RESEARCH

Stimuli-SoS: A Model-Based Approach to Derive Stimuli Generators in Simulations of Systems-of-Systems Software Architectures

Supplementary Material

Full list of author information is available at the end of the article

Listings

Listing 1 shows an excerpt of a code in SosADL that depicts part of the specification of one of the sensors that composes FMSoS. Some parts are hidden because they do not influence in the derivation of stimuli generation. It is possible to see that the gate energy offers two environment connections (Lines 12 and 13): one to receive a threshold (a limit of energy that is considered enough to keep the sensor in operation), and `power`, which is used to receive the level of battery available.

Listing 1 A specification of a Sensor in SosADL.

```
1  //'with' imports declarations suppressed
2  // Description of Sensor as a System Abstraction
3  library WsnSensor is {
4    system Sensor( lps:Coordinate ) is {
5    // Declaration of local types hidden
6    gate measurement is {
7        connection pass is in { MeasureData }
8        environment connection sense is out { MeasureData }
9    }
10
11      gate energy is {
12        connection threshold is in { Energy }
13        environment connection powerLevel is in { Energy }
14      }
15     }
16 }
```

Model Transformation

All SosADL aspects must be traduced to DEVSNL to create a functional simulation. In SosADL, there is a special type of connection called **environment**, which abstracts interaction of an SoS with the surrounding environment, emitting outputs to the environment, or receiving stimuli from it, e.g., when the system is a sensor. However, there are no straightforward elements in DEVSNL to automatically produce environment stimuli. Thus, it is necessary to create a stimuli generator that delivers the expected inputs the constituents wait to perform transitions and to start their execution. The following steps are followed by the transformation chain that generates stimuli generators:

1 Constituents, Mediators, and Gateways are firstly analyzed. All connections are mapped into a specification format and saved in a text file. For convention, constituents are analyzed first, as they are the frontier of a SoS; besides, state transitions for the stimuli generator are created to serve their necessities. After that, mediators are investigated. Lastly, gateways are analyzed. Considering that constituents usually starts the SoS operation due to the stimuli they receive from environment, we prioritize the analysis of constituents. If there are congruent statements between constituents and mediators, i.e., a same type of stimulus that must be received for both, a unique transition is created in the stimulus generator.

2 Connections are read from the text file and analyzed. Such connections were parsed from the architectural description of the SoS to be in the following format: `measurement::sense;RawData-true`. This first part is the name of the gate in which the connection has been specified. The second part is the name of the connection. The third part represents the data type that can be transferred across that communication channel. The last part of each connections descriptions is a boolean: it has a `true` value if the connection is of the type `environment` and `false` if it is not. The transformation algorithm searches for environment connections. Each connection specified as an `environment` connection produces one transition in the specification of the state diagram in the resulting stimuli generator. Hence, the stimuli generator consists of a special type of system (in the context of the simulation) that has a continuous behavior (a behavior materialized as a loop) to emit stimulus by output state transitions, starting and keeping the SoS in operation.

Listings 2, 3, and 4 detail procedures to generate a stimuli generator. In DEVS, such stimuli generator is represented as an Atomic Model. MS4ME environment automatically produces an equivalent Java code for the entire specification of the SoS (including constituents, mediators, and the architecture itself). This Java code is used to drive the simulation.

In Listing 2, an Xtend method called `deriveStimuliGeneratorFor` was created to modularize such generation within the context of the model transformation that transforms the entire SoS specified in SosADL into simulation code specified in DEVSNL. From lines 1 to 15, variables are defined to support operations performed as follows in the code. It is possible to see a file reader being instantiated to receive input data from an external text file. Still in Listing 2, Lines 16-20 show an excerpt of code that will be inserted in the resulting DEVS source code. It corresponds to the part of the DEVSNL functional code that initializes variables. Two random numbers (from 1 to 10) are created for purposes of testing. These numbers can be substituted by data provided from the external files with pre-defined inputs. After that, each line of the files that stores connections metadata will be read line per line, splitting the data according to the pre-defined format (lines 22-35). If a connection has a type `environment` (lines 37-54), one output port is created to deliver that type of data that the connection transports (line 40), a state transition is created in such that every time that such transition occurs, it will trigger an event that sends that type of data by the respective port (lines 42-47), and the respective output event that is triggered when such transition occurs (lines 51-56).

Listing 2 Transformation to derive a stimuli generator - Part I.

```
def deriveStimuliGeneratorFor(SystemDecl constituent) {
      var File file = new File(fileAddress);
    var BufferedReader  reader = null;
    fileAddressStimuli = fileAddressStimuli + "StimuliGenerator"
         + constituent.name.toFirstUpper + ".dnl"

    var String data;
    var String text = "";
    var String initializeVariables = "";
    var String datatypes = "";
    var String events = "";
    var String ports = "";
    var String statemachine = "to start hold in s0 for time 2!\n
        ";
    var int countStates = 0;

    initializeVariables += "Initialize variables
        <%\n
         java.util.Random generator = new java.util.Random();
         int number1 = generator.nextInt(10);\n
         int number2 = generator.nextInt(10);\n"

    try {
      reader = new BufferedReader(new FileReader(file));
           while ((text = reader.readLine()) != null) {
        try
        {
        val String[] vectorConnections = text.split("::")
            val String[] vectorAux = vectorConnections.get(1).
                split(";")
          var String connectionName = vectorAux.get(0).replace("
              ","")
          var String[] vectorDataTypeAndEnvironment = vectorAux.
              get(1).split("-")
          var String dataType = vectorDataTypeAndEnvironment.get
              (0)
          var boolean environmentalConnection = Boolean.valueOf(
              vectorDataTypeAndEnvironment.get(1))

          if(environmentalConnection)
          {
            /** Ports */
            ports += "generates output on " + connectionName.
                toFirstUpper + " with type " + dataType.
                toFirstUpper + "!\n"


            /** State Transitions */
            statemachine += "after s" + countStates + " output "
                 + connectionName.toFirstUpper + "!\n"
            statemachine += "from s" + countStates + " go to s"
                + (countStates+1) + "!\n"
            statemachine += "hold in s" + (countStates+1) + "
                for time 1!\n"
            countStates++

            /** Respective Output event */
            events += "output event for s" + countStates + "\n"
          events += "<%" + "\n"
          events += " output.add(out" + connectionName.
              toFirstUpper + ", " + dataType.toFirstLower + ");\
              n"
          events += "%>!\n\n"
          }
        } catch (IndexOutOfBoundsException e) {
          System.err.println(
            "Problems with the format of the data saved in '
                connections' file text.")
        }
      }
```

Listing 3 Transformation to derive a stimuli generator - Part II.

```
1           var EList<DataTypeDecl> dataTypeList = constituent.
                datatypes
2
3             for(d: dataTypeList)
4             {
5               var DevsType t
6                 if (d.datatype == null) {
7                   if(DEBUG2) System.err.println("Warning! type
                        definition of '" + d.name +
8                     "' null! Assuming 'integer'...")
9                   t = new DevsIntType()
10                    } else {
11                  t = computeDevsType(d.datatype)
12                }
13
14                if (d.datatype == null)
15                {
16                  datatypes += "\nA " + d.name  + " has a value!\n
                        "
17                  datatypes += "the range of " + d.name + "\'s
                        value is Integer!\n"
18                  datatypes += "use " + d.name.toFirstLower + "
                        with type " + d.name + "!\n\n"
19
20                  initializeVariables += '''<<d.name.toFirstLower
                        >> = new <<d.name>>(new Integer(0));'''
21
22
23                } else {
24
25                  if ((d.datatype as DataType) instanceof
                        TupleType) {
26
27                  datatypes += "\n" + d.name.toFirstUpper + " has
                        " + d.datatype.compile
28                  datatypes += '''<<var e = (d.datatype as
                        TupleType)>>><<FOR p : e.fields>>
29                  the range of  <<d.name>>'s <<p.name>> is <<(p as
                         FieldDecl).type.compile>>!<<ENDFOR>>'''
30                  datatypes += "\n use " + d.name.toFirstLower + "
                         with type " + d.name + "!\n"
31
32                  }
33
34                var teste = '''<<d.datatype.computeDevsType>>'''
35
36            if (teste.contains("sequence")) {
37
38              teste = teste.replace("sequence{", "")
39              teste = teste.replace("}", "")
40              initializeVariables += '''<<d.name.toFirstLower>> =
                    new java.util.ArrayList<<<teste>>>();
41              '''
42            } else {
```

Listing 3 details how the data types are handled from the SosADL specification to generate DEVS types. It is important to highlight that data types in SosADL can be of the following types: integer, boolean, tuple, sequence, range, and connection. Table 1 shows how SosADL data types are mapped into DEVS data types. Booleans and integers remain the same data type in DEVS. SosADL connections are mapped into ports, and the concepts of gates disappears. Sequences become a Java ArrayList, whilst tuples, that is, data types composed of fields that can be of any another data type aforementioned become abstract data types in DEVS, that is, data types composed of attributes that can be of other data types.

SoSADL Data Type	DEVS Data Type
Boolean	Boolean
Connection	Port
Integer	Integer
Sequence	Array List
Tuple	Abstract Data Type

Table 1 Mapping between data types of SosADL and DEVS.

The method showed in Listing 3 is executed for each one of the parts that constitute the software architecture of SoS in SoSADL. Line 1 extracts the list of attributes that the constituent being processed now holds. Lines 3-46 process each data type that exists in SosADL. From lines 5 to 12, we deal with raw data types, i.e., data types that are not abstract type of data because they do not have any attribute. They only have a type and a value associated. By convention, all these data types are mapped into integers in DEVS. Lines 5-12 shows the computation of the DEVS Type for syntax processing. However, the resulting source code that will be inserted in the resulting stimuli generator body is created in lines 14-20, which create exactly the excerpts of code showed in Listing **??** corresponding to the lines to create a data type.

Still in Listing 3, if the data type is an abstract data type, defined with attributes, a different type of treatment is done, iterating over its list of attributes, mapping each one of the types in a correspondent DEVS Type.

Sequences in SosADL are mapped into linked lists in DEVS (lines 42-45 in Listing 3). In turn, lines 1-29 in Listing 4 take care of creating the resulting code for DEVSNL that declares each data type that is considered a tuple, that is, an abstract type of data composed and characterized by attributes. Lines 31-43 handle exceptions; and Lines 44-70 organize how the final source file will be written by the transformation, generating the output, creating a functional stimuli generator materialized as an atomic model specified in DEVSNL.

Listing 5 shows the code in DEVSNL that specifies the stimuli generator produced using our approach. Such stimuli generator has data types derived from the specification of the data in the source file (Lines 1-26). After that, a set of three output ports (Lines 28-30) delivers the power level, the threshold, and a data that represents the water level that will be sensed by sensors. Output events that governs the specific data that will be delivered by each specific output port in each transition are specified between lines 32 and 56. An excerpt of code to initialize useful variables is depicted between lines 58 and 66. Lastly, the state diagram that governs the operation of the stimuli generator is showed in lines 68 to 76. The state diagram delivers the aforementioned data, and comes back to the state `s2`, forming a loop that keeps the stimulus continuously running and offering the stimulus for the operation of the SoS. As DEVSNL is a declarative language, the order in which these blocks appears will not influence in the final operation of the generator. We evaluate our approach as follows.

After the automatic derivation, the stimuli generator must be deployed in the simulation code specified in DEVSNL and already deployed in MS4ME environment. This step consists of moving the stimuli generator file to the atomic models directory within the simulation project in MS4ME environment. MS4ME environment automatically generates a Java file that corresponds to the execution entity of the

stimuli generator. This file drives the execution of the stimuli generator during the simulation execution. Besides that, it is necessary to adapt the coupled model code, specifying that the dynamics of the SoS includes new atomic models (materialized by the stimuli generators), and that these generators must emit data to their correspondent constituents, that is, those that hold environment connections that were used as input to produce the respective stimuli generators.

Listing 6 shows the coupled model that was used to add the stimuli generators created by the approach proposed in this paper. At that coupled model, sensors transmit data to their closest mediator (Lines 2, 4, 6, and 8). Then, these mediators receive these data in Lines 3, 5, 7, and 9 forward `Measure` to the next sensors. Since `Sensor2` and `Sensor4` sent their data to `Mediator2` and `Mediator4` respectively (Lines 4 and 6), the gateway is already reached (Lines 5 and 9). When these 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. Thus, the network of exchanged messages between constituents and the flood alert trigger indicate that the SoS mission, i.e., producing flood alerts, has been accomplished by these constituent system.

After adapting this code, some new lines are added to drive the operation of the entire SoS. Listing 7 shows the resultant code after the changes that were performed, highlighting only the lines included or modified. Now, the `FloodMonitoringSosArchitecture` is also formed by four new entities: the stimuli generators. It is necessary to replicate the stimuli generator, creating four identical entities because, even identical, they can emit different data at different time units or moments. Since the stimuli generators received real data stored in external files, this enables distinct stimuli generators delivering distinct data at the same time, simulating the collection of distinct sensors geographically distributed along the river edges, collecting different data depending on the geographic specificities of that place in which the sensors is located or the sensor itself (level of water, level of battery, and GPS coordinates).

Listing 4 Transformation to derive a stimuli generator - Part II.

```
            initializeVariables += '''<<d.name.toFirstLower>> =
                new <<d.name.toFirstUpper>>('''
            if ((d.datatype as DataType) instanceof TupleType) {
              var e = (d.datatype as TupleType)
              var listSize = e.fields.size

              if (listSize == 1) {
                initializeVariables += '''new <<(e.fields.get(0)
                      as FieldDecl).type.compile>>()'''
              }

              if (listSize == 2) {
                initializeVariables += '''new <<(e.fields.get(0)
                      as FieldDecl).type.compile>>(), '''
                initializeVariables += '''new <<(e.fields.get(1)
                      as FieldDecl).type.compile>>()'''
              } else {
                for (var i = 0; i < e.fields.size; i++) {
                  initializeVariables += '''new '''
                  if (i == i < e.fields.size - 1)
                    initializeVariables += '''<<(e.fields.get(i)
                          as FieldDecl).type.compile>>()'''
                  else
                    initializeVariables += '''<<(e.fields.get(i)
                          as FieldDecl).type.compile>>(),'''
                } // end intern for
              } // end else
            }
            initializeVariables += ''');
            '''
          } // end else
                    }//end first else

            }//end for
        initializeVariables += '''

    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        if (reader != null) {
          reader.close();
        }
      } catch (IOException e) {
      }
    }

    statemachine += "from s" + countStates + " go to s0!\n"
    var StringBuffer output = new StringBuffer();
    var BufferedWriter out = null;

      try
      {
        var FileWriter  fstream = new FileWriter(
             fileAddressStimuli, true); // true tells to append
             data.
        out = new BufferedWriter(fstream);

        out.write(datatypes);
        out.newLine();
        out.write(initializeVariables);
        out.newLine();
        out.write(ports);
        out.newLine();
        out.write(statemachine);
        out.newLine();
        out.write(events);
        out.newLine();
      } catch (IOException e) {
        System.err.println("Error: " + e.getMessage());
      } finally {
        if (out != null) {
          out.close();
        }
      }
  }
```

Listing 5 Code DEVSNL for a stimuli generator automatically generated.

```
use count with type int and default "0"!
use listSensorData with type java.util.ArrayList<String>!
use sense with type RawData!

A Abscissa has a value!
the range of Abscissa's value is Integer!
A Ordinate has a value!
the range of Ordinate's value is Integer!

A Coordinate has x and y!
the range of Coordinate's x is Abscissa!
the range of Coordinate's y is Ordinate!

A RawData has coordinate, timestamp, and depth!
the range of RawData's coordinate is Coordinate!
the range of RawData's timestamp is String!
the range of RawData's depth is Depth!

A Depth has a value!
the range of Depth's value is Integer!

A Energy has a value!
the range of Energy's value is Integer!

use threshold with type Energy!
use power with type Energy!

generates output on Threshold with type Energy!
generates output on Power with type Energy!
generates output on Sense with type RawData!

output event for s0
<%
output.add(outThreshold, threshold);
%>!

output event for s1
<%
output.add(outPower, power);
%>!

output event for s2
<%
String line = listSensorData.get(count);
String[] vector = line.split(";");

if(vector.length > 1){

String timestamp = vector[0];
double doubleValue = Double.parseDouble(vector[6]);
int value = (int) doubleValue;
sense = new RawData(new Coordinate(new Abscissa(new Integer(2)), new Ordinate(
    new Integer(4))), new String(timestamp), new Depth(new Integer(value)));
output.add(outSense, sense);
count++;
}
%>!

Initialize variables
<%
count = 0;
sense = new RawData(new Coordinate(new Abscissa(new Integer(2)), new Ordinate(
    new Integer(4))), "", new Depth(new Integer(1)));
threshold = new Energy(new Integer(2));
power = new Energy (new Integer(100));

//reading data from file
%>!

to start hold in s0 for time 2!
after s0 output Power!
from s0 go to s1!
hold in s1 for time 1!
after s1 output Threshold!
from s1 go to s2!
hold in s2 for time 1!
after s2 output Sense!
from s2 go to s2!
```

Listing 6 Coupled Model for FMSoS generated in DEVS.

```
From the top perspective, FloodMonitoring is made of Sensor1, Sensor2,
    Sensor3, Sensor4, Mediator1, Mediator2, Mediator3, Mediator4, and
    Gateway!

//Sending Coordinate for mediators!
From the top perspective, Sensor1 sends Coordinate to Mediator1!
From the top perspective, Sensor2 sends Coordinate to Mediator1!
From the top perspective, Sensor2 sends Coordinate to Mediator2!
From the top perspective, Sensor3 sends Coordinate to Mediator3!
From the top perspective, Sensor4 sends Coordinate to Mediator3!
From the top perspective, Sensor4 sends Coordinate to Mediator4!
From the top perspective, Gateway sends Coordinate to Mediator3!
From the top perspective, Sensor1 sends Coordinate to Mediator1!


//Sending water level for the closest mediator!
From the top perspective, Sensor1 sends Measure to Mediator1!
From the top perspective, Mediator1 sends Measure to Sensor2!
From the top perspective, Sensor2 sends Measure to Mediator2!
From the top perspective, Mediator2 sends Measure to Gateway!

From the top perspective, Sensor4 sends Measure to Mediator4!
From the top perspective, Mediator4 sends Measure to Sensor3!
From the top perspective, Sensor3 sends Measure to Mediator3!
From the top perspective, Mediator3 sends Measure to Gateway!
```

Listing 7 Coupled Model for FMSoS generated in DEVS with Stimuli Generator support.

```
From the top perspective, FloodMonitoring is made of StimuliGenerator,
    StimuliGenerator2, StimuliGenerator3, StimuliGenerator4, Sensor1,
    Sensor2, Sensor3, Sensor4, Mediator1, Mediator2, Mediator3, Mediator4,
    and Gateway!


//Sending Energy, Energy, e Depth!
From the top perspective, StimuliGenerator sends Threshold to Sensor1!
From the top perspective, StimuliGenerator sends Threshold to Sensor1!
From the top perspective, StimuliGenerator sends Power to Sensor1!
From the top perspective, StimuliGenerator sends Sense to Sensor1!
From the top perspective, StimuliGenerator2 sends Threshold to Sensor2!
From the top perspective, StimuliGenerator sends Threshold to Sensor1!
From the top perspective, StimuliGenerator2 sends Power to Sensor2!
From the top perspective, StimuliGenerator2 sends Sense to Sensor2!
From the top perspective, StimuliGenerator3 sends Threshold to Sensor3!
From the top perspective, StimuliGenerator3 sends Power to Sensor3!
From the top perspective, StimuliGenerator3 sends Sense to Sensor3!
From the top perspective, StimuliGenerator4 sends Threshold to Sensor4!
From the top perspective, StimuliGenerator4 sends Power to Sensor4!
From the top perspective, StimuliGenerator4 sends Sense to Sensor4!
```