Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

MODEL TESTING: IS IT ONLY A SPECIAL CASE OF SOFTWARE TESTING?

C. Michael Overstreet

Computer Science Department
Old Dominion University
Norfolk, VA 23462-0529, U.S.A.

ABSTRACT

Effective testing of software is an important concern in the software engineering community. While many techniques regularly used for testing software apply equally well to testing the implementations of simulation models, we believe that testing simulations often raises issues that occur infrequently in other types of software. We believe that many code characteristics that commonly occur in simulation code are precisely those that the software testing community has identified as making testing challenging. We discuss many of the techniques that software engineering community has developed to deal with those features and evaluate their applicability to simulation development.

1 INTRODUCTION

Determining the correctness of a simulation is often a complex task. Even a precise meaning of "correctness" is not easily formulated; it generally includes the idea of the implementation "correctly" implementing model behavior and the model "correctly" replicating salient features of the system it represents. But it can include accuracy requirements, meeting performance goals, the usability of user interfaces, the quality of graphics and many other aspects. In this paper we focus primarily on issues in testing as one of several useful techniques for assessing the correctness of an implementation; see Sargent (2000), Sargent (2001) and Balci and Ormsby (2000) and Balci (2001) for broader discussions of issues in the verification, validation and accreditation of simulation models.

A comment on terminology: we use the term model to mean a representation of a system that utilizes some form of abstraction. Thus models can be physical (for example made of plastic, wood, or paper), iconic (based on drawings or pictures), text based, or some combination of three. Source code in some general purpose or simulation programming language is also a model of a system. It may not be the idea model representation form since of necessity the source code often includes many implementation details that can obscure the underlying model. Some simulation programming languages incorporate iconic representations so that the model they implement is more easily communicated to a reader. When we wish to emphasize the implementation characteristics of the source code, we use the term simulation implementation, but this is still a model.

The software engineering community has developed many techniques to help in determining the correctness of code. In addition to testing, these include very different and often complimentary approaches such as the use of formal inspections of requirements, designs, test plans and code (Fagan 1976) and the use of formal methods (that is, formal mathematical proofs of correctness). But the most widely used technique in industry is testing, though it may be used in conjunction with complementary techniques such as inspections.

While testing has long been an important area in software engineering, it is still an area of active research. This is in large part due to its high cost, often cited as up to 50% of project development costs, even more if the software must be highly reliable, see, for example, Osterweil et al. (1996). This high cost has caused interest in the potential ability of other less expensive techniques to improve software quality; see Chillarege (1999), Hetzel (1993), Beizer (1990), and Roper (1994) for discussions on software testing and other quality-improvement techniques.

Many issues in the use of testing to establish characteristics of code for both simulations and other types of applications are the same. This is true in part since many simulation applications have requirements similar to other application domains and use similar implementation techniques to meet these requirements,. Thus testing to determine if these requirements are met is no different from testing in other application domains. These include, among many possible examples, using testing to estimate performance or testing to evaluate user interfaces.

We believe that many inherent characteristics of simulations and the implementation techniques widely used simulation applications make testing difficult. Many of these specific characteristics occur in other application domains but the combination of issues frequently found in simulations seems to occur rarely if at all in other areas.

In Section 2 we discuss some testing issues that have direct application to simulation. In Section 3, we discuss some common aspects of simulation that make testing difficult, and follow this by conclusions.

2 TESTING ISSUES

A common problem in the testing of software is the difficulty of detecting unanticipated interactions among components. Sometimes these unintended interactions only occur when the components execute in a particular order. This is particularly true of simulations. The typical simulation execution consists of a collection of components whose execution is managed by the modeler and by a simulation executive. In some languages, these components are called events, in others, activities. If the language is process interaction based, then each process typically consists of several components whose individual executions are managed by the simulation executive. Understanding the possible execution orders of the components can be difficult. Adding to this difficulty is the stochastic nature of many simulations.

Many testing methodologies attempt to guide the testing process so that these interactions are more likely detected during testing.

2.1 Using Coverage to Reduce Testing Costs

A key concern in testing is cost. One of many approaches to reducing these costs is identifying a smaller number of test cases that still effectively tests the code. To do this, it is usually desirable that different tests have the possibility of revealing different problems. A common approach for guiding the selection of test cases uses the idea of *coverage*. In general coverage is intended to help with two complementary goals: 1) to ensure that all features of code are tested, and 2) to avoid duplicate tests that check the same feature. The software engineering community has identified many types of coverage; the testing references in Section 1 discuss them.

Because of common techniques used in simulation implementations and the basic characteristics of many simulations, achieving some widely advocated types of coverage is can be more complex than for many other application domains.

These coverage issues are particularly relevant for simulation since executions typically consist of a sequence of components whose order is determined by the modeler, the simulation executive and random numbers.

2.2 Code Coverage Criteria

A basic form of coverage is *full statement coverage*. It involves testing until every executable line of code in the program has run in at least one test case. It is often recommended as a minimal testing goal. But since some code problems are a result of unintended interactions among different code components, these problems may depend on the order of execution of those program components. Thus other more ambitious coverage goals include the concept of testing many (ideally all) of the various possible execution orderings.

A naive testing goal is to execute all program statements in all the orders that are possible. This is almost always infeasible for at least two reasons: one is that identifying all achievable orderings can be difficult or impossible (determining that an apparently feasible execution sequence is in fact impossible can be difficult). The second is that the number of possible orderings is often so large that executing a separate test run for each ordering would take more time than is practical.

A closely related problem in testing is the *oracle problem*, that is, the difficulty of actually detecting incorrect output when a program is tested. A large number of test cases is an added problem from the oracle perspective since, in addition to the time required to design and run these test cases, the task of detecting when a test reveals incorrect behavior is often daunting. In simulation, sometimes data from the system being simulation is available. If so, and if it can be compared with corresponding model output, the real system data can serve in part as an oracle. However, when models components are assumed to have random behaviors, this comparison may require use of appropriate statistical tests.

3 SIMULATION-RELATED TESTING PROBLEMS

In this section we discuss several aspects of simulations that often make their testing more difficult. While none of these issues are necessarily unique to simulation, they occur frequently in simulation testing.

3.1 Access to Source Code

If a simulation application is written in a general purpose programming language such as C++, Java, or Visual Basic, a tester typically have full access to all source code. This source code can be used to guide the testing process; many commercial tools are available to facilitate testing and depend on the source being available. In many cases, however, the tester will not have access to some key code, for example, the executive that selects code components for execution. If the application is written in a simulation programming language, the compiler provides a simulation executive that manages and schedules the execution of some code components. Likewise, even if the application is written in a general-purpose programming language, if it uses a package that supports the building of simulation models, the package likewise provides an executive that can manage the selection of components for execution. The source code for the package may not be available to testers to assist in guiding the testing process. This "unpredictability" of execution orders of components due to lack of access of the details of the executive adds to the complexity of testing—if a testing goal is to test alternative orderings.

Many models assume stochastic behavior of some behaviors. This randomness can also add to testing complexity similar to that discussed above since components may be invoked in a variety of orderings based, say, on the randomness of an arrival process.

3.2 Testing Concurrent Programs

The testing community has frequently asserted that testing of parallel or distributed code is significantly more complex than testing sequential code. We believe that these additional difficulties are due in part to the many possible execution orders, all possibly valid, of program components, where some may result in incorrect interactions among components. This raises complex coverage issues if alternate execution orderings are to be tested. This is an active research area and several tools have been developed to assist in testing by identifying feasible execution paths; see, for example, (Naumovich et al. 1998), (Cleaveland and Smolka1996) (Cleaveland et al. 1994), (Yang et al. 1998).

This identifying and testing of many different possible statement execution orderings, while a problem for several types of applications, can be a central problem in testing simulations. Models often depict a system as having many activities occurring in parallel. In a distributed simulation implementation, several activities may truly occur simultaneously. But valid single processor executions, which must serialize the apparent parallel executions, can have many possible alternative orderings for the execution of code components, all valid. A particular implementation (possibly provided by the simulation executive) will select one of several possible orderings. If so, it could be pointless to test other feasible orderings since the executive will never select them.

3.3 Implications of Abstraction and Simplification

In a traditional view of building simulations, models should be as simple as possible, at the highest-level abstraction possible provided the behaviors produced are sufficient to satisfy the model objectives. Thus, the behavior of some aspects of a model may not resemble the corresponding behavior of the system being simulated. Likewise, simplifications are usually desirable as long as they do not compromise the ability of the model to meet simulation objectives. This "Occam's Raiser" principle of using the simplest possible simulation model is in part motivated by the belief that less complex models will typically be easier and faster to code, easier and faster to test, and often run faster than a more complex version of the same model.

This implies that it is sometimes desirable for simulation models to produce "incorrect" behaviors for some aspects of the system being simulation. These intentionally incorrect (or too simplistic) behaviors can be revealed in testing. The tester must then understand that these apparently incorrect behaviors are acceptable.

We note that in some domains, particularly where the reuse of existing model components is used to reduce the cost of developing new simulations, this minimalist view may conflict with the advantage of using the same component in multiple simulations. In an environment in which execution time or memory requirements are not a primary concern, the development cost of new, more optimally tailored components can easily exceed the added overhead of using of components that do more than is needed for the new simulation objectives.

Closely related to the principle of using the most abstract model possible is the issue of the accuracy of each component in a simulation. Both accuracy and execution overheads can be affected by the simplifying assumptions of an underlying model. The runtime overhead of a model that produces more accurate behaviors is generally assumed to be higher. In some simulations it is difficult to anticipate what accuracy will be required of various components without performing sensitivity analysis. Thus during testing, deciding if outputs are sufficiently accurate, causing the simulation to fail a test, can be difficult. While this flexibility in accuracy is not unique to simulation (it is an issue with numeric methods), it is often an issue in simulations.

3.4 Delayed Component Execution

In simulations, one code component can explicitly cause a second to execute in two ways, either immediately (at least in simulation time), or after a simulation time delay, either time- or state-based. The testing issue this raises is that if the component execution is scheduled for a future simulation time, a variety of code components that effect the correctness of the scheduled component may sometimes (due, say, to random scheduling) execute before it. Ensuring that in one test case the possibly interfering component runs in the delay interval while in another test it does not run can be difficult. This scheduling is typically implemented through two data structures, often called a future events list (components waiting for a particular simulation time) and a current events list (components waiting for a particular state). While these types of delayed invocations are not unique to simulation, it is unusual and commercially available test generation or monitoring coverage tools are unlikely to detect this missing test case.

In some sense, this is similar to the forced serialization of code components that are thought of as executing in parallel. The simulation executive performs this serialization when the program is run on a single processor, and the programmer may not know the how an ordering is selected by the executive. However, this serialization is typically deterministic and has little effect on testing complexity since the same sequence should be used for each run (assuming all code used for testing is compiled with the same compiler). However, when testing code in which other components may run randomly between the scheduling and the execution of a particular component, say due to random scheduling of some event, to continue testing until most possible order have been tested may be infeasible.

3.5 Use of Nonprocedural Languages

While nonprocedural (also called fourth generation) languages) are widely used in some application areas (data base applications, and protocol specifications, for example), they are frequently used in simulation. This use began early in the simulation community with the widespread and still common use of GPSS. Similar newer language are also in wide use in simulation and include, for example Arena (Swets and Drake 2001), Extend (Krahl 2001), and ProModel (Harrell and Field 2001). (Some have been incorrectly advertised as allowing modelers to build and run simulation models without programming; however programming in nonprocedural languages, even if the language is icon rather than text-based, is still tedious and subject to programmer errors just as with text-based languages).

Since the execution order of components written in nonprocedural languages are not under the explicit control of the programmer and are handled implicitly by the language designers, building test cases with an objective of covering possible execution paths requires that the tester understand many implementation details.

3.6 Effects of Nondeterminism

For many application types, testing is performed by the tester providing a collection of test cases that consists, at least conceptually, of pairs: <*program inputs, expected program outputs*>. For each test case, the procedure consists of running the program to be tested with the *program inputs*, then comparing the actual output produced by the program with the *expected program outputs*. Mismatches indicate an error, either in the program or in the *expected program outputs.*

Since many simulations contain stochastic components, their specified behavior is really nondeterministic. Since the stochastic behavior is usually based directly or indirectly on a random number stream, theoretically a tester could provide a seed for that stream, but this is often infeasible. The implication for this discussion is that, since model behavior is nondeterministic, a single set of inputs can produce a possibly wide variety of equally correct outputs. While this adds to the complexity of testing, it is well understood in the simulation community; it is usually dealt with by treating a particular model output as a single observation from a sample space and statistical analysis of outputs is needed.

3.7 Similarities to Requirements Analysis

An area of software engineering that has received significant attention is analysis of requirements to determine their correctness. This can be similar to simulations written in nonprocedural languages in that both the simulation program and requirements are typically nonprocedural, specifying what a system is to do, but not how it should do it. Nonprocedural simulation languages, while often requiring a programming to provide details about some aspects of the implementation, have a similar goal.

A commonly advocated technique for detecting errors in requirements specification is the use of formal inspections described by Fagan (1976). These have been shown to be cost effective in a variety of application areas but are labor intensive since they are manual procedures that require the participation of experienced and trained personnel. Formal inspections can be used for most artifacts produced in a software development process, including requirements, designs, code and test plans. Their effectiveness and applicability in the simulation domain should be similar to other software development areas.

The software engineering community is developing tools to partially automate analysis of requirements since inspections can be tedious and problems easily overlooked. A standard technique for analyzing requirements is to use simulation to generate behaviors. For this reason, the use of executable specifications have been advocated in some areas. This allows the specifier to observe the behavior, that can be generated by the specifications. If any incorrect behaviors are observed, it is assumed that the requirements are in error. Many types of incorrect behaviors can be revealed through watching the sequence of actions produces by the simulation. While the simulation can rarely produce all possible sequences of behaviors that are consistent with the specification, use of simulation is regarded as useful for revealing errors. Note that if the requirements are executable, coverage issues similar to what has been discussed arise. Determining that sufficient simulating has occurred to ensure that all important and feasible behaviors have been observed is the same problem as determining when an implementation has been adequately tested. It also has the same the similar oracle problem.

3.8 Static Analysis

Another approach for discovering errors in requirements is through the use of static analysis techniques. Forms of static analysis are performed by many compilers (for example, flagging of some initialized variables or unused functions), but only as an incidental activity to code generation. Tools such as Lint for C demonstrate that static analysis can be a useful tool for software developers when dealing with source code. Static analyzers for requirements can check for such properties as consistency and completeness (Heimdahl and Leveson, 1995; Chan et al. 1998) or reachability of identified global states (Holzmann 1987). To do so, the requirements must be expressed in a formal language. Analysis tools can also used to assist in construction of test data and compare the consistency of designs with requirements (Adrion 1982). Atlee and Gannon (1991) have used static analysis of event-oriented requirements specifications to check safety requirements.

Several authors, to establish important characteristics of simulations, have explored static analysis of the simulation implementation. Several characteristics of model specifications can be established through static techniques. Some of these analyses may identify problems with the specification; for example lack of connectedness (see Overstreet, et al. (1994) for definitions and discussion). Likewise other analysis can provide potentially useful information to a modeler that the modeler can use to confirm that some correct characteristics exist or identify problems. For example data flow techniques (determining what lines of code use or modify particular variables, directly or indirectly) can by used to identify causality (what can trigger specific events). The modeler can then judge whether everything that should appear does or that items that should not appear are omitted; see Overstreet et al. (1994).

While automated determination of many characteristics related to the correctness of model specifications are either intractable or NP-hard; (see, for example, Overstreet 1982, Jacobson and Yŭcesan 1995, Page and Opper 1999) others have developed tractable solutions, for example, see Yŭcesan and Jacobson (1996). These algorithms are typically *conservative,* that is, if they cannot conclude that an important property holds for a component, they assume it does not. For example, in performing data flow analysis, if source is not available for a component or the analysis is inconclusive, typically an analyzer will conclude that component both uses and changes the value of a variables of concern just to be safe. Thus in many cases the results are often of little use.

3.8.1 Reliance on Subject Matter Experts

The correctness assessment of simulation models may rely in part on statistical analysis of program outputs and other comparisons with real-world data. While not necessarily unique to simulation, a standard technique is the use of subject matter experts (SMEs) who observe the behaviors to assess the believability of those behaviors. This approach is similar to a Turing test, discussed in the artificial intelligence community.

This is really a form of testing and its effectiveness depends both on making important behaviors visible to SMEs and the coverage achieved; that is, ideally sufficient tests should be run so that an SME can observe the full range of possible model behaviors; the SME is functioning as a test oracle, identifying some forms of incorrect model behavior. Our experience in the use of SMEs that they may not understand the benefits of abstraction and simplification, often insisting that the model behave like the real system even when that more realistic behavior does not contribute to the goals of the simulation.

3.8.2 Data Intensive Models

The correctness of many simulation models depend heavily on the data that are incorporated into the code, in simple cases, the numeric values used parameters for speed, range or parameters of statistical distributions. If a model is data intensive, determining the correctness, or even the usability, of the data can be difficult to determine by testing.

4 SUMMARY AND CONCLUSIONS

In the software engineering community, issues related to verification and validation are an active research topic. Finding less expensive ways to determine that software can be used for its intended purpose is an important focus. Many proposed techniques are not widely used by software developers either because their effectiveness has not been conclusively demonstrated (they may not work outside of research labs) or the cost effectiveness of the techniques is unknown. Some proposed techniques may be worth the additional expense when used for safety critical applications, but are generally perceived as uneconomical for use in most application domains.

Too often it seems that available automated techniques that can help to help with determining the correctness of a simulation implementation only work for very small or very simple simulations (where little help is needed) and do not scale up to large applications where errors are more likely and correctness assessment is more difficult and often more crucial. Part of the problem is that to use some of the techniques, the behaviors of the simulation needed to be restated in a formal language (this can easily cost more that the value of the answer produced by the simulation), or the runtime complexity of the technique goes exponentially with the size of the state space and this size often grows exponentially with the size of the requirements. So again, the technique may only work when it is least needed.

Many of the testing techniques developed by the software engineering community are used by parts of the simulation community. Manual techniques such as inspections can be used directly in simulations. Static analysis techniques are promising, but likely need additional development before they are useful to significant portions of the simulation community. Others developments are likely to be useful to the simulation community if they mature. These include the testing of distributed and parallel code and the checking of requirement specifications.

REFERENCES

Adrion, W. R., M. Branstad, J. C. Cherniavsky. 1982. Validation, verification and testing of computer software. *ACM Computing Surveys*. 14 (2): 159-192.

Atlee, J. and J. Gannon. 1991. State-based model checking of event-driven systems. *ACM SIGSOFT Software Engineering Notes*. 16 (5): 16-28.

Balci, O., and W. F. Ormsby. 2000. Verification, validation and accreditation: well-defined intended uses. in *Proceedings of the 2000 Winter Simulation Conference*. ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick. 849-854.

Balci, O. 2001. A Methodology for certification of modeling and simulation applications, *ACM Transaction on Modeling and Computer Simulation*. 11 (4): 352-377.

Beizer, B. 1990. *Software Testing Techniques.*. New York: Van Nostrand Reinhold.

Chan, W., R. Anderson, P. Beame, and D. Notkin. 1998. Improving efficiency of symbolic model checking for sate-based system requirements. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Software Engineering Notes,* 23 (2): 102-112.

Cleaveland, R. and S. A. Smolka. 1996. Strategic direction in concurrency research. *ACM Computing Surveys*. 28 (4): 607-625

Cleaveland, R., J. N. Gada, P.M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. 1994. The concurrency factory—practical tools for specification, simulation, verification and implementation of concurrent systems. In *Proceedings of DIMACS Workshop on Specification of Parallel Algorithms,* Princeton, NJ. 75-90.

Chillarege, R. 1999. Software testing best practices. center for software engineering, IBM Research, Technical Report RC 21457.

Fagan, M. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal.* 15 (3): 182-211.

Harrell, C. and K. Field. 2001. Simulation modeling and optimization using ProModel Technology. In *Proceedings of the 2001 Winter Simulation Conference.* ed. B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer. 226-232.

Heimdahl, M. P. E. and N. G. Leveson. 1995. Completeness and consistency analysis of state-based requirements. *Proceedings of the International Conference on Software Engineer*. 3-14.

Hetzel, B. 1993. *Complete Guide to Software Testing, 2nd Ed.* New York: John Wiley & Sons.

Holzmann, G. J. 1987. Automated protocol validation in Argos: assertion proving and scatter searching. *IEEE Transaction on Software Engineering.* 13 (3) 683-696.

Jacobson, S. H. and E. Yücesan. 1995. On the complexity of verifying structural properties of discrete event simulation models. INSEAD Working Paper Series. TM/95/12.

Krahl, D. 2001. The Extend simulation environment. In *Proceedings of the 2001 Winter Simulation Conference.* . ed. B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer. 217-225.

Naumovich, G., L. Clarke, and L. Osterweil. 1998. Efficient Composite Data Flow Analysis Applied to Concurrent Programs, ACM SIGPLAN Notices. 33 (7): 51-58.

Osterweil, L. et al. 1996. Strategic directions in software quality. *ACM Computing Surveys*. 28 (4): 738-750.

Overstreet, C. M. 1982. Model Specification and Analysis for Discrete Event Simulation, Ph.D. dissertation, Dept. of Computer Science, Virginia Tech, Blacksburg, VA.

Overstreet, C. M., E. H. Page, and R. E. Nance. 1994. Model diagnosis using the condition specification: from conceptualization to implementation. In *Proceedings of the 1994 Winter Simulation Conference.* ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila. 566-573.

Page, E. and J. M. Opper. 1999. Observations on the complexity of composable simulation. *Proceedings of the 1999 Winter Simulation Conference.* ed. P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans. 553-560.

Roper, M. 1994. *Software Testing.*. New York: McGraw-Hill.

Sargent, R. 2000. Introductory tutorials: verification, validation, and accreditation. In *Proceedings of the 2000 Winter Simulation Conference.* ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick. 50-59.

Sargent, R. 2001. Some approaches and paradigms for verifying and validation simulation models. In *Proceedings of the 2001 Winter Simulation Conference.* . ed. B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer. 106-114.

Swets, R., and G. Drake. 2001. The Arena product family: enterprise modeling solutions. In *Proceedings of the 2001 Winter Simulation Conference.* . ed. B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer. 201-208.

Yang, C., A. Souter, and L. Pollock. 1998. All-du coverage for parallel programs, In *Proceedings of the ACM SIGSOFT International Symposium on Software Test-*

ing and Analysis, Software Engineering Notes. 23 (2): 153-162

Yŭcesan, E. and S. H. Jacobson, 1996. Computational issues for accessibility in discrete event simulation. *ACM Transactions on Modeling and Computer Simulation*. 6 (1): 53-76.

AUTHOR BIOGRAPHY

C. MICHAEL OVERSTREET is an Associate Professor of Computer Science at Old Dominion University. A member of ACM and IEEE/CS, he is a former chair of SIGSIM, and has authored or co-authored over 80 refereed journal and conference articles. He received a B.S. from the University of Tennessee, an M.S. from Idaho State University and an M.S. and Ph.D. from Virginia Tech. He has held visiting appointments at the Kyushu Institute of Technology in Iizuka, Japan, and at the Fachhochschule fŭr Technik und Wirtschaft in Berlin, Germany. His current research interests include model specification and analysis, static code analysis and support of interactive distance instruction. Dr. Overstreet's home page is <www.cs.odu.edu/~cmo>. He can be reached by e-mail at <cmo@cs.odue.edu>.