A Theoretical approach to the computational complexity measure of abstract DEVS simulators

Yoro Diouf[*,§], Oumar Y. Maïga[†] and Mamadou K. Traore[‡]

[*]*African University of Science and Technology (AUST)*
P.M.B 681, Garki, Abuja F.C.T, Nigeria

[†]*Université des Sciences*
Techniques et Technologies de Bamako (USTTB)
Hamdalaye ACI 2000 Rue: 405, Porte: 359
BP: E 423, Bamako, Mali

[‡]*Université de Bordeaux*
351 cours de la Libération CS 10004
33405 Talence Cedex, France
[§]*yorodiouf@gmail.com*

DEVS is a sound Modeling and Simulation (M&S) framework that describes a model in a modular and hierarchical way. It comes along with an abstract simulation algorithm which defines its operational semantics. Many variants of such an algorithm have been proposed by DEVS researchers. Yet, the proper interpretation and analysis of the com-putational complexity of such approaches have not been systematically addressed and defined. As systems become larger and more complex, the efficiency of the DEVS simu-lation algorithms in terms of time complexity measure becomes a major issue. Therefore, it is necessary to devise a method for computing this complexity. This paper proposes a generic method to address such an issue, taking advantage of the recursion embedded in the triggered-by-message principle of the DEVS simulation protocol. The applicability of the method is shown through the complexity analysis of various DEVS simulation algorithms.

Keywords: Discrete-Event Systems Specification (DEVS); Modeling and Simulation (M&S); computational complexity; simulation algorithm.

Mathematics Subject Classification 2020: 00A72

[§]Corresponding author.

1. Introduction

DEVS (Discrete-Event Systems Specification) is widely recognized as a universal framework for discrete-event simulation.[1] It proposes a sound formalism that comes along with an abstract simulator, which defines its execution semantics.[2] An abstract DEVS simulator is a technology-agnostic description of a correct DEVS simulation engine. Therefore, it abstracts away all implementation details and only focuses on the description of the operations to be performed in order to ensure the correct generation of DEVS models' behavior. This abstract description is based on a hierarchical architecture of abstract components called simulators and coordinators, and a generic message-based inter-component communication scheme called the DEVS simulation protocol. In this paper, the terms "DEVS abstract simulator" and "DEVS simulation algorithm" are interchangeably used. Since the introduction of the original abstract DEVS simulator,[1] several variants have been proposed in the literature, including sequential simulation algorithms[2–7] as well as parallel and distributed versions.[8–12] Yet, the proper measure of the computational complexity of such approaches has not been systematically addressed. As systems become larger and more complex, the efficiency of the DEVS simulation algorithms becomes a major issue. Such efficiency can be tackled from at least the following two angles:

- By experimentally comparing implementations using standardized or ad-hoc benchmarking study cases — and preferably large-scale simulation models.[13,14] The results will depend on hardware and software factors such as the processor(s) used, the memory access time, the programming language, or the compiler used.
- By theoretically comparing algorithms with each other.[15,16] The advantage of this approach over the preceding one is its independence vis-à-vis the material factors. Its disadvantage is the relative hardness to properly elaborate the theoretical measurement.

This work addresses the theoretical approach to DEVS simulation efficiency. As the DEVS simulation scheme is a tree-based architecture (see Fig. 1), it is important to observe the behavior of the simulation process when the number of simulation components grows width-wise and depth-wise. The term width refers to the number of components per level of the simulation tree, while the term depth refers to the number of levels of such a tree. The paper tries to provide a generic method to such an analysis that can apply to any DEVS abstract simulator.

At this stage, it is important to notice that, similar to the complexity analysis of sorting algorithms, the computational complexity measure of DEVS abstract simulators is not intended to provide efficient implementations, but to allow us understand how an algorithm behaves as the input (i.e., width and depth of the simulation tree generated from the structure of the DEVS model to be simulated) grows larger. Therefore, this computational complexity cannot be used for predicting/comparing specific implementations. For the latter purpose, experimental

results need to be used (taking also into account hardware/middleware-based optimizations).

In Sec. 2, related works are examined. In Sec. 3, the DEVS simulation architecture is outlined, and the reference model and notation that will be the basis for our proposal are introduced. In Sec. 4, the approach used for determining the computational complexity of DEVS abstract simulators is described. In Sec. 5, the method is applied to various well-known variants of DEVS simulation algorithms. The results obtained are discussed in Sec. 6. Section 7 concludes the paper and gives perspectives for future work.

2. Related Works

Performance evaluation and measurement constitute a very important aspect of model design.[17] Some research works have addressed the experimental evaluation of DEVS simulation performances;[13,14,18–20] few have addressed the theoretical evaluation of their computational complexity.

Among the experimental works, a noticeable one is DEVStone, a benchmark and model generator used for performance evaluation[18,21] which enables standardized and exhaustive ways to compare different DEVS-based simulation environments. A performance evaluation of different and successive versions of CD++ (a C++ implementation of DEVS-based cellular automata simulation) has been conducted using DEVStone.[16,17] In another work, the authors presented a revision of DEVStone[18] by introducing new equations and model benchmark, which they used to test five different DEVS-based environments with respect to execution performance and memory footprint on two hardware platforms. In Ref. 13, the authors considered using DEVStone to conduct a technical survey of some of the major DEVS-based simulators and software frameworks, based on a set of criteria to capture the main features of each of them, and furthermore provided a classification upon the tests conducted using these criteria. In Ref. 14, the authors presented a decisional work which enables modelers to choose the best-suited tool to solve their problem and furthermore depicted possible ways to extend (or upgrade) the studied tool.

An early work on theoretical approach to DEVS complexity is given in Ref. 22. For design consideration, a performance evaluation model is considered to find an optimal model decomposition that can be mapped onto a multiprocessor. Another work,[23] which improved upon Ref. 22, was considered by exploiting the parallelism natural to discrete models in general; in addition, the work proposed a complexity computational measure that is used to express the speed-up factor gotten from an optimal model decomposition. This work was done on the basis of the classic DEVS (CDEVS) formalism [as opposed to parallel DEVS (PDEVS); see Appendix A]. It is worth mentioning that distributed simulation in Ref. 23 refers to parallel/concurrent execution rather than the proper "distributed execution" as the term "distributed simulation" holds a much broader meaning. That is, a true distributed simulation execution would require the use of synchronization mechanisms,[24–26]

namely optimistic and conservative mechanisms, to ensure execution correctness. More recently, simulation-based activity metrics have been used to corroborate the choice of a DEVS simulation model among a set of family models based on analytic-based activity metrics.[27] The activity notion for a DEVS system is commonly referred to as the number of execution of its transition functions. However, the authors agreed that getting a good activity metric of models (in order to evaluate their performance) after the simulation is very difficult. Therefore, they do not provide any methodological guideline for the general case of any DEVS model.

Some works not theoretically driven cut into a theoretical performance analysis as a preamble of an experimental work. A performance measure, which relates to the number of atomic models, internal transition, and external transition, was proposed in Refs. 18, 19, and 28. It enabled to represent formulas that can be used to obtain a theoretical execution time, which was compared to the experimental execution time to derive the eventual overhead. A comprehensive performance measurement was proposed in Ref. 17 for DSDEVS for large-scale cellular space models. The authors considered modeling layer and simulation layer for performance analysis and proposed quantitative work on performance by considering the overhead inherent to adding or removing model during simulation in the dynamic simulation scheme. To achieve significant speed-up in execution time, the authors in Ref. 29 proposed an extension of the DEVS formalism to exploit the capability of executing hierarchical and modular DEVS models with the properties of high internal and external event parallelism.

3. DEVS Simulation Architecture

DEVS is a formal Modeling and Simulation (M&S) framework that enables models to be constructed in a hierarchical and modular composition. In the DEVS formalism, there are two basic types of models: atomic model and coupled model (which is a network of atomic models). DEVS considers not only the model behavior but also the model structure in its formalism. Model behavior has to do with model input and output events and states, while the model structure has to do with the model composition (i.e., atomic or coupled components).

A DEVS model can be expressed in either Classic DEVS[1] or Parallel DEVS.[4] The latter represents an improvement of the former by: (1) removing from the coupled model specification the tie-breaking function (Select function), which sequentializes the simulation execution by selecting at each simulation cycle only one component to be active; and (2) introducing in the atomic model specification the confluent function, which takes care of eventual simultaneous events that may occur during simulation.

The DEVS modeling formalism has been abundantly presented in the literature. The interested readers can refer to Appendix A for a short introduction. For further reading on the DEVS, interested readers can refer to a didactic presentation of

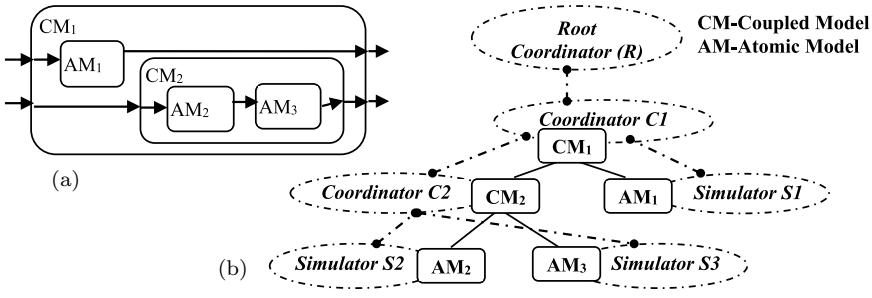

Fig. 1. (a) Hierarchy of model components and (b) the associated simulators/coordinators.

DEVS in Refs. 30 and 33, but also consult some recent works in Refs. 32, 34, and 35 for an ample description of the formalism.

3.1. *DEVS simulation tree*

For simulation executions, a hierarchical DEVS model can be mapped onto a simulation tree, as shown in Fig. 1. DEVS formalism explicitly separates the model from its execution engine. The corresponding coordinators and simulators carry out the simulation of, respectively, the coupled and atomic models.[2,5] The simulation process is dictated by the DEVS simulation algorithm used, to simulate the model of interest. However, each of these simulation algorithms has different message concepts, although following the same semantics.

For simulation executions, a hierarchical DEVS model can be mapped onto a simulation tree, as shown in Fig. 1. DEVS formalism explicitly separates the model from its execution engine. The corresponding coordinators and simulators carry out the simulation of, respectively, the coupled and atomic models.[2,5] The simulation process is dictated by the DEVS simulation algorithm used, to simulate the model of interest. However, each of these simulation algorithms has different message concepts, although following the same semantics.

3.2. *Normalized representation for DEVS simulation algorithms*

Various DEVS simulation algorithms exist in the literature. Let us introduce the following definition of our own.

Definition 1. A DEVS simulation algorithm is in the normal form if it is depicted in a triggered-by-message form, i.e., as a set of action batches to be performed, with each batch being guarded by the receipt of a given type of message.

This normalized representation of the DEVS simulation algorithms mirrors the event-driven programming paradigm, in which the flow of instructions is determined by the triggering of events (i.e., user actions) as presented by Table 1. This form makes it easy to analytically compute the time complexity of the proposed

Table 1. Pattern of DEVS simulation algorithm description in the normal form.

Message	Coordinator	Simulator
Type of message	Computation done by a coordinator at the receipt of a message of this type, and sending of new message(s) or/and forwarding of the message received	Computation done by a simulator at the receipt of a message of this type, and sending of new message(s)

algorithm, as the complexity related to the receipt of each type of message can be computed in isolation and the complexity of the overall algorithm can be derived from the recursion between the receipt of a type of message and the sending of other types of messages or/and the forwarding of the message received. This forms the foundational principles of our approach. Appendix B presents in their normal form the DEVS simulation algorithms to which our method is applied in Sec. 5.

4. Generic Approach to Complexity Analysis of DEVS Algorithms

Traditionally, the algorithmic complexity of a particular problem is tied to the size of the problem's input (as the length of an array to be sorted). In this work, the input is the DEVS simulation tree, and the size is given by two parameters: *width*, which corresponds to the number of children per coordinator, and *depth*, which corresponds to the number of levels of the DEVS simulation tree. Another important aspect is the cost of communication between two computational nodes, in the case the simulation algorithm is distributed over a computer network. The following subsections present the reference architecture and the notation introduced to support our strategy for the determination of the DEVS computational complexity.

4.1. *Reference architecture*

The study of the computational complexity of an arbitrary DEVS model is inherently complex. In order to ease the computation of the time complexity, a balanced DEVS simulation tree is used, where each coordinator (except the root coordinator) possesses the same number N of children, as shown by Fig. 2 (in that case, each coordinator has three children).

The use of a balanced tree enables the exhibition of patterns and eases the use of recursion due to the hierarchical nature of the model. Let us assume the simulation tree levels are numbered from 0 to L (= depth), starting from the leaves (therefore, the root coordinator is the L-level node).

Notice that with such an assumption, L is necessarily greater than or equal to 2. For $L = 2$, the tree is composed of the root coordinator (at level 2), the top-most coordinator (at level 1), and the N simulators (at level 0). Also, N is necessarily greater than or equal to 2, as having a node with only one child does not make sense.

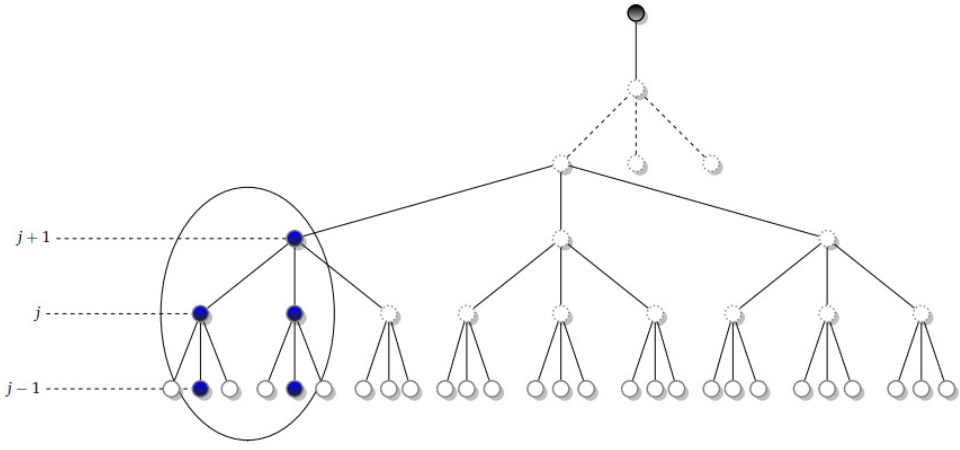

Fig. 2. Example of a balanced simulation tree.

To illustrate the principle of recursion, the sequential processing of the part enclosed within the circle at the bottom left of Fig. 2 is detailed by Fig. 3 in the case of the simulation algorithm proposed in Ref. 5. Let us start with the generation of an output event caused by the reception of an @-message by a $(j-1)$-level node (a simulator in this case). After computing its output (y), the node will send it to its parent coordinator, which in turn processes it and generates new messages to its influences, and/or forwards it to parent (if the latter is part of influences). Each of the influences other than parent will receive a q-message, and will send back a D-message when it is done with the processing of that q-message. The circled area of Fig. 3 is a zoom on the computational process triggered by the receipt of a q-message by a j-level node from its $(j+1)$-level parent node. The cascade of computations goes on, with each receipt of message triggering a batch of computations, and each batch of computations generating messages to be sent. Therefore, the computational complexity due to the starting @-message will be the sum of the complexity due to the computation of the y output by the receiving node and the one due to the sending of the message to its parent node. This latter one is computed the same way, leading to a recursion in the computation process.

4.2. *Reference notations*

There are two main approaches to the theoretical measurement of algorithm complexity: time (taken by the processor) and space (in memory). Space complexity is much less important nowadays because computers have tremendous memory. This work focuses on time complexity, which relates to a scarcer resource. The idea behind the concept of time complexity of an algorithm is mainly measuring the total time required by an algorithm to run successfully until its completion solving a particular problem.

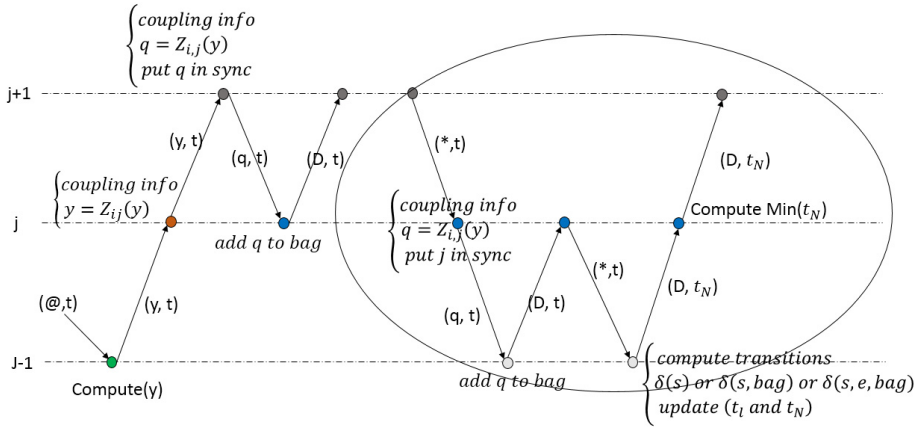

Fig. 3. Example of sequential processing based on the simulation algorithm defined in Ref. 5.

To support our strategy for complexity analysis, a notation is suggested to be adopted with all DEVS simulation algorithms. The following are defined:

M: Set of all types of messages of the algorithm.

$\tau(m, j)$: Time required to completely process a message of type m at level$j$ of the simulation tree. This may include the time needed to do local computations, send some follow-up messages to children, and receive feedback from them. In some other cases, only a local computation is done (and possibly, feedback is given to parent coordinator).

t_{com} : Time required to exchange a message between two components located at contiguous levels (i.e., parent-to-child or child-to-parent message exchange).

t_X : Time required for executing a transition function X by the simulator, $X = \{\text{int}, \text{ext}, \text{con}\}$.

N : Number of children of a coordinator.

L : Number of levels of the simulation tree.

$C(N, L)$: Cost of computations generated by the root coordinator in a cycle of simulation (i.e., one of the repetitive threads of computations performed by the root coordinator).

4.3. *Strategy to complexity analysis*

A DEVS simulation is a repetition of computation cycles until the simulated duration expires or a stopping condition occurs. The main idea in this work is to consider that the computational cost of each of these cycles is decomposable into the cost of computations resulting in the sending of the message that initiates this cycle by the root coordinator and the cost of computations of the receipt of that message by the top-most coordinator. The latter is in turn decomposable into the cost

of computations resulting in the sending of messages to the children of the top-most coordinator and the cost of computations of the receipt of those messages by the top-most coordinator's children. Recursively, this computational approach is applied until a simulation cycle is complete, i.e., when the root coordinator is ready to initiate a new recursive thread of computations.

By considering the time it would take for a message to transit between two adjacent nodes, emphasis is made on the fact that the message-passing process also affects the execution performance,[30] particularly if the model is distributed over a computing network with high latency.

The normal form provides us with a common reading grid for consistently interpreting all DEVS simulation algorithms, while the reference notation provides a unified notation for a unique computational basis. Given a DEVS algorithm in its normal form, our strategy is the following:

Step 1. For each $m \epsilon M$, do calculate $\tau(m,0)$ for top–down messages, and $\tau(m,L)$ for bottom–up messages, in the worst-case scenario.
Step 2. For each $m \epsilon M$, express $\tau(m,j)$ in the worst-case scenario either as a function of $\tau(m,0)$ for top–down messages or as a function of $\tau(m,L)$ for bottom–up messages.
Step 3. Express $C(N, L)$ as a function of N, L, and $\tau(m,j)$, $m \in M$.
Step 4. Replace in $C(N, L)$ each $\tau(m,j)$ by its corresponding function of $\tau(m,0)$ or $\tau(m,L)$.
Step 5. The width-wise complexity of the algorithm is given by $C(N,L)$ as a function of N when N grows infinitely, and the depth-wise complexity of the algorithm is given by $C(N,L)$ as a function of L when L grows infinitely.

The worst-case scenario is considered here, i.e., at a given simulation cycle all simulators are imminent and influence each other and their parents as well. Other scenarios (such as best-case and average-case scenarios) can also be considered without changing the global scheme of the approach.

5. Application

Let us illustrate here the application of our approach with the algorithm proposed in Refs. 4 and 5. The five steps previously presented are detailed hereafter.

5.1. *Step-by- step complexity measure*

Step 1. For each $m \epsilon M$, do calculate $\tau(m, 0)$ for top–down messages, and $\tau(m, L)$ for bottom–up messages, in the worst-case scenario:

- $\tau(@, 0) = 2t_{\text{com}} + \tau(y, 1) + \tau(D, 1) + a$, where a includes time to compute an output event;

- $\tau(y, L) = 0$;
- $\tau(q, 0) = t_{\text{com}} + \tau(D, 1) + b$, where b includes time to manipulate the bag;
- $\tau(*, 0) = \max(t_{\text{int}}, t_{\text{ext}}, t_{\text{con}}) + t_{\text{com}} + \tau(D, 1) + c$, where c includes time to compute t_L and t_N;
- $\tau(D, L) = t_{\text{com}} + \tau(*, L - 1)$;
- $\tau(D, L) = 0$

Note that D' is introduced such that (D',t) = (D, tN), as the algorithm distinguishes between the receipt of (D, t) and of (D, tN) by the root coordinator. Also, if the recursion had to continue over the entire simulation run, it would give $\tau(D, L) = t_{\text{com}} + \tau(@, L - 1)$, since each receipt of D' by the root coordinator initiates a new cycle by the sending of an @-message to the top-most coordinator.

Step 2. For each $m \epsilon M$, express $\tau(m, j)$ in the worst-case scenario either as a function of $\tau(m, 0)$ for top–down messages, or as a function of $\tau(m, L)$ for bottom–up messages:

- For m= @,

$$\tau(@, j) = N\tau(@, j - 1) + (N + 1)t_{\text{com}} + \tau(D, j + 1) + Nd,$$

where d includes time to handle the synchronize set. Applying the same recursive relation, it gives

$$\tau(@, j - 1) = N\tau(@, j - 2) + (N + 1)t_{\text{com}} + \tau(D, j) + Nd$$

Therefore, replacing $\tau(@, j - 1)$ by this formula in the first expression leads to

$$\begin{aligned}\tau(@, j) = N(&N\tau(@, j - 2) + (N + 1)t_{\text{com}} + \tau(D, j) + Nd) \\ &+ (N + 1)t_{\text{com}} + \tau(D, j + 1) + Nd\end{aligned}$$

Hence

$$\begin{aligned}\tau(@, j) = N^2&\tau(@, j - 2) + N(N + 1)t_{\text{com}} + N\tau(D, j) \\ &+ N^2d + (N + 1)t_{\text{com}} + \tau(D, j + 1) + Nd\end{aligned}$$

Arranged differently, it gives

$$\begin{aligned}\tau(@, j) = N^2&\tau(@, j - 2) + (N + 1)(N + 1)t_{\text{com}} \\ &+ N\tau(D, j) + \tau(D, j + 1) + (N^2 + N)d\end{aligned}$$

Let us replace in this last equation,$\tau(@, j - 2)$ by the recursive relation initially established but applied at level $j-2$, i.e., $N\tau(@, j-3)+(N+1)t_{\text{com}}+\tau(D, j-1)+Nd$, which gives

$$\begin{aligned}\tau(@, j) = N^3&\tau(@, j - 3) + N^2\tau(N + 1)t_{\text{com}} + N^2(D, j - 1) + N^3d \\ &+ (N + 1)(N + 1)t_{\text{com}} + N\tau(D, j) + \tau(D, j + 1) + (N^2 + N)d\end{aligned}$$

Arranged differently, it gives

$$\tau(@, j) = N^3\tau(@, j-3) + (N+1)(N^2+N+1)t_{\text{com}} + N^2\tau(D, j-1) \\ + N\tau(D, j) + \tau(D, j+1) + (N^3+N^2+N)d$$

The recursive relation that will successively be applied in every last equation obtained is

$$\tau(@, j-k) = N\tau(@, j-k-1) + (N+1)t_{\text{com}} + \tau(D, j-k+1) + Nd,$$

until $j-k=1$, which also means $k=j-1$. That way, it gives

$$\tau(@, j) = Nj\tau(@, 0) + (N+1)(Nj-1+Nj-2+\cdots+N+1)t_{\text{com}} \\ + Nj-1\tau(D, 2) + Nj-2\tau(D, 3) + \cdots + N^2\tau(D, j-1) \\ + N\tau(D, j) + \tau(D, j+1) + (Nj+Nj-1+\cdots+N^3+N^2+N)d$$

It is known that

$$N^{j-1} + N^{j-2} + \cdots + N + 1 = \frac{(N^j-1)}{N-1}$$

Therefore, one can write

$$\tau(@, j) = N^j\tau(@, 0) + (N+1)\frac{(N^j-1)}{N-1}t_{\text{com}} + \sum_{k=1}^{j} N^{j-k}\tau(D, k+1) + N\frac{(N^j-1)}{N-1}d$$

- For $m = y$,

$$\tau(y, j) = N\tau(q, j-1) + (N+1)t_{\text{com}} + \tau(y, j+1) + Nd + e,$$

where e includes time to handle the synchronize set and other local variables. Therefore,

$$\tau(y, j+1) = N\tau(q, j) + (N+1)t_{\text{com}} + \tau(y, j+2) + Nd + e$$

Replacing $\tau(y, j-1)$ by this expression in the equation giving $\tau(y,\ j)$, it gives

$$\tau(y, j) = N\tau(q, j-1) + (N+1)t_{\text{com}} + N\tau(q, j) \\ + (N+1)t_{\text{com}} + \tau(y, j+2) + Nd + e + Nd + e$$

Arranged differently, it gives

$$\tau(y, j) = N\tau(q, j-1) + N\tau(q, j) + \tau(y, j+2) + 2(N+1)t_{\text{com}} + 2Nd + 2e$$

Replacing $\tau(y, j+2)$ by $N\tau(q, j+1) + (N+1)t_{\text{com}} + \tau(y, j+3) + Nd + e$, it gives

$$\tau(y, j) = N\tau(q, j-1) + N\tau(q, j) + N\tau(q, j+1) + (N+1)t_{\text{com}} + \tau(y, j+3) \\ + Nd + e + 2(N+1)t_{\text{com}} + 2Nd + 2e$$

Arranged differently, it gives

$$\tau(y,j) = N\tau(q,j-1) + N\tau(q,j) + N\tau(q,j+1) + \tau(y,j+3) + 3(N+1)t_{\text{com}} + 3Nd + 3e$$

Repeating the same recursive replacement until reaching the level L, it leads to

$$\tau(y,j) = N\tau(q,j-1) + N\tau(q,j) + \cdots + N\tau(q,L-2) + (y,L) + (L-j)(N+1)t_{\text{com}} + (L-j)Nd + (L-j)e$$

Therefore, one can write

$$\tau(y,j) = \tau(y,L) + N\sum_{k=0}^{L-j-1}\tau(q,j+k-1) + (L-j)((N+1)t_{\text{com}} + Nd + e)$$

- For $m = q$

$$\tau(q,j) = b$$

- For $m = *$

$$\tau(*,j) = N\tau(qj-1) + 2Nt_{\text{com}} + N\tau(*,j-1) + Nf,$$

where f includes time to handle the local variables. By replacing $\tau(*,j-1)$ by $N\tau(qj-2) + 2Nt_{\text{com}} + N\tau(*,j-2) + Nf$, it gives

$$\tau(*,j) = N\tau(q,j-1) + 2Nt_{\text{com}} + N^2\tau(q,j-2) + 2N^{2t_{\text{com}}} + N^2\tau(*,j-2) + N^2f + Nf$$

Arranged differently, it gives

$$\tau(*,j) = N^2\tau(*,j-2) + N\tau(q,j-1) + N^2\tau(q,j-2) + 2N(N+1)t_{\text{com}} + N(N+1)f$$

By repeating the recursive replacement, it gives

$$\tau(*,j) = Nj\tau(*,0) + N\tau(q,j-1) + N^2\tau(q,j-2) + \cdots + N^j\tau(q,0) + 2N(N^{j-1} + \cdots + N + 1)t_{\text{com}} + N(N^{j-1} + \cdots + N + 1)f$$

As $N^{j-1} + N^{j-2} + \cdots + N + 1 = \frac{(N^j-1)}{N-1}$, one can write

$$\tau(*,j) = N^j\tau(*,0) + 2N\frac{(N^j-1)}{N-1}t_{\text{com}} + \sum_{k=1}^{j}N^k\tau(q,j-k) + \frac{(N^j-1)}{N-1}Nf.$$

- For $m = D$,

$$\tau(D, j) = \tau(D, j) = t\text{com} + \tau(D, j + 1) + Ng,$$

where g includes time to manipulate the local variables. By replacing $\tau(D, j + 1)$ by $t_{\text{com}} + \tau(D, j + 2) + Ng$, it gives

$$\tau(D, j) = t_{\text{com}} + t_{\text{com}} + \tau(D, j + 2) + Ng + Ng = \tau(D, j + 2) + 2t_{\text{com}} + 2Ng$$

By repeating the recursive replacement until we reach the level L, it gives

$$\tau(D, j) = \tau(D, j) = (L - j)t_{\text{com}} + N(L - j)g + \tau(D, L).$$

Step 3. Express $C(NL)$ as a function of N, L, and $\tau(m, j), m\epsilon M$:

$$C(NL) = t\text{com} + \tau(@, L - 1) + h,$$

where h includes the initialization time (before simulation cycles).

Step 4. Replace in $C(N, L)$ each $\tau(m, j)$ by its corresponding function of $\tau(m, 0)$ or $\tau(m, L)$:

From Step 2, we know (replacing j by $L - 1$) that

$$\tau(@, L) = N^{L-1}\tau(@, 0) + (N + 1)\frac{(N^{L-1} - 1)}{N - 1}t_{\text{com}} + \sum_{k=1}^{L-1} N^{L-1-k}\tau(D, k + 1) + N\frac{(N^{L-1} - 1)}{N - 1}d$$

Therefore,

$$C(N, L) = t\text{com} + h + N^{L-1}\tau(@, 0) + (N + 1)\frac{(N^{L-1} - 1)}{N - 1}t_{\text{com}}m + \sum_{k=2}^{L} N^{L-k}\tau(D, k) + N\frac{(N^{L-1} - 1)}{N - 1}d.$$

Replacing $\tau(@, 0)$ by $t_{\text{com}} + \tau(y, 1) + \tau(D, 1) + a$, which we know from Step 1, and each $\tau(D, k)$ by $(L - k)t\text{com} + N(L - k)g + \tau(D, L)$, which would be from Step 2, gives

$$C(N, L) = t_{\text{com}} + h + N^{L-1}(2t_{\text{com}} + \tau(y, 1) + \tau(D, 1) + a) + (N + 1)t_{\text{com}} + \sum_{k=2}^{L} N^{L-k}((L - k)t_{\text{com}} + N(L - k)g + \tau(D, L)) + N\frac{(N^{L-1} - 1)}{N - 1}d$$

We know from Step 1 that$\tau(D, L) = \tau(y, L) = 0$, and from Step 2 that$\tau(D, 1) = (L - 1)t_{\text{com}} + N(L - 1)g + \tau(D, L)$, as well as $\tau(y, 1) = \tau(y, L) + N\sum_{k=0}^{L-2}\tau(q, k) +$

$(L-1)((N+1)t_{\text{com}} + Nd + e)$, so we get

$$\tau C(N,L) = t\text{com} + h + N^{L-1}\left(2t\text{com} + N\sum_{k=0}^{L-2}(q,k) + (L-1)((N+1)t_{\text{com}} + Nd + e) + (L-1)t_{\text{com}} + N(L-1)g + a\right) + (N+1)\frac{(N^{L-1}-1)}{N-1}t_{\text{com}} + \sum_{k=2}^{L} N^{L-k}((L-k)t_{\text{com}} + N(L-k)g) + N\frac{(N^{L-1}-1)}{N-1}d$$

Arranged differently, we get

$$C(N,L) = t\text{com}\left(1 + 2N^{L-1} + N^{L-1}(L-1)(N+1) + N^{L-1}(L-1) + (N+1)\frac{(N^{L-1}-1)}{N-1} + \sum_{k=2}^{L} N^{L-k}(L-k)\right) + h + N^{L}\sum_{k=0}^{L-2}\tau(q,k) + N^{L-1}(L-1)(Nd+e) + NL((L-1)g + a) + Ng\sum_{k=2}^{L} N^{L-k}(L-k) + N\frac{(N^{L-1}-1)}{N-1}d$$

From Step 2, we know that $\tau(q,k) = b$, therefore it gives

$$C(NL) = t_{\text{com}}\left(1 + 2N^{L-1} + N^{L-1}(L-1)(N+2) + (N+1)\frac{(N^{L-1}-1)}{N-1} + \sum_{k=2}^{L} N^{L-k}(L-k)\right) + N^{L}(L-1)(b+g+d) + NLa + N^{L-1}(L-1)e + Ng\sum_{k=2}^{L} N^{L-k}(L-k) + N\frac{(N^{L-1}-1)}{N-1}d + h$$

Although this expression can be organized in different ways, there is not much point in doing a lot more calculations, as what matters is to derive the dominant terms when L (respectively, N) grows infinitely.

Step 5. The width-wise complexity of the algorithm is given by $C(N,\ L)$ as a function of N when N grows infinitely, and the depth-wise complexity is given by $C(N,\ L)$ as a function of L when L grows infinitely:

Table 2. Computational loads per cycle of some DEVS simulation algorithms.

Algorithms of	$C(N,L)$
Chow *et al.*[4,5]	$(t_{\text{com}}+\alpha)N^L(L-1)+(2t_{\text{com}}+\beta)N^{L-1}(L-1)+(2t_{\text{com}}+\gamma)N^{L-1}+t_{\text{com}}N+\sum_{k=1}^{L}N^{L-k}(L-k)(t_{\text{com}}+\delta)+2t_{\text{com}}+\varepsilon$
Chow[3]	$N^L(L-1)(t_{\text{com}}+\alpha)+N^{(L-1)}(L-1)t_{\text{com}}+N^{(L-1)}(2t_{\text{com}}+\beta)+N^{L+1}\frac{\lambda}{N-1}\frac{N^{(L-1)}-1}{N-1}-N^{(L+1)}\frac{\gamma}{N-1}(L-2)+N^L(\frac{N^{L-1}-1}{N-1})\delta+N(\frac{N^{L-1}-1}{N-1})(2t_{\text{com}}+\omega\varepsilon)+t_{\text{com}}+\varepsilon$
Zeigler *et al.*[2]	$N^L(L-1)(t_{\text{com}}+\alpha)+N^{(L-1)}(L-1)t_{\text{com}}+N^{(L-1)}(t_{\text{com}}+\beta)+N^{(L+1)}(\frac{t_{\text{com}}+\lambda}{N-1})(\frac{N^{(L-1)}-1}{N-1})-N^{(L+1)}(\frac{t_{\text{com}}+\gamma}{N-1})(L-2)+N^L(\frac{N^{(L-1)}-1}{N-1})\delta+N(\frac{N^{(L-1)}-1}{N-1})t_{\text{com}}+t_{\text{com}}+\varepsilon$
Schwatinski and Pawletta[6]	$L(L-1)N^{L-1}(t_{\text{com}}+\alpha)+\beta(L-1)N^{L-1}+N^{L-1}\lambda+\frac{(N^L-1)}{N-1}(2t_{\text{com}}+\gamma)t_{\text{com}}+\delta$

When N grows infinitely (L being constant), the dominant terms are the higher-degree monomials in N, as $C(N,L)$ is then a polynomial in N. These terms are $t_{\text{com}}N^L(L-1)+N^L(L-1)(b+g+d)+NLa$. Therefore,$C(N,L)\approx(t_{\text{com}}+\alpha)LN^L$, where α aggregates all costs for manipulating auxiliary variables (such as synchronize set, time variables, etc.).

On the other hand, when L grows infinitely (N being constant), the dominant terms are the higher-degree exponential functions in L. Therefore,$C(N,L)\approx(t_{\text{com}}+g)N^LL^2$.

Each complexity measure computed is presented by decreasing order first on the power of N, and then on L. The whole idea in arranging terms the way they appear in Table 2 is to quickly identify the most significant terms to be retained when L or N grows. That way, the complexity order can easily be derived from these expressions presented in Table 2.

For example, consider $C(N,L)$ for Chow *et al.*[4,5] as established in Sec. 5.1. We can develop it in the following way:

$$\begin{aligned}C(N,L) &= t_{\text{com}}+2t_{\text{com}}N^{L-1}+N^L(L-1)t_{\text{com}}+N^{L-1}(L-1)2t_{\text{com}}\\ &\quad+N^{L-1}\frac{(N+1)}{N-1}t_{\text{com}}-\frac{(N+1)}{N-1}t_{\text{com}}+\sum_{k=2}^{L}N^{L-k}(L-k)t_{\text{com}}\\ &\quad+N^L(L-1)(b+g+d)+N^La+N^{L-1}(L-1)e\\ &\quad+g\sum_{k=1}^{L-1}N^{L-k}(L-k)+N^{L-1}\frac{N}{N-1}d-\frac{N}{N-1}d+h\end{aligned}$$

We can arrange this expression by starting with the terms containing $N^L(L-1)$, then terms containing $N^{L-1}(L-1)$, subsequently the terms containing N^{L-1}, and

finally the rest. This gives

$$
\begin{aligned}
C(N,L) = {} & N^L(L-1)t_{\text{com}} + N^L(L-1)(b+g+d) + N^{L-1}(L-1)2t_{\text{com}} \\
& + N^{L-1}(L-1)e + 2t_{\text{com}}N^{L-1} + N^{L-1}\frac{N}{N-1}d \\
& + N^{L-1}\frac{(N+1)}{N-1}t_{\text{com}} - \frac{(N+1)}{N-1}t_{\text{com}} + \sum_{k=2}^{L} N^{L-k}(L-k)t_{\text{com}} \\
& + N^L a + g\sum_{k=1}^{L-1} N^{L-k}(L-k) - \frac{N}{N-1}d + t_{\text{com}} + h
\end{aligned}
$$

Then, some terms can be grouped together to form constant/negligible values (Latin letters).

To derive the complexity order from the resulting expressions (for each algorithm or abstract simulator), only higher-degree monomials in N are to be retained when N grows infinitely (L being constant) since $C(N,L)$ is then a polynomial in N, and only higher-degree exponential functions in L are to be retained when L grows infinitely (N being constant). Table 3 depicts the complexity for each of the simulation algorithms (i.e., abstract simulators) studied in the work (see the computational details in Appendix C).

Algorithms of Chow *et al.*[4,5] and Schwatinski and Pawletta[6] have a better width-wise complexity than those of Chow[3] and Zeigler *et al.*[2] For example, for $L = 2$ (as already mentioned in Sec. 4.1, L is necessarily greater than or equal to 2), the algorithms of Chow *et al.*[4,5] as well as Schwatinski and Pawletta[6] have a width-wise complexity of $O(N^2)$, while those of Chow[3] as well as Zeigler*et al.*[2] have a width-wise complexity of $O(N^4)$.

This does not mean that the specific implementations of Chow *et al.*[4,5] or Schwatinski and Pawletta[6] necessarily outperform specific implementations of Chow[3] or Zeigler *et al.*[2] Rather, it means that if the models to be simulated have more and more immediate sub-components (for example, from a coupled model having 1000 atomic model components to a coupled model having 1000 atomic model components), the degradation of execution performances observed for any specific implementation (whether optimized or not) of Chow[3] or Zeigler *et al.*[2] will be worse than the one observed for an implementation (whether optimized or not) of Chow *et al.*[4,5] or Schwatinski and Pawletta.[6]

Table 3. Complexity order of some DEVS simulation algorithms ($L \geq 2$).

Complexity	Chow *et al.*[4,5]	Chow[3]	Zeigler *et al.*[2]	Schwatinski and Pawletta[6]
Width-wise	$O(N^L)$	$O(N^{2L})$	$O(N^{2L})$	$O(N^L)$
Depth-wise	$O(L^2N^L)$	$O(LN^L)$	$O(LN^L)$	$O(L^2N^L)$

On the contrary, Chow[3] and Zeigler *et al.*[2] have a better depth-wise complexity than Chow *et al.*[4,5] and Schwatinski and Pawletta.[6] For example, for $N = 2$ (as already mentioned in Sec. 4.1, N is also necessarily greater than or equal to 2), Chow *et al.*[4,5] as well as Schwatinski and Pawletta[6] have a depth-wise complexity of $O(L^2 2^L)$, while Chow[3] as well as Zeigler *et al.*[2] have a depth-wise complexity of $O(L2^L)$.

Therefore, if the models to be simulated are more and more decomposed into sub-components and sub-sub-components (for example, from a coupled model having two model components, each of which has two model components in turn, and so on to level 10, to a coupled model having two model components, each of which has two model components in turn, and so on to level 1000), the degradation of execution performances observed for any specific implementation (whether optimized or not) of Chow *et al.*[4,5] or Schwatinski and Pawletta[6] will be worse than the one observed for an implementation (whether optimized or not) of Chow[3] or Zeigler *et al.*[2]

6. Experimental Results

Here, our theoretical results are partly assessed with experimental results obtained from a specific implementation of the DEVS abstract simulators. The focus is on Chow *et al.*'s[4,5] algorithms, which have been implemented in the Java-programmed SimStudio package.[31]

Let us consider a disease-spreading situation in a population composed of individuals, each represented by an atomic model. Each individual has a health status, which is either Susceptible, Infected, or Recovered. While a susceptible individual will remain in his/her health status until being infected, an infected individual will recover after five days (incubation time), and a recovered individual will become susceptible after two days (immune time). A susceptible individual gets infected when in touch with two infected individuals.

The DEVS model of such an individual is defined as follows:

$$M_{\text{Individual}} = \langle X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{conf}}, \lambda, ta \rangle,$$

where

- $X = \{(CONTACT, v) \,|\, v \in \{\text{Susceptible, Infected, Recovered}\}\}$, CONTACT is the name of the unique input port of the model and v is the value received on it;
- $Y = \{(HEALTH, v) \,|\, v \in \{\text{Susceptible, Infected, Recovered}\}\}$, HEALTH is the name of the unique output port of the model and v is the value sent on it;
- $S = \{(status, \sigma) \,|\, \text{status} \in \{\text{Susceptible, Infected, Recovered}\}, \sigma \in \Re\}$, status is the current health status of the individual, while σ is the remaining time to be in that status;
- ta(status, σ) = σ;

- δ_{int} (Infected, σ) = (Recovered, 2), δ_{int} (Recovered, σ) = (Susceptible, $+\infty$), time units are in days, and there is no possible internal transition from the *Susceptible* health status (only a received infection can trigger a transition from that status);
- δ_{ext}((status, σ), e, x) = (Infected, 5) if x contains at least two Infected statuses, (status, $\sigma - e$) otherwise.

Both the width-wise and the depth-wise growths of the model are experimented:

- For the width-wise growth, let us consider $L = 2$ [for which it is known the width-wise complexity is $O(N^2)$], and let us gradually increase the number of individuals in the population. That way, the corresponding simulation tree (which has only a root coordinator at level 2, a top-most coordinator at level 1, and several children at level 0) will gradually have more and more leaves.
- For the depth-wise growth, let us consider $N = 2$ [for which it is known the depth-wise complexity is $O(L^2 2^L)$], and let us initially decompose the population into two clusters, each of which is made of two individuals. Then after, let us gradually decompose each cluster into two sub-clusters, each of which is made of two individuals. That way, the corresponding simulation tree will gradually have more and more levels, with each node at a given level having two children.

The disease-spreading model is always fully connected, i.e., all components within any given component send their statuses to each other. Figure 4 shows the principle of the evolution of the model's structure, width-wise and depth-wise.

Simulation experiments are executed for various scenarios of population's structure, and each experiment lasts for 100 simulated days. Figure 5 depicts the increase of execution time (ET) in milliseconds, as the model grows width-wise (from $N = 2$ to $N = 1024$), while Fig. 6 compares the experimental results (dotted line) with the theoretical results (straight line) established in the previous sections. The execution performances are averaged over 10 simulation executions for each scenario. Obviously, Fig. 6 assesses the worst-case nature of the complexity measure (as compared to the experimental results), and Fig. 5 assesses whether the experimental performances degrade akin to the theoretical ones.

Figure 7 depicts the increase of execution time as the model grows depth-wise (from $L = 2$ to $L = 10$) with a fixed value of N (i.e., 2).

Figure 8 compares the experimental and corresponding theoretical results.

Note that the model considered in this work is a balanced hierarchical DEVS model, where at each level the number of subordinates per coordinator is the same. In practice, DEVS hierarchical models are not balanced. In such a case, our approach can be used to find the upper and lower bounds to the computational complexity to be determined. That is, for any regular DEVS model, one can find two DEVS balanced models that prove the boundedness of the complexity measure of the regular model in terms of complexity analysis.

7. Conclusion

This paper presents a generic approach to the determination of the computational complexity of DEVS simulation algorithms, which exploits their recursive nature when they are expressed in a triggered-by-message form. The approach is based on a balanced simulation tree and uses a reference notation that can apply to any DEVS algorithm. The computational load generated by each type of message of the simulation protocol is first computed, and then used to derive the overall load of the algorithm per simulation cycle. The computational complexity results from the asymptotic growth of the width or the depth of the simulation tree. The approach is applied to four DEVS simulation algorithms, and a comparative study of them is provided.

This approach can be adopted with any DEVS simulation algorithm, whether a variant of CDEVS or PDEVS, or even a DEVS extension that is expressed with CDEVS or PDEVS. However, this paper focuses only on sequential simulation, and only on balanced simulation trees. For simulation that employs a nonbalanced DEVS model, the approach provides lower and upper bound estimates of the complexity. For distributed simulation, additional message types have to be introduced in the DEVS algorithm to ensure the necessary synchronization between simulation components that interact remotely. This has no impact on the applicability of our

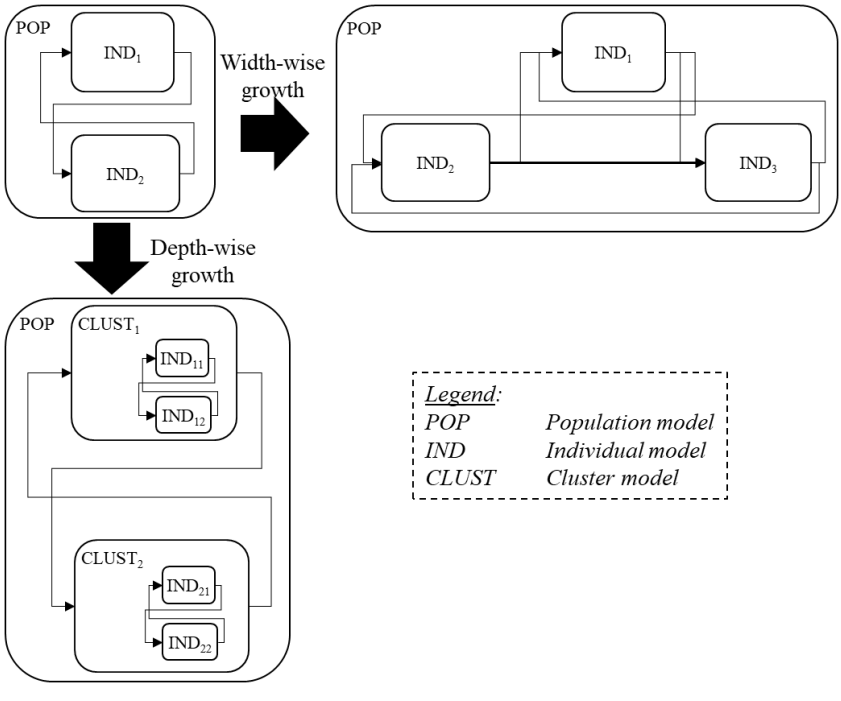

Fig. 4. Experimental model (width-wise and depth-wise growths).

approach, but the number of processors used must also be considered, as well as the way the communication time between two adjacent simulation nodes varies. Examining the case of parallel and distributed DEVS simulation is envisioned as future work.

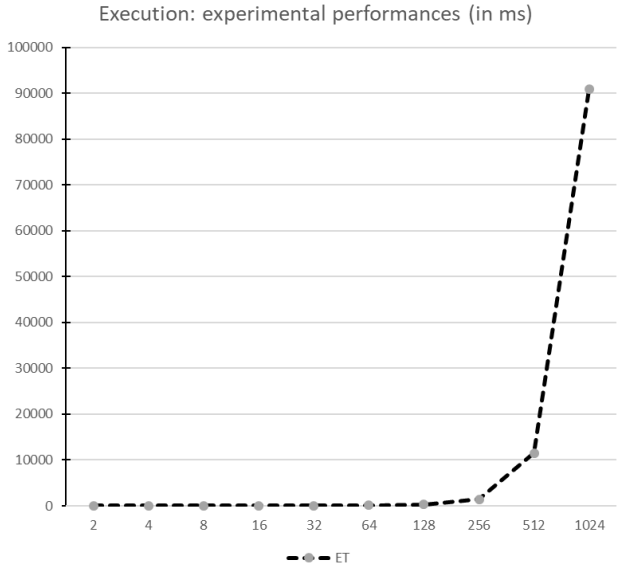

Fig. 5. Execution performances as the model grows width-wise (from $N = 2$ to N = 1024).

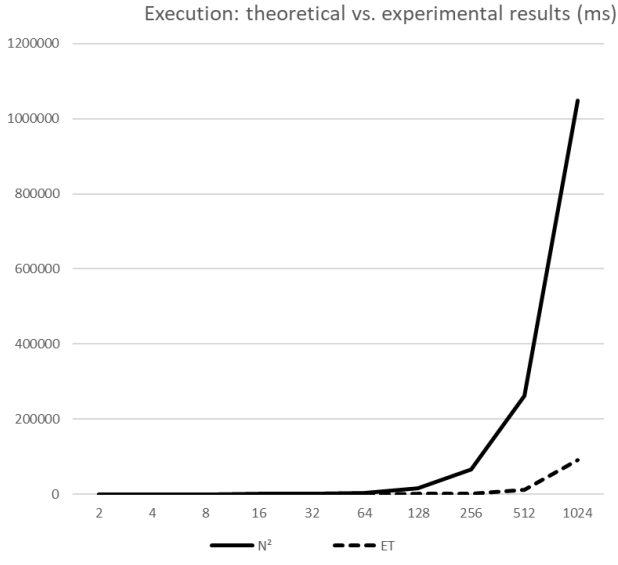

Fig. 6. Theoretical versus experimental results (width-wise growth; from $N = 2$ to $N = 1024$).

It is worth mentioning that even if the focus of this approach is on finding a way to measuring the computational complexity of a DEVS abstract simulator, rather than improving the existing ones, it can serve as a computational mean to comparing various solutions and demonstrating any improvement from one solution to another.

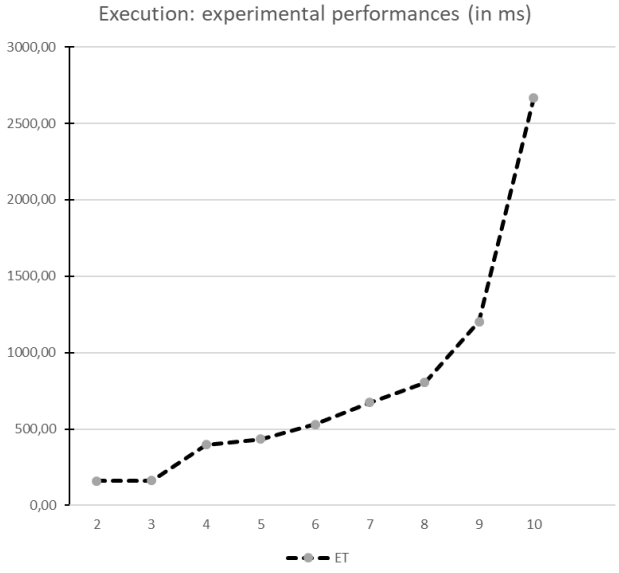

Fig. 7. Execution performances as the model grows width-wise (from $L = 2$ to $L = 10$).

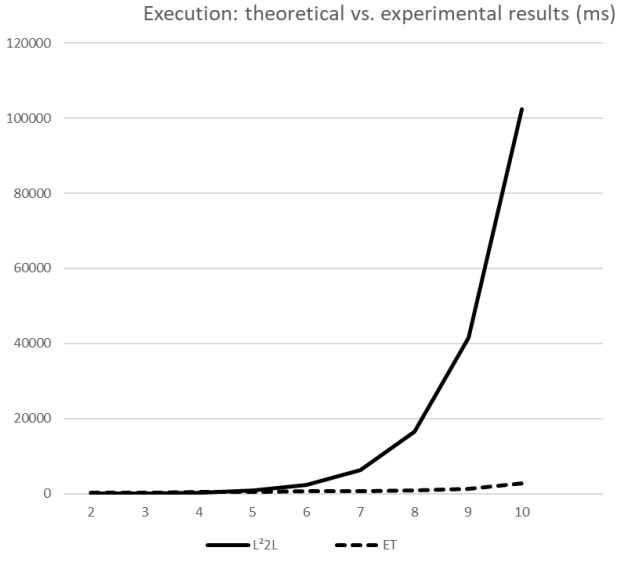

Fig. 8. Theoretical versus experimental result (depth-wise growth; from $L = 2$ to $L = 10$).

Acknowledgments

This work has been partially supported by the African Development Bank (AfDB).

Appendix A. DEVS Modeling Formalism

DEVS formalism was initially known as CDEVS while presenting some limitations to perform parallel implementation. Some of its limitations include a tie-breaking function that handles simultaneously occurring internal transitions of the components of a coupled model, and the fact that it ignores an internal transition function while occurring at the same time with an external input event (collision), in which case the external transition function always takes place. PDEVS has been introduced to alleviate this drawback.[4] In this paper, PDEVS is referred to as DEVS.

An atomic DEVS model is defined by the n-tuple $\langle X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{conf}}, \lambda, ta\rangle$, where

- X, Y, and S are, respectively, the input set, output set, and state set (at any time, the system modeled is in one of the possible states);
- ta: $S \to \Re_0^{+\infty}$ is the time advance function (i.e., it gives the lifespan of each state), with $\Re_0^{+\infty}$ designating the set of nonnegative real numbers, including $+\infty$;
- δ_{int}: $S \to S$ is the internal transition function (i.e., it is triggered only when the elapsed time in the system's current state s_{curr} has reached $\text{ta}(s_{\text{curr}})$ without the system being disturbed by any receipt of input);
- $\lambda : S \to Y$ is the output function (i.e., it computes the output of the system, each time an internal transition is occurring);
- δ_{ext}: $Q \times X \to S$ is the external transition function [i.e., it is triggered only when the system receives an input, while the elapsed time in the system's current state s_{curr} has not reached $\text{ta}(s_{\text{curr}})$], and $Q = \{(s, e) \mid s \in S, 0 \leq e < ta(s)\}$ is called the total state;
- δ_{conf}: $S \times X \to S$ is the confluent transition function [i.e., it is triggered only when the system receives an input at exactly the time when the elapsed time in the system's current state s_{curr} has reached $\text{ta}(s_{\text{curr}})$].

The operational semantics of an atomic model is informally described as follows: At the start, the system is in an initial state and remains there until the time specified by ta is exhausted or until the input event is received. In the former case, an internal transition function occurs then the system switches to another state after sending output event as defined by the output function λ. In the latter case, if input event is received before the specified time, then the external transition function is applied. When a collision occurs, i.e., an external event is received concurrently with the elapsed time equal to the time specified by the time advance function, the confluent function is applied in such a way that the system sends output value and changes to a new state.

A coupled DEVS model is a structure: $\langle X_{\text{self}}, Y_{\text{self}}, \{M_d\}d \in D, \{I_d\}d \in D, \{Z_{i,j}\}i \in D \cup \{\text{self}\}, \text{j} \in \text{I}_\text{i}\rangle$ where

- X_{self} and Y_{self} are defined the same way as X and Y are for atomic models (self being here a reference to the coupled model, while component models are referred to using the indices such as i, j, or d);
- D is the set of component references (thus, not including self);
- M_d is the component model referenced by d, an atomic or a coupled model, with X_d and Y_d as, respectively, its input and output sets;
- I_d is the influence set of component model d, i.e., all other models sending input to d;
- $Z_{\text{self},d\in I_{\text{self}}}: X_{\text{self}} \to X_d$ are the external input transfer functions, which determine how the inputs received by self are translated into the inputs to component models influenced by self;
- $Z_{d/\text{self}\in I_d,self} : Y_d \to Y_{\text{self}}$ are the external output transfer functions, which determine how the outputs sent by component models influencing self are translated into the outputs of self;
- $Z_{i\in D,j\in D-\{i\}}: Y_i \to X_j$ are the internal transfer functions, which determine how the outputs sent by component models are translated into the inputs to component models they influence.

Appendix B. DEVS Abstract Simulation Algorithms in Normal Form

Tables B.1–B.4 given hereafter present the simulation algorithms of coordinators, simulators, and the root coordinator, respectively, of Chow *et al.*,[4,5] Chow,[3] Zeigler *et al.*,[2] and Schwatinski and Pawletta.[6]

Table B.1. Simulation algorithms of Chow *et al.*[4,5]

Message	Coordinator	Simulator
$(@, t)$	If $t = t_N$ then $t_L = t$ Send $(@, t)$ to IMM children Cache child in synchronize set Wait until (D, t) s are received Send (D, t) to parent	Check the simulation time If $t = tN$ then Compute $y = \lambda(s)$ Send (y, t) to parent Send (D, t) to parent
(y, t)	For all influencees j of child i do $q = Z_{i,j}(y)$ Send (q, t) to child j Cache j in synchronize set Wait until (D, t) s are received If self is in I_i $y = Z_{i,\text{self}}(y)$ Send (y, t) to parent	
(q, t)	Add event q to the bag	Add event q to the bag Send (D, t) to parent coordinator

(*Continued*)

Table B.1. (*Continued*)

Message	Coordinator	Simulator
$(*, t)$	If $tl \leq t < tN$ then For all j in I_{self} and all q in bag $q = Z_{\text{self},j}(q)$ Send (q, t) to j Cache $j$in synchronize set Empty bag Wait until (D, t) s are received For all i in synchronize set Send $(*, t)$ to i	Case $t_L \leq t < t_N$ and bag $\neq$ empty $e = t - t_L$ $s = \delta_{\text{ext}}(se\text{bag})$ Empty bag Case $t = t_N$and bag is empty $s = \delta_{\text{int}}(s)$ Case$t = t_N$ and bag is not empty $s = \delta_{\text{con}}(s\text{bag})$ Emptybag Case$t > t_N$or$t < t_L$Raise error $t_L = t$ and $t_N = t + \text{ta}(s)$ Send$(D, t_N)$to parent coordinator
(D, t)	$tl = t$ Compute minimum $t_N$of component's t_N Clear the synchronize set Send(Dt_N) to parent coordinator	

Message	Root coordinator
(D, tN)	Send $(@, t)$ to the top-most coordinator
(D, t)	Send $(*, t)$ to the top-most coordinator

Table B.2. Simulation algorithms of Chow.[3]

Message	Coordinator	Simulator

$(*, x_{\text{count}}t)$	Compute IMM and INF
	For each $e$in IMM $\cup$ INF
	Calculate i_count
	If e is an influencee of self then
	i_count = i_count + x_count
	Send$(*i_\text{count}; t)$ to e
	Increment count
		count = x_count
		If $t = tN$then
		Send$(\#\lambda(s)t)$ to the parent
		If count = then
		$s = \_\text{int}(s)$
		Else
		Block until count = 0
		$s = \delta_{\text{con}}(sx^b)$
		Else
		Block until count= 0
		$s = \delta_{\text{ext}}(s, t - t_L x^b)$
		$t_N = t + \text{ta}(s)$ and $t_L = t$
		Emptyx^b
		Send(Dt_N) to parent coordinator

$(\#, y, t)$	Use $Z_{\text{source},j}$ s If target is the current coupled model d (self) Send$(\#, Z_{\text{source,self}}(y)t)$ to parent Else Send$(\#; Z_{\text{source,target}}(x), t)$ to target	

Table B.2. (*Continued*)

Message	Coordinator	Simulator
$(\#, x, t)$	Use $Z_{\text{source},j}$ s For each child r in the target Send $(\#Z_{\text{source,target}}(x), t)$ to r	Block if count = 0 Lock x^b $x^b = x^b ? x$ Unlock x^b Increment count
(D, t)	Block when count = 0 Cache tN and component sending this message Decrementcount	

Message	Root coordinator
(D, t_N)	$t = t_N$ Send$(*, t)$ to the top-most coordinator

Table B.3. Simulation algorithms of Zeigler *et al.*[2]

Message	Coordinator	Simulator
$(*, t)$	Compute the set IMM in cache with $t = t_N$ Send $(*, t)$ to e in IMM	Check the simulation time If $t = tN$ then compute $y = \lambda(s)$ Send (y, t) to parent coordinator
(y, t)	Add (y_d, d) to mail and mark d as reporting If this is the last d in IMM $y_{\text{parent}} = \emptyset$ For each d in $I_N \& d$ is reporting If $Z_{d,N}(y_d) \neq \emptyset$ then Add y_d to y_{parent} Send y-message(y_{parent}, t) toparent For each $d \epsilon I_r \& d$reporting$\& Z_{d,r}(y_d) \neq \emptyset$ Add $Z_{d,r}(y_d)$ to y_r Send $x -$ messages (y_r, t) to r Foreach e inIMM$\& y_r = \emptyset$ Send $x -$ message $(\emptyset, t)$ to e	
(x, t)	Compute the set receivers (R) of self (N) For each r in $R \& r$ in I_N Send $x -$ message $(Z_{N,r}(x), t)$ to r For each r in IMM and not in R Send $x -$ message $(\emptyset, t)$to$r$	If $(x = \emptyset$ and $t = t_N)$ then $s = \delta_{\text{int}}(s)$ ElseIf$(x \neq \emptyset$ and $t = t_N)$ $s = \delta_{\text{con}}(sx)$ Else$(x \neq \emptyset$ and $(tl \leq t < t_N))$ $e = t - tl$ $s = \delta_{\text{ext}}(sx)$ $tl = t$ and $t_N = tl + \text{ta}(s)$
(D, t)	$tl = t$ $t_N = \min t_{N_d} \vert d \text{in} D$	

Message	Root coordinator
(Dt_N)	$t = t_N$ Send $(*, t)$ to the top-most coordinator

Table B.4. Simulation algorithms of Schwatinski and Pawletta.[6]

Message	Coordinator	Simulator
$(*, t)$	If mail is empty then Set layer=active Send (Y, t) to all children in IMM Wait until mail is not empty and layer = passive For all nonempty y-messages in mail Send (x, t) to corresponding receivers Send $(*, t)$ to all children in IMM with no input in mail Set layer to passive and mail to empty Update tl and t_N	$s = \delta_{\text{int}}(s)$
(Y, t)	Forward Y-message to all children in IMM	$y = \lambda(s)$ Send (y, t) to parent coordinator

(x, t)	If IMM$\neq \emptyset$ then
	Save x in mail
	Send $(*, t)$ to oneself
	Else
	Forward (x, t) according to $Z_{i,d}$
		If $t = t_N$ then
		$s = \delta_{\text{con}}(sex)$
		If $tl \leq t < t_N$ then
		$s = \delta_{\text{ext}}(sex)$
		Update tl andt_N
		Send (Dt_N) to parent

(y, t)	Save y in mail
	Wait until all (y, t) are received from
	children in IMM
	If layer=passive then
	If mail is not empty
	Send all (y, t) in mail to supCoord
	Else
	Send empty (y, t) to supCoord
	Else
	Set layer to passive	

(D, t)	$tl = t$ $t_N = \min t_{N_d} \mid d$ in D Send (D, t_N) to parent coordinator	

Message	Root coordinator
(D, t_N)	$t = t_N$ Send $(*, t)$ to the top-most coordinator

Appendix C. Theoretical Complexity of Some DEVS Simulation Algorithms

C.1. *Simulation algorithms of Chow*[3]

Step 1. For each $m \in M$, do calculate $\tau(m, 0)$ for top–down messages, and $\tau(m, L)$ for bottom–up messages, in the worst-case scenario:

- $\tau(*, 0) = \max(t_{\text{int}}, t_{\text{ext}}, t_{\text{con}}) + 2t_{\text{com}} + \tau(y, 1) + \tau(D, 1) + a$, where a includes time to compute an output event and to manipulate local variables;

- $\tau(y, L) = 0$;
- $\tau(x, 0) = b$ where b includes time to manipulate the bag;
- $\tau(D, L) = 0$, as each D-message received by the root coordinator initiates a new cycle.

Step 2. For each $m \in M$, express $\tau(m, j)$ in the worst-case scenario either as a function of $\tau(m, 0)$ for top–down messages, or as a function of $\tau(m, L)$ for bottom–up messages:

- $\tau(*, j) = N\tau(*, j-1) + Nt_{\text{com}} + Nc$, where c includes time to manipulate local variables,

$$\Rightarrow \tau(*, j) = N^j\tau(*, 0) + N\frac{(N^j - 1)}{N - 1}(t\text{com} + c);$$

- $\tau(y, j) = N\tau(xj - 1) + (N + 1)t_{\text{com}} + \tau(y, j + 1) + Nd$, where d includes time to manipulate local variables,

$$\Rightarrow \tau(y, j) = \tau(y, L) + N\sum_{k=0}^{L-j-1}\tau(x, j + k - 1) + (L - j)((N + 1)t_{\text{com}} + Nd);$$

- $\tau(x, j) = N\tau(x, j - 1) + Nt_{\text{com}} + Ne$, where e includes time to manipulate local variables,

$$\Rightarrow \tau(x, j) = N^j\tau(x, 0) + N\frac{(N^j - 1)}{N - 1}(t_{\text{com}} + e);$$

- $\tau(D, j) = f$, where f includes time to manipulate local variables.

Step 3. Express $C(N, L)$ as a function of N, L, and $\tau(m, j), m \in M$:

$$\tau C(NL) == t\text{com} + (*, L - 1) + g,$$

where g includes time to initialize the simulation (before entering the loop of simulation cycles).

Step 4. Replace in $C(N, L)$ each $\tau(m, j)$ by its corresponding function of $\tau(m, 0)$ or $\tau(m, L)$:

$$\begin{aligned} C(N, L) \;&=\; t_{\text{com}} + g + N^{L-1}\tau(*, 0) + N\frac{(N^{L-1} - 1)}{N - 1}(t_{\text{com}} + c) \\ &\Rightarrow C(N, L) = t_{\text{com}} + g + N^{L-1}(\max(t\text{int}, t\text{ext}, t\text{con}) + 2t_{\text{com}} + \tau(y, 1) \\ &\quad + \tau(D, 1) + a) + N\frac{(N^{L-1} - 1)}{N - 1}(t_{\text{com}} + c) \end{aligned}$$

with $\tau(y, 1) = \tau(y, L) + N\sum_{k=0}^{L-2}\tau(x, mk) + (L - 1)((N + 1) + Nd)$, and $\tau(D, 1) = f$.

Step 5. The width-wise complexity of the algorithm is given by $C(N,\ L)$ as a function of N when N grows infinitely, and the depth-wise complexity is given by

$C(N, L)$ as a function of L when L grows infinitely:

When N grows infinitely (L being constant, and given $L \geq 2$), $C(N, L) \approx (\lambda + \delta)N^{2L-2}$, and when L grows infinitely (N being constant), $C(N, L) \approx (2t\text{com} + \alpha + \gamma)N^L L$. Notice the condition $L \geq 2$ always holds, otherwise the simulation tree is reduced to the root coordinator and its child (which therefore is necessarily a simulator) and there is no way for N to grow (as there is no possible top-most coordinator).

C.2. *Simulation algorithms of Zeigler et al.*[2]

Step 1. For each $m \epsilon M$, do calculate $\tau(m, 0)$ for top–down messages, and $\tau(m, L)$ for bottom–up messages, in the worst-case scenario:

- $\tau(*, 0) = t_{\text{com}} + \tau(y, 1) + a$, where a includes time to compute an output event;
- $\tau(y, L) = 0$;
- $\tau(x, 0) = \max(t_{\text{int}}, t_{\text{ext}}, t_{\text{con}}) + b$, where b includes time to manipulate local variables;
- $\tau(D, L) = 0$, as each D-message received by the root coordinator initiates a new cycle.

Step 2. For each $m \in M$, express $\tau(m, j)$ in the worst-case scenario either as a function of $\tau(m, 0)$ for top–down messages, or as a function of $\tau(m, L)$ for bottom–up messages:

- $\tau(*, j) = Nt_{\text{com}} + N\tau(*, j-1) + c$, where c includes time to manipulate local variables,

$$\Rightarrow \tau(*, j) = N^j \tau(*, 0) + \frac{(N^j - 1)}{N - 1}(Nt_{\text{com}} + c);$$

- $\tau(y, j) = N\tau(xj - 1) + (N + 1)t_{\text{com}} + \tau(y, j + 1) + Nd$, where d includes time for variables update,

$$\Rightarrow \tau(y, j) = \tau(y, L) + N \sum_{k=0}^{L-j-1} \tau(x, j + k - 1) + (L - j)((N + 1)t_{\text{com}} + Nd);$$

- $\tau(x, j) = N\tau(x, j - 1) + Nt_{\text{com}} + Ne$, where $e$includes time to manipulate local variables,

$$\Rightarrow \tau(x, j) = N^j \tau(x, 0) + N\frac{(N^j - 1)}{N - 1}(t_{\text{com}} + e);$$

- $\tau(D, j) = Nf$, where f includes time to manipulate local variables.

Step 3. Express $C(N, L)$ as a function of N, L, and $\tau(m, j), m \epsilon M$:

$$\tau C(N, L) = t_{\text{com}} + (*, L - 1) + g,$$

where g includes time to initialize the simulation.

Step 4. Replace in $C(N, L)$ each $\tau(m, j)$ by its corresponding function of $\tau(m, 0)$ or $\tau(m, L)$:

$$C(N,L) = t_{\text{com}} + g + N^{L-1}\tau(*,0) + \frac{(N^{L-1}-1)}{N-1}(Nt_{\text{com}} + c)$$

$$\Rightarrow C(N,L) = t_{\text{com}} + g + N^{L-1} + (\max(t_{\text{int}}, t_{\text{ext}}, t_{con}) + 2t_{\text{com}} + \tau(y,1) + \tau(D,1) + a) + N\frac{(N^{L-1}-1)}{N-1}(Nt_{\text{com}} + c),$$

with $\tau(y,1) = \tau(y,L) + N\sum_{k=0}^{L-2}(x,k) + (L-1)((N+1)t_{\text{com}} + Nd)$.

Step 5. The width-wise complexity of the algorithm is given by $C(N,\ L)$ as a function of N when N grows infinitely, and the depth-wise complexity is given by $C(N,\ L)$ as a function of L when L grows infinitely:

When N grows infinitely (L constant, and $L \geq 2$),$C(N,L) \approx (t\text{com} + \lambda + \delta)N^{2L-2}$, and when L grows infinitely (N constant), $C(N,L) \approx (3t\text{com} + \alpha + \gamma)N^L L$

C.3. *Simulation algorithms of Schwatinski and Pawletta*[6]

Step 1. For each $m \epsilon M$, do calculate $\tau(m, 0)$ for top–down messages, and $\tau(m, L)$ for bottom–up messages, in the worst-case scenario:

- $\tau(*, 0) = t_{\text{int}}$;
- $\tau(Y, 0) = t_{\text{com}} + \tau(y, 1) + a$, where a includes time to compute an output event;
- $\tau(x, 0) = t_{\text{com}} + \tau(D, 1) + \max(t_{\text{ext}}, t_{\text{con}}) + b$, where b includes time to update time variables;
- $\tau(y, L) = 0$;
- $\tau(D, L) = 0$, as each D-message received by the root coordinator initiates a new cycle.

Step 2. For each $m \epsilon M$, express $\tau(m,j)$ in the worst-case scenario either as a function of $\tau(m,0)$ for top–down messages, or as a function of $\tau(m,L)$ for bottom–up messages:

- $\tau(*, j) = N\tau(Y, j-1) + N\tau(x, j-1) + 2Nt\text{com} + c$, where c includes time to manipulate local variables;
- $\tau(Y, j) = N\tau(Y, j-1)$

$$\Rightarrow \tau(Y, j) = N^j \tau(Y, 0);$$

- $\tau(x, j) = \tau(*, j)$

$$\Rightarrow \tau(x,j) = \tau(*,j) = N^j\tau(*,0) + \sum_{k=1}^{j} N^k \tau(Y, j-k) + \frac{(N^{j+1}-1)}{N-1}(2t_{\text{com}} + c);$$

- $\tau(y, j) = t_{\text{com}} + \tau(y, j+1) + d$, where d includes time to manipulate local variables,

$$\Rightarrow \tau(y, j) = \tau(y, L) + (L - j)(t_{\text{com}} + d);$$

- $\tau(D, j) = t_{\text{com}} + \tau(D, j+1) + Ne$, where e includes time to manipulate local variables,

$$\Rightarrow \tau(D, j) = \tau(D, L) + (L - j)(t_{\text{com}} + Ne).$$

Step 3. Express $C(N,\ L)$ as a function of N, L, and $\tau(m, j), m\epsilon M$:

$$\tau C(N, L) = t\text{com} + (*, L-1) + f,$$

where f includes time to initialize the simulation (before entering the loop of simulation cycles).

Step 4. Replace in $C(N, L)$ each $\tau(m, j)$ by its corresponding function of $\tau(m, 0)$ or $\tau(m, L)$:

$$C(N, L) = t_{\text{com}} + f + N^{L-1}\tau(*, 0) + \sum_{k=1}^{L-1} N^k \tau(Y, L-1-k) + \frac{(N^L - 1)}{N-1}(2t_{\text{com}} + c)$$

$$\Rightarrow C(N, L) = t_{\text{com}} + f + N^{L-1}\tau(*, 0) + \sum_{k=1}^{L-1} N^{kL-1}\tau(Y, 0) + \frac{(N^L - 1)}{N-1}(2t_{\text{com}} + c).$$

Step 5. The width-wise complexity of the algorithm is given by $C(N, L)$ as a function of N when N grows infinitely, and the depth-wise complexity is given by $C(N,L)$ as a function of L when L grows infinitely:

When N grows infinitely (L constant),$C(N, L) \approx (L(L-1)t_{\text{com}} + L^2\alpha + L(\beta - \alpha) + 2t_{\text{com}} + \gamma - \beta + \lambda)N^{L-1}$, and when L grows infinitely (N constant), $C(N, L) \approx (t_{\text{com}} + \alpha)L^2N^L$.

References

1. Zeigler B.,*Theory of Modeling and Simulation*, Wiley, 1976.
2. Zeigler B. P., Kim T. G., Praehofer H., *Theory of Modeling and Simulation*, 2nd edn., Academic Press, San Diego, 2000.
3. Chow A. C., Parallel DEVS: a parallel, hierarchical, modular modelling formalism and its distributed, *Simulation* **13**:55–67, 1996.
4. Chow A. C., Zeigler B. P., Parallel DEVS: A parallel, hierarchical, modular modelling formalism, *Proc. 26th Winter Simulation Conf.*, Lake Buena Vista, IEEE, Piscataway, pp. 716–722, 1994.
5. Chow A. C., Zeigler B. P., Kim D. H., Abstract simulator for the Parallel DEVS Formalism, *Proc. Fifth Annu. Conf. AI, and Planning in High Autonomy Systems*, Gainesville, IEEE, Piscataway, pp. 157–163, 1994.
6. Schwatinski T., Pawletta T., An advanced simulation approach for parallel DEVS with ports, *Proc. 2010 Spring Simulation Multiconf.*, pp. 132–139, 2010.
7. Himmelspach J., Uhmacher A. M., Sequential processing of PDEVS models, *Proc. 3rd EMSS*, pp. 239–244, 2006.

8. Wainer G. A., *Discrete Event Modelling and Simulation: A Practitioner's Approach*, 1st edn., CRC Press, Boca Raton, 2009.
9. Syriani E., Vangheluwe H., Modelling and simulation-based design of a distributed DEVS simulator, *Proc. Winter Simulation Conf.*, pp. 3007–3021, 2011.
10. Bisgambiglia P.-A., Bisgambiglia P., DecPDEVS: New simulation algorithms to improve message handling in PDEVS, *Open J. Model. Simul.* **9**:172–197, 2021.
11. Martin C. R., Trabes G. G., Wainer G. A., A new simulation algorithm for PDEVS models with time advance zero, *Proc. Winter Simulation Conf.*, Orlando, pp. 2208–2220, 2020.
12. Cárdenas R., Henares K., Arroba P., Wainer G., Risco-Martín J. L., A DEVS simulation algorithm based on shared memory for enhancing performance, *Proc. Winter Simulation Conf.*, Orlando, 2020.
13. Franceschini R., Bisgambiglia P. A., Touraille L., Bisgambiglia P., A survey of modelling and simulation software frameworks using Discrete Event System Specification, *Proc. Imperial College Computing Student Workshop*, pp. 40–49, 2014.
14. Van Tendeloo Y., Vangheluwe H., An evaluation of DEVS simulation tools, *Simulation* **93**:103–121, 2016.
15. Knuth D. E., *Selected Papers on Analysis of Algorithms*, CSLI Lecture Notes, Vol. 102, Centre for the Study of Language and Information, Stanford, 2000.
16. Balakirsky S., Kramer T., Comparing algorithms: Rules of thumb and an example, *Proc. 2004 Performance Metrics for Intelligent System (PerMIS) Workshop*, pp. 16–18, 2004.
17. Sun Y., Hu X., Performance measurement of dynamic structure DEVS for large scale cellular space models, *Simulation* **85**:335–351, 2009.
18. Risco-Martin J. L., Fabero J. C., Mittal S., Zapater M., Reconsidering performance of DEVS modelling and simulation environments using the DEVStone benchmark, *Simulation* **93**:459–476, 2017.
19. Glinsky E., Wainer G., DEVSTONE: a benchmarking technique for studying performance of DEVS modelling and simulation environments, *Proc. IEEE Int. Symp. Distributed Simulation and Real-Time Applications*, pp. 265–272, 2005.
20. Wainer G., Glinsky E., Gutierrez A. M., Studying performance of DEVS modelling and simulation environments using the DEVStone benchmark, *Simulation* **87**:555–580, 2011.
21. Gutierrez-Alcaraz M., Wainer G., Experiences with the DEVStone benchmark, *Proc. 2008 Spring Simulation Multiconf.*, pp. 447–455, 2008.
22. Baik D., Zeigler B. P., Performance evaluation of hierarchical distributed simulators, *Proc. 17th Conf. Winter Simulation*, pp. 421–427, 1985.
23. Zeigler B. P., Mapping hierarchical discrete event models to multiprocessor systems: Concepts, algorithm, and simulation, *Parallel Distrib. Comput.* **9**:271–281, 1990.
24. Jefferson D. R., Virtual time, *ACM Trans. Prog. Lang. Syst.* **7**:404–425, 1985.
25. Fujimoto R. M., Parallel and distribution simulation systems, *Proc. 31st Conf. Winter Simulation — A Bridge to the Future*, Vol. 1, pp. 122–131, 1990.
26. Jafer S., Wainer G., Conservative vs. optimistic parallel simulation of DEVS and Cell-DEVS: A comparative study, *Proc. 2010 Summer Computer Simulation Multiconf.*, pp. 342–349, 2010.
27. Capocchi L., Santucci J.-F., Pawletta T., Folkerts H., Zeigler B. P., Discrete-event simulation model generation based on activity metrics, *Simul. Model. Pract. Theory* **103**:102122, 2020.
28. Glinsky E., Wainer G., Performance analysis of DEVS environments, *Proc. AIS Artificial Intelligence, Simulation and Planning*, 2002.

29. Wang Y.-H., Zeigler B. P., Extending the DEVS formalism for massively parallel simulation, *Discrete Event Dyn. Syst.* **3**:193–218, 1993.
30. Van Tendeloo Y., Vangheluwe H., Introduction to parallel DEVS modelling and simulation, *Proc. Model-Driven Approaches for Simulation Engineering Symp.*, pp. 1–12, 2018.
31. Traoré M. K., SimStudio: A next generation modeling and simulation framework, *Proc. 1st Int. Conf. Simulation Tools and Techniques for Communications, Networks, and Systems*, 2008.
32. Van Tendeloo Y., Vangheluwe H., DEVS: Discrete-event modelling and simulation for performance analysis of resource-constrained systems, in *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*, Springer, Cham, pp. 127–153, 2020.
33. Casas P. F., The DEVS formalism, in *Formal Languages for Computer Simulation: Transdisciplinary Models and Applications*, IGI Global, Hershey, pp. 62–102, 2014.
34. Gabriel A. W., Rhys G., Azam K., Introduction to the discrete event system specification formalism and its application for modeling and simulating cyber-physical systems, *Proc. 2018 Winter Simulation Conf.*, Vol. 2, pp. 177–191, 2018.
35. Zeigler B. P., Muzy A., Kofman E., *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*, Academic Press, San Diego, 2018.