Experience Report for WOPR, 4/20/2010, CRIM, Montreal, Quebec, Canada – Fault Density Prediction Experiments:

Gregory Pope, Lawrence Livermore National Laboratory - LLNL-TR-427516

Overview:
One of the purposes of our SQA effort at LLNL is to attempt to determine the "goodness" of the research codes used for various scientific applications. Typically these are two and three dimensional multi-physics simulation and modeling codes. These legacy research codes are used for applications such as atmospheric dispersion modeling and analysis and prediction of the performance of engineered systems. These codes are continually subjected to automated regression test suites consisting of verified and validated expected results. Code is managed in repositories. Experience level of developers is high in the knowledge domain, platforms, and languages used. Code size of the multi-physics code used in this study was 578,242 lines excluding comment and blank lines or 5538.7 function points. Languages were 70% C++, 20% C, and 10% Fortran. The code has 130 users and a development team of 14 and an embedded SQE. The code has achieved 100% prime feature test coverage, 73.6% functional test coverage, and 71.5% statement test coverage. The average cyclomatic complexity of the code was 6.25. The codes have evolved over 10 years.

Research codes are challenging because there is a desire to balance agility with discipline as well as compliance with DOE standards. Agility is important to allow experimentation with new algorithms and addition of the latest physics features. Discipline is important to increase the quality of the codes. Automation of processes and defect prevention/detection are deployed throughout the software development process. Since research codes are a small segment of the software industry, not much information exists in terms of reliability studies on these types of codes. This paper describes attempts to determine the goodness of these research codes. Goodness defined as both correctness of the codes and their fault densities. Correctness is determined by user interviews, peer review; feature based automated testing, and coverage measurement. This paper focuses on the fault density aspect of goodness and reliability of the codes in particular.

The approach taken was to use multiple fault density prediction methods and compare results to actual experimentation and other industry studies on fault density. As a result of the predictions and experiments our confidence in the prediction methods was increased and our confidence in the goodness of the code from a fault density perspective was given more context. A large unintended benefit of these experiments was to find defects hidden for years in the codes when using the Monte Carlo reliability testing results to develop heuristic based bug driven tests.

The Methods:
The six techniques chosen for the defect prediction were:

1. COQUALMO (Constructive Quality Model)
2. EPML (Equivalent Process Maturity Level)
3. Compare fault density predictions to Industry
4. Static Code analysis using Klocwork to determine found fault density
5. Cyclomatic Complexity analysis using Klocwork and historical defect data
6. Musa based reliability MTTF predicted from fault density compared to a reliability demonstration test.

COQUALMO :

Sunita Chulini's COQUALMO model derived from Barry Boehm's et. al. work on the COCOMO and COCOMOII models[i]. This model was developed by using Boehm's risk factors for predicting fault density instead of schedule estimation. The weighted correlation between COCOMO II risk factors and the fault densities were determined by an eight member panel of industry experts in a two pass study called the DELPHI. The COQUALMO method was set up in a spread sheet (answering questions on a spreadsheet tool about defect insertion and defect removal effectiveness) for the code under test risk factors, producing a theoretical fault density for the code expressed in defects per KSLOC.

EPML:

The EPML[ii], was also set up on a spreadsheet and used to estimate the CMM level of the software project, then fault density by CMM maturity level was determined by data available from SEI. The question was would the EPML estimate of maturity level and SEI defect density ranges match the COQUALMO estimation?

Industry Fault Density Table:

Donald Reifer has published fault density ranges by software industry type [iii], would the fault densities estimated make sense when compared to other and like industries? How would our predicted fault densities compare to that of mission critical codes, avionics codes, medical codes, and commercial codes? We expected that research software fault density would not be as low as mission critical or avionics software fault density because those codes are subjected to infrequent changes and more stringent testing, but perhaps better than most commercial software. Would our predicted fault density map to industry research?

Static Code Analysis:

Static code analyzers are used to find questionable coding practices, such as uninitialized variables, overflows, null pointer dereferencing, memory leaks, and many other types of problems. Originally these tools were designed to find security vulnerabilities, but we have found them useful for finding coding problems missed by compilers, inspections, and testing. The tool to be used was Klocworks, which has been used on over six million lines of code at LLNL, finding over 7,000 potential defects to date. Would the fault density prediction correlate to the number of defects found by static analysis?

Cyclomatic Complexity:

Cyclomatic complexity is another metric that the Klocwork tool collects. Complexity can be viewed as the number of different paths through a software component. A loop counts as a single path regardless of the Iterator value. For instance a component consisting of a sequence of statements with no loops or conditional statements would have a cyclomatic complexity of one. (A single test case could cover all paths). A single if-then else statement would be a complexity of two, a single nested if-then else three. A case statement would be a complexity equal to the number of choices. A rule of thumb is to try to keep complexity under 10 in a module of code. The theory is the more complex the code the more likely it will have defects or undesired side effects when it is modified. The actual metric in the Klocwork tool is the number of components with a complexity greater than a user supplied threshold. The plan was to compare the historical defect reports on the codes to the categories of defects found by source directory name in the code and see if the more complex areas of code had more found defects historically. While not a direct comparison to fault density, a good correlation may help us prioritize future testing.

Reliability Demonstration:
The predicted fault density would be used as an input to the Musa model constructed on a spreadsheet to predict the number of hours the software should run without encountering a failure. Traditionally tested software would then be run continuously (24/7) while stimulated by seeded random inputs to see if the software could run as long without a failure as the predication (Monte Carlo Reliability Testing).

Findings:

COQUALMO:
The codes used in the experiment were evaluated using the COQUALMO model. The questions were answered by a certified CSQE who was familiar with the code team tools, processes, and staff. The results of the questionnaire were an estimated fault density of 2.7 per KSLOC. The spreadsheet used is available for evaluation.

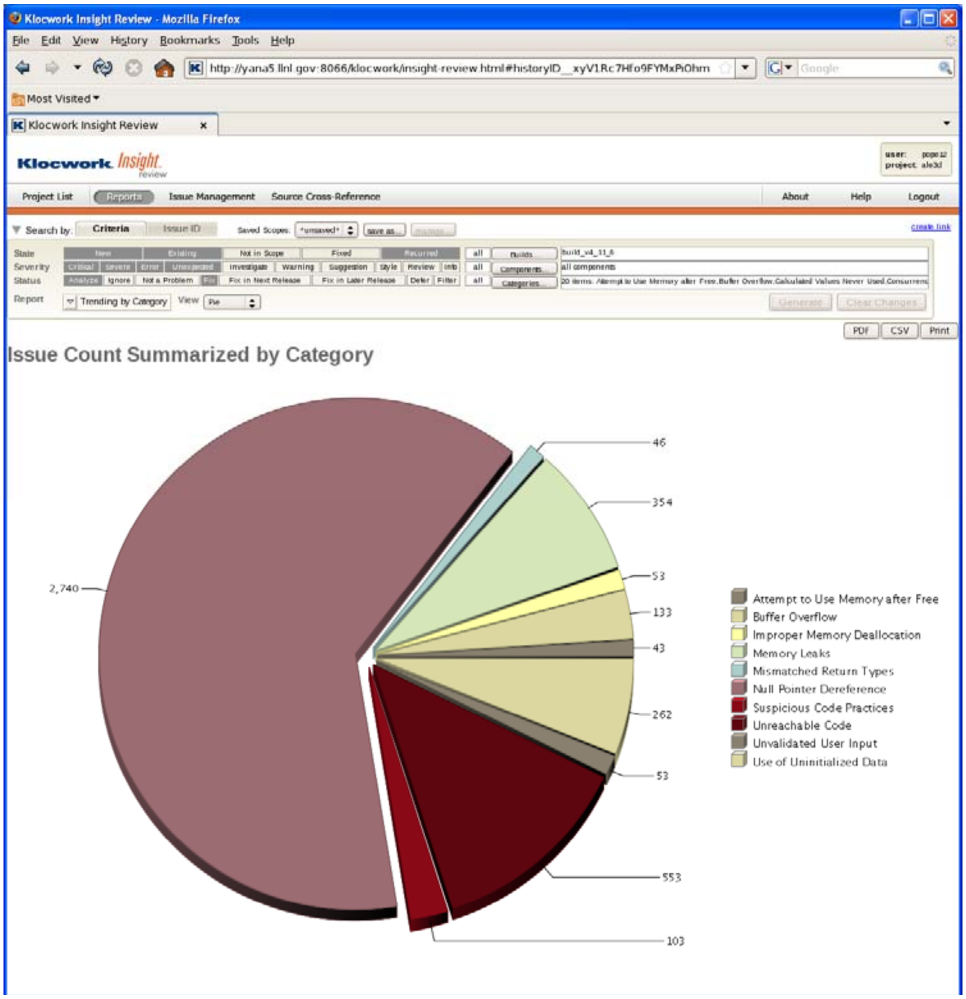
EPML:
An estimated Process Maturity model was completed on the codes used in the experiment. The questions were answered by a certified CSQE who was familiar with the code team tools, processes, and staff. The estimated CMM level was 3.4. The fault density for maturity level 3 codes is .27 defects per function point[iv]. At 104 lines of code per function point for the languages used, this is equivalent to .27 defects per 104 lines of code or 2.6 defects per KESLOC using the backfire method (70% C++, 20% C, 10% Fortran). This is very close to the COQUALMO prediction of 2.7. The spreadsheet used is also available for evaluation.

Industrial Fault Density Table:
The following table gives a compilation of a number of software projects by industry done by Donald Reifer.[v] The fault density prediction of 2.7 per KSLOC or 2.6 per KESLOC would indicate that the codes used in the experiment are within the low range of most commercial codes, but have a higher fault density than most DoD military codes. This is consistent with earlier speculation on the code based on Qualitative observations.

Application Domain	Number Projects	Error Range (Errors/KESLOC)[1vi]	Normative Error Rate (Errors/KESLOC)	Notes
Automation	55	2 to 8	5	Factory automation
Banking	30	3 to 10	6	Loan processing, ATM
Command & Control	45	0.5 to 5	1	Command centers
Data Processing	35	2 to 14	8	DB-intensive systems
Environment/Tools	75	5 to 12	8	CASE, compilers, etc.
Military -All	125	0.2 to 3	< 1.0	See subcategories
§ Airborne	40	0.2 to 1.3	0.5	Embedded sensors
§ Ground	52	0.5 to 4	0.8	Combat center
§ Missile	15	0.3 to 1.5	0.5	GNC system
§ Space	18	0.2 to 0.8	0.4	Attitude control system
Scientific	35	0.9 to 5	2	Seismic processing
Telecommunications	50	3 to 12	6	Digital switches
Test	35	3 to 15	7	Test equipment, devices
Trainers/Simulations	25	2 to 11	6	Virtual reality simulator
Web Business	65	4 to 18	11	Client/server sites
Other	25	2 to 15	7	All others

Static Analysis:
The Klocwork static analyzer used compiles and builds the code, and then compares the code to approximately 1,600 heuristic rules that are considered secure coding practices. The codes in the experiment are not so much concerned with security issues as they are not available to the general public and used on closed networks. However most security vulnerabilities in codes are created by undesirable coding practices, the code may function correctly, but may be more vulnerable to hackers. The Klocwork tool found 4,295 coding practice and flagged them for analysis on the multi-physics code. This was a found suspected defect density of 7.4 per KSLOC. Our experience with the tool is that 85% of the found issues are defects that will need to be fixed, the rest are false alarms. This would lower the defect rate found by the static analyzer to 6.3 per KSLOC. The number is much higher than the predicted defect rate of 2.7, however many of the fixed defects would probably never been found by testing (a fault manifest into a failure). For instance in the defect distribution pie chart shown below, over half of the defects are null pointer dereferences. Most of these would occur only if memory was not available when requested, so finding them would require the test case to control the allocation of memory to find them. If this class of defect is removed then the defect density is (4,295-2740 = 1,555 or 1,555/573 = 2.7, really scary). Which brings up an interesting observation that is how historical fault densities have been determined? If they have been determined by empirical observations of testing, then they may be

low, since it is not possible to prove all defects have been found. Our static analysis is giving new insights into defect rates and allows for finding and fixing of problems difficult to find in inspections or testing.


Klocwork Insight Review - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://yana5.llnl.gov:8066/klocwork/insight-review.html#historyID__xyV1Rc7Hfo9FYMxPiOhm
Google
Most Visited
Klocwork Insight Review
Klocwork Insight review
Project List
Reports
Issue Management
Source Cross-Reference
About
Help
Logout
Search by:
Criteria
Issue ID
save as...
manage
create link
State New Existing Not in Scope Fixed Recurred all
Severity Critical Severe Error Unexpected Investigate Warning Suggestion Style Review Info all
Status Analyze Ignore Not a Problem Fix Fix in Next Release Fix in Later Release Defer Filter all
Generate
Clear Changes
PDF
CSV
Print
46
354
53
133
43
262
53
553
103
2,740
Attempt to Use Memory after Free
Buffer Overflow
Improper Memory Deallocation
Memory Leaks
Mismatched Return Types
Null Pointer Dereference
Suspicious Code Practices
Unreachable Code
Unvalidated User Input
Use of Uninitialized Data

user: pope12
project: ale3d

Saved Scopes: *unsaved*

Builds... build_v4_11_6

Components... all components

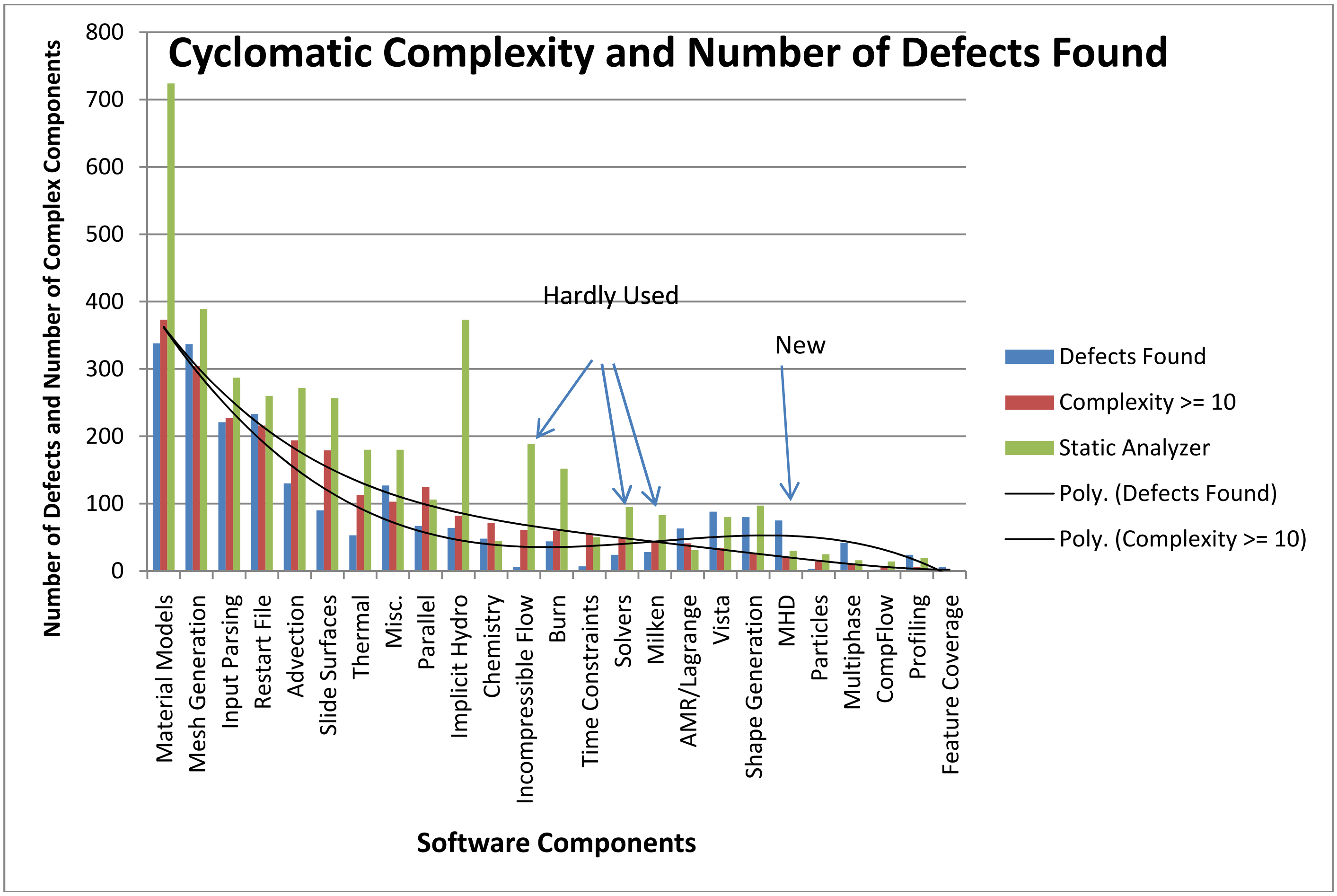
Categories... 20 items. Attempt to Use Memory after Free,Buffer Overflow,Calculated Values Never Used,Concurrenc

Report: Trending by Category View: Pie

Issue Count Summarized by Category

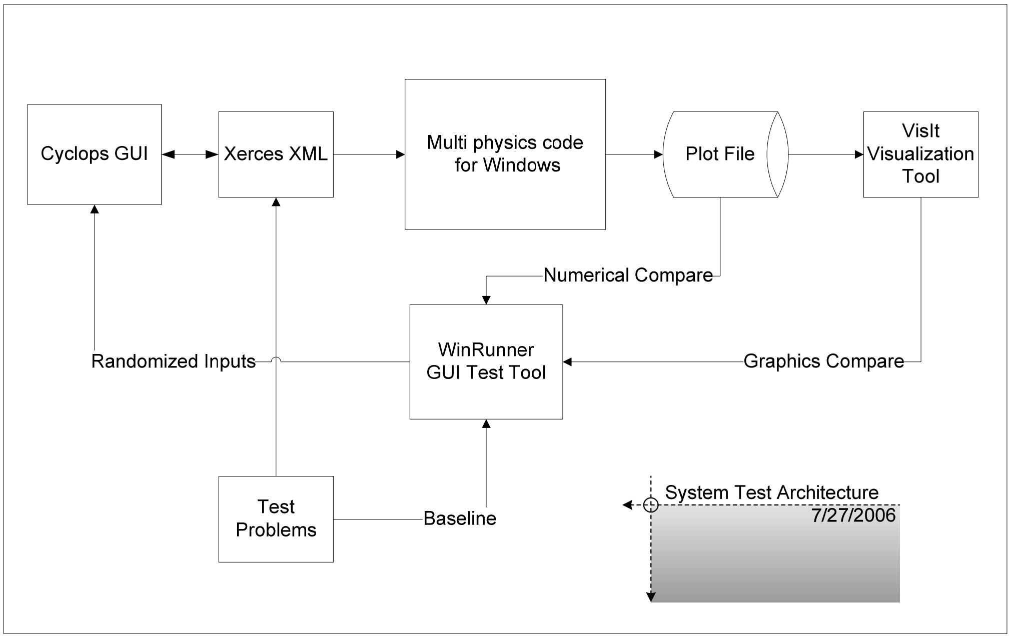
Cyclomatic Complexity:

The Klocwork tool was also used to calculate the cyclomatic complexity of the codes. The tool yields the number of components over certain complexity threshold. Three values were used for the thresholds, 5, 10, and 20. The number of components exceeding threshold complexity was then put on a spreadsheet and plotted by feature (actually a source file directory name). The defect tracker used for the project was also exported to spreadsheet and plotted number of defects found and fixed against feature. The defects had been collected for over five years. Not all of the features and directory file names lined up, so some combining was performed. Lesson learned here, name the source directories the same names as the categories in your defect tracker pull down menu. Interestingly enough the best correlation between defects found and complexity was at the 10 threshold. At five the number of complex components was too great for a good quantitative correlation, at 20 the number of complex components were too small. As shown in the chart below, there is a correlation between the complexity and the defects found over five years. A fourth order polynomial fit line is plotted for defects found and complexity of ten or greater. The maximum gap between the two best fit lines is about 40 defects. Forty defects represents about plus or minus 2%. The same process for a threshold of 5 or greater complexity was plus or minus 15% and for a threshold of 20 plus or minus 5%. A simple interpretation of this result is that for every component with a complexity count of ten or greater, one can expect a defect to be found in that component. It would appear besides this simple and highly questionable quantitative conclusion (heuristic); qualitatively most defects can be expected in areas of higher complexity in the code. The static analyzer defects found are also plotted for reference and do also correlate qualitatively, with some outlier points. The outlier points emphasize that drawing conclusions only by using defects found in testing or inspection can be misleading, as there may be a big pocket of defects yet undiscovered because of the difficulty finding them with conventional tests. What is not shown on the plot is that the leading defect area in the defect was the make and build process, which of course is not related to the source code, but none the less is an important finding in terms of focusing process improvements. Complexity then would seem to be a somewhat reasonable way to predict defects, especially the highest probability areas for defects. As the complexity numbers go down, the correlation tends to widen. Factors that seem to effect the correlation between defects found and complexity are how frequently the code component is used, new code versus modified code, and the experience level of the author, however at higher levels of complexity these factors do not seem to dominate. Further studies are planned to see if other codes exhibit the defect per complexity of ten or greater phenomenon.



Reliability Prediction:

In order to check the ability to predict reliability (a highly controversial area) the plan was to use the predicted defect density from COQUALMO and ECCM (and somewhat confirmed as “in the ballpark” by the static analyzer and industry table) as the fault density for input to the Musa reliability calculation. Musa’s model was developed at Bell Labs for AT&T telephony software many years ago. It treats the software as a statistical problem, there are so many faults in the code, there is a very small chance that when a line of code with a fault is executed it will be in a state that will generate a failure that is perceivable to the user. The faster the CPU speed and duty cycle of the software the more likely that these encountered faults will manifest a failure as a function of time. The Musa model has a simplifying assumption however that faults (or defects) are uniformly distributed throughout the code. In the complexity analysis above there was a good example of this not being true; defects tend to be greater in the more complex areas of code, not uniform. Defects tend to cluster where lesser skilled programmers have created code, etc. However Musa developed his reliability study before complexity was developed by Tom McCabe. An obvious question then is can the basic reliability model be improved by using complexity profiles instead of uniform distributions? The probability of encountering a defect in the material component of the multi-physics code would be much greater than in the MHD component in our example. However, we used the reliability model as described in the literature[vii] to see what would happen. The test bed pictured below was used to stimulate the software with random inputs. The exact inputs to stimulate and over which ranges were selected with help from the developers and code physicists. It took some trial and error both with the software under test and the test tool to attain a viable reliability test. Time synchronization between the test tool and the application was the most challenging.



The reliability prediction tool created, for a fault density or 2.7per KSLOC the following table (it is in the COQUALMO workbook):

Hours	Years	Probability of Failure Free			Probability of Failure	
1		99.91%			0.09%	
3		99.72%			0.28%	
8		99.27%			0.73%	
160		86.30%			13.70%	
480		64.27%			35.73%	
1000		39.81%			60.19%	
5000	0.6	1.00%			99.00%	
10000	1.1	0.01%			99.99%	
50000	5.7	0.00%			100.00%	
100000	11.4	0.00%			100.00%	

Our goal was to see if the software would run about 500 hours failure free when stimulated by random inputs continuously. A failure was defined as a crash, math error, assertion error, system freeze, extremely slow performance, or outside of the range of some rule of thumb values selected by the code physicists. Note that it would be possible to have an error of a slightly wrong answer and not be able to detect it; this is because we did not attempt to duplicate the software under test, as the duplicated software would be no more trustworthy than the software being tested. Many tests had already been run on this software to catch those types of errors with verified and validated expected results using value compares and curve compares within tolerances. The result was we were able to run for 500 hours without a failure as defined for the reliability test.

The Bonus, Bug Driven Heuristic Testing:

An unexpected benefit of performing the randomized (Monte Carlo) reliability testing on the atmospheric dispersion code is that we did trip across certain rare fault types that allowed us to create specific tests to look for more of those fault types. For instance we found a certain chemical that had an atomic weight of null. Null is not a legal atomic weight. So we wrote a heuristic test to check that all chemical's atomic weights were not a negative number, null, 0, space, carriage return, linefeed, etc. This heuristic test did find additional chemicals that had wrong atomic weights. We also found a city of the world with no radius (circle of nearby weather stations to get meteorological data). We wrote a heuristic test to find any city of the world that had a negative number, null, 0, space, carriage return, linefeed, etc. as a radius. We found more cities with no valid radius. We followed with a test to check the minimum and maximum radius values that should be allowed. We also ran another heuristic test to check the maximum wind speed reported by meteorological stations around the world. We found some values over 200 miles per hour that could not be explained. We also ran a minimum value check on meteorological stations around the world, using a heuristic given to us by a meteorologist. The rule was for a wind speed of less than 2.6 knots per hours we should get a direction of null, for a wind speed of greater than 2.6 knots per hour we should get a direction reading of 0-359. (The wind direction can not be determined if these is no or very small amount of wind). This test found numerous meteorological stations throughout the world that were not working correctly. All in all we found several hundred corrupted data values that would have been unlikely to find using conventional testing techniques. We also found another interesting thing, any test running at 3:00am failed. This turned out to be the server, which rebooted itself to clear memory leaks at 3:00am. The code to do this was put in many years ago as a late night fix, but was forgotten about. Since this system is potentially used 24/7 for tracking worldwide events the old patch had to go and a leak checker used to find and fix the leak sources. The same was true for the multi physics code; we were able to create new input tests that checked for conditions that had never been tried before, and failed. Lesson learned here is that a powerful form of testing is the Monte Carlo test to find defects and then use those bugs to create tests that use heuristics to find all instances of that error type, especially in data intensive systems.

Future Work:

The reliability of the software can only be as good as the test suite used to test it. Future work includes defect seeding to improve test suites. The plan is to seed about 8,000 defects, one at a time, into the multi-physics code. Selecting the type of defects to seed and location of the seeded defect is a non-trivial task. A special tool is being developed to help with this task. Some thoughts on what types of defects to seed would be to look at the type of defects already found in the code and the types of defects static analyzers look for. The location of the seeding could be at random or use complexity profiling discussed earlier to seed more complex areas heavier. The use of parallel computers would allow all 8,000 mutated source codes to compile and execute simultaneously against their test suites that have formerly passed all tests. Those processors that do not detect their seeded defect (all tests still pass) would then be checked by another tool using a reverse slicing technique to see what input condition (if any) could be stimulated to detect the seeded error. If an input condition or conditions can be found to stimulate the seed and detect the error, then that input would be a candidate test to add to the test suite to improve it. Also to reduce the wall clock time needed to run reliability tests, use of the parallel testing environment to accumulate hours faster. Lastly as stated before, continue to evaluate other codes to see if the complexity prediction, both quantitative and qualitative holds true.

[i] Chulani, Sunita “Results of Delphi for the Defect Introduction Model” USC-CSE 06/30/97 and Boehm, Barry et.al. “Software Cost Estimation with COCOMOII”, pg 254-266, Prentice Hall, 2000 ISBN 0-13-026692-2

[ii] Barry Boehm on page 34 of “Software Cost Estimation with COCOMOII”

[iii] Donald Reifer, “Industry Software Cost, Quality, and Productivity Benchmarks”, DoD Software Tech News, July 2004

[iv] Capers Jones, “Software Assessments, Benchmarks, and Best Practices”, Addison-Wesley, 2000.

[v] Donald Reifer, “Industry Software Cost, Quality, and Productivity Benchmarks”, DoD Software Tech News, July 2004

[vi] KESLOC is thousands of equivalent source lines of code. KESLOC is derived by converting Function Points (FP) to ESLOC using the language conversion factors supplied from the International Function Point Users Group (e.g., one FP is expressed as so many lines of C or Java using such backfiring tables).

[vii] John D. Musa, *Software Reliability Engineering* (McGraw-Hill, 1998)

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.