Final Document for Project

Reliability Analysis and Visualization for WARP

Objective: Analyze and visualize the end-to-end reliability of message transmissions within the WARP system

Maintainer: Maria Gauna

Developers: Nancy Nahra, Tommy Looi, William Lucas

The team worked closely during this phase of the project, frequently falling back on a pair- or group-programming arrangement to discuss code issues and overcome hurdles that arose during the implementation of buildReliabilityTable. Much of the work of this project was done in a group setting with most or all members present and contributing via discussion; the "roles" below are to be taken in a general sense, as the group often helped out with tasks other members were "in charge" of.

- William and Maria focused on implementing the majority of methods within ReliabilityAnalysis.java. Methods implemented were listed in the project's README.md.
- Nancy and Tommy focused on testwriting, updating artifacts for the Sprint, and JavaDoc comments/generation. Tests were listed in the project's README.md.

buildReliabilityTable() Method Development Process

1. Created the Header Row:
    a. We started by creating a header row for our table, based on the different flows and their nodes. This header row serves as a reference, so that we have something to print at the top of the screen when we need to visualize the reliability table.
2. Build HashMap for Indexing:
    a. To improve efficiency, we created a HashMap from the header. The keys were flow names concatenated with the sink (flowName + snk), and the values were their corresponding indices. This approach allowed us to quickly retrieve indices without repeatedly iterating through an Array List.
3. Initialized Dummy Row:

    a. We created a dummy row to serve as a baseline for calculating the values of the first row. This row provided the initial values needed to perform our first row computations.
4. Calculate the First Row Separately:
    a. Recognizing that the first row's computation was different from subsequent rows, we calculated it separately. So it made no sense for us to include it in the iterations computing the subsequent rows. This is why we decided to do this method separately.
5. Build the ReliabilityTable:
    a. After determining the key features of the method, we implemented the following steps to build the reliability table:
        i. We iterated through each time slot of the scheduler table, as the number of time slots corresponded to the rows in the reliability table.
        ii. For every row iteration, we created a new row, updated with the following iterations, and added it to the reliability table
    b. We iterated through each column of the scheduler table to retrieve the instructions array. This step was necessary to access the instructions needed to compute the reliabilities.
    c. For each instruction in the array we: retrieved the flowName, checked whether the instruction was a “push” or “pull”, and preformed calculations based on whether the period has changed. This is because period would change what the values of the row should be so we need an extra case for this instance.
        i. If the current flow was a period reset, we calculated the newSinkNodeState using prevSinkNodeState, with the values prevSrcNodeState = 1.0 and prevSinkNodeState = 0.0. This was because the reliability was reset, so the initial source and sink values were used.
        ii. If the flow had not reached the end of its period, we calculated the newSinkNodeState using the prevSinkNodeState and prevSrcNodeState from the previous row of reliabilities.
    d. Added Rows to the Reliability Table:
        i. After computing the row, we added it to the reliability table. This ensured the table was updated correctly with each row iteration. The, it resets the prevReliabilityRow, so that there is no conflicts with computations.
    e. Simultaneous Testing:
        i. While writing the code, we simultaneously wrote test cases, especially for helper methods. This approach allowed us to validate our methods in real-time and address issues as they come up.
    f. Started the ReadMe Documentaion:
        i. We began drafting the README.md

g. Updated Sequence Diagrams:
    i. We updated our sequence diagrams to reflect the class calls made during the ReliabilityAnalysis process.
h. Revised the UML Diagrams and Project Plan Documentation:
    i. After adding new methods, we updated the UML diagrams and completed the project plan documentation.
i. Parallel Testing and Documentation:
    i. While some team members worked on steps g-h, others continued writing additional test cases.
j. Final Updates:
    i. Finalized the README.md and merged all changes into the main branch.

buildReliabilityTable() Mistakes

One mistake we made before the final process was starting with the incorrect assumption that each column of each row contained only one instruction in its array. The test cases we initially used to verify our process did not include examples that contradicted this assumption. Consequently, we computed reliabilities based on whether the instruction included a “push” or “pull” We assumed that every unused instruction always had a “push” but not a “pull”. To compute the reliability, we checked whether “pull” had an index in the instruction. If it did, we calculated the reliability of the flow along with the name of the source. Simultaneously, we always computed the flow name along with the sink. This approach worked for most of the examples we tested; however, it produced incorrect outputs in cases where the “pull” flow name did not match the “push” flow name. For example, if the “push” was F1: A ->B, we expected the corresponding pull to be F1:C->A. However, in some cases, the “pull” would instead be something like F5**:**C->A, which caused us to place the “pull” in the wrong index in the table, leading to wrong calculations. This issue was resolved when we met with Professor Goddard, who explained that our assumption was incorrect and helped us correct our logic.

Another mistake we encountered was in creating the first row of our reliability table. This worked for almost all example cases except one, and we initially couldn’t figure out why. Upon reviewing the schedule for that program, we discovered that the issue was due to the program beginning with an unused instruction instead of a used one. This caused an error in our calculations: instead of assigning the source and sink values as 1.0 and 0.0, respectively, we ended up with both values as 0.0 because the column indexing was incorrect for that scenario. We resolved this issue by no longer relying on the column as our index for placing values. Instead, we searched for the index of the flow name we were modifying and used that as our reference point.