ROBUSTNESS MEASUREMENT: AN APPROACH TO ASSESSING SIMULATION PROGRAM RELIABILITY

Daniel Berleant, H. Cheng, P. Hoang, M. Ibrahim, S. Jamil, and P. Krovvidi
Dept. of Computer Systems Engineering
University of Arkansas
Fayetteville, AR 72701

ABSTRACT

We describe a new approach to measuring the quality of simulation software by measuring its *robustness.* Robustness is the ability to give similar or identical numerical results even when mutations — random bugs — are injected into the code. Robustness is important because large programs, such as large simulation programs, always or nearly always have unintended bugs. Thus it is desirable for a program to behave similarly in the presence of bugs to the way it would behave without bugs.

We estimate how well the results of an existing simulation program p_0 may be expected to match the results of its bug free counterpart p by intentionally adding bugs to p_0, producing a set of programs P_1 containing programs $p_{1,j}$ that each differs from p_0 in containing a single added bug. Each program $p_{1,j}$ in P_1 is further mutated to provide a 2-bug program $p_{2,j}$ in 2-bug program set P_2. Likewise, a set P_3 with three bugs in each member is created, etc. Then, we measure how closely the the simulation results predicted by programs in P_N match those predicted by program p_0. These robustness results then constitute knowledge about the relative dependability of p_0: *Robust simulation programs give us more confidence than non-robust ones.*

Readers of this paper are requested to contact us regarding robustness testing of their simulations.

1 INTRODUCTION

We measure robustness by mutating the software in various ways to model random programmer errors, observe the outputs of these modified programs, and analyze these outputs to produce measures of robustness that describe how well the program withstands random faults. Two small discrete event simulation programs were obtained from published code (MacDougall 1987) which will be assumed (perhaps incorrectly) to be bug free. We also tested a continuous simulation model (Berleant et al. 1994).

2 ROBUSTNESS MEASUREMENT

Robustness is measured in the following way:

1. Run p_0, the most bug-free version we have of the program, and record its output.
2. Generate a set of 1-bug programs $P_1 = \{p_{1,1}, p_{1,2}, p_{1,3}, ..., p_{1,j}\}$ whose members each contain one additional bug derived by mutating p_0 in a different way for each member.
3. Run each program in 1-bug set P_1, recording the output of each.
4. Compare the results of running the 1-bug programs with the result of running original program p_0. The way this comparison is done constitutes the specifications of a robustness measure. Three robustness measures are described in this section.
5. Repeat the process for a 2-bug program set P_2, a 3-bug program set P_3, etc.

As sample discrete event simulation programs we used a published program which simulates ether local area networks (LANs). This program is called *ether* (MacDougall 1987). We also used a slight modification of a published program which simulates a multiprocessor system with 8 processors, 4 memories, and 2 buses. This program is called *bws1a* (MacDougall 1987). We also tested MacDougall's simulator itself, an interpreter for a

small language `smpl` written in `C` which runs *ether* and *bws1a* and for which MacDougall provides source code. As an example of a continuous event simulation program we used *HIV-IVDA*, a program that simulates the spread of the HIV virus (which causes AIDS and may be transmitted by sharing drug injection equipment) among intravenous drug abusers (IVDAs) in the United States (Berleant et al. 1994).

Each program mutant was generated by using a random integer, locating the line of code with cardinality corresponding to the random number, and mutating it by hand to have a reasonable-seeming bug.

Next we describe individual robustness measures and the results of measuring the test programs against those measures.

Figure 1: **Mean deviation for mutated versions of *bws1a* vs. % of lines that were mutated.**

2.1 Robustness Measure #1: Average Percentage Of Deviation In Output Value

Consider the question, "By what amount is the output expected to change when a bug is added?" The deviation in results would depend on the particular bug, of course, but there is also an *average* deviation, which is the average deviation of different mutated programs. A program which is highly sensitive to bugs according to this measure would tend to give considerably different results in response to the "average" bug — that is, it would exhibit a high average deviation.

In figures 1 and 2, mean deviation in output value was expressed as a percentage y. A plotted point at $y = 200\%$ for example means the average program in some set P_N (where N is a location on the x-axis) deviates from the output value v produced by the original program p_0 by a factor of 2, or 200%. Either a doubling or a halving of v would be counted as 200%; if the output of program $p_{N,i}$ was double that of p_0, and the output of program $p_{N,j}$ was half that of p_0, both would have exhibited a deviation of 200%, therefore the average deviation of the pair of programs would also be 200%. Deviations of more than a factor of 10 (1000%) were left out of the analysis for two reasons: (1) they would have an inordinately large effect on the mean deviation, and (2) modelers and users would in most cases realize such extremely deviant output values are defective.

Figure 1 shows the mean deviation for the *bws1a* simulation program. Figure 2 shows the mean deviation for the *ether* simulation program. Figure 3 shows the median deviation for the HIV-IVDA model. The median, unlike the mean, provided a curve which does not decrease at any point. The two zero values on the left of the curve indicate that the majority of bugs did not affect the output values (six of them) that were examined for this study, when the program only contained one or two mutations. More mutations caused more severe effects on the output values. The x-axes are calibrated in terms of the percentage of lines of code that are mutated rather than the absolute number of such mutated lines, to facilitate comparison of the robustnesses of different programs which might be of different lengths.

2.2 Robustness Measure #2: Probability That Th Output Of A Mutated Program Changes In T Right Direction When Inputs Change

Consider the question, "What is the percentage of mutated programs that react to a change in input values by producing a change in output which is in *the same direction as the change in output predicted by the unmutated version* of the program?" This percentage forms a robustness metric which would be especially useful in cases where we are interested in seeing what changes to inputs will produce changes in output in a desired direction. This could be useful when the objective of simulation is to show how an output value can be raised or lowered. This objective could be met even if the precise numbers output by the simulation are not needed or are

Figure 2: **Mean deviation for mutated versions of *ether* vs. % of lines that were mutated.**

Figure 3: **Median deviation for mutated versions of *HIV-IVDA* vs. % of lines that were mutated.**

Figure 4: **Percentage of the time output changed in the right direction when input was changed (*ether*) vs. % of lines that were mutated.**

considered unreliable, as is the case (for example) when input values provided to the simulation are not considered reliable. Problems with reliability of input values are often characteristic of models of society. For example, in modeling the spread of AIDS among intravenous drug abusers, initial values for numbers of infected individuals in early years of the epidemic are rough estimates at best (Berleant et al. 1992).

In figure 4, the percentage of the mutated versions which predicted a change in output in the same direction as that predicted by the unmutated program, regardless of the actual values predicted, is plotted against the percentage of code lines in the program that are mutated for *ether*. The program measures as robust as long as the number of bugs is low. In particular, *ether* measures as more robust on this measure than for average deviation. We conjecture that simulations will tend to be more robust for qualitative measures of robustness than for quantitative measures such as average deviation. In the figure, data points toward the right of the graph are based on a small number of mutated programs, hence are unreliable. Quantitative predictions that deviate by a factor of 10 (1000%) or more were left out of the analysis, as for the average deviation measure.

Figure 5: **Percentage of mutated versions (*bws1a*) that run vs. % lines that were mutated.**

2.3 Robustness Measure #3: Percentage Of Mutated Versions That Fail At Run Time

Consider the question, "What is the probability that a mutated program will fail when run?" Failure can consist of termination with an error message, non-termination, or termination with output data that is obviously faulty. (To be consistent with previously described robustness measures, we consider outputs that are greater than 10`x` or less than $\frac{1}{10}$`x` of the output values of the unmutated program to be obviously faulty.) This measure may be useful in determining whether the source of low robustness is the implementation itself, or rather the model or real system (section 3). Results are shown in figures 5, 6, and 7.

2.4 Caveats on the Results

The *ether* and *bws1a* simulation programs were implemented in the `smpl` language, which is an extension to `C`. The source code for the `smpl` language was also tested for robustness in the same way as *ether* and *bws1a,* with mutated versions of the language implementation used to run an unmutated *ether* program. The results were that some mutations led to run time failure, but for mutations that ran successfully *the quantitative output never changed from the output provided by the unmutated version* of `smpl`.

This result is intriguing. The robustness concept

Figure 6: **Percentage of mutated versions (*ether*) that run vs. % lines that were mutated.**

Figure 7: **Percentage of mutated versions (*HIV-IVDA*) that run vs. % lines that were mutated.**

applies to any software for which inaccuracies in quantitative output cause a corresponding decrement in the value of the program. Since `smpl` is not a simulation model itself, but rather an example of a different genre of software (in this case, simulation program interpretation software), these results suggest that what we learn about the meanings of robustness measurements of simulation programs may not apply directly to other kinds of software. Therefore different genres of software will require different interpretations of the same robustness measurements.

The results we have presented are suggestive, and illustrate the feasibility of the robustness measures. However they are based on a relatively small number of samples (twenty 1-mutation versions, twenty 2-mutation versions, ..., twenty 5-mutation versions of each program, of which most produced run time failures rather than quantitative output results). Further research and more reliable conclusions on these programs and others will require more samples.

3 DISCUSSION

Robustness is most closely related to existing work in simulation verification, simulation validation, software testing, and programmer error characterization. (Validation, and programmer error characterization are not discussed here due to space constraints. Contact first author for full paper.)

3.1 Verification

Simulation verification is the

> "Substantiation that a computer program implementation of a conceptual model is correct and performs as intended." — Knepell and Arangno (1993)

Since large simulation programs are essentially guaranteed to contain bugs, verification by this definition will never be completed in large, practical simulation programs. Robustness research, then, relates to verification efforts in that while verification is directed toward removing bugs, robustness is directed toward guaging the significance of bugs. *Robustness results can help organize verification work on a simulation by providing an indication of how much difference additional debugging effort will make.* More research is needed to further elaborate the relation between robustness and verification.

3.2 Software Testing

Testing is gaining increasing visibility with the appearance of a variety of commercial software testing tools (e.g. Hansson 1993; Pure Software Inc. 1993). However software testing, testability, and reliability have been under investigation for some time. The results most closely related to robustness measurement have to do with mutation testing and with testability measurement.

Mutation Testing. Mutation based testing involves modifying a program to be tested, in numerous different small ways. The modifications create buggy programs and are called mutations. The set of mutated programs is tested against a set of test vectors. Those test vectors which best distinguish between the mutated programs and the unmutated program by causing the highest percentage of mutated programs to give different results than the unmutated original are deemed to be good test vectors; test vectors which do not thus distinguish between mutated and unmutated programs are bad test vectors (Howden 1982).

Mutation testing requires generating mutated programs. Testing environments that support mutation based testing by automatically generating mutated versions include Mothra (King and Offutt 1991, and numerous other references).

Robustness measurement differs from traditional mutation testing in the way mutations are used. Ordinary mutation testing makes analyses based on whether the outputs of mutated programs are the same or different from the unmutated program. In contrast, robustness measurement makes analyses based on the *degree* of difference between the outputs of mutated and unmutated programs. The purpose of robustness measurement and mutation testing also differ. Robustness measurement finds the resistance of a program to having its output seriously affected by bugs. In contrast, mutation testing finds test vectors for which outputs are likely to be affected to any degree. We believe that for simulations that give quantitative outputs, robustness measurement provides useful information about depend-

ability not knowable through other kinds of testing.

Testability. Testability (Voas et al. 1991) of a program is in essence the likelihood that bugs will reveal themselves by affecting the program outputs. For example, a segment of code rarely executed will have low testability, because for most inputs the outputs will be unaffected by bugs in that code segment. If mutation testing finds that a high proportion of potential bugs usually do not affect the output, the program is deemed low in testability. Yet a program that is low (bad) in testability will often be high (good) in robustness precisely because such a program's outputs will tend to resist being affected by bugs. Thus while odd it is perhaps not too surprising that a program can be found deficient from the standpoint of testability, yet dependable from the standpoint of robustness, or vice versa. Further, we expect that programs found *low* in quality from one standpoint will tend to be considered *high* in quality from the other standpoint!

The tendency of programs that rate high in either testability or robustness to rate low in the other is only a tendency — there will be exceptions. In particular, testability is determined by *whether* outputs are affected by bugs, whereas robustness is determined by the *degree* to which outputs are affected. Therefore, programs can potentially be high in both testability and robustness and therefore be particularly high in quality overall. Conversely, very low quality programs would be low in both testability and robustness. Research is needed to find techniques for building programs that are high in quality because they are high in both testability and robustness.

REFERENCES

- Berleant, D., R. Goforth, and J. Yuan. 1994. "A Computer Model for Predicting AIDS Among Intravenous Drug Abusers." In *Proceedings of the Arkansas Academy of Sciences, Vol. 47 – 1993* (Arkadelphia, Apr. 2–3). Arkansas Academy of Sciences, submitted, expected to appear.
- Berleant, D., A. Khanna, and R. R. Goforth. 1992. "Comparison of Simulation with Pdfs vs. Fuzzy Values." In *Proceedings of the Symposium on Computer Simulation in Industrial Engineering and in Problems of Urban Development* (Mexico City, Nov. 18–19). Society for Computer Simulation Internaciónal and Universidad Panamericana, Mexico City, I.9–I.15.
- Faltings, B. and P. Struss, eds. 1992. *Recent Advances in Qualitative Physics.* MIT Press, Cambridge, MA.
- Hansson, O. 1993. "Programming Tools Catch Memory Misuse." *Byte* 18, no. 11 (Oct.): p. 28.
- Howden, W. E. 1982. "Weak Mutation Testing and the Completeness of Test Sets." *IEEE Transactions on Software Engineering* SE-8, no. 4 (Jul.): 371–379.
- King, K. N. and A. J. Offutt. 1991. "Fortran Language System for Mutation-Based Software Testing." *Software Practice and Experience* 21, no. 7 (Jul.): 685–718.
- Kotik, G. B. and L. Z. Markosian. 1989. "Automating Software Analysis and Testing Using a Program Transformation System." In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification* (Key West, FL, Dec. 13–15). ACM, NY, 75–84.
- MacDougall, M. H. 1987. *Simulating Computer Systems.* MIT Press, Cambridge, MA.
- Pure Software Inc. 1993. Commercial products for testing software.
- Voas, J., L. Morell, and K. Miller. 1991. "Predicting Where Faults Can Hide from Testing." *IEEE Software* 8, no. 2 (Mar.): 41–48.