Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler eds.

A METHODOLOGY FOR UNIT TESTING ACTORS IN PROPRIETARY DISCRETE EVENT BASED SIMULATIONS

Mark E. Coyne
Scott R. Graham
Kenneth M. Hopkinson
Stuart H. Kurkowski

Department of Electrical and Computer Engineering
Air Force Institute of Technology
2950 Hobson Way
WPAFB, OH 45433-7765, U.S.A

1 ABSTRACT

This paper presents a dependency injection based, unit testing methodology for unit testing components, or actors, involved in discrete event based computer network simulation via an xUnit testing framework. The fundamental purpose of discrete event based computer network simulation is verification of networking protocols used in physical–not simulated–networks. Thus, use of rigorous unit testing and test driven development methodologies mitigates risk of modeling the wrong system. We validate the methodology through the design and implementation of OPNET-Unit, an xUnit style unit testing application for an actor oriented discrete event based network simulation environment, OPNET Modeler.

2 INTRODUCTION

Creation of models for discrete event based simulation engines such as OPNET Modeler(OPNET Technologies 2006), Network Simulator 2 (NS-2), and GloMoSim (Zeng, Bagrodia, and Gerla 1998) is an implementation exercise worthy of the same respect given to any large-scale software development effort. In many instances, models consist of thousands of lines of code, placing their development solidly in the realm of software engineering, warranting best practices development techniques.

However, many individuals involved in the development of these models are concerned primarily with another course of research, such as computer network protocol research and development, not producing robust model implementations. One commonly used mechanism for increasing quality in a development effort's code base is through unit testing.

Unit testing via automated unit testing frameworks such as JUnit, CppUnit, and others, is arguably an integral part of any mature software development effort. Indeed, eXtreme Programming (XP) and other forms of Test Driven Development (TDD) require the use of automated testing frameworks, with their strategy of writing a failing test first, editing the production code to make the test pass, then refactoring the production code to production quality while maintaining the passing status of the test (Meszaros 2007).

Many other benefits to unit testing exist as well. Unit testing has been shown to improve code quality and dramatically reduce debugging times through enhanced defect localization (Meszaros 2003) (Müller and Padberg 2003) (Francel and Rugaber 1999). Additionally, unit tests serve as a communication tool by functioning as an automated, self-checking, technical specification for how the software should behave (Meszaros 2007), and a large battery of unit tests opens the door for "eXtreme Programming" (XP) and other agile forms of development through a robust regression testing process (Fowler 1999) (Freeman, Mackinnon, Pryce, and Walnes 2004). Lastly, the unit testing environment allows for easy re-creation of boundary conditions, useful for verifying correctness of process model implementations, which might otherwise be difficult to produce or re-create in a traditional simulation environment (Meszaros 2007). However, in the domain of many discrete event simulation platforms, several classes

978-1-4244-2708-6/08/$25.00

problems prevent the use of traditional xUnit style testing frameworks:

1. Many simulation platforms provide a library of vendor specific API calls. Many of these API have complex implementations. Abstracting these calls completely with use of a mock object would be time consuming, error prone, and impractical.
2. Simulation vendors may specify artificial programming constructs for defining each actor's behavior. These constructs may not be inherently unit testable.
3. Proper testing of a given actor may require execution in the context of a running simulation as defined by the vendor's simulation platform.

This paper describes a software engineering approach to overcoming these impediments, integrating the techniques it into a testing harness, suitable for unit testing actors with any xUnit compliant testing framework. We use the OPNET Modeler discrete event simulation platform to verify the methodology. OPNET Modeler was chosen for the two fold condition that it is actor oriented and it is a widely used, well documented product.

Section 3 describes actor oriented systems. Section 4 presents the architecture of OPNET-Unit. Section 5 describes OPNET-Unit's use of test doubles. Section 6 shows the dependency injection mechanism that allows the framework to provide a modular and re-usable unit testing solution for the general case. Section 7 describes the execution of simulations. Section 8 discusses observation of simulation state. Section 9 provides a case-study example and section 10 concludes.

3 UNIT TESTING ACTOR ORIENTED SYSTEMS

3.1 Actor Systems

The unit testing methodology described in this paper assumes the discrete event simulation is fundamentally actor oriented. Actor oriented programming defines simple building blocks, called "actors" with interact with each other along predefined data paths. Information passing between actors occurs using "tokens" which actors pass along the data paths. Lastly, each actor executes, or "fires" according to an external scheduler (Janneck 2002).

Some discrete event simulators, such as OPNET, model computer networks as a hierarchy of actor systems (Lee and Neuendorffer 2004). The top level contains the familiar network objects, such as clients, routers, and servers. These network objects compose the top level actor system as each network object executes concurrently with other network objects and communicates via predefined paths (wired or wireless connections) with tokens (packets). Each network object is further defined by another actor system which defined the behavior of each network object. The firing, or execution, of each actor is defined by a schedule, or in the case of discrete event simulation, the event queue.

The goal of unit testing is to verify correctness of modules of implementation. Preferably, these modules should be as small as possible to facilitate defect localization. Under traditional programming paradigms, this would be the individual function or method level of an object. In actor oriented systems, the module of verification is the actor.

To effectively unit test a given actor, a framework must regulate the following three aspects:

1. The schedule of an actor's firing.
2. The input, in the form of tokens, along the actor's data path.
3. The implementation the actor executes when it fires.

3.2 Network Simulations as Actor Systems

In the domain of a DES based computer network simulation, these three aspects manifest them selves as network domain specific events.

3.2.1 Regulating actor scheduling

To regulate the scheduling of actor firing in a DES, we must insert individual events in the event queue that result in the execution of the desired actor. In the domain of network simulation, these events would correspond to packet arrivals at the target actor. Thus, the framework must provide a mechanism for scheduling individual packet arrivals for the Actor Under Test (AUT).

3.2.2 Creating actor input

As stated, actors communicate by passing tokens. In the domain of network simulation, these tokes are represented by packets. Thus, the framework must provide a simulation platform specific packet creation mechanism for various packet types (TCP, UDP, etc.).

3.2.3 Controlling actor implementations

When the scheduler decides to fire an actor, the actor must have an implementation to execute on whatever input tokens may be present. Because the environment of the actor under test (AUT) is fixed (not modifiable without re-compilation), the actual implementation of the actor under test must be readily "swappable" without need for code re-compilation.

4 OPNET-Unit ARCHITECTURE

The overall architecture (Figure 3) of OPNET-Unit consists of three distinct tiers: 1) specialized modules which execute as a part of the OPNET simulation, known collectively as the OPNET Mediator 2) the OPNET framework which interacts with the specialized modules via an API, and 3) the application layer–unit tests themselves. The OPNET-Unit framework defines the entry point for the application, and thus uses the OPNET simulation platform as an external library. The top application level contains the actual unit tests the user wishes to execute on the underlying actor, and uses two libraries: the OPNET-Unit framework to interact with the simulation, and the actual unit testing library of the user's choosing such as CppUnit or CxxTest to enforce assertions and collect test results.

4.1 Mediator

Traditionally, the mediator design pattern allows two objects to depend on the mediator, thus preventing the objects from depending on each other (Gamma, Helm, Johnson, and Vlissides 1995). Likewise, the OPNET Mediator allows the framework and the simulation kernel to interact while eliminating dependencies between them. Unlike the traditional mediator, however, the OPNET mediator performs additional services. First, it eliminates proprietary constraints generated by the OPNET simulation platform. This allows the framework layer to interact with an interface free of proprietary influences. Second, the mediator sets up the generic actor system shown in figure 2, thus opening the door for the generic unit testing environment. Third, the mediator functions as an observer server, allowing clients, such as the framework, to register themselves as observers. The mediator then alerts registered observers of important events, such as packet transmission from the AUT, and allows the clients to take appropriate action.

4.2 Framework

The framework layer utilizes the mediator's standard API to interact with the simulation kernel. Via this API, the framework provides the fundamental services required of unit testing an actor system: regulating actor scheduling, creating actor input, and controlling actor implementations. The framework layer provides these services through another API to the application layer. These API's are high level and sufficiently generic to be usable in a unit testing environment. They contain mechanisms for installing a given implementation into the AUT, creating packets as input tokens, advancing the simulation event by event, and checking for output from the AUT.

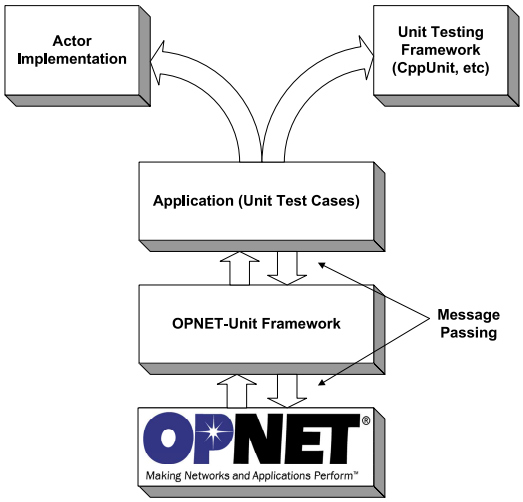

Figure 1: Overall OPNET-Unit Architecture. The architecture consists of three tiers: the OPNET mediator, the OPNET co-simulation, and the actual unit tests as implemented by the user.

4.3 Application

At the highest layer, the application layer executes the user's unit tests. This layer is written by the user utilizing a unit testing library, such as CppUnit or CxxTest. At this layer, the user can utilize the framework's API's to provide input packets to the AUT, check the state of the AUT, and verify correct output from the AUT.

5 SUBSTITUTING PROPRIETARY SUPPORT VIA A TEST DOUBLE

A fundamental reality of implementing such a unit-testing framework for an existing discrete event simulator is mitigating adverse impact caused by the large body of proprietary libraries necessary to run a simulation, and thus a unit test. OPNET Modeler is no different. These libraries provide vital functionality, such as packet creation, inter-actor communication, and other vendor specific functions. Unfortunately, proprietary constraints may disallow their use in a purely unit testing oriented environment. For example, OPNET enforces strict packet ownership, and its packet creation libraries do not permit the creation of a packet independently of an executing simulation. One mechanism for dealing with these libraries is to provide "fake" implementations of all proprietary API calls used by the AUT. Meszaros (Meszaros 2007) refers such implementations in general as "test doubles." Because test doubles that provided simple, test specific implementations are called "fake objects," we refer to our test double implementation as a "fake kernel" as it satisfies the same intent at the simulation platform's

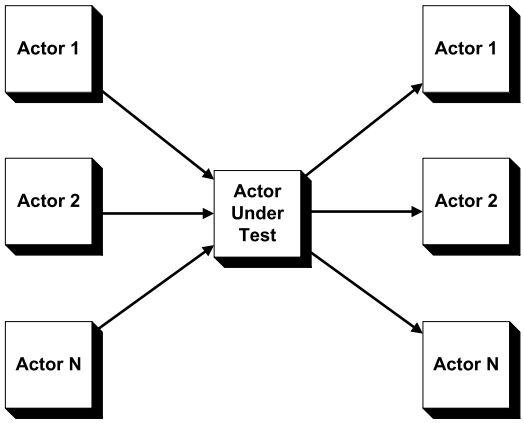

Figure 2: Conceptual Actor Environment. The OPNET Mediator creates a standard API that functions as if this were the executing simulation. This generic actor environment is suitable for testing actors that have a finite number if input actors and a finite number of output actors. The Framework allows unit tests to control the tokens sent by the input actors and observe the tokens received by the output actors, as well as the state of the AUT itself.

kernel level. Using this strategy, the AUT would not be aware of the fact that an actual simulation may not even be running, as all of it's couplings to the proprietary simulation kernel have been removed. While this decoupling is highly advantageous, it comes at a high cost: duplication of existing code.

Thus, our strategy is not to provide test oriented implementations for all of the simulator specific API calls, which would be complicated and unmanageable, but to provide test-specific implementations for a small subset and default to the original implementations for the majority of the required calls. This technique optimizes the ability of the framework to use the original vendor provided implementation wherever possible, while still providing testable access to the AUT for unit test related requirements, such as AUT initialization and state verification.

In order to allow the default implementations to function properly, such as packet creation, the framework must begin an actual simulation. The framework manages the full lifecycle of simulation execution via proprietary API's internally, thus allowing the fake kernel to default to original implementations whenever needed, and allowing the application layer to function without any simulator related dependencies.

6 DEFINING ACTORS VIA DEPENDENCY INJECTION

6.1 Introduction

As mentioned in Section 4, the OPNET-Unit framework contains a module that executes as a part of the OPNET simulation and provides a standard API, independent of proprietary constraints. As part of the API, the module creates, conceptually, the collaboration of actors shown in Figure 2. This actor relationship consists of three categories of actors: 1) "input" actors, 2)the AUT, and 3) "output" actors. Input actors behave as other actors in the system that might provide input tokens to the AUT. Likewise, the output actors function as implementations that might receive tokens produced by the AUT. The framework controls the functioning of both categories of actors. Additionally, the quantity of input and output actors is boundless, as this model is conceptual–implementation details provide for this abstraction.

6.2 Motivation

This conceptual model provides a generic environment for unit testing arbitrary actors. Any number of other test-controlled actors may provide input, and likewise, any number of test-observed actors may receive output form the AUT. Conceptually, when a user wants to test a given actor, they need only to replace the actor under test with their particular implementation.

Ideally, developers performing testing want to change the implementation of the AUT post-compile time. In the OPNET context, we define post compile time as the two-fold condition that 1)an actor's implementation has already been compiled and 2)the network object that contains the actor in question has already been saved and cannot be modified.

This first condition exists because some discrete event based simulators, including OPNET, allow users to define any arbitrary actors in terms of "child actors." These sub-actors behave as regular actors, with the exception that their firings are controlled by the parent actor, not the scheduler. Making liberal use of child actors, the AUT, acting as a parent actor, could dispatch all incoming tokens to any particular child actor, making the child actor the AUT. By switching which child actor received the input tokens, the parent would effectively be swapping between different actor implementations to unit test. However, this mechanism causes unwanted dependencies between the child actor and the parent actor who must depend on the child. This technique would require the parent actor to have a full list of potential child actors to unit test at compile time. Moreover, this casues the simulator to be dependent on both the parent and the child; the addition of child

```
class ActorInterface
{
public:

  virtual void processInterrupt()=0;
  virtual void initialize()=0;

  ActorInterface(void);
  virtual ~ActorInterface(void);
};
```

Listing 1: Actor Interface. This listing shows the interface used by the AUT in the generic actor framework.

implementations would cause the simulation to likewise depend on each new implementation. The elimination of these dependencies is critical, and described in the next section.

The second condition exists because once could simply adjust the actor configuration (Figure 2) to contain the actor desired for testing. However, this is not a viable option because unit testing requires a generic solution. The actor configuration is already sufficiently generic to test arbitrary actors. Re-building the actor collaboration between unit tests is wasteful, akin to rebuilding a house for purposes of changing the light bulbs, and likely requires some code re-compilation. Dependency injection solves all of these problems by providing a method for swapping the implementation of the AUT at runtime. This allows a unit test to swap in its particular actor of interest as apart of test-case initialization.

6.3 Dependency Injection

The principles of dependency injection (DI) and the more general term, inversion of control (IoC), have long been used in many frameworks for the ability to defer establishing dependencies from compile time to a configuration period at run-time and to reduce dependencies of components on specific implementations of other components (Fowler 2004) (Meszaros 2007). Frameworks such as Autumn (C++), Spring (Java, .NET), and PicoContainer (Java, .NET) all utilize DI and IoC (Mürk and Kabanov 2006) (Autumn 2008) (PicoContainer 2008).

Thus, in order to remove the coupling between the simulation kernel and the AUT's implementation, a module external to the simulation must assume control of choosing the appropriate implementation (in this case, the unit testing environment) and provide the implementation to the simulation. Using this method, the AUT depends solely on an interface that all potential AUT's must implement–not an implementation supplied at runtime. OPNET-Unit utilizes type 2 IoC, called "setter injection" by Martin Fowler (Fowler 2004). In this method of dependency injection, the assembler module (the unit testing environment) calls

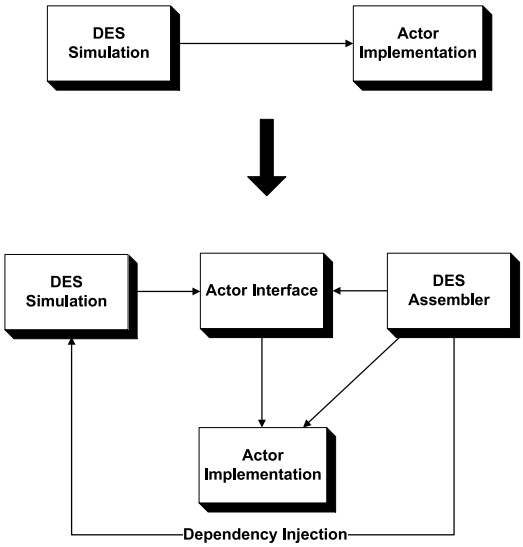

Figure 3: OPNET-Unit Coupling Elimination. In the original OPNET architecture, the DES simulation kernel depends directly on actor implementations. OPNET-Unit eliminates this coupling, allowing the simulation kernel to depend solely on a generic interface shown in listing 1. A separate module then "injects" a particular actor implementation at runtime, thus eliminating the coupling between the simulation kernel and the actor implementation.

a setter method on the client (OPNET-Unit supporting libraries) and sets a field with a reference to the appropriate implementation.

7 EXECUTING SIMULATIONS WITH PRECISION CONTROL

Utilizing standard API's provided by the framework layer, the application layer controls the execution of the underlying OPNET simulation with great precision-down to the event when necessary. Alternatively, the OPNET-Unit framework makes use of the "observer" design pattern to only interrupt the executing simulation when events of interest occur. When the framework initializes the simulation, it registers several call-back methods with the OPNET Mediator. This way, the simulation can run uninterrupted, notifying the framework that the AUT has sent a packet, modified a statistic, or initiated a remote interrupt. When that particular event has occurred, the framework can then decide to check the state of the AUT or continue execution. Thus, the framework can execute the simulation event by event, checking assertions all along the way, or execute the simulation until key events, only then pausing the simulation to check assertions. The framework also recognizes other key events to pause the simulation and check assertions.

8 OBSERVING SIMULATION STATE

In testing a process with OPNET-Unit, there are two important parts of the simulation that should be observed for testing purposes. First, the state of the actual implementation defining the AUT. Second, the state of the surrounding simulation (other nodes, packets, statistics) that the AUT might have altered. Observing the state of the AUT is largely a matter of the way the tester implemented the AUT. Whatever publicly visible state the implementation possess can be verified during unit testing. Observing the state of the surrounding simulation is possible using methods supplied by the framework. The OPNET-Unit Mediator reports pertinent information such as packet arrival and statistic output to the framework's published interface for access and use from a standard unit-testing library. Using these methods, a test scenario is possible that sends a packet to the AUT, and then verifies that the node under test forwards the packet on a given port, and records certain statistics about it's operation.

9 A CASE STUDY

9.1 Introduction

In order to validate the methodology and implementation discussed in this paper, we apply the unit testing mechanism to a pedagogical actor implementation. The actor is implemented as a finite state machine with two states: an initialization state and a wait state. The test cases will exemplify OPNET-Unit's ability to 1) produce packets, 2) control simulation execution and 3) verify AUT state. Additionally, the test cases indirectly utilize the "fake" kernel to handle proprietary issues in the test implementation.

The case study also explores the advantages of a debugging technique known as "slicing." A slice is defined as a the set of statements that affect the value of a variable in a particular statement (Francel and Rugaber 1999). Thus, if a variable has an incorrect value at a certain point in program flow, a debugger should investigate that variable's slice for the solution to the bug. Unit testing is particularly valuable to developers utilizing the slicing technique of debugging, because the specific and focused nature of unit tests facilitate smaller, more manageable slices.

9.2 Test Cases

Figure 4 shows a simple AUT implementation. This implementation performs initialization procedures in the "INIT" state, and then transfers directly to the "WAIT" state where the implementation blocks until another interrupt from the scheduler is received. The INIT state initializes all state variables to 0. The WAIT state performs three simple arithmetic operations involving the incoming packet: calculating

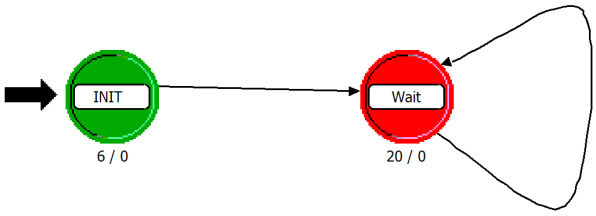

Figure 4: An Example actor implementation. This finite state machine implementation contains only two states with an "INIT" state performing initializations, and the "Wait" state handling all subsequent interrupts from the executing simulation.

```
size_t packet_max_size = 0;
size_t packet_min_size = 0;
size_t num_packets = 0;
size_t sum = 0;
double average = 0;
```

Listing 2: "INIT" State Proto-C. This listing shows the code contained in the "INIT" state of the process implementation under test. Line 2 contains a fault.

the largest, smallest and average packet size. The Code that executes in the "INIT" state is shown in listing 2 and the code that executes in the "WAIT" state when an interrupt is received is shown in listing 3.

To test the implementation, we write several tests to verify proper functioning. In accordance with good testing practices, we are particularly interested in boundary conditions. A few simple boundary conditions are the following:

1. No packets sent to the node.
2. One packet sent to the node (size 0).
3. Two packets sent to the node (sizes 0 and INT_MAX).
4. Two packets sent to the node (sizes 1 and INT_MAX)

Using the CxxTest framework, we can implement the first non-trivial test case, case two (Listing 4).

After running test cases examining all the boundary conditions listed, tests for conditions 3 and 4 fail. Utilizing the debugging concept of slicing discussed in (Francel and Rugaber 1999), we narrow the bug search to lines 13 and 16 and discover empty control statements. We replace the statement:

```
(13) if(packet_size > packet_max_size);
(14)     packet_max_size = packet_size;
```

with:

```
//recognize stream interrupts only
    if(op_intrpt_type() == OPC_INTRPT_STRM){
        //get the packet
        Packet* pkt = op_pk_get(op_intrpt_strm())
            ;
        //get the packet size
        packet_size = op_pk_bulk_size_get(pkt);

            //get the largest packet
            if(packet_size > packet_max_size);
                packet_max_size = packet_size;

             //get the smallest packet
             if(packet_size < packet_min_size);
                packet_min_size = packet_size;
        //calculate average
        num_packets++;
        sum += packet_size;
        average = sum/num_packets;
        op_stat_write(AvgSizeHandle,average);
        delete pkt;
}
```

Listing 3: . "Wait" State Proto-C. This listing shows the code contained in the "Wait" state of the process implementation under test. Lines 9 and 13 contain faults.

```
void test2( void )
{
//get the first packet
Packet* pkt = harness->getPacket(0);
//send the packet to the node under test on
    stream 0
harness->sendPacket(pkt,0)
harness->incrementOneEvent();
TS_ASSERT_EQUALS(manager->getStateVariables()->
    packet_max_size,0);
TS_ASSERT_EQUALS(manager->getStateVariables()->
    packet_min_size,0);


TS_ASSERT_EQUALS(manager->getStateVariables()->
    num_packets,1);
TS_ASSERT_EQUALS(manager->getStateVariables()->
    sum,0);
TS_ASSERT_EQUALS(manager->getStateVariables()->
    average,0);
}
```

Listing 4: Non-Trivial Test Case. This shows the full implementation of the test case that ensures axiom 2 holds true.

```
(13) if(packet_size > packet_max_size)
(14)     packet_max_size = packet_size;
```

and similarly with the "if" statement in line 16. Retest. Now only test 4 fails. Again, slicing leads the debugging search directly to line 2 in the initialization state. We change:

```
(2) size_t packet_min_size = 0;
```

with:

```
(2) size_t packet_min_size = INT_MAX;
```

and retest. All tests now pass. Even in this small sample of code there were several common bugs, extra ";'s" and incorrect initializations, that could have been difficult to debug without an isolated testing environment in which to test axioms and re-create boundary conditions where program faults frequently lie.

10 CONCLUSION

Unit testing has been shown to effectively reduce debugging times for several reasons, most notably through defect localization. However, in many simulation environments, unit testing via traditional unit testing methods quickly becomes infeasible and unmanageable because of close coupling to proprietary simulation kernels. OPNET-Unit proposes a methodology and framework for overcoming these difficulties by building an API around the simulation that makes it appear as a generic actor oriented system.

Through use of a special mediator that allows access to the actor system without reliance on the underlying simulation, we create a framework, suitable for testing in a traditional C++ development environment using a xUnit testing framework. With the support of the new framework, new OPNET development methodologies are now possible including TDD and agile forms of development. Moreover, batteries of unit tests could ship with the accompanying actor implementations and serve as robust regression tests, facilitating future OPNET modeler development; researchers seeking to modify an existing protocol could run the battery of tests after each modification of the source code, clearly identifying which specifications the modification no longer fulfills. Lastly, research efforts can improve, as developers implement new protocol designs for use with the OPNET simulation platform, OPNET-Unit can provide a greater degree of confidence that their implementations are true to their original protocol designs, yielding higher quality and more meaningful results.

ACKNOWLEDGEMENTS

The views expressed in this document are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

REFERENCES

Autumn 2008, January. http://code.google.com/p/autumnframework/.

Fowler, M. 1999. *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Fowler, M. 2004, January. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html.

Francel, M. A., and S. Rugaber. 1999. The relationship of slicing and debugging to program understanding. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, 106. Washington, DC, USA: IEEE Computer Society.

Freeman, S., T. Mackinnon, N. Pryce, and J. Walnes. 2004. Mock roles, objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 236–246. New York, NY, USA: ACM.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Janneck, J. 2002, December. Actors and their composition. Technical Report UCB/ERL M02/37, ERL, UC Berkeley.

Lee, E., and S. Neuendorffer. 23-25 June 2004. Classes and subclasses in actor-oriented design. *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*:161–168.

Meszaros, G. 2003. Test automation manifesto. *XP Universe*.

Meszaros, G. 2007. *xUnit test patterns: Refactoring test code*. Addison-Wesley.

Müller, M. M., and F. Padberg. 2003. On the economic evaluation of xp projects. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, 168–177. New York, NY, USA: ACM.

Mürk, O., and J. Kabanov. 2006. Aranea: web framework construction and integration kit. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, 163–172. New York, NY, USA: ACM.

OPNET Technologies 1987-2006. *Modeler documentation set*. 12.0 ed. 7255 Woodmont Avenue, Bethesda MD 20814-7904 USA: OPNET Technologies.

PicoContainer 2008, January. http://www.picocontainer.org/.

Zeng, X., R. Bagrodia, and M. Gerla. 1998. Glomosim: a library for parallel simulation of large-scale wireless networks. In *PADS '98: Proceedings of the twelfth workshop on Parallel and distributed simulation*, 154–161. Washington, DC, USA: IEEE Computer Society.

AUTHOR BIOGRAPHIES

MARK E. COYNE is a graduate of the Air Force Institute of Technology and is currently an active duty communications officer stationed at Ft. George Meade, Maryland. His interests lie in the areas of networking, simulation, and software engineering. He can be contacted at <mark.coyne@ft-meade.af.mil>.

SCOTT R. GRAHAM is an Assistant Professor of Computer Engineering at the Air Force Institute of Technology. His interests include mobile network protocols, systems engineering, and network simulation. He can be contacted at <scott.graham@afit.edu>.

KENNETH M. HOPKINSON is an Assistant Professor of Computer Science at the Air Force Institute of Technology. His interests include networking, distributed systems, and simulation. He can be contacted at <kenneth.hopkinson@afit.edu>.

STUART H. KURKOWSKI is an Assistant Professor of Computer Science at the Air Force Institute of Technology. His interests include the modeling of mobile networks, scientific visualization, and simulation. He can be contacted at <stuart.kurkowski@afit.edu>.