Abstract

We illustrate the use of heuristic algorithms to improve the verification and validation of software process simulation models. To use this approach, an optimization problem is formulated to guide a heuristic search algorithm that will attempt to locate configurations of the system that yield surprising results. These surprising results often help the modeler to identify flaws in the model logic that would otherwise remain undetected. The general concepts are discussed and a simple example is provided.

I. INTRODUCTION

Verification and validation (V&V) are ongoing issues in the development of effective software and simulation models. It is simply not possible to test the full parameter space of large, complex software process simulation models (SPSMs), therefore V&V approaches emphasize a variety of methods including bounds-checking, sensitivity analysis, and scenario analysis. This paper discusses an idea we refer to as heuristic V&V (HVV), where the SPSM is exercised over a broad range of its parameter space using a heuristic search algorithm. This idea is not new, of course, but we were reminded of its value once again during research that employed genetic algorithms to optimize discrete event simulation models. In this case, the use of a genetic algorithm unexpectedly led to the discovery of a subtle programming error in the simulation model. We then began to look for other examples where heuristic search algorithms can help with V&V. This paper focuses on using HVV to improve the verification and validation of SPSMs.

Heuristic search algorithms have two characteristics useful for verification and validation of simulation models. First, they search a broad range of values in the model’s parameter space. This allows testing of unusual combinations of parameter values that might not be found by ad hoc experimentation, bounds-checking, sensitivity analysis, and the like. Second, they attempt to “exploit” any and all combinations of parameter values whether or not such combinations actually make sense, and whether or not such combinations were intended by the designer.

The concept is illustrated in this paper using an SPSM that was developed using a commercial simulation language with linkages to a commercial optimizer that employs a scatter/tabu search algorithm. The HVV approach was able to detect an unintentional design error that was missed by ad hoc V&V methods. In HVV, the objective is not system optimization per se, although the optimized results may be useful. Rather, the heuristic algorithm is used primarily as a means for thoroughly exploring the parameter space of the model.

II. VERIFICATION AND VALIDATION

According to the DMSO [2001] glossary, verification is “the process of determining that a model or simulation implementation accurately represents the developer’s conceptual description and specification,” whereas validation is “the process of determining the degree to which a model or simulation is an acceptably accurate representation of the real world from the perspective of the intended uses of the model or simulation.” The objective of verification is to determine that the intent of the modeler has been captured by the simulation, whereas the goal of validation is to determine that the model on which the simulation is based is an acceptably accurate representation of reality [Giannanasi et al 2001] [Sadoun 2000].

It is quite difficult to verify computer simulation models. When attempting to do so, one is faced with the same problems that one must
confront when verifying any complex computer program:

1. Large programs (or systems) have a number of residual bugs that will never be completely eliminated [Yourdon 1979, pg. 26].

2. Due to the complexity of the simulation software ..., complete removal of bugs is infeasible. It has long been established in the practice of software engineering that complete removal of bugs by testing, even for relatively simple codes, is impossible from a practical standpoint [Froehlich et al 2000, pg. 421].

3. It's almost impossible to verify totally a model for a complex system [Kelton et al 2001, pg. 513].

As with verification, validation is also not a goal that can actually be achieved, but is rather an ideal for which to strive. Some authors insist that, since there is no such thing as absolute validity, the word validity should not even be used; instead, modelers should simply discuss how their model has been tested. For example:

1. Any 'objective' model validation procedure rests eventually at some lower level on a judgment or faith that either the procedure or its goals are acceptable... without objective proof [Forrester and Senge 1980, pg. 123].

2. There is no completely definitive approach for validating the model... [Law and Kelton 2000, pg. 279].

3. Many modelers speak of model 'validation' or claim to have 'verified' a model. In fact, validation and verification are impossible [Sterman 2000, page 846].

David Kelton and his colleagues [2001] propose one reason for the lack of trust in V&V—that interactions among the many different simultaneous activities occurring in a model can cause unintended consequences that can be very difficult to detect. This means the modeler must "develop tests that will allow you to ferret out the offending interactions or just the plain and simple modeling mistakes" [page 512]. They recommend creating various scenarios and replicating these scenarios in the model to see that the model still performs adequately, and point out the importance of running the simulation for extended periods of time.

John Sterman [2000, page 845], argues that the emphasis should be on model testing...the process to "build confidence that a model is appropriate for the purpose"...not on V&V. His 45-page discussion on model testing provides much detail on the model testing process, including the importance of determining:

1. Is the model robust in the face of extreme variations in input conditions or policies?

2. Are the policy recommendations sensitive to plausible variations in assumptions, including assumptions about parameters, aggregation, and model boundary?

In the context of continuous (differential equation-based) simulation models, he suggests a number of "assessment" tests, including boundary adequacy tests, parameter assessment tests, extreme conditions tests, and sensitivity analysis. These tests call for thoroughly exercising the models over a wide range of parameter values.

Law and Kelton's often-cited discrete system simulation textbook [2000] includes 25 pages on the subject of building valid and credible simulation models. The authors recommend that the modeler "run the simulation under a variety of settings of the input parameters and check to see that the output is reasonable" [page 270]. Of interest to us is their insistence that "The measures of performance used to validate the model should include those that the decision maker will actually use for evaluating system designs" [pg. 265], suggesting to us the possibility of using optimization methods based on plausible objective functions.

Underscoring the importance of V&V, the Handbook on Simulation by Banks includes a 58-page chapter on verification, validation, and testing [Balci 1998].

Regardless of whether one calls it V&V or simply testing, everyone agrees that it is important, and is often not done as thoroughly as it should be. One way to increase the thoroughness of the testing process, without resorting to exhaustive parameter search, is to use a heuristic search algorithm, as we describe here in the context of SPSM.

III. HEURISTIC SEARCH ALGORITHMS

As mentioned in the introduction, heuristic search algorithms are interesting with regard to V&V because they conduct a broad search of
parameter space, and they seek to achieve a specific goal without regard for the intentions of the person who designed the simulation model. Let’s look at each of these aspects in turn.

Often, when testing simulation models, a limited number of operating conditions are evaluated, focused perhaps in the neighborhood of current and/or highly plausible future operating parameters. These neighborhoods might not include regions of parameter space where possible flaws in model logic would become obvious.

The algorithm concentrates its search through parameter space in those regions that provide high benefit in terms of the objective function. The algorithm has one goal, to optimize the fitness function, subject to whatever constraints and business rules the designer built into that function. Since the algorithm is only minimally constrained, it will “exploit” whatever loopholes and errors it discovers in the course of its search. Errors might include poorly written branching logic or incorrectly defined process steps. The hope is that the heuristic search algorithm will identify combinations of parameter values that make the results of that error more obvious and easier to detect.

Scatter/tabu search is a heuristic search algorithm that incorporates decision rules and constraints as inputs [Glover 1998], [Glover et al 2000]. The process operates as follows:

1. A set of candidate solutions are generated using a problem-specific heuristic, and a subset of the best are designated as reference solutions.
2. Linear combinations of the elements of the reference solutions produce new solutions.
3. A new set of reference solutions is selected from the population, and the solution generation heuristic (1) is applied again.

Scatter search algorithms are readily available in commercial packages such as OptQuest from OptTek Systems, Incorporated. Such tools have also been integrated with commercial simulation tools, making it possible for HVV to be employed easily and quickly. For our illustrative application, we use the academic version of the Arena simulation software from Rockwell Software, which features an integrated version of OptQuest.

IV. APPLICATION OF HVV TO SOFTWARE PROCESS SIMULATION

One use, among many, for SPSMs is to find the “appropriate” staffing level for a project, where appropriate means that a balance has been struck between low utilization (indicating improper resource balance) and lengthy project duration due to insufficient resources. Typically, performance improves dramatically when resources are added to a system that has insufficient resources. However, due to other constraints, there are typically diminishing returns from adding additional resources beyond some point. If the logic of the system has not been properly modeled, then the performance of the system might appear to improve beyond what is actually possible.

The modeler may, based on prior experience, believe that the proper resource balance should occur when the ratio of coders to testers is, for example, approximately 2:1, and therefore might never try running the model with ratios far different than this. But, HVV would do exactly that, and, when it results in superior, but impossible results, logical error(s) in the model are likely to be revealed.

Consider the situation, common in SPSMs, where multiple resources are needed to complete a particular task, such as a code review. The modeler might inadvertently fail to seize all of the necessary resources. In some cases, the syntax checker would catch this error. In other cases, watching an animation and/or carefully studying the outputs might reveal the flaw to the modeler. But, in other cases, particularly with more complex models, such an error might remain in the model undetected.

By employing HVV and fully exploring different possible combinations of resources, the modeler is more likely to discover such a flaw. For example, HVV may report that it is possible to complete the software project more quickly by adding more testers and reducing the number of coders. This is likely to look suspicious to the modeler, and lead to discovery of the flawed logic.

1. The Model

The model is simple, but not completely trivial. We wanted the model to be a simple as possible, but we learned as we developed the illustration that Arena, to its credit, does a good
job of revealing obvious errors to the modeler, especially when the logic is simple. In order to make a compelling case for HVV, we felt that the flaw had to be subtle enough that an experienced modeler could easily miss it. This necessitated building a slightly more complex test case than we had originally thought would be necessary.

The model shown in Figure 1 represents a scoped portion of a software development process. Units of code to be developed are created at the start of the simulation and assigned a unique id tag. A copy of each code unit is routed along a second path to initiate the creation of a test package for each code unit. Code units are programmed and then inspected. Inspection requires a walkthrough with the coder and a tester. If the inspection reveals errors, the coder does the rework and the code unit is re-inspected. Once the code passes inspection, the coder is released and the code unit is routed to a special queue to await unit test.

Test packages are created and routed to the same special queue module, where code units and test packages are matched and grouped together based on their unique id. Unit test is then performed. If the code unit passes, both the code unit and test package are counted, and exit the model. In a more complete model, the code units would be checked in to await integration and system test, etc. Code units that fail unit test are reworked, and routed back to be re-matched with its test package and re-tested. The associated test package is routed back as well, since it will be needed again.

Task times are modeled using triangular probability density functions. The specific data utilized is illustrative only. Write code requires T(1,2,4) days, and inspect code requires T(1,3,8) hours. Rework after inspect is T(1,4,8) hours, and write test package is T(5,1,2) days. Unit test is T(2,5,8) hours, and Rework after test is T(5,10,20) hours.

Initially, our plan was create a deliberate flaw in the model that HVV would subsequently reveal. This flaw was to have been to inadvertently seize the tester to perform the rework after test, when it should have been the coder. In fact, however, when we applied HVV, we found that the model contains a subtle flaw in its logic that we had not introduced deliberately! This flaw is that the coder waits until a tester is available to do the code review, which is not realistic. This error was not easily caught because the utilization figures for the resources appeared to be plausible.

Next we describe the ad hoc process of experimenting with the model in order to determine the “best” mix of resources. During this process, the modeler specifies the number of resources (coders and testers) that are available, and then runs the model. At the end of each experiment, the modeler reviews the output report to check the project duration and the utilization for each of the resources. Each experiment consists of 20 replications in order to assure sufficient sample size for estimating the utilizations. The experimental objective is to find the “best” mix of resources that can be relied upon to complete the project quickly.

After a number of iterations, the modeler has determined that 6 coders and 3 testers is the “best” mix of resources. This resource level and mix is also consistent with a priori expectations. Table 1 shows the output from Arena, indicating resource utilizations, including confidence interval data for these estimates.

Notice in Table 1 that the utilization of the resources (coders and testers) is approximately 93%, suggesting that a proper ratio had been established and the total number was neither too high nor too low. The project duration (not shown) ranged from 27.8 to 32.7 days, with a mean value of 30.7 days. The duration is based on working 24 hours per day, 7 days per week. This figure must be multiplied by about 4 to obtain actual project duration in calendar days based on working 5 days per week, 8 hours per day. This could have been calculated within the model by introducing resource schedules, but adding this complexity was not felt to be necessary for this illustrative example. Thus, the 30.7 days corresponds to the project duration being about 123 days or 6 months. The modeler was ready to recommend this mix of resources for the project.

2. The Experiment

What might the modeler learn by utilizing HVV? To apply HVV, one must first be more explicit about what a “good” result is. In this case, we want to minimize cost, subject to the constraint that the schedule is completed in each of the 20 replications for a specific configuration of resources. In Arena, costs may be easily incorporated by specifying how much each resource costs per hour. In this illustration, coders...
cost $100 per hour whether they are busy or idle, and testers cost $50 per hour whether they are busy or idle.

The HVV strategy then uses the OptQuest optimization algorithm that is integrated with Arena to explore different combinations of resources. If there are flaws in the model, the [relatively] unconstrained optimization algorithm may find that it can exploit those flaws and produce a lower cost configuration...that may or may not actually make sense. Even if the model is not flawed, OptQuest may well find attractive solutions that the modeler did not think to try. To use OptQuest, we simply indicate: A) the objective function (minimize total cost), B) which control variables to experiment with (the number of each resource), C) the range to try for each control variable (very broadly in the case of HVV, in this case from 1 to 10 coders and testers), D) constraints amongst the values of the control variables (none in this case), and E) any “requirements” (that all of the code units must be completed for each replication, and that the utilization for each resource must be less than .95). Table 2 shows the results.

In this case, HVV indicates that the project will cost less if we increase the number of resources by over 50%. This does not make sense.
Table 1: Output Report from Flawed Arena Model

<table>
<thead>
<tr>
<th>OUTPUTS</th>
<th>Outputs from Flawed Arena Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codeunits.NumberIn</td>
<td>161.25</td>
</tr>
<tr>
<td>Codeunits.Numberout</td>
<td>161.25</td>
</tr>
<tr>
<td>Coder.NumberSeized</td>
<td>63.250</td>
</tr>
<tr>
<td>Coder.ScheduledUtilization</td>
<td>.93668</td>
</tr>
<tr>
<td>Tester.NumberSeized</td>
<td>209.35</td>
</tr>
<tr>
<td>Tester.ScheduledUtilization</td>
<td>.93907</td>
</tr>
<tr>
<td>System.Numberout</td>
<td>49.000</td>
</tr>
</tbody>
</table>

Simulation run time: 0.05 minutes.
Simulation run complete.

Table 2: HVV Results on Flawed Model

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>545579</td>
<td>49.000</td>
<td>0.938730</td>
<td>0.952586 - Infeasible</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>599382</td>
<td>49.000</td>
<td>0.924375</td>
<td>0.535514</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>570967</td>
<td>49.000</td>
<td>0.878959</td>
<td>0.923354</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>555085</td>
<td>49.000</td>
<td>0.909490</td>
<td>0.945184</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>552112</td>
<td>49.000</td>
<td>0.883171</td>
<td>0.821833</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>Best 13</td>
<td>633917</td>
<td>49.000</td>
<td>0.907194</td>
<td>0.877228</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Current 78</td>
<td>614724</td>
<td>49.000</td>
<td>0.891522</td>
<td>0.515660</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

This result was quite robust regardless of the task times, and even robust with respect to whether or not we introduce or remove the flaw associated with whether or not the correct resource is utilized to perform the Rework after Test. We began to realize that HVV was indicating a truly unintentional flaw in the model logic. A closer inspection of the statistical reports showed the while the coder is utilized at 93%, much of that time is wait time rather than productive time. This led to the realization that having coders simply wait in the model until a tester became available to do the code inspection is not correct. Instead, the coder would start to work on another code unit until a tester is available to do the code review on the completed code unit. The model logic was corrected to reflect this discovery. Table 3 shows the results of rerunning the optimization after correcting the error.

Here we see that in fact only 3 coders and 2 testers are able to complete the project, with the same 93% utilization reported earlier. This is very interesting! The project duration (not shown) varied from 42 to 52 days, with a mean value of 47 days, which as mentioned earlier would be 188 days working 40 hours per week, or about 9 calendar months. Thus, if project cost is the primary driver, the resource mix of 3 and 2 would be preferred. One can see from Table 3, however, that doubling the resources does not double the cost. By adding resources one can significantly shorten the project duration. Thus, if time to market is the driver, one might utilize more resources even though it is somewhat more costly.

- 118 -
Table 3: HVV results with Corrected Model

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Minimize System_TotalCost</th>
<th>Requirement System_NumberOut</th>
<th>Requirement Coder_Utilization Value &lt;= .95</th>
<th>Requirement Tester_Utilization Value &lt;= .95</th>
<th>Coder</th>
<th>Tester</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>496191.</td>
<td>49.000</td>
<td>0.895980</td>
<td>0.695603</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>492347.</td>
<td>49.000</td>
<td>0.904216</td>
<td>0.715038</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>22</td>
<td>490868.</td>
<td>49.000</td>
<td>0.895242</td>
<td>0.783879</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>23</td>
<td>480494.</td>
<td>49.000</td>
<td>0.915391</td>
<td>0.740194</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>37</td>
<td>473447.</td>
<td>49.000</td>
<td>0.903576</td>
<td>0.859275</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Best: 38</td>
<td>443420.</td>
<td>49.000</td>
<td>0.995244</td>
<td>0.927659</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Current: 77</td>
<td>537170.</td>
<td>49.000</td>
<td>0.930894</td>
<td>0.561403</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

and less efficient. The modeler can proceed to report model results such as these to the client with higher confidence than would have been warranted without conducting HVV.

IV. CONCLUSION

We have illustrated how the use of heuristic algorithms may potentially enhance the verification and validation of software process simulation models. In this case, it helped to reveal a subtle model design flaw that went undetected during ad hoc testing and model experimentation.

However, there is no guarantee that by using HVV all important model errors will be detected. It might be the case, for example, that the flawed logic is rarely executed, and is therefore not triggered in the cases generated by the heuristic search algorithm. Further, it is well known that there is no guarantee that a heuristic algorithm will locate all of the pertinent locally optimal configurations. And once located, it may not be possible to determine whether the configuration is truly a local optima or the result of subtle flaws in the logic of the model.

What HVV does do well is to exercise the model over a broad region of the input parameter space, and this significantly increases the likelihood that the modeler will uncover hidden flaws in the model logic that may otherwise go undetected. The fact that heuristic optimization algorithms are now being bundled with popular simulation software packages makes it surprisingly easy to perform HVV analysis. Given this ease of use, there is no reason not to include HVV as a standard component of the SPSM model testing process.

References