Proceedings of the 2009 Winter Simulation Conference
M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin and R. G. Ingalls, eds.

HOW TO TEST YOUR MODELS MORE EFFECTIVELY: APPLYING AGILE AND AUTOMATED TECHNIQUES TO SIMULATION TESTING

James T. Sawyer

TranSystems
500 Church St., Suite 480
Nashville, TN 37219, USA

David M. Brann

TranSystems
512 Via de la Valle, Suite 310
Solana Beach, CA 92075, USA

ABSTRACT

In the industrial engineering community, it's a well-known adage that focusing on *process* can help achieve better results. In this second of a series of papers, we'll focus on the *process of simulation testing* and outline how improving your testing process can lead to better results for your projects. We'll consider model building as a software development exercise, and discuss how best practices from the broader software testing community can be applied for process improvement. In particular, we'll explore various approaches to automated testing and their applicability toward simulation projects, based on recent explorations in our own projects. Part 1 of our series introduced the "milestones" approach to simulation development – based on the popular "agile software" philosophy and our own experiences in real-world simulation consulting practice. This time, we'll discuss how thinking agile can help you become a more effective tester, and help ensure the quality of your models.

1 INTRODUCTION

We opened Part 1 of our series (Sawyer and Brann 2008) with a triumphant tale – how using an agile, milestone-based approach saved the day on a high-visibility, high-pressure, short-deadline project. And that's appropriate when introducing a different approach – you've got to show people that it works. For Part 2, we're covering more familiar territory, albeit from a different slant, and so we thought we'd start this one with a Tale of Woe.

The identities have been changed to protect the innocent, but suffice it to say that it was a standard process simulation project. Entities are created based on a set of input data read in from an external database. The entities execute a series of processes and decisions, and exit the system, collecting output metrics that summarize entity performance and system cost. The project team was following an iterative, agile approach to development (Knoernschild 2006), which included substantial testing at the end of the first major milestone. So far, so good. But then during the second milestone, some minor changes were made to the read routines for the input data. Due to time constraints, and since the changes were so minor, no tests were executed against those code changes at the end of Milestone 2 – nor any of the subsequent milestones until the end of the project.

Fortunately, as we were executing system testing just prior to handoff, the team got lucky. Someone noticed that the costs associated with each entity that was introduced into the system were too high. We finally re-ran the Milestone 1 tests for the input data read routines and discovered that the costs weren't being handled correctly. The model was in very real danger of being delivered to the client reporting costs that were 30-40% higher than they should have been.

But what makes this tale any different from any other "you should really test your work" tale? We feel that the key here is that there was a breakdown in the *process*. The team started out on the right track – doing testing at the end of each milestone. And then the process broke down. Code that had been tested previously was assumed to be solid, when in fact it wasn't. And what really scared us was the fact that even though the cost associated with each entity was a key output metric, the calculation of that cost was regarded as being so basic that no structured or system tests were created to verify the cost. So we came very close to delivering a faulty model to the client because we failed at the very lowest level – unit testing of read routines. We got lucky that time. We're hoping that we (and you!) don't have to count on luck the next time.

This paper discusses how our team has been integrating formal testing into an agile-based model development approach. We begin with a brief outline of formal testing and its relationship to verification and validation, and describe how testing techniques may be divided into categories such as *unit testing, structured testing, system testing,* and *Q/A testing*, each with a place in the model development lifecycle. We then observe that teams can sometimes be tempted to skip the testing step, and that automating the process can help teams avoid this temptation. We discuss how automated testing tools are used in the software engineering community, and the challenges of porting these tools to the simulation environment. Finally, we discuss some recent work we have been exploring with automated testing in projects using commercial off-the-shelf simulation products, by leveraging the JUnit automated testing framework and adding test functions in the model runtime itself. The paper concludes with some remarks on the role of testing in an agile, milestone-based simulation development process.

2 FORMAL TESTING OF SIMULATION MODELS

The typical steps in a successful simulation modeling project are well-documented in simulation textbooks and the tutorials in this conference every year. These steps should look familiar: problem assessment and requirements gathering, data analysis, model construction, model verification, model validation, scenario definition and analysis, documentation and presentation of results. These are the steps that burgeoning simulation practitioners are taught in school, and the steps that we as simulation professionals execute on every project during the course of our careers.

Verification and validation are critical to the success of a simulation project, but they can sometimes be confusing concepts. One good way of thinking about the distinction was illustrated by Osman Balci: "Model verification deals with building the model *right*... Model validation deals with building the *right* model." (Balci 1998). Both are part of a process to ensure that the model is working correctly – that it was created as intended and that it is a sufficient representation of the system being modeled – in order to build confidence in the model's results. There is an active community of research on verification and validation, going back to tutorials in the earlier years of this conference (Sargent 1984), and strategic directions on this broad topic (Sargent et al. 2000) continue to be discussed.

In our work in industry, we apply commercially available simulation software products to solve business problems in a wide variety of domains. In these projects, we've found it useful to consider verification and validation within the context of *formal testing*, terminology that stems from the broad software engineering community. The term "formal" is used to emphasize that the process of testing needs to be included as a specific activity within any simulation project plan. That is, the resources, time, and budget to be allocated for completing a simulation project should not solely be limited to the act of building the model itself – it also needs to include a specific period of time for thorough testing of the model functionality (as well as the other key steps in a simulation project lifecycle).

It can be tempting, given the real-world pressures of schedule and budget, to organize a project in such a way that any kind of formal testing is left to the end of the project. As requirements change during the course of the project (as they always do), time and energy is often spent adding new features and functionality to adapt to those changing requirements, and testing is an easy candidate to be deferred to some future time to be determined. All too often, that future time never comes. The result is a product or analysis that is delivered to the customer, partially or completely untested – and a development team that is crossing their fingers hoping their customers do not encounter a situation that crashes the simulation and embarrasses the developers (or worse, invalidates the results and recommendations).

Looking at the standard list of steps in a simulation project, the sequential approach to development and testing is unfortunately reinforced because testing appears to come after model development! Implicitly, that promotes a strategy of "develop, develop, develop, develop, develop, develop, and then test if you have time". Certainly some informal testing takes place during model construction, just to get the model to compile and run, but unfortunately much of the time in practice, this effort is neither well-structured, nor well-documented (Blade and Lehmann 2000).

Assuming your project has been organized into individual self-contained "milestones", as we've encouraged previously, each milestone would involve an element of design, development, *and testing* for the features to be delivered within that milestone. As Steve McConnell reminds us in the legendary book *Code Complete,* "You can build a little, get a little feedback from your users, adjust your design a little, make a few changes, and build a little more. The key is using short development cycles so that you can respond to your users quickly." (McConnell 1993) Thus, a milestone cannot be considered complete until some aspect of testing is performed.

However, since any given milestone, by design, only contains a subset of the full desired functionality of the final simulation, it isn't reasonable to try to test the entire spectrum of project requirements within every single milestone along the way. To further break down the testing process, it can be helpful to distinguish between various types of formal testing and when they should be applied.

3 TYPES OF FORMAL TESTING

There are countless flavors of testing strategies, all of which can be valuable on any particular project: black box testing, white box testing, glass box testing, load testing, stress testing, integration testing, requirements testing, and our favorite, "sanity testing". We've observed that four general types of testing seem to emerge in both software development projects and simulation modeling projects: *unit testing*, *structured testing*, *system testing*, and *quality assurance (Q/A) testing*. Of course, this is not an exhaustive list of testing types! The four types described here are just classifications that we've found to be useful when considering the typical lifecycle of a simulation project in industry.

One thing these tests have in common is a degree of formality and documentation. To prove that a simulation model is behaving as intended, a simulation developer needs to be able to a) specify and document exactly what was intended in the first place, b) create a test to prove that the model behavior matches what was intended, c) execute the test under the appropriate conditions for the test's assumptions, and d) document the results of that test for the internal project team and external stakeholders. These steps are key to successful formal testing of any category.

Let's consider where each of the four types of steps fits in to a typical simulation project – and keep in mind that testing should be occurring continually within each self-contained milestone of the project.

3.1 Unit Testing

- What: Verifying that individual functions and methods used in the detailed simulation code work as intended.
- When: During development of a project milestone.
- Who: Same developer that wrote the code, assuming the team is not large enough to have dedicated testers.
- Why: Allows developer to be confident that the module is ready to be integrated with the rest of the team, and ready to be more formally tested.
- How: Can be as simple as stepping through code in a debugger to make sure the function is working properly, or that a simulation entity is following the correct logical path through the system. Code walkthroughs by peers can help identify potential errors in algorithm design.

3.2 Structured Testing

- What: Creating, executing, and documenting test plans for broader functional requirements of a model.
- When: After feature coding is completed for a project's internal milestone.
  Who: Ideally, not the same developer who wrote the code. With a smaller development team, this may not be feasible. In that case, developers need to learn to wear different "hats" and think like a tester when it's time to do structured testing. When wearing the developer hat, one often thinks: "This code works, and I'll write a test to prove it." When wearing the tester hat, one needs to think: "This code will not work in all cases, and I'll write tests to find those cases." This distinction in approaching the activity of testing is one of the keys to becoming a good tester, and is critical to the success of a testing process. As Myers et al. noted in the book *The Art of Software Testing* (2004):

  > "*If our goal is to demonstrate that a program has no errors, then we will subconsciously be steered toward this goal; that is, we tend to select test data that have a low probability of causing the program to fail. On the other hand, if our goal is to demonstrate that a program has errors, our test data will have a higher probability of finding errors. The latter approach will add more value to the program than the former.*"

- Why: Proves that the requirement has been implemented as specified, at the level of detail that is appropriate for that project milestone. By incorporating documentation of the process, it allows for quicker *regression testing* at the next milestone. Regression testing refers to the re-testing of previously implemented features to demonstrate that any new feature coding did not break something that was previously working.
- How: Follow formal scripts and document each test thoroughly. A well-written test plan defines the objective of the specific test, the initial conditions necessary to set up the test, and a precise definition of what the expected result should be. In many cases, it can be useful to specify and save sets of input data that support a given set of tests.
  - Use the inputs and outputs provided by the user interface, and run the simulation the same way a user would do it – stepping through the code in the debugger is not appropriate for this type of testing..
  - For example, structured tests are developed to ensure that all entities introduced into the model will successfully complete the model. If you put 1 entity in, there should be exactly 1 entity out, and you should see all of the appropriate output statistics collected for that entity. In addition, the value of those statistics can be compared to the inputs driving the simulation to make sure they are being collected accurately. Trace files – model outputs that capture the flow of activity for any particular entity in human-readable text – can be helpful to verify the entity's behavior as well.

- o If the result of running the model shows any deviation from the expected result, it needs to be tracked down and explained or fixed, not rationalized. It can be tempting to make excuses, particularly in complex simulations, such as "It's probably a rounding error or something" or "That's a synchronization issue between entities". Those are not explanations – they are excuses. A test cannot pass with these excuses in place – the root cause must be determined. Any and all issues found should be documented, no matter how large or small. There are specialized applications designed for defect tracking in software development; we use these tools extensively in our simulation projects.

3.3 System Testing

- What: Thorough end-to-end testing of the complete, integrated simulation model from an end user's perspective.
- When: After all components / modules have been created for the simulation project and it is functionally complete.
- Who: The entire development team. Even colleagues outside the development team, who may have little knowledge of the project itself, can be helpful at this phase. This class of users will typically manipulate the model inputs and user interface in unanticipated ways, helping identify possible design flaws, missing error traps, or areas where clarification is needed.

  Another class of users that can be valuable for system is the subject matter experts (SMEs) for the problem domain. While the users with little knowledge of the domain or project can reveal defects by doing unexpected things as users, the SMEs help guarantee that the program itself doesn't do something that would be unexpected for an end user that knows the problem domain (Dustin 2002).
- Why: Verifying that the simulation developed meets the specified requirements. Also verifies the robustness of the completed system as a whole, anticipating errors or confusion for the end user. Does the simulation answer the questions that it was originally designed to address?
- How: Use the model like a user. If applicable, create new/blank scenario and start configuring the inputs from scratch, preferably with a certain analysis goal in mind ("let's see what happens if we overload the workers in this area?"). Run the model, and examine the output metrics in detail. This is different from the structured testing in that the scope of the test is different. Rather than configuring and running relatively controlled, focused tests ("feed precisely enough work to that area so that the workers are be 50% utilized"), the system testing has a broader perspective – given a set of inputs, is the overall system responding as it should? System testing frequently represents the beginning of the validation process.

3.4 Q/A Testing

- What: Final review of the product to be delivered to the customer.
- When: After System Testing is complete, just before final delivery to the customer.
- Who: Primary developer can do this.
- Why: Often, a simulation developer's computer has a much different profile than the target user's computer. Different applications and utilities are installed; there may be a different operating system, version of Microsoft Office (if used), or version of the simulation software package. This final testing process ensures that the user's first experience with the simulation model or product is error-free. You only have one chance to make a first impression!
- How: The model or product is deployed in a virtual environment such as Virtual PC or VMware configured to match the customer's computing environment. The model is installed along with any supporting files. The developer emulates an end user's first time experience, experimenting with all options in the user interface, running the model to completion, reviewing the output reports. In this type of testing, we are not as interested in the specific content of the outputs – we are verifying that there are no missing files, modules, or other incorrect functionality that raises an immediate error to the user or crashes the simulation.

4 TESTING IS NOT REALLY FUN

One of the biggest problems with testing is that it simply isn't as fun as building new models. At the risk of making a gross generalization, many of us in the simulation community are action-oriented, innovative problem solvers – industrial engineers, business analysts, systems analysts. The field tends to draw creative individuals who are constantly seeking out new and better ways to solve problems or improve processes for our customers. We'd rather be out facing new challenges and evaluating new problems than trying to find flaws in the great new innovation we just came up with. With this mindset, it's understandably challenging to have the patience (or the personality type) to be an excellent tester. In some ways, testing feels like a necessary evil. And partly because it's not fun, testing can often be one of the first things to be dropped when times get tough on a project and a deadline looms ever closer.

In addition, the nature of testing itself is not just time consuming, but a little depressing. The sense of accomplishment in simulation development, just like software development, often comes when you've just completed a challenging task. Say that you've just designed a clever new algorithm and spent days fine-tuning the intricate details. You try it out in the context of a model, and lo and behold, it works the first time! Well, that's a wrap, right? You've built something new. It works. Ship it to the customer!

Well, not exactly. To be a good tester, in trying to prove something works, you must try to find all of the different ways that it could fail. You're not just trying to find the one condition in which a particular algorithm or model works correctly – you're trying to find every condition that it could possibly fail. As Myers et al. note: "A good test case is one that has a high probability of finding an as yet undiscovered error" (2004).

Literally, your success as a tester indicates your failure as a developer. And you have to keep trying to do this over, and over, and over. Trying to fail isn't fun, and it certainly is not good for the ego, but it is necessary for the quality of the project, the robustness of the analysis, and the satisfaction of the customer. And in simulation projects, to do testing the "right way" may require many manual steps: design a failure condition, write a test to catch it, set up the initial conditions of your simulation to match the test assumptions, determine the expected result, execute the test, and manually confirm that the model results match the expected result.

But what if the process could be automated? Wouldn't it be great to press a button and have all of your tests run automatically, and then get a nice clear report of which ones failed so you know where to focus further development efforts? The broader software community has increasingly embraced the value of automated testing approaches over the last 10-20 years (Beck 1998), and automated software testing has now become a common part of professional software development. If the practice of simulation modeling at its core can be treated as a software development exercise, as we have posited in our previous installment (Sawyer and Brann 2008), can we harness the benefits of automated software testing in our work as well?

5 AUTOMATING THE TESTING PROCESS

On a recent project, our team began to explore the possibilities of performing automated testing within a discrete-event simulation environment. The purpose of automated testing is to let the computer do the work for you. Since manual testing can be a tedious, rote activity – just the type of activity that most people do whatever they can to avoid – the process of manual testing may end up being delayed, deferred, or eliminated entirely on a real-world project. *Automated testing* refers to creating computer programs that will test other programs on your behalf.

In the software testing world, automated tools to speed up the testing process are commonplace. Usually, these toolkits can be hooked directly into the integrated development environment (IDE) used to build the application, whether it's Microsoft Visual, Eclipse, or other popular choices. Test scripts are written that execute as programs within the IDE itself, calling specific functions within the target application in order to determine whether the function is working as expected. Tests are written so that each test either passes or it fails; there is no middle ground. The entire collection of test scripts is executed together, controlled by a program called the *test execution engine*. The results are then displayed to the developer, with a specific summary count of tests passed and tests failed. The collection of test scripts, the execution engine, and any supporting data required to run the tests are often referred to as a *test harness*.

One clear benefit to automating the testing process is that it makes life easier for the developer. The easier and more convenient it is to do the testing, the more likely it will actually get done on a project under real-world pressures of deadlines, budget constraints, and demanding clients. However, there is an additional benefit of quantifying the amount of work remaining on a project. Knowing exactly how many functions are broken and need to be fixed can help developers create more precise time estimates for completing the application. This knowledge also allows project managers to monitor a quantitative metric reflecting the true state of completion of a project as opposed to relying on the often-mocked and often-doomed approach of the asking developers to estimate their own percentages toward completion.

There is also an important psychological feeling of accomplishment associated with successfully running a test harness. For example, say the first time you run a test harness for a complex application – your results show 79 / 100 tests passed. Not so good; maybe you rushed a little bit while coding. After fixing some coding errors you had overlooked previously, you re-run the test harness, and it now shows 94 / 100 tests passed. You are making tangible progress! And psychologically, now there are only a "few more" tests left to go before you are done. A couple of hours later, you have fixed all open issues and re-run your test harness: 100 / 100 tests! A perfect score!

This type of continual feedback on progress can be a motivating factor during development, particularly for long and complex applications or projects. However, developers need to keep in mind that a perfect 100 / 100 score, while building confidence in the correctness of the code, does not necessarily guarantee that the model is error free. Tests can only detect the presence of errors, not their absence. But a successful execution of a test suite is certainly one indicator to help drive towards successful completion of the project.

Tools that automate the testing process are often utilized by project teams in the practice of *test-driven development*. This is a relatively recent philosophy in software development in which tests are developed first before any code is written. It has been embraced by the agile software community, as it aligns nicely with the core agile principles of frequent iteration, frequent testing, and frequent deliverables to the customer. It also aligns nicely with "thinking agile" in simulation practice.

Following the test-driven development philosophy, when faced with a new functional requirement, the first thing a developer does is to write a test to prove that the function works. A *stub function* (an empty function with no content) is created to represent the function to be. The test harness is executed to call the new function – and naturally, the test fails because no code has been written yet. Then, and only then, the developer writes the code to fulfill the stated requirement. The test harness is executed again, and the test should pass. The practice has been gaining adherents within the software development community, and naturally, the philosophy meshes well with automated unit testing as described in this section.

6 AUTOMATED TESTING TOOLS FOR SIMULATION

In the simulation world, at least in the private sector, the availability and usage of automated testing tools is far less common. Part of this has to do with the typical development environments used by the simulation practitioner in industry. Our team uses commercial off-the-shelf simulation software packages for the majority of our modeling and analysis work. These packages typically come with their own custom development IDEs, provided by the simulation vendors, which are specific to the needs of the typical simulation developer and the capabilities of the tool itself. This is entirely understandable when the target market for the package is a business user, analyst, or engineer, and not necessarily a software developer. (As a side note, it is the authors' continued belief that for most complex simulations, there will be "some programming required". (Banks and Gibson 2001))

Most of these simulation packages do not easily support third-party "plug-ins" as transparently as a mainstream software developer's IDE would. Many simulation packages do allow the integration of "user-defined functions" written in standard programming languages to some degree. However, it can be more difficult to take entire libraries or set of related functions, such as a test execution framework, and integrate them into a model written in a commercial simulation product.

On a recent project, our simulation team decided to explore the possibility of integrating automated testing tools and techniques into the development process. We are continually looking for opportunities for improving our processes (and eliminating our headaches), and this concept has been on our radar screen for many years.

One of the commercial simulation software tools in our arsenal seemed that it may afford us this capability. AnyLogic, a commercially available product by XJ Technologies, is a full-featured simulation software package that is built on the popular programming language, Java. To wit, the AnyLogic IDE itself, as of a recent release, is derived from the open source Eclipse IDE. While AnyLogic does have the higher-level modeling constructs common to all general-purpose simulation languages (e.g. queues, resources, sources, and sinks), one can always break down any object into its core Java implementation.

User functions in this environment are written natively in Java, using the class libraries provided by AnyLogic for simulation-specific classes and methods. This means that you can use functionality from literally any Java class library that exists in the world, not just those provided with AnyLogic. Note that this is different from other mainstream commercial simulation packages, where user functions are either written in a scripting or programming language unique to that simulation environment, or are required to be developed and compiled as external libraries that have little interactivity with or visibility into the core simulation objects inherent to the platform.

Because Java is so widely used, automated testing packages targeting the Java environment are fairly mature. In the Java community, the most popular test execution framework in widespread use is called *JUnit*. This library is commonly used hand-in-hand with the principles of test-driven development as described earlier. In our case, we decided to explore the technical feasibility – and more importantly, the usefulness – of integrating JUnit with AnyLogic.

7 USING JUNIT TO TEST SIMULATION CLASSES

The process of incorporating JUnit into a project includes several steps. First, the JUnit library is imported as a Java package into a Java IDE environment. Then, individual test functions are created and tagged with a special attribute as "tests". Each of these test functions should in turn execute a small chunk of code, almost always just a single function, and then compare the return value (or some other result of that function's execution) to an expected return value. If they match, the test passes. If they don't match, the test fails.

Ideally, one test function would be created for every function within the simulation model. Then, as development progresses, the entire set of tests created would be run on a recurring basis, in order to confirm that any new features or functions added haven't accidentally broken a feature or function that was previously working.

A JUnit test launcher class is also created to serve as the test execution engine. This object is responsible for executing all of the individual test functions, which are recognized because of the special "test" attribute, and compiling the results.

Our project team explored this approach on two different aspects of the modeling project. The first was a "Flat Space Manager" class that was being created by one developer on the project team, for use by other members on the project team. Without going into project-specific details, one can think of this as a stand-alone class that maintains the state of a checkerboard-like grid representing the flat space on a factory floor. The class is designed to track which spaces are occupied by what objects. A part flowing through the main process flow in the model can query a function within the *FlatSpaceManager* class to determine if it will fit in a particular grid space.

The team deemed this was a good candidate for JUnit testing because it was a self-contained Java class, sometimes known as a *POJO* (plain old Java object). A standard AnyLogic model is not typically built from a collection of developer-defined POJOs; rather it is constructed by connecting together graphical blocks that represent the process flow being modeled. In this sense, AnyLogic's model building process is analogous to the graphical drag-and-drop process used by most current commercial simulation packages for ease of use. While every object in AnyLogic is at its core a Java class, most common modeling objects are derived from AnyLogic's built-in ActiveObject class which contains complexity unique to the specific environment. This contrasts with a POJO, which does not have any AnyLogic-specific features and thus can be tested independently.

With a target object in mind, the team started by identifying a target function of the FlatSpaceManager class: *fDoesPartFitInGrid().* This method performs a single function in accordance with the JUnit assumptions – if the part fits in an unoccupied part of the grid, *fDoesPartFitInGrid()* returns true, otherwise it returns false.

Within an independent testing project (as opposed to the AnyLogic runtime environment – more on this later), the team also created a setup function that takes care of initializing the FlatSpaceManager and a couple of different grid configurations for thoroughness. Then, the team created a number of tests that introduced different part types to the function, with different known grid states, and evaluated if the expected true/false value was returned.

In general, this worked great, and seemed like a good fit. That's largely because this approach was "proper" unit testing in the software development sense. The test focuses on an isolated chunk of code and makes sure it does exactly what the developer intended. Since the FlatSpaceManager was an isolated object, its functions could be easily tested this way.

8 AUTOMATED TESTING OF MODEL FUNCTIONS USING JUNIT

The second aspect of the modeling project for which the team explored automated testing was at the functional level. The function *GetProcessStepDelay()* in the main model was to be called frequently by all parts moving within the simulation. The purpose of this function was to calculate how long any given part should dwell at a particular process step. Since process steps were fairly generic in this project, the delay times by design were generalized into a single function for readability. The team wanted to investigate using JUnit to set up and conduct unit tests on this type of simulation functionality.

The first problem encountered with this effort was the sheer amount of setup needed in order to run a test. In the simulation design, a Process Step is not created independently - it only exists as part of a Work Station. This means that in the test project, a Work Station object must first be created and initialized, and then the Process Step. Since a Part is the simulation entity that will be calling the function, a Part must be created as well. However, for a Part to be valid in this simulation, several internal data structures need to be populated for the data that feeds into the calculation of what its delay is going to be at a given Process Step. The team quickly discovered that there was a fair amount of developer effort required to set up the initial conditions so that the test could even be executed.

After spending time on the setup, the project team was able to successfully run a few tests, and confirmed that the function correctly produced the expected result for the process step delay – it divided one attribute of the part by an attribute of the process step to calculate the delay time. These results were not terribly inspiring: "Of course the two attributes would be divided by each other correctly. Why would the model do anything else?" The question needed to be asked: was the benefit of this test worth the considerable upfront time required to set up the tests?

True, the developer effort was only a one-time setup cost associated with the test environment. For subsequent tests involving Parts, Work Stations, and Process Steps, the initial configuration would already be put into place. And as the project progressed and additional logic was added to the *GetProcessStepDelay*() function beyond simple division (boundary conditions, special cases, etc.), the testing framework was ready. Even better, the test was repeatable – it could be re-run every time new features were added to Parts, Work Stations, and Process Steps, in order to make sure that what used to work before still worked correctly. But was the level of effort worth it, when the function could potentially have been verified simply by doing a thorough code walkthrough with a colleague, and visually confirming the correct attributes were being used in the division?

9 AUTOMATED TESTING OF MODEL FUNCTIONS WITHIN THE SIMULATION RUNTIME – PART 1

To further explore this, the project team also tried creating a test harness within the AnyLogic framework, but without utilizing the JUnit libraries. This had the benefit of allowing the standard model initialization routines to take care of setting up all of the needed model objects, just as if a simulation were about to run. However, instead of running the actual model, the test harness would be executed.

The team did attempt to use the JUnit libraries, marking the test functions with the special "test" attribute just as in the previous experiments. However, at this stage, we were unable to successfully set up JUnit tests to be executed within the context of a running, or even initialized, model. Part of this may have to do with a fundamental difference between the types of functions targeted by automated testing of traditional software applications, versus the need to test simulation model functions.

Unlike traditional software applications, discrete-event models involve an element of simulated time elapsing, with the sequence of events being managed by the simulation engine's event calendar. Thus, what appears to be a single function from a simulation entity's perspective (e.g. delay for X minutes) will usually correspond to a whole series of functions in the underlying simulation engine. For example, the engine will place a future event on the calendar representing the entity emerging from that delay, store the entity in memory to be "woken up" after that delay has elapsed, and then execute all other events on the calendar occurring after the current simulation clock time up until the time that the desired event is the top one on the calendar. What is a single function from the *entity's* perspective may involve the execution of many other functions within the engine itself – most of which do not have anything to do with the entity.

It is that lowest layer of functions that would be targeted by traditional unit testing. But this is not the type of unit testing that is needed in the context of the project! The engine-level functions are assumed to be functional in the simulation product – it is the entity's perspective in the context of our model that we would want to target instead.

Note that this assumption of reliability at the engine level that may not always be true – there may be occasions that the engine-level functions themselves may contain errors. And depending on the nature of the simulation engine, it may contain its own quirks that affect the results of the tests (Leye et al. 2009). Testing processes should consider these possibilities as well.

For the unit testing required in this experiment, the team chose not to use the JUnit libraries, and instead wrote independent test functions that were activated with a global flag. In testing mode, only the test functions would b. In normal execution mode, the model would run as designed.

The downside of this approach is simply that the model is running. If there is any type of interactivity or complexity in the model, it is possible that you may inadvertently execute something within a test function that changes the state of the system – and possibly violates an assumption of a different test function. For example, this may make it a little harder to repeatedly call a given function while modifying different parameters to fully cover a range of values. And after entities execute the tests, we would want to be careful that the state of the system is cleared and the entities have exited the flow of the model logic, so that the initial state is predictable and matches the test conditions.

In this, it complicates the design of the tests themselves – and almost mandates that a developer of the model logic needs to be involved in the test design. An independent tester, who knows what the model is supposed to functionally accomplish but may not have written the code to do it, may not have the depth of knowledge of the model design and construction to make sure no key assumptions are violated by the mere existence of a test that changes the state of the system. It's not unlike Heisenberg's Uncertainty Principle, in that we would not want the act of measuring (in this case, execution of a test function) to interfere with the thing being measured (in this case, the resulting state of the system as produced by the target function).

In general though, the team felt that staying within the context of the simulation software's IDE may be a better way to approach doing automated unit testing. The JUnit library is well-suited for unit testing of software applications, but doesn't really add as much value within the simulation IDE. Further research will need to be done to see if there is a way of executing or extending JUnit testing capabilities within the AnyLogic runtime, so that simulation time can elapse as part of the testing.

10 AUTOMATED TESTING OF MODEL FUNCTIONS WITHIN THE SIMULATION RUNTIME – PART 2

Another example of automated testing within a simulation environment – though not in the context of JUnit – is a feature within an internally developed toolkit that we've been using on rail-related modeling projects for over ten years, the Transportation Modeling Studio (TMS). The TMS is a suite of integrated software tools that allow the rapid configuration of simulation models for network and terminal analysis. The toolkit is specifically designed to focus on the capacity planning issues affecting rail, intermodal and marine terminals. These problems range in scope from the switching movements within a single classification yard, to the train movements supporting all major terminals in a marine port, to the cargo flow across an entire network or region. Customized simulation models, using the TMS functionality as a basis, have been deployed to help address strategic planning issues faced by our customers at major U.S. ports, Class 1 railroads, and companies in industries such as mining and plastic / petrochemicals for which certain rail operations are a critical function of the business.

Technically, there are several components to the TMS. A *Network Designer*, based on Microsoft Visio, is used by simulation developers to rapidly configure a rail network design. A CAD file of the network is imported into Visio, and the developer defines all possible origins and destinations, as well as outlines all tracks, sidings, and switches ascribed by the physical layout. A *Network Generator* component then parses through the diagram, and creates a database of all possible routes between origins and destinations that are possible in the system. The *Network Animator* component reads in this database and automatically draws a corresponding representation of this network within Rockwell Software's Arena simulation environment.

The last component of interest (for the purposes of this paper), the *Train Routing Module,* is a simulation library we developed for use within Arena that will automatically optimize the path of any train through the system so that it arrives at its destination in the best possible way (usually this is the soonest arrival time, but this can be configured for specific project needs). All other competing traffic on the network is accounted for, and trains will automatically stop at a switch on the network until competing trains have cleared the way. The optimization module has been designed so that it is impossible for trains to "gridlock" (something that can occur within some simulation designs but would not provide realistic results). A simulation developer working on a new project simply has to provide a schedule of trains to move through the system (based on whatever project-specific operational techniques are required – this is the customization aspect for each customer's project), and the TMS will execute the move and return to the higher-level project-specific code.

Within the TMS environment, we can illustrate a good example of automated testing with elapsed simulation time – the *TestAllRoute*s function. A global flag is used to turn on this special testing mode in lieu of the normal use. After all model initialization occurs, the database of all possible origins and destinations is read into the simulation. Trains of varying lengths are created at every origin and sent to travel to every possible destination defined in the system, no matter how large or complex of a network is being modeled. By staggering the timing of the train movements over time, it can be demonstrated that every feasible route between every origin-destination pair can be executed and animated correctly. This verifies that the simulation developer has drawn the network correctly and everything is connected and configured appropriately according to the TMS design requirements. In addition, the free-flow travel time for each train (defined as the travel time for any given train assuming that there is no competing traffic on a network) is automatically produced as a simulation output. The model's travel time output can be compared to the travel time calculated by dividing the actual distance between each origin and destination by the speed of the train as specified in the inputs, in order to validate the project's representation of the network.

A second step in the testing process is to adjust the arrival pattern of these trains so that all trains arrive at simulation time 0. This provides a maximal conflict situation within the network, although obviously unrealistic, where all trains are trying to move through the network from all origins to all destinations at exactly the same time. By setting up this test configuration, we prove that every train is guaranteed to eventually complete the system without the possibility of deadlock. This effectively verifies both the logic of the *Train Routing Module* (which may have changed from a prior year's version!) and the way in which the team has designed the project-specific network.

It takes little effort to run these tests, and we've found it very helpful in actual projects. A similar concept could apply to other models involving the movement of vehicles through a known path system, such as other rail modeling frameworks that work differently than the TMS, Automated Guided Vehicle systems, or other vehicle movement systems involving multiple origins and destinations. While this approach does not involve the robust and standardized testing features of the JUnit framework, it still complies with the spirit of automated unit testing and works well within a standard simulation environment.

11 DISCUSSION OF AUTOMATED TESTING

One key take-away from this exercise was that automated unit testing in the typical software testing framework seems to enforce the isolated testing of one function. True unit testing. One function call, one result expected. In a typical project engagement at our firm, we treat this level of testing as a more informal exercise, executed by the developer to confirm that the code is working correctly. When our team talks about testing of any sort, we're usually thinking of the *structured testing* and *system testing* levels described earlier in this paper. Our testing plans typically are framed as: "Given this set of model inputs and initial conditions, does the model output match what was expected?" This tends to be a higher-level question than true automated unit testing is geared toward. The challenge with starting formal testing at this level is that when a given output does not match the expected value, there is often a long list of possible sources of that difference, often leading straight back to unit-level testing.

Following on to that is the question of how much value automated unit testing provides, considering the effort it takes to set up the tests in the first place. The upside is that the process is automated, and once it's written provides the basis for effective regression testing. The downside is that it does take time to set the tests up in the first place, and that the level of testing this seems targeted for is a bit below the level of what we usually think of within a simulation environment. This may be something that each project team will need to decide on their own whether or not it is worth pursuing, considering the requirements, scope, schedule, and budget of each individual project. Perhaps it may also influence our future design of simulation projects to consist of more self-contained objects and functions, in order to realize the benefits of these testing procedures.

We also felt there was substantial benefit to letting standard model initialization take care of the "heavy lifting" for us – declaring and defining all of the initial objects and data structures that would allow us to set up the tests more quickly. The key issue with this was that we were unable to figure out how to configure the JUnit test launcher to execute within the AnyLogic runtime environment (initializing the model objects, starting the simulation clock, etc.). And even if we did, the idea of letting simulation time elapse while the tests are executing is a further problem to be resolved.

Notably, for specialized simulation applications that are written in standard programming languages such as C#, Java, or C++, and not general-purpose commercial simulation languages, the availability of standard and open source tools from the broader software community opens up. Many academic papers published here at the Winter Simulation Conference, and other community-related conferences, illustrate complex solutions or libraries written in just these types of languages. This means that automated testing techniques can be explored more fully, and the benefits more recognizable, in these types of academic and research environments.

Bottom line, it seems like there is some potential in the automation techniques that our project team explored, although the value of the specific JUnit toolkit may not be as great due to the issues described. Maybe JUnit itself isn't a good fit. But hopefully, exposure to tools like this can help our teams become more creative about the testing process. And any tools that we can find to save time and effort can help us stay motivated to do the testing in the first place.

12 CONCLUSION

Testing your simulation model is not an easy task. By focusing on the quality of the formal testing process itself, you can achieve a higher degree of quality in the end result – not just the simulation that is created, but the answers and insights that you learn from using the model to perform analysis. There is a well-established field within the broader software community that can provide guidance to the approaches and tools used to improve testing. In fact, there are journals and conferences devoted specifically to the topic! Since building simulation models can be treated as a software development exercise, best practices from the software testing community can be applied to help us all improve the quality of our work.

The agile philosophy embodies a relatively simple principle: "Test early. Test often." By designing your project using the Milestones approach, the opportunities for frequent and recurring testing become a natural part of the simulation project lifecycle. Automated testing techniques can be helpful to reduce the tedium associated with implementing a thorough testing process – and a beneficial effect of this is to encourage that the testing will actually happen. Automated testing tools that are common within the software community, such as JUnit, may have some applicability within typical simulation practice, but each project team will need to evaluate the costs and benefits of incorporating these tools in their work environment.

By applying the Milestones Approach – and as part of this, the formal testing techniques discussed in this paper – consistently within our consulting engagements, we've found that focusing on an improved process results in better quality project deliverables, improved team performance and morale, and importantly, more satisfied customers. We hope that these techniques will work for your teams as well.

REFERENCES

Balci, O. 1998. Verification, validation, and accreditation. In *Proceedings of 1998 Winter Simulation Conf.,* ed. D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, 41-48. Piscataway, NJ: IEEE.

Banks, J., J.S. Carson, B.L. Nelson, and D.M. Nicol. 2000. *Discrete-event system simulation.* 3rd ed. Upper Saddle River, New Jersey: Prentice-Hall, Inc.

Banks, J. and R.R. Gibson. 2001. Simulating in the Real World. *IIE Solutions.*

Beck, K. 1998. *Kent Beck's Guide to Better Smalltalk.* Cambridge University Press.

Blade, D. and Lehmann, A. 2001. Model verification and validation. In *Modeling and simulation environment for satellite and terrestrial communication networks: proceedings of the European COST Telecommunications Symposium*, ed. A.N. Ince. 281-299. Springer..

Dustin, E. 2002. *Effective Software Testing.* Addison-Wesley.

Knoernschild, K. 2006. *An Agile Resolution.* Agile Journal, December.

Leye, S., Himmelspach, J., and Uhrmacher, A.M. 2009. A discussion on experimental model validation. In *Proceedings of the UKSim 2009: 11th International Conference on Computer Modelling and Simulation*, 161-167.

McConnell, S.C.. 2004. *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press.

Myers, G.J., T. Badgett, T.M. Thomas, and C. Sandler. 2004. *The Art of Software Testing.* John Wiley and Sons

Sargent, R.G. 1984b. A tutorial on verification and validation of simulation models. In *Proceedings of 1984 Winter Simulation Conf.*, ed. S. Sheppard, U.W. Pooch, and C.D. Pegden, 114-121. Piscataway, NJ: IEEE.

Sargent, R.G., P.A. Glasow, J.P.C. Kleijnen, A.M. Law, I. McGregor, S. Youngblood, 2000. Strategic directions in verification, validation, and accreditation research. In *Proceedings of 2000 Winter Simulation Conf.*, ed. J.A. Joines, R.R. Barton, K. Kang, and P.A. Fishwick, Vol 1, 909-916. Piscataway, NJ: IEEE.

Sawyer, J.T. and D.M. Brann. 2008. How To Build Better Models: Applying Agile Techniques To Simulation. In *Proceedings of 2008 Winter Simulation Conf.* S.J. Mason, R.R. Hill, L. Moench, and O. Rose, eds. Piscataway, NJ: IEEE.

Vijay. 2003. Types of software testing. Available via < http://www.softwaretestinghelp.com/types-of-software-testing/> [accessed February, 2009]

AUTHOR BIOGRAPHIES

JAMES T. SAWYER is a Senior Professional partner at TranSystems and serves as the Simulation Team Leader for the firm's Central Region. He is the former Chief Technology Officer of Automation Associates, Inc., a leader in simulation for the global supply chain, which was acquired by TranSystems in 2005 to help expand the Management & Supply Chain Consulting line of business. His technical specialty is the design and development of simulation-based software solutions for the transportation and logistics industries. He is the software architect of the *Modeling Studio*, TranSystems' standardized simulation application and user interface framework enabling rapid development and deployment of simulation projects across teams and industries. His current role is to lead development teams as well as coach customers in simulation modeling and software development best practices. Mr. Sawyer received a B.A.S. in Mathematical & Computational Sciences at Stanford University, and an M.S. in Industrial & Systems Engineering at the Georgia Institute of Technology. His email is <jtsawyer@transystems.com>

DAVID M. BRANN joined Automation Associates, Inc. (AAI) in 1999 and now serves as the Simulation Team Leader for the Western Region of TranSystems, which acquired AAI in 2005. He has over ten years experience in designing, developing, and managing the implementation of simulation-based software solutions across a wide variety of application domains, from fast food restaurants to semiconductor manufacturing to the Internal Revenue Service. He has experience using a variety of simulation languages as well as supporting software such as Visual Basic, Java, and C/C++. In addition to his work in managing and developing custom solutions, Mr. Brann is also responsible for providing training courses on a variety of software tools, including TranSystems' own custom packages and AnyLogic simulation software. If the situation calls for it, Mr. Brann can also serve as the proverbial rocket scientist, having received a B.S. in Aeronautics & Astronautics from the Massachusetts Institute of Technology as well as an M.S. in Industrial & Systems Engineering from the Georgia Institute of Technology. His email is <dmbrann@transystems.com>.