DEVS Coupled Model Specification in SVG to XML

Methodologies for Discrete Event Modelling and Simulation SYSC5104 / ELG6114

Chidiebere Onyedinma
Electrical and Computer Engineering
University of Ottawa
300093261
Carleton University
101161213

ABSTRACT: *The DEVS formalism is a widely implemented modeling and simulation method used for both artificial and natural systems. Scalable Vector Graphics (SVG) is a vector image format based on the Extensible Markup language (XML) for creating two-dimensional graphics that can be animated and are interactive. With the increasing need of modeling and simulation, most domain experts may have ideas of model they would want to create and run simulated tests with, but have little or no idea in computer programing. The aim of this project is to simplify the creation and simulation of DEVS model by allowing the user draw the model that needs to be simulated in SVG and executing it without writing any computer program. This is done by converting the drawn SVG to XML, and then to C++ code ready for execution with Cadmium which is a C++ based library used for defining and executing DEVS models. This paper specifies the standard for defining DEVS coupled model in SVG format and converting them to XML using a parser we implemented. The generated XML files can then be further used to generate C++ Cadmium code for execution*

1. INTRODUCTION

Discrete Event Systems Specifications (DEVS)[1] has gained so much popularity in recent years and has become a widely chosen technique for modeling and simulation. The reason for this is that DEVS allows for building of very complex models, because it splits the system under study into atomic models which can exist in a particular state and time, and have input/output ports for connecting to other atomic models in a hierarchical form. Implementation of these models formally defined in DEVS can sometimes be very complex to understand especially for experts in the domain of modeling and simulation that are interested in modeling but not in computer programing. With the creation of the Scalable Vector Graphics (SVG)[2] standards in 1999 by the World Wide Web Consortium (W3C), SVG images and their interactive behaviors are defined in XML formats which means that they can be indexed, searched, compressed, and scripted. We will show that formally defined DEVS models can be specified in SVG making them graphical and easily convertible for implementation. These domain experts can then create their DEVS models using SVG images (in the standards we will define) with drawing software as well as text editors because they are in XML format.

2. BACKGROUND

A. Extensible Markup Language (XML)

Extensible Markup Language (XML)[3] is a markup language that specifies a set of rules for encoding data in a format that is both machine and human readable. Created by the World Wide Web Consortium (W3C) in 1998, XML and several other related specifications are free open standards.

```
<?xml version="1.0" encoding="UTF-8"?>

<conversation id=”1”>

   <greeting>Hello, DEVS</greeting>

   <response>Hi! Its simulation everywhere</response>

</conversation>
```

Figure 1. Basic structure of an XML file

The fundamental building block of an XML document is an element which is usually defined by tags. Every element has an opening and a closing tag. Elements of an XML document are contained within the outermost element of the document known as the root element. Elements of an XML document can also have various attributes structured in a key-value pair style making the element more datacentric. XML also supports nesting of elements. Figure 1 shows the basic structure of an XML document with the “conversation” element nesting two other elements “greeting” and “response” respectively. This type of heirarchical structure that XML supports, is the main reason we have chosen it to specify our DEVS model.

B. Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG)[2] is an XML based markup language for describing two-dimensional graphics

that can be animated and are interactive. SVG has over seventy (70)
elements, with “svg” being the root element. We will focus on only eight (8) of these elements which are:

- svg
- defs
- marker
- path
- g
- rect
- text
- line

These tags are very crucial in specifying and visualizing DEVS coupled models. We will go in-depth into each one to fully understand the role they would play in the specification.

SVG

The svg element is the root element of every svg document. It contains the attributes that sets the structure for the rest of the document. These attributes are:

1. **xmlns**: This is the XML name space that contains the definitions of SVG
2. **version**: This is the SVG version the document is defined with.
3. **viewBox**: Defines the position and dimension in the user space of an SVG viewport, usually structured as “x y width height”
4. **height**: This is the height of the SVG image
5. **width**: This is the width of the SVG image

```
<svg viewBox=“0 0 220 100”
xmlns=“http://www.w3.org/2000/svg”>
    <rect width= “100” height= “100”/>
</svg>
```

Figure 2. Basic SVG document displaying a rectangle

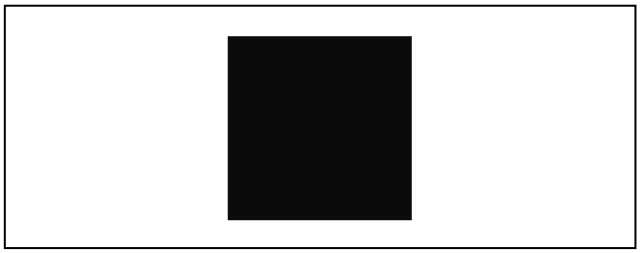

Figure 3. Output of figure 2

DEFS

The defs element is used to store graphical objects that would be used later on in the document. These graphical objects can then be referenced later on using their ‘id’.

Marker

The marker element defines the graphics to be used for drawing edges on a given path, line, polyline or polygon element. They are then attached to shapes using the “marker-start”, “marker-mid” and “marker-end” attributes.

Path

The path element is a generic element for defining shapes and all basic shapes can be created with it. Its primary attribute is “d” which defines the shape of the path being drawn.

G

The g element is used to group other SVG elements together for later referencing, and it also serves as a container.

Rect

The rect element is a basic shape that draws rectangles defined by their position, width and height. The rect element has various attributes, but we are going to be using only four attributes which are:

1. **x**: This defines the start x position of the rectangle.
2. **y**: This defines the start y position of the rectangle.
3. **width**: This defines the width of the rectangle.
4. **height**: This defines the height of the rectangle.

Text

The text element is used to draw graphics consisting of text. If the “text” is included in SVG and not inside of a <text> element, it won’t be rendered.

Line

The line element is a basic SVG shape used to create a line connecting two point. These points are defined using four attributes which are:

1. **x1**: This defines the x-axis of the line starting point
2. **x2**: This defines the x-axis of the line ending point
3. **y1**: This defines the y-axis of the line starting point
4. **y2**: This defines the y-axis of the line ending point

3. Defining DEVS Model in SVG

To define DEVS coupled models in SVG that would later be converted to XML, we need to specify the XML elements that would be correspond to elements of a DEVS coupled model. After specifying these elements, we would then define a standard for creating DEVS model in SVG using some primitive SVG elements and some custom attributes we will define to help match the graphics contained in our SVG to the elements in the XML file.

A. XML DEVS Specification

The various XML elements that would be used for specifying our DEVS coupled model are as follows:

- coupledModel
- ports
- port
- components
- submodel
- param
- connections
- eoc
- eic
- ic

coupledModel

The coupledModel element is the root element of our XML document, it will house all other elements of our model. The coupledModel element has one attribute which is "**name**" that defines the name of the coupledModel.

ports

The ports elements contain all the input and output ports of our DEVS coupled model.

port

The port element defines a port for our model. The port element has some attributes to specify its type and functionalities. These attributes are:

1. **type**: This defines the type of port which could be "in", "out" or "ir"
2. **name**: This defines the name of the port
3. **message_type**: This defines the type of message that would be passed through the port.

components

The components element is the hierarchical part of our model because it contains other submodels as its children.

submodel

The submodel element defines the sub-models of the DEVS coupled model being defined. It has some attributes that give more information about the model, these attributes are:

1. **type**: This specifies the type of the sub-model, which could be "atomic" or "coupled"
2. **name**: This specifies the name of the sub-model
3. **class_name**: This specifies the C++ class implementation of the model.
4. **xml_implementation**: This specifies the xml file implementation of the model.

The submodel may also have parameters defined as its children to indicate the input parameters of the C++ class

param

The param element may be included as a child element of the submodel element used to define the various input parameters of the C++ class that is generated to implement the model. The param element also has some attributes in its definition which are:

1. **type**: This is the C++ type of the input parameter
2. **name**: This is the identifier of the input parameter
3. **value**: This is the value of the input parameter

connections

The connections element contains the definitions of all the connections of the DEVS coupled model which are the external output couplings, external input couplings, and the internal couplings

eoc

This element specifies an external output coupling connection. This element has some attribute that define the connection which are:

1. **submodel**: This is the model where the data is coming from
2. **out_port_submodel**: This is the port of the model where the data is leaving from
3. **out_port_coupled**: This is the port of the coupled model where the data leaves from.

eic

This element specifies an external input coupling connection. This element has some attribute that define the connection which are:

1. **submodel**: This is the model where the data is into
2. **in_port_submodel**: This is the port of the model where the data is coming
3. **in_port_coupled**: This is the port of the coupled model where the data comes in to.

ic

This element specifies the internal coupling connection. This element has some attributes that define the connection which are:

1. **from_submodel**: The model where the data is coming from
2. **out_port_from**: The port of the model where the data leaves from
3. **to_submodel**: The model where the data is going to
4. **in_port_to**: The port of the model the where the data is coming in to

After specifying the XML elements to define a DEVS coupled model, we can now come up with a set of rules and standards to create SVG images that can be easy parsed to the XML file. During parsing, separate XML files are generated for each coupled model in our SVG.

```
<?xml version="1.0" encoding="UTF-8"?>
<coupledModel name="TOP">
  <ports>
    <port type="out" name="outp_ack" message_type="int" />
  </ports>

  <components>
    <submodel type="coupled" name="ABP" class_name="ABP"
xml_implementation="ABP.xml" />
  </components>

  <connections>
    <eoc submodel="ABP" out_port_submodel="outp_ack"
out_port_coupled="outp_ack" />
  </connections>
</coupledModel>
```

Figure 4. DEVS coupled model defined in XML

Figure 4 above is a DEVS coupled model defined using XML based on the elements we've specified.

B. SVG DEVS Specification

To properly define a DEVS model in SVG so that it can be visual and also contain all the components necessary for conversion, we need to create a standard and a set of rules to make sure the models conform to it.

SVG Root Rules

All SVG documents specifying a DEVs model must have a root svg element with all the attributes discussed in chapter two but with specific rules on what their value should be.

1. viewBox: Attribute defined as 'x y width height' must follow these rules:
   - x and y must equal to zero '0'
   - The ratio of the width to height of the viewBox must be equal to the ratio of the width to height of the svg document

2. version: The SVG version supported is from 1.1 to 2 as at the time of creating this document.

```
<svg
  xmlns="http://www.w3.org/2000/svg"
  version="1.1"
  viewBox="0 0 1200 600"
  height="200mm"
  width="400mm">
```

Figure 5. SVG root element for DEVS Model

Defs Rules

The SVG DEVS coupled model can contain only one **defs** element for predefining and storing our maker.

Marker Rules

The marker defined will be used as the edge shape for our connections. Only a single marker should be defined in our model and the rule for its definition is as follows:

1. id: The marker element must specify one 'id' attribute that would be used to identify and make reference to it.

2. viewBox: The marker must have a viewBox attribute with its position set to '0' (i.e. 'x' and 'y') and its width and height set both set to '10'

3. refX and refY: Both the refX and refY attributes must be set to '5' each

4. markerWidth and markerHeight: These must be equal and preferably set to '6'.

5. orient: This attribute sets the orientation of the marker and must be set to 'auto-start-reverse' this is to give the marker an automatic orientation and allow it face which ever direction our connection is facing.

6. path: As of this document we will only use an arrow head as our marker, and the path of the arrow head shape is defined as d="M 0 0 L 10 5 L 0 10 z" using the 'd' attribute.

```
<defs>
    <marker id="arrow" viewBox="0 0 10 10" refX="5" refY="5"
      markerWidth="6" markerHeight="6"
      orient="auto-start-reverse">
      <path d="M 0 0 L 10 5 L 0 10 z" />
    </marker>
</defs>
```

Figure 6. Marker definition

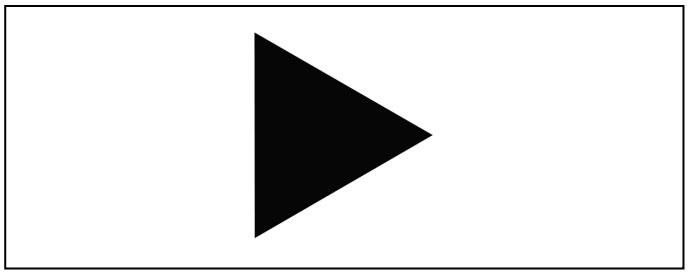

Figure 7. Rendered marker output

G Rules

The g element must be used to specify all the different components of our model and its required visual representations. Naturally the g element does not have any attributes, so we had to come up with a custom attribute called

"component" to specify the kind of model component the g element is representing. The different values of the component attribute are as follows:

- submodel
- graphics
- connections

Other attributes are also added based on these attributes in other to properly define the model component.

Coupled Model Definition Rules

The root of our entire model definition must be the TOP coupled model. Each coupled model must be defined using a g element and must also have an 'id' attribute that would be used to identify and reference it. The TOP model must have an 'id' explicitly set to "TOP". Every coupled model can have three types of children components which are **graphics** **submodel** and **connections**.

1. **submodel**: The submodel component could be an atomic or a coupled model. The g element of a submodel has the following attributes which must adhere to some certain rules and these attributes are:
    - **type**: This is the type of the submodel which could be atomic or coupled.
    - **id**: This is the name of the submodel
    - **class**: This is the C++ class implementation of the submodel
    - **has-param**: This attribute sets if the C++ class has input parameters
    - **param-no**: This is used to define the number of input parameters the C++ class accepts
    - **param-type**: This is the C++ type of the input parameter; it is appended with a number based on the number of the parameter e.g. param1-type.
    - **param-name**: This is the parameter name of the C++ input parameter; it is appended with a number based on the number of the parameter e.g. param1-name.
    - **param-value**: This is the value of the C++ class input parameter; it is appended with a number based on the number of the parameter e.g. param1-value.

A submodel itself can have a graphics component as its child, this is for proper visualization of the submodel.

```
<g id="input_reader" component="submodel" type="atomic" class="InputReader_Int" has-param="true" param-no="1" param1-type="const char*" param1-name="sFilename" param1-value="input_abp_1.txt">
```

Figure 8. Submodel defined in SVG

2. **graphics**: The graphics component specifies two SVG primitive shapes as its children, these elements are "rect" and "text". We must follow some certain rules to make sure these graphical components are rendered correctly. These rules are as follows:

Rect Rules

- The value of the 'x' attribute shouldn't be less than value of the 'x' attribute of the rectangle specified for its parent coupled model
- The value of the 'y' attribute shouldn't be less than value of the 'y' attribute of the rectangle specified for its parent coupled model
- The value of the sum of the 'x' attribute with the 'width' attribute, shouldn't be greater than value of the sum of the 'x' attribute with the 'width' attribute of the rectangle specified for its parent coupled model
- The value of the sum of the 'y' attribute with the 'height' attribute, shouldn't be greater than value of the sum of the 'y' attribute with the 'height' attribute of the rectangle specified for its parent coupled model

Text Rules

- The value of the 'x' attribute shouldn't be less than value of the 'x' attribute of the rectangle of the submodel graphics
- The value of the 'y' attribute shouldn't be less than value of the 'y' attribute of the rectangle of the submodel graphics
- The value of the 'x' attribute shouldn't be greater than value of the sum of the 'x' attribute with the 'width' attribute of the rectangle of the submodel graphics
- The value of the 'y' attribute shouldn't be greater than value of the sum of the 'y' attribute with the 'height' attribute of the rectangle of the submodel graphics

```
<g component="graphics">
  <rect x="150" y="120" width="150" height="300" stroke="black" fill="transparent" stroke-width="2"/>
  <text x="200" y="250" fill="black" stroke-width="0" font-size="16">Sender</text>
</g>
```

Figure 9. Graphics component of SVG DEVS model

Figure 9 above shows the textual description of the graphics

```
<line x1="150" x2="10" y1="280" y2="280" marker-end="url(#arrow)"/>
```

of a submodel specified in SVG, while figure 10 shows the rendered output.

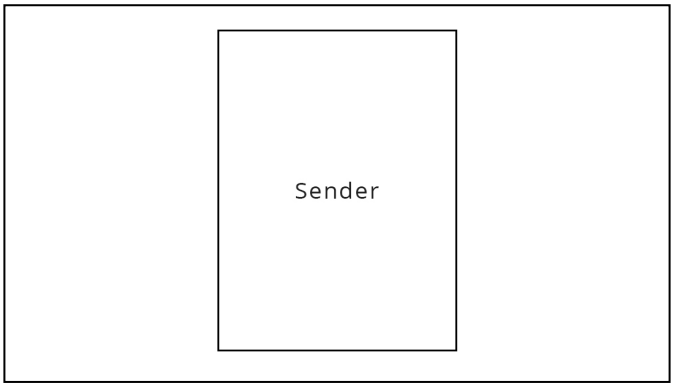

Figure 10. Rendered graphics component of an SVG DEVS model

3. **connections**: The connection component houses other g elements that specify different types of connections, and these g elements for the connections contain 'text' and 'line' elements to help visualize them. The different types of connections are: eoc, eic, and ic.

EOC Connection Rules

The eoc connection g element has five attributes which are:

- **type**: This is the type of connection.
- **out-coupled**: This is the output port from the coupled model.
- **submodel**: This is the model where the data is coming from.
- **out-submodel**: This is the output port of the submodel to the coupled model.
- **message-type**: This is the data exchange type.

The rules for the 'line' element are as follows:

- The line must be straight either vertically or horizontally i.e. x1=x2 or y2=y1
- If the line is pointing to the left (i.e. x1 > x2), x1 must not be more than 3mm away from the model rectangle its preceding from.
- If the line is pointing to the right (i.e. x1 < x2), x1 must be in a 3mm range of the model's rectangle "x + width" its preceding from.
- If the line is pointing up (i.e. y1 > y2), y1 must not be more than 3mm away from the model rectangle its preceding from.
- If the line is pointing down (i.e. y1 < y2), y1 must be in a 3mm range of the model's rectangle "y + height" its preceding from.
- The line must have a previously specified marker ending it

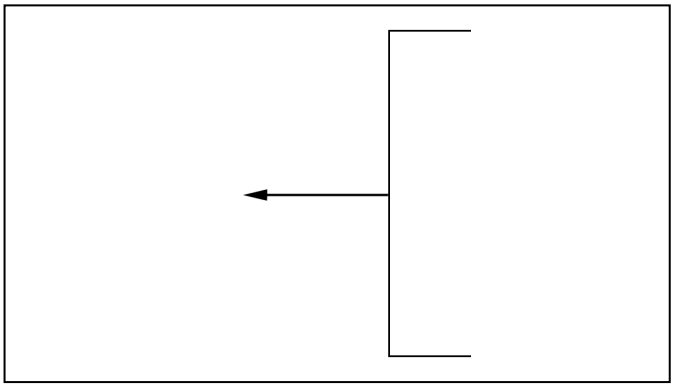

Figure 12. Rendered line element for an eoc connection

The rules for the 'text' element are as follows:

- The text must be at least 2mm above the connection line

```
<text x="25" y="278" fill="black" font-size="14" >ackReceived</text>
```

Figure 13 Text element for a connection

EIC Connection Rules

The eic connection g element has five attributes which are:

- **type**: This is the type of connection.

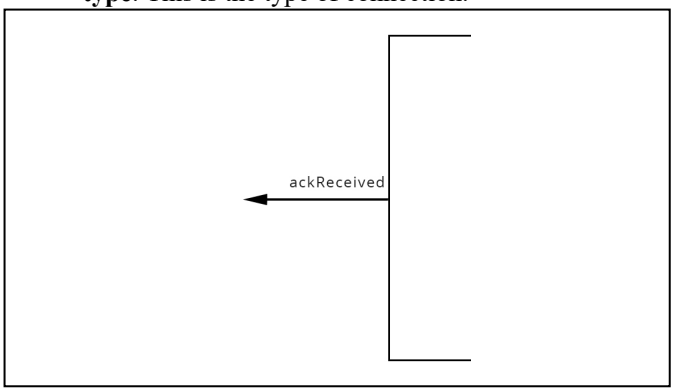

Figure 14 Rendered text element for a connection

- **in-coupled**: This is the input port to the coupled model.
- **submodel**: This is the model where the data is going into.
- **in-submodel**: This is the input port of the submodel from the coupled model.
- **message-type**: This is the data exchange type.

The rules for the 'line' element are as follows:

- The line must be straight either vertically or horizontally i.e. x1=x2 or y2=y1

- If the line is pointing to the right (i.e. x1 < x2), x1 must not be more than 6mm away from the model rectangle its pointing to.
- If the line is pointing to the left (i.e. x1 > x2), x1 must be in a 6mm range of the model's rectangle "x + width" its pointing to.
- If the line is pointing down (i.e. y1 < y2), y1 must not be more than 6mm away from the model rectangle its pointing to.
- If the line is pointing up (i.e. y1 > y2), y1 must be in a 6mm range of the model's rectangle "y + height" its pointing to.
- The line must have a previously specified marker ending it

```
<line x1="10" x2="144" y1="200" y2="200" marker-end="url(#arrow)"/>
```

Figure 15 Line element for an eic connection

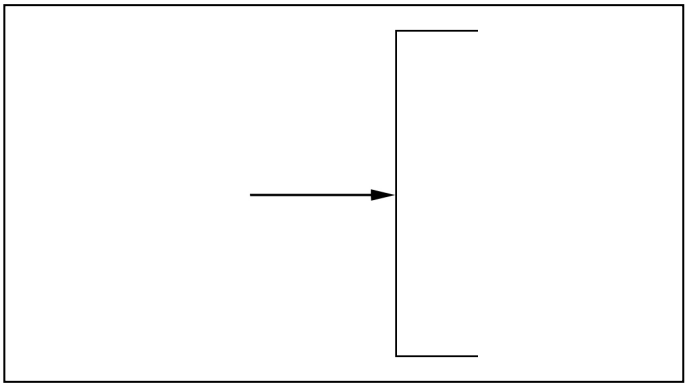

Figure 16. Rendered line element for an eic connection

IC Connection Rules

The ic connection g element has five attributes which are:

- **type**: This is the type of connection.
- **from**: This is the model where the data is coming from.
- **out-from**: This is the output port of the model sending the data
- **to**: This is the model receiving the data.
- **in-to**: This is the input port of the model receiving the data

The rules for the 'line' element are as follows:

- The line must be straight either vertically or horizontally i.e. x1=x2 or y2=y1
- If the receiving model is on the left (i.e. x1 > x2), x1 must be in a 6mm range of the receiving model's rectangle "x + width".
- If the receiving model is on the right (i.e. x1 < x2), x1 must not be more than 6mm away from the receiving model's rectangle.
- If the receiving model is downwards, (i.e. y1 < y2), y1 must not be more than 6mm away from the receiving model's rectangle.
- If the receiving model is upwards (i.e. y1 > y2), y1 must be in a 6mm range of the receiving model's rectangle "y + height".
- The line must have a previously specified marker ending it

```
<line x1="725" x2="323" y1="325" y2="325" marker-end="url(#arrow)"/>
```

Figure 17. Line element for an ic connection

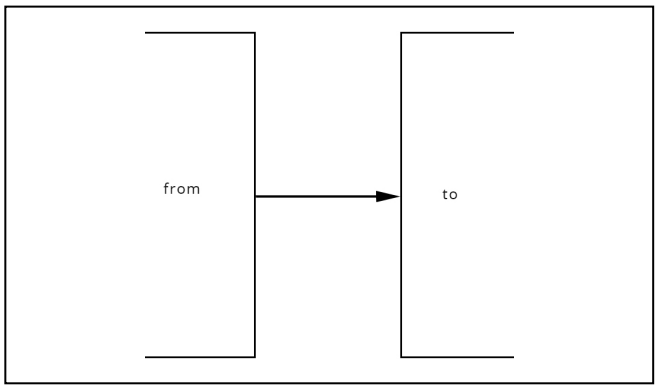

Figure 18. Rendered line element for an ic connection

With all the specified rules and standards, we have outlined, we can now properly define a DEVS coupled model in SVG that can be easily converted to XML and displays a graphical representation of the model.

4. SVG TO XML PARSING RESULTS

Based on the SVG and XML specifications we've come up with to properly define DEVS coupled models, we can create a parser to convert SVG images that have been properly defined with these specified rules and standard into XML. SVG being a type of xml document makes the correlation of data between the SVG file and the corresponding XML files to be generated easy.

The parser was implemented using Java and the apache batik library for SVG. To properly parse the SVG file into XML, we had to come up with different steps to both validate the SVG document to be parsed (to see if it conforms to all the standards), and match data from the SVG file, to the correct XML elements we've defined.

A separate XML file is generated for each coupled model in the SVG document. Coupled models in our SVG are structured in a tree like manner with every coupled model having its own submodels which could also be coupled and so on. To be able to retrieve and parse each coupled model into its respective XML file, we had to use a recursive

approach to traverse the structure. Figure 19 shows the program flow of the parser and the recursive approach.

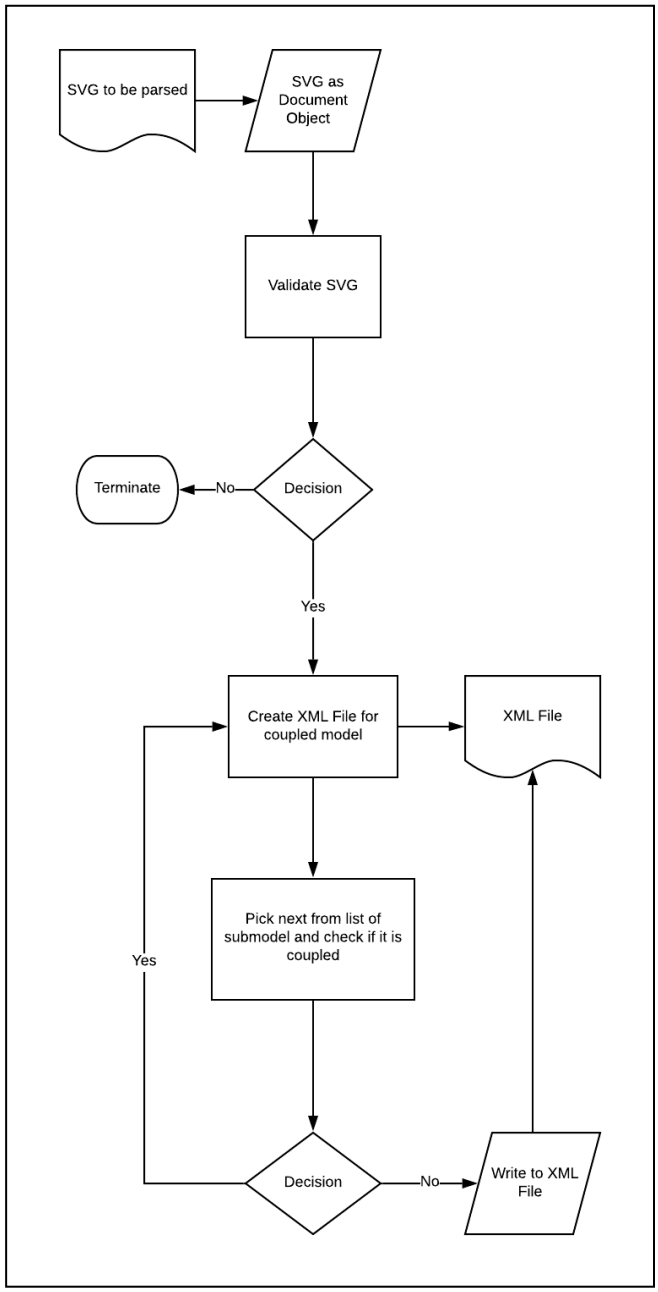

Figure 19. Parser program flow

The SVG file is first converted in an SVG document object using the apache batik library. After conversion, validation is carried out on all elements defined in the SVG file to check if it conforms to the specified standards. If it does, an XML file is created for the topmost coupled model, using its 'id' attribute as the file name. A submodel is picked from the list of submodels nested in the coupled model and then we check if it is also a coupled model. If it's not, we write the atomic models matching elements to the created XML file, but if it is, a XML file is generated and the recursive cycle continues until all the coupled models have been written to their corresponding XML files.

A. Matching and Parsing SVG to XML Elements

Each element of the coupled model defined in the SVG has to match its corresponding element in the XML file.

SVG Connections to XML Port Matching

The SVG connection **g** element has every information to write the ports data for the XML file.

EIC connections in our SVG are used to create input ports in our XML file the matching is as follows

Table 1. SVG EIC to XML input port matching

SVG **g** element connection	XML port of type 'in'
in-coupled	name
message-type	message_type

EOC connections in our SVG are used to create output ports in our XML file the matching is as follows

Table 2. SVG EOC to XML output port matching

SVG **g** element connection	XML port of type 'out'
out-coupled	name
message-type	message_type

SVG Connection to XML Connection Matching

The SVG connection **g** element has every information needed to write the connections data for the XML file.

EIC connections in our SVG are used to create eic elements in our XML connections

Table 3. SVG EIC to XML eic element

SVG **g** element type "eic"	XML eic element
in-coupled	in_port_coupled
submodel	submodel
in-submodel	in_port_submodel

EOC connections in our SVG are used to create eoc elements in our XML connections

Table 4.SVG EOC to XML eoc element

SVG **g** element type "eoc"	XML eoc element
out-coupled	out_port_coupled
submodel	submodel
out-submodel	out_port_submodel

IC connections in our SVG are used to create ic elements in our XML connections

SVG **g** element type “ic”	XML ic element
from	from_submodel
out-from	out_port_from
to	to_submodel
in_to	in_port_to

SVG Submodel to XML Matching

The SVG submodel **g** element has every information needed to write the submodel data for the XML file.

SVG **g** element type “submodel”	XML submodel element
id	name
type	type
class	class_name

The corresponding xml_implementation attribute value is generated using the class.

B. XML Generation Results

We used the Alternating Bit Protocol DEVS model to test our parser. We first defined the model in SVG using our standards, and then parsed it to generate the XML files. The figures below.

```
<g component="connections">
  <g type="eoc" out-coupled="outp_pack"
submodel="ABP" out-submodel="outp_pack"
message-type="int">
  </g>
  <g type="eoc" out-coupled="outp_ack"
submodel="ABP" out-submodel="outp_ack"
message-type="int">
  </g>
  <g type="ic" from="input_reader" out-
from="out" to="ABP" in-to="inp_control"
message-type="int">
  </g>

</g>
```

Figure 20. ABP SVG Top coupled model connections

```
<connections>
  <eoc submodel="ABP"
out_port_submodel="outp_pack"
out_port_coupled="outp_pack" />
  <eoc submodel="ABP"
out_port_submodel="outp_ack"
out_port_coupled="outp_ack" />
  <ic from_submodel="input_reader"
out_port_from="iestream_input_defs&lt;int&gt;
::out" to_submodel="ABP"
in_port_to="inp_control" />
</connections>
```

Figure 21. ABP XML Top coupled model connections

```
<g id="input_reader" component="submodel"
type="atomic" class="InputReader_Int" has-
param="true"
    param-no="1" param1-type="const char*"
param1-name="sFilename" param1-
value="input_abp_1.txt">
    </g>
    <g id="ABP" component="submodel"
type="coupled" class="ABP">
        <g component="graphics">
            <rect id="abp" x="100" y="10"
width="900" height="500"/>
            <text x="120" y="45" fill="black"
stroke-width="0" font-size="16">ABP
Simulator</text>
        </g>
        <g component="submodel" id="sender1"
type="atomic" class="Sender">
            <g component="graphics">
                <rect x="150" y="120"
width="150" height="300" stroke="black"
fill="transparent" stroke-width="2"/>
                <text x="200" y="250"
fill="black" stroke-width="0" font-
size="16">Sender</text>
            </g>
        </g>
size="14">out2</text>
size="14">out</text>
                <line x1="800" x2="704"
y1="325" y2="325" stroke-width="2" marker-
end="url(#arrow)"/>
            </g>
        </g>
    </g>
```

Figure 23. ABP XML Top coupled model submodel

```
<components>
        <submodel type="atomic"
name="input_reader"
class_name="InputReader_Int"
xml_implementation="InputReader">
            <param type="const char*"
name="sFilename" value="input_abp_1.txt" />
        </submodel>
        <submodel type="coupled" name="ABP"
class_name="ABP" xml_implementation="ABP.xml"
/>
    </components>
```

Figure 22. ABP SVG Top coupled model submodel

C. *Building and Running the Parser*

The parser is built using maven which is a build tool for Java.

- To build the source files simply install maven and add it to "path" environment variable.
- Install Java JDK version 8 and above
- Go to the root directory of the project and run the following command:

```
mvn clean compile package
```

This will generate a target folder with the "svg-xml.jar" file inside. If not existing, create a directory called "svgfiles" in the same location as the jar file and put your svg files inside. To parse a svg file, simple run the following command:

```
java -jar svg-xml.jar yourSVGfile.svg
```

The generated XML files will be in a directory called XMLFiles

5. CONCLUSIONS

Defining DEVS coupled models in SVG for conversion to XML can reduce the complexities in implementing these formally defined DEVS models. These XML files can then be converted to C++ Cadmium code for execution.

The SVG and XML specification of DEVS coupled model help give a visual representation of the DEVS model and also makes it ready for further execution without prior knowledge of computer programing. This would help domain experts to easily implement and execute their own models without the much hassle of writing a computer program.

A. *Recommendation and Future Work*

The SVG files used in this project was hand coded in other to do a proof of concept. A graphical user interface would be recommended for drawing and specifying these SVG files as this would reduce the complexities of hand coding them, and also save time.

In our SVG specification, we used our custom defined attributes, and this may cause the SVG file not to render properly in certain editors or web browsers. The solve this issue, an attribute namespace should be created for our defined attributes, and the namespace added to our root svg tag.

The second part of this project which was not implemented is creating a parser that will import the generated XML file and convert them into executable C++ Cadmium programs.

We considered only coupled models in this project, another recommendation would be to add atomic models in our specifications.

REFERENCES

[1] G. K. Bernard, Zeigler, Herbert, Praehofer, Tag, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, 2nd ed. Academic Press, 2000.

[2] "SVG: Scalable Vector Graphics," 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/SVG. [Accessed: 22-Dec-2019].

[3] "XML: Extensible Markup Language," 2019. [Online]. Available: https://en.wikipedia.org/wiki/XML. [Accessed: 22-Dec-2019].