A Model-Driven Round-trip Engineering Approach for Systems-of-Systems Architectural Models - SosADL2DEVS transformation details

Valdemar Vicente Graciano Neto[a,*], Wallace Manzano[b], Lina Garcés[b], Carlos Eduardo Paes[c], Elisa Yumi Nakagawa[b]

[a] *Informatics Institute, Federal University of Goiás, Goiânia, Brazil*
[b] *University of São Paulo, São Carlos, Brazil*
[c] *Pontifical Catholic University of São Paulo, São Paulo, Brazil*

Abstract

For the reader convenience, this document presents details on how a model transformation was specified to map SosADL models into DEVS models. For didactic reasons, such transformation is discussed below in two parts: generation of atomic models, and generation of coupled models. Examples are concerned to a Flood Monitoring system-of-system (SoS).

Keywords: Systems-of-Systems, Model-driven round-trip engineering, Reverse Engineering, Architectural Models, Simulation

1. Generation of Constituent Models

Listing 1 shows a simplified code of mediator specified in SosADL. This code is mapped into an atomic model written in DEVS depicted in Listing 3. The transformation is performed by the code specified in Xtend available in Listing 2. In Listing 1, data types are defined on Lines 2-6. Duties (in this context, only a name for the designation of gates and mediators) with their respective connections are defined on Lines 8-16. Behavior of the mediator is specified between Lines 18 and 23, and shows that a mediator (i) receives constituents coordinates (Lines 19 and 20), (ii) receives data from the sensors (Line 22) and (iii) forward such data towards a gateway (Line 23). This sequence of actions is performed in a loop.

```
1  mediator Transmitter( distancebetweengates:Distance ) is {
2    datatype Abscissa
3    datatype Ordinate
4    datatype Coordinate is tuple { x:Abscissa, y:Ordinate }
5    datatype Depth
6    datatype Measure is tuple { coordinate:Coordinate, depth:Depth }
7
8    duty transmit is {
9      connection fromSensors is in { Measure }
```

☆Supplementary material

*Corresponding author
Email address: valdemarneto@inf.ufg.br (Valdemar Vicente Graciano Neto)

```
10        connection towardsGateway is out { Measure }
11      }
20 12
13      duty location is {
14        connection fromCoordinate is in { Coordinate }
15        connection toCoordinate is in { Coordinate }
16      }
25 17
18      behavior transmitting is {
19        via location::fromCoordinate receive coordinate
20        via location::toCoordinate receive coordinate
21        repeat {
30 22       via transmit::fromSensors receive measure
23          via transmit::towardsGateway send measure
24        }
25      }
26    }
```

Listing 1: Code in SosADL for a mediator.

35 Lines 2-6 in Listing 1 represent the definition of data types in SosADL. Data types are transformed by Lines 1-16 in Listing 2 to produce Lines 1-22 in Listing 3. Lines 8-16, which specify the connections and duties (gates) of a mediator in SosADL in Listing 1, are transformed by Lines 18-23 (more specifically, Lines 21 and 27) in Listing 2 to produce Lines 24-27 in Listing 3, whereas Lines 18-23, which represent the behavior of a mediator specified in SosADL in Listing 1, are transformed by Lines 23-24 and 29-31 in Listing 2 to 40 produce Lines 24-42 in Listing 3.

```
1  def compile(DataTypeDecl d)'''
2      <<IF !d.datatype.isADT>>
3          A <<d.name>> has a value!
4          the range of <<d.name>>'s value is Integer!
45 5       use <<d.name.toFirstLower>> with type <<d.name>>!
6      <<ELSE>>
7          <<d.name.toFirstUpper>> has <<d.datatype.compile>>
8          <<IF (d.datatype as DataType) instanceof TupleType>>
9          <<var e = (d.datatype as TupleType)>>
50 10      <<FOR p : e.fields>>
11         the range of <<d.name>>'s <<p.name>> is <<(p as FieldDecl).type.compile>>!
12         <<ENDFOR>>
13         use <<d.name.toFirstLower>> with type <<d.name>>!
14         <<ENDIF>>
55 15      <<ENDIF>>
16 '''
17
18 def compile(Element e) {
19 if ((connection.type == INPUT) or (action.type == RECEIVE)) {
60 20   if(e instanceOf Connection) ports += '''
21   accepts input on <<connectionName.toFirstUpper>> with type <<dataReceived.type>>!'''
22
```

```
23    if(e instanceOf Action) transitions += '''passivate in s<<fromState>>!
24    when in s<<fromState>> and receive <<dataReceived>> go to s<<toState>>!'''
25
26  }else if((connection.type == OUTPUT) or (action.type == SEND)){
27    if(e instanceOf Connection) ports += '''generates output on <<this.connection.typeName>>
28    with type <<dataSent.type>>!'''
29
30    if(e instanceOf Action) transitions += '''hold in s<<fromState>> for time 1!
31    after s<<fromState>> output <<this.connection.typeName>>!
32    from s<<fromState>> go to s<<toState>>!
33    '''
34    }
35  }
```

Listing 2: Transformation code specified in Xtend.

Regarding the transformation of data types, Xtend code in Listing 2 establishes the following strategy: if the type of data is not an ADT (Line 2), by default, it will be converted to an Integer type (Lines 3-5). Xtend accesses the name of the data type available in the Abstract Syntax Tree and substitutes it in the appropriate places in the template of DEVSNL code (Line 3, Listing 2). In DEVSNL, a simple type has a value (as specified in Line 3), and tfhe range of this type (in case of simple types) is Integer (Line 4). A variable of such a type is declared in another statement to be used for processing purposes (Line 5). Statements in DEVSNL end with an exclamation mark. Conversely, if the data type specified in SosADL is a tuple (such as Coordinate (Line 4), and Measure (Line 6) in Listing 1), then Lines 7-15 (Listing 2) are executed. In this case, a type declaration in DEVS receives the name of the data type in SosADL, and their fields through an iteration structure (Lines 10-12 in Listing 2), and lines, such as Lines 11-14 in Listing 3, are generated.

Lines 18-31 in Listing 2 buffer the definitions of ports and state transitions in DEVSNL to be printed at the end of the process (this detail is hidden from the code presented). If the transformation consists of an input transition, the code in Lines 19-25 (Listing 2) are executed. If it consists of an output transition, the Lines 26-31 (Listing 2) are executed. Since the specification of a state diagram in DEVSNL follows a declarative style, the order of the instructions generated in DESVNL does not matter. The concepts of duty and gate are suppressed in DEVSNL. Therefore, the Xtend code takes the name of the connections in SosADL and uses them (Lines 21 and 27, Listing 2) to create ports in DEVSNL. The type of data received or sent are used in the typification of the data to be transmitted in a DEVS port (Lines 21 and 27, Listing 2). `Receive` instructions in SosADL create receive transitions in the form of Lines 23-24 in Listing 2, whereas `send` instructions in SosADL generate three lines of code in DEVSNL - one that holds in a state for time 1 second (this time was established as default), another that produces the output of some data, and another that performs the state transition to a next state (Lines 29-31). Some of the variables are global variables, and their declaration was hidden.

```
A Distance has a value!
the range of Distance's value is Integer!
use distance with type Distance!

A Abscissa has a value!
the range of Abscissa's value is Integer!
use abscissa with type Abscissa!
A Ordinate has a value!
the range of Ordinate's value is Integer!
use ordinate with type Ordinate!
Coordinate has x and y!
the range of Coordinate's x is Abscissa!
the range of Coordinate's y is Ordinate!
use coordinate with type Coordinate!

A Depth has a value!
the range of Depth's value is Integer!
use depth with type Depth!
Measure has coordinate and depth!
the range of Measure's coordinate is Coordinate!
the range of Measure's depth is Depth!
use measure with type Measure!

accepts input on FromCoordinate with type Coordinate!
accepts input on ToCoordinate with type Coordinate!
accepts input on FromSensors with type Measure!
generates output on Measure with type Measure!

to start hold in s0 for time 1!
hold in s0 for time 1!
from s0 go to s1! //Unobservable
passivate in s1!
when in s1 and receive Coordinate go to s2!
passivate in s2!
when in s2 and receive Coordinate go to s3!
passivate in s3!
when in s3 and receive Measure go to s4!
hold in s4 for time 1!
after s4 output Measure!
from s4 go to s5!
hold in s5 for time 1!
from s5 go to s3! //Unobservable
```

Listing 3: An atomic model for a Mediator generated in DEVSNL.

Each statement of a behavior in SosADL becomes one or more transitions in DEVS. In DEVS, transitions can occur due to (i) a data received (expressed as `?data`), (ii) a data sent (expressed as `!data`), and (iii) a spontaneous transition, with no input or output event. Listing 4 illustrates a behavior called `sensing` of a smart sensor, which senses the depth of the water in a river and is a constituent within the context of

FMSoS. Line 2 is an assignment statement in which a coordinate called `lps` is assigned to a variable called `sensorcoordinate`. In Line 3, the `lps` assigned to a `sensorcoordinate` in Line 2 is sent via a connection called a `coordinate` that has a gate called a `location`. Line 6 depicts a `repeat` statement in which the `powerlevel` is received via a connection called `power` in a gate called `energy`. The system receives the battery level to test whether the energy is enough to perform the instructions. Line 8 tests if the power level is above an established threshold.

```
1 behavior sensing is {
2   value sensorcoordinate is Coordinate = lps
3   via location::coordinate send sensorcoordinate
4   via energy::threshold receive powerthreshold
5
6   repeat {
7       via energy::power receive powerlevel
8       if( powerlevel > powerthreshold ) then {
9           choose {
10            via measurement::sense receive data
11            via measurement::measure send tuple{
12               coordinate = lps, depth = data::convert( ) }
13          } or {
14             via measurement::pass receive data
15             via measurement::measure send data
16          } //end choose
17      } // end if
18  } //end repeat
19 } //end sensing
```

Listing 4: Code in SosADL for a behavior called *sensing* of a smart sensor.

Within the scope of the conditional statement, a `choose` statement (Lines 9 to 16) is triggered if the condition returns a `true` value. System will behave depending on the stimulus received: if it receives a data delivered by its own sensor, Line 10 is executed, and the data is received via a connection called `sense` at the gate called `measurement`. This data is forwarded to the closest mediator towards the gateway via the connection `measure` (Lines 11-12). If the sensor receives a data collected by another sensor and transmitted across the SoS, Line 14 is executed. The data are received via a connection called `pass` at the gate measurement and forwarded via the connection called `measure` at the same gate (Line 15).

A mapping from a SosADL behavior for a DEVS code produces a state diagram. To illustrate this procedure, we depict a state diagram in Figure 1 that corresponds to the behavior `sensing` in Listing 4. A line of code in SosADL (henceforth, a SosADL statement) can be converted to one or more state transitions in a labeled state diagram to produce a DEVS code. Line 2, for example, is directly mapped into one transition between two states ($s_0 \Rightarrow s_1$) (Figure 1). Line 4 expresses a `receive` assignment, with an analogous treatment ($s_2 \Rightarrow s_3$). This is an output event, and as such, originates a labeled output transition ($s_1 \Rightarrow s_2$). The other one-line statements (we term them `action` statements) in line 4, lines 10 and 11, and 14 and 15 follow the same rationale.

As every statement within a `repeat` command must be repeated, a new state called $\mathtt{s}_n$ is created for amalgamating the execution flow at the end of the loop, and transiting it from $\mathtt{s}_n$ to $\mathtt{s}_3$, creating a loop. Line

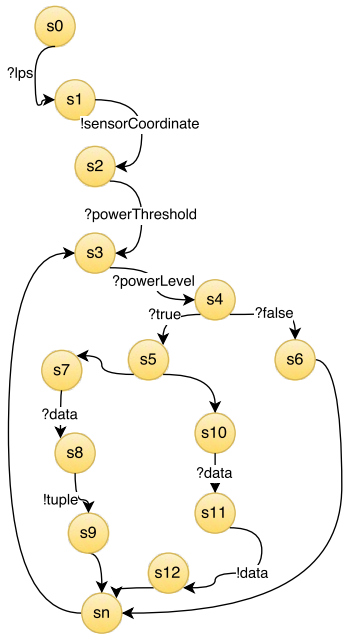


Figure 1: A labeled state diagram corresponding to a *sensing* behavior extracted from SosADL code.

7 depicts the transition `?powerLevel`. Two transitions are created: one for a `true` value returned by the statement, and another for `false` values, i.e., when the battery level is not enough to perform the behavior. Transition to $\mathtt{s}_5$ is created for `?true`, and the transition to $\mathtt{s}_6$ is created for `?false`. Since no `else` statement exists, the flow is forwarded to $\mathtt{s}_n$ spontaneously, i.e., without receiving or sending anything, converging the execution flow. The same procedure is followed for the transformation of simple assignments, and two execution flows are created and chosen depending on which event occurs first. Two flows are created for each set of statements, and finally, the flows are amalgamated again by spontaneous transitions in a state previously created by the `repeat` statement ($\mathtt{s}_n$), and the transformation for the behavior is finalized. A transition ($\mathtt{s}_n$ to $\mathtt{s}_3$) restarts the execution.

```
passivate in s0!
when in s0 and receive lps go to s1!
hold in s1 for time 1!
after s1 output sensorCoordinate!
from s1 go to s2!
when in s2 and receive powerThreshold go to s3!
from sn go to s3!
when in s3 and receive powerLevel go to s4!
when in s4 and receive true go to s5!
when in s5 and receive dataSensed go to s8!
hold in s8 for time 1!
after s8 output tuple!
from s8 go to s9!
from s9 go to sn!
when in s5 and receive dataPassed go to s11!
```

```
16 hold in s11 for time 1!
17 after s11 output data!
18 from s11 go to s12!
19 from s12 go to sn!
20 when in s4 and receive false go to s6!
21 from s6 go to sn!
```

Listing 5: State diagram code corresponding to a Smart sensor behavior generated in DEVSNL.

Listing 5 shows the mapping of the `sensing` behavior of Listing 4 and its equivalent state diagram in Figure 1 into DEVSNL. Each transition produces a transition in DEVSNL. Input transitions correspond to one line of code such as `when in s0 and receive lps go to s1!`. Output transitions produce three lines of code, of which one holds the flow in the state for a pre-defined time, another produces the output, and a final one produces the transition itself, as in Lines 2, 3, and 4 in Listing 5. Finally, transitions created only for moving the flow from one state to another produce one line in DEVSNL, as Line 19 in Listing 5. Transitions `s5 to s7` and `s5 to s10` are substituted during the transformation to DEVSNL by `s5 to s8` and `s5 to s11`, respectively, for the avoidance of non-determinism.

1.1. Patterns for SoS Simulation

ASAS is based on simulations, as they comprise a recurrent practice within the SoS development life cycle [1]. Simulations imitate the environment and the operation of a real-world process or system over time through the generation of artificial stimuli and observation of effects for drawing inferences concerning the operational characteristics of the real system represented [2]. They also enable the prediction and correction of errors, supporting the observation of expected and unexpected emergent behaviors of an SoS. Nevertheless, simulation is also software that relies on dozens of lines of code and demands techniques for its engineering. Simulation codes are often driven by labeled state machines based on discrete input and output events, as Discrete Event Systems (DEVS) [3]. However, some of the instructions are conflicting, and the number of lines of code can be huge, which leads to difficulties and high costs for production and maintainability. Such state machines can reach large dimensions, which makes the production of codes repetitive and error-prone. Under this perspective, the identification of patterns can aid the conception of simulations for SoS, supporting the automatic generation of codes from specifications in a high level of abstraction, such as architectural specifications of software of SoS.

Patterns are standard solutions for recurrent problems that emerge in a domain [4]. As they enable the reuse of solutions, they can (i) contribute to the trustworthiness expected from SoS, once the reuse of a well-succeeded solution can foster the construction of the correct product, and (ii) reduce costs and time. Since simulation has become increasingly dominant and used by various industries, techniques and methods must be designed for the effectiveness of developers [5]. However, the literature lacks techniques and software

engineering methods for simulations in SoS context. SoSE is a novel discipline, and general principles and patterns still must be discovered[1]. DEVS (Discrete Event System Specification) is a well recognized formalism for the specification and execution of SoS simulations. In DEVS, constituents operation are specified via a labeled state machine, i.e., a state machine in which transitions occur due to data input or output, or time elapsed. DEVS variants include probabilistic, non-deterministic, and finite deterministic ones. As non-determinism is unfeasible, deterministic DEVS versions are more common on platforms, such as FD-DEVS (Finite Deterministic DEVS), implemented in platforms as MS4ME[2].

In DEVS, a constituent system is driven by a state machine specified according to a well-defined set of primitives. A state in DEVS can be either a 'hold state', or a 'passive state' (exclusively). The execution flow is maintained at a hold state for a certain amount of time until it is automatically changed to another state (via an internal transition). On the other hand, in a passivate state, simulation remains indefinitely (or until it receives a message that triggers an external transition). Below are the basic constructs for a state machine in DEVS:

Passivate States (PS). States in which the execution flow remains until some event (input or output) has caused a transition to another state.

```
passivate in STATENAME!
```

Hold States (HS). States in which the execution flow remains for a well-defined time, such as 5 seconds.

```
hold in STATENAME for time 5!
```

Initial State (IS). State marked as initial that starts the simulation execution. A simulation holds only one initial state, which can be specified by using the following syntax:

```
to start passivate in STATENAME!
or
to start hold in STATENAME for time 5!
```

Internal Transition (IT). A simple transition that specifies the current state and a next state. Every HS must exhibit one internal transition.

```
from FROMSTATE go to TOSTATE!
```

Output Transition (OT). A transition that produces an output of a message prior to the internal transition.

```
after STATENAME output OUTPUTMESSAGE!
```

[1] J. Fitzgerald, S. Foster, C. Ingram, P. G. Larsen, and J. Woodcock. Model-based engineering for systems of systems: the COMPASS manifesto. COMPASS, October 2013.

[2] http://www.ms4systems.com/pages/main.php

External Transition (ET). A transition that defines an input message the model might receive. Such a message causes a transition from a state to another. Both states and the expected input message must be specified. Any state can have one or more external transitions defined. The syntax is:

```
when in FROMSTATE and receive INPUTMESSAGE go to TOSTATE!
```

The code of a state diagram in DEVS is based on inputs and outputs and consists of an arrangement of statements that guide the operation of a constituent. However, System Engineering guides usually do not guide how to group those statements conveniently to form no conflicting input and output transitions.

We have established one pattern for input and another one for output. The context is the same for both (DEVS simulation models), and recurrent problem changes only in function of the purpose, i.e., input or output. The former prescribes an input causes a transition when, at some state, it receives a datum, whereas, according to the latter, once an output occurs spontaneously (without any triggering event), it should remain in that state for one second (this time can be specified according to convenience), perform the output, and transit to the next state.

Figure 2 depicts our patterns expressed as UML class diagrams. DEVS Input rules specify a `PassivateRule` that passivates in one and only one state, whose name is represented by a label. From this state, an `InputTransition` occurs when a pre-determined type of data is received and causes the transition from one state to one and only one another state. DEVS Output rules specify a `HoldRule` that holds in one and only one state for a pre-determined amount of time. From that state, an output event occurs for the delivery of some data, and the execution flow transits to another state.

Statement/ Statement	PS	HS	IS	IT	OT	ET
HS	X					
IS						
IT	X					X
OT	X					X

Table 1: Conflicts and compatible instructions in DEVS.

Table 1 shows the potential conflicts identified:

- **HS-PS**: A hold state cannot be concomitantly a passivate state, and vice versa. Specification of a state as hold and passivate might cause a conflict in a state, as such state should, at the same time, wait for some pre-specified time and also wait indefinitely. Therefore, such combination is unfeasible;
- **IT-PS**: Internal transitions are automatically triggered after a given portion of time, e.g., after one second, go to another state. As such, a state should not be specified as a passivate state together with a specification of an internal transition, as it might cause a spontaneous transition at any moment;
- **IT-ET**: An internal transition cannot be concomitantly an external transition, and vice versa, since they have a different nature;

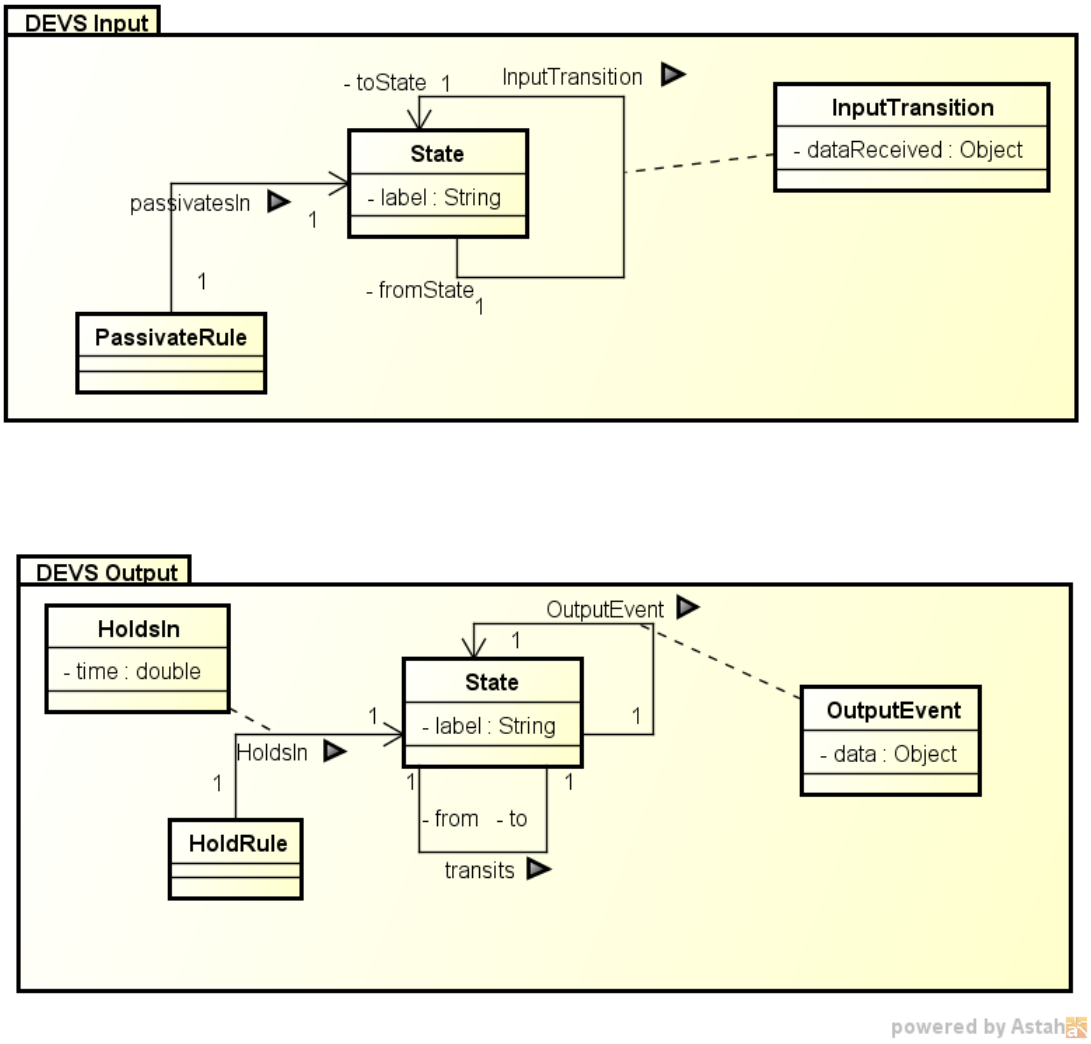


Figure 2: Patterns expressed as class diagrams in UML.

- **OT-PS**: Technically, an output transition is an internal transition. Since a passive state cannot have an internal transition, it cannot have an output transition.

- **OT-ET**: Since an internal transition cannot be an external transition it cannot be an output transition.

Such conflicts must be taken into consideration in the design of robust SoS simulations. Therefore, two patterns have emerged to encapsulate such principles: one that groups codes representing input transitions and their sub-activities involved, and another representing output transitions. Their establishment relied on the classical Gamma's structure [4], i.e., recurrent problem, context, and solution. Since DEVS is based on inputs and outputs, we have established one pattern for input (Table 2) and another for output (Table 3). The context is the same for both (DEVS simulation models) and recurrent problem changes only in function of the purpose, i.e., input or output.

Name	DEVS Input
Recurrent Problem	Specification of a set of simulation instructions that characterize an input event without conflicts with other instructions.

Solution	`passivate in s<<fromState>>!`
	`when in s<<fromState>> and receive <<`
	`  dataReceived>>`
	`go to s<<toState>>!`

Table 2: Patterns for input in DEVS simulation models.

Name	DEVS Output
Recurrent Problem	Specification of a set of simulation instructions that characterize an output event without conflicts with other instructions.

Solution	`hold in s<<fromState>> for time 1!`
	`after s<<fromState>> output <<dataType>>!`
	`from s<<fromState>> go to s<<toState>>!`

Table 3: Output pattern for DEVS simulation models.

DEVS Input

According to the DEVS Input pattern, an input will cause a transition when, at some state, it receives a datum. If `passivate` comes after input instruction `when`, it might cause a conflict with a `hold` of a following output instruction.

DEVS Input (Figure 2) specifies a `PassivateRule` that passivates in one and only one state, whose name is represented by a label. From this state, an `InputTransition` occurs when a pre-determined type of data is received and causes the transition to another state.

DEVS Output

Once an output occurs spontaneously (with no triggering event), the execution flow should remain in a state for one second (this time can be specified according to convenience), perform the output, and transit to the next state. If the next state causes an input, it will be subject to either a `passivate` instruction, or a new `hold`.

DEVS Output (Figure 2) specifies a `HoldRule` that holds in one and only one state for a pre-determined amount of time. From that state, an output event occurs, delivering some data, and transiting from one state to another state.

1.2. Generation of Coupled Models

In ASAS approach, a SoS architectural description specified in SosADL is verified against its metamodel expressed in Xtext[3] during the transformation. If the SosADL code conforms to this metamodel, the code is used as input to an Xtend[4] script that transforms it in a DEVS model. Coupled models in DEVS specify the way constituent systems exchange data with each other to exhibit an emergent behavior. The code of such coupled models systematically specifies the entities involved in the SoS and the way they interact, i.e.,the systems that send data and those that receive them. In SosADL, SoS software architectures are modeled as coalitions. The correspondences between SosADL and DEVS are summarized in Table 4.

Table 4: Mapping of SosADL into SES/DEVS.

SoS concept	**Representation in SosADL**	**SES/DEVS**
Constituent Systems	Coalition	Decomposition
Data Types	Data Type	Data Type
Gate/Connection	Gate/Connection	DEVS Port
Interfaces	Binding	Coupling
SoS Architecture	Coalition + Binding	Coupled Model

Constituent Systems. In SosADL, the list of all constituent systems that compose the software architecture of an SoS is represented by a Coalition. By definition, coalitions are alliances of constituents connected via mediators. When translated into DEVS, coalitions are mapped into a DEVS Decomposition, i.e., a statement of the coupled model that systematically lists all inner structures (e.g., systems, mediators, among others) that form the software architecture of the SoS [3].

Data Types. When a communication is established between constituents and they start to interoperate, data are exchanged between them. Indeed, SoSADL relies on *typed connections*, i.e., connections with a specific type of data. Data types must be preserved by the transformation and properly converted into DEVS format.

Gate/Connection. Gates are a structure through which connections can be established. Since the notion of connection does not exist in DEVS, each SosADL Connection is mapped as a port in DEVS.

Interfaces. The concept of interface encapsulates the communication between two entities (in this case, systems). Consequently, a detailed analysis of an interface should contain (explicitly or not) functions *send* located in one system element, and *receive* located in another [5]. In SosADL, interfaces are specified through bindings, which correspond to the list of all combinations between output ports and input ports that establish

[3]https://eclipse.org/Xtext/
[4]xtend-lang.org/
[5]SEBoK. Guide to the systems engineering body of knowledge, version 1.6, 2016.

a communication between two entities in a SoS. In DEVS, each binding is mapped into a coupling, i.e., a statement describes the way information flows between two systems in the SoS.

Sos Architecture. Finally, the software architecture of an SoS is represented as an abstract architecture in SosADL, which specifies a coalition and a set of bindings, and subsequently mapped in a coupled model, which is a set containing a decomposition and couplings.

Listing 6 depicts a SosADL code that represents the specification of a software architecture of an FMSoS. In The software architecture of SoS represented comprises four `sensors`, one `gateway`, and four `transmitters` (types of mediators) (Lines 4 to 12). `bindings` (Lines 13 to 23) represent the way connections between constituents and mediators are established through gates, and SoS dynamics for data transmission to a gateway. A sensor collects the `water level` through actuators, encapsulates it with the specific `location` in which the collecting was performed, and a time stamp. The sensor then transmits the data to the closest mediator, which forwards them to the next sensor, until the gateway has been reached.

```
1 sos FloodMonitoringSos is {
2   architecture FloodMonitoringSosArchitecture( ) is{
3     behavior coalition is compose {
4     sensor1 is Sensor
5     sensor2 is Sensor
6     sensor3 is Sensor
7     sensor4 is Sensor
8     gateway is Gateway
9     mediator1 is Mediator
10    mediator2 is Mediator
11    mediator3 is Mediator
12    mediator4 is Mediator
13    } binding {
14    relay gateway::notification::alert to warning::alert and
15    relay gateway::request to request and
16    unify one { sensor1::measurement::measure }
17       to one { mediator1::fromSensors } and
18    unify one { mediator1::transmit::towardsGateway }
19       to one { sensor2::measurement::pass } and
20    unify one { sensor2::measurement::measure }
21       to one { mediator2::fromSensors } and
22    unify one { mediator2::transmit::towardsGateway }
23       to one { gateway::notification::measure } and
24    unify one { sensor3::measurement::measure }
25       to one { mediator3::fromSensors} and
26    unify one { mediator3::transmit::towardsGateway }
27       to one { sensor4::measurement::pass } and
28    unify one { sensor4::measurement::measure }
29       to one { mediator4::fromSensors} and
30    unify one { mediator4::transmit::towardsGateway }
31       to one { gateway::notification::measure }
```

```
 1 From the top perspective, FloodMonitoringSosArchitecture is made of
 2 Sensor1, Sensor2, Sensor3, Sensor4, Gateway, Mediator1, Mediator2, Mediator3,
 3 and Mediator4!
 4
 5 From the top perspective,  Sensor1 sends Measure to Mediator1!
 6 From the top perspective,  Mediator1 sends Measure to Sensor2!
 7 From the top perspective,  Sensor2 sends Measure to Mediator2!
 8 From the top perspective,  Mediator2 sends Measure to Gateway!
 9 From the top perspective,  Sensor3 sends Measure to Mediator3!
10 From the top perspective,  Mediator3 sends Measure to Sensor4!
11 From the top perspective,  Sensor4 sends Measure to Mediator4!
12 From the top perspective,  Mediator4 sends Measure to Gateway!
```

Listing 7: Coupled model for FMSoS generated in DEVS.

```
32 }  }
```

Listing 6: Description of an architecture of an FMSoS in SosADL.

In SosADL, a connection is specified as `system :: gate :: connection`. Indeed, the same gate can hold one or more connections. A unification is established for each pair of sensors with a mediator between them by a `unify` statement (Lines 15-23). According to such statements, an output connection `measure` from the gate `measurement` is linked to the input connection `fromSensors` of the closest mediator. A mediator gathers data from a sensor (Lines 11 to 14) and forwards them to the next sensor. Mediators have an output connection termed as `towardsGateway`. Such connections are linked to the sensors through an input connection called `pass` in the gate `measurement` to receive the data transmitted and forward them to the gateway (Lines 16, 18, 20 and 22). Lines 22 and 23 link the output connection of the mediator to the gateway connection called `measure`. In this case, a mediator mediates a constituent and the gateway. The `relay` statement establishes the communication between the SoS and external systems, connecting the notification gate of a gateway to an external connection. Each binding specified in SosADL is mapped into a coupling in DEVS. Listing 7 shows the equivalent code derived from the coalition according to the transformation rules. Line 1 in Listing 7 shows the `FloodMonitoringSoSArchitecture` formed by the same systems specified in the SosADL code. Lines 2 to 9 show the data exchange among all systems and mediators derived from the specification of the coalition. These lines are created by iterating on the unifying statements. A line is created for each unifying connection. Finally, DEVS tool converts that code into an executable simulation model.

According to Listing 7, sensors transmit data to their closest mediator (Lines 2, 4, 6, and 8), which receives them in Lines 3, 5, 7, and 9 forwards `Measure` to the next sensors. Since `Sensor2` and `Sensor4` send their data to `Mediator2` and `Mediator4`, respectively (Lines 4 and 6), the gateway is reached (Lines 5 and 9). When data arrive in the gateway, their values are tested against a pre-determined depth threshold. If they are higher, the gateway emits a flood alert. Therefore, the network of exchanged messages between constituents and the flood alert trigger indicate the SoS mission has been accomplished.

1.3. A Mechanism for Supporting SoS Dynamic Architectures

SoS architectures are inherently dynamic. Therefore, coupled models generated by the model transformation must provide strategies for dealing with changes that may occur in the software structure. Such changes must be well-defined and the final result must be the same whenever a change is performed, as a change should conduct the SoS to a new functional state. The literature reports well-established architectural changes for single systems software architectures [6]. However, when multiple interoperable systems that form a SoS are considered, a gap still must be bridged. This section presents a canonical set of dynamic changes that can affect the SoS software architectures and well-defined steps for their execution.

Dynamic reconfiguration in a SoS is based on four types of architectural changes, namely addition of constituent, deletion of a constituent, substitution of the constituent, and reorganization of the architecture. They are invoked at simulation time by the simulation user and executed by a reconfiguration controller (further explained), which will perform the necessary changes in the architecture.

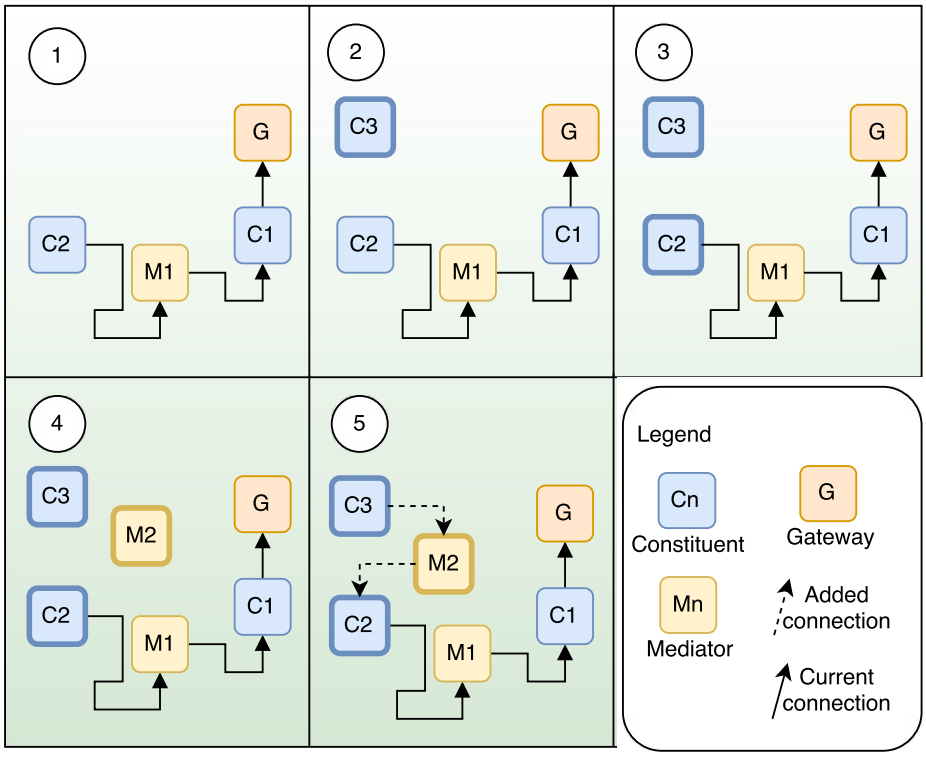


Figure 3: Illustraton of addition of a constituent in a simulation of SoS software architecture.

Addition. Addition of a constituent into the simulation of a SoS software architecture depends on a well-defined set of steps. Firstly, a new constituent must be created and added to the simulation. It must be linked to the SoS for effectively joining it. A constituent is selected in the architecture to be linked to the new constituent. Then an appropriate mediator is instantiated and added to the simulation. Connections

between the constituents and mediator are then established. The following steps are performed, as shown in Figure 3:

1. The SoS simulation user issues a request to add a particular constituent to a reconfiguration controller. Scene 1 represents the state of SoS at that time. The constituent to be added is C3;
2. Constituent C3 is instantiated and added to the simulation;
3. The controller selects another constituent the simulation to be connected to C3 (in this case, C2);
4. The controller instantiates and adds an appropriate mediator M2 to the simulation;
5. The controller adds connections between C2 and C3, which are then mediated by M2.

Deletion. The deletion process is initiated by the user who explicitly chooses a constituent to be removed. Such a constituent sends a unique identifier to the controller, and constituents connected to it are listed. These connections are eliminated and mediators are removed. The controller searches for constituents in the architecture that can be connected to the remaining ones. This important step maintains the SoS operation in progress. At the end of the process, the controller removes the constituent from the simulation, as described in the steps below and illustrated in Figure 4.

1. The SoS simulation user issues a deletion request for a particular constituent. Scene 1 represents the state of SoS at that time. The constituent chosen to be deleted is C1;
2. The connection between constituent 1 and the mediator that connects it with the remainder of the SoS is undone;
3. The binding between mediator 1 and the remainder of the SoS is undone and mediator 1 is selected to be deleted; and
4. Both (C1 and M1) are deleted.

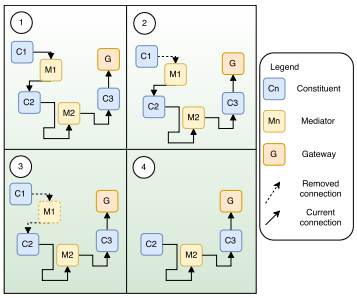


Figure 4: Process of deletion of constituent in simulation of a SoS software architecture.

Substitution. Substitution is performed as a concatenation of the deletion and addition operators. Steps are followed as shown in Figure 5

1. The SoS simulation user issues a request for the removal of constituent C1;

2. The connection between constituent 1 and the mediator that connects it with the remainder of the SoS is undone;
3. The connection between mediator 1 and the remainder of the SoS is undone, mediator 1 is selected to be eliminated and C2 is chosen to connect C3;
4. Both (C1 and M1) are eliminated, and the user adds the constituent C3 which is instantiated and added to the simulation;
5. The controller instantiates and adds a compatible mediator (M2) to the simulation;
6. The controller adds the necessary connections to connect C3 with C2. M2 starts to mediate them.

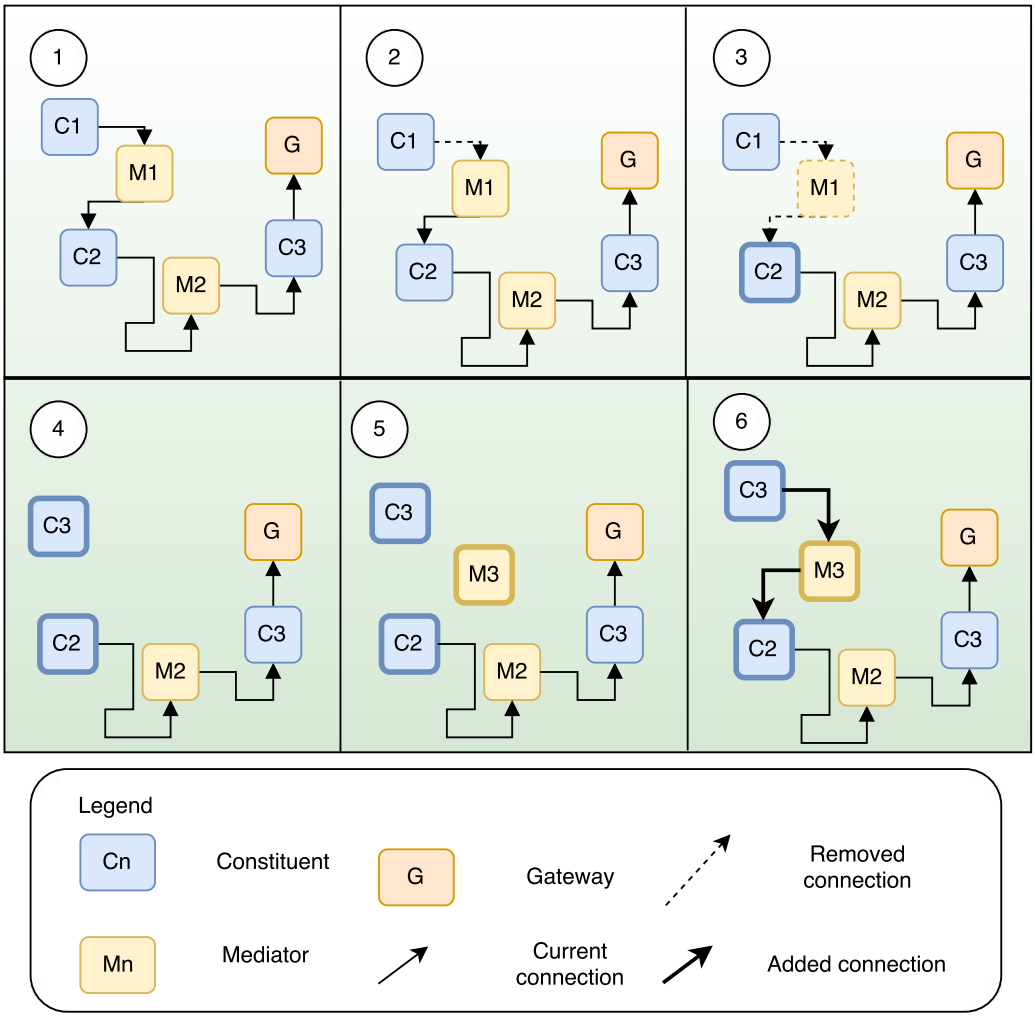


Figure 5: Process of substitution of constituent in the simulation of a SoS.

Reorganization. Architectural reorganization process consists in the complete dissolution of the architectural configuration, and reestablishment of new connections between the constituents, which leads to a new operational state of SoS, as shown in Figure 6. The controller removes all connections between the con-

stituents and all mediators, except connections between the controller and the constituents. The controller arbitrarily chooses a constituent in the simulation (random process) and tries to establish its connections with other constituents. The process is repeated until all elements have been connected in SoS, which results in an architecture configuration different from the original one.

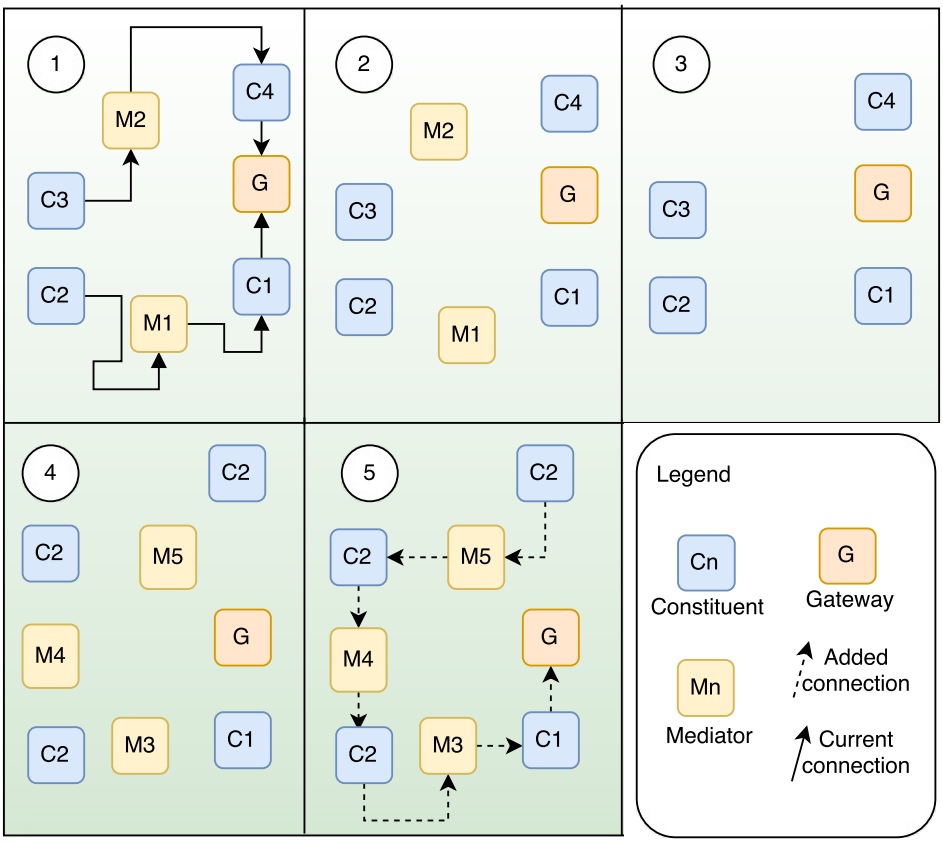


Figure 6: Process of reorganization of architecture in the simulation of a SoS software architecture.

The SoS dynamic architecture is managed at runtime by a *Dynamic reconfiguration controller* (DRC). Figure 7 shows the way DRC interacts with the SoS architecture simulated. DRC is an artificial architectural element that manages every architectural change that occurs. It is added to the simulation to enable the simulation user to perform architectural changes at runtime. From the DEVS simulation model perspective, the reconfiguration controller is an atomic model that (i) adds constituents to the simulation, and the necessary connections and mediators, and relinks the properties of the initial architecture; (ii) removes the constituents of the simulation, connections and mediators, relinking the remaining constituents for maintaining an operational SoS architecture; (iii) removes the constituent and replaces it for another one; and (iv) reorganizes the architecture by removing all connections and mediators and thereafter establishing different mediated connections for creating a new architectural configuration while retaining the initial architectural properties.

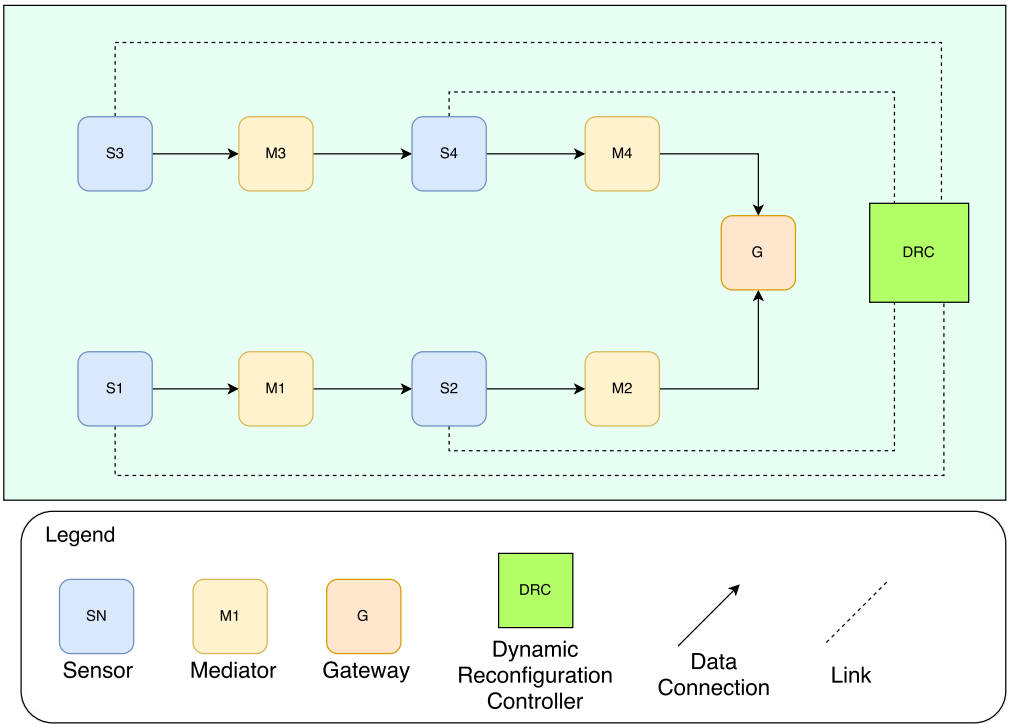


Figure 7: An illustration of the relation between DRC and constituents simulated.

Acknowledgement

Authors thank São Paulo Research Foundation (FAPESP) by the financial support under grants 2014/02244-7, 2017/06195-9, and 2017/17448-5.

References

[1] V. V. Graciano Neto, M. Guessi, L. B. R. Oliveira, F. Oquendo, E. Y. Nakagawa, Investigating the model-driven development for systems-of-systems, in: 8th European Conference on Software Architecture Workshops (ECSAW 2014), ACM, Vienna, Austria, 2014, pp. 22:1–22:8.

[2] J. Banks, Introduction to simulation, in: 31st Conference on Winter Simulation: Simulation - a Bridge to the Future - Volume 1 (WSC 1999), ACM, Phoenix, Arizona, USA, 1999, pp. 7–13.

[3] B. P. Zeigler, H. S. Sarjoughian, R. Duboz, J.-C. Souli, Guide to Modeling and Simulation of Systems of Systems, Springer-Verlag London, London, United Kingdom, 2012.

[4] J. Vlissides, R. Helm, R. Johnson, E. Gamma, Design patterns: Elements of reusable object-oriented software, Reading: Addison-Wesley 49 (120) (1995) 11.

[5] J. Gray, B. Rumpe, Models in simulation, Software & Systems Modeling 15 (3) (2016) 605–607.

[6] E. Cavalcante, T. V. Batista, F. Oquendo, Supporting dynamic software architectures: From architectural description to implementation, in: 12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015), IEEE, Montreal, QC, Canada, 2015, pp. 31–40.