Multi-agent Environment for Complex SYstems COsimulation (MECSYCO) - User Guide: Create your own operations

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`

March 31, 2016

Contents

Introduction

MECSYCO is a platform that allows to use different simulators in one multi-model at the same time. But simulators do not always use the same type of data for their functioning. For example a simulator could be sending real value, but the next simulator need event information such as a Boolean to work or even its proper type (like NetLogoTurtle that belongs to NetLogo).

The operations for data transformation are designed for that issue. It is used by the coupling artifact, more specifically in the receiver side (see *User Guide section "The coupling artifact"*). An operation is designed specifically to transform simulation data (see *User Guide section "Simulation data"*) sent by a simulator's output port, into simulation data needed by a simulator's input port. There is also the equivalent for time conversion in order to adapt the unit, and then put it to scale.

As a consequence, there is no pre-existing operation and the user should define its own operation depending on its simulators and models.

Chapter 1

Operation construction

Defining an operation consists of creating and instantiating a class extending the abstract class **EventOperation** for data operation, and the abstract class **TimeOperation** for time operation. The operation's designer is free to add attributes and other methods to its operation in order to parametrize the operation.

1.1 Data operation

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

apply
Parameters:

- **aEvent** - the event containing the data to process

Returns: the processed event (see *User Guide section "Simulation event"*)

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

apply method in C++ implementation
Not distributed yet

After creating the operation, all you need to do in the multi-model description, is to add it on the link you want the operation to operate. To do so, add the operation to the *CouplingArtifact* used for the link by using this method:

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

1.2 Time operation

In order to put to scale time, every operation must extend **TimeOperation** and define the following method of the **TimeOperation** class:

apply
Parameters:

- **aTime** - the time where the operation should act

Returns: The converted time.

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

apply method in C++ implementation
Not distributed yet

Then, as for the previous operation, you can add it to the *CouplingArtifact* used for the link by using this time, the following method:

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

Pre-made time operation

Some basic operations were already implemented and can be used. Those operations are addition, division, multiplication and exponential, they use the following constructors:

AdditionTimeOperation constructor in Java implementation
public AdditionTimeOperation (double aOperand)

DivisionTimeOperation constructor in Java implementation
public DivisionTimeOperation (double aDivident)

ExponentTimeOperation constructor in Java implementation
public ExponentTimeOperation (int aPower)

MutliplicationTimeOperation constructor in Java implementation
public MutliplicationTimeOperation (double aTimes)

- **aOperand**: value you want to add (subtract if aOperand is negative).
  $- > time + aOperand$
- **aDivident**: the value you will use to divide the time (must be different of 0).
  $- > time/aDivident$
- **aPower**: the power applied to 10.
  $- > time * 10^{aPower}$
- **aTimes**: the value used to multiply the time.
  $- > time * aTimes$

Remark

You can add as many operations you want, they will be applied in the same order you add them.

Operation can also be combine with DDS for decentralized simulation. You can find more information about it in the *User Guide: MECSYCO-com-dds*.

Chapter 2

Template

In this section, we give templates of launcher using time and event operations, template of data operations and template of time operations. Those template are only for Java using. C++ is not implemented yet. When we talk about "flexible" it is just that with this method, you could change easily the value used for the operation thanks to the constructor. It is not "flexible" when you cannot manage these value, as a consequence, the operation will always do one and only one thing.

2.1 Java Example Template: run configuration (with operations)

```java
import mecsyco.core.agent.EventMAgent;
import mecsyco.core.coupling.CentralizedEventCouplingArtifact;
import mecsyco.core.exception.CausalityException;
import mecsyco.core.operation.time.AdditionTimeOperation;
import mecsyco.core.operation.time.DivisionTimeOperation;
import mecsyco.core.operation.time.ExponentTimeOperation;
import mecsyco.core.operation.time.MutliplicationTimeOperation;


public class LauncherWithOperations {
      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");

        /******************************/
        /********* Operations *********/
        /******************************/

        /*
         * On the link from X to x, we will put the freshly made DataOperation
         * We assume here that the operation do not need parameters
         */
        //create the operation
        DataOperationTemplate EventOpe=new DataOperationTemplate();
        //add it to the link
        couplingFrom1To2.addEventOperation(EventOpe);

        /*
         * On the link from Y to y, we will put the freshly made TimeOperation
         * (we assume here that the operation do not need parameters)
         * and all operation to our disposal
         */
        //create the operations
        TimeOperationTemplate TimeOpe1=new TimeOperationTemplate();
        AdditionTimeOperation TimeOpe2=new AdditionTimeOperation(1);
        DivisionTimeOperation TimeOpe3=new DivisionTimeOperation(2);
        ExponentTimeOperation TimeOpe4=new ExponentTimeOperation(3);
        MutliplicationTimeOperation TimeOpe5=new MutliplicationTimeOperation(4);
        //add it to the link
        couplingFrom2To1.addTimeOperation(TimeOpe1);
        couplingFrom2To1.addTimeOperation(TimeOpe2);
        couplingFrom2To1.addTimeOperation(TimeOpe3);
        couplingFrom2To1.addTimeOperation(TimeOpe4);
        couplingFrom2To1.addTimeOperation(TimeOpe5);
        couplingFrom2To1.addTimeOperation(TimeOpe1);
        /*
         * the timestamp send by Model2 will then be:
         * manipulate by the freshly made operation
         * adding 1 to the previous operation
         * divide the result by 2
         * multiply it by 10^3
         * multiply the latest result by 4
         * then manipulate the final result, once again, by the freshly made operation
         */
```

```
    /*******************************/
    /**** 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();
    }
  }
}
```

2.2 Java Example Template: Data operation

```java
import mecsyco.core.operation.event.EventOperation;
import mecsyco.core.type.SimulData;
import mecsyco.core.type.SimulEvent;


public class DataOperationTemplate extends EventOperation{

    /*
     * Constructor
     * You can have empty, depend on if you want a flexible operation or a fixed one
     * (fixed because in any case, the operation will be exactly the same)
     * here, we use a constructor for flexible operation
     */
    public DataOperationTemplate (Type1 aVars1, Type2 aVars2) {
        Vars1=aVars1;
        Vars2=aVars2;
    }

    /*
     *Implementation
     *In the case of flexible, create variable for the operation
     */
    private Type1 Var1;
    private Type2 var2;

    @Override
    public SimulData apply (SimulEvent aEvent) {
        /*the result, can be any SimulData type you want
        *(for example, the freshly made type done in the "User Guide: SimulData manipulation"
        */
        DataTypeTemplate result;

        //extract the simulData from SimulEvent. Instead of SimulData, you can precise
        //the exact SimulData type and then use methods you or we had implemented
        SimulData data=aEvent.getData();
        /*
         * Operation you want to do
         * For making it flexible, use variables you created (Vars1 and Vars 2 here)
         * Do not forget to assign the result of the operation in the variable "result"
         */

        return result;
    }

}
```

2.3 Java Example Template: Time operation

```java

import mecsyco.core.operation.time.TimeOperation;
import mecsyco.core.type.SimulData;
import mecsyco.core.type.SimulEvent;

public class TimeOperationTemplate extends TimeOperation{
    /*
     * Constructor
     * You can have empty, depend on if you want a flexible operation or a fixed one
     * (fixed because in any case, the operation will be exactly the same)
     * here, we use a constructor for flexible operation
     */
    public TimeOperationTemplate (Type1 aVars1, Type2 aVars2) {
        Vars1=aVars1;
        Vars2=aVars2;
    }

    /*
     *Implementation
     *In the case of flexible, create variable for the operation
     */
    private Type1 Var1;
    private Type2 var2;

    @Override
    public double apply(double arg0) {
        //the result, can be any SimulData type you want
        Double result;
        /*
         * Operation you want to do
         * For making it flexible, use variable you created (Vars1 and Vars 2 here)
         * Math operation can be found in java class Math (Math.***)
         * Don't forgot to assign the result of the operation in the variable "result"
         */

        return result;
    }
}
```