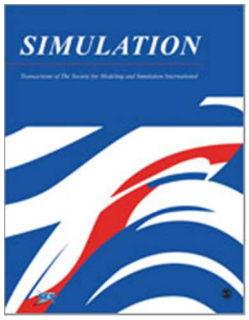
Simulation: Transactions of the Society for Modeling and Simulation International



A Unifying Framework for Specifying DEVS Parallel and Distributed Simulation Architectures

Journal:	*Simulation: Transactions of the Society for Modeling and Simulation International*
Manuscript ID:	S-11-0197.R1
Manuscript Type:	3 Special Issue
Date Submitted by the Author:	09-Aug-2012
Complete List of Authors:	Adegoke, Adedoyin; African University of Science and Technology, Computer Science Togo, Hamidou; Université de Bamako, Mathematiques et Informatiques
Key Words and Phrases:	Theory and Methodology -> DEVS Methodology , Tools and Technology -> Modeling and Simulation Environments , Tools and Technology -> Simulation System Architecture , Tools and Technology -> Parallel and Distributed Computing
Abstract:	DEVS (Discrete Event System Specification) is an approach in the area of Modeling and Simulation (M&S) that provides a means of specifying dynamic systems. A variety of DEVS tools have been implemented without a standard developmental guideline across board consequently revealing a lack of central frameworks for integrating heterogeneous DEVS simulators. When implementing a DEVS Simulator there are salient concepts that are intuitively defined like how events should be processed, what simulation architecture to use, what existing procedures (set of rules/algorithm) can be used, what should be the organizational architecture and so on. From a review of existing implementation approaches, we propose a taxonomy of the identified concepts including some formal definitions as they constitute the essential building blocks of performing PADS by utilizing DEVS. The contribution of this taxonomy and its impact as a unifying framework is that it provides a more systematic understanding of the process of constructing a DEVS simulator. Also, it offers an abstract way for integrating different and heterogeneous DEVS implementation strategies and thus can serve as a contribution to the on-going DEVS standardization efforts.

A Unifying Framework for Specifying DEVS Parallel and Distributed Simulation Architectures

Adedoyin Adegoke [(1)]
aadegoke@aust.edu.ng

Hamidou Togo [(2)]
temena2004@yahoo.fr

Mamadou K. Traoré [(3)]
traore@isima.fr

(1) African University of Science and Technology, Abuja (Nigeria)
(2) Université de Bamako (Mali)
(3) LIMOS, CNRS UMR 6158, Université Blaise Pascal, Clermont-Ferrand 2 (France)

Abstract
DEVS (Discrete Event System Specification) is an approach in the area of Modeling and Simulation (M&S) that provides a means of specifying dynamic systems. A variety of DEVS tools have been implemented without a standard developmental guideline across board consequently revealing a lack of central frameworks for integrating heterogeneous DEVS simulators. When implementing a DEVS Simulator there are salient concepts that are intuitively defined like how events should be processed, what simulation architecture to use, what existing procedures (set of rules/algorithm) can be used, what should be the organizational architecture and so on. From a review of existing implementation approaches, we propose a taxonomy of the identified concepts including some formal definitions as they constitute the essential building blocks of performing PADS by utilizing DEVS. The contribution of this taxonomy and its impact as a unifying framework is that it provides a more systematic understanding of the process of constructing a DEVS simulator. Also, it offers an abstract way for integrating different and heterogeneous DEVS implementation strategies and thus can serve as a contribution to the on-going DEVS standardization efforts.

1. INTRODUCTION

DEVS (Discrete Events Systems Specification) offers a platform for the modeling and simulation of sophisticated systems in a variety of domains. It unifies various formalisms and provides a general description for the construction of a model from an original system and its execution. A DEVS simulator is capable of reproducing behaviors that are identical to that of the system under observation. In doing so, the modeler is provided with some level of abstraction by being able to build models without having knowledge of how the simulator was built.

Due to the growing complexity of systems to be modeled, efficient simulation of such systems cannot be performed on a single physical processor. One way out of this is to make use of distributed strategies by exploiting the computing power of current technologies (grid, cloud, web services etc.). Some benefits of this include; reduction in execution time, improved simulation performance, real time execution and integration of simulators [1]. Parallel and Distributed Simulation (PADS) [1] has been a widely researched area with some potential benefits. First, the use of parallel processors promises an increase in execution speed and a reduction in execution time. Second, the potentially larger amount of available memory on parallel processors will enable the execution of larger simulation models. Third, with the use of multiple processors comes an increased tolerance to a possible processor failure. In addition, it provides a solution to the scientific need to federate existing and naturally dispersed simulation codes. Thus, simulation architecture can be called parallel if its main design goal is to reduce execution time while the term distributed simulation could be referred to as interoperating geographically dispersed simulator [1, 2, 3]. Building a simulation model on a particular world view

significantly reduces implementation complexity [4]. However, DEVS is a specific simulation protocol, which unifies the three classic simulation strategies also known as the world views [2], makes its distribution a challenging issue.

PADS is a matured field of study but its adaptability to some existing modeling and simulation formalisms e.g. DEVS, Petri nets is an arduous task. To our knowledge, DEVS PADS implementation strategies differ from one another due to the absence of a general standard. Also, this heterogeneous factor is a result of not having formal definitions for the intrinsic elements used in developing DEVS simulators. It is necessary that these elements are clarified to easily point out the differences between these strategies in terms of simulation tree structure, concurrency scheme and partitioning strategy. Hence, this work will attempt to identify and capture the elements commonly used in these strategies as well as propose a more generic approach and formal framework which is deemed necessary. As well, a proposal will be made on how to harness the power behind these taxonomies and their bridging. This will help ease the construction of a DEVS simulator and provide a set of minimum requirements for a simulator to be labeled "DEVS-Compliant" as a contribution towards the standardization of DEVS formalism ([30]. Also, it will aid in grasping the concept of DEVS PADS for modeling and simulation by practitioners from different simulation domains.

The rest of the paper is organized as follows: Section 2 presents the foundations of DEVS simulation, i.e. the simulation tree, from which all the distributed strategies are built. Section 3 presents the key concepts in use in this paper. Identified aspects in DEVS PADS as well as classification of research efforts and contributions in this area are presented in Section 4. In Section 5 we present a generic approach and a language useful for building a DEVS PADS implementation. In Sections 6, we give a discussion on the framework and its methodology and then conclude in Section 7.

2. DEVS SIMULATION PROTOCOL

The DEVS formalism [2] provides a comprehensive modeling and simulation framework for modeling and analysis of Discrete Event Systems. It specifies system behavior as well as system structure. System behavior in DEVS is described through its DEVS dynamic functions while system structure is built from the composition of atomic and coupled models. A coupled model is composed of several atomic or coupled models and atomic model is a basic component that cannot be decomposed any further. They are hierarchically organized as shown in Figure 1.

A DEVS model is built according to specification i.e. Classic DEVS or Parallel DEVS. CDEVS (Classic DEVS System Specification) was introduced in 1976 by Zeigler 5] to simulate and execute models sequentially on single processor machine. There is a limitation in the CDEVS that does not allow the proper execution of events that occur concurrently [2]. As a solution, the appropriate execution of these simultaneous events has led to the concept of process and to PDEVS (Parallel DEVS System Specification) [6].

Due to the separation of concerns in DEVS, the modeler needs to focus only on the models being created avoiding the details about the abstract simulator (algorithms). The operational semantics of DEVS models has been defined by abstract algorithms [2]. These algorithms consist of different Nodes (Coordinator, Simulator) organized in a hierarchy that mimics the hierarchical structure of a model. In these algorithms, a DEVS atomic model is executed by assigning a simulator to it and to a DEVS coupled model a coordinator is assigned. From its original definition, the DEVS abstract simulator structure is hierarchical in nature and the hierarchy of models is mapped onto it. The distinctiveness of the DEVS framework is in its hierarchical compositional structures which help in complexity reduction.

During simulation, the interaction/communication between different model components is achieved

through event messages exchanged between the Simulators and Coordinators, each representing an event to be processed:

- (i,t) messages sent from a coordinator to its child to signal initialization
- (*,t) messages sent from a coordinator to its child to signal the occurrence of internal events
- (x,t) messages carry information about external input events from a coordinator to its child
- (y,t) messages transmits a model's output events from a simulator or a coordinator to its parent

"t" represents the time in which a message is sent or received.

Listed above are the basic types of messages used in DEVS. However, other types of messages (e.g. (@, t), (done, t)) can be added as many implementations exist and various extensions to DEVS are defined. The protocol remains the same but actions to be performed by a simulating entity (Simulator/Coordinator) upon the reception of a given message are provided.

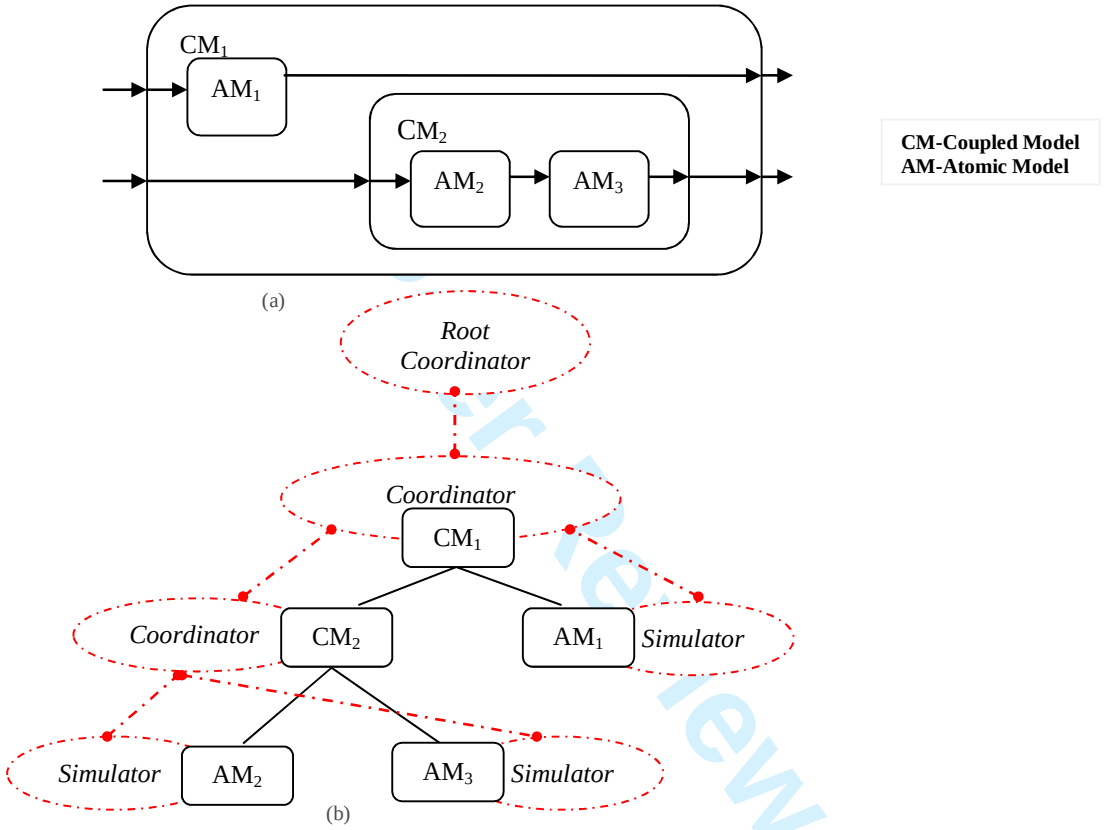


Figure 1: (a) DEVS model (b) Hierarchical mapping of DEVS model to abstract simulator

3. KEY CONCEPTS

In this section we briefly introduce key concepts of the framework. The first 3 concepts are formalized in Definitions 1 and 2 given below. Other concepts will be formalized in subsequent sections.

- Root Coordinator is a special simulating element that drives the global aspects of the simulation on a tree; it initializes and ends the simulation (when a termination condition is detected).
- Nodes: They are simulation entities used for executing DEVS models. These nodes are Coordinators, Simulators and Root Coordinators. The Root Coordinator has an event loop which

sends event messages and controls the simulation cycles while the Coordinator and Simulator are capable of receiving, treating and sending event messages.

- Simulation Tree: A tree is made up of nodes. The Root Coordinator is always at the top of the Tree's hierarchy and has a Coordinator as its descendant. Also, the Coordinator has either a Coordinator or Simulator as its descendant but the Simulator has none.
- Process: We define it as a stream of execution. It contains two types of nodes during execution they are active and passive nodes. An active node is a node that is currently active in an execution stream e.g. Java threads, ADA Tasks. While a passive node is part of an execution stream but not actively involved until it is triggered e.g. function calling in Object Oriented Paradigm. We consider that a process would have at most one active node. If a process has more than one active node, those nodes are then regarded as being autonomous sub-processes. Also, there can be more than one passive node in a process.
- Activity: Set of actions that are performed at the receipt of an event
- Processor is a computing resource that allows the execution of a program (a process, an entire tree, any other executable code) on itself.
- Simulation Graph: A representation of the relationship between the identified aspects in DEVS simulation. An example of Simulation Graph can be seen in Figure 2, details about its components are discussed in the following sections.

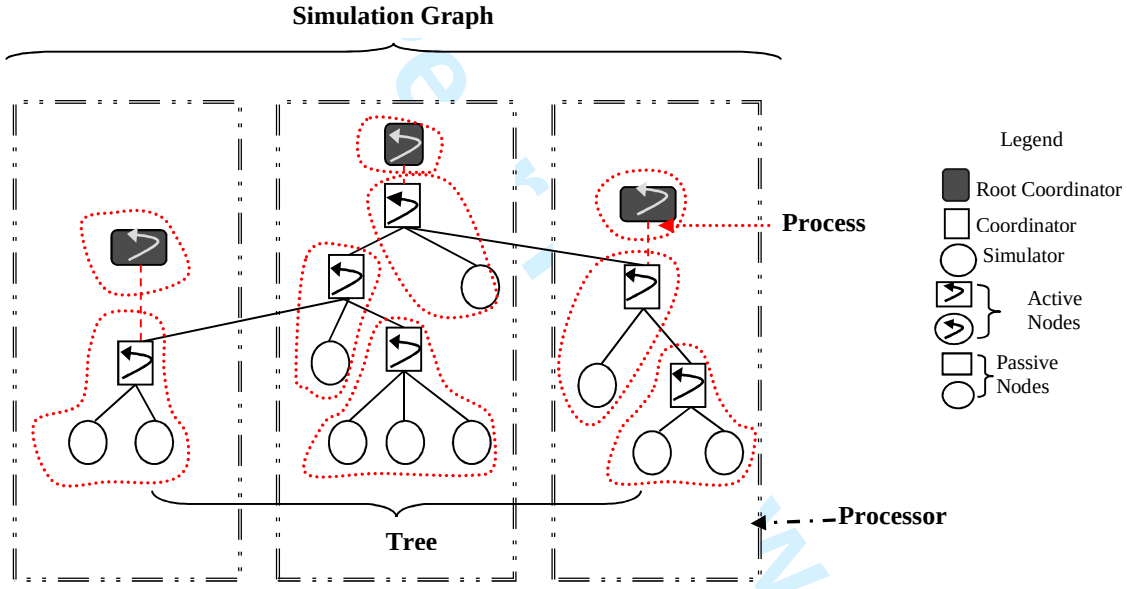


Figure 2: Relationship between Trees, Processes and Processors

Definition 1: We formally define Simulation Tree T as

$$T = < R, N, f >$$

With:

- $R \in N$
- $f: N \rightarrow \wp(N)$ where $\wp(N)$ is Power Set of N
- $f^{-1}(R) = \emptyset$
- $f^{-1}(J) \neq \emptyset, \ \forall J \in N - \{R\}$
- $Cardinal\ f(R) = 1$

Where:

R: The Root Coordinator of the tree

N: The set of nodes of the tree

f: A function that maps a child node to its parent

A node is a "parent" of another node (its child) if it is one step higher in the hierarchy.
For example: Given a Tree T

Tree T will be defined as

$R = A$

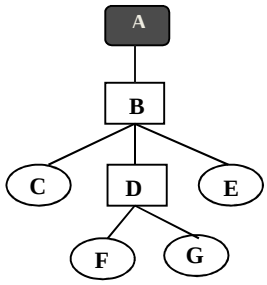
$N = \{A, B, C, D, E, F, G\}$

$f(A) = \{B\}$

$f(B) = \{C, D, E\}$

$f(D) = \{F, G\}$

$f(C) = f(E) = f(F) = f(G) = \emptyset$



Definition 2: Simulation Tree T can also be defined as

$$T = < R, N, F >$$

With:

- $R \in N$
- $F \subset N \times (N - \{R\})$
- $(a, b) \in F \iff b \in f(a)$

Using Definition 2 for the example above, R and N will be defined as same while F will be

$$F = \{(\mathrm{A}, \mathrm{B}), (\mathrm{B}, \mathrm{C}), (\mathrm{B}, \mathrm{D}), (\mathrm{B}, \mathrm{E}), (\mathrm{D}, \mathrm{F}), (\mathrm{D}, \mathrm{G})\}$$

4. TAXONOMY IN DEVS PARALLEL AND DISTRIBUTED SIMULATION

There are different practices behind the concept of exploiting DEVS with Parallel and Distributed Simulation (PADS). Due to this, the concept becomes burdened with variances in opinions on how to build a DEVS simulator. We were able to identify four major factors in use in these practices. First, we realize that some approaches prefer to alter the tree structure; we call this “Tree Transformation” [3, 7, 8, 9, 10, 11, 12, and 13]. Second, we recognize that some approaches consider splitting the model tree, thus based on the number of possible tree structures that can be realized from this splitting we call this “Tree-Splitting” [14, 15, 16, 17]. Also, we recognize as well that some approaches differ on the number of executions/processes that can be performed per simulation run; we call this “Process-Clustering” [19]. Lastly, some approaches prefer to vary the number of computing resources to be used during simulation, we call this “Processor-Mapping” [2, 11]. In the following sections we present these aspects.

4.1. Tree Transformation

It has been observed that altering a simulation tree structure can improve simulation performance and also enhance distribution. This transformation is usually achieved either by reducing or increasing the number of nodes of a tree.

4.1.1. Reduction

As presented in [7] the hierarchical structure of the simulator (which has a one-to-one correspondence with the DEVS model architecture) can increase the communication overhead between Nodes. The process of reducing the number of these nodes on a tree is also known as flattening. A

flattened simulator [8] simplifies the hierarchical simulator while keeping a hierarchical model structure (see Figure 3). Though various studies [9], [10] have shown that a flattened simulator reduces these costs however, maintaining a hierarchical structure eases validation and verification during simulation. Some other approaches prefer to alter the compositional structure of a DEVS model. Kim [11] proposed transforming a hierarchical DEVS model into a non-hierarchical structure to ease synchronization in a distributed simulation and Zeigler [2] also considered building Conservative DEVS simulator for non-hierarchical models.

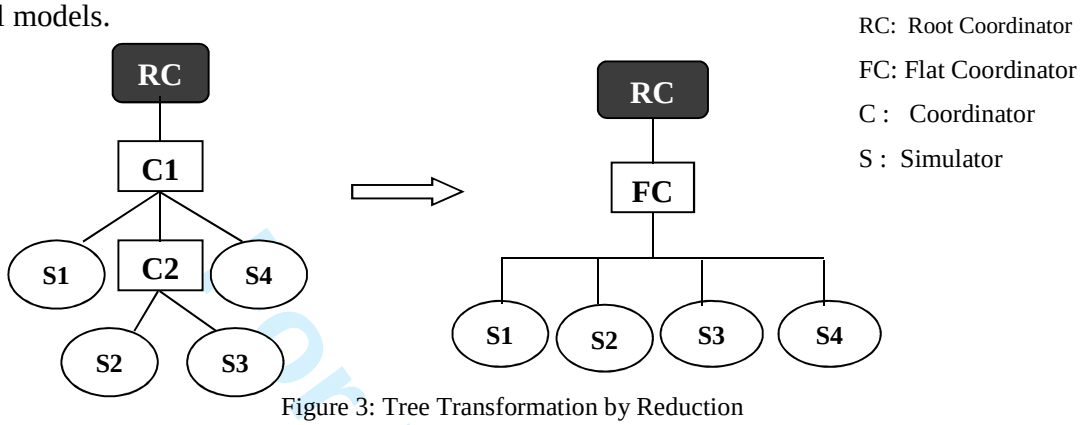


Figure 3: Tree Transformation by Reduction

4.1.2. Expansion

In CD++ ([3], [12]) the expansion was achieved by introducing new simulation nodes into the simulation tree structure as presented in Figure 4. This is to enable the distribution of Nodes on different processors. The introduction of extra components on the tree introduces more concerns as to what type of information each of these new components should contain. Also, communication between these Nodes constitutes an increasing overhead cost as the structure of the messages being passed is altered to accommodate extra information. For example a new sub-coordinator has no coupled model associated with it and therefore contains no coupling information (EIC, EOC, IC), also, it has to correctly identify imminent models and influencees. One way to deal with this is through the composition of messages i.e. by including more information in a message's construct, as seen in [3] and [13].

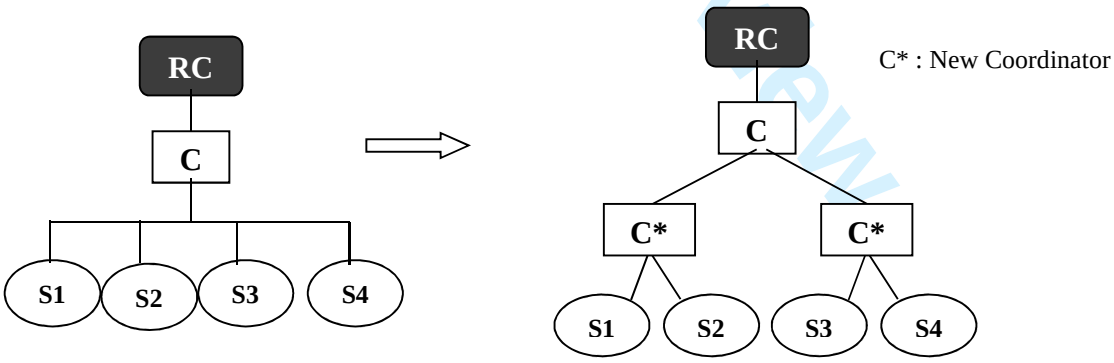


Figure 4: Tree Transformation by Expansion

Formally we define Tree Transformation as:

Definition 3: $Transf[Na, Nr, Fa, Fr]: \tau \rightarrow \tau$

$$Transf[Na, Nr, Fa, Fr](< R, N, F >) =< R, N', F' >$$

With

- $N' = N \cup Nr - Na$
- $F' = F \cup Fr - Fa$

Where

- Na is the set of nodes to be added to N
- Nr is the set of nodes to be removed from N
- Fa is the set of relationships to be added to F
- Fr is the set of relationships to be removed from F

Based on the following conditions:

- $Na \cap N = \emptyset$
- $Nr \subset N - \{R\}$
- $Fa \subset (N \times Na) \cup Na^2 \cup (Na \times N - \{R\})$
- $Fr \subseteq F$

4.2. Tree-Splitting

Tree splitting can be referred to as the decomposition of a simulator tree to form sub-trees based on the analysis of the model's structure. We identified two types namely, single tree structure and multiple tree structure. It is necessary to state here that this section does not deal with how the tree structure can be split, executed or how they can be mapped to available number of processors.

4.2.1. Single Tree Structure

In describing this structure, executing a model with a single tree structure (as shown in Figure 5) can be expressed as having an entire model tree simulated with the use of a central scheduler called the Root Coordinator.

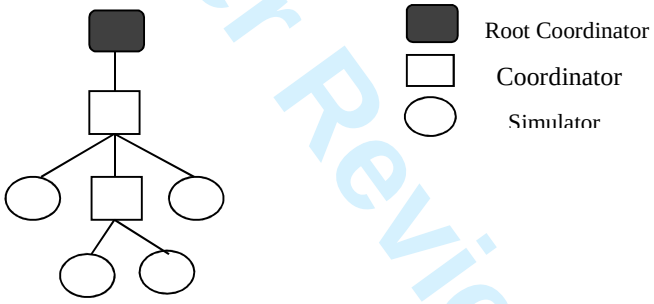


Figure 5: Single Tree Structure

Single tree structures are mostly implemented using Classic DEVS (CDEVS) and Parallel DEVS (PDEVS) algorithms. In CDEVS [2] events are processed in a sequence. This approach is the simplest form of simulation but it is rigid and does not properly reflect the simultaneous occurrence of events in the system being modeled. Also, serialization reduces possible utilization of parallelism during the occurrence of events. On the other hand, Chow and Zeigler [6] introduced PDEVS as a possible solution to the problem of serialization. According to Chow, one desirable property provided by PDEVS is the degree of parallelism which can be exploited in parallel and distributed simulation. It beats the restrictions in CDEVS in both execution time and memory usage.

4.2.2. Multiple Trees

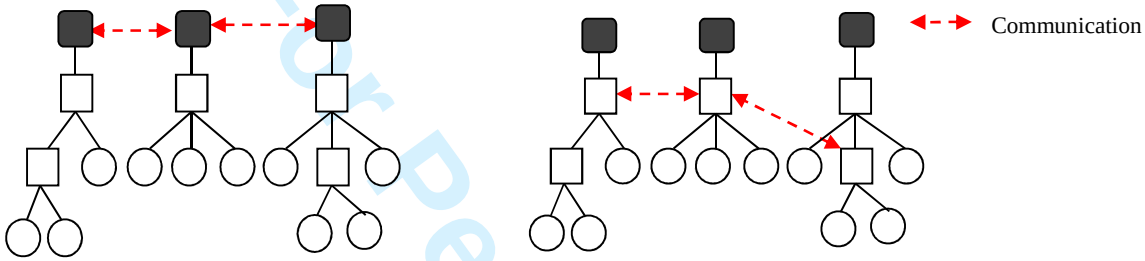
We look at the multiple tree structure as when a tree can be split into different sub-trees with each having its own central scheduler/Root Coordinator and different simulation clocks. This is the preferable solution in distributed simulation. Based on this structure, all events with the same timestamp are

scheduled to be processed simultaneously. It uses asynchronous algorithms which rely on the distributed synchronization protocols for synchronization instead of the Root Coordinator.

The two basic asynchronous algorithms in use are the Optimistic and Conservative (Pessimistic) DEVS algorithms. In distributed simulation, there is an inherent constraint in the time-stamp order (not in their real-time arrival order) with which events occur and are processed, this is called locality constraint. Some algorithms were proposed to resolve this constraint, the Conservative ([14], [15], [16]) algorithm which always prevent this constraint through the use of look-ahead mechanism and Optimistic [17] algorithm which detect and resolves it through the use of the roll-back mechanism. Optimistic algorithms, in contrast to Conservative algorithms, enable increased degrees of parallelism and do not depend on application-specific data to decide on events that are safe for processing.

Communication between these trees is usually between

a. Root Coordinator and Root Coordinator (see Figure 6a) e.g. DEVS Time Warp [2]
b. Coordinators and Simulators (Parents and their Children) e.g. DOHS scheme [6] (see Figure 6b)



a) Communication between Root b) Communication between Coordinators

Figure 6: Communication between tree structures

Formally we define Tree Splitting as:

Definition 4: Split: $\tau \rightarrow \Sigma$

With

- $T = < R, N, f >$
- $Split(T) = < \{R_i\}, N', f' >$

Based on the following conditions:

- $R \in \cup\{R_i\}$
- $N' = N \cup \{R_i\}$
- $f'_{/N} = f$

Where τ is the set of all possible trees and Σ is the set of all possible simulation skeletons. The formal definition for a simulation skeleton has been given in section 5.

4.3. Process Clustering

Events execution is driven through the use of processes. We take a look at the concept of process as an execution stream. A process can be seen as a mechanism that is able to execute events. We categorize based on the number of processes; as “one process” execution and “many processes”. However, we will not be dealing with how execution takes place on processors.

4.3.1. One Process

A one-process execution denotes having events processed in a serially and orderly manner i.e. one

after another. This restricts concurrent execution streams. Himmelspach [18] denotes this form of execution as "sequentialization". In this sense, for example, the "main" program is a process. Though a desired speed up may not be achieved when using a one-process execution stream, on the other hand it is easier and faster to implement. During the one-process execution, interaction between the Nodes is called intra-process communication. Most implementations based on CDEVS make use of one-process type of execution stream.

4.3.2. Multiple Processes

In the case of many processes, execution of events can be split into several logical processes (tasks) for concurrent processing. These logical processes, which are autonomous examples, include Java threads, POSIX threads, Ada tasks and so on.

Using many processes could speed-up execution as each could execute events without interrupting other processes. However this is balanced by the increase of memory consumption and the burden of communication between processors. This type of communication is called inter-process communication. It is possible that during a simulation run only one process, out of many, is scheduled for execution. This situation is called pseudo-parallelism otherwise it is pure-parallelism. During implementation it is essential to manage how processes access resources that are common to all of them e.g. shared data type. Locks, Semaphores, Monitors and other synchronizing mechanisms can be used to coordinate these processes. The CCD++ [19] implementation utilizes many processes for model execution.

Formally we define Process Clustering as:

Definition 5: Cluster: $N \rightarrow Ps$
Where

- Ps is the set of Processes.

Based on the conditions that

- $Cluster^{-1}(p)$ is Connex $\forall p \in Ps$
- $\forall\, p_i, p_j \in Ps,\ p_i \neq p_j,\ Cluster(p_i) \cap Cluster(p_j) = \emptyset,$

4.4. Processor Mapping

We considered that the number of processors play a major role in speed, performance and efficiency that can be achieved during simulation. We therefore categorize this into 2 distinct classes; "one-processor" or "many-processors".

4.4.1. One Processor

On a uniprocessor system, the entire simulation runs on one processor so there is no overhead cost but it is limited to the size of the memory in use. Thus, it is not completely suitable for executing complex models. The type of communication that takes place in this case is called an intra-processor communication.

4.4.2. Multiple Processors

In order to coordinate simulation on many networked processors, some form of inter-processor communications is required to convey data between processors and synchronize each processor's activities. When utilizing multiple processors for simulation, the memory architecture type could either be shared memory (processors have direct access to common physical memory), or distributed memory. Meanwhile, in shared memory only one processor can access the shared memory hereby introducing the need to control access to the memory through synchronization. Distributed memory refers to the fact that

the memory is physically distributed as well. Memory access in shared memory is faster but it is limited to the size of the memory therefore, increasing the number of processors without increasing memory size can cause severe bottlenecks. Inter-processor communications is usually achieved through interoperability mechanisms (e.g CORBA [31]).

As a consequence of using more than one processor, the Nodes can be split into a set of partition blocks based on certain decision criteria and mapped unto the available number of processors, this is called partitioning. In the case of no partition, simulation is performed on a single processor machine. The partitioning problem is one of the most important issues in parallel and distributed simulation as it directly affects the performance of the simulation. Different partitioning algorithms have been proposed an example is Generic Model Partitioning (GMP) algorithm proposed by Park [20]. It uses cost analysis methodology to construct partition blocks, although it makes an effort to guarantee an incremental quality of partitioning but is restricted only to models from which cost analysis can be extracted and processed.

Formally we define Processor Mapping as:

Definition 6: Map: $Ps \rightarrow Pr$
Where

- Ps is the set of Processes
- Pr is the set of Processors

Based on the following condition
$\forall\, p_i, p_j \in Ps,\ p_i \neq p_j,\ Map(p_i) \cap Map(p_j) = \emptyset$

4.5. Simulation Graph Strategies

Due to the increasing number of complex model systems various studies have been conducted to improve efficiencies and performances of DEVS simulators [2, 9, 10, 12, 13, 24, 25] thus giving rise to various graph strategies. In a general overview, most implementation decisions have been observed to be based on the presented aspects in the previous sections. In this section we use figures to illustrate how these aspects are interrelated with one another using a three-categorized view. Since the Tree Transformation and Tree Splitting aspects both focus on the Tree, they will be represented as the same category i.e. the Tree. In each of these categories, the number of elements i.e. Trees, Processes or Processors is put into consideration. This therefore forms the basis for development of any Simulation Graph Strategy presented.

4.5.1. Single Processor – Single Tree – Single Process:

This is the simplest form of mapping strategy which has one root coordinator (one tree) controlling the simulation on one processor (see Figure 7). Execution of events is purely sequential with no need to synchronize communication between the Nodes. In this case, when simultaneous events occur, one event is selected and others are ignored thereby introducing rigidity during execution. PythonDEVS [21] uses this mapping strategy as an implementation of the CDEVS formalism and as a consequence it performs sequential simulation.

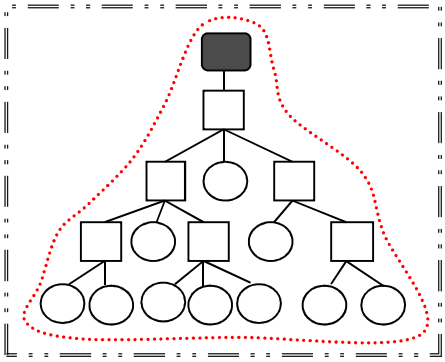


Figure 7: One partition, one tree and one process

4.5.2. Single Processor – Single Tree – Multiple Processes

The entire simulation depends on one Root Coordinator while execution is through the use of many processes as shown in Figure 8. These processes run concurrently and are mostly used to increase execution speed but as the number of processes increase the rate of memory consumption increases thereby slowing down execution and time.

This strategy was proposed for use in Abstract Threaded Simulator [18]. However, depending on the memory size of the processor and the model size, the cost of creating threads gets expensive as the number of models increases. This is a critical factor to be considered when using many processes.

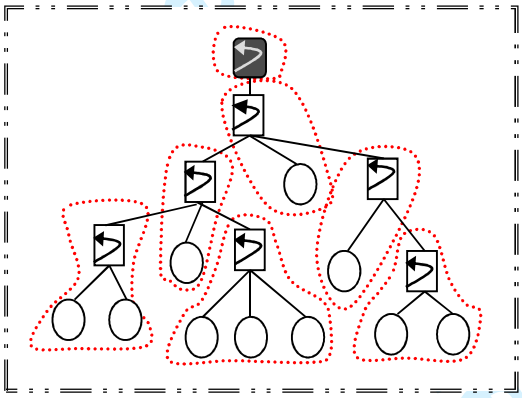


Figure 8: One processor, one tree and many processes

4.5.3. Single Processor – Multiple Trees – Single Process

Several trees (root coordinators) exist on one processor with each performing sequential execution one at a time. An example is shown in Figure 9. This scenario is not realistic because execution is asynchronous and can be done simultaneously using many processes instead.

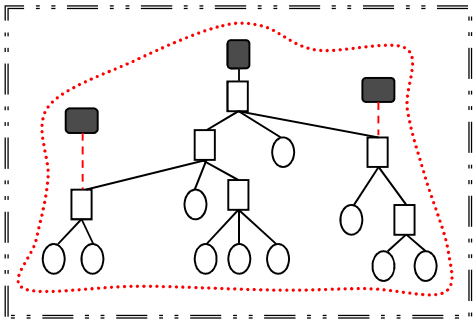


Figure 9: One processor, many trees and one process

4.5.4. Single Processor – Multiple Trees – Multiple Processes

Several root coordinators (trees) exist on a partition (as seen in Figure 10). This makes it easy to implement a synchronization mechanism (optimistic or pessimistic) for dealing with causality errors. Causality errors usually occur when messages are not processed in a time-stamp order [1]. Communication between different trees can be made via the root coordinators or coordinators and simulators. This strategy brings about the idea of federating existing abstract simulators. Since all the trees are on one processor there is intra-processor communication thereby eliminating the need for an interoperability technology.

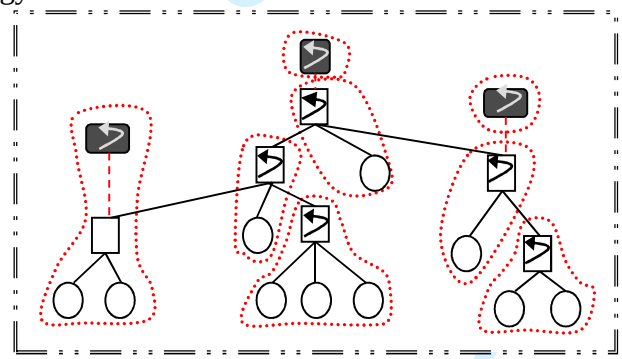


Figure 10: One processor, many trees and many processes

4.5.5. Multiple Processors – Single Tree – Single Process

A root coordinator controls the entire simulation on multiple partitions while execution is sequential. The sequential execution can only take place locally i.e. when a simulating node receives a message from its parent on the same processor. If a simulating node sends a message to a node on another processor (remotely) in this case the number of processes used for executing the tree would increase. An example of this is shown in Figure 11.

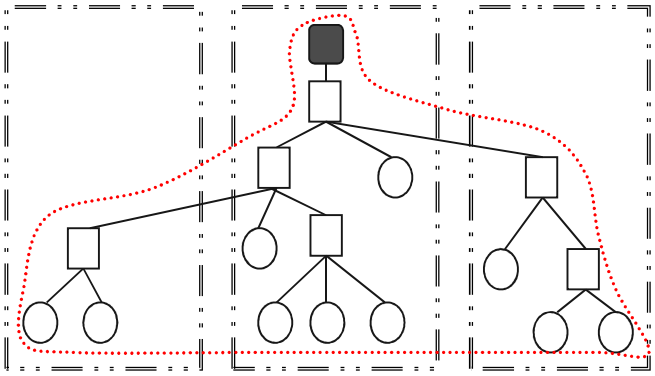


Figure 11: Many processors, one tree and one process

4.5.6. Multiple Processors – Single Tree – Multiple Processes

One Root Coordinator controls the entire simulation on multiple partitions but with multiple processes. The communication mechanism between the Nodes changes since some Coordinators no longer on the same partition as some of their children. There are two types of communication between these nodes i.e. locally (intra-processor) and remotely (inter-processor). At the local level, communication is between Nodes on the same processor. Remote communication is achieved through the use of interoperability technologies (e.g. CORBA [31] etc.) to help overcome the limitation of memory resources otherwise there will be no gain in execution time. This is the case of PDEVS with thread proposed by [13]. Also, as seen in [22] when running parallel and distributed simulations, the entire simulation tree is divided among a set of processes, each of which will execute on a different CPU. In general terms, each process will host one or more simulation nodes as shown in Figure 12.

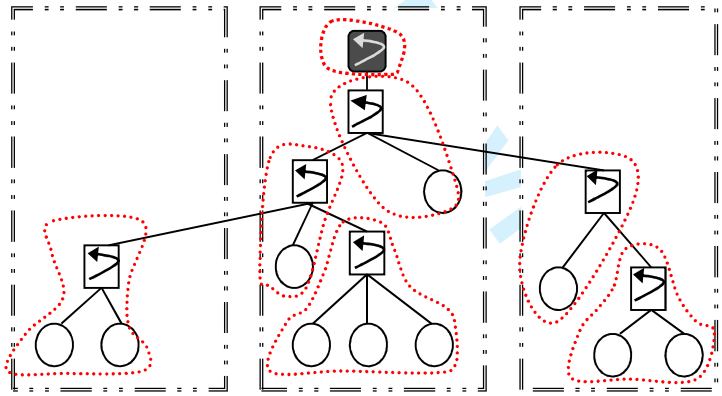


Figure 12: Many processors, one tree and many processes

In [13], the authors proposed an algorithm which is a variant of PDEVS algorithm for implementation on multi-processor using many processes for execution. It allows that sequential and/or parallel execution can be performed among the nodes on each processor. However, extra components were created. It is therefore limited to the issues described previously.

4.5.7. Multiple Processors – Multiple Trees – Single Process

This case, which concerns multiple partitions (each containing at least a tree running independently, as shown in Figure 13), is also not feasible for the same reasons mentioned at "**Single Processor – Multiple Trees – Single Process** ".

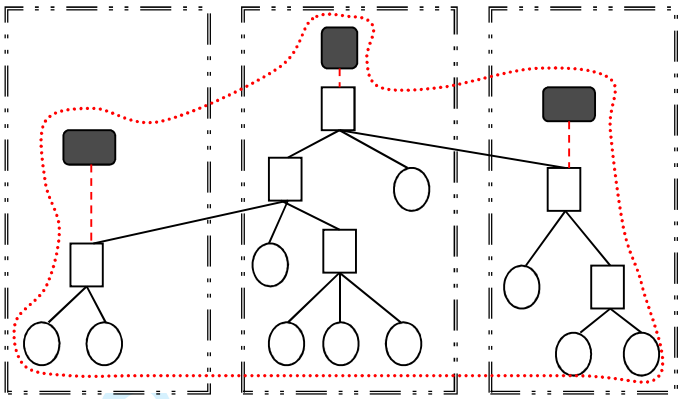


Figure 13: Many processors, many trees and one process

4.5.8. Multiple Processors – Multiple Trees – Multiple Processes

As shown in Figure 14, each partition contains at least one tree and several executions at the same time. Each tree implements a synchronization mechanism for causal errors (because each tree has its own clock). Communications between partitions are either between the root coordinators (as in [2]) or between the coordinators and simulators (between ascendants and descendants as in [11]) via distributed simulation middleware such as HLA [32, 33]

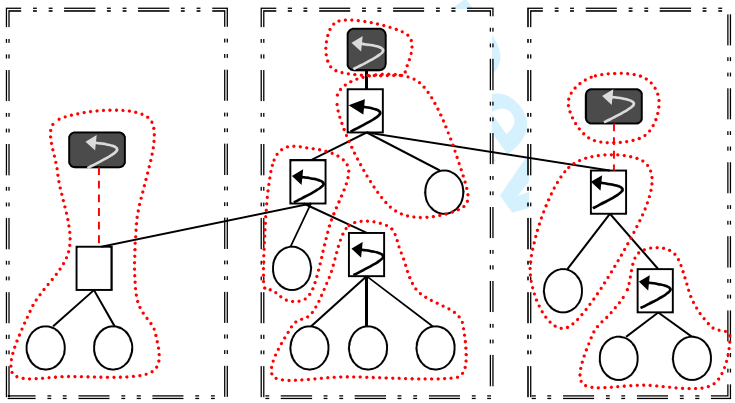


Figure 14: Many processors, many trees and many processes

Optimistic and Conservative strategies are synchronization techniques used for PADS in general. The conservative approach is the first synchronization algorithm that was proposed in the late 1970s by Bryant [14], Chandy and Misra [15]. It is also known as the Chandy-Misra-Bryant (CMB) algorithm and strictly avoids the possibility of processing events out of time stamp order. In contrast the optimistic approaches, introduced by Jefferson's Time Warp (TW) protocol [17], allow causality errors to happen temporarily but provide mechanisms to recover from them during execution. The first attempt to combine DEVS and Time Warp mechanism for optimistic distributed simulation is DEVS-Ada/TW [23]. It is an asynchronous approach that uses the Time Warp mechanism for global synchronization; it treats all Nodes on one processor as one process. In DEVS-Ada/TW, the hierarchical DEVS model can

be partitioned at the highest level of the hierarchy for distributed simulation. As a consequence, the flexibility of partitioning models is restricted.

The DOHS (Distributed Optimistic Hierarchical Simulation) scheme [7] is a method of distributed simulation for hierarchical and modular DEVS models that uses the Time Warp mechanism for global synchronization. In DOHS scheme there is at least one tree per processor (see Figure 15).

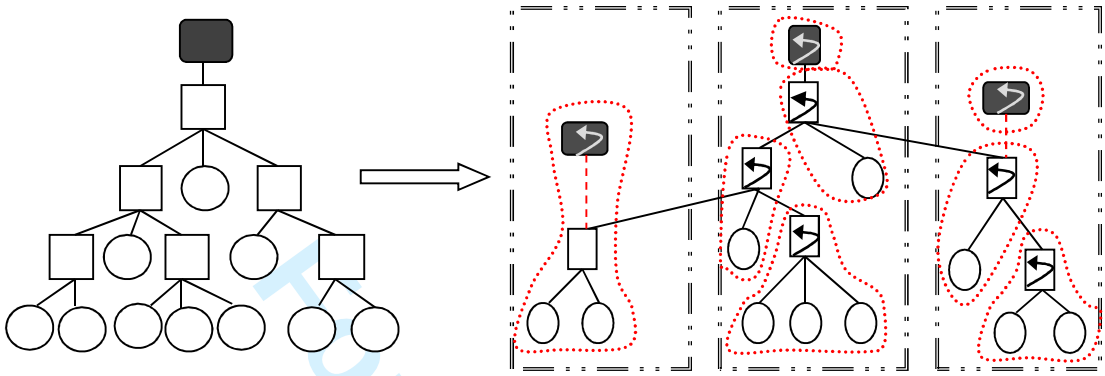


Figure 15: Example Structure of DOHS Scheme

A proposal was made by Zeigler [2] to combine Time Warp with the CDEVS hierarchical simulator as Time Warp DEVS Simulator. In this approach, the overall model is distributed so that each sub-model is a single coupled model. Then, hierarchical execution is done locally on each processor using the classic abstract simulator with extensions for state saving and restores (rollback). In addition, each processor has a root coordinator which realizes the mechanism for Time Warp. On each processor, the root coordinator performs optimistic Time Warp synchronization. For this, it stores the input and output messages of the processor and takes care of anti-messages.

The "Risk-Free" Optimistic Simulator [2] is another version of optimistic DEVS simulator. The operational semantics of the "risk-free" version of optimistic DEVS is based on optimistic DEVS Time Warp. The Time Warp optimistic simulator and coordinator are used without change but the synchronization mechanism in the optimistic root coordinator changes. Local events on each compute node (tree) are processed sequentially but optimistically. That is to say if a straggler event is received by a node, a rollback occurs but is local to that node. This is less costly when compared to Time Warp.

In parallel versions of CD++ (PCD++ [24], [25] & CCD++ ([19], [29]), the authors proposed a distributed simulation architecture for DEVS and Cell-DEVS models. This variant uses the flattening structure of simulators with four simulation nodes on each tree: Simulators, Flat Coordinator (FC), Node Coordinator (NC) and Root Coordinator (RC). Root Coordinator is created on one of the processors to start/end the simulation process and perform I/O operations. FC and NC are created on each processor. FC is in charge of intra-processor communication between its children Simulators. NC is the local central controller on each processor (Figure 16). Simulator executes the DEVS functions defined in its atomic model.

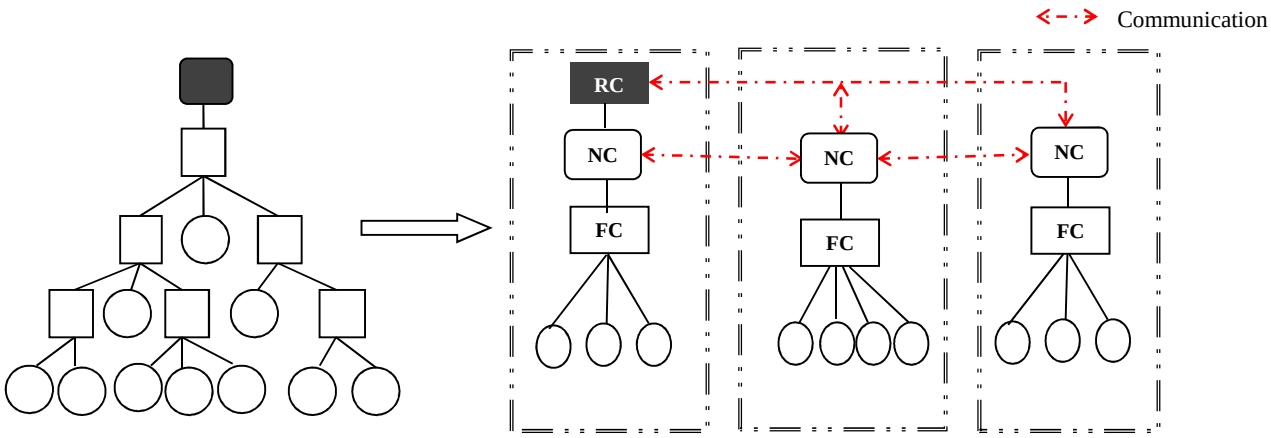


Figure 16 - Structure of Distributed DEVS simulation in CCD++ and PCD++

To briefly explain some of the strategies in the literature in a more formal way we suggest the Tree-Process-Processor notation. It consists of defining the number of elements for each aspect of PADS. We use N for "many elements". For example, a 1-1-1 scheme is a DEVS Simulation Graph strategy with 1 Tree, 1 Process and 1 Processor while N-N-1 is a DEVS tool with many Trees, many Processes and 1 Processor. PythonDEVS [21] is a 1-1-1 strategy. It uses the CDEVS formalism in specifying models and as a consequence it performs sequential simulation. The Abstract Threaded Simulator of the James II [18]) package uses a 1-N-1 strategy with its processes created using Java threads. In [22], the Parallel CD++ Simulator and the Parallel Sequential Simulator by Himmelspach, Ewald, Leye and Uhrmacher [13], which implements the PDEVS formalism, use the 1-N-N strategy. The Conservative CD++ [29] is an N-N-N strategy. Some other approaches that use the N-N-N strategy include DEVS-Ada/TW [23], DOHS scheme by Kim, Seong, Kim and Park [7] and Optimistic Parallel CD++ [24]. A DEVS tool that uses the N-N-N strategy fully supports distributed simulation. Also, we observed that having an implementation which involves the use of N-1-1 strategy or N-1-N strategy is not realistic. A reason for this is that the execution of each tree is asynchronous and can be done simultaneously using many processes instead. This information is represented in Table 1.

Table 1: State of the art and their Simulation Graph Strategy

Approaches	Algorithm	Strategy
PythonDEVS (Bolduc and Vangheluwe 2002)	CDEVS	1-1-1
Abstract Threaded Simulator (Himmelspach, J. et al. 2006)	PDEVS	1-N-1
Parallel Sequential Simulator (Himmelspach, J. et al. 2006)	PDEVS	1-N-N
Parallel CD++ (Troccoli and Wainer 2003)	PDEVS	1-N-N
Optimistic Parallel CD++ (Qi and Wainer 2007)	Optimistic DEVS	N-N-N

CCD++ (Jafer and Wainer 2010[a])	Conservative DEVS	N-N-N
Risk-Free Optimistic DEVS Simulator (Zeigler et al. 2000)	Optimistic DEVS	N-N-N
Time Warp DEVS Simulator (Zeigler et al. 2000)	Optimistic DEVS	N-N-N
DEVS-Ada/TW (Christensen 1990)	Optimistic DEVS	N-N-N
DOHS (Kim et al. 1996)	Optimistic DEVS	N-N-N

We identify that a Simulation Graph strategy for an implementation differs from another however we state here that our definitions of the components used in Simulation Graph construction are at an abstract level. The Links and Nodes in a Simulation Graph strategy are seen as abstract hence they can be implemented in different ways either by using the language in which it was implemented or by using interoperability technologies (e.g. CORBA [31]) in the some cases where simulators are implemented in different languages. The only constraint we define for this is that the simulators implement DEVS simulation algorithm. Thus the issue of DEVS implementation differences can be overcome.

5. UNIFYING DEVS PADS SIMULATION FRAMEWORK

A unifying framework is needed to harness the identified components and their operations (found in the Simulation Graph) in a bid to automate the process of building a DEVS simulator and performing simulation by using DEVS with PADS. We interpret the building of a DEVS Parallel and Distributed Simulation (PADS) Simulator as a move from the original Simulation Tree (ST) to a Simulation Graph (SG). Having taken a look at the possible Simulation Graph strategies, in this section we propose a methodology for building a Simulation Graph. A SG is obtained through the depictions of the relationship and the components of DEVS PADS i.e. Trees, Processes and Processors. This methodology thus describes a structural and behavioral view in exploiting DEVS PADS.

5.1. From Simulation Tree to Simulation Graph

We propose using a layered approach to present the simulation structures involved in DEVS PADS to avoid the pitfalls inherent to the building of a DEVS simulation system suitable for Parallel and Distributed execution. Also, by using a state chart, we present the trajectories which describe the set of all possible paths that can be taken during the construction of the Simulation Graph (SG).

The SG construction is driven based on the analysis of the initial Simulation Tree, the available number of Processes and Processors. An overview of this methodological approach is given by the state chart given in Figure 17. It is worth noting that the methodology iterates on each state until some user-defined satisfaction criteria are reached (optimal splitting, optimal clustering, optimal mapping and optimal transformation).

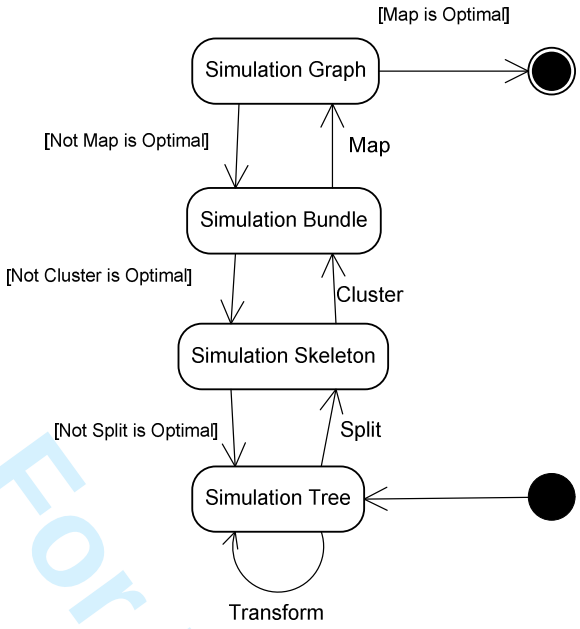


Figure 17: Simulation Graph Methodology

Split is a function that is used for creating a partition of nodes from a *Simulation Tree*. The *Cluster* function takes the available number of nodes and associates them with *Processes*. While the *Map* function takes the set of available *Processes* and plots them onto the set of available *Processors*. Also, the *Transform* function alters the *Simulation Tree* structure either by expansion or reduction. This altering is done on the number of available nodes (not including the Root Coordinators) on the Tree and their relationships. The process of *Transformation*, *Splitting*, *Clustering* and *Mapping* continues until it is sure that a good performance or speed will be gained during simulation from the new *Simulation Graph*. The *Simulation Skeleton* is the structure of the simulation protocol that can fit the PADS scheme. The *Simulation Bun*dle is a collection of cluster of nodes. Examples are shown in Figures 18 and 19 respectively.

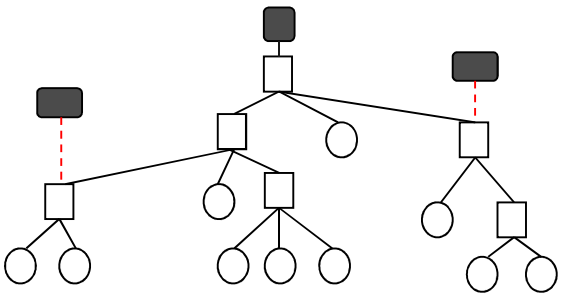


Figure 18: Simulation Skeleton

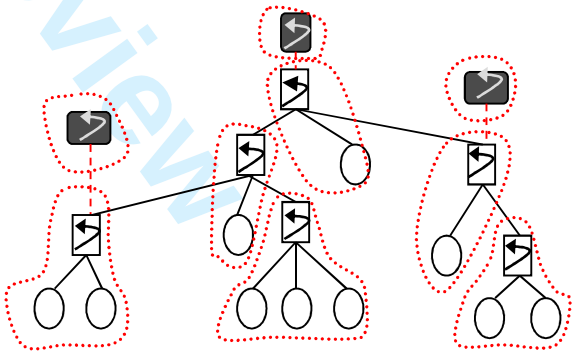


Figure 19: Simulation Bundle

We formally define each of the Simulation Structures that are found in this methodology (the definitions for the Simulation Tree structure are found in Definitions 1 and 2). To have a complete definition we make reference to Clustering and Mapping functions which have been formally defined in Definitions 5 and 6 respectively while definitions for the Transform and Split functions are given in Definitions 3 and 4.

Definition 7: A Simulation Skeleton is defined by

$$S = < \{R_i\}, N, f >$$

With

- $R_i \in N \ \forall$ i
- f: $N \rightarrow \wp(N)$ where $\wp(N)$ is Power Set of N
- $f^{-1}(R_i) = \emptyset \ \forall i$
- $f^{-1}(J) \neq \emptyset, \ \forall J \in N - \cup\{R_i\}$

Definition 8: A Simulation Bundle is formally defined as

$$B = < \{R_i\}, N, f, Ps, Cluster >$$

With

- $< \{R_i\}, N, f >$ as a skeleton
- Ps is the set of Processes
- $Cluster$: $N \rightarrow Ps$

Definition 9: A Simulation Graph is defined by

$$G = < \{R_i\}, N, f, Ps, Pr, Cluster, Map >$$

With

- $< \{R_i\}, N, f, Ps, Cluster >$ is a Simulation Bundle
- Pr is a set of Processors
- Map: $Ps \rightarrow Pr$

5.2. Common Language for the Simulation Structures

We are developing a supporting tool called SimStudio [26] in which the framework has to be applied. To achieve this goal we define an XML (eXtensible Markup Language) representation of the Simulation Structures formally presented in previous sections. We have chosen XML because it is a widely used standard that allows the description of any kind of data and thus can be used for data storage. This language (XML and its platform independent solutions) will be used to specify, construct, describe and preserve information about each Simulation Structure. Figure 20 shows the building blocks of this XML-based language. Each of this description conforms to our proposed methodology and they also take into account multiplicities of the elements that form each Structure. These multiplicities place a constraint on the allowed number of elements in each Structure. The Tree consists of a description about Nodes (Root, Coordinator, and Simulator) and their associated DEVS model. In the specification for Skeleton, it describes the new Root Node(s) that are added to the structure. Bundle consists of Nodes which are clustered into Processes while the VM (Virtual Machine) description contains information about the Processors to be used for simulation. Graph's description uses elements from Bundle and VM i.e. it consists of Processes which are mapped on to Machines.

Such a representation opens the door to the concretization of the abstract process described by Figure 17. As an illustration, Figure 25 shows how key concepts of the framework are assembled and concretely represented from the original tree obtained with Figure 21 to the final Simulation Graph.

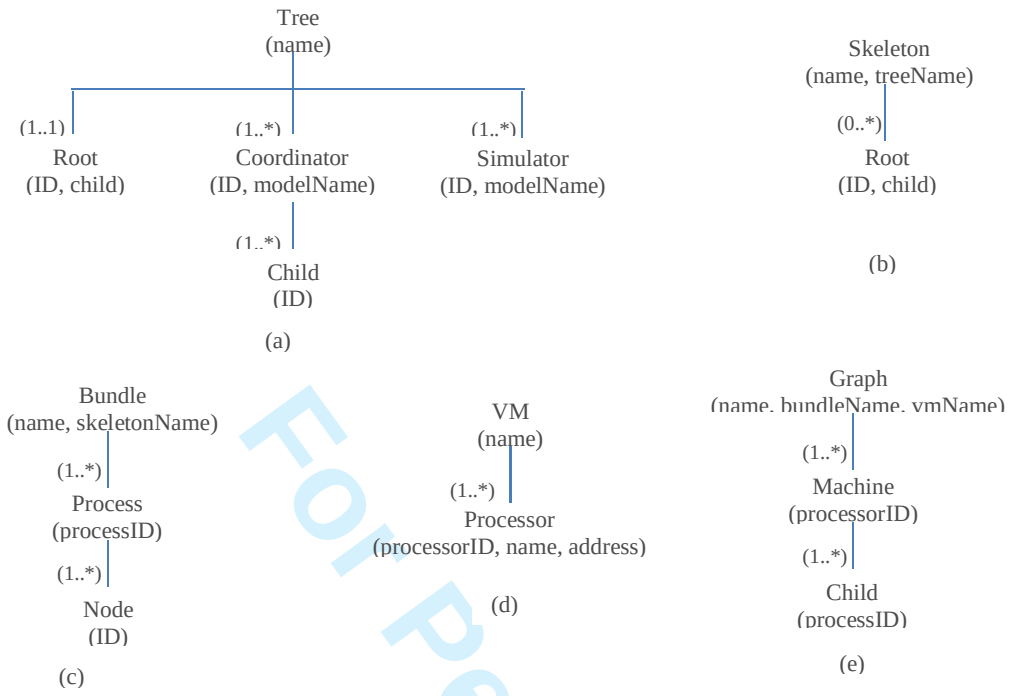


Figure 20: Description for (a) Tree (b) Skeleton (c) Bundle (d) VM (e) Graph

We take a scenario of a university system which offers Masters and PhD degree programs as a DEVS model. Each program's model consists of 2 sub-models. Each student enrolled in the Masters' program is expected to complete and submit a project before moving on to the PhD program. While in the PhD program, the student is first enrolled as a candidate and then he is to write a candidacy exam towards the end of his first year. If he succeeds his status is then upgraded to being a PhD student. Figure 21 is a screenshot of the Eclipse-DDML Modeling tool [28] (part of the SimStudio M&S Framework) which shows the model of the university system. We do not present the entire process from the scenario to the final code. However we show how the Simulation Graph and its language can be constructed from this scenario. Figure 22 draws from this model to illustrate Simulation Graph construction. Nodes C'A, C'B, C'C represents the University, Masters and PhD models of Figure 21 respectively while S'D, S'E, S'F, S'G represents the students, project, candidate and student sub-models respectively.

Starting with this model (Figure 21), a *Simulation Tree* which mimics the hierarchical structure of this model can be obtained. This is done by assigning an Atomic Model to a Simulator and to a Coupled Model a Coordinator is assigned. The Coordinator and Simulator are *Nodes* of a *Simulation Tree*. Also a Root Coordinator which controls the entire simulation process is added to the top of the *Tree*'s hierarchy. The generated *Simulation Tree* structure (Figure 22(a)) is contained in an XML file. The outcome of operations performed is a different Simulation Structure except at the Transform operation stage which produces a modified representation of the *Simulation Tree* structure.

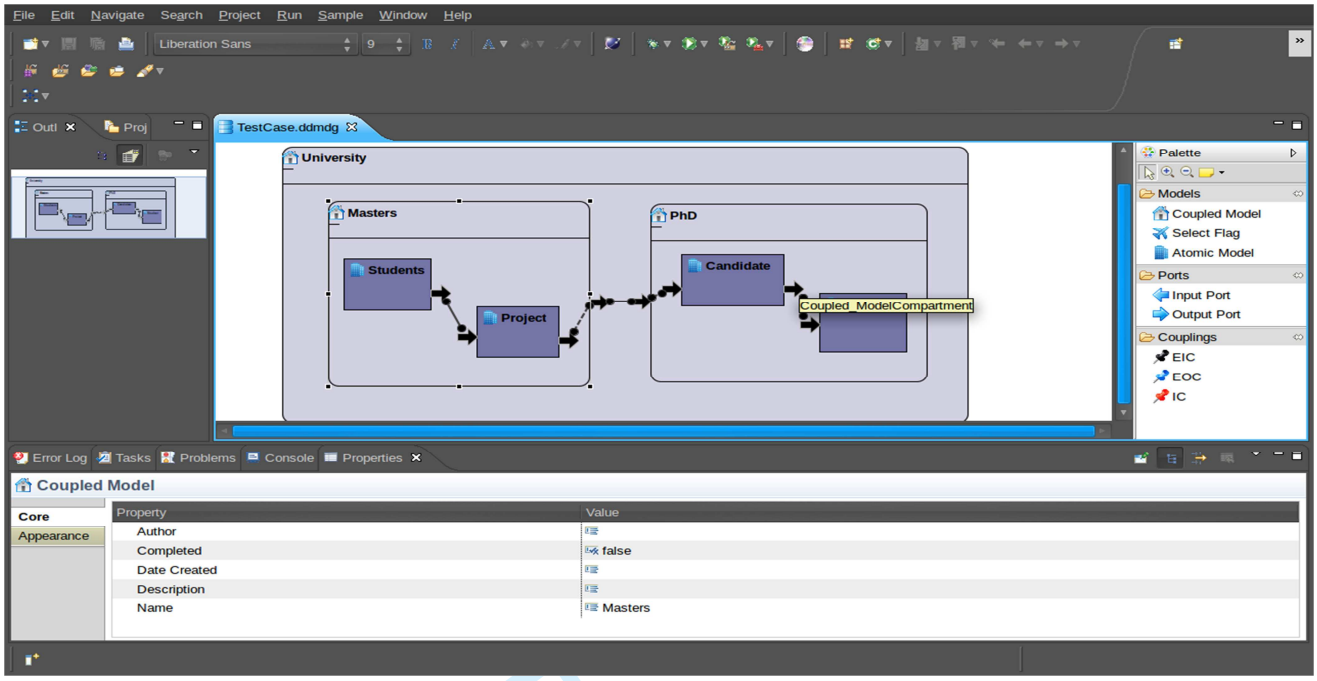


Figure 21: DEVS Model of the University System

Split operation creates a partition of *Nodes* and causes an increase in the number of Root Coordinators on the obtained *Tree* from the previous operation. This functionality is given by the Split plug-in. The existing relationships between these *Nodes* are not modified but new ones are created to accommodate the new Root Coordinators. In this example 2 *Trees* were created from the initial Simulation *Tree* also the initial Root Coordinator still exists as part of the *Trees*. As seen in Figure 22(b), what is usually obtained after a *Split* operation is a *Simulation Skeleton* and stored as an XML file for use by the Cluster plug-in.

During the *Cluster* operation, each *Node* of the *Simulation Skeleton* is grouped into available number of *Processes* thereby creating a *Simulation Bundle* (Figure 22(c)). This is done in such a way that each *Process* contains at least a *Node* and no *Process* shares a *Node* with another *Process*. Also each *Process* has a one-to-one relationship with another *Process*. The Cluster plug-in provides this functionality and produces the *Simulation Bundle* as an XML file.

The information about each of the Processors is contained in a XML file and used by Map plug-in. The *Map* operation takes each of these *Processes* from the *Simulation Bundle* and plots them to an available number of *Processors* thereby producing a *Simulation Graph* (Figure 22(d)). The information about each of these *Processors* is stored in the VM.xml file. Just as in the case of a *Cluster Operation*, each *Process* cannot exist on more than one *Processor* and each *Processor* must contain at least a *Process*. The outcome of this is an XML file representation of *Simulation Graph* structure which can be used for automatic code synthesis (Java, C++, C#, python) and formal analysis.

The complete XML representation for each of the files used in Figure 22 together with their document description is given in the appendix.

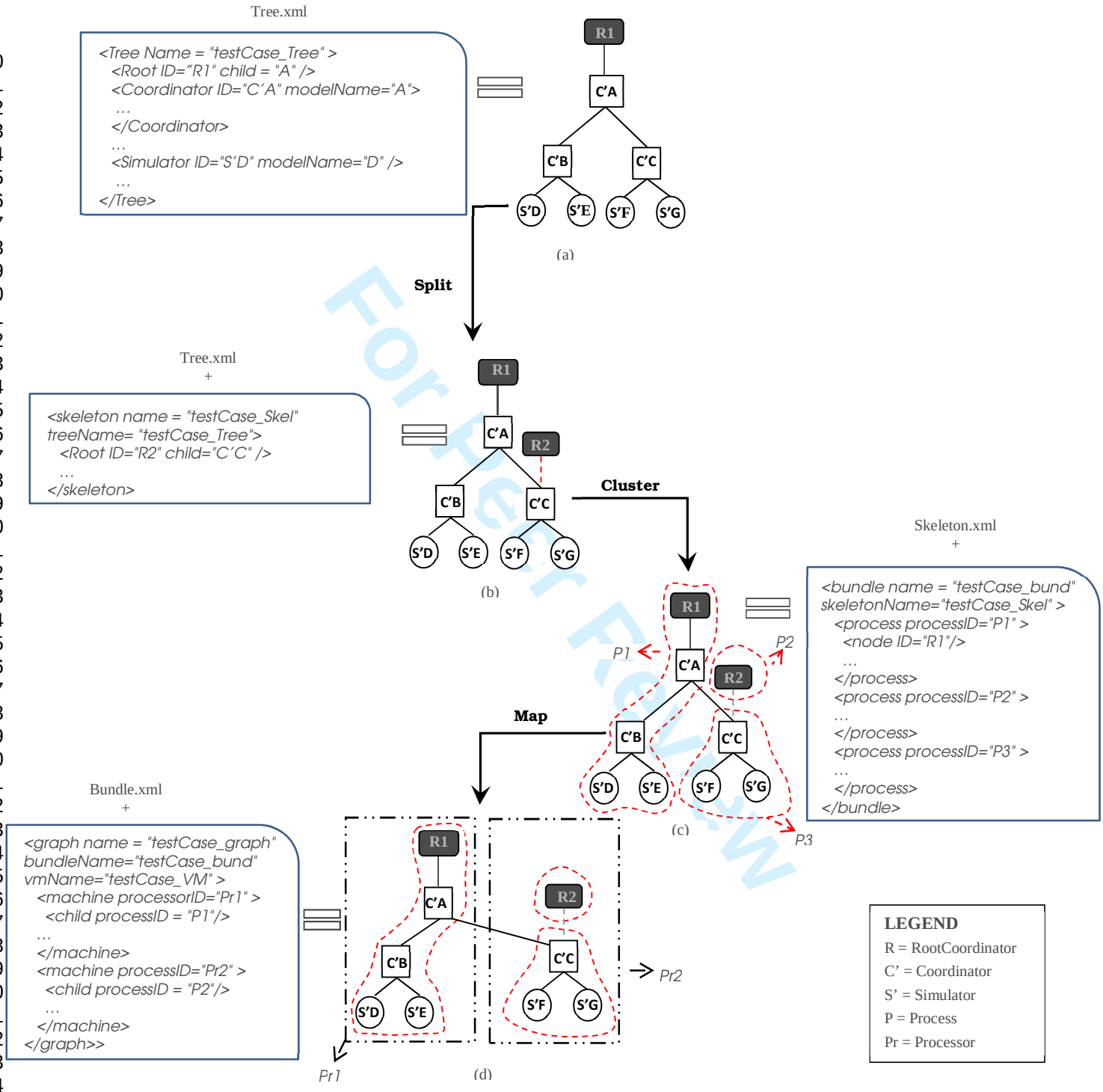


Figure 22: Illustration of Simulation Graph construction using the XML-based Language

6. DISCUSSION

Our work presents an abstract approach to unifying both parallel and distributed simulation main concepts. The difference between the parallel and distributed concept is how abstract links between the processors of the Simulation Graph are implemented; in parallel simulation they are realized by the architecture of the parallel computer while in distributed simulation they are realized through networking protocols (e.g. HTTP, FTP, etc.) or technologies (CORBA and so on).

The purpose of this paper is to propose a generic standard for DEVS PADS implementation strategies and contribute to a better understanding of the Simulation Structures for any implementation strategy. This methodology takes into account the utilization of DEVS PADS for simulation, the Simulation Structures and the operations that can be performed on them. At this point, we discuss some of the potential benefits that can be found in this methodology.

- A simple and efficient guideline: Another essence of this paper is to provide a guideline, help or open library for performing simulation with DEVS PADS. This has been achieved through the proposed methodology which also uses a layered approach in its representation. This approach provides a common frame of reference for performing simulation with DEVS PADS. Also it promotes modularity and Simulation Structure reuse.
- A basis for introducing the evaluation of DEVS PADS strategies: As it has been observed that there are different practices behind the concept of exploiting DEVS PADS which usually results in obtaining a Simulation Graph. The classification of these practices is based on varying the number of elements (i.e. Trees, Processes, and Processors). For example, in Parallel CD++ [22], the Simulation Graph Strategy used consists of a Tree, many Processes and many Processors while the Abstract Threaded Simulator of the James II [18] package a Tree, many Processes and a Processor and Conservative CD++ ([19], [29]) uses many Trees, many Processes and many Processors. With the various practices that have been observed, they introduce the need for evaluation or complexity analysis. This would be used to determine which of the obtained Simulation Graph is more efficient or which would give an improved performance and so on. This methodology therefore gives a foundation for achieving this through its generic nature.
- There has been an on-going research effort in providing standard representation of DEVS to support common understanding, sharing and interoperability. Decisions which are to be taken while implementing a DEVS PADS simulator include how events should be processed, what simulation architecture to use, what should be the organizational architecture and so on. These decisions could be challenging therefore we proposed a solution by going through a review of existing approaches, categorized them (i.e. Simulation Graph strategies) and came up with an integrative view producing a formally specified unified taxonomic framework. As a result, all practitioners dealing with building DEVS PADS simulators can now choose to use this new layered taxonomy as a reference framework for specifying their approaches unambiguously, fostering knowledge reuse across the community. Thus, our work contributes towards implementing DEVS PADS simulators and can be taken as a helpful tool in the context of the DEVS standardization efforts.

Moreover, the underlying methodology uses a layered approach in its representation as shown in Figure 23. This approach permits the easy integration of the identified Simulation Structures into the taxonomic framework. As we move up the layers, each Simulation Structure inherits the DEVS PADS element of the Simulation Structure beneath it. The Simulation Tree consists of just the Tree and the Simulation Graph combines all the elements used in DEVS PADS i.e. Tree, Process and Processor. The

benefit of this approach is that it gives a straight and clear guideline for realizing DEVS PADS by avoiding the possible pitfalls:

- A common pitfall is in trying to distribute the model instead of the simulation (this confusion is due to the fact that most simulations don't separate clearly the concerns). The taxonomy makes it clear what should be distributed i.e. the simulation protocol and not the simulation model.
- When defining the Simulation Graph, a common pitfall is to understand the partitioning as a one-to-one mapping between simulation nodes and processes. The taxonomy makes clear the difference between the nodes that represent the simulation components and the way they can be aggregated at the implementation level as part of a single process (e.g. the same thread can implement a coordinator and its children simulators). As an illustration, trying to model the SG of the university system given in Figure 21 by adopting single tree - single process - single processor strategy will lead to implementing the whole Simulation Tree as the process (which implies that all nodes of the Simulation Tree are implemented as passive nodes except the Root Coordinator which will be the main program).
- From an existing distributed code, it is difficult to build a new distributed solution which must implement a different Simulation Graph strategy. An example is the building of a new N-N-N strategy for an existing N-N-N strategy by increasing or decreasing the number of trees/processes/processors or by reallocating differently processes (respectively nodes) to processors (respectively processes). Taxonomy provides a framework for specifying, comparing and contrasting various implementations and offers a standardized platform for all DEVS PADS methodologies. To modify the Simulation Graph for adaptation to a new computing architecture, starting from the SG layer there would be a need to move to the layer below it or back to the Simulation Tree before making the necessary modification in the Simulation Structure (see Figure 23). Using this layered approach proposed in this methodology, this way, errors can be easily detected and corrected.
- The lack of a unified approach for DEVS PADS development can deter the easy integration of DEVS PADS simulators. For example, combining two existing trees (or combining two existing Simulation Graph or an existing Simulation Tree with an existing Simulation Graph) to get a final Simulation Graph is not obvious. Also, with the various practices involved in building DEVS PADS simulators, it is difficult to evaluate each design and implementation on the same basis. The proposed approach offers a common frame of reference for all DEVS PADS simulators. Failure or inability to facilitate consistency while specifying a Simulation Structure indicates a lack of focus on the essential elements of the taxonomic framework and an understanding of the value that each structure contributes within the framework.

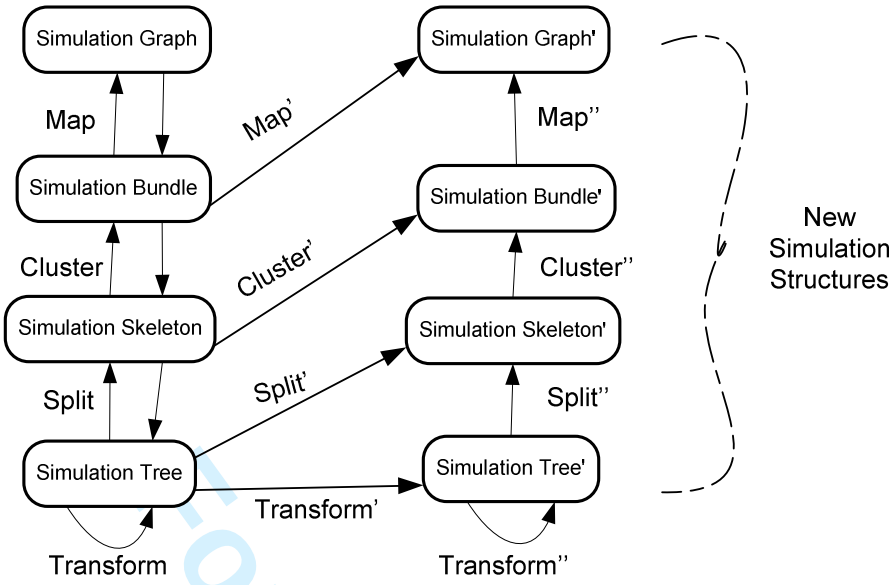


Figure 23: Layered Approach

7. CONCLUSION

This paper has presented a taxonomy and its impact as a methodological framework for providing DEVS PADS solutions. It has provided a more systematic understanding of the process of constructing a DEVS simulator. Also, it proffered an abstract way for integrating different and heterogeneous DEVS implementation strategies. As a consequence we see that this taxonomy would enable practitioners dealing with building DEVS PADS solutions to specify their Simulation Graph strategy explicitly as well as foster knowledge reuse across the community. While some research efforts have been on interoperating DEVS simulators [33] we have focused on how to build a DEVS simulator as an important contribution to DEVS Standardization [34]. We are developing a software tool called SimStudio [26] to support the methodological framework.

REFERENCES

[1] Fujimoto, R. M. “Parallel Discrete Event Simulation,” *Communications of the ACM* 33, no.10 (1990): 30-53.

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

[3] Wainer, Gabriel A. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. New York: CRC Press, 2009.[4] Balci, O. "The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages." *Proceedings of the 1988 Winter Simulation Conference* (1988): 287-295.

[5] Zeigler, B P. *Theory of Modeling and Simulation*. New York: Wiley-Interscience, 1976.

[6] Chow, A C., and B P. Zeigler. "Revised DEVS: A Parallel, Hierarchical, Modular Modeling Formalism." *Proceedings of the Winter Simulation Conference* (1994)

[7] Kim, K H., Y R. Seong, T G. Kim, and K. H. Park. “Distributed Simulation of Hierarchical DEVS Models: Hierarchical Scheduling Locally and Time Warp Globally.” *TRANSACTIONS of the SCS International* 13, no. 3, (1996): 135-154.

[8] Jafer, Shafar, and Gabriel A. Wainer. “Flattened Conservative Parallel Simulator for DEVS and CELL-DEVS.” *Proceedings of CSE (1).* (2009): 443-448.

[9] Glinsky, E., and Gabriel A. Wainer. “Performance Analysis of Real-Time DEVS Models.” *Proceedings of Winter Simulation Conference* (2002).

[10] Kim, K., W. Kang, B. Sagong, and H. Seo. “Efficient Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-Hierarchical One.” *Proceedings of 33rd Annual Simulation Symposium*. (2000)

[11] Kim, K., and W. Kang. “CORBA-based, multi-threaded distributed simulation of hierarchical DEVS models: Transforming model structure into a non-hierarchical one.” *International Conference on Computational Science and Its Applications, Assisi, Italy*. (2004)

[12] Wainer, Gabriel A. "CD++: A Toolkit to Define Discrete-event Models", *Software, Practice and Experience.* Wiley 32, no. 3, (2002): 1261-1306.

[13] Himmelspach, Jan, Roland Ewald, Stefan Leye, and Adelinde Uhrmacher. "Parallel and Distributed Simulation of Parallel DEVS Models." *Proceedings of the 2007 Spring Simulation Multiconference* (2007)

[14] Bryant, R E. “Simulation of Packet Communication Architecture Computer Systems.” Technical Report MIT-LCS-TR-188, Massachusetts Institute of Technology, Cambridge, USA. 1977

[15] Chandy, K M., and J Misra. "Distributed simulation: A Case Study in Design and Verification of distributed programs." *IEEE Transactions on Software Engineering*. (1978): 440-452.

[16] Misra, Jayadev. “Distribute Discrete-Event Simulation.” *ACM Computing Surveys* 18 no.1 (1986): 39-65.

[17] Jefferson, D R. "Virtual Time." *ACM Transactions on Programming Languages and Systems* 7, no. 3 (1985): 404-425.

[18] Himmelspach, Jan, and Adelinde Uhrmacher. "Sequential Processing of PDEVS Models." *Proceedings of the 3rd EMSS* (2006): 239-244.

[19] Jafer, Shafar,and Gabriel A. Wainer. "Conservative DEVS – A Novel Protocol for Parallel Conservative Simulation of DEVS and Cell-DEVS Models." *Proceedings of 2010 Spring Simulation Conference (SpringSim10), DEVS symposium* (2010): 168-175.

[20] Park, S., C. A Hunt, and B. P. Zeigler. "Cost-based Partitioning for Distributed and Parallel Simulation of Decomposable Multi-scale Constructive Models." *SIMULATION* 82 no. 12, (2006): 809–826.

[21] Jean-Sébastien Bolduc and Hans Vangheluwe. "A Modeling and Simulation Package for Classic Hierarchical DEVS." *Technical report, McGill University, School of Computer Science,* (2002)

[22] Troccoli, A., and Gabriel A. Wainer. "Implementing Parallel Cell-DEVS." *Proceedings of the 36th Annual Symposium on Simulation, March 30-April 02 (2003): 273-277.*

[23] Christensen, E.R., and B. P. Zeigler. "Distributed Discrete Event Simulation: Combining DEVS and Time Warp." In Proceedings of the SCS Eastern Multiconference on AI and Simulation Theory and Applications. 1990

[24] Qi, Liu, and Gabriel A. Wainer, "Parallel Environment for DEVS and Cell-DEVS Models." *Simulation*: *Transactions of the Society for Modeling and Simulation International* 83, no. 6 (2007): 449-471.

[25] Glinsky, E., and Gabriel A. Wainer, "New Parallel Simulation Techniques of DEVS and Cell-DEVS in CD++." *Proceedings of 39th Annual Symposium* (2006): 244-251.

[26] Traoré, M. K., "SimStudio. A Next Generation Modeling and Simulation Framework." *SIMUTools'08, ISBN 978-963-9799-20-21, Marseille, France, March 3-7* (2008)

[27] Touraille, L., M. K. Traoré, and D. Hill. "A Model-Driven Software Environment for Modeling, Simulation and Analysis of Complex Systems." *Proceedings of the SCS SpringSim (2011) Conference*.

[28] Ighoroje, Ufuoma B. "An Eclipse-Based Graphical Editing Tool for Dynamic Systems Simulation." *MSc Thesis. African University of Science and Technology, Abuja* 2010.

[29] Jafer, Shafar, and Gabriel A. Wainer. "Conservative vs. Opimistic Parallel Simulation of DEVS and Cell-DEVS: A Comparative Study." *Proceedings of 2010 Summer Simulation Conference (SummerSim10), SCSC symposium* (2010): 342-350.

[30] Wainer G.A., K. Al-Zoubi, O. Dalle, D. Hill, S. Mittal, J.L. Risco-Martin, H. Sarjoughian, L. Touraille, M.K. Traoré and Zeigler B.P.. *DEVS Standardization: Ideas, Trends and Future. Discrete-Event Modeling and Simulation: Theory and Applications (Wainer G., Mosterman P.J.),* Taylor and Francis Group 2010: 389-392.

[31] Cho, Y. W., X. Hu, and B. Zeigler. "The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems." *Simulation 79 (4) (2003): 197–210.*

[32] Sarjoughian, H. S., and B. P. Zeigler. "DEVS and HLA: Complementary paradigms for M&S?" *Transactions of the SCS 17 (2000): 187–197.*

[33] Kim J. H. and T. G. Kim, "DEVS Framework and Toolkits for Simulators Interoperation Using HLA/RTI." Proceedings of Asia Simulation Conference, China (2005): 16–21.

[34] Wainer, G. A. "DEVS Standardization Study Group." *Interim Final Report (Seattle, WA. Apr 26). The SISO Standards Activity Committee (SAC). (*2005).

APPENDIX

- Tree

DTD	XML
<?xml version="1.0" encoding="UTF-8"?> **<!ELEMENT Tree (Root, Coordinator+, Simulator+)>** **<!ATTLIST Tree** **name CDATA #REQUIRED** **xmlns CDATA #IMPLIED>** **<!ELEMENT Root EMPTY>** **<!ATTLIST Root** **ID CDATA #REQUIRED** **child CDATA #REQUIRED>** **<!ELEMENT Coordinator (child+)>** **<!ATTLIST Coordinator** **ID CDATA #REQUIRED** **modelName CDATA #REQUIRED>** **<!ELEMENT child EMPTY>** **<!ATTLIST child** **ID CDATA #REQUIRED>** **<!ELEMENT Simulator EMPTY>** **<!ATTLIST Simulator** **ID CDATA #REQUIRED** **modelName CDATA #REQUIRED>**	<?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE Tree SYSTEM "tree.dtd"> <Tree name="testCase_Tree" xmlns="http://ddml/1.0" > <Root ID="R1" child="C'A" /> <Coordinator ID="C'A" modelName="A"> <child ID="C'B" /> <child ID="C'C" /> </Coordinator> <Coordinator ID="C'B" modelName="B"> <child ID="S'D" /> <child ID="S'E" /> </Coordinator> <Coordinator ID="C'C" modelName="C"> <child ID="S'F" /> <child ID="S'G" /> </Coordinator> <Simulator ID="S'D" modelName="D" /> <Simulator ID="S'E" modelName="E" /> <Simulator ID="S'F" modelName="F" /> <Simulator ID="S'G" modelName="G" /> </Tree>

- Skeleton

DTD	XML
<?xml version="1.0" encoding="UTF-8"?> **<!ELEMENT skeleton (Root+)>** **<!ATTLIST skeleton** **name CDATA #REQUIRED** **treeName CDATA #REQUIRED** **xmlns CDATA #IMPLIED>** **<!ELEMENT Root EMPTY>** **<!ATTLIST Root** **ID CDATA #REQUIRED** **child CDATA #REQUIRED>**	<?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE skeleton SYSTEM "skeleton.dtd"> <skeleton name = "testCase_Skel" treeName= "testCase_Tree" xmlns = "/XML_DEVS/transform"> <Root ID="R2" child="C'C" /> </skeleton>

- Bundle

DTD	XML
<?xml version="1.0" encoding="UTF-8"?> **<!ELEMENT bundle (process+)>** **<!ATTLIST bundle** **name CDATA #REQUIRED** **skeletonName CDATA #REQUIRED** **xmlns CDATA #IMPLIED>** **<!ELEMENT process (node+)>** **<!ATTLIST process** **processID CDATA #REQUIRED>**	<?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE bundle SYSTEM "bundle.dtd"> <bundle name = "testCase_bund" skeletonName="testCase_Skel" xmlns = "/XML_DEVS/skeleton"> <process processID="P1" > <node ID="R1"/> <node ID="C'A" /> <node ID="C'B" />

<!ELEMENT node EMPTY> **<!ATTLIST node** **ID CDATA #REQUIRED>**	<node ID="S'D" /> <node ID="S'E" /> </process> <process processID="P2" > <node ID="R2" /> </process> <process processID="P3" > <node ID="C'C" /> <node ID="S'F" /> <node ID="S'G" /> </process> </bundle>

- VM

DTD	XML
<?xml version="1.0" encoding="UTF-8"?> **<!ELEMENT VM (Processor+)>** **<!ATTLIST VM** **name CDATA #REQUIRED>** **<!ELEMENT Processor>** **<!ATTLIST Processor** **processorID CDATA #REQUIRED** **name CDATA #REQUIRED** **address CDATA #REQUIRED>**	<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE VM SYSTEM "vm.dtd"> <VM name="testCase_VM"> <Processor processorID="P_1" name = "Virtual_1" address="192.168.110.1"/> <Processor processorID="P_2" name = "Virtual_2" address="192.168.110.2"/> </VM>

- Graph

DTD	XML
<?xml version="1.0" encoding="UTF-8"?> **<!ELEMENT graph (vm:machine+)>** **<!ATTLIST graph** **name CDATA #REQUIRED** **bundleName CDATA #REQUIRED** **vmName CDATA #REQUIRED** **xmlns:bundle CDATA #IMPLIED** **xmlns:vm CDATA #IMPLIED>** **<!ELEMENT vm:machine (bundle:child+)>** **<!ATTLIST vm:machine** **processorID CDATA #REQUIRED>** **<!ELEMENT bundle:child EMPTY>** **<!ATTLIST bundle:child** **processID CDATA #REQUIRED>**	<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE graph SYSTEM "graph.dtd"> <graph name = "testCase_graph" bundleName="testCase_bund" vmName="testCase_VM" xmlns:bundle = "/XML_DEVS/bundle" xmlns:vm = "/XML_DEVS/vm"> <vm:machine processorID="Pr1" > <bundle:child processID = "P1"/> </vm:machine> <vm:machine processorID="Pr2" > <bundle:child processID = "P2"/> <bundle:child processID = "P3"/> </vm:machine> </graph>