CS2820: Sprint Project:

Sprint3
Updated as of 05/05/2023

Reminder: Don't forget to fill out your Team Evaluations!

Author: Steve Goddard

ReliabilityAnalysis(Program program):

This makes a call to create the reliability table for the data in createReliabilityAnalysisDataTable.

Team:

- Payton Lovan
- Noah Lawrence
- Minhyuk Lee
- Carter Schminke

Task Assignment & Method:

For our Agile Process, we will be doing **Pair Programming**, where we will delegate tasks into (at most) even teams. *More information on the tasks will be shown below this section.*

Team 1: Payton Lovan & Minhyuk Lee
Team 2: Noah Lawrence, Carter Schminke

Implementation Tasks for ReliabilityAnalysis:

Team 1 - Implement getReliabilityAnalysis

Team 2 - Implement ReliabilityAnalysis(Program program)

Teams 1 & 2: Implement createReliabilityAnalysisTable

Implementation Tasks for ReliabilityVisualization:

Team 1 - Implement createHeaders (Done!)

Team 2 - Implement createColumnHeaders (Done!)

Teams 1 & 2 - Implement createVisualizationData (In Progress..)

Testing Tasks:

Team 1 - Test for headers (graphName,schedulerName, e2e, M, nChannels) **Additionally, the headers can be tested individually if needed.**

Team 1 - Test for nullHeaders

Team 2 - Test for ColumnHeaders (FlowNames)

Team 1 & 2 - Test for reliabilityData (Probability of sink nodes, and updated nodes)

Team 1 or 2 - Test for PriorityOrder timeslots/flowNames

Team 1 - Test for DefaultPriorityOrder timeslots/flowNames

Team 2 - Test for NonPriorityOrder timeslots/flowNames

Overview & Problem:

WARP does not have an Analysis to check on the End-to-End (E2E) reliabilities, so we will need to implement this for WARP so that the communications/messages of nodes are visible to see when it's updated! We will also need to check the probability of the message at that node when it's updated at that time.

Helper Methods & Classes for Implementation:

- ProgramVisualization (Helps with ReliabilityVisualization)
- WarpInterface (Helps with a trace of code)
- VisualizationObject (helps in terms of creating the visualization data) **[Look for not implemented methods]**

Terminology & Dictionary:

Listeners:

Also known as **Observers**, they ‘watch’ for any updates for Methods.

Sink Node:

Sink nodes are nodes where they have no edge to go to, this can be a node that is in the middle of two end nodes (start and end) that **DOES NOT** point back to those nodes or it can be an individual node without any edges.

Tasks to be Implemented in ReliabilityAnalysis:

getReliabilityAnalysisTable:

To get the reliabilityAnalysis, we will need to obtain a reliabilityTable which gets the reliabilityAnalysisData table, **more information will be provided in the buildReliabilityAnalysisTable on how to get the data.**

buildReliabilityAnalysisTable:

Retrieving the nodes in each time slot is crucial to get the node probabilities when they are updated. This can be accessed by using the **ArrayList<InstructionParameters> (String instructions)** which iterates over the **ProgramSchedule** table entries.

We can also access the **InstructionParameters()** for the names of the data, such as the flowName, the src, the sinkNodes, and channels.

To create the ReliabilityAnalysisTable, we will need to implement the data of these flows, which for each flow in src -> sinkNode, can be iteratively used by this formula:

```
newSinkNodeState = (1-M)*prevSnkNodeState + M*prevSrcNodeState
```

This value represents the probability that the message has been received by the node, SinkNode.
Where the newSinkNodeState probability will increase each time a push or pull is executed with SinkNode as a listener/observer.

The number of rows will be the number of timeSlots, and each Column will be the name of the FlowNodes.

overrideReliabilities:

If the values in the current timeslot of the table are less than their corresponding value in the previous timeslot, then the new values are overridden with the previous higher value. The method carryForwardReliabilities takes timeSlot, nodeMap, and reliabilityTable as an parameter. timeSlot indicates the newly finished time slot of the table value, and nodes/table for test.

Class RANode:

Class RANode is implemented to make easier access to nodes of the ReliabilityAnalysis. It creates getters for node column index, node phase, and checks the condition if node is a source code or not. RANode class is also used to set the phase of the nodes if needed.

Class RAMap:

Class RAMap is implemented to make the hashmap of nodes in ReliabilityAnalysis.

Tasks to be Implemented in ReliabilityVisualization:

createHeader (override): [Done!]

Headers like the graph name, Scheduler Name, M, E2E, and nChannels will be implemented and visualized by using strings or descriptions here.

createColumnHeader(override): [Done!]

ColumnHeaders are where the FlowNames will show for the nodes, which is implemented by using an array of strings that are converted from the reliabilityAnalysisDataTable.

createVisualizationData (override): [In Progress..]

Visualization data will make a call to ra.getReliabilities, where afterward, we convert the reliabilityTable to a 2D String array.

Things to Test (more to be added):

testForHeaders:

This JUnit Test will test out if the strings match the headers, by using comparisons or equalTo, we can check if the Graph has the headers, such as the E2E, M, SchedulerName, and the graph name to be identical to where they were grabbed from. **Ideally, this may be split into multiple tests, which can be testForE2E, testForM, testForSchedulerName, testForGraphName.**

testForColumnHeaders:

This JUnit Test will be testing out the FlowNames and checking if those FlowNames are correct and identical to the expected string of FlowNames/ColumnHeaders.

testForNullHeaders:

This JUnit Test will be testing out if the strings are null, by using comparisons, equalTo, or isNull, we can check if the Graph has no headers, such as E2E, M, SchedulerName, and if the graph name has nothing. **Ideally, this may be split into multiple tests, such as testforNullE2E, testForNullM, testForNullSchedulerName, and testforNullGraphName.**

testForReliabilityData:

This JUnit Test will be testing out the reliability analysis data that is received from the ReliabilityTable. If it is equal to the expected value of the probability, then it will pass, if not, it will fail.

testForPriorityOrder:

This JUnit Test will be testing out if the ColumnHeaders/FlowNames are in PriorityOrder. We will implement an expected Order from a StressList and if it matches, the test will pass, if not, the test will fail.

testNumRows:

This JUnit Test will test if the rows in the reliabilityTable making sure it has the correct size for the ReliabilityAnalysis

testNumColumns:

This JUnit Test will test if the columns in the reliabilityTable, making sure it has the correct size for the FlowNames.

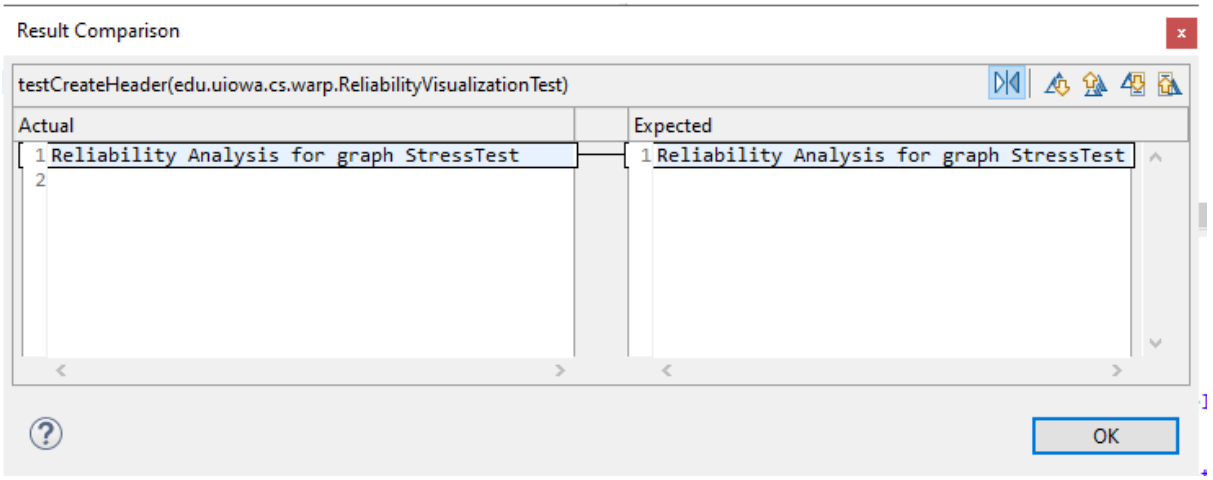
testNullTable:

This JUnit Test will test if the ReliabilityTable is null or not.

testFullReliabilityAnalysisTable:

This JUnit Test will test if the ReliabilityTable is full or not.

Debugs to fix:



There seems to be an AssertionError in our testForHeaders, as there is both an extra line and extra spaces from the Actual than the Expected..