Verifying Distributed Protocols using MSC-Assertions, Run-time Monitoring, and Automatic Test Generation[1]

Doron Drusinsky[2] and Man-Tak Shing
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943, USA
{ddrusins, shing}@nps.edu

Abstract

This paper addresses the need for formal specification and runtime verification of system-level requirements of distributed reactive systems. It describes a formalism for specifying global system behaviors in terms of Message Sequence Chart assertions and a technique for the evaluation of the likelihood of success of a distributed protocol under non-trivial communication conditions via discrete event simulation and runtime execution monitoring. We constructed a proof-of-concept prototype for the leader-election algorithm within a 4-node ring network. The prototype consists of the following components: (i) an OMNeT++ model of the network using non-trivial communication conditions, (ii) C++ code for the network agents, (iii) a system-level assertion stipulating the formal requirement for a correct, time-bound, leader election, (iv) simulation of the formal assertion, (v) automatic scenario generation, and (vi) run-time monitoring of the formal assertion and stochastic-based estimation of the likelihood of success of this assertion under the specified communication conditions.

1 Introduction

The design and implementation of reliable applications on top of asynchronous distributed systems that are prone to processor and network crashes is a difficult and complex task. A distributed system is made up of several components, executing concurrently and interacting with each other under the control of specialized procedures called protocols. Individual components usually do not have real-time knowledge of the global state of the system, and it may not even have the notion of a global clock. Moreover, whenever the application departs from its correct "state" due to processor crashes, the live processors must execute some algorithms (i.e. protocols) to restore the application back to the correct state.

Runtime Execution Monitoring (REM) is a class of methods for tracking the temporal behavior of an underlying application. REM methods range from simple print statement logging methods to run-time tracking of complete formal requirements for verification purposes. NASA used REM to verify the flight code for its Deep Impact project [5]. A recent paper by the authors describes run-time verification of the CARA infusion pump using UML-statechart models combined with statechart-assertions for formal requirement specification [4].

Often, distributed-system protocols are correct for an ideal system but do not operate as well in a less than ideal situation. For example, while the classical leader-election algorithm in a ring network is considered correct, a leader might not be elected within reasonable amount of time when the network suffers from significant communication delays. This paper addresses the problem using REM-based techniques.

In [2], we introduced a classification of formal assertions into the following three categories: (i) test-time assertions, (ii) run-time assertions and (iii) simulation-time assertions. Simulation-time assertions are assertions that use information about the environment not present in run-time. Simulation-time assertions are particularly useful for the validation of global, emerging behaviors of distributed systems, where the global information of the distributed system is unavailable to individual nodes. Modeling and simulation holds the key to the early use of these

[1] The research reported in this article was funded in part by a grant from the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

[2] Also with Time-Rover Software Inc. www.time-rover.com

assertions to validate system behaviors of distributed systems. For example, prototypes augmented with simulation assertions will often be used to force catastrophic behavior of the kind only available in simulation mode.

Harel statecharts were first described in [6]. They are typically used for design analysis and implementation. In his recent book, Drusinsky suggested using statecharts-assertions for formal requirement specification and run-time verification of UML-statechart controller models [1]. While statecharts are used primarily for capturing *intra*-agent behavior, UML Message Sequence Charts (MSC's) are used for capturing *inter*-agent behavior, i.e., system and distributed system behavior. To overcome the limitations of MSC as a formal specification language, researchers have introduced different variants of MSC. For example, in [8], Harel and Morelly described Live Sequence Charts (LCS's), an MSC-like formal specification language tailored for conditional scenario specification in the form of *if a then b*, *a* and *b* being sub-scenario's. The LCS-based specifications can be translated to CTL or LTL for verification [10], or to intra-object state-based specifications for synthesis [7, 9].

This paper builds upon our previous work on statechart-assertions and REM of *intra*-agent behavior. It describes MSC-Assertions, a formal language extension of MSC's, and their application to the formal specification and REM of *inter*-agent behavior. Unlike the other approaches, MSC-Assertions, being a natural extension of the statechart assertions, provide a unified model for both *intra*- and *inter*-agent behavior specification, thus eliminating the need to translate and maintain two models (one for *intra*-agent and the other for *inter*-agent) when designing and analyzing distributed system behaviors. We demonstrate the technique with a proof-of-concept prototype for assessing the failure rate of a time-bound formal requirement for a distributed-system protocol (leader election) operating with non-ideal communication links.

Our proof-of-concept prototype is as follows:

- We developed an OMNeT++ model of a 4-node ring network with parameterized network delays, as detailed in Section 2.
- We developed a UML statechart model and generated code for the network agents, as discussed in Section 3.
- We captured a formal requirement for the timely election of a leader. To this end, we devised a formal language extension to UML Message Sequence Charts, called *MSC-Assertions*, as discussed in Section 4. Absent a code generator for MSC-Assertions we hand crafted the corresponding implementation code. In addition, we simulated the formal MSC-Assertion requirement, as discussed in Section 5.
- We created a large set of test scenarios for the 4-node ring distributed system, using white-box test generation techniques, as discussed in Section 6.
- By executing these scenarios and run-time monitoring the MSC-Assertion we calculated the failure rate of the assertion.

2 OMNeT++ network model

OMNeT++, which stands for *Objective Modular Network Testbed in C++*, is an object-oriented discrete event simulator primarily designed for the simulation of communication protocols, communication networks and traffic models, and multi-processors and distributed systems models. An OMNeT++ simulation model consists of a set of modules communicating with each other via the sending and receiving of messages. Modules can be nested hierarchically. The atomic modules are called simple modules; their code are written in C++ and executed as co-routines on top of the OMNeT++ simulation kernel. Messages are sent out through output gates of the sending module and arrive through input gates of the receiving module. Input and output gates are linked together via connections. *Connections* represent the communication channels and can be assigned properties such as propagation delay, bit error rate and data rate. Figure 1 shows a simple OMNeT++ model of a four-node ring network. The network is made up of four identical nodes (agents), whose behavior is captured using a UML statechart diagram as discussed in Section 3.

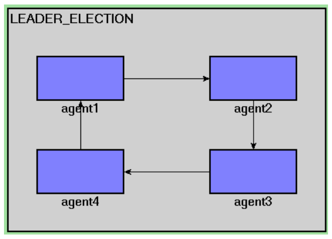


Figure 1. The 4-node ring network model in OMNeT++

3 Network agents

Figure 2a shows the object model of a network node, which is made up of an Agent class and the statechart classes generated from one or more statechart files. In other words, an agent is assumed to be doing more than merely electing a leader, hence the plurality of statecharts per agent. Figure 2b shows an instance of the network node, which contains an instance of the Agent class together with an instance of the *LE* statechart for leader election. The Agent class extends the OMNeT++ cSimpleModule class and serves as the proxy to handle all network communications for the *statechart* object.

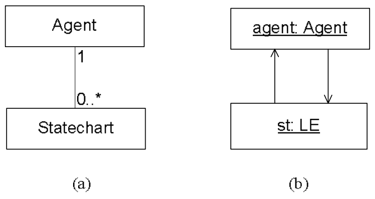
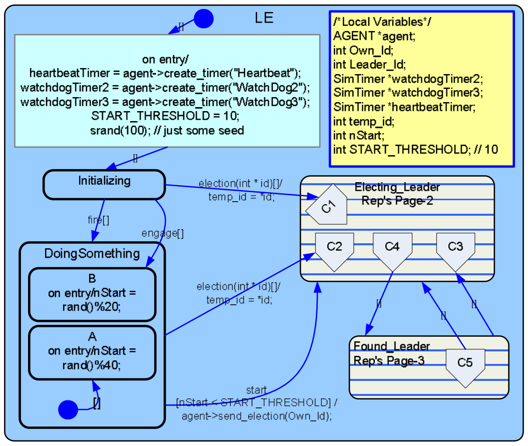


Figure 2. The object model of the network node

Figure 3 shows the top-level statechart of a leader election (*LE*) module. The top-level statechart consists of four states, the *Initializing* state and three composite states named *DoingSomething*, *Electing_Leader* and *Found_Leader*, together with a set of state variables declared in the associated local variable declaration box. As mensioned earlier, a node's statechart is modeled to do more than just leader election. Hence, we added logic to the agent's leader-election statechart to represent situations where the agent is busy performing activities other than electing a leader. The state *DoingSomething* represents (1) that the statechart, in general, has a life – it does work other than merely electing leaders, and (2) that not all agents request election when start occurs.



```
on entry/
heartbeatTimer = agent->create_timer("Heartbeat");
watchdogTimer2 = agent->create_timer("WatchDog2");
watchdogTimer3 = agent->create_timer("WatchDog3");
START_THRESHOLD = 10;
srand(100); // just some seed
```

```
/*Local Variables*/
AGENT *agent;
int Own_Id;
int Leader_Id;
SimTimer *watchdogTimer2;
SimTimer *watchdogTimer3;
SimTimer *heartbeatTimer;
int temp_id;
int nStart;
int START_THRESHOLD; // 10
```

Figure 3. Top-level page of the *LE* statechart

Each *LE* module uses the *Own_Id* variable to store its unique integer identity and uses the *Leader_Id* variable to remember the identity of the current leader, which is the largest identity value among the identities of all the active *LE* modules in the network. The *LE* statechart starts at the *Initializing* state waiting for the arrival of either the *fire()*, *engage()* or *election()* event from the environment. When it receives the *fire()* or *engage()* event, it will transition to the *DoingSomething* state. If the LE statechart receives the *start()* event while it is in the *DoingSomething* state, it starts a new round of leader election by sending an *election()* message to its neighbor in the network using the *send_election()* method of its agent object and then enters the *Electing_Leader* composite state shown in Figure 4. Alternately, if it receives the *election()* event while it is in either the *Initializing* state or the *DoingSomething* state, it will join in the leader election process and transition to the *Electing_Leader* composite state. To simulate the non-deterministic behavior of individual nodes, each LE statechart initializes its variable nStart to a random value when it enters the states *A* or *B* of the *DoingSomething* state and can start a new round of election only if nStart is less than the *START_THRESHOLD*.

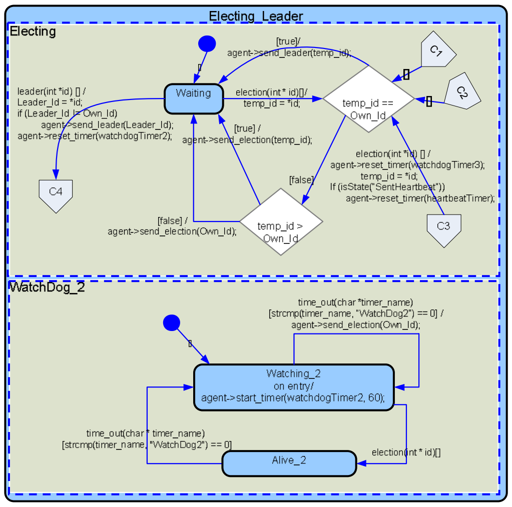

Electing_Leader
Electing
[]
Waiting
[true]/
agent->send_leader(temp_id);
C1
C2
leader(int *id) [] /
Leader_Id = *id;
if (Leader_Id != Own_Id)
agent->send_leader(Leader_Id);
agent->reset_timer(watchdogTimer2);
election(int * id)[]/
temp_id = *id;
temp_id ==
Own_Id
[true] /
agent->send_election(temp_id);
election(int *id) [] /
agent->reset_timer(watchdogTimer3);
temp_id = *id;
If (isState("SentHeartbeat"))
agent->reset_timer(heartbeatTimer);
C4
[false]
[false] /
agent->send_election(Own_Id);
temp_id >
Own_Id
C3
WatchDog_2
time_out(char *timer_name)
[strcmp(timer_name, "WatchDog2") == 0] /
agent->send_election(Own_Id);
Watching_2
on entry/
agent->start_timer(watchdogTimer2, 60);
time_out(char * timer_name)
[strcmp(timer_name, "WatchDog2") == 0]
election(int * id)[]
Alive_2

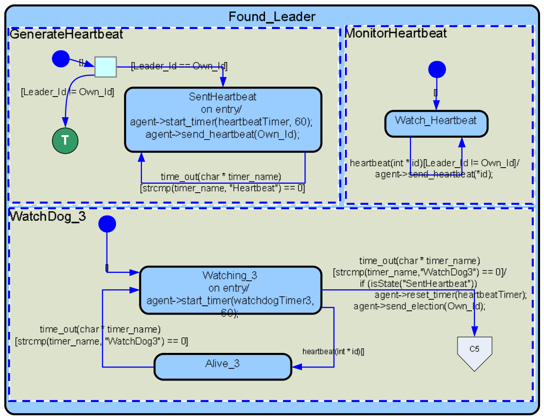
Figure 4. The content of the *Electing_Leader* statechart

The *Electing_Leader* composite state consists of two concurrent threads, *Electing* and *Watchdog_2*. The statechart in the *Electing* thread models the logic of the following simple leader election algorithm.

```
if (event == election(id)) then // on-going election
{
    if (id == Own_Id) then
        send leader(Own_Id) event to its neighbor;
    else if (id > Own_Id) then
        send election(id) event to its neighbor;
    else
        send election(Own_Id) event to its neighbor;
}
else if (event == leader(id)) then
{   // found leader, terminate election
    Leader_Id = id;
    if (Leader_Id != Own_Id) then
        send leader(id) event to its neighbor;
    reset the watchdogTimer2 timer;
    transition to the Found_Leader state
                via the page connector C4;
}
```

The statechart in the *Watchdog_2* thread makes sure that the *LE* statechart receives at least one *election()* message every 60-second cycle while it is participating in an on-going election. If the *LE* statechart receives the event *time_out(timer_name)* with *timer_name* == "*Watchdog2*" while it is in the *Watching_2* state, it will initiate another round of leader election by sending an *election(Own_Id)* message to its neighbor because it has not received any *election()* message within the last cycle.

The leader election algorithm terminates when the *LE* statechart receives a *leader(id)* message, and will transition from the *Electing_Leader* state to the *Found_Leader* state via the page connector *C4*. Note that the correctness of the algorithm relies on a reliable and fully trusted network of cooperating *LE* modules.



Found_Leader
GenerateHeartbeat
[]
[Leader_Id == Own_Id]
[Leader_Id != Own_Id]
T
SentHeartbeat
on entry/
agent->start_timer(heartbeatTimer, 60);
agent->send_heartbeat(Own_Id);
time_out(char * timer_name)
[strcmp(timer_name, "Heartbeat") == 0]
MonitorHeartbeat
[]
Watch_Heartbeat
heartbeat(int * id)[Leader_Id != Own_Id]/
agent->send_heartbeat(*id);
WatchDog_3
[]
Watching_3
on entry/
agent->start_timer(watchdogTimer3, 60);
time_out(char * timer_name)
[strcmp(timer_name,"WatchDog3") == 0]/
if (isState("SentHeartbeat"))
agent->reset_timer(heartbeatTimer);
agent->send_election(Own_Id);
time_out(char * timer_name)
[strcmp(timer_name, "WatchDog3") == 0]
heartbeat(int * id)[]
Alive_3
C5

Figure 5. The content of the *Found_Leader* state

Figure 5 shows the content of the *Found_Leader* composite state. It consists of three concurrent threads. While in the *Found_Leader* state, the leader uses the statechart in the *GenerateHeartBeat* thread to send out a *heartbeat(Own_Id)* message once every 60 seconds, and each *LE* statechart expects to receive at least one *heartbeat()* message from the leader via its neighbor every 60-second cycle. If any *LE* statechart receives the event *time_out(timer_name)* with *timer_name* == "*Watchdog3*" while it is in the *Watching_3* state, it will initiate another round of leader election by sending an *election(Own_Id)* message to its neighbor and transitioning to the *Electing_Leader* state via the page connector *C5* because it has not received any *heartbeat()* message within the last cycle. Other *LE* modules will also enter the *Electing_Leader* state via the page connector *C3* when they receive the *election()* messages from their neighbors in the network while they are in the *Found_Leader* state.

4 System-level assertions using MSC-Assertions

4.1 Limitations of UML MSC's

UML MSC's are considered informal for two primary reasons. First, as discussed by David Harel in his book on Live Sequence Charts [8], a UML MSC does not distinguish between messages that *might* be sent and those that *must* be sent. Consider the MSC shown in Figure 6: the only definite scenario that is acceptable is *req.ack.ackack,* between the respective agents. However, if for example, a *req* event is followed by an *ack* event, that in turn is *not* followed by an *ackack* event, then it is not clear from the specification whether that is an illegal scenario or not. Second, an UML 1.x MSC describes a single scenario (sequence of message events); for example, the MSC in Figure 6 captures a single scenario consisting of three messages. This limitation has been extended in UML 2.0, in which an MSC can describe a finite set of scenarios using the Loop construct. Nevertheless, even UML 2.0 sequence diagrams capture only finite sets of scenarios. From a formal specification standpoint, however, a good formalism should be able to capture an *infinite* set of scenarios. In addition, it is not clear from the MSC of Figure 6 whether the MSC generates the specified events (e.g., *ack*), or witnesses (monitors) those events.

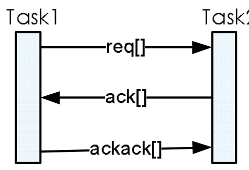


Figure 6. An MSC for the *req-ack* protocol

4.2 MSC-Assertions

MSC-Assertions are a formal-language extension of UML MSC's. They have the look and feel of UML MSC's yet are formal and executable. They are capable of making a distinction between events that *can* occur and those that *must* occur. In addition, MSC-Assertions are capable of specifying infinite sets of scenarios.

MSC-Assertions are based on MSC diagrams augmented with Java (or C++) conditions and actions. Consider the time-bound (60 second) leader-election MSC-Assertion for the requirement R1 of a 4-ring network: "*All agents contain the same ID for the elected leader, which is the largest identity value among the ID's of all the active LE modules in the network, in at most 60 seconds after the first election event detected in the network*" (Figure 7). It looks, for the most part, like a UML MSC, but it enjoys the following unique features:

1. An MSC-Assertion is written from the standpoint of an *observer*, and indeed, in this paper, we will use the MSC-Assertion of Figure 7 for run-time monitoring the OMNeT++ network model. Hence, consider for example the message *leader(Integer m1)* sent from agent #1 to agent #2. While a UML-MSC might consider an interpretation where this event is *generated* by the MSC, for an MSC-Assertion it is meant that the MSC-Assertion should monitor-for, or listen-for, this event flowing from agent #1 to agent #2.
2. An MSC-Assertion *allows loops*. In Figure 7 for example, there is a loop starting from agent #1, through agents #2, #3, and #4, back to #1. This feature is in contrast to UML MSC's where a vertical task bar represents a *timeline* and where clearly a task cannot move back in time. MSC-Assertion however, considers a vertical task bar as a *progression of states*, like a state diagram drawn vertically. It therefore permits loops.
3. *States and actions.* As discussed above and as illustrated in Figure 7, an MSC-Assertion task might contain explicit states. The purpose of these states is to specify actions, which are code snippets (written in Java or C++, depending on the code generator chosen) to be performed, such as *x++* or *n_2=m2.IntValue()*.
4. *Java/C++ underlying language and code generation*. An MSC-assertion is a diagrammatic representation of a Java or C++ class that implements the requirement as a monitor. All variables declared in the local-variables box of Figure 7 are actually properties of this generated class.
5. *Parameterized events*. An MSC-Assertion event can contain objects as actual parameters. In Figure 7, the transition annotated with the message *leader(Integer m1)*, from the *Electing1* state of agent #1 to agent #2, will be traversed if agent #1 is in the *Electing1* state, and the *leader* event with some *Integer* object as an argument is observed from agent #1 to agent #2. Condition guards range over local properties and event arguments (e.g., *m1.intValue() > 0 && m1.intValue() <= 4*).
6. *An MSC-Assertion is an assertion*. It uses the same approach described in [1] for assertion statecharts where it announces a success or failure for every witnessed input scenario. It does so using the built-in *bSuccess* property. The boolean *bSuccess* is true by default. The developer assigns *bSuccess=false* as an action wherever s/he wants the assertion to fail. The JUnit test-case then inspects this property to decide whether a particular test-run failed or not.

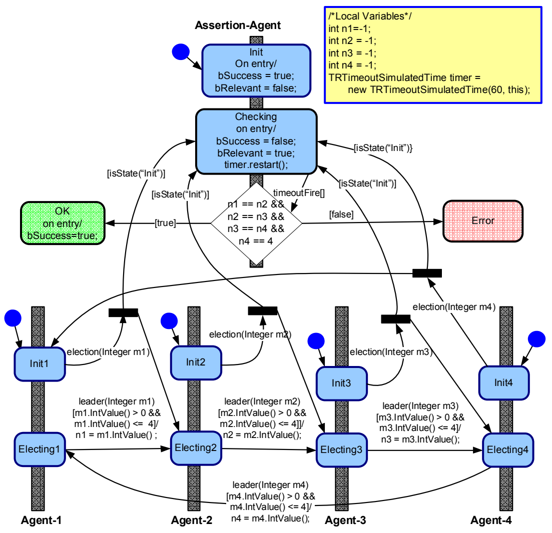

```
/*Local Variables*/
int n1=-1;
int n2 = -1;
int n3 = -1;
int n4 = -1;
TRTimeoutSimulatedTime timer =
     new TRTimeoutSimulatedTime(60, this);
```

Figure 7. A time-bound leader-election MSC-Assertion for a 4-ring network

Figure 7 realized requirement R1 as follows. First note that the assertion contains five tasks (i.e. concurrent threads), one per network agent and an additional task for monitoring the assertion. Also, the assertion contains local variables, n_1 through n_4, where n_i represents the ID of the leader currently elected by agent *i* (-1 means no one has been elected). The MSC-Assertion monitoring starts as the Assertion-Agent task transitions to the *Checking* state when it detects the first *election* event flowing between any two neighboring agents, *bSuccess* is assigned *false* and the 60 second timer is triggered. In other words, the assertion now allows 60 seconds for a successful leader election or else it will fail. Note that the first *election* event does not have to occur on the top-left corner of the diagram, as usually the case with UML MSC's; rather, it can occur anywhere provided that task is in the appropriate state. Suppose that the first *election* event occurs from agent #2 to agent #3. The outgoing *election()* message from agent #2 will cause the Assertion-Agent to transition from the *Init* state to the *Checking* state since the guard *isState("Init")* is true. In addition, it also causes agent #3 to transition from the *Init3* state to the *Electing3* state. Agent #2 will then advance from the *Init2* state to the *Electing2* state as it progresses down its vertical task bar. Whenever agent *i* sends agent *i+1* (modulus 4) a *leader()* message with an Integer argument m, it means that agent *i* has elected agent *m* as the leader. Clearly, *m* must be an id between 1 and 4 or else the agent is not following the protocol. This constraint is manifested as a condition guard on the message transition (in other words, MSC-Assertion message transitions have the same *event[guard]/action* look and feel as UML-statechart transitions). When the MSC-Assertion observes that agent *i* sends a *leader()* message with parameter *m*, it assigns n_i=*m*. Finally, when the Assertion-agent detects a timeout event (60 seconds have elapsed), it tests for a successful leader election according to R1.

Note that R1 is not the only possible leader-election requirement. Consider requirement R2: "all *agents contain the same ID for the elected leader and that ID agrees with the choice of the predecessor agent".* R2 attempts to require that when a leader is elected every agent "knows", and agrees-with, the selection of other agents. The difference between R1 and R2 is subtle. Since the MSC-Assertion shown in Figure 7 only updates n_j's at the sending of a *leader()* message, it is possible that R1 succeeds while R2 fails when a certain link is faulty or exhibits long delays, as follows. Suppose the link from agent #3 to agent #4 is faulty. Suppose also that agent #4 is the first node sending out the *leader* message. A leader (agent #4) will be elected according to R1 even though agent #4 does not "know" that agent #3 has chosen #4 as the leader and hence still in its *Electing_Leader* state in Figure 4. R2 on the other hand will fail under such circumstances since it has to monitor both the sending and receiving of the leader messages. In other words, R1 mandates a leader being elected but does not really deal with agents knowing that they agree on the same elected leader.

Harel's LSC's introduce the notion of cold (blue) vs. hot (red) messages. These two types of messages are used to create a specification in the form of *if a then b,* where *a* is an optional precondition message (cold), and b is a mandatory consequence message (red). MSC-Assertions achieve the same effect using the *bSuccess* property (or any other custom property). In Figure 7 for example, the MSC-Assertion starts up with *bSuccess=true* by default, which effectively means "so far so good". Once an *elect* message is detected between any two neighboring agents, *bSuccess* is set to *false*, which means that the assertion will fail unless some future behavior causes *bSuccess* to be set to true. That mandatory future behavior is requirement R1 (within 60 seconds). MSC-Assertions, like statechart property assertions, use Java/C++ as an underlying language and therefore enjoy Turing-equivalent descriptive power while LSC's are sub-regular (in fact, they are limited to finite sets of scenarios). In addition, MSC-Assertions support non-determinism, which allows the specification of complex real-life scenarios.

5 System-level assertion simulation

Our methodology, as presented in [1, 3, 4], is that formal requirements ought to be simulated to assure that the cognitive understanding of the requirement matches the formal specification. To that end, we developed a run-time monitor for MSC-Assertions that is fully integrated with the popular JUnit testing framework, and created a set of scenarios to be executed by the JUnit testing framework. The scenarios test the MSC-Assertion to assure that it announces a failure if-and-only if a leader is not elected in the prescribed time limit. For example, the following hand-code test case describes a scenario in which the agents successfully complete a round of leader election within an interval of 60 seconds.

```
import junit.framework.*;
public class TestMSC1 extends TestCase {
  private MSC_Assertion  msc = null;
  … // omit the constructor, setup and teardown code
  // Test Scenario:
  public void testExecTReventDiapatcher() {
    msc.setTime(0);   //set clock to 0 sec
    msc.election(3, 4  3); // from agent 3 to agent 4 with id 3
    // Assertion-Agent should now be in Checking state
    this.assertTrue(msc.isState("Checking"));
    msc.incrTime(20); // advance clock by 20 sec
    msc.leader(4, 1, 4);  // from agent 4 to agent 1 with id 4
    msc.incrTime(15); // advance clock by 15 sec
    msc.leader(1, 2, 4); // from agent 1 to agent 2 with id 4
    msc.incrTime(10); // advance clock by 10 sec
    msc.leader(2, 3, 4); // from agent 2 to agent 3 with id 4
    msc.incrTime(10); // advance clock by 10 sec
    msc.leader(3, 4, 4); // from agent 3 to agent 4 with id 4
    msc.incrTime(7); // advance clock by 7 sec
    // the testcase should return bSuccess == true
    this.assertTrue(msc.isSuccess());
  }
}
```

6 Automatic scenario generation

The component level White-Box Automatic Test Generator (WBATG) described in [1] generates a JUnit test suite that repeatedly exercises the component under test using events, time, and data information specified in the component. In other words, the component-level WBATG creates a set of scenarios (sequences) ranging over information received by the component from its surrounding environment. We customized this component-level generator as follows:

1. We distinguish intra-system messages from external messages and data transfers, where the former are those transactions that flow strictly within the system while the latter are transactions with external entities outside of the system. We customized the WBATG to create sequences that range only over external transaction, such as events received from the systems' environment.
2. The WBATG was used to modify inter-system communication delays. In other words, scenarios generated by the WBATG would use varying communication delays along the four links.

7 Estimation of the likelihood of protocol success

The likelihood of success of the leader-election requirement was calculated as the ratio of relevant test scenarios for which the MSC-assertion succeeded to the overall number of relevant tests. A *relevant* test is a test that did trigger a leader-election process; obviously some tests do not trigger the leader election process at all. With the help of OMNeT++, we can now simulate different network conditions and collect statistics to estimate the likelihood of protocol success.

Figure 8 shows the enhanced OMNeT++ model for the 4-ring network with the additional Runtime Execution Monitor and White-Box Tester. The *wb_agent* node contains the C++ code generated by the WBATG described in Section 6. It drives the simulation by sending events to the other nodes in the network via its OMNeT++ proxy and changing the communication delay of the four links. The *msc_agent* node contains the C++ code for the MSC-Assertion shown in Figure 7. It monitors the activities of the other nodes in the network via the messages it receives and notifies the WBATG of the result if the assertion reaches a terminal state.

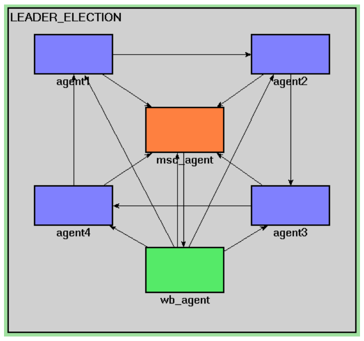


Figure 8. The OMNeT++ model augmented with Runtime Execution Monitor and White-Box Tester

8 Conclusion

The paper brings together several technologies (MSC-Assertion formalism, run-time monitoring, discrete event simulations, JUnit based test methodology) to support the behavior modeling and run-time verification of complex temporal requirements of distributed reactive systems and to asses the likelihood of success of distributed-system protocols under non-ideal circumstances.

The paper presents a new formalism to support the run-time verification of system level requirements of distributed systems using MSC-Assertions. The novelty of the proposed approach include: (1) writing formal specifications of inter-object behavior using MSC-Assertions, (2) JUnit-based simulation and validation of the MSC-Assertions, and (3) the use of discrete event simulation in tandem with automatic, JUnit-based, white-box testing and run-time verification to verify the temporal behavior for distributed system prototypes.

References

[1] D. Drusinsky, *Modeling and Verification Using UML Statecharts*, Elsevier Publishing, 2006.

[2] D. Drusinsky, M. Shing and K. Demir, "Test-time, Run-time, and Simulation-time Assertions for RSP", *Proc 16th IEEE International Workshop on Rapid Systems Prototyping*, Montreal, Canada, June 2005, 105-110.

[3] D. Drusinsky and M. Shing, Creation and Evaluation of Formal Specifications for System-of-Systems Development., *Proc 2005 IEEE International Conference on Systems, Man and Cybernetics*, Waikoloa, Hawaii, Oct 2005, 1864-1869.

[4] D. Drusinsky, M. Shing and K. Demir, "Creation and Validation of Embedded Assertions Statecharts", *Proc 17th IEEE International Workshop on Rapid Systems Prototyping*, Chania, Greece, June 2006, 17-23.

[5] D. Drusinsky and G. Watney, "Applying Run-Time Monitoring to the Deep-Impact Fault Protection Engine", *Proc 28th IEEE/NASA Software Engineering Workshop*, Dec 2003, 127-133.

[6] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming* 8, 1987, 231-274.

[7] D. Harel and H. Kugler, "Synthesizing state-based object systems from LSC specifications". *Int. J. of Foundations of Computer Science (IJFCS)*, 13(1), 2002, 5–51.

[8] D. Harel and R Morelly, *Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine*, Springer, 2003.

[9] I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts", in F. J. Rammig, editor, *DIPES*, *IFIP Conf. Proc*, vol. 155, 1998, 61–72.

[10] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps, "Temporal Logic for Scenario-Based Specifications", in N. Halbwachs and L. D. Zuck, editors, *TACAS*, vol. 3440 of LNCS, Springer, 2005, 445–460.