Predicting the total number of faults using parallel code inspections

Les Hatton
CIS, University of Kingston*

May 25, 2005

Abstract

A simple relationship between the total number of faults in a piece of code and the faults found in common by inspection teams is derived making an assumption of independence in the faults found by different teams. This relationship is then tested in four separate experiments using 22 2-person teams from India, Germany and Austria on the same piece of code in which the total number of faults is known but not revealed to the inspectors. The results show that the relationship holds up well and that somewhat surprisingly given the use of checklists that the assumption of independence is still a statistically reasonable one. This then provides a practical way of predicting the total number of faults in code in general with reasonable accuracy even when a relatively small number of two person teams are used.

$Date: 2005/05/25 16:12:07 $

1 Overview

In an ideal world, testing would simply stop when all the faults that can fail (defined to be *defects* here) had been found and corrected. This is generally impractical given commercial exigencies so, in all engineering systems, there is a risk of failure in the delivered product and engineers traditionally deal with this using the concept of "good enough" or with critical systems, ALARP, (As Low As is Reasonably Practical). With conventional systems such as those in civil, mechanical and aeronautical engineering, historical experience is often an excellent guide to the future behaviour of an engineering system based on its past behaviour. This experience is assembled by a continual feedback process based on the analysis of failure as shown 1. Over a period of perhaps many years as the engineering discipline matures, failing systems are diagnosed to determine why they failed in order to prevent future failures of the same nature. This simultaneously strengthens the ability to predict the future failure nature of a system.

*L.Hatton@kingston.ac.uk, lesh@oakcomp.co.uk

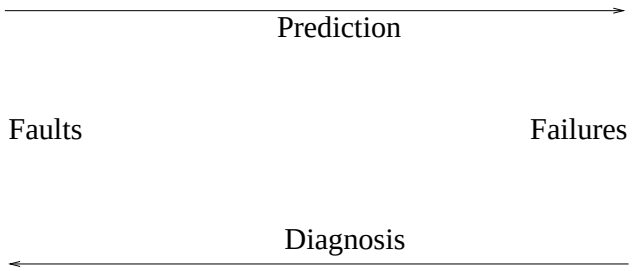

Figure 1: The relationship between diagnosis and prediction in a mature engineering discipline

In other words, in conventional engineering systems, there is not only a mechanism for diagnosis but there is also a mechanism for prediction. In software-controlled systems, historical experience is usually not available, not documented, or is irrelevant because the build techniques have changed too much and no such prediction mechanism exists, primarily because diagnosis is so poor, [8]. This has many negative implications but this paper will focus on one, viz. the lack of a good objective measure of when testing should stop. Since testing is synonymous with finding defects, a minimum requirement would be to be able to predict the number of remaining defects to some degree of precision.

Cost is also a significant factor. The earlier a prediction can be made about future behaviour, the better. If predictions can only be made from existing runtime (i.e. live) behaviour, the system must be built first and defects can then be very expensive to remove. If on the other hand, predictions can be made from say a design, the cost savings can be considerable as remedial work can be done before the system becomes too expensive to change.

There are essentially two forms of testing:-

- Static testing. This covers all test techniques which attempt to discover faults by simply examining a design or piece of code. No attempt is made to run the system. The archetypal static test technique is the design or code inspection, [4]. Static test techniques find *faults*.
- Dynamic testing. This covers all those test techniques which observe a running system and attempt to demonstrate inputs which lead to unexpected outputs. Dynamic test techniques find *failures*. All failures are caused by at least one fault but not all faults fail. As mentioned above, a *defect* is a fault that has failed.

The novel approach described here is to use a simple relation based on independence to predict the total remaining *faults* rather than the total remaining *failures*. Given that the best practitioners of inspection are known to find more than 95% of all faults which lead to eventual failure, [5], the distinction in this case would be unimportant. The number of experiments described here is also sufficient to test the central assumption of independence.

In software-controlled systems, test halting criteria can usually be subdivided into two obvious types:

- Subjective criteria These include but are not limited to:
  - *When the testers 'feel' the application is OK.* This may be quite effective but depends to a large extent on the experience of the testers.
  - *When the budget runs out.* A perfectly reasonable criterion to use provided there is some systematic way of calculating the budget based on risk. This however is rarely the case.
  - *When the testers patience runs out.* This speaks for itself and is not meant to be humorous - the author has seen this on a number of occasions.
- Objective criteria These would be considered a far more satisfactory way of defining when testing of an application should halt. They might include:
  - *When a defined set of tests is completed.* Although objective, it does of course depend on how well the defined set of tests 'cover' the delivered functionality.
  - *When a particular reliability in terms of mean time between failures is reached.* Arguably the most satisfactory from the end-user's point of view in that it is strongly related to the way users might view the quality of the delivered application. For an example, see [3].
  - *When a defined coverage is achieved in some sense.* There are various forms of test coverage from simple statement coverage where some specified percentage of all possible executable statements have to be executed during testing, all the way to path testing, which can be infeasible for all but the simplest programs. There seems little published data on the relationship between coverage and reliability, so to a certain extent, this is also an act of faith.
  - *When more than a certain percentage of all defects have been found.* If the total number of remaining defects was known, this would perhaps be the most satisfactory and certainly obvious objective measure of test quality although of course its relationship with reliability is again opaque. This measure will be the subject of this paper.

1.1 Predicting total faults from common faults

Let P(A) be the probability of a fault being detected by person A in some product. Similarly let P(B) be the probability of a fault being detected by person B in the same product. Then with the single assumption that A and B are independent,

$$P(A \cap B) = P(A)P(B) \tag{1}$$

Further suppose that there are 'n' faults altogether, that person A finds 'a' faults, person B finds 'b' faults and that they find 'q' faults in common. Then, using the above equation,

$$\frac{q}{n} = \frac{a}{n}\frac{b}{n} \tag{2}$$

From this, re-arranging yields:

$$n = \frac{ab}{q} \tag{3}$$

Provided, q $\neq$ 0, this provides a simple way of calculating the total number of faults using data on the number of faults found in common between two inspection groups.

A cautionary note One particular scenario appears to undermine the method and this is when an experienced inspector is teamed with an inexperienced inspector so that the inexperienced inspector's faults are a non-disjoint subset of the experienced inspector's faults, for example b = q. In this case the predicted number of faults simply becomes 'a', the total found by the experienced inspector. The inexperienced inspector adds no relevant information to the prediction and the performance of the team depends entirely on the performance of its experienced member. An example can be seen in one of the datasets from Germany shown below.

1.2 Methodology

Code inspections The benefit of code inspections has been re-iterated many times since the seminal work of [4]. It is almost certainly true to say that inspections are the most successful technology for the removal of defect ever discovered. The reader is referred to [5] and [9] for more details and to [11] for a review of some of the inspection approaches which have been proposed.

Inspections have another important property. Because there is no knowledge of run-time when an inspection is carried out, inspections attack the entire fault space irrespective of any temporal properties. In contrast, dynamic testing only attacks that subset which can be provoked to fail in a given run-time. Figure 2 illustrates. As time goes by, the subset of faults which have actually failed gradually grows within the set of faults which could fail. It may never fill the entire fault space for various reasons. As was demonstrated so emphatically by [1], a significant percentage of defects, around a third in his case take at least 5,000 execution years to fail for the first time so such defects would be effectively impossible to reveal with dynamic testing. For such defects, static techniques such as code inspections remain the only option. Of course, some systems may never see a total of 5,000 execution years but for a reasonably ubiquitous embedded control system for example, such a total can be exceeded collectively within a few weeks or even days. Another source of faults which may never fail during the life-cycle of a program is the set of faults which cannot be executed because there is no possible execution path which can provoke their failure. In some complex systems, significant parts can be effectively unreachable so this may not be small.

Finally, inspections also have the benefit of taking place before run-time testing starts. It is very widely known that defects found no later than this phase are very substantially cheaper than finding the same defects during run-time testing, (unit, system, acceptance ...), (e.g. [2]).

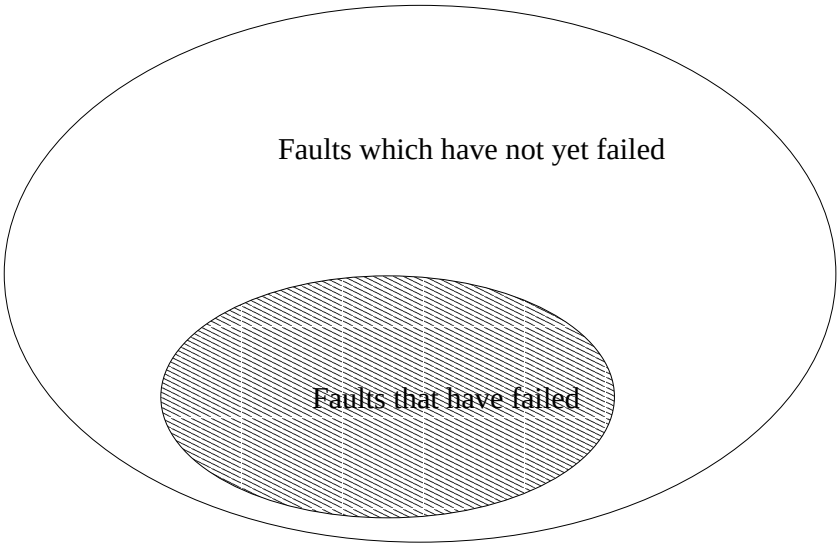

Figure 2: The relationship between faults and defects. Dynamic testing by definition can only attack the faults that can be provoked to fail in the available time shown by the shaded area. In contrast, inspections operate on the entire fault space represented by the larger unshaded area.

1.3 The experiments

Some time was taken to select the right program for the experiment. The principle constraints were engineering time and also the ability to calibrate the experiment in the sense of knowing actually how many faults were in fact present. In addition, it would need enough faults in it to give reasonable results and it would be preferable to have originated from a real system. The chosen program was a C program of 62 lines. It was then slightly modified to remove all identification, (although all faults described below were unaltered by this process). Although small, this program fitted all the criteria well although it of course suffers the disadvantage that it is not known how well the experiment would scale to larger programs and this must be left for further study. Be that as it may, the experiment led to statistically useful conclusions as will be seen.

The program was then analysed both automatically and manually for code fragments known to fail as referenced in [7] and [6]. This allowed the total number of faults to be specified quite precisely as 26 $\pm$2. The error bound although subjective, was added to mirror the fact that some statically detectable failure prone fragments in this program may have noise associated with them in the manner described by [6]. The presence of so many faults in a real system, (actually a high-integrity system), is surprising. Almost all programs analysed in [7] showed the presence of such faults but at a much lower rate of around 1 every 120 lines or so. This program is atypically very poor but nevertheless was part of a real system which is cause for concern on its own.

Examples of some of the fault modes and occurrence rates are shown below. The remainder had some degree of noise associated with them, for example the recommendation that if and else branches be brace-enclosed.

Fault mode	occurrences
Returning the address of a local from a function	1
Test of unsigned variable for negativity	1
Implicit conversion between int and unsigned int or long	3
Unary negative applied to unsigned object	1
Pointer cast to potentially stricter alignment	1
malloc() called in absence of a prototype	1
Non re-entrancy created by unnecessary internal use of static	1
Unconditionally uninitialised variable	1
enum value missing from switch on enum	1
Non-empty case falls directly through to next case	1
Re-initialisation expression of for loop of floating type	1
Comparison of floating types for equality	1
Memory leak	1
Conditionally uninitialised variable	6

It should be noted that these faults fall into the general category of inconsistencies in the use of language. This made it easier to specify the exact number of faults and therefore control the experiment better. They also stress any assumption of independence because they are often the subject of check-lists in real inspections and check-lists might reasonably be expected to degrade independence.

Groups of industrial embedded system engineers in three separate countries, from Germany (4x2 person teams in one location and 4x2 person teams in another), Austria (3x2 person teams) and from India (11x2 person teams) over a 2 year period, were allowed to inspect the described program individually for a total of 30 minutes, corresponding to an inspection speed of approximately 120 lines of code per hour. This lies within the most efficient range quoted by [9], although is a little fast according to [5]. Following this, individual engineers were then combined in teams of two to compare notes for a further 15 minutes to determine which faults they had independently found in common. The results were then collected and are shown below. One additional 'team' was added in the form of two independent compilers, the MS Visual C++ 6.0 compiler and the GNU compiler v. 2.95 both running on Windows.

1.4 Results

1.4.1 Data from India

The raw data is shown below as a series of tables.

faults found by Inspector 1	faults found by Inspector 2	faults in common	Predicted faults	Team type
7	9	3	21	Human
12	13	3	52	Human
14	12	4	42	Human
10	7	4	17.5	Human
10	11	6	18.33	Human
14	10	5	28	Human
8	9	3	24	Human
6	6	3	12	Human
9	6	3	18	Human
15	10	4	37.5	Human
10	13	3	43.33	Human

Average predicted faults = 26.89

1.4.2 Data from Austria

faults found by Inspector 1	faults found by Inspector 2	faults in common	Predicted faults	Team type
15	17	11	23.18	Human
25	20	17	29.41	Human
19	15	9	31.67	Human

Average predicted faults = 28.09

1.4.3 Data from Germany - 1

faults found by Inspector 1	faults found by Inspector 2	faults in common	Predicted faults	Team type
14	13	8	22.75	Human
11	18	9	22	Human
5	17	5	17	Human
17	9	6	25.5	Human

Average predicted faults = 21.81

1.4.4 Data from Germany - 2

faults found by Inspector 1	faults found by Inspector 2	faults in common	Predicted faults	Team type
13	8	2	52	Human
15	10	6	25	Human
17	8	5	27.2	Human
17	10	6	28.33	Human

Average predicted faults = 33.13

1.4.5 General Data

faults found by Inspector 1	faults found by Inspector 2	faults in common	Predicted faults	Team type
6	3	2	9	Two compilers

The overall average is 26.47 which is gratifyingly close to the accepted answer although the assumptions behind this prediction will now be tested formally.

Finally, it should be noted that individual performance was very similar across the four experiments.

2 Testing the assumption of independence

Two events A and B, are independent if and only if $P(A \cap B) = P(A)P(B)$. To test the independence assumption, the statistic $P(A \cap B) - P(A)P(B)$ was tested using the Kolmogorov-Smirnov test for normality, [12]. The result is that this statistic is consistent with a normal distribution $N(\mu = 0.0183, \sigma = 0.07433)$ with a power p = 0.82. The equivalent 95% confidence interval for the mean is (-0.0084,0.0484) which includes 0.0 so we cannot reject the hypothesis that $P(A \cap B) - P(A)P(B)$ is normally distributed with a mean of zero, and so cannot therefore reject the hypothesis of independence at this level of confidence.

Discussion of non-independence This emphatic result is a little surprising as whether or not the assumption of independence made above is appropriate was an exceedingly moot point at the beginning of the experiments because of an anticipated trade-off between being able to predict the total number of faults (apparently depending on the independence assumption) and being able to find the most faults, (which conventional wisdom has it depends on checklists). The trade-off occurs of course because widespread use of checklists would be expected to guarantee that the independence assumption would be questionable at best because inspectors are individually driven towards well-known fault modes. Furthermore, some of these fault modes were present in the inspected program. Some attempt was made to cater for this in the experiments by allowing some groups to use standard checklists as control groups and others not, however, there was no significant difference in the data between the two. The simplest explanation of this seems to be that reasonably experienced engineers appear to use personalised 'internal' methods whether given checklists explicitly or not. In neither case was it possible to reject the independence assumption.

The predicted distribution The actual distribution of the predicted number of faults was also independently tested using the same test. The result was that two *outliers* were identified (i.e using Tukey's exploratory data definition of greater than 1.5 times the interquartile range either above the third quartile or below the first.) When these were removed, the resulting distribution was

reported as normal with a mean of 25.47, a standard deviation of 10.0 and a Power of 0.65.

3 Conclusions

A simple method for predicting the total number of faults from those found by parallel inspection teams has been presented and its underlying assumption of independence tested statistically. The method gave reasonably good predictions but the analysis of independence gave the somewhat surprising result that independence was not rejected by this data and in particular, *there was no evidence that the use of checklists in any way undermined the independence of the inspections*. It should be re-iterated that the method is calibrated on a software component of only 62 lines. The disadvantage of this is that it is unknown how well this method might scale to a larger experiment although this might reasonably be expected to stabilise the results further. The advantage of using a small experiment is that it was much easier to control and cheap enough to do without impacting valuable engineering time.

Given that it is often difficult to predict how effective an inspection process has been, this is a welcome addition to existing techniques and suggests that by diverting some of the resources of a code inspection to parallel inspection, a separate estimate for the total remaining faults can be made over and above any historical data a company might have on the effectiveness of its inspections. If inspections are already effective in finding many of the faults that failed, this would automatically provide a reasonable estimate of the number of faults that might fail and moreover provide them relatively early on in the life-cycle before expensive dynamic testing starts.

As a final note, it is interesting to note how poor the compilers were at finding the kind of faults appearing in the inspected program. This has traditionally been the case with compiler writers largely confining themselves to efficiency concerns rather than inaccurate use of language. Individually the compilers behaved like very inexperienced code inspectors. Modern static analysis tools such as [10] tend to fare much better and should find all the faults noted here leaving inspectors to focus on faults which only humans can find and for which calibration experiments are rather harder to design.

References

[1] E. Adams. Optimising preventive service of software products. *IBM Journal of Research and Development*, (28):2–14, 1984.

[2] B. Boehm. *Software Engineering Economics*. Prentice-Hall, NJ, 1981.

[3] R. Brettschneider. Is your software ready for release ? *IEEE Software*, 6, July 1989.

[4] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 2:182–211, 1976.

[5] T. Gilb and D. Graham. *Software Inspections.* Addison-Wesley, 1993. ISBN 0-201-63181-4.

[6] L. Hatton. EC– a measurement based safer subset of iso c suitable for embedded system development. *Information and Software Technology*, 47:181–187.

[7] L. Hatton. *Safer C: Developing software in high-integrity and safety-critical systems.* McGraw-Hill, 1995. ISBN 0-07-707640-0.

[8] L. Hatton. Characterising the diagnosis of software failure. *IEEE Software*, July 2001.

[9] W. Humphrey. *A discipline of software engineering.* Addison-Wesley, 1995. ISBN 0-201-54610-8.

[10] Oakwood Computing Associates Ltd. The Safer C toolset, 2005. www.oakcomp.co.uk.

[11] S.L. Pfleeger, L. Hatton, and C. Howell. *Solid Software.* Prentice-Hall, 2002. ISBN 0-13-091298-0.

[12] M.R. Spiegel and L.J. Stephens. *Statistics.* Schaum. McGraw-Hill, 3rd edition, 1999.