A DEVS Simulation Algorithm Based on Shared Memory for Enhancing Performance

Román Cárdenas*
r.cardenas@upm.es
Laboratorio de Sistemas Integrados
Universidad Politécnica de Madrid
Madrid, Spain

Kevin Henares*
khenares@ucm.es
Dpt. of Computer Architecture and Automation
Universidad Complutense de Madrid
Madrid, Spain

Patricia Arroba
p.arroba@upm.es
Laboratorio de Sistemas Integrados
Universidad Politécnica de Madrid
Madrid, Spain

Gabriel Wainer
gwainer@sce.carleton.ca
Dpt. of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada

José L. Risco-Martín
jlrisco@ucm.es
Dpt. of Computer Architecture and Automation
Universidad Complutense de Madrid
Madrid, Spain

ABSTRACT

The Parallel Discrete EVent System Specification (PDEVS) formalism provides a standard basis to define accurately any discrete-event system. As the complexity of the system under study increases, the necessity of simulation engines with higher performance rises. In this research, we present Chained DEVS, a PDEVS 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 with a Chained DEVS version of a state-of-the-art simulation engine. Results show that Chained DEVS introduces up to 50% less synchronization overhead than traditional simulation engines.

CCS CONCEPTS

• **Computing methodologies → Massively parallel and high-performance simulations**; *Discrete-event simulation*; Simulation environments.

KEYWORDS

DEVS, Simulation Engine, Performance, Shared Memory

ACM Reference Format:
Román Cárdenas, Kevin Henares, Patricia Arroba, Gabriel Wainer, and José L. Risco-Martín. 2020. A DEVS Simulation Algorithm Based on Shared Memory for Enhancing Performance. In *Proceedings of (SIGSIM PADS'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Both authors contributed equally to this research

SIGSIM PADS'20, June 15–17 2020, Miami, Florida USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Multiple Modeling and Simulation (M&S) methodologies have emerged as a way to conduct preliminary studies of complex systems [1]. Nowadays, M&S is a common practice in science, technology, industry, and governance [2–5], reducing the capital expenses and potential hazards that testing with real systems may imply.

The number of fields that incorporate M&S is constantly growing. The increasing complexity of the models under study places the definition of a formal method for model description and the optimization of simulation engines as an important research field. This optimization also introduces improvements in terms of energy consumption. This is specially relevant in embedded systems, where the energy is one of the main constraints [6]. Although there are different modeling formalisms (e.g., Petri nets [7] or cellular automata [8]), the Discrete EVent System Specification (DEVS) [9] and its Parallel DEVS (PDEVS) [10] variant showed success in expressing any discrete-event formalism [11].

PDEVS enables the description of a model as a hierarchical description of submodels and their relationship [12], 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 simulation engines use message-passing: 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 processing overheads in both sequential and parallel execution. This is specially relevant when messages are sent through different levels of the hierarchy, producing a redundant propagation in the intermediate ports of the coupled components.

In this research, we introduce a novel simulation algorithm for PDEVS that exploits the benefits of shared memory systems. The algorihtm introduces a function-oriented design focused on reducing message-passing overhead. We discuss the potential benefits

of chained DEVS by benchmarking a simulation engine that implements the algorithm. The simulation overhead can be reduced from 10 to 50%, depending on 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 the traditional implementation patterns shared by these tools. Section 3 presents Chained DEVS, 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 an already implemented simulation engine with its equivalent Chained DEVS version. Finally, we present conclusions in Section 5.

2 STATE OF THE ART

In this section, we first present the PDEVS formalism and introduce different PDEVS simulation engines. We identify their common implementation patterns and propose a new implementation approach that aims to improve the overall simulation performance. We also discuss about the main proposals for analyzing and comparing the performance of DEVS environments.

2.1 DEVS and Parallel DEVS

The Discrete EVent System Specification (DEVS) formalism, which was introduced by Zeigler in 1976 [9], provides a rigorous foundation for discrete Modeling and Simulation (M&S). DEVS allows the user to define a mathematical object (i.e., system) that represents an abstraction of real objects [13]. Parallel DEVS (PDEVS) [10] 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 [14]. Given the recursive definition of coupled models, they can itself be a part of a component in a larger coupled model system. This results in a hierarchical DEVS model construction.

Based on DEVS formal specifications, a number of PDEVS tools has been defined. These engines are written in multiple programming languages, like C++ (e.g., CD++ [15], Cadmium [16], DEVSim++ [17]), Java (e.g., DEVSJAVA [18], xDEVS [19]), or Python (e.g., PyPDEVS [20], xDEVS [19]). Regardless of this heterogeneity of implementations, engine implementations for simulating the PDEVS formalism usually share the same design pattern. They are based on the abstract simulator concept presented by Chow and Zeigler [21]. This approach proposes the usage of 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.

Five types of synchronization messages are interchanged between abstract simulators:

- $(@, t)$: 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 their output function λ.
- (q, t): external messages. They contain bags of messages to be forwarded. Parent coordinators send these messages to receiver child processors.
- $(*, t)$: transition messages. Imminent child processors receive this message to execute a transition function δ_x.
- (y, t): output messages. They contain bags of messages to be forwarded from child processors to parent coordinators.
- $(done, t)$: done messages. Child processors send these messages to acknowledge parent coordinators that they have finished processing a given pending task.

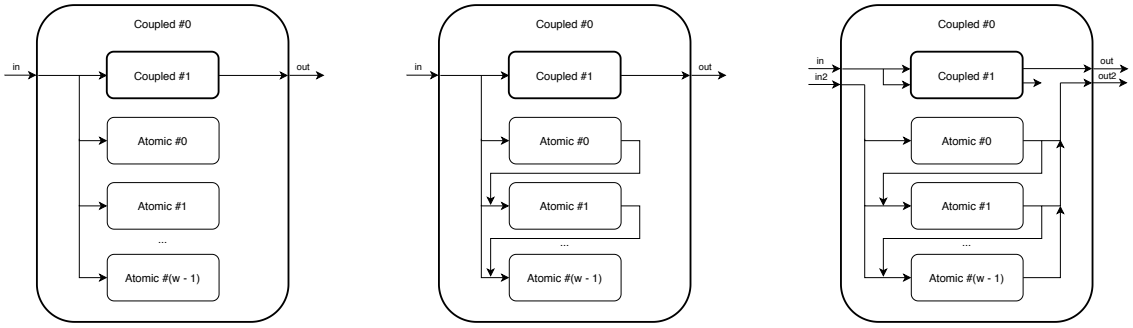
Note that this communication is always hierarchical, and follows a request-response fashion: parent coordinators send requests to their child processors. Child processors end their execution by responding with a $(done, t)$ message.

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 data through the model ((q, t) and (y, t) messages for downwards and upwards communication, respectively). 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., DCD++ [22] or RISE [23]). This message passing process, though necessary for distributed simulation, impose a heavy simulation overhead in both sequential and parallel simulation. 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.

With the aim of reducing this, the flat simulator was defined [24]. This approach creates an equivalent DEVS model that removes all the 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. Researchers has shown that flat simulators have greater performance [25]. However, messages are still copied depending on the number of port couplings.

2.2 Discrete-Event Simulation Performance

With the aim of measuring and comparing the performance of different PDEVS simulation engines, several comparison methods have been presented [26], commonly applied to specific applications. One of the most popular discrete-event simulation benchmarks is DEVStone [27–29]. DEVStone describes several synthetic models with different sizes and complexities. They are defined as coupled models, containing a fixed structure which is recursively replicated in other child coupled components in their inside. This recursion ends with a more simple 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



(a) LI model **(b) HI model** **(c) HO model**

Figure 1: DEVStone models

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. For consuming the necessary CPU clocks, the Dhrystone code is used [30]. Dhrystone code consists of a mix of instructions using integer arithmetic.

DEVStone defines 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). This structures contain a higher level of internal couplings.
- *HO models* (Figure 1c). The number of ports, input and output couplings is increased.
- *HOmod models*. The topology of these models is similar to HO models. However, the number of couplings and outputs grows exponentially.

The structure of the first three ones, that are the ones used in our comparisons, is summarized in Figure 1.

Making profiling of the execution of an arbitrary DEVStone auto-generated model, 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 an HO model of depth 300, width 10, and no internal and external delays. In this research, we introduce the chained simulation algorithm. Chained simulator is compliant with the PDEVS formalism. However, in contrast with the classic message interchange-based abstract simulator [21], we propose a new approach with 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 alternative to the PDEVS simulator that avoids the use of propagation functions. Another meaningful change in Chained DEVS relates to the concept of the abstract simulator. With the removal of couplings and the propagation needs, we update the abstract simulator concept presented by Chow and Zeigler [21]. As a result, we obtain a different function set and replace message transmission by exploiting shared memory mechanisms.

The main structure of the abstract 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 similar to the one reflected by the atomic and coupled components present in the model.

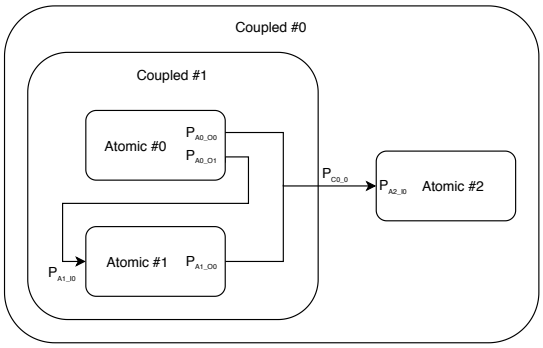
Although all the components have ports, they now are defined accordingly to the aforementioned idea. Only the atomic output ports store the values to be transmitted, while the rest of the ports only reference (directly or indirectly) them. In this way, when an external transition is activated, the input data is read from their original output ports. Figure 2 illustrates how memory management works in the chained simulation algorithm.

Figure 2a shows a PDEVS model composed of three atomic models. Atomic 0 and Atomic 1 models are encapsulated in Coupled 1. Finally, Coupled 0, which is the top model, describes the couplings between Coupled 1 and Atomic 2.

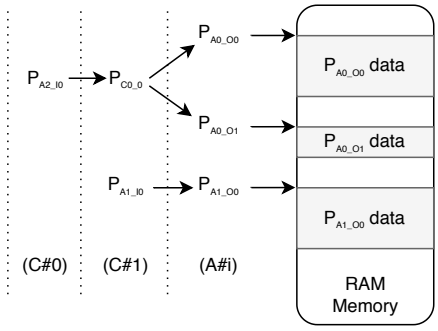
Ports P_{A0_O0} and P_{A0_O1} are output ports of Atomic 0. On the other hand, Atomic 1 has an input port P_{A1_I0} and an output port P_{A1_O1}. As mentioned before, the only ports in which new data is written are output ports. Figure 2b shows how the output ports of the atomic models are pointing directly to memory where output data is written. However, the input port P_{A1_I0} is not pointing directly to memory, but to a port that is pointing to memory. In other words, input ports point to virtual memory zones. This virtual memory mapping is provided by the coordinator of Coupled 1, according to the internal couplings of the model.

If we check Atomic 2, this model only has one input port, P_{A2_I0}. The root coordinator, based on the internal couplings between Coupled 1 and Atomic 2, generates a virtual memory mapping between P_{A2_I0} and P_{C1_O0}. However, P_{C1_O0} is not the output port of an atomic model, and therefore it does not point to memory directly. Instead, the coordinator of Coupled 1 generates a nested virtual memory mapping between P_{A0_O0}, P_{A1_O1}, and P_{C1_O0} based on its external output couplings.

This abstract simulator virtual memory mapping is a recursive process that enables any atomic model to read input data from the original source, avoiding time consuming read/write operations. It



(a) Reference model



(b) Memory management

Figure 2: Example for illustrating memory management in Chained DEVS

is worth mentioning that this change only affects the simulation, and is transparent to the users implementing the models.

Note that the chained simulator is compatible with the flat simulator. Combining both approaches, the model hierarchy disappears, and virtual memory mapping becomes a simpler operation, reducing simulation overheads.

3.1 Implementation

The implementation of the chained simulator can be divided into three parts: (i) the function set of the simulators for managing the behavior of the atomic components, (ii) the function set of the coordinators for the control of the simulation entities, and (iii) the main coordinator, in charge of the main simulation loop, in which the root coordinator is called cyclically.

Keeping in mind a potential parallelization of the simulation algorithm, we ensure access to the atomic output ports until all the dependent external events have been executed. This is done by returning *lock* messages in the simulators output functions, describing their non-empty output ports. Parent coordinators are responsible of managing locks of their child processors. Each coordinator must solve all the depending events before unlocking specific ports in their corresponding simulators.

The functions of both the simulator and coordinator sets are the following:

- $@(t)$: output function. It is invoked before any transition function, and it may return locks pointing to ports with new data.
- $*(t, P)$: transition function. It deals with internal and external events. 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.
- $u(t, l)$: unlock function. It is used to remove the lock l of a given output port and free its memory – i.e., clear data.

Note that the functions are triggered hierarchically: Only parent coordinators can execute a function of a child processor.

Atomic models are the only ones that produce data when an output function 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 the model's output function. If there is data, the simulator locks the port and returns a reference to 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 contains all the functions used to deal with the atomic components and manage its behavior. It is represented in Algorithm 1. An independent simulator is instantiated for each atomic component present in the model. Each of them has a locks list ($locked_{int}$), that contains locks for the non-empty output ports. This list is shared between functions, being used both in the output and unlock function.

The output function, $@(t)$, is called by the parent coordinator when the current simulation time match with the next time scheduled in the related atomic component. It first executes the atomic output function λ. Then, each port $p \in OPorts$ producing output is locked, and its lock l_p is stored in the $locked_{ext}$ list. This prevents the removal of the produced values until the parent coordinator calls to the unlock function $u(t, l_p)$. This call applies to specific ports. Hence, the parent coordinator will generate as many unlock calls to a specific simulator as the ports locked in this function. The $locked_{ext}$ list is returned to the parent coordinator, so that it can manage the pending locked ports.

The transition function, $*(t, P)$, receives the current time and a list of (l_{from}, p_{to}) port references. p_{to} must belong to the input ports set ($IPorts$) of the related atomic component, and the l_{from} represents a reference to the locked source port p_{from}. These source ports corresponds directly to the atomic output ports where the values are physically stored. Therefore, the set of inputs that has to

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** $p \in OPorts$;
- $y(p) := \emptyset$;

be taken into account in this transition function is the one resulting from joining the output sets of all the $p_{from} \in P$. Depending on the time and presence of new input, it is processed either an internal, an external, or a confluent transition event. The internal event is processed when the current simulation time matches the next event time in the simulator and may produce a new state based on the previous one. The external event is processed when there is at least one input and may potentially resulting in a state change. Finally, the confluent transition is processed when the conditions for the these two events met at the same time. At the end of the transition function, next time is calculated based on the current state.

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 ($OPorts$).

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 Chained DEVS provides closure under coupling [12].

Coordinators have two lock lists ($locked_{int}$ and $locked_{ext}$). The first contains all the locks of output ports of child processors that are currently blocked. The second is composed of locks of output ports of the coupled model itself. Additionally, coordinators keep two lock-to-port reference tables: P_{int} and P_{ext}. P_{int} maps virtual memory between locks of output ports and input ports of child processors. 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. Finally, imminent child processors that require to be activated at a given simulation time are tracked in the *sync* set.

The output function, $@(t)$, is called by the parent coordinator when the simulation time matches with the minimum next time scheduled by any child processor. The output 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 coupled model output port is locked and stored in $locked_{ext}$. The lock is sent 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 IC is removed, 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 is removed, and child processors' output ports with no IC dependencies are unlocked.

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 output 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 use the Python branch of the xDEVS engine [19]. The original xDEVS simulation engine is based on the abstract simulator proposed by Chow and Zeigler [10]. We modify this implementation as explained in Section 3 to obtain a renewed version of xDEVS.

To evaluate the original and chained engines, we use two different approaches. First, in Section 4.1 we compare the implementations using a simple queue model representing how the employees deal with the customers in a supermarket. After that, we compare its simulation times using synthetic DEVStone benchmarks [27].

4.1 Customers and Employees Model

In this section, we compare both versions using a parameterizable model that represents how the supermarket employees deal with a queue of customers. In it, we have three types of atomic models: (i) the `CustomerGenerator`, (ii) a `Queue`, and (iii) the `Employees`. The

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**
    - $i :: u(t, l_i)$;
- $P_{ext}(p_{self}) := \emptyset$;

Algorithm 3: Root Coordinator Main Routine

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

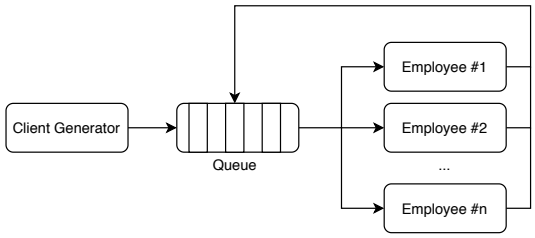


Figure 3: Customers and employees model

`CustomerGenerator` adds customers to the `Queue`, representing the completion of their purchases. The `Queue` notifies to all the `Employees` the existence of new customers. Then, the `Employees` notify the queue if they are available. If some of them are available, the `Queue` assigns the customer to an `Employee`. This scenario is depicted in Figure 3.

To modify the behavior of this scenario, the model allows modifying four variables: (i) the number of employees in the store, (ii) the processing time per customer, (iii) the customer generation speed, and (iv) the simulation time.

In regards to simulations, we set fixed values of 30 minutes of simulation time and 10 seconds as customer processing time. The number of employees varies from 1 to 8500. We set times of 1, 5 and 10 seconds as customer generation delays. Then we combine these parameters and simulate the different generated models.

Figure 4 shows the resulting simulation times. In all the cases, we can see how the chained version of xDEVS performs better than the original, reducing considerably the simulation overhead.

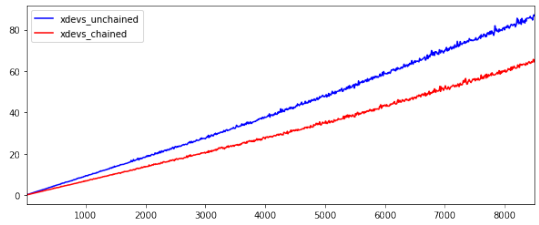
In all the scenarios, the speedup achieved by the chained simulator is 1.36510 ± 0.00075 with a confidence interval of 99%, regardless of the scenario configuration. This result may indicate that the improvement is dependent on the model under study. In the next subsection, we explored the performance of the chained simulator using synthetic DEVStone benchmarks.

4.2 DEVStone Benchmark

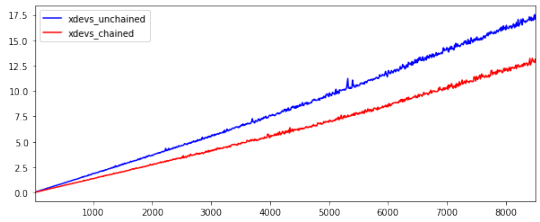
The study presented in this section uses three types of DEVStone models:

- Shape 1: models with a depth of 300 and a width of 10.
- Shape 2: models with a depth of 10 and a width of 300.
- Shape 3: models with a depth of 300 and a width of 300.

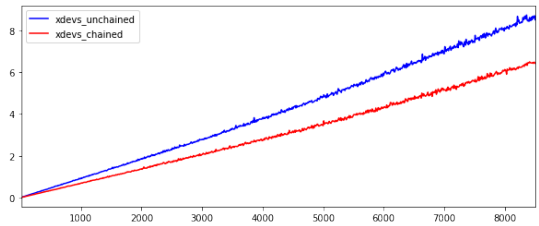
As we are only interested in analyzing the simulation engine execution overhead, we set both internal and external execution time to 0 (these mimic the processing required for computing a



(a) Simulation time with 1s as customer generation time



(b) Simulation time with 5s as customer generation time



(c) Simulation time with 10s as customer generation time

Figure 4: Simulation time comparison using the customers and employees model

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.

For each DEVStone shape, we used three types of models (LI, HI, and HO). HOmod does not scale smoothly, as the width and depth of the model grows exponentially, and the improvement of our proposal is better analyzed using the other three benchmarks. All these combinations of models have been simulated both in the original and renewed implementation of xDEVS, using a workstation with Ubuntu 18.04, Intel Core i7-9700 and 64GB RAM. Table 1 shows the simulation results.

The times shown represent exclusively the simulation time, and therefore it does not include the model instantiation and engine setup times. It also does not include the time spent in injecting the value that triggers all interactions into the input port of the root coupled component. Moreover, we have to take into account that the time shown there for each model corresponds to an average time of ten repetitions of the simulation of the same model. The *Speedup* column represents improvement of the simulation time of the chained implementation over the original one. As it can be seen in Table 1, HO model simulations present a better speed up in each of the shape blocks. This is due to the higher number of couplings

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

		Original (s)	Chained (s)	Speedup
Shape 1	LI	0.007	0.005	1.318
	HI	0.078	0.050	1.549
	HO	0.095	0.049	1.918
Shape 2	LI	0.006	0.005	1.160
	HI	2.162	1.564	1.382
	HO	2.483	1.490	1.667
Shape 3	LI	0.203	0.181	1.120
	HI	85.611	61.599	1.390
	HO	100.189	62.083	1.614

Table 2: Simulation profiling of the xDEVS interpreter

	Original (ms)	Chained (ms)	Speedup
lambdaf	68.027	14.516	4.686
deltfcn	83.683	56.069	1.492
clear	26.653	19.415	1.373
propagate_out	49.203	–	–
propagate_in	21.434	–	–

respect to LI and HI models. Also, Shape 1 tends to produce better results due to a higher ratio of couplings per transition.

To have a better idea of how this simulation time improvement is distributed in the main functions of the xDEVS interpreter, we have analyzed their execution times. For that purpose, we have used the model with the better speed up value (Shape 1, HO model) and the cProfiler of the Python 3.6.9 interpreter. Results are shown in Table 2.

Output functions $@(t)$ in Chained DEVS require less execution time. In case of the original implementation of xDEVS, during the processing of output messages `lambdaf`, data produced by any atomic model is propagated to the corresponding parent coordinator using the `propagate_out` function (i.e., (y, t) messages).

Transition functions $*(t, P)$ (`deltfn` in xDEVS) of Chained DEVS introduce less overhead than the classic approach. The reason is that, in the classic implementation of xDEVS, when a $(*, t)$ message is sent, all the input data is forwarded too using the `propagate_in` function (i.e., (q, t) messages).

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 engine that makes use of shared memory to reduce the computation footprint of simulation engines.

To provide experimental results that support this approach, we developed a rew implementation of the Python branch of xDEVS. In regards to the comparison, we uses 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 a profiling of xDEVS simulation times, confirming the impact that the propagation has on these simulation engines. We demonstrate that the chained simulator allows reducing the overhead introduced by the simulation engine up to 50%.

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 overhead 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 a chained simulation engine for different PDEVS-compliant frameworks. We will also define implementation practices for adding native support to distributed simulation.

REFERENCES

[1] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation. Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press, 2 ed., 2000.

[2] A. S. Ullah, "Modeling and simulation of complex manufacturing phenomena using sensor signals from the perspective of industry 4.0," *Advanced Engineering Informatics*, vol. 39, pp. 1–13, 2019.

[3] R. Cárdenas, P. Arroba, R. Blanco, P. Malagón, J. L. Risco-Martín, and J. M. Moya, "Mercury: A modeling, simulation, and optimization framework for data stream-oriented IoT applications," *Simulation Modelling Practice and Theory*, p. 102037, 2019.

[4] K. Henares, J. Pagán, J. L. Ayala, M. Zapater, and J. L. Risco-Martín, "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*, pp. 223–253, 2019.

[5] A. Georgescu, A. V. Gheorghe, M.-I. Piso, and P. F. Katina, "Governance by emerging technologies—the case for sand and blockchain technology," in *Critical Space Infrastructures*, pp. 237–247, Springer, 2019.

[6] S. Schulz, T. Ewing, and J. W. Rozenblit, "Discrete event system specification (devs) and statemate statecharts equivalence for embedded systems modeling," in *Proceedings Seventh IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2000)*, pp. 308–316, IEEE, 2000.

[7] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, vol. 9, no. 3, pp. 223–252, 1977.

[8] S. Wolfram, "Statistical mechanics of cellular automata," *Reviews of modern physics*, vol. 55, no. 3, p. 601, 1983.

[9] B. P. Zeigler, "Theory of modeling and simulation. jhon wiley & sons," *Inc., New York, NY*, 1976.

[10] A.-H. Chow, "Parallel devs: A parallel, hierarchical, modular modeling formalism and its distributed simulator," *Transactions of the Society for Computer Simulation International*, vol. 13, 01 1996.

[11] H. L. Vangheluwe, "DEVS as a common denominator for multi-formalism hybrid systems modelling," in *Cacsd. conference proceedings. IEEE international symposium on computer-aided control system design (cat. no. 00th8537)*, pp. 129–134, IEEE, 2000.

[12] B. P. Zeigler, "Closure under coupling: Concept, proofs, DEVS recent examples (wip)," in *Proceedings of the 4th ACM International Conference of Computing for Engineering and Sciences*, ICCES'18, (New York, NY, USA), Association for Computing Machinery, 2018.

[13] B. P. Zeigler and T. I. Ören, "Multifaceted, multiparadigm modeling perspectives: tools for the 90's," in *Proceedings of the 18th conference on Winter simulation*, pp. 708–712, 1986.

[14] Y. V. Tendeloo and H. Vangheluwe, "An introduction to classic DEVS," *CoRR*, vol. abs/1701.07697, 2017.

[15] G. Wainer, "CD++: a toolkit to develop DEVS models," *Software: Practice and Experience*, vol. 32, no. 13, pp. 1261–1306, 2002.

[16] "Cadmium: a strong typed PDEVS simulator."

[17] T. G. Kim and C. Choi, "DEVSim++ME: HLA-compliant DEVS modeling/simulation environment with DEVSim++," in *Model Engineering for Simulation* (Lin Zhang and Bernard P. Zeigler and Yuanjun laili, ed.), pp. 355–392, Academic Press, 2019.

[18] H. S. Sarjoughian and B. Zeigler, "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment," *Simulation Series*, vol. 30, pp. 29–36, 1998.

[19] J. L. Risco-Martín, "xDEVS: DEVS M&S Framework." https://github.com/iscar-ucm/xdevs. [Online; accessed 27-Feb-2020].

[20] Y. Van Tendeloo and H. Vangheluwe, "The modular architecture of the Python(P)DEVS simulation kernel: Work in progress paper," in *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*, DEVS '14, (San Diego, CA, USA), Society for Computer Simulation International, 2014.

[21] A. C. Chow, B. P. Zeigler, and D. H. Kim, "Abstract simulator for the parallel DEVS formalism," in *Fifth Annual Conference on AI, and Planning in High Autonomy Systems*, pp. 157–163, IEEE, 1994.

[22] R. Madhoun and G. Wainer, "Studying the impact of web-services implementation of distributed simulation of DEVS and cell-DEVS models.," in *Simulation, Practice and Experience, 2008*, pp. 267–278, 01 2008.

[23] K. Al-Zoubi and G. Wainer, "Distributed simulation of DEVS and Cell-DEVS models using the RISE middleware," *Simulation Modelling Practice and Theory*, vol. 55, 06 2015.

[24] S. Jafer and G. Wainer, "Flattened conservative parallel simulator for DEVS and Cell-DEVS," in *2009 International Conference on Computational Science and Engineering*, vol. 1, pp. 443–448, IEEE, 2009.

[25] K. Kim, W.-S. Kang, B. Sagong, and H. Seo, "Efficient distributed simulation of hierarchical DEVS models: Transforming model structure into a non-hierarchical one," *Simulation Symposium, Annual*, vol. 0, p. 227, 01 2000.

[26] J. L. Risco-Martín, S. Mittal, J. Fabero, M. Zapater, and R. Hermida, "Reconsidering the performance of DEVS modeling and simulation environments using the DEVStone benchmark," *SIMULATION*, vol. 93, p. 003754971769044, 02 2017.

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

[28] M. Gutierrez-Alcaraz and G. Wainer, "Experiences with the DEVStone benchmark," in *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, (San Diego, CA, USA), p. 447–455, Society for Computer Simulation International, 2008.

[29] G. Wainer, E. Glinsky, and M. Gutierrez-Alcaraz, "Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark," *Simulation*, vol. 87, pp. 555–580, 05 2011.

[30] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.