ELSEVIER

Information and Software Technology 42 (2000) 801–807

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

Improving software quality using statistical testing techniques

D.P. Kelly*, R.S. Oshana

Raytheon Company, 13500 N. Central Expressway, P.O. Box 655477, Dallas, TX 75265 USA

Abstract

Cleanroom usage-based statistical testing techniques have been incorporated into the software development process for a program in the Electronic Systems business of Raytheon Company. Cost-effectively improving the quality of software delivered into systems integration was a driving criterion for the program. Usage-based statistical testing provided the capability to increase the number of test cases executed on the software and to focus the testing on expected usage scenarios. The techniques provide quantitative methods for measuring and reporting testing progress, and support managing the testing process. This article discusses the motivation and approach for adopting usage-based statistical testing techniques, and experiences using these testing techniques. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Usage-based statistical testing; Cleanroom

1. Introduction

The Electronic Systems business of Raytheon Company develops real-time embedded software for defense systems. Recently, a large real-time embedded software development effort in Electronic Systems incorporated Cleanroom techniques into their software development process [1–3]. Cleanroom software engineering techniques consist of a body of practical and theoretically sound engineering principles applied to the activity of software engineering (Fig. 1). A main component of Cleanroom is the use of usage-based profiles to test the software system. These usage profiles become the basis for a statistical test of the software, resulting in a scientific certification of the quality of the software system. The fundamental goal driving the incorporation of Cleanroom techniques was to improve the quality of the software delivered to the system integration phase of system development.

2. Historical testing process

Historically, the testing phases of the software development process consisted of unit testing, unit integration and finally product level testing (Computer Software Configuration Item, or CSCI in Department of Defense parlance), which culminates in a formal qualification test. Unit testing and unit integration testing are performed by the software development team. CSCI level testing and qualification testing are performed by a team composed of software development team members, system engineers and software quality engineers.

Prior to unit testing, the development team performed a formal inspection of the software source code. Unit testing is a structural testing activity and the completion criteria for unit testing is 100% path coverage at the unit level and verification of correct functionality of the unit. Unit testing was executed on the host development environment and supported with a custom built automated unit test environment. The test environment was capable of executing the unit under test, feeding input data to the unit, capturing outputs and internal data structures, comparing results to expected values, and generating test reports. Significant effort was spent to develop and maintain the unit test environment.

Unit integration testing has both structural and functional aspects. Early in unit integration testing, the focus is structural as modules of the software are integrated together to form complete functionality. Later in the process, the focus becomes functional as complete functions are tested. Unit integration was partially performed in the unit test environment and partially on the target hardware using another custom built test environment. CSCI testing is a functional test with the testing team developing and executing the detailed test procedures for qualification tests specified by an Independent Verification and Validation (IV&V) agent. Considerable effort was required by the software development team to generate the test procedures associated with the qualification tests. CSCI level testing was performed on the target hardware.

* Corresponding author.
E-mail address: dkelly@raytheon.com (D.P. Kelly).

PII: S0950-5849(00)00124-5

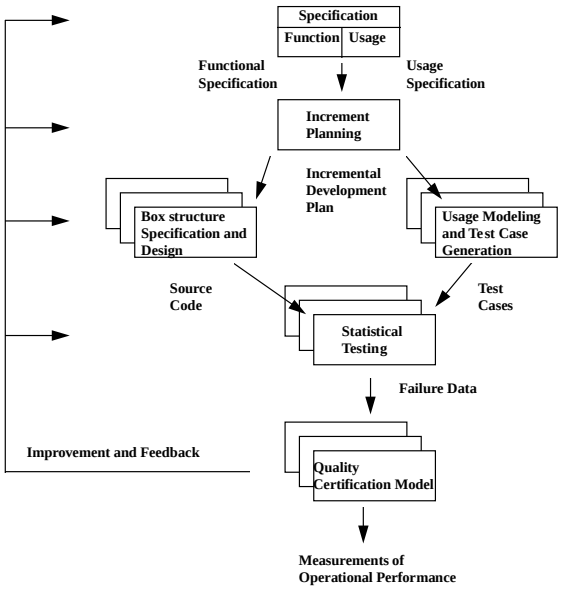

Fig. 1. The cleanroom software engineering process.

The software ultimately delivered by this development process and associated system integration activities exceeded expectations with an error density of 0.07 Software Trouble Reports (STR) per KLOC reported during the first year of operational use. During unit testing, a relatively small number of defects was found and most of the effort was spent in developing test cases to achieve 100% path coverage. Experience showed that the most insidious defects resulted from high level interactions of complete software functionality being used in scenarios, which the lower level testing failed to expose. These defects were found mostly during system integration.

3. Motivation to improve the testing process

As the process for the new development effort was being defined, the software testing process was reviewed for possible enhancements and improvements. While the historic testing approach had yielded ultimately high quality software to the customer, there were issues with the testing approach:

1. The cost of achieving 100% path coverage in unit testing considering the quantity of defects found by the activity.
2. High level defects found in system integration required rework during a schedule choke point in the system's development.
3. The developer's knowledge of the structure of the software tended to bias their tests.
4. The difficulty in accurately judging progress during testing. The testing approach was schedule driven with no quantitative stopping criteria.

Unit testing was an obvious candidate for review for improvements to the historical approach. Adequate unit testing environments are not readily available commercially. To build a unit testing environment, significant effort by the developing organization is required. While in theory unit testing is performed on a unit whenever a change is made to it, in practice unit testing is performed when the software is initially developed and in later stages of the program only when a unit is redesigned or significant new capabilities added. Furthermore, with the static checking capability of modern compilers and the use of structured formal code inspections, the number of defects discovered during unit testing was relatively low. Using Cleanroom box structure design and the associated verification techniques reinforces the inspection process. The verification techniques provide a mechanism for rigorously reviewing software source code for correct functionality. When allocating limited test budgets and schedules, eliminating path coverage-based unit testing has obvious appeal.

Functional unit integration and product level testing approaches needed strengthening to detect defects resulting from the complex streams of input stimuli. Historic functional testing was compromised by the developer's intimate knowledge of code structure and hence biased their view of what should be emphasized during testing. Methods for quantitatively judging progress also need to be improved as previous measures were typically 'M out of N' style metrics of test cases executed, passed, and failed. These measures tend to encourage linear extrapolations to estimate completion dates which in turn proved to be overly optimistic.

4. Cleanroom usage-based statistical testing

The Cleanroom usage-based statistical testing process has several aspects that addressed directly these concerns with the historical testing approach. Usage-based testing focuses effort on modeling expected usage scenarios for the software [4,5]. Usage models can be considered as a complete generalization of Jacobson use scenarios. Jacobson use scenarios typically define a single execution path through the software. A usage model will generate all possible use cases including multiple sequential execution paths. Conceptually, a usage model is a directed graph with nodes and arcs, and transition probabilities associated with each arc in the graph. A hierarchy of usage models with appropriate transition probabilities is used to encompass the full range of usage possibilities including error cases and other 'abnormal' usage scenarios. The fidelity with which the usage models reflect real-world usage is dependent upon the accuracy of the transition probabilities used in the models. While the probabilities do not need to have absolute fidelity with the operational usage their magnitude and relative ratios need to be correct. Information needed to build the models can be derived from

Table 1
Usage model size details

Usage model	# Usage states	# Arcs
Products with both control and algorithm functionality		
Product A—normal usage	162	1455
Product A—adverse usage	16	39
Product B—normal usage	42	379
Product B—adverse usage	24	67
Product C	15	44
Product D	45	65
Average	50.7	341.5
Products with only control functionality		
Product E—adverse usage	16	86
Product E—specialized	8	17
Product F—normal usage	7	38
Product F—normal usage 2	11	37
Product F—adverse usage	10	52
Product G—normal usage	7	11
Product G—normal usage 2	21	50
Product G—normal usage 3	15	37
Product G—normal usage 4	6	19
Average	11.2	38.6

historical data, user experience, operational scenario documents and other such sources.

Usage modeling provides another view of the required software functionality and provides a mechanism for systems engineering, software developers, and customers to review usage scenarios for the software. Using a hierarchy of usage models, it is possible to develop system-level usage models and consistently flow the usage scenarios to lower level software product testing. By focusing test development on operational usage scenarios with model reviews including system engineering personnel, proper focus on system level issues is incorporated and flowed to lower levels of testing than experienced previously. The usage models are developed by a separate team from the team developing the source code and this avoids any biases inadvertently introduced from knowing the details of the software structure.

The statistical testing techniques of cleanroom provide alternative quantitative methods for measuring and reporting progress. The measures of reliability, discrimination and distance are specifically defined in the Cleanroom literature [4]. These measures provide insight into how well the sample of tests executed reflects the input population and quantifies the reliability of the software. Progress can also be measured and reported in terms of node and arc coverage of the usage models. This latter reporting mechanism is especially useful for providing a view of the extent of usage model testing performed and that remaining. The testing techniques of Cleanroom provide support for deciding when to stop testing and advance the software to the next stage of system development. 'What-if' scenarios can be run on the models and quantitative measures returned on the benefits to continued testing. This capability can be used to determine when testing has reached the point of diminishing returns.

Historically, during unit integration and CSCI level testing, a relatively small collection of test cases is used to verify the software product's functionality. Maximizing the practical benefits of the Cleanroom approach to testing requires an automated test environment. Like unit testing environments, commercial statistical testing environments are not readily available. For our development we built an automated test environment. Given that end-to-end user functionality is tested during statistical testing, our experience has been that the Cleanroom test environment required significantly less effort to develop and build than our historic unit test environment (approximately 20% of the effort).

5. Incorporating cleanroom testing techniques

Following a consensus building process that included affected parties: systems engineering, customer representatives, and software developers, a revised test strategy was defined, which incorporated Cleanroom concepts into the testing approach and addressed classic structural testing concerns. Usage models are developed that address the stratification of usage scenarios: normal use, adverse use, error case use. A minimal cost cover sample of test cases for the usage models is generated. This minimal covering set is the first set of test cases executed on the software increments, which are instrumented for code coverage. This approach provides insight into the structural coverage of the software and provides a sanity check of the usage models against the software design. If the usage models accurately model the required software functionality then a high level of code coverage should be expected. Informal reports from organizations using this approach report coverage rates of $>$90%. Code not executed by the minimal covering set is examined to understand why it was not executed. Possible reasons for non-executed code include deficiencies in the usage models, excess functionality in the software, and error case paths that were not executed.

In general, no unit testing is required as part of the defined software development process. The development process does not forbid unit testing, and no explicit steps are taken to prohibit developers from performing unit testing. However, no support is provided for unit testing and developers electing to perform unit testing must meet the same schedule and budget goals of developers not performing unit testing. During the software implementation phase the development teams formally inspect the software. Entry criteria for the review includes clean compilation of the software and appropriate preprocessors to perform static code analysis (i.e. the C language lint tool). The inspection process includes the Cleanroom verification steps for verifying software functionality. Prior to releasing the software to the testing teams, the development teams are responsible

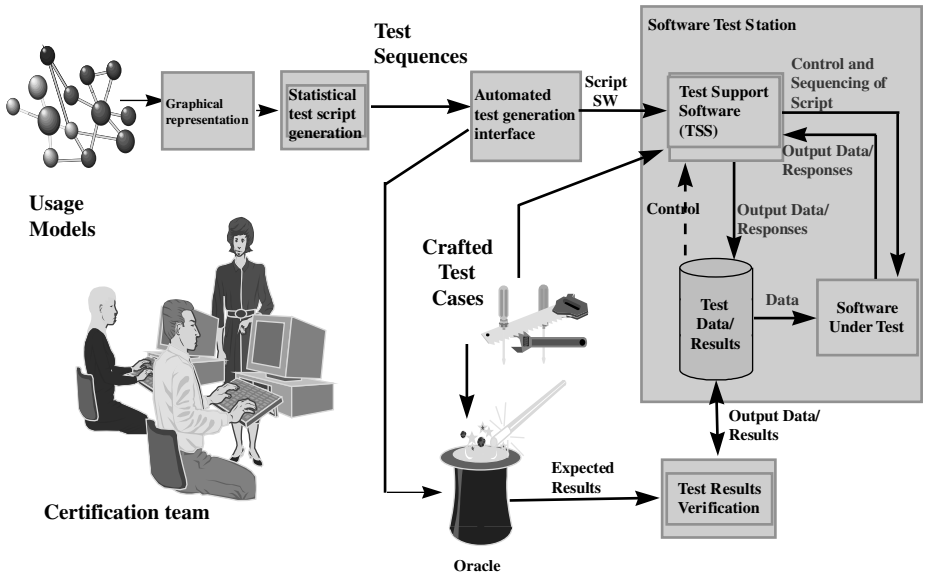

Fig. 2. Automated testing environment.

for successfully compiling the software and generating the load module for the software increment being developed. The development teams do not perform any testing on the resulting load image.

The number of usage models and size of the models varied across the software products. On average there are six models per product with the number ranging from 2 to 8 for the set of products developed. The size of the usage models varied with a wide range in both number of usage states (nodes) and arcs. Details for a representative set of usage models are shown in Table 1. Products containing a mix of control software and algorithm software generally had larger usage models than products containing only control functionality.

The usage models include references to user defined functions, which generate the detailed input data for the software under test. These functions are associated with the arcs in the usage model, which drive the transitions between the usage states (nodes) of the model. Input equivalence partitioning, bounds checking and other decomposition of the input domain of the data are explicitly addressed via the user functions. These user functions are included in the analysis of the ability of the usage model to generate adequately and test samples from the input domain of the software. Taken together, the usage models and the user-defined input data generation functions are capable of generating far more data sets and test sequences for execution than hand crafted test cases.

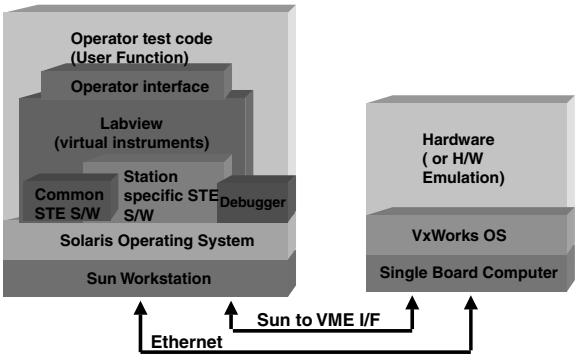

Fig. 3. Software components for the testing environment.

6. Developing an automated environment

An automated testing environment was developed to execute the statistically generated test scripts during the certification phase (Fig. 2) [6]. Usage models are created directly from the box structures developed during the specification phase. The usage models are translated into the appropriate test grammars and executed in a special test equipment (STE) environment. Product level requirements that are difficult to verify using statistical testing are verified via specially crafted test cases. In a few algorithmically intensive products, many of the algorithm related requirements are verified using one or more crafted test cases. Virtually, all of the control-based requirements are verified using statistically generated test scripts based on a set of usage models representing different stratifications of the system (normal use, adverse use, etc.). The minimal covering set of test scripts from the usage models is achieving 80–90% statement coverage in most products.

The test environment includes the following major components (Fig. 3):

- *Operator test software*—also referred to as the ‘user function’. There is a different user function for each of the usage models developed for the software. These

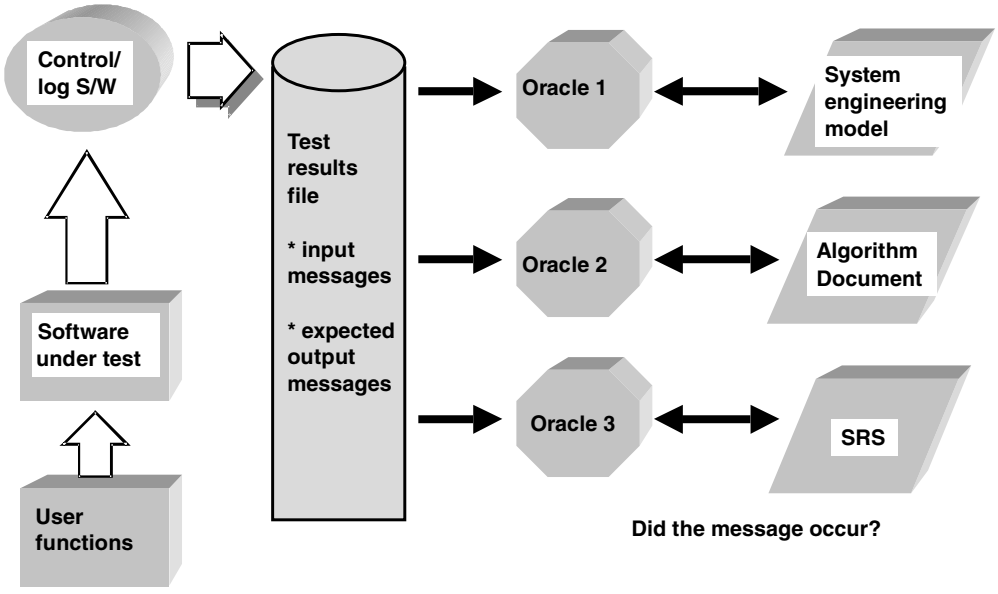

Fig. 4. Testing environment oracle.

programs are written in C and generate the message sequences that drive the software under test for each of the test scripts generated from the usage model.

- *Labview interface*—this software is composed of a number of different Labview 'virtual instruments' used to provide an interface to the operator executing the tests.
- *Station specific Special Test Equipment (STE) software*—this software provides the low level functionality required by each of the different software test stations.
- *Common STE software*—this software is the Application Programming Interface (API) to the rest of the software. It provides the capabilities to watch for certain events occurring in the software under test, log those results, and provide the information to the user function software and Labview virtual instruments.

Once a test has been executed and the input and output events logged in an output file, the data is parsed into several different oracle files (Fig. 4). Each of these oracle files is used for a different pass/fail criteria. 'Generic' oracles are used to determine if the high level sequencing of the test was performed correctly. Other oracles are used to examine the data to determine product specific pass/fail criteria such as:

- Does the data reflect the expected outputs as determined by the system engineering model(s)?
- Does the raw output data match the expected outputs defined in the algorithm document?
- Does the output message sequence match what was expected in the Software Requirements Document?

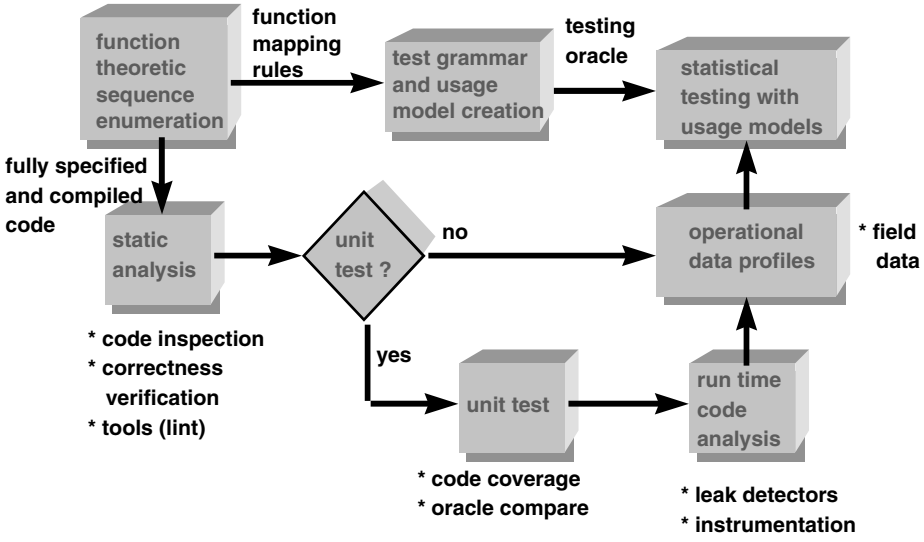

Fig. 5. Hybrid algorithm software testing model.

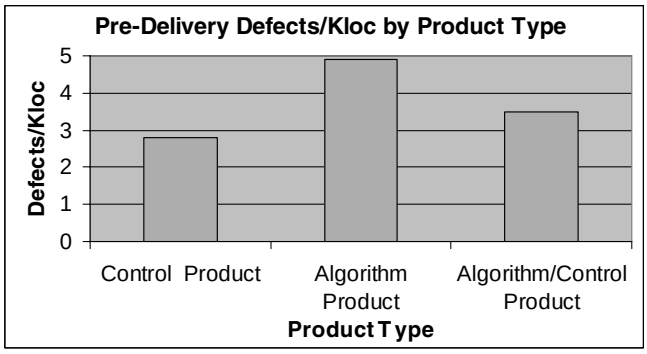

Fig. 6. Pre-delivery defects by software type.

7. Testing algorithmically intensive software

Some of the software products being developed implemented highly mathematically intensive signal processing algorithms. As this type of processing tends to be very sensitive to input data sets that are large and complex, a hybrid testing approach was utilized for this software. A set of cardinal test cases had been developed by the algorithm development community within the program to verify small functional blocks of the algorithms, which equated to routines or modules in the operational software. From a practical perspective, it is impossible to verify fully these modules at a user end-to-end functional level and consequently these signal processing routines undergo 'unit level' testing utilizing the cardinal data sets. These tests are designed to test the mathematical functionality of the routines and are not explicitly designed to guarantee 100% path coverage. Usage model-based testing is utilized for this software to check overall end-to-end control flow and proper functioning in the larger context of the system operation. This hybrid approach to certify algorithmically intensive software included the following steps (Fig. 5):

- a formal code inspection and correctness verification phase;
- an optional level of unit and function testing (based on the nature of the algorithms), which included both static and run time analysis;
- an operational data profile phase using real data collected from various user environments;
- a final statistical testing phase using usage models developed for different user and usage stratifications.

8. Limitations to modeling

There are limitations to modeling software systems that must be understood in order to properly analyze the statistics generated when using statistical techniques such as the ones discussed in this paper. The quote "All models are wrong, but some are useful" has some truth to it. The main point to remember when modeling any type of system using any technique is that all modeling approaches lack, to some extent, the naturalness for representative power. The Cleanroom Markov approach has limitations in handling some of the common issues such as counting and concurrency. There are methods for circumventing some of these issues but they can result in state explosion and hence larger models. Also, the more abstract a model is, the less confident one should be in the predicted reliability generated by the tools. With prudent use, the statistical techniques discussed in this paper can be very useful in any testing organization and help lead to higher quality software products.

9. Results

Cleanroom usage-based statistical testing along with sequence-based specification techniques were incorporated into the program's software development process. Overall, the defects found out of phase during the software development process have been reduced compared to historical data. There have been very few defects found in the software fully specified using sequence-based enumeration (Fig. 6). Most of the defects are traced to language semantics, algorithm misunderstanding, and other semantic mistakes. Software defects were detected during the testing process that historically had not been found using other testing techniques. Cleanroom techniques do require a commitment and a disciplined approach and matched well with the mature CMM framework established for the program [7]. The usage-based statistical testing approach worked very well for control-oriented software. For algorithmically intensive software, a hybrid testing approach needed to be used to properly verify algorithm implementation. The focus on usage-based testing has proven to be very effective. In one case, the certification team found defects immediately after the development team handed over a product they felt was working (based a significant amount of function testing).

10. Conclusions

The decision to incorporate Cleanroom techniques into the software development process was driven by a desire to cost-effectively improve the quality of our software while providing quantitative support for managing the testing phases of software development. Cleanroom testing techniques addressed these goals. Removing the structural path-coverage based unit-testing step was a concern and ultimately the issue came down to whether the probability of defects, historically detected in unit test, slipping through usage-based testing scenarios was sufficient to warrant the cost and time of unit testing for 100% path coverage. This concern was addressed by executing a minimal covering set of test cases on instrumented code, which provided insight into path coverage. As regards our quality improvement, cost-effectiveness and ease of adoption goals, Cleanroom statistical testing techniques have met or exceeded our expectations.

References

[1] H.D. Mills, M. Dyer, R.C. Linger, Cleanroom Software Engineering, IEEE Software September (1987) 19–25.
[2] D.P. Kelly, Robert S.Oshana, Integrating cleanroom software methods into an SEI level 4–5 program, Crosstalk, November 1996.
[3] R.C. Linger, Cleanroom Process Model, IEEE Software March (1994) 50–58.
[4] J.A. Whittaker, J.H. Poore, Markov analysis of software specifications, ACM Transactions on Software Engineering and Methodology, January 1993.
[5] G.H. Walton, J.H. Poore, C.J. Trammel, Statistical testing of software based on a usage model, Software Practice and Experience, January 1993.
[6] R.S. Oshana, An automated testing environment to support operational profiles of software intensive systems, 10th International Software Quality Conference, May 1999.
[7] R.S. Oshana, R.C. Linger, Capability maturity model software development using cleanroom software engineering principles, 32nd Hawaii International Conference on System Sciences, January 1999.