DEVS-Graph Web-Application Developer Manual

Nirmal Patel – 101220783

**Supervisor** – Prof. Cristina Ruiz Martin

**Introduction:**

DEVS (Discrete EVent System Specification) is a formalism to model discrete events using mathematical modal. Originally, the DEVS models for cadmium simulator are developed using the programming language C++. The graphical specification of the DEVS is called DEVS-Graph. One such user-friendly web-based application is developed for DEVS-Graph. The technologies used to develop this application are as follows,

**Frontend:** Vanilla JavaScript (GoJS)

**Backend:** SpringBoot JAVA

**Templating engines:** cogapp (Python) and Freemaker (JAVA)

**Deployment Server:** Ubuntu

**Frontend Documentation**

**Palette Object Configuration:**

This method creates a Palette object and assigns it to the myPalette variable. The Palette is a graphical user interface element that provides a set of pre-defined nodes and links that the user can drag and drop onto a Diagram to create a new diagram.

1. myPalette = $(
2. go.Palette,
3. "myPaletteDiv", // must name or refer to the DIV HTML element
4. {
5. maxSelectionCount: 1,
6. nodeTemplateMap: myDiagram.nodeTemplateMap,
7. linkTemplate: $(
8. go.Link,
9. ...
10. ),
11. model: **new** go.GraphLinksModel(
12. [
13. // specify the contents of the Palette
14. {
15. text: "A",
16. figure: "Ellipse",
17. size: "85 85",
18. fill: "white",
19. },
20. {
21. text: "IEStream",
22. figure: "Rectangle",
23. size: "90 65",
24. fill: "white",
25. }
26. ],
27. [
28. // the Palette also has a disconnected Link, which the user can drag-and-drop
29. {
30. points: **new** go.List(/\*go.Point\*/).addAll([
31. **new** go.Point(0, 0),
32. **new** go.Point(85, 0)
33. ]),
34. color: "black"
35. },
36. {
37. points: **new** go.List(/\*go.Point\*/).addAll([
38. **new** go.Point(0, 0),
39. **new** go.Point(85, 0)
40. ]),
41. color: "red",
42. arrow\_color: "red",
43. fill\_arrow: "red"
44. },
45. {
46. points: **new** go.List(/\*go.Point\*/).addAll([
47. **new** go.Point(0, 0),
48. **new** go.Point(85, 0)
49. ]),
50. dash\_array: [6, 3],
51. }
52. ]
53. ),
54. }
55. );

The Palette is created using the go.Palette constructor and is associated with a DIV HTML element with the id myPaletteDiv. The Palette is initialized with a set of options that define its behavior and appearance.

The maxSelectionCount property is set to 1, which means that only one node or link can be selected at a time from the Palette. The nodeTemplateMap property is set to myDiagram.nodeTemplateMap, which specifies the node templates that should be used to render the nodes in the Palette.

The linkTemplate property is set to a go.Link object that defines the default appearance and behavior of the links in the Palette. This Link object has several properties that control its behavior, such as selectable, relinkableFrom, relinkableTo, and reshapable, which enable the user to select, relink, and reshape the links in the Palette.

The Link object also has a Binding property that is used to bind the points of the link to a data value. This allows the Palette to display links with different shapes and configurations.

The Palette also has a model property that is set to go.GraphLinksModel object. This object defines the nodes and links that should be displayed in the Palette. The model is initialized with an array of node data objects and an array of link data objects. These data objects define the appearance and behavior of the nodes and links in the Palette.

**Node (Shapes) Object Configuration**

This method defines a Node template for a Diagram object. The Node template is created using the go.Node constructor and is assigned to the nodeTemplate property of the myDiagram object.

1. myDiagram.nodeTemplate =
2. $(go.Node, "Spot",
3. { locationSpot: go.Spot.Center },
4. **new** go.Binding("location", "loc", go.Point.parse).makeTwoWay(
5. go.Point.stringify
6. ),
7. {
8. selectable: **true**,
9. selectionAdornmentTemplate: nodeSelectionAdornmentTemplate,
10. },
11. {
12. resizable: **true**,
13. resizeObjectName: "PANEL",
14. resizeAdornmentTemplate: nodeResizeAdornmentTemplate,
15. },
16. **new** go.Binding("angle").makeTwoWay(),
17. // the main object is a Panel that surrounds a TextBlock with a Shape
18. $(
19. go.Panel,
20. "Auto",
21. { name: "PANEL" },
22. **new** go.Binding("desiredSize", "size", go.Size.parse).makeTwoWay(
23. go.Size.stringify
24. ),
25. $(
26. go.Shape,
27. "Rectangle", // default figure
28. {
29. ...
30. },
31. **new** go.Binding("figure"),
32. **new** go.Binding("fill")
33. ),
34. $(
35. go.TextBlock,
36. {
37. ...
38. },
39. **new** go.Binding("text").makeTwoWay()
40. )
41. ),
42. // four small named ports, one on each side:
43. makePort("T", go.Spot.Top, **true**, **true**)
44. );

The Node template is initialized with several properties that define its behavior and appearance. The locationSpot property is set to go.Spot.Center, which means that the location of the Node is determined by its center point. The location property is bound to the loc data property using the Binding constructor, which allows the Node to be moved around the Diagram by dragging it with the mouse.

The selectable property is set to true, which allows the user to select the Node by clicking on it. The selectionAdornmentTemplate property is set to nodeSelectionAdornmentTemplate, which specifies the adornment that should be displayed when the Node is selected.

The resizable and rotatable properties are set to true, which allows the user to resize and rotate the Node using the adornments specified by the resizeAdornmentTemplate and rotateAdornmentTemplate properties, respectively.

The Node template also has a Panel object that defines the appearance and behavior of the Node. The Panel has a Shape object and a TextBlock object as its children. The Shape object is used to render the background of the Node, and the TextBlock object is used to display the Node's text. Both of these objects have Binding properties that are used to bind their visual attributes to data values.

The Node template also has four named ports, which are defined using the makePort function. These ports can be used to attach Link objects to the Node. The Node also has event handlers that are used to show and hide the ports when the mouse enters and leaves the Node, respectively.

Finally, the Node template has a toolTip and a contextMenu property that specify the adornments that should be displayed when the user hovers over the Node with the mouse or right-clicks on it, respectively. These adornments provide additional information and functionality for the Node.

**Link Template Configuration**

This code defines a Link template for a Diagram object. The Link template is created using the go.Link constructor and is assigned to the linkTemplate property of the myDiagram object.

1. myDiagram.linkTemplate =
2. $(
3. go.Link, // the whole link panel
4. {
5. selectable: **true**,
6. },
7. { relinkableFrom: **true**, relinkableTo: **true**, reshapable: **true** },
8. {
9. routing: go.Link.Orthogonal,
10. smoothness: 0.01,
11. curve: go.Link.Bezier,
12. },
13. $(
14. go.Shape, // the link path shape
15. { isPanelMain: **true**, strokeWidth: 8, stroke: "transparent" },
16. ),
17. $(go.Shape,
18. **new** go.Binding("strokeDashArray", "dash\_array"),
19. { isPanelMain: **true** },
20. **new** go.Binding("stroke", "color")),
21. $(
22. go.Shape, // the arrowhead
23. { toArrow: "Standard", stroke: "black" },
24. **new** go.Binding("stroke", "arrow\_color"),
25. **new** go.Binding("fill", "fill\_arrow")
26. ),
27. {
28. mouseEnter: **function** (e, link) { link.elt(0).stroke = "rgba(0,90,156,0.3)"; },
29. mouseLeave: **function** (e, link) { link.elt(0).stroke = "transparent"; }
30. },
31. $(go.TextBlock,  // the "from" label
32. {
33. ...
34. },
35. **new** go.Binding("text", "text")),
36. $(go.TextBlock,  // the "from" label
37. {
38. ...
39. },
40. **new** go.Binding("text", "text")),
41. $(
42. go.Panel,
43. "Auto",
44. // new go.Binding("visible", "isSelected").ofObject(),
45. $(
46. go.Shape,
47. "RoundedRectangle", // the link shape
48. { fill: "grey", stroke: **null** }
49. ),
51. $(
52. go.TextBlock,
53. {
54. ...
55. },
56. **new** go.Binding("text", "label").makeTwoWay(),
57. BindSelection("stroke", "white", "black"),
58. BindSelection("background", "grey", **null**)
59. )
60. )
61. );

The Link template is initialized with several properties that define its behavior and appearance. The selectable property is set to true, which allows the user to select the Link by clicking on it. The relinkableFrom, relinkableTo, and reshapable properties are all set to true, which allows the user to relink and reshape the Link using the mouse.

The routing property is set to go.Link.Orthogonal, which means that the Link will be drawn as a series of orthogonal segments. The curve property is set to go.Link.Bezier specifies that the segments should be connected using Bezier curves.

The Link template also has several Binding properties that are used to bind the Link's visual attributes to data values. This allows the Link to be customized based on the data that it is associated with.

The Link template has a Shape object that is used to render the path of the Link. This Shape has a strokeDashArray property that is bound to the dash\_array data property, which allows the Link to be drawn with a dashed line.

The Link template also has a TextBlock object that is used to display labels on the Link. This TextBlock has an editable property that is set to true, which allows the user to change the label by clicking on it and typing a new value.

The Link template also has a toolTip and a contextMenu property that specify the adornments that should be displayed when the user hovers over the Link with the mouse or right-clicks on it, respectively. These adornments provide additional information and functionality for the Link.

**Diagram Object Configuration**

This code creates a go.Diagram object, which represents a diagramming or graph-drawing surface, and assigns it to the myDiagram variable. The go.Diagram object is created by passing several options to the go.Diagram constructor function.

1. myDiagram = $(
2. go.Diagram,
3. "myDiagramDiv", // must name or refer to the DIV HTML element
4. {
5. grid: $(
6. go.Panel,
7. "Grid"
8. ),
9. "draggingTool.dragsLink": **true**,
10. "draggingTool.isGridSnapEnabled": **true**,
11. "linkingTool.isUnconnectedLinkValid": **false**,
12. "linkingTool.portGravity": 20,
13. "relinkingTool.isUnconnectedLinkValid": **true**,
14. "relinkingTool.portGravity": 20,
15. "relinkingTool.fromHandleArchetype": $(go.Shape, "Diamond", {
16. segmentIndex: 0,
17. cursor: "pointer",
18. desiredSize: **new** go.Size(8, 8),
19. fill: "tomato",
20. stroke: "darkred",
21. }),
22. "relinkingTool.toHandleArchetype": $(go.Shape, "Diamond", {
23. segmentIndex: -1,
24. cursor: "pointer",
25. desiredSize: **new** go.Size(8, 8),
26. fill: "dark red",
27. stroke: "tomato",
28. }),
29. "linkReshapingTool.handleArchetype": $(go.Shape, "Diamond", {
30. desiredSize: **new** go.Size(7, 7),
31. fill: "lightblue",
32. stroke: "deepskyblue",
33. }),
34. "rotatingTool.handleAngle": 270,
35. "rotatingTool.handleDistance": 30,
36. "rotatingTool.snapAngleMultiple": 15,
37. "rotatingTool.snapAngleEpsilon": 15,
38. "undoManager.isEnabled": **true**,
39. "commandHandler.archetypeGroupData": { text: "Model", isGroup: **true**, color: "blue", I: [], O: [] },
40. }
41. );

The myDiagramDiv element is the HTML element that the diagram that will be drawn in. The grid option sets the grid that will be displayed on the diagram's background. The draggingTool.dragsLink option determines whether or not a link is being dragged when the user starts dragging from a link. The draggingTool.isGridSnapEnabled option determines whether or not the diagram should snap to the grid when objects are moved.

The linkingTool.isUnconnectedLinkValid option determines whether or not unconnected links are valid in the diagram. The linkingTool.portGravity option determines the strength of the force that will pull a link toward a port when the link is connected.

The relinkingTool.isUnconnectedLinkValid option determines whether or not unconnected links are valid when relinking in the diagram. The relinkingTool.portGravity option determines the strength of the force that will pull a link toward a port when relinking. The relinkingTool.fromHandleArchetype and relinkingTool.toHandleArchetype options determine the appearance of the handles that are used to relink links.

The linkReshapingTool.handleArchetype option determines the appearance of the handles that are used to reshape links. The rotatingTool.handleAngle, rotatingTool.handleDistance, rotatingTool.snapAngleMultiple, and rotatingTool.snapAngleEpsilon options determine the behavior of the tool that is used to rotate objects in the diagram.

The undoManager.isEnabled option determines whether or not the diagram's undo manager is enabled. The commandHandler.archetypeGroupData option sets the default data for a new group that is created in the diagram.

**Download Model JSON**

1. **function** downloadModel() {
2. **var** blob = **new** Blob([myDiagram.model.toJson()],
3. { type: "application/json;charset=utf-8" });
4. saveAs(blob, "model.json");
5. }

This function creates a blob object from the diagram json and uses saveAs library to download the raw json data file as “model.josn”.

**Load Model JSON**

1. **function** load() {
2. **let** file = document.getElementById("formFile").files[0];
3. **var** reader = **new** FileReader();
4. **if** (file !== undefined) {
5. reader.onloadend = **function** () {
7. myDiagram.model = go.Model.fromJson(reader.result);
9. }
10. reader.readAsText(file);
11. }
12. **else** {
13. myDiagram.model = go.Model.fromJson(
14. document.getElementById("mySavedModel").value
15. );
16. }
18. loadDiagramProperties(); // do this after the Model.modelData has been brought into memory
20. }

This function allows loading a JSON file into a diagram div. Firstly, read the file uploaded to element formFile and read the file contents using FileReader(). Update the myDiagram.model with the uploaded JSON data. This function is used by the application when the application is initialized (or the page is refreshed), therefore, in such case no file would be loaded instead we use the default empty JSON from the mySavedModel div value.

**Transform the Model JSON to Simplified JSON**

The function transformData takes in a data object and converts it into a simplified JSON. The data object is expected to be in JSON format, so the first step is to parse it using the JSON.parse function.

1. nodeData.forEach(d => {
2. **if** (d.hasOwnProperty("isGroup") && d.isGroup) {
3. model.push(d);
4. group\_keys.push(d.key);
5. }
6. **else**
7. states.push(d);
8. })

Firstly, it iterates through nodeDataArray to identify which are independent nodes and which are group objects. The independent nodes will be states for our model and the groups will be atomic and coupled models.

Secondly, to identify the atomic and coupled models we again loop through nodeDataArray and check if the node has both the properties “group” and “isGroup” then the node is a coupled model else the node is an atomic model.

1. nodeData.forEach(d => {
2. **if** ((d.hasOwnProperty('group') && d.hasOwnProperty('isGroup')) || d.figure === "Rectangle") {
3. **if** (coupled\_id.hasOwnProperty(d.group))
4. coupled\_id[d.group].push(d.key);
5. **else**
6. coupled\_id[d.group] = [d.key];
7. }
8. **else** **if** (d.hasOwnProperty('group') && !d.hasOwnProperty('isGroup')) {
9. **if** (atomic\_id.hasOwnProperty(d.group))
10. atomic\_id[d.group].push(d.key);
11. **else**
12. atomic\_id[d.group] = [d.key];
13. }
14. })

In the next steps, the array for an atomic and coupled model is built. The structure of simplified JSON is as followed,

1. {
2. "top": {
3. "name": "top\_model",
4. "out\_port": "output"
5. },
6. "atomic": [
7. {
8. "name": "Table",
9. "initial\_state": "idle",
10. "imports": [
11. {
12. "name": "in5",
13. "type": "int"
14. }
15. ],
16. "outports": [
17. {
18. "name": "out4",
19. "type": "int"
20. }
21. ],
22. "states": {
23. "idle": "inf",
24. "ready": 15
25. },
26. "internal\_transitions": [
27. {
28. "curr\_state": "ready",
29. "new\_state": "idle"
30. }
31. ],
32. "external\_transitions": [
33. {
34. "port": "in5",
35. "value": "1",
36. "curr\_state": "idle",
37. "new\_state": "ready"
38. }
39. ],
40. "output": [
41. {
42. "curr\_state": "ready",
43. "port": "out4",
44. "value": "3"
45. }
46. ]
47. }
48. ],
49. "coupled": [
50. {
51. "name": "top\_model",
52. "components": [
53. "Lobby",
54. "Restaurant",
55. "IEStream"
56. ],
57. "top": {
58. "name": "top\_model",
59. "out\_port": "output"
60. },
61. "inports": [],
62. "outports": [
63. {
64. "name": "output",
65. "type": "int"
66. }
67. ],
68. "couplings": [
69. {
70. "from\_model": "Lobby",
71. "to\_model": "Restaurant",
72. "from\_port": "out1",
73. "to\_port": "in2"
74. },
75. {
76. "from\_model": "Lobby",
77. "to\_model": "Restaurant",
78. "from\_port": "out2",
79. "to\_port": "in3"
80. },
81. {
82. "from\_model": "IEStream",
83. "to\_model": "Lobby",
84. "from\_port": "out",
85. "to\_port": "in1"
86. },
87. {
88. "from\_model": "Restaurant",
89. "to\_model": "",
90. "from\_port": "out5",
91. "to\_port": "output"
92. }
93. ]
94. }
95. ]
96. }

**Generate Cadmium Code**

1. **function** generate() {
2. **let** data = transformData(save());
3. fetch('http://localhost:8080/cadmium/generate', {
4. method: 'POST',
5. headers: {
6. 'Content-Type': 'application/json',
7. },
8. body: JSON.stringify(data),
9. }).then((res) => {
10. **return** res.blob();
11. }).then((blob) => {
12. saveAs(blob, document.getElementById("modalname").value + ".zip")
13. }).**catch**((e) => console.log(e));
14. }

To generate cadmium code, first, we need to get the raw JSON data from the myDiagram object using the save() method. Then we can use this raw JSON as an argument to the transformData method to convert it into simplified JSON. Fetch API is used to call the backed service URL path “/cadmium/generate” and the response object is returned to the browser using saveAs library as a ZIP file.

**Backend Service Documentation**

**Generate Cadmium Code**

This method uses JAVA process builder to run a python command to cogapp (a utility that uses the template to replace variables with values). If we want to run this command in cmd or shell we can use the following command,

python -m cogapp -D fileData=data -o test\_model.hpp model\_atomic.hpp

Here, fileData is the variable that points to the simplified JSON object of the model which we want to generate code for, the -o flag indicates the output file name, and the last argument model\_atomic.hpp is the template file used to generate the code. As of now, there are two template files, model\_atomic.hpp and model\_coupled.hpp

1. **private** **boolean** generateCode(String data, String projectID, String modelName, String modelType)
2. **throws** IOException, InterruptedException {
4. File dir = **new** File(root + "/" + projectID);
6. List<String> command = **new** ArrayList<String>(
7. Arrays.asList(python, "-m", "cogapp", "-D", "fileData=" + data, "-o", modelName));
8. String modelfile = modelType == "atomic" ? "model\_atomic.hpp" : "model\_coupled.hpp";
9. command.add(modelfile);
10. ProcessBuilder pb = **new** ProcessBuilder(command).redirectErrorStream(**true**);
11. pb.directory(dir);
12. pb.redirectError();
14. Process p = pb.start();
15. InputStream is = p.getInputStream();
16. **int** value = -1;
17. **while** ((value = is.read()) != -1) {
18. System.out.print((**char**) value);
19. }
20. **int** exitCode = p.waitFor();
21. **return** exitCode == 0;
22. }

A list of Strings is created with the command we want to execute for the python interpreter. Process builder takes the commands list as an argument and the redirect error is set to true, which will print the error to the console if there is any. The process is executed using the start method and then the application waits for the process to finish to ensure the execution or error of the command. If the command runs successfully, the exit code returned is 0.

**Controller to Generate Cadmium Code**

For each request made to the backend service, this function is executed. First, a unique random string is generated to create a workspace for the current request, so that the files of multiple simultaneous requests don’t conflict with each other. After that, the template files for atomic and coupled models are copied into this newly created folder. Then, the atomic and coupled array objects are filtered from the request body, which is a simplified JSON sent from the front end. And these filtered objects are used to call the genrateCode to generate the source code. After the header source files are generated, the main.cpp and CMakeLists.txt files are generated using the Freemake template library of java. Finally, to return the multipart-data response to the front end the processOutput file is called with the project ID (the unique random string).

1. **public** ResponseEntity<**byte**[]> generate(@RequestBody Model data) **throws** IOException, InterruptedException,
2. IllegalArgumentException, IllegalAccessException, NoSuchFieldException, SecurityException {
3. String projectID = UUID.randomUUID().toString();
4. String projectDir = root + "/" + projectID;
5. File dir = **new** File(projectDir);
6. dir.mkdir();
7. Files.copy(Paths.get(root + "/model\_atomic.hpp"), Paths.get(dir.getAbsolutePath() + "/model\_atomic.hpp"),
8. REPLACE\_EXISTING);
9. Files.copy(Paths.get(root + "/model\_coupled.hpp"), Paths.get(dir.getAbsolutePath() + "/model\_coupled.hpp"));
10. Files.copy(Paths.get(root + "/iestream.hpp"), Paths.get(dir.getAbsolutePath() + "/iestream.hpp"));
12. File simplifiedModel = **new** File(projectDir + "/simplifiedModel.json");
13. FileWriter writer = **new** FileWriter(simplifiedModel);
14. ObjectWriter ow = **new** ObjectMapper().writer();
15. writer.write(ow.writeValueAsString(data));
16. writer.close();
17. **for** (Object atomic : data.getAtomic()) {
18. LinkedHashMap<String, Object> map = (LinkedHashMap<String, Object>) atomic;
19. String a = ow.writeValueAsString(atomic).replaceAll("\"", "\\\\\"");
20. generateCode(a, projectID, map.get("name") + ".hpp", "atomic");
21. }
22. **for** (Object coupled : data.getCoupled()) {
23. LinkedHashMap<String, Object> map = (LinkedHashMap<String, Object>) coupled;
24. String c = ow.writeValueAsString(coupled).replaceAll("\"", "\\\\\"");
25. generateCode(c, projectID, map.get("name") + ".hpp", "coupled");
26. }
27. HashMap<String, Object> map = (HashMap<String, Object>) data.getTop();
28. String absolutePath = "src/main/resources/templates/main.ftlh";
29. writeMain(map, absolutePath, projectDir + "/main.cpp");
30. String cmakePath = "src/main/resources/templates/cmake.ftlh";
31. writeCMake(projectID, cmakePath, projectDir + "/CMakeLists.txt");
33. **return** processOutput(projectID);
34. }

**Process Output**

This method constructs a byte array response entity from the workspace folder for the current request. The method takes the project ID as the argument and iterates through all the files of the folder and puts them in the zip file object. Before adding the files to the zip file, the cogapp comments are removed from the generated source files using the regex pattern for the comments.

1. **private** ResponseEntity<**byte**[]> processOutput(String projectID) **throws** IOException {
2. File output = **new** File(root + "/" + projectID);
3. List<File> filelist = **new** ArrayList<File>();
4. FileFilter modelFileFilter = (file) -> {
5. **return** !file.getName().contains("model") || file.getName().contains("top\_model");
6. };
7. **for** (File f : output.listFiles(modelFileFilter)) {
8. **byte**[] raw = Files.readAllBytes(Paths.get(f.getAbsolutePath()));
9. String clean = **new** String(raw, StandardCharsets.UTF\_8).replaceAll("/\\\*[^\*]\*(?:\\\*(?!/)[^\*]\*)\*\\\*/|//.\*",
10. "");
11. FileWriter writer = **new** FileWriter(f);
12. writer.write(clean);
13. writer.close();
14. filelist.add(f);
15. }
16. ZipFile zf = **new** ZipFile(filelist);
17. FileUtils.deleteDirectory(output);
18. **return** **new** FilesResponse(projectID + ".zip", zf.toByteArray()).response;
19. }

**Deployment:**

Login to the server devs-simulators.sce.carleton.ca with the credentials. The type of this command to navigate to the deployment directory.

1. cd /var/lib/tomcat9/webapps

Copy and paste the front-end source code in this folder. This will be the path to the web-application. Suppose we create a folder called DEVS and paste the source code in this folder. So, the path to web-application is devs-simulators.sce.carleton.ca/DEVS

To deploy the backend application, first we need to build the WAR executable from the SpringBoot application. To build executable run following command,

1. mvn package

The executable WAR will be created in the target folder of the springboot application. Copy and paste this file in the webapps folder of the server.

Top of Form