TOWARDS A UNIFYING FRAMEWORK FOR ENGINEERING DEVS DISTRIBUTED SIMULATIONS

A Thesis submitted to the Department of Computer Science and Engineering

African University of Science and Technology

In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

BY

ADEDOYIN ADEGOKE

December 2017

CERTIFICATION

This is to certify that the thesis titled "*Towards a Unifying Framework for Engineering DEVS Distributed Simulations*" submitted to the school of postgraduate studies, African University of Science and Technology (AUST), Abuja, Nigeria for the award of the Doctor of Philosophy degree is a record of original research carried out by *Adedoyin Adegoke* in the *Department of Computer Science and Engineering*.

TOWARDS A UNIFYING FRAMEWORK FOR ENGINEERING DEVS DISTRIBUTED SIMULATIONS

By:

ADEDOYIN ADEGOKE

A THESIS APPROVED BY THE COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

RECOMMENDED:

……………………………………… [illegible] …………..

Supervisor, Professor Mamadou Kaba Traoré

…………………………………………………………………

Committee Member, Professor Ousmane Thiaré

…………………………………………………………………

Committee Member, Professor Peter Onwualu

…………………………………………………………………

Head of Department, Professor David Amos

Approved by: …………………………………………………………………

Vice President (Academic), Professor Charles Chidume

December 18th, 2017

Date: …………………………………………………………………

Abstract

Simulation engineering employs the use of simulation models, simulation environments, and simulation execution platforms in providing applicable Modeling and Simulation (M&S) solutions for studying complex systems. Discrete Events Systems Specification (DEVS), a well-known formalism, offers a framework that separates simulation model specifications from simulation environments and adapts well to distributed simulation engineering. Over the years, there are noticeable progresses made in using DEVS to 1) describe models for different systems, 2) implement simulation environments, and 3) describe execution on parallel/distributed platforms. Adopting DEVS as a solution for distributed simulation engineering poses an important challenge in terms of how to manage the migration of models, simulation engines, or platforms in the event of an unforeseen change in requirement. Consequently, it is necessary to devise techniques to systematically support these migrations.

A variety of DEVS environments have been implemented without a standard developmental guideline across board consequently revealing the need of central frameworks for integrating heterogeneous DEVS simulators. There are salient concepts that are intuitively defined when implementing a DEVS simulator, for example, how should events be processed, what simulation platform to use, what existing procedures (set of rules/algorithm) can be used, what should be the organizational architecture and so on. However, defining these concepts further reveals the lack of a systematic and quantifiable approach that can guide the DEVS simulator development process.

In order to address the identified issues, this research proposes two major contributions. First is a conceptual and formalized guide that models the process of building a DEVS implementation strategy. 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. This contribution offers abstract way for integrating heterogeneous DEVS implementation strategies and thus can serve as a contribution to the on-going DEVS standardization efforts. The second contribution is, Model-Driven Distributed Simulation Engineering Framework (MD2SEF), a multi-layered framework that supports the systematic and agile engineering of DEVS distributed simulations by allowing to conceptually represent DEVS models, specify DEVS simulation environment, define the execution platform, integrate simulation components, and automate the generation of distributed codes.

Keywords: DEVS, parallel/distributed simulation, model-driven framework, simulation engineering, agility

ACKNOWLEDGEMENT

First, I am super-duper grateful to God, the Olowogbogboro, who has made everything work out for my good.

I deeply appreciate my supervisor, Prof. Mamadou K. Traore, for his insightful suggestions and warm encouragements throughout the duration of this thesis. His diligence and commitment to science have been and will be a great influence on me for many years to come. I am grateful for having the opportunity to learn from him and work with him.

I would like to thank the people who participated in my research, who contributed either through emails or meetings and whose insights and views shaped my work. To members of GReP research group, your friendships at research and personal levels paved way for me to succeed. All of you have been the witnesses of my complaints, my results and achievements, my failures and my frustrations during these years. I sincerely appreciate every one. I would like to extend my gratitude to all my present/past colleagues and staff in AUST, you all are amazing.

The love and the gracious support I have received from the entire Durojaiye and Adegoke clan has been tremendous. Thank you for your prayers and sacrifices in every form. To my Mom, you are not here; I can only imagine how joyful and proud you will be.

My deepest thanks go to Oluwakorede, my K-bobo, for sharing my attention with my studies in addition; you gave me all the love, hugs, and kisses you could muster. I love you very much.

A very special appreciation to my husband 'dimeji, your love, trust and support enabled me to complete this thesis. You stood beside me in the darkest moments of the journey and helped me overcome it with success. Thank you for being with me all the way.

TABLE OF CONTENTS

List of Figures

List of Abbreviations

ATL	ATLAS Transformation Language
CDEVS	Classic DEVS
CIM	Computation Independent Model
CORBA	Common Object Request Broker Architecture
DESS	Differential Equation System Specification
DEVS	Discrete Events Systems Specification
DTSS	Discrete Time System Specification
HLA	High Level Architecture
HPC	High Performance Computing
M&S	Modeling and Simulation
M2M	Model to Model
M2T	Model to Text
MD2SEF	Model Driven Distributed Simulation Engineering Framework
MDA	Model Driven Architecture
ML	Middleware Layer
OCL	Object Constraints Language
PDES	Parallel Discrete-Event Simulation
PDEVS	Parallel DEVS
PDM	Platform Description Model
PIM	Platform Independent Model
PSM	Platform Specific Model
SB	Simulation Bundle
SCL	Simulation Code Layer
SG	Simulation Graph

SiS	Simulation Structure
SML	Simulation Modeling Layer
SS	Simulation Skeleton
SSL	Simulation Structure Layer
ST	Simulation Tree
UML	Unified Modeling Language
VM	Virtual Machine
XMI	XML Metadata Interchange
XML	EXtensible Mark-Up Language
XSLT	EXtensible Stylesheet Language Transformations

List of Tables

1. Introduction

1.1. General Introduction

Modeling and Simulation (M&S) play a key role in the study and development of complex systems from different domains. An M&S solution consists of models and simulators as basic components; while models are an abstraction of the system under study, simulators are computational devices used for generating the behavior of the models. In addition, computing infrastructures are fundamental requirements in an M&S study as they determine the success of the study. Due to the growing complexity of systems to be modeled, efficient simulation of such systems should not be performed on a single physical processor. One solution will be to exploit platforms containing millions of cores for large-scale simulation applications.

Parallel discrete-event simulation (PDES) (Fujimoto, 2001), also often referred to as parallel/distributed simulation, is a blend between modeling and simulation, and high-performance computing. PDES is concerned with executing discrete-event simulation programs on high computing infrastructures such as parallel or distributed computers. As such, it can overcome the limitations imposed by sequential simulation in both the execution time and the memory usage. It is a widely researched area with some potential benefits. Firstly, the use of parallel processors promises an increase in execution speed and a reduction in execution time. Secondly, the potentially larger amount of available memory on parallel processors will enable the execution of larger simulation models. Thirdly, 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 simulators (Fujimoto, 2001; Ga Wainer, 2009; Bernard P. Zeigler, Praehofer, & Kim, 2000).

DEVS (Discrete Events System Specification) (Bernard P. Zeigler et al., 2000) offers a platform for the modeling and simulation of sophisticated systems in a variety of domains. DEVS has gained popularity due to the need to build and obtain quality simulation models. It provides a clear separation between models and simulators, which makes it easy for a modeler to focus on developing models while making use of existing and implemented simulators to generate the behavior of the system under study. 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. In addition, a DEVS simulator has been proven (Bernard P. Zeigler et al., 2000) to be capable of reproducing behaviors that are identical to that of the system under observation.

1.2. Research Motivation

As DEVS simulation models get bigger and more complex there is the need to harness PDES. However, mapping the simulation models to high performance infrastructures to perform distributed simulation is a demanding task as these infrastructures are built to harness the power of aggregated processors and do not provide guidelines for implementing and executing distributed simulation algorithms.

On the other hand, PDES is a matured field of study, but its adaptability to DEVS is also arduous task. Two main reasons concur with this point;

- Various concerns are involved in the process of building a DEVS PDES. A good building strategy should be based on a clear separation of concerns. The formal specification of the key concepts and transformation that are found in these concerns would remove ambiguity and reduce accidental complexity (i.e., wrong implementation due to misunderstanding of concepts).

- The absence of a systematic and quantifiable approach that can guide this DEVS PDES building process.

DEVS simulation principle takes a DEVS model specification and maps it to DEVS Simulation Tree (ST) using well-defined DEVS operational semantics. As it has been observed from existing works, DEVS PDES implementation strategies (Franceschini, Bisgambiglia, Touraille, Bisgambiglia, & Hill, 2014) differ from one another (see Figure 1-1) thereby lacking a standard approach. There is the need to ensure that a given DEVS PDES implementation strategy is consistent with the DEVS formalism and simulation protocol. In addition, there is the need for a strategy to support the integration of models in the event of unanticipated changes.

Existing DEVS PDES strategies are tightly coupled with a high computing platform (Cho, Zeigler, & Sarjoughian, 2001; Mittal, Risco-Martín, & Zeigler, 2009; Park, Kim, Hunt, & Park, 2007; Ming Zhang, Zeigler, & Hammonds, 2006) so as to achieve distributed simulation. This inflexibility places represents an hindrance when adapting a strategy to a new platform. There is therefore the need to ensure that a DEVS simulation model built for one distributed platform can be easily adapted to another.

This dissertation therefore attempts to answer the following question,

“How do we map any DEVS simulation model on to any parallel/distributed infrastructure?”

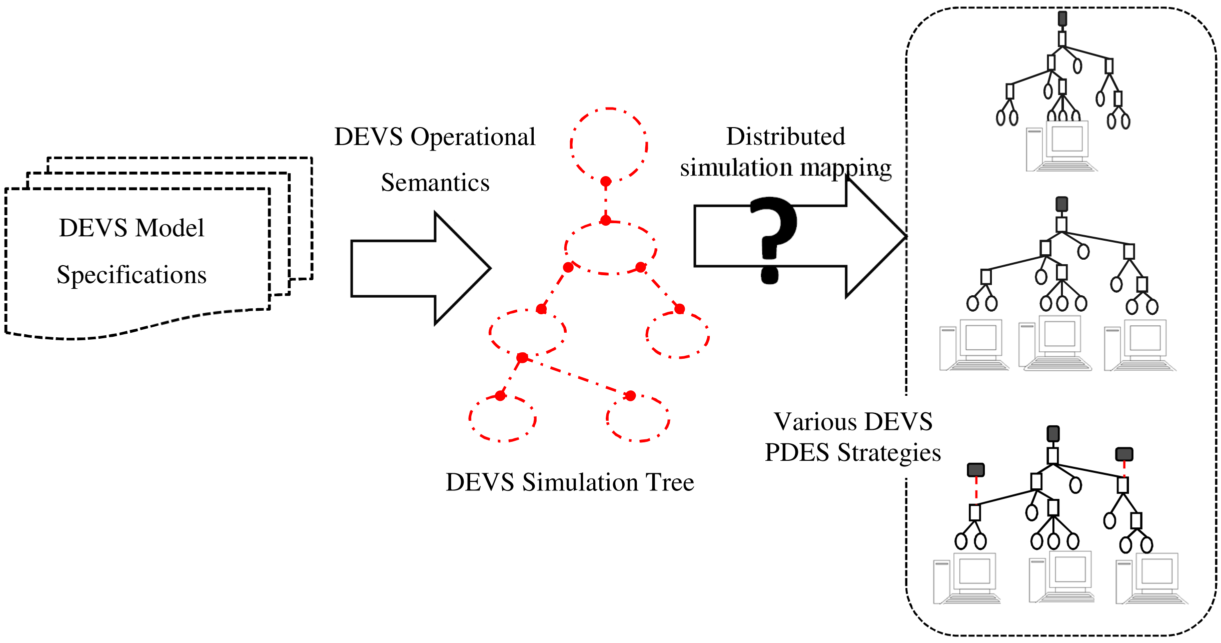

DEVS Model Specifications
DEVS Operational Semantics
DEVS Simulation Tree
Distributed simulation mapping
?
Various DEVS PDES Strategies

Figure 1-1: Process of mapping DEVS to Parallel Discrete-Event Simulation

1.3. Research Objectives

Therefore, the objective of this research is to provide a unifying framework that supports the process of mapping any domain specific system specified as a DEVS model on to any parallel/distributed infrastructures and hence obtain simulation codes.

To achieve this objective, three partial objectives are considered throughout this research:

O 1: To provide support for the specification of DEVS PDES implementation strategies

O 1.1: To provide a conceptual understanding and specification of concepts found when developing a DEVS PDES implementation strategy

O 1.2: To specify a generic methodology for the development of DEVS PDES strategies.

O 2: To provide support for the modeling and migration of DEVS models

O 3: To specify a model that has an extended simulation architectural scope i.e. allows the integration and migration of parallel/distributed infrastructures

O 4: To specify a unifying framework that enables the engineering and re-engineering of DEVS distributed simulations

1.4. Summary of Contributions

Overall, the primary contribution (see Figure 1-2) of this research is the development of an approach methodology to engineer DEVS distributed simulation starting from the modeling stage to the code generation stage. In addition, this thesis proposes the specification DEVS distributed simulations independent of any DEVS implementation or platform. It uses the separation of concerns approach in maintaining the independence of distributed simulation components and bridging them together to obtain distributed simulation codes. It also offers the integration of DEVS models and demonstrates the suitability of a DEVS PDES implementation strategy to adapt to changes in a parallel/distributed simulation infrastructure.

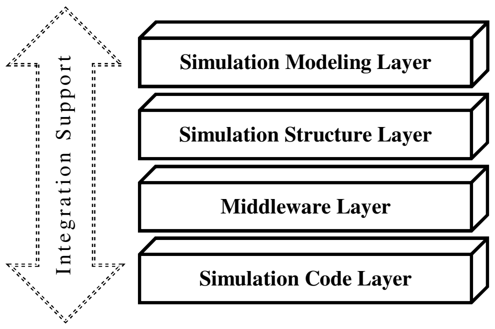


Figure 1-2: Multilayered framework for the development of DEVS distributed simulations

The outcome of this work is a conceptualized and implemented unifying framework. This framework supports the multilayer development approach as it is built to capture the development of DEVS simulations at four layers, namely; the modeling, structure, middleware and the code layer. The modeling layer describes a DEVS model specification. The structure layer captures definitions of DEVS simulation protocol, the middleware layer describes the platform and the code layer provides an abstraction from which simulation codes can be generated. The framework is defined to be as generic as possible to enable the integration of heterogeneous implementations of the DEVS simulation protocol and eliminate the need for the user to start building an implementation from scratch. Full and partial automated integration processes needed for the layers to interoperate are captured at the integration support layer.

1.5. Thesis Organization

Remaining Chapters of this thesis are organized as follows:

Chapter 2 provides with a background of the research context and literature review. To that end, section 2.2 introduces DEVS as a framework for modeling and simulation. Next, the context for executing simulation models on parallel/distributed infrastructures is reviewed in section 2.3. A state of the art on model driven concepts and techniques useful for achieving the stated objectives of this thesis is discussed in section 2.4.

Chapter 3 is a proposal that can support the understanding and development of DEVS PDES strategies. To that purpose, section 3.2 describes a pattern that is common to different versions of the DEVS simulation protocol. Sections 3.3 to 3.6 introduce concepts used by DEVS simulation practitioners during the development of a DEVS PDES implementation strategy.

Chapter 4 builds on the knowledge gained from Chapter 3. To that purpose, sections 4.1 to 4.3 present formal specifications of the components and processes involved in building a DEVS PDES strategy. A methodology that can guide a strategy building process is proposed in Section 4.4 while the principle behind the methodology is examined in section 4.5. Section 4.6 presents a case study on the use of the methodology.

Chapter 5 proposes the unifying framework as an integrative solution for engineering DEVS distributed simulations. To that purpose, Section 5.2 brings forward the conceptual architecture of framework. In the remainder of the chapter, the framework in examined with respect to its design principle and the contribution of model driven techniques to its development.

Chapter 6 addresses the framework, examines each of the layers and their contribution to the overall framework. To this end, Sections 6.2 to 6.4 examines the layers for specifying DEVS models, DEVS PDES strategy, and parallel/distributed infrastructures respectively while Section 6.5 presents a layer for generating simulation codes. Chapter 7 proposes techniques and tools for enabling integration between each of these layers. It presents a proof of concept implementation of the unifying framework in the DEVS distributed simulation domain. A case study was instantiated to show the applicability of the framework in Chapter 8.

In Chapter 9, related works are examined. Finally, the concluding remarks and future research directions are reported in Chapter 10.

2. Background

2.1. Introduction

This chapter presents the research context of this thesis. First, there is an introduction to a comprehensive modeling and simulation framework for modeling and analysis of Discrete Event Systems. Then, parallel/distributed concepts, relevant to the current research are explored. That is, synchronization and communication techniques. Finally, an architecture including its concepts, techniques, and technologies used in providing automation support is discussed. Furthermore, software tools that implement the aforementioned technologies are examined in this chapter.

2.2. Discrete Events Systems Specification

A general modeling and simulation methodology for describing discrete-event systems including continuous systems after quantization (B.P. Zeigler, Sarjoughian, & Praehofer, 2000) or discretization (Bernard P. Zeigler et al., 2000) is the Discrete Event System Specification (DEVS). It supports hierarchical construction of reusable models in a modular way. DEVS formalism can be considered as a universal formalism as it is capable of subsuming other formalisms (Vangheluwe, 2000). Extensions to the DEVS formalism include Parallel DEVS (Chow & Zeigler, 1994), DEV&DESS for combined continuous-time and discrete-event systems (Bernard P. Zeigler et al., 2000), Real-Time DEVS for real-time discrete-event systems (Hong, Song, Kim, & Park, 1997), Cell-DEVS for cellular automata (Ga Wainer, 2009), Fuzzy-DEVS (Kwon, Park, Jung, & Kim, 1996) and Dynamic Structuring DEVS (Barros, 1995).

DEVS formalism (Bernard P. Zeigler et al., 2000) 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 structure, which defines a DEVS modeling operation, 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.

A DEVS model is created according to a structural specification i.e. Classic DEVS (CDEVS) or Parallel DEVS (PDEVS). CDEVS was introduced to simulate and execute models sequentially on single processor machine. PDEVS has later been introduced to increase the potential of parallelism of DEVS simulation (Chow & Zeigler, 1994; Bernard P. Zeigler et al., 2000). CDEVS and PDEVS can be used to specify the same class of systems. However, this work focuses on using PDEVS as it permits increased degrees of parallelism and proper handling of events collision that can be exploited in parallel and distributed environments. Due to this, it is the preferred choice for our work on High Performance DEVS Simulation.

PDEVS formalism is an extension to DEVS that removes serial execution of events constraints and provides an environment for executing simultaneous DEVS models in parallel. PDEVS was introduced to properly handle collisions during tie breaking, i.e. the behavior when a model receives external events at the same time as its prescheduled internal transition. Previous solutions attempt to define the collision behavior implicitly either through the select function or by saying that an internal transition should occur before a colliding external transition (Chow & Zeigler, 1994). The PDEVS formalism structure explicitly requires a modeler to define the collision behavior by using the confluent transition function. According to Chow (Chow & Zeigler, 1994) other desirable properties provided by PDEVS are degrees of parallelism which can be exploited in parallel and distribution environments and uniformity through the hierarchical construction of models.

2.2.1. Atomic Model Specification

An atomic PDEVS model is formally defined as

$$M = < X_M,\ Y_M, S,\ \delta_{ext}, \delta_{int}, \delta_{conf}, ta, \lambda >$$

Where

$X_M = \{\ (p,v) \mid p\ \in IPorts, v\ \in\ X_p\}$	is the set of input ports and values;
$Y_M = \{\ (p,v) \mid p\ \in OPorts, v\ \in\ Y_p\}$	is the set of output ports and values;
S	is the set of sequential states;
$\delta_{ext} : Q\ x\ X_{M^b}\ \rightarrow S$	is the external state transition function;
$\delta_{int} : S\ \rightarrow S$	is the internal state transition function;
$\delta_{conf} : Q\ x\ X_{M^b}\ \rightarrow S$	is the confluent transition function;
$\lambda : S\ \rightarrow Y_{M^b}$	is the output function;
$ta : S\ \rightarrow R_{0^+ \rightarrow \infty}$	is the time advance function;

With $Q = \{(s,e) \mid s\ \in S, 0 \leq e \leq ta(s)\}$ the set of total states

e is the elapsed time and X_{M^b} is the bag of input events and Y_{M^b} is the bag of output events.

Essentially, the structure of an atomic DEVS model consists of a set of states, inputs, outputs and four functions. The model interacts (i.e., receives and sends values given by λ) with its environment through its ports and remains in a particular state for a given period of time determined by the time advance function. The dynamic behavior of the model is described as follows:

- Internal Transition: the model changes its state at the expiration of time determined by the ta. In this case the elapsed time $e = ta(s)$.
- External Transition: the model changes its state if it receives an external event through its input port before the expiration of time. In this case $e \neq ta(s)$

- Confluent Transition: this transition determines the new state of the model once there is a collision between the internal and external transition i.e., it acts as the tiebreaker.

2.2.2. Coupled Model Specification

A coupled PDEVS model is a model composed of several interconnected atomic or coupled models, that communicate externally using the input and output ports of the coupled model interface. Direct feedback is not permitted on the ports for either the atomic or coupled models.

A coupled PDEVS model is formally defined as

$$CM = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC)$$

Where

$X = \{ (p, v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{ (p, v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

D is the set of component names

Components are PDEVS models. For $d \in D$, M_d is a subcomponent. It can be either Atomic PDEVS model or Coupled PDEVS model.

EIC (External Input Coupling) connects external inputs to component inputs

EOC (External Output Coupling) connects external outputs of components to external outputs

IC (Input Coupling) connects components outputs to component inputs

A coupled model, due to the closure under coupling property, can be regarded as a new DEVS model (Praehofer, Sametinger, & Stritzinger, 2001). This property ensures that the overall behavior of a coupled model is equivalent to a basic atomic model, and therefore, allows for hierarchical model construction and decomposition. (Shiginah, 2006) demonstrates how this property (i.e., the derivation of an atomic model from a coupled model) can be obtained.

2.2.3. DEVS Simulation Protocol

DEVS simulation protocol consists of two major components, that is, the simulation tree and the messages transmitted along the tree. We discuss about the simulation tree in this section and examine the kind of messages on a tree in Section 3.2. Subsequently, in this work, we refer to DEVS PDES implementation strategy as an implementation of DEVS simulation protocol.

The simulation tree is composed of abstract simulators that react based on the kind of message they received. These abstract simulators or Nodes are Root Coordinators, Coordinators, and Simulators; they are used for executing DEVS models. They are organized in a hierarchy that imitates the hierarchical structure of a model as shown in Figure 2-1. The Root Coordinator has an event loop that sends event messages and controls the simulation cycles, while the Coordinator and Simulator are capable of receiving, treating and sending event messages. 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 concept of separation of concerns between model and simulator significantly improves model validation and simulator verification, allows model reusability as well as later extension of the model (Bernard P. Zeigler et al., 2000).

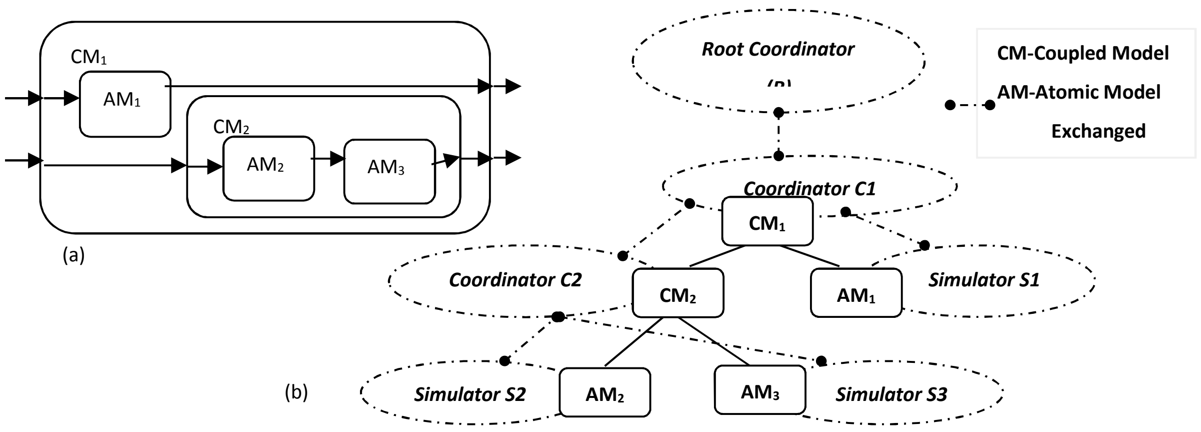


Figure 2-1: (a) DEVS model (b) Hierarchical mapping of DEVS model to abstract simulator

2.2.4. DEVS Implementations

In this section, we briefly look at various DEVS implementations whose architecture is derived from the abstract simulator concepts associated with the hierarchical, modular DEVS formalism.

PythonDEVS is an interpreted (using Python) and an object oriented implementation of the CDEVS formalism (Bolduc & Vangheluwe, 2002). PythonDEVS consists of a modeling layer and a simulation engine. While the modeling layer is built to be consistent with hierarchical DEVS model definition, the execution layer is loosely based on the DEVS simulation protocol. The DEVS-Scheme (Bernard P. Zeigler, 1990) environment is an implementation of the CDEVS formalism in the Scheme functional language (a Lisp dialect) which enables the modeler to specify models directly in its terms in a hierarchical and modular manner. DEVS-Scheme is written in the PC-Scheme language, which runs on DOS compatible microcomputers, and under a Scheme interpreter for the Texas Instruments Explorer. It is implemented, as a shell that sits upon PC-Scheme in such a way, that all of the underlying Lisp-based and object oriented programming language features is available to the user.

So as to remove the indications of sequential processing (found in DEVS-Scheme) and enable full exploitation of parallel processing, DEVS-C++ was proposed (B P Zeigler, Moon, Kim, & Kim, 1996; Bernard P. Zeigler, 1990).It is a portable DEVS-based modeling and simulation environment written in C++, which implements parallel execution and supports large-scale systems. Its execution layer is based on PDEVS formalism. DEVSim++ (T. Kim, Sung, Hong, & Hong, 2010) is an object-oriented DEVS simulator implemented in C++. ADEVS (Nutaro James, n.d.) provides a C++ library based on DEVS, which developers can use to build their own models, and supports integration with other simulation environments.

DEVSJAVA (H. Sarjoughian, Kim, Ramaswamy, & Yau, 2008) is a DEVS-based modeling and simulation environment that provides Java classes for users to implement their own models. It also supports PDEVS models with software real-time, variable structure, 2D/3D cellular automata, and animation. SimBeans (Praehofer et al., 2001) is a component-based software architecture based on Java and JavaBeans. The idea is to provide a set of layered components that can be used in model creation, result output analysis, and visualization using DEVS. JDEVS (Filippi & Bisgambiglia, 2004) is a DEVS modeling and simulation environment written in Java. It allows general-purpose, component-based, object-oriented, visual simulation of models. (Adegoke, 2010) proposed a portable and reusable DEVS Virtual Machine (VM) built using object-oriented Java programming language to execute DEVS models. The VM realizes a purely sequential version of DEVS (i.e., sequential in nature, sequential in execution), a thread-based PDEVS implementation (i.e., parallel in nature, parallel in execution) and a non-thread based PDEVS implementation (i.e., parallel in nature, sequential in execution). Descriptions and comparisons of some DEVS implementations are given in (Van Tendeloo & Vangheluwe, 2017)

2.3. PDES

It is easy to simulate DEVS models, however PDES techniques must be engaged when models and simulation requirements become complex, critical as well as expensive. These techniques introduce the need to achieve proper synchronization and communication between distributed components. We take a look at these features in this section.

2.3.1. Synchronization

Parallel Discrete Event Simulation (PDES) is a technique that can be used for the modeling and simulation of large DEVS models on high performance architectures. In PDES, the sub-systems that make up the original complex system are simulated concurrently on different logical processors (LP). In cases where these sub-systems need to obtain results from each other, they communicate this through their LPs. PDES helps to reduce the computational costs that can be incurred during these communications however it introduces a new problem related to the need of synchronizing the exchange of events between LPs. Importantly, synchronization between different LPs is required since each process needs to know the results of other processes in order to correctly simulate its own subsystem. If the LPs are not correctly synchronized, causality constraints may occur. Causality constraints happen when an LP receives an event with a timestamp lower than its actual simulation time, which can lead to an incorrect result.

There are many synchronization approaches to overcome this problem. An early solution is the one proposed by Chandy, Misra, and Bryant (CMB algorithm) (Bryant, 1977; Chandy & Misra, 1979; Misra, 1986). CMB implements a conservative synchronization where none of the LPs advances its simulation time until it is safe to produce the next event. The synchronization is achieved by enforcing the correct order of all messages, making an LP wait until it is certain that it will never receive a message with an earlier time–stamp than its own logical simulation time before producing the next event. The major drawback is that parallelism exploitation is diminished by the Null message mechanism used to enforce synchronization. By introducing look-ahead and structural analysis, the efficiency of the approach can be improved. In contrast, the optimistic approach introduced by (Jefferson, 1985) that is, Time Warp protocol, allow causality errors to happen temporarily but provide a mechanism called rollback to recover from them during execution. The rollback involves sending anti-messages and saving state values for each LP and thus, memory requirements are larger when compared with other synchronization approaches. Finally, an approach called NOTIME (Rao, Thondugulam, & Radhakrishnan, 1998) explores the effects of not synchronizing the processes thereby eliminating completely the need to rollback or look-ahead. However, this introduces errors in the simulation and is suitable for when simulation speed is more important than simulation accuracy. Using this approach, the different LPs simulate as fast as they can and high level of parallelism can be exploited.

2.3.2. Distributed Communication Middleware

A distributed communication middleware plays an essential role in developing distributed simulation systems and forms a layer between processes and distributed platforms. We sometimes refer to a distributed communication middleware as middleware (see Figure 2-2). In (Bernstein, 1996), a middleware classification and their properties were proposed. A middleware should be able to define higher-level domain-independent features like application programming interfaces (APIs) without the need to interact with low-level details (for example security). In addition, it should be transparent enough to enable communication between processes on available distributed platforms. A platform can be an operating system, a programming language, hardware, or a networking protocol and so on. A distribution communication middleware will therefore need to provide solutions to heterogeneity issues found at the platform level. It should be able to provide remote access to services or other processes irrespective of the kind of platform in use. A remotely accessible middleware service usually includes a client part, which supports the service's API running in the application's address space, and a server part, which supports the service's main functions and may run in a different address space (i.e., on a different system) (Bernstein, 1996).

We consider using a class of distributed communication middleware that supports remote access to other services or processes to enable interoperating geographically dispersed processes. Another class of middleware we consider are those that support the client-server model. It should allow to have a process designated as a client and a server, this is the case a process requests (as a client) and responds (as a server) to a service.

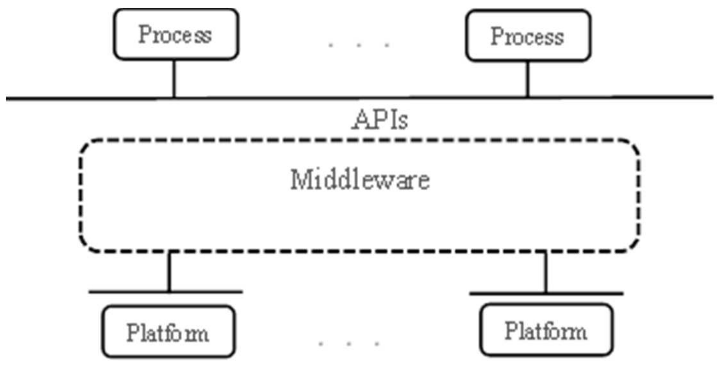


Figure 2-2: Middleware and surrounding context

(Krakowiak, 2007) proposes a global model for viewing the interactions between the client and the server. The server implements a service and registers it in a directory. The directory allows the client to lookup for available services then the client binds itself to the server to access it. In Figure 2-3, we show the abstract entities in this model. This global view is expressed with concrete representations in technologies such as Java Remote Method Invocation (RMI)

(Downing & Bryan, 1998), Common Object Request Broker Architecture (CORBA) (Object Management Group (OMG), n.d.) and Web Services (Booth et al., 2004).

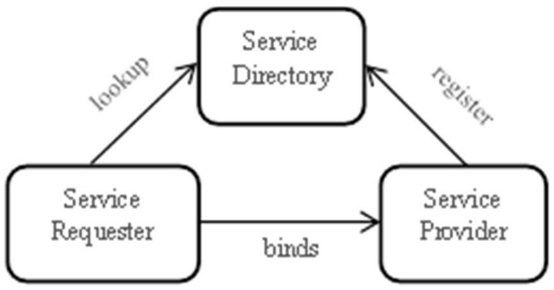


Figure 2-3: Abstract entities of service provisioning (Krakowiak, 2007)

Queries for services and responses need to be transmitted over the network in a serialized format suitable for transmission. They are transmitted in formats that are independent of programming languages and networking protocols an example format is the Extensible Markup Language (XML) for Web Services. Converting the queries and assembling them into a form suitable for transmission is called marshaling while conversion is called unmarshalling. There are different kinds of network protocols that can be used for this transmission they include; Internet Inter-ORB Protocol (IIOP) used by CORBA, Hypertext Transfer Protocol (HTTP) used by Web Services. The services need to be specified in a format that can support implementation in heterogeneous platforms, Interface Definition Language (IDL) is used by CORBA while Web Services make use of Web Service Definition Language (WSDL). Java RMI unlike CORBA and Web Services do not provide support for interoperability at the level of the programming language.

2.3.2.1. CORBA

Common Object Request Broker Architecture (CORBA) is an OMG standard (Object Management Group (OMG), n.d.) that can enable applications to invoke operations on distributed objects without concern for object location, programming language, operating system, communication protocols, interconnections, or hardware (Krishna, Schmidt, Klefstad, & Corsaro, 2005). CORBA enables remote objects to be accessed over a network. These objects are specified using the IDL and can share the same location with either a server. An object is implemented using any languages it supports such as Java, C++ and so on. Then, there is the Object Request Broker (ORB) that moderates interactions between the clients and servers using a communication protocol, typically Internet Inter-ORB Protocol (IIOP). For this access to be feasible, stub and skeleton objects are created on the client and server respectively. The client invokes the stub to create a request, the ORB forwards this request to the skeleton. On the server side, the targeted object is located and executed with the results sent back to the client. During the process of transmitting requests and results, the ORB marshalls the request and unmarshalls the results back to the client.

2.3.2.2. Web Services

Web services adhere to a collection of standards that allow them to be discovered and accessed over the Internet by client applications that follow those standards as well. These standards form the core of Web services architecture (see (Booth et al., 2004)), they include the Universal Description, Discovery and Integration standard (UDDI) (Booth et al., 2004), Web services Definition Language (WSDL) (Booth & Liu, 2007) and the Simple Object Access Protocol (SOAP) (Mitra & Lafon, 2007). The UDDI is a registry used for Web service publishing and discovery. WSDL is an XML based language used for describing available services. It contains information such as message types, signatures of operations, and a location of a service, for clients to consume the services. SOAP is an extensible framework for packaging and exchanging XML messages via network protocols such as HTTP, FTP, and so on.

In order for processes to communicate with each other, service provider (which can be a client/server process) describes a web service by the using the WSDL and publishes its availability using UDDI. The UDDI broadcasts the web service for service requestors, in this case client processes, to find the service. Once the service is found, the client then binds to the service provider. In the case of a successful bind, the processes can either request or provide services by using SOAP messages typically conveyed using network protocols.

2.3.3. DEVS PDES

We look at some implementation strategies that have attempted combining PDES with DEVS.

Himmelspach (Himmelspach, Ewald, Leye, & Uhrmacher, 2007) proposed a Parallel Sequential Simulator which was implemented to ease the distribution of DEVS models on several physical processors consequently introducing the need for partitioning and load balancing. In addition, it proposes performing Sequential and/or Parallel execution. Parallel variant of the CD++ (Ga Wainer, 2009) tool was designed to execute DEVS models on parallel memory architectures i.e. with the idea of distributing the simulating entities on different physical processors.

DEVS-Ada/TW (Christensen, 1990) is the first attempt to combine DEVS and Time Warp mechanism for Optimistic Distributed simulation. The DOHS (Distributed Optimistic Hierarchical Simulation) scheme proposed by (K. H. Kim, Seong, Kim, & Park, 1996) is a method of distributed simulation for hierarchical and modular DEVS models. It uses the Time Warp mechanism for global synchronization. Each node of the simulation tree structure is revised to adapt to a simulation parallel/distributed environment. P-CD++ (Liu & Wainer, 2007) is an optimistic version of the CD++ tool which was developed for the simulation of DEVS and Cell-DEVS models. Contrary to optimistic approaches, few parallel DEVS simulators belong to the conservative class. In (Bernard P. Zeigler et al., 2000), a distributed simulation framework (Conservative Parallel DEVS Simulator) is described for non-hierarchical DEVS models using conservative synchronization.

In the case of distributed simulation, DEVS/P2P (Cheon, Seo, Park, & Zeigler, 2004) makes use of JXTA middleware technology to handle communication and the sharing of resources between peers in a distributed and parallel computing environment. DEVS/GRID (Seo, Park, Kim, Cheon, & Zeigler, 2004) is an implementation for a grid computing environment that is used for DEVS modeling and simulation. It uses the Globus Toolkit as the middleware technology, which provides high performance computational resources. DEVS/CLUSTER (K. Kim & Kang, 2004) utilizes Common Object Request Broker Architecture (CORBA (Object Management Group (OMG), n.d.)) as a communication system which is designed to enable collaboration between heterogeneous platforms, different programming languages and operating systems. DEVS/RMI (M. Zhang, Zeigler, Hammonds, & Raj, 2006) is a distributed version of DEVSJAVA while its underlying middleware technology is Java RMI.

2.4. Model Driven Architecture (MDA)

PDES can benefit a lot from Model Driven Engineering (MDE) approaches, tools and methods; one of such approaches is the Model Driven Architecture (MDA) (J. Bézivin & Ploquin, 2001; OMG, 2014). It is an architecture-focused approach developed by OMG (Object Management Group, Version, Kennedy, Carter, & Technologies, 2003) with three major goals including model reusability, portability and interoperability (Hailpern & Tarr, 2006) as well as model continuity (Deniz Cetinkaya, Verbraeck, & Seck, 2015). It also uses the principle of separation of concerns to distinguish between the conceptualization, development, and implementation of models of a system. Models are representations of a system at an abstract level and form the core artifacts in the MDA. The MDA therefore focuses more on models and model transformations rather than on code consequently improving the communication between different participants (e.g., domain expert and modeler). In general, with MDA, it is possible to obtain a model that has been developed systematically, can be validated and is less sensitive to change in system requirements, platform or technology (Hailpern & Tarr, 2006). In addition, MDA simplifies model development process and reduce the amount of efforts needed for development (because it is an automated process).

2.4.1. MDA Model Abstraction Levels

MDA is a methodology based on a collection of standards that focuses on separating the business concerns from the platforms/technologies on which applications are deployed. The MDA proposes model abstraction at three levels; a Computation Independent Model (CIM) (corresponds to a view defined by a computation independent viewpoint) which defines the domain and the requirements of a system. It aims at elaborating high abstraction level model from a modeler's point of view through conceptualization. A Platform Independent Model (PIM) (Stahl & Voelter, 2006) (corresponds to a view defined by a platform independent viewpoint) describes the operations of a system independent of execution platforms. A Platform Description Model (PDM) (Object Management Group et al., 2003) (J. Bézivin & Ploquin, 2001) has been introduced to capture the descriptions about the platforms (e.g., hardware, software, etc.). A Platform Specific Model (PSM) (corresponds to a view defined by a platform specific viewpoint) describes the operations of a system as it uses a chosen set of execution platforms. MDA proposes that models are transformed from a CIM to a PIM, then this model is transformed by merging with information from the PDM with the help of one or more tools to PSM, and finally each PSM is transformed into code. It is also the case that a PIM may be transformed into one or more PIMs to be merged with a PDM. This type of transformation is known as model refinement or abstraction (Biehl, 2010). Furthermore, MDA incorporates the use of tools for the automation of models transformation (i.e., CIM to PIM, (PIM, PDM) to PSM, PSM to code (see Figure 2-4)).

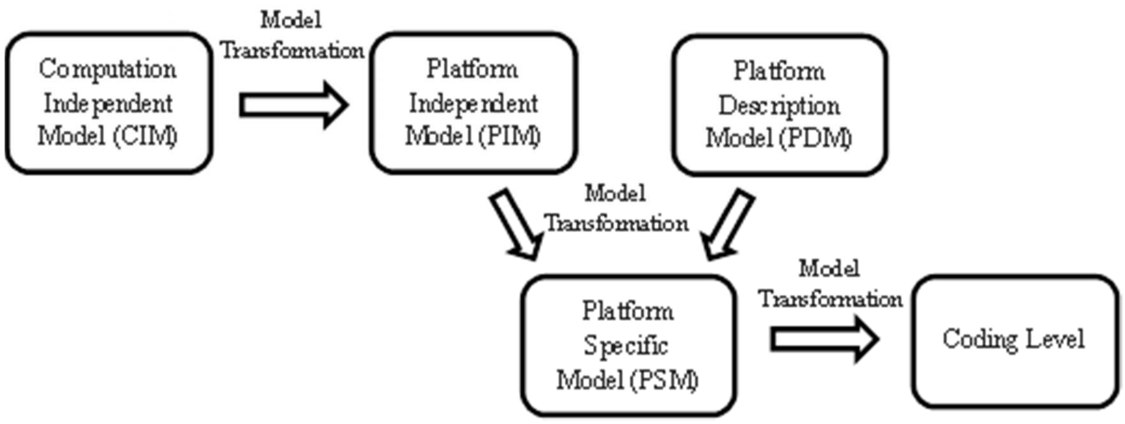

Computation Independent Model (CIM)
Model Transformation
Platform Independent Model (PIM)
Platform Description Model (PDM)
Model Transformation
Platform Specific Model (PSM)
Model Transformation
Coding Level

Figure 2-4: MDA Abstraction Levels (OMG, 2014)

2.4.2. Meta-modeling

Meta-modeling can be seen as the design process for developing meta-models. (Clark, Sammut, & Willans, 2008) defines a meta-model as a model of a language that captures its essential properties and features. These include the language concepts it supports, its textual and/or graphical syntax and its semantics (what the models and programs written in the language mean and how they behave). This meta-modeling language is described by a meta-meta-model. One important characteristic of meta-models is that they are seen as a model that can conform to its own meta-model (consequently a meta-model that can conform to its own meta-meta-model). In addition, a meta-meta-model conforms to a language whose abstract syntax it represents. This is relationship is described in Figure 2-5. The language can be specified using the Meta Object Facility (MOF) (Omg, 2006) or Unified Modeling Language (UML) (Rumbaugh , James; Jacobson, Ivar; Booch, 1999). Metamodels defined in this work have been specified using the Ecore meta-modeling language (of the Eclipse Modeling Framework (EMF) (Steinberg, Budinsky, Paternostro, Merks, & Paternostro, 2008)).

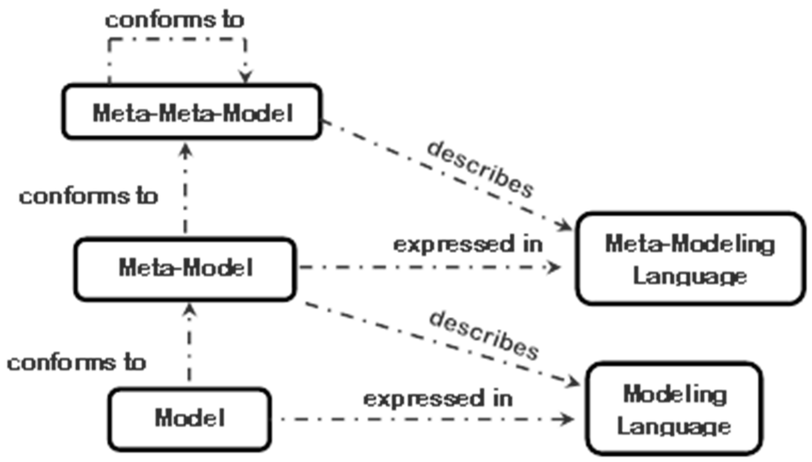


Figure 2-5: Relationship between Meta-models and modeling languages

2.4.3. Model Transformation

A model transformation process can be described as the process of creating one or several target models from one or several source models. The main objects in a transformation process are models, meta-models and transformation rules. Transformation rules are a set of operations or instructions applied on one or more source models so as to obtain one or more target models.

In the MDA approach, there is a series of model transformation techniques that can be applied on a model. These techniques are classified as model-to-model (M2M) which converts a source model to a target model and model-to-code

(M2T) which converts the source model to text or code. In order to transform models, it is necessary to express them in some modeling language (e.g., UML) whiles the syntax of the modeling language itself is expressed by a meta-model. Based on the language in which the source and target models of a transformation are expressed, it is possible for transformation to occur between models expressed in the same (respectively different) language also known as endogenous (respectively exogenous) transformation. Based on the model's abstraction level, it is possible to achieve transformations where source and target models reside on the same (respectively different) levels also known as horizontal (respectively vertical) transformation. The process of transformation can be either manual or automated. A more detailed survey on model transformation approaches and tools that can be used for manual or automated model transformation is presented in (K. Czarnecki & Helsen, 2006; Mens & Van Gorp, 2006).

One approach is the use of Extensible Stylesheet Language Transformations (XSLT) (WSC, n.d.) for M2M transformation. XSLT is a language for transforming XML documents into other XML documents as well XML into text/codes. It is very attractive for use with XML due to serialization of models using XMI. However there are claims that manual implementation of model transformations in XSLT quickly leads to non-maintainable implementations because of the verbosity and poor readability of XMI and XSLT (Boas, 2004; Krzysztof Czarnecki & Helsen, 2003).

Another considered tool for automating M2M transformation is the matured and popular ATLAS Transformation Language (ATL) (Jean Bézivin, Dupé, Jouault, Pitette, & Rougui, 2003; Jouaullt, 2009). It is integrated into Eclipse (Moore, Dean, Gerber, Wagenknecht, & Vanderheyden, 2004) through its EMF feature: for example, a debugger, an interpreter and syntax high-lighting editors are available. It is a hybrid language, declarative and imperative, that allows defining a model-to-model transformation (called Module) in the form of a set of mapped rules. It allows also defining model to text transformation (called Query). The transformation takes one or many source models (which can be defined with an Ecore meta-model (Moore et al., 2004)). It has been used in this work to support M2M transformation due to the availability of online support and documentations.

Automating M2T transformations involves producing models that are textually represented. One of such tools is the Acceleo Model to Text Language (Acceleo, 2015). Acceleo is a model-to-text transformation tool that can be used to generate source code from an instance model of a meta-model. Acceleo uses template based model transformation paradigm to implement transformation rules. Each rule in a template maps an element from the meta-model (and hence the instance model) to the text/code to be generated. A main feature of Acceleo is that static text can be mixed with Acceleo syntax when writing transformation rules.

2.4.4. MDA in Modeling and Simulation

MDA has been used in defining frameworks for different domains. An example is the Model Driven Service Engineering Architecture (MDSEA) (Bazoun, Zacharewicz, Ducq, & Boyé, 2014) which has been successfully applied to service system modeling and implementation. MDSEA provides an integrated methodology for dealing with modeling languages at various abstraction levels to support service model design and implementation. It offers to map to DEVS at the last MDA/MDI abstraction level (i.e., Technology Specific Model (TSM) that corresponds to the PSM abstract level in the MDA) for the modeling of business processes.

While some works do not consider the CIM level of the MDA proposal, some others do not explicitly give a conceptual model representation at the CIM level. (Tolk & Muguira, 2004) focuses on facilitating modeling and simulation by using MDA techniques. It deals with distributed simulation using DEVS however; it does not address the DEVS domain of the system at the CIM level. Rather it describes DEVS models by using UML at the PIM and maps it to HLA using MDA techniques unlike (Bernard P Zeigler, 1998) which maps DEVS directly to High Level Architecture (HLA). Differently, our work conceptually specifies DEVS models at the CIM level thereby proposing a unifying model for any kind of DEVS model.

Some works consider three MDA abstract levels in the modeling and simulation field. The Model Driven Development For Modeling And Simulation (MDD4MS) framework (Deniz Cetinkaya, Verbraeck, & Seck, 2011; Deniz Cetinkaya et al., 2015) uses BPMN modeling language for conceptual modeling at the CIM level, DEVS for the specification modeling at the PIM level and DEVSDSOL for the implementation at the PSM level. The framework also uses model transformation techniques to move between the MDA abstract levels. (Hu, Huang, Wu, Cao, & Chang, 2014) proposes a Model Driven Service Engineering (MDSE) methodology which is based on MDA to specify a unified model-driven design and validation approach to Service Oriented Architecture. This work specifies the use of UML/BPMN at CIM level, DEVS model specification at the PIM level and DEVS simulation at the PSM level.

Some works rather consider the use of MDA techniques from the view of model transformation techniques and languages for which models can be formulated. With model transformation techniques, it is possible to ensure model continuity from one abstract level to another in the MDA. (Janoušek, Polášek, & Slavíček, 2006; Mittal, Risco-Martín, & Zeigler, 2007) propose the use of XML formats to represent DEVS models thereby contributing towards the transformations of Platform Specific Models (PSMs) to Platform Independent Models (PIMs). (De Lara & Vangheluwe, 2002) propose a tool for meta-modeling and model transformations for simulation, namely AToM3, that combines the use of multi-formalism modeling and meta-modeling to facilitate computer assisted modeling of large systems. (Gonzalez, Luna, Cuello, Perez, & Daniele, 2015) proposes the automatic model transformation of UML state machines to DEVS models by using MDA techniques. (Mittal et al., 2009) gave ways to automatically generate DEVS models from various types of business process specifications and realize distributed simulation execution using web services. (D'Ambrogio et al., 2010) proposes the automatic development of DEVS/SOA simulations from the UML specification of DEVS models by using MDA techniques. In its present form, the behaviors of the atomic models are not implemented and the approach only produces the core skeleton of Java classes that implement DEVS models. A state of the art of the MDA contributions to DEVS formalism including a comparative study of the most recent ones is given in (Garredu, Vittori, Santucci, & Poggi, 2014).

2.5. Conclusion

In this chapter, we presented a literature survey of the context of this research. This forms the foundation of our work and provides us building blocks with which to develop the unifying framework. The relevant concepts were discussed based on the reviewed literature.

First, there was a review of DEVS, PDES and MDA concepts; DEVS for its provision of a modeling and simulation framework, PDES for its provision of facilities for achieving parallel/distributed simulation and MDA for its provision of automation facilities. Using Figure 1-2, DEVS modeling is captured at the modeling layer, DEVS simulation protocol is captured at the structural layer, PDES components are integrated at the architectural layer while a combination of DEVS and PDES facilities are captured at the code layer. The tool support layer makes use of MDA techniques and tools to provide automation support for each layer of the unifying framework. Finally, a literature review of the application of these concepts was provided.

In Chapter 3, we will provide contributions to knowledge of the development of DEVS PDES implementation strategies. In a subsequent chapter, we develop on this by proposing formal specifications, a guideline, and methodology with which a DEVS PDES strategy can be built.

3. A conceptual approach for the understanding of DEVS PDES strategies

3.1. Introduction

From an extensive review of literature, we see varying opinions in how simulation practitioners choose to approach the development of a DEVS PDES implementation strategy. Subsequently, in this work, we refer to DEVS PDES implementation strategy as an implementation of DEVS simulation protocol. In this chapter, we present a discussion on these approaches to guide the understanding of the process of building a DEVS PDES implementation strategy. This discussion forms the basis of this work, i.e. the proposal of the unifying framework presented in Figure 1-2. We start from a message exchange perspective of the DEVS simulation protocol to show an identified pattern in the specification of DEVS simulation algorithms. Then we examine how the hierarchical structure of simulation components can be altered, grouped or distributed. First, we consider that some approaches prefer to alter the structure of a DEVS model or a DEVS Simulation Tree (ST) structure. Alteration of a DEVS model structure has been mentioned, we will however focus on alteration of the ST structure (Jafer & Wainer, 2009; Kihyung Kim, Wonseok Kang, Bong Sagong, & Hyungon Seo, 2000; K. H. Kim et al., 1996; Ga Wainer, 2009) in this chapter and subsequently and we call this approach ‘‘Tree Transformation’’. Secondly, some approaches split the ST into sub-trees (Ezequiel Glinsky & Wainer, 2006; Bernard P. Zeigler et al., 2000). We call this ‘‘Tree Splitting’’. Also, some approaches differ on the number of executions/processes that can be performed per simulation run (Himmelspach & Uhrmacher, 2006). We call this ‘‘Node Clustering’’. Lastly, some approaches vary the number of computing resources to be used during simulation (K. Kim & Kang, 2004; Bernard P. Zeigler et al., 2000). We call this ‘‘Process Distribution’’.

We briefly introduce the key concepts used in this chapter and subsequent ones:

- Root Coordinator (RC): the simulating element that manages the time of a DEVS Simulation Tree (ST).
- Nodes: the simulation elements used for executing DEVS models. These nodes are Coordinators, Simulators, and RCs. The RC has an event loop that sends event messages and controls the simulation cycles while the Coordinator and Simulator are capable of receiving, treating and sending event messages.
- DEVS Simulation Tree (ST): a ST is made up of nodes we sometime refer to it as tree. The RC is always at the top of the tree's hierarchy and has a Coordinator as its descendant. In addition, the Coordinator has either a Coordinator or Simulator as its descendant but the Simulator has none.
- Process: we define this as a stream of execution. It contains two types of nodes during execution: they are active and passive nodes. An active node is anode that is currently active in an execution stream (e.g., Java threads or Ada tasks). While a passive node is part of an execution stream, it is 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. In addition, there can be more than one passive node in a process.
- Activity: a set of actions that is performed at the receipt of an event.
- Processor: a computing resource that allows the execution of a process on itself.
- Simulation Graph (SG): a representation of the relationship between the identified aspects in DEVS simulation. An example of a SG can be seen in

Figure 3. Details about its components are discussed in the following sections.

These concepts are illustrated in Figure 3-1.

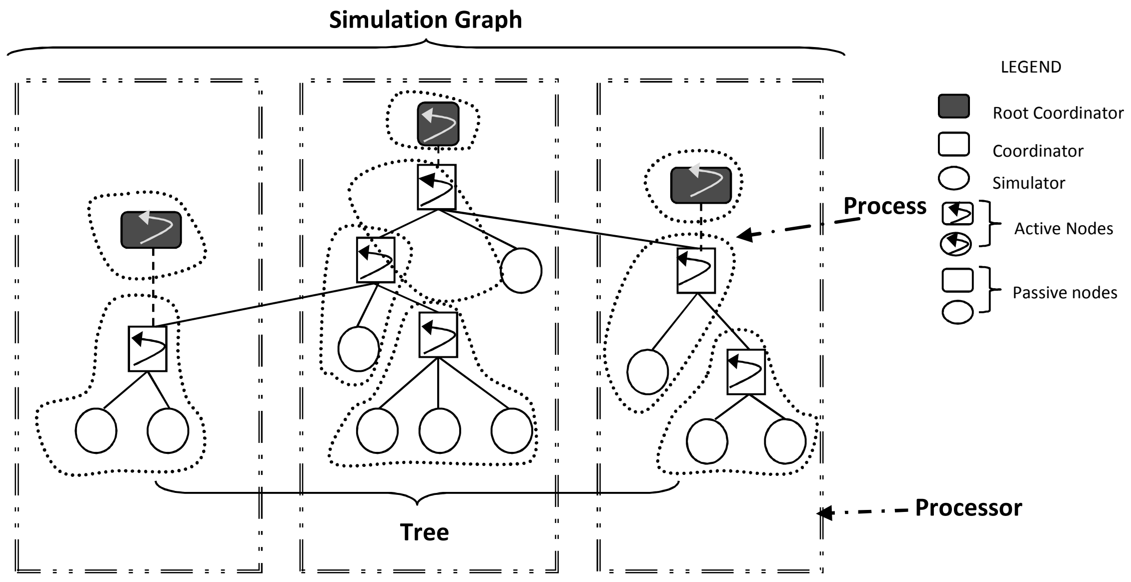


Figure 3-1: Relationship between Trees, Processes and Processors

3.2. A Pattern as seen in DEVS Simulation Protocol

There exist various DEVS simulation algorithms in literature; they are designed to capture communication on a simulation tree for reasons such as improving time efficiency, optimizing execution and distributing models. Three major features (B. Zeigler & Muzy, 2017) expressed by these variants include time management, data exchange management and state update management. In this section, we propose a unified way to describe these variants based on the data exchange management feature.

DEVS simulation protocol consists of nodes that exchange messages and perform some actions. Two key pieces of information carried by these messages are their category and time stamp. The category of a message is associated to a certain form of treatment the receiver component (Coordinator or Simulator) must perform while the time stamp indicates the simulation time this message has been generated. In CDEVS (Bernard P. Zeigler et al., 2000), categories are *, i, x and y. In a version of PDEVS (Chow & Zeigler, 1994), categories are *, i, q, done, @ and y. The subsequent versions propose various sets of categories (with associated sets of treatments). However, all DEVS-based algorithms adhere to the same simulation protocol (i.e., generalized or specialized Coordinators and Simulators exchanging messages and performing specific actions on receipt of specific categories of message).

From this adherence, a send – receive message pattern can be seen in these algorithms; be it in sequential, parallel or distributed DEVS simulation algorithms. We use concept of "send-receive message" pattern as a basis for unifying various forms of DEVS algorithms. This pattern is illustrated in Table 3-1 where we examined literature and extracted the message pattern for each described node. We see this pattern as being common among all these variants and if more algorithms are to be examined, they will follow this pattern. It is also possible for this pattern to accommodate a differently specified message, for example, the "%" message used in DEVSP2P (Park et al., 2007). This form of pattern can support the standardization and integration of heterogeneous DEVS algorithm specifications. We have included in Appendix A the simulation algorithm for PDEVS (Bernard P. Zeigler et al., 2000) in details showing how each node conforms to the send-receive message pattern when exchanging events. By using the send-receive message pattern new nodes can be defined.

Table 3-1: Send – Receive message pattern in DEVS simulation Protocol

Literature	**Node**	**Send**	**Receive**
CDEVS and PDEVS by (Bernard P. Zeigler et al., 2000)	Simulator	y	i, *, x
	Coordinator	i, *, x, y	i, *, x, y
	Root Coordinator	i, *	

PDEVS (Chow & Zeigler, 1994)	Simulator	y, done	@, q, *,
	Coordinator	@, done, q, y, *	@, y, q, *, done
	Root Coordinator	@, *	done
Risk-free optimistic Simulator (Bernard P. Zeigler et al., 2000)	Optimistic Root Coordinator	i, *, y, rollback, x	y, x
Time-warp DEVS simulator (Bernard P. Zeigler et al., 2000)	Optimistic Simulator	i, x, *, y	i, x, *, rollback, y
	Optimistic Coordinator	rollback	rollback
	Optimistic Root Coordinator	i, x, *, y, rollback, anti-message	y, x, anti-message
Conservative parallel DEVS simulator (Bernard P. Zeigler et al., 2000)	Conservative Simulator	i, y, *, x	i, x, y
Optimistic DOHS Scheme (K. H. Kim et al., 1996)	p-simulator	y, done	*, x
	p-coordinator	y, done	*, x, y, done, flush, collect
DEVSP2P (Park et al., 2007)	Atomic Simulator	%, y, done	%, @, x, *
	Coupled Simulator	%, y, done	%, @, x, *
	Peer	%, @, y, x, *	done, x
PCD++ (Ezequiel Glinsky, 2005; Ezequiel Glinsky & Wainer, 2006; Liu & Wainer, 2007)	Simulator	init, @, q, *	done, y,
	Flat Coordinator	init, @, y, q, *	init, done, @, y, q, *
	Node Coordinator	init, done, q, y,	init, q, @, *, y
	Root Coordinator	init, q, y	q, y

3.3. Hierarchical Simulation Component Transformation

Various studies have shown that altering the hierarchical structure of simulation components can improve simulation efficiency (Franceschini & Bisgambiglia, 2014; Lee & Kim, 2003), reduce the cost of event routing (E Glinsky & Wainer, 2002; Kihyung Kim et al., 2000) and support the inclusion of synchronization mechanisms so as to achieve parallel/distributed simulation. The transformation process is by either increasing the number of components in the hierarchy or reducing them.

3.3.1. Reduction

Reduction also known as flattening is the process of reducing the number of models or nodes on a DEVS model or simulation tree. It is a process used for improving simulation performance and reducing communication overhead during simulation execution. From literature two flattening methods can be found; one is direct connection (Bae, Bae, Moon, & Kim, 2016; Chen & Vangheluwe, 2010) and the other is composition (Lee & Kim, 2003). When applying direct connection method to flatten a simulator, one coordinator manages all the nodes. It serves as the top-most coordinator positioned just below the root and gives an execution order to simulators. The method recursively traverses all nodes on the hierarchical simulation tree while copying simulators and the relationship between them to the top-most coordinator. The composition method of flattening, when applied on models, would recursively merge all models belonging to a coupled model until the top-most coupled model is reached. By applying the coupling under closure property of DEVS models this coupled model is transformed into an atomic model.

DEVS framework provides a flexible foundation by separating models and simulators. In keeping with this philosophy, it is possible to flatten either the model or the simulator. In the following sections we discuss flattening as it applies to the model and the simulator.

3.3.1.1. Model Flattening

The closure under coupling property (Bernard P. Zeigler et al., 2000) of DEVS models make it easy to apply the flattening process on models because it shows that any coupled DEVS model can offer the same behavior as a resultant atomic model. With this property, one can easily remove coupled models in the hierarchy. Typically, a flattened model simplifies the hierarchical model while keeping a hierarchical simulator structure. (Chen & Vangheluwe, 2010) proposed to symbolically transform a hierarchical coupled model into a coupled model with one-level depth to simplify the complexity and reduce the redundancy in DEVS models. (Zacharewicz & Hamri, 2007) proposed transforming the hierarchical structure of a DEVS model by reducing the number of coupled models to one to which atomic models will be connected as direct successors. (Bae et al., 2016) proposed a flattening transformation for DEVS models whose coupling structure changes during simulation execution. In the case whereby the hierarchical model structure is flattened, ports and coupling information need to be modified. This can be resolved by copying them into the top parent coordinator (Chen & Vangheluwe, 2010; Zacharewicz & Hamri, 2007) or by storing the information in the simulators as state variables (Franceschini & Bisgambiglia, 2014). Extra data structures will then be required to achieve this.

3.3.1.2. Simulator Flattening

CD++ uses a flat simulation approach that eliminates the need for intermediate coordinators to improve the performance of simulation (Ga Wainer, 2009; Gabriel Wainer, 2002) (see Figure 3-2). (Franceschini & Bisgambiglia, 2014) used flattening methods to reduce the number of messages exchanged between simulation nodes while (Muzy & Nutaro, 2005) used flattening to reduce the cost of event scheduling in nodes. In the case whereby the simulator is flattened and the hierarchical DEVS model structure is maintained, the information about the model structure is stored to be explored during simulation execution. The process of transforming the simulator by flattening involves reducing the number of nodes and consequently the number of messages being exchanged is reduced. This improves the manageability of the nodes in the simulator.

However, in the case of simulating a complex model structure, a flattened simulator will require a deep-tree search algorithm to retrieve model information. This will slow down simulation execution.

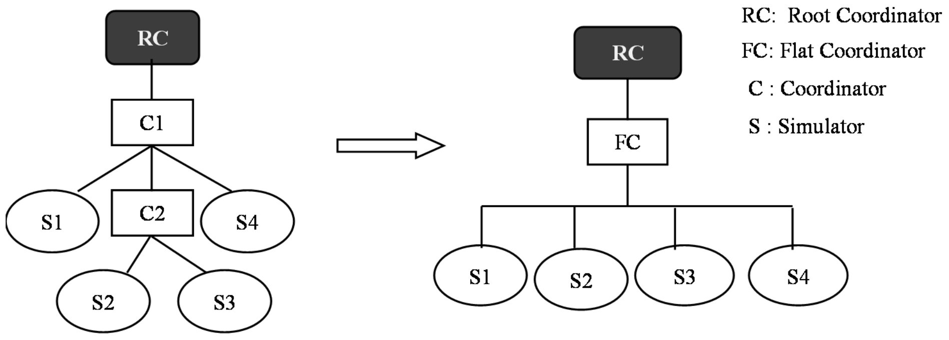


Figure 3-2: Tree Transformation by Reduction

3.3.2. Tree Expansion

Himmelspach et al. (Himmelspach et al., 2007) achieved the expansion by introducing new simulation nodes into the ST structure (see Figure 3-3. 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. Communication between these nodes constitutes an increasing overhead cost as the structure of the messages exchanged maybe altered to accommodate extra information. For example, a new sub coordinator has no coupled model associated with it and therefore contains no coupling information in addition; it has to correctly identify imminent models and influences. One way to deal with this is through the composition of messages, that is, by including more information in a message's construct, as seen in Himmelspach et al. (Himmelspach et al., 2007).

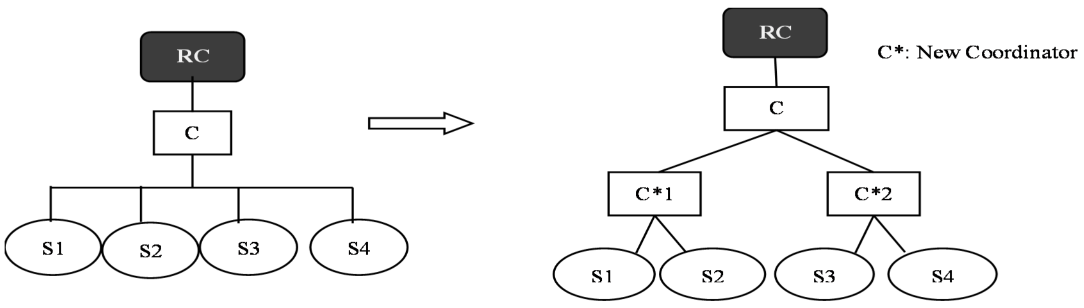


Figure 3-3: Tree Transformation by Expansion

3.4. Tree Splitting

Tree splitting refers to 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 is split, executed, or how they can be mapped to available number of processors.

3.4.1. Single Tree Structure

Executing a model with a single-tree structure simulator can be expressed as having an entire tree simulated with the use of a central scheduler called the Root Coordinator (RC). It is applicable to sequential simulation, which is the simplest form of simulation however; it does not properly reflect the simultaneous occurrence of events in the modeled system.

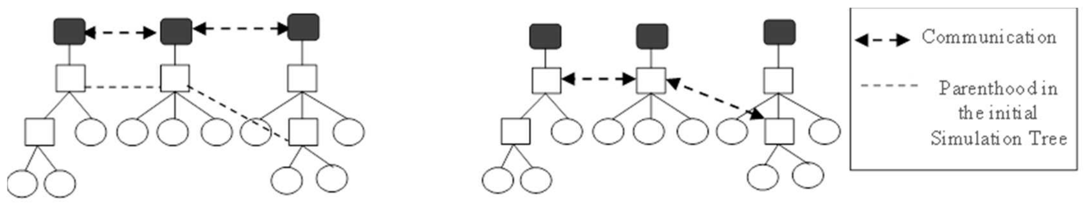
Single-tree structures are mostly implemented using CDEVS and PDEVS algorithms. In CDEVS (Bernard P. Zeigler et al., 2000) events are processed in a sequence however serialization reduces possible utilization of parallelism during the occurrence of events. On the other hand, Chow and Zeigler (Chow & Zeigler, 1994) 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 is exploited in parallel and distributed simulation. It beats the restrictions in CDEVS in both execution time and memory usage.

3.4.2. Multiple Trees

In the case of distributed simulation, it is necessary to split the simulation tree. A tree is split into different sub-trees with each having its own central scheduler/RC and different simulation clocks. Based on this structure, all events with the same time stamp can be processed simultaneously.

Additionally, it is possible to perform tree transformation operation on the sub-trees of a split tree. The expansion operation can be used to introduce synchronization mechanisms into the sub-trees as nodes and as such these nodes do not carry the same kind of information as the existing ones on the sub-trees (see Conservative DEVS Simulator (Bernard P. Zeigler et al., 2000)).

Distributed simulation algorithms are used to synchronize trees. The two basic distributed simulation algorithms in use are the Optimistic (Jefferson, 1985) and Conservative (Pessimistic) (Bryant, 1977; Chandy & Misra, 1979; Misra, 1986) algorithms. Optimistic algorithms, in contrast to Conservative algorithms, enable increased degrees of parallelism. However, they also result in more algorithms that are complex. In PCD++ (Ezequiel Glinsky & Wainer, 2006; Liu & Wainer, 2007) the inclusion of Node Coordinators (NCs) on the sub-trees was to done enable optimistic synchronization and communication in a distributed environment. Communication between these sub-trees is usually between; the RCs (see Figure 3-4a), for example, DEVS Time Warp (TW) (Bernard P. Zeigler et al., 2000); Coordinators on sub-trees (see Figure 3-4b), for example, Distributed Optimistic Hierarchical Simulation (DOHS) scheme (K. H. Kim et al., 1996) and the newly added nodes (Conservative DEVS Simulator (Bernard P. Zeigler et al., 2000)).



a) Communication between Root Coordinators b) Communication between Coordinators

Figure 3-4: Communication schemes between tree structures

3.5. Node Clustering

This is the association of one or many nodes to various processes. Processes drive events execution. We 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 or “many processes” execution. However, we will not be dealing with how execution takes place on processors.

3.5.1. One Process

A one-process execution denotes having events processed in a serial and orderly manner, i.e., one after another. This restricts concurrent execution streams. In (Himmelspach & Uhrmacher, 2006) the authors denote this form of execution as “sequentialization”. In this sense, for example, the “main” program is a process. 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.

3.5.2. Multiple Processes

In the case of many processes, execution of events can be split into several logical processes (autonomous tasks) for concurrent processing. Examples of such processes are 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++ implementation utilizes many processes for model execution (Jafer & Wainer, 2010).

3.6. Process Distribution

Process Distribution can be referred to as the allocation of one or many processes to available number of processors. 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 two distinct classes: “one-processor” or “many-processors”.

3.6.1. 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.

3.6.2. 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 may be achieved through interoperability mechanisms (e.g., CORBA (Object Management Group (OMG), n.d.), or more recently Web Services (Booth et al., 2004)).

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 (Park, Hunt, & Zeigler, 2006). It uses cost analysis methodology to construct partition blocks, although it makes an effort to guarantee an incremental quality of partitioning. However, it is restricted only to models from which cost analysis can be extracted and processed.

3.7. Conclusion

This chapter is an improvement of the work already started in (Adegoke, Togo, & Traore, 2013). We examined the various approaches, as seen in literature that can be used in the development of DEVS PDES implementation strategies. This effort was aimed at contributing to a better understanding of the components and procedures involved in a DEVS PDES development process.

We first considered the development process from the different ways DEVS simulation algorithms are specified in literature. It was viewed from the data exchange perspective then a send-receive message pattern was proposed. Other approaches from the view of the DEVS ST, processes, processors were examined. Insights was given as to how the structure of the ST can be modified, how the nodes on an ST can be grouped together into process and mapped on processors.

In Chapter 4, we further examine some of the concepts introduced in this chapter to provide a deeper understanding of the process of building a DEVS PDES implementation strategy. These concepts will be formally described and a guideline for building a strategy will be proposed.

4. From Simulation Tree to Simulation Graph

4.1. Introduction

In chapter 3, we observed that there are many DEVS PDES strategies in existence and that there are some elements commonly used in these strategies. These elements include; Tree, Process and Processor. From the integration of these elements, one can obtain a set of Simulation Structures. The integration process also requires that a set of Simulation Operations be performed on the structures in order to obtain a new structure. In this chapter, we examine the Simulation Structures, Simulation Operations and their formalization. Numerous formal definitions are given for the following reasons:

- They provide a clear understanding (by reducing ambiguity) of simulation structures and operations we introduce.
- They provide mathematical objects that one can use for analyzing a strategy
- To enable consistency checking or validity checking.
- To ease the automation of processes defined with them.

In addition, we examine a methodology by which a DEVS PDES strategy can be obtained and illustrate it with a sample case study.

4.2. Simulation Structures

We define Simulation Structures as conceptual components used in building a DEVS PDES strategy.

4.2.1. Simulation Tree

Simulation Tree (ST) is the basic building block of a strategy. It consists of nodes and the relationship between them. A ST is obtained after DEVS operational semantics mapping have been applied on DEVS model specifications. Structurally it consists of one Root Coordinator, several

Coordinators and Simulators. See Figure 4-1. We provide two formal definitions for specifying a ST and this is for convenience sake.

Definition 1: *Simulation Tree* $T = \langle R, N, f \rangle$

With:

- $R \in N$
- $f: N \rightarrow \wp(N)$ *where* $\wp(N)$ *is the Power Set of N*
- $f^{1}(R) = \varnothing$
- $f^{1}(J) \neq \varnothing, \ \forall J \in N - \{R\}$
- $f^{-1}(n) \neq \{n\}, \ \forall n \in N$
- $f^{-1}(n)| = 1, \ \forall n \in N - \{R\}$
- $f(n) \neq \varnothing, \ \forall n \in (N\text{-}\{Simulators\})$

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 (the one at one step higher in the hierarchy).

For example, the tree given in Figure 2-1b is defined as T = <R, {R, C1, C2, S1, S2, S3}, f>, where f(R) = {C1}, f(C1) = {C2, S1}, f(C2) = {S2, S3}, and f(S1) = f(S2) = f(S3) = ∅

Definition 2*: Simulation Tree can also be defined as* $T = \langle R, N, F \rangle$

With:

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

Using Definition 2 for the example of Figure 2.b, R and N will be defined as same while F will be {(R, C1), (C1, C2), (C1, S1), (C2, S2), (C2, S3)}

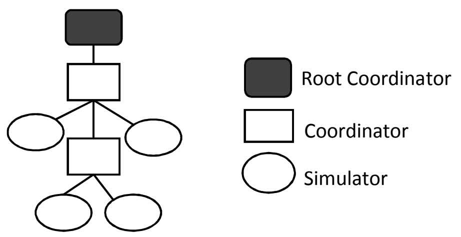


Figure 4-1: Simulation Tree

4.2.2. Simulation Skeleton

Simulation Skeleton (SS): is the structure of the simulation protocol that can fit the PDES scheme. It is obtained by using operations that analyzes a ST and may consist of one or more Root Coordinators thereby decomposing the ST into multiple trees (Ezequiel Glinsky & Wainer, 2006; Liu & Wainer, 2007).

Definition 3*: A Simulation Skeleton is formally defined as*

$S = \langle \{R_i\}, N, f \rangle$

Where

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

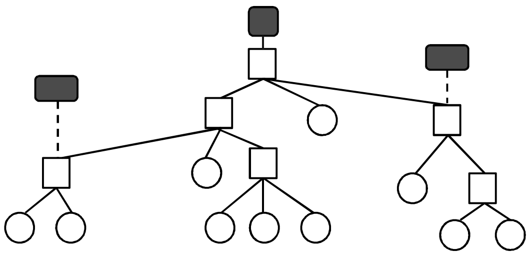


Figure 4-2: Simulation Skeleton

4.2.3. Simulation Bundle

Simulation Bundle (SB): is obtained after operations, which takes each of the nodes on a SS and maps them on the available number of processes. See Figure 4-3.

Definition 4*: A Simulation Bundle is formally defined as*

$B = \langle \{R_i\}, N, f, Ps, Cluster \rangle$

Where

- $\langle \{R_i\}, N, f \rangle$ *is a skeleton*
- *Ps is the set of Processes*
- *Cluster (see Definition 8)*

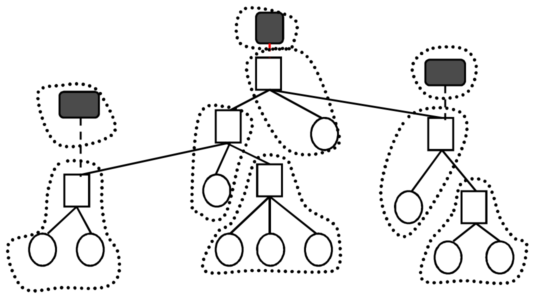


Figure 4-3: Simulation Bundle

4.2.4. Simulation Graph

Simulation Graph (SG): depicts the relationship between main components of a DEVS PDES implementation strategy i.e., the DEVS tree, Processes and Processors. It is obtained by taking the cluster of nodes in SB and mapping them to processors. See Figure 4-4

***Definition 5**: A Simulation Graph is formally defined as*

SG = <{R_i}, N, f, Ps, Pr, Cluster, Distrib >

Where

- *{R_i}, N, f, Ps, Cluster> is a Simulation Bundle*
- *Pr is the set of Processors*
- *Distrib (see Definition 9)*

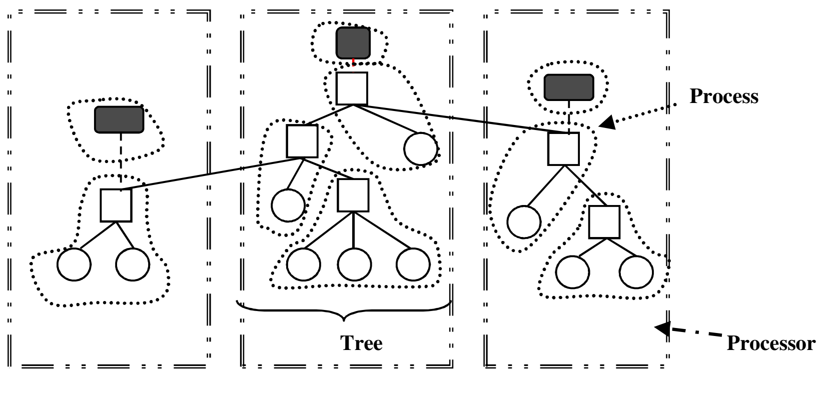


Figure 4-4: Simulation Graph

4.3. Simulation Structure Operations

Simulation Operations are generic functions that are used in mapping elements of a simulation structure to another thereby realizing a structure. These operations are:

- Transform: This is a generic operation takes a ST and alters it by reducing and maybe increasing the number of nodes on the ST. This alteration is made on the number of available nodes (not including the Root Coordinators) on the Tree and their relationships. It is important to note that this operation can also be performed on the SS, SB and SG.

Definition 6:

Tree Transformation is formally defined as

$Transf[Na,Nr,Fa,Fr]: \tau \rightarrow \tau$ *where* τ *is the set of all possible STs*

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

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:

- $Na \cap N \wedge (Fa \cap F) = \varnothing$ *(new nodes and relationships should not belong to the old tree)*
- $Nr \subset N - \{R\}$ *(only nodes of the old tree can be removed except for Root)*
- $Fa \subset (N \times Na) \cup Na^2 \cup Na \times (N-\{R\} \cup (\{R\} \times Na))$ *(a new parenthood must exist either between an old and a new node or between two new nodes or between a new and an old node or between an existing root and a new node)*
- $Fr \subseteq F$ *(only parenthood of the old tree can be removed)*

- Split: is a generic operation that is used for creating a partition of nodes from a ST.

***Definition 7**: Formally Tree Splitting is defined as*

$Split: \tau \rightarrow \Sigma$

$Split(<R, N, f>) = <\{R_i\}, N', f'>$

Where

- *Σ is the set of all SSs (see the definition of this structure in definition 3)*
- $R \in \{R_i\}$
- $N' = N \cup \{R_i\}$
- $f'_{/N} = f$

- Cluster: A generic operation takes the available number of nodes and associates them with *Processes.*

***Definition 8**: Node Clustering is formally defined as*

$Cluster: N \rightarrow Ps$

Where

- *Ps is the set of Processes.*
- *$(Cluster)^{-1}(p)$ is Connex $\forall p \in Ps$*
- $\forall p_i, p_j \in Ps, p_i \neq p_j, Cluster^{-1}(p_i) \cap Cluster^{-1}(p_j) = \emptyset$

- Distrib: A generic operation that takes the set of available Processes and plots them onto the set of available Processors.

***Definition 9**: Process Distribution is defined as*

$Distrib: Ps \rightarrow Pr$

Where

- *Pr is the set of Processors*
- $\forall p_i, p_j \in Pr, p_i \neq p_j, Distrib^{-1}(p_i) \cap Distrib^{-1}(p_j) = \emptyset$

4.4. Methodology

The adapted methodology describes a structural and behavioral view for the construction of any Simulation Structure. It consists of performing simulation operations on simulation structures. A state chart has been used to present the trajectories that describe the set of all possible paths that can be taken during the construction of the structures. Any user-defined Simulation Structure construction process is described as an instantiation of the state chart given in Figure 4-5, driven by the analysis of the initial ST and the available number of Processes and Processors. The methodology allows iterating on each state until some user-defined satisfaction criteria are reached (optimal splitting, optimal clustering, and optimal distribution, which we specify in Figure 4-5 respectively as [Split is Optimal], [Cluster is Optimal] and [Distrib is Optimal]). However, this methodology does not restrict the use of any algorithm or technique for the SPLIT, CLUSTER or DISTRIB operations. Therefore, the use of any efficient model-partitioning algorithm, clustering techniques or mapping heuristics is permitted as long as user-satisfaction criteria are met. The use of this methodology is illustrated with a case study in Section 4.6

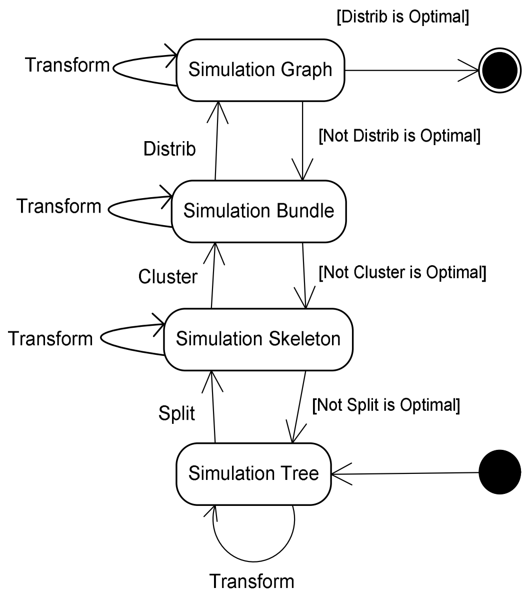


Figure 4-5: Simulation Graph Methodology

4.5. Layered Approach

The methodology is modular as it uses the layered approach to separate functionalities and structures used in a DEVS PDES implementation strategy into layers. Each layer is a simulation structure implemented using simulation operations. The main advantage of the layered approach is simplicity of simulation structure construction, verification, and debugging. Once the first layer is free from errors, its correct functioning can be assumed while the second layer is being checked for errors, and so on. If an error is found in a particular layer, the error must be on that layer, because the layers below it are already free from errors. Thus, the design and implementation of the structures is simplified. The layers are: ST layer, SS Layer, SB Layer and SG Layer.

To modify the SG 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 ST before making the necessary modifications in the simulation structure (see Figure 4-6). Such disciplined approach makes it easy to avoid mistakes and as such increases the flexibility of the methodology.

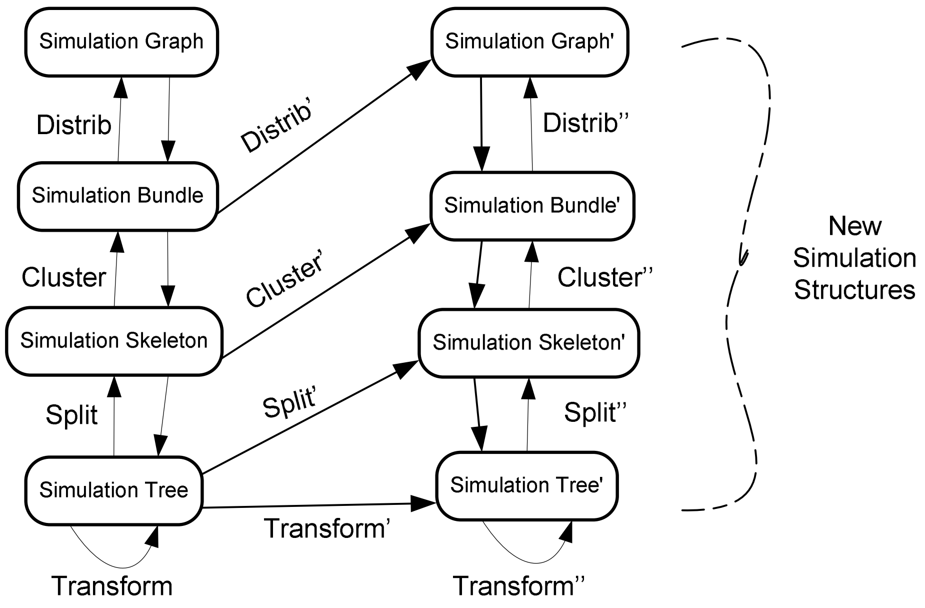


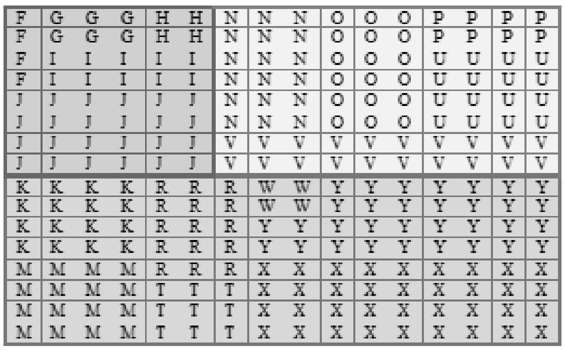
Figure 4-6: Flexibility of the layered approach

4.6. A Case Study on the use of the Simulation Graph Methodology

To illustrate the use of our methodology, we consider an existing work from literature as an example. PCD++ (Ezequiel Glinsky & Wainer, 2006; Liu &

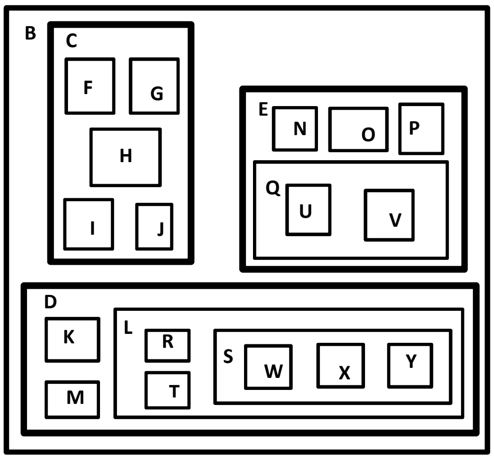
Wainer, 2007) consists of simulation nodes which are concrete implementations of abstract DEVS simulators. We describe how an instantiation of our generic approach can provide a formal guideline for the process of mapping a DEVS simulation protocol onto the PCD++ strategy.

We consider a fire propagation model (n x n cells) as the one presented in (Ezequiel Glinsky & Wainer, 2006; Liu & Wainer, 2007) and the distributed simulation strategy introduced as well. We assume the initial DEVS model (without the couplings) and DEVS tree structure in use are as shown in Figure 4-7.

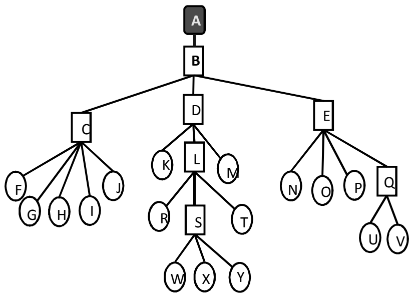


F	G	G	G	H	H	N	N	N	O	O	O	P	P	P	P
F	G	G	G	H	H	N	N	N	O	O	O	P	P	P	P
F	I	I	I	I	I	N	N	N	O	O	O	U	U	U	U
F	I	I	I	I	I	N	N	N	O	O	O	U	U	U	U
J	J	J	J	J	J	N	N	N	O	O	O	U	U	U	U
J	J	J	J	J	J	N	N	N	O	O	O	U	U	U	U
J	J	J	J	J	J	V	V	V	V	V	V	V	V	V	V
J	J	J	J	J	J	V	V	V	V	V	V	V	V	V	V
K	K	K	K	R	R	R	W	W	Y	Y	Y	Y	Y	Y	Y
K	K	K	K	R	R	R	W	W	Y	Y	Y	Y	Y	Y	Y
K	K	K	K	R	R	R	Y	Y	Y	Y	Y	Y	Y	Y	Y
K	K	K	K	R	R	R	Y	Y	Y	Y	Y	Y	Y	Y	Y
M	M	M	M	R	R	R	X	X	X	X	X	X	X	X	X
M	M	M	M	T	T	T	X	X	X	X	X	X	X	X	X
M	M	M	M	T	T	T	X	X	X	X	X	X	X	X	X
M	M	M	M	T	T	T	X	X	X	X	X	X	X	X	X

(a) A n × n cells model (the letter in a cell indicates the atomic model to which belongs the cell)



(b) A DEVS representation of a n x n cells model whose cells are shared among various atomic models then later grouped in coupled models



(c) DEVS ST for the n × n cells model

Using Definition 2, the initial tree is: T = <A, N, F> with
N = {A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y}
F = {(A,B), (B,C), (B,D), (B,E), (C,F), (C,G), (C,H), (C,I), (C,J), (D,K), (D,L), (D,M), (L,R), (L,S), (L,T), (S,W), (S,X) (S,Y), (E,N), (E,O), (E,P), (E,Q), (Q,U), (Q,V)}

(d) Formal Specifications of the ST

Figure 4-7: Initial DEVS nXn cells model, its ST structure and formal definition

In PCD++, we see that the first operation is to reduce (transform) the number of simulation nodes on the tree hence the introduction of Flat Coordinators (FC). Each FC synchronizes its child simulators, routes messages among them and is in charge of intra-process communication between its children. The approach of reducing the simulation nodes was used to achieve improved simulation performance. At the end of this operation, the resultant SiS is also a ST. The formal definition of the transformation done includes the formal algorithms of the nodes involved in both sides of the transformation (i.e., initial DEVS algorithms for coordinators at one side, and new algorithms for FCs at the other side). Figure 4-8presents the resulting tree.

Using Definition 3, the *Transform* operation is

Transf[{FC1,FC2,FC3}, {B,C,D L,S,E,Q}, Fa, Fr](T) = T' with

Fa = {(A,FC1), (FC1,FC2), (FC1,FC3), (FC2,K), (FC2,R), (FC2,W), (FC2,X), (FC2,X), (FC2,Y), (FC2,T), (FC2,M), (FC1,F), (FC1,G), (FC1,H), (FC1,I), (FC1,J), (FC3,N), (FC3,O), (FC3,P), (FC3,U), (FC3,V)}

Fr = {(A,B), (B,C), (B,D), (B,E), (C,F), (C,G), (C,H), (C,I), (C,J), (D,K), (D,L), (D,M), (L,R), (L,S), (L,T), (S,W), (S,X) (S,Y), (E,N), (E,O), (E,P), (E,Q), (Q,U), (Q,V)}

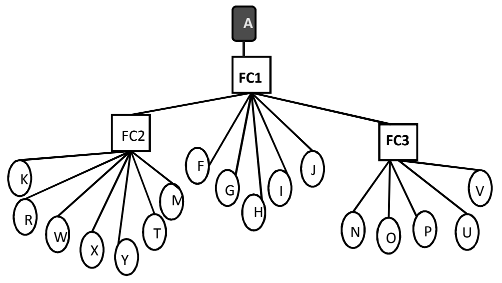


Figure 4-8: Simulation Tree transformation by reduction

The next operation to be performed is a Split operation. It takes the transformed tree and partitions it into several sub-trees with a node coordinating the timing on each sub-tree. In PCD++ the Node Coordinator (NC) has been introduced to be the local central scheduler on each sub-tree. The Root Coordinator shares its time scheduling capabilities with a NC. While the Root starts the simulation and performs I/O operations, the NC schedules simulation time and manages inter-process communication. The resultant structure is a SiS as shown in Figure 4-9.

Split(T') = <{A+NC1, NC2, NC3}, N', f'> with

N' = {A+NC1, NC2, NC3, FC1, FC2, FC3, K, R, W, X, Y, T, M, F, G, H, I, J, N, O, P, U, V}

f'(A+NC1) = {FC1}

f'(NC2) = {FC2}

f'(NC3) = {FC3}

f'(FC1) = {F, G, H, I, J}

f'(FC2) = {K, R, W, X, Y, T, M}

f'(FC3) = {N, O, P, U, V}

f'(K) = f'(R) = f'(W) = f'(X) = f'(Y) = f'(T) = f'(M) = f'(F) = f'(G) = f'(H) = f'(I) = f'(J) = f'(N) = f'(O) = f'(P) = f'(P) = f'(U) = f'(V) = ∅

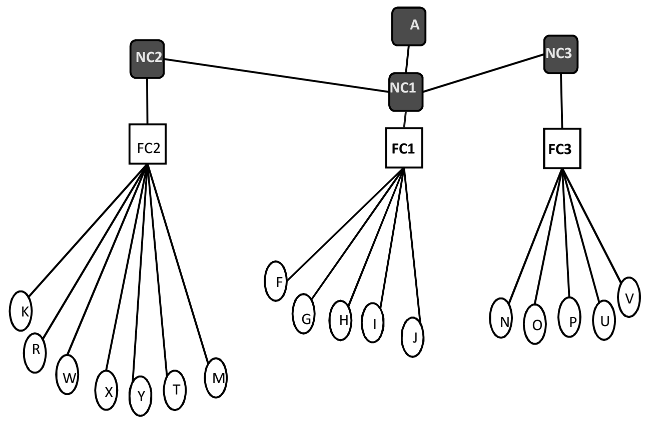


Figure 4-9: Intermediary Simulation Skeleton

In PCD++, there is at most one of each of NCs and FCs clustered into a process also called Logical Process. We see that the authors varied the number of processes/processors for checking the performance of PCD++. We however assume the use of three processes in this example. Figure 4-10 shows the SB resulting from the Cluster operation performed.

Cluster: $N' \rightarrow \{Ps_1, Ps_2, Ps_3\}$

$Cluster^{-1}(Ps_1) = \{A+NC1, FC1, F, G, H, I, J\}$

$Cluster^{-1}(Ps_2) = \{NC2, FC2, K, R, W, X, Y, T, M\}$

$Cluster^{-1}(Ps_3) = \{NC3, FC3, N, O, P, U, V\}$

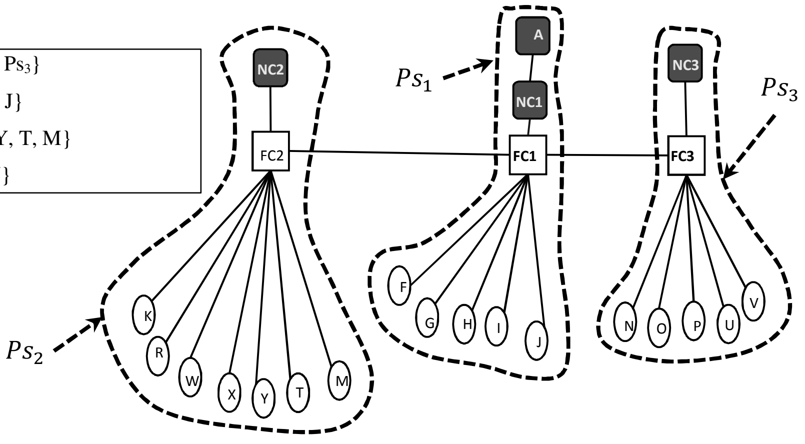


Figure 4-10: Intermediary Simulation Bundle

In PCD++, the constraint used is to permit only one Logical Process on a Processor thereby allowing intra-LP communication via the Node Coordinator. The FC is in charge of communication between its child Simulators. The final SG is obtained by mapping each Logical Process onto a Processor as shown by Figure 4-11 and suggested in (Park et al., 2006; Seo & Zeigler, 2009).

Distrib: $\{Ps_1, Ps_2, Ps_3\} \rightarrow \{Pr_1, Pr_2, Pr_3\}$

$Distrib(Ps_i) = Ps_i, \forall i \in \{1, 2, 3\}$

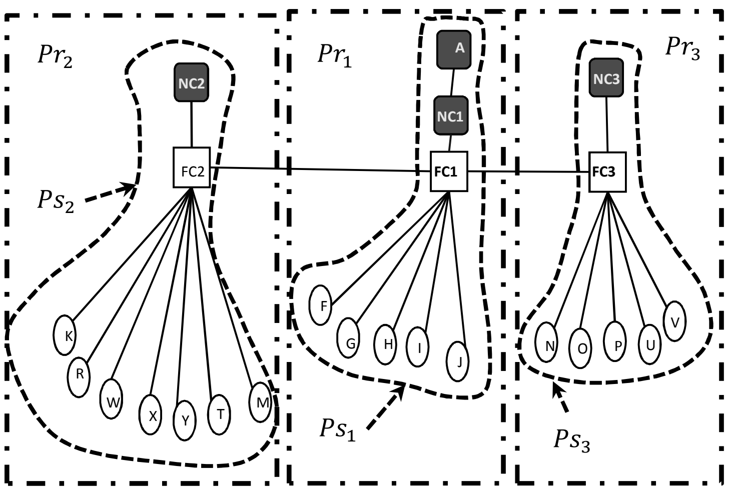


Figure 4-11: Simulation Graph obtained applying the PCD++ strategy

4.7. Conclusion

In this chapter, we gave descriptions of the Simulation Structures and Operations found in DEVS PDES simulation strategies. These descriptions were formally specified to provide an unambiguous understanding of the process of implementing a strategy. An approach was proposed to guide the construction of these structures. This approach was illustrated in a case study.

In Chapter 5, we describe the proposed unifying framework (see Figure 1-2) that integrates the Simulation Structures in a bid to engineer DEVS distributed Simulation process. The framework will include layers for the specification of the DEVS model, the specification of components identified in this chapter, the specification of the platform and the layer for generating simulation codes.

5. Model Driven Distributed Simulation Engineering Framework

5.1. Introduction

We present a multilayered framework to enable the process of DEVS distributed simulation engineering. There are four layers, namely; the Simulation Modeling Layer (SML), the Simulation Structure Layer (SSL), the Middleware Layer (ML), and the Simulation Code Layer (SCL) (see Figure 1-2). We call this framework the Model Driven Distributed Simulation Engineering Framework (MD2SEF). In this chapter, we examine this framework.

5.2. MD2SEF

The development of the multilayered framework is motivated by the need to obtain a systematic approach that can guide DEVS distributed simulation engineering, starting from the modeling stage to the code generation stage. It proposes a generic way for describing DEVS distributed simulations, independent of any DEVS implementation or platform. It uses the separation of concerns approach in maintaining the independence of distributed simulation components and bridging them together to obtain executable distributed simulation codes. It also offers the integration of DEVS models regardless of its requirements and demonstrates the suitability of a DEVS PDES implementation strategy to adapt to changes in a parallel/distributed platform. These contributions are captured in a framework called Model-Driven Distributed Simulation Engineering Framework (MD2SEF).

The MD2SEF consists of layers that contribute a separate functionality (see Figure 5-1). Figure 5-1 is a detailed description of Figure 1-2. Each layer consists of a set of components that deals with various pieces of concerns. For example, the Simulation Modeling layer consists of DEVS model component, which is useful for creating DEVS models.

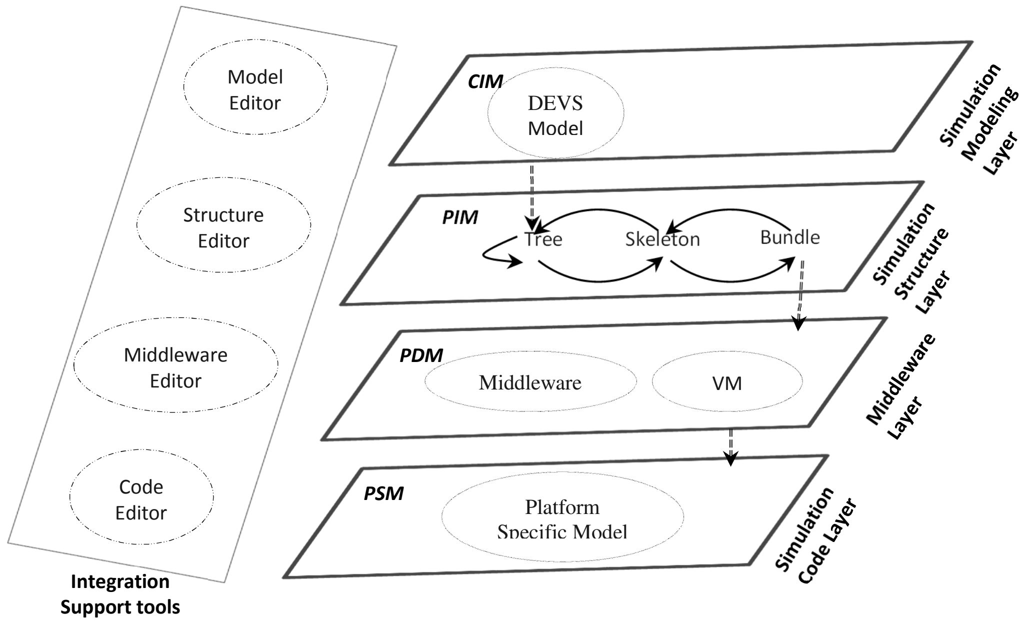


Figure 5-1: A multilayered framework for engineering DEVS distributed simulations

Layers in the framework make use of meta-modeling techniques to contribute a modeling language to the MD2SEF. It is also possible that a layer can have multiple sub layers for example the Simulation Structure layer. Simulation modeling layer deals with conceptual modeling and is useful for capturing any kind of DEVS model. The basic function of the Simulation Structure layer is the specification of components used in building a DEVS PDES strategy. The parallel/distributed platform, which can provide a complete DEVS PDES solution, is described at the Middleware layer and the code layer provides an abstraction from which executable codes can be generated. The framework is defined to be as generic as possible to enable the integration of heterogeneous implementations of the DEVS simulation protocol and eliminate the need for the user to start building an implementation from scratch. Full and partial automated integration processes needed for the layers to interoperate are captured at the integration support layer. In the rest of this work, we describe each of the layers in a top-bottom fashion.

5.2.1. MD2SEF Design Principle

The development of MD2SEF is based on the principle of “separation of concerns (Dijkstra, 1976; Parnas, 1972)”; a software engineering technique for controlling the complexities in a system by modular decomposition. This principle is used to isolate independent, or loosely related, aspects of building a DEVS simulation and to deal with each of them separately. In the MD2SEF, products of this decomposition are what we have called layers. The layers help to differentiate between the different kinds of tasks performed by the components, making it easier to create a design that supports modularity, extensibility, traceability, and reusability of components. The interactions between these layers are defined in a top-down fashion thereby eliminating circular dependences. These interactions are implemented using MDA principles. The integration support provides an interface for a simulation practitioner to interact with the MD2SEF without needing to know internal details of each layer.

The MD2SEF is flexible enough to allow the integration of user-defined components and address the demands of complex models and platforms. The constraint is that any user-defined component should follow the MD2SEF ideology. That is, the new components to be added should address; 1) simulation modeling issues at the Simulation Modeling Layer. 2) components useful for building a strategy at the Simulation Structure Layer. 3) platform specifications at the Middleware Layer. 4) distributed simulation code generation at the Simulation Code Layer.

5.2.2. From CIM to PSM

Model Driven Architecture (MDA) (Kleppe, Warmer, & Bast, 2003) has been used to support the development of the MD2SEF through the specification and integration of abstract models. DEVS models are specified in the Computation-

Independent Model (CIM); the Platform-Independent Model (PIM) describes the DEVS simulation protocol independent of the implementation platform; Platform-Description Model (PDM) gives the description for implementation platform; the Platform-Specific Model (PSM) contains information from PIM and PDM, which is used to develop an operating distributed simulation environment. Figure 5-2 describes the interactions in the MD2SEF layers such that, at each MDA abstract level the proposed modeling language and the model transformation technique (Mens & Van Gorp, 2006) (model to model (M2M) or model to text (M2T)) is captured. In addition, it describes how the layers interoperate as a unified whole in the MD2SEF.

Subsequently, we describe each of these concepts in their associated chapters.

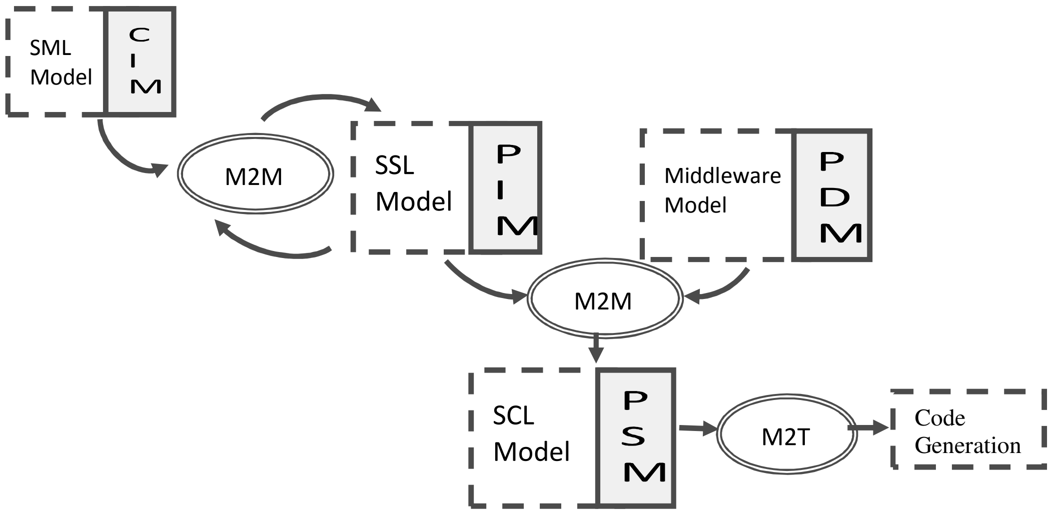


Figure 5-2: Interactions between MD2SEF Layers

5.3. Conclusion

In this chapter, we introduced the Model Driven Distributed Simulation Engineering (MD2SEF) as a multilayered framework for engineering DEVS distributed simulations. We examined the approach used in developing this framework. Afterward, we discussed the design principle and stated how each of the layers integrates with another. The integration was useful for capturing the components necessary for performing a parallel/distributed simulation.

In chapter 6, we examine each of the MD2SEF layers.

6. MD2SEF Layers

6.1. Introduction

In Chapter 5, we examined the MD2SEF as a multilayered framework for engineering DEVS distributed simulations useful for enabling the integration of DEVS components (DEVS model and DEVS PDES strategies) and distributed platforms. In this chapter, the discussion will be about the layers in the MD2SEF.

We start with a discussion about the topmost layer encountered when developing a DEVS distributed application i.e., the modeling layer. Then we examine the layer for specifying a DEVS PDES strategy followed by the layer for specifying distributed platforms and layer useful for obtaining simulation codes.

6.2. Simulation Modeling Layer (SML)

Computation-Independent Model (CIM) is used to describe the business model or the domain of the system, with no direct reference to how the system will be implemented (OMG, 2014). In the MD2SEF, DEVS (Bernard P. Zeigler et al., 2000) has been chosen as the CIM. DEVS is suitable as it is possible to use it to describe models independent of technologies and it is capable of elaborating high abstraction level model of the domain being considered through conceptualization. It can be used to give an unambiguous representation of the conceptual model, formal specification of inputs and outputs and categorization of static and dynamic variables. At the CIM, the modeler builds his DEVS model following the DEVS hierarchical specification approach: (1) atomic models, (2) then coupled models, (3) then the final model is composed of the model to simulate and the experimental frame coupled together.

We propose a DEVS meta-model (see Figure 6-1) as the CIM. There are other proposals in the literature (D Cetinkaya, Verbraeck, & Seck, 2012; Gamar & Agrawal, 2017; Lei, Wang, Li, & Zhu, 2009; H. S. Sarjoughian, Alshareef, & Lei, 2015), we however focus on developing one that can guide the definition of DEVS model on the MD2SEF. Constraints have been included to guard against accidentally describing a DEVS model that does not conform to DEVS model specifications (Bernard P. Zeigler et al., 2000). A description of the CIM, which embeds constraints directly into the meta-model, has been included in Appendix B.

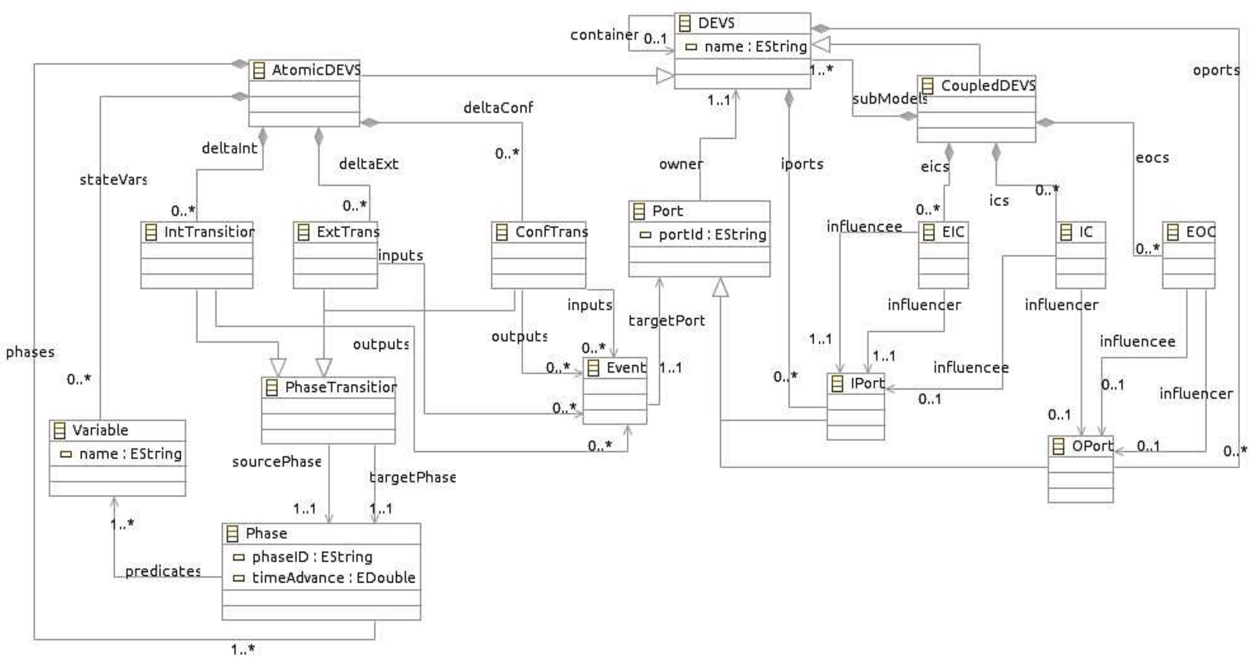


Figure 6-1: SML Meta-model

The meta-model is described as thus; *AtomicDEVS* and *CoupledDEVS* are *DEVS* models. An *AtomicDEVS* remains a basic *DEVS* model while a *CoupledDEVS* can be composed of *DEVS* models. A *Container* refers to the parent of a *DEVS* model. A *DEVS* model has *Ports*, which can be an *IPort* or an *OPort*. A *CoupledDEVS* is composed of *IC*, *EOC* and *EIC* coupling specifications. An *IC* couples an *Influencer OPort* to an *Influencee IPort* while an *EIC* couples an *Influencer IPort* to an *Influencee IPort*. An *EOC* couples an *influencer OPort* to an *Influencee OPort*. An *AtomicDEVS* is composed of *Variables*, *Phases* and *PhaseTransition*. A *Phase* is a state an *AtomicDEVS* can be in at time t. It makes use of a predicate *Variable.* A *PhaseTransition* defines the current and next phases of an *AtomicDEVS*. It is used in producing *Events* that are transmitted through a *Port.* The *PhaseTransition* are *ExtTrans, IntTransition* and *ConfTrans*.

6.3. Simulation Structure Layer (SSL)

Definitions useful for the implementation of DEVS simulation protocol are examined in the Simulation Structure Layer (SSL). They are described at a level that is independent of any platform. The layer, which is built on the Simulation Structures (see Chapter 4), provides support for the integration of heterogeneous implementations of the protocol. The use of Simulation Structures can guide the implementations of DEVS simulation protocol thus making it suitable for use in the SSL. In addition, the SSL includes a meta-model i.e., the Platform Independent Model (PIM) containing abstractions of the concepts found in Chapter 4.

DEVS PDES implementation strategies have been identified to have essentially three components in common namely; Tree (composed of Nodes i.e., Root, Coordinator, and Simulator), Process and Processor. There are four Simulation Structures proposed for implementing DEVS simulation protocol namely; Simulation Tree, Simulation Skeleton, Simulation Bundle, and Simulation Graph. However, the mapping between the Simulation Bundle and the Simulation Graph introduces platform-specific information. For the Simulation Structures to fit into the SSL, the platform information needs to be separated to ensure that the SSL only describes concepts that are independent of any platform.

The abstract model (see Figure 6-2) is described as thus. At the top of a Simulation *Tree* hierarchy is the *Root* (i.e. Root Coordinator) which coordinates the entire simulation and has a *BasicNode* (*Coordinator* or *Simulator*) as a descendant. Sub-classes of *BasicNode* have been included to capture existing nodes in literature and to show how new nodes can be included in the metamodel. For example, (Ezequiel Glinsky & Wainer, 2006) describes *FlatCoordinator* and *NodeCoordinator* as a proposal for implementing DEVS distributed simulation protocol. A *Coordinator* is composed of at least a subcomponent, which is a *BasicNode*. A Simulation *Skeleton* is composed of one or more *Root* that helps in splitting the Simulation *Tree* into sub-trees. The *Root* and *BasicNode*s are *Node*s that are mapped to a *Process* thereby achieving a many–to–one relationship and creating a Simulation *Bundle*. The *Middleware* captures conceptual information about the distributed platforms for enabling simulation execution and communication between components. A description of the PIM, which embeds constraints directly into the meta-model, has been included in Appendix C.

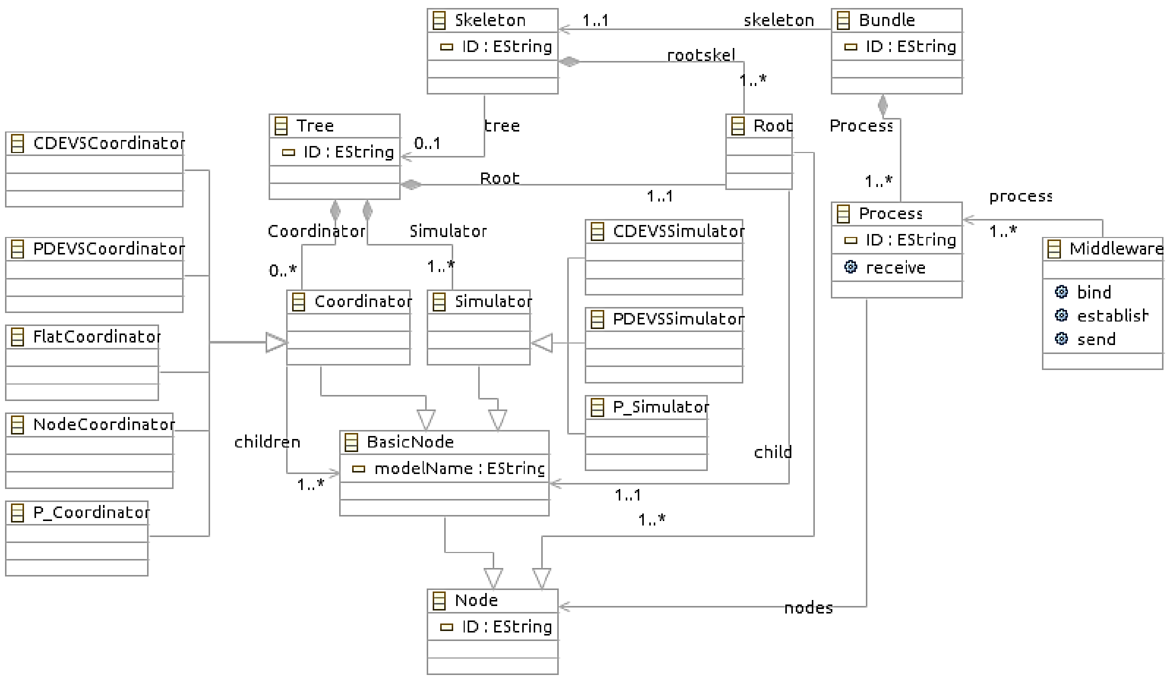


Figure 6-2: PIM at the Simulation Structure Layer

In the PIM, a conceptual view of the middleware is presented at a more abstract level to avoid describing platform-specific information. However, in the next section the middleware will be viewed from the perspective of a platform.

6.4. Middleware Layer (ML)

In Chapter 4, we proposed a generic layer for the understanding of simulation structures and operations. The framework was proposed based on the knowledge gained from the review of literature as presented in Chapter 3. Based on early results from experimenting with the structures, we observed that it does not specify how to integrate a middleware to obtain a distributed solution. The Middleware Layer (ML) has been introduced to resolve this issue. In this section, we describe a generic model that captures the platform specifications for DEVS PDES implementation strategies and show how it can be extended to specific platforms.

6.4.1. Platform Description Model (PDM)

Platform-Description Models (PDMs) are created to give information about the platforms to be used. They are usually combined with PIMs in a systematic way so as to obtain a PSM, that is, by using mechanisms for model transformation and model description of the PDM (J. Bézivin & Ploquin, 2001). In the MD2SEF, PDMs are introduced to capture information about the platforms or technologies that can be used to obtain a parallel/distributed execution. It is therefore necessary to consider how the mappings and communication between the simulation elements (i.e., simulation nodes, processes and processors) can be specified. The MD2SEF does not limit one to a specific the choice for a middleware rather it encourages the description of any other middleware e.g., Java RMI. This way, the MD2SEF is flexible.

The PDM has a Middleware class that has been defined to be generic to ease the integration of user-defined middleware specifications into the Middleware Layer through meta-model refinement techniques. The refinement is achieved by extending the Middleware class with specific information about the communication middleware and integrating the VM class to contain information about the processors (see Figure 6-3). This way, the integration of the abstract models into a unified whole can be achieved and it makes the layer flexible enough to capture and integrate any technology. It is therefore possible to consider any other parallel/distributed communication technology, other than the ones used in this work, without increasing the complexity of integrating the models. We make the difference here that the choice of the middleware technology is restricted to technologies that can enable communication and interoperability between simulation components. Other kinds of middleware can be considered if, in these technologies, mechanisms for synchronizing the simulation components can be separated from mechanisms that will enable communication between them.

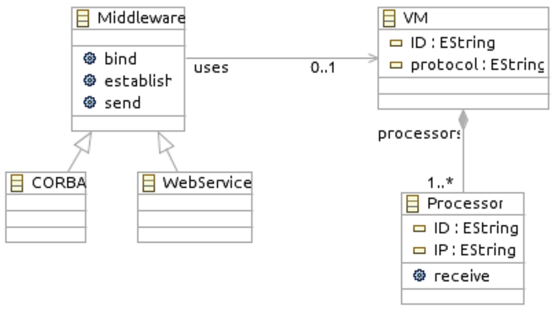


Figure 6-3: MD2SEF generic Platform Dependent model

6.4.2. Middleware Layer Extension

To integrate a platform, a model of the platform needs to be specified in such a way that it extends the Middleware. This extension makes it possible to replace the Middleware with a specific user-defined platform. In addition, we note here that it is possible to define UML profiles (Fuentes-Fernández & Vallecillo-Moreno, 2004) for developing customized extensions of the platform. However, the constraint is that it extends the Middleware class. Figure 6-4 describes, as an example, how this extension is possible for two middleware technologies. First, we propose a model to capture CORBA (Object Management Group

(OMG), n.d.) platform-definitions at an abstract level. CORBA has been defined to provide facilities for communication between server and client processes. It consists of reference to remote objects provided by the server and called by the client. For a successful communication the client needs to look-up and bind to available remote objects listed in the Registry. Then, the client can invoke the remote object through the Proxy. The remote objects are defined in the *Contract* and implemented in *ContractImpl*. The Contract specifies the list of parameters and operations to be implemented.

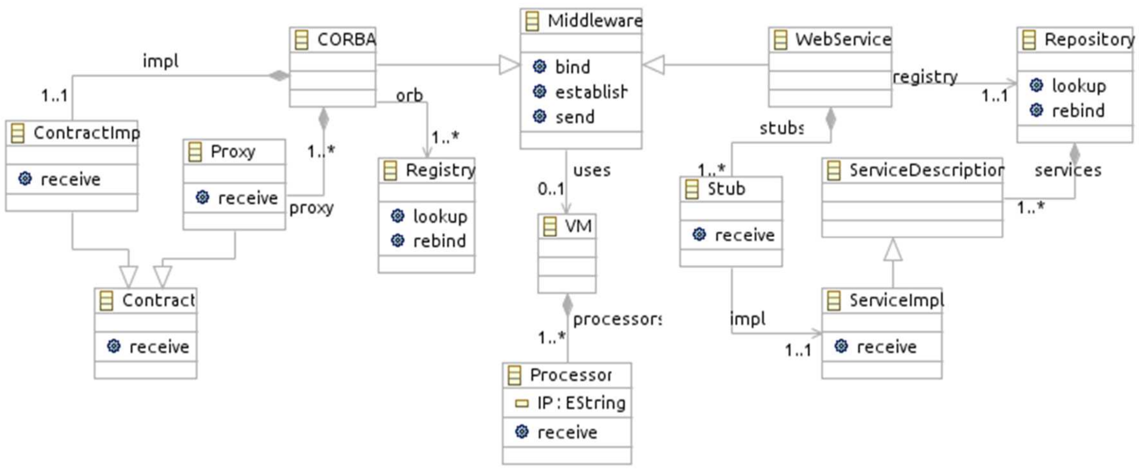


Figure 6-4: Platform definitions at the Middleware Layer

Second, in order for processes (see Figure 6-4) to communicate with each other using Web Services (Booth et al., 2004), service provider describes a web service by the using the service descriptor and publishes its availability using repository. The repository broadcasts the web service for service requestors, a client for example, to find the service. Once the service is found, the client uses the information to locate and bind to the service provider. In the case of a successful bind, the client can either request or provide services by using XML messages typically conveyed using network protocols.

In the following section, we describe the layer below the ML that is, the Simulation Code Layer

6.5. Simulation Code Layer (SCL)

The Simulation Structure Layer provides the MD2SEF with details of DEVS simulation protocol at a level that is independent of the platform while the platform details are captured at the Middleware layer. The Simulation Code Layer (SCL) merges the details from these layers to produce an abstract model i.e., Platform Specific Model (PSM), for the MD2SEF. The merging procedure is described in Section 7.4. The abstract model in Figure 6-5 is described for Web Service platform; still, it is possible to describe a model that integrates a different kind of middleware. This integration involves using model extension techniques in the Middleware Layer.

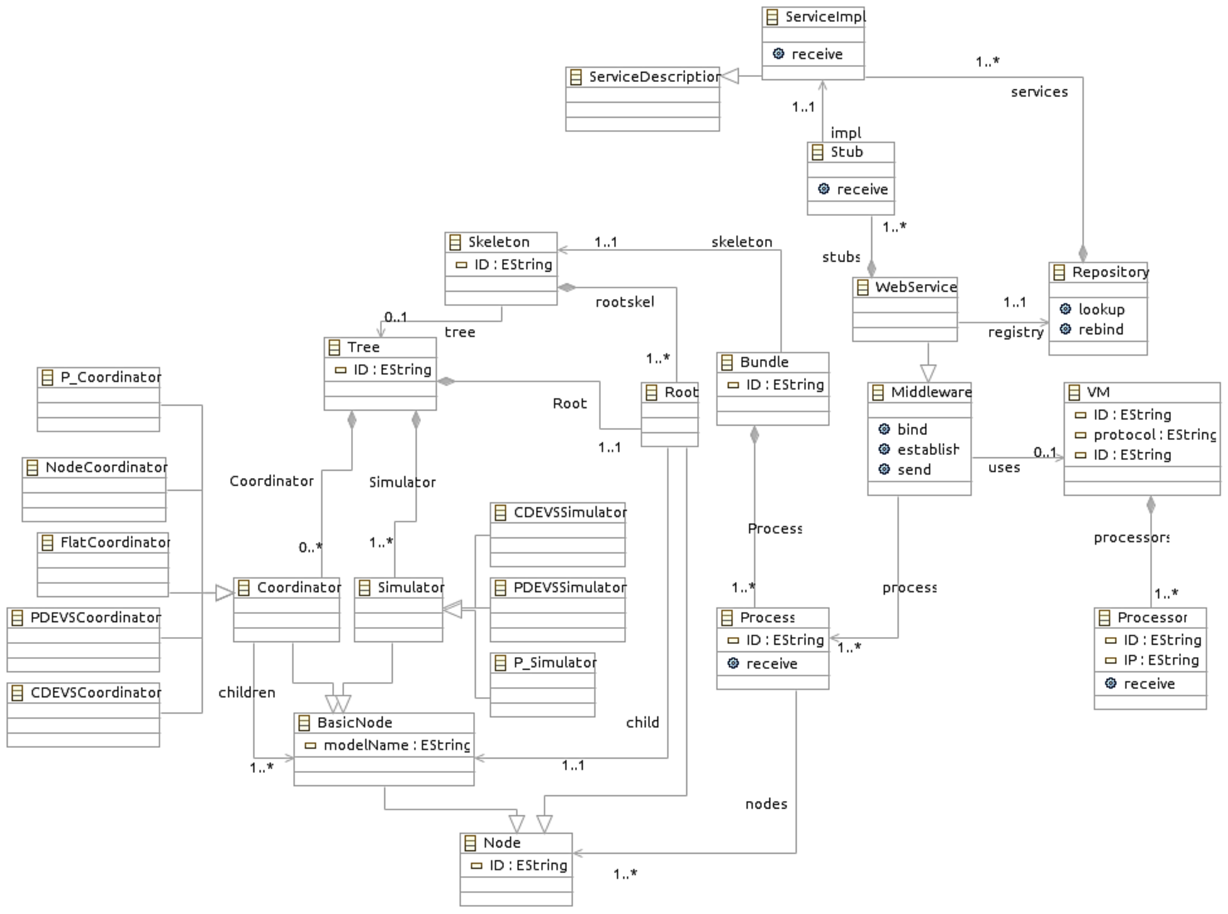


Figure 6-5: Platform-specific model at the SCL

DEVS distributed simulation codes are generated using elements of the PSM. To generate these codes, a specific programming language has to be selected so that proper transformation from the model to the appropriate code format (Java, C++, and so on) may be defined. As a result, the generated codes will contain artifacts representing elements of the abstract model. In Section 7.5, we describe how simulation codes are generated from the model in the SCL.

6.6. Conclusion

In this chapter, we examined each of the layers of the MD2SEF with which to engineer DEVS distributed simulations. DEVS modeling specification is examined at the SML, DEVS simulation protocol is captured at the SSL, platform components are integrated at the ML while a merger of SSL and ML features are captured at the code layer. There is the need to allow the MD2SEF layers interoperate as a unified whole, this and how it has been achieved are what we examine in Chapter 7.

7. Integration Support for MD2SEF Layers

7.1. Introduction

This chapter will illustrate how each layer integrates with another in the frame of MD2SEF, and in particular, it will present the integration using model transformation techniques. The integration process is described as thus. First, the DEVS model specification of the domain under consideration is described. Next, the specifications are adapted to a DEVS PDES implementation strategy. The strategy is laced with sufficient technological details to obtain a DEVS distributed simulation model. Finally, the executable DEVS distributed simulation codes are generated.

In the following sections, we describe how model transformation is achieved in the MD2SEF.

7.2. CIM to PIM

The M2M is achieved in a series of steps. In the first step, an instance of a DEVS model is used as source model. The modeling tool verifies the correctness of this model. In the second step, transformation rules are applied on the source file to generate a target XML file, which is a Simulation Tree. In general, the transformation rules search the source DEVS model file and specify the way target Simulation Tree elements can be initialized from each matched source DEVS model. It starts by creating a Tree element. The rule searches for the topmost element in the source model and assigns a Root Coordinator to it. It then maps every coupled model to a Coordinator and atomic model to Simulator. If an element is a coupled model, the rule searches to find if it has child models (i.e., coupled or atomic) as descendants and maps them unto the corresponding objects (Coordinator or Simulator) in the target model. The rules of transformation are illustrated in Figure 7-1 while the implementation is given in Appendix D.

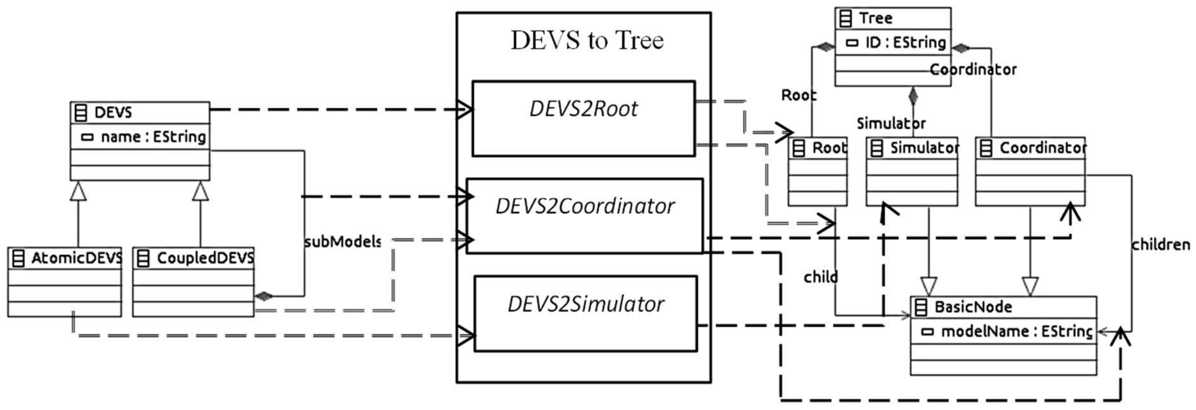


Figure 7-1: CIM to PIM Rule Mapping

7.3. PIM to PIM

The Simulation Tree, generated from CIM to PIM transformation, is transformed into a Simulation Skeleton through the addition of one or more Roots in the MD2SEF framework. The Simulation Skeleton is transformed into Simulation Bundle by mapping Nodes into Processes with the constraint that no two processes share the same node and every process has at least a node. The transformation technique used at this level is called endogenous transformation. The technique takes a source element and transforms it into another using the same meta-model as source and target. Thus, in the MD2SEF, a PIM to PIM transformation is achieved. The transformation pattern is illustrated in Table 7-1.

Table 7-1: PIM to PIM transformation pattern

Pattern	Source	Target
Tree2Skeleton	Tree +Root +Coordinators +Simulators	Skeleton +Root +Coordinators +Simulators +new Roots
Skeleton2Bundle	Skeleton +Root +Coordinators +Simulators +new Roots	Bundle +Processes

7.4. (PIM,PDM) to PSM

The essence of a merging process in the MD2SEF is to integrate the platform capability and obtain a model that can completely describe a DEVS distributed simulation environment at the PSM level. As a result, many PSMs can be generated for each differently specified PDM model. The model transformation, which takes the model obtained from the PIM and merges it with the model in the PDM, is described. The merging process is summarized as combining two models, following a recursive union-like copy approach. The process starts by investigating the overlapping class, i.e., the Middleware Class, which is copied to the new model along with the disjointed classes. Then links, i.e., inheritances, references, and compositions are considered and copied as-is in the new model. Thus, the new model obtained from merging PIM with PDM is enriched with technology specific details using model transformation techniques, thereby enabling the model for execution on parallel/distributed architectures. This new model can be used to describe models at the PSM abstract level. The merging rules are illustrated in Figure 7-2 while the implementation is given in Appendix E.

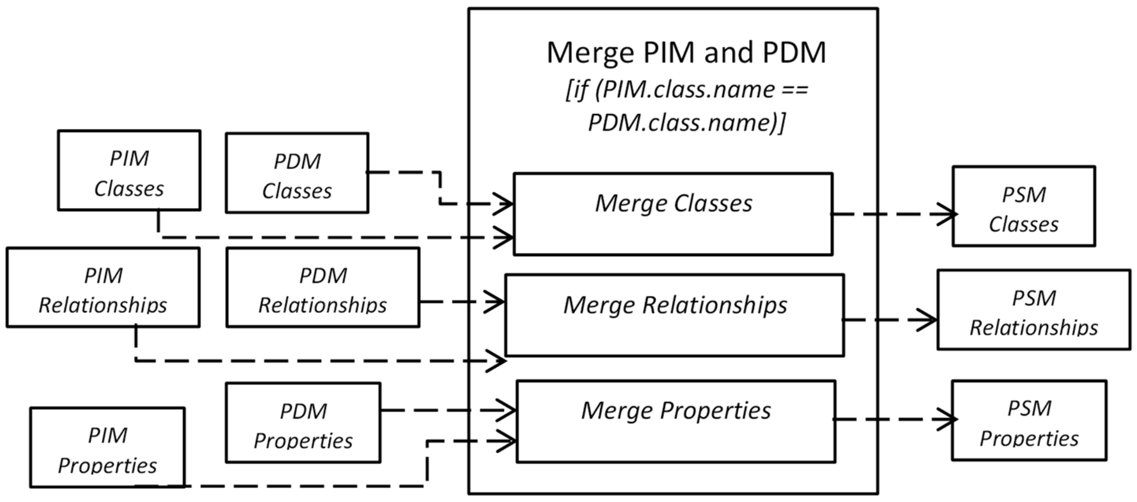


Figure 7-2: (PIM, PDM) to PSM rule mapping

7.5. PSM to Code

The generated PSM includes the required information to obtain simulation source codes in the MD2SEF. The PSM contains information about the DEVS model, the DEVS simulation protocol and the middleware. Before getting to this point, specifications about the DEVS simulation protocol implementation and the chosen middleware ought to have been clarified. This is the case for using an already existing implementation of DEVS simulation protocol. To describe the code generation, as an example, we have chosen PDEVS (Bernard P. Zeigler et al., 2000) for the simulation protocol, Web Service for the middleware and Java as the language. The transformation process is partially automated to allow the integration of user-defined specifications for example variables and data structures, geared towards the final generation of distributed executable simulation codes. Table 7-2 shows how specific PSM elements are mapped to codes, for example Java. A separate file containing codes is generated for each of the transformed PSM elements.

Table 7-2: PSM to Codes transformation pattern

PSM	CODES
Coordinator +children, modelName, ID	Class +variables
Simulator	Class
Root +ID, child	Class +variables
Process +ID +receive	Class +variables +operations
Middleware +process, uses +bind,establish,send	Class +variables +operations
VM +ID,processors	Class +variables
Processor +ID, IP	Class +variables
ServiceImpl	Class

+receive	+operations

7.6. MD2SEF Integration Tools

This integration is achieved through a series of model transformations (see Figure 5-2). To perform model transformations in the MD2SEF framework, model instances are represented in an eXtensible Markup Language (XML) (Bray, Paoli, Sperberg-McQueen, Maler, & Yergeau, 2008) format. This is for enabling model continuity, reusability and interoperability. The instance model serves as the input model, which is parsed using some transformation rules to produce a target model. These models and their transformations have be implemented using the Eclipse Modeling Framework (EMF) (Steinberg et al., 2008); EMF/Ecore (Steinberg et al., 2008) for specifying the models, ATL (Jouault, Allilaire, Bézivin, & Kurtev, 2008) for model to model transformations, Acceleo Model Transformation Language (MTL) (Acceleo, 2015) for model to text transformation and Sirius (Viyovic, Maksimovic, & Perisic, 2014) for a graphical representation of the simulation structures. The MD2SEF has been developed on the EMF to take advantage of its compatibility with other tools. The instance model obtained from each MD2SEF layer can be; edited on the EMF, stored as an XML file and validated against the associated meta-model. A guide for using the MD2SEF is proposed in Figure 7-3 and the plugins can be found in https://goo.gl/eXEsHr.

7.7. Conclusion

In this Chapter, we examined how each MD2SEF layer integrates with the next layer via the abstract model defined for each layer and application of model transformation techniques. We showed how information from the Simulation Modeling Layer can be mapped to a DEVS PDES strategy in the Simulation Structure Layer. Then we showed how the model in the Middleware Layer can be extended with user-defined middleware and merged with a DEVS PDES strategy to obtain a DEVS distributed simulation model. We also showed how simulation codes can be generated from this model. To make this chapter complete, we included a discussion about the tools that enabled the development of MD2SEF.

The next Chapter will demonstrate further the MD2SEF and evaluate the feasibility of the framework for engineering DEVS distributed simulations.

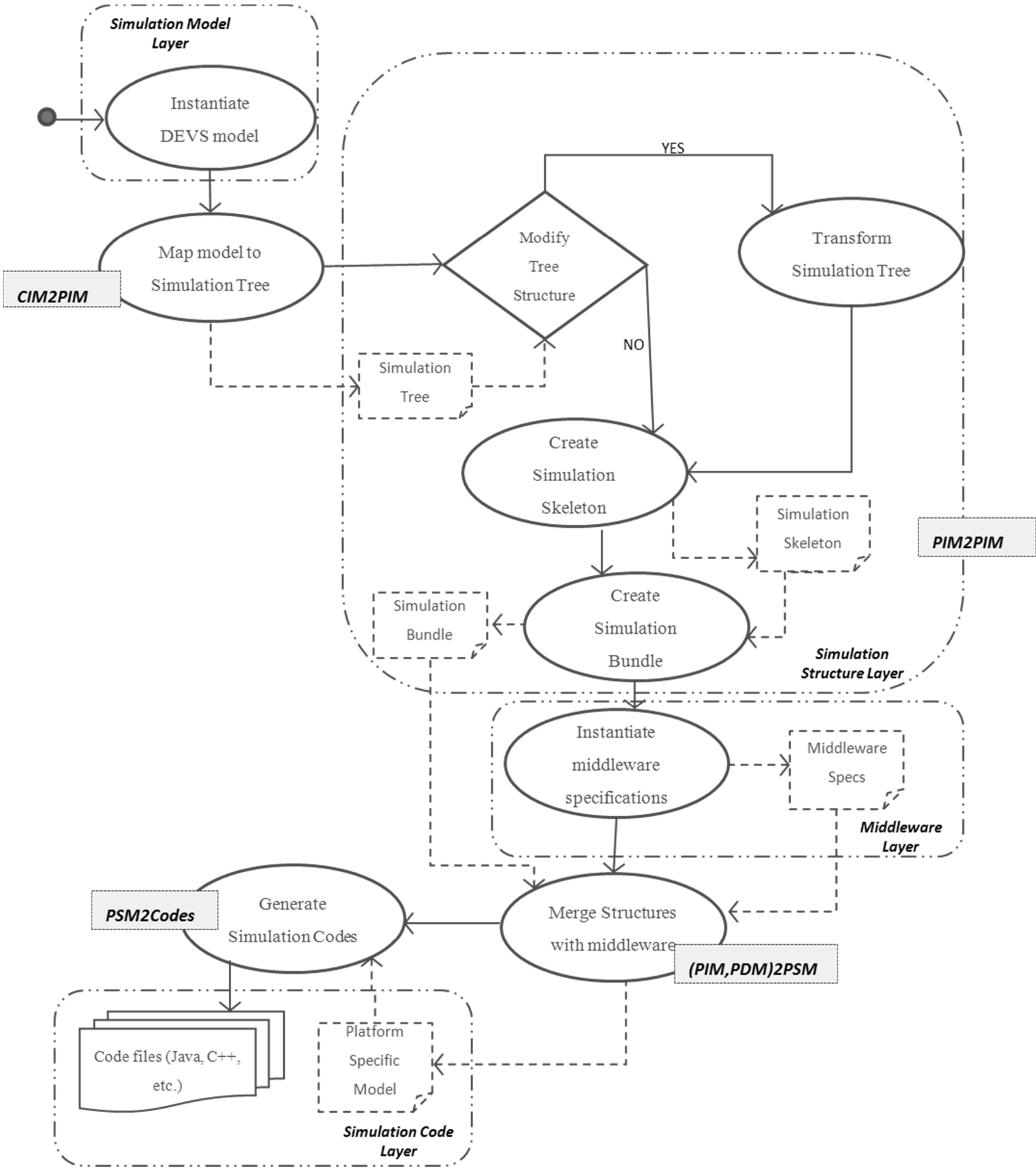


Figure 7-3: MD2SEF guide

8. MD2SEF Application

8.1. Introduction

The aim of the study is to illustrate how the MD2SEF contributes to resolving migration challenges that may occur due to evolving models or unanticipated technological changes. The study involves a model of the Lassa Hemorrhagic Fever (LHF), how it undergoes changes in context: by model enlargement through the increase of the dimension parameter, which expresses the geographical area, and by transfer from one execution platform to another.

8.2. Lassa Hemorrhagic Fever (LHF)

Lassa Fever, which has been in existence for a longer period, is yet to be completely eradicated in Nigeria. More recently, there are recorded occurrences of this disease. Descriptions of the Lassa Fever, which is also known as Lassa hemorrhagic fever (LHF), dates from the 1950s. The virus was first described in 1969 from a case in the town of Lassa, in Borno State, Nigeria (Frame, Baldwin, Gocke, & Troup, 1970). It is endemic in Nigeria, Liberia, Sierra Leone, Guinea, and other West African countries. Also, there are some documented cases of LHF being imported from Africa into other countries like UK (Sogoba, Feldmann, & Safronetz, 2012) and Germany (Günther et al., 2000). Humans are infected with this disease by eating foods that are contaminated with saliva, urine or excreta of the hosted Lassa virus rat. It can be transmitted from human to human via aerosol transmission (coughing), or from direct contact with infected human blood, urine, or semen. According to the World Health Organization (World Health Organization, 2017), between August 2015 and May 2016, there has been 273 cases of LHF, including 149 deaths in Nigeria. Of these, 165 cases and 89 deaths have been confirmed through laboratory testing (Case Fatality Rate = 53.9%). These cases were reported from 23 states in Nigeria. Two major characteristics can describe the spread of LHF. Firstly, the incubation period for LHF is about 24 days while death can occur within 14 days in severe cases. Secondly, the outbreak of LHF is within a population (endemic) and can occur in another population as a result of the movement of the disease carrier from one population to another (epidemic). Therefore, population dynamics is an inclusive mechanism of LHF spread.

8.3. LHF DEVS models

The three DEVS-specified models are:

- The disease spread model (Epidemiology)
- The population dynamics model (Demography)
- The LHF model (coupling between Epidemiology and Demography)

The disease spread model classifies Susceptible (S), Infected (I) and Recovered (R) as the three disease statuses for a population. Susceptible individuals are not affected but are at risk for infection. Infected individuals are capable of transmitting the virus. Recovered individuals have gained permanent immunity from the disease. The population is distributed over a geographical area that is squared into cells of dimension 1 km^2. The total number of cells is DIM×DIM. That way, each cell can be identified by $(S,I,R)_{ij}$, where i and j belong to {1, …, DIM}, and S_{ij}, I_{ij} and R_{ij} are the numbers of Susceptible, Infected and Recovered in that cell, respectively. The mathematical compartmental model from which derives the DEVS model assumes a closed population (i.e., no birth, no death, and no migration). β is the average number of virus-spreading contacts made by each infected individual in unit time (it is given as 0.95833), γ is the recovery rate (thus, 1/γ is the average infectious period, which is given as 0.04167).

MEpidemiology = $<X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta>$, where:

- $X = Y = S = (S,I,R)_{ij} \,/\, i \in \{1..DIM\}, j \in \{1..DIM\}, S \in \aleph, I \in \aleph, R \in \aleph$
- $ta : S \rightarrow \Re_0^{+\infty}$
  $ta(current) = 1$
- $\delta_{int} : S \rightarrow S$

$\delta_{int}((S,I,R)_{ij}) = (S',I',R')_{ij}$ / $i \in \{1..DIM\}$, $j \in \{1..DIM\}$, $S \in \aleph$, $I \in \aleph$, $R \in \aleph$

with:

$S'_{ij} = S_{ij} - \beta S_{ij} I_{ij}$

$I'_{ij} = \beta S_{ij} I_{ij} + (1-\gamma) I_{ij}$

$R'_{ij} = R_{ij} + \gamma I_{ij}$

$\beta=0.95833$

$\gamma=1/0.04167$

- $\lambda: S \rightarrow Y$
  $\lambda(\text{current}) = \text{current}$
- $\delta_{ext} : Q \times X \rightarrow S$, with $Q = \{(s,e) / s \in S, 0 \leq e < 1\}$
  $\delta_{ext}(\text{current}, e, x) = x \quad \forall e \in [0, 1]$
- $\delta_{conf} : S \times X \rightarrow S$
- $\delta_{conf}(\text{current}, x) = \delta_{int}(\delta_{ext}(\text{current}, 0, x))$

The population dynamics model also considers that the population is distributed over a geographical area that is squared into cells of dimension 1 km². The total number of cells is DIM×DIM. Each cell is identified by $(S,I,R)_{ij}$, where i and j belong to {1, …, DIM}, and S_{ij}, I_{ij} and R_{ij} are sub-categories of population with specific net growth rates (i.e., birth – death +/- migrations from/towards the cell). The rate of migration between any pair of cells depends on the population distribution in both cells, and the relative attractivity of each cell to the other.

$M_{Demography} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta \rangle$, where:

- $X = Y = S = (S,I,R)_{ij}$ / $i \in \{1..DIM\}$, $j \in \{1..DIM\}$, $S \in \aleph$, $I \in \aleph$, $R \in \aleph$
- $ta : S \rightarrow \Re_0^{+\infty}$
  $ta(\text{current}) = 1$
- $\delta_{int} : S \rightarrow S$
  $\delta_{int}((S,I,R)_{ij}) = (S',I',R')_{ij}$ / $i \in \{1..DIM\}$, $j \in \{1..DIM\}$, $S \in \aleph$, $I \in \aleph$, $R \in \aleph$

  with:

  $S'_{ij} = N_{ij}' S_{ij} / N_{ij}$

  $I'_{ij} = N_{ij}' I_{ij} / N_{ij}$

  $R'_{ij} = N_{ij}' R_{ij} / N_{ij}$

  where:

  $N_{ab} = S_{ab} + I_{ab} + R_{ab} \quad \forall a \in \{1..DIM\}, \forall b \in \{1..DIM\}$

  $N_{ij}' = G_{ij} N_{ij} + \sum_{(i,j) \neq (k,l)} (\alpha_{ij} - \alpha_{kl}) |N_{ij} - N_{kl}| \exp(D(ij,kl))$

  G_{ij} is the net growth rate of cell (i,j)

α_{ij} is the relative attractivity of cell (i,j)

D(ij,kl) is the Cartesian distance between cells (i,j) and (k,l)

- $\lambda: S \rightarrow Y$
  $\lambda(\text{current}) = \text{current}$
- $\delta_{ext} : Q \times X \rightarrow S$, with $Q = \{(s,e) / s \in S, 0 \leq e < 1\}$
  $\delta_{ext}(\text{current}, e, x) = x \quad \forall e \in [0, 1]$
- $\delta_{conf} : S \times X \rightarrow S$
  $\delta_{conf}(\text{current}, x) = \delta_{int}(\delta_{ext}(\text{current}, 0, x))$

The LHF model couples the previous two models into a bigger model, where each component feeds the other.

$$M_{LHF} = \langle X, Y, D, \{M_d\}_{d \in D}, \{I_d\}_{d \in D}, \{Z_{m,n}\}_{m,n \in D}\rangle, \text{ where:}$$

- $X = Y = \varnothing$
- D = {Epidemiology, Demography}
- $I_{Epidemiology}$ = {Demography}
- $I_{Demography}$ = {Epidemiology}
- $Z_{Epidemiology,Demography}: Y_{Epidemiology} \rightarrow X_{Demography}$
  $Z_{Epidemiology,Demography}(x) = x$
- $Z_{Demography,Epidemiology}: Y_{Demography} \rightarrow X_{Epidemiology}$
- $Z_{Demographyy,Epidemiology}(x) = x$

8.4. LHF model description on MD2SEF

Figure 8-1 exhibits the agile scenario resulting from the application of the MD2SEF to the engineering of distributed DEVS simulation solutions, as the context changes. Steps and milestones of this scenario are marked and described hereafter.

At the SML level (Step 1), the system model is defined as a coupled DEVS model composed of two atomic models: one for population dynamics (called Demography), and the other for LHF spread (called Epidemiology). Each of these atomic models has been built, based on theoretical models of the literature (Kermack & McKendrick, 1927), respectively in population dynamics modeling and disease spread modeling. The Epidemiology model doesn't capture the demographics of a particular location, while the Demography model lacks of information about the health status of individuals of a population. As a result, these models have been coupled to feed each other with necessary information, i.e., the Demography model sends to the Epidemiology model, the distribution of population in a location resulting from its demographics, while the Epidemiology model sends back a distribution of individuals that are either infected, recovered or susceptible.

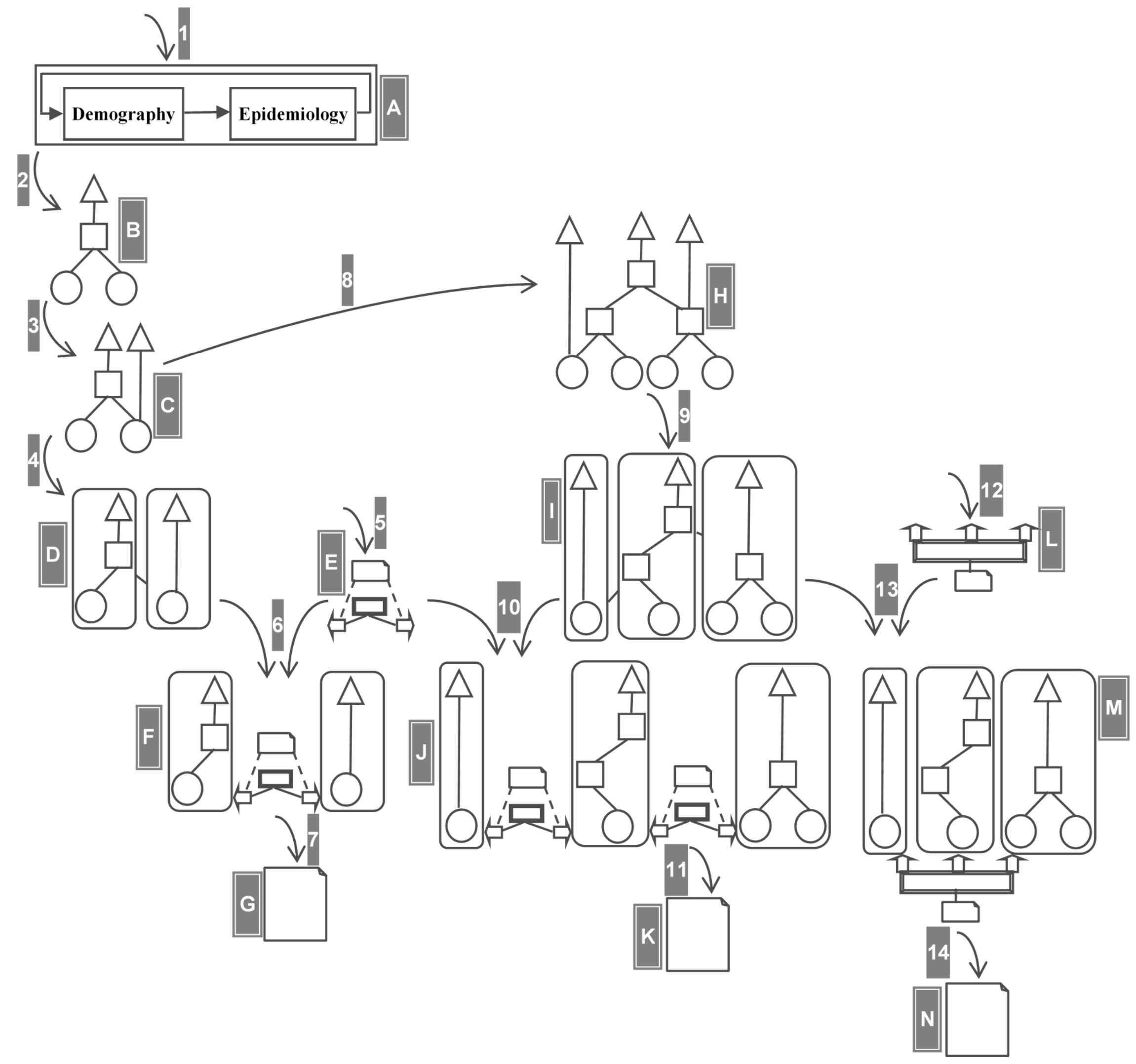
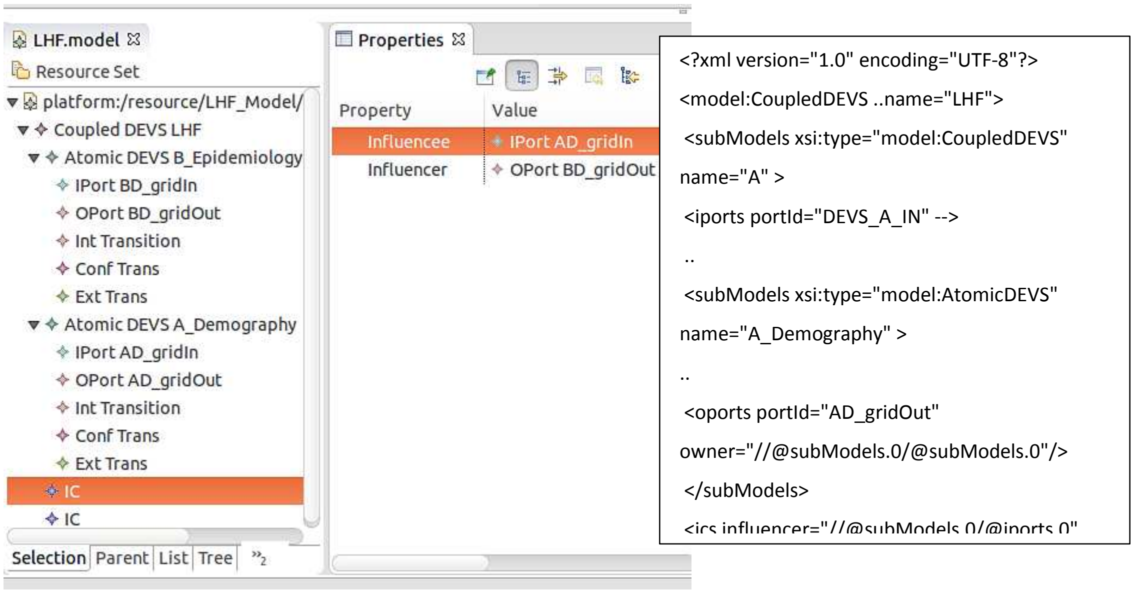


Figure 8-1: Agile scenario in distributed DEVS simulation engineering

The DEVS specification of each of the two models, as well as the resulting coupled model, is given in Section 8.3. The DEVS model, which is stored in an XML/XMI file (Document A), is parsed for transformation (Step 2) into the simulation Tree (Document B). Figure 8-2 shows the description of the model in the MD2SEF user interface and an excerpt of the corresponding XML. Appendix F gives the complete description of the modified model in XML.


LHF.model ☒
Resource Set
- platform:/resource/LHF_Model/
  - Coupled DEVS LHF
    - Atomic DEVS B_Epidemiology
      - IPort BD_gridIn
      - OPort BD_gridOut
      - Int Transition
      - Conf Trans
      - Ext Trans
    - Atomic DEVS A_Demography
      - IPort AD_gridIn
      - OPort AD_gridOut
      - Int Transition
      - Conf Trans
      - Ext Trans
    - IC
    - IC

Selection | Parent | List | Tree

Properties ☒

Property	Value
Influencee	IPort AD_gridIn
Influencer	OPort BD_gridOut

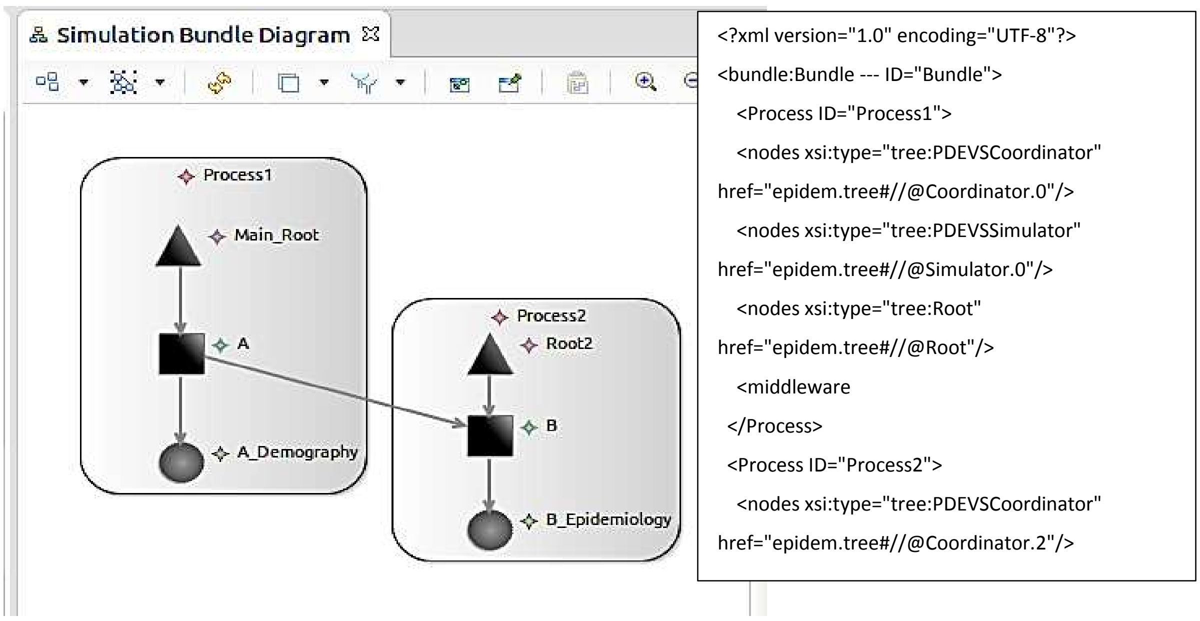
```
<?xml version="1.0" encoding="UTF-8"?>
<model:CoupledDEVS ..name="LHF">
 <subModels xsi:type="model:CoupledDEVS"
name="A" >
 <iports portId="DEVS_A_IN" -->
 ..
 <subModels xsi:type="model:AtomicDEVS"
name="A_Demography" >
..
 <oports portId="AD_gridOut"
owner="//@subModels.0/@subModels.0"/>
 </subModels>
 <ics influencer="//@subModels.0/@iports.0"
```

Figure 8-2: LHF model description and its representation in XML.

User-defined choices made in the MD2SEF interface, at the SSL level, lead successively to splitting the Tree (Step 3) into a Skeleton (Document C), by addition of a Root on top of one simulator, then (Step 4) into a Bundle (Document D), by clustering nodes into two logical processes. Figure 8-3 shows how the MD2SEF interface captures graphically this bundle. Triangles, squares and circles represent Roots, Coordinators and Simulators respectively, while round rectangles represent Processes. An excerpt of the corresponding XML is displayed as well.

At the ML level (Step 5), a CORBA-based Middleware is established (Document E), defining the Contract, ContractImpl, Proxy and the VM. The newly integrated middleware is merged (Step 6) with the existing model, at the SCL level, to obtain a platform specific model (Document F). This model is used during automated code generation (Step 7) to obtain Java distributed simulation codes (Document G) for the LHF model.

To take into account larger geographical regions, the initial skeleton is modified (Step 8) such that the initial grid manipulated by simulator is divided into two regions, each under the control of a simulator (Document H). It is important to notice that changes done at the level of a DEVS simulation tree have always a counterpart at the DEVS model specification level. It is also important to notice that the tree can be altered in a way that reduces the number of nodes (instead of increasing them, as done in this example). Such a situation arises for example when the DEVS simulator is flattened for the purpose of improving the computational performances (Ezequiel Glinsky, 2005). The new Skeleton obtained in Document H is turned (Step 9) to a new Bundle (Document I), which is merged (Step 10) with the already defined Middleware in Document E, to produce the new PSM (Document J). The latter is translated (Step 11) to a new executable Java code of distributed DEVS simulation (Document K).



```
<?xml version="1.0" encoding="UTF-8"?>
<bundle:Bundle --- ID="Bundle">
  <Process ID="Process1">
  <nodes xsi:type="tree:PDEVSCoordinator"
href="epidem.tree#//@Coordinator.0"/>
  <nodes xsi:type="tree:PDEVSSimulator"
href="epidem.tree#//@Simulator.0"/>
  <nodes xsi:type="tree:Root"
href="epidem.tree#//@Root"/>
  <middleware
 </Process>
 <Process ID="Process2">
  <nodes xsi:type="tree:PDEVSCoordinator"
href="epidem.tree#//@Coordinator.2"/>
```

Figure 8-3: Simulation Bundle structure of the LHF model

To migrate to a different middleware, say Web Services, modifications are made at the ML level (Step 12) to replace CORBA with Web Services (Document L), specifying the ServiceDescription and ServiceImpl classes. The Stub class is customized as a UML stereotype to enable the integration of stubs that will be generated after compiling the ServiceDescription. The ServiceDescription has a receive operation that is called through the Stub and executed by processes during communication. The operation is composed of the information about the recipient process, the events to be treated, and the associated time. Figure 8-4 shows an excerpt of the XML representation of the service description.

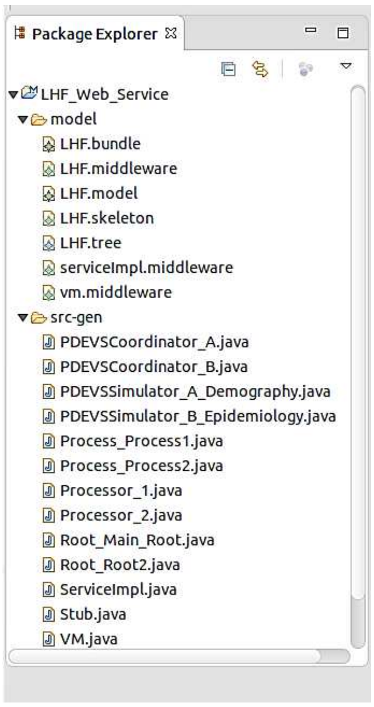
```
<definitions targetNamespace="http://stubs.service.org"
name="ServiceDescription>
…
<message name="receive"><part name="parameters"
element="tns:receive"/></message>
<operation name="receive">
<soap:operation soapAction=""/>
  <input><soap:body use="literal"/></input>
  <output><soap:body use="literal"/></output>
</operation>
<service name="ServiceDescription">
<port name=" ServiceDescriptionPort" binding="tns:
ServiceDescriptionPortBinding">
<soap:address
location="http://localhost:8080/SimStudioWebServices/SimStudioServi
ce"/>
</port></service></definitions>
```

```
@WebService(name="ServiceDescription",
targetNamespace="http://stubs.service.org")
public interface Service {
        @WebMethod
        void receive(Events event, double time,
String recipient);
}
```

Figure 8-4: Excerpt of the Web Service Description

The existing Bundle defined in Document I is now merged (Step 13) with this new Middleware to produce another PSM (Document M), which is then translated (Step 14) to an executable Java-based distributed DEVS simulation code (Document N). Figure 8-5 shows Java classes that are generated in the MD2SEF interface, during the process of code generation.

Using the MD2SEF, during the repeated migrations of the initial model, helped to eliminate the need to start from scratch at each change of context. As a result, the overall development time and costs have been improved.



```
Package Explorer
▼ LHF_Web_Service
  ▼ model
      LHF.bundle
      LHF.middleware
      LHF.model
      LHF.skeleton
      LHF.tree
      serviceImpl.middleware
      vm.middleware
  ▼ src-gen
      PDEVSCoordinator_A.java
      PDEVSCoordinator_B.java
      PDEVSSimulator_A_Demography.java
      PDEVSSimulator_B_Epidemiology.java
      Process_Process1.java
      Process_Process2.java
      Processor_1.java
      Processor_2.java
      Root_Main_Root.java
      Root_Root2.java
      ServiceImpl.java
      Stub.java
      VM.java
```

Figure 8-5: Auto generated Java files from the Web Service PSM

8.5. Conclusion

In this Chapter, we showed the applicability of the MD2SEF by instantiating it with a model of Lassa Hemorrhagic Fever and illustrated the DEVS distributed simulation engineering processes involved in mapping the LHF model to distributed platforms. We also showed how the framework supports agility when transforming the initial model to a more complex model and transferring it from one middleware to another. This proof of concept validates the MD2SEF as a framework for flexible engineering and reengineering of DEVS distributed simulations.

In Chapter 9, we examine model-driven frameworks for modeling and simulation in relation to the MD2SEF

9. Related Works

9.1. Introduction

As regards to the blend between DEVS, DS engineering and MDA several proposals have been made, they include Model Driven Development for M&S (MDD4MS) (Deniz Cetinkaya et al., 2011) and DEVS Unified Process (DUNIP) (Mittal & Martin, 2013). In this chapter, we examine these related works in literature, as they are model-driven frameworks geared towards the development of DEVS distributed simulations.

9.2. MDD4MS

MDD4MS framework is a formalized model-driven framework that consists of a M&S life cycle, meta-model definitions for each MDA abstract model, model transformations and a tool architecture for the overall process. The process of development was described for transforming BPMN models into DEVS models then for transforming into an executable form on a specific DEVS distributed simulator called DEVSDSOL (Seck & Verbraeck, 2009). It has been presented as an integrated approach to bridge the gaps between different steps of a simulation study by using meta-modeling and model transformations.

9.3. DUNIP

DUNIP is a development and testing environment that is built on Bifurcated Model Continuity-Based Lifecycle Methodology and allows DEVS-based Modeling and Simulation (M&S) over net-centric platforms using DEVS/SOA (Mittal et al., 2009). The complete process of DUNIP starts from the automated generation of DEVS models from various Domain Specific Language (DSL) requirements specifications. Then, the DEVS models are integrated with distributed simulation features before they are executed on service-oriented architectures.

9.4. Model-Driven Frameworks for Engineering DEVS distributed simulations

DUNIP and MDD4MS are implementation platforms for engineering Distributed Simulation (DS). The methodology used in DUNIP is based on systems engineering, model evolution and agile development process while MDD4MS methodology is based on software engineering and model evolution. MD2SEF development methodology is based on similar concepts as MDD4MS. Formalization of concepts in these frameworks has been provided except in the case of DUNIP. MDD4MS provides guidance for formal model transformations between the models at different abstraction levels. MD2SEF is based on Simulation Graph concepts that provide formal definitions for elements that enable DEVS DS. These features are summarized in Table 9-1.

Table 9-1: Identified features in the frameworks

Features	MD2SEF	MDD4MS	DUNIP
Formalism	DEVS	DEVS, BPMN	DEVS
Model-Driven	MDA	MDD	MDSE
Theoretical Framework	Y	Y	N
Tool Support	Y	Y	Y
Formalism Independent	N	Y	N
DEVS PDES Strategy Independent	Y	N	N
Middleware Independent	Y	N	N

Y=Yes, N=No, DEVS=Discrete Events Systems Specification, MDA=Model Driven Architecture, MDD=Model Driven Development, MDSE=Model Driven Systems Engineering

MD2SEF has proposed an integrative support for engineering DEVS simulations. MDD4MS and DUNIP also provide toolset support for performing M&S however, they are tightly coupled with a specific middleware, that is, Service Oriented Architecture. In this work, we propose the use of any kind of middleware in the PDM to enable DS. That is, it can either be HLA, Web Services, CORBA and so on. This thus makes the MD2SEF generic. MDD4MS and DUNIP are tightly coupled with DEVSDSOL and DEVS/SOA respectively for simulating DEVS models. In contrast, we propose that the use of a DEVS distributed simulator or programming language can be made based on the modeler's level of expertise. This is achieved at the point of adapting the PSM to the chosen DEVS simulator. The tool architecture presented in(Mittal & Martin, 2013) accepts the description of different kinds of Domain Specific Models (DSMs) and supports interoperability between DEVS simulators over the web. It however may not be used to define other kinds of DEVS implementation strategies as described in (Adegoke et al., 2013). MDD4MS and DUNIP describe MDA features for the transformation of a Domain Specific Model (DSM) into a DEVS model before mapping to a specific DEVS simulation engine and specific simulation architecture; we however focus on providing modeling languages and transformation rules that describe the process of mapping any DEVS model on any high performance infrastructure.

9.5. Conclusion

Differently from the identified related works, the MD2SEF specifies DEVS model specification at the CIM level, DEVS simulation protocol at the PIM level, parallel/distributed concepts (e.g., communication) at PDM level, and an integrative solution model as the PSM. It includes modeling languages and transformation techniques for the development thus;

- Ensuring that concerns involved in performing DEVS simulation engineering are clearly identified, specified, and formalized.
- Eliminating the need to have DEVS simulation applications which are tightly coupled to either a particular DEVS model, DEVS PDES strategy or a simulation platform

- Supporting the partial automation of the process of building parallel/distributed simulation strategy starting from a DEVS model to its execution.
- Providing a guide for the mapping of any specified DEVS model on any specified parallel/distributed platform

In Chapter 10, we present the summary of the current thesis.

10. General Conclusion

10.1. Introduction

This Chapter provides a summary of the thesis. Thereafter, the aim of this research is discussed and the way that the objectives, stated in Chapter 1, were met in order to achieve the aim. Subsequently, the contributions of the thesis are highlighted and discussed. Finally, the limitations of this work are discussed together with the further work. There are different aspects of this research and the contributions are mainly in the field of DEVS distributed simulation engineering, model driven frameworks, and agile development.

10.2. Research Summary

This research study proposes a unifying framework for engineering DEVS distributed simulations starting from the specification of DEVS models to obtaining simulation codes. The framework suggests four layers and their associated generic models that describe components necessary for developing a DEVS distributed simulation application. We show that the proposed framework is capable of integrating these components as a unified whole via model transformations. We show the feasibility of the framework by using a case study.

We introduced the Simulation Graph as an answer to the question of how to build and map a DEVS simulation strategy on to parallel/distributed infrastructures. This answer includes formalized concepts and a methodology consisting of a meta-model of the process of mapping a DEVS simulation tree onto a DEVS simulation graph. As such, it proffers an abstract way for integrating heterogeneous DEVS PDES implementation strategies. By providing, a systematic and quantifiable generic approach that can be instantiated to fill the gap identified from taking a DEVS simulation model specification to mapping it onto a parallel/distributed infrastructure. Thus, each instantiation provides a guideline to applying a specific strategy.

10.3. Research objectives

The aim of this research was to provide a unifying framework that supports the process of mapping any domain specific system specified as a DEVS model on to any parallel/distributed infrastructures and hence obtain simulation codes. In doing so, the following research questions were addressed:

How do we map any DEVS simulation model on to any parallel/distributed infrastructure?

To achieve the aim of this thesis and to address the research questions, three objectives were identified. These objectives were met through the Chapters of this thesis as follows:

Objective 1: To provide support for the specification of DEVS PDES strategies

In Chapter 2, we looked at the underlying research context that establishes the research aim. It provided the theoretical background and the literature on all the relevant aspects of the proposed framework. Also, in the same Chapter were reviews of existing solutions to building DEVS implementation strategies. Sub-objectives were developed as a result of the knowledge extracted from these reviews.

Objective 1.1: To provide a conceptual understanding and specification of concepts found when developing a DEVS PDES implementation strategy

Chapter 3 provided an approach to guide the understanding of the concepts and procedures involved in a DEVS PDES development process. The approach is centered around the DEVS simulation protocol.

Objective 1.2: To specify a generic methodology for the development of DEVS PDES strategies.

Chapter 4 provided formal definitions and a methodology for the concepts identified in Chapter 3. The methodology proposes a guideline with which a DEVS implementation strategy may be specified. In addition, it defines the Simulation Structure Layer of the MD2SEF in Section 6.3.

Objective 2: To provide support for the modeling and migration of DEVS models.

This objective was achieved in the span of two chapters i.e., Chapters 5 and 6. Chapter 5 describes the integrative and unifying framework that includes a layer for specifying DEVS models i.e., the Simulation Modeling Layer. Section 6.2 illustrates how this specification has been achieved.

Objective 3: To specify a model that has an extended simulation architectural scope i.e. allows the integration and migration of parallel/distributed infrastructure.

The Middleware layer of the MD2SEF was defined to meet this objective. The specification of a model that has an extended architectural scope was presented in Section 6.4.

Objective 4: To specify a unifying framework that enables DEVS distributed simulation engineering.

Chapter 5 introduced the MD2SEF framework while its layers were described in Chapter 6. Chapter 7 describes how a layer is integrated with another below it in an effort to present the MD2SEF as a unified whole fit for engineering DEVS distributed simulations.

10.4. Main Contributions (pros)

We have shown that it is possible to have a generic framework that supports the process of mapping a domain specific system specified as a DEVS model on to parallel/distributed infrastructures using MDA methods and tools to obtain executable simulation codes.

There are different practices behind the concept of exploiting DEVS in terms of the model construction, the simulation language implementations, algorithmic code expressions and many more. As a result, it gives rise to heterogeneous DEVS PDES strategies. This therefore complicates the development, sharing and execution of models on single or multi-processor infrastructure. A DEVS standardization group was set up to look into this. This group identified three main issues to tackle; define a “DEVS kernel”, i.e. the set of minimum requirements a tool should fulfill to be labeled “DEVS-compliant”; make existing DEVS tool interoperate, e.g. by relying on standard interfaces and some underlying communication protocol; and come up with a standardized format for representing DEVS models, to facilitate their exchange between scientists and their use in different tools (G. A. Wainer et al., 2010).

(Adegoke et al., 2013) proposes a contribution to the standardization effort. The proposal is a framework that provides a means of specifying, comparing and contrasting various implementations as well as a standardized platform for all DEVS implementation strategies. It contributes by providing a clear guideline and a reference framework for specifying any DEVS simulation implementation strategy. These provisions and contributions have been defined as Simulation Graph. We build on this work by providing a complete and integrative solution that includes definitions for a DEVS model, Simulation Structures, middleware and tool support

We identify that a DEVS PDES implementation strategy differs from another. However, it should be noted here that the definitions of the components used in Simulation Graph (SG) construction are at an abstract level. The Links and Nodes in a SG 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 (if simulators are implemented in different languages). The only constraint defined is that the simulators implement DEVS simulation algorithm. Thus, the issue of DEVS implementation differences can be overcome.

The SSL models the process of mapping a DEVS simulation tree to a graph of simulation components distributed over a network. Then, each builder of DEVS distributed simulation can instantiate this generic model to get his own mapping strategy. It can therefore serve as a guideline for any user to when building a strategy. It consists of identified simulation structures, simulation operations that are used for mapping these structures and well-defined formal specifications for the structures and operations. These specifications may be used to remove ambiguity, reduce accidental complexity (i.e., wrong implementation due to misunderstanding of concepts), for consistency or validity checking and ease the automation of simulation components and operations. The simulation structures and operations have been identified as being generic in various DEVS implementation strategies. A crucial point is that the approach is independent of any DEVS implementation. Thus at the implementation level, MD2SEF can reuse one of the several existing DEVS platforms of the literature.

The Simulation Modeling Layer and the Middleware Layer were also included to identify the MD2SEF as a complete framework that can enable the engineering of DEVS distributed simulations. They capture the specification of a DEVS model and middleware respectively while the Simulation Structure Layer is for defining DEVS PDES strategies. Each of the layers was defined independently thereby supporting modularity and reusability while their integration was achieved using MDA techniques. The MD2SEF, as a result, contributes the flexibility needed to support the migration of DEVS models, DEVS PDES implementation strategies, or middleware platforms that may arise in the event of unanticipated changes.

10.5. Limitations (cons)

In this work, our aim was to provide a guideline by which any DEVS model can be mapped on to any high performance platform. We focused on identifying and separating the concerns found in this mapping process. One limitation of this framework is that it is only suitable for use in cases where synchronization mechanisms can be separated from the communication mechanism

When using the unifying framework, a user is allowed to define their own variant of the simulation protocol provided it follows the proposed procedures and the send-receive pattern. A consequence of this freedom is the introduction of heterogeneous behavioral definitions for any Node on the Simulation Tree. The framework is limited such that it does not provide a way to validate or verify the behavioral conformity of this user-defined variant to the standard DEVS simulation protocol.

MD2SEF is not sufficient to be described as an infrastructure that allows the integration of several simulation formalisms due to the framework being tightly coupled with DEVS. This places a limit to the comparison of MD2SEF with other distributed environments as well as a limit to the integration of other kinds of formalisms into MD2SEF. In this case, Distributed Simulation Engineering and Execution Process (DSEEP) (Xplore & IEEE Computer Society, 2011) can be used as the support whereby similar steps taken to develop MD2SEF can be replicated or compared with other frameworks built with DEVS.

In this work, MDA has been used to align MD2SEF as a framework that supports DEVS simulation engineering. The CIM, PIM, PDM, and PSM levels and the transformations between them were developed as plug-ins and integrated into the Eclipse platform. MD2SEF has an extended simulation architectural scope i.e. allows the integration of different middleware in the development process. A limitation with the MD2SEF is that it is assumed that only one kind of middleware can be in use per time. Although it is possible to identify suitable architectures, however, only one can be used at a time. A future work can be the integration of an HLA abstract model into the MD2SEF or possibly a PDM that describes the combination of CORBA and HLA. This can be interesting considering that the absence of adequate technical solutions in one is fixed in the other (Buss & Jackson, 1998; D Ambrogio & Gianni, 2004).

10.6. Perspectives (future work)

10.6.1. Patterns in Simulation Graph Operations

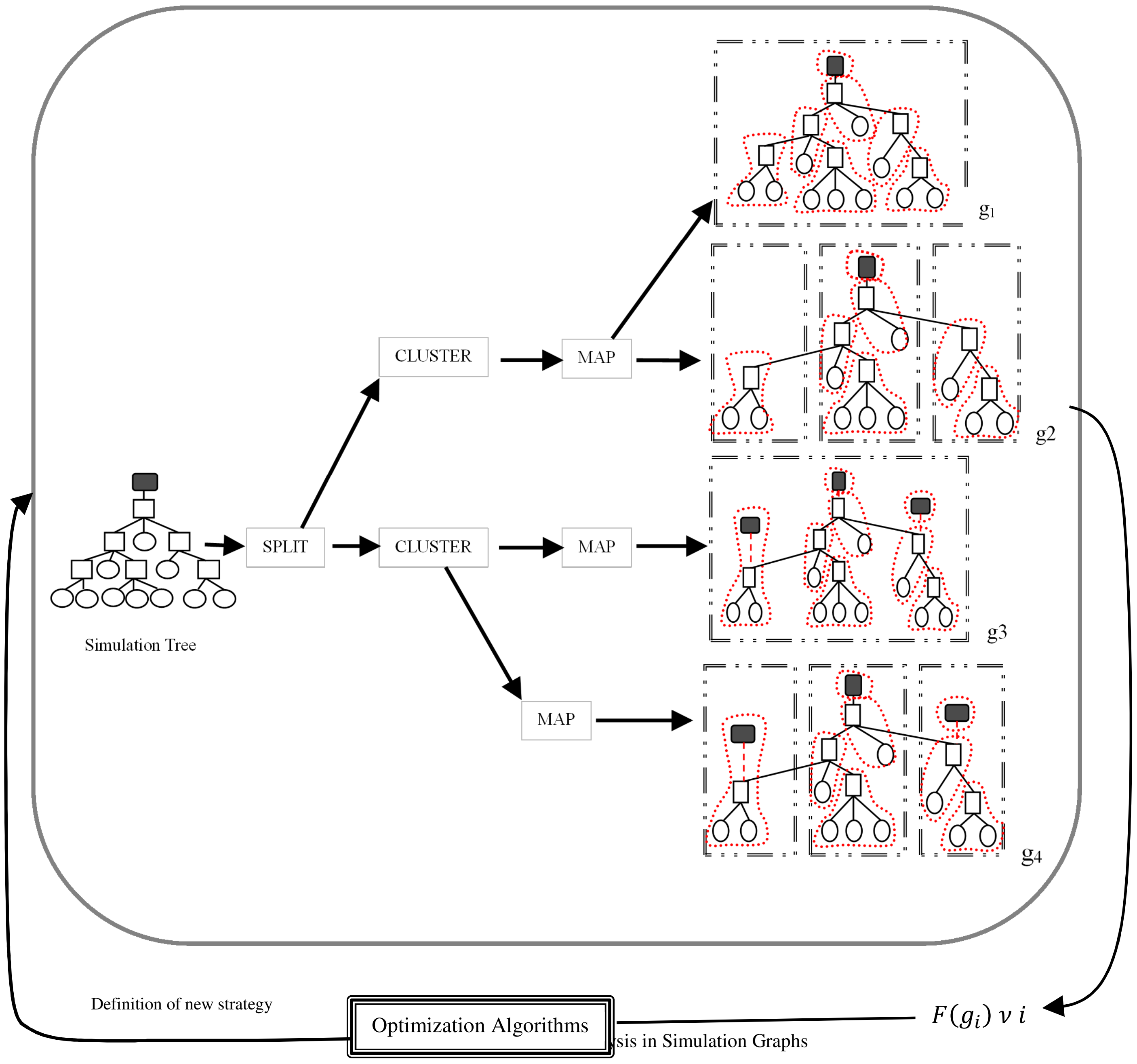
In literature, several algorithms exist to achieve the operations used in developing Simulation Graph strategies. We can site: Simulation Tree flattening and expansion algorithms (Chen & Vangheluwe, 2010; Zacharewicz & Hamri, 2007); Simulation Tree splitting approaches (Ezequiel Glinsky & Wainer, 2006; Liu & Wainer, 2007); Simulation Nodes clustering approaches (Himmelspach & Uhrmacher, 2006); and Simulation Graph mapping (Park et al., 2006). It will be interesting to find in each of the operations common patterns that can be standardized and used to enhance the Simulation Structures and operations.

10.6.2. Model Composition Techniques

Merging the PIM and PDM meta-models require some form of equivalence relation between the meta-models. In this work we considered merging using class equivalence (i.e. class name) however this merging procedure is limited to the level of classes. It will be necessary to consider a more generic situation whereby a user of the MD2SEF framework will want to merge the given PIM with a differently specified PDM using any kind of equivalence relation.

10.6.3. Performance Analysis in Simulation Graphs

Simulation graph methodology offers different strategies for mapping a ST to a SG and proffers the potential of having each of the strategy analyzed. Figure 10-1 shows a series of paths that can be taken to build a SG from a ST. These paths are some of the mapping strategies found in the presented methodology. A path (Split, Cluster, and Map) can be considered as not being optimal if another path produces a better performance after comparison. Therefore, the use of any efficient model-partitioning algorithm, clustering techniques or mapping heuristics in the MD2SEF can be evaluated. However, performance analysis or optimization is not the focus of this work. Rather, this work's objective is to provide a way for integrating heterogeneous mapping strategies in a framework as well as to automate (i.e., using software engineering methods) such a process. Thus, performance evaluation (in terms of time, memory and so on) of each of the strategies can be considered as a further work.



Simulation Tree
SPLIT
CLUSTER
MAP
CLUSTER
MAP
MAP
g1
g2
g3
g4
Definition of new strategy
Optimization Algorithms
$F(g_i)\ \nu\ i$
ysis in Simulation Graphs

References

Acceleo. (2015). Acceleo MTL. Retrieved November 1, 2017, from http://www.acceleo.org/

Adegoke, A. (2010). *Efficient Object Oriented Implementations For The Devs Formalism*. African University of Science and Technology.

Adegoke, A., Togo, H., & Traore, M. K. (2013). A unifying framework for specifying DEVS parallel and distributed simulation architectures. *Simulation-Transactions of the Society For Modeling and Simulation International*, *89*(11), 1293–1309. https://doi.org/10.1177/0037549713504983

Bae, J. W., Bae, S. W., Moon, I.-C., & Kim, T. G. (2016). Efficient Flattening Algorithm for Hierarchical and Dynamic Structure Discrete Event Models. *ACM Transactions on Modeling and Computer Simulation*, *26*(4), 1–25. https://doi.org/10.1145/2875356

Barros, F. J. (1995). Dynamic Structure Discrete Event System Specification: A New Formalism for Dynamic Structure Modeling and Simulation. *Proceedings of the 1995 Winter Simulation Conference*, 781–785. https://doi.org/10.1145/224401.224731

Bazoun, H., Zacharewicz, G., Ducq, Y., & Boyé, H. (2014). SLMToolBox: An Implementation of MDSEA for Servitisation and Enterprise Interoperability. *Enterprise Interoperability VI*, *7*, 101–111. https://doi.org/10.1007/978-3-319-04948-9_9

Bernstein, P. (1996). Middleware: a model for distributed system services. *Communications of the ACM*.

Bézivin, J., Dupé, G., Jouault, F., Pitette, G., & Rougui, J. E. (2003). First experiments with the ATL model transformation language : Transforming XSLT into XQuery. *2nd OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture.*, (1).

Bézivin, J., & Ploquin, N. (2001). Combining the Power of Meta-Programming and Meta-Modeling in the OMG. In *MDA Framework, OMG's 2nd Workshop on UML for Enterprise Applications*.

Biehl, M. (2010). Literature study on model transformations. *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, (July), 1–28.

Boas, G. van E. (2004). Template Programming for Model-Driven Code Generation. *19th Annual ACM SIGPLAN Conference on Object-*.

Bolduc, J.-S. J.-S., & Vangheluwe, H. (2002). *A Modelling and Simulation Package for Classical Hierarchical DEVS*. McGill University, School of Computer Science.

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., & Orchard, D. (2004). Web Services Architecture. Retrieved November 1, 2017, from https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

Booth, D., & Liu, C. K. (2007). Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. Retrieved from https://www.w3.org/TR/wsdl20-primer/

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition) W3C Recommendation. Retrieved January 1, 2015, from https://www.w3.org/TR/2008/REC-xml-20081126/

Bryant, R. E. (1977). Simulation of Packet Communication Architecture Computer Systems. *Tr-188*.

Buss, A., & Jackson, L. (1998). Distributed simulation modeling: a comparison of HLA, CORBA, and RMI. *1998 Winter Simulation Conference. Proceedings (Cat.*

No.98CH36274), *1*(May), 819–825. https://doi.org/10.1109/WSC.1998.745071

Cetinkaya, D., Verbraeck, A., & Seck, M. (2012). Model transformation from BPMN to DEVS in the MDD4MS framework. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium* (p. 28:1-28:6). Society for Computer Simulation International.

Cetinkaya, D., Verbraeck, A., & Seck, M. D. (2011). MDD4MS: a model driven development framework for modeling and simulation. *SCSC '11 Proceedings of the 2011 Summer Computer Simulation Conference*, (June), 113–121.

Cetinkaya, D., Verbraeck, A., & Seck, M. D. M. (2015). Model Continuity in Discrete Event Simulation: A Framework for Model-Driven Development of Simulation Models. *ACM TRANSACTIONS ON MODELING AND COMPUTER SIMULATION*, *25*(3), 17.

Chandy, K. M., & Misra, J. (1979). Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, *SE-5*(5), 440–452. https://doi.org/10.1109/TSE.1979.230182

Chen, B., & Vangheluwe, H. (2010). Symbolic Flattening of DEVS Models. *Proceedings of the 2010 Summer Computer Simulation Conference*, 209–218.

Cheon, S., Seo, C., Park, S., & Zeigler, B. P. (2004). Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Networked System. In *Advanced Simulation Technologies Conference*.

Cho, Y. K., Zeigler, B. P., & Sarjoughian, H. S. (2001). Design and implementation of distributed real-time DEVS/CORBA. In *2001 IEEE International Conference on Systems, Man and Cybernetics* (Vol. 5, pp. 3081–3086). https://doi.org/10.1109/ICSMC.2001.971989

Chow, A. C. H., & Zeigler, B. P. (1994). Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of Winter Simulation Conference* (pp. 716–722). https://doi.org/10.1109/WSC.1994.717419

Christensen, E. (1990). Hierarchical optimistic distributed simulation: Combining DEVS and Time Warp.

Clark, T., Sammut, P., & Willans, J. (2008). Applied metamodelling: a foundation for language driven development. *Book To Be Published*. https://doi.org/10.1016/j.jbusres.2014.06.013.The

Czarnecki, K., & Helsen, S. (2003). Classification of Model Transformation Approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, *45*(3), 1–17. https://doi.org/10.1147/sj.453.0621

Czarnecki, K., & Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, *45*(3), 621–645. https://doi.org/10.1147/sj.453.0621

D'Ambrogio, A., Gianni, D., Risco-Martín, J., Pieroni, A., Risco-Martin, J. L., & Pieroni, A. (2010). A MDA-based approach for the development of DEVS/SOA simulations. In *Spring Simulation Multiconference 2010, SpringSim'10*. Society for Computer Simulation International. https://doi.org/10.1145/1878537.1878685

D Ambrogio, A., & Gianni, D. (2004). Using CORBA to Enhance HLA Interoperability in Distributed and Web-Based Simulation. *Lecture Notes in Computer Science*, 696–705.

De Lara, J., & Vangheluwe, H. (2002). Atom3: A tool for multi-formalism and meta-modelling. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 2306, pp. 174–188). Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45923-5_12

Dijkstra, E. W. (1976). *A discipline of programming*. Upper Saddle River, NJ: Prentice-Hall.

Downing, T., & Bryan, T. (1998). *Java RMI: remote method invocation*. IDG Books Worldwide.

Filippi, J.-B. B., & Bisgambiglia, P.-A. (2004). JDEVS: an implementation of a DEVS based formal framework for environmental modelling. *Environmental Modelling & Software*, *19*(3), 261–274. https://doi.org/10.1016/j.envsoft.2003.08.016

Frame, J. D., Baldwin, J. M., Gocke, D. J., & Troup, J. M. (1970). Lassa fever, a new virus disease of man from West Africa. I. Clinical description and pathological findings. *American Journal of Tropical Medicine and Hygiene*, *19*(4), 670–676.

Franceschini, R., & Bisgambiglia, P.-A. (2014). Decentralized Approach for Efficient Simulation of Devs Models. In *IFIP International Conference on Advances in Production Management Systems (APMS)* (pp. 336–343). Springer. https://doi.org/10.1007/978-3-662-44733-8_42

Franceschini, R., Bisgambiglia, P.-A., Touraille, L., Bisgambiglia, P., & Hill, D. (2014). A survey of modelling and simulation software frameworks using Discrete Event System Specification. *2014 Imperial College Computing Student Workshop*, 40–49. https://doi.org/10.4230/OASIcs.ICCSW.2014.40

Fuentes-Fernández, L., & Vallecillo-Moreno, A. (2004). An Introduction to UML Profiles. *European Journal for the Informatics Professional*, *V*(2), 6–13. https://doi.org/10.1016/B978-0-12-416619-6.00002-X

Fujimoto, R. M. (2001). Parallel and distributed simulation systems. In *Simulation Conference, 2001. Proceedings of the Winter* (Vol. 1, pp. 147–157). https://doi.org/10.1109/WSC.2001.977259

Gamar, J., & Agrawal, B. S. (2017). Business Process Simulation: Transformation of BPMN 2.0 to Discrete Event System Specification. *International Journal on Recent and Innovation Trends in Computing and Communication*, *5*(6).

Garredu, S., Vittori, E., Santucci, J.-F., & Poggi, B. (2014). A Survey of Model-Driven Approaches Applied to DEVS - A Comparative Study of Metamodels and Transformations. *Proceedings of the 4th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, 179–187. https://doi.org/10.5220/0005041001790187

Glinsky, E. (2005). *New Parallel Simulation Techniques of DEVS and Cell-DEVS in CD++*. Carleton University, Ottawa.

Glinsky, E., & Wainer, G. (2002). Performance analysis of real-time DEVS models. In *Proc. Winter Simulation Conf* (Vol. 1, pp. 588–594). https://doi.org/10.1109/WSC.2002.1172935

Glinsky, E., & Wainer, G. (2006). New parallel simulation techniques of DEVS and cell-DEVS in CD++. In *Proceedings - Simulation Symposium* (Vol. 2006, pp. 244–251). IEEE Computer Society. https://doi.org/10.1109/ANSS.2006.32

Gonzalez, A., Luna, C., Cuello, R., Perez, M., & Daniele, M. (2015). Towards an automatic model transformation mechanism from UML state machines to DEVS models, *18*(2), 1–27.

Günther, S., Emmerich, P., Laue, T., Kühle, O., Asper, M., Jung, A., … Schmitz, H. (2000). Imported Lassa fever in Germany: Molecular characterization of a new Lassa virus strain. *Emerging Infectious Diseases*, *6*(5), 466–476. https://doi.org/10.3201/eid0605.000504

Hailpern, B., & Tarr, P. (2006). Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, *45*(3), 451–461. https://doi.org/10.1147/sj.453.0451

Himmelspach, J., Ewald, R., Leye, S., & Uhrmacher, A. M. (2007). Parallel and distributed simulation of Parallel DEVS models. In *Proceedings of the 2007 spring simulation multiconference* (Vol. 1, pp. 249–256).

Himmelspach, J., & Uhrmacher, A. M. (2006). Sequential Processing of PDEVS Models. In *Proceedings of the 3rd European Modeling \& Simulation Symposium (EMSS)* (pp. 239–244).

Hong, J. S., Song, H.-S., Kim, T. G., & Park, K. H. (1997). A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development. *Discrete Event Dynamic Systems: Theory and Applications*, *7*(4), 355–375. https://doi.org/10.1023/A:1008262409521

Hu, J., Huang, L., Wu, R., Cao, B., & Chang, X. (2014). Model-Driven Design and Validation of Service Oriented Architecture Based on DEVS Simulation Framework. *International Journal of Services Computing (IJSC)*, *2*(2), 17–29.

Jafer, S., & Wainer, G. (2009). Flattened Conservative Parallel Simulator for DEVS and CELL-DEVS. *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 01*, 443–448. https://doi.org/10.1109/cse.2009.52

Jafer, S., & Wainer, G. (2010). Conservative DEVS - A novel protocol for parallel conservative simulation of DEVS and cell-DEVS models. In *Simulation Series* (Vol. 42, pp. 168–175). https://doi.org/10.1145/1878537.1878683

Janoušek, V., Polášek, P., & Slavíček, P. (2006). Towards DEVS meta language. In *Proceedings of the 38th Winter Simulation Conference* (pp. 69–73).

Jefferson, D. R. (1985). Virtual time. *ACM Transactions on Programming Languages and Systems*, *7*(3), 404–425. https://doi.org/10.1145/3916.3988

Jouaullt, F. (2009). Model Transformation with ATL. *International Workshop MtATL 2009*, (March).

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*, *72*(1–2), 31–39. https://doi.org/10.1016/j.scico.2007.08.002

Kermack, W. O., & McKendrick, a. G. (1927). Contributions to the mathematical theory of epidemics. *Proceedings of the Royal Society of London*, *115*(772), 700–721. https://doi.org/10.1098/rspa.1927.0118

Kihyung Kim, Wonseok Kang, Bong Sagong, & Hyungon Seo. (2000). Efficient distributed simulation of hierarchical DEVS models: transforming model structure into a non-hierarchical one. In *Proceedings 33rd Annual Simulation Symposium (SS 2000)* (pp. 227–233). IEEE Comput. Soc. https://doi.org/10.1109/SIMSYM.2000.844920

Kim, K. H., Seong, Y. R., Kim, T. G., & Park, K. H. (1996). Distributed Simulation of Hierarchical DEVS Models: Hierarchical Scheduling Locally and Time Warp Globally. *Transactions of the Society for Computer Simulation International*, *13*(3), 135–154.

Kim, K., & Kang, W. (2004). CORBA-based, multi-threaded distributed simulation of hierarchical DEVS models: Transforming model structure into a non-hierarchical one. In *International Conference on Computational Science and its Applications* (pp. 167–176). Springer Berlin Heidelberg.

Kim, T., Sung, C., Hong, S., & Hong, J. (2010). DEVSim++ toolset for defense modeling and simulation and interoperation. *The Journal of.*

Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. *AddisonWesley Professional* (Vol. 83). https://doi.org/10.1016/S0031-9406(05)65759-8

Krakowiak, S. (2007). *Middleware Architecture with Patterns and Frameworks*. INRIA Rhône-

Alpes, France. https://doi.org/10.1.1.140.3783

Krishna, A. S., Schmidt, D. C., Klefstad, R., & Corsaro, A. (2005). Real-Time CORBA Middleware. In *Middleware for Communications* (pp. 413–438). Chichester, UK: John Wiley & Sons, Ltd. https://doi.org/10.1002/0470862084.ch17

Kwon, Y. W., Park, H. C., Jung, S. H., & Kim, T. G. (1996). Fuzzy-DEVS Formalism: Concepts, Realization and Applications. In *Proceedings of the 1996 Conference on AI, Simulation and Planning in High Autonomy Systems* (pp. 227–234).

Lee, W., & Kim, T. (2003). Simulation Speedup for DEVS Models By Composition-based Compilation. *Summer Computer Simulation 2003*, 395–400.

Lei, Y., Wang, W., Li, Q., & Zhu, Y. (2009). A transformation model from DEVS to SMP2 based on MDA. *Simulation Modelling Practice and Theory*, *17*(10), 1690–1709. https://doi.org/10.1016/J.SIMPAT.2009.08.003

Liu, Q., & Wainer, G. (2007). Parallel Environment for DEVS and Cell-DEVS Models. *Simulation*, *83*(6), 449–471. https://doi.org/10.1177/0037549707085084

Mens, T., & Van Gorp, P. (2006). A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, *152*(1–2), 125–142. https://doi.org/10.1016/j.entcs.2005.10.021

Misra, J. (1986). Distributed discrete-event simulation. *ACM Computing Surveys (CSUR)*, *18*(1), 39–65. https://doi.org/10.1145/6462.6485

Mitra, N., & Lafon, Y. (2007). SOAP Version 1.2 Part 0: Primer (Second Edition). Retrieved September 14, 2017, from https://www.w3.org/TR/soap12-part0/

Mittal, S., & Martin, J. L. R. (2013). Model-driven systems engineering for netcentric system of systems with DEVS unified process. *Proceedings of the 2013 Winter Simulation Conference - Simulation: Making Decisions in a Complex World, WSC 2013*, (July), 1140–1151. https://doi.org/10.1109/WSC.2013.6721503

Mittal, S., Risco-Martín, J. L. J., & Zeigler, B. P. B. (2007). DEVSML: automating DEVS execution over SOA towards transparent simulators. *Proceedings of the 2007 Spring Simulation Multiconference*, *2*(August 2015), 1–9. https://doi.org/10.1145/1404680.1404725

Mittal, S., Risco-Martín, J. L., & Zeigler, B. P. (2009). DEVS / SOA : A Cross-Platform Framework for Net-centric Modeling & Simulation in DEVS Unified Process. *Simulation*, *85*(7), 1–35. https://doi.org/10.1177/0037549709340968

Moore, B., Dean, D., Gerber, A., Wagenknecht, G., & Vanderheyden, P. (2004). Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework. *IBM Redbooks*, *1*, 1–256. https://doi.org/10.1147/JRD.2010.2041693

Muzy, A., & Nutaro, J. J. (2005). Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. *In 1st Open International Conference on Modeling & Simulation, OICMS*, 273--279.

Nutaro James. (n.d.). ADEVS. Retrieved August 6, 2014, from http://web.ornl.gov/~1qn/adevs/index.html

Object Management Group (OMG). (n.d.). Common Object Request Broker Architecture (CORBA) Specification. Retrieved November 1, 2017, from http://www.omg.org/spec/CORBA/2.6.1/

Object Management Group, Version, M. D. a G., Kennedy, A., Carter, K., & Technologies, W. F. X. (2003). MDA Guide Version 1.0.1. *Object Management Group*, *234*(June), 51. https://doi.org/10.1074/jbc.M312687200

Omg. (2006). Meta Object Facility (MOF) Core Specification. *Management*, *80907*(January), 1–76. https://doi.org/citeulike-article-id:1557124

OMG. (2014). OMG MDA Guide rev. 2.0. *OMG Document Ormsc*, *2.0*(June), 1–15.

Park, S., Hunt, C. A., & Zeigler, B. P. (2006). Cost-based Partitioning for Distributed and Parallel Simulation of Decomposable Multiscale Constructive Models. *SIMULATION*, *82*(12), 809–826. https://doi.org/10.1177/0037549706075479

Park, S., Kim, S. H. J., Hunt, C. A., & Park, D. (2007). DEVS Peer-to-Peer Protocol for Distributed and Parallel Simulation of Hierarchical and Decomposable DEVS Models. In *2007 International Symposium on Information Technology Convergence (ISITC 2007)* (pp. 91–95). IEEE. https://doi.org/10.1109/ISITC.2007.47

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058. https://doi.org/10.1145/361598.361623

Praehofer, H., Sametinger, J., & Stritzinger, A. (2001). Concepts and architecture of a simulation framework based on the JavaBeans component model. *Future Generation Computer Systems*, *17*(5), 539–559. https://doi.org/10.1016/S0167-739X(00)00038-8

Rao, D., Thondugulam, N., & Radhakrishnan, R. (1998). Unsynchronized parallel discrete event simulation. In *Proceedings of the 30th conference on Winter simulation*. IEEE Computer Society Press.

Rumbaugh , James; Jacobson, Ivar; Booch, G. (1999). *The Unified Modeling Language Reference Manual. New York: Addison-Wesley*. New York: Addison-Wesley.

Sarjoughian, H., Kim, S., Ramaswamy, M., & Yau, S. (2008). A simulation framework for service-oriented computing systems. *2008 Winter Simulation Conference*, (Acims 2001), 845–853. https://doi.org/10.1109/WSC.2008.4736148

Sarjoughian, H. S., Alshareef, A., & Lei, Y. (2015). Behavioral DEVS metamodeling. In *2015 Winter Simulation Conference (WSC)* (pp. 2788–2799). IEEE. https://doi.org/10.1109/WSC.2015.7408384

Seck, M., & Verbraeck, A. (2009). DEVS in DSOL: adding devs operational semantics to a generic event-scheduling simulation environment. *Proceedings of the 2009 Summer Computer Simulation Conference*. Society for Modeling & Simulation International.

Seo, C., Park, S., Kim, B., Cheon, S., & Zeigler, B. P. (2004). Implementation of Distributed high-performance DEVS Simulation Framework in the Grid Computing Environment. In *Advanced Simulation Technologies Conference–Design, Analysis, and Simulation of Distributed Systems Symposium* (pp. 9–15). SCS.

Seo, C., & Zeigler, B. P. (2009). Automating the DEVS Modeling and Simulation Interface to Web Services. *Proceedings of the 2009 Spring Simulation Multiconference*.

Shiginah, F. B. (2006). *Multi-Layer Cellular DEVS Formalism for Faster Model Development and Simulation Efficiency*. University of Arizona.

Sogoba, N., Feldmann, H., & Safronetz, D. (2012). Lassa Fever in West Africa: Evidence for an Expanded Region of Endemicity. *Zoonoses and Public Health*. https://doi.org/10.1111/j.1863-2378.2012.01469.x

Stahl, T., & Voelter, M. (2006). *Model-Driven Software Development: Technology, Engineering, Management. John Wiley and Sons ISBN9780470025703*.

Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., & Paternostro, M. (2008). *EMF: eclipse modeling framework. Engineering*. https://doi.org/10.1108/02641610810878585

Tolk, A., & Muguira, J. A. (2004). M&S within the model driven architecture. *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*, *2004*(1), 1–13.

Van Tendeloo, Y., & Vangheluwe, H. (2017). An evaluation of DEVS simulation tools. *SIMULATION*, *93*(2), 103–121. https://doi.org/10.1177/0037549716678330

Vangheluwe, H. (2000). DEVS as a common denominator for multi-formalism hybrid systems modelling. In *CACSD. Conference Proceedings. IEEE International Symposium on Computer-Aided Control System Design (Cat. No.00TH8537)* (pp. 2–7). https://doi.org/10.1109/CACSD.2000.900199

Viyovic, V., Maksimovic, M., & Perisic, B. (2014). Sirius: A rapid development of DSM graphical editor. In *INES 2014 - IEEE 18th International Conference on Intelligent Engineering Systems, Proceedings* (pp. 233–238). IEEE. https://doi.org/10.1109/INES.2014.6909375

Wainer, G. (2002). CD++: A toolkit to develop DEVS models. *Software Practice and Experience*, *32*(13), 1261–1306. https://doi.org/10.1002/spe.482

Wainer, G. (2009). *Discrete-event modeling and simulation: a practitioner's approach. Design*. https://doi.org/10.1201/9781420053371.ch12

Wainer, G. A., Al-Zoubi, K., Hill, D. R. C., Mittal, S., Risco-Martín, J., Sarjoughian, H., … P., Z. B. (2010). An Introduction to DEVS Standardization,. In Wainer G. A. & P. Mosterman (Eds.), *Discrete-Event Modeling and Simulation: Theory and Applications*. Taylor & Françis.

World Health Organization, W. H. O. (2017). Emergencies preparedness, response. Retrieved March 20, 2017, from http://www.who.int/csr/don/27-january-2016-lassa-fever-nigeria/en/

WSC. (n.d.). XSL Transformations (XSLT). Retrieved January 1, 2015, from https://www.w3.org/TR/1999/REC-xslt-19991116

Xplore, I., & IEEE Computer Society. (2011). IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP). *IEEE Std 1730-2010 (Revision of IEEE Std 1516.3-2003)*, *2010*(January), 1–79. https://doi.org/10.1109/IEEESTD.2011.5706287

Zacharewicz, G., & Hamri, M. E.-A. (2007). Flattening G-DEVS / HLA structure for Distributed Simulation of Workflows. In *AIS-CMS International modeling and simulation multiconference* (pp. 11–16). Buenos Aires, Argentina.

Zeigler, B., & Muzy, A. (2017). From Discrete Event Simulation to Discrete Event Specified Systems (DEVS). In *Conference invitee International Federation of Automatic Control (IFAC)* (pp. 9–14). Toulouse, France.

Zeigler, B. P. (1990). *Object-Oriented Simulation with Hierarchical, Modular Models. Object-Oriented Simulation with Hierarchical, Modular Models*. https://doi.org/10.1016/B978-0-12-778452-6.50005-4

Zeigler, B. P. (1998). The DEVS / HLA Distributed Simulation Environment And Its Support for Predictive Filtering. *Computer Engineering*, (September).

Zeigler, B. P., Moon, Y., Kim, D., & Kim, J. G. (1996). DEVS-C++: a high performance modelling and simulation environment. In *Proc. Twenty-Ninth Hawaii Int System Sciences Conf.* (Vol. 1, pp. 350–359). https://doi.org/10.1109/HICSS.1996.495481

Zeigler, B. P., Praehofer, H., & Kim, T. G. (2000). *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems* (2nd ed.). Academic Press.

Zeigler, B. P., Sarjoughian, H., & Praehofer, H. (2000). Theory of quantized systems: DEVS simulation of perceiving agents. *Cybernetics and Systems*, *31*(6), 611–647. https://doi.org/10.1080/01969720050143175

Zhang, M., Zeigler, B. P., & Hammonds, P. (2006). DEVS/RMI - An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies. *ITEA Journal Of Test And Evaluation*.

Zhang, M., Zeigler, B. P., Hammonds, P., & Raj, J. (2006). DEVS/RMI-A dynamic and flexible distributed simulation environment. In *Simulation Series* (pp. 79–85).

Appendices

Appendix A Send – Receive Message Pattern

Here, we make use of the pattern as discussed in Section 3.2 to describe PDEVS (Bernard P. Zeigler et al., 2000).

Message	Simulator	Coordinator
Receive i	$tl = t - e$ $tn = tl + ta(s)$	for each $d \in D$ do **Send i-message** to child d Sort event-list according to tn_d $tl = \max(tl_d \mid d \in D)$ $tn = \min(tn_d \mid d \in D)$
Receive *	if $t = tn$ then $y = \lambda(s)$ **Send y-message** (y,t) to parent coordinator	if $t \mathrel{!=} tn$ then error : bad synchronization $IMM = (d \mid (d, tn_d) \in$ (event-list & $tn_d = tn$)) for each $r \in IMM$ **Send *-messages** to r
Receive x	if ($x = \emptyset$ and $t = tn$) then $s = \delta int(s)$ else if ($x \mathrel{!=} \emptyset$ and $t = tn$) then $s = \delta conf(s, x)$ else if ($x \mathrel{!=} \emptyset$ and ($tl \leq t \leq tn$)) $e = t - tl$ $s = \delta ext(s, e, x)$ $tl = t$ $tn = tl + ta(s)$	if $!(tl \leq t \leq tn)$ then error : bad synchronization receivers = ($r \mid r \in$ children, $N \in I_r$, $Z_{N,r}(x) \mathrel{!=} \emptyset$ for each r in receivers **Send x-message** ($Z_{N,r}(x)$, t) with input value $Z_{N,r}(x)$ to r for each $r \in IMM$ and not in receivers **Send x-message** ($\emptyset$, t) to r sort event-list according to tn_d $tl = t$ $tn = \min(tn_d \mid d \in D)$
Receive y		if this is not the last d in IMM add (y_d, d) to mail mark d as reporting else if this is the last d in IMM $y_{parent} = \emptyset$ for each $d \in I_N$ & d is reporting if $Z_{d,N}(y_d) \neq \emptyset$ then

```
add $y_d$ to yparent
Send y-message (yparent, t) to parent
for each child r with some $d \in Ir$ & d is reporting & $Z_{d,r}$
 $y_d \neq \emptyset$
for each child r with some $d \in Ir$ & d is reporting & $Z_{d,r}(y_d) \neq \emptyset$
 add $Z_{d,r}(y_d) \neq \emptyset$ to $y_r$
Send x-messages ($y_r$, t) to r
for each $r \in IMM$ & $y_r = \emptyset$
 Send x-messages ($\emptyset$, t) to r
tl = t
tn = min ($tn_d$ | $d \in D$)
sort event-list according to $tn_d$
```

Appendix B CIM Metamodel in OCLinEcore

```
import ecore : 'http://www.eclipse.org/emf/2002/Ecore' ;

package Model : model = 'http://model.org'
{
	abstract class DEVS
	{
		attribute name : String[1];
		property container : DEVS[?];
		property iports : IPort[*] { composes };
		property oports : OPort[*] { composes };
	}
	class AtomicDEVS extends DEVS
	{
		property deltaInt : IntTransition[*] { composes };
		property deltaConf : ConfTrans[*] { composes };
		property deltaExt : ExtTrans[*] { composes };
		property phases : Phase[+] { composes };
		property stateVars : Variable[*] { composes };
	}
	class CoupledDEVS extends DEVS
	{
		property subModels : DEVS[+] { composes };
		property eics : EIC[*] { composes };
		property ics : IC[*] { composes };
		property eocs : EOC[*] { composes };
	}
	class IPort extends Port;
	class OPort extends Port;
	abstract class Port
	{
		attribute portId : String[1];
		property owner : DEVS[1];
		property portType : ecore::EClassifier[1];
	}
	class EIC
	{
        property influencer : IPort[1];
        property influencee : IPort[1];
}
class IC
{
        property influencer : OPort[?];
        property influencee : IPort[?];
}
class EOC
{
        property influencer : OPort[?];
        property influencee : OPort[?];
}
class IntTransition extends PhaseTransition
{
        property outputs : Event[*];
}
class ConfTrans extends PhaseTransition
{
        property outputs : Event[*];
        property inputs : Event[*];
}
class ExtTrans extends PhaseTransition
{
        property inputs : Event[*];
}
class Event
{
        property targetPort : Port[1];
        property value : ecore::EClassifier[1];
}
class Phase
{
        attribute phaseID : String[1];
        attribute timeAdvance : ecore::EDouble[1];
        property predicates : Variable[+];
}
```

```
        abstract class PhaseTransition
        {
                property sourcePhase : Phase[1];
                property targetPhase : Phase[1];
        }
        class Variable
        {
                attribute name : String[1];
                property domain : ecore::EClassifier[1];
        }
}
```

Appendix C PIM Metamodel in OCLinEcore

```
package sgf : sgf = 'http://devs.org/sgf'
{
	package tree : tree = 'http://devs.org/tree'
	{
		class Tree
		{
			attribute ID : String[1];
			property Root : Root[1] { composes };
			property Coordinator : Coordinator[*] { composes };
			property Simulator : Simulator[+] { composes };
			invariant UniqueTreeID: self.ID
				->forAll(n | Tree.allInstances().ID
					->count(n) = 1);
			invariant IDCanNotBeEmpty:
				self.ID.size() > 0;
		}
		class Root extends Node
		{
			property child : BasicNode[1];
		}
		class Coordinator extends BasicNode
		{
			property children : BasicNode[+];
			invariant UniqueChildren: self.children
				->forAll(n | Coordinator.allInstances().children
					->count(n) = 1);
			invariant ChildrenCantIncludeSelf: self.children
				->excludes(self);
			invariant CantOccurAsChildOfRootAndCoordinator: not
			(Coordinator.allInstances().children
				->includes(self) and Root.allInstances().child
				->includes(self));
		}
		class Simulator extends BasicNode;
		class CDEVSCoordinator extends Coordinator;
		class PDEVSCoordinator extends Coordinator;
		class FlatCoordinator extends Coordinator;
		class NodeCoordinator extends Coordinator;
		class P_Coordinator extends Coordinator;
		abstract class Simulator extends BasicNode;
		class CDEVSSimulator extends Simulator;
		class PDEVSSimulator extends Simulator;
		class P_Simulator extends Simulator;
		abstract class BasicNode extends Node
		{
			attribute modelName : String[?];
			invariant MustBeAChild: Coordinator.allInstances().children
				->includes(self) or Root.allInstances().child
				->includes(self);
		}
		abstract class Node
		{
```

```
                attribute ID : String[?];
                invariant UniqueID: self.ID
                        ->forAll(n | Node.allInstances().ID
                                ->count(n) = 1);
                invariant IDCanNotBeEmpty:
                        self.ID.size() > 0;
        }
}
package skeleton : skeleton = 'http://devs.org/skeleton'
{
        class Skeleton
        {
                attribute ID : String[1];
                property tree : sgf::tree::Tree[?];
                property rootskel : sgf::tree::Root[+] { composes };
                invariant uniqueID: self.ID
                        ->forAll(n | Skeleton.allInstances().ID
                                ->count(n) = 1);
                invariant IDCanNotBeEmpty:
                        self.ID.size() > 0;
        }
}
package bundle : bundle = 'http://devs.org/bundle'
{
        class Bundle
        {
                attribute ID : String[1];
                property skeleton : sgf::skeleton::Skeleton[1];
                property Process : Process[+] { composes };
                invariant UniqueBundleID: self.ID
                        ->forAll(n | Bundle.allInstances().ID
                                ->count(n) = 1);
                invariant AllRootsFromSkeletonAreNotInBundle: self.Process.nodes
                        ->includesAll(self.skeleton.rootskel);
                invariant AllRootsFromTreeAreNotInBundle: self.Process.nodes
                        ->includes(self.skeleton.tree.Root);
                invariant AllCoordinatorsAreNotInBundle: self.Process.nodes
                        ->includesAll(self.skeleton.tree.Coordinator);
                invariant AllSimulatorsAreNotInBundle: self.Process.nodes
                        ->includesAll(self.skeleton.tree.Simulator);
                invariant IDCanNotBeEmpty:
                        self.ID.size() > 0;
        }
        class Process
        {
                attribute ID : String[1];
                property nodes : sgf::tree::Node[+];
                invariant UniqueProcessID: self.ID
                        ->forAll(n | Process.allInstances().ID
                                ->count(n) = 1);
                invariant UniqueNodes: self.nodes
                        ->forAll(n | Process.allInstances().nodes
                                ->count(n) = 1);
                invariant ProcessIsNotInBundle: Bundle.allInstances().Process
                        ->includes(self);
                invariant IDCanNotBeEmpty:
```

```
                                        self.ID.size() > 0;
                }
        }
        package vm : vm = 'http://devs.org/vm'
        {
                class VM
                {
                        attribute ID : String[1];
                        attribute protocol : String[1];
                        property processors : Processor[+] { composes };
                        invariant UniqueVMID: self.ID
                                ->forAll(n | VM.allInstances().ID
                                        ->count(n) = 1);
                        invariant IDCanNotBeEmpty:
                                self.ID.size() > 0;
                }
                class Processor
                {
                        attribute ID : String[1];
                        attribute IP : String[1];
                        invariant UniqueProcessorID: self.ID
                                ->forAll(n | Processor.allInstances().ID
                                        ->count(n) = 1);
                        invariant UniqueProcessorIP: self.IP
                                ->forAll(n | Processor.allInstances().IP
                                        ->count(n) = 1);
                        invariant IDCanNotBeEmpty:
                                self.ID.size() > 0;
                }
        }
        package graph : graph = 'http://devs.org/graph'
        {
                class Graph
                {
                        attribute ID : String[1];
                        property bundle : sgf::bundle::Bundle[1];
                        property vm : sgf::vm::VM[1];
                        property mappings : Mapping[+] { composes };
                        invariant UniqueGraphID: self.ID
                                ->forAll(n | Graph.allInstances().ID
                                        ->count(n) = 1);
                        invariant AllProcessesMustBeInMapping: self.mappings.processes
                                ->includesAll(self.bundle.Process);
                        invariant AllProcessorsMustBeInMapping: self.mappings.processor
                                ->includesAll(self.vm.processors);
                        invariant IDCanNotBeEmpty:
                                self.ID.size() > 0;
                }
                class Mapping
                {
                        attribute ID : String[1];
                        property processes : sgf::bundle::Process[+];
                        property processor : sgf::vm::Processor[1];
                        invariant UniqueMappingID: self.ID
                                ->forAll(n | Mapping.allInstances().ID
                                        ->count(n) = 1);
```

```
                    invariant MappingIsNotInGraph: Graph.allInstances().mappings
                              ->includes(self);
                    invariant IDCanNotBeEmpty:
                              self.ID.size() > 0;
                    invariant UniqueProcessorsInMapping:
                              Mapping.allInstances()->forAll(p1, p2 |
                                        p1 <> p2 implies p1.processor <> p2.processor );
                    invariant UniqueProcessesInMapping:
                              self.processes ->forAll(n | Mapping.allInstances().processes
                                        ->count(n) = 1);
          }
     }
}
```

Appendix D ATL Rules for CIM to PIM transformation.

```
module model2Tree2;
create OUT: SG from IN: DEVS;

helper context DEVS!DEVS def: hasNoParent(): Boolean =
        self.refImmediateComposite().oclIsUndefined();

rule DEVS2Tree {
        from
                s: DEVS!DEVS (s.hasNoParent())
        to
                w: SG!Tree (
                        ID <- 'Simulation_Tree',
                        --ID <- s.name,
                        Root <- thisModule.DEVS2Root(),
                        Coordinator <- DEVS!CoupledDEVS.allInstances() -> collect(e |
thisModule.DEVS2Coordinator(e)),
                        Simulator <- DEVS!AtomicDEVS.allInstances() -> collect(e |
thisModule.DEVS2Simulator(e))
                )
}

unique lazy rule DEVS2Root {
        from
                s: DEVS!CoupledDEVS
        to
                rr: SG!Root (
                        ID <- 'Main_Root',
                        child <- thisModule.DEVS2Coordinator(DEVS!CoupledDEVS.allInstances() ->
        flatten() -> first())
                )
 }

unique lazy rule DEVS2Coordinator {
        from
                s: DEVS!DEVS
                (s.oclIsTypeOf(DEVS!CoupledDEVS))
	to
		w: SG!PDEVSCoordinator (
			ID <- s.name,
			modelName <-s.name,
			children <- s.subModels->select(p | p.oclIsTypeOf(DEVS!AtomicDEVS))-
>collect(e | thisModule.DEVS2Simulator(e)) ->union(s.subModels->select(p |
p.oclIsTypeOf(DEVS!CoupledDEVS))->collect(e | thisModule.DEVS2Coordinator(e)))
		)
}

unique lazy rule DEVS2Simulator {
	from
		s: DEVS!DEVS (
			s.oclIsTypeOf(DEVS!AtomicDEVS)
		)
	to
		w: SG!PDEVSSimulator (
			ID <- s.name,
			modelName <-s.name
		)
}
```

Appendix E ATL Rules for merging PIM and PDM

```
create OUT : Ecore from LEFT : Ecore, RIGHT : Ecore;

helper def : eClassifierByName : Map(String, Ecore!EClassifier) =  Ecore!EClassifier.allInstances()-
>iterate(e; acc : Map(String, Ecore!EClassifier) = Map{} |
 if (acc.get(e.name).oclIsUndefined()) then
 acc.including(e.name, e)
 else
 acc
 endif
 );

rule EPackage {
 from
 s : Ecore!EPackage in LEFT
 to
 t : Ecore!EPackage (
 name <- 'PDMs',
 nsURI <- s.nsURI,
 nsPrefix <- s.nsPrefix,
 eAnnotations <- s.eAnnotations,
 eSubpackages <- s.eSubpackages,
 eClassifiers <- Ecore!EClassifier.allInstances()->select(c | thisModule.eClassifierByName.get(c.name) = c)
 )
}

rule EClass {
 from
 s : Ecore!EClass (thisModule.eClassifierByName.get(s.name) = s)
 to
 t : Ecore!EClass (
		name <- s.name,
 instanceClassName <- s.instanceClassName,
 instanceTypeName <- s.instanceTypeName,
 "abstract" <- s."abstract",
 interface <- s.interface,
 eAnnotations <- s.eAnnotations,
 eTypeParameters <- s.eTypeParameters,
 eSuperTypes <- s.eSuperTypes,
 -- eOperations <- s.eOperations->union(Ecore!EOperation.allInstancesFrom('RIGHT')->select(e |
e.refImmediateComposite().name = s.name )),
              eOperations <- s.eOperations,
 eStructuralFeatures <- s.eStructuralFeatures->union(Ecore!EStructuralFeature.allInstancesFrom('RIGHT')-
>select(e | e.refImmediateComposite().name = s.name ))
 )}

rule EAttribute {
 from s : Ecore!EAttribute
 to t : Ecore!EAttribute (
 name <- s.name,
 ordered <- s.ordered,
 "unique" <- s."unique",
 lowerBound <- s.lowerBound,
 upperBound <- s.upperBound,
 changeable <- s.changeable,
 volatile <- s.volatile,
 transient <- s.transient,
 defaultValueLiteral <- s.defaultValueLiteral,
 unsettable <- s.unsettable,
 "derived" <- s."derived",
 iD <- s.iD,
 eAnnotations <- s.eAnnotations,
 eType <- s.eType)
}

rule EDataType {
 from s : Ecore!EDataType in LEFT (s.oclIsTypeOf(Ecore!EDataType))
 to t : Ecore!EDataType (
 name <- s.name,
 instanceClassName <- s.instanceClassName,
 instanceTypeName <- s.instanceTypeName,
 serializable <- s.serializable,
 eAnnotations <- s.eAnnotations,
 eTypeParameters <- s.eTypeParameters)
}

rule EOperation {
 from s : Ecore!EOperation
 to t : Ecore!EOperation (
 name <- s.name,
 ordered <- s.ordered,
 "unique" <- s."unique",
 lowerBound <- s.lowerBound,
 upperBound <- s.upperBound,
 eAnnotations <- s.eAnnotations,
 eType <- s.eType,
 eTypeParameters <- s.eTypeParameters,
 eParameters <- s.eParameters,
 eExceptions <- s.eExceptions)
}

rule EParameter {
 from s : Ecore!EParameter in LEFT
 to t : Ecore!EParameter (
 name <- s.name,
 ordered <- s.ordered,
 "unique" <- s."unique",
 lowerBound <- s.lowerBound,
 upperBound <- s.upperBound,
 eAnnotations <- s.eAnnotations,
 eType <- s.eType)
}

rule EReference {
 from s : Ecore!EReference
 to t : Ecore!EReference (
 name <- s.name,
 ordered <- s.ordered,
 "unique" <- s."unique",
 lowerBound <- s.lowerBound,
 upperBound <- s.upperBound,
 changeable <- s.changeable,
 volatile <- s.volatile,
 transient <- s.transient,
 defaultValueLiteral <- s.defaultValueLiteral,
 unsettable <- s.unsettable,
 "derived" <- s."derived",
 containment <- s.containment,
 resolveProxies <- s.resolveProxies,
 eAnnotations <- s.eAnnotations,
 eType <- s.eType,
 eOpposite <- s.eOpposite,
 eKeys <- s.eKeys)
}

rule EAnnotation {
 from s : Ecore!EAnnotation in LEFT
 to t : Ecore!EAnnotation (
 source <- s.source,
 eAnnotations <- s.eAnnotations,
 details <- s.details)
}

rule EStringToStringMapEntry {
 from s : Ecore!EStringToStringMapEntry in LEFT
 to t : Ecore!EStringToStringMapEntry (
 key <- s.key,
 value <- s.value)
}

rule ETypeParameter {
 from s : Ecore!ETypeParameter in LEFT
 to t : Ecore!ETypeParameter (
 name <- s.name,
 eAnnotations <- s.eAnnotations)
}
```

Appendix F LHF model specification

```
<?xml version="1.0" encoding="UTF-8"?>
<model:CoupledDEVS xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:model="http://model.org" name="LHF">
 <subModels xsi:type="model:CoupledDEVS" name="A" container="/">
 <iports portId="DEVS_A_IN" owner="//@subModels.0"/>
 <oports portId="DEVS_A_OUT" owner="//@subModels.0"/>
 <subModels xsi:type="model:AtomicDEVS" name="A_Demography" container="//@subModels.0">
 <iports portId="AD_gridIn" owner="//@subModels.0/@subModels.0"/>
 <oports portId="AD_gridOut" owner="//@subModels.0/@subModels.0"/>
 </subModels>
 <eics influencer="//@subModels.0/@iports.0" influencee="//@subModels.0/@subModels.0/@iports.0"/>
 <eocs influencer="//@subModels.0/@subModels.0/@oports.0" influencee="//@subModels.0/@oports.0"/>
 </subModels>
 <subModels xsi:type="model:CoupledDEVS" name="B" container="/">
 <iports portId="DEVS_B_IN" owner="//@subModels.1"/>
 <oports portId="DEVS_B_OUT" owner="//@subModels.1"/>
 <subModels xsi:type="model:AtomicDEVS" name="B_Demography" container="//@subModels.1">
 <iports portId="BD_gridIn" owner="//@subModels.1/@subModels.0"/>
 <oports portId="BD_gridOut" owner="//@subModels.1/@subModels.0"/>
 </subModels>
 <eics influencer="//@subModels.1/@iports.0" influencee="//@subModels.1/@subModels.0/@iports.0"/>
 <eocs influencer="//@subModels.1/@subModels.0/@oports.0" influencee="//@subModels.1/@oports.0"/>
 </subModels>
 <ics influencer="//@subModels.0/@oports.0" influencee="//@subModels.1/@iports.0"/>
 <ics influencer="//@subModels.1/@oports.0" influencee="//@subModels.0/@iports.0"/>
</model:CoupledDEVS>
```