Multi-agent Environment for Complex SYstems COsimulation (MECSYCO) - User Guide: SimulData manipulation

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

A simulation data corresponds to the data exchanged between simulators during a multi-simulation. In order to be exchanged, a simulation data need to be contained in a simulation event (*User Guide section "Simulation event"*). Each simulation data is specific to one or several simulator ports (input or output). As a consequence, the designer may have to create its ows simulation data.

Chapter 1

Structure

In order to make MECSYCO the more generic as possible, we defined a special class for the data exchanged: *SimulData*. Each time a simulation data is created, it only needs to inherit of *SimulData* (Figure 1.1). Thanks to this, all simulation types will have the same properties and will then be easily managed.

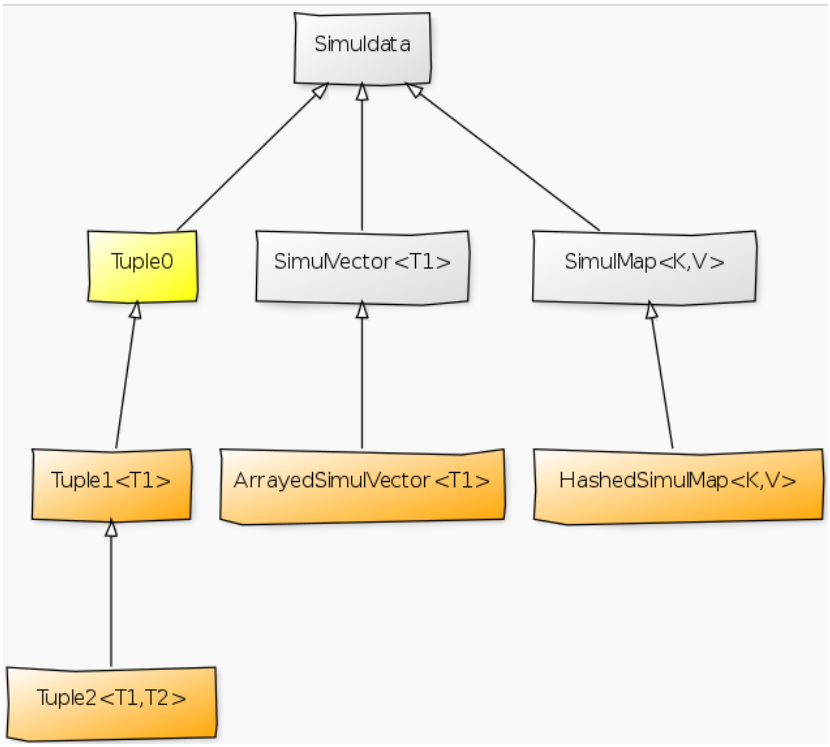

Figure 1.1: Predefined simulation data types

- **Grey box:** Abstract/Deferred class
- **Orange box:** Instantiable class
- **Yellow box:** Singleton

All simulation types are immutable types. An immutable object is an object which cannot be modified during the software lifecycle. Thereby a call to a method on an object do not change the object and call to a method returns a result. Immutable objects are thread-safe by nature. And more, immutability removes a lot of complex mistakes which arises in large systems.

These SimulData are used in any multi-model, distributed or not, then, in order to make them compliant with DDS, you need to use Jackson (add of getter and setter, or by using @FsonProperty in the constructor, cf section 3).

Chapter 2

Predefined

As shown with figure 1.1, we already defined some simulation data types. Most of the time you will not need to create your own, because what we defined are generic and can manage most of the data. We detail here the predefined types: Tuples, Vectors and Maps.

2.1 Tuples

A tuple is a finite ordered list of potentially heterogeneous items. An n-tuple is a tuple with n items. A tuple should be use in simple cases. In particular when few items need to be exchanged between two models. In more complex cases, prefer vector (section 2.2), maps (section 2.3) or design your own type.
The type of a tuple is defined as a product type. For example, Integer x String is a 2-tuple with an integer as first item and a string as second item. MECSYCO provides six tuples:

- **Tuple0** is a 0-tuple
- **Tuple1<T1>** is a 1-tuple of type T1
- **Tuple2<T1, T2>** is a 2-tuple of type T1 x T2
- **Tuple3<T1, T2, T3>** is a 3-tuple of type T1 x T2 x T3
- **Tuple4<T1, T2, T3, T4>** is a 4-tuple of type T1 x T2 x T3 x T4
- **Tuple5<T1, T2, T3, T4, T5>** is a 5-tuple of type T1 x T2 x T3 x T4 x T5

Examples:

- Tuple1< String> t1 = new Tuple1<>(”str”);
- Tuple2<Integer, String> t2 = new Tuple2<>(1, ”str”);
- Tuple3<Integer, String,Double> t3 = new Tuple3<>(1, ”str”,1.0);
- Tuple4<Integer, String, String, Double> t4 = new Tuple4<>(1, ”str”, ”str2”,1.0);
- Tuple5<Double, String, Double, Integer,Integer> t5 = new Tuple5<>(1.0, ”str”, 2.0, 1, 2);

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)

2.2 Vectors

A vector is a finite ordered list of items. A vector should be use to exchange a list of items of same types (parameter G). MECSYCO provides a single implementation of *SimulVector*: *ArrayedSimulVector*. *ArrayedSimulVector* uses internally a native array making an efficient access to each item. *ArrayedSimulVector* is a null-safe and immutable structure. Thereby you must initialize all items of the vector at creation time.

ArrayedSimulVector constructor in Java implementation
public ArrayedSimulVector (@JsonProperty("items") G... aItems)

Example:
SimulVector<Integer> v = new ArrayedSimulVector<>(1, 2, 3, 4, 5);

ArrayedSimulVector should be used only to instantiate a vector. Use *SimulVector* for your variable types.

2.3 Maps

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). MECSYCO provides a single implementation of *SimulMap*: *HashedSimulMap*. *HashedSimulMap* uses internally a hash map, making an efficient access to each value.

HashedSimulMap constructors in Java implementation
public HashedSimulMap (Tuple2<K, V>... aPairs)
public HashedSimulMap (@JsonProperty("mappings") Map<K, V> aMapping)
public final static <K extends Serializable, V extends Serializable> HashedSimulMap<K, V> empty ()

K for the key and V for the value.
Examples

- Create an HashMap beforehand:
  Map<String, Integer> m = new HashMap<>();
  m.put("a", 1);
  m.put("b", 2);
  SimulMap<String, Integer> sm = new HashedSimulMap<>(m);
- Create an empty one:
  SimulMap<String, Integer> sm = new HashedSimulMap<>();
  sm = sm.with("a", 1).with("b", 2);
- or like the others, directly:
  SimulMap<String, Integer> sm = new HashedSimulMap<>(new Tuple2<>("a", 1), new Tuple2<>("b", 2));

2.4 Use of a SimulData

SimulData are how data exchanged between agents are perceived by MECSYCO. The conversion is done when the data transmitted become a *SimulEvent* (*User Guide section Simulation event*). That means that when you want to implement them, you will need to do it (most of the time) inside a *ModelArtifact* (see *User Guide section "The interface artifact"* and *User Guide: Model Artifact*).

2.4.1 Methods

We will take the example from the Lorenz case, but how to instantiate the *SimulData* depends on the constructor of the *SimulData*.
So in the *ModelArtifact*, each time you need to manage or create a *SimulEvent*, you will need to fill it with a *SimulData*.

SimulEvent constructor in Java implementation
public SimulEvent (@JsonProperty("data") SimulData aData, @JsonProperty("time") double aTime)

- **aData:** The simulation data to manage
- **aTime:** Time when the SimulEvent was send

```
final SimulEvent result;

if ("obs".equalsIgnoreCase(port)) {
    final Double x = equation.getVariable("X");
    final Double y = equation.getVariable("Y");

    result = new SimulEvent(new Tuple2<>(x, y), equation.getTime());
}
else if ("obs3d".equalsIgnoreCase(port)) {
    final Double x = equation.getVariable("X");
    final Double y = equation.getVariable("Y");
    final Double z = equation.getVariable("Z");

    result = new SimulEvent(new Tuple3<>(x, y, z), equation.getTime());
}
else {
    SimulData data = new Tuple2<>(equation.getVariable(port), port);
    result = new SimulEvent(data, equation.getTime());
}

return result;
```

Figure 2.1: *getExternalOutputEvent* implementation in *EquationModelArtifact*.

In the Lorenz case, the *ModelArtifact* is *EquationModelArtifact*. It is a simple example of *SimulData* instantiation because Lorenz use Double (value calculated) and String (name indication).
In *EquationModelArtifact*, we need *SimulData* when creating the *SimulEvent* to send (Fig 2.1).

In this method, we defined the agent outputs. There is three different kinds depending on the one you are calling in the launcher. You need to know that the model *"Equation"* was defined specific output names (X, Y and Z) and they send real.

- **obs:** If you want to call to a new port name "obs". We first get the original value (real) from the outputs X and Y then convert them in a *Tuple2* inside the *SimulEvent*.
- **obs3D:** Same as for "obs" but we created a port that sends *Tuple3*.
- **Outputs:** The last condition (else) allows us to transform the original port in *SimulData*. Here we decided to put not only the value but the name of the output port too. The simulation data then become *Tuple2* of Double and String. As a consequence, in the multi-model, each agent will be able to transmit the value by using the outputs' name (X, Y and Z).

Usually, agents do not only send but can receive too. Thanks to the *getExternalOutputEvent* method, data transmitted are contains in a *SimulEvent* with the form of a *SimulData*. At the reception we need to manage them with *processExternalInputEvent*! (Fig 2.2)

```
final SimulData data = event.getData();

if (data instanceof Tuple1) {
    final Object value = ((Tuple1<?>) event.getData()).getItem1();

    if (value instanceof Number) {
        equation.setVariable(port, ((Number) value).doubleValue());
    }
    else {
        modelLogger.error("processExternalInputEvent",
                new UnexpectedTypeException(Tuple1.of(Number.class), Tuple1.of(value.getClass())));
    }
}
else {
    modelLogger.error("processExternalInputEvent",
            new UnexpectedTypeException(Tuple1.of(Number.class), data.getClass()));
}
```

Figure 2.2: processExternalInputEvent implementation in EquationModelArtifact.

In this method, we first extracted the *SimulData* from the *SimulEvent* then checked if that was the type expected (Tuple). If yes, we just convert it back to its original state (Double) and put it in the inputs of *equation* in order to process the next step of the calculation.

2.4.2 Note

You can notice that we did not follow the notation Tuplex<T1,...,Tx>, but use Tuplex<> instead. When the types are not defined, you can use whatever you want, the type will be automatically detected. You just need to respect that vectors use only one type!

Chapter 3

Create a SimulData

The pre-made *SimulData* types can manage most of the models since data are usually real with name attached. But if they have a different meaning or are more complex, you can create your own *SimulData* type. Even if tuple are a generic type, for any kind of new type, you should avoid to inherit of tuple but directly inherits of *SimulData*. *SimulData* depends on certain components that implie that some part are really important when designing a new type. The commented template contains the essential part.

The component is Jackson, and is needed for using the distribution tool *MECSYCO-com-dds*. In order to make your SimulData you then need to have a particular structure. There is two methods:

- **Method 1:** For each instance attribute of the SimulData, create a getter and a setter. If some attribute are not to be transmitted, add the annotation *@JsonIgnore* before the getter.
- **Method 2:** In the constructor, add the annotaion @JsonProperty("name_attribute") before the constructor's parameter use to set the attribute.

3.1 Java Example Template: SimulData type construction

```
1  import com.fasterxml.jackson.annotation.JsonIgnore;
   import com.fasterxml.jackson.annotation.JsonProperty;
3
   import mecsyco.core.type.SimulData;
5
7
9  public class DataTypeTemplate implements SimulData{
       //Version for avoid warning from Serializable
11     /**
        *
13      */
       private static final long serialVersionUID = 1L;
15
       /*
17      * Implementation
        * variables contain in your new SimulData type
19      * "Type_x" can be anything, double, int, string long, array, matrix, whatever you want and as many you need
        */
21     private Type_1 Var1;
       private Type_2 Var2;
23     private Type_3 Var3;
25
       /*
27      * constructor
        */
29     public DataTypeTemplate (Type_1 aVar1,  Type_2 aVar2, Type_3 aVar3){
           Var1=aVar1;
31         Var2=aVar2;
           Var3= aVar3;
33     }
35     /*
        * For Jackson, in order that this type can be use in DDS (see User Guide: MECSYCO-com-dds)
37      */
       //empty constructor
39     public DataTypeTemplate (){}
41     //getter and setter
       //Since Jackson 1.9, setter and empty constructor are not mandatory
43     public final Type_1 getVar1(){
           return Var1;
```

```
45      }
        public final void setVar1(Type_1 aVar1){
47          Var1=aVar1;
        }
49
        public final Type_2 getVar2(){
51          return Var2;
        }
53      public final void setVar2(Type_2 aVar2){
            Var2=aVar2;
55      }

57
        //If you don't want that a variable is send by Jackson in DDS, you can ignore it
59      //by using @JsonIgnore before a getter or a setter (or you don't use getter and setter for this variable)
        @JsonIgnore
61      public final Type_3 getVar3(){
            return Var3;
63      }
        public final void setVar3(Type_3 aVar3){
65          Var3=aVar3;
        }
67
        /*
69       * Jackson, 2nd method to use it
         * Instead of using empty constructor, you can use @JsonProperty("variable_name") in the normal constructor
71       * You still need at least getter (and setter for Jackson under 1.9)
         */
73      public DataTypeTemplate (@JsonProperty("Var1") Type1 aVar1, @JsonProperty("Var2") Type2 aVar2, @JsonProperty("Var2") Type3 aVar3){
            Var1=aVar1;
75          Var2=aVar2;
            Var3= aVar3;
77      }

79      /*
         * Optional methods
81       * any methods you think can be helpful (debugging, status, functional etc...)
         */
83

85  }
```