Open access • Journal Article • DOI:10.1109/99.609829

The T experiments: errors in scientific software — Source link

Les Hatton

Published on: 01 Jan 1997 - Computational Science and Engineering

Topics: Social software engineering, Software construction, Software measurement, Software sizing and Software requirements

THE T-EXPERIMENTS: ERRORS IN SCIENTIFIC SOFTWARE

L. HATTON
Oakwood Computing,
Oakwood House, 11, Carlton Road, New Malden, Surrey, KT3 3AJ
Tel/Fax: +44-181-336-1151
lesh@oakcomp.demon.co.uk, http:/www.oakcomp.demon.co.uk/

ABSTRACT

This paper covers two very large experiments carried out concurrently between 1990 and 1994, together known as the T-experiments. Experiment T1 had the objective of measuring the consistency of several million lines of scientific software written in C and Fortran 77 by *static* deep-flow analysis across many different industries and application areas, and experiment T2 had the objective of measuring the level of *dynamic* disagreement between independent implementations of the same algorithms acting on the same input data with the same parameters in just one of these industrial application areas.

Experiment T1 showed that C and Fortran are riddled with statically detectable inconsistencies independent of the application area. For example, interface inconsistencies occur at the rate of one in every 7 interfaces on average in Fortran, and one in every 37 interfaces in C. They also show that Fortran components are typically 2.5 times bigger than C components, and that roughly 30% of the Fortran population and 10% of the C population would be deemed untestable by any standards.

Experiment T2 was even more disturbing. Whereas scientists like to think that their results are accurate to the precision of the arithmetic used, in this study, the degree of agreement gradually degenerated from 6 significant figures to 1 significant figure during the computation. The reasons for this disagreement are laid squarely at the door of software failure, as other possible causes are considered and rejected.

Version 2nd Apr 97, for CS & E

Taken with other evidence, these two experiments suggest that the results of scientific calculations involving significant amounts of software should be treated with the same measure of disbelief as an unconfirmed physical experiment.

1. INTRODUCTION

The results of the two experiments described in this paper should intrigue and perhaps perturb any scientific user of software interested in the accuracy of their results. They are based on pure measurement rather than speculation or anecdote and contain no abstruse theory whatsoever. Together they paint a rather gloomy picture, suggesting that the accuracy we actually get in scientific computation is rather less than we would like or expect, with all the attendant risk to the development of our theories. However, detection is half the battle, and enough insight emerged to suggest ways forward, although the path is not easy. Note that the experiments described here cover only serial code and not parallel code, although I believe (without supporting data) that the situation is no better for parallel systems, particularly of course when they simply lay out serial computation across an array or matrix of processors.

When we test our scientific software, we frequently find errors. We correct them until there comes a great day, when our precious scientific thought is encapsulated in a piece of robust and above all, absolutely error-free software :-). Then some time later, we find another error, and a little time after that, another, and so on. So how good was it all in the first place ? There are two ways of assessing this. First of all, many software failures, perhaps as many as 40%, (Hatton 1995), are *statically* detectable as faults. In other words, they can be found without running the program first. In this sense, a fault is defined to be a misuse of the language which will very likely fail in some context. In Fortran, the scientist will be familiar with dependence on uninitialised variables, inconsistent interfaces and so on. In C, the delights of pointers add many new ways of getting it all wrong. In C++, even more sybaritic delights await the unwary, with the language rapidly becoming so complex that any underlying science seems almost irrelevant amidst the magic kingdom of polymorphism, inheritance, object-orientation, over-loading, virtual methods, encapsulation and vast numbers of hidden and frequently surprising actions performed on behalf of the unwitting scientist by a grateful compiler. In contrast, predicting the existence of a new sub-atomic particle seems a relatively straightforward exercise.

The second way of assessing the effects of software failure is to run it and see giving a *dynamic* perspective. Unfortunately, scientists tend to swap code rather than relying on the independent verification they pursue naturally for an experimental result, thus reducing the opportunities for carrying out such an experiment. Fortunately commercial rivalry comes to the rescue in the form of seismic data processing, where scientists have to develop their software in conditions of strict competitive confidentiality. This has naturally evolved parallel independent implementations of the same algorithms. Even more fortunately, significant parts of this

software have been developed from identical published mathematical signal-processing specifications greatly reducing a significant source of uncontrollable variability.

In order to compare these two views of software, two concurrent experiments to measure their effects were conducted in the period 1989-1994, although I never realised how long they would take to do enough to get a good picture. What started as an interesting hobby messing around with other people's code finished up as two 4 year long experiments, which became known as the T experiments. The relationship between the two T-studies is shown in Figure 1.

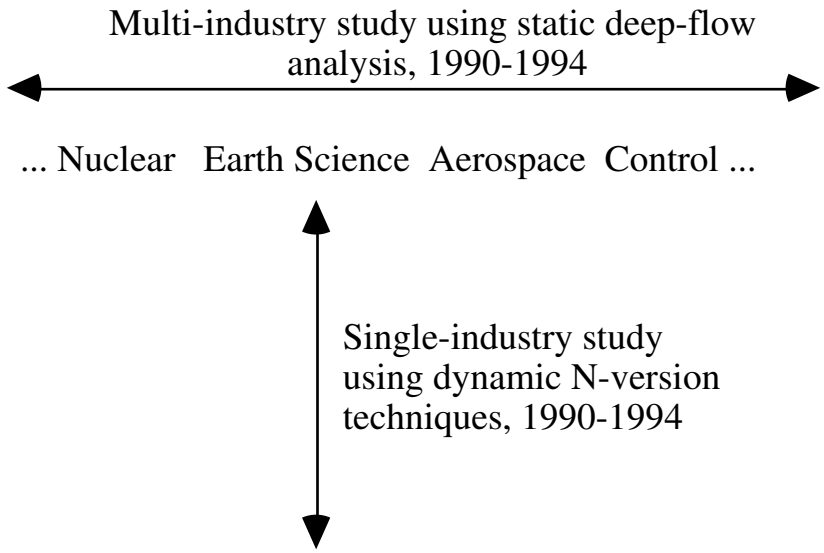

Figure 1:T-diagram illustrating the relationship between experiment T1, which concerns static measurements along the cross-bar of the T and experiment T2, which concerns the vertical bar of the T, covering dynamic measurements.

2. EXPERIMENT T1: STATIC ANALYSIS RESULTS

Experiment T1 focused on static measurements of software whereby the source code is automatically checked for consistency but not run. This corresponds to the cross-bar of the T. The codes studied were submitted for analysis mostly by companies, but also by government agencies and universities from around the world and cover some 40 application areas including for example graphics, nuclear engineering, mechanical engineering, chemical engineering, civil engineering, communications, database, medical systems and aerospace. Both safety-critical and non safety-critical environments as well as environments with and without quality systems were comprehensively represented. The age of the codes is evenly spread between 1 and 20 years old. All codes submitted are "mature" in the sense that they are in regular use by their intended users. The details of the populations as of 31st July, 1995 were:

Fortran 66/77

- Total lines analysed	3,305,628
- Number of participating organisations	47
- Largest package in lines	770,444
- Smallest package in lines	806
- Average package size in lines	60,102
- Total packages	55
- Number of different disciplines	20
- Total executable lines	1,737,536

C

- Total pre-processed lines analysed	1,928,011
- Number of participating organisations	26
- Largest package in lines	431,655
- Smallest package in lines	361
- Average package size in lines	28,353
- Total packages	68
- Number of different disciplines	41
- Total executable lines	1,389,712

So what exactly are we looking for ? In static deep-flow analysis, we are looking for inconsistent or undefined use of language. A deep-flow analysis tool studies code rather like a compiler but its back-end is a knowledge base of items known to lead to failure rather than an object-code generator. It might surprise the scientist, who relies on a tool which is thousands of years old and supported by rigorous and proven methods, i.e. mathematics, that the programming language he or she uses to express the science numerically is considerably less well-defined. It is therefore a common fallacy, that if something compiles, it is OK, apart from errors of the mind. In this study, we carried out three forms of analysis:

- Unsafe dependency analysis by measurement of dependence on unsafe features of the programming language. Such features fall into two categories: those explicitly defined as unsafe by the standard itself, (e.g. ISO C lists 119 constructions of uncertain definition), and a rather larger number of features which have been found by experience to be unsafe or to lead to unsafe behaviour, even though apparently well-defined. Some items are language-independent at least in part and occur in both C and Fortran, for example, any dependence on uninitialised variables. Other items are language-dependent such as the well-known problem in C of casting a pointer to a narrower integral type. The occurrence rate of several hundred items was measured in both C and Fortran.
- Programming standards adherence. This is unlikely to be of interest to readers of this paper and will not be considered further beyond noting that experiment T1 proved conclusively that attempts to maintain programming standards were risible.
- Population complexity analysis using a range of well-documented software measures.

All of the measures are based on the source code itself rather than the design and such measures are therefore entirely repeatable.

The submitted codes were analysed by two static deep-flow analysers, *QA C* for C source code and *QA Fortran* for Fortran 66/77 source code. The biggest C package analysed in one pass was Motif™ 1.1.4 and X11R5 together during an interface consistency check, totalling some 700,000 source lines, (which revealed a total of 1,885 interface faults). The parser of the C analysis tool, *QA C*, is based on the model C implementation, which was amongst the first to be validated by the British Standards Institute. The biggest Fortran package analysed in one pass, a seismic data processing package, totalled some 770,444 source lines.

Although only C and Fortran were studied here, users of other languages should not assume that they are therefore free of any of the problems reported here. All languages have similar problems of one kind or another, (Ghezzi and Jazayeri 1982), (Hatton 1995) with politically- and accidentally-induced ambiguity and redundancy both rife.

It should be re-emphasised that the measurements are based on huge amounts of source code in day to day use which its developers believe to constitute fully-tested products.

Resulting static fault rates

Precise occurrence rates in occurrences per lines of code were published in part in (Hatton 1995), but brevity forbids us from requoting these results here. Instead, each statically detectable unsafe item in a list containing around 100 such items was categorised as to severity between 5 and 100% where 5% represents a relatively low probability that such a fault will mature into a failure, (for example casting a pointer to an integral type in C) and 100% represents effective certainty that this will take place in a reasonable software life-cycle, (for example, unconditional reliance on an uninitialised variable). Of course there is no guarantee that any particular fault will mature into a failure but independent observers were in close agreement as to the severity category for each item, and the weighting is intended only to give some kind of risk factor.

For C, the results are shown as a function of industrial application area in Figure 2, whilst equivalent figures for Fortran 77 are shown in Figure 3.

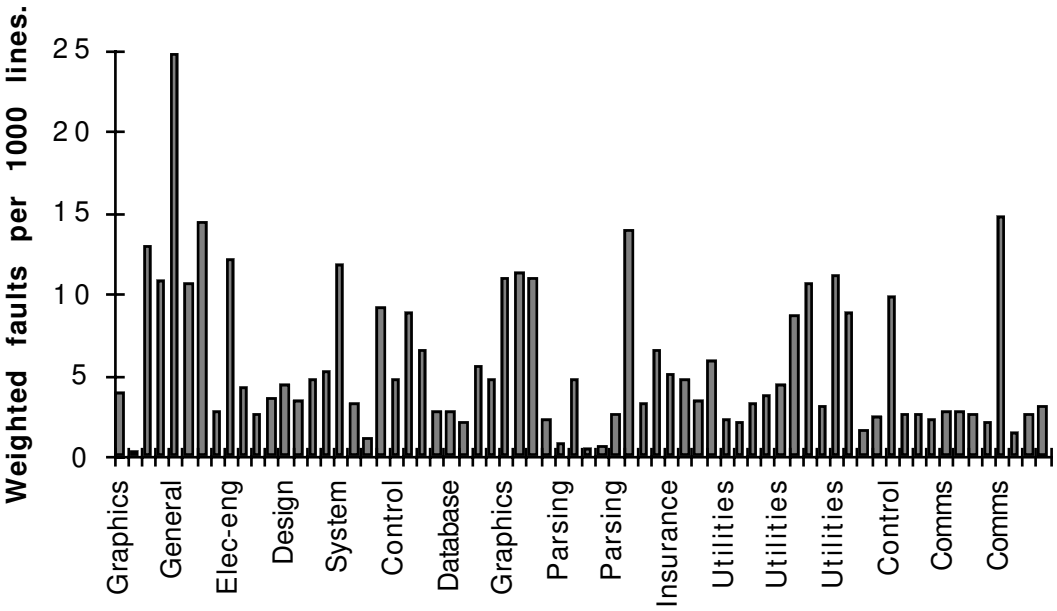

Figure 2: Weighted fault rates per 1000 lines of code for a wide variety of commercially released C applications plotted as a function of industry.

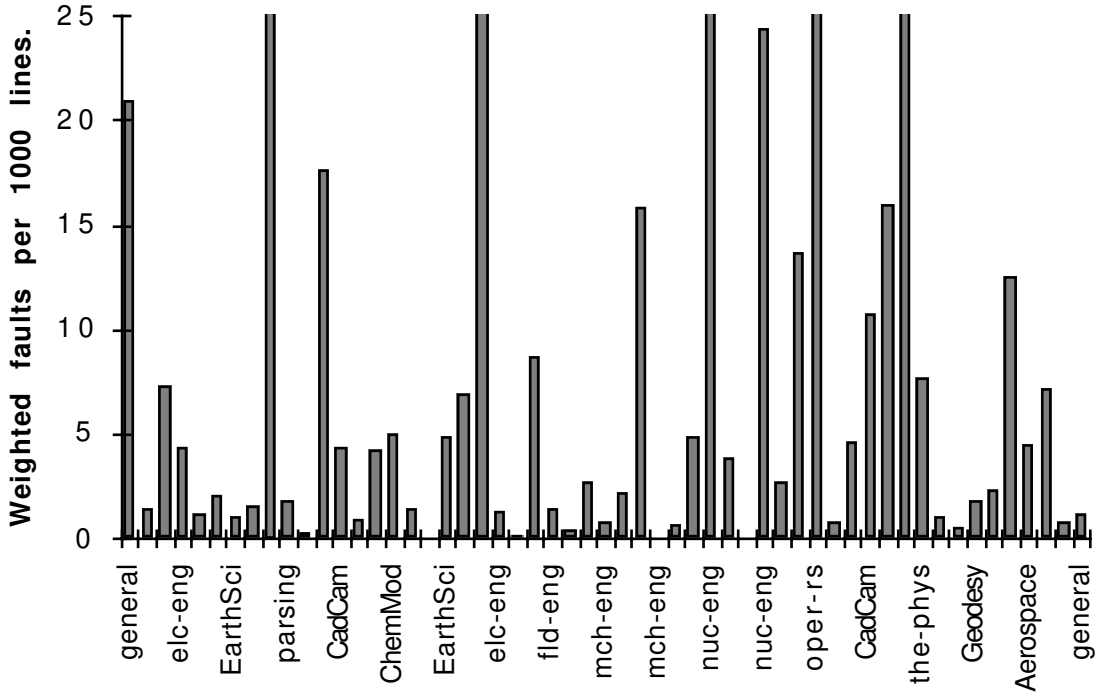

Figure 3: Weighted fault rates per 1000 lines of code for a wide variety of commercially released Fortran 77 applications plotted as a function of industry.

These two figures have the same vertical scale. As can be seen, the patterns are essentially similar in that there is no discernible relation with application area, (or with integrity level when the data are inspected more closely). The only essential difference seems to be that when a Fortran 77 package is bad, it is really bad, as exemplified by the nuclear engineering code about 2/3 the way along the applications axis of Figure 3. This package climbed to an awe-inspiring 140 weighted static faults per 1000 lines of code, and in spite of the aspirations of its designers, amounted to no more than a very expensive random number generator.

Complexity results

A vast amount of complexity data emerged from this study. Distributions for some 40 published metrics were extracted, however only two of the most well-known are shown in Table 1.

Table 1 This shows a simple distribution of two testability metrics for both the C and Fortran populations

Statistic	Fortran	C
Percentage of population with more than 200 static paths.	28%	10%
Percentage of population with more than 10 decisions	32%	8%

The static path count is essentially a count of the paths through a program assuming that all predicates are independent. It is related to the NPATH metric of (Nejmeh 1988), who recommended a value of not more than 200. The cyclomatic complexity first appeared in (McCabe 1976), who recommended a value of not more than 10.

A few of the more amusing hall of fame statistics were:

Deepest level of nesting encountered in C	121
Largest number of decisions encountered in one C component	2224
Largest number of external variables encountered in one C component	134
Largest number of decisions encountered in one Fortran component	642
Greatest number of static paths encountered in C and Fortran	500,000,000
Greatest number of knots encountered in one Fortran component:	95031

Here a component is a function in C and a function or subroutine in Fortran. Note also that in the last statistic, a *knot* is a crossing of control flow. A value of 50 represents densely woven interlocking logic so one can only speculate as to what was going through this particular author's mind at the time, perhaps something illegal.

Static comparison of C and Fortran 77

Table 2 summarises some basic comparisons between the use of the two languages.

Table 2. Some basic comparison parameters between the two languages

Statistic	Fortran	C
Average executable lines per function or subroutine	106	40
Average number of function or subroutine references per interface fault.	7	37
Average number of arguments per function or subroutine call	6.1	2.4
Average number of executable lines per serious fault.	86	205
Ratio of executable lines to total lines	0.53	-
Ratio of executable lines to total pre-processed lines	-	0.72

As can be seen, Fortran functions or subroutines tend to be about 2.5 times larger than their C counterpart. Interestingly, this is achieved by an interface having around 2.5 times more parameters, showing that *the ratio of executable lines per function to parameters per function is very similar for the two languages*. In terms of serious faults, which is that set of faults deemed to be likely to cause a serious problem in some environment, C is more than twice as good. Although there is a subjective element to this, what was deemed serious had a strong consensus of agreement amongst experienced programmers questioned.

In C, note that function prototypes were well used only around 60% of the time and as a result, interface faults accounted for about 24% of the total. *In other words, if function prototypes were mandated in all C functions, 24% of all serious faults would disappear.* The computational scientist should not use this as an argument in favour of C++ or Ada in which they are mandated. A large number of new failure modes result from this action, which lack of space prohibits further discussion here. The net result of changing languages appears to be that

the overall defect density appears to be about the same, (Hatton 1997). In other words, when a language corrects one deficiency, it appears to add one of its own.

EXPERIMENT T2: DYNAMIC ANALYSIS RESULTS

Overview

Experiment T2 explored one of the application areas studied in T1, that of seismic data processing in the Earth Science industry, in an exhaustive series of *dynamic* tests based around the concept of N-version programming or diversity. A fuller description of this experiment first appeared in (Hatton and Roberts 1994).

An overview of seismic data processing

Seismic data processing is the dominant tool in the search for oil and gas and is also used extensively in earthquake studies and also in site surveys before the construction of major civil engineering projects such as bridges and nuclear reactors.

A typical dataset might be perhaps 10^{11} bytes, or around 5×10^{10} digital values which arrive at the data processing centre in various formats. Seismic data in its raw state is of quite poor quality by the standards of most sciences and rarely exceeds a signal-to-noise ratio of 1. As a result, this dataset is subjected to a large number of well-known mathematical and image-processing operations. These include specialised statistical operations aimed at improving the basic quality of the data by exploiting the high degree of redundancy as well as numerous algorithms familiar to scientists in other numerate disciplines, for example, the multi-dimensional Fast Fourier Transform, deconvolution of numerous kinds, wave-equation techniques using finite-difference and other methods, the solution of very large ill-conditioned sparse sets of linear equations, and many others. The aggregate effect is that each sample of a seismic dataset is routinely subject to between 10^2 and 10^3 floating point operations. Putting the above figures together yields a load of around 10^{14} floating point operations required to process the data acquired by a typical marine seismic survey vessel over a period of 4 weeks.

Experimental design

Seismic data processing is carried out by successively applying 30 or so mathematical processes to the input data. The output of each step is the input of the next, rather like a conveyor-belt or pipeline, (c.f. (Hatton, Wright et al. 1988)). The first step was to partition the algorithms into two categories:

- *Published (i.e. unambiguous) algorithms*. 14 such algorithms were placed on the *main sequence*, (i.e. seen by all data) with a data comparison point (*primary calibration point*) after the application of each algorithm.

- *Proprietary (i.e. involving some ambiguity) algorithms.* 20 such algorithms were arranged to appear off the main sequence as *branch* algorithms with a data comparison point after each (*secondary calibration point*), with only subsets of data seeing them. It is worth noting that although they contain proprietary differences from package to package, they involve the same physical process and are referred to synonymously by the end-user, the geoscientists.

The next stage of experimental design involved defining the user-disposable parameters associated with each algorithm. Main sequence processes are characterised by a small number of well-defined disposable parameters, and the existence of these alone with consistent definitions in the various implementations, confirmed this view. In contrast, branch algorithms allow more ambiguity, leading to a more diverse, although still closely-related set of disposable parameters. After some considerable amount of work, a 46-page document was produced which precisely defined the processing sequence and the exact values of all disposable parameters, against which compliance was carefully checked.

The 9 individual packages represent several distinct software architectures and very different machine environments from supercomputer to workstation. Although each one is typically around 750,000 source lines, the current experiment probably illuminated only around 150,000 of these. (Note that an individual process requires on average 5000 lines or so to implement it). Note that each participating company had a recognisable quality system and a written testing policy, some of which were exceptionally thorough.

The input data in their raw form are effectively a three-dimensional matrix, (x,y,z=0,t), recorded in the marine environment, where x is the surface profile direction, y is the distance between the source and a particular transducer and t is time. There are perhaps 400 x-positions and up to 200 y-positions and 2000 t-positions. Numerous windows of the data were defined in which detailed statistical comparisons were performed to supplement other comparisons performed on all the data.

The accuracy of the analysis software (also written in two independent versions) was validated by the fact that the first primary calibration point, where the data is read from the tape,

is merely a publicly defined bit-shifting operation involving negligible floating point computation in which the data is simply re-formatted and re-normalised from a floating point format used by IBM (an industry standard in the seismic industry) to the host specific floating point format. The analysis software showed that all packages (apart from two subtle errors uncovered in two of them and later verified with their developers) agreed to within around 0.001%, the maximum precision available with single precision floating point.

Results

The results are fascinating and deeply disturbing. First and foremost, shifts of one or two in the t-component of the seismic trace occur throughout the processed datasets. These occur between different packages and in the same packages between different calibration points. They are a manifestation of the well-known "one-off" array index problem (c.f. for example (Koenig 1988)) and are obvious evidence of errors. Before differences were computed, all these shifts were taken into consideration, as they would otherwise artificially exaggerate data-point by data-point differences.

After removing the shifts, the normalised average absolute differences between the traces , f_d, was computed using the following formula:

$$f_d = \frac{1}{n_x} \sum_{x=1}^{n_x} \frac{1}{n_t} \sum_{t=1}^{n_t} \frac{1}{n_c} \sum_{c=1}^{n_c} \left\{ a_{dcxt} \cdot \left| a_{dcxt} - \frac{1}{n_c} \sum_{c=1}^{n_c} a_{dcxt} \right| \right\} \qquad (1)$$

where a_{dcxt} is the amplitude at coordinate d, package c, trace x and time sample t. Note that the summations were computed as trimmed means, with a trim factor of two means, to control gross outliers which occurred periodically throughout the data, (c.f. (Tukey 1977)). All amplitudes were pre-normalised to have a maximum absolute amplitude of 1 prior to data analysis. There are many potential choices for computing the differences between the data. Several were tried independently, but all gave the same qualitative behaviour and the central attraction of the above method was that it gave values of around 5% when differences became visually obvious, roughly correspondingly with the visual bandwidth of the eye on data of this kind.

The statistic computed in equation (1) is averaged over all time. However, in order to see whether agreement differed depending on the temporal properties of the data, estimates of this statistic for around thirty t-subsets of the data, some including signal and some noise only, were computed. The results lead to an averaged disagreement for each calibration point summarised in Figure 4. They show that the data agrees to within 0.001% after being read from tape at the first primary calibration point, coordinates 1 and 2, the re-formatting and re-normalising stage.

After this, agreement deteriorates steadily until calibration point 4 where there is a spread of around 8%. Calibration point 5 is empty because the nature of the process means it cannot affect the data, but between primary calibration points 6 and 10, the spread jumps dramatically to around 30-100% with one or two even larger outliers. Interestingly this corresponds to processes involving significant amounts of computation. After this point, there is a data compression stage whose natural redundancy reduces the spread to around 20% and then things deteriorate rapidly to *a spread of around 100% at the coordinate where the geoscientist inspects the data* ! Note that the data compression stage reduces the overall number of t-subsets substantially giving less calibration values between coordinates 11-14 than at coordinates 10 and earlier. Note also that only eight companies contributed at coordinate 8 and only four companies at coordinate 12.

As was expected the spread of disagreement between secondary calibration points was 2-7 times worse than that for the primary calibration points. This is suggestive that specification differences further contribute to the overall disagreement just as inadvertent implementation errors do in the case of the primary calibration points.

Detailed analysis of the disagreement showed also that *it is spatially non-random* and tends to track the underlying data. Furthermore, the non-randomness was not due to a single consistently deviant package, but spread amongst the packages, with different packages assuming the dubious honour of being most deviant at different calibration points. To illustrate this, Figure 5 shows the overall percentage of data at each processing coordinate for which a particular company's package represents the furthest outlier. The size of the circles gives the percentage. Although package 7 gives a consistently poor performance, other packages come and go as can be seen by the deterioration in package 6 from processing coordinate 7 onwards, and the improvement in packages 1 and 2 after a shaky start. It was also observed that the results from the packages are non-Gaussian distributed and cluster into distinct groups with outliers.

Finally, to see the data from the geoscientists point of view, Figure 6 shows the 9 different views of the same data at primary coordinate 14, the stage at which the process of data interpretation by the geoscientist begins. From a geoscientists's point of view however, these differences are not subtle, corresponding to alternative but equally legitimate lithological views which can fundamentally affect the conclusions reached as to the nature of potential hydrocarbon accumulations. This has been confirmed independently by showing the datasets to a number of experienced geoscientists in different companies not directly affiliated with this

study, (Hatton and Roberts 1992). Put simply, this could lead to a 20 million dollar well being drilled in the wrong place !

Feedback

It is of obvious interest to see if obvious discrepancies in the compared datasets could be related back to tangible software failure. We have already seen from the presence of "one-off" errors that software failure is present, but causal experiments whereby discrepancies can be observed, fed back to the developers, related to specific faults which when corrected cause the discrepancy to disappear, are so much more appealing to the scientific method. *Such feedback was attempted several times in different packages and in* ***all*** *cases led to the discovery of a fault causing the failure whose correction caused the discrepancy to disappear.* All of these faults had been in their respective packages for some time. The reader should refer to (Hatton and Roberts 1994) for more details.

Precision of the results

It is interesting to compare the loss of precision reported in experiment T2 due to software fault, with the loss of precision due to other better understood sources of error. Table 3, indicates the approximate number of significant figures of accuracy associated with various experiments or environments. It can be seen, that the departures reported here due to software fault probably dwarf any of the other sources of fault thereby pointing out a problem which needs urgently resolving before we even think of tackling any of the others.

Environment	Number of significant figures
Agreement of single-precision floating point arithmetic, (32 bit)	6
Agreement when running the same reflection seismic data processing package on different architectures and compilers, given the same data, (Hatton, Wright et al. 1988).	4
Agreement when using a single processing package while the package is subjected to continual enhancement, (the norm), (Hatton and Roberts 1994).	1-2
Agreement when processing the same data through independently-developed different implementations of the same seismic data processing algorithms, running on different architectures and compilers, (Hatton and Roberts 1994).	1

Table 3: This table shows the deterioration in agreement with different environments for processing reflection seismology data as cited in the text. Unfortunately, modern seismic data processing interpretation by geologists relies on 2-3 significant figure accuracy, for some of its

deductions. As can be seen, this is not available in general, in fact, it is about an order of magnitude better than is achievable.

How does static fault rate and dynamic failure correlate ?

To answer this quite simply, we don't know, without more experimentation. However, there is good indirect evidence from other sources such as (Pfleeger Lawrence and Hatton 1997) that the kind of static fault we are measuring in experiment T1 is highly correlated with the kind of dynamic failure observed in experiment T2. Figure 7 illustrates the occurrence rate in the population at large of serious static faults, (those with above 50% ascribed severity in experiment T1, which corresponds to about half of all the static faults measured), described as 'average' dynamic testing, compared with the serious static fault rate measured in the much more thoroughly tested system described by (Pfleeger Lawrence and Hatton 1997).

The ubiquity of the static faults reported in experiment T1 suggests therefore that there is every reason to believe that the dynamic failures occuring in the earth science study also occur in other application areas. It would certainly be stretching credulity a little far to think that only the application area studied dynamically in experiment T2 was afflicted with these failures.

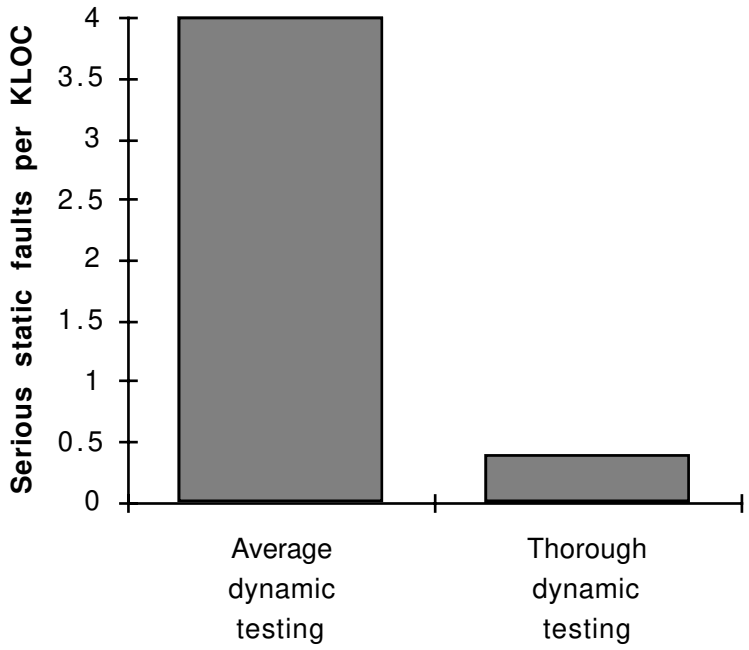

Figure 7: A comparison of the serious static fault rate in the population at large, compared with a much more thoroughly dynamically tested system. This supports the view that static fault measurement and dynamic failure occurrence are strongly correlated.

Conclusions

The evidence for the observed disagreement resulting from software problems is overwhelming and is summarised in more detail in (Hatton and Roberts 1994), so here we will content ourselves with attempting to summarise the lessons from both T-experiments. These are:

- Commercially released C and Fortran software are provably full of statically detectable faults, irrespective of the existence of any quality system, level of criticality, or application area. In fact, there are about 8 serious faults per 1000 executable lines in C which are statically detectable in commercially released code and about 12 per 1000 executable lines in Fortran. This is almost certainly true for other languages also. If a language contains a hole, programmers will fall into it. All languages contain holes.

- In one application area which emerged better than the average in terms of statically detectable fault in T1, the disagreement between 9 different implementations of the same published mathematical algorithms written in the same language using the same input data and same disposable parameters is much worse than anticipated, and in this case reduces output data agreement to around 1-2 significant figures. In comparison, porting the same software to different machines and using the same data gave 4 significant figures of agreement according to (Hatton, Wright et al. 1988). So it isn't the compiler or the hardware implementation. It is interesting to note that the Earth Science industry is now beginning to use techniques which require at least 3 significant figure accuracy in order to find ever-smaller hydrocarbon accumulations. This is clearly a questionable venture given the current state of affairs.

- The disagreement between algorithms which have a somewhat less well-specified definition is several times worse than those which are defined formally using mathematics. This is very worrying given the poor specification common in many software implementations.

- Fortran functions are on average about 2.5 times bigger than their C counterparts, but with correspondingly more parameters passed

What options are open to the computational scientist ? We could do several things:

a) Switch languages. However, given that all languages contain problems, this seems definitely a case of jumping out of the frying-pan into the fire. For example, one of the

languages studied here is C. As reported by (Hatton 1995) and (Pfleeger Lawrence and Hatton 1997), C is responsible for producing some of the most reliable systems ever written. Furthermore, modern languages tend to produce similar levels of defect density, (Hatton 1997).

b) Switch paradigms in the hope that it will all magically come together. However, there is no data to my knowledge anywhere in the world to suggest that, for example, object-orientation leads to more accurate or reliable systems. In fact, such evidence as there is suggests that there is no such benefit.

c) Whenever a computational result is announced, attempt to verify it by at least one independent software implementation. This is a step in the right direction.

d) Instead of using parallel systems to compute things faster, use them to make independent computations of the same thing to improve confidence.

e) Only use safe well-defined subsets of languages, e.g. (Hatton 1995).

On this somewhat polemic note, I will finish with an open plea to computational scientists. All but the most trivial of programs is overwhelmingly likely to contain faults, however well 'tested', and the underlying failure rate is virtually unchanged in the last 15 years, (Schwartz 1991). I understand all too well the urgency to make progress but simply swapping software on whose calculations you depend is inherently high-risk. Even when independent implementations agree, there may still be problems as reported for example by (Knight and Leveson 1986), but at least its a considerable step in the right direction. I am sure that the many capable scientists who produced the code analysed here would agree. All the evidence of the T-experiments suggests that the current state of software implementations of scientific activity is rather worse than we would ever dare to fear, but at least we are forewarned, and can therefore do something about it.

ACKNOWLEDGEMENTS

I would first and foremost like to acknowledge Andy Roberts, my consummately capable collaborator at Enterprise Oil company in experiment T1. Many people including my colleagues

at Programming Research Ltd. where this work was done, contributed to experiment T2 and I would like to thank them all. Finally, I would like to thank all the companies who took part. We were all conscious that our sole purpose was to find faults, something which no scientist finds easy to accept, but from which we can all learn something.

REFERENCES

Ghezzi, C. and M. Jazayeri (1982). Programming Language Concepts. New York, John Wiley & Sons.

Hatton, L. (1995). Safer C: Developing for High-Integrity and Safety-Critical Systems., McGraw-Hill.

Hatton, L. (1997). "Re-examining the fault density - component size connection." IEEE Software **14(2)**(March/April 1997): p. 89-97.

Hatton, L. and A. Roberts (1992). Analysing the agreement between seismic software packages: A Seismic Software Calibration Experiment. 62nd. S.E.G., New Orleans, Society of Exploration Geophysicists.

Hatton, L. and A. Roberts (1994). "How accurate is scientific software ?" IEEE Transactions on Software Engineering **20**(10 (October 1994)): p. 785-797.

Hatton, L., A. Wright, et al. (1988). "The Seismic Kernel System - A Large-Scale Exercise in Fortran 77 Portability." Software Practice and Experience **18**(4): 301-329.

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

Koenig, A. (1988). C Traps and Pitfalls. Reading, Mass., Addison-Wesley.

McCabe, T. A. (1976). "A complexity measure." IEEE Trans Soft. Eng. **SE-2**(4): 308-320.

Nejmeh, B. A. (1988). "NPATH: A measure of execution path complexity and its applications." Comm ACM **31**(2): 188-200.

Pfleeger Lawrence, S. and L. Hatton (1997). "Investigating the influence of formal methods." IEEE Computer, Feb. 1997 **30**(2): pp 33-43.

Schwartz, E. I. (1991). Turning software from a black art into a science. Business Week**:** p. 80-81.

Tukey, J. W. (1977). Exploratory Data Analysis. Reading, Mass., Addison-Wesley.