Multi-agent Environment for Complex SYstems COsimulation (MECSYCO) - Architecture Documentation

Benjamin Camus[1,2], Julien Vaubourg[2], Yannick Presse[2],
Victorien Elvinger[2], Thomas Paris[1,2], Alexandre Tan[2]
Vincent Chevrier[1,2], Laurent Ciarletta[1,2], Christine Bourjot[1,2]
[1]Universite de Lorraine, CNRS, LORIA UMR 7503,
Vandoeuvre-les-Nancy, F-54506, France.
[2]INRIA, Villers-les-Nancy, F-54600, France.
`mecsyco@inria.fr`

April 5, 2016

Contents

Introduction

MECSYCO (Multi-agent Environment for Complex SYstems COsimulation) is a software to model and simulate complex systems. It enables to realize heterogeneous and digital simulations from existing simulators.

The main problematic of MECSYCO is the integration of distinct sub-systems. We propose to design the system as a multi-model and to use a multi-agent paradigm to run it.

The multi-model approach of MECSYCO is applicable on different levels:

- **multi-domain level.**
- **coupling level:** how the information are exchanged between the models.
- **formalism level:** discrete or continuous.
- **simulator level :** pre-existing softwares.
- **programming language level:** some tool are only available with a specific language.
- **hardware level :** some tool needs a particular hardware.

Chapter 1

MECSYCO's features

1.1 MECSYCO's purpose

MECSYCO manages the **decentralized multi-simulation** of a **multi-model** in a **parallel** and/or **distributed** way. Let us introduce the different terms of this previous definition:

- **Multi-model:** A multi-model is a model composed of several models in interaction. Each model may use its own formalism and spatial and temporal scales. [Siebert, 2011]
- **Multi-simulation:** A multi-simulation consists of simulating a multi-model using a simulator for each model
- **Decentralized simulation:** A multi-simulation executed without global coordinator.
- **Parallel simulation:** A multi-simulation performed using different processes but with a shared memory [Fishwick, 2007]
- **Distributed simulation:** A multi-simulation *"performed on loosely connected nodes, such as a cluster of workstations connected by a local network, or a geographically separated set of servers connected over a wide area network."* [Fishwick, 2007]

1.2 Modeling hypothesis

In MECSYCO we have made the hypothesis that all the models composing the multi-model are compliant with the DEVS formalism (see Appendix B for more details on this formalism). This formalism has the advantage of being an execution model proved to be compatible with all other formalisms. The DEVS simulation protocol imposes some assumptions on the model:

- Model's inputs can NOT immediately (in term of simulation time) generate model's outputs
- Models have to conform to the primitives of the protocols. This implies the possibility to define the operations listed in section 2.3.3.

1.3 Simulation strategy

The execution of a multi-simulation must respect the **causality constraint**, that is to say: **each simulator must processes events in timestamp order** [Fujimoto, 2001]. Two types of approaches exist to fulfill this requirement [Fishwick, 2007]:

- **The conservative approaches** consists of insuring that the causality is never broken during the simulation.
- **The optimistic approaches** consists of detecting when the causality constraint is broken and then rolling back the simulation to this point.

The optimistic approaches required all the simulators to have a roll back capability implemented either with a state saving or inverse computation strategy [Fishwick, 2007]. As this requirement strongly restricts the type of the simulators which can be used, we have made the choice in the current MECSYCO specification, to take a **conservative approach**. However, the DEVS optimistic strategy [Zeigler et al., 2000] may also be implemented in future version of MECSYCO as it is compatible with the concept and hypotheses of MECSYCO.

The MECSYCO conservative simulation algorithm is based on the DEVS version [Zeigler et al., 2000] of the Chandy Misra Bryant algorithm [Chandy and Misra, 1979]. This algorithm is detailed in Appendix C. Conservative approaches have a well-known limitation: every simulator involved in the decentralized multi-simulation must define a non-null **minimum propagation delay** in order to computes its lookahead (for more details on this point see Section 2.3.1 and Appendix C).

Despite being proved to respect the causality constraint, the algorithm may break the causality if the **minimum propagation delay** is wrongly defined. In order to ensure that a multi-simulation respect the causality constraint, MECSYCO will throw a **Causality Exception** if the causality is broken.

1.4 Integrated simulators

This section presents simulators already include in MECSYCO, that is to say that all extensions and libraries needed for simulate the model are included.

1.4.1 NetLogo

NetLogo is a multi-agent programmable modeling environment used for simulating natural and social phenomena. It is a tool well suited for modeling complex systems developing over time thanks to the agent system that permit the explore the connection between the micro-level behavior and the macro-level patterns that emerge from their interaction.

It is used by tens of thousands of students, teachers and researchers worldwide. It also powers HubNet participatory simulations. It is authored by Uri Wilensky and developed at the CCL. You can download it free of charge. It also has extensive documentation and tutorials [1].

1.4.2 FMU

Not distributed yet

FMI defines a tool-independent standard for exchanging models and running standalone simulation tools [2]. A simulation model exported according to the FMI standard is called a Functional Mockup Unit (FMU). The FMU is like a black box with inputs and outputs port, and explicit interface to interact. It is an archive file that contains the source or object code needed to execute the model (*.bin), and an XML file describing the capabilities of the model (functions, connecting port, parameters etc...). The FMI standard has two different ways of working, Model Exchange and Co-Simulation (Figure 1.1). The first one is a dynamic component representation using differential, algebraic and discrete equations. For Model Exchange, it is needed that the master algorithm provide the necessary solvers. In the other hand, Co-Simulation defines an interface for coupling independent simulation tools. That is to say that the FMU provides the associated solvers. In both cases, the master algorithm coordinates time and exchanges between FMUs [Blochwitz et al., 2011].

The FMI standard has two version (1.0 and 2.0), but these two versions are not supported by every modeling tools yet. For now, MECSYCO can use well Co-Simulation models for version 1.0 and need more development for the 2.0.

[1]`https://ccl.northwestern.edu/netlogo/index.shtml`
[2]`https://www.fmi-standard.org`

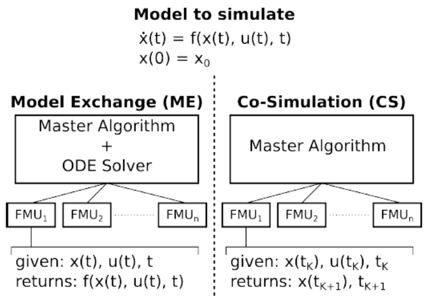

Model to simulate
$\dot{x}(t) = f(x(t), u(t), t)$
$x(0) = x_0$
Model Exchange (ME)
Master Algorithm + ODE Solver
FMU_1
FMU_2
FMU_n
given: x(t), u(t), t
returns: f(x(t), u(t), t)
Co-Simulation (CS)
Master Algorithm
FMU_1
FMU_2
FMU_n
given: $x(t_K)$, $u(t_K)$, t_K
returns: $x(t_{K+1})$, t_{K+1}

Figure 1.1: FMI for ModelExchange and for Co-Simulation

1.5 Implementation choices

1.5.1 Middleware for distributed simulation

To perform a distributed multi-simulation, MECSYCO currently uses the **Data Distribution Service (DDS)** [OMG, 2007] communication middleware. DDS is an OMG standard for real-time communication and scalable systems. Currently, the DDS implementation used in MECSYCO is **Prismtech Opensplice Community Edition 6.3**[3].

At the deployment level, the constraints of DDS are that communications are done using the multicast protocol. This implies that:

- the IP addresses of the connected devices have to be set in the same network (LAN).
- the firewall(s) used inside the network should NOT block multicast messages.

For the communication through DDS, messages are serialized using the open standard format JSON [ECMA, 2013].

DDS is currently the only available choice, but MECSYCO is not dependent on it and can switch from a communication middleware to another easily.

[3]`http://www.prismtech.com/opensplice`

1.5.2 Programming Languages

MECSYCO is currently implemented in Java and C++ programming languages. In the following sections, we detail each of these implementations. Note that we care to develop the C++ sources as close as possible of the Java sources, for easier maintenance.

Java implementation of MECSYCO

The Java code of MECSYCO supports Java 1.7 version or higher either with the OpenJDK or Oracle Java Virtual Machine. The MECSYCO's features have been tested with this implementation in various OS configurations (see table 1.2). Using Eclipse[4] for developing is encouraged.

OS Configurations		**Parallel simulation**	**Distributed simulation (OpenSplice)**	**HLA Connection**	**FMI Connection**
Windows 8.1	x64	✓	?	✓	✓
Windows 7	x64	✓	✓	?	✓
	x86	✓	✓	?	✓
Ubuntu 12.04	x64	✓	✓	?	✓
	x86	✓	?	?	?
GNU/Linux Debian Jessie	x64	✓	✓	?	✓
Mac OS X 10.9.4	x64	✓	Not supported	?	N/A
Mac OS X 10.7.5	x64	✓	Not supported	?	N/A

Table 1.1: Compatibility of the MECSYCO Java features with different OS configurations. Green check boxes indicate that the MECSYCO Java feature supports the configuration, whereas orange question marks indicate that the configuration has not been tested for a feature yet.

C++ implementation of MECSYCO (not distributed yet)

The C++ sources are compiled with GCC[5] for GNU/Linux, and Microsoft Visual Studio 2013[6] for Windows. In the both case, we use a CMakeFiles system, detecting the current OS and compiling all subprojects through auto generated Makefiles. Using QtCreator[7] for developing is encouraged. Both versions use Boost[8] libraries, with *shared pointers*.

OS Configurations		**Parallel simulation**	**Distributed simulation (OpenSplice)**	**FMI Connection**
Windows 8.1	x64	✓	✓	?
GNU/Linux Debian Jessie	x64	✓	✓	?
Ubuntu 14.10	x64	✓	✓	?

Table 1.2: Compatibility of the MECSYCO C++ features with different OS configurations. Green check boxes indicate that the MECSYCO C++ feature supports the configuration, whereas orange question marks indicate that the configuration has not been tested for a feature yet.

[4]`http://eclipse.org`
[5]`https://gcc.gnu.org`
[6]`http://msdn.microsoft.com/en-us/library/dd831853.aspx`
[7]`http://qt-project.org/wiki/Category:Tools::QtCreator`
[8]`http://www.boost.org`

Chapter 2

MECSYCO's User Guide

In this section we provide you an installation guide and a global view of the structure and componentd of MECSYCO. This User Guide is an introduction for the advance guides provided. You will find some information about each component of MECSYCO, and a tutorial for a basic launcher of a multi-model (section 2.4.5)

Level	1			
For	**Tester:** Learning the base, in other words, learning how to construct the easiest multi-model (using only pre-made object), and how to visualize and distribute them.			
Documentation needed	Getting Started 1 and 2	User Guide: MECSYCO-visu	User Guide: MECSYCO-com-dds	User Guide: Scholar applications
Download	MECSYCO-re, MECSYCO-visu, MECSYCO-com-dds, MECSYCO-world-netlogo, Getting started eclipse/maven proect, Scholar applications. And do not forget all the dependencies.			
Note	If you need help to install libraries (MECSYCO or dependencies), you can find a guide section 2.2. You can find templates for a simple multi-model launcher at the end too.			

Table 2.1: User guide list for begginer.

Level	2		
For	**Advanced user:** Base acquired. Learning how to improve your launcher and your multi-model.		
Documentation needed	All level 1 guides	Check section *Simulation data* (2.4.1) and operation (2.4.3 and 2.4.4)	User Guide: Create your own operations
Download	Same as level 1. Add the source for MECSYCO-world-netlogo (see section **??**).		
Note	The source can help you to understand what kind of data are used by NetLogo, and then what kind of operation you will need.		

Table 2.2: User guide list for advances.

Level	**3**				
For	**Developper user:** Base acquired. Learning how to create your own multi-model from the scratch (model, ModelArtifact, SimulData).				
Documentation needed	All Level 1 and 2 guides	Check Appendix B	User Guide: SimulData manipulation	User Guide: Multiplexer	User Guide: Model Artifact
Download	Same as previous levels. Add all sources.				
Note	In order to fully use MECSYCO, you need to understand MECSYCO but also DEVS and the simulators you want to work with (new or current one). We just provide help for the implementation part.				

Table 2.3: User guide list for a full use of MECSYCO.

2.1 MECSYCO's overview

For a multi-simulation, we consider a set of **models** with **input** and **output ports**. During a multi-simulation the simulators of models exchange **simulation events** consisting of time stamped **simulation data**.

The MECSYCO meta-model relies on the Agents and Artifacts paradigm [Ricci et al., 2007]. In this perspective, a multi-agent system is described by two kinds of concepts:

- The **agents** are the autonomous part of the system. In MECSYCO, each simulator of the multi-simulation is associated with an agent named **m-agent**.
- The **artifacts** are reactive parts of the environment that can be used by the agent in order to achieve their goals. The artifacts propose services to the m-agents allowing them to communicate with each other, to coordinate their works or to modify their environment. At the implementation level, these services correspond to methods. In MECSYCO, two kinds of artifacts exist: **interface artifacts** are used by the m-agents to collect and put simulation events in their models, and **coupling artifacts** are used by the m-agent to exchange simulation events. The coupling artifacts use **transformation operations** to transform simulation data between simulators.

In the next sections, we detail how these concepts can be manipulated by the user in order to described a multi-model with MECSYCO. For each concept, we detail (if relevant):

- What is its purpose in the MECSYCO meta-model
- How to create it
- What are the methods the user needs to manipulate in order to connect the concept with other ones
- What are the methods the user needs to manipulate in order to operationalize the multi-simulation
- What are the methods the user needs to manipulate in order to debug the multi-simulation

For each method and constructor we detail the corresponding implementation in Java (C++ programming language is not distributed yet).

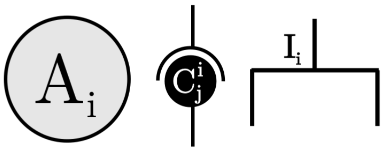


Figure 2.1: Symbols of the MECSYCO components (a) m-agent, (b) coupling artifact, (c) interface artifact.

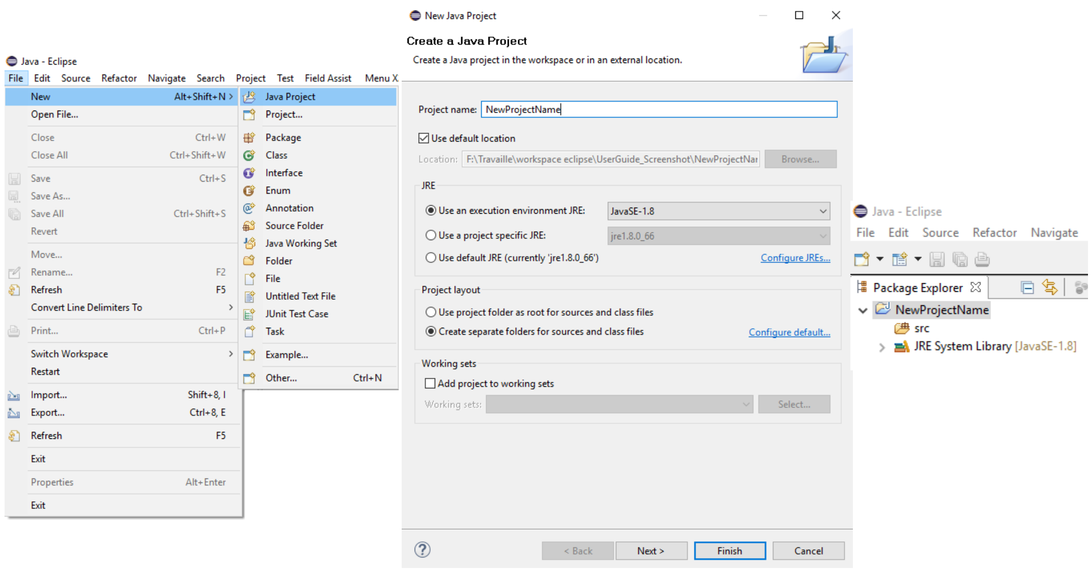
2.2 MECSYCO's installation guide

2.2.1 Java implementation

Remember that the Java code of MECSYCO supports Java 1.7 version or higher with OpenJDK or Oracle Java Virtual Machine. As said, we used Eclipse for developing it. In this section we show how to install MECSYCO's core in your projects (in the case of Eclipse Luna, but it will be the same for all version of Eclipse).

First, you need to download it at MECSYCO's website, "Download" section[1] (MECSYCO-re 2.0.0). If you had already used the old version of MECSYCO, you can still launch your projects thanks to the compatible version of the core (MECSYCO-re-compatible). But we still advice you to update your previous project.

MECSYCO is used as a library for your Eclipse project, so after creating a new project (Figure 2.2), you will be able to import libraries thanks to the properties of the project (Figures 2.3 to 2.5).



Create a Java project in the workspace or in an external location.

Project name: NewProjectName

☑ Use default location

Location: F:\Travaille\workspace eclipse\UserGuide_Screenshot\NewProjectNa

JRE

● Use an execution environment JRE: JavaSE-1.8

○ Use a project specific JRE: jre1.8.0_66

○ Use default JRE (currently 'jre1.8.0_66')

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files

Working sets

☐ Add project to working sets

Working sets:

NewProjectName
src
JRE System Library [JavaSE-1.8]

Figure 2.2: Creation of a new project in Eclipse.

[1] http://mecsyco-website.gforge.inria.fr/dev/download/

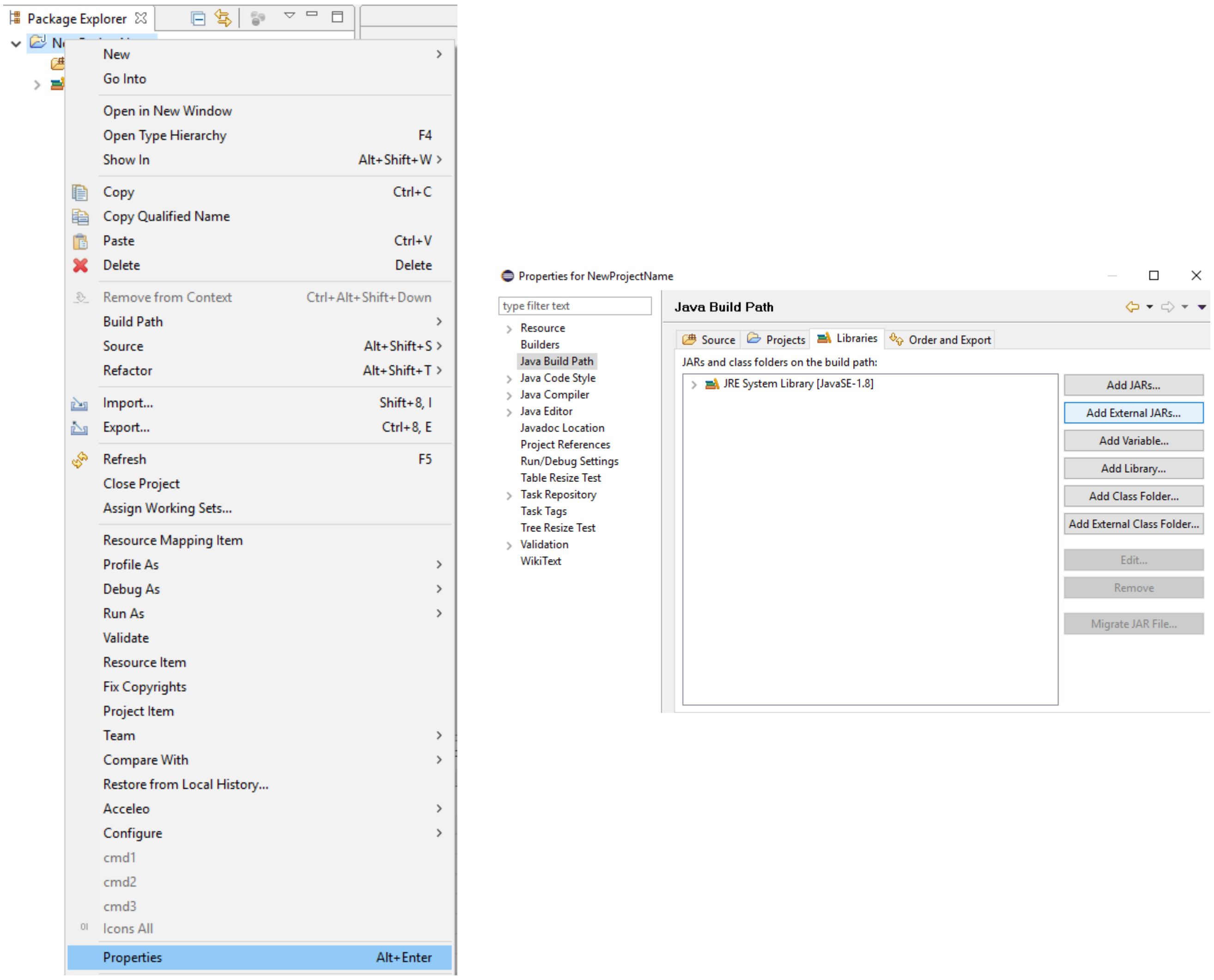


Figure 2.3: Project's properties.

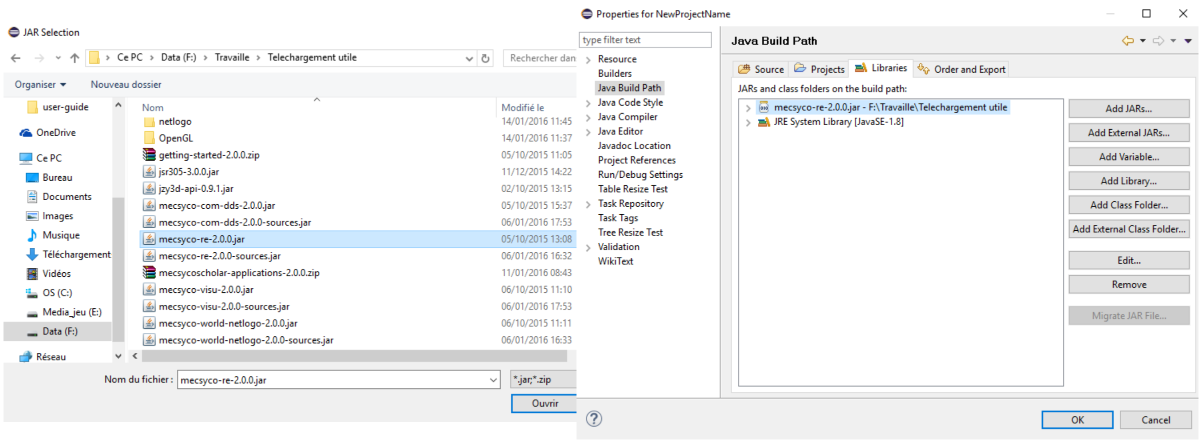

JAR Selection
Ce PC › Data (F:) › Travaille › Telechargement utile
Rechercher dan
Organiser
Nouveau dossier
user-guide
OneDrive
Ce PC
Bureau
Documents
Images
Musique
Téléchargement
Vidéos
OS (C:)
Media_jeu (E:)
Data (F:)
Réseau
Nom
Modifié le
netlogo 14/01/2016 11:45
OpenGL 14/01/2016 11:37
getting-started-2.0.0.zip 05/10/2015 11:05
jsr305-3.0.0.jar 11/12/2015 14:22
jzy3d-api-0.9.1.jar 02/10/2015 13:15
mecsyco-com-dds-2.0.0.jar 05/10/2015 15:37
mecsyco-com-dds-2.0.0-sources.jar 06/01/2016 17:53
mecsyco-re-2.0.0.jar 05/10/2015 13:08
mecsyco-re-2.0.0-sources.jar 06/01/2016 16:32
mecsycoscholar-applications-2.0.0.zip 11/01/2016 08:43
mecsyco-visu-2.0.0.jar 06/10/2015 11:10
mecsyco-visu-2.0.0-sources.jar 06/01/2016 17:53
mecsyco-world-netlogo-2.0.0.jar 06/10/2015 11:11
mecsyco-world-netlogo-2.0.0-sources.jar 06/01/2016 16:33
Nom du fichier : mecsyco-re-2.0.0.jar
.jar;.zip
Ouvrir
Properties for NewProjectName
type filter text
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project References
Run/Debug Settings
Table Resize Test
Task Repository
Task Tags
Tree Resize Test
Validation
WikiText
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
mecsyco-re-2.0.0.jar - F:\Travaille\Telechargement utile
JRE System Library [JavaSE-1.8]
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
OK
Cancel

Figure 2.4: Libraries importation

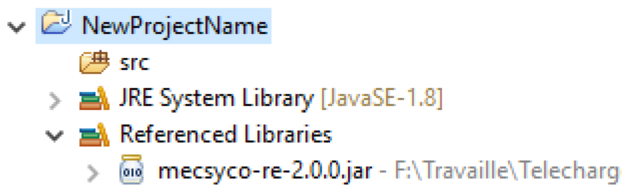

NewProjectName
src
JRE System Library [JavaSE-1.8]
Referenced Libraries
mecsyco-re-2.0.0.jar - F:\Travaille\Telecharg

Figure 2.5: Final result

Note that if you put the jar file in the files of the project before importing it, you can use *"Add JARS"* instead of *"Add External JARS"*

2.2.2 C++ implementation

For now, the C++ implementation is not distributed yet.

2.2.3 Other libraries

You can notice that in the "Download" section, there is not only *MECSYCO-re-2.0.0.jar* (or the compatible version) but there is more. Each librarie is installed in the same way as *MECSYCO-re-2.0.0.jar* and offer new functions:

- MECSYCO-visu is used for observing and logging the results
- MECSYCO-com-dds enables the distribution of the co-simulation on several machines
- MECSYCO-world-netlogo contains the set of special type of variables needed for NetLogo
- Getting started is the set of pre-made models and artifact needed for the "Getting Started" section of the website
- MECSYCO application is a set of pre-made models that allows to test and show examples of the features of MECSYCO. It also contains the full version of the models used in "Getting Started".

For more information about those different libraries see chapter *Extensions of MECSYCO* (4). You can also find an "User Guide" associated to each librairies in the" Documentation" section.

2.2.4 Dependencies

Each one of the libraries have external dependencies. It means that they need other libraries to work. Some are other libraries from MECSYCO, and some need to be downloaded elsewhere. Those dependencies and the different links are given in the download section of the website. In order to add them easely in your project, put all the jar in a libs folder in the project. In some case this manipulation is mandatory.

2.3 MECSYCO's concepts

2.3.1 The m-agent

An **m-agent** $\mathcal{A}_i$, manages a model m_i and is in charge of interactions of this model with the other ones (symbol in Figure 2.1a). Each agent is autonomous and its behavior corresponds to the parallel abstract simulator of the model (see Appendix C). At the implementation level, each m-agent matches a thread.

Constructor of the m-agent

An m-agent corresponds to the class **EventMAgent**. Two parameters are used to create an m-agent:

- **aAgentName:** the name of the agent. It is used in the console for information or debugging.
- **aMinPropagationDelay:** the minimum time delay an input needs to propagate through the system to the output (see the Appendix C for more details). This parameter is optional, by default it is equals to $+\infty$ (no modification of the stack of the events). If inaccurate, this parameter may cause a **CausalityException** during the multi-simulation.
- **aMaxSimTime:** the maximum time of the multi-simulation.

EventMAgent constructor in Java implementation
public EventMAgent (String aAgentName, double aMinPropagationDelay, double aMaxSimTime)
public EventMAgent (double aMinPropagationDelay, double aMaxSimTime))
public EventMAgent (String aAgentName, double aMaxSimTime)
public EventMAgent (double aMaxSimTime)

EventMAgent constructor in C++ implementation
Not distributed yet

Methods for connecting the m-agent with the other concepts

An m-agent can be connected to several output and input coupling artifacts (see section 2.3.2) and one interface artifact (see section 2.3.3) using the following methods:

$addInputCouplingArtifact$
Parameters:

- **coupling** - the input coupling artifact (see section 2.3.2) to be connected with the m-agent
- **port** - the name of the input port of the model where the m-agent will put the events coming from the artifact.

addInputCouplingArtifact method in Java implementation
public void addInputCouplingArtifact(EventCouplingArtifactReader coupling, String port)

addInputCouplingArtifact method in C++ implementation
Not distributed yet

Connects an input coupling artifact to the m-agent.

$addOutputCouplingArtifact$
Parameters:

- **coupling** - the output coupling artifact (see section 2.3.2) to be connected with the m-agent
- **port** - the name of the model output port where the outgoing events will be send to the artifact.

addOutputCouplingArtifact method in Java implementation
public void addOutputCouplingArtifact(EventCouplingArtifactWriter coupling, String port)

addOutputCouplingArtifact method in C++ implementation
Not distributed yet

Connect an output coupling artifact 2.3.2 to the m-agent.

setModelArtifact
Parameters:

- **artifact** - the interface artifact (see section 2.3.3) to be connected with the m-agent

setModelArtifact method in Java implementation
public void setModelArtifact (GenericModelArtifact artifact)

setModelArtifact method in C++ implementation
Not distributed yet

Set the model artifact of the m-agent.

M-agent's methods for simulation

The m-agent is the class used to launch a multi-simulation. For a given m-agent, the following methods should be called in that order (if needed):

startModelSoftware

startModelSoftware method in Java implementation
public void startModelSoftware()

startModelSoftware method in C++ implementation
Not distributed yet

Initialize the m-agent's model by calling the *init* methods of the m-agent's interface artifact (see section 2.3.3).

setModelParameters
Parameters:

- **args** - the set of the initial parameters

setModelParameters method in Java implementation
public void setModelParameters (@Nullable String... args)

setModelParameters method in C++ implementation
Not distributed yet

Set the initial parameters of the model by calling the *setIntialParameters* method of the interface artifact (see section 2.3.3).

coInitialize

coInitialize method in Java implementation
public void coInitialize()

coInitialize method in C++ implementation
Not distributed yet

Send initialization data to the others m-agents (see Appendix C for more details on the initialization process).

run
Do not use this method directly, use the "start" method of Thread object

run method in Java implementation
public void run()

run method in C++ implementation
Not distributed yet

Start the simulation of the m-agent's model. Thanks to *start*(), the agent will be given a thread.

2.3.2 The coupling artifact

Each interaction between any m-agents $\mathcal{A}_i$ and $\mathcal{A}_j$ is reified by a **coupling artifact** $\mathcal{C}_j^i$ (symbol in Figure 2.1b). A coupling artifact $\mathcal{C}_j^i$ has a direction: it is the support of the exchanges of events from $\mathcal{A}_i$ to $\mathcal{A}_j$. It works like a mailbox: the artifact has a buffer of events where the m-agents can post their external output events and get their external input events. Therefore, a coupling artifact $\mathcal{C}_j^i$ has two roles: for $\mathcal{A}_i$, it is an output coupling artifact, whereas for $\mathcal{A}_j$ it is an input coupling artifact.

Constructor of the coupling artifacts

A coupling artifact for sending external events corresponds to the class **EventCouplingArtifactWriter**. For receiving external events, you should use the class **EventCouplingArtifactReader**. These classes are abstract and so cannot be used directly. Depending on the communication middleware you choose for the decentralization, you have to use the corresponding instantiation.

If you want to use DDS as communication middleware, a coupling artifact for sending external events corresponds to the class **DDSEventCouplingArtifactSender**:

DDSEventCouplingArtifactSender constructor in Java implementation
public DDSEventCouplingArtifactSender(String topic)

DDSEventCouplingArtifactSender constructor in C++ implementation
Not distributed yet

With DDS, a coupling artifact for receiving external events corresponds to the class **DDSEventCouplingArtifactReceiver**:

DDSEventCouplingArtifactReceiver constructor in Java implementation
public DDSEventCouplingArtifactReceiver (String aTopic, Class<? extends SimulData> aType) public DDSEventCouplingArtifactReceiver (String aTopic, Type aDataType)

DDSEventCouplingArtifactReceiver constructor in C++ implementation
Not distributed yet

The "topic" arguments is the DDS topic to use. No matter what you choose, but the topic must be the same between a couple Sender and Receiver. With the Receiver, you have to add the type of the expected value from the coupling (derived from **SimulData**, see section 2.4.1, in the first constructor).

The special coupling artifact **CentralizedEventCouplingArtifact** is also available, for centralized simulation (with agents in the same launcher). This class is both a Sender and a Receiver and does not need any argument or at least a name for information (debugging) purpose:

CentralizedEventCouplingArtifact constructor in Java implementation
public CentralizedEventCouplingArtifact (String aCouplingId) public CentralizedEventCouplingArtifact()

CentralizedEventCouplingArtifact constructor in C++ implementation
Not distributed yet

Example templates of run configurations using centralized coupling artifacts are available in the section 2.4.5. For decentralize example, see "*User Guide: MECSYCO-com-dds*".

Connection of the coupling artifact with other concepts

In order to transform data between simulators or put to scale the time, operations (see section 2.4.3) can be added to the coupling artifact. These operations are sequentially called by the coupling artifact in order to transform data or time. The operations are on the Receiver side, that is to say classes derived from EventCouplingArtifactReader, including **DDSEventCouplingArtifactReceiver** and **CentralizedEventCouplingArtifact**.

addEventOperation
Parameters:

- **aOperation** - the operation to be added

addEventOperation method in Java implementation
public void addEventOperation (EventOperation aOperation)

addEventOperation method in C++ implementation
Not distributed yet

Add an operation for data to the coupling artifact.

addTimeOperation
Parameters:

- **aOperation** - the operation to be added

addTimeOperation method in Java implementation
public void addTimeOperation (TimeOperation aOperation)

addTimeOperation method in C++ implementation
Not distributed yet

Add an operation for time to the coupling artifact.

2.3.3 The interface artifact

The **interface artifact** $\mathcal{I}_i$ reifies interactions between an m-agent $\mathcal{A}_i$ and its model m_i (symbol in Figure 2.1c). An interface artifact contains primitives to manipulate a simulator. As a consequence, each interface artifact is specific: to a simulator, if the interface is relatively for all models (FMI); to a model otherwise (NetLogo), and must be defined by the user.

Constructor of the interface artifact

Defining an interface artifact consists of creating and instantiating a class implementing the interface **GenericModelArtifact**. The designer is free to add attributes and other methods to its artifact.

Methods for the simulation

For the m-agent to be able to manipulate a specific simulator, the following methods must be implemented by the users. These methods are to be used by the m-agent and should NOT be called directly by the user.

initialize

initialize method in Java implementation
public void initialize()

initialize method in C++ implementation
Not distributed yet

Used by the m-agent to initialize the model, it launches the simulation software and sets the initial state of the model.

setIntialParameters
Parameters:

- **args -** the set of the initial parameters

setIntialParameters method in Java implementation
public void setInitialParameters (@Nullable String[] args)

setIntialParameters method in C++ implementation
Not distributed yet

Used by the m-agent to set the model's initial parameters.

processInternalEvent
Parameters:

- **time -** the simulation time of the internal event to be processed. This parameters may be useful for the designer only with pure DEVS simulator. When designing the method, the value of the parameter can be ignored and, in that case, the function will process the next internal event of the model

processInternalEvent method in Java implementation
public void processInternalEvent(double time)

processInternalEvent method in C++ implementation
Not distributed yet

Used by the m-agent to process the (next) internal event situated at a given simulation time.

processExternalInputEvent
Parameters:

- **event -** the incoming simulation event (see section 2.4.2)
- **port -** the input port of the model where the simulation event has to be sent

processExternalInputEvent method in Java implementation
public void processExternalInputEvent(SimulEvent event, String port)

processExternalInputEvent method in C++ implementation
Not distributed yet

Used by the m-agent to process the incoming external event in the input port of the model.

getExternalOutputEvent
Parameters:

- **port -** the output port of the model where the simulation event has to be collected

Returns: External outputs of the model, at the current time, attached to the given port. The user has to create the simulation event himself. **the timestamp of the returned simulation event must be equals to the current simulation time of the model**.

getExternalOutputEvent method in Java implementation
public @Nullable SimulEvent getExternalOutputEvent (String port)

getExternalOutputEvent method in C++ implementation
Not distributed yet

Used by the m-agent to create the events to send by the given output port of the model.

getNextInternalEventTime
Returns: the simulation time of the model's earliest scheduled internal event

getNextInternalEventTime method in Java implementation
public double getNextInternalEventTime()

getNextInternalEventTime method in C++ implementation
Not distributed yet

Used by the m-agent to get the simulation time of the model's earliest scheduled internal event.

getLastEventTime
Returns: the simulation time of the model's last processed internal event

getLastEventTime method in Java implementation
public double getLastEventTime()

getLastEventTime method in C++ implementation
Not distributed yet

Used by the m-agent to get the simulation time of the model's last processed internal or external event (the current simulation time of the model).

finishSimulation

finishSimulation method in Java implementation
public void finishSimulation()

finishSimulation method in C++ implementation
Not distributed yet

Used by the m-agent to terminate the simulation (for instance, closing the simulation software).

2.4 Manipulation of simulation data

2.4.1 Simulation data

A simulation data corresponds to data exchanged between simulators during a multi-simulation. In order to be exchanged a simulation data is contained in a simulation event (see section 2.4.2). Each simulation data is specific to one or several simulator port (input or output). As a consequence, the designer should create its own simulation data.

Constructor for a simulation data

Defining a simulation data consists of creating a class extending the abstract class **SimulData**. The designer of the simulation data is free to add any attributes and constructors. **However, due to serialization reason (i.e. translating the simulation data into JSON format and rebuilding it from a JSON description for machine-to-machine communication purpose), empty constructor is mandatory, and complex type attributes (for instance list of objects) must contain only simulation data objects.**

Existing simulation data:

In order to make MECSYCO as generic as possible, we implemented some type of simulation data that can already cover a lot of simulation. For that, we based those types on **Tuple** [2]. **Tuple** are like vector with a predefined length. Its particularity is that each element can be of a completely different type and nature, and it would still work perfectly. In MECSYCO, we implemented 5 levels of **Tuple**:

[2]urlhttp://www.javatuples.org/

Tuple1 constructor in Java implementation
public Tuple1 (@JsonProperty("item1") T1 aItem1)

Tuple2 constructor in Java implementation
public Tuple2 (@JsonProperty("item1") T1 aItem1, @JsonProperty("item2") T2 aItem2)

Tuple3 constructor in Java implementation
public Tuple3 (@JsonProperty("item1") T1 aItem1, @JsonProperty("item2") T2 aItem2, @JsonProperty("item3") T3 aItem3)

Tuple4 constructor in Java implementation
public Tuple4 (@JsonProperty("item1") T1 aItem1, @JsonProperty("item2") T2 aItem2, @JsonProperty("item3") T3 aItem3, @JsonProperty("item4") T4 aItem4)

Tuple5 constructor in Java implementation
public Tuple5 (@JsonProperty("item1") T1 aItem1, @JsonProperty("item2") T2 aItem2, @JsonProperty("item3") T3 aItem3, @JsonProperty("item4") T4 aItem4, @JsonProperty("item4") T5 aItem5)

To use them, you have to define the type of element (T1, T2, T3, T4, T5). Then in the **Tuple5** case, we have: *new Tuple5 <type of T1, type of T2, type of T3, type of T4, type of T5>(aItem1, aItem2, aItem3, aItem4, aItem5)*

- **type:** This can be any type of Java (Double, String...) or data you created (Tuple2 for example). You can also use "*?*", it corresponds to "all types".
- **aItem:** This is the corresponding data. For example, if T1 is *"Integer"*, aItem1 will be any natural number.

In order to manage large number of data at once, you can use **SimulVector** and **SimulMap**:

ArrayedSimulVector
public ArrayedSimulVector (@JsonProperty("items") G... aItems)

HashedSimulMap
public HashedSimulMap (Tuple2<K, V>... aPairs)

As previously, you have to define the type (G, K and V), then you will be able to give the corresponding items.

- **SimulVector:** All element will have then the same type. For example:
  SimulVector<SimulData> test=new ArrayedSimulVector(aItem1, aItem2, ...)
  is a vector where all elements will be a simulation data type (Tuple1, or even another SimulVector)
- **SimulMap:** A map is a function with a finite set of inputs and of outputs. An input is a key and an output is a value. The map is defined by a list of pairs (key, value):
  HashedSimulMap test=new HashedSimulMap(new Tuple2<String, SimulData>(Item1, Item2), new Tuple 2<Integer,Integer>(Item1_2, Item2_2), ...)
  This is a map where the firs element will have a String for "key", and a SimulData for data; the second element will have for both natural number, and you can keep adding different kind of entries like that.

For more information, see "*User Guide: SimulData manipulation*".

Methods to manipulate a simulation data

The designer is free to add any methods in order to manipulate a simulation data. **However, due to serialization reason, getter and setter are mandatory for each attribute of a simulation data. (i.e. translating the simulation data into JSON format and rebuilding it from a JSON description for machine-to-machine communication purpose)**.

MAJ: Since Jackson 1.7, setter are not mandatory anymore.

2.4.2 Simulation event

A simulation event is an external event occurring at a given point in the simulated time during a multi-simulation. The simulation events constitute the messages exchanged between the simulators. They are composed of simulation data (see section 2.4.1) and a timestamp.

Creation of a simulation event

Simulation events are created in the *getExternalOutputEvent* methods of the interface artifacts (see section 2.3.3). A simulation event corresponds to the class **SimulEvent**. Two parameters are used to create an SimulEvent:

- **aData:** the simulation data (see section 2.4.1) contains by the simulation event
- **aTime:** the timestamp of the simulation event. **When created in the *getExternalOutputEvent* of the interface artifact, it must be equals to the model's current simulation time.**

SimulEvent constructor in Java implementation
public SimulEvent (SimulData aData,double aTime)

SimulEvent constructor in C++ implementation
Not distributed yet

Methods for the manipulation of a simulation event

A simulation event is manipulated by an interface artifact in its *getExternalOutputEvent* and *processExternalInputEvent* methods (see section 2.3.3). The simulated event can be manipulated using the following methods:

withData
Parameters:

- **aData** - the simulation data (see section 2.4.1) to be contained by the simulation event.

withData method in Java implementation
public SimulEvent withData (SimulData aData)

withData method in C++ implementation
Not distributed yet

Get a new event with the same timestamps and a new data (see section 2.4.1).

getData
Returns: the simulation data (see section 2.4.1) contained by the simulation event.

getData method in Java implementation
public SimulData getData()

getData method in C++ implementation
Not distributed yet

Used to get the simulation data (see section 2.4.1) contained by the simulation event.

withTime
Parameters:

- **aTime** - the timestamp of the simulation event.

withTime method in Java implementation
public SimulEvent withTime (double aTime)

withTime method in C++ implementation
Not distributed yet

Used to get a new event with the same data and another timestamps.

getTime
Returns: the timestamp of the simulation event.

getTime method in Java implementation
public double getTime()

getTime method in C++ implementation
Not distributed yet

Used to get the timestamp of the simulation event.

2.4.3 Operation for data transformation

The operations for data transformation are used by the coupling artifact. An operation is designed specifically to transform simulation data (see section 2.4.1) sent by the output port of a simulator, into simulation data needed by the input port of a simulator. As a consequence, there is no pre-existing operation and the user should define its own operation depending on its simulators and models.

Creation of an operation

Defining an operation consists in creating and instantiating a class extending the abstract class **EventOperation**. The designer is free to add attributes and other methods to its operation in order to parametrize the operation.

Method of the operation

In order to transform data, every operation must define the following method of the EventOperation class:

apply
Parameters:

- **aEvent** - the event containing the data to process (see section 2.4.1).

Returns: the processed event (see section 2.4.1).

apply method in Java implementation
public @Nullable SimulData apply (SimulEvent aEvent)

apply method in C++ implementation
Not distributed yet

Transform simulation data.

2.4.4 Operation for time transformation

The operations for time have the same pattern than operation for data transforming (section 2.4.3). This time, operations are extending the abstract class **TimeOperation**, and the method *apply* has a different parameter:

apply
Parameters:

- **aTime** - the time to put to scale.

Returns: The converted time.

apply method in Java implementation
public double apply (double aTime)

apply method in C++ implementation
Not distributed yet

Scale a simulation time.

2.4.5 Launcher

This section is a tutorial for building easily a run configuration (or launcher) for your multi-model without any optional function (operation, DDS, observing etc...). In order to have advanced example, see the corresponding *User Guide* for the functions you want to add.

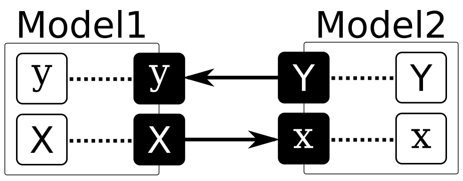


Figure 2.6: Simple example of coupling for the templates (black boxes are ports and white boxes are state variables).

The example will be two models interacting with each other. Each model has two input ports, and two output ports (Figure 2.6).

Java version

First, create a new Java class containing a *main*. Before working in the *main*, you can define some variables in order to make your launcher easily configurable, for example the stop time of your simulation:
public final static double maxSimulationTime = 10;

Now, in the *main*, we will construct the launcher, and at the same time the multi-model. First, instantiate an agent (section 2.3.1) per model, and link them to the model thanks to the interface artifact (section 2.3.3):
EventMAgent agent1 = new EventMAgent("Name1",maxSimulationTime);
MyModel1Artefact ModelArtefact1 = new MyModel1Artefact();
agent1.setModelArtefact(ModelArtefact1);
The interface artifact of each agent can be the same or different, it should match the model you want to use.

You can now start to create the interaction. In order to do that you need to define the links from the scratch. You need then to create the "wires", that is to say the *CouplingArtifact* (section 2.3.2), and precise where they are plugged-in (which input and output):
CentralizedEventCouplingArtifact couplingFrom1To2 = new CentralizedEventCouplingArtifact();
agent2.addInputCouplingArtifact(couplingFrom1To2, "x");
agent1.addOutputCouplingArtifact(couplingFrom1To2, "X");
Try, to be explicit when you name your variables.

The multi-model is built, we now need the launcher part. First step is the initialization. For

that, you need to start the softwares associated to each model (not mandatory in some case), and if your models need initialization parameters,set them:
agent1.startModelSoftware();
String [] args_model1 = { "value1","value2", ... };
agent1.setModelParameters(args_model1);

You can now start the simulation. Do it in a *try, catch* section in case of errors:
try {
agent1.coInitialize();
agent1.start();
}
catch (CausalityException e){
e.printStackTrace();
}

C++ version

Not distributed yet

Run configuration example templates

This section proposes commented example templates, for the model. They are available in this documentation and viewable in the Appendix section.

Commented template of Java run configurations is available:

- appendix A: run configuration for a centralized execution with two agents (`Templates/java/centralized/Launcher.java`) ;

Commented template of C++ run configurations are not available yet.

Chapter 3

MS4SG's concepts

3.1 Model Artifacts

3.1.1 The HLA Model Artifact

3.1.2 The FMI Model Artifact

3.2 Data transformation operations

Chapter 4

Extensions of MECSYCO

4.1 Deployment of a multi-simulation (MECSYCO-com-dds)

The communication package gathers the implementations of means to connect several machines or platforms from the network. As a consequence, MECSYCO enables distributed and decentralized simulations in Java, C++, or hybrid code. In order to do that, this package adapt MECSYCO to the use of OpenSlice DDS[1].

MECSYCO-com-dds is used instead of the usual *CouplingArtifact* but *DDSEventCouplingArtifactSender* and *DDSEventCouplingArtifactReceiver* instead (section 2.3.2).

All primitives and classes needed for communication are in *MECSYCO-com-dds 2.0.0.*

A detailed guide is provided (*User Guide: MECSYCO-com-dds*) and two templates in order to help building DDS based model.

4.2 Observation of a multi-simulation (MECSYCO-visu)

MECSYCO-visu is a set of agents and artifacts dedicated to the display of the results from simulations on different type of graph. The observer agent has the same behavior than any other MECSYCO's agent, which makes its use and parametrization easier.

MECSYCO-visu is built upon observing agents (agents that manage a visualization tool), an observing dispatcher (artifact that will connect the agent to the observing artifacts) and observing artifacts to connect agents and visualization tools.

All primitives and classes needed for visualization are in *mecsyco-visu-2.0.0.jar*.

A detailed guide is provided (*User Guide: MECSYCO-visu*) and a template to help you include observing tools in your launcher.

[1]`http://www.prismtech.com/dds-community`

Chapter 5

Future upgrades

5.1 Reliable time representation

5.2 Prevention of busy waiting

Appendix A

Java Example Template: run configuration (centralized)

```java
import mecsyco.core.agent.EventMAgent;
import mecsyco.core.coupling.CentralizedEventCouplingArtifact;
import mecsyco.core.exception.CausalityException;


public class Launcher {
      public final static double maxSimulationTime = 10;

      public static void main(String args[]) {

        /**********************************/
        /**** AGENTS & MODEL ARTEFACTS ****/
        /**********************************/

        // First agent with first model (model1)
        EventMAgent agent1 = new EventMAgent("Name1",maxSimulationTime);
        MyModel1Artefact ModelArtefact1 = new MyModel1Artefact();
        agent1.setModelArtefact(ModelArtefact1);

        // Second agent with second model (model2)
        EventMAgent agent2 = new EventMAgent("Name2",maxSimulationTime);
        MyModel2Artefact ModelArtefact2 = new MyModel2Artefact();
        agent2.setModelArtefact(ModelArtefact2);


        /***************************/
        /**** COUPLING ARTEFACTS ****/
        /***************************/

        //        Model1                      Model2
        //   .--------------.            .--------------.
        //   | .---.      .---.        .---.      .---. |
        //   | | y |------| y |<-------| Y |------| Y | |
        //   | '---'      '---'        '---'      '---' |
        //   |              |            |              |
        //   | .---.      .---.        .---.      .---. |
        //   | | X |------| X |------->| x |------| x | |
        //   | '---'      '---'        '---'      '---' |
        //   '--------------'            '--------------'
        //
        // "y" and "X" are model1's state variables (typed Double)
        // "Y" and "x" are model2's state variables (typed Double)
        // We consider that the port names correspond to the state variable with which they are linked
        // During the simulation, X and Y are exchanged and models states are modified,
        // with model1.y = model2.Y and model2.x = model1.X

        CentralizedEventCouplingArtifact couplingFrom1To2 = new CentralizedEventCouplingArtifact();
        CentralizedEventCouplingArtifact couplingFrom2To1 = new CentralizedEventCouplingArtifact();

        // Agent1 will update "y" with the value received from couplingFrom2To1 (input events)
        // Agent2 will update "x" with the value received from couplingFrom1To2 (input events)
        agent1.addInputCouplingArtifact(couplingFrom2To1, "y");
        agent2.addInputCouplingArtifact(couplingFrom1To2, "x");

        // Agent1 will send "X" to couplingFrom1To2 (output events)
        // Agent2 will send "Y" to couplingFrom2To1 (output events)
        agent1.addOutputCouplingArtifact(couplingFrom1To2, "X");
        agent2.addOutputCouplingArtifact(couplingFrom2To1, "Y");


        /******************************/
        /**** MODELS INITIALIZATION ****/
        /******************************/

        // Start the simulation softwares associated to model1 and model2
        // This is not systematically necessary, it depends on the simulation software used
        agent1.startModelSoftware();
        agent2.startModelSoftware();

        // Initialize model1 and model2 parameters
        // e.g. time discretization or constants
        // This is not systematically necessary, it depends on the model
        String [] args_model1 = { "0.001" };
        String [] args_model2 = { "0.01" };
        agent1.setModelParameters(args_model1);
        agent2.setModelParameters(args_model2);

        /****************************************/
        /**** CO-SIMULATION INIT & STARTING ****/
        /****************************************/

        try {
          // Co-initialization with first exchanges
          // This is necessary only when the model initial states are co-dependent
          agent1.coInitialize();
          agent2.coInitialize();

          // Start the co-simulation
          agent1.start();
          agent2.start();

        // This should never happen
        } catch (CausalityException e) {
          e.printStackTrace();
        }
      }
    }
```

Appendix B

The DEVS Formalism

This appendix is taken from [Camus et al., TBP].

B.1 Introduction

The use of different formalisms may be required when modeling a complex system [Vangheluwe et al., 2002]. At the execution level, that means managing simulators with different scheduling policies: cyclic or variable time-steps, and event-based.

The Discrete EVent System (DEVS) formalism [Zeigler et al., 2000] is the most general formalism for discrete event model. An important feature of this formalism is that it can integrate all others formalisms [Vangheluwe, 2000] [Quesnel et al., 2009]. In other words, taking an execution point of view, it is sufficiently generic to describe all the different scheduling policies. That is why it can serve as an execution model for the evolution of a heterogeneous co-simulation.

In this Appendix, we detail the DEVS formalism. In section B.2, we give an overview of DEVS formalism. In section B.2.1 and B.2.2 we gives a formal description of the two kinds of models composing the DEVS formalism, respectively the DEVS coupled and atomic models. Finally we described the sequential simulation of a DEVS model B.3.

B.2 Definition of an event-based model

An event-based model considers that a system evolves in a continuous time base but changes its state at discrete points in time. These points are called event and can happen at any point in simulated time, contrary to cyclic model which change their states regularly every time-step (Figure B.1). Two types of events exist in event-based models: internal events and external events. The former correspond to events internally scheduled by the model, while the later correspond to:

- input events coming from other models or inputs generator, disturbing the model's internal functioning,
- and output events send by the model to other ones or to some outputs analyzer.

A model sends an external event each time it changes its state. When a model receives an external event, it must process it. This action modifies the state of the model and may schedule new internal event.

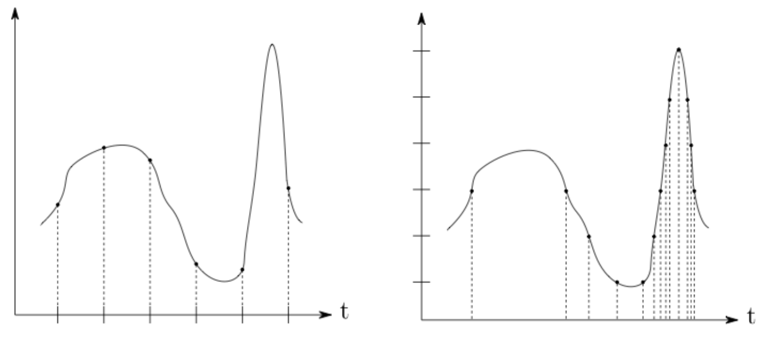


Figure B.1: (a) A cyclic simulation. Time is represented in a discrete manner and state changing happens regularly, every time-step. (b) An event-based simulation. The simulation time is continuous and state changing can happen irregularly anytime. (taken from [Siebert, 2011])

The DEVS formalism is the most general formalism for discrete event model. This formalism separates the description of a system in two kinds of model:

- The behavior of the system is described by DEVS atomic models. This is the formal equivalent of a model in the multi-model
- The organization of the system, that is to say how the atomic models are interconnected, is described by DEVS coupled models. This is the formal equivalent of a multi-model

In the next sections, we detail these two kinds of model.

B.2.1 Atomic DEVS Model

An atomic DEVS model m_i formally corresponds to the tuple $< X_i, Y_i, S_i, \delta_{int_i}, \delta_{ext_i}, \lambda_i, ta_i >$. S_i corresponds to the set of the model's states. Each state $s_i \in S_i$ is associated with a sojourn time given by the $ta_i(S_i \to \mathbb{R}^+_{0,\infty})$ function. When this sojourn time is over, the internal transition function $\delta_{int_i}(S_i \to S_i)$ is called in order to change the state of the model. Note that the sojourn time can be infinite. In this case, the model is said to be passive as it will stay in this state until the reception of an output event.

X_i and Y_i respectively correspond to the set of input and output ports where the model receives and sends its external events. The k^{th} input port of m_i is noted $x_i^k \in X_i(0 < k \leq card(X_i))$, and the n^{th} output port of m_i is noted $y_i^n \in Y_i(0 < n \leq card(Y_i))$.

Before any change of the model's state, the output function $\lambda_i(S_i \to Y_i)$ is called in order to generate the model output event(s) in the model output port(s). When the model receives an external event in one of its input port, the external transition function $\delta_{ext_i}(Q_i \times X_i \to S_i)$ is called in order to change the model's state. This function uses the total state $Q = (s, e)$ of the DEVS model. $e(0 \leq a \leq ta(s))$ corresponds to the elapsed time between the time of the last internal event and the timestamp of the external event.

The figure B.2 sums up the behavior of a DEVS atomic model. This behavior is executed by a DEVS abstract simulator (Figure B.3).

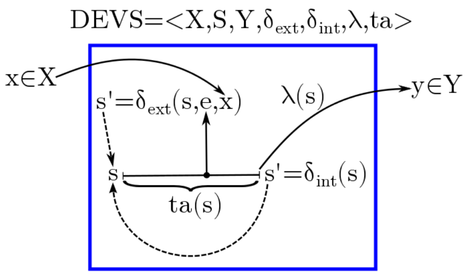


Figure B.2: Behavior of a DEVS atomic model (adapted from [Zeigler et al., 2000])

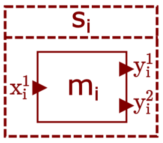


Figure B.3: An atomic DEVS model m_i and its abstract simulator s_i

B.2.2 Coupled DEVS Model

In this section we present the formalism of a DEVS coupled model. DEVS supports hierarchical modeling thanks to its proved closure under the coupling property: to each coupled model corresponds an atomic DEVS model. We present here a simplified, flattened version of the DEVS coupled model.

A DEVS coupled model is composed of a set M of interconnected models. An interconnection is defined as a link between the output port of a DEVS atomic model, and the input port of another DEVS atomic model. The set of the coupled model's links IC is composed of the couples $((i,n),(j,k))$, mapping the ports y_i^n and x_j^k. When an external event is send out of an output port, it must be immediately received in all the input ports sharing a link with this output port. Note that the DEVS formalism prevents an external input event from immediately generating an external output event. This may cause an immediate (in term of simulation time) infinite external events exchange between models, blocking the simulation execution. The figure B.4 shows an example of a DEVS coupled model. The table B.1 shows how this coupled model is formalized in DEVS.

An external event is composed of the tuple (t, e), where t corresponds to the timestamp of the event, that is to say the (simulated) time where the event is sent. e corresponds to the event message. It can be symbolic (i.e. describing the event in a symbolic way, like 'doors open') or numeric (i.e. describing the event in a quantitative way, like a set of positions).

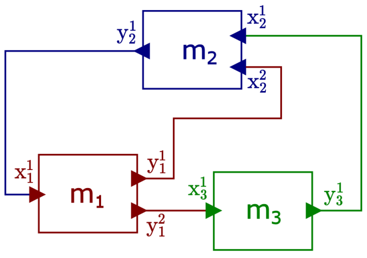


Figure B.4: A DEVS coupled model composed of three DEVS atomic models: m_1, m_2 and m_3

Descriptions	Notations
Atomic models	$M_d = \{m_1, m_2, m_3\}$
Links between the atomic models	$\mathsf{IC} = \{((1,1),(2,2)),((1,2),(3,1)),((2,1),(1,1)),((3,1),(2,1))\}$

Table B.1: Formalization of the coupled model of figure B.4 in DEVS

B.3 Sequential execution of a coupled model

In this section, we present how a DEVS coupled model is sequentially simulated with a monolithic execution. The simulation of a coupled model must respect the causality constraint, that is to say:

Definition: to respect the **causality constraint**, each atomic model must process its events (internals and externals) in an increasing temporal order [Zeigler et al., 2000,Fujimoto, 2001].

The sequential simulation of a DEVS coupled model is performed by a global coordinator (Figure B.5). It coordinates the DEVS simulators execution thanks to the maintainability of a global stack of the internal events scheduled for the simulation. These events are ordered according to their timestamps (Figure B.6).

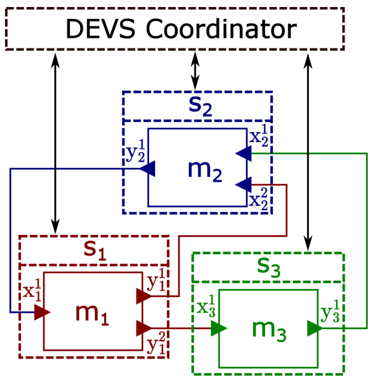


Figure B.5: Sequential execution of the DEVS coupled model of figure B.4. A global DEVS coordinator manages the execution of the DEVS simulators.

The coordinator behavior follows the cycle:

1. Selection of the most imminent internal event to be processed in the simulation.
2. Processing of this event by the corresponding atomic model.
3. Propagation of the resulting external(s) event(s) in the coupled model.
4. Processing of the external events by the receiver(s) atomic model(s). This step potentially lead to the scheduling of new internal event.
5. Updating the stack of events.

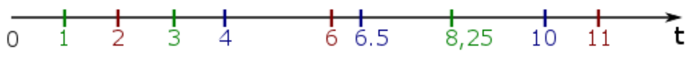


Figure B.6: Example of a stack of event for a sequential simulation of the coupled model of figure B.4. The events are arranged on the horizontal axis corresponding to the simulated time. The color of each event corresponds to the model processing the event. (red for m_1, blue for m_2, and green for m_3.)

Appendix C

Decentralized Execution of a DEVS Coupled Model with MECSYCO

This appendix is taken from [Camus et al., TBP].

C.1 Introduction

In this appendix, we explain the MECSYCO decentralized simulation algorithm based on the DEVS formalism (see Appendix B for more details on the DEVS formalism). In section C.2 we show how MECSYCO represents a DEVS coupled-model. In section C.3, we describe the MECSYCO algorithm of execution for decentralized simulation of discrete-event system. Finally, in section C.4 we discuss the advantages of this algorithm for the multi-simulation of complex systems.

C.2 Representation of a DEVS coupled model in MECSYCO

The translation of a DEVS coupled model, into the concepts of MECSYCO is done as follow [Siebert, 2011]. Each model m_i is associated with an m-agent $\mathcal{A}_i$ thanks to an interface artifact $\mathcal{I}_i$. The links between two models m_i and m_j corresponds a coupling artifact $\mathcal{C}_j^i$ between the m-agents $\mathcal{A}_i$ and $\mathcal{A}_j$ (Figure C.1).

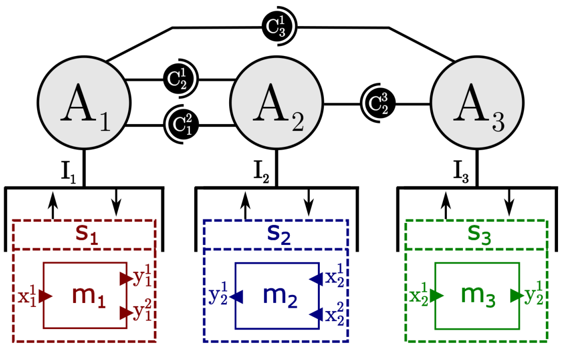


Figure C.1: The MECSYCO configuration corresponding to the coupled model of figure B.4

In MECSYCO, there is no DEVS global coordinator synchronizing the multi-simulation: the multi-simulation execution is done in a decentralized way by the m-agents. Therefore, the behavior of an m-agent corresponds to the parallel abstract simulator of the model. In MECSYCO, an m-agent only has a local knowledge of the coupled model's links. It does not know neither the other m-agents nor their model's input and output ports. The coupled model's IC is split in the multi-model. An m-agent $\mathcal{A}_i$ only knows:

- which input coupling artifact has the event buffer of each of its model's input ports. We define the set of input links IN_i of $\mathcal{A}_i$ as being composed of the couples (j, k) mapping the input coupling artifact $\mathcal{C}_i^j$ with the input port x_i^k.
- to which output coupling artifact it must sends the external events of each of its model's output ports. We defines the set of output links OUT_i of $\mathcal{A}_i$ as being composed of the couples (n, j) mapping the ouput port y_i^n with the output coupling artifact $\mathcal{C}_j^i$.

The mapping of the output ports of a model m_i with the input ports of a model m_j is done by the coupling artifact $\mathcal{C}_i^j$. We defines the set L_j^i of links from m_i to m_j as being composed of the couples (n, k) mapping the output port y_i^n with the input port x_j^k. The table C.1 shows how the coupled model of table B.1 is formalized in the MECSYCO formalism.

Descriptions	**Notations**
Output links of m_1	$\mathsf{OUT}_1 = \{(1,2),(2,3)\}$
Input links of m_1	$\mathsf{IN}_1 = \{(2,1)\}$
Output links of m_2	$\mathsf{OUT}_2 = \{(1,1)\}$
Input links of m_2	$\mathsf{IN}_2 = \{(1,2),(3,1)\}$
Output links of m_3	$\mathsf{OUT}_3 = \{(1,2)\}$
Input links of m_3	$\mathsf{IN}_3 = \{(1,1)\}$
Links from m_1 to m_2	$\mathsf{L}_2^1 = \{(1,2)\}$
Links from m_1 to m_3	$\mathsf{L}_3^1 = \{(2,1)\}$
Links from m_2 to m_1	$\mathsf{L}_1^2 = \{(1,1)\}$
Links from m_3 to m_2	$\mathsf{L}_2^3 = \{(1,1)\}$

Table C.1: Formalization of the coupled model of figure B.4 and Table B.1 in the MECSYCO formalism.

C.3 Decentralized simulation of a multi-model with MECSYCO

C.3.1 Issue of a distributed execution

In MECSYCO, there is no global coordinator for managing the execution of the global multi-simulation. According to the concept of agent, each m-agent is autonomous and manages only its own model. The simulation of the multi-model is made thanks to a distributed execution of the multi-model.

The consequence of the memory distribution is that each m-agent only has access to the stack of events of its own model: the global stack of event is unknown (as illustrated in the figure C.2). The problem is that the information required for the coordination of the simulation execution is split in the different m-agents. There is then a risk of breaking the causality constraint (i.e. that the events will be not processed in an increasing timestamp order).

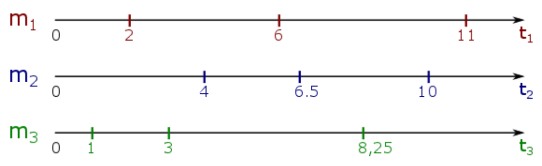


Figure C.2: Example of the stacks of events for a distributed execution of the coupled model of figure B.4. The events are arranged on the horizontal axis corresponding to the simulated time. Each m-agent $\mathcal{A}_i$ can only know the stack of event of its model m_i

Following a conservative approach in MECSYCO, all the m-agent have to be sure before any processing to respect the causality constraint. As, the information required to ensure this point is spread across several models (Figure C.2), the m-agents may be waiting on information of each other. In this case, deadlock will occurs. Any conservative approach must then be proved to be deadlock free.

We take the principle of the Chandy-Misra-Bryant algorithm [Chandy and Misra, 1979] for DEVS: in parallel of the external events flow, the temporal information required for the models synchronization is homogeneously disseminated in the multi-model during the simulation. In the following we present this algorithm. Proof that this algorithm is deadlock free and respect the causality constraint can be found in [Zeigler et al., 2000].

In the next sections, we detail how this algorithm works. In section C.3.2, we show how this algorithm ensures the respect of the causality constraint. The section C.3.3 details the processing of event. Sections C.3.4 and C.3.5 respectively detail the simulation initialization and stopping condition. The algorithm is detailed in section C.3.6.

C.3.2 Determining which events are safe to process

In order to fulfill the causality constraint, before processing any event (internal or external) with a timestamp t an m-agent has to be sure that no external input event will arrive with a timestamp inferior to t. If an event fulfills this condition, it is then said safe to process.

In order to know which events are safe to process, each m-agent $\mathcal{A}_i$ needs to determine its estimated earliest input time (EIT_i).

Definition: The EIT_i of an m-agent $\mathcal{A}_i$ corresponds to the (simulation) time below which it will not receive any new external input event. All the events (internal or external) with a timestamp inferior or equal to this time are safe to process.

The computation EIT_i, is made possible thanks to the communication by all the m-agents of their estimated Earliest Output Time (EOT). To determine EOT_i, each m-agent $\mathcal{A}_i$ computes its lookahead.

Definition: The **lookahead** of an m-agent corresponds to the date (in simulation time), below which it guarantees it will not send new external output event (even if it receives new external input events). The lookahead of an m-agent A_i is then equal to the minimum between:

- the date of its model's next internal event nt_i
- the date of its earliest (in term of simulation time) external event waiting to be processed in its input coupling artifacts t_{in_i} plus its model's minimum delay transfer Dt_i. This date corresponds to the limit below which it is guaranty that all the already received (but not processed yet) external events cannot schedule new internal events.
- the date of EIT_i plus its model's minimum delay transfer Dt_i. This date corresponds to the limit below which it is guaranty that all the future external events that will be dropped in its input coupling artifacts cannot schedule new internal events

$Dt_i(Dt_i > 0)$ corresponds to the minimum delay below which the processing of an external event cannot schedule a new internal event in m_i. The Dt_i of each model in the multi-model has to be determined.

$$Lookahead_i = min(nt_i, EIT_i + Dt_i, t_{in_i} + Dt)$$

Each m-agent updates its EOT in its output coupling artifacts, and can access to the EOT of the other m-agents with its input coupling artifacts. Therefore, each coupling artifact $\mathcal{C}_j^i$ proposes two functions to store and access to EOT_i.

- $\mathcal{A}_i$ can use the *setEOT* operation to update EOT_i in the artifact.
- $\mathcal{A}_j$ can use the *getEOT* operation to get EOT_i.

The EIT_i of an m-agent $\mathcal{A}_i$ corresponds to the minimum EOT of the EOTs given by all the input coupling artifacts of $\mathcal{A}_i$.

C.3.3 Processing safe events

In order to process safe events in a temporal increasing order, the behavior of an m-agent $\mathcal{A}_i$ follows the cycle:

1. Get the time nt_i of its next internal event ($nt_i \in \mathbb{R}$).
2. Get the time t_{in_i} of the earliest external event of all its input coupling artifacts e_{in_i}. ($nt_i \in \mathbb{R}$)
3. Determine what is the earliest event between the next internal event or e_{in_i} .
4. Determine if this event is safe to process.
5. If yes, process the earliest event.
6. If this event is an internal event, propagate the resulting output external events to other agents.

C.3.4 Initialization of the simulation

We assume here that the distributed simulations begin at simulation time 0. Before starting the distributed simulation, each model may be initialized using information dependent on some other models. In other word, the models need to send and integrate external events of initialization. The initialization phase allows models to send external events to other models.

As these events are for the initialization of the multi-model, they must happen before (in term of simulation time) the beginning of the simulation. That is why the timestamp of every initialization events must be strictly inferior to 0. The coupling artifacts' original $\mathtt{EOT}$s are equals to $-\infty$ to prevent the m-agents from starting processing their model's internal events before receiving their initialization ones. When each m-agent has sent all its initialization events, it sets its output coupling artifacts' EOT to 0. This procedure guaranties that each m-agent integrates its initialization input event before processing the simulation ones. Figure C.3 illustrates this initialization phase.

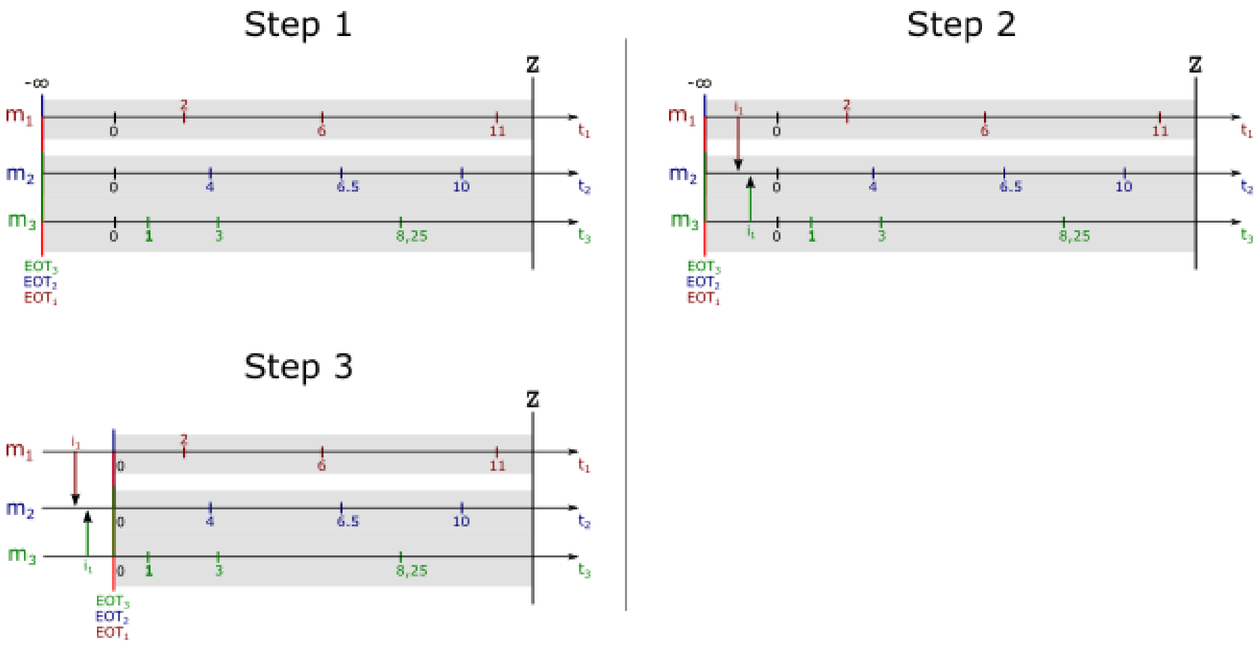


Figure C.3: Example of the initialization phase. Planned internal events are on the time axis (horizontal). Each vertical arrow shows the sending of an external event. Events outside the darkest zone are secured events. **Step 1:** *Link Times* are fixed at $-\infty$. M-Agents cannot process the internal events of their models. **Step 2:** M-Agents that need to send initialization events, send them. **Step 3:** All initialization events are sent, agents are now fixing the *Link Times* of their coupling artifact (sender) to 0.

C.3.5 Stopping condition for the end of the simulation

We consider here that we want to simulate the system up to time Z ($Z \in \mathbb{R}$). Two conditions exist in order to determine when the end of the simulation is reached. An m-agent stops the simulation when all of these two conditions are respected:

- It has processed all its model's internal events up to simulation time Z. This condition is observed when $\mathsf{EOT}_i > Z$.
- It has processed all its external events up to simulation time Z. This condition is heeded when:
  - It has received all the external events up to Z ($EIT_i > Z$).
  - It has processed all of these events ($t_{in_i} > Z$).

C.3.6 The behavior of the m-agents

The table C.2 sums up the notations of the MECSYCO meta-model. Tables C.3 and C.4 respectively shows the operations proposed by the coupling and the interface artifacts. The algorithm 1 describes the m-agent behavior.

Figure C.4 describes an example of distributed simulation execution of the coupled model of figure B.4. The values of the simulation's parameters for this example are described by table C.5. The state of the m-agents is described in table C.6. For sake of simplicity, we assume in that example that external events do not schedule new internal event in the models.

Level	**Description**	**Notation**
Conceptual	M-agent	$\mathcal{A}_i$
	Interface artifact	$\mathcal{I}_i$
	Coupling artifact from $\mathcal{A}_i$ to $\mathcal{A}_j$	$\mathcal{C}_j^i$
Semantic	Model	m_i
	Input links of m_i	IN_i
	Ouput links of m_i	OUT_i
	Links between m_i's output and m_j's input	L_j^i
Syntactic	Set of m_i's input ports	X_i
	Set of m_i's output ports	Y_i
	k^{th} input port of m_i	$x_i^k(0 < k \leq card(X_i))$
	n^{th} output port of m_i	$y_i^n(0 < k \leq card(Y_i))$
Dynamic	Maximum simulation time	Z
	Simulation time	t_i
	Time of the next internal event of m_i	nt_i
	Time of earliest external input event of m_i	t_{in_i}
	Earliest estimated input time of m_i	EIT_i
	Earliest estimated output time of m_i	EOT_i
	Earliest external input event of m_i	e_{in_i}
	External output event of port y_i^n	$e_{out_i}^n$

Table C.2: Notations of the MECSYCO meta-model

Operations	**Description**
$post(e_{out_i}^n, t_i)$	Sends the external event $e_{out_i}^n$ at time t_i
$getEarliestEvent(k)$	Returns the earliest external event at the input port x_i^k (or *null* if they is no external event)
$getEarliestEventTime(k)$	Returns the time of the earliest external event at the input port x_i^k (or $+\infty$ if they is no external event)
$removeEarliestEvent(k)$	Removes the earliest external event for the port x_i^k
$setEOT(t_i)$	Sets the time of the earliest estimated output time at time t_i
$getEOT()$	Returns the time of the earliest estimated output time of the coupling artifact

Table C.3: Operations proposed by the coupling artifacts

Operations	**Description**
$init()$	Initializes the simulator and sets the model's parameters
$getNextInternalEventTime()$	Returns the time of the next internal event of m_i
$getExternalOutputEvent(y_i^n)$	Returns the external output event of the output port y_i^n
$processExternalInputEvent(e_{in_i}, t_i, x_i^k)$	Processes the external input event e_{in_i} at time t_i in the input port x_i^k
$processInternalEvent(t_i)$	Processes the internal event scheduled at time t_i

Table C.4: Operations proposed by the interface artifacts

Parameters	**Values**
Dt_1	3
Dt_2	2
Dt_3	1
Z	12

Table C.5: Parameters of the simulation Figure C.4.

Step	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
EIT_1	0	2	3	4	6.5	6.5	9	10	12.5	13	EOS	EOS
EIT_2	0	1	3	6	6	7	9.5	10.5	12	12	16.5	EOS
EIT_3	0	2	5	6	6	9.5	9.5	11	11	15.5	EOS	EOS
EOT_1	2*	5	6*	6*	9.5	9.5	11*	11*	15.5	16.5	EOS	EOS
EOT_2	2	3	4*	6.5*	6.5*	9	10*	12.5	13	14	14	EOS
EOT_3	1*	3*	6	7	7	10.5	10.5	12	12	16.5	EOS	EOS

Table C.6: State of the m-agents from the simulation Figure C.4. Asterisk point out that $EOT_i = tn_i$. Else, $EOT_i = EIT_i + Dt_i$.

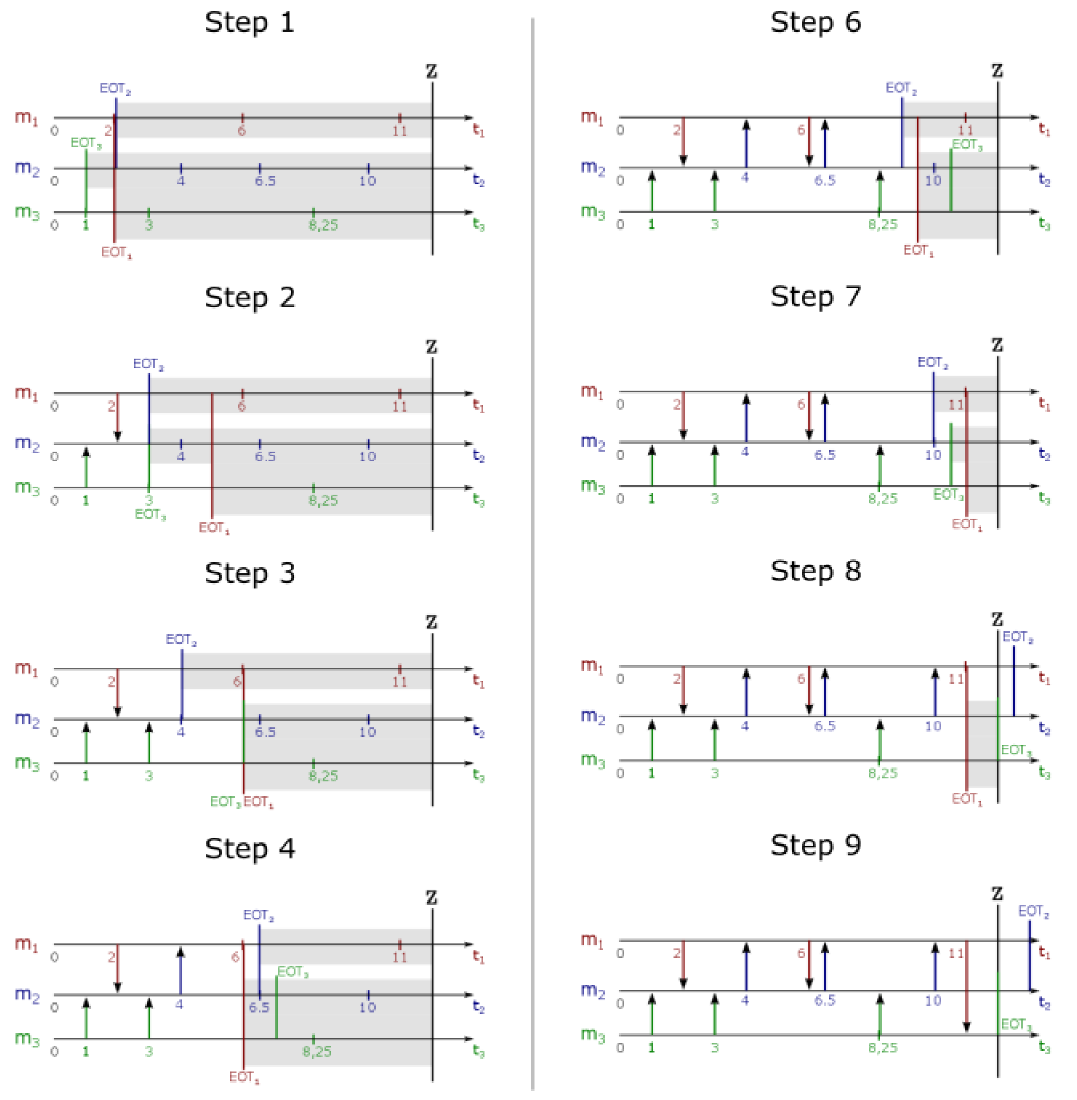


Figure C.4: Example of a run of a distributed simulation of a multi-model. Planned internal events are on the time axis (horizontal). Each vertical arrow shows the sending of an external event. Events outside the darkest zone are secured events.

Algorithm 1 Algorithm of the m-agent's behavior

```
INPUT: $\mathsf{IN}_i$, $\mathsf{OUT}_i, Z, Dt_i$
OUTPUT:
  $nt_i \leftarrow \mathcal{I}_i.getNextEventTime()$
  $t_{in_i} \leftarrow +\infty$
  $\mathsf{EOT}_i \leftarrow 0$
  $\mathsf{EIT}_i \leftarrow 0$
                         ▷ Until all internal and external events have been processed up to Z, do
  while $(\mathsf{EOT}_i \le Z)$ or $(\mathsf{EIT}_i \le Z)$ or $(t_{in_i} \le \mathsf{Z})$ do
    $nt_i \leftarrow \mathcal{I}_i.getNextInternalEventTime()$
    $\mathsf{EIT}_i \leftarrow +\infty$
    $t_{in_i} \leftarrow +\infty$
    for $(j,k) \in \mathsf{IN}_i$ do
      if $\mathcal{C}_i^j.getEOT() < \mathsf{EIT}_i$ then                        ▷ Compute the EIT
        $\mathsf{EIT}_i \leftarrow \mathcal{C}_i^j.getEOT()$
      end if
      if $\mathcal{C}_i^j.getEarliestEventTime(k) < t_{in_i}$ then         ▷ Get the earliest input event
        $t_{in_i} \leftarrow \mathcal{C}_i^j.getEarliestEventTime(k)$
        $e_{in_i} \leftarrow \mathcal{C}_i^j.getEarliestEvent(k)$
        $p \leftarrow k$
        $c \leftarrow j$
      end if
    end for
                         ▷ If there is, process the earliest safe event (internal or external)
    if $(nt_i \le t_{in_i})$ and $(nt_i \le \mathsf{EIT}_i)$ and $(nt_i \le \mathsf{Z})$ then      ▷ If it is an internal event
      $\mathcal{I}_i.processInternalEvent(nt_i)$                       ▷ Process the internal event
      $nt_i \leftarrow \mathcal{I}_i.getNextInternalEventTime()$
      for $n \in [1, card(Y_i)]$ do                        ▷ Send the resulting external events
        $e_{out_i}^n \leftarrow \mathcal{I}_i.getExternalOutputEvent(y_i^n)$
        if $e_{out_i}^n \neq \varnothing$ then
          for $(n,j) \in \mathsf{OUT}_i$ do
            $\mathcal{C}_j^i.post(e_{out_i}^n, lt_i)$
          end for
        end if
      end for
    else if $(t_{in_i} < nt_i)$ and $(t_{in_i} \le \mathsf{EIT}_i)$ and $(t_{in_i} \le \mathsf{Z})$ then    ▷ If it is an external event
      $\mathcal{I}_i.processExternalInputEvent(e_{in_i}, t_{in_i}, x_i^p)$         ▷ Process the external event
      $\mathcal{C}_i^c.removeEarliestEvent(p)$
      $nt_i \leftarrow \mathcal{I}_i.getNextInternalEventTime()$
    end if
                         ▷ Compute the EOT and update the output coupling artifacts
    $t_{in_i} \leftarrow +\infty$                                   ▷ Get the earliest input event time
    for $(j,k) \in \mathsf{IN}_i$ do
      if $\mathcal{C}_i^j.getEarliestEventTime(k) < t_{in_i}$ then
        $t_{in_i} \leftarrow \mathcal{C}_i^j.getEarliestEventTime(k)$
      end if
    end for
    if $\mathsf{EOT}_i \neq Lookahead_i(nt_i, EIT_i, t_{in_i})$ then       ▷ Update the output coupling artifacts
      $\mathsf{EOT}_i \leftarrow Lookahead_i(nt_i, EIT_i, t_{in_i})$
      $\forall (k,j) \in \mathsf{OUT}_i : \mathcal{C}_j^i.setEOT(\mathsf{EOT}_i)$
    end if
  end while
```

C.4 Discussion

An advantage of this algorithm is that it allows the integration of different scheduling policies:

- If the model is discrete in time, the simulator cannot advance the simulation time between two internal events. In consequence, the *processExternalInputEvent* operation only computes the external event in the model without changing the time of the model. This may lead to slight errors in the simulation results, but they are inherent to a discrete time representation of the system.
- Cyclic time-step models can be considered as discrete event models with internal event scheduled in a regular manner in simulation time. For these models, the *getNextInternalEventTime* can simply return the current time of the model plus the size of a time-step. In this case, the *processInternalEvent* operation executes the model for one simulation step.
- If external events cannot schedule new internal event in a model m_i, then $Dt_i = +\infty$. As a consequence, EOT_i will always be equal to tn_i.

Moreover, this algorithm takes directly account of the dependencies between models. Indeed, if a model m_i does not have any input link, then $\mathsf{EIT}_i = t_{in_i} = +\infty$. The m-agent will then always compute all its model's internal events and propagate the resulting external events as fast as it can until $\mathsf{EOT}_i > Z$.

Bibliography

[Blochwitz et al., 2011] Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clau, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T., and et al. (2011). The functional mockup interface for tool independent exchange of simulation models. In *8th International Modelica Conference, Dresden.*

[Camus et al., TBP] Camus, B., Bourjot, C., and Chevrier, V. ((TBP)). An execution algorithm for distributed discret-event simulation with aa4mm. Technical report, Université de Lorraine.

[Chandy and Misra, 1979] Chandy, K. M. and Misra, J. (1979). Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, (5):440–452.

[ECMA, 2013] ECMA (2013). Ecma-404 the json data interchange standard. 1st edition.

[Fishwick, 2007] Fishwick, P. A. (2007). *Handbook of dynamic system modeling.* CRC Press.

[Fujimoto, 2001] Fujimoto, R. M. (2001). Parallel simulation: parallel and distributed simulation systems. In *Proceedings of the 33nd conference on Winter simulation*, WSC '01, pages 147–157, Washington, DC, USA. IEEE Computer Society.

[OMG, 2007] OMG (2007). Data distribution service (dds), version 1.2.

[Quesnel et al., 2009] Quesnel, G., Duboz, R., and Ramat, É. (2009). The virtual laboratory environment – an operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simulation Modelling Practice and Theory*, 17(4):641 – 653.

[Ricci et al., 2007] Ricci, A., Viroli, M., and Omicini, A. (2007). Give agents their artifacts: the a&#38;a approach for engineering working environments in mas. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, AAMAS '07, pages 150:1–150:3, New York, NY, USA. ACM.

[Siebert, 2011] Siebert, J. (2011). *Approche multi-agent pour la multi-modélisation et le couplage de simulations. Application à l'étude des influences entre le fonctionnement des réseaux ambiants et le comportement de leurs utilisateurs.* These, Université Henri Poincaré - Nancy I.

[Vangheluwe, 2000] Vangheluwe, H. (2000). Devs as a common denominator for multi-formalism hybrid systems modelling. In *Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium on*, pages 129–134.

[Vangheluwe et al., 2002] Vangheluwe, H., De Lara, J., and Mosterman, P. J. (2002). An introduction to multi-paradigm modelling and simulation. In *Proc. AIS2002. Pp*, pages 9–20.

[Zeigler et al., 2000] Zeigler, B., Praehofer, H., and Kim, T. (2000). *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press.