City Research Online

City, University of London Institutional Repository

Citation: van der Meulen, M. (2008). The Effectiveness of Software Diversity. (Unpublished Doctoral thesis, City University London)

This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: http://openaccess.city.ac.uk/19624/

Link to published version:

Copyright and reuse: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

The Effectiveness of Software Diversity

Meine Jochum Peter van der Meulen

Centre for Software Reliability
City University
London EC1V 0HB
United Kingdom

December 2008

PhD Thesis

BRITISH LIBRARY

IMAGING SERVICES NORTH

Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

THE FOLLOWING HAVE BEEN REDACTED AT THE REQUEST OF THE UNIVERSITY

EPIGRAPH ON PAGE 71

PUBLISHED ARTICLES ON PAGES 111-125, 127-139, 141-163, 165-175,177-190, 192-205,207-212 AND 214-224

© Meine van der Meulen, 2008

I grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council via the Interdisciplinary Research Collaboration on the Dependability of Computer Based Systems (DIRC), and via the Diversity with Off-The-Shelf Components (DOTS) project, GR/N23912.

Contents

List of Tables 8

List of Figures 11

Acknowledgements 13

Abstract 15

1 Introduction 17

1.1 Background 17

1.2 Terminology 19

1.3 The Quest 20

1.4 The Online Judge 21

1.5 Research Questions 24

2 Research in Software Diversity 27

2.1 Introduction 27

2.2 Diversity Experiments 28

2.2.1 Overview 28

2.2.2 Knight and Leveson 30

2.3 Commercial Software 31

2.3.1 Seismic Data Processing 31

2.3.2 SQL Servers 32

2.3.3 Smart Sensors 32

2.4 Fault Injection 33

2.5 Recovery Blocks 35
2.6 Modelling of Software Diversity 35
2.6.1 Eckhardt and Lee 35
2.6.2 Littlewood and Miller 37
2.6.3 Strigini and Popov 38

3 Experiments 41
3.1 Selection of Problems 41
3.2 Selection of Programs 42
3.2.1 For all Analyses 42
3.2.2 For the Analysis of Run-Time Checks 43
3.2.3 For the Analysis of Software Metrics 43
3.3 Test Harness 44
3.4 Analysis 44
3.4.1 Main Parameters 45
3.4.2 Parameters of C and C++ Programs 46
3.4.3 Graphs for Software Diversity 48
3.4.4 Outliers 49
3.5 Problems Encountered 50

4 Results 55
4.1 Introduction 55
4.2 General Observations 57
4.3 Debugging Process 58
4.4 Multiple-Version Software Diversity 59
4.5 Language Diversity 60
4.6 Run-Time Checks 60
4.7 Software Metrics 61
4.8 Validity of the Results 61

5 The Approach and its Validity 63
5.1 Introduction 63
5.2 Advantages 64

5.2.1 Access to a Diverse Participant Population 64
5.2.2 Bringing the Experiment to the Participant 65
5.2.3 Motivational Confounding 65
5.2.4 High Statistical Power 66
5.2.5 Cost Savings 66
5.2.6 Validation across Problem Domains 66
5.3 Disadvantages 67
5.3.1 Multiple Submissions 67
5.3.2 Lack of Experimental Control 68
5.3.3 Self-Selection 68
5.3.4 Dropout 69
5.3.5 Small Size of the Programs 69
5.3.6 Abnormal Debuging Process 69

6 Further Research **71**
6.1 Correlation with Country 71
6.2 Psychology 72
6.3 Correlation Between Faults 73
6.4 Priors for Bayesian Analysis 74
6.5 Bigger Programs 74
6.6 Other m-out-of-n Configurations 75

7 Conclusion **77**

References **81**

Names Index **89**

A Problems Analysed **93**
A.1 Properties of the Specifications 93
A.2 Specifications used in the Publications 96
A.3 Problem Domains 98

B Problem Analysis **99**

B.1 Directory Structure 99
B.2 Extracting E-mails 102
B.3 Extracting Source Code 102
B.4 Compiling Source Code 103
B.5 Running the Programs 103
B.6 Generating Standard Graphs 105
B.7 Example Session 106

C Publications **109**

C.1 TÜV Symposium 2004 110
C.2 Safecomp 2004 126
C.3 ISSRE 2004 140
C.4 ICCBSS 2005 153
C.5 EDCC 2005 164
C.6 Safecomp 2005 176
C.7 Safecomp 2006 191
C.8 ISSRE 2007 206
C.9 Transactions on Software Engineering 213

List of Tables

1.1 Distribution of the authors over continents and countries. 22

2.1 Some software diversity experiments. 29

3.1 Main parameters calculated for the "3n+1"-problem. 45

3.2 Parameters of the C and C++ programs of the "3n+1"-problem. 46

3.3 Correlations between the parameters of the C and C++ programs of the "3n+1"-problem. 47

6.1 For two faults in the "3n+1"-problem and their common occurrence: the equivalence classes in which they occur, the number of programs which contain them and their probability. 73

B.1 Directory structure for conducting the experiments. 100

B.2 Fields in the file History.txt. 101

C.1.1 FMEA of power supply. 122

C.1.2 FMEA of electrical output subsystem. 122

C.1.3 FMEA of sensor subsystem. 123

C.1.4 FMEA of communications interface. 123

C.1.5 FMEA of data processing subsystem. 124

C.1.6 FMEA of human interface. 124

C.1.7 FMEA of clock signal. 124

C.1.8 Rating of failure modes with respect to common cause failure. . 125

C.2.1 Population v1 equivalence classes. 131

C.2.2 Population vFinal: equivalence classes. 132
C.2.3 Mean difficulty values for different program populations. 135
C.2.4 Comparison of expected probability of failure on demand. 136

C.3.1 Example program with typical algorithm. 143
C.3.2 Equivalence classes and faults. 145
C.3.3 The probability of staying in the same equivalence class and other parameters. 147
C.3.5 Expected pfd's, from [2]. 149
C.3.6 Transitions from equivalence classes EC01 to EC36 to equivalence classes EC00 to EC36. 152

C.5.1 Some statistics on the three problems. 168

C.6.1 Some statistics on the three problems. 179
C.6.2 Classification of execution results with plausibility checks. 180
C.6.3 Classification of execution results with self-consistency checks. . 181

C.7.1 Sample inputs and sample outputs for the primary. 194
C.7.2 Sample inputs and sample outputs for the checker. 195
C.7.3 Classification of execution results with RTCs. 196
C.7.4 Most frequent equivalence classes in the first submissions for the primary. 199
C.7.5 Most frequent equivalence classes in the first submissions for the checker. 200

C.9.1 Statistics on submissions to the "3n+1"-problem. 217
C.9.2 Example program with typical algorithm. 217
C.9.3 Additional improvements for the language pairs. 222
C.9.4 Equivalence classes and faults for the first submission of the "3n+1"-problem. 224

List of Figures

1.1 Authors' Birth years. 23

2.1 Primary/Checker model. 38
2.2 An example of a wrapper in a boiler control system. 39

3.1 Example of the graphs of the parameters of the C and C++ programs. 47
3.2 Example of the graphs of the correlation between parameters of the C and C++ programs. 48
3.3 Box and whisker plot of the main parameters of the correct programs of the "Archeologist"-problem. 49
3.4 Reliability improvement of a diverse pair, relative to a single version for the "3n+1"-problem. 50
3.5 Reliability improvement for 1-out-of-2 pairs with diverse language diversity. 51

4.1 Histogram of the ratios between the number of equivalence classes and the number of incorrect submissions for the first attempt. . . 57
4.2 Transitions between equivalence classes with a given number of faults for the "3n+1"-problem. 59

C.1.1 Generic model of the design of a smart sensor. 115

C.2.1 Failure regions for some of the infrequent equivalence classes. . . 132
C.2.2 Difficulty function for the v1 population. 133
C.2.3 Adjusted difficulty function for the v1 population. 134

C.2.4 Adjusted difficulty function for the vFinal distribution. 135

C.3.1 Number of submissions until last or correct solution per author. 142

C.3.2 Failure sets for the equivalence classes. 146

C.3.3 Reliability profile (successive releases). 147

C.3.4 Difficulty function for the final version of the "3n+1"-problem. . 148

C.3.5 Difficulty function for the final version of the alternate problem. 149

C.4.1 The boiler control system used as an example. 156

C.5.1 Reliability improvement over a single version for the "3n+1"-problem. 173

C.5.2 Reliability improvement over a single version for the "Factovisors"-problem. 174

C.5.3 Reliability improvement over a single version for the "Prime Time"-problem. 175

C.6.1 The improvement of the pufd for the "3n+1"-problem. 182

C.6.2 Values of the error detection coverage. 183

C.6.3 Improvement in the average pufd of the primary for the various self-consistency checks for "3n+1". 184

C.6.4 Improvement in the average pufd of the primary for combinations of run-time checks for "3n+1". 185

C.6.5 The effectiveness of the run-time checks for "Factovisors". 186

C.6.6 The effectiveness of the run-time checks for "Prime Time". . . . 187

C.6.7 Improvement of the pufd of a pair of randomly chosen C programs for "3n+1", over a single version. 188

C.7.1 Primary/Checker model. 193

C.7.2 Histograms of the pfd's of the primaries and the pufd with RTC8.201

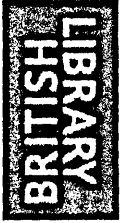
C.7.3 The improvement of the pufd for various run-time checks. 202

C.7.4 The effectiveness of the eight run-time checks with decreasing average pfd of the pool of primaries. 203

C.7.5 Improvement of the pufd of a pair of randomly chosen primaries, relative to a single version. 204

C.8.1 Histograms of internal software metrics of the correct submissions for the "Unidirectional TSP"-problem. 208
C.8.2 Histograms of the correlations between the various internal software metrics. 209
C.8.3 Density plots of the LOC/Volume and LOC/CC distribution of all correct programs of all specifications combined. 210
C.8.4 Histograms of the ratio of internal metrics of correct programs and incorrect programs. 211
C.8.5 Histograms of the correlations between various internal software metrics and PFD or Number of Defects. 212

C.9.1 Failure sets for the equivalence classes. 218
C.9.2 Reliability improvement of a diverse pair, relative to a single version. 218
C.9.3 Relative frequency of occurrence rates of equivalence classes for loops. 219
C.9.4 Difficulty functions for three pools with different pfd. 220
C.9.5 Reliability improvement for 1-out-of-2 pairs with diverse language diversity. 220
C.9.6 Histogram of the number of submissions used for the specifications. 221
C.9.7 The reliability improvement of multiple-version diversity. 221
C.9.8 Histograms of the PFD improvement of implementing 1-out-of-2 diversity for the sixty specifications. 222



IMAGING SERVICES NORTH
Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

MISSING PAGE/PAGES
HAVE NO CONTENT

Acknowledgements

One of the deepest wishes in my life has been to get a PhD. Until now, I have not succeeded, in spite of the many opportunities that presented themselves.

After my MSc thesis at Philips Research, I turned down the offer to start a PhD project there; at the time I thought it was wiser to get work experience; a decision I still think was right. Later, I got another opportunity at the Dutch Organisation for Applied Scientific Research. This attempt soon failed, because the idea I had was not really good enough. Then, around 2001, I started writing a thesis based on an old idea of mine: the design of asynchronous fault-tolerant logic. In spite of this being a great idea, I aborted this attempt because I had accepted a job at City University in London in the beginning of 2003.

Now, finally, the moment has come close. This is entirely due to the opportunities City University gave me. Although my assignment was to be a Research Fellow and do various research assignments, mainly for Lorenzo Strigini, I was given many opportunities to find my own way and to investigate my own ideas. After three years, I assembled enough material to continue some of the research independently and write my thesis.

Therefore, my sincerest gratitude goes to City University. With the risk of omitting people, I wish to name Lorenzo Strigini and Bev Littlewood first, for their many positive remarks and challenging attitude; Robin Bloomfield for his support; Peter Bishop for the enormous knowledge he shared with me.

Miguel Revilla plays a central role, because he selflessly provided the programs submitted to the Online Judge. This is something I almost didn't dare ask, but Miguel immediately reacted with the utmost enthusiasm. And of course, my gratitude goes to the thousands of programmers who unknowingly

contributed to my work.

I also wish to thank Andrey Povyakalo for introducing me to the statistical package R, and helping me to use it; Kizito Salako for invaluable discussions on the Eckhardt & Lee model and its extensions; Peter Popov for his insights on almost every part of the work; Kostas Tourlas for the help in writing bash scripts and C programs; Ton van Boven for maintaining my Linux server which safely ran experiments in the silence of the basement of his house.

Three month of the research were done at the Budapest University of Technology and Economics, Department of Measurement and Information Systems, in which time András Pataricza provided me with all the necessary resources.

Finally, I thank Viktória Kaposi, my wife, who supported me at all times, and Andreas Jan, my son, who made me realize that it is time to finish the work.

Abstract

This research exploits a collection of more than 2,500,000 programs, written to over 1,500 specifications (problems), the Online Judge programming competition. This website invites people all over the world to submit programs to these specifications, and automatically checks them against a benchmark. The submitter receives feedback, the most frequent responses being "Correct" and "Wrong Answer".

This enormous collection of programs gives the opportunity to test common assumptions in software reliability engineering about software diversity, with a high confidence level and across several problem domains. The previous research has used collections of up to several dozens of programs for only one problem, which is large enough for exploratory insights, but not for statistical relevance.

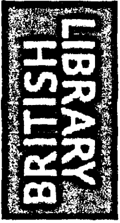
For this research, a test harness was developed which automatically extracts, compiles, runs and checks the programs with a benchmark. For most problems, this benchmark consists of 2,500 or 10,000 demands. The demands are random, or cover the entirety or a contiguous section of the demand space.

For every program the test harness calculates: (1) The output for every demand in the benchmark. (2) The failure set, i.e. the demands for which the program fails. (3) The internal software metrics: Lines of Code, Halstead Volume, and McCabe's Cyclomatic Complexity. (4) The dependability metrics: number of faults and probability of failure on demand.

The only manual intervention in the above is the selection of a correct program. The test harness calculates the following for every problem: (1) Various characteristics: the number of programs submitted to the problem, the number of authors submitting, the number of correct programs in the first attempt, the average number of attempts until a correct solution is reached, etc. (2) The grouping of programs in equivalence classes, i.e. groups of programs with the same failure set. (3) The difficulty function for the problem, i.e. the average probability of failure of the program for different demands. (4) The gain of multiple-version software diversity in a 1-out-of-2 pair as a function of the pfd of the set of programs. (5) The additional gain of language diversity in the pair. (6) Correlations between internal metrics, and between internal metrics and dependability metrics.

This research confirms many of the insights gained from earlier studies: (1) Software diversity is an effective means to increase the probability of failure on demand of a 1-out-of-2 system. It decreases the probability of undetected failure on demand by on average around two orders of magnitude for reliable programs. (2) For unreliable programs the failure behaviour appears to be close to statistically independent. (3) Run-time checks reduce the probability of failure. However, the gain of run-time checks is much lower and far less consistent than that of multiple-version software diversity, i.e. it is fairly random whether a run-time check matches the faults actually made in programs. (4) Language diversity in a diverse pair provides an additional gain in the probability of undetected failure on demand, but this gain is not very high when choosing from the programming languages C, C++ and Pascal. (5) There is a very strong correlation between the internal software metrics.

The programs in the Online Judge do not behave according to some common assumptions: (1) For a given specification, there is no correlation between selected internal software metrics (Lines of Code, Halstead Volume and McCabe's Cyclomatic Complexity) and dependability metrics (pfd and Number of Faults) of the programs implementing it. This result could imply that for improving dependability, it is not effective to require programmers to write programs within given bounds for these internal software metrics. (2) Run-time checks are still effective for reliable programs. Often, programmers remove run-time checks at the end of the development process, probably because the program is then deemed to be correct. However, the benefit of run-time checks does not correlate with pfd, i.e. they are as effective for reliable as for unreliable programs. They are also shown to have a negligible adverse impact on the program's dependability.



IMAGING SERVICES NORTH
Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

MISSING PAGE/PAGES
HAVE NO CONTENT

Chapter 1

Introduction

There never were, in the world, two opinions alike, no more than two hairs, or two grains; the most universal quality is diversity.

Michel Eyquem de Montaigne (1533–1592)

1.1 Background

In this age of automation, software dependability is a major concern. In 1994, Peter Neumann [53], Donald McKenzie [44] and Les Hatton [27] published lists of software-related incidents. The lists show that the cost of software failure can be very high, both in terms of risks to human life and well-being as to economy. Up to the present day, Peter Neumann continues publishing incidents in his RISKS forum on the internet, and there is no reason to believe the problem has disappeared.

There are therefore good reasons to increase the dependability of software, and there are many ways to do so: testing, quality control, design, etc. One of the possible measures is fault tolerance, of which software diversity is an

example.

Lee and Anderson [3] were very early in publishing their work on fault-tolerance, "Fault Tolerance; Principles and Practice". In one of the chapters they propose a list of practices for software fault tolerance, which also addresses some words to software diversity. Voges [65] edited the book "Software Diversity in Computerized Control Systems" in 1988, showing the interest in that issue at that time.

Also in the 1980s, various researchers addressed the issue of software diversity in experiments, see Section 2.2. They were able to prove beyond doubt that it was an effective measure to increase software dependability. Also, mathematical theories behind software diversity were developed, see Section 2.6.

The growing awareness also found its way into standardization, especially concerning safety. Germany was early; employees at the TÜV, Holscher and Rader, wrote "Microcomputer in der Sicherheitstechnik" [30] in 1984. Their ideas were later implemented in DIN-19250 [13] and VDE-0801 [14]. Then many countries and organisations (for automotive, defence, space, rail, aviation, healthcare, etc. applications) derived their own guidelines and standards.

The International Electrotechnical Committee, IEC, took up the task and wrote what is now called IEC61508 [32]. IEC61508 mentions diverse programming in Table A.2, item 3c. It is Recommended (R) for SIL1 to SIL3 and Highly Recommended (HR) for SIL4. IEC61508 describes diverse programming as follows:

Aim: Detect and mask residual software design and implementation faults during execution of a program, in order to prevent safety critical failures of the system, and to continue operation for high reliability.

Description: In diverse programming a given program specification is designed and implemented N times in different ways. The same input values are given to the N versions, and the results produced by the N versions are compared. If the result is considered to be valid, the result is transmitted to the computer outputs. The N versions can run in parallel on separate computers, alternatively all versions can be run on the same computer and the results subjected

to an internal vote. Different voting strategies can be used on the N versions, depending on the application requirements, as follows.

- *If the system has a safe state, then it is feasible to demand complete agreement (all N agree) otherwise an output value is used that will cause the system to reach the safe state. For simple trip systems the vote can be biased in the safe direction. In this case the safe action would be to trip if either version demanded a trip. This approach typically uses only two versions (N=2).*
- *For systems with no safe state, majority voting strategies can be employed. For cases where there is no collective agreement, probabilistic approaches can be used in order to maximise the chance of selecting the correct value, for example, taking the middle value, temporary freezing of outputs until agreement returns, etc. This technique does not eliminate residual software design faults, nor does it avoid errors in the interpretation of the specification, but it provides a measure to detect and mask before they can affect safety.*

This describes software diversity very well. In this thesis, I will only address the case where N=2, often referred to as a 1-out-of-2 system.

1.2 Terminology

The previous section already shows that software diversity is known under different names. Roughly in order of the year of first appearance, authors and standards refer to it as "dual programming" [22], "parallel programming" [22], "dual code" [22, 63], "distinct software" [23], "N-version programming" [4], "dissimilar software" [42], "multi-version software" [17, 33, 35, 37], "software diversity" [24], "multi-version programming" [36], and "diverse programming" [32].

I will use the term "multiple-version software diversity" and its shorthand variant "software diversity". There is no specific reason to prefer this term over

the others, except that it seems to fit best to earlier work done at the Centre for Software Reliability.

1.3 The Quest

When I arrived at the Centre for Software Reliability, in February 2003, I joined a group with a respectable background in the research of software diversity. I was presented the work on modelling software diversity (the Eckhardt & Lee model, and its extension the Littlewood & Miller model) and the research project DOTS, "Diversity with Off-The Shelf Components".

The approach to modelling software diversity at CSR is primarily theoretical. With their mathematical background, Kizito Salako and Bev Littlewood were working on theoretical aspects of the modelling (e.g. [59]). It was clear that—with my background—my focus would be more pragmatic and, within the framework of DOTS, Lorenzo Strigini motivated me to start thinking and writing about smart sensors and the use of software diversity [46]. This made me very aware of the difficulty of doing this kind of research. Although interesting observations can be made, it is hard to give hard recommendations, because their effectiveness can not be measured.

One of my research interests has always been the verification of "facts" vs. "urban legends". In the field of software diversity, there are many of these. The fact that software diversity helps improve reliability is fairly well researched, how much improvement is far more an educated guess, since large collections of data have not been used for testing hypotheses of this kind. Other assumptions, for example about the effectiveness of language diversity in a 1-out-of-2 pair, have never been measured.

Another observation I made is that the Eckhardt & Lee model is elegant in its simplicity, and is useful for many a philosophical discourse, but has never really been applied. What does a difficulty function look like? What happens if you multiply real difficulty functions? In other words, in my pragmatic mind, it needed some legs to stand on to get some grasp of its usefulness.

With this in mind, I realized that to accomplish these two aims—measure

effectiveness, and apply the Eckhardt & Lee model—I needed a large set of programs written to the same specification. I started my quest, and quickly find out that indeed some interesting examples exist of these.

One example is chess programs. They are written to a very precise specification, and dozens of chess programs can easily be found. However, these programs struck me as too complicated for my analyses.

Then a simpler game "Rock, Paper, Scissors", is the subject of "The International RoShamBo Programming Competition". Not only are there many programs written for this specification, they are also available: all the best programs of the 1999 competition can be downloaded at `https://www.cs.ualberta.ca/~darse/rsbpc.html`. This game is a rather interesting programming challenge, because at first sight it seems that there is only one strategy: play random. However, as can be easily seen, a random program will also randomly win and lose, and therefore will never win a competition. In spite of the availability of several dozens of programs and the simplicity of the specification, I decided not to take this route, because the difference between these programs is strategy. This will be very hard to analyse. Apart from that, it does not really fit the concept of software diversity, since that normally addresses programs that aim at producing the same output.

Of course there are also many commercial programs with comparable specifications, for example spreadsheets, word processing, and smart sensors. This is certainly a possible way forward, but I decided not to pursue it, because it has been tried by others before, see Section 2.3.

Many attempts later, I arrived at the website of the Online Judge. It presented me a very large set of specifications, and also presented some statistics. It showed that very many programs were written to each specification. I realized that this could give me the opportunity I had been looking for.

1.4 The Online Judge

The "UVa Online Judge"-Website, `http://acm.uva.es`, is an initiative of Miguel Revilla of the University of Valladolid [61]. It contains program specifi-

Continent	%	Country	%
Africa	0.7		
Asia	51.4	Bangladesh	13.1
		China	11.8
		Hong Kong	2.4
		India	2.1
		Indonesia	2.4
		Japan	1.3
		Korea	1.9
		Taiwan	13.8
Oceania	1.4	Australia	1.2
Europe	25.5	Bulgaria	1.0
		France	1.4
		Germany	1.4
		Poland	3.0
		Portugal	1.4
		Russia	5.0
		Spain	1.9
		Ukraine	1.2
North America	12.2	Canada	2.5
		USA	9.7
South America	8.7	Brazil	4.7
		Mexico	1.2

Table 1.1: Distribution of the authors over the continents, and the countries with more than 1% of the authors (as of May 2004).

cations for which anyone may submit programs in C, C++, Java or Pascal. The correctness of a program is automatically judged by the "Online Judge". Most authors submit programs repeatedly until one is judged correct. Many thousands of authors contribute and together they have produced more than

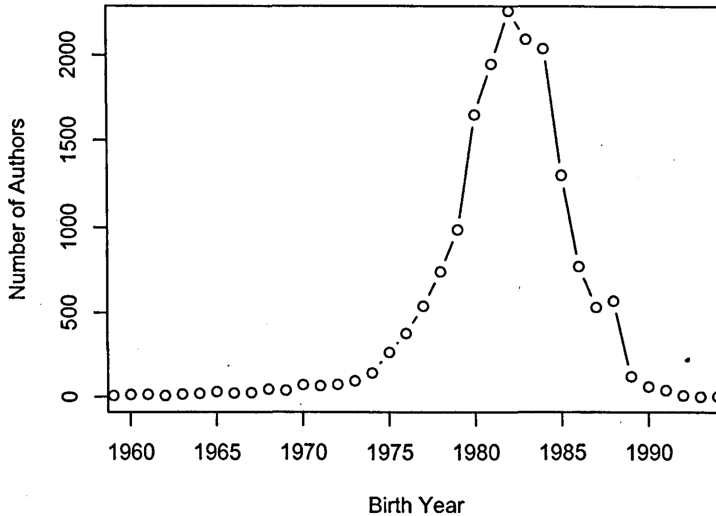


Figure 1.1: Authors' birth years, for those authors that provided their birth year (as of May 2004).

5,000,000 programs for the approximately 1,500 specifications on the website.

Most of my research has been done using the data available in May 2004: 2,545,979 submissions for around 1,500 problems.

Table 1.1 shows the distribution of the authors over the continents and countries. It appears that they are located all over the world, with some countries being surprisingly productive and Africa almost absent. Figure 1.1 shows the birth year of the authors. It appears most of the authors are between 18 and 25 years old, i.e. they are most probably university students. The average birth date is in July 1981, i.e. the average age is around 22 at the time of submission.

The Online Judge generates feedback to the author of a program, which is one of the following:

Correct (AC). The program's output matches the Online Judge's output.

Wrong Answer (WA). The output of the program does not match what the Online Judge expects.

Presentation Error (PE). Presentation errors occur when the program

produces correct output for the Online Judge's secret data but does not produce it in the correct format.

Runtime Error (RE). This error indicates that the program performs an illegal operation when running on the Online Judge's input. Some illegal operations include invalid memory references such as accessing outside an array boundary. There are also a number of common mathematical errors such as divide by zero error, overflow or domain error.

Time Limit Exceeded (TL). The Online Judge has a specified time limit for every problem. When the program does not terminate in that specified time limit this error will be generated.

Compile Error (CE). The program does not compile with the specified language's compiler.

Memory Limit Exceeded (ML). The program requires more memory to run than the Online Judge allows.

Output Limit Exceeded (OL). The program produces more than 4 MB output within the time limit.

Restricted Function (RF). The program uses some system function call or tries to access files.

In my experiments I only use programs that are either classified as accepted (AC), wrong answer (WA) or presentation error (PE), see Section 3.1.

1.5 Research Questions

The quantification and analysis of the various aspects of software diversity requires the availability of large amounts of programs written to the same specification. This is exactly what the Online Judge can provide. Therefore, the use of the programs provided by the Online Judge gives a unique opportunity to research software diversity.

The main questions I try to answer in this research project are:

- How effective is multiple-version software diversity as a means to enhance software reliability?

- How effective is language diversity as a means to increase the effectiveness of multiple-version software diversity?
- How effective are run-time checks? (Run-time checks constitute a special variety of software diversity.)

In this research I actually try to measure effectiveness of software diversity. Additionally, I encountered other interesting and related results, which I will also report.

BRITISH LIBRARY

IMAGING SERVICES NORTH

Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

MISSING PAGE/PAGES
HAVE NO CONTENT

Chapter 2

Research in Software Diversity

The greatest of faults, I should say, is to be conscious of none.

Thomas Carlyle (1795–1881)

This chapter presents the existing theory in software diversity in order to provide the necessary background for the experiments. It also describes some relevant existing case studies, which will later be used to put the results of the experiments into perspective.

2.1 Introduction

As explained in the introduction, this research addresses the effectiveness of software diversity using a large set of programs submitted to an on-line programming competition. Although this has not been done before there is other research addressing similar issues. In this chapter I will sketch out this related research and my work within this context.

Starting in the 1980s, there have been quite a few experiments in which (teams of) students wrote programs to the same specification. I will address

some of these in Section 2.2.

A possible way to circumvent much of the criticism on these experiments (and my work) is to analyse various commercial software packages implementing the same application. I will address this approach in Section 2.3.

In all of the above research, the basic idea is that different versions of the software are generated because different programmers/design teams make different mistakes. Another approach to get many different versions of programs is fault injection. I will address this in Section 2.4.

A related strategy for software fault tolerance is recovery blocks, which I will address in Section 2.5.

Furthermore, I need to address the issue of how to model software diversity, which is necessary to be able to do the calculations. I will address the mathematical models for software diversity in Section 2.6.

2.2 Diversity Experiments

2.2.1 Overview

Peter Bishop compiled Table 2.1 [8], which shows some typical software diversity experiments. Most famous is the work of John Knight and Nancy Leveson, performed in the 1980s, which includes one of experiments addressed (the Launch Interceptor). I will describe this work separately in the next section.

Some of Peter Bishop's observations relevant to this thesis are:

1. A significant proportion of the faults is similar, and the major cause of this is (the interpretation of) the specification [12, 24, 34]. The use of relatively formal notations is effective in reducing specification-related faults caused by incompleteness and ambiguity.

2. Some experiments address the impact of the choice of programming language [5, 9, 12], and fewer faults seem to occur in the strongly typed, highly structured languages such as Modula-2 and Ada. However, the choice of language seems to bear little relationship to the incidence of common specification-related or design-related faults.

Experiment	Application	Specs	Languages	Versions	Ref.
Halden	Reactor Trip	1	2	2	[12]
NASA	First Generation	3	1	18	[34]
KFK	Reactor Trip	1	3	3	[24]
NASA/RTI	Launch Interceptor	1	3	3	[15]
UCI/UVA	Launch Interceptor	1	1	27	[35]
Halden (PODS)	Reactor Trip	2	2	3	[9]
UCLA	Flight Control	1	6	6	[5]
NASA	Inertial Guidance	1	1	20	[16]
UI/Rockwell	Flight Control	1	1	15	[41]

Table 2.1: Some software diversity experiments.

My research differs from these experiments in various ways:

1. The number of programs written to the same specification. The highest number of versions in these experiments is 27. There are other experiments with slightly higher numbers, e.g. by Per Runeson [58], but there are no experiments with thousands of versions.

2. The way the experiment is controlled. The amount of control in these experiments is typically very high. This cannot be said of the contributions to the Online Judge. Little is known about the programmers, except their nationality and age.

3. The programming languages used, and whether different programming languages are used for the different versions. Many experiments do address the issue of language diversity, some even forced it [40]. Some general observations have been done but, given the low number of versions, no statistically relevant results are available.

4. The number of specifications used. In some experiments more than one specification is used. This is done because a major source of faults appears to be the specification. The Online Judge only provides one specification.

(There are some different specifications of the same problem of the Online Judge, I did however not analyse this.)

5. The size of the programs. The size of the programs written for the experiments are typically up to 1000 lines of code. This is larger than the programs written for the Online Judge. In both cases however, the difference with "real" software is big.

6. Realism of the experiment. In many of the experiments there was an effort to provide a specification of a realistic problem. However, the specifications remained fictitious, and as such I don't see much difference with the specifications of the Online Judge.

For a more elaborate discussions of the limitations of this research see Chapter 5.

2.2.2 Knight and Leveson

In 1986, John Knight and Nancy Leveson reported an experiment in which 27 programmers, from two universities, programmed to the same specification, describing a (fictitious) defence application launching a missile based on input radar data [36]. The 27 programmers wrote programs in Pascal that varied largely in size: from 327 to 1,004 lines of code. The programs were all submitted to a random test with 1,000,000 demands, their probability of failure on demand was in all cases lower than 0.01.

A major result was that the authors could reject the assumption of independence of failure of the diverse versions to a high confidence level [8].

In a subsequent publication the authors investigated the effectiveness of software diversity [35]. They were able to show that a three-version system, with majority voting, yielded a reliability gain of a factor 20 to 50 over a single version. Other researchers investigated the same dataset, e.g. Susan Brilliant [10] and Les Hatton [28].

2.3 Commercial Software

The ideal software diversity experiment is of course one in which "real" software is used, i.e. different commercial implementations for the same application. In theory this would be possible in many application domains, but it is in practice difficult and very time consuming to do. This is because different commercial packages are never exactly the same. Their inputs and output formats as well as the results they provide may be different, and it is therefore hard to exercise them in ways that are comparable. Additionally, even if a test harness can be made, it may be hard to find faults, because each test may take some time, and it may be hard to run many tests.

In this section I wish to present the results of Les Hatton, who compared various seismic data processing packages, Ilir Gashi, who compared various SQL servers, and myself on smart sensors.

The reader will notice that, although all these cases clearly give very useful insights into the possible effectiveness of diversity, it is hard to see how they can be used to quantitatively assess the possible reliability gain of using software diversity.

2.3.1 Seismic Data Processing

In 1994, Les Hatton and Andy Roberts reported their findings concerning the accuracy of scientific software [29]. It concerns a four year study of nine commercial seismic data processing packages written independently in the same programming language and calibrated with the same input data and input parameters. They show that, after a normal processing sequence, the nine packages may differ in the first and second decimal place, which essentially makes these results useless. They also show that these differences are entirely due to software errors.

The experiment is carried out in a branch of the earth sciences known as seismic data processing. Hatton and Roberts identified around fifteen independently developed large commercial packages that implement mathematical algorithms from the same or similar published specifications in the same pro-

gramming language (Fortran) in their experiments. They reported the results of processing the same input dataset, using the same user-specified parameters, for nine of these packages.

Hatton and Roberts show that the differences between the results of the various packages are big, far bigger than the arithmetical precision, and that the nature of the disagreement is non-random.

2.3.2 SQL Servers

Ilir Gashi [18, 19, 20] assessed the dependability gains that may be achieved through software fault tolerance via modular redundancy with diversity in complex off-the-shelf software. He used SQL database server products in his studies: they are a very complex, widely-used category of off-the-shelf products.

Gashi used bug reports of the servers as evidence in the assessment: they were the only direct dependability evidence that was found for these products. He studied a sample of bug reports from four SQL database server products (Postgre SQL, Interbase, Microsoft SQL Server, Oracle) and later releases of two of them to check whether they would cause coincident failures in more than one of the products. He found very few bugs that affect more than one product, and none caused failures in more than two. Many of these faults caused systematic, non-crash failures, a category ignored by most studies and standard implementations of fault tolerance for databases.

For one of the products, he found that the use of different releases of the same product appears to tolerate a significant fraction of the faults.

2.3.3 Smart Sensors

In sensors, a silent revolution has taken place: almost all sensors now contain software. This has lead to increased functionality, increased precision, and other advantages. There are also disadantages. The "smart" sensors have many more failure modes and they also introduce new mechanisms for common mode failure. I investigated the issue in [45] (Appendix C.1).

In this paper, I compare the dependability aspects of deploying smart sen-

sors vs. conventional ones using an Failure Modes and Effects Analysis. There appear to be some significant differences. Some failure modes do not exist in conventional sensors, e.g. those involving information overload and timing aspects. Other failure modes emerge through the use of different technologies, e.g. those involving complexity, data integrity and human interface. When using smart sensors I suggested the use of a set of guidelines for their deployment:

1. Do not send data to the smart sensor.
2. Use the smart sensor in burst mode only.
3. Use a smart sensor with the least possible number of operational modes.
4. Use the simplest possible sensor for the application.

In redundant sensor configurations (including those without diversity) common cause failure becomes the dominant failure scenario. The failure modes of smart sensors suggest that smart sensors are more susceptible to common cause failure than conventional ones. Dominant are failures that have their origin in the human interface, complexity and information overload. The guidelines given will also reduce the probability of common cause failure.

In redundant sensor configurations a possible design method is the use of diversity. Diversity has the advantage that it can reduce the probability that two or more sensors fail simultaneously, although this effect is limited by the fact that diverse sensors may still contain the same faults. A disadvantage of diversity can be the increased complexity of maintenance, which in itself can lead to a higher probability of failure of the smart sensors. Whether the use of diversity is advisable depends on the design of the smart sensors and the details of their application.

2.4 Fault Injection

It is possible to create many different programs, mutants, by changing a given program. These changes can either be random or based on a set of rules, which are generally based on observations of frequently made mistakes. This approach

of changing programs is used in "mutation testing" [1, 54, 55, 64]. Mutation operators exist for various programming languages, e.g. C [1] and Ada [55].

The Diverse Software Project (DISPO) of the Centre for Software Reliability (CSR, City University) and the Safety Systems Research Centre (SSRC, University of Bristol), included a set of fault injection experiments [62, 21, 39]. One of the objectives of these experiments was to study the relationship between fault diversity and failure diversity, i.e. is there a correlation between the type/location of faults and software failure?

The types of faults injected in these experiments are mainly slips/mistakes, and not the kind of faults that may be caused by requirements ambiguities or other higher-level misunderstandings. In [62], the author concedes that this does not pose a major concern for testing the hypothesis, because the faults are realistic; for testing the hypothesis it is not necessary that the faults reflect all possible faults nor their distribution. Examples of injected faults are replacing "<" with ">" and "==" with "=", and are as such comparable to those suggested in mutation testing.

The location of faults is varied by injecting faults in the different files that constitute the entire program. In the experiment, mutants with a pfd higher than 0.01 are discarded, because they are deemed to be unrealistic.

The experiments can not confirm a relationship between fault diversity and failure diversity.

The experiments also lead to some other results, one of which is relevant for this thesis. When assuming that the mutants form the complete population of possible programs, and that they are equiprobable (as the author points out, these are both unlikely assumptions), pairs of programs in a 1-out-of-2 configuration fail on average more often than failure independence would imply. The improvement of the pfd over a single version differs for various experiments, but is generally in the range of one to two orders of magnitude.

2.5 Recovery Blocks

The recovery block scheme [31] [56] is a strategy for software fault tolerance which also employs software diversity. A recovery block consists of a conventional block which is provided with a means of error detection (an acceptance test) and zero or more stand-by spares (the additional alternates). If none of the alternate programs passes the acceptance test, there will be no result.

In contrast to 1-out-of-2 systems, the recovery block scheme may also work when the required output is not unique.

In 1988, Anderson et. al. [2] published the results of an experiment with a realistic application, a command and control system constructed by professional programmers to normal commercial standards. Several fault tolerance features were added, including recovery blocks. In the experiment it was possible to monitor the improvement of the reliability from the use of fault tolerance. The experiment showed that 74% of the failures were masked by the use of software fault tolerance.

2.6 Modelling of Software Diversity

The mathematical theory on modelling software diversity is not very large. Only Eckhardt & Lee and Littlewood & Miller have made significant contributions to mathematically understanding the mechanisms of software failure in redundant configurations. Only recently, Lorenzo Strigini and Peter Popov added their work on asymmetric software diversity, i.e. run-time checks and wrappers.

2.6.1 Eckhardt and Lee

The best known probability model in the domain of multiple-version diversity is that of Eckhardt & Lee [17]. The assumptions underlying the model are that:

1. Failures of an individual program are deterministic and a program version either fails or succeeds for each input value x. The failure set of a program π can be represented by a "score function" $\omega(\pi, x)$ which produces a zero if the program succeeds for a given x or a one if it fails.

2. There is randomness due to the development process. This is represented as the random selection of a program, Π, from the set of all possible program versions that can feasibly be developed and/or envisaged. The probability that a particular version π will be produced is $P(\Pi = \pi)$.

3. There is randomness due to the demands in operation. This is represented by the random occurrence of a demand, X, from the set of all possible demands. The probability that a particular demand will occur is $P(X = x)$, the demand profile.

Using these model assumptions, the average probability of a program version failing on a given demand is given by the difficulty function, $\theta(x)$, where:

$$\theta(x) = \sum_{\pi} \omega(\pi, x) P(\Pi = \pi) \tag{2.1}$$

The average probability of failure on demand of a randomly chosen single program version Π_A can be computed using the difficulty function for method A and the demand profile:

$$\text{pfd}_A := P(\Pi_A \text{ fails on } X) = \sum_{x} \theta_A(x) P(X = x) \tag{2.2}$$

The average pfd for a pair of diverse programs, Π_A and Π_B, assuming the system fails when both versions fail, i.e. a 1-out-of-2 system. would be:

$$\text{pfd}_{AB} = \sum_{x} \theta_A(x) \theta_B(x) P(X = x) \tag{2.3}$$

The Eckhardt & Lee model assumes similar development processes for the two programs A and B and hence identical difficulty functions, i.e. that $\theta_A(x) = \theta_B(x)$. Therefore, the average pfd for a pair of diverse programs is:

$$\text{pfd}_{AB} = \text{pfd}_A^2 + var_X(\theta_A(X)) \tag{2.4}$$

If the difficulty function is constant for all x, and therefore $var_X(\theta_A(X), \theta_B(X)) = 0$, the reliability improvement for a diverse pair will (on average) satisfy the independence assumption:

$$\text{pfd}_{AB} = \text{pfd}_A . \text{pfd}_B = \text{pfd}_A^2 \tag{2.5}$$

It is always the case that $var_X(\theta_A(X)) \geq 0$, and therefore:

$$\text{pfd}_{AB} \geq \text{pfd}_A^2 \tag{2.6}$$

In practice the variance plays an important role, and is often the main factor of system unreliability. The intuitive explanation for this is that it is harder for the program developers to properly deal with some demands. The difficulty function will then be "spiky", and the diverse program versions tend to fail on the same demands. Diversity is then likely to yield little benefit and pfd_{AB} could be close to pfd_A.

I will use the Eckhardt & Lee model in most of our multiple-version diversity experiments, except those with different programming languages.

2.6.2 Littlewood and Miller

In 1989, Littlewood and Miller published a generalization of the Eckhardt & Lee model: the Littlewood & Miller extended model [37]. The difference between the models is that Littlewood and Miller do not assume that the development process of the two program versions is the same. Therefore, in their model the difficulty functions may differ.

The average pfd for a pair of diverse programs, Π_A and Π_B, developed using methods A and B (assuming the system fails when both versions fail, i.e. a 1-out-of-2 system) would be:

$$\text{pfd}_{AB} := P(\Pi_A \text{ fails on } X \text{ and } \Pi_B \text{ fails on } X) = \sum_x \theta_A(x)\theta_B(x)P(X = x) \tag{2.7}$$

And, as shown in [37]:

$$\text{pfd}_{AB} = \text{pfd}_A.\text{pfd}_B + cov_X(\theta_A(X), \theta_B(X)) \tag{2.8}$$

Because $cov_X(\theta_A(X), \theta_B(X))$ can be smaller than zero, the improvement can, in principle, be better than the independence assumption. The intuitive explanation of negative covariance is a circumstance where the "dips" in $\theta_A(x)$ coincide with the "spikes" in $\theta_B(x)$, so the programs on average fail on different demands, and the performance of the pair is better than under the independence assumption (and vice versa).

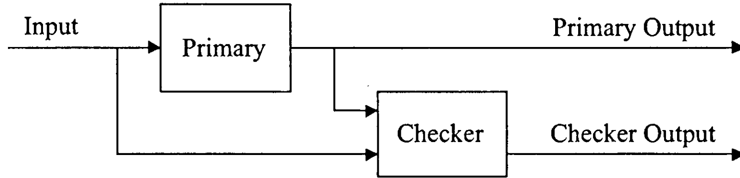


Figure 2.1: Primary/Checker model.

We will use the Littlewood & Miller model in our language diversity experiments with different programming languages.

2.6.3 Strigini and Popov

Both the Eckhardt & Lee and the Littlewood & Miller model consider symmetrical redundancy, i.e. both channels have the same functionality. In practice however, there are many examples of systems where the redundancy is not symmetrical. Examples are wrappers and run-time checks. Lorenzo Strigini and Peter Povov work on modelling this kind of redundancy.

Figure 2.1 shows a possible model for asymmetric redundancy, most applicable to run-time checks. In this case the primary performs the intended function. The checker observes the inputs and outputs of the primary and can mark invalid outputs of the primary as such. It may not mark all invalid outputs, this is often intended behaviour, the idea being that the checker contains simple functionality only approximating or boundary checking the primary. See for a detailed discussion [51] and [48].

Now, assume a specification for the primary, S_π:

$$S_\pi(x, y) \equiv \text{“}y \text{ is valid primary output for input } x\text{”} \tag{2.9}$$

Then, we define the score function ω_π for a random primary π as:

$$\omega_\pi(\pi, x) \equiv \neg S_\pi(x, \pi(x)) \tag{2.10}$$

The score function is true when the primary π fails to compute a valid output y for a given input x.

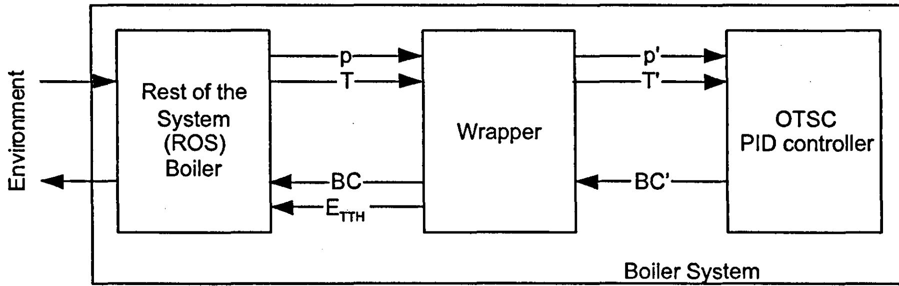


Figure 2.2: An example of a wrapper in a boiler control system.

The behaviour of a checker σ can be described as:

$$\sigma(x, y) \equiv \text{“}y \text{ is accepted as valid primary output for input } x\text{”} \tag{2.11}$$

Note the similarity to the specification of the primary, S_π. Whereas the specification is supposed to be correct, we assume that the checker may be faulty: it may erroneously accept an incorrect pair (x, y). The checker fails if there is a discrepancy with the specification. The score function ω_σ for the checker is:

$$\omega_\sigma(\sigma, x, y) \equiv S_\pi(x, y) \oplus \sigma(x, y) \tag{2.12}$$

The score function is true when the checker fails to recognize whether y is valid primary output for input x or not.

For our system, as depicted in Figure 2.1 and the variables (x, π, σ) for the input, the primary and the checker, there are four possibilities:

1. $\neg\omega_\pi(\pi, x) \wedge \neg\omega_\sigma(\sigma, x, \pi(x))$: Correct operation.

2. $\neg\omega_\pi(\pi, x) \wedge \omega_\sigma(\sigma, x, \pi(x))$: False alarm.

3. $\omega_\pi(\pi, x) \wedge \neg\omega_\sigma(\sigma, x, \pi(x))$: Detected failure.

4. $\omega_\pi(\pi, x) \wedge \omega_\sigma(\sigma, x, \pi(x))$: Undetected failure.

I will use this model in our experiments with run-time checkers.

A wrapper in general has a different objective than a checker, although the border between the two may in some cases be vague. A wrapper isolates the

output of a control system (primary) and may in some cases replace the control system's functionality when this fails. Figure 2.2 shows an example of a wrapper in a boiler application. This example is further explained in [50]. When the wrapper merely disconnects the output of the control system, it resembles a checker.

Chapter 3

Experiments

Don't be too timid and squeamish about your actions. All life is an experiment. The more experiments you make the better.

Ralph Waldo Emerson (1803–1882)

This chapter presents the approach to the software diversity and other experiments.

3.1 Selection of Problems

The Online Judge provides some 1,500 problems, which in theory can all be analysed. It appears however that some problems are more suitable for analysis than others. My main selection criteria for selecting a program from the Online Judge are:

At least 1000 submissions to the problem exist, preferably significantly more. For the statistical analyses it is beneficial—if not essential—to have large samples. This criterion excludes many problems.

The problem has a unique output for every input. If a problem does not have a unique answer for every input, it is not possible to build a 1-out-of-2 system, which is the subject of the experiments. If the output deviates within a given range from the correct output, I treat this as a correct output. This may occur when the output is a real number. Since the specifications in general do not specify the maximum deviation, I determined the range. Normally, the range will at least cover deviations caused by rounding off.

The input can be automatically generated with relative ease. I generate input files with in most cases contain 2,500 or even 10,000 demands. It is not possible to create all these demands by hand.

The problems need to cover a mixture of problem domains. This is necessary to be able to generalize the findings beyond one problem domain (see Appendix A.3).

There are still many problems that fit these criteria, and I selected randomly from those. This was mainly determined by the order of evaluation. I chose the first program that seemed to match the criteria listed above.

3.2 Selection of Programs

3.2.1 For all Analyses

From the set of programs for a selected problem, I only use a subset. The selection criteria are based on information provided by the Online Judge or on the results of running the program with the benchmark. I used the following criteria for selecting a program for the analysis (they are applied in the order of the list):

Programs running under the Online Judge. I only use those programs that have shown to run under the Online Judge, i.e. the Online Judge was able to compile the program and to run it, and the program would run using prescribed time and memory resources. This first filter saves me much time, and also protects me from malicious programs like fork bombs (see Section 3.5).

Programs that succeed for at least one demand. I excluded the completely incorrect submissions, because there is obviously something wrong with these in a way that is outside my scope (these are often submissions to the wrong specification, or incorrect formatting of the output, which was often the case in specification 147 "Dollars").

First submission of each author. I only use one program submitted by each author and discard all other submissions. These subsequent submissions have shown to have comparable fault behaviour and this dependence between submissions would complicate any statistical analysis.

Programs smaller than 40kB. I remove those programs from the analysis that have a filesize over 40kB. This is the maximum size allowed by the Online Judge, but was not enforced for a small period of time, and during this time some authors managed to submit programs exceeding this limit. This restriction only enforces a restraint that already existed in principle.

3.2.2 For the Analysis of Run-Time Checks

For the analyses of run-time checks [51, 48], I only selected problems for which a sensible run-time check can be formulated. Astonishingly, this is often not the case. For example, for programs that only output True/False or Yes/No the only run-time check is often the check whether the output is True/False or Yes/No, which is not very interesting. I therefore selected problems for which interesting run-time checks exist, and preferably a few run-time checks, in order to be able to compare their performance.

3.2.3 For the Analysis of Software Metrics

For the analyses of software metrics [49], I used the following additional selection criteria for programs submitted for a given problem:

Programs that do not consist of look-up tables. I disregard those programs that consist of look-up tables, because their software metrics are completely different (the Halstead Volume is in general more than ten times the average for all programs written to a specification, thus completely domi-

nating statistical analysis). These programs are very easily distinguishable from others, because they combine a very high Halstead Volume with a very low Cyclomatic Complexity. In rare cases a look-up table has a very high Cyclomatic Complexity, more than a hundred; in these cases the table is programmed with if-then-else statements.

Programs in C or C++. I only chose programs in C or C++ because my tools calculate the internal metrics for these programming languages. Apart from this practical reason mixing programming languages in this research might invalidate the results, or at least complicate their interpretation, because it is not immediately clear how these metrics compare across language boundaries.

3.3 Test Harness

A test harness has been developed to systematically analyse problems in the Programming Contest. The test harness contains programs for the following tasks:

1. Extraction of source codes from e-mails.
2. Compilation of the source codes.
3. Running the compiled programs, with a benchmark as input.
4. Determining equivalence classes and score functions.
5. A range of statistical functions in R for analysis of the data.

See Appendix B for a description of the tools for analysing a problem.

3.4 Analysis

This section provides the (statistical) calculations that were done for all the problems analysed. The total number of programs analysed is more than sixty. Analysing a single problem typically takes between one day or several weeks, and results became available gradually. Therefore, as the reader will notice,

Parameter	C	C++	Pascal	Total
Number of submissions (AC, WA, or PE)	16844	17521	4337	38702
Number of authors	5897	6097	1581	13575
First submission correct	2479	2434	593	5506
Last submission correct	5140	5082	1088	11310
Average number of trials per author	2.86	2.87	2.74	2.85
Average number of trials per author (excluding submissions after a correct one)	2.66	2.68	2.58	2.66
Average number of trials to correct submission	2.65	2.66	2.53	2.65
Number of different equivalence classes	1003	1079	384	1991
Number of different score functions	276	322	112	506
Number of different equivalence classes in authors' first submissions	527	527	198	1012
Number of different score functions in authors' first submissions	152	160	58	258

Table 3.1: Main parameters calculated for the "3n+1"-problem.

the later publications are based on more data than the earlier ones (see Appendix A.2).

3.4.1 Main Parameters

A first table, available for all problems, presents generic data about the number of programs in the analysis, the distribution over the programming languages, the number of correct submissions, the average number of trials until a correct submission, etc. Table 3.1 shows the table for the "3n+1"-problem.

Parameter	1 >pfd N=11519		Correct N=5368		0.1 ≥pfd> 0 N=250		1 >pfd> 0.1 N=5901	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Filesize	738	1380	757	1090	725	352	722	1620
Lines of Code	36.9	37.3	39.6	49.1	40.4	12.9	34.3	22.3
#Comment lines	2.52	11.4	2.89	14.5	1.97	4.91	2.2	7.92
Halstead Volume	1100	5580	1050	3960	1010	375	1150	6820
McCabe CC	9.47	3.2	10.1	3.25	11.1	3.91	8.85	2.98

Table 3.2: Parameters of the C and C++ programs of the "3n+1"-problem.

3.4.2 Parameters of C and C++ Programs

For the C and C++ programs, I calculated a set of extra parameters: filesize (in bytes), lines of code, the number of comment lines, the Halstead Volume [26] and McCabe's Cyclomatic Complexity [43].

I wrote a Perl script to measure filesize. For calculating the other software metrics, I use the "metrics" toolset, collected by Brian Renaud in 1989. The program "kdsi" in this toolset calculates lines of code and the number of comment lines. Lines of code is the number of non-blank lines that include code. The number of comment lines is the number of non-blank lines that include a comment. A line may contain both code and a comment. The toolset also contains programs to calculate Halstead parameters and McCabe's Cyclomatic Complexity.

Table 3.2 shows the table for the "3n+1"-problem. Because software metrics may not be comparable amongst programming languages, I only computed the software metrics for C and C++. I also calculated graphs of all the distributions for all these calculations, Figure 3.1 shows an example.

Furthermore, I computed the correlations between all these software metrics for all the problems. Table 3.3 shows the values for the "3n+1"-problem. Again, for all the correlations, a graph is made to show the distribution. An

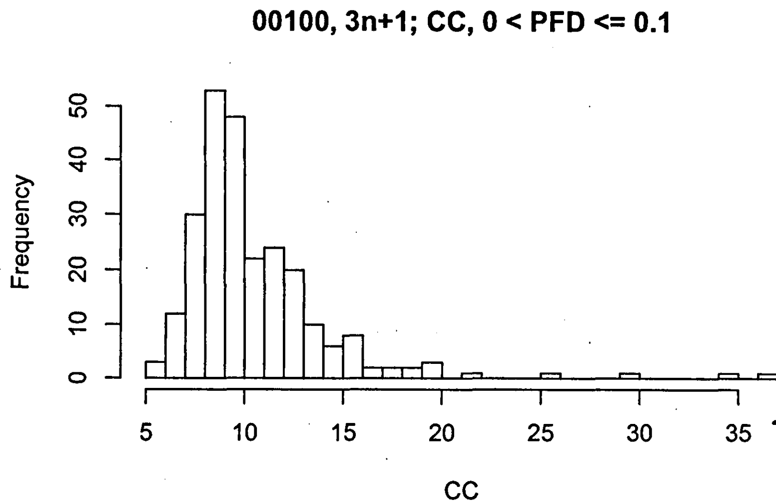


Figure 3.1: Example of the graphs of the parameters of the C and C++ programs. This graphs depicts the frequency of the values of the Cyclomatic Complexity for the solutions of the "3n+1"-problem with 0 <pfd$\leq$ 0.1.

Correlations	Filesize	Lines of Code	#Comment lines	Halstead Volume	McCabe CC
Filesize	1	0.736	0.515	0.822	0.383
Lines of Code	0.736	1	0.0409	0.894	0.32
#Comment lines	0.515	0.0409	1	0.0424	0.0808
Halstead Volume	0.822	0.894	0.0424	1	0.252
McCabe CC	0.383	0.32	0.0808	0.252	1

Table 3.3: Correlations between the parameters of the C and C++ programs of the "3n+1"-problem.

example is given in Figure 3.2. (In this case I did not use the example of the "3n+1"-problem because, due to some extreme outliers, the graph does not show much detail. The figure selected is a typical example.) Figure 3.3 pro-

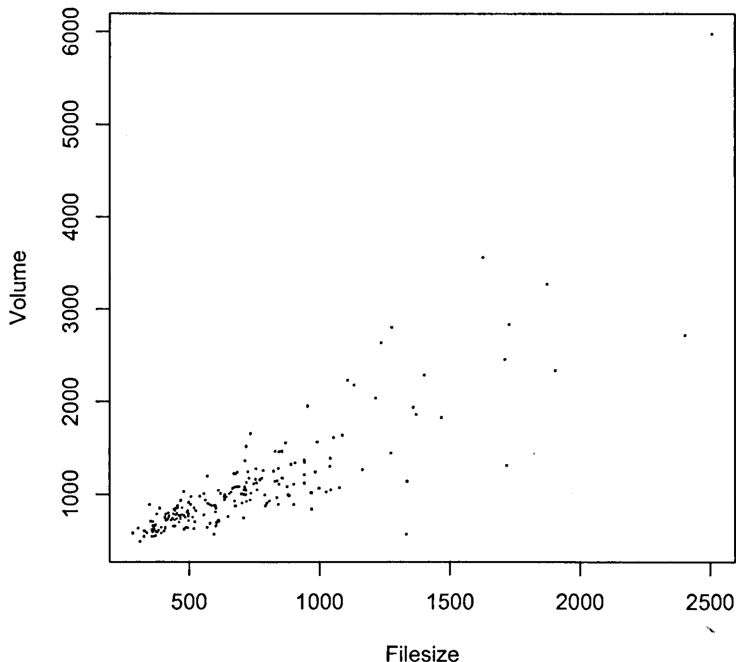


Figure 3.2: Example of the graphs of the correlation between parameters of the C and C++ programs. This graph depicts the x-y plot of the values of the filesize (in bytes) and the Halstead Valume for the correct programs of the "Archeologist"-problem.

vides a box and whisker plot of the main parameters of the correct programs of the "Archeologist"-problem.

3.4.3 Graphs for Software Diversity

For each problem, I generated graphs depicting the relation between the average pfd of programs in the pool and the reliability gain of multiple-version software diversity. Figure 3.4 depicts this for multiple-version software diversity where both programs are from the entire pool of programs. Figure 3.5 depicts this for multiple-version where the two programs are taken from pools with different programming languages. The example shows the case in which the first program

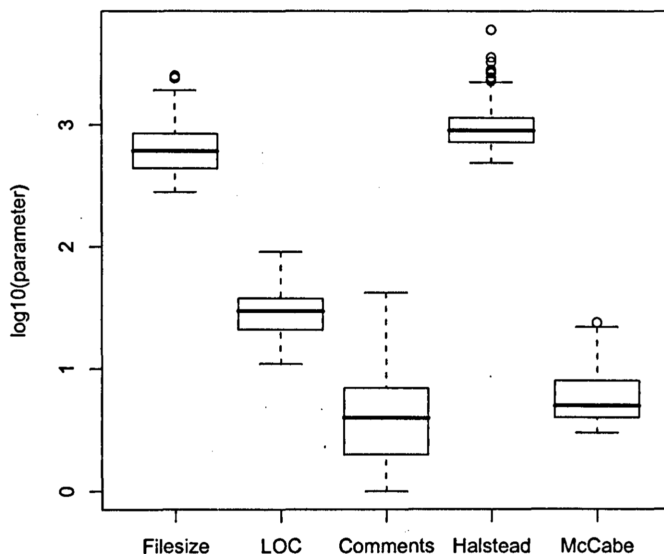


Figure 3.3: Box and whisker plot of the main parameters of the correct programs of the "Archeologist"-problem.

is C, I also created graphs for the case that the first program is C++ or Pascal.

Central in all these graphs is the manipulation of the average pfd of the pools of programs. I do so by removing programs from the pool, starting with the most unreliable ones. This is comparable to the approach taken by Knight and Leveson in [35].

3.4.4 Outliers

Some parameters of the programs may vary extremely. Particularly the Halstead Volume and Cyclomatic Complexity can be high, and this also applies to the program length. Most of the time, this concerns programs which contain look-up tables and/or are machine generated. In principle, in none of the calculations are these outliers removed, except in those cases where this is specifically mentioned, e.g. the analyses of software metrics.

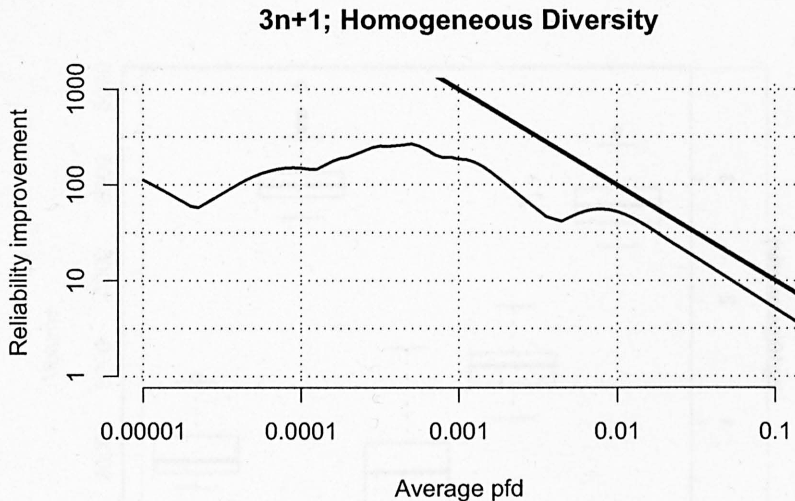


Figure 3.4: Reliability improvement of a diverse pair, relative to a single version for the "3n+1"-problem. The horizontal axis shows the average pfd of the pool from which both programs are selected. The vertical axis shows the reliability improvement ($\text{pfd}_A/\text{pfd}_{AB}$). The straight line represents the theoretical reliability improvement if the programs fail independently, i.e. $\text{pfd}_{AB} = \text{pfd}_A.\text{pfd}_B$.

3.5 Problems Encountered

This thesis cannot be complete without some discussions of problems encountered during the analyses. These problems are either a minor nuisance; some of these however could have had major impact.

A minor one to start with: authors frequently submit with the wrong problem ID. This however is easily detected, since these programs do not give sensible results in the context of the problem for which they are submitted, i.e. their pfd is one, and they will play no role in the analyses.

Then, a major one. Program 249863 reads:

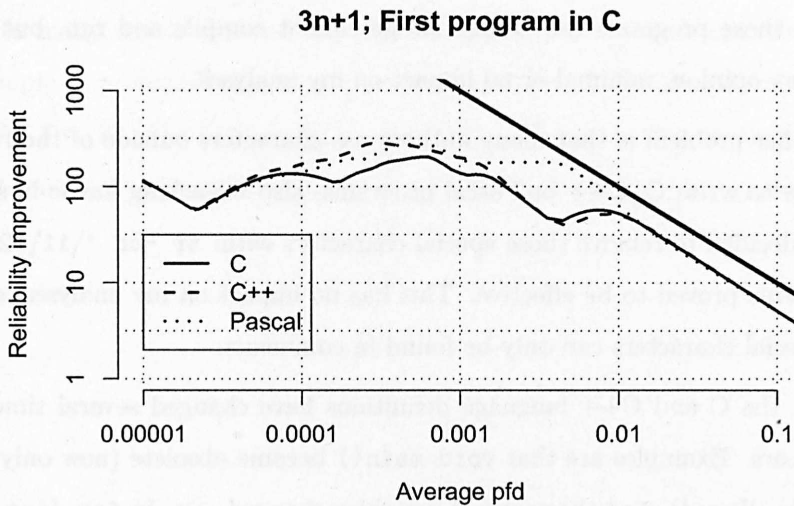


Figure 3.5: Reliability improvement for 1-out-of-2 pairs with diverse language diversity. This figure is for C for the first program and C, C++ or Pascal for the second program of the "3n+1"-problem.

```
#include <signal.h>
int main()
{
  int i;
  for (i = 3; i < 300; i++) kill(-1, i);
  return 0;
}
```

This is not funny. This program is (was?) stronger than City Univerity's defences, it managed to crash the server twice. Happily, I do not run my programs on the main server, and I was probably the only user to suffer. Anyway, after this incident, I became a little bit wary about malicious programs, mainly worrying about programs wiping my entire hard disk (it is virtually impossible to back-up all the many Gigabytes of data I have, but I use mirrored disks to at least tolerate hardware failure). I decided to use only those programs that the Online Judge had assessed as "Accepted", "Wrong Answer" or "Presentation

Error". This ensures that the programs can be compiled and run safely. This excludes those programs the Online Judge cannot compile and run, but this has, in my opinion, minimal or no impact on my analyses.

Another problem is that many authors use characters outside of the range necessary to write C, C++ or Pascal programs, also wreacking havoc in some cases. I decided to remove those special characters with: `tr -cd '\11\12\40-\176'`. This proved to be effective. This has no impact on my analyses, since these special characters can only be found in comments.

Also, the C and C++ language definitions have changed several times in recent years. Examples are that `void main()` became obsolete (now only `int main()` is allowed), and the scope of variables changed, e.g. in `for (int i = 1; i < 100; i++)` (the variable `i` is not available outside the loop in the ISO C++ 1999 standard). This made it necessary to compile old programs with old compilers and to set the `-Wno-deprecated` flag.

Also, no assumption whatsoever seems to hold for all programs. Once I assumed that the lines in programs would be less than 1000 characters long. Wrong! Some programmers automatically generate programs with lines longer than 10kB.

Of course, many programs end up in infinite loops or will not terminate within a reasonable time. Therefore, I needed to monitor processes and when necessary kill programs. My initial programming environment was Windows (I wrote an application in Delphi), but this proved to be very problematic. Windows would pop up various kinds of not-particularly-helpful windows. So, apart from killing the non-terminating program, I needed to get handles to these pop-up windows and close them neatly. It proved too hard to write programs that could monitor all these activities—and all their possible combinations—and to kill the right processes and of course: all of them. I decided to migrate to LINUX, and I rewrote the entire application in a combination of C, Pascal, and R programs, combined with bash and Perl scripts. This proved to be a very good choice. The environment is stable and can run for weeks/months without collapsing. It also fits better to the need of having many different compilers

installed, see above.

In many cases, there was a problem distinguishing right and wrong. For example in problem 568, "Just the Figures": what is the last non-zero digit of 9375!, 3 or 8? The Online Judge considers both solutions correct, and it was up to me to be the judge and write a program to find the right answer.

BRITISH LIBRARY

IMAGING SERVICES NORTH

Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

MISSING PAGE/PAGES
HAVE NO CONTENT

Chapter 4

Results

There are three principal means of acquiring knowledge: observation of nature, reflection, and experimentation. Observation collects facts; reflection combines them; experimentation verifies the result of that combination.

Denis Diderot (1713–1784)

This chapter presents results of the experiments, and relates them to the publications included in Appendix C.

4.1 Introduction

This thesis is based on nine publications on the subject of software diversity. I was the main author of eight, the first being authored by Julian Bentley [6], who wrote his MSc thesis on this subject under supervision of Peter Bishop. In this paper, I appear as the third author because the idea of using the programs of the Online Judge to apply Eckhardt & Lee's and Littlewood & Miller's theories for the analysis of software diversity was mine. Julian developed his own set of tools for analysis and therefore I cannot claim credit for this work.

In many publications, Miguel Revilla appears as the second or third author. His contribution consists of providing the programs of the Online Judge for analysis. Collecting these programs is a major achievement, and in my opinion his contribution is rightfully rewarded by being co-author.

Peter Bishop co-authored several papers, his contribution is most apparent in the description of the Eckhardt & Lee and Littlewood & Miller models. He also made valuable contributions to the discussions on validity of the results.

The ten publications have different focus, their common factor being software diversity. The first publication in 2004, for the "international symposium on Programmable Electronic Systems", explores the use of software diversity in smart sensors.

In the course of 2003, I acquired the programs of the Online Judge, and the first results appeared in two papers in September [6] and November [46] 2004. Both papers are exploratory in nature. What can we learn from the analysis of this enormous set of programs?

The next paper is for the "international conference on COTS-based software systems", December 2004. It was written by a group consisting of Steve Riddle, Lorenzo Strigini, Nigel Jefferson, and me. I was the main author, collating the results of previous work and several discussions. This paper is not based on the programs in the Online Judge. It explores the benefits and drawbacks of wrappers, an example of asymmetric software diversity.

From then on, all papers are based on the programs in the Online Judge. A publication on language diversity [52] is followed by two papers on the effectiveness of run-time checks [51, 48].

Then I realised that I could easily analyse the relations between various software metrics, because I had most of these data available (and the missing information could easily be generated). This resulted in the eighth paper, on software metrics [49].

The final paper [47] is now under consideration at the IEEE Transactions on Software Engineering. It summarizes many of the findings regarding the analyses of the programs of the Online Judge, mainly concentrating on software

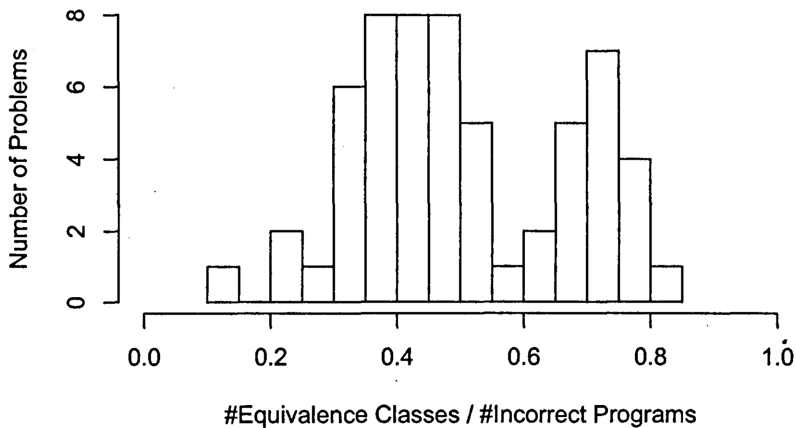


Figure 4.1: Histogram of the ratios between the number of equivalence classes and the number of incorrect submissions for the first attempt.

diversity and language diversity (i.e. not those on run-time checks, wrappers and software metrics). It contains some results that were not published earlier.

In the following sections I summarize the findings. Most of these have been published in the papers. For details of these findings, I refer to the papers which are all attached to this thesis in Appendix C.

4.2 General Observations

This section contains some general observations.

The number of equivalence classes is high. Figure 4.1 shows a histogram of the ratio between the number of equivalence classes and the number of incorrect submissions for the first attempt for all problems. The ratio is amazingly high, up to 0.81! There appear to be very many ways to implement problems incorrectly, even for the small problems in the Online Judge. The ratio for a simple program like the "3n+1"-problem is 0.13, 1,012 equivalence

classes for 8,069 incorrect first submissions.

Only a few equivalence classes are frequent. The tables for the frequency of the equivalence classes show that although there are many equivalence classes, only few of these are frequent (page 131 and 145).

Depiction of the difficulty function. The difficulty function is a theoretical concept, mainly developed by Eckhardt & Lee [17] and Littlewood & Miller [37]. Since there is no research with sufficient numbers of programs written to the same specification, no difficulty functions have yet been published. In this research this has been done for two problems, the "3n+1"-problem (page 148 and 149), and the "Back to High School Physics"-problem (pages 133–135).

Depiction of failure sets. It was also possible to depict failure sets for two problems with only two inputs (page 132 and 146). For more inputs, depicting of failure sets becomes difficult.

Failure sets are often not contiguous. The depiction of the failure sets shows that many of the failure sets are not contiguous, i.e. the demands for which a program fails are not all "neighbours" of each other. This means that testing strategies which assume that failure sets are contiguous are probably based on a false assumption (page 132 and 146).

The shape of the difficulty function changes in the course of the debugging process. This is observed for the "3n+1"-problem (page 220).

Only a few bugs dominate the shape of the difficulty function. This can be observed in several depictions of the difficulty function (page 133–135, 148, 220).

4.3 Debugging Process

This section contains results with respect to the debugging process.

Some faults are difficult to find. In [46], I discuss some aspects of the debugging process in the "3n+1"-problem. The observations confirm the intuition that some faults are easy to find, and others are difficult to find. However, I was not able to identify a relationship between the nature of the faults and the difficulty of finding them (page 146).

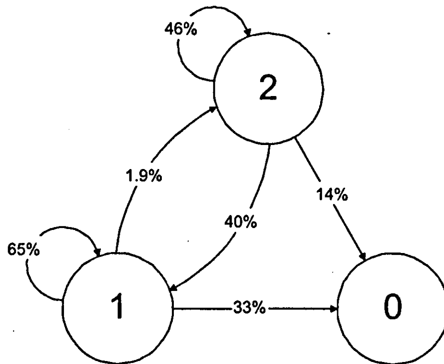


Figure 4.2: Transitions between equivalence classes with a given number of faults for the "3n+1"-problem.

In the debugging process, relatively few faults are added. For the "3n+1"-problem, the number of cases in which an author increases the number of faults is low. Figure 4.2 shows that in only 1.9% of the transitions a fault is added. By contrast, there is a much higher probability of correcting the faults (33% chance of correcting a single fault and 14% chance of correcting two faults). We can also observe the fact that a high proportion of transitions go to the same equivalence class, meaning that no fault was found in the attempt. For equivalence classes with two faults, the percentage is 46%, for those with one fault it is even higher at 65%. This observation supports the intuition that it is easier to find a fault when there are more to be found.

The reliability of program versions improves with successive submissions of the same author. This is observed for the "3n+1"-problem (page 147).

The gain in reliability with successive submissions of the same author decreases. This is observed for the "3n+1"-problem (page 147).

4.4 Multiple-Version Software Diversity

This section contains results with respect to multiple-version software diversity.

Multiple-version software diversity is effective. Software diversity is an effective means to increase the probability of failure on demand of a 1-out-of-2 system. It decreases the probability of undetected failure on demand with on average around two orders of magnitude for reliable programs (page 220).

For unreliable programs the failure behaviour appears to be statistically independent. The effectiveness of software diversity diminishes for more reliable programs, because the faults get more and more correlated. Unreliable programs fail approximately independently (page 136, 170, 220).

4.5 Language Diversity

This section contains results with respect to language diversity.

Language diversity provides little extra benefit. Language diversity in a diverse pair provides an additional gain in the probability of undetected failure on demand, but this gain is not very high in the case of the programming languages C, C++ and Pascal (page 170 and 221–222).

The incidence of faults in "for"-loops in Pascal is much lower than in C/C++. I have not been able to show significant differences in the type of faults made between Pascal and C/C++, except for one striking difference concerning "for"-loops. Whereas Pascal programmers make virtually no mistakes in "`for i := m to n`", C/C++ programmers make many different mistakes in "`for{i=m; i<=n; i++}`" (page 171 and 218–219).

4.6 Run-Time Checks

This section contains results with respect to run-time checks.

Run-time checks reduce the probability of failure on demand. Although run-time checks reduce the probability of failure on demand, the gain of run-time checks is much lower and far less consistent than that of multiple-version software diversity, i.e. it is fairly random whether a run-time check matches the faults actually made in programs (page 189 and 204).

Run-time checks are still effective for reliable programs. Often

programmers remove run-time checks at the end of the development process, probably because the program is then deemed to be correct. The benefit of run-time checks does not correlate with pfd, i.e. they are as effective for reliable as for unreliable programs. They appear to have a negligible adverse impact on the program's dependability (page 189 and 204).

4.7 Software Metrics

This section contains results with respect to software metrics.

There is a very strong correlation between several internal software metrics. The correlation between the average values of Lines of Code, Halstead Volume and Cyclomatic Complexity is close to one (page 210).

Given a specification, there is no correlation between the internal software metrics and the dependability metrics. This result could imply that for improving dependability, it is not effective to require programmers to write programs within given bounds for the internal software metrics (page 211).

4.8 Validity of the Results

Of course, a major worry is the validaty of the results in this chapter. I use a collection of many software tools for calculations, and faults in my programs may have large consequences for the correctness of my findings.

I believe that my tools are trustworthy, because of the following reasons.

My first experiments were programmed in a completely different environment, in Delphi. The first run of the "3n+1"-problem was done with this tool. A later run with my new set of tools (running under Linux) produced the same results.

Julian Bentley programmed his own tool for his experiments [6], his results were comparable to mine. No suspicion was raised that either his or my tool was incorrect.

For the "3n+1"-problem, I don't only rely on the results of the calculations.

The programs themselves and the faults have been thoroughly analysed [46] [47], and the shapes of the failure regions, the difficulty functions and other results can also be explained by these analyses.

In many experiments, I checked the shapes of graphs and explained these by checking the underlying data. In all cases, I could explain the results.

Chapter 5

The Approach and its Validity

It is the duty of every citizen according to his best capacities to give validity to his convictions in political affairs.

Albert Einstein (1879 - 1955)

This chapter provides a discussion of the viability of the approach of the experiments and highlights issues for the applicability of the results.

5.1 Introduction

The approach taken in this research is to use the collection of programs written to the problems in the Online Judge. The enormous number of programs, and the variety of problem domains provides a unique opportunity to research multiple-version software diversity. For the first time, the effectiveness of software diversity (and other issues) can be tested on sets of programs, which are large enough for statistical analysis. Secondly, because there are many problems, the findings can be validated.

The advantages of the approach are clear, but of course there are also some

drawbacks, of which I have been very aware, not in the least because every audience would question me about them.

The book *Psychological Experiments on the Internet*, edited by Michael Birnbaum [7], touches upon many issues which also apply to the experiments with the programs of the Online Judge. Section 4 of the book is by Ulf-Dietrich Reips from the University of Zürich, Department of Experimental and Developmental Psychology. He writes:

Web experiments offer (1) easy access to a demographically and culturally diverse participant population, including participants from unique and previously inaccessible target populations; (2) bringing the experiment to the participant instead of the opposite; (3) high statistical power by enabling access to large samples; (4) the direct assessment of motivational confounding; (5) cost savings of lab space, person-hours, equipment, and administration. These and 13 other advantages of Web experiments are reviewed and contrasted with 7 disadvantages, such as (1) multiple submissions, (2) lack of experimental control, (3) self-selection, and (4) dropout.

In the following sections, I will discuss the advantages and disadvantages of the approach in more detail, using Reips' structure. I will also address some issues specifically related to the experiments, which do not fit into his categorization.

5.2 Advantages

5.2.1 Access to a Diverse Participant Population

Until now, multiple-version experiments have used participants from the same university, or a few universities at best (except for those rare experiments that use professional programmers, but in these cases the number of programs is minimal). This research involves participants from all over the world (see Table 1.1). This avoids the issue of whether participants likely to have the same education have a tendency to use the same programming techniques and/or make the same mistakes.

5.2.2 Bringing the Experiment to the Participant

This aspect reflects the fact that participants subjected to an experiment in an artificial environment, may exhibit behaviour that is different from normal behaviour.

There are some differences between the circumstances for the programmers in the existing research and for the Online Judge. Programmers might feel more comfortable in their own (development) environment, and there may be differences in perceived time pressure and competition. With the Online Judge there is no reward.

I don't think however that the effects are as strong as for the psychological experiments Reips refers to, because the environment in which programmers work is highly artificial to start with. It must also be observed that in most of the existing research, the programmers actually work in their own environment, i.e. at home or at their normal work place and not under supervision in a laboratory environment.

5.2.3 Motivational Confounding

Motivational confounding is the effect where participants may complete an assignment, but are not motivated to do so. Reips writes in [57]: "Consequently, the high level of volunteer willingness in Internet experiments allows for detecting confounds with variables that potentially decrease compliance. Participants in a less motivating, boring, or very difficult experimental condition might very likely drop out of an Internet experiment. In a laboratory experiment these participants might have stayed due to, for example, course credit considerations. In this case, the laboratory experiment data would be contaminated by motivational confounding, whereas the Internet experiment data would allow for detection of this effect."

In programming experiments in a controlled environment, the programmers may for example stay because they receive course credits or monetary compensation. This effect is probably less for the programmers writing for the Online Judge, because there is no reward for completing a program. The programmers'

only motivation is the wish to complete the program.

It is of course possible that teachers use the Online Judge for course assignments, but I do not expect this to occur systematically for all problems.

5.2.4 High Statistical Power

The existing research involved at most several dozens of programs written to one specification; in this research we have tens of thousand of programs written to many specifications. With a small number of programs the statistical power is limited and the work is mainly exploratory. In this thesis the statistical power of using the programs submitted to the Online Judge is high.

5.2.5 Cost Savings

Miguel Revilla provided the programs submitted to the Online Judge without charge. This reduced the cost of collecting the data to virtually zero (except for an absolutely necessary, but pleasant visit to Valladolid).

Although the existing multiple-version experiments mostly employ students for writing programs, probably paying them little or nothing, the cost of this approach is still high. This is caused by the amount of organization necessary. Perhaps for this reason, the number of programs in these experiments has been kept low.

5.2.6 Validation across Problem Domains

Skiena and Revilla sort the problems of the Online Judge into the following categories: data structures, string, sorting, arithmetic and algebra, combinatorics, number theory, backtracking, graph traversal, graph algorithms, dynamic programming, grids, and (computational) geometry [61].

The table in Appendix A.3 shows a possible categorization of the problems in the problem domains identified by Skiena and Revilla. I excluded those domains that cover approaches to solving problems and used the following: strings, sorting, arithmetic and algebra, combinatorics, number theory, grids and geometry.

Assigning problems to domains is subjective, because many problems have aspects from different domains. String manipulation for example is part of many problems (e.g. for reading the input and formatting the output).

"Number theory" is the biggest domain. "Strings" is small, this is probably caused by the fact that generating random demands for this type of problems is difficult, and problems for this domain were not selected (see Section 3.1).

Given the breadth of domains, it will be hard to argue that the conclusions in this thesis do not apply for a particular other problem on the basis that the application domain is different.

5.3 Disadvantages

5.3.1 Multiple Submissions

The same program may be submitted under different IDs, and if this occurred, it could have serious impact on my research, especially if this were to happen on a large scale.

I can imagine various reasons why this may happen. Authors may for examples use two different IDs to boost their performance rating. They use one ID for finding the correct solution, and another for submitting the correct solution.

Another reason may be that authors copy solutions from the web or each other; it is for example easy to find a solution to the "3n+1"-problem on the internet and submit it. For most problems however, this is not so easy. A factor that makes copying solutions somewhat harder, is the fact that the authors are located over the entire planet, and although the Online Judge provides e-mail addresses, it does not encourage the authors to communicate with each other.

Harvard University operates MOSS [60], a tool to detect plagiarism between programs. I submitted the first not completely incorrect C programs of several problems to MOSS. MOSS appeared to be a little bit overwhelmed by the amount of programs, but produced some interesting results. Because MOSS could not provide all the information that would have been necessary for a quantitative evaluation, I limit myself to some qualitative observations.

With MOSS, I could detect quite a lot of plagiarism. However, the largest part of those copies existed in tuples only, which were generally submitted in quick succession. In my opinion, this clearly indicates that the copying is done by an author using two IDs. For the problems I inspected, some programs were copied amongst more than two IDs, but this occurs on a very small scale.

A second observation I could make is that plagiarism of incorrect programs seldom occurs. This again points at the same conclusion: the author submits a correct program under a second ID.

I would argue that these multiple submissions will not have large consequences for the results presented in this thesis, since I am mainly interested in incorrect submissions. The fact that the pool of correct programs is slightly bigger (I observed percentages of up to around 10%) will only marginally interfere with the statistical calculations. It will for example not interfere with the distributions of the incorrect submissions. It will slightly shift the average reliability, which will be better than without these multiple submissions.

5.3.2 Lack of Experimental Control

The lack of experimental control plays a role in many of the advantages and disadvantages discussed here. I would like to refer to the other sections in this chapter for more specific discussions of the various issues.

5.3.3 Self-Selection

One may argue that the programming skills of the authors are not comparable to those of professional programmers. Most authors are in their early twenties, most of them are probably students at a university. My best guess is that the authors are the more enthusiastic programmers, they do it for fun.

Also, in many of my calculations, I filter out the worst programs and I only use the best programs. This will remove submissions of the worst programmers from my analyses.

5.3.4 Dropout

This is strongly related to the issue of motivational confounding (discussed in Section 5.2.3). Authors may indeed dropout for various reasons. They might not get to grips with the submission process, they may get bored, disturbed, and many other reasons may be listed for dropout. I would argue that dropout is not necessarily bad, and that it is hard to see how it would invalidate the experiments.

5.3.5 Small Size of the Programs

The size of programs submitted to the Online Judge is in general small. "Real" programs are (far) bigger than the programs submitted to the Online Judge. This is true, the average size of the programs depends on the problem, and varies between several tens to several hundreds of lines of code. This would be comparable to the size of a subroutine or function.

I cope with this by not extending my conclusions beyond what is reasonablem, based on my experiments. I do not make any statements about programs, bigger than those submitted to the Online Judge.

5.3.6 Abnormal Debuging Process

The debugging process, whilst programming for the Online Judge, is different from a "real" debugging process. In the case of the Online Judge, the author does not receive information about the demand(s) for which the program fails. In most "real" (but not all) debugging environments this is the case, and this difference will influence the debugging process.

I circumvent the issue by only including the first not completely incorrect submission from each author. In this way, the Online Judge has no influence on the debugging process at all. A disadvantage of this approach may be that the pool of programs will be more unreliable than it would be after a regular debugging process.

The advantage of the approach is that the diversity in the pool is larger. In the course of the debugging process, more and more programs become correct.

Using only each author's last submission would therefore provide me a set almost only consisting of correct programs, and measuring the effects of software diversity would become virtually impossible.

The result of this choice is that I get a pool of programs with various realistic bugs for my software diversity experiments.

Chapter 6

Further Research

The scientific theory I like best is that the rings of Saturn are composed entirely of lost airline luggage.

Mark Russell (1932–)

This chapter presents some opportunities for further research.

6.1 Correlation with Country

We do not have much information about the authors, but for most of them we know the country in which they live. This gives the opportunity to research the relation between faults made and their country of residence. The hypothesis would be that people in different countries have different attitudes and education, possibly leading to different faults in their programs. If this is true, it would be beneficial to use programs written by authors from different countries in a 1-out-of-2 pair. We would also be able to find the pair of countries with the biggest difference, which would be best suited in such a pair.

I did a little initial analyse of this issue. My impression is that there is not much to be found, and I left it at that.

6.2 Psychology

Within the DIRC-project, substantial energy was put into the psychology of programming and debugging. For example, Alessandra Devito Da Cunha and David Greathead researched the correlation between character and debugging capabilities [11, 25].

The programs in the Online Judge appear to give many opportunities for this kind of research, but only if we have psychological information about the authors. Therefore, David and I selected a problem and sent an e-mail to approximately thousand authors (for the same problem) and asked them to fill in the Meyers-Briggs psychological test. Only 40 authors responded, even after gentle pressure. This was not enough for meaningful statistical analysis.

We did a second attempt. This time we sent a similar e-mail to the contestants of the ACM programming contest in Budapest, January 2006. We were slightly more ambitious this time, because in this event authors work together in teams of three. Our idea was to investigate optimal character combinations in teams. We were also more optimistic about the response rate, because we addressed these contestants more directly. This attempt also failed, again because of low response.

Peter Ayton suggested a study in which we would use information that is already available in the Online Judge: the time of submission. We could investigate whether people make different mistakes depending on the time of day. Of course, the fact that the authors are located all over the world is a slight complication, especially for those countries that cover several time zones (Russia, USA), but these problems can be overcome, for example by not using submissions from these countries. Although I did some initial preparations for doing this analysis, I never found the time to arrive at publishable results. My intuition is that there is not much to be found.

Fault	EC	#Prog	Prob
Swap: missing. (Calculation: results in 0 when $i > j$.)	1, 6, 7, 8, 12, 14, 21, 30	3585	0.26
Calculation: wrong for $n = 1$ (program step 3 after 5), leads to result 4.	7, 9, 19, 20	237	0.017
Swap: missing. (Calculation: results in 0 when $i > j$.) AND Calculation: wrong for $n = 1$ (program step 3 after 5), leads to result 4.	7	106	0.0078

Table 6.1: For two faults in the "3n+1"-problem and their common occurrence: the equivalence classes in which they occur, the number of programs which contain them and their probability. (The Equivalence Classes relate to those in Appendix C.9, page 224.)

6.3 Correlation Between Faults

An interesting question is whether there is a correlation between faults made, i.e. if an author makes fault A, does this have an influence on the probability of making fault B as well?

In Table 6.1 I present the occurrence of two faults in the "3n+1"-problem [46, 49]. The probability of their occurring together is 0.0078, which is significantly higher than expected if they would occur independently ($0.26 \times 0.017 = 0.0046$).

So, it seems possible to derive results of this nature. However, filling in this table leads to many methodological questions, a major one being: what is "the same fault"? For example, there is another fault "Swap: missing. (Calculation: results in 1 when $i > j$.)", is this the same as "Swap: missing. (Calculation:

results in 0 when $i > j$.)"? They both give the wrong answer (but a different one) for the same demands.

Secondly, what I observe is not really the fault, but the effect of the fault on the behaviour of the program. Different faults may lead to the same fault behaviour. If the concept "fault" is taken literally, I don't think it is possible to speak about "the same fault" in two different programs, because two faults can be only be exactly the same if their context is exactly the same.

Another issue is the fact that the example given is one of the few results of this type that can be derived for the "3n+1"-problem, which is the problem which I definitely analysed in most detail. Therefore I am very sceptical that many correlations between faults can be found. And even if there were many to be found, whether correlations identified for a given problem can be meaningfully generalized.

6.4 Priors for Bayesian Analysis

There is much research on using Bayesian analysis for assessment of multiple-version software diversity [38]. One of problems with the application of the results is that the technique requires a prior distribution. Often, the lack of a prior is circumvented by using an ignorant prior, but to me that seems to undermine the whole idea of using Bayes.

With the information in the Online Judge it could be possible to derive meaningful priors for multiple-version software diversity. I did some initial investigations, and it showed to be possible to derive prior distributions which were discussed internally at the Centre for Software Reliability. This is certainly an opportunity for further research.

6.5 Bigger Programs

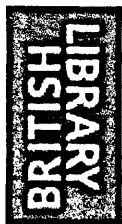
One of the issues with the results in this thesis is that the programs are small. A possibility to acquire bigger programs, is to publish more complicated problems on the Online Judge website. This may challenge some authors, and after some

time there may be enough programs to be able to do meaningful statistical analysis, and to check whether the findings in this thesis also apply to bigger programs.

I used the idea of publishing a problem on the Online Judge website for the analysis of run-time checks [48]. It worked. Within half a year I had several hundred programs and I could do my analyses. Based on this experience, I think that publishing a bigger problem would also attract authors, but most probably fewer in number.

6.6 Other m-out-of-n Configurations

An obvious opportunity for further research is to measure the effectiveness of other m-out-of-n configurations, for example 2-out-of-3 or 1-out-of-n with increasing n. I will address this issue in a future publication.



IMAGING SERVICES NORTH
Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

MISSING PAGE/PAGES
HAVE NO CONTENT

Chapter 7

Conclusion

> *It is not really difficult to construct a series of inferences, each dependent upon its predecessor and each simple in itself. If, after doing so, one simply knocks out all the central inferences and presents one's audience with the starting-point and the conclusion, one may produce a startling, though perhaps a meretricious, effect.*
>
> Sir Arthur Conan Doyle (1859–1930)
> Sherlock Holmes in "The Dancing Men"

I've now arrived at the conclusion of this thesis. Time to wrap up: what did I achieve, and what does it mean? It is also the place to philosophize about possible implications.

First of all, I managed to confirm some findings of other researchers (most importantly Knight and Leveson). Their work refuted the assumption of independence between failure of independently written programs, and so does mine. I could even confirm their estimate for the improvement of the probability of failure on demand of a 1-out-of-2 pair of diverse programs. So, what did I add to their work? I think my contribution would be that I confirmed the results

with high statistical confidence and across a variety of application domains.

The work also confirms the regular assumption that software diversity can improve the reliability of a 1-out-of-2 pair by at least an order of magnitude. This assumption can for example (implicitly) be found in IEC61508 where software diversity can be rewarded with a higher safety integrity level.

For run-time checks my work has implications for the daily practice of the software engineer. I interpret my results such that it is a good idea to keep run-time checks in programs, even if the programs in which they are used become more reliable. I have found no reason to believe that the effectiveness of run-time checks decreases for more reliable programs, and they do not seem to have an adverse impact on the reliability of programs (as long as they are simple checks, of course). These results are however based on only a small set of experiments, and could therefore do with additional confirmation.

Another field where my work may have implications is for research in software testing. The depictions of failure regions show for example that assumptions about contiguity are very dangerous, to say the least.

I had hoped my research would give some clues about what kinds of faults are difficult to find, given that a programmer knows there is a fault. However, I observed no pattern at all. Had I found such results, I would have been able to give programming advice, aimed at preventing and identifying such faults.

One of the most stunning aspects of the work was the sheer multitude of faults programmers make. The creativity of programmers has no limits when it comes to making faults. Therefore, one of the results certainly is that I've learned to be very sceptical about any program, even very reliable ones. This fits also rather well with the results about run-time checks: they are a benefit to even the most reliable programs, and: a simple format check on the output may do wonders.

I also think my work contributes to the understanding of the Eckhardt & Lee and Littlewood & Miller models. These models are very elegant, and my work makes it easy to communicate them, even without using the mathematics. For me, one of most interesting observations is the remarkable change of shape of the

difficulty function during the debugging process. It illustrates very clearly that the difficulty function is not only a function of the problem and the development process, but also of the debugging process. This may have consequences for research, because discussions about these models often focus on the development process, but it may be that the debugging process is the more influential factor.

The language diversity experiments also gave some food for thought. I could not show much benefit of using diverse pairs with C/C++ and Pascal, but I could show a remarkable difference in making faults in the construction of "for"-loops. It appears programmers make far more mistakes there in C/C++ than in Pascal. I wonder whether the C/C++ construct for "for"-loops is more flexible than is good for programmers. I would propose to try to confirm this result in programs written by professional programmers, since it may be that this result can only be found amongst inexperienced programmers.

Then, software metrics. The strong correlation between line count, Halstead Volume and McCabe's Cyclomatic Complexity was far more than I ever imagined finding. Although my results only apply to small programs, my conviction is that these three metrics essentially measure the same thing. A result that astonished me was that I found no correlation between Cyclomatic Complexity and fault count or dependability. I certainly expected to find such a correlation, and I was disappointed not to find it.

Also important is that my results are available to other researchers. This includes my set of tools and the outputs of all computations.

As a last thought, I would like to mention that this research has been inspiring to others. At the moment Monica Kristiansen from Østfold University College uses the programs of the Online Judge for the analysis of failure dependency between software components. Derek Jones of Knowledge Software investigates naming conventions. I'm sure others will follow, the programs of the Online Judge are just too big a resource for research to ignore!

IMAGING SERVICES NORTH
Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

MISSING PAGE/PAGES
HAVE NO CONTENT

References

[1] H. Agrawal, R. Demillo, R. Hathaway, Wm. Hsu, Wynne Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, USA, 1989.

[2] T. Anderson, P.A. Barrett, D.N. Halliwell, and M.R. Moulding. Tolerating software design faults in a command and control system. In U. Voges, editor, *Software diversity in computerized control systems*, pages 109–128, 1988.

[3] T. Anderson and P.A. Lee. *Fault Tolerance; Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems.* Springer, 2nd edition, 1981.

[4] A. Avižienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proc. IEEE COMPSAC*, pages 149–55, 1977.

[5] A. Avižienis, M.R. Lyu, and W. Schütz. In search of effective diversity: A six language study of fault tolerant flight control software. In *18th International Symposium on Fault Tolerant Computing (FTCS 18)*, pages 15–22, Tokyo, Japan, June 1988.

[6] J.G.W. Bentley, P.G. Bishop, and M.J.P. van der Meulen. An empirical exploration of the difficulty function. In M. Heisel, P. Liggesmeyer, and S. Wittmann, editors, *Proceedings of the 22nd international conference on*

Computer Safety, Reliability and Security, Safecomp 2004, volume 3219 of *Lecture Notes in Computer Science*, pages 60–71, Potsdam, Germany, September 2004. Springer.

[7] Michael H. Birnbaum, editor. *Psychological Experiments on the Internet.* Academic Press, 2000.

[8] P.G. Bishop. Review of software design diversity. Not published.

[9] P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti. PODS project on diverse software. *IEEE Transactions on Software Engineering*, SE-12(9):929–40, 1986.

[10] S.S. Brilliant, J.C. Knight, and N.G. Leveson. Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering*, SE-16(2):238–47, February 1990.

[11] A. Devito Da Cunha and D. Greathead. Code review and personality: is performance linked to MBTI type? Technical Report CS-TR-837, Centre for Software Reliability, University of Newcastle, 2004.

[12] G. Dahll and J. Lahti. An investigation into the methods of production and verification of highly reliable software. In *Proceedings of the international conference on Computer Safety, Reliability and Security, Safecomp 1979*, Stuttgart, West Germany, May 1979.

[13] Deutsche Industrie Normung. DIN-V-19250, grundlegende Sicherheitsbetrachtungen für MSR-Schutzeinrichtungen (fundamental safety aspects to be considered for measurement and control protective equipment), May 1994.

[14] Deutsche Industrie Normung. DIN-V-VDE-0801, grundsätze für rechner in systemen (principles for computers in safety), 1994.

[15] J.R. Dunham. Experiments in software reliability: life critical applications. *IEEE Transactions on Software Engineering*, Vol SE-12(1):110–23, 1986.

[16] D.E. Eckhardt, A.K. Caglayan, J.C. Knight, L.D. Lee, D.F. McAllister, M.A. Vouk, and J.P.J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transaction on Software Engineering*, 17(7):692–702, July 1991.

[17] D.E. Eckhardt and L.D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–17, December 1985.

[18] I. Gashi. *Software Dependability with Off-The-Shelf Components*. PhD thesis, City Univerisity, Centre for Software Reliability, 2007.

[19] I. Gashi, P. Popov, V. Stankovic, and L. Strigini. On designing dependable services with diverse off-the-shelf SQL servers. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems II*, volume 3069 of *Lecture Notes in Computer Science*, pages 191–214. Springer Verlag, 2004.

[20] I. Gashi, P. Popov, and L. Strigini. Fault diversity among off-the-shelf SQL database servers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '04)*, pages 389–98, Florence, Italy, 2004. IEEE Computer Society Press.

[21] C. Gierl, P. Popov, and L. Strigini. The DISPO-2 experiments - specification of the test harness and experimental set-up. Technical Report CG-DISPO2-01, Centre for Software Reliability, City University, 2003.

[22] T. Gilb. Parallel programming. *Datamation*, 20(10):160–1, October 1974.

[23] T. Gilb. *Distinct software: a redundancy technique for reliable software*, pages 117–33. InfoTech, 1977.

[24] L. Gmeiner and U. Voges. Software diversity in reactor protection systems: an experiment. In R. Lauber, editor, *Safety of computer control systems*, New York, 1980. Pergamon.

[25] D. Greathead and A. Devito Da Cunha. Code review and personality: The impact of MBTI type (fast abstract). In *Proceedings of the Fifth European Dependable Computing Conference (EDCC-5)*, volume 3463 of *Lecture Notes in Computer Science*, pages 73–4, Budapest, Hungary, April 2005. Springer.

[26] M.H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.

[27] L. Hatton. *Safer C: Developing Software for High-Integrity and Safety-Critical Systems.* McGraw-Hill, 1995.

[28] L. Hatton. Are N average software versions better than 1 good version? *IEEE Software*, 14(14):71–6, 1997.

[29] L. Hatton and A. Roberts. How accurate is scientific software? *IEEE Transactions on Software Engineering*, 20(10):785–97, 1994.

[30] H. Holscher and J. Rader. *Mikrocomputer in der Sicherheitstechnik (Microcomputers in Safety Technique).* Verlag TÜV-Rheinland, 1984.

[31] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell. Program structure for error detection and recovery. *Proc. Conf. Operating Systems: Theoretical and Practical Aspects, IRIA, April 23-25, 1974*, pages 177–193, 1974.

[32] International Electrotechnical Commission. IEC61508, functional safety of E/E/PES systems.

[33] J.P.J. Kelly. Specification of fault-tolerant multi-version software: Experimental studies of a design diversity approach. Technical Report CSD-820927, UCLA Computer Science Department, Los Angelos, California, September 1982.

[34] J.P.J. Kelly and A. Avižienis. A specification-oriented multi-version software experiment. In *Thirteenth International Symposium on Fault Tolerant Computing (FTCS-13)*, Milan, Italy, June 1983.

[35] J.C. Knight and N.G. Leveson. An empirical study of failure probabilities in multi-version software. In *16th International Symposium on Fault-Tolerant Computing (FTCS-16)*, pages 165–70, Vienna, Austria, 1986.

[36] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, 1986.

[37] B. Littlewood and D.R. Miller. Conceptual modelling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering*, 15(2):1596–614, December 1989.

[38] B. Littlewood, P.T. Popov, and L. Strigini. Assessment of the reliability of fault-tolerant software: a bayesian approach. In F. Koornneef and M.J.P. van der Meulen, editors, *Proceedings of the 19th international conference on Computer Safety, Reliability and Security, Safecomp 2000*, volume 1943 of *Lecture Notes in Computer Science*, pages 294–308, Rotterdam, Netherlands, 2000. Springer.

[39] B. Littlewood and L. Strigini. A discussion of practices for enhancing diversity in software designs. Technical Report LS-DI-TR-04, Centre for Software Reliability, City University, 2000.

[40] M. Lyu, J. Chen, and A. Avižienis. Software diversity metrics and measurements. In *Proceedings of the Sixteenth Annual International Computer Software and Applications Conference*, pages 69–78, 1992.

[41] M.R. Lyu and Y. He. Improving the N-version programming process through the evolution of a design paradigm. *Proc. IEEE Transactions Reliability*, 42(2):179–89, June 1993.

[42] D.J. Martin. Dissimilar software in high integrity applications in flight controls. In *Proceedings of AGARD Symposium on Software for Avionics*, 1982.

[43] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), December 1976.

[44] D. McKenzie. Computer-related accidental death: an empirical exploration. *Science and Public Policy*, pages 233–48, August 1994.

[45] M.J.P. van der Meulen. On the use of smart sensors, common cause failure and the need for diversity. In *Proceedings of the 6th International Symposium on Programmable Electronic Systems in Safety Related Applications*, TUV, Cologne, 2004.

[46] M.J.P. van der Meulen, P.G. Bishop, and M. Revilla. An exploration of software faults and failure behaviour in a large population of programs. In *Proceedings of the 15th IEEE International Symposium of Software Reliability Engineering*, pages 101–12, St. Malo, France, November 2004.

[47] M.J.P. van der Meulen and M.A. Revilla. The effectiveness of software diversity in a large population of programs. *To be published.*

[48] M.J.P. van der Meulen and M.A. Revilla. Experiences with the design of a run-time check. In J. Gorski, editor, *Proceedings of the 24th international conference on Computer Safety, Reliability and Security, Safecomp 2006*, volume 4166 of *Lecture Notes in Computer Science*, pages 302–15, Gdansk, Poland, 2006. Springer.

[49] M.J.P. van der Meulen and M.A. Revilla. Internal software metrics and software dependability in a large population of small C/C++ programs. In *Proceedings of the 18th IEEE International Symposium of Software Reliability Engineering*, pages 203–8, Sweden, November 2007.

[50] M.J.P van der Meulen, S. Riddle, L. Strigini, and N. Jefferson. Protective wrapping of off-the-shelf components. In X. Franch and D. Port, editors, *Proceedings of the 4th International Conference on COTS-Based Software Systems (ICCBSS '05)*, volume 3412 of *Lecture Notes in Computer Science*, pages 168–77, Bilbao, Spain, 2004. Springer.

[51] M.J.P. van der Meulen, L. Strigini, and M.A. Revilla. On the effectiveness of run-time checks. In B.A. Gran and R. Winter, editors, *Proceedings of the 23rd international conference on Computer Safety, Reliability and Security, Safecomp 2005*, volume 3688 of *Lecture Notes in Computer Science*, pages 151–64, Fredrikstad, Norway, September 2005. Springer.

[52] M.J.P. van der Meulen and M.A. Revilla. The effectiveness of choice of programming language as a diversity seeking decision. In *Proceedings of the Fifth European Dependable Computing Conference (EDCC-5)*, volume 3463 of *Lecture Notes in Computer Science*, pages 199–209, Budapest, Hungary, April 2005. Springer.

[53] P. Neumann. *Computer-Related Risks.* ACM Press, 1994.

[54] A. Jefferson Offutt. A practical system for mutation testing: Help for the common programmer. In *International Test Conference*, pages 824–30, 1994.

[55] A. Jefferson Offutt, Jeff Voas, and Jeff Payne. Mutation operators for Ada. Technical Report ISSE-TR-96-09, Department of Information and Software Systems Engineering, George Mason University, Fairfax, Virginia, USA, March 1996.

[56] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1:220–232, 1975.

[57] U.-D. Reips. Internet-based psychological experimenting; five dos and five don'ts. *Social Science Computer Review*, 20(3):241–49, 2002.

[58] P. Runeson, M. Holmstedt Jöhnsson, and F. Scheja. Are found defects an indicator of software correctness? An investigation in a controlled case study. In *The 15th IEEE International Symposium of Software Reliability Engineering, 2–5 November 2004, St. Malo, France*, pages 91–100, 2004.

[59] K. Salako and L. Strigini. Diversity for fault tolerance: effects of "dependence" and common factors in software development. Technical Report

KS-DISPO5-01, Centre for Software Reliability, City University, August 2006.

[60] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 76–85, June 2003.

[61] S. Skiena and M. Revilla. *Programming Challenges.* Springer Verlag, March 2003.

[62] L. Strigini. DISPO-2 fault injection experiments with DARTS software: findings. Technical Report LS-DISPO2-06, Centre for Software Reliability, City University, December 2003.

[63] Unknown. *Software Metrics.* Studentlitteratur, 1976.

[64] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. In *International Symposium on Software Testing and Analysis*, pages 139–48, 1993.

[65] U. Voges, editor. *Software Diversity in Computerized Control Systems.* Springer Verlag, Wien, 1988.

Names Index

This index contains the pages on which the name itself occurs (page numbers in bold) or where a citation occurs in which the person is (co-)author. There are no references to the reference lists in the main text or in the publications.

Agrawal, Hiralal, 34

Aidemark, Joakim, 178, 193

Aiken, Alex, 67

Amman, Paul, 160

Anderson, Tom, **18**, **35**, **177**, **193**

Arlat, Jean, 155

Avižienis, Algirdas, 19, 28, 29, **127**, 214

Badger, Lee, 155

Barnes, Mel, 29, 127, 214

Barrett, Peter, 35

Beizer, Boris, 150

Bentley, Julian, **55**, 149

Beurden, Iwan van, 114, 118

Birnbaum, Michael, **64**

Bishop, Peter, 28, **28**, 29, 127, 149, 214, 215

Blum, Manuel, 177, **180**, 193

Brilliant, Susan, **30**, 165

Caglayan, Alper, 29, 165, 170, 214

Canning, James, 207

Carlyle, Thomas, **27**

Cha, Stephen, 178, 193

Cheung, Steven, 155

Clark, N., 112, 114, 117

Conan Doyle, Sir Arthur, **77**

Dahll, Gustav, 28, 29, 127

Devito Da Cunha, Alessandra, **72**

Diderot, Denis, **55**

Dobbing, A., 112

Dunham, Janet, 29

Eckhardt, Dave, 19, 29, 35, **58**, **127**, **129**, **135**, **148**, 165, **168**, **170**, **214**, **215**, **218**

Einstein, Albert, **63**

Emerson, Ralph Waldo, **41**

Esp, David, 29, 127, 214

Fabre, Jean-Charles, 155

Feldman, Mark, 155
Fenton, Norman, **207**
Fetzer, Christof, 155, 161, 177, 193
Folkesson, Peter, 178, 193
Frazer, Timothy, 155

Gärtner, Felix, 177, 193
Gashi, Ilir, **32**
Ghosh, Anup, 155
Gierl, Claude, 34
Gilb, Tom, 19
Gmeiner, Lothar, 19, 28, 29, 214
Goble, William, 114, 118
Godfrey, David, 112
Greathead, David, **72**
Grebe, John, 114, 118

Halliwell, Dave, 35
Halstead, Maurice, **46**, **207**, **209**
Harris, Peter, 112, 114, 117
Hatton, Les, **17**, **30**, 31, 165
He, Y., 29, 214
Hill, Frank, 155
Horning, James, 35
Humphreys, Peter, 29, 127, 214

Jefferson, Nigel, **56**
Jhumka, Arshad, 177, 193

Kafura, Dennis, 207
Kamruzzaman, Md., **194**
Karlsson, Johan, 178, 193
Kelly, John, 28, 29, 127, 165, 170, 214
Knight, John, 19, **28**, 29, **30**, 49, 77, **127**, 150, 160, 165, 170, 178, 193, 214
Koga, Shuichi, 117

Lahti, J., 29
Lauer, Hugh, 35
Lee, Larry, 19, 29, 35, **58**, **127**, **129**, **135**, **148**, 165, **168**, **170**, **214**, **215**, **218**
Lee, Peter, **18**, **177**, **193**
Leveson, Nancy, 19, **28**, 29, **30**, 49, 77, 117, **127**, 150, 165, 178, 193, 214
Levitt, Karl, 155
Littlewood, Bev, **13**, **20**, 34, 37, **37**, **58**, **127**, **129**, **148**, 150, 165, **168**, **170**, **214**, **215**
Lyu, Michael, 29, 150, 214

Madeira, Henrique, 178, 193
Martin, D., 19
McAllister, David, 29, 165, 170, 214
McCabe, Thomas, **46**, **207**, **209**
McKenzie, Donald, **17**
Melliar-Smith, Mike, 35
Meyer, Bertrand, 157, 177, 193
Miller, D. Richard, 37, **37**, **58**, **127**, **129**, **148**, 165, **168**, **170**, 214, 215
Montaigne, Michel Eyquem de, 17
Moulding, Michael, 35

Neill, Martin, 207

Neumann, Peter, **17**

Offutt, A. Jefferson, 34

Parkin, Graeme, 112
Pfleeger, Shari, 207
Pinnel, L. Denise, 117
Popov, Peter, 34, 127, 166, 188, 203
Pullum, Laura, 119

Randell, Brian, 35
Reese, Jon Damon, 117
Reips, Ulf-Dietrich, **64**, **65**
Rela, Mário Zenha, 178, 193
Renaud, Brian, 46
Revilla, Miguel, 21, **21**, **56**, 66, **66**, **98**, 128, **141**, **166**, **178**, **194**, **207**, **214**, 215, 219
Riddle, Steve, **56**
Romanovsky, Alexander, 166
Runeson, Per, **29**, 211
Russell, Mark, **71**

Salako, Kizito, **20**
Salles, Frédéric, 155
Sandys, Sean David, 117
Schütz, Werner, 29
Schleimer, Saul, 67
Schmid, Matt, 155
Schutz, Werner, 214
Shimeall, Timothy, 178, 193
Silva, João Gabriel, 178, 193
Skiena, Steven, 21, 66, **98**, 128, 141, 166, 178, 194, 207, 214
Stevens, Mark, 112
Strigini, Lorenzo, **13**, **20**, 34, **35**, **38**, **56**, 127, 166, 188, 203
Suri, Neeraj, 177, 193

Untch, Roland, 34

Vinter, Jonny, 178, 193
Voges, Udo, 29, 165, 214
Vouk, Mladen, 29, 165, 170, 214

Wasserman, Hal, 177, 180, 193
Wichmann, Brian, 112, 114, 117
Wilkerson, Daniel, 67

Xiao, Zhen, 155, 161

BRITISH LIBRARY

IMAGING SERVICES NORTH

Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

MISSING PAGE/PAGES
HAVE NO CONTENT

Appendix A

Problems Analysed

A.1 Properties of the Specifications

Nr = Number on the Online Judge website, #Prog = Number of programs submitted to this problem, #Aut = Number of authors that submitted at least one program to the specification, Diff = Difficulty, the percentage of authors not submitting a correct program on the first attempt; and for the correct programs: LOC = Average Lines of Code; Vol = Average Halstead Volume, CC = Average Cyclomatic Complexity.

Nr	Name	#Prog	#Aut	Diff	LOC	Vol	CC
00100	3n+1	38,702	13,575	59	36.9	1102	9.5
00102	Ecological Bin Packing	18,283	6,264	36	50.7	2252	11.8
00106	Fermat vs. Pythagoras	4,658	1,348	55	63.8	2431	16.2
00108	Maximum Sum	7,950	3,747	56	43.1	1696	13.7
00116	Unidirectional TSP	6,160	1,907	72	76.6	3482	21.4
00145	Gondwanaland Telecom	2,944	979	64	72.8	3448	17.7
00147	Dollars	3,294	925	89	36.8	1847	8.9
00149	Forests	500	160	69	117.2	5222	25.8

Nr	Name	#Prog	#Aut	Diff	LOC	Vol	CC
00160	Factors and Factorials	7,104	3,112	42	59.5	2379	15.1
00191	Intersection	5,703	1,563	82	73.6	3570	17.9
00231	Testing the Catcher	3,722	1,514	69	44	1336	11.3
00253	Cube Painting	1,935	887	56	68.4	2955	16.9
00271	Simply Syntax	2,193	1,044	35	45.1	1285	15.3
00344	Roman Digititis	3,449	2,116	34	68.7	2582	19.2
00374	Big Mod	3,922	2,007	43	29.8	864	7.2
00402	M*A*S*H	3,088	1,174	69	49.5	1444	13.7
00534	Frogger	2,394	1,052	40	55.9	2381	14.5
00558	Wormholes	1,371	633	62	58.7	2135	14.6
00568	Just the Facts	3,341	1,852	70	41.2	3548	7.7
00572	Oil Deposits	2,635	1,623	15	58.6	2411	16.8
00591	Box of Bricks	8,299	3,643	33	28.4	776	6.4
00594	One Little, Two Little, Three Little Endians	1,649	996	33	27.7	873	5.5
00602	What Day Is It?	4,362	749	67	99.6	3980	33.1
00612	DNA Sorting	6,154	1,773	64	50.3	1794	12.8
00623	500!	2,731	913	58	80.3	3165	17.2
00637	Booklet Printing	1,928	918	74	49.5	1502	15.8
00674	Coin Change	3,067	1,245	28	138.8	10007	7.1
00686	Goldbach's Conjecture (II)	4,042	2,162	28	54.6	3923	11.5
00696	How Many Knights	2,012	648	60	35	1320	11
00701	The Archaeologists' Dilemma	2,158	626	56	45.5	1542	9.5
00713	Adding Reversed Numbers	5,327	2,750	92	42.1	1323	8.7
00748	Exponentiation	1,065	717	82	125.7	4978	31.2

Nr.	Name	#Prog	#Aut	Diff	LOC	Vol	CC
00763	Fibinary Numbers	1,920	501	53	101.1	3805	25.1
00808	Bee Breeding	1,104	381	40	145.2	6671	19
00861	Little Bishops	725	208	53	85	4647	19.2
10003	Cutting Sticks	2,027	972	26	41.4	1623	10.7
10035	Primary Arithmetic	8,683	2,846	55	43.7	1326	12
10038	Jolly Jumpers	6,152	2,242	60	34.6	969	10.2
10042	Smith Numbers	2,506	748	57	80.5	5248	17.3
10061	How Many Zero's and How Many Digits?	1,492	411	67	63	2216	14.4
10083	Division	609	223	52	156.5	5945	33
10104	Euclid Problem	1,788	794	38	37.9	1154	7.2
10106	Product	4,921	1,615	37	89	3537	20.1
10116	Robot Motion	1,506	840	29	60.1	2107	16.6
10127	Ones	2,246	1,509	38	31.1	2220	5.4
10130	Super Sale	1,127	469	29	46.1	1834	11.7
10139	Factovisers		865				
10157	Expressions	391	148	89	100.7	4906	26.6
10161	Ant on a Chessboard	1,980	1,140	20	42.4	1087	11.5
10162	Last Digit	1,340	679	39	44.9	1781	11
10176	Ocean Deep! Make it Shallow!!	1,096	491	31	33.3	902	9.6
10183	How many Fibs?	1,554	627	44	128.4	5192	29.4
10200	Prime Time	2,056	1,527	20	77.8	7890	11.2
10220	I Love Big Numbers!	1,873	1,282	26	107.3	5662	18.5
10235	Simply Emirp	3,411	1,123	86	232	20430	14.3
10271	Chopsticks	545	209	32	47.6	2181	11.3
10285	Longest Run on a Snowboard	872	451	24	60.2	2762	16.6

Nr	Name	#Prog	#Aut	Diff	LOC	Vol	CC
10299	Relatives	1,399	498	69	61.3	3668	13.2
10311	Goldbach and Euler	2,663	421	83	117.6	6580	24.4
10392	Factoring Large Numbers	655	285	23	62.7	2574	15.4
10453	Make Palindrome	566	206	66	73.5	2990	18.6
10848	Make Palindrome Checker						
	Average	3,540	1,382	52	56.2	2548	12.9

The parameters have not been computed for 10848, and they are not included in the calculation of the averages.

A.2 Specifications used in the Publications

Publication	Reference	Specifications
On the use of smart sensors, common cause failure and the need for diversity.	[45], C.1	None
An empirical exploration of the difficulty function.	[6], C.2	100, 10139
An exploration of software faults and failure behaviour in a large population of programs.	[46], C.3	100

Publication	Reference	Specifications
Protective Wrapping of Off-the-Shelf Components.	[50], C.4	None
The Effectiveness of Choice of Programming Language as a Diversity Seeking Decision.	[52], C.5	100, 10139, 10200
On the Effectiveness of Run-Time Checks.	[51], C.6	100, 10139, 10200
Experiences with the Design of a Run-Time Check.	[48], C.7	10453, 10848
Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs.	[49], C.8	100, 102, 106, 108, 116, 145, 147, 149, 160, 191, 231, 253, 271, 344, 374, 402, 534, 558, 568, 572, 591, 594, 602, 612, 623, 637, 674, 686, 696, 701, 713, 748, 763, 808, 861, 10003, 10035, 10038, 10042, 10061, 10083, 10104, 10106, 10116, 10127, 10130, 10157, 10161, 10162, 10176, 10183, 10200, 10220, 10235, 10271, 10285, 10299, 10311, 10392

Publication	Reference	Specifications
The Effectiveness of Software Diversity in a Large Population of Programs.	[47], C.9	100, 102, 106, 108, 116, 145, 147, 149, 160, 191, 231, 253, 271, 344, 374, 402, 534, 558, 568, 572, 591, 594, 602, 612, 623, 637, 674, 686, 696, 701, 713, 748, 763, 808, 861, 10003, 10035, 10038, 10042, 10061, 10083, 10104, 10106, 10116, 10127, 10130, 10157, 10161, 10162, 10176, 10183, 10200, 10220, 10235, 10271, 10285, 10299, 10311, 10392

A.3 Problem Domains

Program Category	Problems
Strings	271, 10453, 10848
Sorting	612, 637
Arithmetic and Algebra	145, **701**, 713, 748, **10035**, 10083, 10106, 10200
Combinatorics	591, 674, 10003, 10130, **10157**, **10183**, 10271
Number Theory	100, 106, 160, 231, 344, 374, 402, 568, 594, 602, 623, 686, 763, 10038, **10042**, 10061, **10104**, 10127, **10139**, 10162, 10183, 10235, 10299, 10311, 10392
Grids	108, 116, 572, 696, 808, 861, **10161**, 10285
Geometry	149, 191, 253, 534, 558, 10116

Problems in bold have been categorized by Skiena and Revilla in [61].

Appendix B

Problem Analysis

This appendix contains a description of the tools and their use for analyzing a single problem.

B.1 Directory Structure

Table B.1 presents a schematic representation of the directory structure for conducting the experiments.

`PC/` (for Programming Contest) is the root directory and contains everything for conducting the experiments.

`Archive/` contains the E-mails of all submissions in 2547 zipped files; each of these files contains 1000 submissions and has a name in the range `00000.TGZ` ... `02547.TGZ`. The submissions have names 00000000 ... 02547979 (which are the IDs of the submission, see below). The file History.txt contains a line of information for the submissions 00000000 ... 02545979, its state in May 2004. Table B.2 contains a description of the information available for every submission.

`Utils/` contains utilities useable for all problems.

Every experiment concerns one problem in the programming contest, e.g. problem 100 "The 3n+1"-Problem, and has its own directory, in this case `00100_3n+1/`. For every problem, we have to

- extract the E-mails to `Emails/`,

```
PC/ Archive/       *.TGZ
                   History.txt
    Utils/         ExtractID_list
                   ExtractEmailbatch
                   ema2srcBatch
                   src2exeBatch
                   exe2outBatch
                   equ2scoBatch
                   EquList
                   ScoreList
                   CreateDatabase
                   GenerateGraphs.r
                   killafter
    00100_3n+1/    ID_list.txt
                   ProblemList.txt
                   Emails/           EmailIDs.txt
                                     *.ema
                   Sources/          *.c
                                     *.pas
                   Execs/            *.exe
                   Scripts/
                   Output1/          TestBatch.in
                                     MakeInputFile
                                     input.txt
                                     OutputScript
                                     OutputScriptPerl
                                     Normalize
                                     *.out
                                     EquList.txt
                                     *.score
                                     ScoreList.txt
                                     Results/         Database.r
                                                      HomDiv.ps
                                                      FirstC.ps
                                                      FirstC++.ps
                                                      FirstPascal.ps
                   Other output
                   directories
    Other problem
    directories
```

Table B.1: Schematic representation of the directory structure for conducting the experiments, including the location of some key files and scripts.

Field	Format	Description
ID	8 digits	A unique ID given to each submission.
DATE	17 digits	The date and time of the submission in the format YYYYMMDDhhmmssmmm.
ST	2 chars	The status of the submission, the most important ones being: AC=Accepted; WA=wrong Answer; PE=Presentation Error; TL=Time Limit exceeded; ML=Memory Limit exceeded; CE=Compilation Error.
USER	5 digits	A unique number given to a user.
PROBL	5 digits	A unique number for each problem in the programming contest.
SOURCE	6 chars	The language of the submission: C, C++, PASCAL or JAVA.
MAXMEM	not used	
CPU	not used	
SG	not used	
ALGORITH	not used	

Table B.2: Fields in the file History.txt.

- extract the program submissions from the E-mails to `Sources/`,
- compile the programs and store the executables in `Execs/`,
- run the various experiments and store the results in `Output1/`, `Output2/`, etc.,
- and finally, generate graphs and results for each experiment, and store these in `Results/`.

The `Scripts/` directories contain scripts specific for a problem, e.g. for generating score functions when determining whether an answer is correct requires more than a simple comparison. For most problems, there are no special

scripts because the experiment can be done completely using the generic scripts in `Utils/`.

B.2 Extracting E-mails

The script `ExtractID_list` in `Utils/` extracts the lines from History.txt concerning the problem to the file `ID_list.txt` in the problem's main directory. It takes the problem number as its argument.

The script ExtractEmailBatch in `Utils/` first determines the IDs of submissions written in C, C++ or Pascal, and with status AC, WA and PE, and writes these to the file `Emails/EmailIDs.txt`. Then the script extracts all these submissions to `Emails/` (using the script ExtractEmail, which extracts one submission only to the current directory). We use the submissions with status AC, WA and PE only, because the other submissions have already been classified by the Online Judge as being incorrect in ways outside our range of interests. We are not interested in programs that cannot be compiled, or exhibit major problems while being executed. We do not extract Java programs, because we have as yet not implemented the functionality to process these.

B.3 Extracting Source Code

The script `ema2srcBatch` in `Utils/` extracts source code from the emails in `Emails/`, using the Pascal program `ema2src`, which does this for one source only. The script adds the extension .c to C and C++ programs, and .pas to Pascal programs.

Extracting source code from the emails proved to be more difficult than it seemed. In later submissions, the users had to adhere to a prescribed format, which forced them to use keywords—`@BEGIN` and `@END`—in their E-mail; these keywords can be easily detected. In early submissions however, we have to detect the beginning and end of programs in different ways. This is easiest for programs written in Pascal, since these almost always start with the keyword "`program`", and end with "`end.`"; of course there are programs that violate

this rule. For C and C++ programs this is more difficult. Programs may start in many different ways; an "`#Include`" is one of the best indicators of the beginning, but this is certainly not conclusive.

The program `ema2src` does the job rather well, but not always. We can detect mistakes, because compiling damaged sources will almost inevitably fail. `ProblemList.txt` contains the IDs of these sources. The extraction then needs to be done manually; fortunately, this is an easy task and does not happen very often.

B.4 Compiling Source Code

The script `src2exeBatch` compiles the programs in `Sources/`, using the list of IDs in `Emails/EmailIDs.txt`.

For C and C++ programs, it first tries gcc version 2.95.3, then g++ version 2.95.3, and finally the gcc version installed on the computer (in our case version 3.4.2). We use version 2.95.3 because this is prescribed by the Programming Contest. Later versions of gcc can often not compile programs written for version 2.95.3.

For Pascal, the script first tries gpc, and then fpc. The Programming Contest used to prescribe fpc, but later changed its preference to gpc, so we try both.

The script writes the IDs of sources for which no executable is generated to `ProblemList.txt`.

The script adds the extension .exe to the executables. This is done to be able to differentiate executables from E-mails, which do not have an extension.

B.5 Running the Programs

Running the programs requires the collaboration of several scripts.

The first thing to be done is to determine the demands to the programs. These will be stored in `input.txt`. If we use a program to generate the input file, we call it `MakeInputFile` with appropriate extension if necessary. This can

either be a compiled program, or a script.

The main script is exe2outBatch in Utils/, taking the name of the output directory as its argument, e.g.: exe2outBatch Output1.

exe2outBatch uses OutputScript and OutputScriptPerl in the output directory. These are scripts controlling the execution of a single program. These scripts are specific to a problem, since every problem may pose different challenges. In general the scripts will not need big changes. A difference may occur for the time a program is allowed to use to compute the answer. The utility used for this purpose is killafter in Utils/; it terminates programs that take too long to complete.

exe2outBatch also uses the script Normalize in the output directory. This is a script that normalizes the output of the programs. This may be unnecessary in which case the script will either be empty or absent. In most cases the only task of Normalize is to remove empty lines, repeated spaces, or spaces at the end of lines. It may also convert the output to lower case, or remove characters. This is to cope with the fact that outputs may be slightly different whilst still being within the boundaries of the specification. We want these to be exactly the same.

OutputScriptPerl uses the Pascal program EquList to determine the equivalence classes of the outputs of the programs. Initially the output of a program is written to a $ID.out file; EquList checks whether an earlier program has produced exactly the same output: this is called an equivalence class. If so, EquList removes the output of the latter. The file EquList.txt contains the mapping between IDs and the equivalence classes.

After running exe2outBatch, the output directory contains *.out files. These files contain the outputs of the programs in all of the equivalence classes. The file EquList.txt contains a mapping between the IDs of the programs and the equivalence classes.

(OutputScriptPerl converts the equivalence classes to score functions. These files contain a line with a 0 for each correct output and a 1 for each incorrect output. equ2sco takes three arguments: the output directory, the ID of the

correct equivalence class and the ID of the equivalence class for which the score function is to be generated.

Finally, the script `ScoreList` computes equivalence classes in the score functions. It stores a mapping between matching score functions in `ScoreList.txt`.

B.6 Generating Standard Graphs

We use R to generate graphs. However, R is very slow with loops, and therefore we wrote a Pascal program `CreateDatabase`, to do the preprocessing. The program does the following:

- It takes the `ID_list.txt` file and sorts it; first on author, then on ID.
- It selects the lines with IDs in `TestBatch.in`.
- It adds a field Equiv, which contains the ID of the equivalence class.
- It adds a field Score, which contains the ID of the equivalent score function.
- It adds a field NextEquiv, the ID of the equivalence class of the next submission of the same author. If it's the last submission of an author, the ID is 99999999.
- It adds a field NextScore, the ID of the equivalent score function of the next submission of the same author. If it's the last submission of an author, the ID is 99999999.
- It adds a field Trial, which numbers the submissions of each author, starting with 1.
- It adds a field BeyondOK. BeyondOK is True when an earlier submission of the same author is correct.
- It adds a field TrialsLeft, which numbers the number of trials left, starting with 0 for the last submission or the first correct submission. If BeyondOK is true, TrialsLeft is zero.

It stores the resulting database in `Database.r` in `Results/`.

Now, it is possible to generate a set of graphs using the statistical package R and the program `GenerateGraphs.r` in `Utils/`. This program gives several options to make graphs, the main outputs are:

- A graph for homogeneous diversity.
- Graphs for language diversity.

B.7 Example Session

The following gives the sequence of commands for a normal experiment, starting in the directory `PC/`:

```
$ mkdir 00160_FactorsAndFactorials
$ cd 00160_FactorsAndFactorials
$ ../Utils/ExtractID_list 160
Finding solutions for problem 160... 8623 solution(s) found
$ ../Utils/ExtractEmailBatch
Finding subset of ID_list.txt... 7105 solution(s) found
Extracted 00000462.ema.
...
Extracted 02545969.ema.
Extracted emails in EmailIDs.txt to Emails/.
$ ../Utils/ema2srcBatch
First ID to evaluate is 0
00000462    C
...
02545969    C++
Extracted sources from emails in Emails/ and saved these to Sources/.
$ ../Utils/src2exeBatch
00000462
...
02545969
Compiled sources in Sources/ and saved executables in Execs/.
$ mkdir Output1
```

```
$ perl Output1/MakeInputFile.perl  # --> See example of perl script below.
$ cp Emails/EmailIDs.txt Output1/TestBatch.in
$ exe2outBatch Output1
00000462
               Normalizing...
                              Find equivalence class...
Equivalent: 00000462 00000462
00000748
               Normalizing...
                              Find equivalence class...
...
All programs executed and outputs written to Output1.
$ equ2scoBatch Output1 00000462
Directory: Output1, Benchmark: 00000462
00000462
...
02544837
Score functions generated and written to Output1.
$ ../Utils/ScoreList Output1
00000462
...
02544837
ScoreList.txt generated and written to Output1.
$ ../Utils/CreateDatabase Output1 462 -v
Output directory: Output1
Directory Results/ already exists.
Benchmark: 00000462.
Reading ID_list.txt.
Reading TestBatch.in.
Reading Output1/Equlist.txt.
Reading Output1/ScoreList.txt.
Numbering trials.
Numbering trials left.
Assigning equivalence and score classes for the next submissions.
Determining filesizes.
```

```
Determining unreliabilities.
Database saved to Output1y/Results/Database.r.
$ R
> load("../Utils/GenerateGraphs.r")
> GenerateGraphs()
Give the name of the directory of the output files [Output1]:
Give the name of the problem, e.g. '10083, Division': 00160, Factorials
[1] You can choose from the following options:
[1] 1. Generate graph for homogeneous diversity.
[1] 2. Generate graphs for language diversity.
[1] 3. Generate graph for special single property.
[1] 4. Generate graph for special multiple property.
[1] 5. Generate graph for filesize.
[1] 7. Change selection of submissions.
[1] 8. Change settings.
[1] 9. Exit
Give your option: 1
[1] Now generating graph for homogeneous diversity.
[1] 1e-04
...
[1] 0.7943282
[1] 1
[1] You can choose from the following options:
[1] 1. Generate graph for homogeneous diversity.
[1] 2. Generate graphs for language diversity.
[1] 3. Generate graph for special single property.
[1] 4. Generate graph for special multiple property.
[1] 5. Generate graph for filesize.
[1] 7. Change selection of submissions.
[1] 8. Change settings.
[1] 9. Exit
Give your option:
...
```

Appendix C

Publications

C.1 TÜV Symposium 2004

M.J.P. van der Meulen. On the use of smart sensors, common cause failure and the need for diversity. In *Proceedings of the 6th International Symposium on Programmable Electronic Systems in Safety Related Applications*, TUV, Cologne, 2004.

On the use of smart sensors, common cause failure and the need for diversity

Meine van der Meulen

City University, Centre for Software Reliability
mjpm@csr.city.ac.uk,
WWW home page: http://www.csr.city.ac.uk

Abstract

The use of smart sensors in highly critical (safety) applications is still being debated. In this paper, we compare the dependability aspects of deploying smart sensors vs. conventional ones using an FMEA. There appear to be some significant differences. Some failure modes do not exist in conventional sensors, e.g. those involving information overload and timing aspects. Other failure modes emerge through the use of different technologies, e.g. those involving complexity, data integrity and human interface. When using smart sensors we suggest the use of a set of guidelines for their deployment:

1. Do not send data to the smart sensor.
2. Use the smart sensor in burst mode only.
3. Use a smart sensor with the least possible number of operational modes.
4. Use the simplest possible sensor for the application.

In redundant sensor configurations common cause failure becomes the dominant failure scenario. The failure modes of smart sensors suggest that smart sensors might be more susceptible to common cause failure than conventional ones. Dominant are failures having their origin in the human interface, complexity and information overload. The guidelines given will also reduce the probability of common cause failure.

In redundant sensor configurations a possible design method is the use of diversity. Diversity has the advantage that it can reduce the probability that two or more sensors fail simultaneously, although this effect is limited by the fact that diverse sensors may still contain the same faults. A disadvantage of diversity can be the increased complexity of maintenance, which in itself can lead to a higher probability of failure of the smart sensors. Whether the use of diversity is advisable depends on the design of the smart sensors and the details of their application.

1 Introduction

The last decade has seen a massive increase in the use of smart sensors. On the whole, users of smart sensors appear satisfied, focusing on their advantages,

such as the higher accuracy made possible through better signal processing and the use of digital communication. It is argued that the quality of smart sensors is now comparable to conventional sensors, and that their use in highly critical (safety) functions can be justified. The question we will address is whether this assertion is true or false.

For highly critical functions redundant configurations of sensors are normally being used, for example 2-out-of-3. In redundant configurations, the achievable failure rate or the probability of failure on demand is normally limited by common cause failure[1]. Therefore, when using smart sensors, the question is whether common cause failure is a general concern.

If common cause failure is a concern, a second question one might ask is whether the use of diverse[2] sensors might mitigate this.

In this paper we will investigate the reasons why smart sensors fail and compare this with those for conventional sensors.

1.1 Research

The amount of published research in the area of dependability of smart sensors is limited. Particularly relevant is the work produced in the framework of the Software Support for Metrology (SSFM) project ([2], [3] and [4]). The work resulted in some guidelines for the use of smart sensors. It also incorporated an analysis of the software of a specific flow sensor, manufactured by Druck. We are not aware of any other publications in the field.

1.2 Standards

At the moment, IEC61508 [5] and IEC61511 [6] are the most important standards regarding the use of software in the process industry. The standards also address sensor software, even though its inclusion in the introduction of IEC61508 is rather indirect:

*In most situations, safety is achieved by a number of protective systems which rely on many technologies (for example mechanical, hydraulic, pneumatic, electrical, electronic, programmable electronic). Any safety strategy must therefore consider not only all the elements within an individual system (for example **sensors**, controlling devices and actuators) but also all the safety-related systems making up the total combination of safety-related systems. Therefore, while this standard is concerned with electrical/electronic/programmable electronic (E/E/PE) safety-related systems, it may also provide a framework within which safety-related systems based on other technologies may be considered.*

[1] Common cause failure: Failures of multiple items occurring from a single cause which is common to all of them [1].

[2] Diversity: existence of different means of performing a required function [6]. We will use the word diversity in the sense that we address smart sensors of different manufacturers or sensors for which a convincing case can be made that they contain different software, e.g. because they are built by different parts of a company.

As opposed to this, IEC61511 is very clear in its introduction:

This international standard addresses the application of safety instrumented systems for the Process Industries. The safety instrumented system includes sensors, logic solvers and final elements.

So the requirements for software in both these standards also apply to the sensor software.

We will not copy all software requirements here, but list the following topics as an illustration:

1. Software configuration management ([5], part 3, §6.2.3).
2. Documentation of all activities in the safety life cycle of the software ([5], part 3, §7.1.2 and Table 1).
3. There are extensive requirements for all activities in the safety life cycle and the design of the software.
   - Specification methods.
   - The use of formal methods.
   - Error detection and correction.
4. Etc. etc.

The effort that manufacturers are obliged to invest in demonstrating the compliance of their software to these standards is substantial, and many might argue it prohibitive. On the other hand, these standards present the state-of-the-art and it will be hard for manufacturers to assert that their software meets appropriate standards while not complying.

For the current generation of smart sensors the situation is that most (all?) of them do not conform to the requirements as set in these standards. There is a simple reason: IEC 61508 is approved in 1999/2002, IEC 61511 has just been approved; most smart sensors on the market predate both. Manufacturers will certainly adhere to these standards for future developments and they will have their sensors certified by independent organisations like TÜV. Given the amount of software in smart sensors (some contain as little as two kilobytes of assembler code) this should be an achievable target. Meanwhile many smart sensors enjoyed so much use that a "proven in use"-argument can be acceptable.

At the moment the two IEC standards are worthwhile mentioning, one of these is still in its draft phase. The first, IEC62098 [9], covers evaluation methods for microprocessor-based instruments. The second, IEC60770-3 [7] gives guidelines for the evaluation of intelligent transmitters, including fault-injection testing. We use the generic models and their description as proposed in these draft standards for our analyses.

2 Independent assessment

A problem that remains is the confidentiality of information. Manufacturers do not want to disclose details on the design of their sensors. This explains why

reports on failure rates and diagnostic coverage exist[3] of sensors, but that they do not provide detailed design information[4]. These evaluations give the numerical data—most importantly failure rates and fault coverage—required by IEC61508 for the calculations as presented its part 6.

Sometimes independent assessors will perform fault insertion testing. In principle, this is comparable to performing a FMEDA. One advantage is that it concerns real insertion of faults, a disadvantage is that the number of faults that can be tested is lower and that some tests might be catastrophic. Also the distribution of inserted faults might not reflect the distribution in actual use, and thus give a biased assessment of the fault behaviour that is to be expected. At the moment the approach to smart sensor assessment and fault insertion testing is the subject of a new IEC standard [7], now in the draft phase.

The absence of design details makes it hard for users to assess the validity of the results and their applicability to each user's setting. Independent assessors (like EXIDA.com, TÜV, TNO and Factory Mutual) are therefore essential.

However, it appears that none of these assess the dependability of the software in the sensors. They all concentrate on[5]:

1. Functionality of the hardware and the software. Does the smart sensor behave as specified? These analyses do not essentially differ from those of conventional sensors.
2. Failure rate of the hardware.
3. Diagnostic coverage. These analyses are either paper exercises or fault-injection tests.

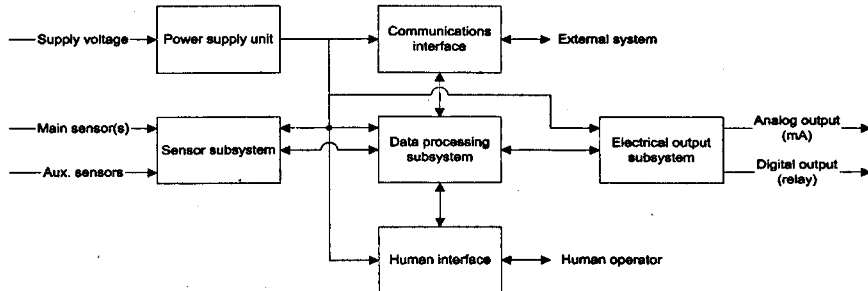
Independent assessment of software in smart sensors has not been done frequently, one example is a project done by NPL [3]. The project didn't lead to any changes becoming necessary in the software (on the other hand: the study identified unreachable code). Assessment of the software appeared to be diffi-

[3] Diagnostic coverage: Fractional decrease in the probability of dangerous hardware failure resulting from the operation of the automatic diagnostic tests [5]. A dangerous failure is defined as: Failure which has the potential to put the safety-related system in a hazardous or fail-to-function state [5]. These two definitions of course pose a problem to the assessor, because he cannot assess which effect failure of a sensor may have in a particular situation.

[4] See for example the reliability evaluations of the Fisher Rosemount 3051C Smart Sensor [10], the Honeywell STT250 temperature transmitter [11], and the Siemens 345 Critical Transmitter [12]. These studies contain a reference to an assessment of the design of the sensor using a FMEDA (=Failure Modes, Effects, and Diagnostic Analysis), and based on that a calculation of the failure rate and the fault coverage. They do not contain design details.

[5] We had a look at the documentation and (confidential) independent analysis of the following smart sensors: Smart Vortex Flowmeter Model 8800 of Rosemount [14], Smart transmitter Model 133DP for differential pressure and Model 131GP for gauge pressure of Foxboro Eckardt [15] [16], Smart multifunction converter Model TSV 175 of Eckardt [17] and Buoyancy transmitters model Eckardt 134 and 144 LVD of Foxboro Eckardt [18] [19].

Fig. 1. Generic model of the design of a smart sensor.



cult as the structure of the software did not permit easy analysis and automatic assessment tools for assembler were not available.

We expect that assessment of the software will become normal practice in the near future when manufacturers start applying IEC61508 and IEC61511 to their smart sensors.

3 Failure of Smart Sensors

To get insight into the specific nature of the failure of smart sensors we performed an FMEA on a generic model of a smart sensor. It should be clear that for the assessment of a specific smart sensor more detailed analyses are necessary. The generic model is depicted in Figure 1, it is based on [7]. The following describes the components of the model:

1. The **data processing subsystem**, whose main function is to provide and process the measured quantity(ies) for further real-time use by the human and communication interfaces and/or at the electrical output subsystem.
2. The **sensor subsystem** converts the physical or chemical quantity(ies) into electrical signals, which are conditioned and digitised for use in the data processing unit.
3. The **human interface** consists of means at the instrument for reading out data (local display) and provisions for entering and requesting data (local pushbuttons).
4. The **communications interface** connects the instrument to external systems.
5. The **electrical output subsystem** primarily converts the digital information into one or more analogue electrical signals. It may also be equipped with one or more binary (digital) electrical outputs.
6. The **power supply unit** supplies power to the subsystems of the smart sensor.

The FMEA (given in Appendix A) considers failure modes of these subsystems and their consequences.

The failures found in the FMEA specific to smart sensors can be classified in a few groups: those concerning time, data integrity, communication, interface, complexity and fault diagnosis. The examples in these subsections are realistic, they have been observed in smart sensors.

3.1 Time

In conventional sensors time and timing are normally not an issue. The sensor continuously measures the parameter and presents a 4 20 mA signal as a result. The time lag between measurement and output is normally negligible.

For smart sensors this is not so straightforward. The sensor has to keep track of time. The following problems might occur:

1. In an architecture based on interrupts the sensor might get many interrupts of high priority, and might subsequently fail to attend to the tasks of lower priority. Example 1: the lower priority task involves updating the sensor's display, and the user gets incorrect reading on the display. Example 2: the lower priority task is the integrating part of the PID function and the sensor's output becomes incorrect. Example 3: To reduce bandwidth requirements of the communication, measurements are not time labeled by the smart sensors, but at the receiver end. Due to prioritisation at the receiver, measurements of different sensors are not time labeled in the correct order, leading to confusion on the cause of a disturbance.
2. In general the theory for discrete signal processing is difficult and many faults can be made in algorithms, aliasing is one of the known problems.

3.2 Data Integrity

Data integrity is not a large problem in conventional sensors. Parameters are set using hardware (resistors for example) and are therefore highly insensitive to external influences. In smart sensors this is more complicated. Parameters are set using buttons and displays and stored in RAM and/or EEPROM. The storage in EEPROM is necessary if the sensor is to maintain the parameters after a power failure. Data in RAM and EEPROM is sensitive to corruption by, for example, radiation and heat, and need error detection and correction. The whole process of dependably storing, retrieving and using data is certainly not trivial. Example 1: A smart sensor looses settings during a power failure. Example 2: A write line to the EEPROM is defective, new settings are written but the memory contents is not checked. The settings are lost.

3.3 Communication

Faults in communication can cause a variety of problems, ranging from lacking or incorrect output, to incorrect parameters and settings. Example: informa-

6

tion overload of the communications interface causes events happening simultaneously to be communicated sequentially, thus confusing the operators when deciding what caused the disturbance.

Most problems that communication can cause, can be solved by the following rules (see also [4]):

1. Wire independent channels separately. This however offsets the advantage of the need of less cabling when using smart sensors nor will it solve all problems: the Logic Unit may treat all sensors equally, and may overload them at the same time.
2. Use sensors in burst mode only.
3. Do not allow on-line reprogramming of sensors.

Conventional sensors use 4–20 mA loops for communication. These loops can be used over very long distances. Digital buses are often not capable of reaching the same distance. In that case, repeaters are necessary, thus increasing the amount of equipment and the possibility of failure.

3.4 Human interface

Some failure modes in smart sensors are related to maintenance and use of the sensor using the human interface. This is caused by:

1. The human interface mainly consists of pushbuttons and displays. Not all information is visible at the same time, this may introduce errors on the information being entered.
2. A smart sensor can be in different modes, modal behaviour is notoriously difficult for users to grasp [8].

These failure modes correlate with those in the next paragraph on complexity.

3.5 Complexity

However simple the functionality of a smart sensor might be, it always consists of highly complex components: integrated circuits. Microprocessors have been shown to contain many faults, most of which are almost unobservable [3].

In principle the functionality of smart sensors could be the same as conventional sensors. In reality this is not the case:

1. The signal processing in smart sensors can involve more elaborate calculations than are possible in conventional sensors. These calculations might for example correct for non-linearity.
2. Smart sensors might have more modes of operation and more parameters than their conventional versions.
3. Smart sensors might measure multiple process variables at the same time.
4. Smart sensors might provide more information than conventional sensors do. Flow sensors for example often contain temperature sensors for improving the accuracy of the measurement. Conventional sensors do not transmit this temperature, some smart sensors however do.

5. Smart sensors contain fault diagnosis, this inevitably leads to complicated algorithms in software and provisions in the hardware of the sensor.

For manufacturers, higher complexity may mean a higher probability of making errors being introduced in the design of the smart sensor. For users, higher complexity may lead to a larger probability of making errors when setting up and maintaining a sensor.

4 Common Cause Failure

The failure rates of smart sensors seem to be comparable to those of conventional sensors[6]. There is no reason to believe that the failure rate of smart sensors will be significantly better, because typically conventional sensors are built using robust components. Smart sensors may contain more vulnerable components and their failure rate might then even be worse.

Given the fact that failure rates are comparable, the difference between the dependabilities of a redundant smart sensor configuration and its counterpart using conventional technology is determined by common cause failure. The question then reduces to: Are smart sensors more prone to common cause failure than conventional sensors? We will first answer the following question: How many of the failure modes addressed in the FMEA are likely to be candidates for common cause failure?

In Table 8 we list the failure modes and assess their likelihood for common cause failure on a relative scale. The table uses the following terms:

1. Information overload. This can occur when the data processing subsystem or the communications interface (other subsystems can also suffer information overload, but this is less likely) have to cope with too many demands. Information overload can largely be avoided by not sending information to a smart sensor and by using it in burst mode only.
2. Core functionality. The core functionality of a smart sensor is thoroughly tested and is therefore least likely to lead to common mode failures.
3. Complexity. See paragraph on complexity above. To reduce the contribution to common mode failure the complexity of a smart sensor should be as low as possible. Also, the environment should whenever possible not use functionality of the smart sensor that is not part of the core functionality. This reduces the probability that possible errors in the software are triggered.
4. HCI. The human computer interaction plays a role in common cause failure. The HCI component can be reduced by making it is simple as possible. The smart sensor should have the least possible number of operational modes and settings.

[6] Some examples: Failure rate of the Honeywell STT250 smart temperature transmitter is calculated to be 3.9×10^{-7}/h [11]. Failure rate of the Fisher-Rosemount 3051C Pressure Transmitter is 7×10^{-7}/h [10]. Both failure rates are for all failure modes.

5. EMI. Electromagnetic interference might cause failure of different smart sensors since the components used might be more susceptible, but it is highly unlikely that both sensors fail the same way. The effects of EMI can be mitigated by proper shielding.

Given the above, there is sufficient reason to believe that common cause failure might be a larger problem for smart sensors. However, there are many ways to mitigate the problem:

1. Do not send data to the smart sensor. No setting of parameters, no change of operational mode, no requests for data.
2. Use the smart sensor in burst mode only. The data the smart sensor produces should be sufficient for all purposes, it should include the necessary information on fault diagnosis, operational mode, settings and parameters. The latter information is necessary to detect possible errors during maintenance.
3. Use a smart sensor with the least possible number of operational modes, settings and parameters. A user can assess this using the documentation.
4. Use the simplest possible smart sensor for the application. This includes signal processing and fault diagnosis. This is very hard to assess for a user, and the opinion of an independent assessor plays an important role.
5. Diversity. See next paragraph.

5 The Case for Diversity

To answer the question whether diversity improves the dependability of redundant smart sensor configurations, we first consider how diversity influences the main causes of common cause failure—information overload, complexity and HCI—as identified in the previous paragraph.

1. Information overload. Diverse sensors might react differently to overload conditions. However, it remains best to not send information to the sensors and to avoid the problem.
2. Complexity. Smart sensors designed for the same use will have comparable design, algorithms and complexity. Although it can not be excluded that different sensors contain the same errors, diversity will reduce the possibility of common cause failure due to complexity (see for example [21]).
3. HCI. Diverse smart sensors will have different human interfaces. Although it is possible that a user makes the same mistake twice, it is less likely.

It is clear that the use of diversity has advantages with respect to dependability. However, there are also disadvantages that have to be considered. First of all, maintaining diverse sensor configurations is more difficult and maintenance personnel has to be trained to maintain different types of equipment. One might even argue that this can lead to more maintenance errors. These errors will increase the failure rate, not the probability of common cause failure. There might also be a larger need for keeping stocks of spare parts, so diversity can increase cost.

The trade-off is hard to make and depends on details of the application. At this stage, it is not possible to give general recommendations for the use of diversity. It is obvious that the advantages of diversity become less when more of the design measures described in the previous section are applied. For the extreme case where very simple smart sensors are deployed and all design measures are effectively applied, the advantage of diversity seems to be very small indeed. However, when the complexity of the smart sensor increases and it is not possible to apply the recommended design measures, diversity becomes an option to consider when dependability is a major concern. When in doubt reliability analyses are necessary to assess the specific application.

6 Conclusion

The use of smart sensors in applications where a high dependability is required is still being debated. It seems that the dependability of smart sensors is becoming comparable to that of their conventional counterparts. The special nature of the design of smart sensors gives the possibility of some new failure modes. Especially the problems of information overload, increased complexity and human interfacing are to be mentioned. Most of these problems can be mitigated by applying simple guidelines.

Some of the failure modes of smart sensors might lead to common cause failure. Therefore, in redundant configurations there might still be a difference between the use of smart sensors versus the use of conventional sensors. The usage guidelines will also mitigate many of the problems of common cause failure.

Some causes of common cause failure can not be addressed in this way, especially with respect to human interfacing and complexity. These problems can be addressed by using diversity. Diversity also has disadvantages, mainly of operational nature. The decision when to apply diversity depends on the details of the application. Further research will investigate guidelines for taking this decision and the procurement of field data on common cause failure of smart sensors to support the decision process.

Acknowledgements

This work is supported by EPSRC/UK in the framework of the *Diversity with Off-The-Shelf Components*-Project (DOTS), GR/N24056.

References

1. European Space Agency, ECSS-P-001A, *Glossary of terms*, Rev. 1, 1997.
2. Dobbing, A., N. Clark, D. Godfrey, P.M. Harris, G. Parkin, M.J. Stevens & B.A. Wichmann, NPL and Druck Ltd., *Reliability of Smart Instrumentation*, NPL, 4 September 1998.
3. Clark, N., P.M. Harris & B.A. Wichmann, *Reliability of SMART Instrumentation*, IMC Reference: PC/GNSR/5001, 14 August 1998.

4. Wichmann, B.A., *Software Support for Metrology; Best practice guide No. 1; Measurement System Validation: Validation of Measurement Software*, April 2000.
5. IEC, IEC61508, *Functional safety of electrical/electronic/ programmable electronic safety-related systems.*
6. IEC, IEC61511, *Functional Safety: Safety Instrumented Systems for the process industry sector*, .
7. IEC, IEC60770-3/CD, *Transmitters for use in industrial process control systems; Part 3: Methods for evaluation of intelligent transmitters*, Committee Draft.
8. Leveson, N.G., L.D. Pinnel, S.D. Sandys, S. Koga and J.D. Reese, *Analyzing software specifications for mode confusion potential*, Proceeding of the Workshop on Human Error and System Development Conference, Glasgow, Scotland, 1997.
9. IEC, NPR-IEC/TS62098, *Evaluation methods for microprocessor-based instruments*, 2001.
10. Goble, W. & Iwan van Beurden, Exida.com, *PFDavg calculation 3051c Pressure Transmitter*, Report No. ROS 01/10-01 R110, 15 October 2001.
11. Goble, W., I. van Beurden, J. Grebe, Exida.com, *Failure Modes, Effects and Diagnostics Analysis; STT250 temperature transmitter*, HON 01/08-01 R110, Version V1, Rev. R1.0, August 2001.
12. Siemens Moore Process Automation, Inc., *Application Data; Safety Integrity Level Verification Failure Rate Data for the 345 Critical Transmitter*, ADQL-6, Rev. 1, August 2000.
13. Houtermans, M. & D. Baer, Factory Mutual Research, *Reliability Evaluation; Fisher Rosemount 3051C Smart Sensor*, J.I. 0003004515 - ES, Rev. 0.2, October 2000.
14. Rosemount, *Model 8800 Smart Vortex Flowmeter*, 00813-0100-4003, January 1996.
15. Foxboro/Eckardt, *Product Specifications; 133DP Intelligent d/p Transmitter*, PSS EMP0530 A-(en), August 1995.
16. Foxboro/Eckardt, *Product Specifications; 131GP Intelligent Gauge Pressure Transmitter; 132AP Intelligent Absolute Pressure Transmitter*, PSS EMP0510 A-(en), PSS EMP0520 A-(en), September 1995.
17. Eckardt, *Smart multifunction converter 19" TSV 175*, Data Sheet 5 175 100, July 1993.
18. Foxboro/Eckardt, *Product Specification; 134LVD Intelligent Bouyancy Transmitter for Liquid level, Interface and Density*, PSS EML1510 A-(en), July 1995.
19. Foxboro/Eckardt, *Product Specifications; 144LVD Buoyancy Transmitter for Liquid Level, Interface and Density*, PSS EML1610 A-(en), October 1998.
20. Metso Automation, *Neles ND800 Valve Controller*, 7ND 20EN, August 2001.
21. Pullum, L.L., *Software Fault Tolerance; Techniques and Implementation*, Artech House, 2001.

Appendix. Failure Modes and Effects Analysis of a Generic Smart Sensor

The FMEA is based on Figure 1.

Table 1. FMEA of power supply.

Failure mode	Possible cause	Possible consequence	Possible measure	Comparison with conventional sensor
Power supply to all components fails.	Defect.	Failure of entire sensor. Loss of calibration information. Loss of settings information. Loss of signal history information.	Redundant power supply. Uninterruptable power supply.	Conventional sensors might be better at keeping calibration information.
Power supply to sensor subsystem fails.	Wire defect.	No measurement of input signal.		Same.
		Wrong measurement of input signal (e.g. because reference voltage incorrect).		Same.
Power supply to communication interface fails.	Wire defect.	No or incorrect output. No or incorrect settings.	Fault diagnosis.	Not existent in conventional sensors.
Power supply to data processing subsystem fails.	Wire defect.	No or incorrect output. Loss of calibration information. Loss of settings information. Loss of signal history information.	Fault diagnosis.	Conventional sensor has analog data processing, in general without signal history memory. Calibration and settings are done using hardware.
Power supply to human interface fails.	Wire defect.	No display of data to user. Setting of parameters not possible.	Fault diagnosis.	Depends on implementation.
Power supply to electrical output subsystem fails.	Wire defect.	No or incorrect output.	Fault diagnosis.	Same.

Table 2. FMEA of electrical output subsystem.

Failure mode	Possible cause	Possible consequence	Possible measure	Comparison with conventional sensor
Defect in D/A conversion.	Hardware error.	Incorrect output.	Fault diagnosis. Robust design.	Not existent in conventional sensors.

Table 3. FMEA of sensor subsystem.

Failure mode	Possible cause	Possible consequence	Possible measure	Comparison with conventional sensor
Digitisation of data from sensor is incorrect in the value domain.	Defect in A/D conversion. Software error.	Incorrect output.	Redundant digitisation.	Not existent in conventional sensors.
Digitisation of data from sensor is incorrect in the time domain.	Software error.	Incorrect output (e.g. integration might be incorrect).	Software validation.	Not existent in conventional sensors.
Conditioning of data from sensor is incorrect.	Software error.	Incorrect output.	Software validation.	Conditioning of input data can be more elaborate in smart sensor.
Incorrect setting (e.g. of measurement range).	Software error. Wire defect. Maintenance error.	Incorrect output.	Fault diagnosis.	Not existent in conventional sensors.
Processing of signal of auxiliary input sensors.	Software error (e.g. in compensation algorithm).	Incorrect output.	Software validation.	Not existent in conventional sensors.

Table 4. FMEA of communications interface.

Failure mode	Possible cause	Possible consequence	Possible measure	Comparison with conventional sensor
Corrupts signals to external system.	Software error. Information overload.	No or incorrect output. Incorrect information on Fault diagnosis to external system.	Software validation. Change of external system or communication interface, such that information overload cannot occur.	Not existent in conventional sensors.
Corrupts signals to data processing subsystem.	Software error. Information overload.	No or incorrect output. No or incorrect setting of sensor subsystem.	Same.	Not existent in conventional sensors.

13

Table 5. FMEA of data processing subsystem.

Failure mode	Possible cause	Possible consequence	Possible measure	Comparison with conventional sensor
Error in signal processing.	Software error.	Incorrect output	Software validation.	Signal processing can be more elaborate in smart sensor.
Error in fault diagnosis.	Software error.	Incorrect information on fault diagnosis to external system.	Software validation.	Not existent in conventional sensors.
Error in calibration.	Software error. Failure of communications interface.	Incorrect output.	Software validation. Fault diagnosis.	Calibration can be more elaborate in smart sensors.
Error in data storage	Software error. Hardware error.	Incorrect settings. Incorrect output. Incorrect calibration.	Software validation. Redundancy, e.g. error correcting/detecting codes. Hardware checks.	Not existent in conventional sensors.

Table 6. FMEA of human interface.

Failure mode	Possible cause	Possible consequence	Possible measure	Comparison with conventional sensor
Incorrect display of data.	Software error. Human error.	Software error. Incorrect action by controller.	Software validation. Improvement of interface.	Conventional sensors exhibit other failure behaviour, e.g. graceful degradation is more likely.
No or incorrect setting of parameters.	Software error. Human error.	Incorrect output.	Software validation. Improvement of interface.	Setting of parameters might be more visible with conventional sensors.

Table 7. FMEA of clock signal.

Failure mode	Possible cause	Possible consequence	Possible measure	Comparison with conventional sensor
Clock not present.	Hardware error.	No or incorrect output.	Fail-safe design.	Not existent in conventional sensors.
Clock has wrong or changing frequency.	Hardware error.	Incorrect output (e.g. integration might be incorrect).	Robust design. Fault diagnosis.	Not existent in conventional sensors.

14

Table 8. Rating of failure modes with respect to common cause failure. The rating "R" is a relative measure, 1 is for the lowest likelihood of common cause failure, 5 for the highest. (HCI=Human Computer Interface, EMI=Electromagnetic Interference.)

Failure mode	R	Rationale for rating
Power supply failure	4	Only when sensors have the same power supply.
Digitisation of data from sensor is incorrect in the value domain.	1	Core functionality.
Digitisation of data from sensor is incorrect in the time domain.	2	Core functionality. Information overload.
Conditioning of data from sensor is incorrect.	1	Core functionality.
Incorrect setting (e.g. of measurement range).	5	HCI.
Processing of signal of auxiliary input sensors.	2	Complexity.
Communications interface corrupts signals to external system.	3	Information overload.
Communications interface corrupts signals to data processing subsystem.	3	Information overload.
Error in signal processing.	2	Core functionality. Complexity.
Error in fault diagnosis.	1	Although faults in the software for fault diagnosis are the same for sensors of the same type, the error only becomes apparent when a fault occurs.
Error in calibration.	2	Core functionality. New calibration or loss of calibration might have same cause.
Error in data storage	1	EMI. Core functionality. But: the occurrence of errors in data storage will be highly independent.
Incorrect display of data.	4	HCI.
No or incorrect setting of parameters.	5	HCI.
Defect in D/A conversion.	1	Core functionality. EMI.
Clock not present.	1	Failure of clocks in different sensors will be highly independent.
Clock has wrong or changing frequency.	1	Failure of clocks in different sensers will be highly independent.

C.2 Safecomp 2004

J.G.W. Bentley, P.G. Bishop, and M.J.P. van der Meulen. An empirical exploration of the difficulty function. In M. Heisel, P. Liggesmeyer, and S. Wittmann, editors, *Proceedings of the 22nd international conference on Computer Safety, Reliability and Security, Safecomp 2004*, Volume 3219 of *Lecture Notes in Computer Science*, pages 60-71, Potsdam, Germany, September 2004, ©Springer. With kind permission of Springer Science and Business Media.

An Empirical Exploration of the Difficulty Function

Julian G W Bentley, Peter G Bishop, Meine van der Meulen

Centre for Software Reliability
City University
Northampton Square
London EC1V 0HB, UK

Abstract. The theory developed by Eckhardt and Lee (and later extended by Littlewood and Miller) utilises the concept of a "difficulty function" to estimate the expected gain in reliability of fault tolerant architectures based on diverse programs. The "difficulty function" is the likelihood that a randomly chosen program will fail for any given input value. To date this has been an abstract concept that explains why dependent failures are likely to occur. This paper presents an empirical measurement of the difficulty function based on an analysis of over six thousand program versions implemented to a common specification. The study derived a "score function" for each version. It was found that several different program versions produced identical score functions, which when analysed, were usually found to be due to common programming faults. The score functions of the individual versions were combined to derive an approximation of the difficulty function. For this particular (relatively simple) problem specification, it was shown that the difficulty function derived from the program versions was fairly flat, and the reliability gain from using multi-version programs would be close to that expected from the independence assumption.

1. Introduction

The concept of using diversely developed programs (N-version programming) to improve reliability was first proposed by Avizienis [1]. However, experimental studies of N-version programming showed that the failures of the diverse versions were not independent, for example [2, 4] showed that common specification faults existed, and Knight and Leveson [6] demonstrated that failure dependency existed between diverse implementation faults to a high level of statistical confidence. More generally, theoretical models of diversity show that dependent failures are likely to exist for any pair of programs. The most notable models have been developed by Eckhardt and Lee [5] and Littlewood and Miller [7]. A recent exposition of these theories can be found in [8]. These models predict that, if the "difficulty" of correct execution varies with the input value, program versions developed "independently" will, on average, not fail independently. A key parameter in these models is the "difficulty function". This function represents the likelihood that a randomly chosen

program will fail for any given input scenario (i.e. the probability that the programmer is more likely to make a mistake handling this particular input scenario).

While there has been considerable theoretical analysis of diversity, and empirical measurement of reliability improvement, there has been little research on the direct measurement of the difficulty function. This paper presents an empirical analysis of many thousands of "independently" developed program versions written to a common specification in a programming contest. The objectives of the study were:

- to directly measure the failure regions for each program version,
- to examine the underlying causes for faults that lead to similar or identical failure regions,
- to compute the difficulty function by combining the failure region results
- to assess the average reliability improvement of diverse program pairs, and compare it with the improve expected if the failures were independent.

The focus of this study was on diverse *implementation* faults. The correctness, completeness and accuracy of the specification were considered to be outside the scope of this project. However, specification-related problems were encountered in the study, and are discussed later in the paper.

In Section 2 of the paper we describe the source of the program versions used in this study, Section 3 summarises the difficulty function theory, Section 4 describes the measurements performed on the programs, while Sections 5 and 6 present an analysis of the results. Sections 7 and 8 discuss the results and draw some preliminary conclusions.

2. The Programming Contest software resource

In the past, obtaining many independently developed program versions by different authors to solve a particular problem would have been difficult. However, with wider use of the Internet, the concept of "programming contests" has evolved. "Contest Hosts" specify mathematical or logical challenges (specifications) to be solved programmatically by anyone willing and able to participate. Participants make submissions of program versions that attempt to satisfy the published specification. These are then "judged" (usually by some automated test system at the contest site) and then accepted or rejected.

We established contact with the organiser of one of these sites (the University of Valladolid) which hosts contest problems for the ACM and additional contest problems maintained by the University [9]. The organiser supplied over six thousand program submissions for one of its published problems. The programs varied by author, country of origin, and programming language. Authors often submitted several versions in attempting to produce a correct solution to the problem. This program corpus formed the basis for our research study.

Clearly, there are issues about realism of these programs when compared to "real world" software development practices, and these issues are discussed in Section 7. However the availability of so many program versions does allow genuine statistical studies to be made, and does allow conjectures to be made which can be tested on

other examples. In addition such conjectures can be evaluated on actual industrial software and hence have the potential to be extended to a wider class of programs.

3. Probability of failure and the difficulty function

Two of the most well known probability models in this domain, are the Eckhardt and Lee model [5], and, the Littlewood and Miller extended model [7]. Both models assume that:

1. Failures of an individual program π are deterministic and a program version either fails or succeeds for each input value x. The failure region of a program π can be represented by a "score" function" $\omega(\pi, x)$ which produces a zero if the program succeeds for a given x or a one if it fails.
2. There is randomness due to the development process. This is represented as the random selection of a program from the set of all possible program versions Π that can feasibly be developed and/or envisaged. The probability that a particular version π, will be produced is $P(\pi)$.
3. There is randomness due to the demands in operation. This is represented by the (random) set of all possible demands X (i.e. inputs and/or states) that can possibly occur, together with the probability of selection of a given input demand x, $P(x)$.

Using these model assumptions, the average probability of a program version failing on a given demand is given by the *difficulty function*, $\theta(x)$, where:

$$\theta(x) = \sum_{\pi} \omega(\pi, x) \mathrm{P}(\pi) \tag{1}$$

The average probability of failure per demand (*pfd*) of a randomly chosen single program version can be computed using the difficulty function and the demand profile $P(x)$:

$$\mathrm{E}(\mathrm{pfd}_1) = \sum_{x} \theta(x) \mathrm{P}(x) \tag{2}$$

The average pfd of randomly chosen pair of program versions (π_A, π_B) taken from two possible populations A and B is:

$$\mathrm{E}(\mathrm{pfd}_2) = \sum_{x} \theta_A(x) \theta_B(x) \mathrm{P}(x) \tag{3}$$

The Eckhardt and Lee model assumes similar development processes for A and B and hence identical difficulty functions

$$\mathrm{E}(\mathrm{pfd}_2) = \sum_{x} \theta(x)^2 \mathrm{P}(x) \tag{4}$$

where $\theta(x)$ is the common difficulty function. If $\theta(x)$ is constant for all x (i.e. the difficulty function is "flat") then, the reliability improvement for a diverse pair will (on average) satisfy the independence assumption, i.e.:

$$\mathrm{E}(\mathrm{pfd}_2) = \mathrm{E}(\mathrm{pfd}_1)^2 \tag{5}$$

However if the difficulty function is "bumpy", it is always the case that:

$$\mathrm{E}(\mathrm{pfd}_2) \geq \mathrm{E}(\mathrm{pfd}_1)^2 \tag{6}$$

If there is a very "spiky" difficulty surface, the diverse program versions tend to fail on exactly the same inputs. Consequently, diversity is likely to yield little benefit and pfd_2 is close to pfd_1. If, however, there is a relatively "flat" difficulty surface the program versions do not tend to fail on the same inputs and hence pfd_2 is closer to pfd_1^2 (the independence assumption).

If the populations A and B differ (the Littlewood and Miller model), the improvement can, in principle, be *better* that the independence assumption, i.e. when the "valleys" in $\theta_A(x)$ coincide with the "hills" in $\theta_B(x)$, it is possible for the expected pfd_2 to be less than that predicted by the independence assumption.

4. Experimental study

For our study we selected a relatively simple Contest Host problem. The problem specified that two inputs, velocity (v) and time (t) had to be used to compute a displacement or distance (d). The problem had defined integer input ranges. Velocity v had a defined range of ($-100 \leq v \leq 100$), whilst time t was defined as ($0 \leq t \leq 200$). A set of 40401 unique values would therefore cover all possible input combinations that could be submitted for the calculation. However, this was not the entire input domain, because the problem specification permitted an arbitrary sequence of input lines, each specifying a new calculation. If all possible sequences of the input pairs (v, t) were considered, assuming no constraints on sequencing or repetition, the input domain for the program could be viewed as infinite. However, as each line of input should be computed independently from every other line, the sequence order should not be relevant, so the experiment chose to base its analysis on the combination of all possible values of v and t. This can be viewed as a projection of the input domain (which has a third "sequence" dimension) on to the (v, t) plane.

The experiment set up a test harness to apply a sequence of 40401 different values of v and t to the available versions. The results for each version were recorded and compared against a selected "oracle" program. The success or failure of each input could then be determined. Some versions were found to have identical results to others for all inputs. The identical results were grouped together in "equivalence classes".

In terms of the difficulty function theory outlined, each equivalence class was viewed as a possible program, π, taken from the universe of all programs, Π, for that specification. The record of success/failure for each input value is equivalent to the score function, $\omega(\pi, x)$ for the equivalence class as it represents a binary value for every point in the input domain, x, indicating whether the result was correct or not. For the chosen problem, the input domain, x, is a two-dimensional space with axes of velocity (v) and time (t), and the score function represented the failure region within that input domain.

$P(\pi)$ was estimated by taking the ratio of the number of instances in an equivalence class against the total number of programs in the population. The size of the failure region was taken to be the proportion of input values that resulted in failure. The failure regions can be represented two dimensionally on the v, t plane, but it should be

emphasised that this is only a projection of the overall input domain. It is only possible to sample the total input domain.

5. Results

The results revealed that the 2529 initial program versions produced by the authors (the "v1" population) formed 50 equivalence classes. The five most frequent equivalence classes accounted for approximately 96% of the population. The results of the analysis are summarised in Table 1.

Table 1. Population v1 equivalence classes (frequent)

Equivalence Class (π)	Number of versions	P(π)	Size of Failure Region
EC1	1928	0.762	0.000
EC2	201	0.079	1.000
EC3	189	0.075	0.495
EC4	90	0.036	0.999
EC5	27	0.011	0.990

Equivalence class 1 agrees with the oracle program. There are no known faults associated with this equivalence class result, consequently the size of the failure region was 0%.

For equivalence class 2, analysis of the programs revealed a range of different faults resulted in complete failure across the input domain.

For equivalence class 3, failures always occurred for $v < 0$. This was due to a specification discrepancy on the Contest Host web site. Two specifications existed on the site—one in a PDF document, the other on the actual web page. The PDF specification required a *distance* (which is always positive) while the web specification required a *displacement* which can be positive or negative. The "displacement" version was judged to be the correct version.

Equivalence class 4, typified those versions that lacked implementation of a loop to process a sequence of input lines (i.e. only computed the first input line correctly).

For equivalence class 5, inspection of the program versions revealed a variable declaration fault to be the likely cause.

A similar analysis was performed on the final program version submitted by each author (the "vFinal" population). The results revealed that of the 2666 final program versions could be grouped into 34 equivalence classes. The five most frequent equivalence classes accounted for approximately 98% of the population. The results of the analysis are summarised in Table 2.

Table 2. Population vFinal: equivalence classes

Equivalence Class (π)	Number of versions	$P(\pi)$	Size of Failure Region
EC1	2458	0.922	0.000
EC2	70	0.026	1.000
EC3	40	0.015	0.495
EC4	21	0.008	0.999
EC5	13	0.005	0.990

Note that there is some overlap between the "first" and "final" populations as some authors only submitted one version. It can be seen that the dominant equivalence classes are the same as in the first version, but the proportions of each equivalence class have decreased (apart from EC1) presumably because some programs have been successfully debugged.

Figure 1 shows examples of the less frequent equivalence class failure regions.

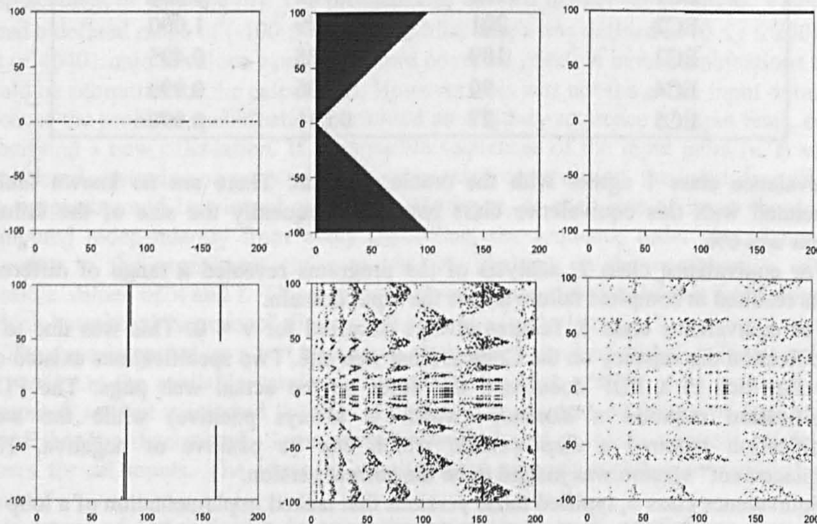


Fig. 1. Failure regions for some of the infrequent equivalence classes

These graphs show that there is a remarkable variation in the failure regions even for such a simple problem. The failure regions for the frequent EC1 to EC5 equivalence classes are simpler in structure, i.e. all "white" for EC1, almost all "black" for EC2, EC4 and EC5 and a black rectangle for EC3 covering the negative portion of the input domain.

6. Analysis

The "score functions" and frequency data of the equivalence classes can be combined to estimate the difficulty function for the specific problem. Note that this is an approximation to the actual difficulty function which should be an average taken over the population of all possible programs. It is unlikely that the set of all possible programs (Π) are limited to the 50 equivalence classes identified in this study. However, a computation of $\theta(x)$ based on the known equivalence classes should give a good approximation to the difficulty function, as 95% (v1) and 97% (vFinal) of the program versions belonged to four of the most frequently occurring known equivalence classes so uncertainties in the "tail" of the population of programs will only have a marginal effect on the difficulty function estimate.

One issue that needed to be considered in the analysis was the effect of the specification discrepancy. The discrepancy will bias the estimate of implementation difficulty as equivalence class EC3 might not have occurred if the specification had been unambiguous. On the other hand, such specification problems might be typical of the effect of specification ambiguity on the difficulty function. We therefore calculated the difficulty function in two ways:

- including all equivalence classes
- all equivalence classes except EC3 (the adjusted difficulty function).

6.1 Calculation of the difficulty function

For each input value, x, the difficulty function value $\theta(x)$ was estimated using equation (1) and the result for the v1 population is shown in Figure 2.

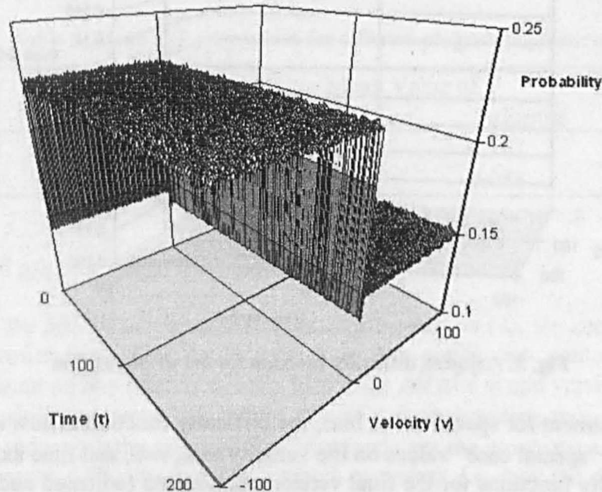


Fig. 2. Difficulty function for the v1 population

This calculation assumes that π is the same as an equivalence class, the score function $\omega(\pi, x)$ is the same as the observed failure region and $P(\pi)$ is the relative frequency of the equivalence class in the population. Effectively the calculation takes a weighted average of the individual failure regions in the v1 population.

Note that the difficulty function shown in Figure 2 has not accounted for any bias introduced by the specification discrepancy and the "step" in difficulty for $v<0$ is due to the specification ambiguity.

The probability of failure also decreases for certain "special" values—the velocity axis v=0, and time axis t=0. This might be expected since an incorrect function of v and t might well yield the same value as the correct function for these special values (i.e. a displacement of zero). There is also a low probability value at v=-100, t=0 which is due to faults that fail to execute subsequent lines in the input file, and the first test input value happens to be v=-100, t=0. If the test input values had been submitted in a random order, this point would have been no more likely to fail than adjacent points.

It can also be seen that there is a certain amount of "noise" on the two "flat" regions of the difficulty surface. This is caused by some of the highly complex failure patterns that exist for some of the infrequent equivalence classes (as illustrated in Figure 1).

The results were adjusted to account for specification bias by eliminating the equivalence class EC3 and Figure 3 shows the adjusted difficulty function.

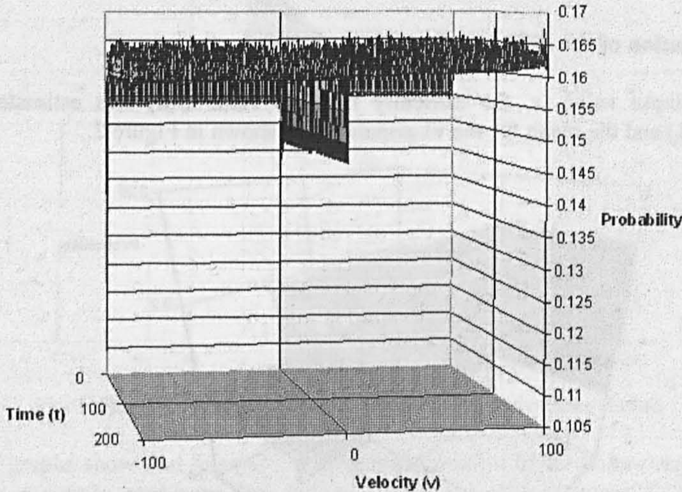


Fig. 3. Adjusted difficulty function for the v1 population

With the adjustment for specification bias, the difficulty function is now almost "flat" apart from the "special case" values on the velocity axis, v=0, and time axis, t=0.

The difficulty functions for the final version populations (adjusted and unadjusted) are very similar in shapes observed in the v1 population. The adjusted vFinal difficulty function is shown in Figure 4.

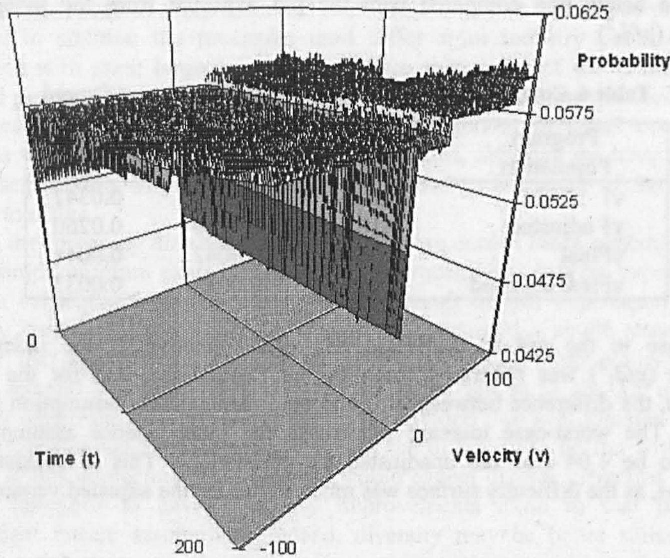


Fig. 4. Adjusted difficulty function for the vFinal population

While the difficulty functions are similar in shape to the v1 population difficulty functions, the mean value of $\theta(x)$ is about one third that of the v1 population—mainly because the vFinal population contains a higher proportion of correct program versions.

The mean values for $\theta(x)$ are summarised in the table below.

Table 3. Mean difficulty values for different program populations

Program Population	Mean Value of θ Unadjusted	Adjusted
v1	0.186	0.161
vFinal	0.064	0.058

6.2 Expected *pfd* of a single version and a pair of versions

To compute the *pfd* for an average program from equation (2), we need to know the execution profile P(x). This could vary from one application context to another. However, assuming any input is equally likely, the *pfd* of a single version is the mean value of θ, while the dangerous failure rate of a fault-detecting pair, pfd_2, given in equation (4) reduces to the mean of $\theta(x)^2$ averaged over the input space. Note that this assumes the same difficulty function for both programs, i.e. they are drawn from the same population (the Eckhardt and Lee assumption [5]).

The expected *pfd*s for a single version and a pair of versions were computed for the v1 and vFinal program populations (and the adjusted versions). The results are shown

in the table below and compared with the pfd expected from the independence assumption (pfd_1^2).

Table 4. Comparison of expected probability of failure on demand

Program Population	pfd_1	pfd_2	pfd_1^2
v1	0.186	0.0361	0.0347
v1 adjusted	0.161	0.0260	0.0260
vFinal	0.064	0.0042	0.0041
vFinal adjusted	0.058	0.0033	0.0033

The increase in the *pfd* of a diverse pair (pfd_2) relative to the independence assumption (pfd_1^2) was relatively small for all populations, and for the adjusted populations, the difference between pfd_2 and the independence assumption is almost negligible. The worst-case increase relative to the independence assumption was observed to be 1.04 (for the unadjusted v1 population). This is consistent with expectations, as the difficulty surface was much flatter for the adjusted versions.

7. Discussion

While the results are interesting, we have to be cautious about their applicability to "real world" programs. Programming contests can provide many thousands of versions and this is a clear benefit for statistical studies. On the other hand, the results may be unrepresentative of software development in industry, especially in that:

1. many of the developers are probably amateurs or students rather than professional developers;
2. the program specifications are not overly complex, so that the programs are not typical of software developed in industry, and whole classes of faults that arise in the development of complex software may be missing;
3. the development process is different from the processes applied in industry;
4. there is no experimental control over program development, so independence could be compromised, e.g. by copying other participants' programs, or by submitting programs produced collectively by multiple people.

Discussions with the contest organiser suggests that plagiarism is not considered to be a major issue and, in any case, the main effect of plagiarism of correct versions would be to increase the number of correct versions slightly. In principle, it should be feasible to trap programs from different authors that are identical or very similar in structure.

With regard to programming expertise, the top participants are known to take part in international programming contests under controlled conditions (in a physical location rather than on the internet). So it seems that there is a very broad range of expertise. In future studies we might be able to obtain more information about the participants so that the level of expertise can be more closely controlled.

The example we have studied is "programming in the small" rather than "programming in the large". It is therefore likely that there are classes of "large

program" fault, such as integration and interface faults, that will not be present in our example. In addition the processes used differ from industry practice. However experience with quite large programs indicates that many of the faults are due to localised programming errors that remained undetected by industrial verification and validation phases. So it is likely the errors committed in small contest-derived programs will also arise in large industrial programs, although we have to recognise that the set of faults will be incomplete and the relative frequency of the fault classes is likely to differ.

From the foregoing discussion, it is clear that we cannot make general conclusions from a single program example. However, the results can suggest hypotheses to be tested in subsequent experiments. One clear result of this experiment is that the difficulty surface is quite flat. The specification required a single simple "transfer function" that applies to the whole of the input domain. One might conjecture that, for a fixed transfer function, the difficulty would be the same for all input values. Similarly where the program input domain is divided into sub-domains which have different transfer functions, we might expect the difficulty to be flat within each sib domain. If this conjecture is correct, we would expect diverse programs with simple transfer functions to have reliability improvements close to that predicted by independent failure assumption. Indeed, diversity may be better suited to simple functions rather than entire large complex programs. However we emphasise that this is a conjecture, and more experiments would be needed to test this hypothesis.

It should also be noted the *pfd* reduction derived in Table 4 is the *average reduction*. For a specific pair of program versions it is possible for the actual level of reduction to vary from zero to complete reduction. A zero reduction case would occur if a pair of versions from the same equivalence class are selected. Conversely, complete reduction occurs if an incorrect version is combined with a correct version. In Table 2, for instance, 92% of versions in the final population are correct so the chance that a pair of versions will be faulty is $(1-0.92)^2$, i.e. 0.64%. It follows that the chance of a totally fault detecting pair (where at least one version is correct) will 99.36%. Pairs with lower detection performance will be distributed within in the remaining 0.64% of the population of possible pairs and some of these versions will behave identically and hence have zero failure detection probability.

Another issue that we did not plan to examine was the impact of specification problems. However it is apparent that the problem we encountered was a particular example of specification ambiguity that arises in many projects. This illustrates how N-version programming can be vulnerable to common specification problems, and the need for appropriate software engineering strategies to ensure that specifications are sound.

At a more general level, we have to ask whether such experiments are of practical relevance to industry. As discussed earlier, the examples we use are not typical as they are not as complex as industrial software and the development processes differ. However, the experiments could lead to conjectures that *could* be tested on industrially produced software, (such as the assumption of constant difficulty over a sub-domain). If such conjectures are shown to be applicable to industrial software, this information could be used to predict, for example, the expected *variation* in difficulty over the sub-domains and hence the expected gain from using diverse software. It has to be recognised that relating the research to industrial software will

be difficult and, at least initially, is most likely to be applicable to software implementing relatively simple functions (like smart sensors). We hope to address this issue in future research.

8. Conclusions and further work

We conclude that:

1. One significant source of failure was the specification. We were able to allow for the specification discrepancy in our analysis, but it does point to a more general issue with N-version programming, i.e. that it is vulnerable errors in the specification, so a sound specification is an essential prerequisite to the deployment of N-version programming.
2. For this particular example, the difficulty surface was almost flat. This indicates that there was little variation of difficulty and a significant improvement in reliability should (on average) be achieved, although the reliability of arbitrary pair of versions can vary significantly from this average.

We conjecture that for programs with a single simple transfer function over the whole input domain (like this example), the difficulty function might turn out to be relatively flat. In that case, reliability improvements close to the assumption of independent failures may be achievable. However, more experiments would be needed to test this hypothesis.

There is significant potential for future research on variations in difficulty. The possibilities include:

1. Variation of difficulty for *different* sub-populations (e.g. computer language, author nationality, level of expertise, etc). The extended Littlewood and Miller theory suggests it is possible to have reliability better than the independence assumption value. An empirical study could be envisaged, to determine if this is observed when versions from different populations are combined.
2. Extension to other contest host program examples, and more wide-ranging experiments to assess conjectures like the flat difficulty conjecture discussed above.
3. Relating the hypotheses generated in the experiments to industrial examples.

Acknowledgements

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council via the *Interdisciplinary Collaboration on the Dependability of computer based systems*, (DIRC), and via the *Diversity with Off-The-Shelf Components* Project (DOTS), GR/N24056. The authors would like to thank Miguel Revilla, University of Valladolid, Spain, for providing the program versions and information relating to the programming contest.

References

[1] A. Avizienis and L. Chen, "On the Implementation of N-version Programming for Software Fault Tolerance during Execution", Proc. the First IEEE-CS International Computer Software and Applications Conference (COMPSAC 77), Chicago, Nov 1977.
[2] A. Avizienis and J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments", Computer, Vol. 17, No. 8, August 1984.
[3] A. Avizienis, "Software Fault Tolerance", Information Processing (G.X. Titter, Ed.), pp. 491-498, Elsevier Science Publishers, Holland, 1989.
[4] P.G. Bishop, M. Barnes, et/ al., "PODS a Project on Diverse Software", IEEE Trans. Software Engineering, Vol. SE-12, No. 9, 929-940, 1986.
[5] D. E. Eckhardt, L. D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors", IEEE Transactions on Software Engineering, SE-11 (12), pp.1511-1517, 1985.
[6] J. C. Knight and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming", IEEE Transactions on Software Engineering, SE-12 (1), pp. 96-109, 1986.
[7] B. Littlewood and D. R. Miller, "Conceptual Modelling of Coincident Failures in Multiversion Software", IEEE Transactions on Software Engineering, Vol. 15, No. 2, pp. 1596-1614, December 1989.
[8] B. Littlewood, P. Popov and L. Strigini, "Modelling software design diversity - a review", ACM Computing Surveys, vol. 33, no. 2, 2001, pp.177-208.
[9] S. Skiena and M. Revilla, Programming Challenges, ISBN: 0387001638, Springer Verlag, March, 2003, (http://acm.uva.es/problemset/)

C.3 ISSRE 2004

M.J.P. van der Meulen, P.G. Bishop, and M.A. Revilla, An exploration of software faults and failure behaviour in a large population of programs. In *Proceedings of the 15th IEEE International Symposium of Software Reliability Engineering*, pages 101–12, St. Malo, France, November 2004.

An Exploration of Software Faults and Failure Behaviour in a Large Population of Programs

M.J.P. van der Meulen, P.G. Bishop

Centre for Software Reliability
City University
EC1V 0HB London, UK

M. Revilla

Department of Applied Mathematics
University of Valladolid
47011 Valladolid, Spain

Abstract

A large part of software engineering research suffers from a major problem—there are insufficient data to test software hypotheses, or to estimate parameters in models. To obtain statistically significant results, a large set of programs is needed, each set comprising many programs built to the same specification. We have gained access to such a large body of programs (written in C, C++, Java or Pascal) and in this paper we present the results of an exploratory analysis of around 29,000 C programs written to a common specification.

The objectives of this study were to characterise the types of fault that are present in these programs; to characterise how programs are debugged during development; and to assess the effectiveness of diverse programming. The findings are discussed, together with the potential limitations on the realism of the findings.

1. Introduction

To date software engineering research has been based on relatively small samples of programs; at most a few tens of programs have been used in controlled experiments to test hypotheses. Ideally far more programs, written to a common specification, are needed to undertake statistical analyses, and many different specifications are needed to demonstrate results are generally applicable. In this paper we identify such a body of programs, and present the results of our exploratory analysis.

The UVa Online Judge Website is an initiative of Miguel Revilla of the University of Valladolid [8]. It contains problems to which everyone can submit solutions. The solutions are programs written in C, C++, Java or Pascal. The correctness of the programs is automatically judged by the "Online Judge". Most authors submit solutions until their solution is judged as being correct. There are many thousands of authors and together they have produced more than 2,500,000 solutions to the approximately 1500 problems on the website.

From the perspective of algorithm design, the programming contest is a treasure trove. There appear to be numerous ways to solve the same problem. But also for software reliability engineers this is the case: there are even more ways to *not solve* the problem. Most authors' first submission is incorrect. They take some trials to—in most cases—finally arrive at the correct solution. What happens between this first submission and their final one is illuminating.

Ideally analyses should be performed on different sets of programs to identify common features. But in this paper we focus on a single set of 29,000 C programs version written to a common specification, the "3n+1"-problem. In this exploratory study, we examine three different aspects in software engineering:

- what types of faults are introduced;
- how programs are debugged during development;
- whether diverse programs are likely to be effective.

In the following sections we introduce the "3n+1"-problem, describe the environment used to test the programs and the results of our exploratory studies of these issues. The relevance of our findings are discussed and we make some conjectures that can be evaluated in future studies.

2. The "3n+1"-problem

The "3n+1"-problem can be summarised as follows:

```
1. input n
2. print n
3. if n = 1 then STOP
```

©2007 IEEE. Reprinted, with permission, from the Proceedings of the 15th IEEE International Symposium of Software Reliability Engineering, pages 101–12, St. Malo, France, November 2004.

```
4.    if n is odd then n := 3n + 1
5.    else  n := n/2
6. GOTO 2
```

For example, given an initial value 22, the following sequence of numbers will be generated 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1.

It is conjectured that the algorithm above will terminate (i.e. stop at one) for any integer input value. Despite the simplicity of the algorithm, it is unknown whether this conjecture is true. It has been verified, however, for all integers n such that $0 < n < 1,000,000$ (and, in fact, for many more numbers than this).

Given an input n, it is possible to determine the length of the number sequence needed to reach the final value of one. This is called the cycle-length of n. In the example above, the cycle length of 22 is 16.

The "3n+1"-problem specification includes the following requirements:

- For any two numbers i and j you are to determine the maximum cycle length over all integers between and including i and j.
- The input will consist of a series of pairs of integers i and j, one pair of integers per line. All integers will be less than 1,000,000 and greater than 0.
- For each pair of input integers i and j the output is i, j, and the maximum cycle length for integers between and including i and j. These three numbers should be separated by at least one space with all three numbers on one line and with one line of output for each line of input.

The specification is supplemented by sample input and output examples, e.g.:

Sample Input.

```
1 10
100 200
```

Sample Output.

```
1 10 20
100 200 125
```

3. Program submissions

The number of programs submitted to this problem is 66,696 at the moment of this analysis, of which 29,102 are written in C. (We consider only those programs that are designated as being written in C by the author, at this moment we do not include C++ programs that are C compatible.) The online judge classifies 7,132 (24.5%) of these as correct, 10,335 (35.5%) as "wrong answer" and 273 (0.9%) as "presentation error". The latter category contains solutions that do not exactly conform to the output specification, but give the correct answer. The remaining 11,362 (39.0%) programs contain fatal errors, take to long to complete, use too much memory, or have other problems. In our analysis we only consider those programs that are either marked as "correct", "wrong answer" or "presentation error".

The number of authors that submitted C programs is 4317, 3444 (79.8%) of whom managed to solve the "3n+1"-problem.

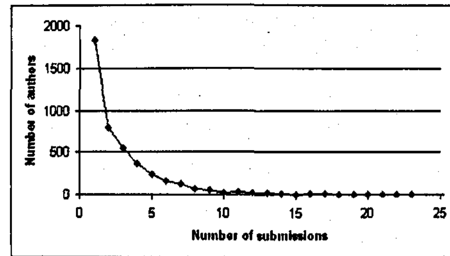


Figure 1: Number of submissions until last or correct solution per author.

The number of programs submitted per author, excluding those submissions after a correct submission, is depicted in Figure 1, the average is 2.9.

4. Solutions to the problem

The example C program in Table 1 shows the approach most programs take. We will use the program's characterisation to describe the faults that authors make.

Of course, the actual programs differ from this example, but most programs take a similar approach and only differ in aspects such as: the use of subroutines for the cycle length calculation and the determination the maximum value. The programs that differ most from the example are those that optimize on speed. These programs can be lengthy and complex, but constitute a minority.

5. Program testing

We submitted all the programs to a benchmark test. The benchmark input is a list of 2,500 pairs of numbers with all combinations of numbers between 1 and 50. The outputs of the programs' executions are written to a file for later analysis. We deleted all output

Table 1: Example program with typical algorithm.

Program	Characterisation
`#include <stdio.h>`	
`#include <stdlib.h>`	
`main()`	
`{`	
`  int a, b, min, max, num;`	Variable declaration
`  register n, cycle, cyclemax;`	
`  while (fscanf(stdin, "%d %d", &a, &b) != EOF) {`	Read inputs
`    if (a < b) min=a; max=b; else min=b; max=a;`	Swap inputs
`    for (cyclemax=-1,`	Reset maximum cycle length
`      num=min; num<=max; num++) {`	Loop between bounds
`      for (n=num, cycle=1; n != 1; cycle++)`	
`        if (n % 2) n=3*n+1; else n >>= 1;`	Calculate cycle length
`      if (cycle > cyclemax) cyclemax=cycle;`	Determine maximum
`    }`	
`    printf ("%d %d %d\n", a, b, cyclemax);`	Write outputs
`  }`	
`}`	

files smaller than half the size of the correct output and larger than twice its size, because we deemed these programs to be incorrect. The output files smaller than half the size are in general either programs that do nothing at all (fake submissions) or only process one or a few inputs. The output files larger than double the size mostly contain intermediate results or text.

It has to be noted that this approach does not identify all faults in the programs. An example of a known fault that is not considered in this analysis is numerical overflow, caused by intermediate results becoming very large. In our assessment, we discarded the programs by authors who needed more than 30 attempts as they were considered to be too incompetent to be typical of normal programming (some authors managed to submit over 500 trial versions).

We were slightly more generous than the online judge in assessing the output files. We only compared the numbers in the output file, so if the output file contains commas, empty lines or short text like "The answer is:" we still treat it as a correctly formatted output. The reason for ignoring commas and short texts might be questioned, but this decision significantly reduces the number of different equivalence classes generated and enhances the opportunities for analysis.

After submitting a correct program, many authors continue submitting, probably to optimise their program, or to make it faster. We are not interested in these aspects, so we discard all programs of an author after submission of a correct program.

When running and comparing these programs it was striking how many different behaviours were observed. In total, 516 different output results were generated. Many are only slightly different, but the fact that such a simple program can be programmed incorrectly in so many ways is surprising.

After eliminating programs that did not conform to the criteria outlined above, a set of 11,951 program versions were available for subsequent analysis.

Three main analyses were performed in this exploratory study:

- analysis of the types of fault introduced (see section 6.);
- analysis of the debugging process, e.g. what faults are removed in successive "releases" submitted by the author (see section 7.);
- assessment of the effectiveness of diversity (see section 8.).

6. Analysis of program faults

6.1 Equivalence classes

We observed that there were many different programs that produced *identical results*. These were generally due to the existence of similar faults in the different versions. We grouped the program versions that produced identical results into "equivalence classes" and used these equivalence classes in our subsequent

analysis. Given the limited space, in this paper we will only consider equivalence classes that contain more than 5 programs.

After grouping the output files of the programs into equivalence classes, we characterised them by the faults they contained (see Table 2). The 36 most frequent equivalence classes are shown, with their total frequencies, the frequencies with which they occur in the first and last programs submitted by the authors, their reliability (i.e. the fraction of correct responses to the 2,500 demands), and a description of the faults that were identified as being present in that class of programs. An assumption we made here is that programs that behave similarly contain the same kind of faults. This may not always be correct, but no counterexamples have yet been found.

6.2 Types of fault

The characteristicis of the faults found in each equivalence class are described below.

Swap: missing or incorrect. This is related to test cases where input i is larger than input j. This is normally handled by swapping the two input values. (Strictly speaking a swap is not necessary, because this functionality can also be implemented in the loop by counting down, but most authors do not use this alternative solution. So we have labelled this a "Swap" problem).

A missing swap indicates incorrect interpretation of the specification: the author did not anticipate the possibility that the second input may be smaller than the first. This is the most frequent mistake: 31% of the programs in the selected equivalence classes exhibit this problem.

Incorrect implementation of the swap is less frequent (14%), in most cases the author did not consider the consequences for bouncing i and j. In some cases it is caused by a slip in a routine programming task.

Write: incorrect order. Returning the input values is one of the possible consequences of implementing the swap incorrectly. The specification clearly specifies that the returned inputs should appear in the same order. The author manages to implement the swap, but forgets to consider the consequences for the write step. The problem is in general solved by either returning the inputs before swapping or by remembering the order of the inputs in separate variables.

Reset maximum cycle length. The author forgets to reset the maximum cycle length for the next set of input values (3.7%). In this case the program will fail if the maximum cycle length for these i, j values is lower than the highest one calculated since the start of the program. This problem is caused by not initialising the loop correctly.

Loop. There appear to be many ways to implement the loop incorrectly (3.6%). Most frequent is the omission of the last value in the loop. An example is shown below:

```
for (StartSequence = StartCounter;
StartSequence < LastCounter; StartSequence++)
```

Another case is the omission of the first and the last values in the loop, e.g.:

```
for(i=min(a,b)+1;i<max(a,b);i++)
```

New line. Some programs (3.4%) do not output a new line between subsequent iterations.

Calculation. Very few programs (0.3%) contain a fault in the calculation of the maximum cycle length. This is probably due to the fact that if the algorithm responds well to the sample outputs given in the problem specification, it will perform well for all inputs. The main problem found is putting step 3, testing for $n = 1$, after step 4 and 5 in the program (see pseudocode in introduction). The program will not check for $n = 1$ immediately, leading to the sequence "1 4 2 1" and a calculated sequence length of 4 instead of 1.

6.3 Failure sets

We also plotted the failure sets that characterised each equivalence class, i.e. for each input pair i, j we noted whether the result was a success or a failure and plotted the failure set as a two-dimensional map. The failure sets are shown in Figure 2.

The triangular pattern, e.g. a), i) and o), is related to the i, j swap problem, i.e. the correct answer is only generated when i is less or equal to j. The diagonal structures like h), p), q) and s) are related to loop implementation problems where either one or both of the i, j endpoint values is not included in the cyclic length calculation. An entirely black square, n), is associated with problems like failing to generate any output, outputting in the wrong format or generating too much output. The most common equivalence class is a completely blank square (not depicted), which represents the case where all test inputs were correct.

The shape of some failure sets depends on the order of the numbers in the input file: b), f), w) and x).

We also see regions that appear to be the superposition of two different failure sets, for example, v) seems to be the superposition of a) and h). This might be the explanation for the large number of different equivalence classes found in the study. For example, 256 different failure set patterns can be generated with only 8 basic patterns found.

Table 2: Equivalence classes and faults. EC: Equivalence Class; Freq.: Frequency of the equivalence class; First: Frequency of the EC as the first attempt; Last: Frequency of the EC as the last attempt; Rel.: Fraction of correct responses to the 2,500 demands; Description: Description of the faults found in the EC, with the consequences of the fault for another program step between brackets.

EC	Freq.	First	Last	Rel.	Description
EC00	3444	1512	3444	100.00%	Correct program.
EC01	1735	707	133	51.00%	Swap: missing. (Calculation: results in 0).
EC02	921	158	67	51.00%	Swap: incorrect. (Write: bounces i and j in incorrect order when $i > j$).
EC03	426	168	37	51.00%	Swap: missing. (Calculation: leads to result 1).
EC04	295	77	17	52.84%	Reset maximum cycle length: not included after first calculation. Swap: missing. (Calculation: results in maximum cycle length of all previous calculations.)
EC05	277	63	9	0.04%	New line: no new line between outputs. (Often hides other faults.)
EC06	211	77	29	58.00%	Swap: missing. (Loop: only lowest number when $i > j$.)
EC07	76	17	3	99.88%	Calculation: wrong for $n = 1$ (program step 3 after 5), leads to result 4.
EC08	74	12	2	26.96%	Reset maximum cycle length: not included after first calculation.
EC09	63	33	3	43.76%	Loop: highest element not included. Swap: missing (Calculation: results in 0.)
EC10	63	16	1	87.52%	Loop: highest number not included, leads to result 0 when $i = j$.
EC11	60	11	22	52.96%	Swap: incorrect. (Write: After first time $i > j$ bounces inputs written in reversed order when $i > j$.)
EC12	39	10	3	54.96%	Swap: incorrect, leads to $i = j = \max(i, j)$ when $i < j$.
EC13	38	6	5	0.04%	Calculation: missing, leads to result 1.
EC14	36	8	1	40.24%	Loop: lowest and highest number not included. Swap: missing, leads to result 0.
EC15	36	2	1	87.52%	Loop: highest number not included. (Calculation: leads to result -1 when $i = j$.)
EC16	35	4	3	99.96%	Calculation: aborts when $n = 1$, leads to result 0.
EC17	32	16	1	50.92%	Swap: missing. (Calculation: results in 0). Calculation: wrong for $n = 1$ (program step 3 after 5), leads to result 4.
EC18	25	4	1	0.00%	Calculation: result one too low. Swap: missing. (Calculation: results in 0.)
EC19	24	6	1	50.96%	Calculation: aborts when $n = 1$, leads to result 0. Swap: missing (Calculation: leads to result 0.)
EC20	21	3	1	92.00%	Loop: lowest element not included.
EC21	21	7	2	50.92%	Loop: only lowest number when $i < j$ Calculation: wrong for $n = 1$ (program step 3 after 5), leads to result 4.
EC22	21	3	9	0.00%	Other: second output is zero.
EC23	20	4	3	51.00%	Swap: incorrect, leads to $i = j = \min(i, j)$.
EC24	19	4	2	80.48%	Loop: lowest and highest number not included.
EC25	16	1	1	2.00%	Swap: incorrect, leads to loop being only correct for $i = j$.
EC26	15	4	2	50.92%	Swap: incorrect, bounces i and j in incorrect order. Calculation: wrong for $n = 1$.
EC27	14	5	13	89.52%	Loop: highest number not included, except when $i = j$.
EC28	14	2	1	2.00%	Other: no output line when $i < j$.
EC29	14	4	1	43.76%	Loop: highest number not included. Swap: incorrect. (Write: bounces i and j in incorrect order when $i > j$.)
EC30	12	2	1	2.00%	Swap: incorrect (swaps when $i < j$), leads to incorrect answer when $i \neq j$.
EC31	11	5	1	43.80%	Swap: missing, leads to result 1. Loop: last element missing.
EC32	10	2	1	51.00%	Swap: incorrect, leads to $i = j = \max(i, j)$. (Write: bounces "i i" if $i > j$.)
EC33	10	3	3	54.76%	Swap: missing, leads to last calculation result.
EC34	9	1	2	99.96%	Calculation: wrong for $n = 1$ (increment of cycle length incorrect for $n = 1$), leads to result 2.
EC35	6	1	1	48.32%	Loop: incorrect, leads to result being one too low if maximum cycle length of longest sequence is one higher than the next highest length.
EC36	5	2	1	99.92%	Calculation: wrong for $(i, j) = (1, 2)$ or $(i, j) = (2, 1)$ (program step 3 after 5), leads to result 4.
Total	8148	2960	3828		

7. Analysis of the debugging process

We have seen that there are a several types of faults which can appear in a number of different "varieties" for the same underlying programming fault. We have also seen that these basic faults appear to be superimposed on each other, i.e. an equivalence class consists

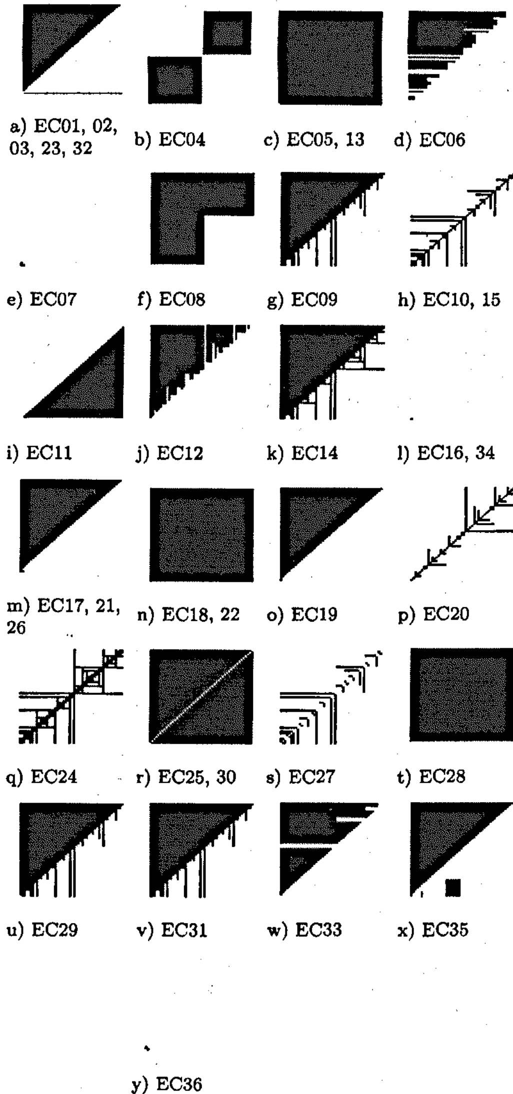


Figure 2: Failure sets for the equivalence classes.

a combination of one or more "basic" faults.

We might therefore expect the debugging process to result in the removal of successive bugs and hence there would be a transition from one equivalence class to another with fewer basic faults. If this supposition is correct, we would expect that:

- relatively few transition steps are needed before the final "correct" equivalence class is reached;
- few equivalence classes are "reachable" from another class (i.e., only the ones with one more or one less basic fault).

At a more global level, it may be the case that some faults are more difficult to eliminate than others, so we might expect to see different proportions of basic defects as debugging progresses. These issues are examined in the sections below.

7.1 Transitions between the equivalence classes

An analysis was performed of the transitions between equivalence classes. In Table 3 we show the mean number of transition steps needed to arrive at a correct program from a program in a given equivalence class. The table also shows the number of steps it takes to arrive at a correct program, for those cases in which the author manages to do this, and the percentage of submissions for which the author does not manage to correct the program.

An "ideal" debugging process would eliminate a basic fault at each step, but in practice it can be seen that the number of steps needed to correct faults is higher: on average 2.8 steps for correcting one fault and 3.1 for correcting two faults. The number of faults here is the number identified in Table 2; this number is however not exact, because some authors may correct two faults at the same time, and a program can contain more faults than listed. One possible cause of these extra transitions might be correction-induced faults, where a new fault is sometimes added to the set. However our analysis indicates that the primary reason for the additional transitions is that the next release has the *same* equivalence class. The probable explanation for that is that the Contest Host provides no debugging information, i.e. it does not provide any information about the test input values that caused the failure or which element of the answer is incorrect. This could result in the programmer making cosmetic changes rather than addressing the actual problem. Table 3 also shows the probability of staying in the same class for the different equivalence classes.

It can be seen that for some equivalence classes there is a 100% probability of staying in the same class, while in other cases there is only a 4% percent probability. However there is no obvious relationship between the faults present in the class and the transition probability.

A full transition matrix between equivalence classes is given in the final table at the end of the paper. From this table it is clear that authors do not insert faults of

Table 3: The probability of staying in the same equivalence class for a program in a given equivalence class, the mean number of steps to correct the program, and the percentage of programs that are never corrected. (#Tr.: Number of transitions.)

Equiv. class	#Tr. to same EC	Total #Tr. from EC	% within EC	Mean #Steps to correct	% never corrected
EC01	702	1602	44 %	3.1	23 %
EC02	324	854	38 %	2.1	16 %
EC03	174	389	45 %	3.3	25 %
EC04	96	278	35 %	2.8	17 %
EC05	117	265	44 %	3.3	19 %
EC06	80	182	44 %	3.1	38 %
EC07	32	73	44 %	2.1	14 %
EC08	28	72	39 %	3.7	20 %
EC09	14	60	23 %	4.6	16 %
EC10	24	62	39 %	2.3	13 %
EC11	20	38	53 %	1.7	67 %
EC12	19	36	53 %	1.5	51 %
EC13	22	33	67 %	2.6	76 %
EC14	4	35	11 %	4.2	22 %
EC15	21	35	60 %	4.1	3 %
EC16	11	32	34 %	1.7	31 %
EC17	12	31	39 %	4.6	16 %
EC18	1	24	4 %	4.3	12 %
EC19	6	23	26 %	3.5	21 %
EC20	6	20	30 %	2.2	19 %
EC21	11	19	58 %	3.8	38 %
EC22	11	12	92 %	N/A	100 %
EC23	6	17	35 %	5.3	20 %
EC24	5	17	29 %	2.8	16 %
EC25	4	15	27 %	1.7	6 %
EC26	3	13	23 %	2.4	53 %
EC27	1	1	100 %	N/A	100 %
EC28	5	14	36 %	2.1	0 %
EC29	1	14	7 %	4.2	14 %
EC30	2	12	17 %	2.0	8 %
EC31	3	11	27 %	5.8	18 %
EC32	2	9	22 %	1.4	20 %
EC33	3	7	43 %	1.5	40 %
EC34	2	7	29 %	1.5	33 %
EC35	1	6	17 %	1.7	50 %
EC36	2	4	50 %	N/A	100 %

an entirely different category. e.g. there are no transitions from EC07 to EC06 where the faults are disjoint. The diagonal is a dominant feature of the transition table. These are transitions within the same equivalence class.

7.2 Reliability of successive releases

It is difficult to talk about the reliability of a program version without defining its operational profile. Take for example, the program failure set in Figure 2a. If the input profile was restricted to the top triangular portion the program would always fail, while an input profile that remained in the bottom triangle would never fail. However on average, we would expect reliability to be better when the failure sets become smaller, and if we assume that each input value is equally likely, the probability of failure is proportional to the size of the failure set.

In Figure 3, we show the distribution of the reliability (assuming each input value is equally likely). for successive program "releases" by the authors. The lowest line is the distribution of reliabilities of the first submissions. The second lowest is the second submission, and so forth. It can be seen that:

- the reliability of the program versions improves with successive attempts;
- the gain in reliability per release is decreasing.

This is consistent with the reliability growth behaviour that might be expected if the faults present in a program are removed in successive releases, and the faults with the highest failure rates are removed first.

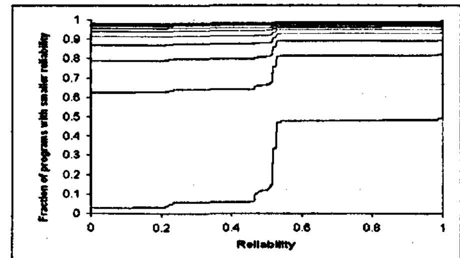


Figure 3: Reliability profile (successive releases).

We also see that there are significant "steps" at certain reliability values, for example a significant fraction of the programs have reliabilities clustered around 0.5. This is caused by one of the basic faults—a missing or incorrect swap (the triangular failure set shown in Figure 2a) which occupies 50% of the input space. We also note that these steps in the distribution remains similar relative to each other, suggesting that there is little difference in the debugging of the difference basic faults within the programs. If for example, the 50% triangle fault was easy to remove, the large step at 50% would disappear after the first release, but in fact all "steps" seem to be removed at a similar rate.

8. Effectiveness of diversity

Two of the most well known probability models in this domain are the Eckhardt and Lee model [3] and the Littlewood and Miller extended model [5]. Both models assume that:

1. Failures of an individual program are deterministic and a program version either fails or succeeds for each input value x. The failure set of a program π can be represented by a "score function" $\omega(\pi, x)$ which produces a zero if the program succeeds for a given x or a one if it fails (see the example in Figure 5).
2. There is randomness due to the development process. This is represented as the random selection of a program from the set of all possible program versions Π that can feasibly be developed and/or envisaged. The probability that a particular version π will be produced is $P(\pi)$. This can be related to the relative numbers of equivalence classes in Table 2.
3. There is randomness due to the demands in operation. This is represented by the (random) set of all possible demands X (i.e. inputs and/or states) that can possibly occur, together with the probability of selection of a given input demand x, $P(x)$.

Using these model assumptions, the average probability of a program version failing on a given demand is given by the difficulty function, $\theta(x)$ where:

$$\theta(x) = \sum_{\pi} \omega(\pi, x) P(\pi) \tag{1}$$

The average probability of failure on demand of a randomly chosen single program version can be computed using the difficulty function and the demand profile,

$$E(\text{pfd}_1) = \sum_{x} \theta(x) P(x) \tag{2}$$

The Eckhardt and Lee model assumes similar development processes for the two programs A and B and hence identical difficulty functions. So the average pfd for a pair of diverse programs (assuming that only agreement on the wrong answer is dangerous) would be:

$$E(\text{pfd}_2) = \sum_{x} \theta(x)^2 P(x) \tag{3}$$

If $\theta(x)$ is constant for all x (i.e. the difficulty function is "flat") then, the reliability improvement for a diverse pair will (on average) satisfy the independence assumption, i.e.:

$$E(\text{pfd}_2) = E(\text{pfd}_1)^2 \tag{4}$$

However if the difficulty function is "bumpy", it is always the case that:

$$E(\text{pfd}_2) > E(\text{pfd}_1)^2 \tag{5}$$

If the difficulty surface is very "spiky" the diverse program versions tend to fail on the exactly the same inputs (where the "spikes" are). In this case, diversity is likely to yield little benefit and pfd_2 could be close to pfd_1. If, however, there is a relatively "flat" difficulty surface there is no a priori reason for program versions to fail on the same inputs and hence pdf_2 should be closer to the value implied by the independence assumption.

If the development processes for A and B differ (the Littlewood and Miller model), the improvement can, in principle, be better than the independence assumption, i.e. when the "dips" in $\theta_A(x)$ coincide with the "spikes" in $\theta_B(x)$, it is possible for the expected value of pfd_2 to be *less* than that predicted by the independence assumption.

At this stage however we have not used programs that can be readily separated into different populations (e.g. by programming language) so our study of effectiveness was confined to deriving a difficulty function for the whole population.

This is fairly simple to derive: for each point in the input space we add up the number of program version that fail and divide by the total number of program versions. The resultant difficulty surface $\theta(x)$ for the "3n+1"-problem is shown in Figure 4.

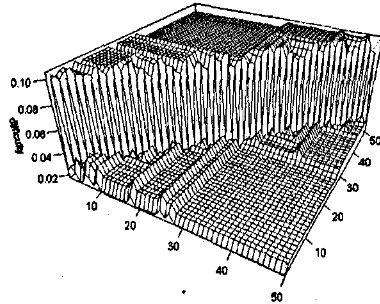


Figure 4: Difficulty function for the final version of the "3n+1"-problem.

As the difficulty surface is the weighted average of the failure sets of the individual equivalence classes, it

is not surprising that the surface is dominated by the most frequently occurring failure set—the triangular region of the "swap"-fault.

To estimate the pfd using equations 2 and 3, we need to specify the input profile $P(x)$. Assuming that all inputs are equally likely we can compute the expected pfd for a single version and a diverse pair:

Table 4: Expected pfd's (from the difficulty function).

Parameter	Initial version	Final version
pfd_1	0.283	0.0624
pfd_2	0.118	0.00518
pfd_1^2 (independent)	0.0800	0.00390
Ratio	1.48	1.33

This can be compared with another difficulty function study using a different problem from the same archive [2]. The difficulty surface for the final release versions is shown in Figure 5.

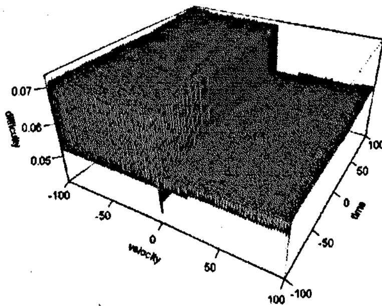


Figure 5: Difficulty function for the final version of the alternate problem.

The equivalent pfd results for the second problem are given in Table 5.

Table 5: Expected pfd's, from [2].

Parameter	Initial version	Final version
pfd_1	0.186	0.064
pfd_2	0.0361	0.0042
pfd_1^2 (independent)	0.0347	0.0041
Ratio	1.04	1.02

It is notable that, in both problems, the dominant failure set in the difficulty surface was a specification problem. A specific sub-domain of the input space ($i > j$ in the first example and $v < 0$ in the second example) was not handled in the correct way. This resulted in a large failure set zone that was present in many different program versions. The other notable feature is that the expected pfd of a diverse pairs is actually quite close to the independence assumption in both examples.

9. Discussion

9.1 Relevance of results

In presenting these exploratory results it is important to note any limitations in their applicability to software engineering in general. There are a number of issues involved in using programs from a contest host site.

- disparities in programmer experience and expertise;
- disparities in the size and complexity of the specifications and the programs;
- disparities in the software development process;
- bias in program submissions, e.g. multiple submissions under different names or by submitting programs produced collectively by multiple people.

As there are no large-scale data sources that are free from such bias, the only way forward is to take account of the limitations and to be careful about what observations can be generalised. In particular, we have attempted to eliminate programmers who do not appear to be competent (judged by the number of submissions). We also know that at the other end of the spectrum, there are some very professional authors who participate in international time-limited competitions under controlled conditions. We hope to get more information on the backgrounds of authors for subsequent analyses.

Despite these precautions it must be recognised that both the specifications and the programs are much smaller than those used in industrial scale software. Also there is no control over the engineering process used to develop individual releases. So the results produced here a more typical of "programming in the small" rather than "programming in the large" and the faults might be similar to those present in a single program module produced by a programmer prior

to verification and validation. These caveats apply to the discussions below.

9.2 Characterisation of faults

Most faults related to poor interpretation of the specification: In particular common faults were related to:

- Not realising that the second input can be smaller than first. This was not mentioned in the specification but the author should not assume otherwise.
- Not realising that returned input values should be in the same order. This is explicitly mentioned in the specification.

It was also notable that almost no faults were found in the mathematical part of problem. Possibly because the algorithm is "homogeneous", i.e. the same algorithm is used regardless of the input value. So if the program works for the sample inputs, it likely works for all inputs.

Most of the implementation faults were related to well known programming "slips", e.g.

- first value of loop forgotten;
- last value of loop forgotten;
- first and last value of loop forgotten;
- initialisation of variable omitted.

In this respect the faults found in the study are similar to those found in more typical software examples [1]. However, they do lack "large program" faults like inconsistent procedure calls and inappropriate use of functions.

One interesting feature of this study is that faults are not arbitrary; there are certain basic faults that appear many times over in different versions. From our exploratory analysis it seems that the majority of equivalence classes are combinations of the basic faults and the failure sets are a superposition of the failure sets of the basic faults. This supports a common assumption in software engineering [7] that software faults can be viewed as separate entities that can be inserted or corrected individually.

9.3 The debugging process

As noted earlier, the debugging environment is atypical because there is no diagnostic feedback to help identify the error. The programmer does not know which of the test values used by the on-line judge resulted in failure, and this information should help to locate the cause. However it appears that this difficulty did not result in the introduction of new faults, it just delayed the removal of faults.

The delay in removal differs from a standard reliability modelling assumption that defects are removed once a failure occurs [6]. This situation could be due to the lack of failure information, and it would be interesting to see what impact additional test result information would have on the debugging process.

On the other hand, the study supports the common assumptions that there is a specific set of faults in the program and the debugging process removes these faults one by one.

9.4 Effectiveness of diversity

The results obtained for the two problems are rather surprising as they predict the reliability improvement could be close to the independence assumption. This result is not supported by other experiments on larger programs, particularly [4]. However we must be careful not to over-interpret our results. In both problems, the difficulty surface is dominated by quite "large" faults that are related to the specification, and one might question whether such large faults would remain in a real software development. It may be better to examine the difficulty surface that would be obtained if we excluded all the large faults (on the assumption that these would be removed by the standard debugging process). On the other hand, one might argue that we might expect a fault to occupy particular sub-domains of the input space, so "flat" difficulty functions over the subdomain might be the norm, even for large programs.

The current study did not attempt to identify different populations that could (potentially) lead to different difficulty surfaces and more effective diversity. Furthermore, another issue we need to consider is that the theories predict the reliability improvement on average. As we have seen in Figure 5, it is possible for two program versions to have identical failure sets and in this case diversity would be ineffective (although a disparity would be detected for different "varieties" where wrong, but different, answers are produced). In other cases, the failure sets could be disjoint or not exist at all. So we need to look at the distribution of possible reliability improvement for the range of equivalence classes. We plan to look at these aspects of diversity in later studies.

10. Conclusions

The analysis of programs submitted to the UVa Online Judge Website gives numerous opportunities for

software engineering research. This paper presents some exploratory findings related to:

- the types of faults that are introduced;
- the debugging process;
- the effectiveness of diversity.

The results tend to support some of the common assumptions made in software engineering such as:

- a distinct set of faults;
- progressive removal of these faults during debugging.

However the study also suggests that other assumptions such as:

- immediate detection and removal of faults;
- large variations in "difficulty" for different input values in diverse programs;

were not supported.

It must be emphasised however that the programs used may not be typical of normal software engineering practice, and further studies are planned to address some of the limitations of the current study and to investigate some the conjectures made in this paper.

Acknowledgements

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council via the Interdisciplinary Collaboration on the Dependability of computer based systems, (DIRC), and via the Diversity with Off-The-Shelf Components Project (DOTS), GR/N24056.

References

[1] B. Beizer, *Bug Taxonomy and Statistics*, Van Nostrand Reinhold, New York, 1990.

[2] G.W. Bentley, P.G. Bishop and M.J.P. van der Meulen, "An Empirical exploration of the Difficulty Function", *Computer Safety, Reliability and Security*, Proceedings of the 22nd international conference, Safecomp 2004, Potsdam, 2004.

[3] D.E. Eckhardt, L.D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors", *IEEE Transactions on Software Engineering*, SE-11 (12), 1985, pp.1511-1517.

[4] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming", *IEEE Transactions on Software Engineering*, SE-12 (1), 1986, pp. 96-109.

[5] B. Littlewood and D.R. Miller, "Conceptual Modelling of Coincident Failures in Multiversion Software", IEEE Transactions on Software Engineering, Vol. 15, No. 2, December 1989, pp. 1596-1614.

[6] B. Littlewood, "Software reliability growth models", *Software Reliability Handbook* (P. Rook, ed.), Elsevier (Amsterdam), 1990, pp. 401-412 (Ch. 16).

[7] M.R. Lyu, *Software Reliability Engineering*, McGraw Hill, 1995.

[8] S. Skiena and M. Revilla, *Programming Challenges*, ISBN: 0387001638, Springer Verlag, March, 2003.

Table 6: Transitions from equivalence classes EC01 to EC36 to equivalence classes EC00 to EC36.

	To EC:																																				
From:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
EC01	429	702	211	15	3	12	3	1	1	5	3	8	3		19		3	1	17	5			2	2	1	4			5		7		4				
EC02	412	15	324	2	1	5	2		4			9	1	1							1					1				2	1						
EC03	97	21	45	174		2	3					1	1	1						1		1		3								1		1			
EC04	83	38	3	8	96	1	3		7								1							1	1									1			
EC05	55	19	9	4	16	117	2		3	1	2	3	1	1	1	1					1									1				1			
EC06	49	3	14		1	1	80					1	1							1				3				1	1								
EC07	32							32						1							1																
EC08	12		12	1	9	1			28																												
EC09	5	26	4			1				14	2				1																	1					
EC10	26	1									24					3					1				1					2							
EC11	10					1						20																									
EC12	12	1	1	1									19		1																						
EC13	5	1												22																							
EC14	3	10	7							1					4										3												
EC15	9		1													21																					
EC16	14					1								1			11																				
EC17		4	2	1				4										12	1								2										
EC18	5	11	3	1		1													1																		
EC19	2	5		2													3			6																	
EC20	8	1					1									1					6																
EC21	1		1	2				3														11															
EC22																							11														
EC23	1		5				2						1											6													
EC24	6	1									1										2				5												
EC25	7	1																								4											
EC26	2		4																								3										1
EC27																												1									
EC28	4	2																											5								
EC29	2		4			1					5																			1							
EC30	5		3																							2					2						
EC31				2												2	1															3					
EC32	5		1																														2				
EC33	3				1																													3			
EC34	4																																		2		
EC35	1	1																																	1	1	
EC36																																					2
Total	1309	863	654	213	127	144	96	40	43	21	37	42	27	27	26	28	19	13	19	13	12	12	13	15	11	11	5	2	11	6	10	5	6	6	3	1	3

C.4 ICCBSS 2005

M.J.P. van der Meulen, S. Riddle, L. Strigini, and N. Jefferson. Protective wrapping of off-the-shelf components. In X. Franch and D. Port, editors, *Proceedings of the 4th International Conference on COTS-Based Software Systems (ICCBSS '05)*, volume 3412 of *Lecture Notes in Computer Science*, pages 168–77, Bilbao, Spain, 2004. ©Springer. With kind permission of Springer Science and Business Media.

Protective Wrapping of Off-the-Shelf Components

Meine van der Meulen[1], Steve Riddle[2], Lorenzo Strigini[1], and Nigel Jefferson[2]

[1] City University, Centre for Software Reliability, London
WWW home page: http://www.csr.city.ac.uk
[2] University of Newcastle, School of Computing
WWW home page: http://www.csr.ncl.ac.uk

Abstract. System designers using off-the-shelf components (OTSCs), whose internals they cannot change, often use add-on "wrappers" to adapt the OTSCs' behaviour as required. In most cases, wrappers are used to change "functional" properties of the components they wrap. In this paper we discuss instead "protective wrapping", the use of wrappers to improve the dependability – i.e., "non-functional" properties like availability, reliability, security, and/or safety – of a component and thus of a system. A wrapper improves dependability by adding fault tolerance, e.g. graceful degradation, or error recovery mechanisms. This paper discusses the rational specification of such protective wrappers in view of system dependability requirements, and highlights some of the design trade-offs and uncertainties affecting system design with OTSCs and wrappers, and differentiating it from other forms of fault-tolerant design.

1 Introduction

As building "component-based" software systems becomes more common, it becomes more often necessary to combine existing components – hardware as well as software – that were not necessarily designed to work together. *Wrapping* is a popular, often cost-effective technique for integrating pre-existing components into a system. When designing a new system, ad hoc "wrappers" are developed, i.e. new, small components that will be interposed between the others, reading and altering the contents of the communications they exchange. Wrapping has the advantage of not requiring detailed knowledge of the internal structure of the components being wrapped.

In most cases, wrappers are used to adapt the functionality of a component to the requirements set for it by the system's design: they often perform simple functions like translation between the argument formats used by two communicating components. In this paper we look instead at the use of wrappers for improving dependability. We call such wrappers *protective* wrappers. Protective wrapping is a way of structuring the provision of standard fault tolerance functions, like error detection, confinement and recovery, plus the less common function of *preventing* component failures, in a component-based design where dependability is a concern. We wish to clarify how these wrappers can be rationally specified, the trade-offs facing system designers (simply "designers" for

the rest of the paper), and the peculiarities of this form of fault-tolerant design, compared to the general case.

When designing a system with off-the-shelf components ("OTSCs"), it is often the case that an OTSC's functionality, and even more often its dependability, is insufficiently documented. Both these deficiencies are threats to system dependability: wrong assumptions about how an OTSC is intended to behave lead to system design faults; optimistic assumptions about an OTSC's probability of behaving as intended may lead to overestimating the dependability levels achieved by the chosen system design. Wrapping can help a designer to compensate for this lack of information.

Wrapping for dependability has been addressed by other authors. Wrappers are used to transform or filter unwanted communications that may cause failures. Fault injection may be used to identify such failure-causing values [1,6,3]. Wrappers are proposed to protect OTS applications that do not deal properly with kernel-raised exceptions, by transforming these into other exceptions or error return codes [1]; or to protect OTS kernels against inappropriate requests ([6]; here, an extended notion of wrappers is proposed that can access the kernel's internal data). In [3], the goal is automatic protection of library components against failure-causing parameter values, submitted by accident or malice. In [2], wrappers protect name servers from receiving unverifiable requests. A somewhat general approach to wrappers for common security concerns is described in [4].

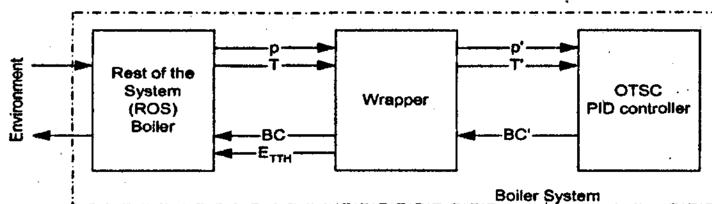
Most of this previous work assumes that a good knowledge can be gained about which communications will cause OTSC failure. We propose a more general view of protective wrapping, taking into account the fact that this knowledge is usually deficient, the specification of the OTSC may be incomplete, and a designer need to be concerned with both failures of the OTSC and of the ROS. We discuss issues of design, verification and quantitative dependability trade-offs.

In the rest this paper, Section 2 introduces terminology and an illustrative example of a system using wrapped OTSCs. Section 3 introduces the specifications of components in relation to system-level requirements, including those concerning fault-tolerant behaviour. Sections 4 and 5 discuss the actual semantics of wrappers, i.e. the "cues" that may trigger their intervention and the forms of these interventions, in general terms and then through examples. Section 6 sets the previous discussion of wrapper specifications in the context of probabilistic system dependability requirements and discusses the important design trade-offs that arise. Our conclusions follow.

2 System Model and Example

Throughout the paper, we will use a simple example to clarify the concepts introduced. The example system (Figure 1) is a water boiler. We focus on a single OTSC, in this case a PID (Proportional-Integral-Derivative) controller which provides feed-back control for the burner of the boiler, and on its communications with the rest of the system, seen as a single black box ("ROS"); the ROS may contain other OTSCs. This example omits some of the possible compli-

Fig. 1. The boiler control system used as an example.



cations of a real system (an OTSC may have direct communication links with the environment around the system, or communication links with the ROS that cannot be intercepted by a wrapper) but will suffice for this brief discussion.

The ROS outputs readings of the pressure and temperature in the boiler, (p, T), and accepts a burner control input, BC', and an exception signal, E_{TTH}, which causes an alarm to a human operator. The OTSC accepts as inputs two real numbers (p', T') and a *reset* signal, and outputs a (real) control signal for the burner, BC'.

The designer is concerned with the dependability of this system: how frequently the components will behave "abnormally" (will fail), whether these component failures will cause system failure, and whether the frequency and severity of these failures will be acceptably low. Because of this concern, instead of connecting the ROS outputs directly to the OTSC's inputs and vice versa, the designer introduces a protective wrapper between the ROS and the OTSC, as depicted, which transforms p into p', etc.

The wrapper monitors communications between the ROS and OTSC, and possibly changes the values transmitted to the ROS or the OTSC. The ROS sees the combination of the OTSC and wrapper as one component, which we call the "wrapped OTSC" (WOTSC); likewise, the OTSC sees a "wrapped ROS".

For the sake of simplicity, we assume here that the OTS and ROS, if connected without the protective wrapper, would, in the absence of failures, produce the combined behaviour required from the system. So, the OTSC in Figure 1 does not need "functional" wrapping, limiting our discussion to "protective" wrapping.

Our discussion does not depend on whether the OTSC, ROS and wrapper are realised in hardware or software (or both).

3 Roles of Components and Protective Wrappers

3.1 System requirements, components and interfaces

The designer's problem is how to ensure the required behaviour of the whole system, using a given OTSC. When considering dependability, a designer usually deals with multiple sets of requirements on system behaviour. First, there is a specified "nominal behaviour": what the system ought to do, at least if none of

its components fail. The designer usually has an understanding of a "nominal behaviour" for each component, and makes sure that if all components exhibit their nominal behaviours, then so will the system. Making the system fault-tolerant means ensuring that even if components violate their nominal behaviours (they fail), the system will still exhibit nominal behaviour (failure masking) or some "degraded" but acceptable behaviour (graceful degradation) or at least will remain within an envelope of "safe behaviours"; the choice being determined by the system dependability requirements and by the costs of these various options.

The complete dependability requirements will inevitably be probabilistic: in addition to defining a nominal behaviour and zero or more degraded behaviours or "modes" of operation, it will include required probabilities of these assertions holding during operation of the system[1]. A similar hierarchy of a nominal behaviour and more or less acceptable failure behaviours applies to dependability requirements for any component or subsystem.

In this and the next two sections, we will discuss the deterministic part of these dependability properties. In a proper design, the specified system-level properties need to be *verifiable*, in the sense that, given clear descriptions (*models* in what follows) of how the various components will behave (in their nominal and degraded modes) and of their connections, one can deduce that the requirements for the whole system (for a nominal or degraded mode, as specified) are satisfied. The required and expected behaviours of the components and of the system need to be described in some unambiguous language, e.g., preconditions and postconditions characterising the relation between their inputs and outputs [7].

These descriptions need not specify all details of behaviour of a component, i.e. they may be partial specifications. We might for instance describe a component in a numerical library as computing a certain floating-point result with a relative error of less than 1%, although in reality the relative error is smaller, and variable; or, rather than trying to describe in detail what a component would do if it failed, we would rather describe an envelope of plausible behaviours it may exhibit, and prove that some system-level requirement will be satisfied provided the component remains within that envelope.

The behaviour that the designer expects the OTSC, as procured, to exhibit can be described abstractly as pairs of pre and post-conditions [7]. The looser the postconditions (the fewer the restrictions assumed on the behaviour of the OTSC), the more arbitrary behaviours of the OTSC one will need to require the wrapper and ROS to cope with in order to guarantee any given system-level requirement. This may make the system more robust, but at a cost, which will be the more acceptable, the more likely the extra erroneous behaviours allowed by

[1] It is true that such a formal way of specifying dependability requirements is only in common use for few categories of systems. For many everyday systems, probabilities may not be mentioned at all, for instance. Yet, we think that any rational definition of requirements will include some idea of what probabilities would be unacceptably high for each given failure mode, and a partial ordering between more and less acceptable modes.

the less restrictive model of the OTSC are in reality. Symmetrical considerations apply to the designer's expectations about the ROS's behaviour.

3.2 The Model of the OTSC

We assume that the designer has chosen a particular OTSC, either procured on the market or already available within the same company. If the OTSC has been procured on the market, some documentation will be available, but its quality may be low and procuring extra information is often cumbersome and expensive. On the other hand, the component may be in frequent use, and suppliers sometimes have reliable data on their dependability.

For some OTSCs, publicly available dependability data exist – e.g., collections of bug reports for software packages – and may provide valuable input Relevant other information about the OTSC may concern maintenance requirements, failure modes and their failure rates.

The documentation of the OTSC may not specify its behaviour in certain circumstances, and the designer's most prudent approach would then be to assume that it is completely undetermined. At the opposite extreme, designers may choose to guess the OTSC's behaviour, based on previous experience, expert knowledge or other information.

Boiler example A PID controller has been chosen. Its documentation is unclear about what happens when either p or T becomes negative. The designer assumes that its behaviour is undefined for these values in his OTSC model.

3.3 Requirements on the Wrapped OTSC

The designer's specification for the WOTSC may differ from the model of the OTSC even in its nominal behaviour, e.g. by hiding from the ROS some of the functions offered by the OTSC. In addition, the WOTSC specification has to describe dependability requirements, which determine the need for fault tolerance provisions in the wrapper.

Boiler example The boiler needs from the PID controller a control signal, BC, derived from the pressure and the temperature of the boiler according to a PID control law. If either p or T becomes negative, the burner has to be switched off ($BC = 0$), as a system safety precaution.

3.4 The model of the ROS

The designer can control the properties of the ROS to a very large extent, this in contrary to the ROS. As already stated: the looser the postcondition (the fewer restrictions one assumes on the behaviour of the ROS), the more arbitrary behaviours of the ROS the OTSC has to cope with.

Boiler example The designer may choose to pose no restriction at all on the outputs p and T. In this case, if the OTSC requires p and T to be always positive, the wrapper has to guarantee this property.

3.5 Requirements on the (Wrapped) ROS

The OTSC sees the ROS through the wrapper. The wrapper is introduced to protect the OTSC against violations of its input requirements by the ROS. The more robust the OTSC, the fewer restrictions it requires on the interface behaviour it sees, and the less the wrapper is required to do.

Boiler example The OTSC requires p and T to be positive.

4 Specifying the Protective Wrapper: Cues for Action

Usually, the designers of a fault-tolerant system use the detection of errors to trigger defensive actions. This depends on a fairly accurate knowledge of the behaviour of all components when failure-free. In designing with OTSCs, though, this knowledge cannot be assumed. Furthermore, the design of OTSCs often precludes close monitoring for early error detection. So, designers may want their wrappers to react to a pattern of component behaviour that merely suggests a failure, although it may be correct, especially if the type and circumstances of the suspected failure would cause severe consequences to the system.

So, designers may take an attitude similar to that frequently taken in designing for safety: aiming more at keeping the behaviour of components within an envelope of behaviours that prevent unacceptable damage at system level, than at guaranteeing correct (nominal) behaviour of the components. They also face the same kind of trade-offs: the interventions of the wrapper will usually prevent some requested operation of the OTSC, possibly providing in its place a safe failure, or an alternative, degraded or less efficient service. Designers thus know that the more cues they decide to react to, the less likely that the system will fail in unpredictable ways, but also the more likely for any wrapper intervention to be the result of a false alarm, and the more degradation in performance or availability.

The wrapper, as depicted in Figure 1, can observe the outputs of the ROS and of the OTSC, and can manipulate the corresponding inputs to the OTSC, and to the ROS. These signals may include exceptions, indicating error conditions.

The wrapper gets its cues by monitoring signals from the ROS and the OTSC. It checks for violations of properties of (combination of) signals (possibly depending on the previous history of the signals), which define either requirements on the various components' behaviours, or bounds of the region of operation of the OTSC in which the OTSC is trusted to behave reliably. In the wrapper's specifications, pre-conditions describing the possible cues will be matched with postconditions about the actions the cues must elicit from the wrapper.

5 Examples of Specifications for Wrapper Actions

For any given cue, the designer may choose among various possible reactions by the wrapper, depending on the system's architecture and dependability requirements. For instance, assume that the postcondition of the ROS states that the

pressure output will always be positive. If a failure leads to a negative value, this also violates a precondition for the PID controller, whose behaviour is then unspecified. The wrapper could mitigate the consequences of such a failure by *substituting this erroneous, dangerous or suspicious signal value with other values*: e.g. if $p < 0$, p' will be 0. This keeps the PID controller in a region of operation for which its behaviour is predictable. This may not ensure correct *system* behaviour, but it may be sufficient protection e.g. against noise spikes on sensor readings, given the robustness of the PID control law.

In many cases, though, it is not judged useful to correct a "suspicious" input value to the ROS or the OTSC. It may still be possible to prevent harm by checking and if necessary correcting their subsequent outputs. Suppose that a failure causes "suspicious" values of p. The designer may decide that the wrapper will then perform additional plausibility checks on the output of the PID controller. If the checks fail, the wrapper could ensure *graceful degradation* by providing a simpler version of the OTSC's (or ROS's) function. The designer might specify this kind of switch if the degraded control were proven to keep the boiler in an acceptable degraded mode of operation for as long as the OTSC cannot be trusted to perform correctly.

All these palliative measures may only be acceptable for a short time. A reaction can be for the wrapper enforce at least safe system-level behaviour, switching the burner off ($BC' = 0$): this is an extreme form of graceful degradation suitable for all undesired situations.

Another possibility is *error recovery*. In many OTSCs it is an established fact that when they fail, a reset is sufficient to restore an internal state such that the OTSC will subsequently function according to its specifications. In our example, the wrapper could reset the PID controller (OTSCS) if its output is clearly out of bounds. Reset erases the OTSC's memory of previous history: it does not generally guarantee that its future behaviour will be correct *from a system viewpoint*, but it may in a control system like our example, if the designer can demonstrate that the internal state of the OTSC will then return to a correct state (through the OTSC reading and processing its inputs) quickly enough for the boiler not to be affected badly by the transient.

More complex recovery actions can be specified. If, for instance, an OTSC has an "undo" operation, the wrapper could use it for *backward recovery and retry*; a wrapper could store sequences of input messages to an OTSC and replay them after recovery, possibly even with slight variations to reduce the risk of repeated failure ("retry blocks" architecture [5]). The possibilities here are bounded by the risk implicit in increasing the complexity of the wrapper, and thus the risk of specification or implementation errors. For instance, designers may often limit themselves to stateless wrappers.

A wrapper can also deal with *exception* signals. Often failure of the OTSC is not catastrophic and can be recovered by actions of other system components or by human intervention. In these cases the wrapper may generate exception signals. For instance, the wrapper can be used to generate an exception signal to the ROS, E_{TTH}, when the temperature becomes too high.

Last, many of the actions described so far may not be effective, e.g. if the cue to which they react is caused by a permanent or recurrent fault. If this event is considered too likely, wrappers may be designed to escalate to more drastic and safer actions (multi-level recovery). E.g., once it has entered a "graceful degradation" state, a wrapper may be programmed to become sensitive to cues that it would otherwise ignore, and trigger a more drastic action if any of these cues occurs. After the wrapper has reset the PID controller, it may wait for a pre-set length of time for it to restart issuing apparently correct control signals again, after which it may shut down the boiler ($BC = 0$). The wrapper could also or instead count its own interventions, and escalate to a more drastic one if they become too frequent. Again, designers will need to judge at which point the added complexity becomes counterproductive.

6 Probabilistic Dependability Properties

Up to this point, we have approached wrapper design mostly from a deterministic viewpoint: the designer considers the possibility of certain unplanned-for sequences of actions of the OTSC or ROS, and specifies the wrapper so that it will mask or alter those behaviours in ways that appear desirable, to achieve one of the specified nominal or degraded modes of operation.

This desirability must be determined in view of the system-level dependability requirements, which are inevitably, in their general form, probabilistic, as outlined in Section 3.

A wrapper may be meant to avoid a system failure, i.e. to increase the probability of nominal behaviour, or to mitigate it, i.e. to shift some probability from more severely to less severely degraded behaviours.

As always with fault tolerance, wrapping faces two kinds of trade-offs, i.e. between the improvement in dependability that it produces by avoiding or mitigating some failures, and (i) its direct costs, in development effort and in terms of run-time resources; and (ii) the dependability loss that wrappers cause by *causing* failures or making them more severe.

Costs are generally the easiest factor to estimate. Estimating dependability improvements may be difficult, especially when a system is already reasonably dependable before the improvement. Even with this uncertainty, designers will think it reasonable to provide abilities at least to deal with predictable component failures that have a clear potential for severe effects and can be avoided or tolerated at low cost. This appears to be the approach, for instance, of the HEALERS project [3]. In other cases, system failures due to specific failure modes of OTSCs are observed often enough that it is easy to assess the expected effect of avoiding them.

The second trade-off is also complex. There are two ways for a wrapper to cause failures. First, it may simply be faulty, and deliver a wrong output towards the ROS despite having received a correct OTSC output. This could be due to mistakes in developing the wrapper, which are always possible, especially for complex wrappers. In many design situations this problem will not arise,

because very complex wrappers are not affordable; otherwise, the designer must decide how sophisticated the wrapper may be specified to be before this very sophistication becomes counterproductive. To make the transition less sharp, it may be worthwhile to seek wrapper design techniques that bias wrappers towards "benign" failures, whose consequences can be assessed, rather than uncontrolled ones, e.g. injecting arbitrary values in a communication stream.

But, more importantly, a wrapper may also cause failure because of the designer's direct decision to behave prudently in response to a "cue". The designer may wish to specify wrappers to prevent as many unpredicted, uncontrolled behaviours as possible. However, this may mean that many occurrences of "cues" will be "false alarms", triggered by communications that are neither erroneous (for the sender) nor dangerous (for the receiver); and many systems will differ from our boiler example in that the effects of wrapper interventions on the behaviour of the whole system will be complex to trace. If, for instance, a wrapper changes any message values from the ROS that *may* cause uncontrolled OTSC failure to default values for which the OTSC's behaviour is known, it becomes easier to deduce what level of service (nominal, or which degraded level) the system will exhibit after the ROS outputs those values; but not to know whether this is better or worse than what would have occurred without the wrapper's intervention, and thus whether, statistically, dependability will be improved.

Uncertainty also affects the failure of wrappers to react effectively to component failures. In many practical uses of protective wrapping, the main cause of such wrappers "failures" may not be, as it is for other components, development mistakes or hardware failures, but rather inescapable limits of the algorithms that the designer can feasibly apply in the wrapper. Error detection, for example, often depends on reasonableness checks, which cannot flag values that are erroneous but "reasonable"; algorithms that guarantee perfect state recovery may take too long for the real-time requirements of the system, or require more resources than can be made available[2]. The designer may, on the other hand, have some control on the stringency of the checks applied for error detection, and decide, as indicated before, to err on the side of "prudence".

7 Conclusion

We have tried to clarify some issues concerning *protective* wrapping. Protective wrappers are components that monitor and ensure the non-functional properties at interfaces between components. We have described the role of wrapping as a special case of fault-tolerant design, from both the viewpoints of deterministic and of probabilistic dependability properties.

These considerations should help designers in specifying wrappers, using the spectrum of fault-tolerance techniques within the special constraints of wrapping

[2] So, designers may know with almost complete certainty which component failure modes the wrappers will not detect or tolerate. Unfortunately, they still do not usually know the frequency of these failure modes, so that the uncertainty on the actual dependability improvement achieved by wrapping is not resolved.

as a design structuring scheme. These peculiarities are not always acknowledged in previous literature. The main considerations we have made are: wrappers can be rigorously specified on the basis of the designers' specification of the OTSC's behaviours in its possibly multiple modes of operation: from nominal, correct behaviour to manageable, non catastrophic failure modes. Also, due to poor ability to detect run-time errors inside off-the-shelf components, protective wrappers may have to act on "cues" of potentially erroneous and/or potentially error-causing communications between components. All of this increases the importance of design trade-offs between reducing the probabilities of the more dangerous system failure modes and avoiding too frequent false alarms leading to degraded service or " safe" system failures.

Research developments that appear desirable concern formal proof, probabilistic modelling and experimental evaluation. Formal proof methods are desirable that are simple to apply to the restricted sets of structures defined by wrapping and the kinds of properties it involves. Probabilistic modelling should support designers in choosing trade-offs as discussed here; this modelling must include both the structural aspects of how component failures cause system failure, aspects that are well developed in modelling of hardware fault tolerance, and the uncertainty on the reliability of the individual components and their probabilities of failing together, as studied in software reliability and the assessment of software diversity. And finally, experimental evaluation of systems using wrapping is required, to document the levels of coverage and of system dependability achieved with various classes of wrapper designs and of OTSC components and thus some basis for rational, probabilistically based decisions.

References

1. Ghosh, A.K., M. Schmid, F. Hill, *Wrapping Windows NT Software for Robustness*, Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing. June 15 - 18, Madison, Wisconsin, USA, 1999.
2. Cheung, S., K. N. Levitt, *A Formal-Specification Based Approach for Protecting the Domain Name System*, International Conference on Dependable Systems and Networks (DSN 2000), June 25-28, New York, 2000.
3. Fetzer, C. and Z. Xiao. "HEALERS: A Toolkit for Enhancing the Robustness and Security of Existing Applications", in DSN 2003, International Conference on Dependable Systems and Networks, 2003.
4. T. Fraser, L. Badger, and M. Feldman, "Hardening COTS Software with Generic Software Wrappers," 1999 IEEE Symposium on Security and Privacy, Oakland, USA, 1999.
5. Ammann, P. E., J. C. Knight, *Data Diversity: An Approach to Software Fault Tolerance*, IEEE Transactions on Computers, C-37, 1988, pp. 418-425.
6. J. Arlat, M. R. J.-C. Fabre, F. Salles, *Dependability of COTS Microkernel-Based Systems'*, IEEE Transactions on Computers, vol. 51, pp. 138-163, 2002. In: Proceedings of the 27th Euromicro Conference, Warsaw, Poland, 4-6 September 2001, pp. 22-29. IEEE Computer Society Press, 2001.
7. Meyer, B., *Design by Contract*, in Computer (IEEE), vol. 25, no. 10, pages 40-51, October 1992.

C.5 EDCC 2005

M.J.P. van der Meulen and M.A. Revilla. The effectiveness of choice of programming language as a diversity seeking decision. In *Proceedings of the 5th European Dependable Computing Conference (EDCC-5)*, volume 3463 of *Lecture Notes in Computer Science*, Budapest, Hungary, April 2005, ©Springer. With kind permission of Springer Science and Business Media.

The effectiveness of choice of programming language as a diversity seeking decision

Meine J.P. van der Meulen[1] and Miguel Revilla[2]

[1] Centre for Software Reliability, City University, London
WWW home page: http://www.csr.city.ac.uk
[2] Department of Applied Mathematics
University of Valladolid, Spain
WWW home page: http://www.mac.cie.uva.es/~revilla/

Abstract. Software reliability can be increased by using a diverse pair of programs (1-out-of-2 system), both written to the same specification. The improvement of the reliability of the pair versus the reliability of a single version depends on the degree of diversity of the programs. The choice of programming language has been suggested as an example of a diversity seeking decision. However, little is known about the validity of this recommendation. This paper assesses the effect of language on program diversity.

We compare the effects of the choice of programming language as a diversity seeking decision by using programs written to three different specifications in the "UVa Online Judge". Thousands of programs have been written to these specifications; this makes it possible to provide statistical evidence.

The experiment shows that when the average probability of failure on demand (pfd) of the programs is high, the programs fail almost independently, and the choice of programming language does not make any difference. When the average pfd of the pools gets lower, the programs start to fail dependently, and the pfd of the pairs deviates more and more from the product of the pfds of the individual programs. Also, we observe that the diverse C/Pascal or C++/Pascal pairs perform as good as or better than the other possible pairs.

1 Introduction

The use of a diverse pair of programs has often been recommended to achieve high reliability [1] [2] [3] [4] [5]. Software diversity may however not lead to a dramatically high improvement. This is caused by the fact that the behaviour of the programs cannot be assumed to be independent [3] [5] [6] [7]. Two program versions written by independent teams can still contain similar programming mistakes, thus limiting the gain in reliability of the diverse pair.

In spite of this, the case for diversity for achieving high reliability remains strong. The possible gain using diversity appears to be higher than can be achieved by trying to write a high reliability single program [6].

Several techniques have been proposed to decrease the likelihood that different programs fail dependently. These are called "Diversity Seeking Decisions" in [9]. Examples are:

- **Data diversity.** Using random perturbations of inputs; using algorithm specific re-expression of inputs.
- **Design diversity.** Separate ("independent") development; diversity in programming language; diverse requirements/specifications; different expressions of identical requirements; etc.

In this paper we will concentrate on design diversity and specifically on programming language diversity. This is a potential defence against some programming slips, and provides some, limited, cognitive diversity against mistakes in higher-level problem solving; the efficacy will however depend heavily on "how different" the programming languages are.

The "UVa Online Judge"-website (`http://acm.uva.es`) provides many programs written to many specifications, and gives us the opportunity to compare diverse pairs. In this research we use the programs written in C, C++ and Pascal, written to three different specifications. Our aim is to compare the reliability performance of diverse pairs with each other and with single programs.

2 The experiment

2.1 The UVa Online Judge

The "UVa Online Judge"-Website is an initiative of Miguel Revilla of the University of Valladolid [10]. It contains problems to which everyone can submit solutions. The solutions are programs written in C, C++, Java or Pascal. The correctness of the programs is automatically judged by the "Online Judge". Most authors submit solutions until their solution is judged as being correct. There are many thousands of authors and together they have produced more than 3,000,000 solutions to the approximately 1500 problems on the website.

In this paper we will analyse the programs written to three different problems on the website. We will submit every program to a test set, and then compare their failure behaviour.

There are some obvious drawbacks from using this data as a source for scientific analysis. First of all, these are not "real" programs: the programs under consideration solve small, mostly mathematical, problems. We have to be careful to not overinterpret the results.

Another point of criticism might be the fact that the Online Judge does not give feedback on the demand on which the program failed. This is not necessarily a drawback. It is certainly not comparable to a development process involving a programmer and a tester, because in that case there will be feedback on the input data for which the program fails. It has however similarities with a programmer's normal development process: a programmer will in spite of the fact that there are no examples of inputs for which a program fails, assume that it is not yet

correct. The programmer works until he is convinced that the program is correct, based on his own analysis and testing. From this perspective, the Online Judge only confirms the programmer's intuition that the program is not yet correct. In this experiment, we circumvent this drawback by only using first submissions.

A last possible criticism on our approach is that programmers may copy each other's results. This may be true, but it is possible to limit the consequences of this plagiarism for the analyses by assuming that authors will only copy correct results from each other. For the analyses in this paper, the consequence is that we cannot trust absolute results, and we will limit ourselves to observing trends in relative performance.

2.2 Problems selected

We selected problems from conforming to the following criteria:

- The problem does not have history, i.e. subsequent inputs should not influence each other. Of course, some programmers may implement the problem in such a way that it has history. Given our test approach, see below, we will not detect these kinds of faults.
- The problem has a limited demand space: two integer inputs.

Both restrictions lead to a reduction of the size of the demand space and this keeps the computing time within reasonable bounds (the necessary preparatory calculations for the analysis of these problems take between a day and two weeks to complete).

Below, we provide a short description of the problems, although this information is not necessary for reading this paper: we will not go into detail with respect to functionality. It gives some idea of the nature and difficulty of the problems, which is useful for interpreting our results. See the website `http://acm.uva.es` for more detailed descriptions of the problems.

The "3n+1"-Problem. A number sequence is built as follows: start with a given number; if it is odd, multiply by 3 and add 1; if it is even, divide by 2. The sequence length is the number of these steps to arrive at a result of 1. Determine the maximum sequence length for the numbers between two given integers $0 < i, j \leq 100,000$.

The "Factovisors"-Problem. For two given integers $0 \leq i, j \leq 2^{31}$, determine whether j divides factorial i.

The "Prime Time"-Problem. Euler discovered that the formula n^2+n+41 produces a prime for $0 \leq n \leq 40$; it does however not always produce a prime. Write a program that calculates the percentage of primes the formula generates for n between two integers i and j with $0 \leq i \leq j \leq 10,000$.

2.3 Running the programs

For all problems chosen, a "demand" is a set of two integer inputs. Every program is restarted for every demand; this is to ensure the experiment is not influenced

Table 1. Some statistics on the three problems.

	3n+1 C	3n+1 C++	3n+1 Pas	Factovisors C	Factovisors C++	Factovisors Pas	Prime Time C	Prime Time C++	Prime Time Pas
Number of authors	5897	6097	1581	212	582	71	467	884	183
First attempt correct	2483	2442	593	113	308	42	356	653	127
First version completely incorrect	723	761	326	27	97	9	93	194	49

by history, e.g. when a program crashes for certain demands. We set a time limit on each demand of 200 ms. This time limit is chosen to terminate programs that are very slow, stall, or have other problems.

Every program is submitted to a series of demands. The outputs generated by the programs are compared to each other. Programs that produce exactly the same outputs form an "equivalence class". These equivalence classes are then converted into score functions. A score function indicates which demands will result in failure. The difference between an equivalence class and its score function is that programs that fail in different ways (i.e. different, incorrect outputs for the same demands) are part of different equivalence classes; their score functions may however be the same. The score functions are used for the calculations below.

For all three problems, we chose the equivalence class with the highest frequency of occurrence as the oracle, i.e. the version giving all the correct answers.

"3n+1" and "Factovisors" were run using the same set of demands: two numbers between 1 and 50, i.e. a total of 2500 demands. In both cases the outputs of the programs were deemed correct if they exactly match those of the oracle.

"Prime Time" was run using a first number between 0 and 79, and a second number between the first and 79, i.e. a total of 3240 demands. The outputs of the programs were deemed correct if they were within 0.01 of the output of the oracle, thus allowing for errors in round off (the answer is to be given in two decimal places).

Table 1 gives some statistics on the problems.

3 Effectiveness of diversity

Two of the most well known probability models in this domain are the Eckhardt and Lee model [3] and the Littlewood and Miller extended model [7]. Both models assume that:

1. Failures of an individual program are deterministic and a program version either fails or succeeds for each input value x. The failure set of a program π can be represented by a "score function" $\omega(\pi, x)$ which produces a zero if the program succeeds for a given x or a one if it fails.
2. There is randomness due to the development process. This is represented as the random selection of a program, Π, from the set of all possible program versions that can feasibly be developed and/or envisaged. The probability

that a particular version π will be produced is $P(\Pi = \pi)$. This can be related to the relative numbers of programs in the pools.

3. There is randomness due to the demands in operation. This is represented by the random occcurrence of a demand, X, from the set of all possible demands. The probability that a particular demand will occur is $P(X = x)$, the demand profile. In this experiment we assume a contiguous demand space in which every demand has the same probability of occurring.

Using these model assumptions, the average probability of a program version failing on a given demand x is given by the difficulty function, $\theta(x)$, where:

$$\theta(x) = \sum_{\pi} \omega(\pi, x) P(\Pi = \pi) \tag{1}$$

The average probability of failure on demand (pfd) of a randomly chosen single program version, can be computed using the difficulty function and the demand profile, in our case for two program versions, Π_A and Π_B:

$$\text{pfd}_A := \sum_{x} \theta_A(x) P(X = x), \quad \text{pfd}_B := \sum_{x} \theta_B(x) P(X = x) \tag{2}$$

The Eckhardt and Lee model assumes similar development processes for the two programs A and B and hence identical difficulty functions: $\theta_A(x) = \theta_B(x)$. So the average pfd for a pair of diverse programs (assuming the system only fails when both versions fail, i.e. a 1-out-of-2 system) would be:

$$\text{pfd}_{AB} := \sum_{x} \theta_A(x) \theta_B(x) P(X = x) = \sum_{x} \theta_A(x)^2 P(X = x) \tag{3}$$

And:

$$\text{pfd}_{AB} = \text{pfd}_A^2 + var_X(\theta_A(X)) \tag{4}$$

The Littlewood and Miller model does not assume that the development processes are similar, and thus allows the difficulty functions for the two versions, $\theta_A(x)$ and $\theta_B(x)$, to be different. Therefore, the probability of failure of a pair remains:

$$\text{pfd}_{AB} := \sum_{x} \theta_A(x) \theta_B(x) P(X = x) \tag{5}$$

And:

$$\text{pfd}_{AB} = \text{pfd}_A . \text{pfd}_B + cov_X(\theta_A(X),\, \theta_B(X)) \tag{6}$$

In this experiment, we wish to investigate the reliability improvement gained by choosing different programming languages for the programs in the pair, and we therefore need to use the Littlewood and Miller model.

First, we establish pools of programs, each pool containing programs in C, C++ or Pascal. For comparison of the individual programs and pairs, we need pools with the same pfd. To manipulate the pfd of the pools, we remove programs from them, starting with those with the highest pfd, until the average pfd of the

pool has the desired value. This is a possible way of simulating testing of the programs; the tests remove the programs with the highest pfd first. Pools with the same pfd could then be assumed to have undergone the same scrutiny of testing.

We select a first program from one of the pools. Then we select a second program from a pool, and calculate the ratio of the pfd of the first program and the pfd of the pair:

$$R = \frac{\mathrm{pfd}_A}{\mathrm{pfd}_{AB}} = \frac{\sum_x \theta_A(x) P(X = x)}{\sum_x \theta_A(x)\theta_B(x) P(X = x)} \tag{7}$$

We do so for varying values of the pfd of the pools. The varying pfd is shown on the horizontal axis in the graphs.

Figures 1, 2 and 3 show these ratios for the three problems for different choices of the programming language of the first program. The graphs show R on the vertical axes on a logarithmic scale, because we are interested in the reliability improvement; with a logarithmic scale, equal improvements have equal vertical distance.

4 Analysis and Discussion

All graphs clearly show that for pairs of programs with higher pfds (pfd $> 10^{-1.5}$) the choice of programming language does not make any difference. The pfd of the pair is fairly close to the product of the pfds of the individual programs. This indicates that the programs fail almost independently.

For the lower pfds (pfd $< 10^{-2}$) the pfd of the pair deviates more and more from the product of the pfds of the individual programs, the programs fail dependently. The reliability improvement reaches a "plateau" at between one to two orders of magnitude better than a single version. This is in accordance with the generally accepted assumption that the gain from redundancy is limited, and is certainly not simply the product of the pfds of the individual programs (e.g. in IEC61508 [13] the reliability improvement one can claim for applying redundancy is one SIL, i.e. a factor of 10; Eckhardt and Lee reach a comparable conclusion in [4]).

If we now compare the effect of choice of programming language in the pairs, we can observe that the C/Pascal and the C++/Pascal pairs almost always outperform the other pairs, most notably also the C/C++ pairs. The effect is most clearly visible in the "Factovisors"-problem, and in the middle region (10^{-3} < pdf < 10^{-2}) of the "3n+1"-problem. It is also visible in the "Prime Time"-problem, but in this case it is only visible in a small, unreliable, region of the graph. This graph is unreliable for pfd $< 10^{-2.5}$, because the pool of Pascal programs is almost empty.

The Littlewood and Miller model provides a description of diversity, in which the reliability change compared to the independence assumption is given by a covariance term, see Equation 6. Since the model cannot predict the shapes of the difficulty functions in C, C++ or Pascal, the model can not predict how

large this change will be. The model however provides an intuition that when programmers make different faults in different programming languages their respective difficulty functions will be different. These differences could then lead to a better performance of diverse language pairs.

Analysis of the differences in programming faults of the "3n+1"-problem shows that faults in programming "for"-loops in Pascal are rare compared to C and C++. This accounts for the reliability improvement of Pascal/C and Pascal/C++ pairs in the middle pfd-region.

In the low-pfd region, pfd $< 10^{-4}$, we observe for "3n+1" that the reliability improvements of the pairs become approximately the same, whereas for "Factovisors" the diverse language pairs (C/Pascal and C++/Pascal) show an enormous improvement, even approaching infinity for the lowest pfd-values. This observation gives rise to some thoughts[3]

First, these observations in the low-pfd region confirm the theoretical result of the Littlewood and Miller model that the reliability of a diverse pair can be better than under the independence assumption.

Second, why do we observe this effect? As explained above, the pfd of a pool is established by subsequently removing the most unreliable programs. For low pfd-values this approach leads to a monoculture, and in the end only few different program behaviours will be present in the pool. In the case of "3n+1" these program behaviours happen to be roughly the same. In the case of "Factovisors" however, the last programs in the Pascal pool fail on different demands than those left in the C and C++ pools, thus leading to an enormous improvement in the reliability of the pair.

This result should be considered with care, because it could be an artefact of our experiment, caused by the way in which we establish a given pfd for a pool. On the other hand, a normal debugging process will have a similar effect: eliminating some behaviours, starting with the most unreliable ones, thus eventually also leading to a monoculture of behaviours.

5 Conclusion

We analysed the effect of the choice of programming language in diverse pairs using three different problems from the "UVa Online Judge". The results seem to indicate that diverse language pairs outperform other pairs, but the evidence is certainly not strong enough for a definite conclusion. Analysis of more problems could help to strengthen the evidence and also to identify the factors that influence the gain possibly achieved by diversity of programming language.

Acknowledgement

This work is supported by EPSRC/UK in the framework of the *Interdisciplinary Research Collaboration in Dependability*-Project (DIRC).

[3] It has to be noted here that the amount of Pascal programs in this pfd-region for "Factovisors" is rather low, and the graph has to taken cum grano salis.

References

1. Brilliant, S.S., J.C. Knight, N.G. Leveson, *Analysis of Faults in an N-Version Software Experiment*, IEEE Transactions on Software Engineering, SE-16(2), pp. 238-47, February 1990.
2. Voges, U., *Software diversity*, Reliability Engineering and System Safety, Vol. 43(2), pp. 103-10, 1994.
3. Eckhardt, D.E., L.D. Lee, *A Theoretical Basis for the Analysis of Multi-Version Software Subject to Coincident Errors*, IEEE Transactions on Software Engineering, Vol. SE-11(12), pp. 1511-1517, December 1985.
4. Eckhardt, D.E., A.K. Caglayan, J.C. Knight, L.D. Lee, D.F. McAllister, M.A. Vouk, J.P.J. Kelly, *An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability*, IEEE Transaction on Software Engineering, Vol. 17, No. 7, July 1991.
5. Knight, J.C., N.G. Leveson, *An Experimental Evaluation of the Assumption of Independence in Multiversion Programming*, IEEE Transaction on Software Engineering, Vol. SE-12(1), pp. 96-109, 1986.
6. Hatton, L., *N-Version Design Versus One Good Version*, IEEE Software, 14, pp. 71-6, 1997.
7. Littlewood, B., D.R. Miller, *Conceptual Modelling of Coincident Failures in Multiversion Software* IEEE Transactions on Software Engineering, Vol. 15, No. 2, pp. 1596-1614, December 1989.
8. Lyu, M.R., *Software Reliability Eningeering*, McGraw Hill, 1995.
9. Popov, P., L. Strigini, A. Romanovsky, *Choosing Effective Methods for Design Diversity - How to Progress from Intuition to Science*, In: Proceedings of the 18th International Conference, SAFECOMP '99, Lecture Notes in Computer Science 1698, Toulouse, 1999.
10. Skiena, S., M. Revilla, *Programming Challenges*, Springer Verlag, March 2003.
11. Lee, P.A., T. Anderson, *Fault Tolerance; Principles and Practice*, Dependable Computing and Fault-Tolerant Systems, Vol. 3, Second, Revised Edition, 1981.
12. Chen, L., A. Avizienis, *N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation*, Digest of 8th Annual International Symposium on Fault Tolerant Computing, Toulouse, France, pp. 3-9, June 1978.
13. IEC, IEC61508, *Functional Safety of E/E/PE safety-related systems*, Geneva, 2001-2.

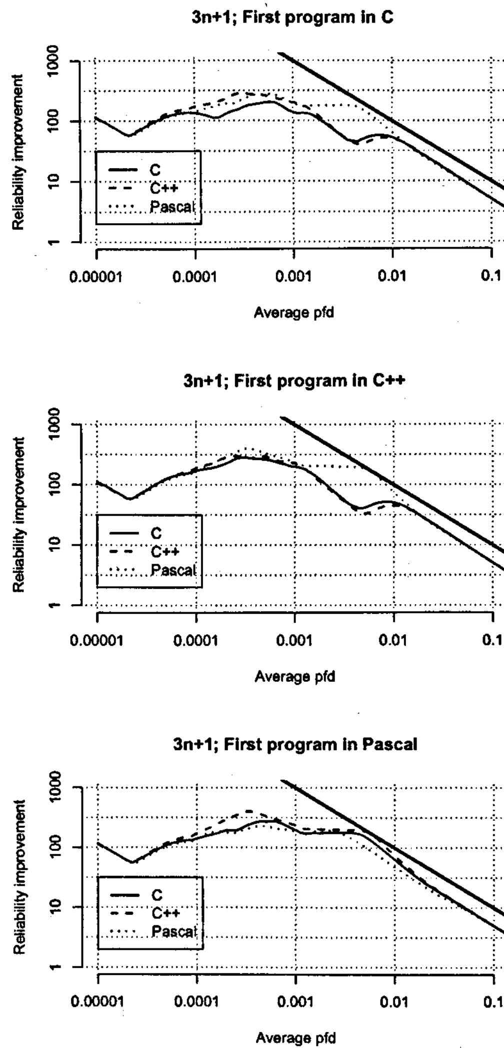


Fig. 1. These graphs show the reliability improvement of a pair of programs over a single version for the "3n+1"-problem. The horizontal axis gives the average pfd of the pools of programs involved in the calculation. In every graph, the programming language of the first program is given. The curves show the reliability improvement for the different possible choices of the programming language for the second program as function of the average pfd of the pools. The diagonal in the graphs shows the reliability achievement if the programs' behaviours were independent.

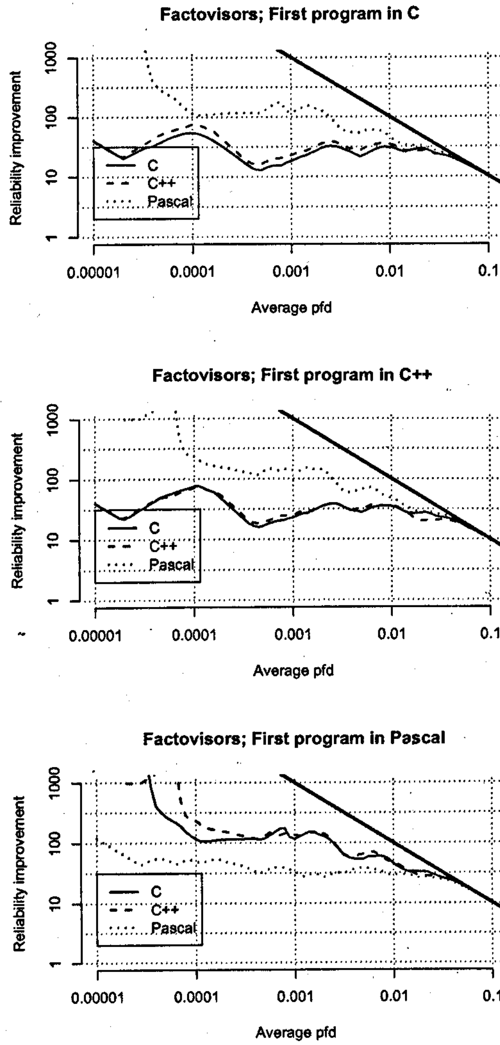


Fig. 2. The same graphs, for the "Factovisors"-problem.

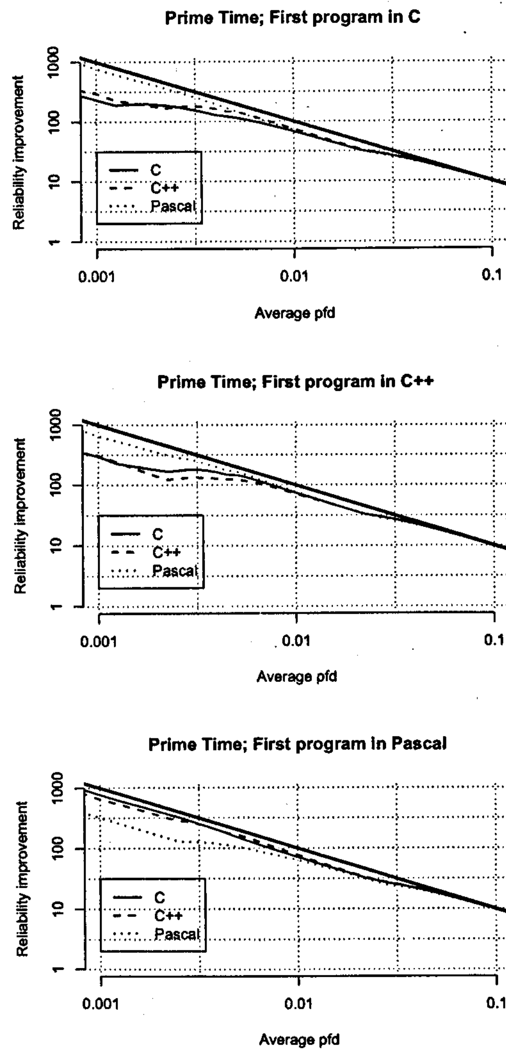


Fig. 3. The same graphs, for the "Prime Time"-problem. These graphs do not show the results for pfds below $10^{-2.5} = 3.2 \times 10^{-3}$, because the pool of Pascal programs is almost empty.

C.6 Safecomp 2005

M.J.P. van der Meulen, L. Strigini, and M.A. Revilla. On the effectiveness of run-time checks. In B.A. Gran and R. Winter, editors, *Proceedings of the 23rd international conference on Computer Safety, Reliability and Security, Safecomp 2005*, volume 2688 of *Lecture Notes in Computer Science*, pages 151–64, Fredrikstad, Norway, September 2005, ©Springer. With kind permission of Springer Science and Business Media.

On the Effectiveness of Run-Time Checks

Meine J.P. van der Meulen, Lorenzo Strigini[1] and Miguel A. Revilla[2]

[1] City University, Centre for Software Reliability, London, UK
WWW home page: http://www.csr.city.ac.uk
[2] University of Valladolid, Valladolid, Spain
WWW home page: http://www.mac.cie.uva.es/~revilla

Abstract. Run-time checks are often assumed to be a cost-effective way of improving the dependability of software components, by checking required properties of their outputs and flagging an output as incorrect if it fails the check. However, evaluating how effective they are going to be in a future application is difficult, since the effectiveness of a check depends on the unknown faults of the program to which it is applied. A programming contest, providing thousands of programs written to the same specifications, gives us the opportunity to systematically test run-time checks to observe statistics of their effects on actual programs. In these examples, run-time checks turn out to be most effective for unreliable programs. For more reliable programs, the benefit is relatively low as compared to the gain that can be achieved by other (more expensive) measures, most notably multiple-version diversity.

1 Introduction

Run-time checks are often proposed as a means to improve the dependability of software components. They are seen as cheap compared to other means of increasing reliability by run-time redundancy, e.g. N-version programming.

Run-time checks (also called *executable assertions* and other names) can be based on various principles (see e.g. Lee and Anderson [3] for a summary), and have wide application. For instance, the concept of design by contract [5] enables a check on properties of program behaviour.

Some run-time checks can detect all failures, for example checks that perform an inverse operation on the result of a software component [1,2]. If the program computes $y = f(x)$, an error is detected if $x \neq f^{-1}(y)$. This is especially attractive when computing $f(x)$ is complex, and the computation of the inverse f^{-1} relatively simple. The argument is then that because computing f^{-1} is simple, the likelihood of failure of this run-time check is low. Also, it seems unlikely that both the primary computation and the run-time check would fail on the same invocation and in a consistent fashion. Together, these factors lead to a high degree of confidence that program outputs that pass the check will be correct. However—as these authors readily admit—such theoretically perfect checks do not exist in many cases, maybe even not in the majority of cases. Run-time checks can then still be applied, but they will in general not be capable of finding all failures. Examples of these partial run-time checks are given by e.g. [12].

Previous empirical evaluation of run-time checks have generally used small samples of programs, or single programs [4,7,11]. Importantly, we run these measures on a large *population* of programs. Indeed, if we wish to learn something general about a run-time check, we need this statistical approach. Measuring the effectiveness of a run-time check on a single program could, given a certain demand profile and enough testing, determine the fraction of failures that the check is able to detect (coverage) for that program, given that demand profile. But in practice, this kind of precise knowledge would be of little value: if one could afford the required amount of testing, at the end one would also know which bugs the program has, and thus could correct them instead of using the run-time check. However, a software designer wants to know whether a certain run-time check is worth the expense of writing and running it, without the benefit of such complete knowledge. The run-time check can detect certain failures caused by certain bugs: the coverage of the check depends on which faults the program contains; and the designer does not usually know this. What matters are the statistics of the check's coverage, given the statistics of the bugs that *may* be present in the program. If a perfect check cannot be had, a check that detects most of the failures caused by those bugs that are likely to be in a program has great value. A check that detects many failures that are possible but are not usually produced, because programmers do not make the mistakes that would cause them, is much less useful. In conclusion, the coverage of a check depends on the distribution of possible programs in which it is to be used.

Here, we choose three program specifications for which we have large numbers of programs, and for each of the three we choose a few run-time checks, then study their coverage. We thus intend to provide some example "data points" of how the coverage can vary between populations of programs. In addition to such anecdotal evidence—evidence that certain values or patterns of values *may* occur—such experiments may contribute to software engineering knowledge if they reveal either some behaviour that runs contrary to the common-sense expectations held about run-time checks, and/or some apparent common trend among these few cases, allowing us to conjecture general laws, to be tested by further research.

For lack of space, we only discuss coverage, or equivalently the probability of undetected failure. We will also not discuss other dependability issues like availability (possibly reduced by false alarms from run-time checks), although these should be taken into account when selecting fault tolerance mechanisms.

2 The Experiment

2.1 The UVa Online Judge

The "UVa Online Judge"-Website [8] is an initiative of one of the authors (Revilla). It contains program specifications for which anyone may submit programs in C, C++, Java or Pascal intended to implement them. The correctness of a program is automatically judged by the "Online Judge". Most authors submit

Table 1. Some statistics on the three problems.

	3n+1			Factovisors			Prime Time		
	C	C++	Pascal	C	C++	Pascal	C	C++	Pascal
Number of authors	5,897	6,097	1,581	212	582	71	467	884	183
First submission correct	2,479	2,434	593	112	294	41	345	636	125

programs repeatedly until one is judged correct. Many thousands of authors contribute and together they have produced more than 3,000,000 programs for the approximately 1,500 specifications on the website.

We study the C, C++ and Pascal programs written to three different specifications (see Table 1 for some statistics, and `http://acm.uva.es/problemset/` for more details on the specifications). We submit every program to a test set, and compare the effectiveness of run-time checks in detecting their failures.

There are some obvious drawbacks from using these data as a source for scientific analysis. First, these are not "real" programs: they solve small, mostly mathematical, problems. Second, these programs are not written by professional programmers, but typically by students, which may affect the amount and kind of programming errors. We have to be careful not to overinterpret the results.

All three specifications specify programs that are memory-less (i.e. earlier demands should not influence program behaviour on later ones), and for which a demand consists of only two integer input values. Both restrictions are useful to keep these initial experiments simple and the computing time within reasonable bounds. The necessary preparatory calculations for the analysis of these programs took between a day and two weeks, depending on the specification.

2.2 Running the Programs

For a given specification, all programs were run on the same set of demands. Every program is restarted for every demand, to ensure the experiment is not influenced by history, e.g. when a program crashes for certain demands or leaves its internal state corrupted after execution of a demand (we accept the drawback of not detecting bugs with history-dependent behaviour). We set a time limit on the execution of each demand, and thus terminate programs that are very slow, stall, or crash. We only use the first program submitted by each author and discard all subsequent submissions by the same author. These subsequent submissions have shown to have comparable fault behaviour and this dependence between submissions would complicate any statistical analysis.

For each demand, the outputs generated by all the programs are compared. Programs that produce exactly the same outputs on every demands form an "equivalence class". We evaluate the performance of each run-time check for each equivalence class.

For all three specifications, we chose the equivalence class with the highest frequency as the *oracle*, i.e. the version whose answers we consider correct. We challenged each oracle in various ways, but never found any of them to have

Table 2. Classification of execution results with plausibility checks.

Output of primary	Output valid	Plausibility check	Effect from system viewpoint
Correct	Yes	Accept	Success
Correct	Yes	Reject	False alarm
Incorrect	Yes	Accept	Undetected failure
Incorrect	Yes	Reject	Detected failure
Incorrect	No	-	Detected failure

failed. For each specification, the test data were chosen to exhaustively cover a region in the demand space. In other words, we assume (arbitrarily) a demand profile in which all demands that occur are equiprobable.

2.3 Outcomes of Run-Time Checks

Run-time checks test properties of the output of a software component (the primary), based on knowledge of its functionality. In the rest of this paper we distinguish two types of run-time checks: *plausibility checks* and *self-consistency checks* (SCCs). The latter, inspired by Blum's "complex checkers" [12], use additional calls to the primary to validate its results, by checking whether some known mathematical relationship that must link its outputs on two or more demands does hold.

Checks on the values output by the primary are only meaningful if the output satisfies some minimal set of syntactic properties, one of which is that an output exists. Other required properties will be described with each specification. We call an output that satisfies this minimal set of properties "valid" (in principle this validaty check also constitutes a run-time check). We separate the check for "validity" from the "real" run-time checks, because it otherwise remains implicit and a fair comparison of run-time checks is not possible.

Table 2 shows how we classify the effects of plausibility checks. There are two steps: first, a check on the validity of the output of the primary; second, if this output is valid, a plausibility check on the output. There is an undetected failure (of the primary) if both the primary computes an incorrect valid output and the checker fails to detect the failure. Our plausibility checks did not cause any false alarms. Also note that a correct output cannot be invalid.

With self-consistency checks, the classification is slightly more complex (Table 3): we have to consider that one way the self-consistency check may fail is because its additional calls to the primary do not elicit valid outputs (e.g., they cause the primary to crash). We then assume that the self-consistency check will fail to reject the primary's output, i.e., that an undetected failure ensues. We could have made the decision to reject the output of the primary if the self-consistency check fails in this way; this would lead to slightly different results. False alarms did occur, which we do not analyse here for lack of space.

Table 3. Classification of execution results with self-consistency checks.

Output of primary	Output valid	Output of second call to primary by self-consistency check	Effect from system viewpoint
Correct	Yes	Consistent	Success
Correct	Yes	Inconsistent	False alarm
Correct	Yes	Invalid output	Success
Incorrect	Yes	Consistent	Undetected failure
Incorrect	Yes	Inconsistent	Detected failure
Incorrect	Yes	Invalid output	Undetected failure
Incorrect	No	-	Detected failure

3 Results for the "3n+1" specification

Short specification. A number sequence is built as follows: start with a given number n; if it is odd, multiply by 3 and add 1; if it is even, divide by 2. The sequence length is the number of required steps to arrive at a result of 1. Determine the maximum sequence length (max) for all values of n between two given integers i, j, with $0 < i, j \leq 100,000$. The output of the program is the triple: i, j, max.

We tested "3n+1" with 2500 demands ($i, j \in 1..50$). The outputs of the programs were deemed correct if the first three numbers in the output exactly matched those of the oracle. We consider an output "valid" if it contains at least three numbers. In the experiment we discard non-numeric characters and the fourth and following numbers in the output. The programs submitted to "3n+1" have been analysed in detail in [9]; this paper provides a description of the faults present in the equivalence classes.

3.1 Plausibility Checks

We use the following plausibility checks for the "3n+1"-problem:

1. The maximum sequence length should be larger than 0.
2. The maximum possible sequence length (given the range of inputs) is 476.
3. The maximum sequence length should be larger than $log_2(max(i, j))$.
4. The first output should be equal to the first input.
5. The second output should be equal to the second input.

We measure the effectiveness of a run-time check as the improvement it produces on the average *probability of undetected failure on demand* (*pufd*). Without run-time checks, a program's probability of undetected failure equals its probability of failure per demand (*pfd*).

Figure 1 shows the improvement in average *pufd* given by these plausibility checks, depending on the average *pufd* of a pool of programs. We manipulate this average by removing, one by one, from the original pool of 13575 programs,

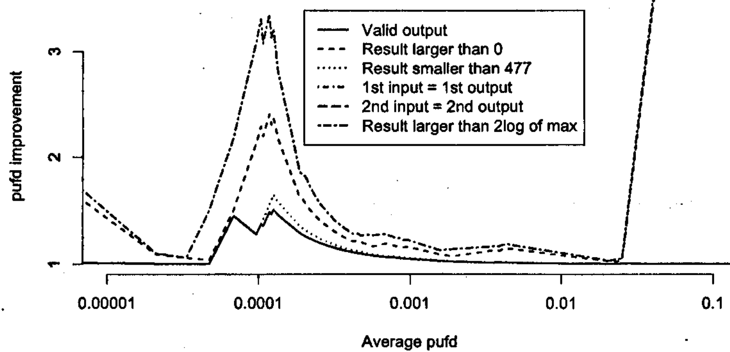


Fig. 1. The improvement of the *pufd* of the primary for the various plausibility checks for "3n+1". The curves for "1st input = 1st output" and "2nd input = 2nd output" are invisible because they coincide with the curve for "Valid output".

the programs with the highest *pufd*. The more programs have been removed, the lower the average *pufd* of the remaining pool.

The graph clearly shows that many of these run-time check are very effective for unreliable programs (the right-hand side of the graph). More surprising is that the impact is quite pronounced at a *pufd* of the pool around 10^{-4}, while it is much lower for the rest of the graph. Apparently, these checks are effective for some equivalence classes that are dominant in the pool for that particular *pufd* range. Upon inspection, it appears that these programs fail for $i = j$.

The gain in *pufd* is for most of the graph only about 20%, but the peak reaches a factor of 3.2 for the plausibility check "Result > $log_2(max(i, j))$", a significant improvement over a program without checks. The check "Result>0" is mainly effective for programs that initialise the outcome of the calculation of the maximum sequence length to 0 or -1, if they abort the calculation before setting the result to a new value. This appears to be caused by an incorrect "for"-loop which fails when $i > j$. The check "Result<477" is not very effective. The failures it detects have mostly to do with integer overflow and uninitialised variables.

The check "Result > $log_2(max(i, j))$" is the most effective of all. It catches a few more programming faults than "Result > 0", especially of those programs that do not cover the entire range between the two inputs i and j for the calculation of the maximum sequence length.

Figure 2(a) gives some more detail of the performance of this plausibility check. It shows the percentage of failures detected for each equivalence class. We can make various observations. First, for many equivalence classes there is no effect (many crosses with a coverage of 0%). Second, since there are more crosses

6

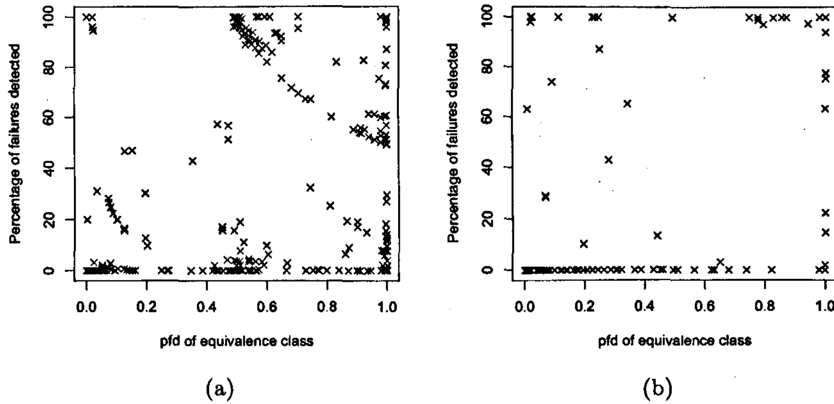


Fig. 2. Values of the error detection coverage of (a) the plausibility check "Result > $log_2(max(i,j))$" for the equivalence classes of "3n+1" programs, and (b) the plausibility check "$i \leq j$" for the equivalence classes of "Factovisors" programs. Each cross represents an equivalence class. The horizontal axis gives the average *pfd* of the equivalence class, the vertical axis the percentage of its incorrect outputs that the check detects.

in the right-hand side of the graph, this check seems to be more effective when the primary programs tend to be less reliable (i.e., for development processes that tend to deliver poor reliability). We must say "seem" here, because this graph lacks information about the frequencies of the various programs (sizes of the equivalence classes). Third, this plausibility check still detects faults in the left-hand side of the graph, i.e. for the more reliable programs.

The plausibility check "First output equals first input" mainly catches problems caused by incorrect reading of the specification: some programs do not return the inputs, or not always in the correct order. These faults lead to very unreliable programs, and the effects of this plausibility check are not visible in Figure 1 because they manifest themselves (i.e. differ from the curve for "Valid output") for average *pufd*s larger than 0.1.

The result of the plausibility check "Second output equals second input" is almost equal to the previous one. There are a few exceptions, for example when the program returns the first input twice.

3.2 Self-Consistency Checks

If we denote the calculation of the maximum sequence length as $f(i,j)$, then:

$$f(i,j) = f(j,i) \tag{1}$$

and:

$$f(i,j) = max(f(i,k), f(k,j)) \quad \text{for} \quad k \in i..j \tag{2}$$

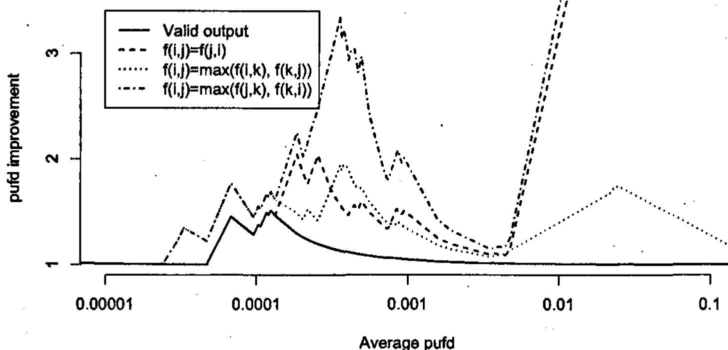


Fig. 3. Improvement in the average *pufd* of the primary for the various self-consistency checks for "3n+1".

and, if we combine these two properties:

$$f(i,j) = max(f(j,k), f(k,i)) \quad \text{for} \quad k \in i..j \tag{3}$$

Figure 3 presents the effectiveness of these self-consistency checks (for the experiment, we choose $k = \lfloor (i+j)/2 \rfloor$). Like our plausibility checks, these self-consistency checks appear to be very effective for unreliable programs.

The first self-consistency check mainly detects failures of programs in which the calculation of the maximum sequence length results in 0 or -1 for $i > j$. The second mainly finds failures caused by incorrect calculations of the maximum sequence length.

The third self-consistency check attains an improvement comparable to that of the plausibility check "Result > $log_2(max(i,j))$", but with a shifted peak. It appears that they catch different faults in the programs. As already stated, the peak of "Result > $log_2(max(i,j))$" is caused by programs failing for $i = j$ (which none of our self-consistency checks can detect) while this self-consistency check detects failures caused by faults in the calculation of the maximum sequence length as well as programs that systematically fail for $i > j$.

The fact that the plausibility checks and the self-consistency checks tend to detect different faults is highlighted by Figure 4, which shows the performances of the combined plausibility checks, the combined self-consistency checks and the combination of all run-time checks.

4 Results for the "Factovisors" specification

Short specification. For two given integers $0 \leq i, j \leq 2^{31}$, determine whether j divides $i!$ (factorial i) and output "j divides i!" or "j does not divide i!".

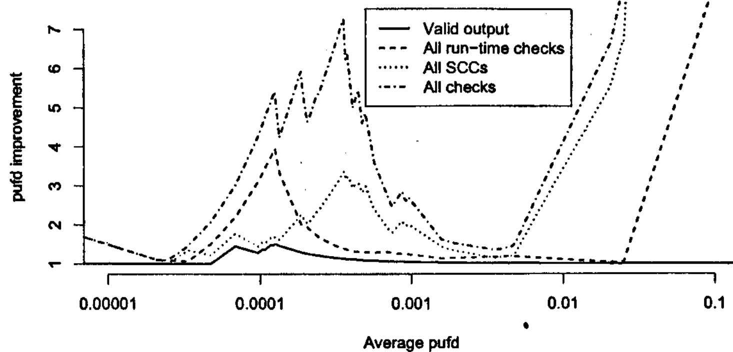


Fig. 4. Improvement in the average *pufd* of the primary for combinations of run-time checks for "3n+1".

We tested "Factovisors" with the 2500 demands ($i, j \in 1..50$). We consider an output "valid" if it contains at least two strings and the second is "does" or "divides". The main reason for invalid outputs appears to be absence of outputs.

4.1 Plausibility Checks

We use the following plausibility check for "Factovisors":

1. If $i \geq j$, the result should be "j divides i!".

Figure 2(b) shows the coverage of the run-time check "$i \geq j$" for each equivalence class. It is remarkable, again, that the crosses are spread over the entire plane: this check has some effect for equivalence classes with a large range of reliabilities. We also again observe the large number of crosses for a coverage of 0%, showing the check to detect no failure at all for that class of programs.

Figure 5 shows the *pufd* improvement caused by the plausibility check. As for "3n+1", we observe that the run-time check is very effective for unreliable programs. For pools of programs with average *pufd* between 10^{-4} and 10^{-2} the reliability improvement varies between 1 and 1.6.

The graph shows a peculiarity for *pufd*s smaller than 10^{-4}: the improvement approaches infinity. This is because as we remove programs from the pool, the faulty programs in the pool eventually become a "monoculture", a single equivalence class, and the check happens to detect all the failures of this class of incorrect programs. Here, the pool with the lowest non-zero average *pufd* contains 447 correct programs and 21 incorrect ones in the same equivalence class; the plausibility check detects the failures of these 21 incorrect programs..

9

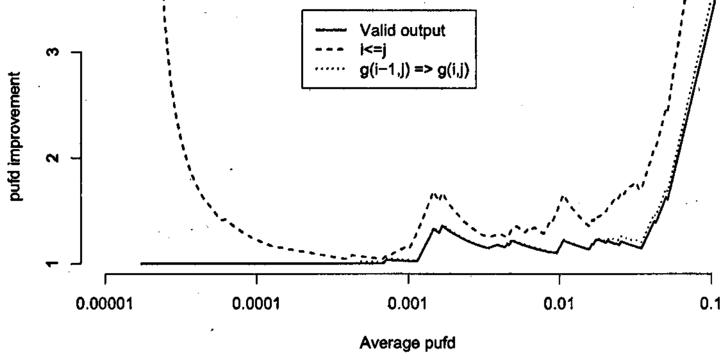


Fig. 5. The effectiveness of the run-time checks for "Factovisors".

4.2 Self-Consistency Checks

If we call $g(i,j)$ the Boolean representation of the output of the program, with $g(i,j) = true \equiv$ "j divides i!", $g(i,j) = false \equiv$ "j does not divide i!", then:

$$g(i-1,j) \implies g(i,j) \quad \text{with} \quad i \neq 1 \tag{4}$$

As can be seen in Figure 5, the effect of this self-consistency check is minimal: the reliability improvement is never substantially greater than that given by the validity check.

5 Results for the "Prime Time " specification

Short specification. Euler discovered that the formula $n^2 + n + 41$ produces a prime for $0 \leq n \leq 40$; it does however not always produce a prime. Calculate the percentage of primes the formula generates for n between two integers i and j with $0 \leq i \leq j \leq 10,000$.

We tested "Prime Time" on 3240 demands ($i \in 0..79$, $j \in i..79$). The outputs were deemed correct if they differed by most 0.01 from the output of the oracle, allowing for round-off errors (the answer is to be given with two decimal digits).

The output is considered "valid" when it contains at least one number. We discard all non-numeric characters and subsequent digits from the output.

5.1 Plausibility Checks

The programs for "Prime Time" calculate a percentage, therefore:

1. The result should be larger than or equal to zero.

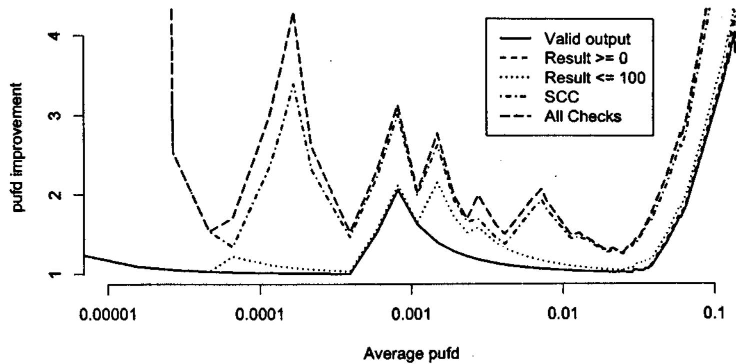


Fig. 6. The effectiveness of the run-time checks for "Prime Time". The curve for the plausibility check "Result $\geq$ 0" is not visible, because it coincides with the one for "Valid output".

2. The result should be smaller than or equal to a hundred.

Figure 6 presents the effectiveness of the plausibility checks for "Prime Time". The plausibility check "Result $\geq$ 0" appears to have virtually no effect. The plausibility check "Result $\leq$ 100" has some effect, but not very large.

5.2 Self-Consistency Checks

If we denote the result of the calculation of the percentage with $h(i,j)$, then:

$$h(i,j) = \frac{h(i,k) \times (k-i+1) + h(k+1,j) \times (j-k)}{j-i+1} \quad \text{for} \quad i \leq k < j \tag{5}$$

Obviously, this check is not available when $i = j$. It is quite elegant: the computing time will not be excessively more than computing $h(i,j)$. For the experiment, we choose $k = \lfloor (i+j)/2 \rfloor$.

The effectiveness of the self-consistency check is shown in Figure 6. It is much more effective than the plausibility check "Result $\leq$ 100". We observe the same phenomenon for low *pufd*s as for "Factovisors": the effectiveness of the self-consistency check approaches infinity. When we combine the plausibility checks and the self-consistency check, we observe that the two complement each other: the combination is (slightly) more effective than the self-consistency check alone.

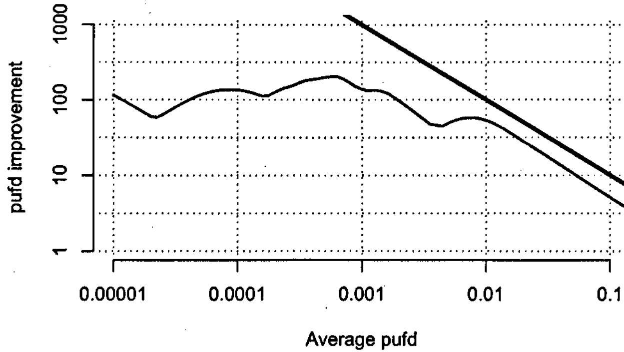


Fig. 7. Improvement of the *pufd* of a pair of randomly chosen C programs for "3n+1", relative to a single version. The horizontal axis shows the average *pufd* of the pool from which both C programs are selected. The vertical axis shows the *pufd* improvement ($pufd_A/pufd_{AB}$). The diagonal represents the theoretical reliability improvement if the programs fail independently, i.e. $pufd_{AB} = pufd_A . pufd_B$. (This figure is based on [10].)

6 Run-Time Checks vs. Multiple-Version Diversity

A question that begs answering is: how do run-time checks compare to other forms of run-time fault tolerance? Using results we reported previously [10], we can compare our run-time checks against multiple-version diversity for "3n+1".

We observed (see Figure 7) that two-version diversity would become more effective with decreasing mean probability of failure on demand of the pool of programs from which the pair is selected, until a "plateau" is reached (between a *pufd* of 10^{-5} and 10^{-3}) with an improvement factor of about a hundred (note that the opposite trend—effectiveness decreasing with decreasing mean *pfd*—is also possible, as proved by models and empirical results [6]). For run-time checks the opposite occurs: their effectiveness decreases with decreasing average *pufd* of the primary reaching a fairly low improvement factor. The improvement factor of using diversity is significantly higher than that of applying run-time checks.

For these programs, it seems that these run-time checks could be the better choice for testing in the early phases of development, when the *pufd* of programs is still high, and multiple-version diversity when *pufd*s of programs become low.

7 Conclusion

The results in this paper are of course specific to these three specifications, the programs submitted by these anonymous authors, the run-time checks we devised, and the demand profiles we used (uniform in a subset of the demand

space). There are however some commonalities among the three sets of results, and we will tentatively discuss these here, while keeping in mind the limitations of this research.

First, we observe that the majority of the run-time checks considered are very effective for unreliable programs or have no effect at all.

Then, if we only look at pools of primaries with average *pufd* between 10^{-4} and 10^{-2}, the *pufd* improvement factor of the primary-checker pair is comparable for all three specifications: in the range 1–4. Over this range, the average improvement is less than 2 for all run-time checks considered.

Some run-time checks provide almost no benefit. It would be of great importance to be able to predict which checks are effective and which are not, but for the time being this seems not to be possible.

These plausibility checks appear to detect a different set of failures than the self-consistency checks, so that combining them is more effective than applying either one alone. So, the apparent "diversity" between the two kinds of checks did bring the benefit of some complementarity.

For pools of primaries with an average *pufd* lower than 10^{-2}, the *pufd* improvement achieved by the run-time checks considered for "3n+1" is far less than would have been achieved by applying multiple-version redundancy. In these analyses, the *pufd* improvement realised by multiple-version redundancy is at least a factor of a hundred better.

A natural comment on this work could be that since we have implemented simple-minded checks, it is not surprising that they only catch the simple-minded programming errors that cause highly unreliable programs. But this is actually a *non sequitur*. It is true that we do not expect expert programmers to produce highly unreliable programs, but our checks are "simple-minded" only in being based on simple mathematical properties of these specifications. There is no a priori reason why they should only catch simple-minded implementation errors: implementation errors are often caused by misunderstanding details of the specification or of the program itself, not of some mathematical property of the specification that is of little interest to the programmers. Likewise, there is no a priori reason for naive errors normally to cause faults which cause very high failure rates.

A tempting conjecture generalising the results we observed is that for some reason simple run-time checks tend (in some types of programs?) only to detect the failures in very unreliable programs. This would be an attractive "natural law" to believe and would simplify many decisions on applying run-time checks, so that it may be worth exploring further, since without some solid, plausible explanation (e.g. based on the psychology of programmers) or overwhelming empirical evidence, it would appear wholly unjustified.

Our measure of effectiveness as average improvement in *pufd* may be questioned. It is such that even if a check C has 100% coverage for the failures produced by a set of dangerous possible bugs, B, it will still be assessed as having negligible effectiveness if the bugs in set B occur with negligible probability in actual software development. Some may object that if C is the only check

that can detect the effects of B-type bugs, and given the uncertainty on the probabilities of B these bugs being actually produced, a prudent designer will still use C. This objection is certainly right if C has negligible cost (implementation cost, cost in run-time resources, risk of bugs in C causing false alarms, etc). But whenever these costs are non-negligible, they must be weighted against C's potential benefits, as we do.

Acknowledgement

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council via the Interdisciplinary Research Collaboration on the Dependability of Computer Based Systems (DIRC), and via the Diversity with Off-The-Shelf Components (DOTS) project, GR/N23912.

References

1. M. Blum and H. Wasserman. Software reliability via run-time result-checking. Technical Report TR-94-053, International Computer Science Institute, October 1994.
2. A. Jhumka, F.C. Gärtner, C. Fetzer, and N. Suri. On systematic design of fast and perfect detectors. Technical Report 200263, École Polytechnique Fédérale de Lausanne (EPFDL), School of Computer and Communication Sciences, September 2002.
3. P.A. Lee and T. Anderson. *Fault Tolerance; Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer, 2nd edition, 1981.
4. N.G. Leveson, S.S. Cha, J.C. Knight, and T.J. Shimeall. The use of self checks and voting in software error detection: An empirical study. In *IEEE Transactions on Software Engineering*, volume 16(4), pages 432–443, 1990.
5. B. Meyer. Design by contract. *Computer (IEEE)*, 25(10):40–51, October 1992.
6. P. Popov and L. Strigini. The reliability of diverse systems: A contribution using modelling of the fault creation process. *DSN 2001, International Conference on Dependable Systems and Networks, Göteborg, Sweden*, July 2001.
7. M. Rela, H. Madeira, and J.G. Silva. Experimental evaluation of the fail-silent behavior of programs with consistency checks. In *FTCS-26, Sendai, Japan*, pages 394–403, 1996.
8. S. Skiena and M. Revilla. *Programming Challenges*. Springer Verlag, March 2003.
9. M.J.P. van der Meulen, P.G. Bishop, and M. Revilla. An exploration of software faults and failure behaviour in a large population of programs. In *The 15th IEEE International Symposium of Software Reliability Engineering, 2–5 November 2004, St. Malo, France*, pages 101–12, 2004.
10. M.J.P. van der Meulen and M. Revilla. The effectiveness of choice of programming language as a diversity seeking decision. In *EDCC-5, Fifth European Dependable Computing Conference, Budapest, Hungary, 20–22 April, 2005*, April 2005.
11. J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson. Reducing critical failures for control algorithms using executable assertions and best effort recovery. In *DSN 2001, International Conference on Dependable Systems and Networks, Goteborg, Sweden*, 2001.
12. H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

C.7 Safecomp 2006

M.J.P. van der Meulen and M.A. Revilla, Experiences with the design of a run-time check. In J. Gorski, editor, *Proceedings of the 24th international conference on Computer Safety, Reliability and Security, Safecomp 2006*, volume 4166 of *Lecture Notes in Computer Science*, pages 302-15, Gdansk, Poland, 2006. ©Springer. With kind permission of Springer Science and Business Media.

Experiences with the Design of a Run-Time Check

Meine J.P. van der Meulen[1] and Miguel A. Revilla[2]

[1] City University, Centre for Software Reliability, London, UK
WWW home page: http://www.csr.city.ac.uk
[2] University of Valladolid, Valladolid, Spain
WWW home page: http://www.mac.cie.uva.es/~revilla

Abstract. Run-time checks are often assumed to be a cost-effective way of improving the dependability of software components, by checking required properties of their outputs and flagging an output as incorrect if it fails the check. Run-time checks' main point of attractiveness is that they are supposed to be easy to implement. Also, they are implicitly assumed to be effective in detecting incorrect outputs. This paper reports the results of an experiment designed to challenge these assumptions about run-time checks.
The experiment uses a subset of 196 of 867 programs (primaries) solving a problem called "Make Palindrome". This is an existing problem on the "On-Line Judge" website of the university of Valladolid. We formulated eight run-time checks, and posted this problem on the same website. This resulted in 335 programs (checkers) implementing the run-time checks, 115 of which are used for the experiment.
In this experiment: (1) the effectiveness of the population of possibly faulty checkers is very close to the effectiveness of a correct checker; (2) the reliability improvement provided by the run-time checks is relatively small, between a factor of one and three; (3) The reliability improvement gained by using multiple-version redundancy is much higher. Given the fact that this experiment only considers one primary/Run-Time Check combination, it is not yet possible to generalise the results.

1 Introduction

Redundancy is a means to improve the reliability of software components. Much research has been invested in multiple-version redundancy, e.g. 1-out-of-2 systems. Much less research has been invested in asymmetrical redundancy, e.g. the use of run-time checks, RTCs, see Figure 1. In these cases a primary program is checked by an, ideally relatively simple, RTC.

RTCs are often proposed as a means to improve the dependability of software components. They are seen as cheap compared to other means of increasing reliability by run-time redundancy, e.g. N-version programming. We are interested in answering questions like whether RTCs are effective and how their performance compares to that of symmetrical redundancy. We also want to confirm or reject

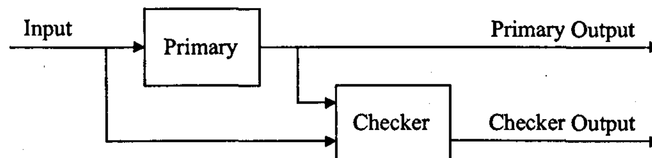


Fig. 1. Primary/Checker model.

common conjectures, such as that RTCs are so simple that we may assume that they are correctly programmed.

RTCs (also called *executable assertions* and other names) can be based on various principles (see e.g. Lee and Anderson [3] for a summary), and have wide application. For instance, the concept of design by contract [5] enables a check on properties of program behaviour.

Some RTCs can detect all failures, for example checks that perform an inverse operation on the result of a software component [1,2]. If the program computes $y = f(x)$, an error is detected if $x \neq f^{-1}(y)$. This is especially attractive when computing $f(x)$ is complex, and the computation of the inverse f^{-1} relatively simple. The argument is then that because computing f^{-1} is simple, the likelihood of failure of this RTC is low. Also, it seems unlikely that both the primary computation and the RTC would fail on the same invocation and in a consistent fashion. Together, these factors lead to a high degree of confidence that program outputs that pass the check will be correct. However—as these authors readily admit—such theoretically perfect checks do not exist in many cases, maybe even not in the majority of cases. RTCs can then still be applied, but they will in general not be capable of finding all failures. Examples of these partial RTCs are given by e.g. [11].

Previous empirical evaluation of RTCs have generally used small samples of programs, or single programs [4,7,10]. Importantly, our experiment involves a *population* of programs, both of the primary and of the checker, because we think that in order to learn something general about RTCs, we need a statistical approach. We can now study the complex interplay between (possible faulty) primaries combined with (partial, possibly faulty) checkers.

The interaction between the primary and the checker is complex because the performance of the checker is dependent on that of the primary. An example of this interaction is that it may be that improving the primary may lead to a rise in the probability of undetected failure. Suppose that a primary (e.g., to compute $f(n) = (n+2)(n-2)$) is incorrect ($f(5) = 27$) and that the checker (e.g., $f(n) \leq n^2$) detects the incorrect outputs of the primary. Now, the programmer changes the primary ($f(5) = 23$); its output is now closer to the correct answer, but still incorrect. It may now be that the checker is unable to detect the incorrect outputs. As a result, the probability of undetected failure may have increased.

Table 1. Sample inputs and sample outputs for the primary.

Sample Input	Sample Output
abcd	3 abcdcba
aaaa	0 aaaa
abc	2 abcba
aab	1 baab
abababaabababa	0 abababaabababa
pqrsabcdpqrs	9 pqrsabcdpqrqpdcbasrqp

2 The Experiment

2.1 The UVa Online Judge

The "UVa Online Judge"-Website (`http://acm.uva.es`, [8]) is an initiative of one of the authors (Revilla). It contains program specifications for which anyone may submit programs in C, C++, Java or Pascal intended to implement them. The correctness of a program is automatically judged by the "Online Judge". Most authors submit programs repeatedly until one is judged correct. Many thousands of authors contribute and together they have produced more than 3,000,000 programs for the approximately 1,500 specifications on the website.

2.2 Specification of the primary

For the primary, we took a specification from the Online Judge formulated by Md. Kamruzzaman: a program to generate palindromes. It takes an input string of 1000 or less lower case characters and makes it into a palindrome by inserting lower case characters into the input string at any position. The number of characters inserted shall be as low as possible. The output is the number of characters inserted, followed by the resulting palindrome. See for a complete specification the Online Judge website, `http://acm.uva.es/p/v104/10453.html`, and Table 1 for some examples of correct input and output combinations.

2.3 Specification of the checker

Based on the specification of the primary, we formulated a specification for its checker. The checker (`http://acm.uva.es/p/v104/10848.html`) takes as its input a string of 5000 or less ASCII characters (5000 to allow for faults in the primary, leading to the output of many characters). It tests this string for the following properties:

P1. It consists of a first string of lower case characters (length ≤ 1000), a single space, an integer (≥ 0, ≤ 1000), a single space, and a second string of lower case letters (length ≤ 2000).

3

Table 2. Sample inputs and sample outputs for the checker.

Sample Input	Sample Output
abcd 3 abcdcba	TTTTTTT The solution is accepted
aaaa 3 abcdcba	TTTFFTT The solution is not accepted
abc 2 abdcba	TFTTTFT The solution is not accepted
aab b baab	FFFFFFF The solution is not accepted
abababaababababa 0 abababaabababa	TTTTTTT The solution is accepted
pqrsabcdpqrs 9 pqrsabcdpqrqpdcbasrqp	TTTTTTT The solution is accepted

P2. P1 & the second string is a palindrome.
P3. P1 & all letters of the first string appear in the second string.
P4. P1 & the frequency of every letter in the second string is at least the frequency of this letter in the first string.
P5. P1 & the first string can be made out of the second string by removing 0 or more letters (and leaving the order of the letters intact).
P6. P1 & the length of the second string is equal to the length of the first string plus the value of the integer.
P7. P1 & the value of the integer is smaller than the length of the first string.

Obviously, when all properties are true, the output of the primary may still be faulty.

The output consists of the value "T" or "F" (for True and False) for every property in the list above, and a statement "The solution is accepted" if all properties are true, and "The solution is not accepted" otherwise. We will call this property P8. See Table 2 for some examples of correct input and output combinations.

Although the checkers implement all eight different properties, we will analyse these separately, as if the checker programs only implement one of these. When we address any of the run-time checks, we will use the abbreviation RTC. When we address the implementations of one of the properties P1-8, we will use the abbreviations RTC1-8.

2.4 System Behaviour

Table 3 shows how we classify the effects on the system based on the outputs of the primary and the run-time checks. The effects from the system viewpoint are rather obvious, except for the consequences of "No output" from the RTC (this includes invalid output). We have chosen to accept the output of the primary in these cases; the other option would also have been possible, it would have increased the number of false alarms. Our choice is based on the assumption that false alarms of RTCs are in general very undesirable.

RTC1 differs from RTC2-8, because it is purely a syntactical check of the input. It is a necessary precondition to be able to do any of the other checks. It

Table 3. Classification of execution results with RTCs.

Output of primary	run-time check	Effect from system viewpoint
Correct	Accept	Correct
Correct	Reject	False alarm
Correct	No output	Correct
Incorrect	Accept	Undetected failure
Incorrect	Reject	Detected failure
Incorrect	No output	Undetected failure

is interesting to separate this check from RTC2-8, because it gives us an idea how much the more application specific RTC2-8 add to this basic syntactic check.

2.5 Equivalence Classes

We subjected the primary to 10,000 demands: strings of lower case characters. Each string has a random length between 1 and 30 characters with random characters from the set "a".."e". The reason for the limited character set is that cases with character repetition will more frequently occur. The reason for the maximum length of the string is to limit the execution time of the primary.

We sorted the primaries in "equivalence classes", i.e. sets of programs producing exactly the same output. There is more than 1 correct equivalence class, because for almost all inputs there is more than one correct solution, e.g. the correct output to the input "ab" may be "1 aba" or "1 bab".

The primaries gave mostly equal, but also many different outputs to the 10,000 demands; in total there were 529,433 different outputs to the 10,000 inputs. to reduce computing time, we randomly selected 17,241 of these (approximately 1/30). We generated an input file to the checkers by combining each output with the corresponding input. This input file is used to determine the equivalence classes of the checkers and was only used for this purpose; for the rest of the experiment we used the 10,000 demands as used for determining the primary equivalence classes. We assumed that these 17,241 demands are sufficient to discern the different checker equivalent classes.

For every primary equivalence class we made an input file for the checkers by combining the 10,000 demands (the same for every primary) and their outputs. We executed every combination of primary and checker equivalence classes with the appropriate input files. This was computationally quite intensive; the computation took approximately three days.

2.6 Score Functions

Assume a specification for the primary, S_π:

$$S_\pi(x, y) \equiv \text{“}y \text{ is valid primary output for input } x\text{”} \qquad (1)$$

Then, we define the score function ω_π for a random primary π as:

$$\omega_\pi(\pi, x) \equiv \neg S(x, \pi(x)) \qquad (2)$$

I.e., the score function is true when the primary π fails to compute a valid output y for a given input x.

The behaviour of an RTC σ can be described as:

$$\sigma(x, y) \equiv \text{“}y \text{ is accepted as valid primary output for input } x\text{”} \qquad (3)$$

Note the similarity to the specification of the primary, S_π. Whereas the specification is supposed to be correct, we assume that the checker may be faulty: it may erroneously accept an incorrect pair (x, y). The checker fails if there is a discrepancy with the specification. The score function ω_σ for an RTC is:

$$\omega_\sigma(\sigma, x, y) \equiv S_\pi(x, y) \oplus \sigma(x, y) \qquad (4)$$

I.e., the score function is true when the checker fails to recognize whether y is valid primary output for input x or not.

For our system as depicted in Figure 1 and the variables (x, π, σ) for the input, the primary and the checker, there are four possibilities:

1. $\neg\omega_\pi(\pi, x) \wedge \neg\omega_\sigma(\sigma, x, \pi(x))$: Correct operation.
2. $\neg\omega_\pi(\pi, x) \wedge \omega_\sigma(\sigma, x, \pi(x))$: False alarm.
3. $\omega_\pi(\pi, x) \wedge \neg\omega_\sigma(\sigma, x, \pi(x))$: Detected failure.
4. $\omega_\pi(\pi, x) \wedge \omega_\sigma(\sigma, x, \pi(x))$: Undetected failure.

We have calculated the score functions $\omega_\pi(\pi, x)$ and $\omega_\sigma(\sigma, x, y)$ for the primary and the checker equivalence classes.

2.7 Subsets for the Experiment

There are 867 submissions for the primary specification. We included the primaries that are written in C, C++ or Pascal, compile and provide output within one second. This left 566 primaries. Then we excluded primaries that fail for all inputs, that left 484 primaries. From these we used the first submission of each author: 196 primaries. There are various reasons for selecting the first submissions:

1. We do not want to include more than one submission of a single author, because subsequent submissions are shown to be highly similar, and that would corrupt our statistical analyses.
2. The variability between first versions is higher, e.g. because later submissions are more likely to be correct. This gives more room for statistical analyses.

3. A first submission is most comparable to a first submission in a "normal" development process, because feedback to the authors differs from normal feedback in various ways, e.g. the Online Judge will not communicate for which input the program failed.

There are 395 submissions for the checker specification. We included the checkers that are written in C, C++ or Pascal, compile and provide output within one second. This left 335 checkers.

For these checkers we compute the average FAR (false alarm rate, the fraction of false alarms) for a specific primary π for RTC8 (for the calculations: $TRUE = 1, FALSE = 0$):

$$FAR(\pi) = \sum_{x \in D} \sum_{\sigma \in R_\sigma} (1 - \omega_\pi(\pi, x)) \cdot \omega_\sigma(\sigma, x, \sigma(x)) \cdot Q(x) \qquad (5)$$

R_σ is the set of checkers, $Q(X)$ is the demand profile over the demand space D. D is the test set of 10,000 demands; we assume that each of the 10,000 demands is equiprobable, and therefore: $Q(x) = 1/10,000$.

We excluded checkers that have a FAR of more than 0.1 for any primary or do not provide sensible output at all (manual check), that left 306 checkers. From these we used the first submission of each author: 118 checkers.

The rationale for excluding checkers with a high FAR is that these will normally be quickly detected during development. This is our modelling of the debugging process of the checkers. There is only one checker left with a FAR larger than zero. This checker fails for RTC1 (and subsequently often for RTC2-8) in a rather erratic way.

3 Observations

3.1 General

The first observation was already done during selection of a suitable primary for this research. It appeared that it is only possible for a small subset of the problems of the Online Judge to formulate meaningful RTCs. In many cases, the output of a program is a (set of) number(s) for which it is not possible to formulate an inverse function to the input, or even an interesting weaker relationship.

We chose this primary because it is possible to define RTCs, and a sufficient number of submissions for the primary is available.

3.2 The Specification of the Checker

The checker specification appeared to be incomplete: we forgot to specify a lower bound on the length of the strings. This leaves it to the programmers to decide whether an empty string is correct input or not. As it appears, some of the authors allow empty strings. This ambiguity has consequences for property 2: is

Table 4. Most frequent equivalence classes in the first submissions for the primary.

Fault	Freq.	PFD	Detection
Correct	129	0.00	No detection.
Adds too many characters to input string.	5	0.54	No detection.
Forgets last character in input string.	2	0.69	P3 (28%), P4 (55%), P5 (100%), P8 (100%)
Adds too many characters to input string.	2	0.55	No detection.
Output string is not always a palindrome.	2	0.12	P2 (100%), P8 (100%)
...			
Often fails to output second string.	1	0.94	P1-8 (100%)
Often outputs very large integers.	1	0.99	P126 (93%), P345 (96%), P8 (100%)
...			
No integer in output.	0	1.00	P1-8 (100%)
Often outputs control character at end of second string.	0	0.94	P1-8 (100%)

an empty string a palindrome or not? As it happens some authors who accept empty strings consider an empty string to be a palindrome, others don't.

A peculiar problem occurs for property 4: P1 & the frequency of every letter in the second string is at least the frequency of this letter in the first string. We intended to write: P1 & the frequency of every letter in the first string is smaller than or equal to the frequency of this letter in the second string. Peculiarly, most authors interpreted it this way. They thus wrote a stronger test than was specified.

We argue that problems with specifications are common, and that this observation does therefore not invalidate the conclusions of the paper. Maybe even to the contrary: they might even be supporting it, since specifying is part of the development process, and a possible source of errors.

There is one equivalence class containing a correct checker, i.e. a program that does not accept empty strings and interprets P4 as written in the specification. Nobody submitted a correct program as their first submission. This implies there is no correct submission in the set of programs we do our analyses on in this paper, c.f. 2.7.

3.3 Faults in the primary

To give an idea of the kinds of faults made, Table 4 presents some of the equivalence classes of the primaries. There are 17 correct equivalence classes, with in total 129 submissions. There are 67 incorrect submissions in 59 equivalence classes. Only five equivalence classes contain more than one submission, these are listed in the table. The fact that equivalence classes tend to only contain one program indicates that authors tend to choose different approaches and tend to make different mistakes. Furthermore, the presence of many different equivalence

Table 5. Most frequent equivalence classes in the first submissions for the checker.

Fault	Freq.
Correct, except that P4 is interpreted more extensively.	54
Fails when input contains control characters.	5
Fails when integer in input string is very large.	4
Always outputs "TTTTTT The solutions is accepted".	3
Fails when second string is absent.	2
Fails when there is a control character in the second string.	2
Fails when integer in input string is very large.	2
Fails when there is a control character in the integer or in the second string.	2
...	
Fails for P4, behaves partly as specified.	1
Fails for P4, behaves as specified, but strange other fault.	1
Accepts empty second string, assumes empty string is palindrome.	1
...	
Correct.	0
...	

classes for correct solutions indicates that authors do not tend to copy solutions from each other, one of the worries for the usefulness of the data.

The table also shows whether the faults are detected by a correct checker, and how effective these checks are.

3.4 Faults in the checker

Table 5 presents some of the equivalence classes of the checkers. There are 54 correct submissions in one equivalence class (there is only one way to solve this problem). There are 64 incorrect submissions in 51 equivalence classes, only seven of these contain more than one submission. This again indicates that the authors do not tend to make exactly the same mistake.

Seven submissions give no output when the second string is empty.

The most frequent mistake is that a checker fails when there are special ASCII characters in the input string, e.g. the NULL character. This is problematic, because some primaries fail in a way that produces exactly these characters. This is caused by the fact that the solution to the "Make Palindrome"-problem typically includes array manipulation. This, combined with a bug leading to a pointer being out of array bounds, leads to possibly outputting these characters. Important is here that this observation may undermine the conjecture of independence between primaries and checkers.

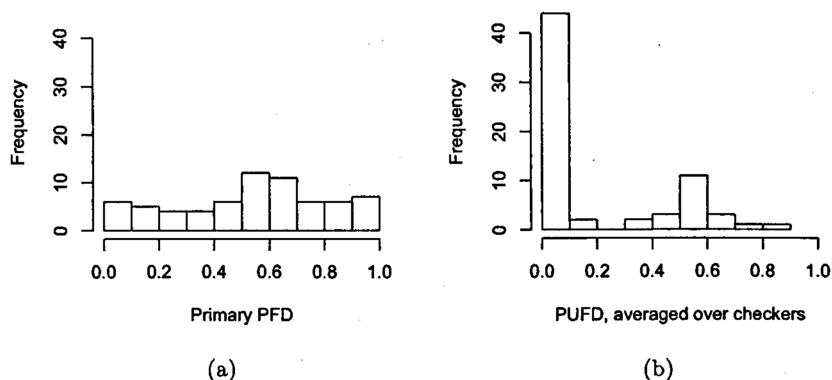


Fig. 2. (a) Histogram of the PFDs of the primaries; (b) Histogram of the average probability of undetected failure for the same primaries, with RTC8. Both graphs exclude the 129 correct primaries.

4 Statistical Calculations

4.1 Reliability Improvement

To know the reliability improvement the checkers give, we first have to compute the probability of failure on demand of the primaries:

$$PFD(\pi) = \sum_{x \in D} \omega_\pi(\pi, x) \cdot Q(x) \tag{6}$$

Figure 2(a) presents the distribution of the PFDs of the primaries in a histogram.

The probability of undetected failure of a primary, averaged over the checkers is:

$$PUFD(\pi) = \sum_{x \in D} \sum_{\sigma \in R_\sigma} \omega_\pi(\pi, x) \cdot \omega_\sigma(\sigma, x, \sigma(x)) \cdot Q(x) \tag{7}$$

Figure 2(b) shows the distribution of the probability of undetected failure after applying RTC8. As can be expected, there is a significant shift to the left.

We now calculate the improvement of the primary PFD for various checkers for subsets of the primary programs:

$$I(PFD_{min}, PFD_{max}) = \frac{\sum_{\pi \in \hat{R}_\pi} PFD(\pi)}{\sum_{\pi \in \hat{R}_\pi} PUFD(\pi)} \tag{8}$$

with $\hat{R}_\pi = \{\pi | PFD_{min} < PFD(\pi) \leq PFD_{max}\}$.

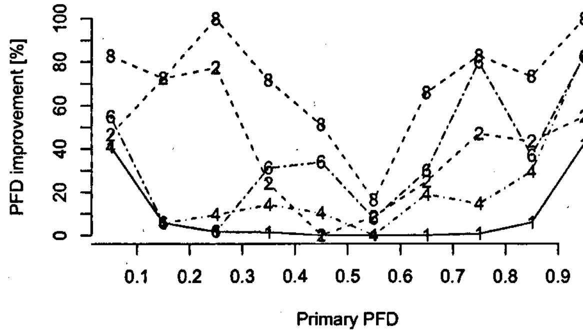


Fig. 3. The improvement of the probability of undetected failure in various PFD ranges for RTCs 1, 2, 4, 6 and 8 for a correct checker. The improvement in the range 0-0.1 excludes correct primaries.

We choose 10 subsets $\hat{R}_\pi$ of the primary programs such that: $0 < PFD(\pi) \leq 0.1$, $0.1 < PFD(\pi) \leq 0.2$, and so on. Figure 3 shows the improvement of the probability of undetected failure in the various PFD ranges for RTCs 1, 2, 4, 6 and 8 for a correct checker (we do not show all, because this makes the figure unreadable; the other RTCs show similar erratic behaviour). There appears to be no obvious relation between the PFD of the primaries and the effectiveness of RTCs. Who would have expected that RTC6 would be very effective for reliable primaries?

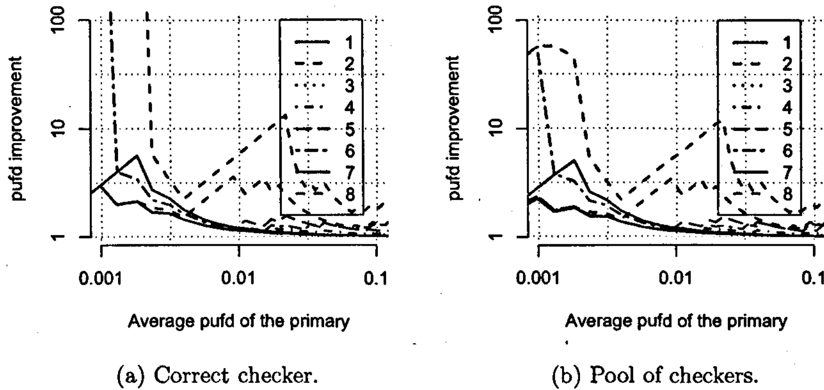
Some RTCs are very effective, others are not. Some are effective for low primary PFDs, others for high. It is however not predictable which RTCs will be effective, since this depends on factors as the demand space and the programming faults made in the primary.

The graph gives rise to one possible conclusion: RTCs may still be effective for reliable primaries.

4.2 Effectiveness of the RTCs for Decreasing Average PUFD

We now investigate the effectiveness of RTCs as a function of the average PFD of primaries. To vary the PFD, we take the pool of 196 primaries and we one after another remove primaries with the highest PFD. The result is shown in Figure 4.

As observed in our earlier paper [9], the effectiveness of RTCs shows a rather unpredictable pattern. The PFD-improvement of RTC1-7 remains well below a factor three in almost the entire graph. RTC6 and 8 become infinity for low PUFDs, but that is mainly caused by the fact that the number of primaries in this region becomes very small and the checkers manage to capture the faults in



(a) Correct checker.

(b) Pool of checkers.

Fig. 4. The effectiveness of the eight RTCs as with decreasing average PFD of the pool of primaries. The PFD is lowered by subsequently removing the most unreliable programs. (a) For a correct checker; (b) averaged over the pool of checkers.

these few programs. The numbers computed in this region of the graph should be considered with care. RTC8 is the conjunction of RTC1-7, and reaches an average PFD improvement of about a factor three.

When we compare the PUFD of the correct checker with the average of the checkers, we can observe that there is little difference, except for highly reliable primaries. Here, the performance of the checkers is reduced, because of faults in some checkers.

There may be a turning point at which a checker's effectiveness becomes questionable: when the ratio between false alarms and detection of primary failure becomes worse. Here we see a complex interplay between improving the quality of the primary and the checker. Improving the quality of the checker may have low priority, thus possibly resulting in poor specificity of the average checker.

5 RTCs vs. Multiple-Version Diversity

We now compare the effectiveness of RTCs with 1-out-of-2 diversity. We make a graph in the same way as Figure 4, except that we take two primaries from the pool instead of one.

We observe (see Figure 5) that 1-out-of-2 diversity becomes more effective with decreasing PFD of the pool of primaries from which the pair is selected. The reliability improvement ranges from a factor 25 to 100 for primary PFDs between 0.01 and 0.001. The effectiveness seems to reach a peak at a PFD of 0.001. (note that the opposite trend—effectiveness decreasing with decreasing mean PFD—is also possible, as proved by models and empirical results [6]). The

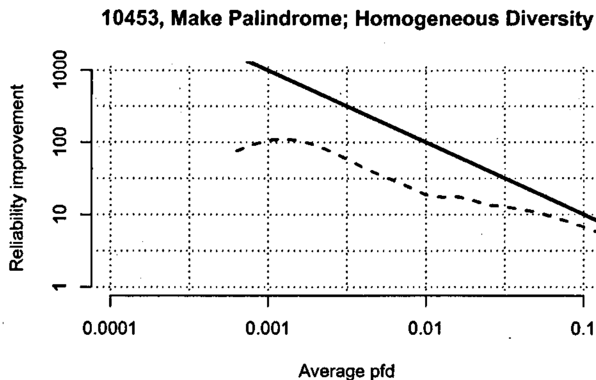


Fig. 5. Improvement of the PUFD of a pair of randomly chosen primaries, relative to a single version. The horizontal axis shows the average PUFD of the pool from which both primaries are selected. The vertical axis shows the PUFD improvement ($PUFD_A/PUFD_{AB}$). The diagonal represents the theoretical reliability improvement if the programs fail independently, i.e. $PUFD_{AB} = PUFD_A \cdot PUFD_B$.

improvement factor of most run-time checks remains fairly constant between 1 and 3 over this range, depending on the RTC. Only RTC8 reaches a factor of 10, when the PFD of the primaries is around 0.02. The improvement factor of using diversity is significantly higher than that of applying RTCs.

These results also confirm those in our earlier publication on the effectiveness of RTCs [9].

6 Conclusions

In this paper, we examined the effectiveness of Run-Time Checks without the assumption that these are fault-free. The effectiveness of the average checker appears to not deviate much from that of a perfect checker.

The effectiveness of the Run-Time Checks is comparable to that in our earlier study [9], a factor between one and three. We also confirmed the earlier result that multiple-version diversity is far more effective for decreasing the PUFD.

As yet, it seems not predictable which Run-Time Checks will be most effective, although in this study it is obvious that RTC8 will have the best coverage, simply because it is the conjunction of all the others. As a side effect, RTC8 also has the highest false alarm rate. Here again, it is hard to make a well-informed choice, because it is virtually impossible to predict the false alarm rate.

The results also show that Run-Time Checks may remain effective, also for the more reliable primaries. It may therefore be useful to keep the RTCs in primaries, also after extensive debugging. This however needs to be a trade-off with the possibility of raising false alarms.

We have to keep in mind that this research only considers one primary/Run-Time Checker combination, and that we can as yet not generalise, except perhaps for those observations that confirm those in our earlier publication on Run-Time Checks.

Acknowledgement

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council via the Interdisciplinary Research Collaboration on the Dependability of Computer Based Systems (DIRC), and via the Diversity with Off-The-Shelf Components (DOTS) project, GR/N23912.

References

1. M. Blum and H. Wasserman. Software reliability via run-time result-checking. Technical Report TR-94-053, International Computer Science Institute, October 1994.
2. A. Jhumka, F.C. Gärtner, C. Fetzer, and N. Suri. On systematic design of fast and perfect detectors. Technical Report 200263, École Polytechnique Fédérale de Lausanne (EPFDL), School of Computer and Communication Sciences, September 2002.
3. P.A. Lee and T. Anderson. *Fault Tolerance; Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems.* Springer, 2nd edition, 1981.
4. N.G. Leveson, S.S. Cha, J.C. Knight, and T.J. Shimeall. The use of self checks and voting in software error detection: An empirical study. In *IEEE Transactions on Software Engineering*, volume 16(4), pages 432–43, 1990.
5. B. Meyer. Design by contract. *Computer (IEEE)*, 25(10):40–51, October 1992.
6. P. Popov and L. Strigini. The reliability of diverse systems: A contribution using modelling of the fault creation process. *DSN 2001, International Conference on Dependable Systems and Networks, Göteborg, Sweden*, July 2001.
7. M. Rela, H. Madeira, and J.G. Silva. Experimental evaluation of the fail-silent behavior of programs with consistency checks. In *FTCS-26, Sendai, Japan*, pages 394–403, 1996.
8. S. Skiena and M. Revilla. *Programming Challenges.* Springer Verlag, March 2003.
9. M.J.P. van der Meulen, L. Strigini, and M. Revilla. On the effectiveness of run-time checks. In B.A. Gran and R. Winter, editors, *Computer Safety, Reliability and Security, Proceedings of the 23nd international conference, Safecomp 2005*, pages 151–64, Fredrikstad, Norway, September 2005.
10. J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson. Reducing critical failures for control algorithms using executable assertions and best effort recovery. In *DSN 2001, International Conference on Dependable Systems and Networks, Goteborg, Sweden*, 2001.
11. H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–49, 1997.

C.8 ISSRE 2007

M.J.P. van der Meulen and M.A. Revilla, Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs. In *Proceedings of the 18th IEEE International Symposium of Software Reliability Engineering*, pages 203–8, Trollhättan, Sweden, November 2007. Copyright ©2007 IEEE.

Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs

Meine J.P. van der Meulen
Centre for Software Reliability
City University
London EC1V 0HB, UK

Miguel A. Revilla
Department of Applied Mathematics
University of Valladolid
47011 Valladolid, Spain

Abstract

Software metrics are often supposed to give valuable information for the development of software. In this paper we focus on several common internal metrics: Lines of Code, number of comments, Halstead Volume and McCabe's Cyclomatic Complexity. We try to find relations between these internal software metrics and metrics of software dependability: Probability of Failure on Demand and number of defects.

The research is done using 59 specifications from a programming competition—The Online Judge—on the internet. Each specification provides us between 111 and 11,495 programs for our analysis; the total number of programs used is 71,917. We excluded those programs that consist of a look-up table.

The results for the Online Judge programs are: (1) there is a very strong correlation between Lines of Code and Halstead Volume; (2) there is an even stronger correlation between Lines of Code and McCabe's Cyclomatic Complexity; (3) none of the internal software metrics makes it possible to discern correct programs from incorrect ones; (4) given a specification, there is no correlation between any of the internal software metrics and the software dependability metrics.

1 Introduction

Software metrics have been subject of research since the seventies, and expectations were high that metrics would exist to help managerial decision making during the software lifecycle. Software metrics come in many flavours (e.g. described by Fenton e.a. in [3]). Essentially any metric is an attempt to measure or predict some attribute (internal or external) of some product, process or resource. Normally, the internal attributes are those that we can directly measure, and the external ones those that we are interested in [2]. In this paper we concentrate on a few internal, product-related software metrics: Lines of Code, number of comments, Halstead Volume and McCabe's Cyclomatic Complexity. We contrast these with two external software metrics: number of defects and Probability of Failure on Demand (PFD).

There have been many attempts to use software metrics in the development of software. Kafura reports that a collection of software metrics can be used to identify those components which contain an unusually high number of errors or which require significantly more time to code than the average [5]. In general however, the results have been ambiguous at least. Fenton, in [1], states: "Specifically, we conclude that the existing models are incapable of predicting defects accurately using size and complexity metrics alone. Furthermore, these models offer no coherent explanation of how defect introduction and detection variables affect defect counts."

In this paper we investigate which correlations between internal software metrics and dependability metrics exist by analysing a very large collection of small C/C++ programs submitted to the "Online Judge".

2 The experiment

2.1 The UVa Online Judge

http://acm.uva.es, the "UVa Online Judge"-Website [8], is an initiative of one of the authors (Revilla). It contains program specifications for which anyone may submit programs in C, C++, Java or Pascal intended to implement them. The correctness of a program is automatically judged by the "Online Judge". Most authors submit programs repeatedly until one is judged correct. Tenthousands of authors contribute and together they have produced more than 3,000,000 programs for the approximately 1,500 specifications on the website (as of May 2004, the programs submitted at that date are used in this experiment).

©2007 IEEE. Reprinted, with permission, from the Proceedings of the 18th IEEE International Symposium of Software Reliability Engineering, pages 203–8, Trollhättan, Sweden, November 2007.

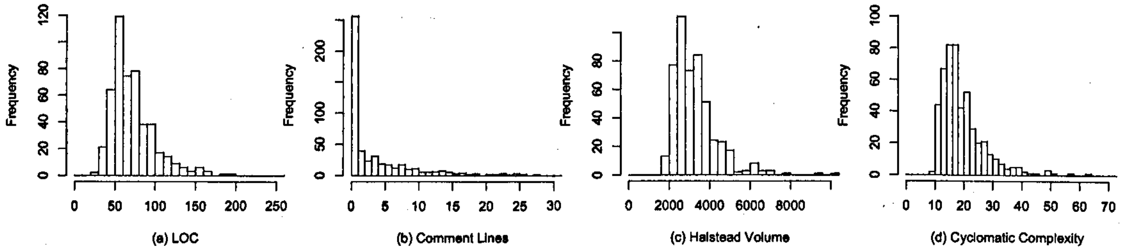


Figure 1. Histograms of internal software metrics of the correct submissions for the "Unidirectional TSP"-problem. The vertical axis presents the number of programs.

The program specifications of the Online Judge contain a short description of the problem, mainly in natural language, and some example input and output. The formalism of the specifications differs from specification to specification, but most are more or less informal.

Note that we do not use the test results of the Online Judge, except that we only use those programs that the Online Judge *can* run, see Section 2.2. We test the programs ourselves for determining the external metric PFD as explained in Section 3.2.

2.2 Selection of specifications and programs

We selected 59 specification from the Online Judge from different domains: graph theory, string operations, mathematical puzzles, etc. To enable statistical analysis, we selected specifications for which many submissions existed. From all the programs submitted to the Online Judge for these 59 specifications, we use:

Programs running under the Online Judge. We only use those programs that the Online Judge can execute, i.e. the Online Judge was able to compile the program, was able to run it, and the program runs using prescribed time and memory resources. This first filter saves us much time, and also protects us from malicious programs like fork bombs.

Programs in C or C++. We only chose programs in C or C++, because our tools calculate the internal metrics for these programming languages. Apart from this practical reason, mixing programming languages in this research might invalidate the results, or at least complicate their interpretation.

Programs that succeed for at least one demand. We excluded the completely incorrect submissions, because there is obviously something wrong with these in a way that is outside our scope (these are often submissions to the wrong specification, or apply incorrect formatting to the output).

First submission of each author. We only used one program submitted by each author and discard all other submissions (except for the determination of the number of defects, see Section 3.2). Subsequent submissions have comparable fault behaviour and this dependence between submissions would invalidate the statistical analysis.

Programs smaller than 40kB. A manipulation of the data we allowed ourselves is that we remove those programs from the analysis that have a filesize over 40kB. This is the maximum size allowed by the Online Judge, but was not enforced for a small period of time, and during this time some authors managed to submit programs exceeding this limit. Imposing this restriction does therefore only enforce a restraint that already in principle existed.

No look-up tables. We also disregarded those programs that consist of look-up tables, because their software metrics are completely different (the Halstead Volume is in general more than ten times the average for all programs written to a specification, thus completely dominating statistical analysis). These programs are very easily distinguishable from others, because they combine a very high Halstead Volume with a very low Cyclomatic Complexity. In rare cases a look-up table has a very high Cyclomatic Complexity, more than a hundred; in these cases the table is programmed with if-then-else statements. We deleted these programs manually from the analysis. In total 314 of the 41,685 remaining correct programs (0.75%), and 132 of the 30,232 remaining incorrect programs (0.43%) were disregarded.

The total number of submissions to the 59 specifications used in our analyses was 71,917, on average 1,219 per specification.

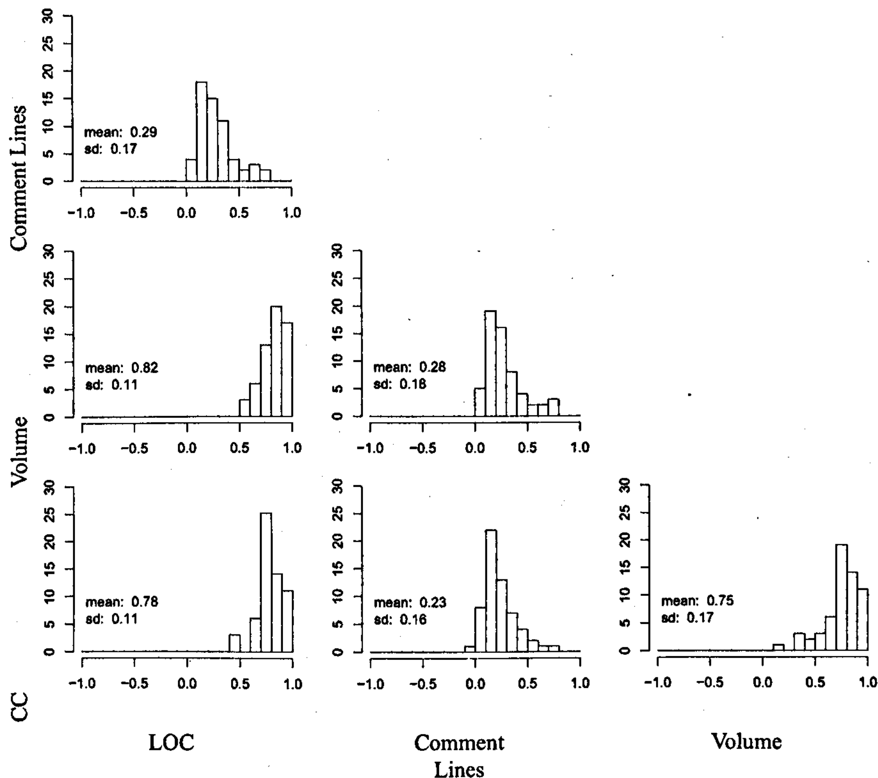


Figure 2. Histograms of the correlations between the various internal software metrics. The vertical axis depicts the number of specifications.

3 Measurement of software metrics

3.1 Internal metrics

We automatically measured the following internal software metrics: Lines of Code (LOC), the number of comment lines, the Halstead Volume (Volume) [4], and McCabe's Cyclomatic Complexity (CC) [6]. These four are very commonly used in the assessment of programs and many (commercial) tools give the possibility to measure them. The distributions of these metrics is similar for all specifications. Figure 1 presents a typical example for one specification, named "Unidirectional TSP".

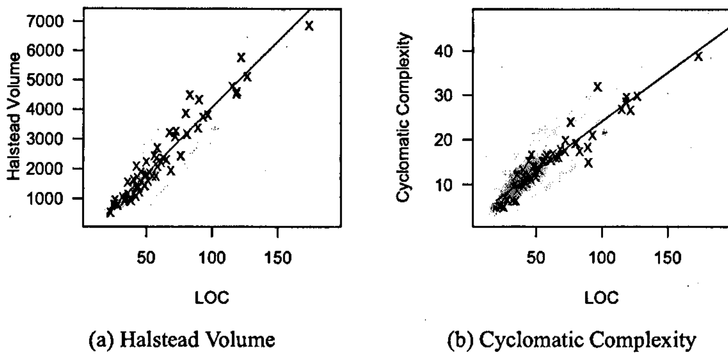
We determine the Cyclomatic Complexity as follows. We first measure the CC of the main body of the program and the subroutines and functions separately. We determine the CC of the entire program by summing the CCs of the constituent parts.

The Cyclomatic Complexity appears to have a broad range for every specification, Figure 1(d) gives it for the correct programs for the specification "Unidirectional TSP". The CC goes well beyond 50, and many correct programs have a CC above 20. This observation is valid for all specifications.

3.2 Dependability metrics

We also measured two external software metrics related to dependability: the Probability of Failure on Demand (PFD) and the number of defects (D).

To determine the PFD, we used three different testing strategies. For some specifications, a complete test is possible. For other specifications this is not the case, we then completely tested part of the demand space or we did a random test. The number of demands is either 2,500 or 10,000 for all specifications, except for those for which we did a complete test, in those cases the number of demands is equal to the number of possible demands.



(a) Halstead Volume

(b) Cyclomatic Complexity

Figure 3. Density plots of the LOC/Volume and LOC/CC distribution of all correct programs of all specifications combined. The x's are the means for the 59 specifications. The lines are unweighted regressions of these means. The background of the figure shows a density plot of the Volume-LOC and CC-LOC pairs of all the programs involved, with every specification having the same weight.

The PFD is the fraction of demands for which the program fails, i.e. we assume that every demand is equiprobable. This assumption seems to be as good as any other assumption, and we do not have any indication that it influences our results.

We determined the number of defects as follows. In our approach this is only possible for those authors who manage to submit a correct program (we disregard all submissions after a first correct submission). We assumed that the number of defects in this correct submission is zero, and in the penultimate submission is one. Then, we assess the submission before the penultimate submission of this author; if its behaviour is different from that of the penultimate submission, we add one to the defect count, otherwise we assume the defect count is the same. We repeat this procedure until the first submission of the author. The defect count of this submission is used for the analysis.

4 Correlations between internal software metrics

For every specification, we determined the correlation between the internal software metrics. This gives us 59 measurements for every pair of metrics. These are presented in histograms in Figure 2.

There are a some very strong correlations: CC vs. LOC (mean=0.78), Halstead Volume vs. CC (0.75), Halstead Volume vs. LOC (0.82).

The correlations between Comment Lines and LOC/Volume/CC (mean=0.29, 0.28 and 0.23) are rather unexpected. They might be explained by assuming that programmers who write comments, also tend to write more elaborate code. Writing comments in code for the Online Judge is completely voluntary.

5 Lines of Code vs. Halstead Volume and Cyclomatic Complexity

The correlations in Figure 2 suggest a very strong relationship between LOC on the one hand, and Volume/CC on the other. We would like to investigate this a little bit further.

We plot the mean Volume and CC of all the specifications against the mean LOC, see Figure 3. In both cases, the correlation is very strong (0.97 and 0.95). For the means, we determined the following regression lines:

$$Volume = 45 \times LOC - 428 \quad (1)$$

$$CC = 0.22 \times LOC + 1.9 \quad (2)$$

For CC this can informally be interpreted as: on average, programmers write a branch in almost every five lines of C/C++ code.

6 Are internal software metrics different for incorrect programs?

To discover whether there is a relationship of any of the internal metrics with correctness of program, we determined the ratio between the means of the internal metrics of the correct programs and those of the incorrect programs for every specification. This gives us 59 measurements for the

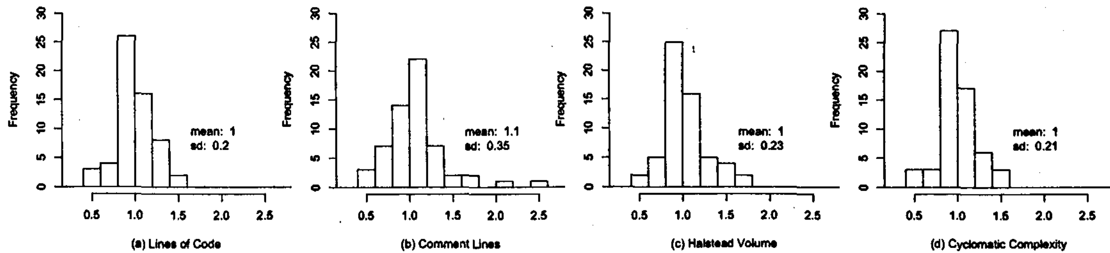


Figure 4. Histograms of the ratio of internal metrics of correct programs and incorrect programs for 59 specifications, for (a) LOC, (b) Comment Lines, (c) Halstead Volume, (d) Cyclomatic Complexity.

four internal metrics, which we depict in histograms in Figure 4. It appears that although the change in internal metrics can be large for some specifications, these changes go either way, and on average the internal metrics are practically the same for correct and incorrect programs.

Number of Comment Lines. There is no requirement to insert comments in the code submitted. Beforehand, we conjectured that the number of comment lines in submissions would correlate with lower PFD (also reported by Runeson [7]), the rationale being that someone who voluntarily adds comments, thinks about the program more carefully. However, only a small change in this metric was found, correct programs have on average 10% more comment lines; the spread on the distribution is very large.

Halstead Volume and Cyclomatic Complexity. On average, the Volume and the CC of correct and incorrect programs are the same.

7 Correlations between internal metrics and dependability

We determined the correlation between the internal software metrics and the two dependability software metrics. We depict this information in eight histograms in Figure 5.

Lines of Code. Figure 5 shows that in our experiment the mean correlation between LOC and number of defects or PFD is close to zero.

Number of Comment Lines. Figure 5 shows that the average correlation of the number of comment lines with the number of defects and PFD is very close to zero. This adds to our finding in Section 6: although correct programs have slightly more comment lines, it now appears that the *number* of comment lines is not correlated to the *number* of defects.

Cyclomatic Complexity and Halstead Volume. In our experiments, there is no correlation between these metrics and PFD nor number of defects.

8 Discussion

This research suffers from one major point of criticism: the programs are not 'real', they are most probably written by students and not by professional programmers. Even so, we argue that if there is a relationship between one of these internal software metrics and PFD and number of defects, this relationship would even be stronger if less rigorous programming methods are followed. A higher Cyclomatic Complexity is more problematic in a 'trial and error program' then in a rigorously developed program and becomes irrelevant if the correctness of a program is formally proved.

Another point of criticism is the size of the programs, varying between several dozens and several hundreds of lines of code. This is comparable to the size of subroutines, and the results should be interpreted at that level. We can only conjecture that these results are applicable to bigger, more realistic programs.

In our approach to counting defects, we assume that programs of the same author that have the same behaviour have the same number of defects and that the number of defects removed in one step is one. Both assumptions most probably lead to underreporting of defects. A possible other way to measure the number of defects is to simply use the number of attempts to the first correct submission. This approach would probably lead to overreporting, assuming that authors on average remove less than one fault per attempt. We also tried this approach, and the results are virtually the same, and for that reason we do not publish these.

9 Conclusion

We analysed the relations between various internal and external software metrics in a large collection of C/C++ programs submitted to a programming competition, the Online Judge.

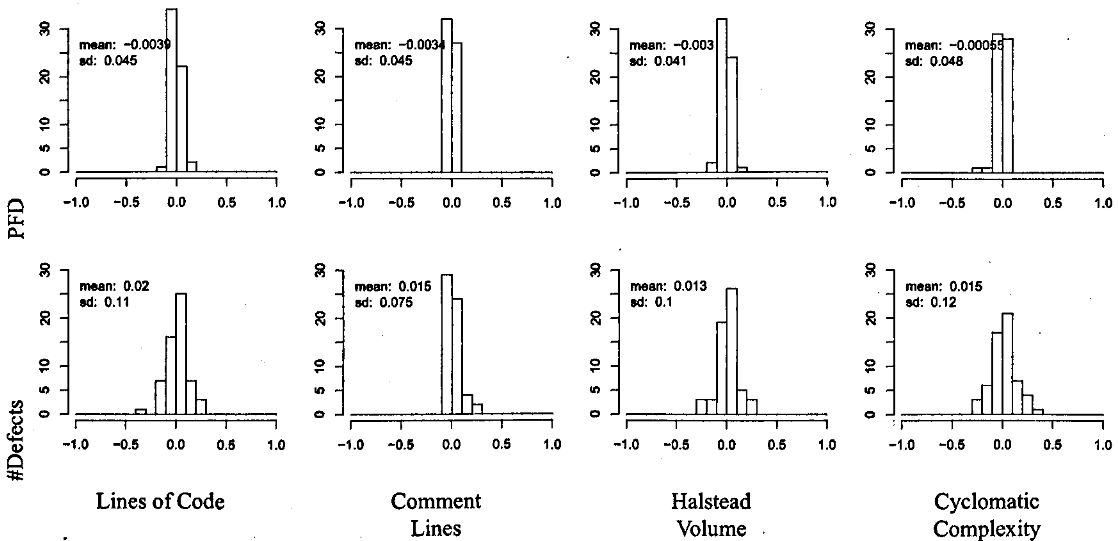


Figure 5. Histograms of the correlations between various internal software metrics and PFD or Number of Defects. The vertical axis reflects the number of specifications.

The experiment shows very strong correlations between the internal software metrics Lines of Code (*LOC*), Halstead Volume (*V*) and Cyclomatic Complexity (*CC*). We derived the following relations between the means of their distributions: $V = 45 \times LOC - 428$ and $CC = 0.22 \times LOC + 1.9$. These give the best estimates for V and CC when LOC is given.

In the experiment, the internal software metrics have no predictive power with respect to the Probability of Failure on Demand nor the number of defects of programs. The internal metrics are on average the same for correct and incorrect programs.

Acknowledgement

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council via the Interdisciplinary Research Collaboration on the Dependability of Computer Based Systems (DIRC), and via the Diversity with Off-The-Shelf Components (DOTS) project, GR/N23912.

References

[1] N.E. Fenton and M. Neill. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, SE-25(5):675–89, 1999.

[2] N.E. Fenton and M. Neill. Software metrics: Roadmap. In *Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland*, pages 357–70, 2000.

[3] N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigourous and Practical Approach.* International Thomson Computer Press, 1996.

[4] M.H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.

[5] D. Kafura and J. Canning. A validation of software metrics using many metrics and two resources. In *Proceedings 8th International Conference on Software Engineering, London*, pages 378–85, 1985.

[6] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), December 1976.

[7] P. Runeson, M. Holmstedt Jöhnsson, and F. Scheja. Are found defects an indicator of software correctness? an investigation in a controlled case study. In *The 15th IEEE International Symposium of Software Reliability Engineering, 2–5 November 2004, St. Malo, France*, pages 91–100, 2004.

[8] S. Skiena and M. Revilla. *Programming Challenges.* Springer Verlag, March 2003.

C.9 Transactions on Software Engineering

M.J.P. van der Meulen and M.A. Revilla, The Effectiveness of Software Diversity in a Large Population of Programs. Submitted to the Transactions on Software Engineering, October 2007.

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, TO BE PUBLISHED

The Effectiveness of Software Diversity in a Large Population of Programs

Meine J.P. van der Meulen, *Member, IEEE,* Miguel A. Revilla, *Member, IEEE*

***Abstract*—In this paper, we first present an exploratory analysis of the aspects of multiple-version software diversity using 36,123 programs written to the same specification. We do so within the framework of the theories of Eckhardt & Lee and Littlewood & Miller. We analyse programming faults made, explore failure regions and difficulty functions, show how effective 1-out-of-2 diversity is and how language diversity increases this effectiveness.**

The second part of the paper generalizes the findings about 1-out-of-2 diveristity, and its special case language diversity by performing statistical analyses of 89,402 programs written to 60 specifications. Most observations in the exploratory analysis are confirmed; however, although the benefit of language diversity can be observed, its effectiveness appears to be low.

***Index Terms*— Reliability, fault-tolerance, redundant design.**

I. Introduction

To date, software reliability research has been based on relatively small samples of programs; at most a few tens of programs have been used in controlled experiments to test hypotheses about multiple-version diversity [1], [2], [3], [4], [5], [6], [7]. Ideally far more programs—written to a common specification—are needed to undertake statistical analyses, and many different specifications are needed to demonstrate that the results are generally applicable.

The UVa Online Judge Website is an initiative of Miguel Revilla of the University of Valladolid [8]. It contains problems to which everyone can submit solutions. The solutions are programs written in C, C++, Java or Pascal. The correctness of the programs is automatically judged by the "Online Judge". Most authors submit solutions until their solution is judged as being correct. There are many thousands of authors and together they have produced more than 3,000,000 solutions to the approximately 1500 problems on the website.

From the perspective of algorithm design, the programming contest is a treasure trove. There appear to be numerous ways to solve the same problem. But also for software reliability engineers it is interesting: there are even more ways to *not solve* the problem. The first submission is often incorrect, but most authors eventually arrive at the correct solution.

Section II of this paper explains the theory of multiple-version diversity of Eckhardt & Lee [9], later generalized by Littlewood & Miller [10]. This gives the mathematical background for the calculations in the rest of the paper.

Then, in Section III, an exploratory analysis of multiple-version diversity follows, using the submissions to one of the specifications (the "3n+1 "-problem). This analysis highlights the issues involved, mainly confirming earlier published results.

Manuscript received 28 October, 2007.

Meine van der Meulen is with the Centre for Software Reliability at City University, London.

Miguel Revilla is with the University of Valladolid, Spain.

The main difference with these earlier studies is the amount of submissions: our study uses tens of thousands of submissions. Additionally, we are able to depict failure regions and difficulty functions.

Finally, in Section IV, we generalize the issues by calculating some of the observed properties and averaging them over similar analyses of sixty other specifications. In total, the study covers 89,402 programs, on average 1,466 per specification.

II. Theory

A. Modelling of Multiple-Version Diversity, Including Language Diversity

Two of the most well-known probability models in the domain of multiple-version diversity are the Eckhardt & Lee model [9] and its generalization, the Littlewood & Miller extended model [10]. Both models assume that:

1) Failures of an individual program are deterministic and a program version either fails or succeeds for each input value x. The failure set of a program π can be represented by a "score function" $\omega(\pi, x)$ which produces a zero if the program succeeds for a given x or a one if it fails.
2) There is randomness due to the development process. This is represented as the random selection of a program, Π, from the set of all possible program versions that can feasibly be developed and/or envisaged. The probability that a particular version π will be produced is $P(\Pi = \pi)$.
3) There is randomness due to the demands in operation. This is represented by the random occurrence of a demand, X, from the set of all possible demands. The probability that a particular demand will occur is $P(X = x)$, the demand profile. In this experiment we assume a contiguous demand space in which every demand has the same probability of occurring.

Using these model assumptions, the average probability of a program version failing on a given demand is given by the difficulty function, $\theta(x)$, where:

$$\theta(x) = \sum_{\pi} \omega(\pi, x) P(\Pi = \pi) \quad (1)$$

The average probability of failure on demand of a randomly chosen single program version Π_A developed using some method A can be computed using the difficulty function for method A and the demand profile:

$$\text{pfd}_A := P(\Pi_A \text{ fails on } X) = \sum_{x} \theta_A(x) P(X = x) \quad (2)$$

The average pfd for a pair of diverse programs, Π_A and Π_B, developed using methods A and B (assuming the system fails

when both versions fail, i.e. a 1-out-of-2 system) would be:

$$\begin{aligned} \text{pfd}_{AB} &:= P(\Pi_A \text{ fails on } X \text{ and } \Pi_B \text{ fails on } X) \\ &= \sum_x \theta_A(x)\theta_B(x)P(X=x) \end{aligned} \quad (3)$$

And with [10]:

$$\text{pfd}_{AB} = \text{pfd}_A.\text{pfd}_B + cov_X(\theta_A(X), \theta_B(X)) \quad (4)$$

If the difficulty function is constant for all x, and therefore $cov_X(\theta_A(X), \theta_B(X)) = 0$, the reliability improvement for a diverse pair will (on average) satisfy the independence assumption:

$$\text{pfd}_{AB} = \text{pfd}_A.\text{pfd}_B \quad (5)$$

If the difficulty function is not constant, the impact of the difficulty functions differs between the Eckhardt & Lee model and the Littlewood & Miller model. The Eckhardt & Lee model assumes similar development processes for the two programs A and B and hence identical difficulty functions. i.e. that $\theta_A(x) = \theta_B(x)$. So the average pfd for a pair of diverse programs reduces to:

$$\begin{aligned} \text{pfd}_{AB} &= \sum_x \theta_A(x)\theta_B(x)P(X=x) \\ &= \text{pfd}_A^2 + var_X(\theta_A(X)) \end{aligned} \quad (6)$$

It is always the case that $var_X(\theta(X)) \geq 0$, and therefore:

$$\text{pfd}_{AB} \geq \text{pfd}_A^2 \quad (7)$$

In practice the variance plays an important role, and is often the main factor of system unreliability. The intuitive explanation for this is that it is harder for the program developers to properly deal with some demands. The difficulty function will then be "spiky", and the diverse program versions tend to fail on the same demands. Diversity is then likely to yield little benefit and pfd_{AB} could be close to pfd_A.

The Littlewood & Miller model does not assume similar development processes for program versions A and B, and the pfd of the pair remains:

$$\text{pfd}_{AB} = \text{pfd}_A.\text{pfd}_B + cov_X(\theta_A(X), \theta_B(X)) \quad (8)$$

We will use the Littlewood & Miller model in our language diversity experiments in which the pair consists of two programs in different programming languages.

III. Exploratory Analysis

A. The Problem

We analyse a single set of 107,522 C, C++ and Pascal programs version written to a common specification, the "3n+1"-problem. The "3n+1"-problem can be summarised as follows:

```
1. input n
2. print n
3. if n = 1 then STOP
4.     if n is odd then n := 3n + 1
5.     else  n := n/2
6. GOTO 2
```

For example, given an initial value 22, the following sequence of numbers will be generated 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1.

It is conjectured that the algorithm above will terminate (i.e. stop at one) for any integer input value. Despite the simplicity of the algorithm, it is unknown whether this conjecture is true.

Given an input n, it is possible to determine the length of the number sequence needed to reach the final value of one. This is called the cycle-length of n. In the example above, the cycle-length of 22 is 16.

The "3n+1"-problem specification includes the following requirements:

- For any two numbers i and j, determine the maximum cycle-length over all integers between and including i and j.
- The input will consist of a series of pairs of integers i and j, one pair of integers per line. All integers will be less than 1,000,000 and greater than 0.
- For each pair of input integers i and j the output is i, j, and the maximum cycle-length for integers between and including i and j. These three numbers should be separated by at least one space with all three numbers on one line and with one line of output for each line of input.

The specification is supplemented by sample input and output examples, e.g.:

Sample Input:

```
1 10
100 200
```

Sample Output:

```
1 10 20
100 200 125
```

B. Testing

We submitted all the programs to a benchmark test. The benchmark is constituted of 2,500 pairs of numbers with all combinations of numbers between and including 1 and 50. We call every pair of inputs a demand. The demands are processed by the programs separately, i.e. the programs are restarted for every demand. The outputs of the programs' executions are written to a file for later analysis.

It has to be noted that this benchmark approach does not claim to identify all faults in the programs. An example of a known fault that will not be found in this analysis is numerical overflow, caused by intermediate results becoming very large, because the small numbers used in the benchmark will not trigger these overflow conditions. Another fault that this benchmark approach will not identify is an arbitrary limit on the number of lines of input a program can accept (e.g. 4, 10, or 50 lines) because the program is restarted for each demand. It also follows that we cannot identify faults that depend on particular sequences of input lines to a program as we only use a single line of input per demand. Examples of the latter two types of fault were identified in an earlier study presented at the ISSRE conference in St. Malo [11].

We were slightly more generous than the Online Judge in assessing the output files. We only compared the numbers in the output file, so if the output file contains commas, empty lines or short text like "The answer is:" we still treat it as a correctly formatted output. The reason for ignoring commas and short texts might be questioned, but this decision significantly reduces the number of different equivalent failure behaviours observed and enhances the opportunities for analysis.

A side effect of our testing programme was an independent check of the effectiveness of the Online Judge's benchmark. The

benchmarks of the Online Judge proved to be quite accurate, only few of the programs that we identified as incorrect were accepted. We were able to generate minimum sets of extra tests to add to the benchmarks. The augmented benchmarks findy all the programs that we classified as incorrect.

When running and comparing these programs it was striking how many different behaviours were observed. In total, 1991 different output behaviours were generated when the benchmark test was applied to the set of programs. In many cases the outputs were only slightly different, but the fact that such a simple program can be programmed incorrectly in so many ways is surprising.

In the following paragraphs, we provide some statistics of the submissions, Section III-C, and the structure of the submitted programs, Section III-D.

Three main analyses were performed in this exploratory study:

- Analysis of the types of fault introduced (see Section III-E).
- Assessment of the effectiveness of multiple-version diversity (see Section III-F).
- Assessment of the effectiveness of language diversity (see Section III-G).

C. Program submissions

The number of programs submitted to this problem was 107,522 when this analysis was performed. Table I presents some statistics, sorted by programming language.

The table also presents the judgement of the Online Judge. The Online Judge assesses all submissions, and has many categories ranging from "Compilation error", "Time limit exceeded", "Memory limit exceeded" to "Run-time error". In our analysis we only consider those submissions that are in the categories "Accepted", "Wrong answer" or "Presentation error". The latter category contains solutions that do not exactly conform to the output specification, but slightly differ in terms of presentation of the output. We do not consider the other categories in our analyses because these do not produce meaningful output.

There are some discrepancies between the verdict of the Online Judge and ours. The most important reason for these is that we are more generous in accepting solutions, e.g. by accepting solutions that output some additional text. Also, the Online Judge does not perform a complete test, and sometimes our tests reveal incorrect outputs, not identified by the Online Judge. We therefore always retest submissions, irrespective of the verdict of the Online Judge.

D. Solutions to the problem

The example C program in Table II shows the approach most programs take. We will use the program's characterisation to describe the faults that authors make.

Of course, the actual programs differ from this example, but most programs take a similar approach and only differ in aspects such as: the use of subroutines for the cycle-length calculation or the determination of the maximum value. The programs that differ most from the example are those that optimize on speed. These programs can be lengthy and complex, but constitute a minority.

E. Analysis of program faults

1) Equivalence classes: We observed that there were many different programs that produced identical results. These were generally due to the existence of similar faults in the different versions. We grouped the program versions that produced identical results into "equivalence classes" and used these equivalence classes in our subsequent analysis.

After grouping the output files of the programs into equivalence classes, we characterised them by the faults they contained (see Table IV). The 36 equivalence classes are shown, with their total frequencies and their totals for each programming language, their reliability (i.e. the fraction of correct responses to the 2,500 demands), and a description of the faults that were identified as being present in that class of programs. An assumption we made here is that programs that behave similarly contain the same kind of faults. This may not always be correct, but only very few counterexamples have been found.

2) Types of fault: We can characterise the faults found in each equivalence class as follows.

Swap: missing or incorrect. This is related to demands where input i is larger than input j. This is normally handled by swapping the two input values. (Strictly speaking a swap is not necessary, because this functionality can also be implemented in the loop by counting down, but most authors do not use this alternative solution. So we have labelled this a "Swap" problem).

A missing swap indicates incorrect interpretation of the specification: the author did not anticipate the possibility that the second input may be smaller than the first. This is the most frequent mistake: 33% of the authors' first submissions in the selected equivalence classes exhibit this problem.

Incorrect implementation of the swap is less frequent (5.5%), in most cases the author did not consider the consequences for exchanging i and j. In some cases it is caused by a slip in a routine programming task.

Returning the input values in the incorrect order is one of the possible consequences of implementing the swap incorrectly. The specification clearly states that the returned inputs should appear in the same order. The author manages to implement the swap, but forgets to consider the consequences for the "write"-step. The problem is in general solved by either returning the inputs before swapping or by remembering the order of the inputs in separate variables.

Loop. There appear to be many ways to implement the loop incorrectly (2.9%). Most frequent is the omission of the last value in the loop. An example is: `for (StartSequence = StartCounter; StartSequence < LastCounter; StartSequence++)`.

Another case is the omission of the first and the last values in the loop, e.g.: `for(i = min(a,b) + 1; i < max(a,b); i++)`.

Calculation. Very few programs (3.1%) contain a fault in the calculation of the maximum cycle-length. This is probably due to the fact that if the algorithm responds well to the sample outputs given in the problem specification, it will perform well for all inputs. The main problem found is putting step 3, testing for $n = 1$, after step 4 and 5 in the program (see pseudocode in introduction). The program will not check for $n = 1$ immediately, leading to the sequence "1 4 2 1" and a calculated sequence length of 4 instead of 1.

Output. There are also various ways to incorrectly format the output (1.8%). Programs may not print the two inputs, and they may not print any output at all (sometimes conditionally, e.g. when $i > j$).

TABLE I
STATISTICS ON SUBMISSIONS TO THE "3N+1"-PROBLEM. THE FILTER INCLUDES ALL SUBMISSIONS ASSESSED BY THE ONLINE JUDGE AS "ACCEPTED", "WRONG ANSWER" OR "PRESENTATION ERROR", AND EXCLUDES ALL SUBMISSIONS AFTER A CORRECT SUBMISSION OF THE SAME AUTHOR.

Filtered	Parameter	C	C++	Pascal	Total
	Number of authors	5,897	6,097	1,581	13,575
	Number of authors solving the problem	5,020	5,141	1,152	11,313
	Percentage of authors solving the problem	85%	84%	73%	83%
Not filtered	Number of submissions	46,397	49,191	11,934	107,522
	Online Judge assessment: accepted	12,056	12,999	2,453	27,508
	Online Judge assessment: presentation error	391	365	26	782
	Online Judge assessment: wrong answer	15,271	17,048	4,809	37,128
Filtered	Number of submissions	15,705	16,225	4,193	36,123
	Average number of submissions	2.6	2.7	2.7	2.7
	Online Judge assessment: accepted	4,442	4,198	816	9,456
	Online Judge assessment: presentation error	160	184	7	351
	Online Judge assessment: wrong answer	11,103	11,843	3,370	26,316

TABLE II
EXAMPLE PROGRAM WITH TYPICAL ALGORITHM. (NOTE THAT THIS PROGRAM DOES NOT COMPLY WITH THE CURRENT VERSION OF THE C STANDARD.)

Program	Characterisation
`#include <stdio.h>`	
`#include <stdlib.h>`	
`main()`	
`{`	
`int a, b, min, max, num;`	Variable declaration
`register n, cycle, cyclemax;`	
`while (fscanf(stdin, "%d %d", &a, &b) != EOF) {`	Read inputs
`if (a < b) {min=a; max=b;} else {min=b; max=a;}`	Swap inputs
`for (cyclemax=-1,`	Reset maximum cycle-length
`num=min; num<=max; num++) {`	Loop
`for (n=num, cycle=1; n != 1; cycle++)`	
`if (n % 2) n=3*n+1; else n >>= 1;`	Calculation
`if (cycle > cyclemax) cyclemax=cycle;`	
`}`	
`printf ("%d %d %d\n", a, b, cyclemax);`	Output
`}`	
`}`	

Most faults related to poor interpretation of the specification: In particular common faults were related to:

- Not realising that the second input can be smaller than first. This was not mentioned in the specification but the author should not assume otherwise.
- Not realising that returned input values should be in the same order. This is explicitly mentioned in the specification.

3) Failure sets: Figure 1 shows graphic representations of 36 failure sets. For each input pair i, j a failure is indicated by a black dot, the 50*50 demands produce two-dimensional maps for each equivalence class.

The triangular pattern, e.g. (b), (e) and (k), is related to the i, j swap problem, i.e. the correct answer is only generated when i is less or equal to j. The diagonal structures like (h), (i) and (r) are related to loop implementation problems where either one or both of the i, j endpoint values is not included in the cyclic length calculation. An entirely black square, (f), is associated with problems like failing to generate any output or outputting in the wrong format. The most common equivalence class is a completely blank square, (a), which represents the case where all test inputs were correctly evaluated.

We also see regions that appear to be the superposition of two different failure sets, for example, (d) seems to be the superposition of (b) and (h). This might be the explanation for the large number of different equivalence classes found in the study. For example, 256 different failure set patterns can be generated with combinations of 8 basic patterns. Even more combinations are possible when we note that different equivalence classes can have the *same* pattern but have *different* (but equally wrong) output values.

F. Multiple-Version Diversity

We will first investigate the effectiveness of multiple-version diversity using a common difficulty function (the Eckhardt & Lee model). For this analysis, we assume that both programs in the diverse pairs are taken at random from the first submissions of all the authors (a common "pool" of programs). The difficulty functions for both programs are therefore the same. To estimate the pfd using equations 2 and 3, we need to specify the input profile $P(x)$. Assuming that all inputs are equally likely, we can compute the expected pfd for a randomly chosen single version and a randomly chosen diverse pair. The difficulty function $\theta(x)$

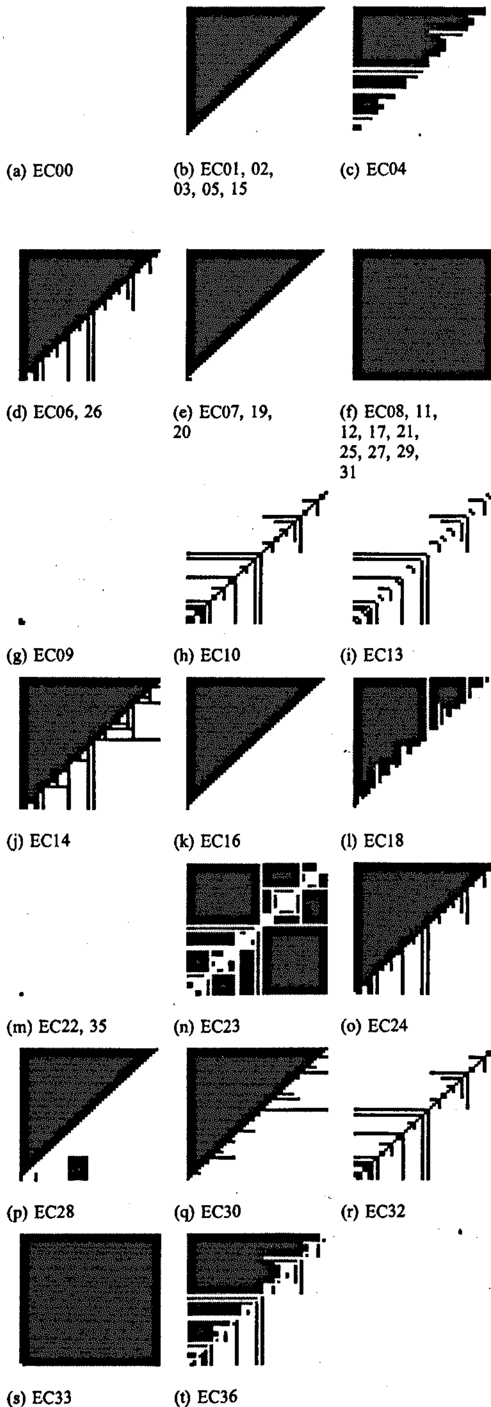


Fig. 1. Failure sets for the equivalence classes.

is fairly simple to derive: for each point in the input space we add up the number of program version that fails and divide by

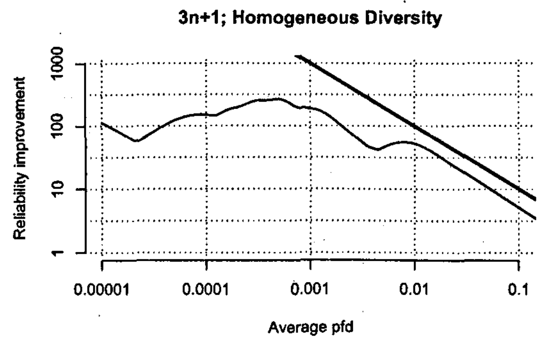


Fig. 2. Reliability improvement of a diverse pair, relative to a single version. The horizontal axis shows the average pfd of the pool from which both programs are selected. The vertical axis shows the reliability improvement ($\text{pfd}_A/\text{pfd}_{AB}$). The diagonal represents the theoretical reliability improvement if the programs fail independently, i.e. $\text{pfd}_{AB} = \text{pfd}_A . \text{pfd}_B$.

the total number of program versions.

As the difficulty surface is the weighted average of the failure sets of the individual equivalence classes, it is not surprising that the surface is dominated by the most frequently occurring failure set: the triangular region of the "swap"-fault.

Since the effect of diversity is likely to be dependent on the quality of the "pool" of programs, we artificially manipulate the average pfd of the pool by removing programs from the pool, starting with the most unreliable ones. This is comparable to the approach taken in the "four university experiment" [3].

Figure 2 shows the reliability improvement of the pair with respect to a single version. The horizontal axis shows the average pfd of the programs of the pool. For every pool, we compute the reliability of the pair using equations 2 and 3. The vertical axis shows the reliability improvement, i.e. $\text{pfd}_A/\text{pfd}_{AB}$.

We can observe that for the more unreliable pools, on the right hand side of the graph, the reliability improvement is close to the independence assumption. When the pools become more reliable, the effectiveness of diversity flattens. For pools with a pfd lower than 5.10^{-3} the reliability improvement is approximately a factor of a 100. Eckhardt & Lee earlier observed this in [6].

Figure 4 illustrates how the difficulty function of the pool changes as function of its pfd. It shows (e.g. by comparing the difficulty functions with the shapes of the failure sets in Figure 1, or by looking at the distribution of equivalence classes, as we did), and their description in table IV) that "swap"-faults are dominant in (a), "loop"-faults in (b), and "calculation"-faults in (c) (although some other faults seem to also play a role in this latter category).

G. Language Diversity

To improve the reliability of diverse pairs, one may try to force diversity between the pools from which the programs are selected. A frequently made assumption is that programmers using different programming languages make different mistakes, and that the failure behaviour of the pairs will be better than it would have been when the pools would have been the same (as in the previous section).

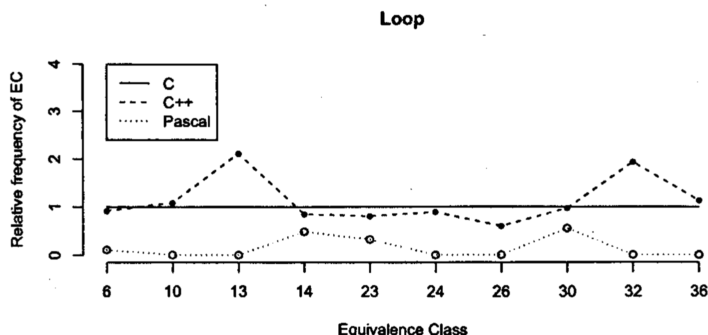


Fig. 3. Relative frequencies of occurrence rates of equivalence classes, as compared to their occurrence rate in C, for those equivalence classes with a fault classified as "loop".

The present experiment provides some indications that the programmers indeed make different mistakes in different programming languages. In this case the difference is largest and most clearly observable in the equivalence classes with a fault in the construction of the loop. Figure 3 shows the relative frequencies of the occurrence rates of the equivalence classes as compared to their occurrence rates in C (obviously, the relative frequencies for C are all 1). The figure clearly shows that faults in the construction of the loop are rare in Pascal in most equivalence classes.

Further investigation of the Pascal programs in these equivalence classes provides the following information:

- There are some occurrences of faults in the loop in equivalence class 6. Investigation of these cases shows that the author did not use the normal Pascal "for"-loop, but constructed it using a "while" and made a mistake.
- The cases in equivalence class 14 are caused by authors writing e.g.: `for f:=i+1 to j-1 do`. Even Pascal cannot protect against this kind of mistake.
- The fault in equivalence class 23 is caused by misinterpretation of the specification. The program only calculates the sequence length for i and j, and returns the maximum of the two as the result of the calculation. This fault can be made both in C and in Pascal.
- The cases in equivalence class 30 are caused by authors writing: `for f:=i+1 to j do`, or by constructing the "for"-loop with a "while" and making a mistake.

We conclude that if an author uses the "for"-loop in Pascal, the probability of a mistake in the construction of the loop is far smaller than when using the "for"-loop in C or constructing the loop with "while".

Note that the issue of the discussion above is not to conclude that Pascal is a better program language than C. The point we make is that different mistakes are made, depending on the programming language. This makes language diversity interesting for improving the effectiveness of multiple-version diversity.

From the above, it appears that the relative frequencies of faults made in Pascal programs differ from those in C or C++ programs, and indeed, this has also been observed by other researchers e.g. [6]. Based on this, language diversity may lead to more reliable pairs than homogeneous diversity. To examine this conjecture, we conducted the same diversity experiment as in Section III-F, but selected the two programs from different pools, each pool only containing programs in C, C++ or Pascal. Figure 5 presents the results of these calculations.

The figure shows the reliability improvement trend is similar to that for homogeneous diversity: the reliability improvement of the pair is close to the independence assumption when the average failure rate of the programs in the pool is high, and reaches a "plateau" of around 100 when the average failure rate is low. However, between these two extremes, we can observe differences between same/similar language pairs (C/C, C/C++, C++/C++ and Pascal/Pascal) and diverse language pairs (C/Pascal, C++/Pascal). The diverse language pairs systematically perform better in the reliability region of the pools between 10^{-3} and 10^{-2}.

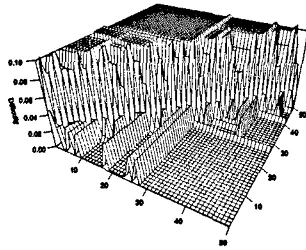
IV. Statistical Analysis

We will now investigate whether the observations in our exploratory analysis can be generalized. We do so by repeating some of the analyses for sixty more specifications. We calculate the effectiveness of multiple-version diversity and language diversity for all these specifications and we then statistically analyse the results.

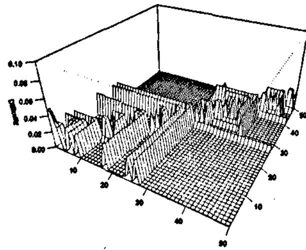
A. Selection of the specifications

We selected specifications from the Online Judge, based on the following criteria:

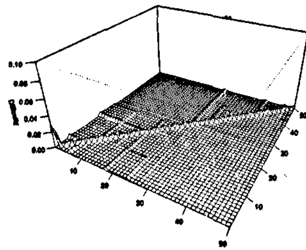
- There are more than 1,000 submissions for the problem. It is necessary to have a high number of submissions, to be able to perform the statistical analyses in this paper.
- There is only one correct output for each input. In some cases, we accept rounding-off, although none of the problems specifies the maximum error. In these cases, we use a "reasonable" estimate of the maximum round-off error.
  We do not accept problems that can have different correct answers, because multiple-version diversity is not a valid fault tolerance technique in these cases: if the answers differ, the system must be deemed to have failed, even if they are both correct. (However, other diversity mechanisms like Run-Time Checks may still work [12].)
- It is relatively easy to generate test cases automatically. In some cases this is difficult, e.g. when the input consists of



(a) pfd=5.10^{-2}

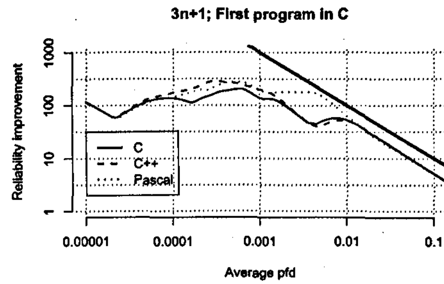


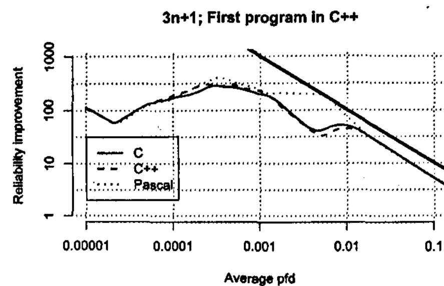
(b) pfd=5.10^{-3}



(c) pfd=5.10^{-4}

Fig. 4. Difficulty functions for three of the pools of programs used to generate Figure 2, for average pfds of 5.10^{-2}, 5.10^{-3}, and 5.10^{-4}. The three graphs show the gradual removal of failure sets from the pools. In (a) "swap"-faults are dominant, in (b) "loop"-faults, and in (c) "calculation"-faults.





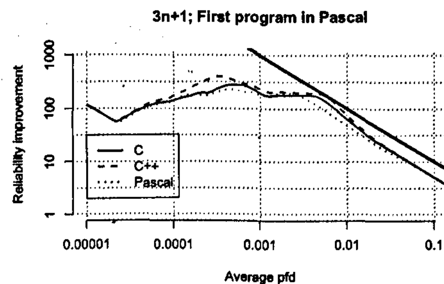


Fig. 5. Reliability improvement for 1-out-of-2 pairs with diverse language diversity. Each figure is for a given first choice of programming language for the first program and depicts the reliability improvement depending on the choice of programming language for the second program.

plain text or must obey a complicated mathematical criterion, these specifications have not been selected, because the creation of the benchmark may then in itself be error-prone.

As for the "3n+1"-problem, we only use first submissions, that were classified as either "AC", "WA", or "PE" by the Online Judge. This reduces the number of submissions actually used in our analysis. Figure 6 shows a histogram of the number of submissions used for the specifications. We analysed 61 specifications, with a total number of 89,402 submissions, of which 30,130 (34%) were written in C, 48,369 (54%) in C++ and 10,877 (12%) in Pascal.

B. Testing

We systematically tested all the programs, using three different test regimes:

- Complete test. This is used when the demand space is small, less than 5,000 possible demands.
- Random test. This is used when the demand space is large. The Perl "rand()"-function is used for this purpose. The number of demands is 2,500 or 10,000.
- Complete test of a part of the demand space. This testing strategy is sometimes used instead of Random Test, for example in the "3n+1"-problem in our exploratory analysis. The number of demands is at least 2,500.

We found no differences with respect to the findings in this paper related to the choice of test regime or number of demands.

Depending on the problem, testing takes between approximately several hours and several weeks on a 3.2 GHz Linux machine with 2 Gigabyte RAM.

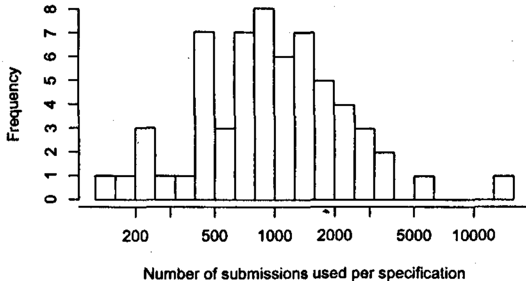


Fig. 6. Histogram of the number of submissions used for the specifications.

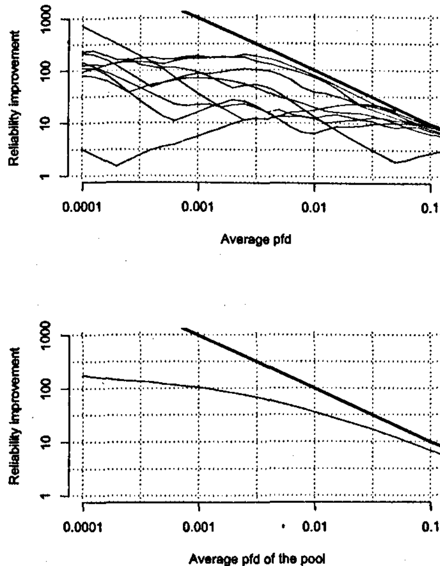


Fig. 7. The reliability improvements of multiple-version diversity (a) for ten randomly chosen specifications, (b) averaged over sixty specifications.

C. Multiple-Version Diversity

For each specification, we computed the average PFD-improvement for pairs of two programs randomly taken from the pool of programs, in the same way as described in Section III-F. We cut off the graphs on both sides, on the left hand side, when there is no incorrect program in the pool left; on the right hand side, when the pfd of the pool is reached.

Figure 7(a) shows the reliability improvements for ten different specifications (depicting more specifications makes the graph unreadable, the observations remain the same). Figure 7(b) shows the average over the sixty specifications. These figures confirm the earlier observations in our exploratory analysis:

- For unreliable pools, the average reliability improvement is close to the independence assumption.
- For reliable pools, the average reliability improvement levels off to a factor of around a hundred.

However, as can be seen in Figure 7(a), the graphs of some specifications do not follow this pattern. Sometimes, the effectiveness of multiple-version diversity decreases with decreasing pfd of the pool. This is the case when only one fault becomes dominant in the most reliable programs. This for example applies to the specification "Just the facts (00568)", in which programs tend to fail for inputs that are multiples of powers of five. The most reliable incorrect programs all fail for the input 9375 (3×5^5). In those cases, the effectiveness of multiple-version completely disappears.

The average PFD-improvement over the range 10^{-4}–10^{-3} is 135, over the range 10^{-3}–10^{-2} it is 68 and over the range 10^{-3}–10^{-2} it is 19.

Figure 8 presents a histograms of the PFD-improvement for all specifications at PFDs of the pool of 5.10^{-4}, 5.10^{-3} and 5.10^{-2}. We observe that the spread of the improvement is very large, especially for the most reliable pool, which means that the actual improvement of the pfd is hard to predict. But even though there is no guarantee about the reliability of a particular diverse pair, it nevertheless seems sensible to use results such as these as a guide for selecting methodologies for building diverse systems.

There is no correlation between the effectiveness of 1-out-of-2 diversity and the number of submissions to a specification (0.09). We checked this, because one may argue that the diversity increases with an increasing number of submissions, and therefore the effectiveness of diversity. This effect is not observeable, indicating that the size of the experiment is large enough.

D. Language Diversity

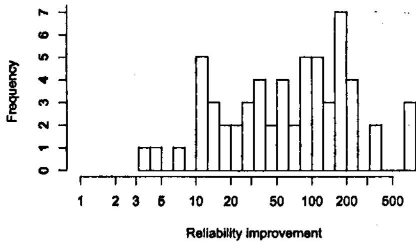
We will now address the average effectiveness of language diversity. This is complicated by the fact that partitioning the entire pool of programs in different pools will in itself increase diversity between the pools, and this effect will be different for the three programming languages, because the size of their pools differ. To judge the effect of language diversity, we have to differentiate between the effects of the partitioning and of language diversity. This is mainly important for the Pascal pools, because the number of Pascal programs for some problems is not very high, and the effects of partitioning will then be most apparent.

We measured the effect of the partioning itself by randomly partitioning the set of programs of a problem in the pools, whilst retaining the proportion of C, C++ and Pascal programs for that problem. We subsequently measured the effectiveness of software diversity of programs taken from these different pools. For every problem, we did 20 of such measurements and our estimate was the average over those measurements.

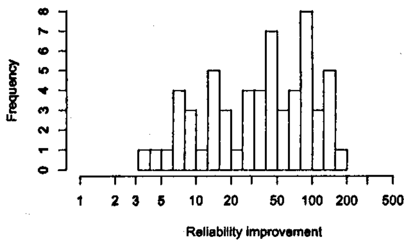
We then computed the average PFD-improvement for pairs of two programs taken from three different pools of programs: C, C++ and Pascal, in the same way as in our exploratory analysis (Section III-G).

Table III shows the average additional (as compared to the random improvement as calculated above) PFD-improvements for the nine possible language pairs.

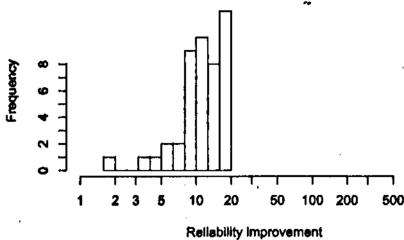
The PFD-improvements of the C/C, C++/C++, and Pascal/Pascal pairs are less than unity. We explain this by the fact that these pairs actually have less diversity than the random pairs.



(a) 5.10^{-4}



(b) 5.10^{-3}



(c) 5.10^{-2}

Fig. 8. Histograms of the PFD improvement of implementing 1-out-of-2 diversity for the sixty specifications, at a pfd of the pool of (a) 5.10^{-4}, (b) 5.10^{-3} and (c) 5.10^{-2}.

We also observe that a C/C++ pair is only slightly better than random. This improvement is probably so little, because most programmers do not use the special C++ language features.

The C/Pascal and C++/Pascal pairs perform best, but the improvement is only around 10%. The improvement may be so small because the difference between the C and Pascal language is not very big.

V. Discussion

In presenting these results it is important to note any limitations in their applicability to software engineering in general. There are a number of issues involved in using programs from a contest host site.

- Disparities in programmer experience and expertise.

TABLE III

ADDITIONAL PFD-IMPROVEMENTS FOR THE LANGUAGE PAIRS.

	C	C++	Pascal
C	0.95	1.02	1.13
C++	1.02	0.97	1.07
Pascal	1.13	1.07	0.97

- Disparities in the size and complexity of the specifications and the programs.
- Disparities in the software development process.
- Bias in program submissions, e.g. multiple submissions under different names or by submitting programs produced by a group of people.

As there are no large-scale data sources that are free from such bias, the only way forward is to take account of the limitations and to be careful about what observations can be generalised.

It must be recognised that both the specifications and the programs are much smaller than those used in industrial scale software. Also there is no control over the engineering process used to develop individual releases. So the results produced here may be more typical of "programming in the small" rather than "programming in the large" and the faults might be similar to those present in a single program module produced by a programmer prior to verification and validation.

VI. Conclusion

Our analyses confirm many of the findings in earlier work about software diversity. General observations are:

- The effectiveness of multiple-version diversity is close to the independence assumption for unreliable programs.
- The effectiveness of multiple-version diversity levels off for reliable programs, on average a factor of approximately a hundred for program pools with an average pfd of better than 0.001.
- In different programming languages, programmers may make different faults. In our exploratory analysis of the "3n+1"-problem this was shown to be the case for loops. Pascal programmers appear to be much less likely to make mistakes in loops then C/C++ programmers.
- Because programmers may make different faults when programming in different programming languages or similar mistakes may have different effects, language diversity may lead to a higher benefit of multiple-version diversity. In our experiments, we observed a maximum additional pfd gain of around 13% for C/Pascal pairs. This gain is small compared with the gain of multiple-version diversity only.

A correlation with program size (Lines of Code) has not been found in any of the observations. It must be stressed however that the programs analysed are all small.

Acknowledgment

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council via the Interdisciplinary Research Collaboration on the Dependability of Computer Based Systems (DIRC), and via the Diversity with Off-The-Shelf Components (DOTS) project, GR/N23912.

References

[1] J. Kelly and A. Avizienis, "A specification-oriented multi-version software experiment," in *Thirteenth International Symposium on Fault Tolerant Computing (FTCS-13)*, Milan, Italy, June 1983.

[2] L. Gmeiner and U. Voges, "Software diversity in reactor protection systems: an experiment," in *Safety of computer control systems*, R. Lauber, Ed. New York: Pergamon, 1980.

[3] J. Knight and N. Leveson, "An empirical study of failure probabilities in multi-version software," in *16th International Symposium on Fault-Tolerant Computing (FTCS-16)*, Vienna, Austria, 1986, pp. 165–70.

[4] P. Bishop, D. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti, "PODS project on diverse software," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp. 929–40, 1986.

[5] A. Avizienis, M. Lyu, and W. Schütz, "In search of effective diversity: A six language study of fault tolerant flight control software," in *18th International Symposium on Fault Tolerant Computing (FTCS 18)*, Tokyo, Japan, June 1988, pp. 15–22.

[6] D. Eckhardt, A. Caglayan, J. Knight, L. Lee, D. McAllister, M. Vouk, and J. Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability," *IEEE Transaction on Software Engineering*, vol. 17, no. 7, pp. 692–702, July 1991.

[7] M. Lyu and Y. He, "Improving the N-version programming process through the evolution of a design paradigm," *Proc. IEEE Transactions Reliability*, vol. 42, no. 2, pp. 179–89, June 1993.

[8] S. Skiena and M. Revilla, *Programming Challenges*. Springer Verlag, Mar. 2003.

[9] D. Eckhardt and L. Lee, "A theoretical basis for the analysis of multi-version software subject to coincident errors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1511–17, Dec. 1985.

[10] B. Littlewood and D. Miller, "Conceptual modelling of coincident failures in multiversion software," *IEEE Transactions on Software Engineering*, vol. 15, no. 2, pp. 1596–614, Dec. 1989.

[11] M. van der Meulen, P. Bishop, and M. Revilla, "An exploration of software faults and failure behaviour in a large population of programs," in *Proceedings of the 15th IEEE International Symposium of Software Reliability Engineering*, St. Malo, France, Nov. 2004, pp. 101–12.

[12] M. van der Meulen and M. Revilla, "Experiences with the design of a run-time check," in *Proceedings of the 24th international conference on Computer Safety, Reliability and Security, Safecomp 2006*, ser. Lecture Notes in Computer Science, J. Gorski, Ed., vol. 4166. Gdansk, Poland: Springer, 2006, pp. 302–15.

Meine J.P. van der Meulen is consultant in safety and reliability engineering, currently working for Det Norske Veritas in Norway. Between 2003 and 2006, he was Research Fellow at the Centre for Software Reliability at City University (London) and combined this position with a position as Senior Consultant with Adelard (London). His research interests are in the reliability and safety of computer-based systems. He has been involved dependability studies since 1988 when he joined the safety instrumentation department of Dutch Railways. After that he worked for the Netherlands Organisation for Applied Scientific Research TNO, where he performed safety studies for mainly chemical industry and off-shore operations. Before joining CSR he worked for SIMTECH Engineering in Rotterdam (Netherlands), concentrating on safety studies in traffic (mainly railways, but also shipping) and infrastructure (mainly tunnels and storm surge barriers).

Meine van der Meulen holds an MSc degree in Electrotechnical Engineering from Eindhoven University, where he specialised in Measurement and Control. He also holds a degree in Dutch Law from the Dutch Open University. He is a member of the Dutch Society for Risk Analysis and Dependability, and the IEEE Reliability Society. He is member of the International Program Committee of the yearly Safecomp conference.

Miguel A. Revilla is a professor of applied mathematics and algorithms at the University of Valladolid, Spain. He is the official website archivist of the ACM ICPC and creator/maintainer of the primary robot judge and contest-hosting website. He received the 2005 Joseph S. DeBlasi Outstanding Contribution Award.

TABLE IV

EQUIVALENCE CLASSES AND FAULTS FOR THE FIRST SUBMISSIONS. EC: EQUIVALENCE CLASS; NUMBER OF OCCURRENCES OF THE EC IN THE VARIOUS PROGRAMMING LANGUAGES; REL.: FRACTION OF CORRECT RESPONSES TO THE 2,500 DEMANDS; DESCRIPTION: DESCRIPTION OF THE FAULTS FOUND IN THE EC, WITH THE CONSEQUENCES OF THE FAULT FOR ANOTHER PROGRAM STEP BETWEEN BRACKETS.

EC	C	C++	Pascal	Total	Rel.	Description
EC00	2483	2442	593	5518	100.00 %	Correct program.
EC01	1308	1456	350	3114	51.00 %	Swap: missing. (Calculation: results in 0 when $i > j$.)
EC02	272	314	54	640	51.00 %	Swap: incorrect. (Write: returns i and j in incorrect order when $i > j$.)
EC03	242	229	70	541	51.00 %	Swap: missing. (Calculation: leads to result 1 when $i > j$.)
EC04	95	96	40	231	58.00 %	Swap: missing. (Loop: only lowest number when $i > j$.)
EC05	62	73	10	145	51.00 %	Swap: missing. (Calculation: leads to result -1 when $i > j$.)
EC06	72	68	2	142	43.76 %	Loop: highest element not included. Swap: missing. (Calculation: results in 0 when $i > j$.)
EC07	44	40	22	106	50.92 %	Swap: missing. (Calculation: results in 0 when $i > j$.) Calculation: wrong for $n = 1$ (program step 3 after 5), leads to result 4.
EC08	33	46	21	100	0.00 %	Swap: missing. (Calculation: results in 0 when $i > j$.) Output: does not bounce inputs.
EC09	37	33	11	81	99.88 %	Calculation: wrong for $n = 1$ (program step 3 after 5), leads to result 4.
EC10	33	37	0	70	87.52 %	Loop: highest element not included, leads to result 0 when $i = j$.
EC11	22	28	5	55	0.00 %	Output: does not return inputs.
EC12	14	17	5	36	0.00 %	Swap: missing. (Calculation: results in 0 when $i > j$.) Calculation: all results one too low.
EC13	11	24	0	35	89.52 %	Loop: highest element not included, except when $i = j$.
EC14	16	14	2	32	40.24 %	Loop: lowest and highest number not included. Swap: missing. (Calculation: results in 0 when $i > j$).
EC15	12	16	3	31	51.00 %	No output line when $i < j$.
EC16	9	18	2	29	50.96 %	Calculation: aborts when $n = 1$, leads to result 0. Swap: missing. (Calculation: leads to result 0 when $i > j$.)
EC17	21	7	1	29	0.00 %	Output: prints the sequence for the first input.
EC18	13	12	1	26	54.96 %	Swap: incorrect, leads to $i = j = \max(i, j)$ when $i < j$.
EC19	9	13	4	26	50.92 %	Swap: missing. (Calculation: results in 1 when $i > j$.) Calculation: wrong for $n = 1$ (program step 3 after 5), leads to result 4.
EC20	9	12	3	24	50.92 %	Swap: incorrect. (Write: returns i and j in incorrect order when $i > j$.) Calculation: wrong for $n = 1$ (program step 3 after 5), leads to result 4.
EC21	12	9	2	23	0.00 %	Swap: missing. (Calculation: leads to result 0 when $i > j$.) Calculation: all results one too low.
EC22	9	12	2	23	99.96 %	Calculation: aborts when $n = 1$, leads to result 0.
EC23	12	10	1	23	22.64 %	Loop: only calculates sequence length for the two inputs. (Output: highest of the two sequence lengths.)
EC24	12	11	0	23	43.80 %	Loop: highest element not included when $i < j$, lowest element not included when $i > j$.
EC25	9	9	3	21	0.00 %	Output: only outputs result.
EC26	13	8	0	21	43.76 %	Loop: highest element not included. Swap: incorrect. (Write: returns i and j in incorrect order when $i > j$.)
EC27	10	8	0	18	0.00 %	Calculation: all outputs are equal to 1108544020, overflow, because of error in recursion.
EC28	5	9	3	17	48.32 %	Swap: missing. (Calculation: results in 0 when $i > j$.) Calculation: incorrect, leads to result being one too low if maximum cycle-length of longest sequence is one higher than the next highest length.
EC29	5	9	1	15	0.00 %	Output: all results are 0.
EC30	7	7	1	15	46.00 %	Swap: missing. (Calculation: results in 0 when $i > j$.) Loop: lowest element not included.
EC31	7	8	0	15	0.00 %	Initialisation: variable not initialised.
EC32	5	10	0	15	87.56 %	Loop: highest element not included.
EC33	4	10	0	14	0.04 %	Calculation: always leads to result 1.
EC34	7	6	1	14	51.00 %	Swap: incorrect, leads to $i = j = min(i, j)$.
EC35	3	11	0	14	99.96 %	Calculation: wrong for $n = 1$ (increment of cycle-length incorrect for $n = 1$), leads to result 2.
EC36	6	7	0	13	52.76 %	Loop: highest element not included. Loop: only first element when $i > j$.
Total	5897	6097	1581	13575		