**Generating System Test Cases from UML Interaction Diagrams**

|  |  |  |
| --- | --- | --- |
| Nephi Calvin Bandela,  *College of Computer & Information Science,*  *Northeastern University,*  *Boston, MA.*  *bandela.n@husky.neu.edu* | Nikhil Reddy Sudireddy  *College of Computer & Information Science,*  *Northeastern University,*  *Boston, MA.*  *sudireddy.n@husky.neu.edu* | Michael Weintraub,  Associate Clinical Professor,  *College of Computer & Information Science,*  *Northeastern University,*  *Boston, MA.*  *m.weintraub@northeastern.edu* |

**Introduction:**

System testing is crucial to any project. Good testing promotes confidence in a system, especially if the number and scope of test cases are - robust and the system passes every test case. Creating sufficient test suites is a challenge for a project. Often test creation and testing itself are deferred in favor of design and implementation, which creates challenges for delivering tested software. It would be beneficial to be able to generate test suites, especially system tests directly from designs. As UML diagrams [1] are a useful design tool, it makes sense that generating test cases directly from UML would be useful. This paper describes an extension to previous work that first proposed how to generate system test cases from simple UML Interaction/Sequence diagrams –to include sequences that include both branches and loops. Further, we describe an extension that introduces the chance for generating issues with the communications between modules.

**Background and Related Work:**

There has been work that generates test cases from UML diagrams. [2] describes an approach that converts an UML activity diagram to Input Output Activity Diagram, which is then converted into a directed graph and from where the test cases are derived. [3] proposes an approach that uses collaboration diagrams to produce the test cases. [4] proposes an approach that transforms sequence diagrams into graphs, called a *sequence diagram graph*, and test cases are derived by traversing this graph.

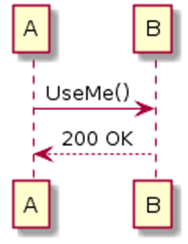


Figure 1: A Simple Sequence Diagram

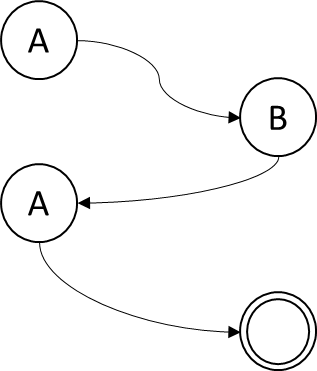


Figure 2: Corresponding Sequence Diagram Graph

**Approach:**

[4] is very interesting. However, the scope of the UML it covers is simple sequence diagrams. Moreover, while UML naturally supports distributed object systems, the graphs being created do not include negative path situations such as failures that happen in practice. Our work extends the sequence diagram graphs to capture a subset of negative path situations, in particular inter-object/system communication failures, and sequence diagrams that include conditional (ALT) or loop constructs. To illustrate, consider a simple UML sequence diagram shown in Figure 1 where entity A invokes the *UseMe* service of B and B returns a typical http 200 OK response.

Building off [4], we take as input a UML sequence diagram and then construct the corresponding directed graph. The nodes in the graph represent components links represent messages passing between them. Scenarios are then composed by different traversals of the graph. In effect, each scenario is a separate system test case. For our example, the sequence diagram given in Figure 1 would result in a correspondingly simple sequence diagram graph, shown in Figure 2.

Figure 3: The sequence diagram graph with communication delays or loss added

To address communication issues such as delay or loss (time-outs), we introduce a new node (*Delayi,j*) into each link in the graph to represent that some time is needed for the message to traverse the network from nodei to nodej. To represent losing a message, one can set the delay time to be infinite or one may consider adding an alternate path off *Delayi,j* to a sink state, which would accomplish the same thing. We describe *Delayi,j* in terms of some distribution of time. Appropriate action is taken based if *Delayi,j*  exceeds a threshold. If one is not concerned with delay or loss between two nodes, we simply set *Delayi,j* to 0. Figure 3 illustrates this for the simple sequence diagram originally shown in Figure 1. Here we chose to represent

Many sequence diagrams rely on some kind of conditional control flow, and UML allows for branching (Alt) and looping (Loop). Alt specifies a condition which results in two different paths whereas a Loop specifies a condition which iterates over a set of events until the terminating (or break) condition is met. So, each Alt condition doubles the number of paths whereas a Loop repeats over a single path for the specified set of events. Two graphs are generated when Alt is used in the interaction diagram denoting the alternate paths. For Loop a single graph is generated which includes a self-loop with the terminating condition over a sub-path.

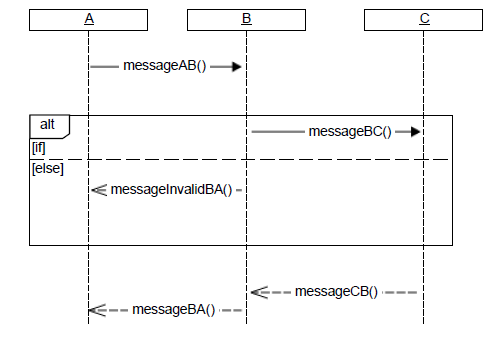


Figure 4: A Sequence diagram with “alt”

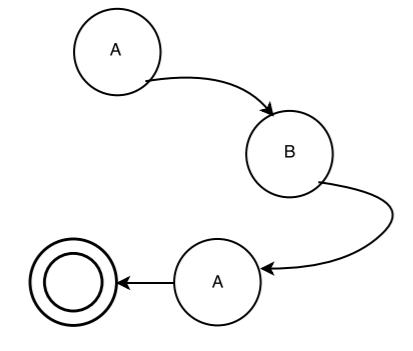
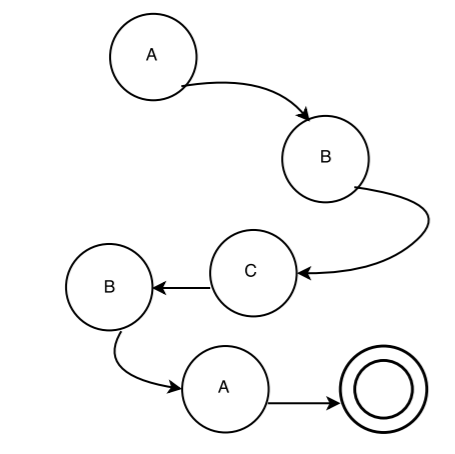


Figure 5: Corresponding diagrams – alt

As shown in Figure 5, the Alt in the Interaction diagram (Figure 4) produces alternate graphs.

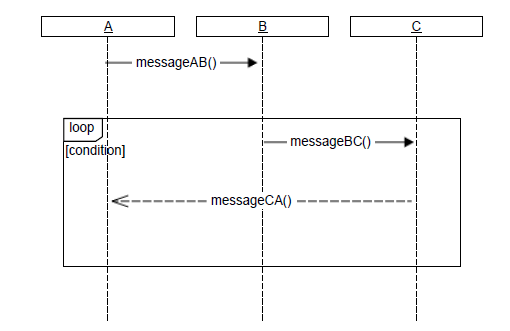


Figure 6: A Sequence diagram with “loop”

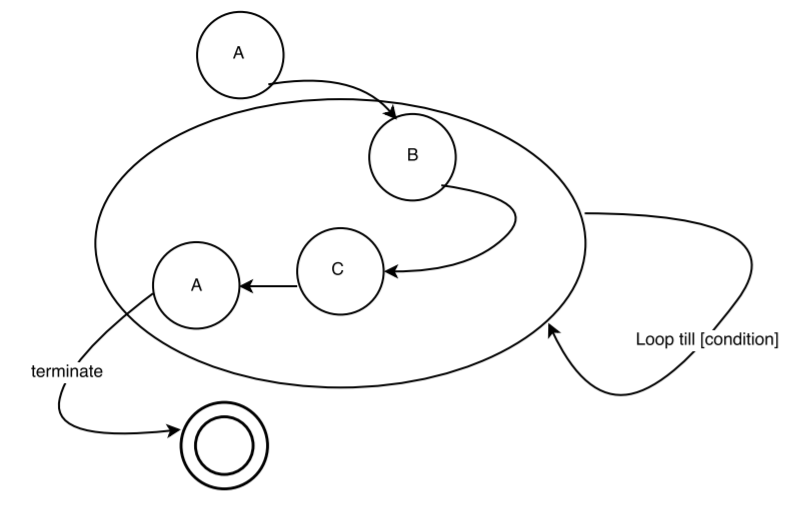


Figure 7: Corresponding diagram - loop

As shown in Figure 7, the loop in the Interaction diagram (Figure 6) produces a graph with iteration over a sub-path[B-C-A].

**Implementation: TestCaseGen**

We have implemented these extensions in a system called TestCaseGen*.* It takes as input a UML interaction diagram expressed as XML and produces test suites. Most UML tools produce XML. TestCaseGen presently works with the XML produced by *Papyrus* and produces test suites in Java, with a target test tool like JUnit. TestCaseGen starts by parsing the XML file and retrieves the relevant information about lifelines (nodes from which messages are sent) and messages. TestCaseGen identifies components and messages by an ID. TestCaseGen groups each message with a source and destination component. This combination of (message ID, source ID, destination ID) is then considered as an *event*.

Next, TestCaseGen creates scenarios by grouping *events* together using each *event’s* source and destination. In effect, a scenario denotes a sequence of operations (messages), which becomes a test case within the downstream test suite.

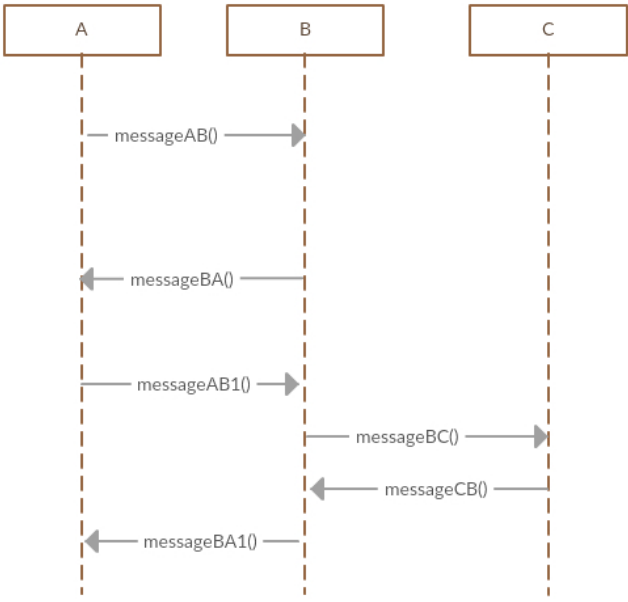
**

Figure 8: Simple Interaction Diagram

For example, consider the simple interaction diagram shown in Figure 8 where there are three components. There are four events (described as *source, destination,* and *message*):

1. (A, B, messageAB)
2. (B, A, messageBA)
3. (A, B, messageAB1)
4. (B, C, messageBC)
5. (C, B, messageCB)
6. (B, A, messageBA1)

These events may be combined as a scenario:

Scenario: {(A, B, messageAB), (B, A, messageBA), (A, B, messageAB1), (B, C, messageBC),

(C, B, messageCB), (B, A, messageBA1)}

This scenario become one system test case. TestCaseGen provides the following graph for the Interaction Diagram in Figure 8:

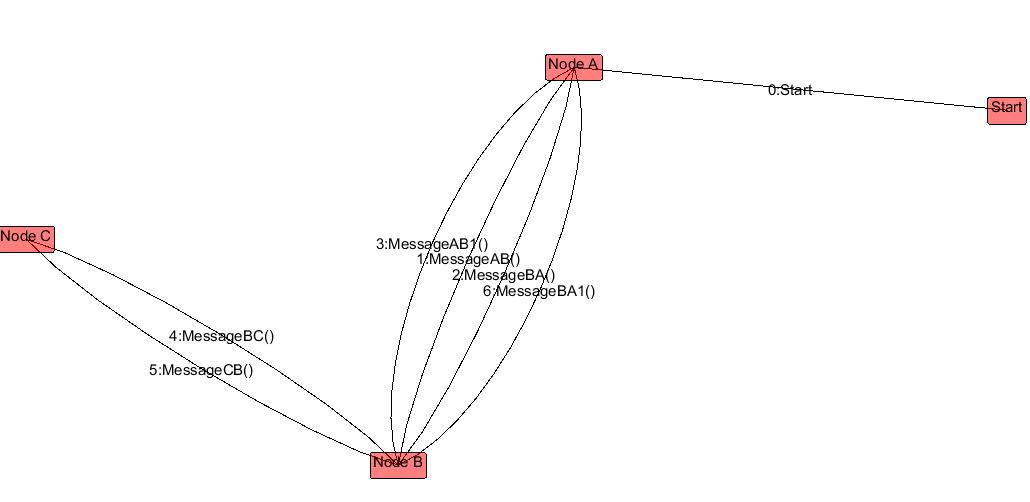


Figure 9: Graph Generated for Figure 4 using TestCaseGen

In UML interaction diagrams, branching and looping is implemented by using ‘*alt’* and ‘*self-loops’* respectively. Given a sequence diagram which contains an ‘*alt’*, TestCaseGenintroduces a conditional statement at the beginning of the *‘alt’*. This conditional statement uses the *guard* condition extracted from the XML*,* which creates two scenarios – one for the condition being satisfied and one for the condition failing. In effect, each *alt* doubles the number of scenarios.

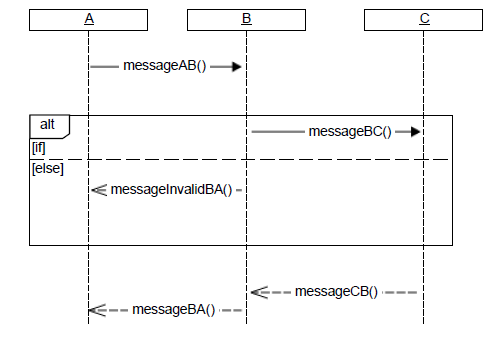


Figure 10: Interaction Diagram with ‘alt’

For example, consider the interaction diagram shown in Figure 10 with the conditional operator, the scenarios are split into two: one for ‘if’ and ‘else’ conditions respectively as shown below:

Scenario 1: {(A, B, messageAB()), (B, C, messageBC()), (C, B, messageCB()), (C, B, messageCB())}

Scenario 2: {(A, B, messageAB()), (B, A, messageInvalidBA())}

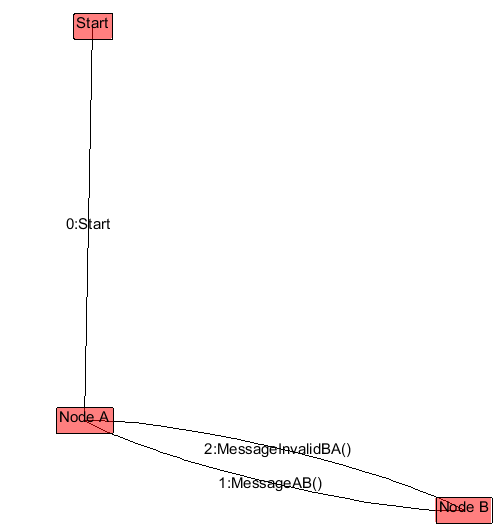
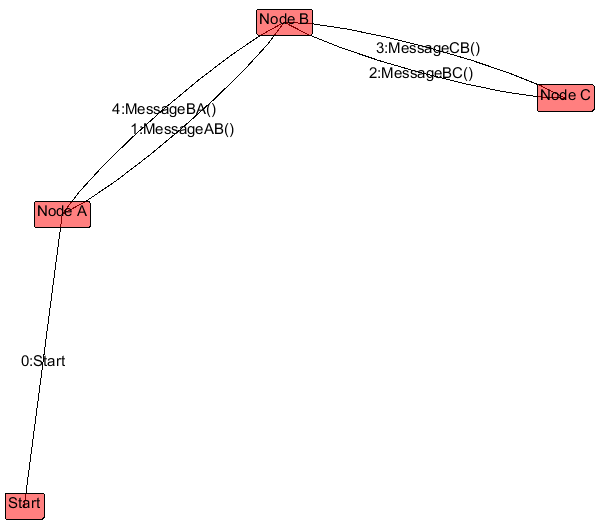


Figure 11: Graphs Generated for Figure 6 using TestCaseGen

Handling loops is done in a similar way. We introduce a conditional loop statement at the beginning of *loop.* This conditional loop statement uses the *guard* condition extracted from the XML, and loops over a set of events until the loop terminates. In contrast to branching, loops do not multiply the number of scenarios, a single scenario exists throughout the loop.

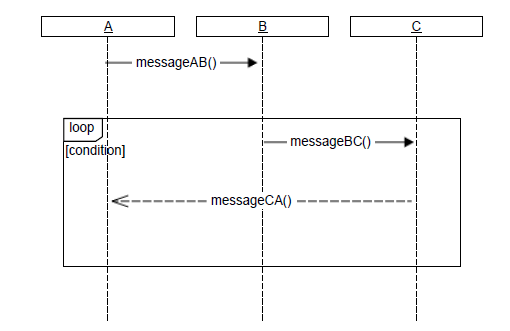


Figure 12: Interaction Diagram with ‘loop’

For example, consider the interaction diagram shown in Figure 11 with the conditional loop operator, the specific events within the loop condition are repeated till the terminating condition is met.

Scenario 1: {(A, B, messageAB()), [(B, C, messageBC()), (C, A, messageCA)]\*}

where \* denotes iteration of events 0 or more times based on the loop condition. This is explained in action in the below flow chart (Figure 4).

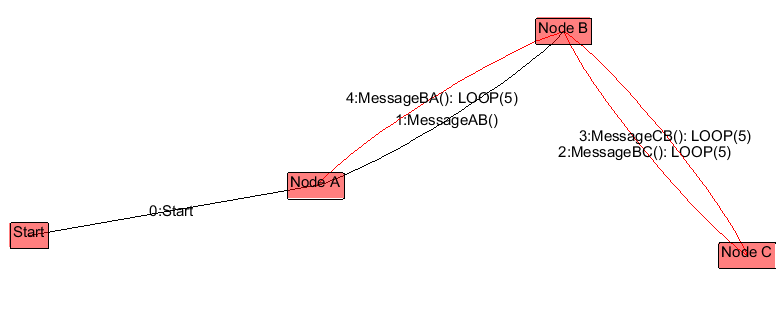


Figure 13: Graph generated for figure 12 from TestCaseGen. The sub path within the loop is shown in red(edges) with the loop count specified, here it is 5

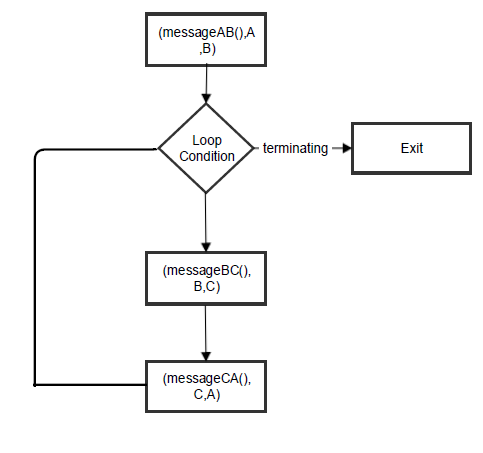


Figure 14: Flow chart of ‘loop’

As described, system failures that result from issues with message passing – such as congestion conditions – is handled by introducing a delay into an event. Thus, the length of a *scenario* effectively doubles as the test case moves from componenti sending messagem to componentj.  We shorten this expression to (componenti, componentj, delayi,j,*,* messagem). In effect, the delay node acts as a conditional block acting on a specific condition. This condition is generally the delay (time in seconds/milliseconds) that is allowed for a message to reach from source to destination. Appropriate action is taken based on this condition. For certain instances, where the interacting modules do not require a ‘delay/loss’ condition, we manipulate the condition to always return true.

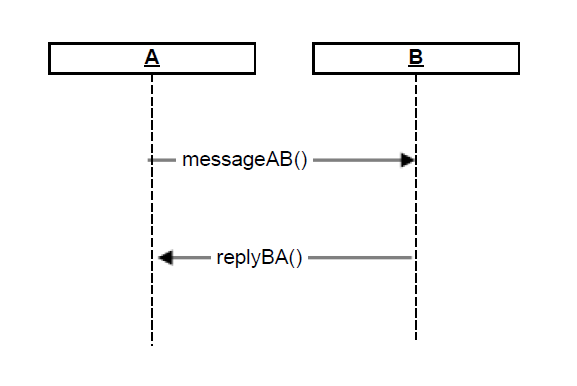


Figure 15: Negative Path Example

For the above diagram a scenario consists of:

Scenario: {(messageAB(), A, d, B), (replyBA(), B, d, A)} where ‘d’ is the delay node.

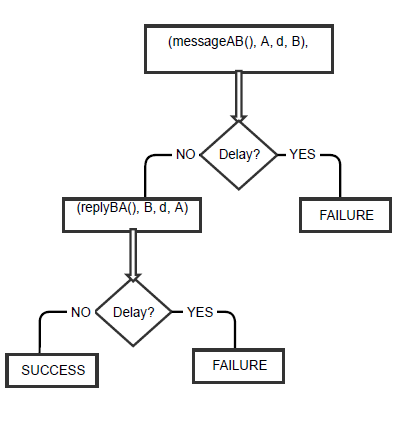


Figure 16: Negative Path Flow Chart

As shown above every event has a delay and based on the delay condition next event is either processed or goes into the failure state.

Possible Scenarios from Figure 6:

Scenario 1: {(messageAB(), A, d, B), (Failure)}

Scenario 2: {(messageAB(), A, d, B), (replyBA(), B, d, A), (Failure)}

Scenario 3: {(messageAB(), A, d, B), (replyBA(), B, d, A)}

**Conclusion:**

We have extended a methodology of [4] to include system test scenarios involving complex control paths and communications delays. Test cases are generated based on scenarios which are formed by grouping of events. An event is denoted by (message, source, destination). Delays are handled by introducing a new delay node between source and destination. Branching is handled by splitting the scenarios into other scenarios. The tool used to create UML Interaction diagram is ‘Papyrus’ which generates corresponding XML file for the diagram and is used for retrieving the required information. Based on the scenarios generated from the Interaction diagram we automatically produce a test suite which contains unimplemented methods each representing a scenario.

# References

|  |  |
| --- | --- |
| [1] | J. I. J. a. G. B. A.-W. 1. 1. Rumbaugh, The Unified Modeling Language Reference Manual, Reading, MA: Addison Wesley, 1999. |
| [2] | S. K. J. B. a. I. K. H. Kim, Test Cases Generation from UML Activity Diagrams, IEEE Computer Society, 2007. |
| [3] | A. A. a. J. Offutt, "Using UML collaboration diagrams for static checking and test generation," in *The Third International Conference on the Unified Modeling Language*, 2000. |
| [4] | D. K. a. R. M. M. Sarma, "Automatic Test Case Generation from UML Sequence Diagrams," in *15th International Conference on Advanced Computing and Communications*. |