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 their 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;*
- *characterise how programs are debugged during development;*
- *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 of 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 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 solutions 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
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 integral 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.

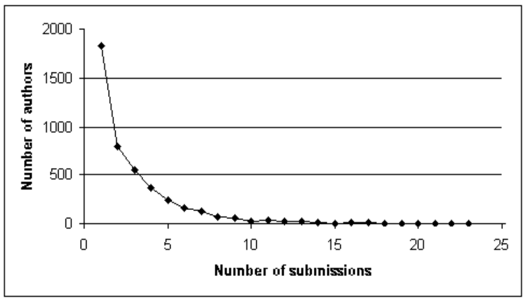
```
1 10 20
100 200 125
```

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 sequence length calculation and the determination the maximum value. The programs that dif-

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 sequence 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 sequence length
`      if (cycle > cyclemax) cyclemax=cycle;`	Determine maximum
`    }`	
`    printf ("%d %d %d\n", a, b, cyclemax);`	Write outputs
`  }`	
`}`	

fer 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' execution are written to a file for later analysis. We deleted all output 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.

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 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, 400 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 (correct: 3 305, 27.7%; wrong answer: 8 510, 71.2%; presentation error: 136, 1.1%). Three main analyses were performed in this exploratory study:

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

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. We only considered 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, their total frequencies, their frequencies of first and last occurrence, together with 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 sequence length. The author forgets to reset the maximum sequence length for the next set of input values (3.7%), In this case the program will fail if the maximum sequence 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 element 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++)
```

Write. 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 sequence length. This is probably due to the fact that if the algorithm responds well to the sample outputs the algorithm 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).

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.

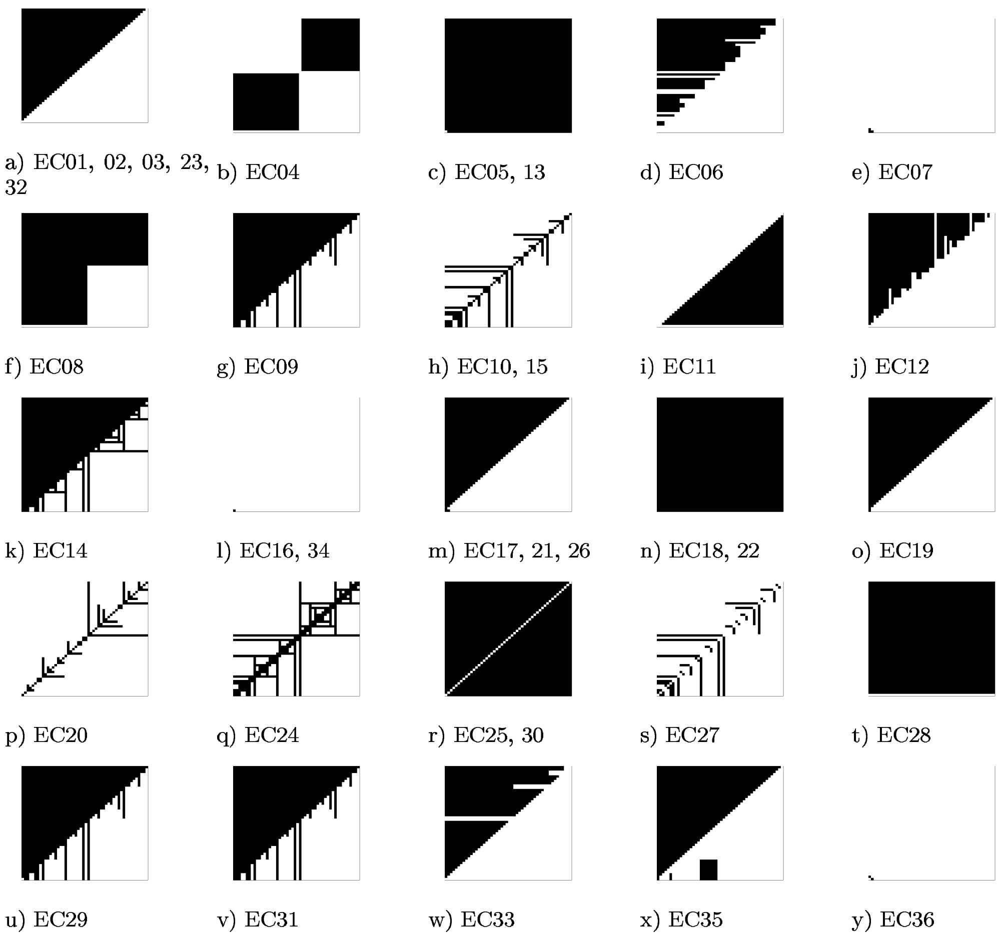
The triangular pattern, e.g. a), i) and o), is related to the i, j swap problem, i.e. the correct answer is only generated when i is less or equal to j, but it can be seen that there are several versions with identical shapes. The difference is due to the different default values used by the versions when there are zero iterations. 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 was a completely blank square, which represent the case where all test inputs were correct.

Table 2: Equivalence classes and faults. Between brackets: consequences of the fault for another program step.

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 sequence length: not included after first calculation. Swap: missing. (Calculation: results in maximum sequence length of all previous calculations).
EC05	277	63	9	0.04 %	Output: 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 $n = 1$, leads to result 4.
EC08	74	12	2	26.96 %	Reset maximum sequence 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 $n = 1$, 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 %	Print: 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 %	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 sequence 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 sequence length of longest sequence is one higher than the next highest length.
EC36	5	2	1	99.92 %	Calculation: wrong for $(i, j) = (0, 1)$ or $(i, j) = (1, 0)$ (program step 3 after 5), leads to result 4.
Total	8148	2960	3828		

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 so far (and even more combinations are possible if we include "varieties" of the basic faults).

Figure 2: Failure sets for common equivalence classes.



7 Analysis of the debugging process

We have seen that there are a number of basic 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 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 ar "reachable" from another class (i.e., only the ones with one more or 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, and the basic faults contained in each equivalence class.

An "ideal" debugging process would eliminate a basic fault at each step, but in practice it can be seen that there are more transitions than the number of faults by a factor of 2 or 3. 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 that authors do not insert faults of an entirely different category. e.g. there are no transitions from EC07 to EC06 where the faults are disjoint. It is also rare to have a transition from a program containing one fault to a program with two faults. When analysing the few cases where this happens, it appears to be that the failure set of the first fault "covers" the failure sets of two other faults.

The diagonal is a dominant feature of the transition table. These are transitions within the same equivalence class.

Table 3: Probability of staying in the same equivalence class for a program in a given equivalence class and mean number of steps to final submission. (#Tr.: Number of transitions.)

EC	**#Tr. to same EC**	**Total #Tr. from EC**	**% within EC**	**Mean #Steps to correct**
EC01	702	1602	44 %	2.9
EC02	324	854	38 %	2.0
EC03	174	389	45 %	2.8
EC04	96	278	35 %	2.6
EC05	117	265	44 %	3.5
EC06	80	182	44 %	2.6
EC07	32	73	44 %	2.0
EC08	28	72	39 %	3.5
EC09	14	60	23 %	4.2
EC10	24	62	39 %	2.4
EC11	20	38	53 %	2.4
EC12	19	36	53 %	2.5
EC13	22	33	67 %	6.0
EC14	4	35	11 %	3.6
EC15	21	35	60 %	4.0
EC16	11	32	34 %	1.6
EC17	12	31	39 %	4.1
EC18	1	24	4 %	4.2
EC19	6	23	26 %	3.2
EC20	6	20	30 %	2.0
EC21	11	19	58 %	3.0
EC22	11	12	92 %	1.2
EC23	6	17	35 %	4.3
EC24	5	17	29 %	2.4
EC25	4	15	27 %	1.6
EC26	3	13	23 %	1.9
EC27	1	1	100 %	0.1
EC28	5	14	36 %	2.1
EC29	1	14	7 %	4.6
EC30	2	12	17 %	1.9
EC31	3	11	27 %	5.0
EC32	2	9	22 %	1.2
EC33	3	7	43 %	1.0
EC34	2	7	29 %	1.1
EC35	1	6	17 %	1.8
EC36	2	4	50 %	1.2

7.2 Reliability of successive releases

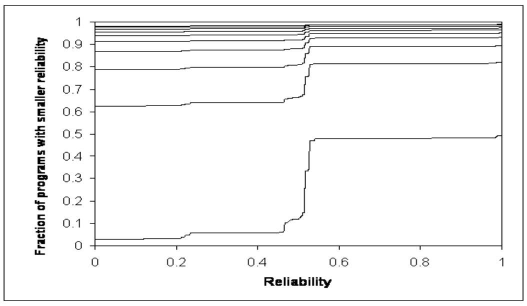
It is difficult to talk about the reliability of a program version without defining its operational profile. Take for example, the program failure set in Figure 5a). If the input profile that was restricted to the top triangular portion the program would always fail, while an input profile that remained 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 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,

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

The Eckhardt and Lee model assumes similar development processes for 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:

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

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.:

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

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

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

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 the value implied by the independence assumption)

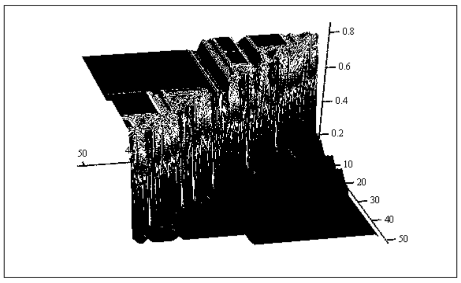
If the populations A and B differ (the Littlewood and Miller model), the improvement can, in principle, be *better* that the independence assumption, i.e. when the "dips" in $\theta_A(x)$ coincide with the "spikes" in $\theta_B(x)$, it is possible for the expected value of pfd_2

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 below.

Figure 4: Difficulty function for 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 release	Final release
pfd_1	0.32	0.15
pfd_2	0.024	0.026
pfd_1^2 (independent)	0.010	0.023
Ratio	2.4	1.13

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 are shown in Figure 4.

The equivalent pfd results for the second problem were:

Figure 5: Difficulty function for the alternate problem.

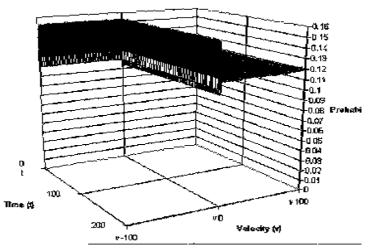


Table 5: Expected pfd's, from [2].

Parameter	Initial release	Final release
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 official 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 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 is likely work for all inputs.

Most of the implementation faults were related to well known programming "slips", e.g.

- first element of loop forgotten;
- last element of loop forgotten;
- first and last element 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, inappropriate use of functions, etc.

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 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 are 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 of 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 sub-domain night 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 Valladolid Programming Contest 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 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" (in preparation).

[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), pp.1511-1517, 1985.

[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), pp. 96-109, 1986.

[5] 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.

[6] B. Littlewood, "Software reliability growth models", Software Reliability Handbook (P. Rook, ed.), Elsevier (Amsterdam), pp. 401-412 (Ch.16), 1990.

[7] M.R. Lyu, "Software Reliability Engineering", McGraw Hill, 1995.

[8] S. Skiena and M. Revilla, Programming Challenges, ISBN: 0387001638, Springer Verlag, March, 2003, (http://online-judge.uva.es/).

Table 6. Transitions from equivalence classes EC01 to EC32 to equivalence classes EC00 to EC32.

	To EC:																																			
	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