SE 320 Homework 5 Write-up

Consider the process of creating an event flow graph covering traces with up to 1 item in the todo list. What kinds of issues would you encounter in creating such a model? Which events have restrictions on their ordering relative to other events (i.e., which events would not have arrows directly between them because they would require other things to happen in between)?

There are a few issues that are encountered when creating an event flow graph covering traces with up to 1 item in the todo list. First, the “add to list” button would always be present, even though this model assumes that up to one item can be added. Thus, the event flow graph for this scenario would bounce between “add” and “delete,” where each “add” presents a “delete” and removes the “add,” and each “delete” presents the “add” again and removes the “delete.” The amount of finite states here is 2 (“no item” and “item”), which is not representative of the actual GUI application — we never actually reach the maximum state where no more items can be added. Furthermore, adding more than one item should display said item with its respective “delete” button after the previous items. By only tracing one item, we encounter issues where multi-item list ordering is not tracked appropriately, thus not exemplifying the “delete” functionality per each item in order. Creating such a model is difficult because certain states that are expected from the GUI application will never actually occur under the assumption of one item max. We will never see the “add” button go away, meaning we are representing a functionality that has limitations in such a way that said limitations are not represented in the event flow graph. Additionally, the “delete” function is also represented in a binary manner in this one-item event flow graph, where it can be pressed and said item no longer exists. However, the “delete” functionality can be exercised many times before any “adds” occur again, meaning the one-item event flow graph is inaccurate even further to the full capability of the GUI application.

Assume you did draw out the event flow graph above. That would be a model of the UI that assumes there is never more than one todo item. But per your tests above, you know the application actually allows up to 3. Please consider the consequences of this hypothetical mismatch. If this event flow graph modeling a maximum of 1 list item were the basis for all of your tests, what sorts of problems are unlikely to be discovered? Can you describe a test whose behavior would not be captured by such an event flow graph?

Having an event flow graph of only one item max be the basis of testing the GUI application that, in actuality, allows up to three items max, would have consequences. Some of these consequences were touched upon in the previous prompt, but I will order them here with some additional consequences in more full detail. If there were issues with the ordering of the to-do items with more than one item, this issue would not present itself. Do additional items get appended properly? Will the previous to-do items accidentally get altered upon adding additional items? Furthermore, having one item max would not allow for accurate testing of the “delete” functionality. What if adding more than one item made each “delete” actually delete the wrong item? Also, ensuring that the “add” functionality goes away when the max items are reached could not be verified, as a one-item max list will never reach the max that the GUI application can handle. Similarly, this one-item list could also not verify that the “add” functionality reappears once the max items are reached and then an item is deleted. As mentioned previously, a one-item max event flow graph can only go between “add” and “delete,” where one item is the most you will ever see in the list. This is quite unrepresentative of the actual GUI application and does not expose the various other events that can occur with more than one item and the potential problems having this type of functionality might invoke. There are a couple tests I can think of that a one-item max event flow graph would not test for. One I touched upon previously is that items cannot be added after the max has been reached. If a test was written to add the max amount of items, or even just more than one item, the one-item event flow graph could not trace this behavior. Additionally, having multiple items means having multiple “delete” buttons. While the one-item event flow graph can test that said one item is deleted, it cannot test multiple deletes or deletes of items other than the one item that exists in the list. The actual GUI application is meant to have up to three items in it, which results in a larger and more complex event flow graph than that of one item max. Events will flow into other events and back around, and certain events have stipulations. While the one-item event flow graph can trace the basis of the functionality, it will be very limited in that “add” and “delete” work together back-and-forth, and does not expose the complexities of all possible events.

Conversely, are there any ways in which the fixed size assumption might simplify testing in a good way? Does it encourage you to write tests for impossible scenarios? Would any of these issues be affected by working with a larger event flow graph (say, up to 5-element TODO lists)?

While having a one-item size assumption proves to have its consequences, there are benefits to having some fixed size assumptions. With the current state of the to-do GUI application, we can have three items max. However, a useful and practical to-do application would have a max that is much larger. Writing the types of tests that were done here would have to be scaled up largely to accommodate for this, and would potentially change each time this limit is increased. This is where a proper fixed size assumption can come in handy. With the current GUI application, we have a few properties that can be partitioned. First, we have a minimum, which is no items in the list. Conversely, we have a maximum, which is currently three items in the list. Then, we have a case which I will call “some” items. For this, it is whenever we have one or two items in the list. This is where we are neither at the minimum or maximum number of items for the list. In the case of no items, we are able to add items. With the case of max items, we are able to delete items. In the case of “some” items, we are able to both add and delete items from the list. This type of partitioning will apply to a to-do list application where the minimum of max items that can be added is two, and the maximum of max items that can be added is a number much larger than three. While the min and max are important edge cases, as their functionality has specificities, the number of items in the list besides these edges is almost irrelevant as long as you are partitioning for this case appropriately. Being able to test the “add” and “delete” functionality when not at the min or max should be the same whether the max is three, five, or one thousand. The “state” of this non-edge case (being able to “add” and “delete") is all-encompassing. By doing this, the core functionality of the GUI application is tested without having to potentially write an impossible number of tests for lists that can handle an extreme number of items. Scenarios that are both just a small increase (e.g. 5-element to-do list) and impossible (e.g. million-element to-do list) have the capability of being tested by having a fixed size assumption and partitioning the tests in a manner that is effective in regard to the core functionality of the list application.