Experience Report: Error Distribution in Safety-Critical Software and Software Risk Analysis Based on Unit Tests

Stephan Ramberger, Thomas Gruber , Wolfgang Herzner

Division Information Technologies
ARC Seibersdorf research
Tech Gate Vienna, Donaucitystraße 1/4
A-1220 Vienna, Austria
stephan.ramberger@arcs.ac.at
thomas.gruber@arcs.ac.at
wolfgang.herzner@arcs.ac.at

Abstract: Systematic verification and validation measures are of essential importance in particular for safety-critical software. After a short introduction into the test environment and error categories, the paper presents the results of a unit test performed on a C++ software package for the European Space Agency in the ARC Seibersdorf research test lab. The authors analyse the error distribution and relations between software metrics and software faults and recommend guidelines for a less error-prone design of safety-critical software.

1 Introduction

Today there is an increasing trend to control safety critical systems by means of software. High safety requirements imposed by the authorities, as for instance in the rail domain, or an irrecoverable material loss in case of a system failure, like in space technologies, demand an extremely low failure probability. Accordingly, from the system level down to subsystems and components, quality management and software development process technologies have to be obeyed thoroughly, and especially systematic testing deserves a maximum of attention.

ARC Seibersdorf research has a long ranging expertise in the development and the quality management of safety-critical systems and maintains an accredited software test laboratory. This paper describes the results of a unit test in the test laboratory as part of the overall software quality assurance measures and an analysis of the relation between some of the software metrics and the failure rate, similar to [FE00].

The SUT (software under test) consisted of numerous C++ classes, and the task was a verification of the documentation and the code based on the European standard ISO/IEC 61508 [IC01]. As a test tool we used Cantata++ of IPL, running on an Alpha station with a TRUE64 operating system on it.

The goal of the software test was 100 % statement coverage with an appropriate set of test cases. The errors we encountered during the test were categorized and documented as SPRs (Software Problem Reports).

For managing the SPRs and the workflow of fixing and re-testing the bugs, we used the web-based tool WWQM (World Wide Quality Management), an in-house development of ARC Seibersdorf research. This was very helpful in analysing the results is detail and finding interesting facts about the correlation between static metrics of the particular software and the observed error rate.

The results of the test were documented in detail and are - of course - confidential. But some interesting findings of the test will be shown in this paper.

2. Distribution of Error Types

During our research we encountered more than one hundred various Software Problem Reports, which can be categorized as documentation errors, coding errors and incomplete coverage. The following figure shows the distribution of the errors found during the whole verification process. As we can see, the majority of the Software Problem Reports have been due to misleading, incorrect or insufficient documentation, followed by 30% of coding errors and one sixth of incomplete coverage.

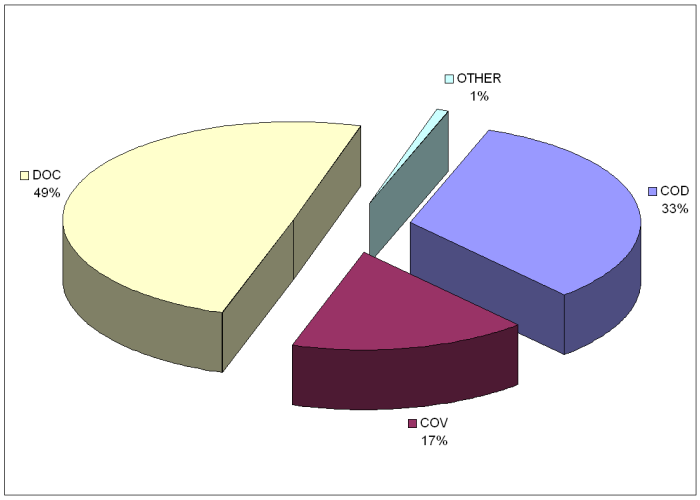

Figure 1: Error Type Distribution Chart

The three categories can be subdivided into smaller and more specific categories for this particular Software Under Test.

2.1 Documentation Errors

value never returned	54%	These methods define values that will not be returned in any possible case but have been defined by the method's description as being returned in some cases.
value not documented	31%	The value returned by the method has not been defined as a valid return value.
documentation error	10%	This relates to textual errors like quoting wrong OUT values or describing used structures incorrectly.
doc. not sufficient	5%	The provided information is incomplete or does not specify the method properly.

2.2 Coding Errors

implementation: dependencies	38%	These are semantic or implementation specific errors, like variables that will not be initialized in every case.
wrong pointer handling	23%	A typical C or C++ error is to handle a pointer in an incorrect manner.
int checked against bool	13%	If using own structures for expressing the current state, return values should not be checked against the keywords “true” or “false” to prevent misleading comparisons.
Other	26%	Various errors could not be assigned to one of the first three groups but are not enough to be split into a distinct group.

2.3 Incomplete Coverage

defensive programming	65%	Assertions which are always true due to prior checks.
default or else branch	35%	The use of default or else branches that can not be reached due to checking every possible value before.

3 Relation Between Complexity and Error Count

After having analysed the different error types we took a closer look at every method to measure its potential risk. Therefore it was necessary to exclude the documentation errors, as they are not directly related to the software metrics we wanted to investigate.

The applied metrics has been derived from the NASA SATC (Software Assurance Test Center) [SA03], [Ro98], which assigns every single method to one of seven "Risk Areas" based on the lines of code and the cyclomatic complexity, which is the number of linear independent paths as a specialisation of McCabe's measure (edges – nodes + (2 * connected regions)) [MC82] with only one connected region. The SATC propagates that the lower the risk area number is, the lower is the risk of a potential error in one of the methods. In prior research work we have already matched Risk Areas of the NASA SATC model with the test results, cf. [RG03].

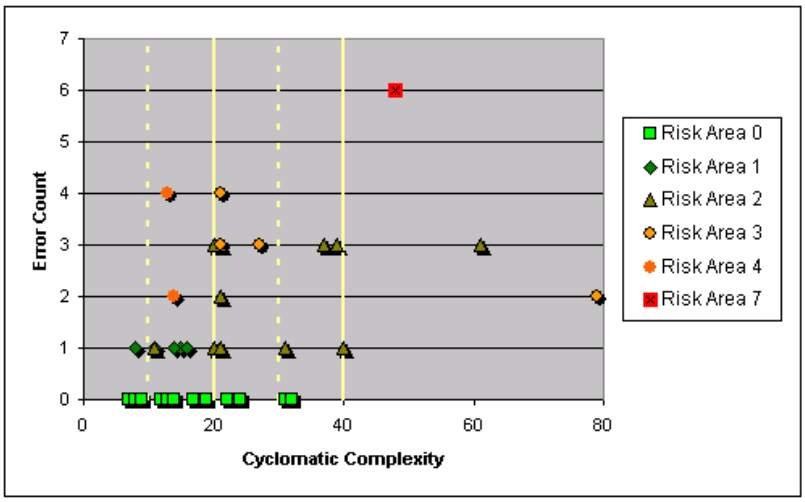

Figure 2: Error distribution over medium method complexity per C++ class

The results we present here deal with the relation between complexity and error count, cf. [IE89].We analysed the medium cyclomatic complexity of the methods in each class in relation to the total error count in the class. The results show an approximately equal distribution in the lower complexity categories, i.e. we cannot sufficiently evaluate the programming difficulty of error-proneness by the complexity alone. Based on our results, one might assume that there is a positive correlation between complexity and error probability as there were no flaw-free classes with a complexity above 35, and there were hardly errors detected in classes with a complexity lower than 10, but this is too imprecise for a sound evaluation.

It is also remarkable that the two classes in category 4 had a "safe" complexity value of less then 20; this is one more reason to evaluate not only on the basis this kind of analysis.

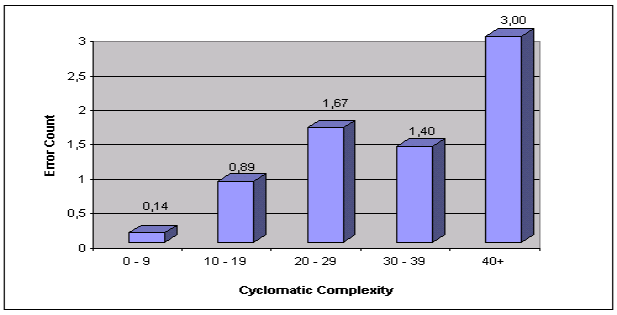

Figure 3: Error count over complexity categories

The mean error count for the complexity categories showed the expected trend of the error probability raising in a positive correlation with the complexity (cf. fig. 3). All in all we might state that there was a weak correlation between errors and complexity with some singularities.

4 Conclusion

The results of the unit test described above have shown that software, even if it comes from a professional organisation, deserves a thorough verification. As a consequence of our analysis work on error types we recommend detailed design and code reviews to avoid the majority of errors, the documentation errors.

Furthermore we have seen that there is a relation between software metrics and the errors encountered during the test. But as we learned, a low complexity alone is no guarantee for a low error rate, so we recommend systematic and careful unit tests in any case.

References

[FE00] N. E. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System", IEEE Transactions on Software Engineering, Vol.26, Nr.8, 2000

[IC01] IEC 61508, Functional Safety of Electric/Electronic/Programmable Electronic Systems, Part 1 – Part 7 (1998 – 2001)

[IE89] IEEE 982.2-1988 “IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software”, IEEE Press, June 1989

[MC82] Thomas J. McCabe, “Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric”, National Bureau of Standards, Washington 1982

[RG03] St. Ramberger, Th. Gruber, “Software-Verifikation und -validation - Risikoanalyse auf Basis von Unit-Tests im Prüflabor’’, ÖVE-Schriftenreihe Nr.33, Beiträge der Informationstagung Mikroelektronik 2003, Wien, ISBN 3-85133-030-7

[Ro98] Dr. Linda H. Rosenberg, Applying and Interpreting Object Oriented Metrics, Software Technology Conference, April 1998 (http://satc.gsfc.nasa.gov/metrics as of Feb.2003)

[SA03] Software Assurance Technology Center, http://satc.gsfc.nasa.gov as of Feb.2003