Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

A DEVS SIMULATION ALGORITHM BASED ON SHARED MEMORY FOR ENHANCING PERFORMANCE

Román Cárdenas

Laboratorio de Sistemas Integrados
Universidad Politécnica de Madrid
ETSI Telecomunicación, Avenida Complutense 30
Madrid 28040, Spain

Kevin Henares

Dpt. Of Computer Architecture and Automation
Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9
Madrid 28040, Spain

Patricia Arroba
Laboratorio de Sistemas Integrados
CCS-Center for Computational Simulation
Universidad Politécnica de Madrid
ETSI Telecomunicación, Avenida Complutense 30
Madrid 28040, Spain

Gabriel Wainer

Dept. Of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa ON K1S 5B6, Canada

José L. Risco-Martín

Dpt. of Computer Architecture and Automation
CCS-Center for Computational Simulation
Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9
Madrid 28040, Spain

ABSTRACT

The Discrete EVent System Specification (DEVS) formalism provides a unified method to define any discrete-event system accurately. As the complexity of the system under study increases, the necessity of simulation engines with higher performance rises. In this research, we present a *chained DEVS simulator*, a DEVS-compliant, function-oriented simulation algorithm that exploits shared memory patterns to improve the performance of sequential and parallel simulations. We also illustrate the positive impact of this novel approach executing a set of DEVStone synthetic benchmarks and comparing a state-of-the-art simulation engine with an updated version that implements the chained algorithm. Results show that the chained simulator introduces up to 40% less synchronization overhead than the traditional simulation approach.

1 INTRODUCTION

Multiple Modeling and Simulation (M&S) methodologies have emerged as a way to conduct preliminary studies of complex systems (Mittal and Tolk 2020). Nowadays, M&S is a common practice in science, technology, industry, and governance, reducing the capital expenses and potential hazards that testing with real systems may imply (Ullah 2019). The increasing complexity of the models under study places the definition of a formal method for model description as an important research field, as well as the optimization of simulation engines in terms of speed and energy consumption (Guérout, et al. 2013).

Although there are different modeling formalisms, the Discrete EVent System Specification (DEVS) (Zeigler, Praehofer and Kim 2000) and its Parallel DEVS (PDEVS) variant (Chow 1996) showed success in expressing any discrete-event formalism (Risco-Martín and Mittal 2019). PDEVS enables the description of a model as a hierarchy of submodels and their relationship, including numerous advantages (e.g., modularity, reusability, and shorter model description times).

The construction of models as a composite of submodels introduces the need for inter-model communication. State-of-the-art PDEVS-compliant simulation engines use message-passing patterns: models are managed by asynchronous, independent processors that follow a communication protocol for synchronization. While this architecture enables a simple distribution of simulations, it introduces unnecessary synchronization processing overheads in both sequential and parallel execution. This is especially relevant with complex models, where messages are sent through multiple levels of the hierarchy, producing a redundant propagation in the intermediate ports of the coupled components.

In this research, we introduce the *chained DEVS simulator*, a novel simulation algorithm for PDEVS that exploits the benefits of shared memory systems. The algorithm introduces a function-oriented design focused on reducing message-passing overhead. We discuss the potential benefits of the chained simulator by benchmarking a simulation engine that implements this algorithm. The simulation overhead can be reduced from 15 to 40%, depending on the structure of the model under study.

The paper is organized as follows. In Section 2, we provide a brief description of the PDEVS formalism and enumerate different PDEVS-compliant simulation engines. Additionally, we discuss their implementation patterns and procedures for benchmarking their performance. Section 3 presents the chained DEVS algorithm, a novel implementation pattern for PDEVS simulation engines based on shared memory for boosting the overall simulation performance in sequential and parallel simulation. We illustrate the benefits of our proposal in Section 4, comparing a simulation engine with an equivalent version that implements the chained algorithm. Finally, we present conclusions and future work in Section 5.

2 RELATED WORK

In this section, we first present the PDEVS formalism. We introduce different PDEVS simulation engines and identify their most common implementation patterns. We also discuss about the main proposals for analyzing and comparing the performance of DEVS environments.

2.1 DEVS and Parallel DEVS

The DEVS formalism (Zeigler, Praehofer and Kim 2000) provides a rigorous foundation for discrete M&S. DEVS allows the user to define a mathematical object (i.e., system) that represents an abstraction of real objects. PDEVS (Chow 1996) is a popular variant of the original formalism, which addresses some deficiencies of the original DEVS. In PDEVS, the behavior of a system can be described at two levels: atomic models, which describe the autonomous behavior of a system as a series of transitions between states and its reactions to external events, and coupled models, which describe a system as the interconnection of coupled components. The formal definition of an atomic model is described as the following:

$$A = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$$

An atomic model's state is $s \in S$ at any given time. If no external events occur, its state remains in s for a period of time $ta(s)$ (i.e., time advance function). When the lifetime expires, the atomic model sends a set of output events $y \in Y$ according to its output function λ(s), and changes its state to a new one given by the internal transition function $\delta_{int}(s)$. If one or more input events $x \in X$ occur before the expiration of $ta(s)$, the model changes to a new state determined by the external transition function $\delta_{ext}(s, e, x)$. The confluent transition function δ_{con} determines the next state in the case of collisions when a model receives external events at the same time of its internal transition.

On the other hand, the formal definition of a coupled model is described as follows:

$$M =< X, Y, C, EIC, EOC, IC >$$

where X is the set of inputs; Y is the set of outputs; C is the set of DEVS component models; EIC is the external input coupling relation, from external inputs of M to component inputs of $c_i \in C$; EOC is the external output coupling relation, from component outputs of $c_i \in C$ to external outputs of M; and IC is the internal coupling relation, from components outputs of $c_i \in C$ to component inputs of $c_j \in C$.

2.2 PDEVS Simulation Algorithms

There are multiple simulation tools defined according to the PDEVS formal specification. These engines are written in a wide variety of programming languages, like C++ (e.g., Cadmium (Belloli, Vicino and Ruiz-Martín 2019)), Java (e.g., xDEVS (Risco-Martín 2014)), or Python (e.g., PyPDEVS (Van Tendeloo and Vangheluwe 2014)). All these simulation engines are based on the same simulation algorithm: the PDEVS abstract simulator (Chow, Zeigler and Kim 1994). This approach proposes the use of independent, coordinating simulation engines that interchange messages to synchronize any parallel task to be distributed across asynchronous processors. In this abstract simulator, there are two types of simulation components: simulators and coordinators. Simulators are attached to atomic models. On the other hand, coordinators manage coupled models and are in charge of synchronizing their child simulators and coordinators (i.e., child processors). Abstract simulators exchange five synchronization messages:

- *(@, t)*: collection messages. Parent coordinators send these messages to imminent child processors (i.e., those processors whose next transition event is scheduled at time *t*) to execute the output function λ of their corresponding atomic model.
- *(q, t)*: external messages. They contain bags of input events to be forwarded (i.e., $x \in X$). Parent coordinators send these messages to receiver child processors.
- *(*, t)*: transition messages. Imminent child processors receive this message to execute the corresponding transition function (i.e., δ_{int}, δ_{ext}, or δ_{con}) of their atomic model.
- *(y, t)*: output messages. They contain bags of output events (i.e., $y \in Y$) to be forwarded from child processors to parent coordinators. Parent coordinators forward these messages to the corresponding processors according to the couplings defined in the model.
- *(done, t)*: done messages. Child processors send these messages to acknowledge their parent coordinator that they have finished processing a given pending task.

The communication between parent coordinators and child processors is always hierarchical, and follows a request-response fashion: parents send requests to their child processors, and child processors notify that they finished executing the requested action by responding with a done message.

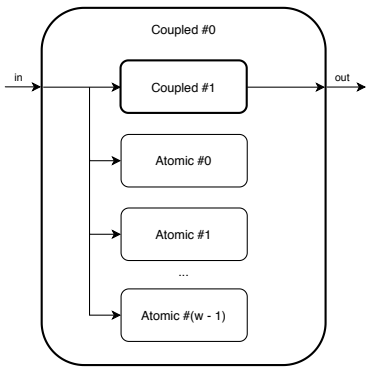
Simulation engines based on this abstract simulator include explicit definitions of ports and couplings, and each simulation cycle calls explicitly to functions in charge of propagating input/output events through the model. Hence, for each coupling in the system, the same values in source ports are copied multiple times until they reach the destination port. This approach is suitable for distributed simulation. In fact, multiple research work is focused on distributed architectures to enhance simulation performance (e.g., RISE (Wainer, et al. 2016) or the DEVSML 3.0 stack (Mittal and Risco-Martín 2017)). This message passing process, though necessary for distributed simulation, impose a heavy simulation overhead in both sequential and parallel simulation, where all the entities that participate in the simulation are hosted in the same physical machine. As the main loop of the simulation engine is constantly copying messages sources to consumers, the grouped propagation time results in a significant percentage of the overall execution time.

Multiple research work is focused on improving the performance of DEVS simulation engines (Uhrmacher, Himmelspach and Ewald 2018). For example, the flat DEVS simulator was defined (Kim, et al. 2000).This approach creates an equivalent DEVS model that removes all the intermediate coupled models. Although the new model behaves as the original, the complexity is reduced. Doing so, the root coordinator of the simulation engine is the parent coordinator of all the atomic models that describe the behavior of the system. This algorithm reduces the simulation synchronization overhead. However, messages are still copied depending on the number of port couplings.

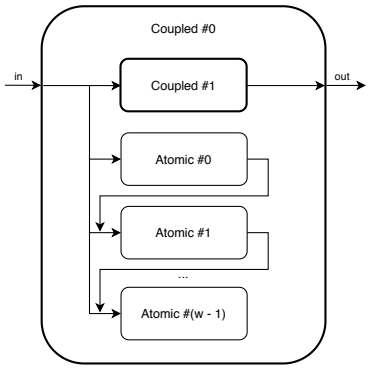
2.3 Measuring Discrete-Event Simulation Performance: the DEVStone benchmark

To measure and compare the performance of different PDEVS simulation engines, several comparison methods have been presented, commonly applied to specific applications. The DEVStone synthetic (Wainer et al. 2011) has been used to study a variety of DEVS engines. DEVStone describes several synthetic models with assorted sizes and complexities. They are defined as coupled models, containing a fixed structure which is recursively replicated in other children coupled components. This recursion ends with a simpler coupled model, that only contains an atomic component. All the models presented in DEVStone can be customized with four parameters: (i) *width*, that specifies the number of atomic components per layer, (ii) *depth*, that specifies the number of nested coupled models, (iii) *internal transition delay*, and (iv) *external transition delay*. The internal and external transition functions are programmed to execute a fixed amount of time specified in these delays. There are four types of models:

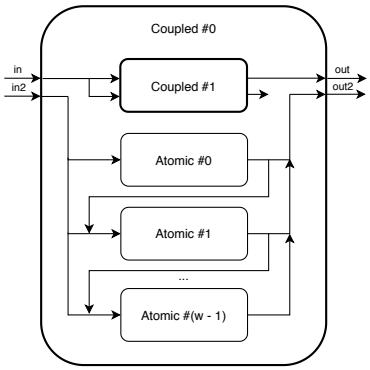
- LI models (Figure 1a). This model has the simplest structure, with a low level of interconnections for each coupled model.
- HI models (Figure 1b). These structures contain a higher level of internal couplings than LI models.
- HO models (Figure 1c). The number of ports, input and output couplings increases.
- HOmod models (Figure 1d). The number of ports of these models is like HO models. However, the number of atomic models, couplings, and outputs grows exponentially.



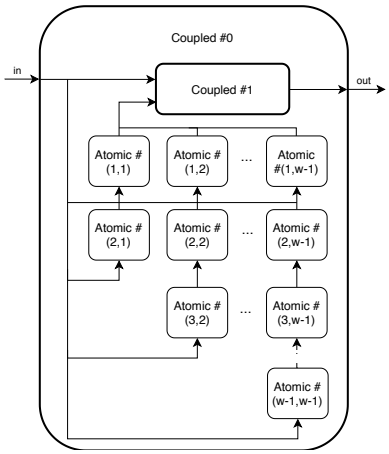
(a) LI Model



(b) HI Model



(c) HO Model



(d) HOmod Model

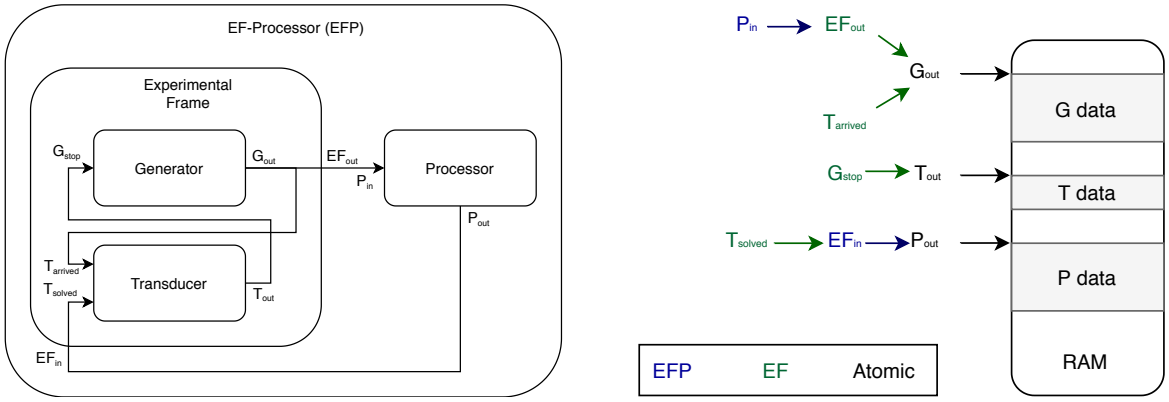
Figure 1: DEVStone models.

By profiling the execution of an arbitrary model generated using the DEVStone benchmark, we can have a sense of the simulation time spent just in propagating messages. For instance, in the xDEVS simulation engine, message propagation takes 39.61% of the total simulation time for a HO model of depth 300, width 10, and no internal nor external delays. Based on these results, in this research, we propose a chained simulation algorithm, which is compliant with the PDEVS formalism. In contrast with the classic message interchange-based abstract simulator, the chained simulation algorithm uses a reduced function set that avoids data propagation and enhances the simulation performance by carefully enabling autonomous, asynchronous processors to share the same memory space.

3 THE CHAINED SIMULATOR

In this section we present the chained simulation algorithm, an equivalent to the classic PDEVS abstract simulator that avoids the use of propagation functions. With the removal of couplings and the propagation needs, we update the abstract simulator concept (Chow, Zeigler and Kim 1994). As a result, we obtain a different function set and replace message transmission by shared memory mechanisms.

The main structure of the chained simulator is similar. The behavior of each atomic component is controlled by a simulator. The control flow in each coupled component is controlled by a coordinator. Hence, simulators and coordinators reproduce a hierarchy like the one reflected by the atomic and coupled components present in the model. Although all the components have ports, they now are defined accordingly to the aforementioned idea: only the output ports of atomic models store new events to be transmitted, while the rest of the ports only reference (directly or indirectly) them. In this way, when an external transition is activated, the input data is read from their original source. Figure 2 illustrates how memory management works in the chained simulation algorithm with a common example in the DEVS literature: the Experimental Frame-Processor (EFP) model.



(a) EFP model. (b) Memory management.

Figure 2: Example for illustrating memory management in the chained DEVS simulator.

New data is only generated in the output ports of the atomic models. Therefore, in the example shown in Figure 2, when the *Generator* atomic model produces a new output message via its output port G_{out}, this message is written in the memory zone assigned to the *Generator* model (in Figure 2b, *G data*). The simulator attached to this model creates a pointer to a memory zone that contains new data. This pointer is represented in Figure 2b as the black arrow G_{out} that is pointing to the memory zone *G data*. The simulator sends this pointer to its parent coordinator, which is in charge of the *Experimental Frame (EF)* coupled model. This coordinator resolves all the couplings according to the *IC* and *EIC* sets of the *EF* model. In case of the *IC*, the output port G_{out} is coupled to the input port $T_{arrived}$ of the *Transducer* model. The coordinator of the *EF* then creates a virtual memory pointer (in Figure 2b, the green arrow labeled as $T_{arrived}$)

that points to the memory position pointed by the pointer G_{out}.The pointer $T_{arrived}$ is sent to the simulator in charge of the *Transducer* model, so it can read the data that the *Generator* sent from the original memory zone *G data*. On the other hand, the external coupling between *Generator* and the output port EF_{out} of *EF* creates another virtual pointer (in Figure 2b, this pointer is represented as the green arrow labeled as EF_{out}). The coordinator of the *EF* model sends this pointer to its parent coordinator (the one in charge of the *EFP* model), which resolves the couplings and detects a coupling between the *EF* and *Processor* models. The coordinator generates another virtual pointer (shown as the blue arrow P_{in} in Figure 2b) that points to the same memory zone pointed by the virtual pointer EF_{out} and sends it to the simulator that controls the *Processor* atomic model.

This memory mapping is performed equivalently for the rest of output ports of atomic models (i.e., T_{out} and P_{out}). Virtual memory mapping is a recursive process that enables any atomic model to read input data from its original source, avoiding time-consuming intermediate read/write operations. The chained simulator is a flat simulator, in which the model hierarchy disappears, and virtual memory mapping becomes a simpler operation, reducing even more simulation overhead.

3.1 Implementing the chained simulator

The implementation of the chained simulator is divided in three parts: (i) the function set of the simulators, (ii) the function set of the coordinators, and (iii) the root coordinator main routine, in charge of executing the simulation loop. Keeping in mind a potential parallelization of the simulation algorithm, we ensure access to the data corresponding the atomic output ports until all the dependent external events have been executed. This is done by returning memory locks in the simulators collection functions *@(t)* indicating their non-empty output ports. Parent coordinators are responsible of managing locks of their child processors. Each coordinator must solve all the depending events due to couplings before unlocking ports in their corresponding simulators. The function set of simulators and coordinators is the following:

- *@(t)*: collection function. It is invoked on imminent simulators before the transition function, and it may return locks pointing to ports with new data.
- **(t, P)*: transition function. It deals with internal and external state transitions. The argument *P* contains tuples (l_{from}, p_{to}). For each tuple, the input port p_{to} reads input data from the port that the lock l_{from} points to. Processors executing this function return their next time advance, t_N.
- *u(t, l)*: unlock function. It is used to remove the lock *l* of a given output port and free its memory.

Functions are triggered hierarchically: only parent coordinators can execute a function of their child processors. Atomic models are the only ones that produce data when the collection function of their corresponding simulator is triggered. Furthermore, data is exclusively read when an atomic model triggers an external transition function. Atomic models read input data directly from the original source, with no intermediate copies required. Simulators check which output ports contain data after executing their model's output function. If there is data, the simulator locks the port and returns a reference of the port to the parent coordinator. A lock provides read-only permits. While a port is locked, writing or deleting data from the port is forbidden. Coordinators receive locks from their child processors. A new lock means that the child processor's port that is locked contains new data. If there is any *IC*, the coordinator sends the lock to the influencees in the next transition function, so they have access to the data of the port. As locks have read-only permissions, we ensure that simulators with access to the port's values do not add nor remove anything. If there is an *EOC*, the output port of the coupled model is blocked and sent to the parent coordinator. Notice that a lock can be recursive: it can either point to data or to a set of locks that point to data.

3.1.1 Simulator Function Set

The simulator function set (represented in Algorithm 1) contains all the functions used to deal with the atomic components and manages its behavior. An independent simulator is instantiated for each atomic component present in the model. Each of them has a locks list, $locked_{ext}$, that contains locks of the non-

empty output ports. This list is shared between functions, being used both in the collection and unlock function.

The collection function, *@(t)*, is called by the parent coordinator when the current simulation time *t* matches with the next time scheduled in the related atomic component, t_N. It first executes output function of its atomic model, λ. Then, each port $p \in OPorts$ with new output events is locked, and its lock l_p is stored in the $locked_{ext}$ list. This prevents the removal of new messages until the parent coordinator triggers the unlock function $u(t, l_p)$. Simulators return their $locked_{ext}$ list, giving up the control of locked ports to their parent coordinator.

The transition function, **(t, P),* receives the current time and a list *P* containing tuples (l_{from}, p_{to}) of port references. p_{to} must belong to the input ports set ($IPorts$) of the related atomic component, and l_{from} is a virtual memory reference to the data generated by a locked source port p_{from} with new data. These source ports correspond directly to the atomic output ports where the values are physically stored. Therefore, the set of inputs that must be considered in this transition function is the one resulting from joining the output sets of all the ports $p_{from} \in P$. Depending on the time and presence of new input data, either an internal, an external, or a confluent transition event of the atomic model is processed. At the end of the transition function, the next internal transition time t_N is returned to the parent coordinator.

The unlock function, *u(t, l)*, unlocks a previously locked port and frees its assigned memory. The locked port p must belong to the output ports set ($OPorts$).

Algorithm 1: Simulator function set.

Function $@(t)$**:**
- **assert** $t = t_N$;
- $y := \lambda(s)$;
- **foreach** $(p_{from}, v) \in y$ **do**
  - $l_{from} := lock(p_{from})$;
  - $locked_{ext} := locked_{ext} \cup l_{from}$;
- **return** $locked_{ext}$

Function $*(t, P)$**:**
- $x := \emptyset$;
- **foreach** $(l_{from}, p_{to}) \in P$ **do**
  - **assert** $p_{to} \in IPorts$;
  - $x := x \cup y(l_{from})$;
- **if** $t_L \leq t < t_N \wedge x \neq \emptyset$ **then**
  - $e := t - t_L$;
  - $s := \delta_{ext}(s, e, x)$;
- **else if** $t = t_N \wedge x = \emptyset$ **then**
  - $s := \delta_{int}(s)$;
- **else if** $t = t_N \wedge x \neq \emptyset$ **then**
  - $s := \delta_{con}(s, x)$;
- **else**
  - raise error
- $t_L := t$;
- $t_N := t_L + ta(s)$;
- **return** t_N;

Function $u(t, l)$**:**
- **assert** $l \in locked_{ext}$;
- $locked_{ext} := locked_{ext} - l$;
- $p := unlock(l)$;
- **assert** $p \in OPorts$;
- $y(p) := \emptyset$;

3.1.2 Coordinator Function Set

The coordinator function set contains all the functions used to deal with the coupled components and synchronize its child processors. It is represented in Algorithm 2. The name of the functions, as well as the expected parameters, coincides with the simulator function set. This proves that the chained algorithm provides closure under coupling (Zeigler 2018). Coordinators have two internal lock lists: $locked_{int}$ and $locked_{ext}$. The first contains all the locks of output ports of child processors that contain new data. The second is composed of locks of output ports of the coupled model itself that point to one or more ports of child processors with new data. Additionally, coordinators keep two lock-to-port reference tables: P_{int} and P_{ext}. P_{int} maps virtual memory between locks with new data and input ports of child processors, mimicking the *EIC* and *IC* sets defined in the coupled model. On the other hand, P_{ext} keeps record of memory references between locks of output ports of child processors and output ports of the coupled model

according to its *EOC*. Finally, imminent child processors that require to be activated at a given simulation time are tracked in the *sync* set.

The collection function, *@(t)*, is triggered by the parent coordinator when the simulation time matches with the minimum next time scheduled by any child processor. The collection function is forwarded to imminent child processors, and if any lock is returned, the coordinator stores it in $locked_{int}$ and resolves the corresponding port memory mapping in P_{int} according to the coupled model's *IC*, adding influencees to the $sync$ list. If any *EOC* is triggered, the corresponding output port of the coupled model is locked and stored in $locked_{ext}$. This last set is returned to the parent coordinator. Additionally, the virtual memory mapping corresponding to the *EOC* is stored in P_{ext}.

The transition function, **(t, P)*, is forwarded to all the child processors in the *sync* set. Memory mapping related to the *IC* set is cleared, and child processors' output ports with no *EOC* dependencies are unlocked.

If an unlock function *u(t, l)* is triggered by the parent coordinator, the output port corresponding to the given lock is unlocked. Memory mapping related to the *EOC* set is removed, and child processors' output ports with no *IC* dependencies are unlocked.

Algorithm 2: Coordinator function set.

Function $@(t)$**:**
- **assert** $t = t_N$;
- $t_L := t_N$;
- **foreach** $i \in ImminentChildren$ **do**
  - $sync := sync \cup i$;
  - **foreach** $l_i \in i :: @(t)$ **do**
    - $locked_{int} := locked_{int} \cup l_i$;
    - **foreach** $(p_i, p_j) \in IC$ **do**
      - $sync := sync \cup j$;
      - $P_{int}(j) := P_{int}(j) \cup (l_i, p_j)$;
    - **foreach** $(p_i, p_{self}) \in EOC$ **do**
      - $P_{ext} := P_{ext} \cup (l_i, p_{self})$;
      - $l_{self} := lock(p_{self})$;
      - $locked_{ext} := locked_{ext} \cup l_{self}$;
    - **if** $l_i \notin P_{int} \wedge l_i \notin P_{ext}$ **then**
      - $locked_{int} := locked_{int} - l_i$;
      - $i :: u(t, l_i)$);
- **return** $locked_{ext}$;

Function $*(t, P)$**:**
- **assert** $t_L \leq t \leq t_N$;
- $locked_{tmp} = \emptyset$;
- **foreach** $(l_{from}, p_{self}) \in P$ **do**
  - **foreach** $(p_{self}, p_j) \in EIC$ **do**
    - $l_{self} := lock(p_{self})$;
    - $locked_{tmp} := locked_{tmp} \cup l_{self}$;
    - $sync := sync \cup j$;
    - $P_{int}(j) := P_{int}(j) \cup (l_{self}, p_j)$;
- **foreach** $i \in sync$ **do**
  - $t_{N,i} := i :: *(t, P_{int}(i))$;
  - **foreach** $(l_j, p_i) \in P_{int}(i)$ **do**
    - **if** $l_j \notin P_{ext}$ **then**
      - $j :: u(t, l_j)$;
  - $P_{int}(i) := \emptyset$;
- $sync := \emptyset$;
- **foreach** $t_{self} \in locked_{tmp}$ **do**
  - $unlock(l_{self})$;
- $t_L := t$;
- $t_N :=$ minimum of child components' t_N;
- **return** t_N;

Function $u(t, l_{from})$**:**
- **assert** $l_{from} \in locked_{ext}$;
- $locked_{ext} := locked_{ext} - l_{from}$;
- $p_{self} := unlock(l_{from})$;
- **foreach** $(l_i, p_{self}) \in P_{ext}$ **do**
  - **if** $l_i \notin P_{int}$ **then**
    - $i :: u(t, l_i)$;
- $P_{ext}(p_{self}) := \emptyset$;

3.1.3 Root Coordinator Routine

The root coordinator contains the same function set as any regular coordinator. However, as root of the entire hierarchy, it has an additional main routine, which corresponds to the simulation workflow. The main routine is described in Algorithm 3. The root coordinator only triggers collection and activation functions. As the root coordinator has no parent, its activation function will never generate any external lock.

4 CASE STUDY: THE XDEVS SIMULATOR

To study the effectiveness of the chained simulation algorithm, we used the Python branch of the xDEVS simulation engine. The original xDEVS version is based on the classic abstract simulator. We modified this implementation following the function sets described in Section 3 to obtain a renewed version of xDEVS.

Algorithm 3: Root coordinator main routine.

Function *Main*:
  $t := t_N$;
  while $t \neq \infty$ **do**
    $locked := @(t)$;
    assert $locked = \emptyset$;
    $*(t, \emptyset)$;
    $t := t_N$;

To evaluate how much the chained algorithm improved the performance compared to the original, we ran a heterogeneous set of synthetic DEVStone benchmarks, comparing the simulation time using different models and shapes. For the LI, HI, and HO structures, we explored 100 different models with widths and depths going from 20 to 200 with increments of 20. On the other hand, as the complexity of HOmod models is significantly higher than the rest of the DEVStone structures, we explored a reduced set of 50 models with widths and depths going from 10 to 50 with increments of 10.

As we were only interested in analyzing the simulation engine execution overhead, we set both internal and external execution time to 0 (these mimic the processing required for computing a given model's next state – i.e., it depends on the model under study). By setting these parameters to 0, we ensure that simulation time only depends on the model's structure and the simulation algorithm of the engine.

All these combinations of models were simulated 5 times both in the original and renewed implementations of xDEVS to work with a confidence interval of 95% in the results. We used a workstation with Ubuntu 18.04, Intel Core i7-9700 and 64GB RAM. All the experiments were run sequentially.

Table 1 shows the simulation results of a subset of corner cases of the experiments. For each structure, we show the deepest, the widest, and the most complex models.

Table 1: Chained/original xDEVS simulation time comparison for several DEVStone models

Structure	Width	Depth	Original (s)	Chained (s)	Speedup
LI	20	200	0.0191 ± 0.0003	0.0140 ± 0.0005	1.3578
	200	20	0.0173 ± 0.0002	0.0139 ± 0.0003	1.2444
	200	200	0.1985 ± 0.0035	0.1582 ± 0.0013	1.2552
HI	20	200	0.2306 ± 0.0016	0.1546 ± 0.0033	1.4913
	200	20	2.1440 ± 0.0859	1.5375 ± 0.0374	1.3945
	200	200	25.5225 ± 0.1836	17.6190 ± 0.3642	1.4486
HO	20	200	0.2730 ± 0.0038	0.1558 ± 0.0124	1.7517
	200	20	2.5063 ± 0.0476	1.5435 ± 0.0442	1.6238
	200	200	29.6602 ± 0.6885	17.6750 ± 0.4832	1.6781
HOmod	10	50	4.3406 ± 0.0429	2.6092 ± 0.1474	1.6636
	50	10	18.8785 ± 0.2663	10.9209 ± 0.1459	1.7287
	50	50	615.0647 ± 11.3729	367.3651 ± 9.9269	1.6743

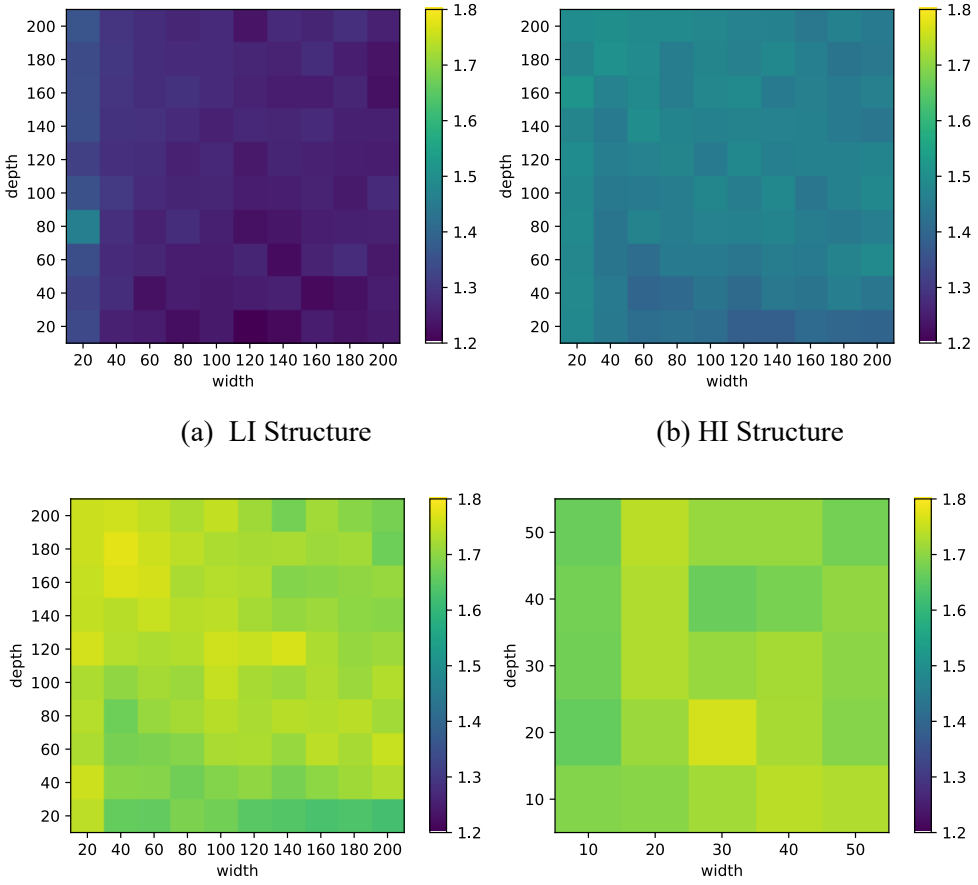
Times shown in the table represent exclusively the simulation time, and therefore they do not include the model instantiation and engine setup times. Even though model instantiation times are the same for both simulation algorithms, the chained simulation engine has then to adapt the models before starting to simulate. This adaptation is required for keeping backwards compatibility with the original simulation algorithm. In future versions of xDEVS, the chained algorithm will be the standard, and this additional pre-processing will not be necessary.

The *speedup* column represents the improvement of the simulation time of the chained implementation over the original one. From the obtained results, we can infer that the chained algorithm shows better

performance when simulating more complex models. This trend is not accomplished by models of the HOmod structure, which presented less mean speedup than HO models. Note that, for a given width and depth, all the DEVStone structures except HOmod contain the exact number of atomic models and coupled models. In contrast, the number of components within HOmod models grows exponentially faster with changes in width or depth, increasing the amount of time required for processing external/internal transition functions. As the chained simulation algorithm focuses on reducing message propagation times, the speedup obtained in HOmod models is slightly smaller than in HO models.

Looking at the results of a specific structure, models with greater depth and smaller width tend to present higher speedups, regardless of the structure. However, the difference is not significant enough, and a more in-depth research should be performed before inferring any further conclusion.

Figure 3 shows the speedup obtained using the chained simulator over the original implementation of xDEVS, using the depth of the structures as the ordinates and the width as the abscissae.



(a) LI Structure (b) HI Structure

(c) HO Structure (d) HOmod Structure

Figure 3: Speedup of the chained simulator over the original implementation of xDEVS

Experiments reported that, even though cases with greater depth and smaller width had slightly greater speedups, the obtained speedup remained uniform for a given structure, regardless of its shape. However, if we compare the obtained results depending on the structure, the simulation time improvement of the chained algorithm over the original implementation is greater in more complex structures: using LI structures, the mean speedup was 1.2668, whereas HI structures reported a mean speedup of 1.4557, and of 1.7149 and 1.7026 in HO and HOmod structures, respectively. This is because the main benefit of using

the chained simulation algorithm resides in how events are propagated through components in the hierarchy of the model. As the number of couplings increments, so do the number of events triggered during the simulation. The original algorithm spends a greater portion of the simulation time forwarding events, while the chained algorithm does not need to forward them, as any coupled atomic model can access to these new events directly to their original source.

5 CONCLUSIONS

M&S tools require higher computing capabilities to explore increasingly complex scenarios. New technologies and algorithms that enhance the performance of simulation tools are needed. PDEVS is an M&S formalism that allows the simulation of a variety of complex systems. Its modular and hierarchical structure comes to several benefits, but it raises the need for communicating messages between the modules of the system. This operation represents a significant percentage of the simulation time spent just in synchronization issues. Our chained simulation algorithm is a PDEVS-compliant simulation routine that makes extensive use of shared memory patterns to reduce the computation footprint of simulation engines.

To provide experimental results that support this approach, we developed a new implementation of the Python branch of xDEVS. Regarding the comparison, we used all the structures defined in the DEVStone benchmarks to evaluate the different algorithms. This method is a convenient method for analyzing DEVS environments, allowing them to generate synthetic test models with a variety of structures and behaviors. We showed that the chained simulator allows reducing the simulation engine overhead up to 40%.

This memory-shared approach can improve performance of PDEVS simulation engines. Besides, any PDEVS framework can integrate it with no backwards compatibility issues. With it, we contribute to continue towards faster simulators that introduce less simulation time overheads to the model computation time while reducing the energy consumption.

As future work, we will perform an in-depth study of several simulation engines using the DEVStone benchmark. Using profiling techniques, we will identify the potential benefits of implementing the chained simulation algorithm for different PDEVS-compliant frameworks. We will also define implementation practices for adding native support to distributed simulation.

ACKNOWLEDGMENTS

This work has been partially supported by the Education and Research Council of the Community of Madrid (Spain), under research grant S2018/TCS-4423.

REFERENCES

Belloli, Laouen, Vicino Damián, and Cristina Ruiz-Martín. 2019. "Building DEVS Models with the Cadmium Tool." *2019 Winter Simulation Conference (WSC).* IEEE. 45-59.

Chow, Alex C. H. 1996. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator." *Transactions of the Society for Computer Simulation* 13: 55-68.

Chow, Alex C., Bernard P. Zeigler, and Doo H. Kim. 1994. "Abstract simulator for the parallel DEVS formalism." *Fifth Annual Conference on AI, and Planning in High Autonomy Systems.* IEEE. 157-163.

Guérout, Tom, Thierry Monteil, Georges Da Costa, Rodrigo Neves Calheiros, Rajkumar Buyya, and Mihai Alexandru. 2013. "Energy-Aware Simulation with DVFS." *Simulation Modelling Practice and Theory* (Elsevier) 39: 76-91.

Kim, Kihyung, Wonseok Kang, Bong Sagong, and Hyungon Seo. 2000. "Efficient distributed simulation of hierarchical DEVS models: transforming model structure into a non-hierarchical one." *Proceedings 33rd Annual Simulation Symposium (SS 2000).* 227-233.

Mittal, Saurabh, and José L. Risco-Martín. 2017. "DEVSML 3.0 stack: rapid deployment of DEVS farm in distributed cloud environment using microservices and containers." *Proceedings of the Symposium on Theory of Modeling & Simulation.*

Mittal, Saurabh, and Andreas Tolk. 2020. *Complexity Challenges in Cyber Physical Systems: using Modeling and Simulation (M&S) to Support Intelligence, Adaptation, and Autonomy.* John Wiley & Sons.

Risco-Martín, José L. 2014. *xDEVS: M&S Framework.* Accessed April 20, 2020. https://github.com/iscar-ucm/xdevs.

Risco-Martín, José L., and Saurabh Mittal. 2019. "Chapter 14 - Model Management and Execution in DEVS Unified Process." In *Model Engineering for Simulation*, by Lin Zhang, Bernard P. Zeigler and Yuanjun Iaili, 291-313. Academic Press.

Uhrmacher, Adelinde M., Jan Himmelspach, and Roland Ewald. 2018. "Chapter 6: Effective and Efficient Modeling and Simulation with DEVS." In *Discrete-Event Modeling and Simulation*, by Gabriel A. Wainer and Pieter J. Mosterman, 38. CRC Press.

Ullah, AMM Sharif. 2019. "Modeling and simulation of complex manufacturing phenomena using sensor signals from the perspective of Industry 4.0." *Advanced Engineering Informatics* (Elsevier) 39: 1-13.

Van Tendeloo, Yentl, and Hans Vangheluwe. 2014. "The modular architecture of the Python (P) DEVS simulation kernel." *Proceedings of the 2014 Symposium of Theorry og Modeling and Simulation-DEVS.* 387-392.

Wainer, Gabriel A., Khaldoon Al-Zoubi, Olivier Dalle, Saurabh Mittal, José L. Risco-Martín, Hessam Sarjoughian, and Bernard P. Zeigler. 2016. "Chapter 18: Standardizing DEVS Simulation Middleware." In *Discrete-Event Modeling and Simulation: Theory and Applications*, by Gabriel A. Wainer and Pieter J. Mosterman, 459. CRC Press

Wainer, Gabriel A., E. Glinsky, M. Gutiérrez-Alcaraz. "Studying Performance of DEVS Modeling and Simulation Environments using the DEVStone Benchmark". SIMULATION: Transactions of the Society for Modeling and Simulation International. Vol. 87, No. 7, pp. 555–580. July 2011.

Zeigler, Bernard P. 2018. "Closure under coupling: concept, proofs, DEVS recent examples (wip)." *Proceedings of the 4th ACM International Conference of Computing for Engineering and Sciences* 1-6.

Zeigler, Bernard P., Herbert Praehofer, and Tag G. Kim. 2000. *Theory of Modeling and Simulation. Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press.

AUTHOR BIOGRAPHIES

ROMÁN CÁRDENAS received the M.Sc. degree in Telecommunication Engineering in 2019 from Technical University of Madrid (UPM), where he is currently pursuing the Ph.D. in Electronic Systems Engineering in Cotutelle with Carleton University (CU). His research interests include modeling and simulation with applications in the IoT domain. He can be reached at r.cardenas@upm.es.

KEVIN HENARES is a Ph.D. candidate at the Complutense University of Madrid (UCM). His work focuses on the development of robust modeling and simulation methodologies to study the behavior of complex systems. His email address is khenares@ucm.es.

PATRICIA ARROBA is an Assistant Professor at the Technical University of Madrid (UPM). She received her Ph.D. degree in Telecommunication Engineering from UPM in 2017. Her research interests include energy and thermal-aware modeling and optimization of data centers. Her email address is p.arroba@upm.es.

GABRIEL WAINER received the Ph.D. degree from Université d'Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada), where he is now a Full Professor. His current research interests are related with modelling methodologies and tools, parallel/distributed simulation and real-time systems. His e-mail is gwainer@sce.carleton.ca. His website is www.sce.carleton.ca/faculty/wainer.

JOSÉ L. RISCO-MARTÍN received his Ph.D. from Complutense University of Madrid, and currently is Associate Professor in the Department of Computer Architecture and Automation at Complutense University of Madrid (UCM). His research interests include computer-aided design and modeling, simulation and optimization of complex systems. He can be reached at jlrisco@ucm.es.