DISTRIBUTED DISCRETE EVENT SIMULATION ARCHITECTURE WITH CONNECTORS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

İSMET ÖZGÜR ÇOLPANKAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

AUGUST 2014

Approval of the thesis:

DISTRIBUTED DISCRETE EVENT SIMULATION ARCHITECTURE WITH CONNECTORS

submitted by **İSMET ÖZGÜR ÇOLPANKAN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. Halit Oğuztüzün
Supervisor, **Computer Engineering Department, METU** _____

Dr. Ahmet Kara
Co-supervisor, **TÜBİTAK BİLGEM İLTAREN** _____

Examining Committee Members:

Prof. Dr. Ahmet Coşar
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Department, METU _____

Assist. Prof. Dr. Selim Temizer
Computer Engineering Department, METU _____

Dr. Atilla Özgit
Computer Engineering Department, METU _____

Dr. Cumhur Doruk Bozağaç
TÜBİTAK BİLGEM İLTAREN _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: İSMET ÖZGÜR ÇOLPANKAN

Signature :

ABSTRACT

DISTRIBUTED DISCRETE EVENT SIMULATION ARCHITECTURE WITH CONNECTORS

Çolpankan, İsmet Özgür
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Halit Oğuztüzün
Co-Supervisor : Dr. Ahmet Kara

August 2014, 69 pages

In this thesis we propose a distributed approach to Simulation Modeling Architecture (SiMA) with software connectors via Windows Communication Foundation (WCF) as a middleware technology. SiMA is a DEVS-based modeling and simulation framework developed in TÜBİTAK BİLGEM İLTAREN. Discrete Event System Specification (DEVS) is a formalism that arranges complex system models with a well-defined execution protocol. A connector is a first class entity which performs interaction among components and plays an important role in a component-based architecture. Connectors in Distributed SiMA are behavioral models that perform data conversions between models which have communication data type mismatches and data marshalling/unmarshalling for remote model communication. We claim that using a connector instead of modifying an already developed model increases the model reusability and keeps model developer from spending lots of time. We enable SiMA to run in a distributed environment via WCF which is Microsoft's distributed systems technology. It offers Service Oriented Architecture (SOA) development environment and lots of configurable features in a single .NET API. At the end we also compare this approach with the existing distributed DEVS approaches in terms of base formalism, network layer technology, model partitioning, remote node synchronization scheme and message exchange pattern.

Keywords: DEVS; SiMA; distributed DEVS; connectors; modeling and simulation

ÖZ

BAĞLAYICILARI KULLANAN DAĞITIK KESİKLİ OLAY BENZETİM MİMARİSİ

Çolpankan, İsmet Özgür
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Halit Oğuztüzün
Ortak Tez Yöneticisi : Dr. Ahmet Kara

Ağustos 2014 , 69 sayfa

Bu tezde Simülasyon Modelleme Altyapısı'na (SiMA) ara katman yazılımı olarak WCF kullanarak yazılım bağlayıcıları ile dağıtık bir yaklaşım sunuyoruz. SiMA TÜBİTAK BİLGEM İLTAREN'de geliştirilmiş DEVS tabanlı bir modelleme ve benzetim çatısıdır. DEVS iyi tanımlanmış bir çalıştırma protokolü ile karmaşık sistem modelleri düzenleyen bir biçimciliktir. Bağlayıcı, bileşenler arası etkileşim sağlayan ve bileşen tabanlı mimarilerde önemli bir rol oynayan birincil sınıf bir öğedir. Dağıtık SiMA'daki bağlayıcılar modeller arası iletişimdeki veri tipi uyumsuzluklarına veri dönüştürme yapan ve uzak model iletişimi için veriyi seri bitlere çevirme/sekizli paralel akışa geri çevirme yapan davranışsal modellerdir. Önceden geliştirilmiş bir modeli değiştirmek yerine bağlayıcı kullanmanın, modelin yeniden kullanılabilirliğini arttırdığını ve model geliştiriciyi çok zaman harcamaktan kurtardığını iddia ediyoruz. SiMA'nın Microsoft'un dağıtık sistem teknolojisi WCF kullanılarak dağıtık bir çevrede çalışmasını sağlıyoruz. Dağıtık SiMA SOA geliştirme ortamı ve tek bir .NET uygulama programlama arayüzünde bir çok ayarlanabilen özellik sunar. Son olarak bu yaklaşımı var olan dağıtık DEVS yaklaşımlarıyla temel biçimcilik, ağ katman teknolojisi, model bölüntüleme, uzak birim eşzamanlama planı ve mesaj takas şekli açılarından karşılaştırıyoruz.

Anahtar Kelimeler: Kesikli olay benzetimi; SiMA; dağıtık DEVS; bağlayıcılar; modelleme ve benzetim

Dedicated to my family

ACKNOWLEDGMENTS

I would like to express my special appreciation and thanks to my supervisor Assoc. Prof. Dr. Halit Oğuztüzün for all the support and priceless advice on my research he gave me. His motivation and immense knowledge helped me a lot in all the time of research and writing of this thesis.

I would like to express my gratitude to my co-supervisor and colleague Dr. Ahmet Kara for the insightful comments, useful remarks and engagement through the learning process of this thesis. His experience, patience and efforts have made completion of this thesis possible.

I would like to thank my thesis committee members Prof. Ahmet Coşar, Assist. Prof. Selim Temizer, Dr. Atilla Özgit, and Dr. Cumhur Doruk Bozağaç, who is also my colleague, for brilliant comments and suggestions.

I would also like to thank TUBİTAK BİLGEM İLTAREN for supporting my graduate studies.

I give my special thanks to my classmate, roommate, sports coach, flatmate and colleague Ahmet Can Bulut for tolerating my madness through years of our undergraduate and Master of Science studies. I am also grateful to Hüseyin Dirican for offering me different aspects about almost everything. And I am very grateful to my beloved Kübranur for her patience and support during my studies.

I would like to express my thanks to my parents and my brother. Words cannot express how grateful I am for being in such a great family. Their prayers for me were what sustained me thus far. Their unconditional love and support make me who I am.

Finally I am very thankful to Ferdi Tayfur for motivating me with his songs and contributing to all my successes since high school.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CLI	Common Language Infrastructure
CORBA	Common Object Request Broker Architecture
DEVS	Discrete Event System Specification
DEVSML	DEVS Modeling Language
DLL	Dynamic Link Library
DNS	Domain Name Service
DS	Deployment Service
GIG	Global Information Grid
GIIS	Grid Index Information Service
GUI	Graphical User Interface
HLA	High Level Architecture
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
KODO	Code Generator
M&S	Modeling and Simulation
OIDS	Object ID Service
PS	Port Service
PyRO	Python Remote Objects
REST	Representational State Transfer
RMI	Remote Message Invocation
RPC	Remote Procedure Call
RSS	Remote Simulation Service
RT	Real Time
SiMA	Simulation Modeling Architecture
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer

TCP	Transmission Control Protocol
UDDI	Universal Description Discovery and Integration
UDP	User Datagram Protocol
WCF	Windows Communication Foundation
WSDL	Web Services Description Language
WSN	Wireless Sensor Network
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

As Maria stated in [32] "A model is a representation of construction and working of some system of interest". Simulation is the manipulation of a model in a compressed time and space to understand the interactions of the parts of the system. Modeling and simulation is the perception of the behavior of a system of interest. It is cheaper and safer than conducting experiments such as collisions. Moreover, simulations can be carried out faster than real time. Free configuration of models and environment also helps to collect a wide range of information about a system.

Discrete Event System Specification (DEVS); which was introduced by Zeigler in 1976 [50], arranges complex system models with a well-defined execution protocol in a formalism. DEVS defines a formal basis on structural and behavioral specification of hierarchical complex model orchestration, and a simulation execution protocol. Furthermore, it consolidates the formalism by offering a hybrid system [38, 29], continuous [30] and discrete event.

There are several implementations of DEVS formalism, such as: PowerDEVS [7], DEVSimPy [10], DEVSJAVA [54], DEVSim++ [28]. In course of time, extended versions of DEVS formalism have been implemented, such as: Parallel DEVS [52, 16], Cell DEVS [46], Real-Time DEVS [21], dynDEVS [45], dynPDEVS [19], SiMA-DEVS [23, 24], dynamic SiMA-DEVS [17]. Among these formalisms Parallel DEVS is used as a base for SiMA-DEVS formalism.

Complex model hierarchy, high level of detail in models and large simulations cause the processor and memory of a computer to become insufficient to run simulations in a

reasonable time. Therefore, the need for a scalable performance leads to development of parallel and distributed simulation systems. Distributed DEVS idea was launched in 1985 by Zeigler [51] and until today lots of distributed DEVS applications have been developed, including: DEVS/CLUSTER [26], SOAD [40], DEVS/RMI [55], DEVS/MPI [12], DEVS/P2P [14], DEVS/REST [1] and DEVS/SOA [35]. Some of them are shown in Figure 1.1 under the base formalism that they work.

In this thesis, we propose a distributed approach to enable Simulation Modeling Architecture (SiMA) to execute in a distributed environment. Our approach is using Windows Communication Foundation (WCF) [13] as an underlying middleware technology and integrating the concept of software connectors for adaptation of distributed nodes and models. SiMA is a DEVS-based modeling and simulation framework developed in TÜBİTAK BİLGEM İLTAREN. It implements the SiMA-DEVS formalism which is an extended version of Parallel DEVS formalism as it is shown in Figure 1.1.

WCF offers a set of APIs in the .NET framework [34] for establishing service-oriented applications [13]. Core simulation engine of SiMA was developed in .NET framework, hence WCF might be attuned to SiMA easily. Furthermore, in a WCF-to-WCF application the fastest message encoding formatting and transfer protocol methods can be utilized compared to the other facilities that WCF offers.

The increase in modeling complexity leads to utilization of already developed reusable models. Furthermore, model reuse saves developers development effort and time, and more importantly regression tests to verify and validate the modified model. However, it is not always feasible to use a legacy model in a new simulation scenario in terms of detail of computation and data types for communication among models. Connectors engage at this point by providing interactions among components by transferring control or data, and playing important role in component-based architectures. Besides, DEVS formalism is highly appropriate for a component-based framework design when each model is considered as a component. Therefore, in our methodology we have introduced use of connectors as introduced by Kara [25] for Distributed DEVS environments to perform data conversions and data marshalling/unmarshalling.

The rest of the thesis is organized as follows: Chapter 2 describes subjects in the

background of our research, Chapter 3 provides distributed DEVS approaches related to our research, Chapter 4 explains our approach in detail, Chapter 5 presents a case study using our implementation and our discussions about the importance of our approach, and finally Chapter 6 includes our resultant comments and future work ideas.

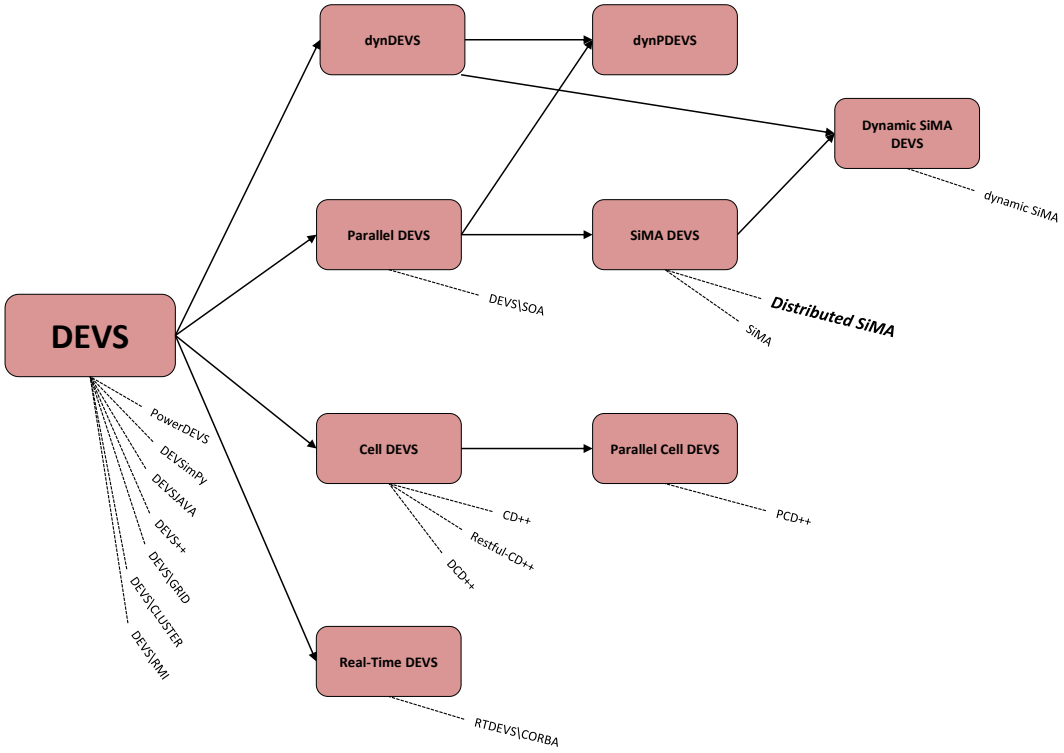


Figure 1.1: DEVS Extensions and Frameworks

CHAPTER 2

BACKGROUND

2.1 DEVS

The DEVS formalism was first introduced in 1976 by Bernard Zeigler[50] to propose a discrete event model simulated by a simulator engine. There are two kinds of models in the formalism; *atomic* and *coupled*. Atomic models are the indecomposable simulation entities and they have the behavioral logic of the component. In DEVS formalism an atomic model is defined as:

$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$

where

X is the set of external events

S is the set of states

Y is the set of output events

$\delta_{int} : S \to S$ is the internal transition function

$\delta_{ext} : Q \times X \to S$ is the external transition function, where

$Q = \{(s, e) \,|s \in S, 0 \leq e \leq ta(s)\}$ is the total state set, and

e is the elapsed time since last state transition

$\lambda : S \to Y$ is the output function

$ta : S \to R^{+}_{0,\infty}$is the time advance function

External events X are the received events through input ports and the output events Y are the events sent through output ports. These ports form the communication mechanism of models. The states S are determined by the time advance function ta and

executed sequentially. Internal transition function δ_{int} is applied for the calculation of the new state of the model. Before internal transition function, output function λ is executed to send the output events. External transition function δ_{ext} is executed when an event is received in elapsed time after last state transition e, and modifies the current model state.

Coupled models can include both atomic and other coupled models, and they just hold the coupling information of contained models. They make the DEVS structure hierarchic. They do not have the behavioral logic. In original DEVS formalism a coupled model is defined as:

$$CM = \langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

where

X is the set of input events

Y is the set of output events

D is the set of DEVS component names

for each i in D

$\{M_i\}$ is a DEVS component model, can be either an atomic DEVS model or a coupled DEVS model

$\{I_i\}$ is the set of influencees of i, the components influenced by $i \in D \cup \{self\}$

for each j in I_i

$\{Z_{i,j}\}$ is the i to j output translation function

$select$ is the tie-breaker function

X consists of the input events of the coupled model and Y consists of the output events of the coupled model. D keeps the name of the component models. The set of all influencees, I, describes the coupling network structure. Output event of a component model is associated with a corresponding target component model by using the influences and output to input translation functions. When a simultaneous state transition occurs, $select$ function is invoked and gives priority to a transition function for tie-breaking. This selection generally depends on the implementer decision.

The DEVS formalism was enhanced to handle the transition function collision while keeping the DEVS closure under coupling feature and hierarchical consistency. Thus Parallel DEVS formalism was developed [16, 52]. In Parallel DEVS formalism the atomic model structure is as:

$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$

where

X is the set of external events

S is the set of states

Y is the set of output events

$\delta_{int} : S \to S$ is the internal transition function

$\delta_{ext} : Q \times X^b \to S$ is the external transition function, where

X^b is a set of bags over elements in X

$\delta_{ext}(s, e, \phi) = (s, e)$

$Q = \{(s, e) \,|s \in S, 0 \leq e \leq ta(s)\}$ is the total state set, and

e is the elapsed time since last state transition

$\delta_{con} : S \times X^b \to S$ is the confluent transition function

$\lambda : S \to Y^b$ is the output function

$ta : S \to R^+_{0,\infty}$is the time advance function

Different from DEVS, Parallel DEVS allows receiving more than one external events in the same simulation step. When a component receives external events at the same time of internal transition, confluent transition function δ_{con} is applied. This transition does external and internal transitions sequentially. The sequence can be decided by the developer.

Parallel DEVS also removes the $select$ function from the DEVS formalism. It is not required since confluent transition handles the simultaneous events issue. The modified coupled model structure of Parallel DEVS is as follows:

$CM = \langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle$

where

X is the set of input events

Y is the set of output events

D is the set of DEVS component names

for each i in D

$\{M_i\}$ is a DEVS component model

$\{I_i\}$ is the set of influencees of i, the components influenced by $i \in D \cup \{self\}$

for each j in I_i

$\{Z_{i,j}\}$ is the i to j output translation function

2.2 SiMA

SiMA [24] is a modeling and simulation framework that is built on DEVS formalism. For complex model construction it uses a strong formalism which extends the Parallel DEVS formalism. SiMA has two extensions to the parallel DEVS formalism; strongly-typed inter model connection environment and direct feed through transition function. In port definitions of models there are some constraints and this makes port types type-safe. Moreover, the new transition function provides that in the same simulation time a model can receive data, make computation on it, and send the modified data without any state change.

2.2.1 Formalism

In SiMA formalism an atomic model structure is [24]:

$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \delta_{df}, \lambda, ta \rangle$

where

X is the set of external events arriving from set of input ports, P_{in}

Y is the set of output events sent from set of output ports, P_{out} where

P_{in} and P_{out} are the set of input and output ports such that

$P_{in} = \{(\tau, I_x) \,|\Gamma \rightarrow \tau \wedge I_x \subseteq X \wedge \forall x \in I_x, \tau \rightarrow x\}$, and

$P_{out} = \{(\rho, O_y) \,|\Gamma \rightarrow \rho \wedge O_y \subseteq Y \wedge \forall y \in O_y, \rho \rightarrow y\}$, where

Γ is the XML Schema type system

τ, ρ are data types defined via XML Schema type system

δ_{df} is the direct feed through transition function, where

$$PDFT_{in} \in P_{in} \times S \rightarrow P'_{out} \subseteq P_{out}$$

Port definitions in SiMA are made by using an XML Schema type system. Each input port definition pair is unique in P_{in}. First element of a pair τ is a data type from the XML Schema type system Γ. Second element of the pair is I_x input data value which pairs off the data type τ. Similar semantic rules applly to output ports.

This strong typing in port definitions has some benefits [24]:

- In externally visible model interfaces there is a definition of a type semantic for the modeling level information convenience and the run-time level data robustness.
- The type system dependency helps the model construction and port matching to be automated. Moreover, in SiMA architecture there are some tools for these operations such as a code generator, a model linker and a model builder.
- In model implementation there are input/output ports and state managers. State managers provide a run-time data type conformation control. They are bound to the input/output ports in order to manage the data flow in a type-safe manner. Thus, this makes the data transferring mechanism among models type-safe. State managers also offer access to received events and methods to send events; therefore, it simplifies the model development.

The new transition function δ_{df} has its own port type, direct feed through port, in order to process incoming events and send them in the same simulation interval. To avoid deadlocks, an application independent loop-breaking logic is defined in this port type. Processing incoming events and sending them in the same simulation time property can be reached by adjusting the next time of a model to the current time of the simulation and after required calculations setting it back to model's normal interval time. However, with the definition of a new transition function and its specific port type some advantages are gained:

- Without any touch to the simulation engine or making custom implementations, models gain a zero-lookahead behavior as a first class entity.

- While sharing states and communicating in the same simulation time interval there may be loops and deadlocks. Thus, a specific port type implementation makes it possible to obviate deadlocks and this makes model validity reliable.

2.2.2 Architecture

SiMA is a modeling and simulation framework through which one can develop models and execute simulations. There are two main components in SiMA; SiMA Core and C++ Interface. While SiMA Core is implemented in .NET, C++ Interface is implemented in both C++ and C++/CLI, which is a specialized C++ language in .NET that is an adapter between C++ and .NET with respect to methods and data types [20].

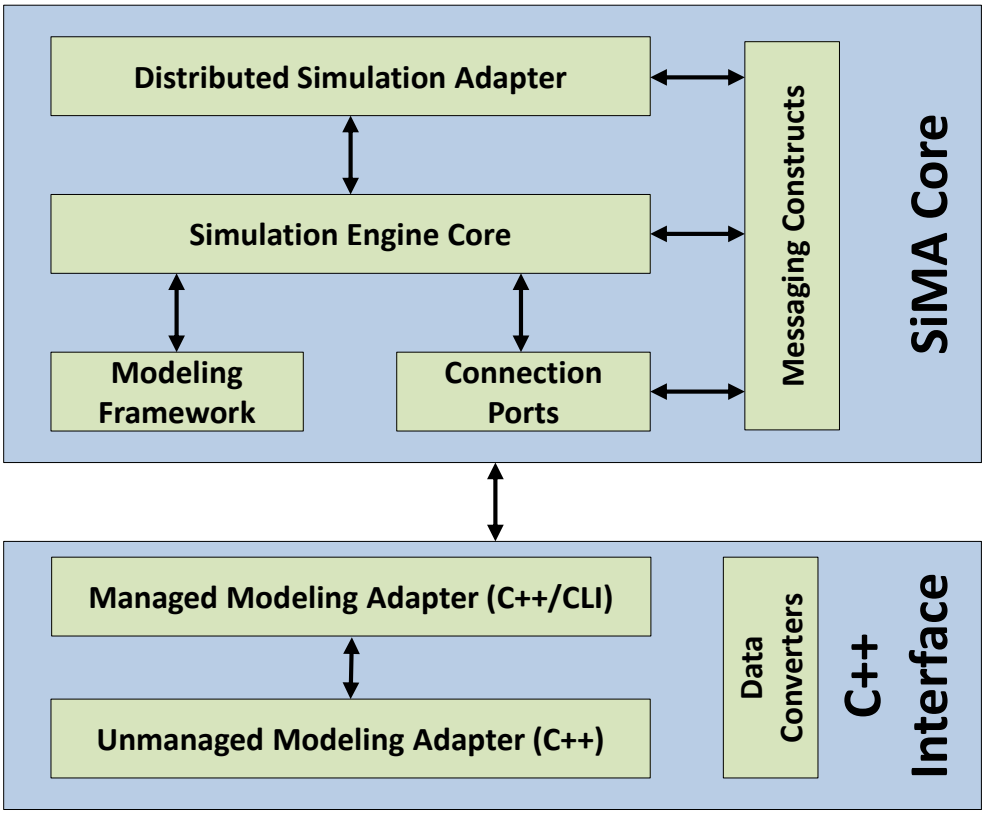


Figure 2.1: SiMA Architecture (reproduced from [24])

SiMA Core has five sub components as a framework. In *Simulation Engine Core* DEVS simulation protocol is executed. It also provides the features of monitoring and tracking of the simulations at run-time by introducing interfaces. It uses Modeling Framework and Connection Ports for the simulation execution. *Modeling Framework* covers atomic model base classes and data types used in model development. *Connection Ports* component includes port definition classes and connections. *Distributed Simulation Adapter* is an interface exposed for HLA adapters. *Messaging Constructs* has all the base data type classes and rules for model and SiMA core component communications.

SiMA supports model development in both .NET and C++ environments. C++ Interface layer presents a model development environment in C++. It consists of three sub components. Unmanaged Modeling Adapter (C++) exposes same interfaces of Modeling Framework but in pure C++. Managed Modeling Adapter (C++/CLI) allows accessing to Unmanaged Modeling Adapter methods and data types. SiMA core simulation engine runs in .NET, so this component helps the models developed in C++ to be employed in simulation execution. Data Converters are the data serialization adapters between .NET and C++ data types. KODO tool is used for the auto-generation of data converter classes by model developers. SiMA has various tool supports for the automatic simulation model construction. Figure 2.2 shows the simulation construction pipeline with five tools in four steps.

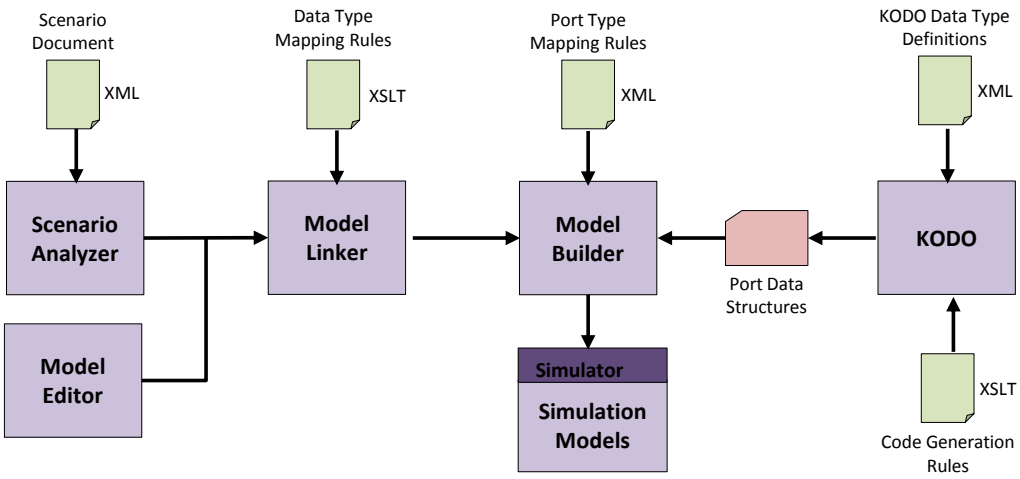


Figure 2.2: SiMA Simulation Construction Pipeline (reproduced from [24])

KODO plays a very important role in defining model initialization and port definitions for the model developer. It generates the serialization/deserialization codes for the .NET-C++ interoperability, all port and initial data class generations. These classes include the methods for the input configuration file accessing and state initializations. It also generates the logic for serializing and type conversion. Moreover, it provides methods for the trace information. KODO uses XML for the definitions and generates all required classes.

Scenario Analyzer is one of the starting points in the simulation construction pipeline. It takes a Scenario Document file as an input and sends the intermediary data to the Model Linker. The Scenario Document consists of model definition, model coupling information and initialization data of the models. After it analyzes the input file, all atomic models defined in scenario are identified and composition of hierarchical model is done.

Model Editor is the other way of starting the pipeline. In model editor, model coupling information and port connections can be defined visually. After model development in .NET or C++, all of the models are assembled in DLL files. Model editor exposes a graphical user interface for the complex model definitions using compiled model DLL files.

Model Linker takes the intermediary scenario definition file and separates the hierarchical model definition and the initialization data of the atomic models. The first output file consists of data type mapping information, port connection definitions, names and locations of atomic models by keeping the hierarchical structure. The second file is the SiMA configuration file including initialization data of the atomic models.

Model Builder reads the model definitions and builds the model structure by creating object instances of the models in memory. The root coupled model is returned as a result of building process. SiMA simulator uses it for the simulation execution.

2.3 Connectors

"A software system's architecture is the set of principal design decisions made about the system" as pointed out by Taylor [44]. These design decisions are connected to the organization of architectural elements, functional and nonfunctional properties, interaction between system elements, and implementation. Components are the software units providing services and requiring services. A comprehensive definition of a component is given by Taylor [44]; "A software component is an architectural entity that encapsulates a subset of the system's functionality and/or data, restricts access to that subset via an explicitly defined interface, and has explicitly defined dependencies on its required execution context".

Connectors are the architectural building blocks that manage the interactions among components [3]. Some examples of interactions are procedure calls, method invocations, data flow, communication protocol, and pipelines. Connectors have several functionalities as component interactions; communication, coordination, conversion and facilitation. Communication services manage the data transfers among components. Coordination services transfer the control among components. Shared memory access in a multi-threaded program, method invocations and function calls are examples of coordination services. Conversion services allow component interaction in heterogeneous environments. They modify the data required by the receiver component. They solve the mismatch issues in type, number, frequency and order of the interactions. Facilitation services provide mechanisms such as load balancing, scheduling, and control in concurrent systems to reduce inter dependencies among components.

According to Bures [9] connectors are classified into four groups; procedure call, messaging, streaming and blackboard. On the other hand, Mehta and his colleagues [33, 44] proposes eight connector types; procedure call, event, data access, linkage, stream, arbitrator, adaptor, and distributor. In our study, we take the classification of Mehta into account.

Procedure Call Connectors: These connectors undertake the communication, coordination and sometimes facilitation roles. Message invocations and procedure calls

provide the flow of control among components. The procedure calls use parameters and return values, thus data transfer occurs. Object-oriented methods, operating system calls, remote procedure calls (RPCs), and callback invocations are examples of procedure call connectors.

Event Connectors: Flow of control among components by triggering events is managed by the event connectors. In addition, data can be transferred by the event notifications as parameters. They play the communication and coordination roles. They are more virtual than procedure call connectors since they can be activated or not according to component behavior. They are mostly used in GUI (graphical user interface) applications.

Data Access Connectors: They manage the accessing data of the data store component. Saving or loading data may require a conversion in the format. Thus, communication and conversion services are operated. Database queries and repository accesses are examples of persistent data access connectors. Additionally, heap and stack memory usage in programs, and caching mechanisms are the temporary data access connector examples.

Linkage Connectors: They only provide facilitation services for the higher order connectors . They can form binding between components before or during compilation, or system execution. They can take role in configuration management, system building, linking variables, procedures, functions, constants and types within the linked components, and interaction protocols.

Stream Connectors: Large amount of data transfers and data transfer protocols in client-server architectures are achieved by the stream connectors. They apply communication services in their system. Pipes, TCP/UDP communication sockets, and client-server middleware are some examples of stream connectors.

Arbitrator Connectors: They employ the facilitation and coordination services. They channel the flow of control to resolve the conflicts between components. Shared memory access synchronization in multi-threaded systems, scheduling and load balancing services are provided by arbitrator connectors. They ensure system reliability, safety and security.

Adaptor Connectors: They bring interoperability characteristic to the components that are not designed as interoperable. Required conversion and facilitation services are provided by them. Mostly heterogeneous environments employ them in order to establish communication policies and interaction protocols. Remote procedure call is the basic form of these connectors.

Distributor Connectors: They never exist by themselves, but they help other connectors, such as stream and procedure call. Information routing and interaction path identification between components are performed by distributor connectors. They provide the facilitation services. They are used mostly in distributed systems to identify the component locations and paths. Some example services of distributor connectors are domain name service (DNS), routing and switching like network services.

2.4 Windows Communication Foundation

Windows Communication Foundation (WCF) is Microsoft's distributed systems technology that covers the features of .NET distributed technologies in a single API[13]. It facilitates building, deploying, and operating distributed application infrastructures. It promotes the service-oriented architecture (SOA). It can provide the interaction between client-server via SOAP, a WCF-to-WCF binary communication protocol, other main WS-* specifications, peer-to-peer networking, inter-process communication protocols, and RESTful architecture [11]. The following are some of WCF features [42]:

Service Orientation: WCF proposes a service oriented development environment. SOA ensures data send and receive operations.

Interoperability: WCF supports the interoperability with applications built on other distributed technologies.

Multiple Message Patterns: There are several message exchange patterns such as one-way, duplex, request/reply patterns. In one-way messaging system one endpoint sends a message and does not wait for the reply. In duplex channels callback methods can be defined and not only does the source endpoint send a message, but also the

target endpoint can send a message to the source. In request/reply patterns source endpoint sends a message and target endpoint replies.

Service Metadata: WCF uses some standards such as WSDL, XML Schema and WS-Policy in publishing service metadata. It is published over HTTP, HTTPS, or Web Service Metadata Exchange standards.

Data Contracts: In data transfer WCF uses the data contract property provided by .NET framework. In order to be serializable by WCF, the data contract identifier is added to implemented classes. This allows a service to generate the metadata of the data type.

Security: Standards such as SSL or WS-SecureConversation can be applied to encrypt messages.

Multiple Transports and Encodings: Mostly used transport and encoding technique is XML text encoded in SOAP messages over HTTP. Moreover, WCF allows sending messages over TCP, pipes, or MS-MessagingQueue. Messages are also encoded in binary format.

Reliable and Queued Messages: There are sessions implemented over WS-Reliable Messaging using MS-Messaging Queue for the reliable message exchange.

Durable Messages: These are the never-lost messages due to network problems while transferring. Durable messages are saved to a database and after re-establishing the network connection, sending processes resume.

Extensibility: WCF offers several extensibility points. The service behaviors can be customizable.

CHAPTER 3

RELATED WORK

This chapter consists of some Distributed DEVS approaches. Before describing them, some features used by distributed DEVS implementations are explained in following subjects:

Model Deployment: Model deployment is the initial process of Distributed DEVS approaches. Before the simulation starts the DEVS model is partitioned and deployed to nodes. Partitioning can be made according to some algorithms in master node or manually. Among partitioning algorithms, mostly used one is the Cost-based hierarchical model partitioning algorithm [37]. In some implementations hierarchical DEVS model is converted to a non-hierarchical one. After partitioning, each partitioned DEVS model is distributed over the remote nodes and is initialized to be made ready for the simulation.

Time Management: Since DEVS models are being executed on remote nodes, there has to be a synchronization among them in terms of simulation time. There are two time management approaches in distributed DEVS implementations; *conservative* and *optimistic*. In conservative scheme, global time which is maintained by a central node is used between remote nodes. In every simulation step, central node gets the minimum time and advances the simulation time, so this brings overhead over network. However, it provides global time synchronization and it guarantees that during simulation no time causality violation occurs. On the other hand, in optimistic scheme each simulator manages its own simulation time without any synchronization with the other remote simulators. When a time causality violation occurs, simulation results after that time are thrown away and simulation is rolled back to the last

time the simulators were in synch. This requires time-tagged model data to be saved for rolling back. Time Warp [27] is the most well-known algorithm for optimistic approach. When a simulator receives an event or object with a prior time stamp according to already processed events or objects, rollback happens and the simulator reprocesses those events or objects in time stamp order.

Communication: Inter-model communication mechanism varies among distributed DEVS architectures:

1. In system-central architectures, firstly, each model on each node sends events to the coordinator of the corresponding node. After that, coordinator sends the event package to the central coordinator located on central node. Central coordinator dispatches the events through the network to the related coordinators and coordinators send events to models.

2. A faster version of this approach is making each node center of communication, node-central architecture. In this approach, each model sends events to its coordinator located on each node. Coordinators on each node make event transfers by direct communication with remote coordinators.

3. The last approach discards all redundant data exchanges and enables direct inter-model communication as if it is a non-distributed DEVS implementation.

3.1 DEVS/P2P

DEVS/P2P [14] is a distributed DEVS implementation that proposes a peer-to-peer (P2P) simulation protocol to operate a DEVS simulation on a distributed and parallel computing environment. The proposed protocol uses advantage of P2P infrastructure, in which inter-connected peers share resources with each other without using any centralized administrative system, to gain optimal performance compared to existing protocols. As middleware it uses JXTA [48] technology which is a P2P network system implementation. DEVS/P2P uses a customized distributed DEVS simulation protocol in which each simulator blocks itself until receiving a "DONE" message from all other simulators in each DEVS call. So decentralized conservative approach

is used in synchronization of remote simulators.

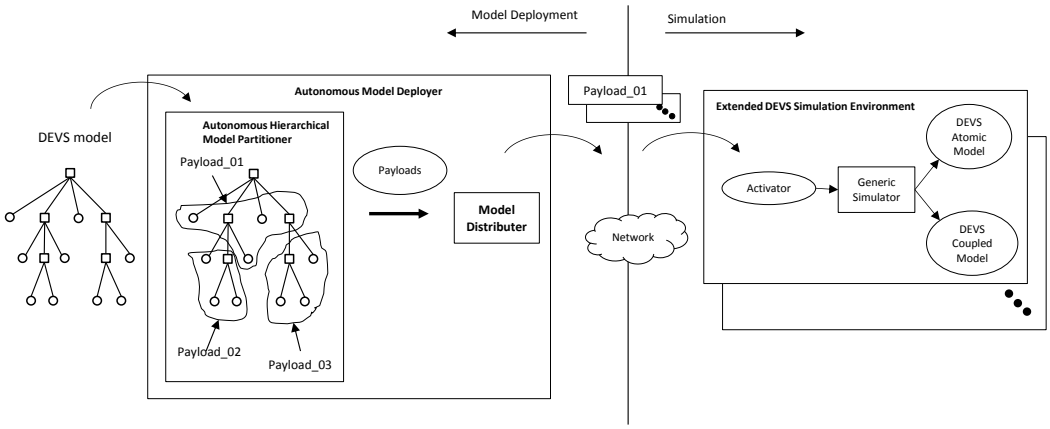


Figure 3.1: DEVS/P2P Architecture (reproduced from [14])

The overall system can be inspected in two parts. The first part is the Autonomous Model Deployer (AMD). Its task is partitioning DEVS model and dispatching the partitioned models over local and remote simulators. Firstly, Autonomous Hierarchical Model Partitioner partitions the DEVS model according to the cost-based hierarchical model partitioning algorithm. After that, Model Distributer deploys partitioned DEVS models to the local and remote simulators. The second part of the system is Extended DEVS Simulation Environment (EDSE). All simulators have EDSE including the Activator and the Generic Simulator (GS). AMD sends the model to Activator through JXTA message exchange service. Then GS is created by the Activator and calls the DEVS atomic or coupled simulator.

In inter-node communication, DEVS/P2P uses JXTA message which is a specialized XML document. DEVS message is converted into JXTA message and is transferred to the target remote model's input port. In remote node this message is reconverted to DEVS message. Source model's output port is mapped to source node's output JXTA pipe. Similarly, target model's input port is mapped to target node's input JXTA pipe. And these two pipes are connected to each other. Through the JXTA pipe, the target model can receive the output message of the source model.

3.2 DEVS/GRID

In DEVS/GRID [41], Grid computing infrastructure is used for DEVS modeling and simulation activities. DEVS/GRID proposes new functionalities to the existing DEVS M&S frameworks as mentioned by Seo [41]: "cost-based hierarchical model partitioning, dynamic coupling restructuring, automatic model deployment, remote simulator activation, self-communication setup, M&S name and directory service, etc." As middleware, Globus Toolkit which is widely used in grid computing is used. This software provides high performance computational Grid resources usage. The network infrastructure consists of interconnected Grid components which are high performance computers, storage devices, etc.

There are three non-simulation components which are *model partitioner*, *model deployer* and *activator*. *Model partitioner* uses hierarchical model partitioning and constructs a set of partition blocks with the help of a cost-based measurement system. After forming a cost tree from the DEVS model the Generic Model Partitioning Algorithm is performed in partition process. While creating partition blocks the original structure of DEVS model can be broken. Thus, in each partition block restructuring the coupled model according to new model information can occur. Then *model deployer* takes place and identifies which nodes are joining the simulation by Grid Index Information Service (GIIS). Partition block and the coupling data of it are packed in a message and are sent to the activators on those nodes. The *activators* receive the payload and set up remote simulators for simulation execution. After simulator creation, node information including IP address, name of the simulator, model name, input/output ports, etc. is published through GIIS.

There are two communication channels in DEVS/GRID; *user channel* and *system channel*. *User channel*'s objective is transferring output data of a model to an input port of a remote model. *System channel* is used for synchronization. It synchronizes and advances the simulation time and applies locks for data transfer synchronization. The communication channels are constructed after remote simulators are launched and information of each node is published. For optimization of communication channels some filtering, grouping and optimization methods are used.

There is no central coordinator for the simulation management in DEVS/GRID. Each simulator calls DEVS methods individually. After calling each method, simulator blocks itself until it receives all other simulators' protocol messages via system channels. When the termination condition is encountered, the DEVS cycle ends.

3.3 DEVS/CLUSTER

The novelty of DEVS/CLUSTER [26] is transforming hierarchical DEVS model structure into a non-hierarchical one to ease the synchronization of remote models. DEVS/CLUSTER utilizes CORBA as a communication system which is designed to perform collaboration between heterogeneous platforms, different programming languages and operating systems. CORBA supports object oriented programming paradigm and this suits the DEVS formalism. DEVS/CLUSTER is also a "multi-threaded distributed simulation" that works on distributed nodes as multi-threaded. This ensures that while processing external events no deadlock condition is encountered. DEVS/CLUSTER employs Time Warp algorithm [27] for time synchronization.

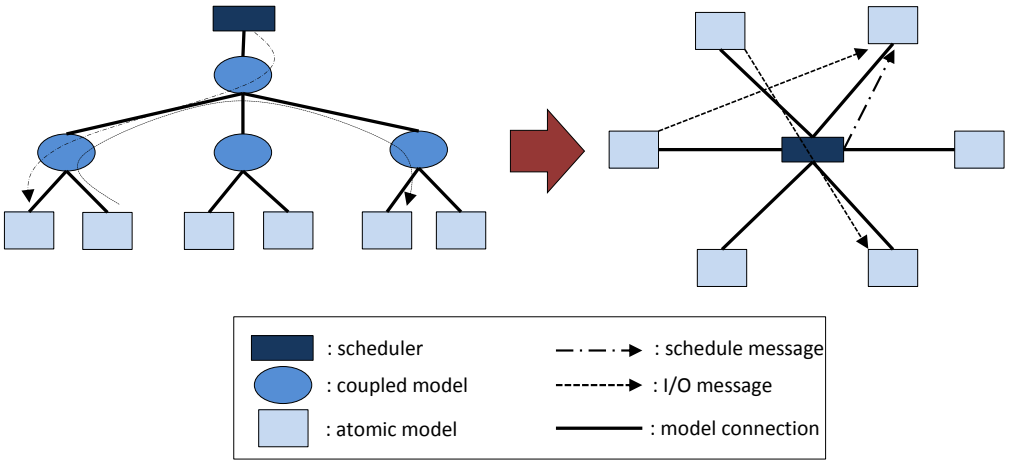


Figure 3.2: DEVS/CLUSTER Simulation Mechanism (reproduced from [26])

Unlike other distributed DEVS implementations DEVS/CLUSTER converts the hierarchical DEVS structure into a non-hierarchical one. This operation is carried out

without loss of information. Coupled models are removed and the hierarchical structure becomes flattened. Input/output ports of the atomic models lose connections because before removal process they were connected to coupled models. Hence, new connections are made directly between atomic models whether they reside in local or remote node. A central scheduler is placed for the internal event scheduling in place of coupled models.

3.4 DEVS/RMI

DEVS/RMI [55] focuses mostly the reconfiguration of the simulation structure dynamically in runtime unlike other approaches. It is a distributed version of DEVS-JAVA [4, 39]. Its underlying communication technology is Java RMI. Because of Java virtual machine implementation it is platform independent. RMI provides synchronization in making remote method calls, so DEVS/RMI does not require an extra time synchronization mechanism. RMI also handles serialization and marshalling for the objects transferring between remote nodes. This object used as an entity object in DEVSJAVA has to extend Java Serializable.

Partitioning can be performed in two ways; *static* or *dynamic*. In *static* partitioning, before the simulation execution, main model is partitioned and distributed over remote nodes. *Dynamic* partitioning is activated in runtime. When a dynamic model partition is required, simulation cycle is stopped in simulation/model migration process. After migration, simulation continues to simulation execution with new partition plan.

DEVS/RMI is composed of 4 components; *configuration engine*, *simulation controller*, *monitoring engine* and *remote simulator*. *Configuration engine* applies the partitioning algorithm to the given model structure and sends the partition plan to the simulation controller. In addition to this static partitioning, it can decide dynamic partitioning by applying repartition algorithm to the model information received from monitoring engine. *Simulation controller* manages the simulation with start, restart, continue and stop commands. It also applies static and dynamic partitioning. It commands the transportation of simulators and models to the remote nodes or creation

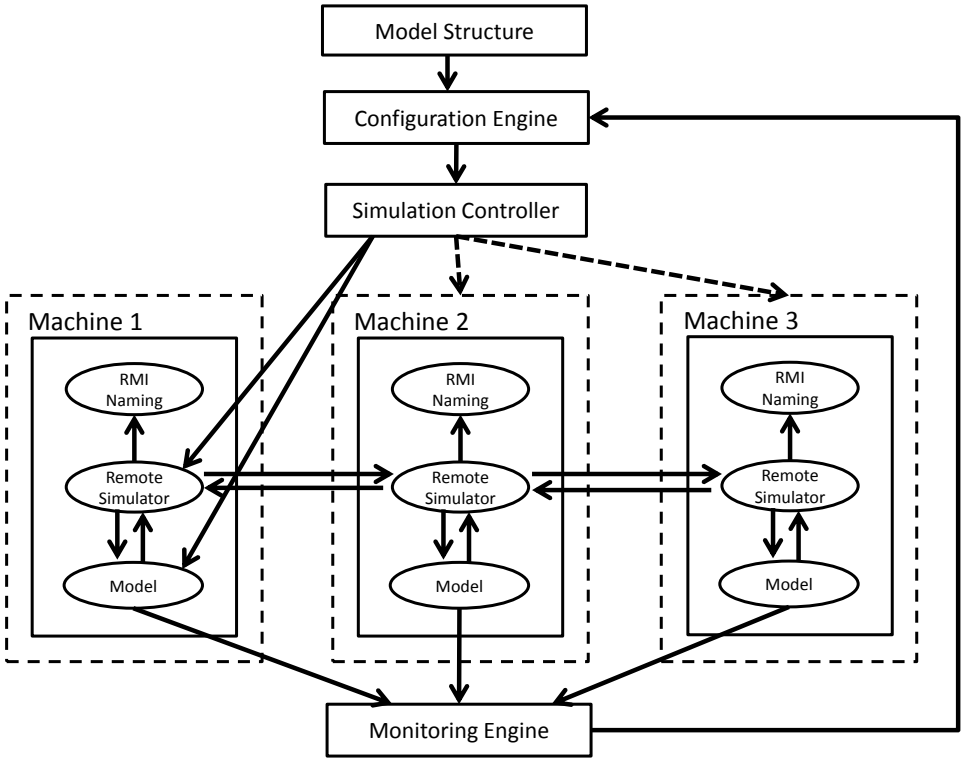


Figure 3.3: DEVS/RMI Architecture (reproduced from [55])

of remote simulators on them. *Monitoring engine* gathers information of all models in the simulation, keeps track of their actions and sends that information to the configuration engine. *Remote simulator* maintains the DEVS simulator objectives as a remote interface. It is created by the simulation controller and is connected to RMI naming server in order to publish network address information with a unique name. A related model is attached to remote simulator to process with and made ready for the simulation start.

3.5 DEVS/SOA

The main distinction of DEVS/SOA [35] from other approaches is providing a solution to cross-platform distributed M&S in a client-server architecture using SOA. It uses Java for the implementation. Messages between remote nodes are serialized with SOAP. In communication, XML is used to make the system interoperable. Simulation engine is developed as transparent to provide model level interoperability to the

developers and users. DEVS models can be developed in different languages and they are represented in DEVSML, a specialized XML language for model definitions. Remote simulators in DEVS/SOA behave as web services and composed of some core features such as SOAP, WSDL and UDDI.

There are two kinds of simulation protocols in DEVS/SOA; *centralized* and *real-time*. In *centralized* simulation, there resides a central coordinator in main server and it is connected to all simulation services over network. At run-time, all output messages are passed through this central coordinator since only it knows the coupling information of all simulation services. In *real-time* simulation, wall-clock time takes over the logical time in DEVS protocol execution. RT coordinator and RT simulation services take part in execution. Output message of an RT simulation service is transferred directly to the target RT simulation service since each of them knows the coupling information. In centralized simulation, simulation time is controlled by the central coordinator. However, in real-time simulation each simulator manages its own time.

DEVS/SOA also supports platform independent simulation execution. While one simulation service works in DEVSJAVA, the other one can work in DEVSC++ [53]. Furthermore, the similar fact is eligible for the simulators. DEVSML provides a wide environment for the multi-platform simulation definition. Between remote simulators, platform independent messages wrapped with SOA framework are used and in local communication these messages are converted to platform specific messages.

3.6 DEVS/PyRO

DEVS/PyRO [43] is designed to make quantitative analysis of reliability and performance of different simulator designs with detection of failures in computational and network resources. It is implemented in Python and uses pythonDEVS [36, 8]. PyRO which is an RMI-based python implementation is used in middleware layer. Simulation protocol is executed asynchronously for performance.

DEVS/PyRO uses the *solvers* as a simulation protocol fashion and simulation is executed by the communications between solvers. Each solver has a management interface with the corresponding model. A coupled model's solver is named *coordinator*.

Classical DEVS protocol is applied by root coordinator. All the atomic solvers, coordinator and the root coordinator are atomic DEVS models. There are also three important atomic DEVS models for fault detection, toleration and restoration; *Monitor*, *Log* and *Master Servers*. *Monitor server* watches each remote node in order to detect faults and notifies master server if required. *Log server* keeps information of atomic solvers, coordinators, and the root coordinator from the logging ports. And the *master server* manages the whole system. It knows the coupling information and executes the DEVS simulation protocol.

When a simulation message is sent, target simulator is firstly searched locally by looking into Local Coupling Table. If the target simulator is not there, Remote Coupling Table is searched. All simulation messages wait for the callback, so although simulation is executed asynchronously, it is always in sync.

3.7 Other Distributed DEVS Approaches

There are also some distributed DEVS approaches which utilize different DEVS formalisms; such as Cell DEVS [47] and Real-time DEVS [21], in the field. We do not go into details of these since we have focused on Parallel DEVS and SiMA DEVS formalisms. However, some information about approaches and architectures of them is presented as:

PCD++ /.NET [18]: It can apply both Parallel DEVS and Cell DEVS formalisms. For client-server routine, .NET Remoting API is used. As message transfer protocol both HTTP and TCP are supported. Furthermore, both SOAP and binary encoding formatters can be adopted in PCD++. Models are implemented in C++, thus, for the message exchange C++/CLI is utilized.

DCD++ [31]: It can also apply both Parallel DEVS and Cell DEVS formalisms. On the remote nodes Web Service technology with SOAP is adopted. Interoperability between JAVA and C++ models is achieved through XML as a message transfer format. Both optimistic scheme with Time Warp algorithm, and conservative scheme with synchronization messages or central simulation management are implemented.

Restful-CD++ [1]: As a middleware technology Representational State Transfer (REST) is used. Restful-CD++ follows HTTP transfer protocol. The distributed simulation is maintained by a root coordinator. Simulation clock is managed by the root coordinator. Simulation message is propagated downward in the hierarchy and a 'DONE' message is propagated upward until it reaches the root coordinator. Synchronization is provided in this way.

RTDEVS/CORBA [15]: It is the real-time extension of DEVS/CORBA. Real-time DEVS formalism is used. In the middleware, ACE/TAO extension of CORBA is utilized to be in sync with real time. Coordinator no longer keeps components synchronized as in DEVS/CORBA. In order to ensure conformity with the real time, simulators in each node work asynchronously and a globally synchronized real-time clock is used.

Table 3.1: Overview of Distributed DEVS Approaches

	Formalism	**Middleware Technology**	**Partitioning**	**Synchronization Scheme**	**Message Exchange**
DEVS/CLUSTER	DEVS	CORBA	hierarchical to non-hierarchical structure	optimistic	CORBA remote method invocations
DEVS/GRID	DEVS	Globus	cost-based hierarchical partitioning	conservative	GIIS
DEVS/P2P	DEVS	JXTA	autonomous hierarchical model partitioning	conservative	JXTA message format
DEVS/RMI	DEVS	JAVA/RMI	applying built-in partition algorithm	conservative	JAVA serializable object
DEVS/PyRO	DEVS	PyRO/RMI	user specified or automatic	conservative	serialized objects
DEVS/SOA	Parallel DEVS	GIG/SOA	user specified	conservative	JAVA serialization
PCD++/.NET	Parallel DEVS or Cell DEVS	.NET Remoting	user specified	optimistic or conservative	SOAP or binary encoding
DCD++	Parallel DEVS or Cell DEVS	Web Service	user specified	optimistic or conservative	XML format
Restful-CD++	Cell DEVS	REST	user specified	conservative	XML messages within HTTP envelopes
RTDEVS/CORBA	Real-Time DEVS	ACE/TAO extension of CORBA	user specified	conservative	CORBA remote method invocations

CHAPTER 4

DISTRIBUTED SIMA

Distributed SiMA is a framework that enables SiMA to execute in a distributed environment. It also aims to increase model reusability by bringing software connector notion to M&S. Connectors in Distributed SiMA assist inter node communication and adaptation of models in terms of port data types. Distinctively from other distributed DEVS approaches Windows Communication Foundation is adopted as an underlying middleware technology. Like SiMA core engine Distributed SiMA has been implemented in .NET $C\#$. Besides, WCF is convenient to adapt two .NET products with each other in terms of performance. In Distributed SiMA most of the WCF features mentioned in section 2.4 are benefited such as service oriented development environment for simplicity, request/reply message exchange pattern for synchronization, binary encoding over TCP transportation for performance and customized service behavior for object serialization.

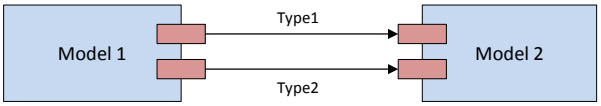
Novelty of our approach is the explicit use of connectors for adaptation of distributed nodes and models via favorable WCF features in a distributed DEVS environment.

4.1 Connectors in Distributed SiMA

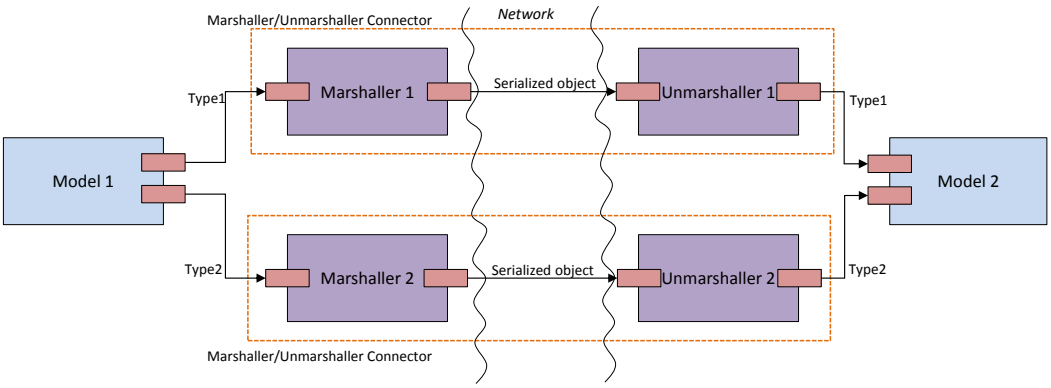
Distributed SiMA uses connectors which play important roles in simulation execution since connectors play important roles in a component-based architecture. Connectors in Distributed SiMA are the specialized atomic models. There are two kinds of connectors in terms of the tasks they carry out: marshaller/unmarshaller connector and data conversion connector.

4.1.1 Marshaller/Unmarshaller Connector

It consists of two built-in atomic models used in communication among remote models. If there is a port coupling between two models (coupled or atomic) located at different nodes, a marshaller/unmarshaller connector is placed between the models in order to serialize and deserialize the port data. The marshaller part of the connector is placed in the source node, and the source model's output port is connected to its input port. Similarly the unmarshaller part of the connector is placed in the target node, and its output port is connected to the target model's input port. A port coupling of two models in Figure 4.1a can be transformed for a distributed environment as in Figure 4.1b with marshaller/unmarshaller connectors.



(a) Located in the same node



(b) Located in remote nodes

Figure 4.1: Marshaller/Unmarshaller Connectors

Conceptually, there is a one connector that performs marshalling and unmarshalling, in addition to networking chores. In terms of implementation, it is composed of two atomic DEVS models (one for marshalling, the other for unmarshalling), and the underlying network. Marshalling/Unmarshalling connector uses direct feed through transition port defined in SiMA formalism (Section 2.2.1). Therefore there is no simulation time step loss between remote models despite the connectors between them. *Atomic 1* sends data to *Marshaller 1*. In the same simulation time step *Marshaller*

1 serializes the data and sends it through network to *Unmarshaller 1*. *Unmarshaller 1* deserializes the data and sends it to the *Atomic 2*. The sequence diagram in Figure 4.2 clarifies the operations. With the help of WCF there is no need for an extra synchronization while sending port data through network. Because even if a remote procedure call does not return a value, WCF ensures that it caller waits until calling process is done.

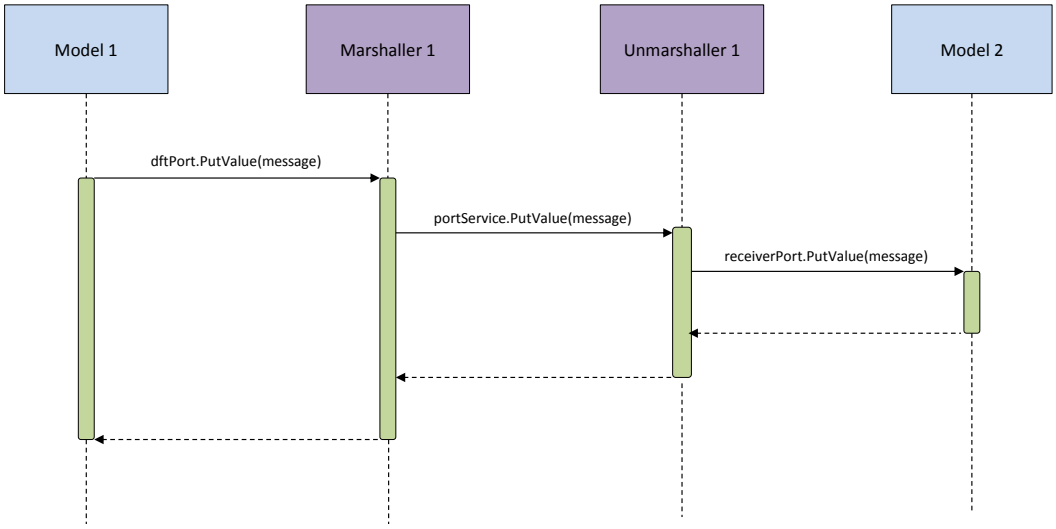


Figure 4.2: Remote Model Communication Sequence Diagram

4.1.2 Data Conversion Connectors

Model continuity is a profitable characteristic when huge simulation projects are considered. Already implemented atomic models can be used in other projects without modifications. Modelers might not want to break the integrity of already developed models since the verification and validation time of the modified model may take more time than developing a new model. However, data types processed by the legacy models might not match the data types of the new models. Evidently, composable models may reflect the same real world entities and facts; however, they might have non-identical data representations. These models have to communicate with each other but port data types they are using are not the same. Data conversion connectors [25] accomplish this bridging task. They receive a data packet in *Type1*, apply conversion to it, and send it in *Type2* like in Figure 4.3. They are implemented by the model developers as atomic models. Distributed SiMA provides a base class

for the developers to develop connector models that utilize DFT ports for conversion routines.

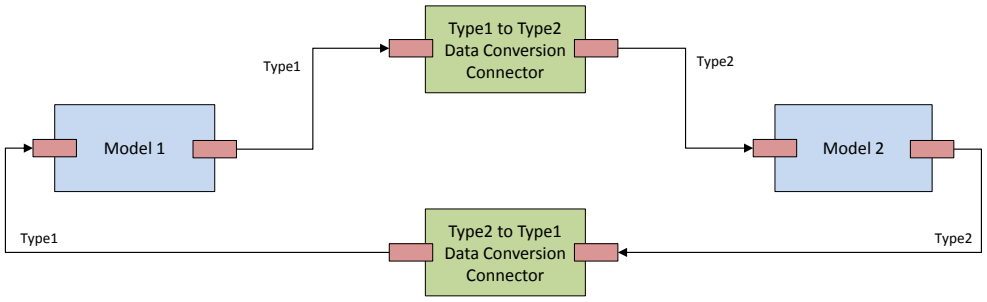


Figure 4.3: Data Conversion Connectors

4.1.3 Connector Roles According to Mehta Classification

Mehta and his colleagues stated eight connector types mentioned in Section 2.3; procedure call, event, data access, linkage, stream, arbitrator, adaptor, and distributor. A connector may have more than one type among them.

Marshaller/Unmarshaller connectors are *procedure call* connectors since they make remote procedure calls. They are also *linkage* connectors as they are not defined as a simulation entity before simulation construction, but they are inserted to simulation when a remote model port data transportation is needed. They establish the WCF communication protocol and serializes data to be sent over network, so they are *adaptor* connectors.

Data conversion connectors are mostly *adaptor* connectors, because they provide data conversion service among components. They also form binding between different typed models and this makes them *linkage* connectors.

4.2 WCF Services in Distributed SiMA

There are four services in Distributed SiMA that are provided by the SOA feature of WCF: *Deployment Service (DS)*, *Remote Simulation Service (RSS)*, *Port Service (PS)*

and *Object ID Service (OIDS)*.

Deployment Service: Purpose of Deployment Service is creating a model deployment interface that is used by the central node. All nodes joined in the distributed simulation open a Deployment Service and central node takes the role of the client. The central node can get the names of remotely located models, deploy models, test them and order nodes to start Remote Simulation Service.

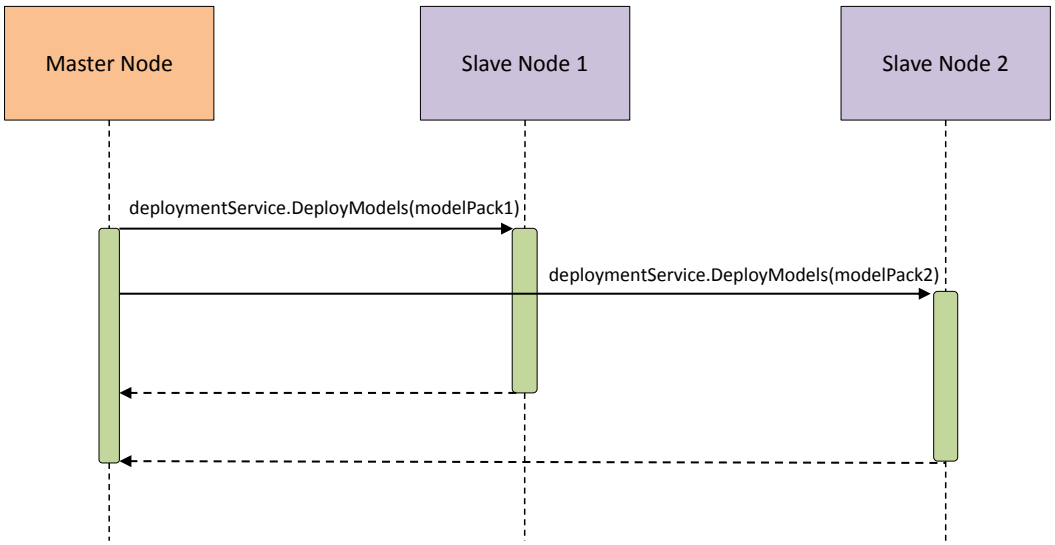


Figure 4.4: Deployment Service Usage Sequence Diagram

Remote Simulation Service: For this service, central node is a client again that connects the other nodes via Remote Simulation Service. Central node builds and maintains the simulation execution with the help of this service. It sends required information to the other nodes for distributed simulation construction and builds remote model couplings on them. After distributed simulation construction, SiMA protocol is maintained by central node via this service.

Port Service: This service is used for the data transfer between ports of remote models. For each port coupling of a data type, one Port Service is started on the receiver model. The source model connects to this service in distributed simulation construction process.

Object ID Service: It is used for keeping the integrity of SiMA port data usage principle. In classical SiMA, the objects transferring port data are created before the

first transfer and a unique object id is assigned to them by the simulator. Not to break this uniqueness, central node hosts the Object ID Service and provides the generated unique id when a node requests.

4.3 Distributed SiMA Architecture

We used the existing SiMA implementation as a basis to develop Distributed SiMA. We added new packages, extended the core classes, and modified some classes for the distributed version. As shown in Figure 2.1 SiMA already has a distributed simulation adapter for interoperability. We did not touch it, but we implemented new packages. Figure 4.5 shows the added components: Distributed SiMA, Model Deployment Server and Model Deployment Client.

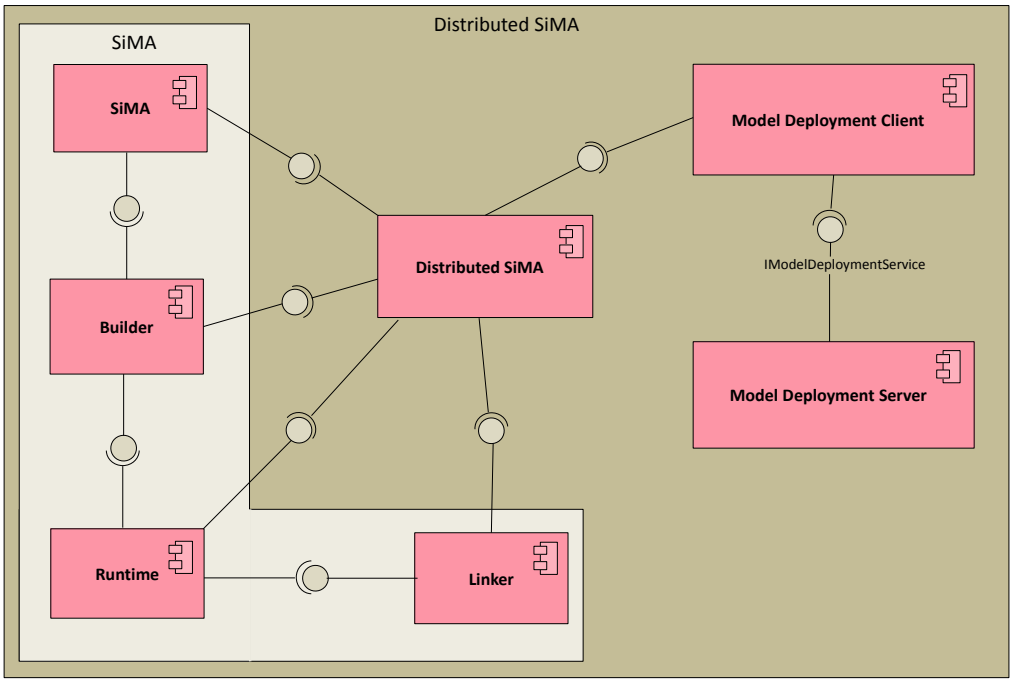


Figure 4.5: Distributed SiMA Components

The central node managing the whole system by deploying models, building simulation, and executing simulation is called the Master Node. Other remote nodes joining the simulation execution are called Slave Nodes. The only difference in package distribution between master and slave nodes is that Model Deployment Client works

in master node and Model Deployment Server works in slave nodes as illustrated in Figure 4.6.

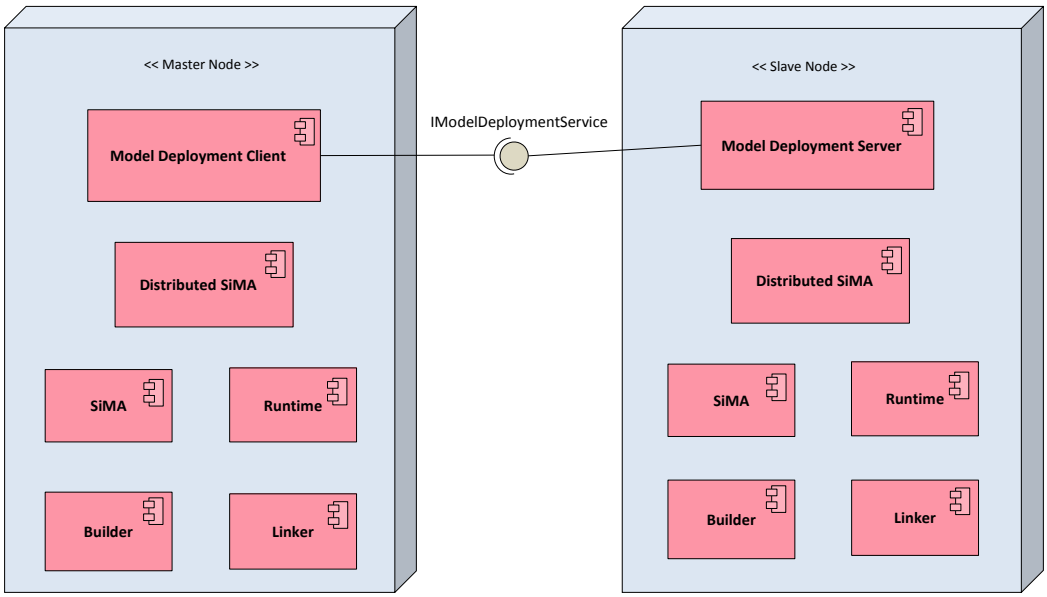


Figure 4.6: Deployment Diagram

Distributed SiMA could be examined in two main phases with respect to distributed simulation execution: *model deployment*, and *simulation build and run* as illustrated in Figure 4.7.

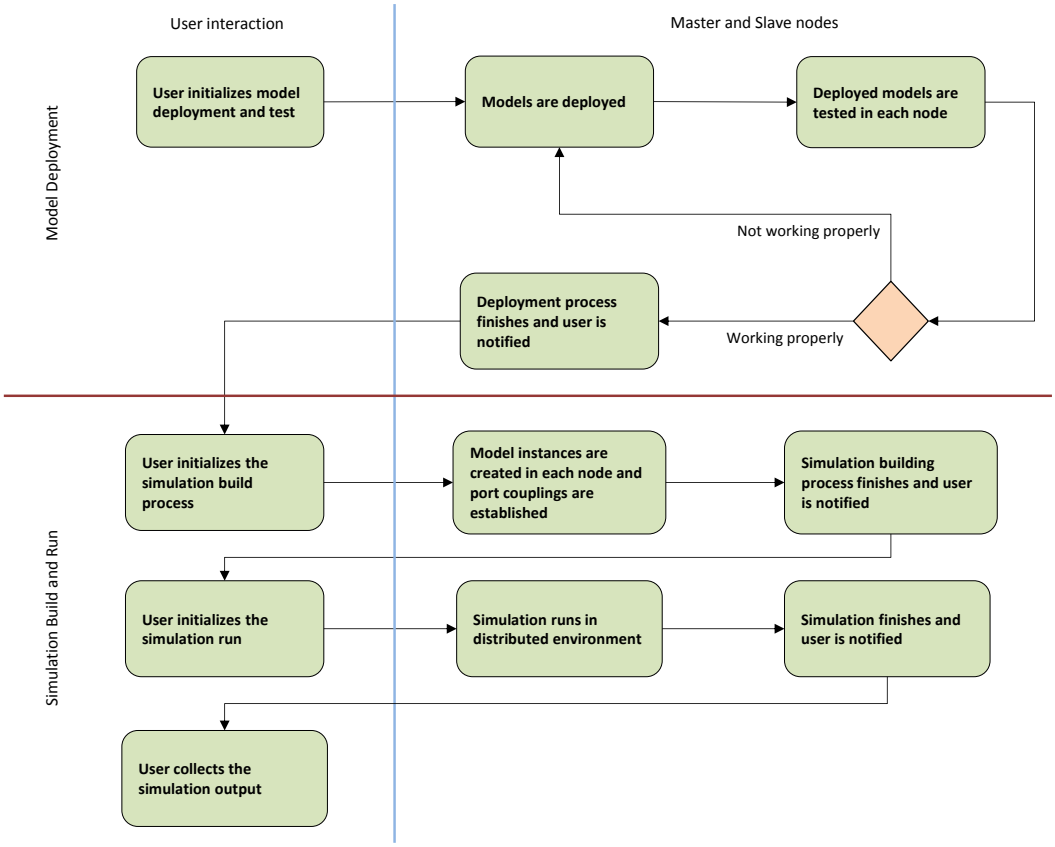


Figure 4.7: General System Flow Diagram

4.3.1 Model Deployment

This phase covers identifying slave nodes, establishing connections, deploying and testing of models in them. Before all of these, slave nodes must be ready for the model deployment. For the model deployment slave nodes need starting the Model Deployment Server and creating a service listener for the connection. We use Windows services that operate in the background, and are managed by the operating system. We developed a Windows service to instantiate the Deployment Service. In order to make a slave node ready for model deployment and distributed simulation execution, the following steps are to be performed (Figure 4.8).

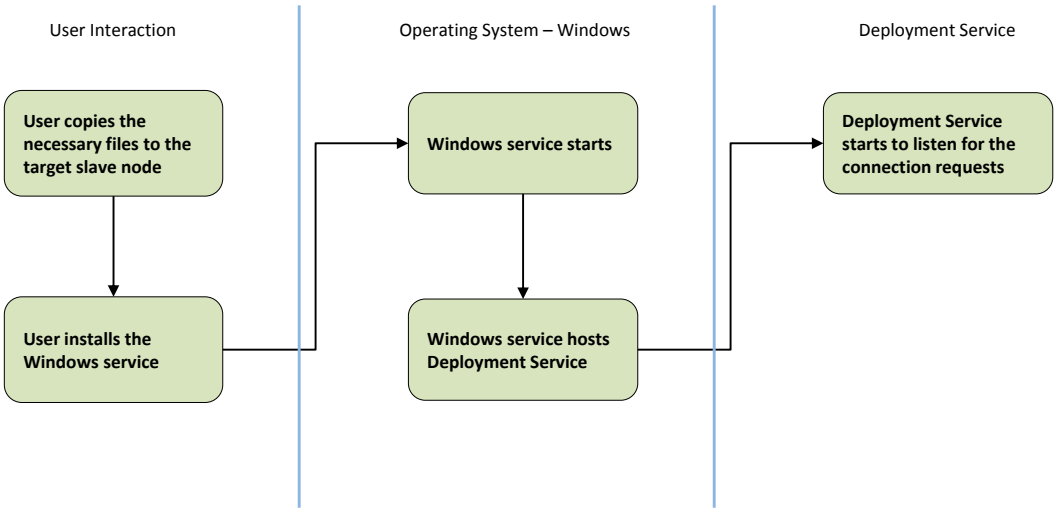


Figure 4.8: Deployment Service Start

1. User copies the necessary files to each target slave node: distributed SiMA execution DLLs and Windows service installation file.
2. User installs the Windows service.
3. Windows service starts.
4. Windows service hosts Deployment Service.
5. Deployment Service starts to listen for the connection requests until someone manually stops it or computer is shut down.

Slave nodes behave like servers after these steps and start to listen via Deployment Service port for the connection requests. This service is used in the deployment phase of Distributed SiMA. The activity diagram in Figure 4.9 indicates the parts of the deployment phase in order. User initializes the model deployment process with the distributed scenario definition document which consists of model definition, coupling information, initialization parameters of models, and endpoint addresses of slave nodes added to coupled or atomic model definitions.

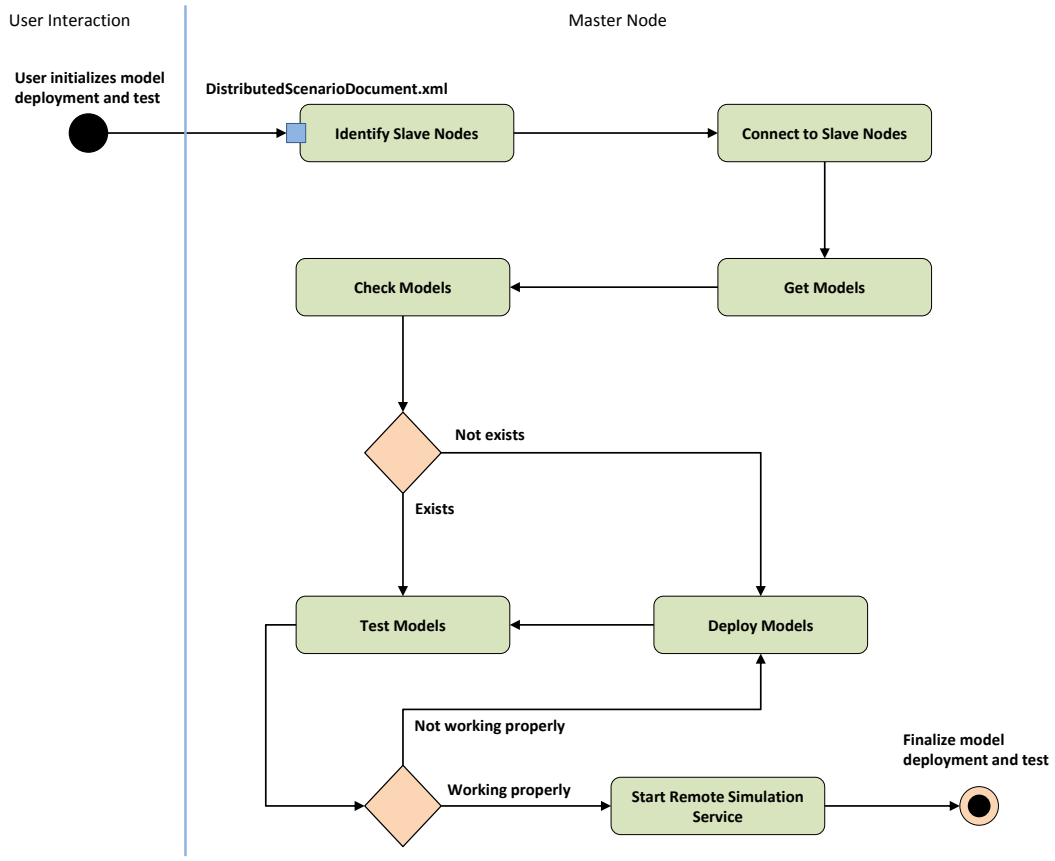


Figure 4.9: Deployment Activity Diagram

1. **Identify Slave Nodes:** From the input XML document called DistributedScenarioDocument the endpoint addresses of slave nodes are gathered. Endpoint address format is:

   ```
   <WCFBindingTypeSpecifier>//<IpAddress>:<PortNumber>/<ServiceName>
   Example: net.tcp//127.51.52.05:24000/ModelDeploymentService
   ```

   WCF bindings (defined by WCFBindingTypeSpecifier) represent the communication mechanism to be used while communicating with an endpoint and how WCF channels provide the required communication features. We use *nettcp-binding* in communication. This binding provides an optimized cross-machine communication between WCF applications. In order to use this binding, the two connected applications must be WCF applications.

2. **Connect to Slave Nodes:** Endpoint address information of a slave is enough to make connection between the master and the slave nodes. Master node connects

to all slave nodes through Model Deployment Service.

3. **Get Models:** Master node gets the model names from each slave.

4. **Check Models:** Master node has the information of which model will be located on which slave node. Hence, master node checks if each slave has the corresponding models.

5. **Test Models:** Master node tests those remote models to indicate that they are correctly installed on the slave node with the help of a testing interface proposed by models.

6. **Deploy Models:** If a Check and Test operation fails on a slave node, master node deploys the necessary models to the slave node via WCF.

7. **Start Remote Simulation Service:** At the end of the model deployment phase a new process is created in the master node for the simulation build and run. We use multi-processes instead of multi-threads because if an error occurs during simulation build and run, Deployment Service will continue execution to serve new connection requests. In slave nodes, preparation for the simulation build and run is finished with the followings:

   (a) Master node calls a method from the Model Deployment Service for the Remote Simulation Service start on slave nodes.

   (b) In slave node a new process is created.

   (c) An inter process communication (IPC) channel is established between main and new process for the data exchange. Additionally an operating system level mutex provided by .NET framework is used for the process synchronization.

   (d) Remote Simulation Service starts to listen for the incoming connection requests in created process.

   (e) Remote Simulation Service endpoint address information is sent to main process through the IPC channel.

   (f) The method called from Master node returns the endpoint information.

   (g) The main process on slave node starts to sleep until awakened by the created process when the simulation ends with or without any error.

4.3.2 Simulation Build and Run

Model Deployment guarantees that all simulation models are dispatched over the network and tested on the corresponding slave nodes. In this phase distributed simulation will be built and run. Distributed SiMA employs basic SiMA simulation construction pipeline showed in Figure 2.2 as a base and extends it to build distributed simulation. Again master node maintains the system building process. The distributed version of the SiMA simulation construction pipeline is in Figure 4.10.

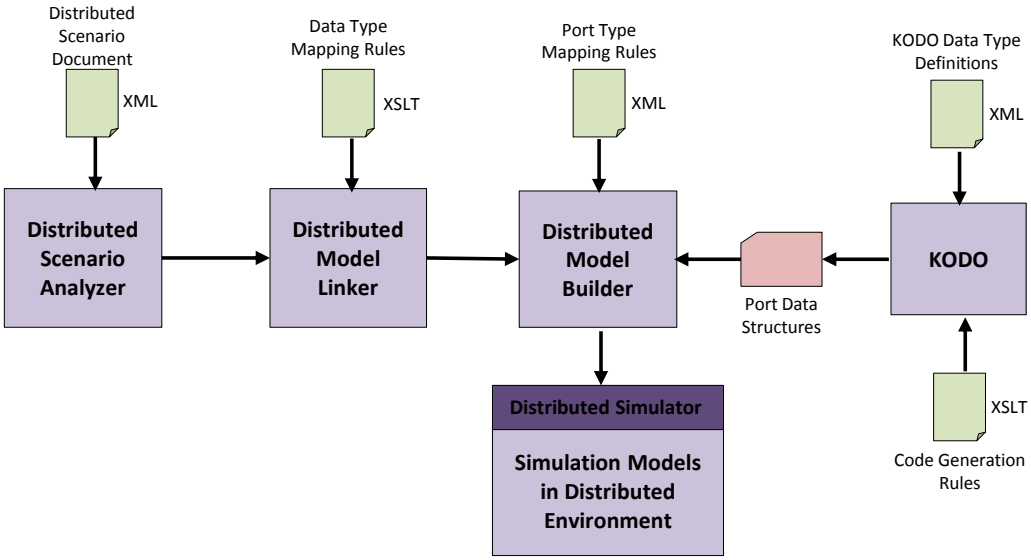


Figure 4.10: Distributed SiMA Simulation Construction Pipeline

4.3.2.1 Distributed Scenario Analyzer

Distributed Scenario Analyzer starts the distributed simulation construction pipeline. In Distributed SiMA, Scenario Document is modified as Distributed Scenario Document that includes the definition of the partition plan. User writes Deployment Service (DS) endpoint address of a slave node inside the definition of the model (atomic or coupled) he wants to work as remote. This modification on scenario document is enough to make a basic SiMA scenario distributed. Distributed Scenario Analyzer takes a Distributed Scenario Document file as an input and creates an intermediary data for Distributed Model Linker. The Distributed Scenario Document consists of model definition, model coupling information with remote node endpoint addresses,

and initialization data of the models.

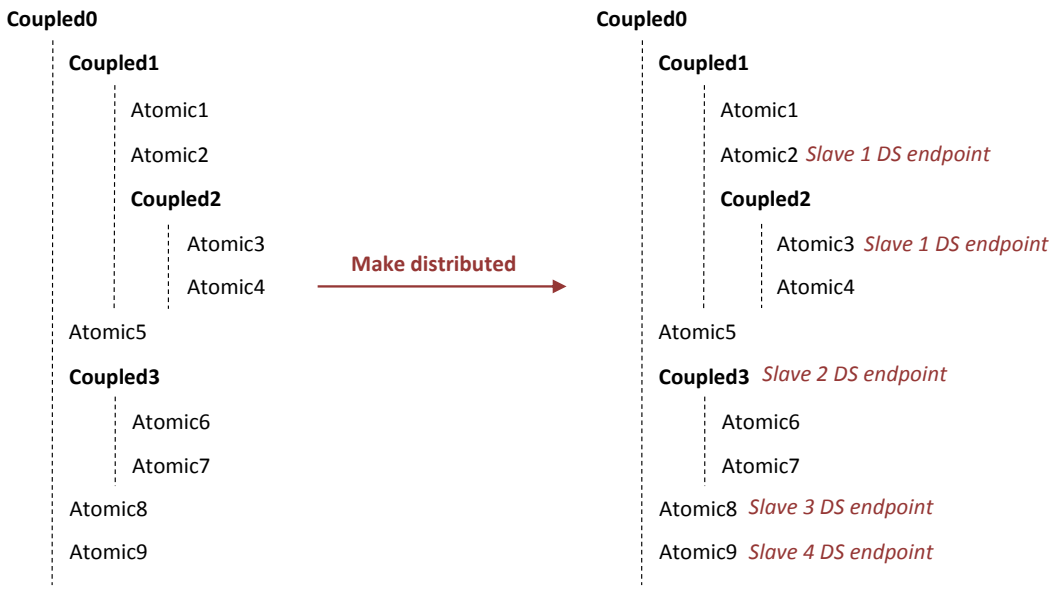


Figure 4.11: Hierarchical View of Making a Basic SiMA Scenario Distributed

Hierarchical view of a basic SiMA scenario definition can be observed on the left side of the Figure 4.11. Coupling information is defined in XML syntax inside the scenario definition file. According to the given partition Atomic2 and Atomic3 will work remotely on Slave 1, Coupled 3 on Slave 2, Atomic8 on Slave 3, and Atomic9 on Slave 4. Remaining models will work locally on Master. This document is also used in model deployment processes.

While defining the partition plan the only constraint is that a remote coupled model cannot contain a remote atomic or coupled model. As illustrated in the Figure 4.12 invalid definitions are highlighted and a valid scenario definition is defined on the right hand side.

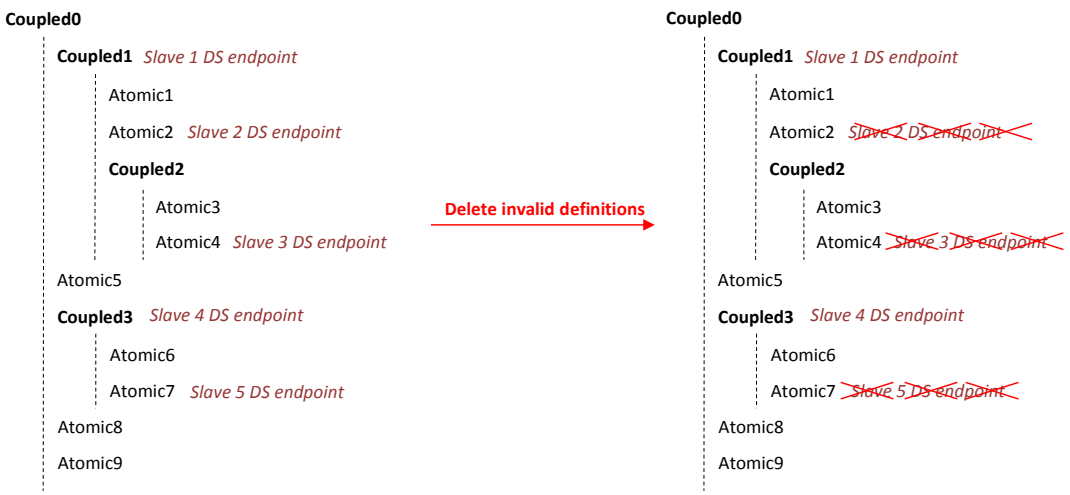


Figure 4.12: Invalid Partition Plan Definitions

4.3.2.2 Distributed Model Linker

Distributed Model Linker is the core component among the others taking role in the pipeline. It prepares the distributed simulation structure to be built. The intermediary scenario received from Distributed Scenario Analyzer is separated into two files in this component. One of the files is the distributed model link map file including the coupling information, model definitions, and port connection definitions using a hierarchical style. The other file is the distributed simulation configuration file consisting of initialization data of the atomic models.

User specifies the required data conversion connectors; however, the document does not include marshaller/unmarshaller connector definitions. Distributed Model Linker inserts them where they are required. There are some conditions to be considered while partitioning the remote models in order to make some optimizations.

Partitioning Conditions

Condition 1 *Each slave node has only one remote model*

This is the basic condition which does not require any operation except forming models defined as remote and their port connection definitions. Firstly, Distributed Model

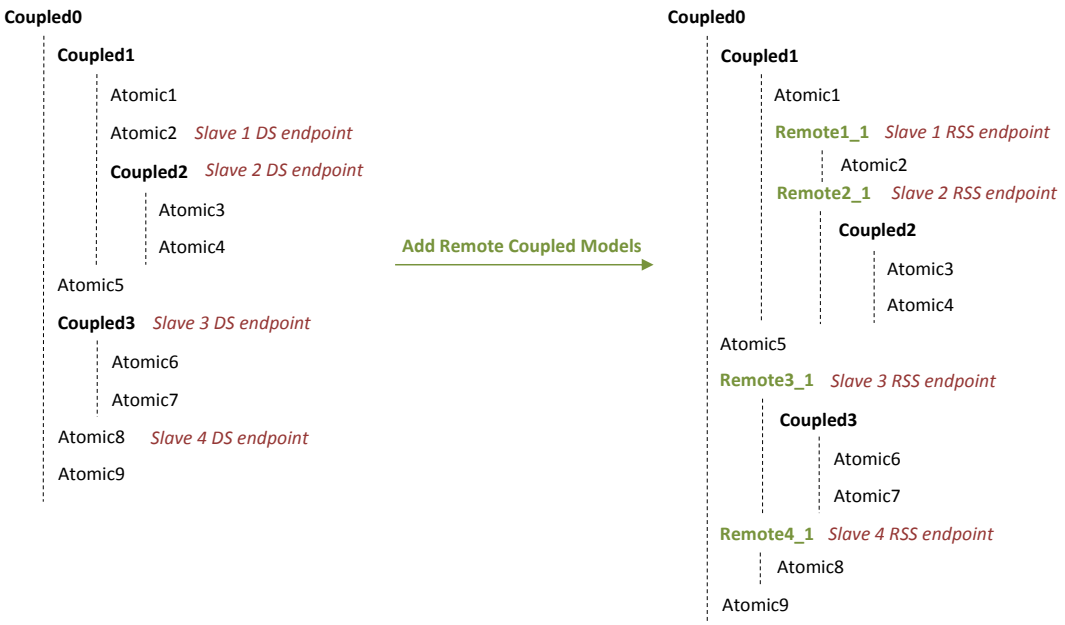


Figure 4.13: Condition 1: Adding Remote Coupled Models

Linker adds remote coupled models covering the models defined as remote for each slave node as displayed in Figure 4.13. In addition, it changes the Deployment Service (DS) endpoint addresses to Remote Simulation Service (RSS) endpoint addresses and writes them near remote coupled models.

Numbers in the names of the remote coupled models specify the index number of slave nodes and the remote session number held for each slave node respectively. The logic of remote sessions will be explained in the second condition. After restructuring the model hierarchy with remote coupled models, Distributed Model Linker analyzes the new structure and adds marshaller/unmarshaller connectors according to remote model connections as displayed in Figure 4.14. Moreover, the existing connections are altered and new connections are made with the new connectors.

Marshaller/unmarshaller connectors are inserted at the same levels as their corresponding source/target models. They are named with the same number as suffixes of marshaller part of the connectors with the counterpart unmarshaller part of the connectors. After making new couplings Port Service (PS) endpoint addresses are written near marshaller-unmarshaller connections. In Figure 4.15 new simulation structure after marshaller/unmarshaller connectors is illustrated clearly.

Figure 4.14: Condition 1: Adding Marshaller/Unmarshaller Connectors and Making New Couplings with New Models

Condition 2 *A slave node has more than one **distinct** remote model*

Slaves may have more than one remote model. If there is not any connection among them, this condition takes the charge. For each model defined as remote, one remote coupled model is created. WCF opens a session for each of them and they are processed unaware of each other. There is an instance context per session and this maintains lifetime of each session separately. Sessions are indicated with the last number in the remote coupled model name as in Figure 4.16.

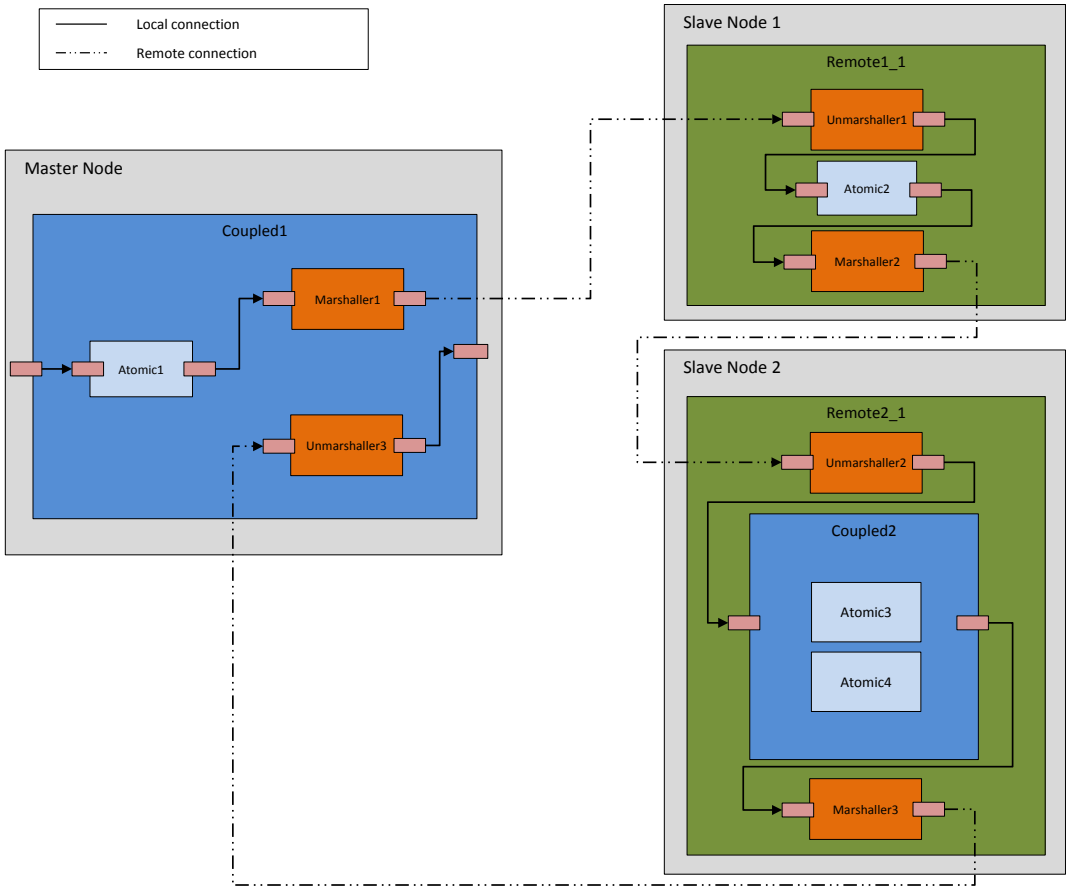


Figure 4.15: Condition 1: New Simulation Structure With Marshaller/Unmarshaller Connectors

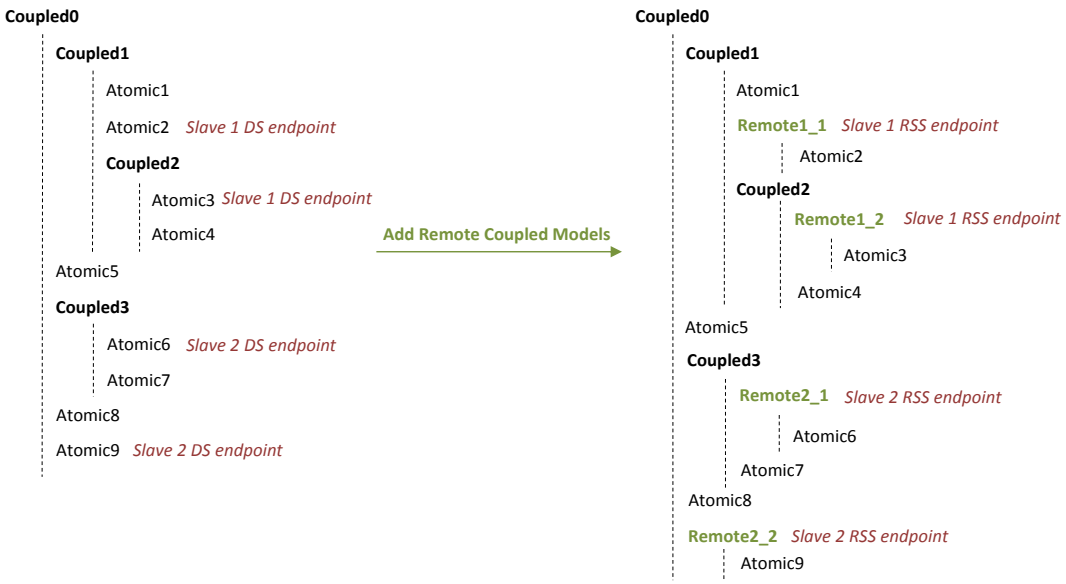


Figure 4.16: Condition 2: Adding Remote Coupled Models

Marshaller/unmarshaller connectors are inserted and new connections are made similar to condition 1.

Condition 3 *A slave node has more than one* ***connected*** *remote models*

In order to maximize local communication, if there are connected models defined as remote in the same slave node they are taken in the same remote coupled model as displayed in Figure 4.17.

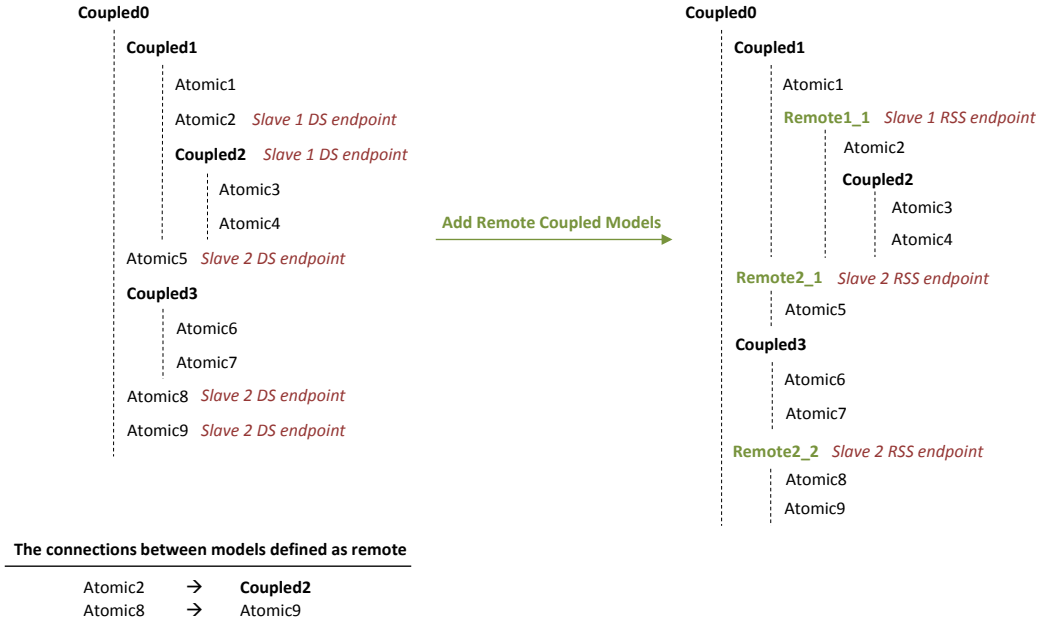


Figure 4.17: Condition 3: Adding Remote Coupled Models

4.3.2.3 Distributed Model Builder

Distributed Model Builder is the constructor of the Distributed SiMA. It manages the creation of instances of model classes. It establishes one root coupled model by combining the instances according to hierarchical structure for each node. This hierarchical structure is obtained from the distributed model link map file produced by Distributed Model Linker. For the models to be run in master node, it builds them locally. And for the remote models it commands the slave nodes to build the remote coupled models via Remote Simulation Service. There are also remote model proxies located in the master node for each remote coupled model. Their task is to provide a communication setup in order to communicate with Remote Simulation Service.

They are images of the remote coupled models in master node. Distributed Simulator manages the distributed simulation on master node as if it is local. It thinks that all models are located in one computer. Actually remote model proxies create this illusion.

At the initialization phase of the construction process Distributed Model Builder opens an Object ID Service (OIDS) on master node. This service provides an access facility for the object id generator working on Distributed Simulator. SiMA ensures that all port data objects created have unique ids in simulation. These ids are generated by SiMA simulator. To conserve this logic, OIDS is opened in master node and slave nodes request ids from simulator. After preparing OIDS, remote model building process commits in the following steps shown in Figure 4.18 by the master node.

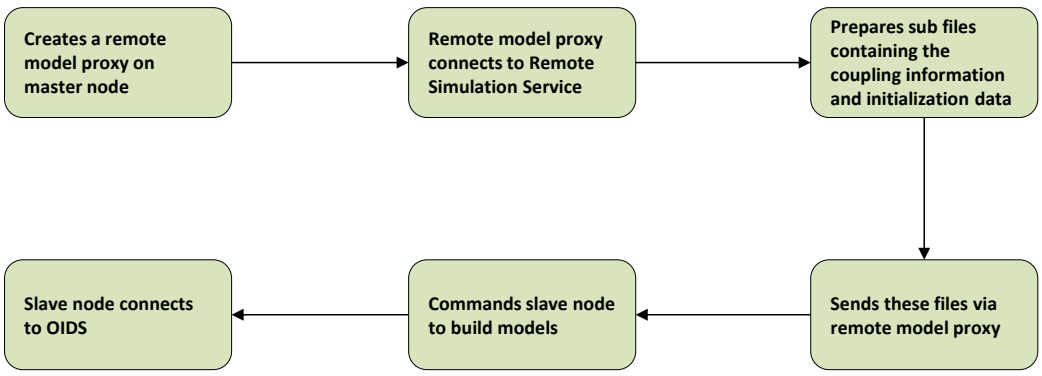


Figure 4.18: Remote Model Building

1. Distributed Model Builder creates a remote model proxy on master node.
2. Remote model proxy connects to Remote Simulation Service located on the slave node.
3. Distributed Model Builder prepares two sub files including only the remote coupled model coupling information and initialization data to be created on related slave node.
4. These files are sent via remote model proxy to the slave node.
5. Build model command is sent to slave node and remote coupled model instance is created with the given coupling information and initialization data on the slave node.
6. Slave node connects to OIDS.

This instance includes all remotely defined models without remote port connections. In order to connect remote ports marshaller/unmarshaller connectors should be ready on associated nodes. Remote port connections are established in two steps: a Port Service (PS) is opened in unmarshaller part of the connector and the counterpart marshaller part of the connector connects it. Here in Figure 4.19 there is an overview of the constructed distributed simulation system of the example from Figure 4.17.

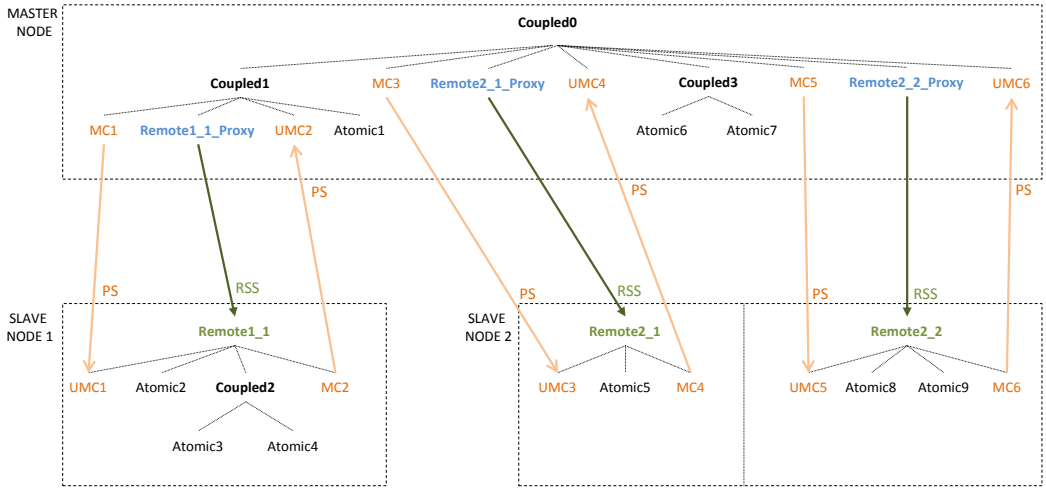


Figure 4.19: Distributed Simulation Structure Example Overview

4.3.2.4 KODO

KODO is not adjusted to distributed SiMA since the port and initial data classes generated are used as they are. Actually in WCF user specifies the port data classes transported through network by tagging them with *Data Contract* property. The .NET framework detects that the class will be serialized by WCF. However, we did not develop a new distributed version of KODO or modify the generated classes. Because it brings some work load and also ruins the easiness of making an already developed SiMA scenario distributed. One would have to run the new distributed KODO to add *Data Contract* property to all generated classes. We resolved this problem by changing the Port Service data serialization behavior. In default WCF expects to serialize only Data Contracts. With the changing service serialization behavior, all port data classes can be serialized into the *object* class which is the base class of all classes in .NET. This feature of WCF serialization behavior can be used because of WCF-to-

WCF communication. There is not a cross-platform communication and we can use the benefits of WCF-to-WCF communication in both data transport optimization and serialization.

4.3.2.5 Distributed Simulator

Distributed Simulator maintains the simulation execution as in SiMA. However, it is distributed since there are remote model proxies in place of the actual models. At the management level Parallel DEVS protocol is applied. Simulation execution is synchronized in terms of simulation time management as a default because of its central architecture and request/reply feature in remote procedure calls of WCF. There is no need for an extra global or local simulation time synchronization among remote nodes. Considering the distributed DEVS systems mentioned in Chapter 3 time management scheme of Distributed SiMA can be named as conservative but not with a global time synchronization mechanism. There is a sequence diagram of a Parallel DEVS protocol method in Figure 4.20. Distributed Simulator gets the next active time of the models to find the minimum next time. Different from the classical SiMA there will be a network delay cost in terms of the process time of the method.

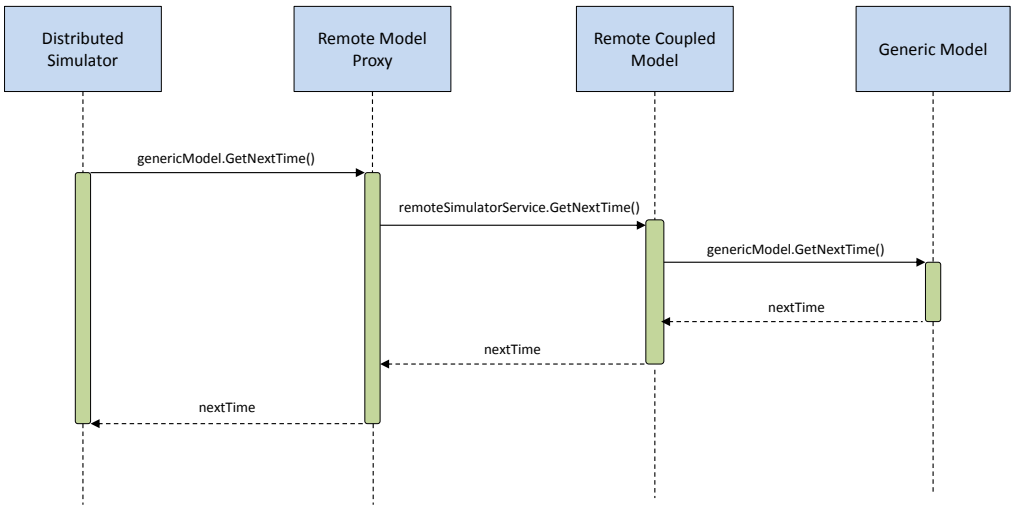


Figure 4.20: A DEVS Protocol Method Call via Remote Simulation Service

CHAPTER 5

CASE STUDY

This chapter includes a case study to demonstrate Distributed SiMA. The scenario used in the case study is a wireless ad hoc sensor network [49]. There are two kinds of sensor models in the scenario: *detailed sensor model* and *regular sensor model*. There is also a *sink model* gathering sensor information. A *logger model* is used to trace the local and distributed models' activities. A *platform model* is used as a target to be detected by sensors and make them send detection information to the sink model. Moreover, there are connectors; RegularToDetailedSensorInfo and DetailedToRegularSensorInfo connector between regular sensor models and detailed sensor models, and DetailedToRegularPlatformInfo connector between platform and regular sensor models. All scenario models and relations between them are observed from Figure 5.1. As the aim of this case study is not a WSN evaluation, the implementation of models does not reflect the exact calculations needed to be done in a real wireless ad hoc sensor network.

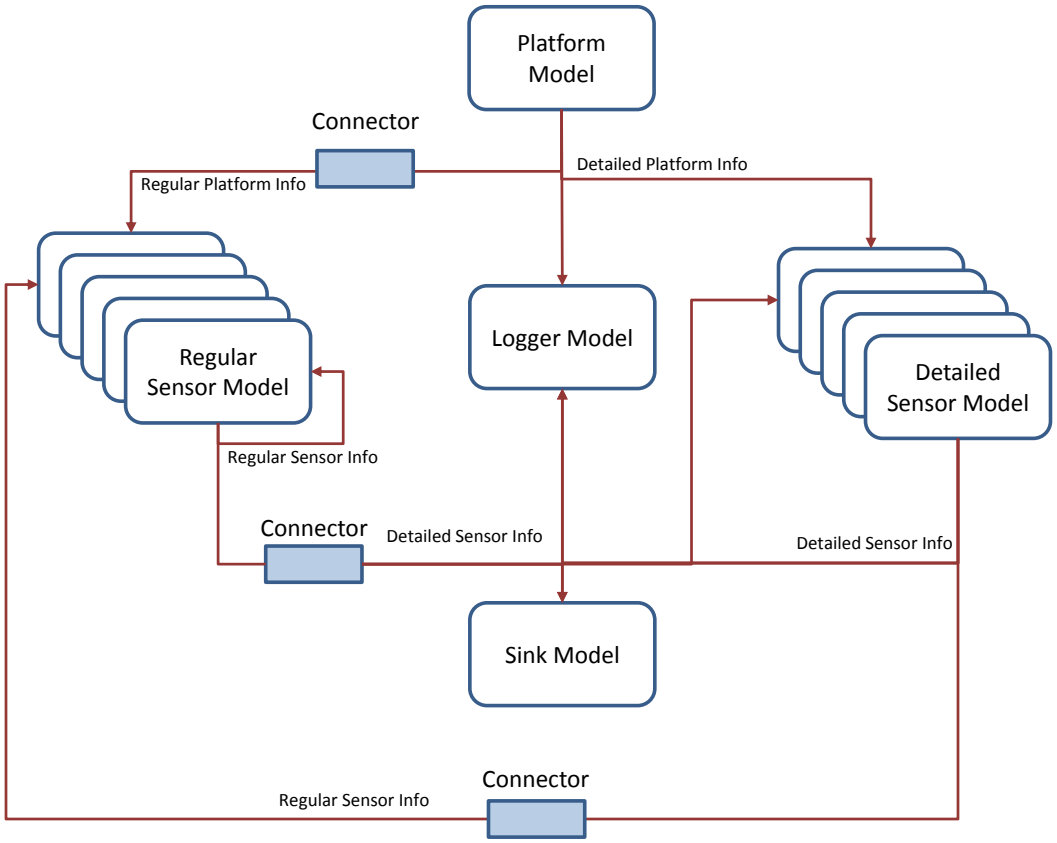


Figure 5.1: Models in the Case Study Scenario

5.1 Description of Models

Regular Sensor Model: Regular Sensor Model forwards the sensor information received from other sensor models to the closest sensor model until the information reaches to the sink model. It also receives the platform information from the Platform Model. If the platform is in range of the Regular Sensor Model, it processes the data and sends the data to the sink through sensors. It expects the sensor information in type of *Regular Sensor Info* from its input port and sends the sensor information with the same type. Also it receives the platform information in type of *Regular Platform Info*.

Detailed Sensor Model: The task of Detailed Sensor Model is the same as a Regular Sensor Model; however, it processes the data in a detailed form. Thus its calculations need more CPU time. It receives and sends the sensor information with type of

Detailed Sensor Info. And it receives the platform information with type of *Detailed Platform Info*.

Sink Model: The Sink Model initiates the WSN and keeps track of the platform position. It receives *Detailed Sensor Info* from all sensors and sends *Detailed Sensor Info* to them. The received sensor information consists of position of the sensor and the platform, signal strength, and some id numbers related to the sensor.

Logger Model: It receives the *Detailed Sensor Info* typed data from the sensor models and *Detailed Platform Info* typed data from the Platform Model, and writes the logs to an XML file.

RegularToDetailedSensorInfo Connector Model: This data conversion connector takes charge in converting the Regular Sensor Info type to Detailed Sensor Info. It converts the data received from Regular Sensor Models and sends it to Detailed Sensor Models, Sink Model and Logger Model.

DetailedToRegularSensorInfo Connector Model: This data conversion connector takes charge in converting the Detailed Sensor Info type to Regular Sensor Info. It converts the data received from Detailed Sensor Models and Sink Model, and sends it to Regular Sensor Models.

DetailedToRegularPlatformInfo Connector Model: This data conversion connector takes charge in converting the Detailed Platform Info type to Regular Platform Info. It converts the data received from Platform Model, and sends it to Regular Sensor Models.

5.2 Scenario Build

Distributed SiMA is tested with 2 slave nodes. Partition plan of the models is shown in Figure 5.2. Slave Node 1 has 25 Detailed Sensor Models, one Sink Model, one Logger Model and one RegularToDetailedSensorInfo Connector Model. Slave Node 2 has 25 Regular Sensor Models, one DetailedToRegularSensorInfo Connector Model and one DetailedToRegularPlatformInfo Connector Model. Additionally, Master Node runs one Platform Model.

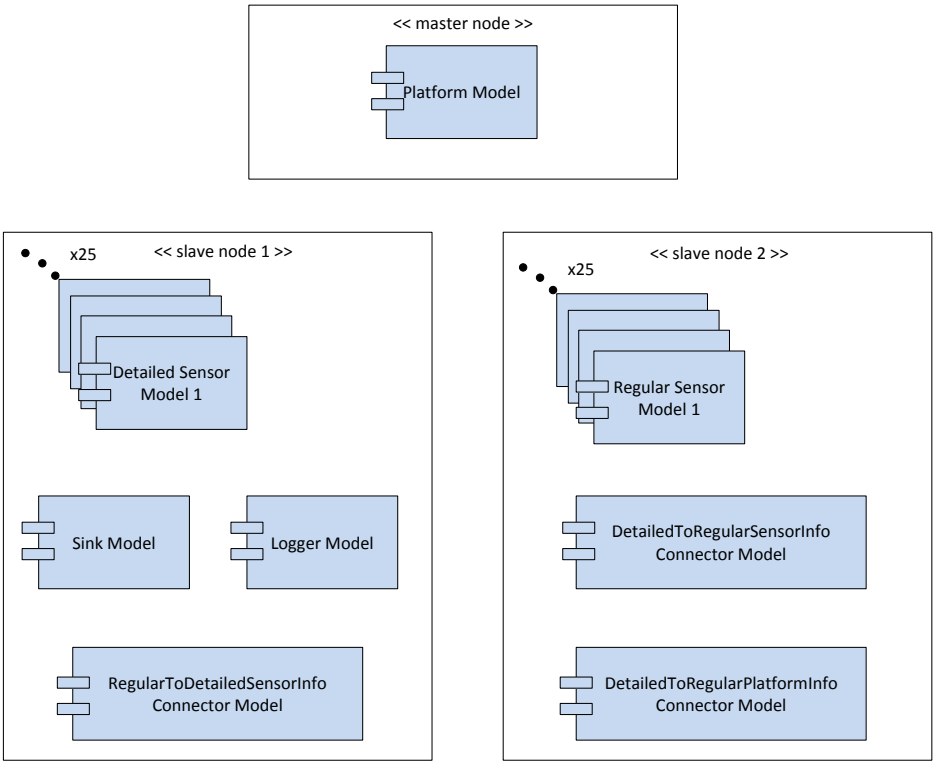


Figure 5.2: Partition Plan Overview of Case Study Scenario

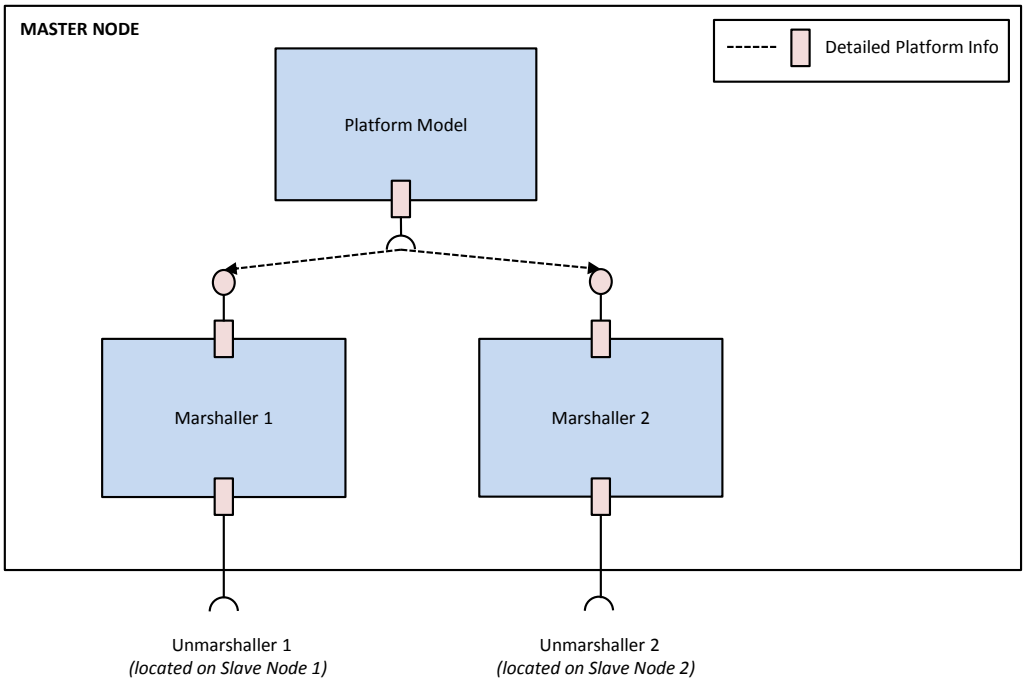


Figure 5.3: Master Node

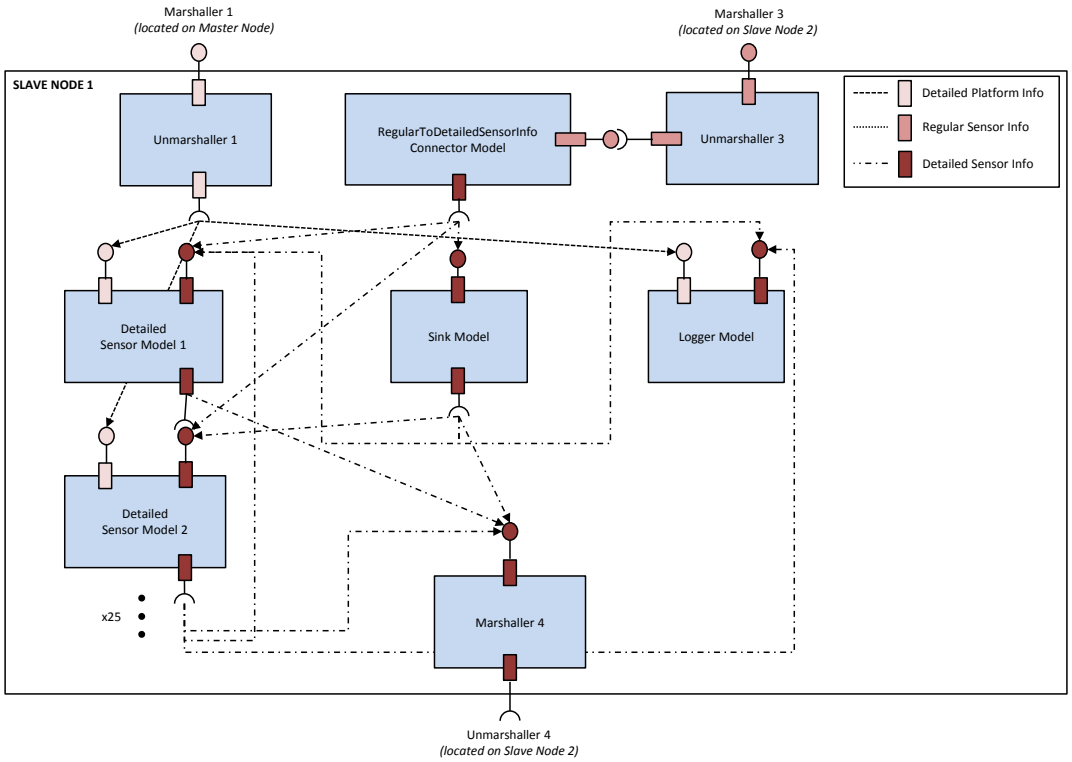


Figure 5.4: Slave Node 1

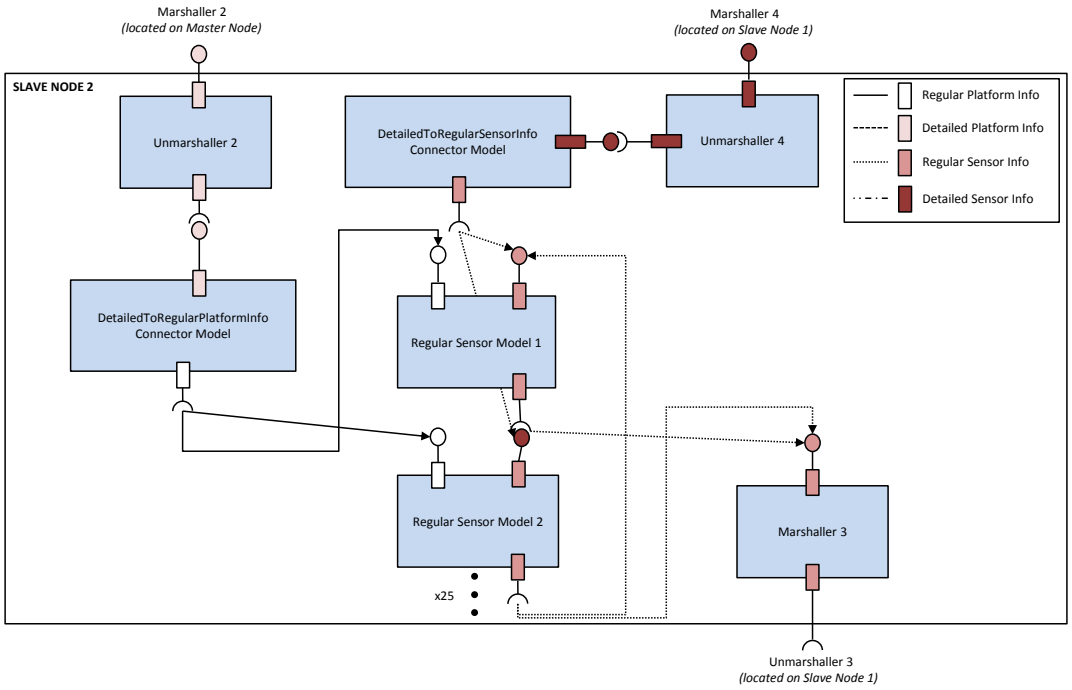


Figure 5.5: Slave Node 2

After the construction of the system, node overviews are as shown in Figures 5.3, 5.4, and 5.5 with created remote coupled models, inserted marshaller/unmarshaller connectors and established port connections.

5.3 Discussion

To specify component interactions user-defined connectors are the most flexible approach [2]. There are also some studies for the definitions of connector roles in a component-based architecture, for example Balek's studies [6, 5]. Our data conversion connectors are the adaptors that tie two or more atomic model components designed to interoperate. However, it can be considered that adding new models to a scenario brings burden to the simulator and slows the simulation execution. Moreover, one has to develop the new data conversion connector model.

For example, there are two already developed and tested scenarios. The first one uses Detailed Sensor Models and the second one uses Regular Sensor Models. In a new scenario we want to use both sensor models. There are two ways in order to prepare a scenario like this: modifying one type of sensor model to convert the received data when it is making calculations and again convert the calculated data back to send, or developing data conversion connectors. We implemented the case study scenario without data conversion connectors. We removed all the data conversion connectors and modified the Regular Sensor Model to adapt the new environment. Input and output port data types changed to detailed ones. Required conversions are done before calculations and before sending the calculated data. By doing this we achieved two results:

1. We executed the simulation with and without data conversion connectors and the simulation execution time did not change. Thus, a connector is not a burden to the simulator.

2. The Regular Sensor Model had 182 lines at first. After the modification 18 lines were changed and 26 new lines are added. As a ratio *10%* of code lines is modified and *14%* of code lines is added. Thus, when atomic models consist of massive data and calculations are considered, modifying them for the

adaptation to the new scenario models costs developer more than anticipated.

5.4 Evaluation

We have conducted some tests on the case study scenario and obtained the data shown in Figure 5.6. The scenario is executed both with data conversion connectors as *DCC* and without them. The X axis of the chart shows the number of Detailed Sensor Model as *DS* and Regular Sensor Model as *RS* in the scenario. The Y axis of the chart shows the wall clock time of simulation execution in seconds. The result data shows that when number of sensor models in the scenario increases, Distributed SiMA executes faster than SiMA. Moreover, in distributed simulation execution, the effect of data conversion connectors is unremarkable with only 0.5% increase in execution time. There is an unanticipated result: in the scenario with 50DS-50RS SiMA executed, simulation is executed faster with data conversion connectors. Because with data conversion connectors 200 port connections are established between Detailed Sensor Models and Regular Sensor Models. However, when data conversion connectors are removed 5000 port connections are established and this slows down the execution.

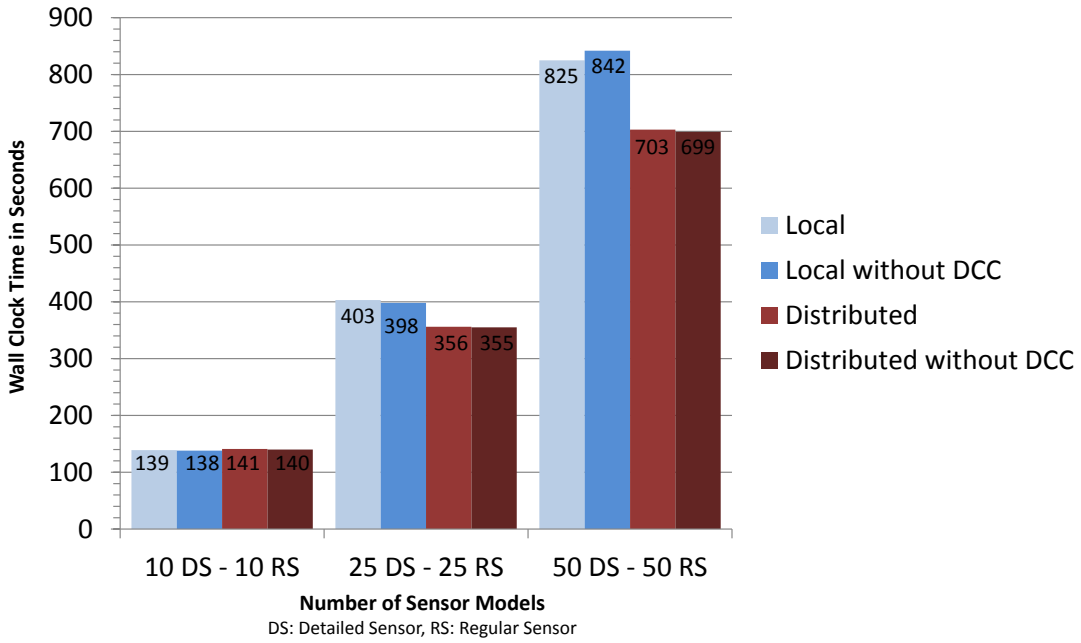


Figure 5.6: Case Study Scenario Test Results

CHAPTER 6

CONCLUSION

In this thesis, we have proposed a distributed approach for SiMA via WCF. WCF takes charge in communication and transportation among distributed nodes. Moreover, our approach use connectors which provide adaptation of distributed nodes and models and increase model reuse. Compared to other distributed DEVS implementations Distributed SiMA brings novelty in these three points:

1. Distributed SiMA uses SiMA-DEVS formalism
2. In order to increase model reusability, and distributed node and model adaptation, connectors are used
3. WCF is adopted in the network communication and data transportation layer

In development process of Distributed SiMA, by the help of direct feed transition functionality, connectors are not a burden to simulation execution in terms of time step. Both data conversion and data marshalling/unmarshalling operations are done in the same simulation step. In fact, this feature can be achieved with classical DEVS formalism by setting next execution time of those models to zero. Nevertheless, direct feed through transition function simplifies the management of this feature while preventing simulation from deadlock conditions. Moreover, a data conversion connector increases model reusability by placing between a new and a legacy model. This also hinders developers from applying regression tests for the validation and verification. If a data conversion connector was not placed, the legacy model would have to be modified to be compatible with the new scenario models. Benefits of connectors can be listed as:

1. Modularity and object-oriented design approach usage which is supported strongly by the DEVS [50] increases when connectors are used [25].

2. Development time is decreased since developing a new atomic model is faster than modifying an already developed atomic model. The existing atomic model would be implemented by another person, so it may be hard to understand the code and change it. Also the model may have a lot of lines of code and again this increases the changing time.

3. When we change an existing atomic model, we disrupt the code integrity. It was a verified atomic model when it was used in previous scenario. Therefore, the modified atomic model needs to be verified, and unit and regression tests have to be done. Connectors save developer from these issues.

4. Since we do not change the existing model and use it as it is, model reusability is promoted.

According to the statistics given in Section 5.3, this situation causes 10% of code lines to be modified and 14% of code lines to be added for our case. In addition, this operation removes the validation and verification properties of it. Moreover, it is easier to develop a data conversion connector in a shorter time. On the other hand, marshaller/unmarshaller connector facilitates the port data serialization/deserialization operations along the network. For example, if a marshaller part of the connector was not placed at the source node, connection with the corresponding unmarshaller part of the connector would have to be made and the data serialization operation would have to be done in source model. When a marshaller part and an unmarshaller part of a connector are placed mutually at remote nodes, client/server connection is automatically done in Distributed SiMA. This procedure has to be implemented in model if marshaller or unmarshaller part of the connector is removed.

There are roles of connectors in the literature. While data conversion connectors are adaptor and linkage connectors, marshaller/unmarshaller connectors are also procedure call connectors in addition to adaptor and linkage in terms of Mehta's and his colleagues' classification of connectors [33].

In Distributed SiMA most of the WCF features are utilized. These features and utilization ways are given as:

1. *Service oriented development environment:* This provides a simple development environment. It presents a service interface in a server and a client makes remote procedure calls after connecting the service. In service interface since data type to be sent and received is specified, no extra data type transport definition is needed.
2. *Request/reply message exchange pattern:* This feature plays the most important role in simulation execution in a distributed environment. When a client makes a remote procedure call, it waits for a reply from the server. With the help of this no extra synchronization mechanism needs to be implemented. It ensures that the distributed simulation execution is always in sync.
3. *Binary encoding over TCP transportation:* This feature is used since all distributed nodes run a WCF application. The main benefit of it is the fastest way of communication among WCF-to-WCF applications.
4. *Extensibility:* In order to make explicit use of binary encoding possible, we customized the service behavior. Thus, a .NET serialized object can be transported through network.

6.1 Distributed SiMA Approach

In Chapter 3 we have described various distributed DEVS approaches. And here is a table summing up those information with the Distributed SiMA approach.

DEVS is the underlying formalism in all distributed DEVS approaches, but DEVS\SOA uses an upgraded version of DEVS, parallel DEVS formalism, and Distributed SiMA uses SiMA formalism.

Middleware technologies vary from Globus providing control over grid computers, and JXTA enabling processing in a peer-to-peer network system to RMI and WCF which are common in RPCs in an object-oriented fashion.

When partitioning is considered, there are algorithms applied on the DEVS root cou-

Table 6.1: Comparing Different Distributed DEVS Approaches

	Formalism	Middleware Technology	Partitioning	Synchronization Scheme	Message Exchange
DEVS/CLUSTER	DEVS	CORBA	hierarchical to non-hierarchical structure	optimistic	CORBA remote method invocations
DEVS/GRID	DEVS	Globus	cost-based hierarchical partitioning	conservative	GIIS
DEVS/P2P	DEVS	JXTA	autonomous hierarchical model partitioning	conservative	JXTA message format
DEVS/RMI	DEVS	JAVA/RMI	applying built-in partition algorithm	conservative	JAVA serializable object
DEVS/PyRO	DEVS	PyRO/RMI	user specified or automatic	conservative	serialized objects
DEVS/SOA	Parallel DEVS	GIG/SOA	user specified	conservative	JAVA serialization
Distributed SiMA	**SiMA**	**WCF**	**user specified**	**conservative**	**.NET binary serialized objects**

pled model to partition the models efficiently in terms of resource usage and cross-node communication. DEVS/CLUSTER firstly transforms the hierarchical DEVS structure into a non-hierarchical one and after that partitions the models. While DEVS/GRID uses a cost-based hierarchical partitioning algorithm, DEVS/RMI applies a built-in algorithm for partitioning. Moreover, in DEVS/PyRO, DEVS/SOA and Distributed SiMA user specifies the partition plan.

As a synchronization scheme only DEVS/CLUSTER implements an optimistic approach which uses Time Warp algorithm to synchronize when it is broken. DEVS/GRID ensures the synchronization with protocol messages. DEVS/RMI and DEVS/PyRO is always synchronized with the help of RMI technology. Similarly, Distributed SiMA is always in sync due to WCF.

Each message exchange format depends on the implemented language and working environment. DEVS/P2P uses JXTA message format since it uses JXTA as a middleware. DEVS/CLUSTER sends or receives data by remote method invocations in

CORBA. DEVS/RMI and DEVS/SOA use JAVA serialized objects since they are distributed versions of DEVS/JAVA and the language allows it. Besides, Distributed SiMA uses .NET binary serialized data since in a WCF-to-WCF application it is a fast way of transfer compared to other options [11].

6.2 Future Work

6.2.1 Connector Repository

Distributed SiMA automatically places marshaller and unmarshaller part of the marshaller/unmarshaller connector mutually on the remote nodes. However, data conversion connectors are implemented and defined in Distributed Scenario Document by developers. In order to automatize these operations there can be a repository including data conversion connectors located in a remote server [22]. The Distributed Scenario Document is analyzed before simulation construction and incompatible port data types among models are detected. This connector repository is searched for the required data conversion connector models and they can be downloaded. And also definitions of them can be added to the file automatically.

6.2.2 Model Editor and KODO Adaptations

Although there is a Model Editor in basic SiMA simulation construction pipeline, it is not adapted for Distributed SiMA yet. Model Editor generates Scenario Document after a user prepares a scenario via GUI. To enable a SiMA simulation to work on Distributed SiMA only endpoint addresses of remote nodes are written next to model definitions in Scenario Document. Model Editor can be adapted to Distributed SiMA for arranging a distributed scenario visually.

KODO generates the initialization and port data classes for both SiMA and Distributed SiMA. In Distributed SiMA for the transportation of port data through network, service behavior of Port Service is modified and all port data are serialized into .NET objects. However, WCF offers a property that can be added to the classes to be transported through network without changing service behavior. This should

save users from extra type castings and speeds up the marshalling/unmarshalling processes.

REFERENCES

[1] Khaldoon Al-Zoubi and Gabriel Wainer. Performing Distributed Simulation with RESTful Web-services. In *Winter Simulation Conference*, WSC '09, pages 1323–1334. Winter Simulation Conference, 2009.

[2] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[3] Abdelkrim Amirat, Mourad Oussalah, et al. Reusable Connectors in Component-Based Software Architecture. In *Proceedings of the ninth international symposium on programming and systems,(ISPS 2009)*, pages 28–35, 2009.

[4] ASU. Arizona Center For Integrative Modeling and Simulation. http://acims.asu.edu/software/devsjava, last visited on June 2014.

[5] Dusan Bálek. Connectors in software architectures. Technical report, Charles University, Czech Republic, 2002.

[6] Dusan Bálek and Frantisek Plasil. Software Connectors and Their Role in Component Deployment. In *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 69–84, Deventer, The Netherlands, 2001. Kluwer, B.V.

[7] Federico Bergero and Ernesto Kofman. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *SIMULATION*, 2010.

[8] Jean Sébastien Bolduc and Hans Vangheluwe. The Modelling and Simulation Package PythonDEVS for Classical Hierarchical DEVS. *MSDL technical report MSDL-TR-2001-01*, June 2001.

[9] Tomas Bures and Frantisek Plasil. Scalable element-based connectors. In *Proceedings of SERA*, 2003.

[10] L. Capocchi, J. F Santucci, B. Poggi, and C. Nicolai. DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2011 20th IEEE International Workshops on*, pages 170–175, June 2011.

[11] David Chappell. Dealing with Diversity: Understanding WCF Communication Options in the .NET Framework 3.5. September 2007.

[12] Bin Chen and Xiao-gang Qiu. MPI-Based Distributed in DEVS Simulation. In *Proceedings of the 2009 Third International Symposium on Intelligent Information Technology Application - Volume 02*, IITA '09, pages 78–81, Washington, DC, USA, 2009. IEEE Computer Society.

[13] Steven Cheng. *Microsoft Windows Communication Foundation 4.0 Cookbook for Developing SOA Applications*. Packt Publishing Ltd, 2010.

[14] Saehoon Cheon, Chungman Seo, Sunwoo Park, and Bernard P. Zeigler. Design and implementation of distributed DEVS simulation in a peer to peer network system. *Advanced Simulation Technologies Conference–Design, Analysis, and Simulation of Distributed Systems Symposium. Arlington, USA*, 2004.

[15] Y.K. Cho, Bernard P. Zeigler, and H.S. Sarjoughian. Design and Implementation of Distributed Real-time DEVS/CORBA. In *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, volume 5, pages 3081–3086 vol.5, 2001.

[16] Alex Chung Hen Chow and Bernard P. Zeigler. Parallel DEVS: A Parallel, Hierarchical, Modular, Modeling Formalism. In *Proceedings of the 26th Conference on Winter Simulation*, WSC '94, pages 716–722, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[17] Fatih Deniz, M Nedim Alpdemir, Ahmet Kara, and Halit Oğuztüzün. Supporting dynamic simulations with Simulation Modeling Architecture (SiMA): a Discrete Event System Specification-based modeling and simulation framework. *Simulation*, 88(6):707–730, 2012.

[18] Bo Feng and G. Wainer. A .NET Remoting-Based Distributed Simulation Approach for DEVS and Cell-DEVS Models. In *Distributed Simulation and Real-Time Applications, 2008. DS-RT 2008. 12th IEEE/ACM International Symposium on*, pages 292–299, Oct 2008.

[19] Jan Himmelspach and Adelinde M. Uhrmacher. Processing Dynamic PDEVS Models. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '04, pages 329–336, Washington, DC, USA, 2004. IEEE Computer Society.

[20] Gordon Hogenson. *Foundations of C++/CLI: The Visual C++ Language for .NET 3.5*. Springer, Dordrecht, 2008.

[21] Joon Sung Hong, Hae-Sang Song, Tag Gon Kim, and Kyu Ho Park. A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time

Software Development. *Discrete Event Dynamic Systems*, 7(4):355–375, October 1997.

[22] Ahmet Kara. *A Methodology for Cross-Resolution Modeling in DEVS using Event-B Refinement*. PhD thesis, Graduate School of Natural and Applied Sciences, Middle East Technical University, 2014.

[23] Ahmet Kara, Doruk Bozağaç, and Mahmut Nedim Alpdemir. Simülasyon Modelleme Altyapısı (SiMA): DEVS Tabanlı Hiyerarşik ve Modüler bir Modelleme ve Koşum Altyapısı. *İkinci Ulusal Savunma Uygulamaları Modeleme ve Simulasyon Konferansı (USMOS)*, 2007.

[24] Ahmet Kara, Fatih Deniz, Doruk Bozağaç, and M. Nedim Alpdemir. Simulation Modeling Architecture (SiMA), a DEVS Based Modeling and Simulation Framework. In *Proceedings of the 2009 Summer Computer Simulation Conference*, SCSC '09, pages 315–321, Vista, CA, 2009. Society for Modeling & Simulation International.

[25] Ahmet Kara, Halit Oğuztüzün, and M. Nedim Alpdemir. Heterogeneous DEVS Simulations with Connectors and Reo Based Compositions (WIP). In *Proceedings of the 2014 Spring Simulation Multiconference*, SpringSim '14, pages 291–296, San Diego, CA, USA, 2014. Society for Computer Simulation International.

[26] Ki-Hyung Kim and Won-Seok Kang. CORBA-based, multi-threaded distributed simulation of hierarchical DEVS models: transforming model structure into a non-hierarchical one. In *Computational Science and Its Applications–ICCSA 2004*, pages 167–176. Springer, 2004.

[27] Ki Hyung Kim, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park. Distributed Simulation of Hierarchical DEVS Models: Hierarchical Scheduling Locally and Time Warp Globally. *Trans. Soc. Comput. Simul. Int.*, 13(3):135–154, September 1996.

[28] Tag Gon Kim, Chang Ho Sung, Su-Youn Hong, Jeong Hee Hong, Chang Beom Choi, Jeong Hoon Kim, Kyung Min Seo, and Jang Won Bae. DEVSim++ Toolset for Defense Modeling and Simulation and Interoperation. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 8(3):129–142, 2011.

[29] E. Kofman, M. Lapadula, and E. Pagliero. PowerDEVS: A DEVS-Based Environment for Hybrid System Modeling and Simulation. Technical Report LSD0306, School of Electronic Engineering, Universidad Nacional de Rosario, Rosario, Argentina, 2003.

[30] Ernesto Kofman and Sergio Junco. Quantized-state Systems: A DEVS Approach for Continuous System Simulation. *Trans. Soc. Comput. Simul. Int.*, 18(3):123–132, September 2001.

[31] Rami Madhoun and Gabriel Wainer. Studying the Impact of Web-services Implementation of Distributed Simulation of DEVS and Cell-DEVS Models. In *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*, SpringSim '07, pages 267–278, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[32] Anu Maria. Introduction to modeling and simulation. In *Proceedings of the 29th Conference on Winter Simulation*, WSC '97, pages 7–13, Washington, DC, USA, 1997. IEEE Computer Society.

[33] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 178–187, New York, NY, USA, 2000. ACM.

[34] Jose Luis Latorre Millas. *Microsoft .Net Framework 4.5 Quickstart Cookbook*. Packt Publishing Ltd, 2013.

[35] Saurabh Mittal, José L. Risco-Martín, and Bernard P. Zeigler. DEVS/SOA: A Cross-Platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process. *Simulation*, 85(7):419–450, July 2009.

[36] MSDL. Modeling, Simulation and Design Lab. http://msdl.cs.mcgill.ca/projects/projects/DEVS/PythonDEVS, last visited on June 2014.

[37] Sunwoo Park. *Cost-based Partitioning for Distributed Simulation of Hierarchical Modular DEVS Models*. PhD thesis, 2003. AAI3090011.

[38] Herbert Praehofer. System Theoretic Formalisms For Combined Discrete-Continuous System Simulation. *International Journal of General Systems*, 19(3):226–240, 1991.

[39] H Sarjoughian and Bernard P. Zeigler. DEVSJAVA: Basis for a DEVS-based collaborative M&S environment" proceedings of the International Conference on Webbased Modeling & Simulation. *San Diego, CA*, 1998.

[40] Hessam Sarjoughian, Sungung Kim, Muthukumar Ramaswamy, and Stephen Yau. A Simulation Framework for Service-oriented Computing Systems. In *Proceedings of the 40th Conference on Winter Simulation*, WSC '08, pages 845–853. Winter Simulation Conference, 2008.

[41] Chungman Seo, Sunwoo Park, Byounguk Kim, Saehoon Cheon, and Bernard P. Zeigler. Implementation of distributed high-performance DEVS simulation framework in the Grid computing environment. 2004.

[42] John Sharp. *Microsoft Windows Communication Foundation Step by Step*. Microsoft Press, 2007.

[43] Eugene Syriani, Hans Vangheluwe, and Amr Al Mallah. Modelling and Simulation-based Design of a Distributed DEVS Simulator. In *Proceedings of the Winter Simulation Conference*, WSC '11, pages 3007–3021. Winter Simulation Conference, 2011.

[44] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[45] A. M. Uhrmacher. Dynamic Structures in Modeling and Simulation: A Reflective Approach. *ACM Trans. Model. Comput. Simul.*, 11(2):206–232, April 2001.

[46] Gabriel Wainer. CD++: A Toolkit to Develop DEVS Models. *Softw. Pract. Exper.*, 32(13):1261–1306, November 2002.

[47] Gabriel Wainer and Norbert Giambiasi. Timed Cell-DEVS: Modeling and Simulation of Cell Spaces. In *Discrete Event Modeling and Simulation Technologies*, pages 187–214. Springer, 2001.

[48] Brendon J. Wilson. *JXTA*. Pearson Education, 2002.

[49] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Comput. Netw.*, 52(12):2292–2330, August 2008.

[50] Bernard P. Zeigler. *Theory of Modelling and Simulation*. A Wiley-Interscience Publication. John Wiley, 1976.

[51] Bernard P. Zeigler. Discrete Event Formalism For Model Based Distributed Simulation. In *SCS Conf. Distributed Simulation*, pages 3–7, 1985.

[52] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 2000.

[53] Bernard P. Zeigler, Yoonkeon Moon, Doohwan Kim, and George Ball. The DEVS Environment for High-Performance Modeling and Simulation. *IEEE Comput. Sci. Eng.*, 4(3):61–71, July 1997.

[54] Bernard P. Zeigler and Hessam S Sarjoughian. Introduction to DEVS Modeling and Simulation With Java: Developing Component-based Simulation Models. *Technical Document, University of Arizona*, 2003.

[55] Ming Zhang, Bernard P. Zeigler, and Phillip Hammonds. DEVS/RMI-An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies. *ITEA Journal*, 2005.