mittal and Patricia Arroba;** all authors have read and agreed to the published version of the manuscript.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

José L. Risco-Martín https://orcid.org/0000-0002-3127-6507
Kevin Henares https://orcid.org/0000-0002-2637-5316

REFERENCES

1. Srba J. Comparing the expressiveness of timed automata and timed extensions of petri nets. *International Conference on Formal Modeling and Analysis of Timed Systems*; 2008:15-32.
2. Zeigler BP, Muzy A, Kofman E. *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*. *Academic Press*; 2018.
3. Xie K, Zhang L, Laili Y, Wang X. XDEVS: a hybrid system modeling framework. *Int J Model Simulat Sci Comput*. 2022;13(2):2243001. doi:10.1142/S1793962322430012
4. Mittal S, Martin JLR. *Netcentric System of Systems Engineering with DEVS Unified Process*. 1st ed. CRC Press; 2013.
5. Mittal S, Risco-Martín JL. DEVSML 3.0 stack: rapid deployment of DEVS farm in distributed cloud environment using microservices and containers. *Proceedings of the 2017 Spring Simulation Multiconference*; 2017:19:1-19:12.
6. Mittal S, Risco-Martín JL. DEVSML studio: a framework for integrating domain-specific languages for discrete and continuous hybrid systems into DEVS-Based M&S Environment. *Proceedings of the 2016 Summer Simulation Multiconference*; 2016.
7. Nutaro J. ADEVS (a discrete EVent system simulator). *Arizona Center for Integrative Modeling & Simulation (ACIMS)*. University of Arizona. Available at: http://www.ece.arizona.edu/nutaro/index.php; 1999.
8. Wainer G. CD++: a toolkit to develop DEVS models. *Soft Pract Exp*. 2002;32(13):1261-1306.
9. Cárdenas R, Henares K, Arroba P, Risco-Martín JL, Wainer GA. The DEVStone metric: performance analysis of DEVS simulation engines. *ACM Trans Model Comput Simul*. 2022;32(3):1-20. doi:10.1145/3543849
10. Bolduc JS, Vangheluwe H. *A Modeling and Simulation Package for Classic Hierarchical DEVS*. MSDL, School of Computer McGill University, Tech. Rep; 2002.
11. Sarjoughian HS, Zeigler B. DEVSJAVA: basis for a DEVS-based collaborative M&S environment. *Simulat Series*. 1998;30:29-36.
12. Seo C, Zeigler BP, Coop R, Kim D. DEVS modeling and simulation methodology with MS4 me software tool. *SpringSim (TMS-DEVS)*; 2013:33.
13. Quesnel G, Duboz R, Ramat E. The virtual laboratory environment – an operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simulat Modell Pract Theory*. 2009;17:641-653.
14. Zeigler B, Mittal S, Traoré M. MBSE with/out simulation: state of the art and way forward. *Systems*. 2018;6:40. doi:10.3390/systems6040040
15. Risco-Martín JL. *xDEVS: A Cross-Platform Discrete Event System Simulator*; 2014 Accessed at: https://github.com/iscar-ucm/xdevs.
16. Chow AC, Zeigler BP, Kim DH. Abstract simulator for the parallel DEVS formalism. *Fifth Annual Conference on AI, and Planning in High Autonomy Systems*; 1994:157-163.
17. Martin A, Marangozova-Martin V. Automatic benchmark profiling through advanced workflow-based trace analysis. *Soft Pract Exp*. 2018;48(6):1195-1217.
18. Cárdenas R, Henares K, Arroba P, Wainer G, Risco-Martín JL. A DEVS simulation algorithm based on shared memory for enhancing performance. *2020 Winter Simulation Conference (WSC)*; 2020:2184-2195.
19. Henares K, Risco-Martín JL, Zapater M. Definition of a transparent constraint-based modeling and simulation layer for the management of complex systems. *2019 Spring Simulation Conference (SpringSim)*; 2019:1-12.
20. Henares K, Risco-Martín JL, Ayala JL, Hermida R. Unit testing platform to verify DEVS models. *Proceedings of the 2020 Summer Simulation Conference*; 2020:1-11.
21. Mittal S, Risco-Martín JL, Zeigler BP. DEVS-based simulation web services for net-centric T&E. *Proceedings of the 2007 Summer Simulation Multiconference*; 2007:357-366.
22. Moreno A, Risco-Martín JL, Besada E, Mittal S, Aranda J. DEVS/SOA: towards DEVS interoperability in distributed M&S. *2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*; 2009:144-153.
23. Zeigler BP. Closure under coupling: concept, proofs, DEVS recent examples (WIP). *Proceedings of the 4th ACM International Conference of Computing for Engineering and Sciences*; 2018:1-6.
24. Martin J, Henares K, Mittal S, Almendras L, Olkoz K. A unified cloud-enabled discrete event parallel and distributed simulation architecture. *Simulat Modell Pract Theory*. 2022;118:102539.
25. Risco-Martín JL, Mittal S. *Cloud-Based M&S for Cyber-Physical Systems Engineering*. Springer; 2020:3-23.
26. Risco-Martín JL, Mittal S, Fabero Jiménez JC, Zapater M, Hermida Correa R. Reconsidering the performance of DEVS modeling and simulation environments using the DEVStone benchmark. *Simulation*. 2017;93(6):459-476.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

27. Glinsky E, Wainer G. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments. *Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications*; 2005:265-272.
28. Franceschini R, Bisgambiglia PA, Touraille L, Bisgambiglia P, Hill D. A survey of modelling and simulation software frameworks using discrete event system specification. *2014 Imperial College Computing Student Workshop*; 2014.
29. Van Tendeloo Y, Vangheluwe H. The modular architecture of the python (P) DEVS simulation kernel. *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*; 2014:387-392.
30. Penas I, Zapater M, Risco-Martín JL, Ayala JL. SFIDE: a simulation infrastructure for data centers. *Proceedings of the Summer Simulation Multi-Conference*; 2017;1-12.
31. Cárdenas R, Arroba P, Blanco R, Malagón P, Risco-Martín JL, Moya JM. Mercury: a modeling, simulation, and optimization framework for data stream-oriented IoT applications. *Simulat Modell Pract Theory*. 2020;101:102037.
32. Pagán J, Moya JM, Risco-Martín JL, Ayala JL. Advanced migraine prediction simulation system. *Proceedings of the Summer Simulation Multi-Conference*; 2017:24.
33. Henares K, Risco-Martín JL, Hermida R, Roselló GR, Cárdenas R. Modular framework to model critical events in stroke patients. *Proceedings of the 2019 Summer Simulation Conference*; 2019:48.
34. Pérez-Vilarelle L, Risco-Martín JL, Ayala JL. Modeling and simulation of wind energy production in the smart-grid scenario. *Proceedings of the Symposium on Modeling and Simulation of Complexity in Intelligent, Adaptive and Autonomous Systems*; 2018:2.
35. Henares K, Pagán J, Ayala JL, Zapater M, Risco-Martín JL. Cyber-physical systems design methodology for the prediction of symptomatic events in chronic diseases. *Complexity Challenges in Cyber Physical Systems: Using Modeling and Simulation (M&S) to Support Intelligence, Adaptation and Autonomy*; 2019:223-253.
36. Henares K, Pagan J, Ayala JL, Risco-Martín JL. Advanced migraine prediction hardware system. *Proceedings of the 2018 Summer Simulation Multionference*; 2018.
37. Cárdenas R, Arroba P, Moya JM, Risco-Martín JL. Multi-faceted Modeling in the analysis and optimization of IoT complex systems. *Proceedings of the 2020 Summer Simulation Conference*; 2020.
38. Mittal S. Emergence in stigmergic and complex adaptive systems: a formal discrete event systems perspective. *Cognitive Syst Res*. 2013;21:22-39.

How to cite this article: Risco-Martín JL, Mittal S, Henares K, Cardenas R, Arroba P. xDEVS: A toolkit for interoperable modeling and simulation of formal discrete event systems. *Softw: Pract Exper*. 2023;53(3):748-789. doi: 10.1002/spe.3168

APPENDIX A. XDEVS API IMPLEMENTATION EXAMPLE FOR EFP MODEL

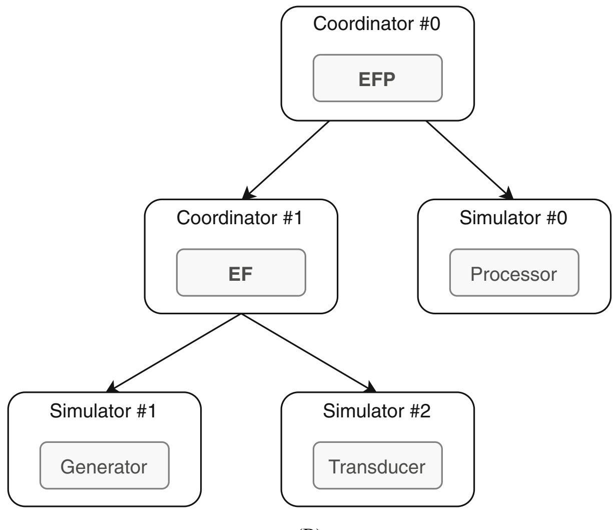
Figure A1 depicts how these layers are structured using the Experimental Frame - Processor (EFP) model as an example. The EFP model is a common reference model where a `Generator` component generates `Jobs` with an specific period. The `Processor` receives these `Jobs` and simulates some internal processing. Usually, the generation time is configured to be less than the processing time in this model, and the `Processor` only accepts `Jobs` when it is in the idle state. The `Transducer` is the component in charge of counting the number of generated and processed jobs and computing the ratio of processed jobs. As shown in Figure A1A, the `Generator` and the `Transducer` are grouped in the `EF` coupled component. Also, there is a root coupled model that contains the `EF` and the `Processor` components. This modeling hierarchy is also followed in the simulation layer (as can be seen in Figure A1B). A coordinator is created for each coupled component, and a simulator is created for each atomic component. The arrows depict the dependencies among the simulation entities. These dependencies between coordinators and simulators are the same as those expressed between the coupled and atomic components of the model.

In the following, we present the implementations of the `Processor` component of the `EFP` example (shown in the previous section) for the three available xDEVS Application Programming Interface (APIs). This component starts with a *passive* state (set in the *initialize* method) and waits until a `Job` arrives at its input port. When that happens, the external event method is activated. In this method, if the `Processor` is idle at that simulation time, the component changes its state to *active* and sets its sigma to the processing time. All the jobs received while the `Processor` is in this state are discarded. When the time specified in sigma is consumed, the output function (*lambda*) is activated, and the `Job` is sent through the output port. Right after that, the internal event method is invoked, which changes the status to *passive* again, indicating that it is available for processing new jobs.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

EF-Processor (EFP)
Experimental Frame (EF)
Generator
Processor
Transducer

(A)



(B)

FIGURE A1 Experimental frame-processor (EFP) model. (A) Structure of the model, (B) hierarchy of simulators and coordinators

We can see the `Processor` in the Java branch of xDEVS in Listing 7. It can be seen how the ports are added to the atomic component in the constructor, and a *processingTime* parameter is received. This parameter is saved and used later in the external event method (*deltext*) to specify the duration of the *active* state. When this time is consumed, the *lambda* and *deltint* methods are called. First, *lambda* outputs the suitable values (in this example, the original input job), and then *deltint* calls the *passivate* method. This is a shortcut for specifying the *passive* phase with an infinity duration. In this way, the component only can be activated again due to an external event. It is worth noting that both the *active* and *passive* phases do not have any special behavior and are used in some auxiliary methods only for usability reasons.

In the Python implementation of Listing 8, we can see how both the structure and the nomenclature are equivalent to the Java one. However, the format of the method names is changed to snake case to comply with the well-accepted nomenclature conventions of the Python language. Also, the *lambda* output method is renamed to *lambdaf* to avoid overwriting the Python *lambda* keyword. In C++ (Listing 9), although it keeps the camel case nomenclature, for the API methods, it introduces the *Event* additional object for the message passing. This wrapper object creates a shared pointer to the memory address of the actual message to release it when it is no longer used, simplifying the memory management of the values.

```
public class Processor extends Atomic {

   protected Port<Job> in = new Port<>("in");
   protected Port<Job> out = new Port<>("out");
   protected Job currentJob = null;
   protected double processingTime;

   public Processor(String name, double processingTime) {
      super(name);
      super.addInPort(in);
      super.addOutPort(out);
      this.processingTime = processingTime;
   }

   @Override
   public void initialize() { super.passivate(); }
```

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
    @Override
    public void exit() {}

    @Override
    public void deltint() { super.passivate(); }

    @Override
    public void lambda() { out.addValue(currentJob); }

    @Override
    public void deltext(double e) {
        if (super.phaseIs("passive")) {
            currentJob = in.getSingleValue();
            super.holdIn("active", processingTime);
        }
    }
}
```

Listing 7: Atomic module definition in xDEVS (Java)

```
class Processor(Atomic):
    def __init__(self, name, proc_time):
        super().__init__(name)

        self.in = Port(Job, "in")
        self.out = Port(Job, "out")

        self.add_in_port(self.in)
        self.add_out_port(self.out)

        self.current_job = None
        self.proc_time = proc_time

    def initialize(self):
        self.passivate()

    def exit(self):
        pass

    def deltint(self):
        self.passivate()

    def deltext(self, e):
        if self.phase == PHASE_PASSIVE:
            self.current_job = self.in.get()
            self.hold_in(PHASE_ACTIVE, self.proc_time)

    def lambdaf(self):
        self.out.add(self.current_job)
```

Listing 8: Atomic module definition in xDEVS (Python)

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
class Processor : public Atomic {
protected:
      Event nextEvent;
      double processingTime;
public:
      Port in;
      Port out;
      Processor(const std::string & name, double processingTime):
          Atomic(name), nextEvent(), processingTime(processingTime), in("in"), out("out") {
          this->addInPort(&in);
          this->addOutPort(&out);
      }

      ~Processor() {}
      virtual void initialize() { Atomic::passivate(); }
      virtual void exit() {}
      virtual void deltint() { Atomic::passivate(); }
      virtual void lambda() { out.addValue(nextEvent); }

      virtual void deltext(double e) {
            if (Atomic::phaseIs("passive")) {
                  nextEvent = in.getSingleValue();
                  Atomic::holdIn("active", processingTime);
            }
      }
};
```

Listing 9: Atomic module definition in xDEVS (C++, header)

As an example of coupled components implementation in xDEVS, we show the definition of the `EFP` component shown in Figure A1A. In Listing 10 we can see Java implementation of this component. It does not have to implement special methods, so only the constructor is defined. The internal components (`Generator` and `Transducer`) are instantiated and added as part of the coupled component. After that, the suitable links are established using the *addCoupling* method. These actions are repeated in the Python (Listing 11) and C++ (Listing 12) versions with no remarkable changes.

```
public class Efp extends Coupled {

   public Efp(String name, double generatorPeriod, double processorPeriod, double transducerPeriod) {
      super(name);

      Ef ef = new Ef("ef", generatorPeriod, transducerPeriod);
      super.addComponent(ef);
      Processor processor = new Processor("processor", processorPeriod);
      super.addComponent(processor);

      super.addCoupling(ef.out, processor.in);
      super.addCoupling(processor.out, ef.in);
   }
}
```

Listing 10: Coupled module definition in xDEVS (Java)

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
class Efp(Coupled):
   def __init__(self, name, generator_period, processor_period, transducer_period):
      super().__init__(name)

      ef = EF("ef", generator_period, transducer_period)
      proc = Processor("processor", processor_period)

      self.add_component(ef)
      self.add_component(proc)

      self.add_coupling(ef.out, proc.in)
      self.add_coupling(proc.out, ef.in)
```

Listing 11: Coupled module definition in xDEVS (Python)

```
class Efp : public Coupled {
protected:
      Ef ef;
      Processor processor;
public:
      Efp(const std::string& name, const double& generatorPeriod, const double& processorPeriod,
          const double& transducerPeriod): Coupled(name),
            ef("ef", generatorPeriod, transducerPeriod),
            processor("processor", processorPeriod) {
          Coupled::addComponent(&ef);
          Coupled::addComponent(&processor);
          Coupled::addCoupling(&ef, &ef.out, &processor, &processor.in);
          Coupled::addCoupling(&processor, &processor.out, &ef, &ef.in);
      }

      ~Efp() {}
}
```

Listing 12: Coupled module definition in xDEVS (C++, header)

APPENDIX B. EXAMPLE FOR COORDINATOR API FOR EFP MODEL

The Coordinator API is invoked for EFP model for the three languages: Java, C++, and Python (Listings 13-15).

```
Efp efp = new Efp("efp", 1, 3, 1000);
Coordinator coordinator = new Coordinator(efp);
coordinator.initialize();
coordinator.simulate(Long.MAX_VALUE);
coordinator.exit()
```

Listing 13: Launching a simulation in xDEVS (Java)

```
efp = Efp("efp", 1, 3, 1000)
coord = Coordinator(efp)
coord.initialize()
coord.simulate_time(INFINITY)
coord.exit()
```

Listing 14: Launching a simulation in xDEVS (Python)

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
Efp efp("efp", 1, 3, 1000);
Coordinator coordinator(&efp);
coordinator.initialize();
coordinator.simulate((long int)10000);
coordinator.exit();
```

Listing 15: Launching a simulation in xDEVS (C++)

APPENDIX C. EXAMPLE WRAPPER FOR ATOMIC ADEVS

Listing 16 shows the aDEVS wrapper. This wrapper wraps the aDEVS model for execution with xDEVS simulation engine.

```
AtomicADEVS::AtomicADEVS(const std::string& name,
                     adevs::Atomic<PortValue>* model,
                     const std::list<int>& in_ports,
                     const std::list<int>& out_ports) : Atomic(name) {
  this->model = model;
  for(int port : in_ports) {
    Component::addInPort(new Port(std::to_string(port)));
  }
  for(int port: out_ports) {
    Component::addOutPort(new Port(std::to_string(port)));
  }
}

AtomicADEVS::~AtomicADEVS() {
  delete model;
}

void AtomicADEVS::initialize() { }

void AtomicADEVS::exit() { }

double AtomicADEVS::ta() {
  double sigmaAux = model->ta();
  if (sigmaAux >= DBL_MAX) {
    sigmaAux = std::numeric_limits<double>::infinity();
  }
  return sigmaAux;
}

void AtomicADEVS::deltint() {
  model->delta_int();
}

void AtomicADEVS::deltext(double e) {
  adevs::Bag<PortValue> msg = buildMessage();
  model->delta_ext(e, msg);
}

void AtomicADEVS::deltcon(double e) {
  adevs::Bag<PortValue> msg = buildMessage();
  model->delta_conf(msg);
}

void AtomicADEVS::lambda() {
  adevs::Bag<PortValue> msg;
  model->output_func(msg);
```

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
  std::list<Port *> ports = this->getOutPorts();

  for(auto port_adevs : msg) {
    for(auto port_xdevs : ports) {
      if(port_adevs.port==std::stoi(port_xdevs->getName())) {
        port_xdevs->addValue(port_adevs.value);
      }
    }
  }
}

adevs::Bag<PortValue> AtomicADEVS::buildMessage() {
  adevs::Bag<PortValue> msg;
  std::list<Port *> ports = getInPorts();

  for(auto port : ports) {
    const std::string& port_name = port->getName();
    const std::list<Event>& events = port->getValues();
    for(auto event : events) {
      PortValue pv(std::stoi(port_name), event);
      msg.insert(pv);
    }
  }
  return msg;
}
```

Listing 16: aDEVS wrapper implemented in xDEVS/C++

APPENDIX D. COORDINATOR INVOCATIONS FOR SEQUENTIAL, REAL-TIME, PARALLEL AND DISTRIBUTED EXECUTION FOR EFP

The following code excerpt shows how straightforward it is to define a sequential, real-time, and parallel coordinator and perform simulation through the respective paradigm. The real-time coordinator has a time-scale attribute that can be modified to change the resolution of the simulation clock, defining time in seconds, milliseconds and so forth. The parallel coordinator also uses by default a number of threads equal to the number of cores in the simulation entity.[‡] In an xDEVS parallel simulation, the transition functions and the output functions are equally distributed among all the threads and executed in parallel. The modeler can always create particular mechanisms to distribute transition and output functions better (Listing 17).

```
// Sequential coordinator
coordinator = new Coordinator(new Gpt("gpt", 1, 100));
coordinator.initialize();
coordinator.simulate(Long.MAX_VALUE);
coordinator.exit();
// Real-time coordinator
coordinator = new RTCentralCoordinator(new Gpt("gpt", 1, 100));
coordinator.initialize();
coordinator.simulate(Long.MAX_VALUE);
coordinator.exit();
// Parallel coordinator
coordinator = new CoordinatorParallel(new Gpt("gpt", 1, 100)); // The number of threads is equal to
coordinator.initialize();                                      // the number of cores by default.
coordinator.simulate(Long.MAX_VALUE);
coordinator.exit();
```

Listing 17: Use of sequential, real-time, and parallel coordinators in xDEVS

[‡]With entity we refer to a computer, virtual machine, container and so forth. Any virtual or physical device able to simulate an xDEVS model.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

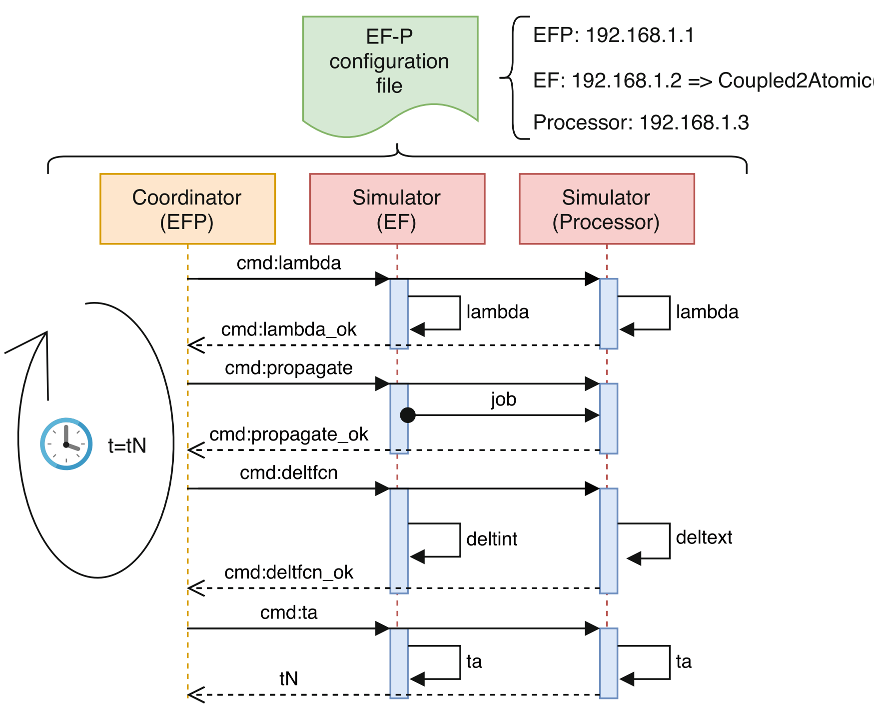


FIGURE D1 Sequence diagram of the xDEVS distributed simulation

In the distributed simulation, every simulation entity must have installed the xDEVS simulation engine and at least the part of the model being simulated into each entity. Each simulation entity is identified by its IP address. A text configuration file defines which entity is simulating each atomic or coupled model. An entity can simulate one or more atomic models. If a coupled model is linked to an IP address, that model is transformed into an atomic model using the `Coupled2Atomic` class and simulated with a `SimlutadorDistributed` class. Otherwise, the coupled model is flattened and distributed among the simulation entities specified in the atomic models through their IP addresses.

Figure D1 illustrates a brief description of the distributed simulation sequence. The configuration file is uploaded to each simulation entity. In the particular example of Figure D1, the experimental frame processor (EFP) model[2] is being simulated, where the experimental frame (labeled as EF) is located at simulation entity 192.168.1.2. Thus, the Coupled2Atomic wrapper encapsulates this coupled model that is handled as an atomic model. Next, the processor is being simulated at the simulation entity 192.168.1.3. Three different processes are executed independently, in an infinite loop. Each simulator is waiting for commands coming from the coordinator.

First, the coordinator commands, via sockets, the execution of the output function. Each simulator listens to this command and runs the output function of their respective atomic models.[§] Second, the coordinator commands the propagation of the output, sent and executed by all the simulators. After that, the execution of the transition function is requested, and each simulator tests if the transition function must be the external, internal, or confluent function, depending on the current simulation time and the state at the input ports. Finally, the next time event (*tN* in Figure D1) is requested to start the DEVS simulation loop again. This is executed until the number of DEVS iterations is reached, or all the models enter into a *passive* state (i.e., $\sigma = \infty$).

APPENDIX E. EXAMPLE FOR PROFILING A DEVS SIMULATION

Table F1 illustrates an example. It shows the results of a profiling report of a complex Search And Rescue (SAR) mission model. In these cases, data is grouped by atomic models that belong to the same modeling class.

[§] λ is executed if and only if the simulation clock is equal to the next time event, accordingly to the DEVS formalism.

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

APPENDIX F. EXECUTION OF CHAINED ALGORITHM FOR SPEEDUP USING SHARED MEMORY

Figure F1 illustrates how the chained algorithm works with the EFP model. Figure F1A depicts the EFP model, while Figure F1B represents the chained simulator manages the system's memory.

If the Generator model outputs a new event when executing its output function λ, this event is written in a memory region reserved for this model's port. Figure F1B represents this memory region as *Gdata*. The simulator in charge of the Generator model returns to its parent coordinator a reference to every nonempty output port of its atomic model. For this example, Figure F1B depicts this reference as G_{out}, which is pointing to the *Gdata* memory region. Then, the coordinator in charge of the Experimental Frame (EF) coupled model creates chained references to G_{out} according to the internal and external output couplings which source port is G_{out}. In this example, there is an internal coupling to the *arrived* port of the Transducer model and an external output coupling to the *out* port of the EF model. These chained references are represented by $T_{arrived}$ and EF_{out} in Figure F1B. Chained references of external output couplings are forwarded to the parent coordinator. In this way, the coordinator in charge of the EFP model (i.e., the root coordinator) is able to create a chained reference for the input port of the Processor model (P_{in} in Figure F1B). By doing so, the Processor model will be able to read output events directly from the original source by simply resolving the chained references to *Gdata*.

TABLE F1 Profiling of a simulation, grouped by class

Class	#	$t(\delta_*)$	$t(\lambda)$	$t(\delta + \lambda)$	%
DynamicSensorPayload	100	1849.408	5.933	1855.341	34.0
DynamicTargetControl	50	2774.442	6.276	2780.718	50.0
EvaluatorFunction	50	274.078	2.007	276.085	5.0
SensorMotion	100	15.300	12.763	28.063	1.0
StaticSensorPayload	50	315.514	2.580	318.094	6.0
TargetMotion	50	9.594	114.581	124.175	2.0
UavControl	100	26.886	4.953	31.839	1.0
UavMotion	100	16.523	46.070	62.593	1.0

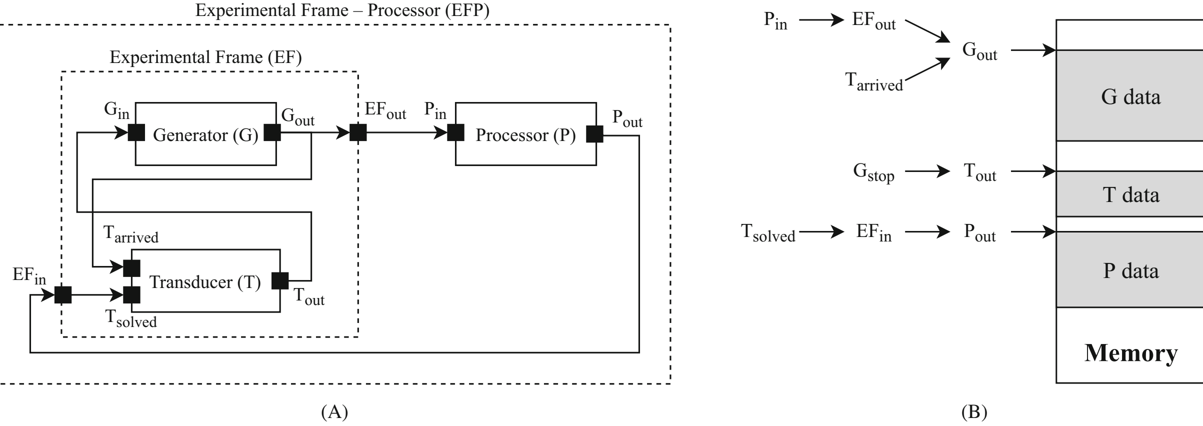


FIGURE F1 Example of memory management in the chained DEVS simulator. (A) Experimental frame-processor model, (B) chained simulator memory management

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

APPENDIX G. DEVSTONE PERFORMANCE DATA

For measuring the performance of xDEVS we ran all the DEVStone models with a wide range of depth and width for each xDEVS implementation. These experiments were run sequentially in a n1-standard-1 Google Cloud Platform virtual machine, with Debian 10, Intel(R) Xeon(R) CPU @ 2.30 GHz, and 3.75 GiB RAM. For LI, HI, and HO, the combinations from 100 to 1200 with step 50 have been generated for both of the parameters. The resulting simulation times have been compared against those generated by a DEVStone implementation developed in the aDEVS simulator. Table G1 shows information of the used engines and environments. Specifically, it includes the engine version and programming language, as well as the interpreters or compilers used to run or compile the DEVStone implementations. Also, it is worth noting that these simulation times does include the model creation and engine set-up times, evaluating all the aspects of the simulation engine.

Table G2 shows the performance values for specific balanced configurations, for each model type and implementation. As can be seen xDEVS/C++ offers similar execution times to aDEVS, whereas xDEVS/Java is slower, typically because the Java Virtual Machine demands more resources and have an extra loading time. However, this deficit is compensated with the parallel and distributed facilities of the Java programming languages. Finally, the xDEVS/Python engine is the slowest one, since this is an interpreted language, mainly. However, the Python version also offers some advantages like its facilities to use a distributed simulation or the huge amount of resources provided by Python libraries for data analysis.

In order to have a single metric comparing all the simulation engines, Table G3 show the averages for each DEVStone model class. The conclusions are similar to those obtained with the previous table, but with a more condensed information. aDEVS is the fastest simulation engine, closely followed by xDEVS/C++, and then by xDEVS/Java and finally xDEVS/Python.

TABLE G1 Engine versions and environments used for the DEVStone simulations

Engine	Version	Programming language	Interpreter/Compiler
adevs	3.3	C++17	g++ 7.5 (-o3)
xDEVS (1)	1.0.0	C++11	g++ 7.5 (-o3)
xDEVS (2)	1.1.0	Java	OpenJDK 11.0.7
xDEVS (3)	1.1	Python3	CPython 3.6.9

TABLE G2 Simulation times for specific DEVStone balanced configurations

Model	Depth	Width	aDEVS	xDEVS/C++	xDEVS/JAVA	xDEVS/Python
LI	400	400	0.27	0.35	1.60	2.16
	600	600	0.63	0.79	3.17	4.80
	800	800	1.12	1.83	4.84	8.49
	1000	1000	1.74	2.21	7.57	13.08
	1200	1200	2.55	3.27	9.24	18.97
HI	400	400	36.34	49.37	72.09	332.21
	600	600	128.06	258.26	249.30	1088.55
	800	800	338.94	431.46	795.57	2649.91
	1000	1000	745.52	826.01	1272.91	5312.32
	1200	1200	1462.11	1463.87	2340.96	7879.57
HO	400	400	60.43	62.12	81.12	351.48
	600	600	185.54	213.82	274.17	1197.56
	800	800	392.61	558.46	712.02	2864.07
	1000	1000	1252.72	1030.42	1401.38	6554.49
	1200	1200	2553.18	2004.62	2613.55	6554.49

TABLE G3 Simulation time averages for all the DEVStone configurations

Simulator	LI	HI	HO
aDEVS	0.77	269.51	318.47
xDEVS/C++	0.96	304.02	375.49
xDEVS/Java	3.39	443.31	483.18
xDEVS/Python	5.57	1853.30	1942.80

APPENDIX H. UAVS DEVS MODEL

For this use case, each example is constructed upon multiple DEVS atomic components that exemplify UAV dynamics and behavior, together with multiple DEVS coupled components that characterize the line of action of an ADU. This Appendix shows the Finite State Machine (FSM) design of two key components: UAV and Missile.

UAVs are models that receive states of tracking radars and missiles of each ADU correspondingly. They also send their state out. A UAV keeps an internal state variable with an array of these UAV states reflecting the UAV dynamics. Whenever a UAV realizes that it has been detected by an ADU, if possible, it starts an evasive maneuver to escape from the ADU firepower range and prevent being shot down. Basically, the UAV FSM model works in the following way. Every time the internal time event function is triggered (simulation time is equal to sigma), the next necessary collection of states is computed, unless this collection is not empty or the UAV has reached the end of the trajectory. These states store intermediate values of position, orientation, and velocity that describe the UAV's movement across the current coordinate to the next trajectory point. Then, sigma (time of next internal time event) is updated to the next computed time or set to infinity only if the UAV reached the end of the assign path. Whenever the external transition is executed (received an input), the UAV verifies if any radar is tracking its path and whether the distance from a missile aimed at overthrowing it is less than the established minimum. If the former case is positive, the UAV attempts to escape through an intersecting trajectory to flee away from the corresponding ADU and afterward updates sigma. If the latter is positive and according to a certain probability of destruction, the UAV is destroyed and sigma is set to infinity. On every occasion that the output function is activated, the current UAV state is sent thought the output port. Figure H1A and Table H1 depict this behavior.

As an illustrative example, Listing 18 shows the source code of the internal xDEVS transition function.

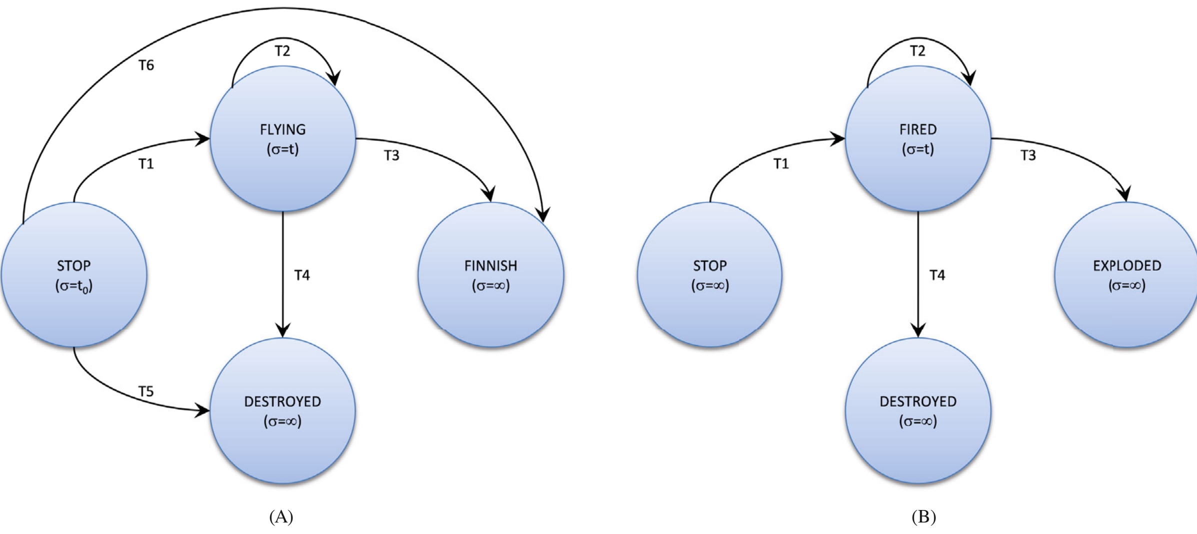


FIGURE H1 Example of (A) UAV and (B) Missile state diagrams

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE H1 UAV State Transitions

	S_k	S_{k+1}	δ_{int}	δ_{ext}
T1	STOP	FLYING	States is NOT empty	
T2	FLYING	FLYING	States is NOT empty	Dist. to missile is > MIN
T3	FLYING	FINISH	States is empty	
T4	FLYING	DESTROYED		Dist. to missile is ≤ MIN
T5	STOP	DESTROYED		
T6	STOP	FINISH	States is empty	

```
public void deltint() {
  boolean finish = false;
  if (super.phaseIs(phases[FINISH]))
    finish = true;
  if (uavStates.isEmpty() && !finish)
    finish = uavModel.update(uavStates, refUavState.t, dt, refUavState);
  if (!uavStates.isEmpty()) {
    double t0 = refUavState.t;
    refUavState = uavStates.remove();
    if (finish)
      super.holdIn(phases[FINISH], refUavState.t - t0);
    else
      super.holdIn(phases[FLYING], refUavState.t - t0);
  }
  else
    super.holdIn(phases[FINISH], Constants.INFINITY);
}
```

Listing 18: UAV internal transition

Missile models accept states of UAVs intended to be blown down, and pass these UAVs' state to the next missile only if their status is fired, and another output port to communicate its state to the UAVs so that they can check whether they are destroyed or not. Like UAVs, they also keep an internal state variable with an array of missile states (same as UAVs) reflecting changes in time of the missile dynamics. Essentially, as seen in Figure H1B and Table H2, missiles wait for an external command from any tracking radar model of its corresponding ADU to shift from the initial state stop to be fired. Then, sigma is updated from infinity to the next immediate state time. Afterward, every time the internal time event function is triggered, it jumps to the next computed state. Sigma is updated to the next computed time, unless the array of states is empty and the missile has reached its goal or exceeded its limits, or the distance sigma is set to ∞.

TABLE H2 Missile state transitions

	S_k	S_{k+1}	δ_{int}	δ_{ext}
T1	STOP	FIRED		NO assigned target
T2	FIRED	FIRED	States is not empty	Assigned target
T3	FIRED	EXPLODED		Reached goal
T4	FIRED	DESTROYED		Exceeds limits or distance

1097024x, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3168 by Cochrane Canada Provision, Wiley Online Library on [09/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License