Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

DEBUGGING SIMULATION MODELS

David Krahl

Imagine That, Inc.
6830 Via Del Oro
Suite 230
San Jose, CA 95119, U.S.A.

ABSTRACT

While much has been written about model validation and verification, the actual process of correcting, or debugging, a model is presented as an afterthought. This paper will describe different types of bugs and will present techniques, drawn from both the simulation modeling and application programming worlds, for determining the cause of an error in the model.

INTRODUCTION

Late at night, an engineer is cursing at the monitor. Not that this will do any good, but it's a ritual that is repeated throughout the world. This engineer is trying to find the source of a bug or error in a model. Eventually, the bug will be found and the engineer can go home for the night (or is it morning already?).

Simulation modeling is programming. Building a model is instructing the computer to execute a sequence of steps. Even if the model is built graphically, a modeler is, essentially, creating a computer program. As this engineer has discovered, what is known as Greer's third law: "A computer program does what you tell it to do, not what you want it to do" is frustratingly true. There are, however a number of steps that can be taken to minimize the possibility of "bugs" and, should they occur, help determine their cause.

1 TYPES OF ERRORS

There are a number of categories of bugs in a software program or simulation model. An understanding of these categories will have an impact on the techniques and tools used to locate the source of the problem.

These different types of errors will show up at different points in the simulation process. Syntactical errors, for example are found by the simulation software program. Data errors require a careful examination of the model input and results.

1.1 Specification

An incorrect model specification will lead to an incorrect model. Specification problems are a typical part of the model building process. Modelers must iteratively compare the simulation model to the real system. An incorrect specification may even result in the invalidation of the entire model.

1.2 Omission

A necessary piece of the model was not included. Part of the art of simulation is the ability to determine what details should be left out and what should be included in the simulation model. Sufficient detail should be included so that the model accurately represents all important aspects of the real system. However, this needs to be balanced against "overmodeling" the system. Fixing this kind of error involves adding to the content of the model.

1.3 Data

Incorrect data can cause problems in any phase of the simulation project. Even a different distribution might cause a model to behave differently. One of the most insidious types of errors, this is typically revealed in model validation and verification phases. Correcting the data often requires additional data collection and there may be no "quick fix" to this kind of problem.

1.4 Syntactical

This is an error in the format of the model statement. This type of error is flagged by the simulation software either immediately after the statement is entered, when it is compiled, at the start of the simulation, or when it is first executed. A possible syntactical error that a C programmer might use in their first Visual Basic program would be:

```
If .value == 1 Then
```

This statement combines the syntax of two different programming languages. It would not work as the modeler intended in either one. Visual Basic requires a single equals sign to test for equality, and the overall syntax for an if statement in C is quite different.

This is probably the easiest type of bug to identify as the simulation model will simply stop processing and generally present a message informing the modeler of the error.

1.5 Logical

Simulation models typically contain significant logical calculations or statements. An error in any one of these can cause the model to behave incorrectly. One example (in C or ModL) would be:

```
if(A == 1 || 2)
```

Although grammatically correct in the English language, this condition would never evaluate to false. Presumably, the modeler would want the condition:

```
if(A == 1 || A == 2)
```

1.6 Time Sequence

Simulation models evaluate parallel processes with a sequence of serial calculations. A race condition, or time tie, occurs when two events or calculations occur at the same simulated time. However, because the simulation engine processes events serially, these calculations will be evaluated in sequential real time. This order of execution may be important to the behavior of the model. Time also plays a part in other calculations in a simulation model. A common mistake is to reference a value in the model before it has been set, or to access that value after it has been re-set by another entity.

1.7 Analysis

The statistical aspects of the model need to be debugged as well. Factors such as run length and number of runs need to be properly determined. If a non-terminating model begins the run "empty and idle" with no entities in the system, then the statistical accumulators must be cleared after an appropriate warm-up time.

1.8 Vendor

It is also possible that the source of a problem in a model is a bug in the simulation software program itself. These can be very subtle and range from a difference of assumptions (how a conveyor works) to an outright programming error.

A vendor bug might not be the direct responsibility of the simulation software developer. It might be in their development tools, the operating system, or even in a video or printer driver.

2 CHALLENGES OF SIMULATION MODELING

There are a number of challenges to model building that are not found in mainstream programming. These are related to the complexity of the model and generalized use of high level simulation modeling tools.

Simulation models are usually built using a purpose-built simulation program. This is generally an advantage. Simulation software is well tested by the developer and provides features that reduce the amount of time to build a model. Simulation software also contains a number of debugging tools unique to simulation modeling.

Using commercial simulation software does mean that the developer is using the assumptions of the software developer and much of the actual code is hidden from the end user. A single modeling component can easily represent thousands of lines of source code. The modeler must take care to ensure that he/she understand the developer's assumptions. Failure to do this can result in a model that is not behaving as the modeler intends it to.

Data problems in simulation models may come from unusual sources. For example, in some cases one random distribution, such as the beta, may have multiple meanings for its parameters. A good way to test the parameters is to build a very simple model that generates a set of samples from the distribution. Use a distribution fitting program to analyze the results to double-check the parameters. Given enough samples, the distribution fitting program should find a fit for the selected distribution and generate parameters that are statistically close to the original inputs.

Finally, simulation models must not only behave properly on the computer, they must be a reasonable facsimile of a real system. Failure of the model to accurately represent the outside world can lead to misleading results.

3 THE DEBUGGING PROCESS

Assuming that it has been determined through validation or verification that a bug does exist, the modeler can begin the actual debugging process. Robbins (2000) gives the following sequence of steps that will help to speed the modeler through the debugging process. These steps are oriented towards general software development, but they apply equally well to simulation modeling.

3.1 Duplicate the bug

Begin by ensuring the problem is repeatable. Every time the model is run, exactly the same problem should occur at exactly the same time. Note the random seed used and verify that the bug occurs exactly the same way every time the model is run. There may be external factors that will alter how the bug occurs. Without this repeatability, the debugging process can become nearly impossible. From time-to-time a bug occurs that depends on a factor that is not predictable (such as the system clock or what is currently loaded into memory). In these cases, the lack of repeatability can become a valuable clue. However, the vast majority of simulation models will behave the same way from one run to the next. This is also a necessary step should the debugging process result in a call to the simulation software vendor.

If possible, reduce the size of the model to be as small as possible with the bug intact. This will make the model easier to debug as there will be fewer factors and the model will run faster.

3.2 Describe the Bug

Describing the bug often helps to fix it. Bring in someone else if possible. The very act of stating the problem often brings the source of the bug to the surface. Even if the listener does not completely understand the model, this can be a valuable aid in short-circuiting the debugging process. A pencil and paper are also useful here. Hand calculation of expected results as well as any diagrams or notes, are helpful in determining the existence and source of the bug. This is a place where having a good relationship with the software vendor is especially helpful. It is essential to have a well stated description of the problem, however, before calling in someone else.

3.3 Always Assume the Bug Is Yours

Most bugs are created by the person doing the programming/modeling. It is easy, but usually incorrect, to blame someone else. By assuming that the modeler has created the bug, the modeler is able to focus on the most likely cause of the problem. Calling the vendor and claiming that the bug is theirs before fully investigating the source of the problem can be counterproductive.

3.4 Divide and Conquer

Find out where the bug is not. Check the values of the variables and step through the model at a high level. Do all of the values and actions match the expected values? Start with a hypothesis. If this is correct on the first try, then the problem is quickly solved. If not, try something else.

Build the simplest model that duplicates the error. This will make the model run faster and there will be fewer variables to consider when debugging.

3.5 Think Creatively

Bugs don't always come from the expected locations. If there is no measurable progress toward a solution, take some time away from the problem. Although project deadlines may make this impossible, developers view the morning shower or drive to work as one of the most inspirational times of the day. Maybe there is an alternate way to approach the model. The bug may not be solved, but this may eliminate it from the model. This is a good work-around, but it is generally best to determine the source of the problem so that it can be avoided in the future.

"Thinking out of the box" is certainly an overused cliché, but it truly applies in this case. Its not unusual for the evidence of a modeling error to show up at a different point in the model than the actual error. Keeping an open mind about the source of the problem can assist the modeler in moving from one hypothesis to the next.

3.6 Leverage Tools

Learn the debugging tools in the simulation software. Call technical support and ask them how they debug models. Build tools if necessary (and if possible). In short, wage guerilla warfare on the bugs and use every tool available. In developing these tools, go back to the "think creatively step". The author has developed a number of modeling components specifically for debugging that have found their way into the released version of the software.

3.7 Start Heavy Debugging

Close the door, turn on music if that helps, and focus on the problem. Take notes about every variable or step that is appropriate. Use a calculator to check the math. The objective here is to become absorbed in the problem. This phase of debugging requires focused concentration. The modeler must mentally track logical behaviors and variable values in trying to determine the discrepancy between the desired and the actual model behavior.

At the end of this phase, the bug should be located and fixed. If not, keep working on it. Bring in new tools, hypotheses, or people if necessary.

3.8 Verify That the Bug Is Fixed

Test in every possible configuration. Do not forget that sometimes fixing a bug can create new ones. It is easy to focus on the problem at hand and forget that there may be undesirable side effects to a particular change in the model. Ideally, validate and verify the model again.

3.9 Learn and Share

Engineers love to talk about their bugs and how they fixed them. For some reason, crashing an entire mainframe is a source of pride. There is a use to this, however, in that sharing this with others and keeping notes on fixed problems can help to avoid them in the future.

4 DEBUGGING TOOLS

Modern simulation environments should contain a rich set of tools for determining the source of a modeling error. There can be, however, a great deal of variability in the quality and level of detail obtained by the available debugging tools. Invariably, debugging capabilities are overlooked in the simulation software selection process. In simulation software surveys (OR/MS Today, 2001), the debugging capabilities are a single yes/no checkbox. This is one area where the modeler is well advised to “test-drive” the software before the purchase.

From the online wikipedia (www.wikipedia.org, debugging): “Debugging is, in general, a cumbersome and tiring task. The debugging skills of the programmer is probably the biggest factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the programming language used and the available tools…”. This is certainly true of simulation models as well as computer programs. A modeler would certainly be wise to look over the wall at the tools that software developers use to solve problems in addition to those tools that are unique to simulation modeling.

Following is a list of tools that are helpful in determining the source of a bug.

4.1 RTM

“Read the manual”. In most cases, there is a wealth of information in there waiting to get out. Not only should the manual contain debugging tips, but there should also be information explaining how the underlying software works (Imagine That, 2003). This can be invaluable in determining the problems in the model. It is also a great tool in reducing the possibility of a bug. A modeler who has a good understanding of the simulation software will better understand the assumptions of the developers. A nice side benefit of this is that the modeler will probably discover previously unknown software features.

4.2 Animation

A great tool for model verification, animation can also help to identify the source of the bug. Using animation is a great first step in the debugging process. The modeler can quickly get an overview of the model behavior and some classes of modeling errors, such as omission or specification may become immediately obvious. An animation is a very effective way to communicate the behavior of the model to a non-modeler that is familiar with the real-world system. Often, outside observers such as this may spot modeling problems that were missed by the model builder.

The modeler should take care not to rely too much on animation as a verification and validation tool. Animation does not typically show the same level of detail as other techniques.

4.3 Model Traces

Recording a model trace to a file is simple, primitive, and effective. Reading through the trace forces the modeler to slow down and evaluate the model execution in a step-by-step fashion.

Trace files create a permanent record of information that is specified by the modeler. Making changes and then comparing the changes to the text file created by the trace can yield clues to the source of the bug.

4.4 Preempt the Bug

In the process of building the model, add additional modeling constructs that alert the modeler to a potential problem. A statement that pops up a window with “Item should not be here” if the item takes what should be an invalid path can help to identify a modeling problem at an early point in model development. While this may not locate the bug directly, it will alert the modeler of a problem before the day of the presentation.

4.5 Interactive Debugger/Model Execution

Interactive debuggers and models allow the modeler to step through the simulation viewing the system status at any point. Variables can be examined as the simulation progresses and the each step of the simulation is displayed before the modeler’s eyes. Powerful debuggers can give the modeler insight into the actions of the simulation model. The best of them can show the actual event scheduling procedures and data structures. Using a debugger at this level can require some expertise, but is unsurpassed in its ability to locate a bug. Given enough time, virtually all problems can be found using this method.

4.6 Common Sense

Just possibly, the model is working properly. Keep a pocket calculator handy to check the model parameters. This method is particularly good at locating problems in the underlying data. Recently, the author was presented with a test case model that was unstable. The software evaluator wanted to test the robustness of the software and intentionally specified data, without alerting the vendor, that would lead to an impractical system. The arrival rate was far too high for the resources available. A little common sense and a pocket calculator verified the discrepancy, the software was dealing with the specification correctly, and the modeling continued. This is a case of something that initially appeared to be a bug, but was merely the simulation software accurately representing the specification.

4.7 Technical Support

Engineers love to solve problems. Do the basic research on the problem. If a solution is not forthcoming, give technical support a call. If there is a good engineer on the other end of the phone line, they will become a valuable resource for this and future problems. A positive relationship with a support person can be helped by good preparation and attitude. Define the problem well, give them all of the information that is requested, and keep an open mind. It is in their best interest to solve the problem quickly and effectively.

5 BECOMING A BETTER DEBUGGER

The author is often asked "how hard is it to build a simulation model in your software?" Of course, for a specific person, the answer can only be determined after they have learned how to use the software. There are however, three skills that the author has found are common to the best modelers are: a background in statistics; the understanding of at least one programming language; and a bit of common sense. Of the three, common sense is the most important. If the other two are not there, a savvy person will recognize that they need to hire a consultant to build models for them. In addition to these basics, a better understanding of the following topics will improve debugging skills.

5.1 Develop a System for Modeling and Debugging

Nearly every simulation textbook (Banks and Carson, 2001; Law and Kelton 2000) specifies the sequence of steps for simulation project. Following this sequence will help to avoid errors of omission and specification at the beginning of the project and data and analysis errors as the project progresses.

Develop a system for debugging as well as for modeling. Line up the tools that you need before you begin the debugging process. Develop a plan of attack for debugging the model. Systematically eliminate the possible sources of the problem.

5.2 A Basic Understanding of Computers

Every pilot should know something about aerodynamics. Understanding data structures, data types, how math is evaluated, and how logical statements are executed will go a long way in creating a better modeler and debugger. If possible, some exposure to binary data types and machine code can be very useful in locating certain types of bugs such as overflows and round-off errors (these are generally handled by the simulation software, but can easily show up in user programmed segments). One of the most useful college courses for modeling in the author's experience is low level machine code programming.

5.3 Learn about Discrete Event Simulation

Learn the basic data structures and algorithms of a simulation model. Most simulation textbooks such as Law and Kelton (2000) have a good example of how a simple discrete event simulation works. At a minimum, the modeler should understand each line of that sample model. Read the manual or contact technical support if it is not clear how the software works. Engineers like to discuss the innards of software almost as much as they like talking about bugs.

Schriber and Brunner (2004) have an excellent discussion of how small differences between simulation programs can impact simulation results. The modeler should always keep in mind the assumptions that underlie higher level simulation constructs. Modelers must also be very careful when changing from one simulation program to another. A conveyor in package A may have somewhat different behavior than the conveyor in package B.

5.4 Step through a Working Model

Gain a familiarity with how the model is actually working. It is probable that some discoveries will be made about the operation of the model and the underlying software. Use the debugger or trace files to show the detailed model behavior. Use a spreadsheet or calculator to add up the numbers and make sure that they reflect the design. Animation can be helpful here as well. Be careful, however, not to rely too much on the animation as it typically will not show the same detail as a trace or debugger.

5.5 Build Models That Are Easy to Understand

There are numerous advantages to this approach. Well built models will be better understood by others and even the model developer. In software development, Spaghetti code is unnecessarily convoluted programming. In simulation, spaghetti modeling are poorly organized models that are difficult to read and interpret. A spaghetti model can hide modeling errors and frustrate the debugging process.

5.6 Test Test Test

Test each phase of the model. Make sure that each model segment makes sense and that they interact properly. Always be suspicious of the model. This is particularly important if the model will be delivered to someone else for analysis or if the model is expected to be used over a long time period.

- Use the model as if the end-user were experimenting with it. Specify parameters that are out of range, does the model deal with these properly?
- Try extreme cases to make sure that the model reacts sensibly.
- Try the model on different computers. The author recently found a case in which Excel reacted differently depending on some seemingly unrelated system settings.
- Ideally, someone else should test the model as well. Its often easy to be too close to a project to do a quality test. Allow someone that is not familiar with the details of the model to operate it.

6 DEBUGGING "DON'TS"

It is easy to fall into these traps when time schedules are tight and the solution seems intractable. Its is best to work through the debugging process in order and with a clear focused objective.

6.1 Don't Blame Someone Else

Don't start with "Your (software, model, code) is buggy". This will not solve any problems and may force some serious "crow eating" when the problem is located. If the bug was self-created, there will be a loss of credibility for future bugs. Remember to always first assume that the problem is the modeler's.

6.2 Don't Build the Model All At Once

This is an old axiom of simulation modeling for a reason. Building models in stages allows the modeler to incrementally test each phase of the model. As it is generally easier to locate problems in a small model than a large one, incremental model building makes for easier debugging.

6.3 Don't Change Multiple Factors At Once

Change one variable, construct, or line of code at a time. This goes back to the "divide and conquer" step in model debugging. At some point, a change will be made and the bug will go away or be altered in some way. This change may be the source of the bug or a valuable clue to its cause. Changing multiple factors obfuscates this process.

6.4 Don't Write Spaghetti Models

Its an easy trap to fall into when deadlines are tight and there are multiple revisions to the model. Not only will it make it more difficult for the modeler to locate the source of the bug, it will more difficult to get outside help on the model. It is time well spent though to organize the model and make it understandable by others. This very process can help to identify or avoid modeling errors.

7 CONCLUSION

Simulation modeling is programming. Even when the model is built entirely with graphical components, the definition of "computer programming" from the online wikipedia (www.wikipedia.org, Programming): "the craft of implementing one or more interrelated abstract algorithms using a particular programming language to produce a concrete computer program. Programming has elements of art, science, mathematics, and engineering." holds true. Because of this, adapting techniques from the sofware engineering community can help a modeler prevent bugs and locate modeling errors more efficiently.

ACKNOWLEDGMENTS

The author would like to thank James Dailey of James Dailey and Associates and Stephen Chick of INSEAD for their input on debugging techniques as well as Peter Kozik and everyone at Imagine That, Inc. for their input on technical content and readability.

REFERENCES

Banks, J., J.S. Carson, B.L. Nelson, and D.M. Nicol. 2001.*Discrete-event system simulation*. 3rd ed. Upper Saddle River, N.J.: Prentice-Hall.

Imagine That, Inc. 2003. *Extend 6 developer's reference* San Jose, CA.

Law, A. M., and W. D. Kelton. 2000. *Simulation modeling and analysis*, 3rd ed. New York : McGraw-Hill.

OR/MS Today February 2001. *Simulation software survey.*(http://www.lionhrtpub.com/orms/surveys/Simulation/Simulation.html) [accessed April 6, 2005]

Robbins, Jack. 2000. *Debugging applications*. Redmond, Washington : Microsoft Press

Schriber, Thomas J., and Brunner, Daniel T., Inside discrete-event simulation software: How it works and why it matters. In *Proceedings of the 2004 Winter Simulation Conference,* ed. R .G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters. 142-152. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

www.wikipedia.org. Debugging [accessed April 4, 2005]

www.wikipedia.org. Programming [accessed April 4, 2005]

BIOGRAPHY

DAVID KRAHL is currently debugging models and simulation applications at Imagine That, Inc. in San Jose, California. He has worked with modelers in solving simulation problems in half-a-dozen different simulation programs. Mr. Krahl has a Bachelors Degree in Industrial Engineering, a Masters Degree in Project and Systems Management, and has spent too much time hanging around with computer programmers.