“Dross ex machina: A long hard look at the influence of software defects in computer modelling"

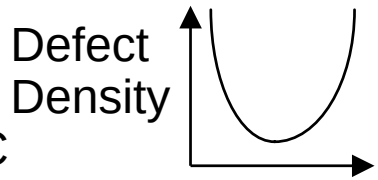
Les Hatton

Professor of Forensic Software Engineering,
CISM, Kingston University, London
lesh@leshatton.org, l.hatton@kingston.ac.uk

Version 1.1: 26/Jan/2006

1984-2005: a personal view

1989-1995



1990-92:
T1: ~10 static faults/KLOC in F77, C. (C++ worse)

1990-1993:
T2: 9-version dynamic experiment.
Only 1 sig. fig. agreement left at end.

1997:
Compression and accuracy

1996:
O-O/C++ has 2x defect correction cost.

1998-1999:
Why do we have so much repetitive failure in software ?

1999-:
Necessary and unnecessary complexity

2005-:
Inspection quality very poor

2003-:
Embedded system paranoia to test arithmetic

2001-:
Thermodynamic properties of software failure

2005-:
Gives up and joins blues band

1995-1999:
Win'95 1 defect every 42 mins.
Mac - 1 defect every 188 mins.
Linux - Almost never.

2006:
Maintenance surprises

1984-1988:
Porting same F77 package gave 4 sig.fig. agreement on different platforms.

1995-7:
Formal methods better ? – Sometimes.
Static fault highly correlated to dynamic failure.

1997:
N-version might be better than we thought

2000-2003:
Formalisation of safer Subsets and definition of signal to noise.

Preparing the ground

Fixing the definitions

- A *fault* is a statically detectable property of a piece of code or a design
- A *failure* is a fault or set of faults which together cause the system to show unexpected behaviour at run-time
- *Asymptotic defect density* is the upper bound of the density of faults that failed in the life-cycle.

v **Note:**

- All failures are caused by at least one fault
- Not all faults fail, (in fact surprisingly few do).

The current state of the art

- **For asymptotic defect density,**
  - The lower limit appears to be around 0.1 defect / KXLOC
  - The current state of the art for good systems appears to be about 0.1 - 1 defect / KXLOC*
  - A reasonable commercial system can expect to have about 2-6 defects / KXLOC
  - A poor system is likely to have > 10 defects / KLOC.
- **Note:**
  - Language choice does not appear to be a major issue.

* Numerous examples of C systems quoted in this range

Overview

- **1984-1988: Portability experiments**
- **1988-1997: Fault experiments**
- **1990-1996: Failure experiments**
- **1996-1997: Correlating fault and failure**
- **1995-2000: Does paradigm shift help ?**
- **2001-: Some interesting questions**

1984-1988: Portability

The Seismic Kernel System (SKS)

- About a million lines of Fortran 77 developed for processing seismic data
- Ported to 10 different architectures, Cray down to Data General with attached FPS array processor. Porting time about 2 weeks.
- Portable graphics based on GKS
- Inhouse portable meta description language for array processing.
- Cost about $3million to develop

1984-1988: Portability

The Seismic Kernel System

- Achieved 4 significant figures of agreement (eventually*) across all architectures on typical seismic data processing benchmarks. Single precision floating point arithmetic used, 32-38 bit.

  * The following statement cost 2 of those until it was found in the middle of a 2-D Fourier Transform:

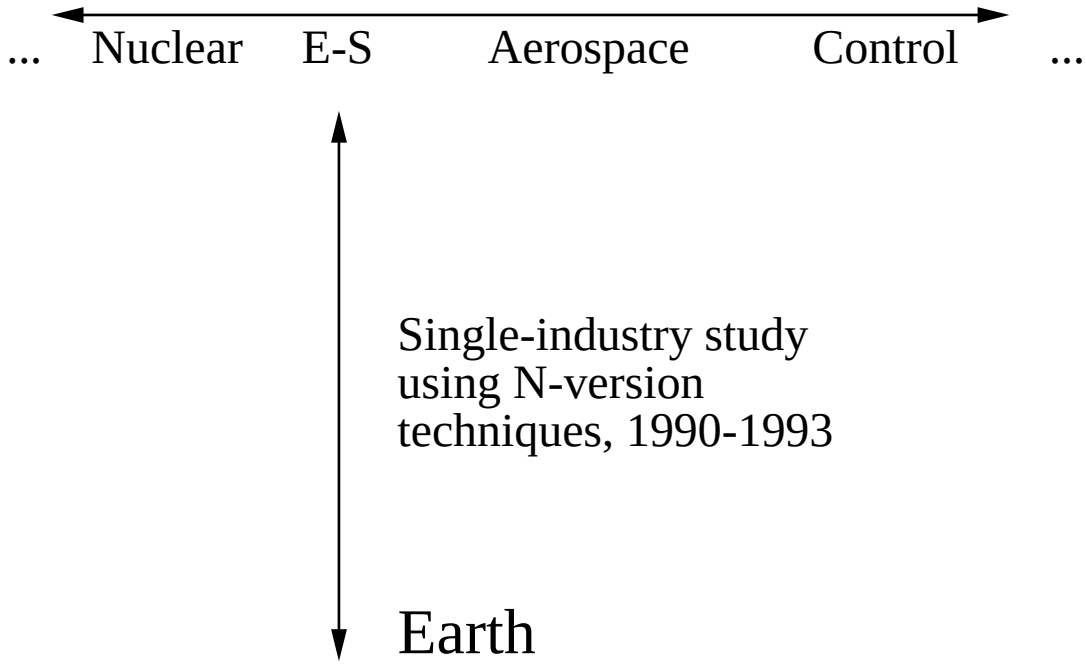
  ```
  if ( ABS(a-b) .gt. 1E-3 ) then ...
  ```

Overview

- **1984-1988: Portability experiments**
- **1988-1997: Fault experiments**
- **1990-1996: Failure experiments**
- **1996-1997: Correlating fault and failure**
- **1995-2000: Does paradigm shift help ?**
- **2001-: Some interesting questions**

The T-experiments

Multi-industry study using static inspection, 1990-1992



1988-1997: The T1 Fault experiments

Stages

- Observed many repeating faults in development of SKS
- Developed F77 parsing engine to study other packages, 1988-1992
- Developed C parsing engine to study similar problems in different language, 1990-1994
- Measured around 100 major systems 1988-1997
- Developed more advanced C parsing engine 1996-2000, restart experiments on embedded control systems

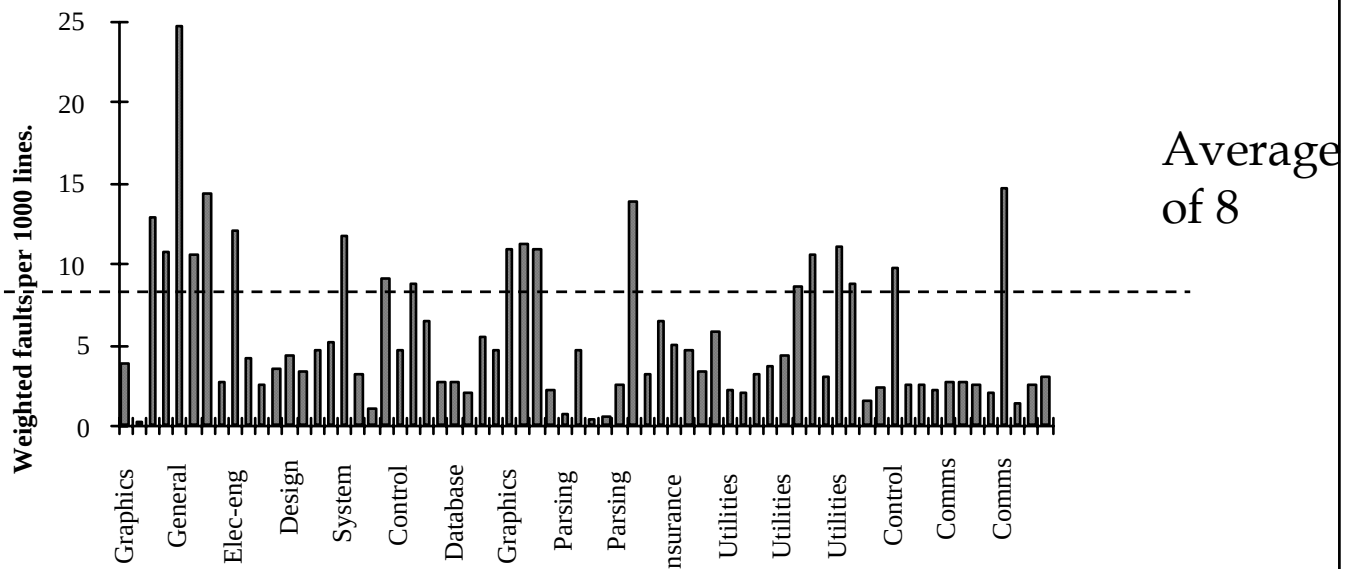
Fault frequencies in C applications

Recent examples:

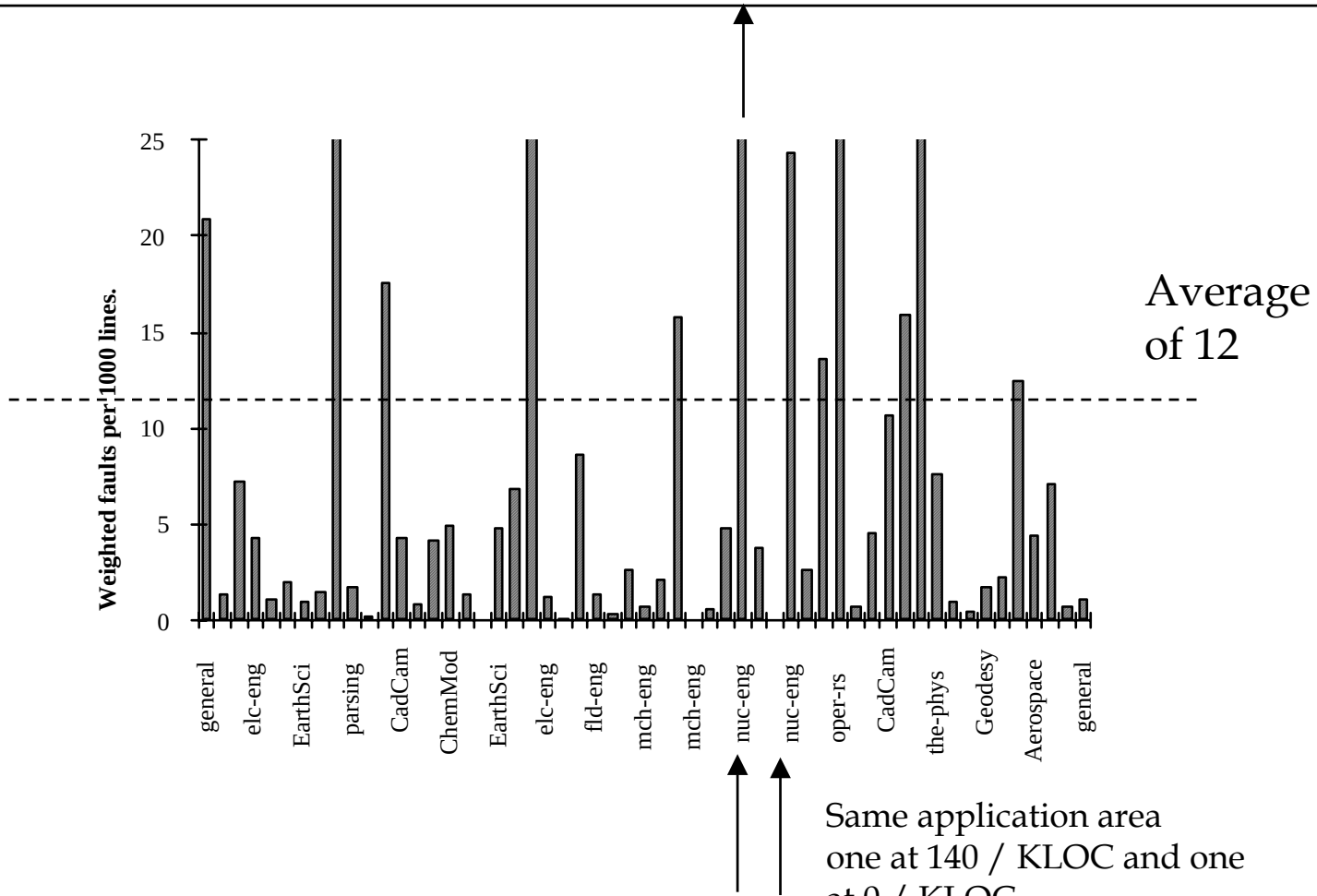
Netscape Javascript Interpreter, 2003
14.78 per KSLOC

F1 racing car software 2003
13.47 per KSLOC

Government agency, 2005
0 per KSLOC



Fault frequencies in Fortran 77 applications



Overview

- **1984-1988: Portability experiments**
- **1988-1997: Fault experiments**
- **1990-1996: Failure experiments**
- **1996-1997: Correlating fault and failure**
- **1995-2000: Does paradigm shift help ?**
- **2001-: Some interesting questions**

1990-1996: Failure experiments

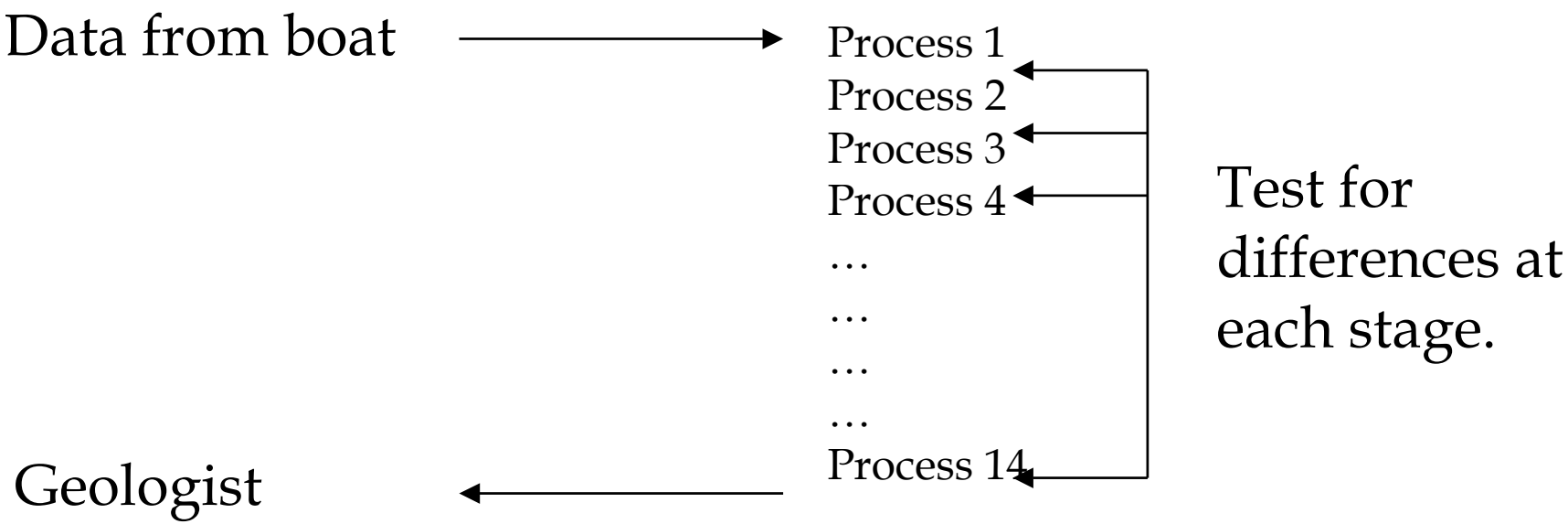
Stages

- An observation: Failure experiments are REALLY expensive compared with fault experiments
- "T2" experiment, 1990-1993
  - Funded by Enterprise Oil plc in the UK
  - Compared the output of 9 packages all in Fortran 77 developed independently
  - Carried out with a colleague, Andy Roberts

T2 details

- 9 independently developed commercial versions of same ~750,000 F77 package of signal processing algorithms.
- Same input data tapes.
- Same processing parameters, (46 page monitored specification document).
- All algorithms published with precise specification, (e.g. FFT, deconvolution, finite-difference wave-equation solutions, tridiagonal matrix inversions and so on).
- All companies had detailed QA and testing procedures.

How seismic data are processed and how the experiment was done



Similarity v. coordinate: No feedback

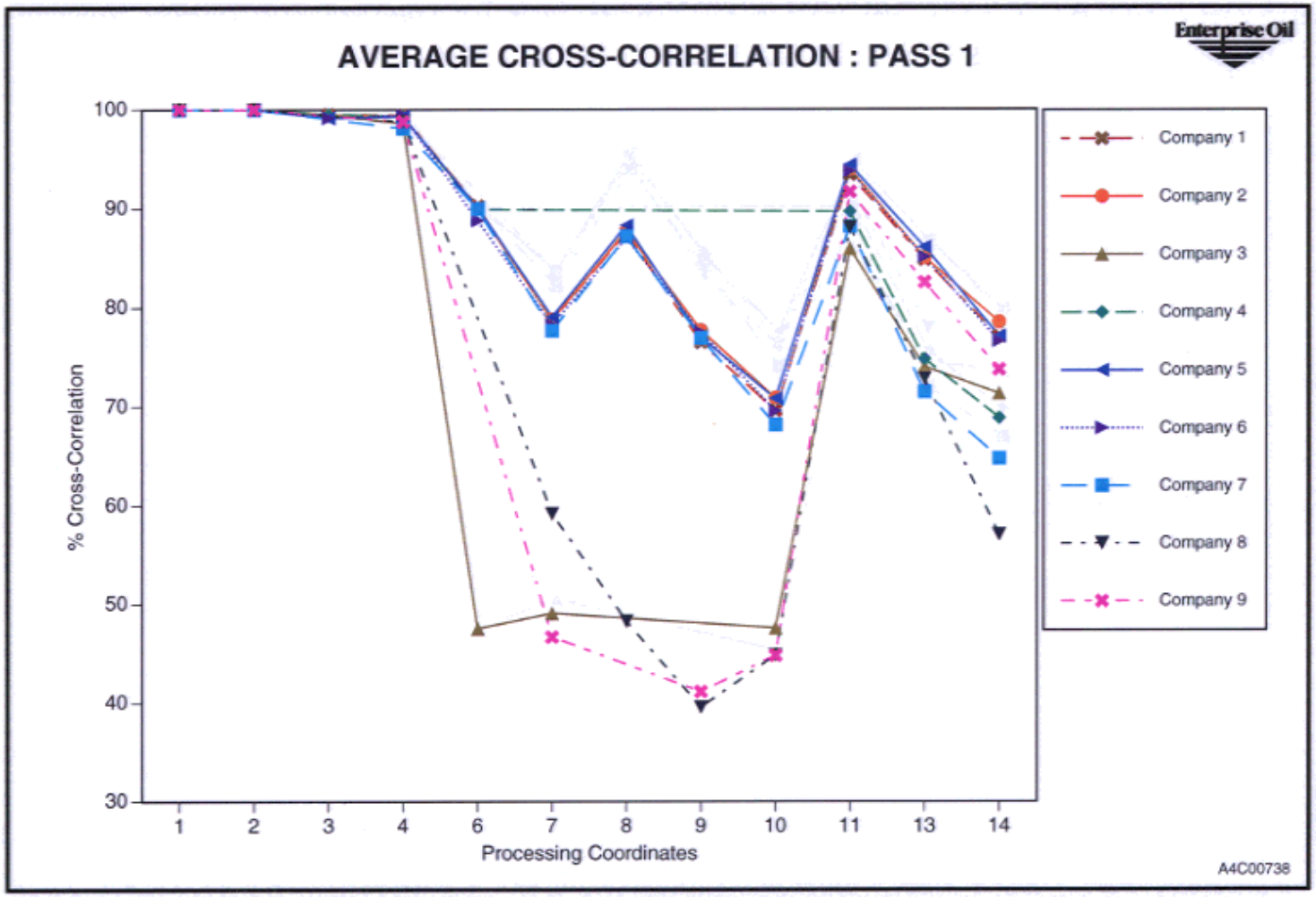
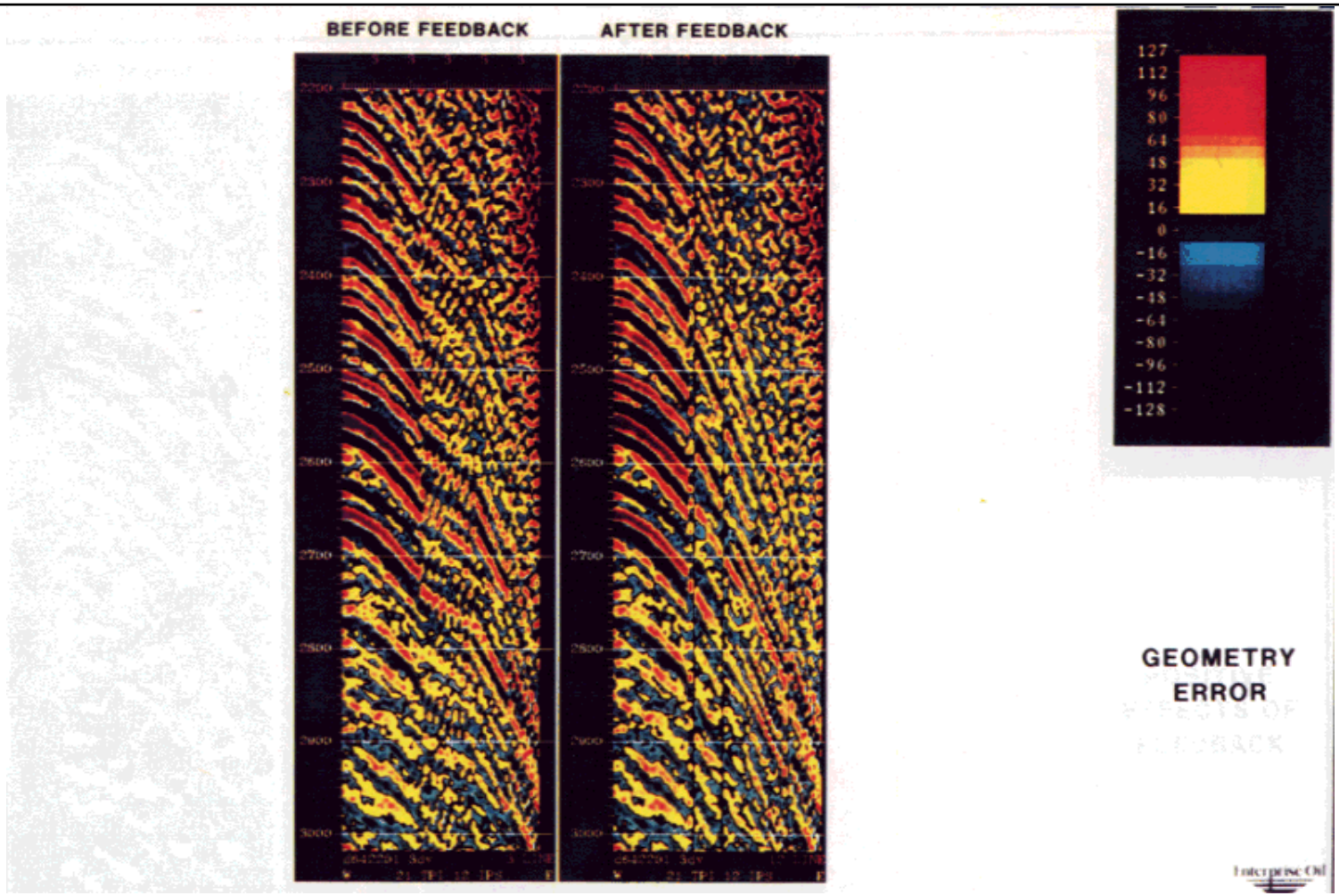


Figure 2.1

Defect example 1: feedback detail



Similarity v. coordinate: Feedback to company 8

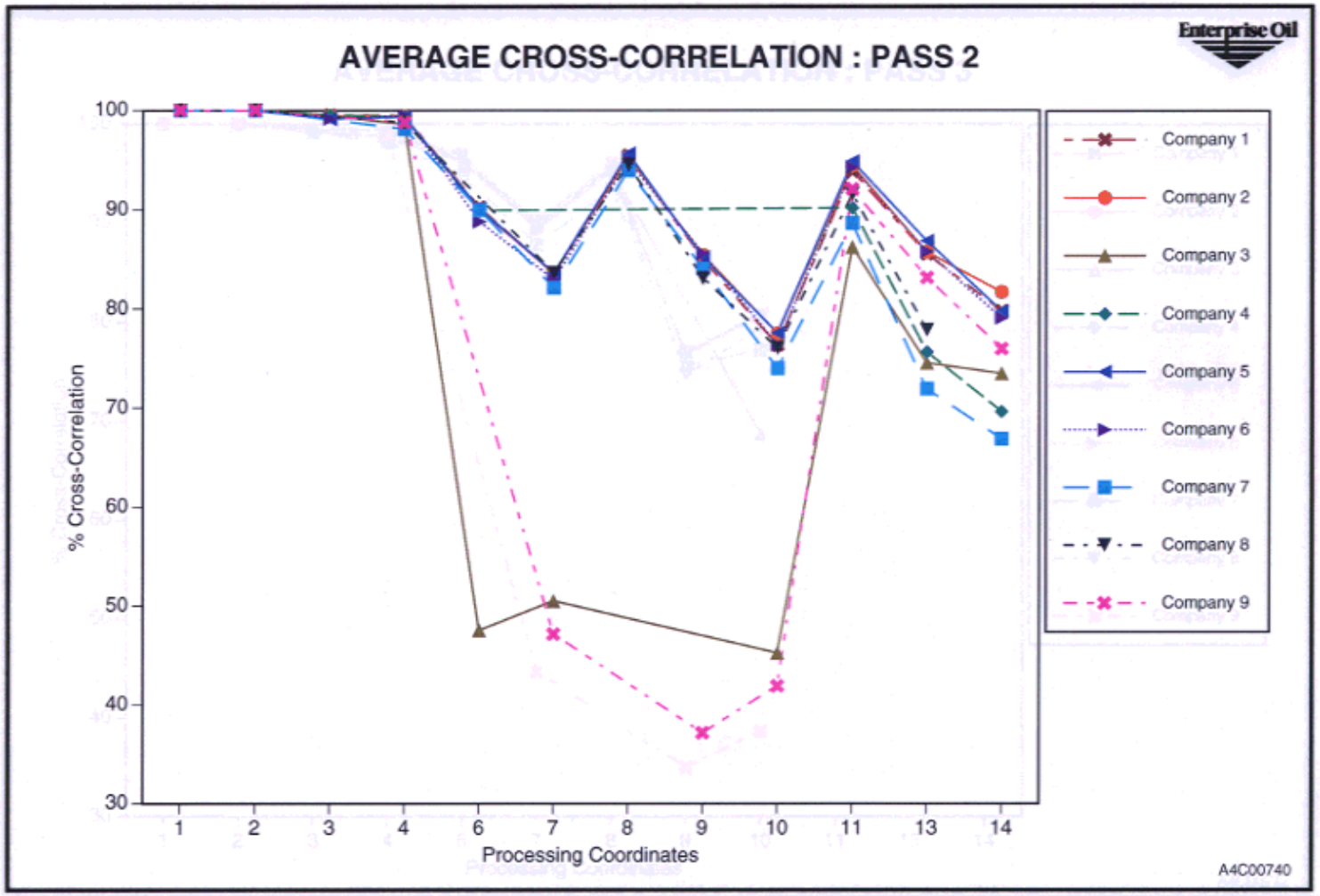
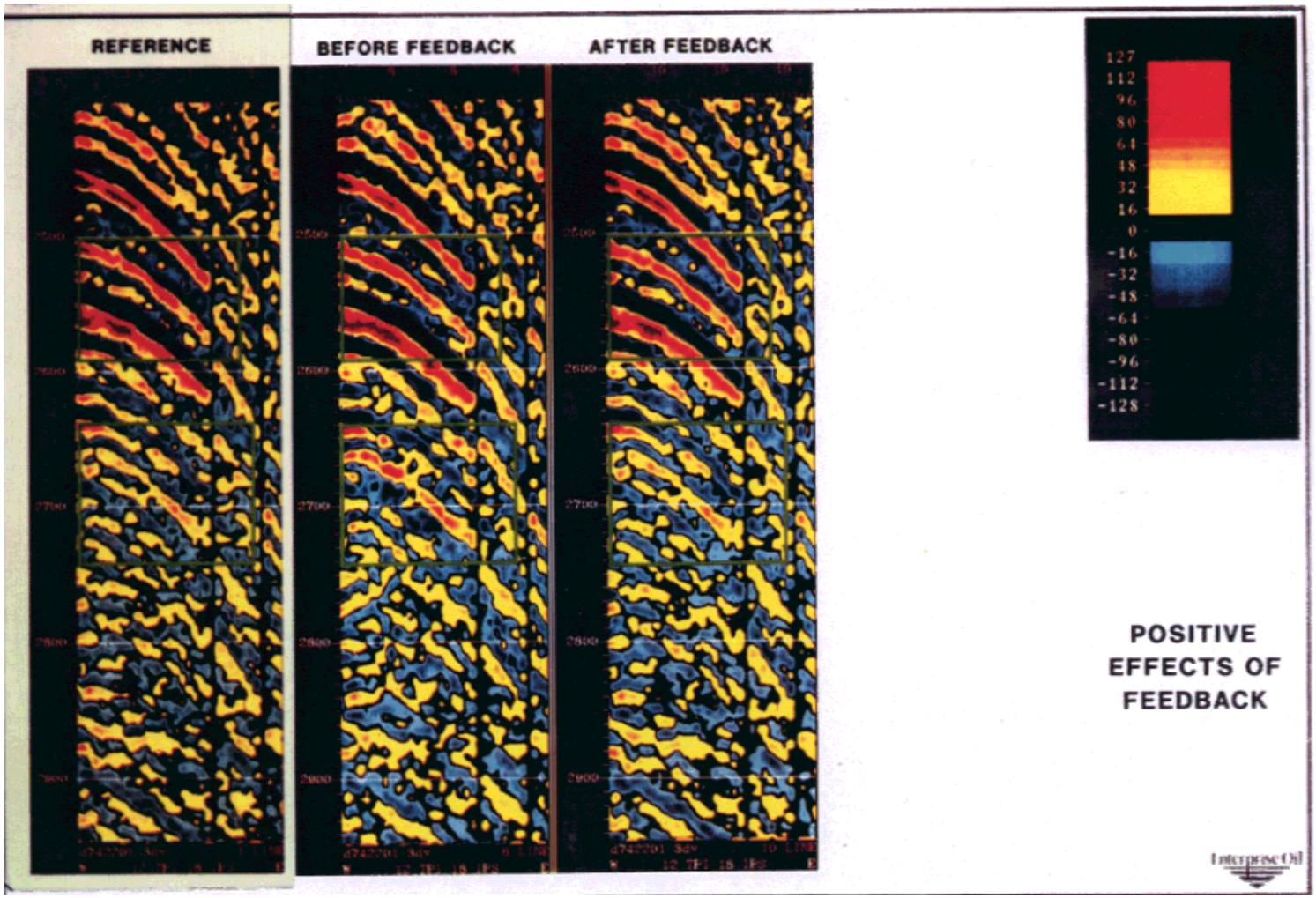


Figure 2.2

Defect example 2: feedback detail



Similarity v. coordinate: Feedback to company 3

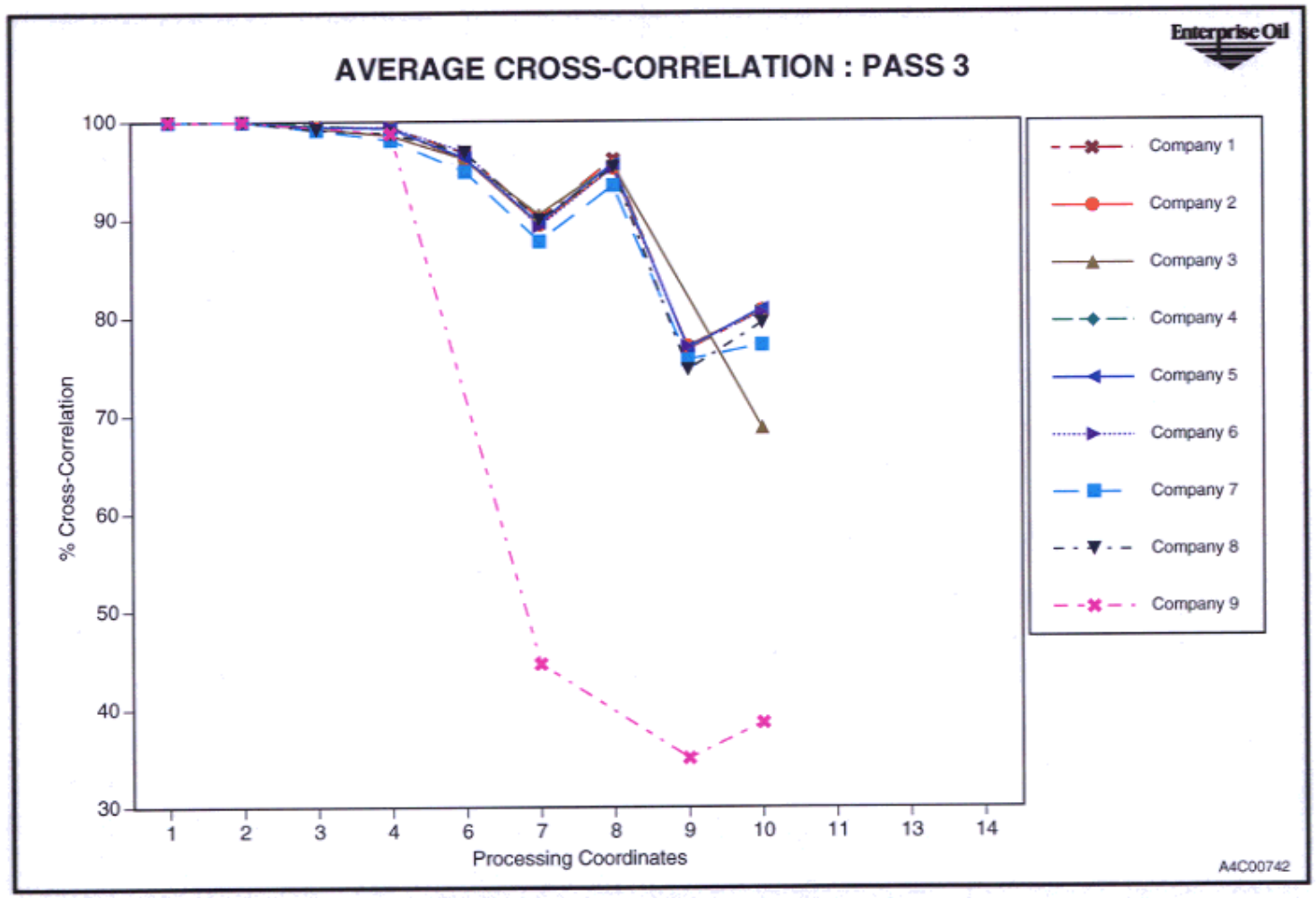
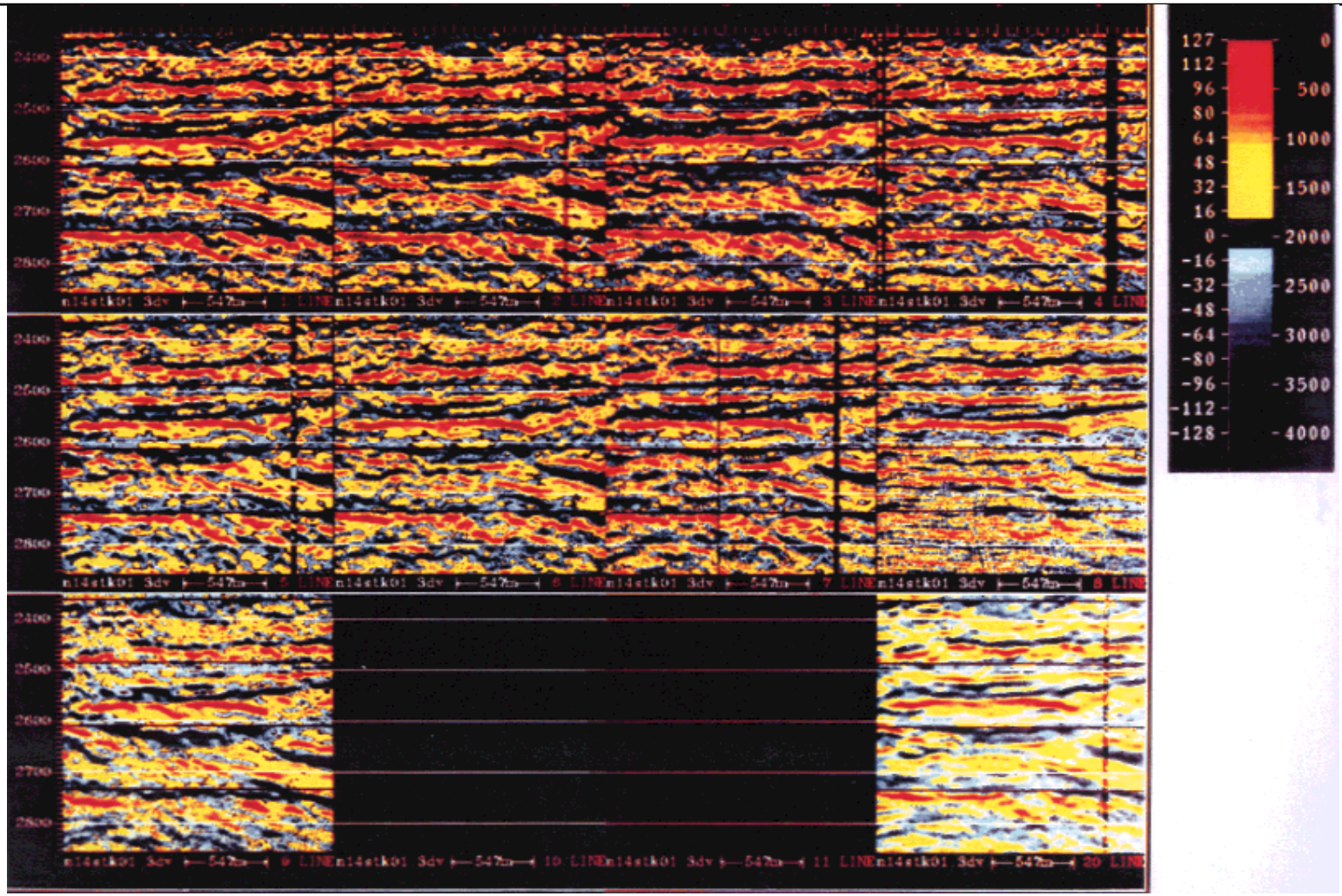


Figure 2.3

The end product: 9 subtly different views of the geology



T2 Results

- **The accompanying slides illustrate:**
  - Only 1-2 significant figures agreement after processing.
  - Disagreement is non-random and alternate views seem equally plausible
  - Feedback of anomalies along with other evidence confirms source of disagreement as software failure.

T2 Summary

To conclude:

- Numerical accuracy in simulations is probably not as good as we believe
  - Software failure is a significant factor
  - Regular maintenance degrades accuracy
  - Moving platforms degrades accuracy
  - Diversity is an excellent method for identifying long standing defects

A summary of 10 years of failure experiments

Seismic processing software environment	Number of significant figures agreement
32 bit floating point arithmetic.	6
Same software on different platforms, same data.	4
Same software on same platform, 5-1 lossy compression.	3-4
Same software subjected to continual 'enhancement'	1-2
T2: different software, same specs, same data, same language, same parameters.	1

Portability degradation

Compression degradation

Maintenance degradation

Diversity degradation

Overview

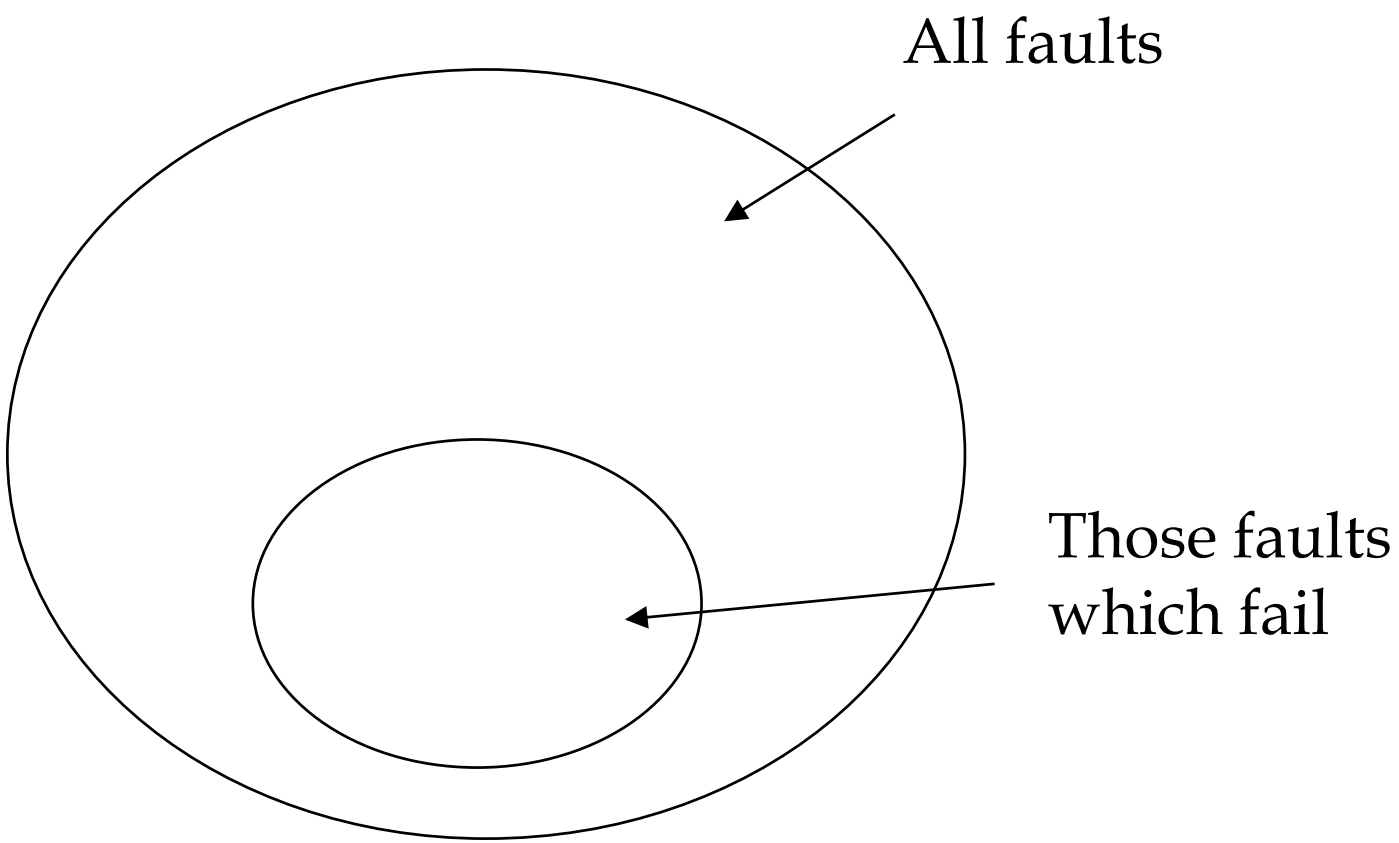
- **1984-1988: Portability experiments**
- **1988-1997: Fault experiments**
- **1990-1996: Failure experiments**
- **1996-1997: Correlating fault and failure**
- **1995-2000: Does paradigm shift help ?**
- **2001-: Some interesting questions**

1996-1997: Correlating fault and failure

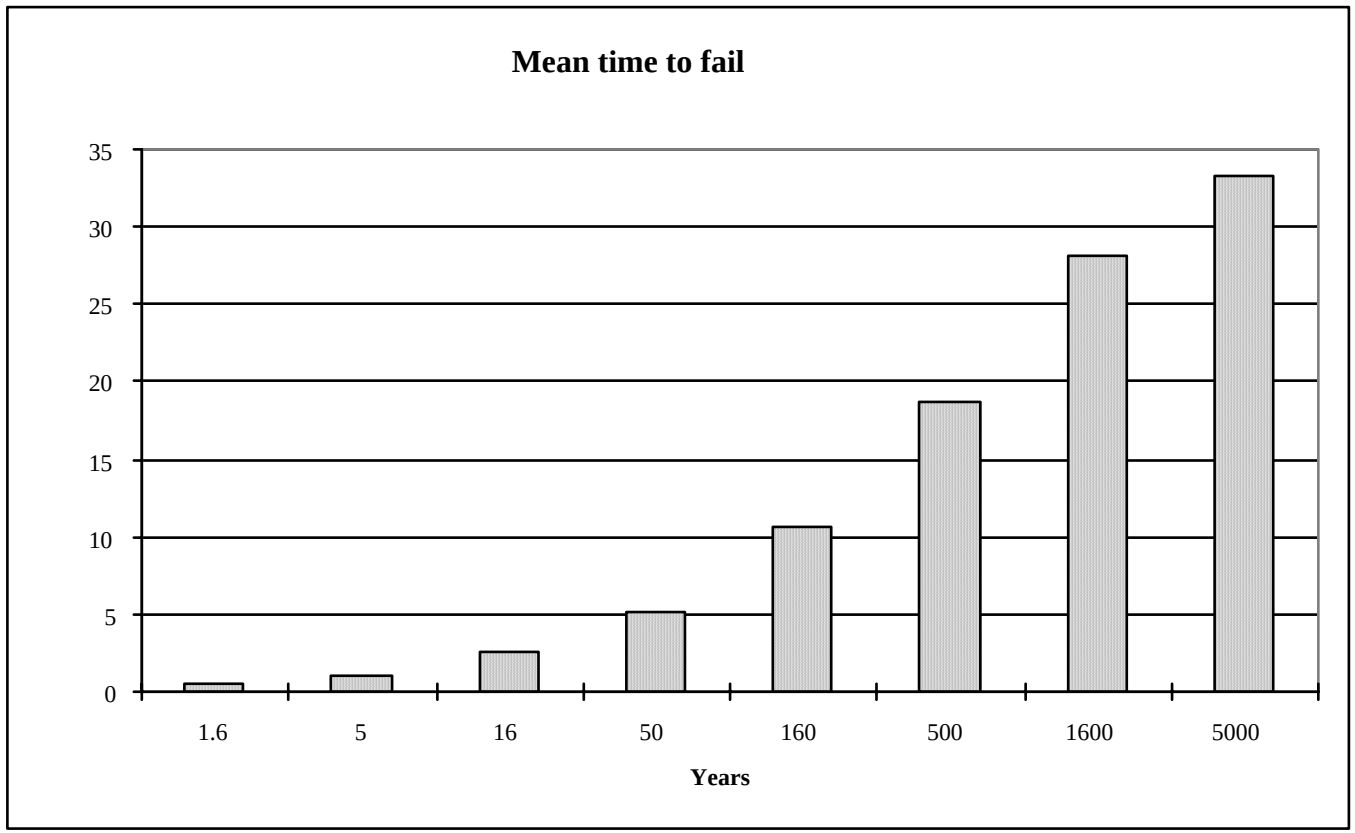
Stages

- How and when do faults fail ?
- An indirect relationship - the Heathrow air-traffic control system and others

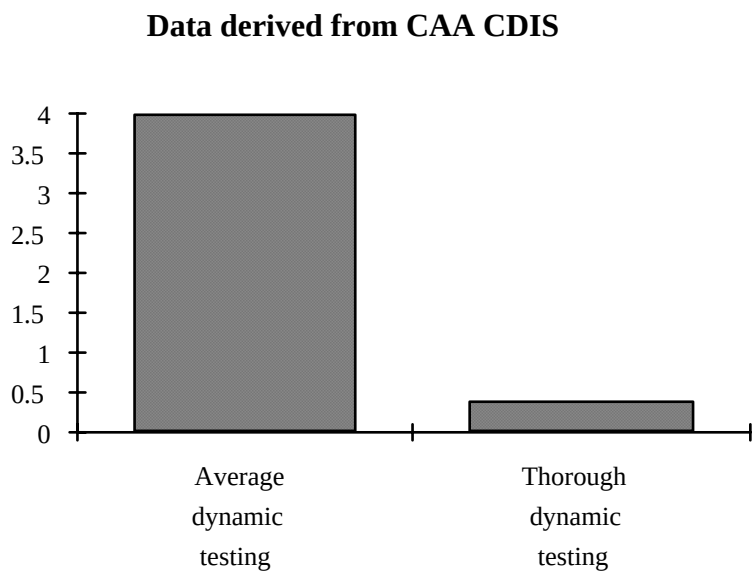
Where and how do faults fail historically ?



Mean time to fail in Adams (1984)

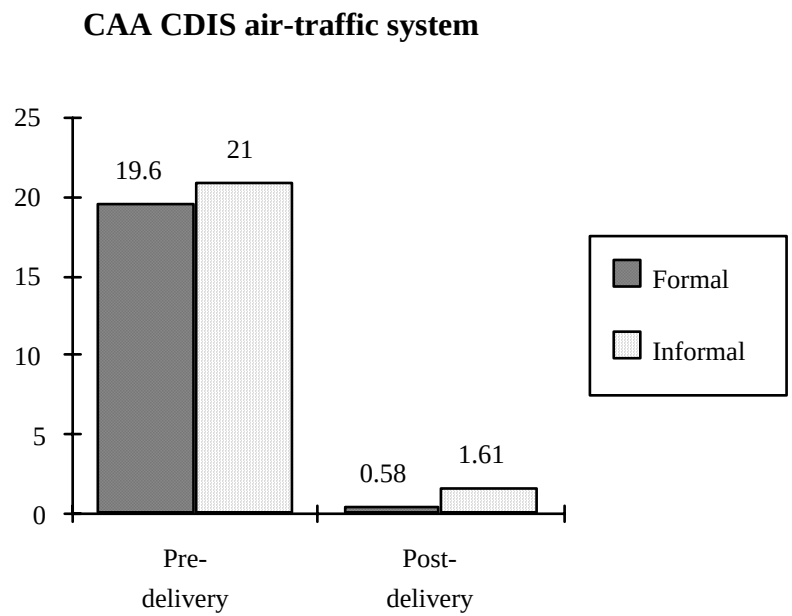


Where and how do faults fail historically ?



This study shows that this class of statically detectable faults do in fact fail during the life-cycle of the software.

Do formal methods work or even help ?



This appears to imply that on their own, formal methods would not be effective.

Overview

- **1984-1988: Portability experiments**
- **1988-1997: Fault experiments**
- **1990-1996: Failure experiments**
- **1996-1997: Correlating fault and failure**
- **1995-2000: Does paradigm shift help ?**
- **2001-: Some interesting questions**

1995-2000: Does paradigm shift help ?

Points to consider

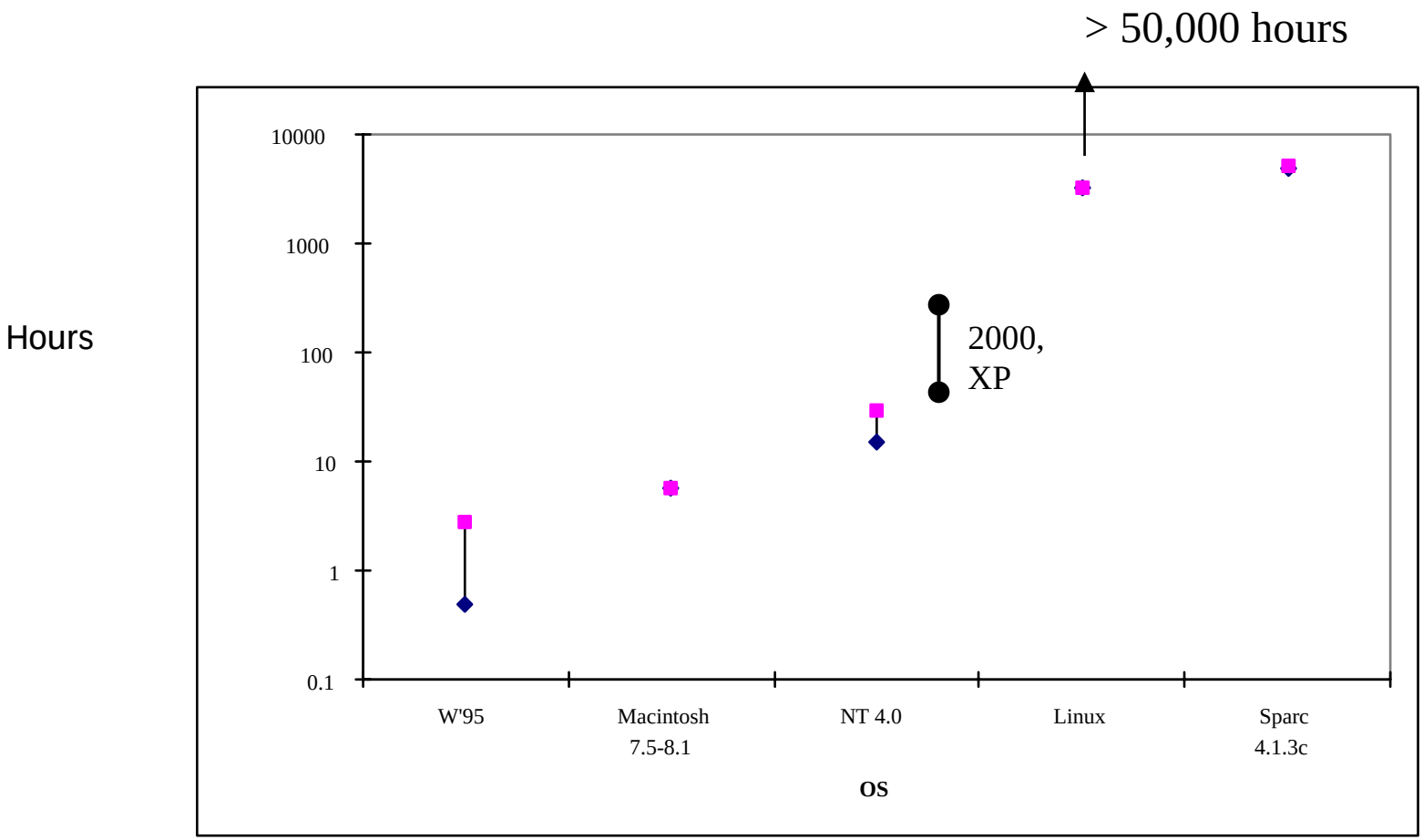
- Software Process
- Development paradigms, for example OO
- Control process feedback

Software Process

Points to consider

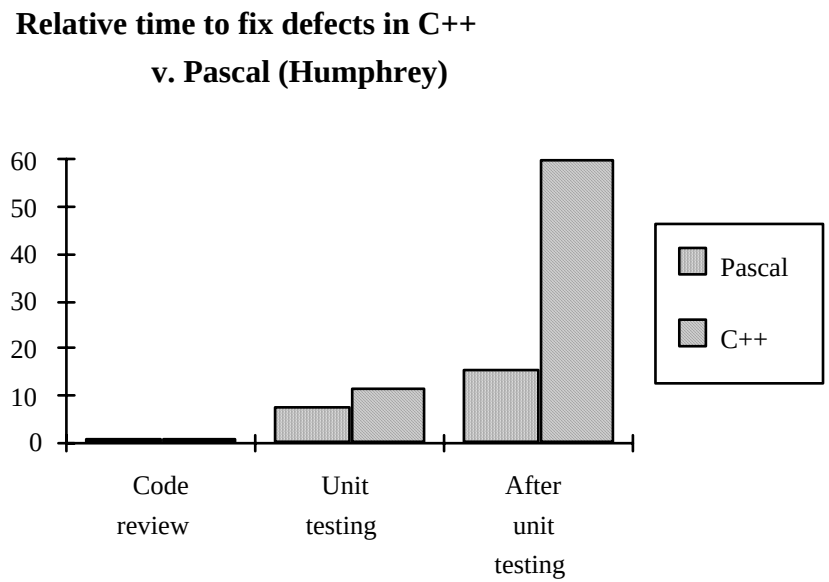
- There is an inherent belief that a good process implies a good product
- Why is Linux so good ?
  - Linux is categorically CMM level 1 so is the CMM wrong or does Open Source development have important properties that we don't understand well yet ?
  - Is the reliability of Linux incremental ?

OS Reliability



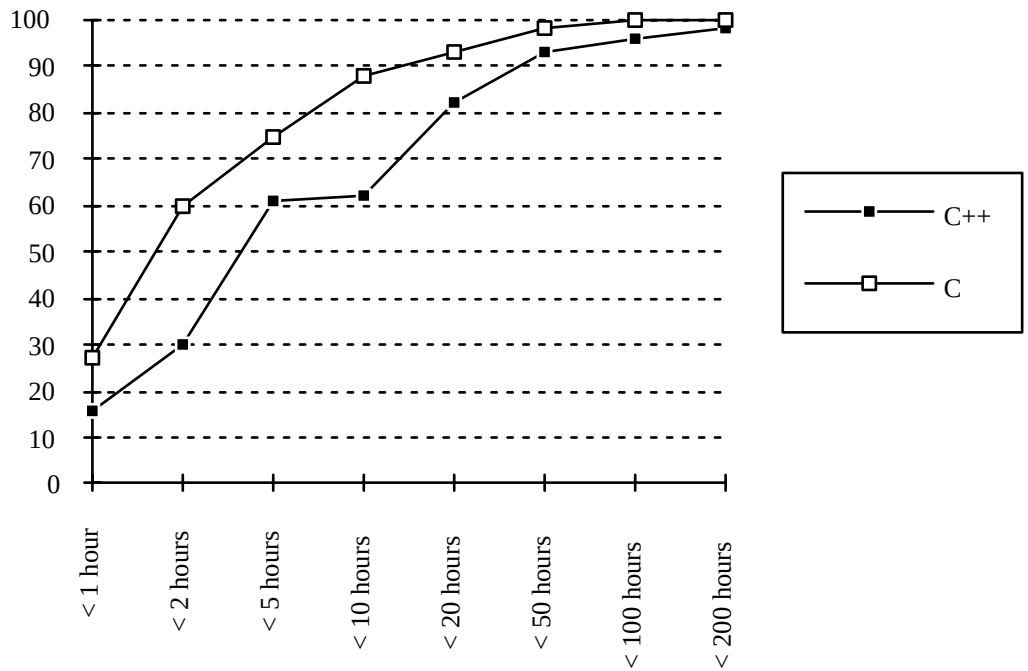
Mean Time Between Failures of various operating systems

Measurement feedback on OO development, (Humphrey)



Measurement feedback on OO development, (Hatton)

Percent of defects found and fixed.



How long it took

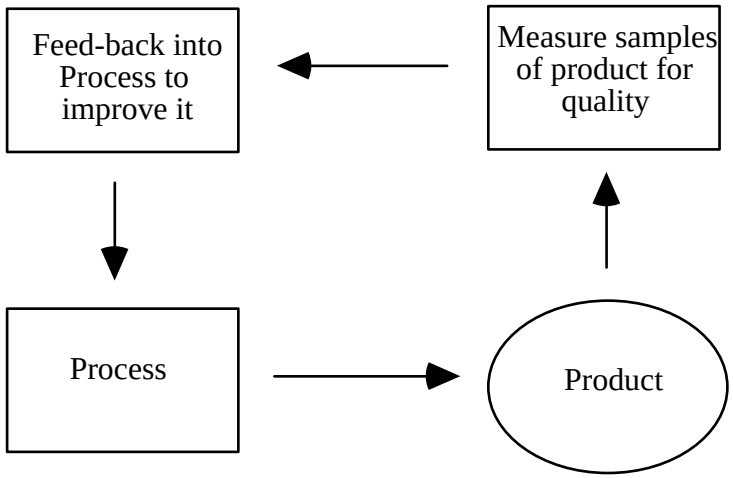
Measurement feedback on OO development

Summary of known measurements

- C++ OO systems have comparable defect densities to conventional C or Pascal systems.
- Each defect in a C++ OO system takes about twice as long to fix as in a conventional system. This is true for both simple defects AND difficult ones. The whole distribution is right shifted.
- Components using inheritance have been observed to have 6 times the defect density.

How much of this is attributable to C++ is unknown.

Control Process feedback - the essence of engineering improvement



If you want to improve reliability, measure and analyse failures.

Overview

- **Paradigm shift is characterised by:-**
  - Fashion / marketing focus
  - Creativity driven
  - The complete absence of measurement
  - Maximises things the engineer CAN do.
- **Control process feedback is characterised by:-**
  - Engineering focus
  - Measurement and analysis of failure
  - Ruthless elimination of known failure modes
  - Maximises things the engineer can NOT do.

Overview

- **1984-1988: Portability experiments**
- **1988-1997: Fault experiments**
- **1990-1996: Failure experiments**
- **1996-1997: Correlating fault and failure**
- **1995-2000: Does paradigm shift help ?**
- **2001-: Some interesting questions**

2001-2006: Some interesting questions

Points to consider

- Can we reverse or even halt linguistic decay ?: Aristotleans v. Babylonians
- Why do defects cluster and/or why are they not linearly distributed ?
- How do we characterise necessary and unnecessary complexity ?
- Why is floating point arithmetic still indifferently implemented ?

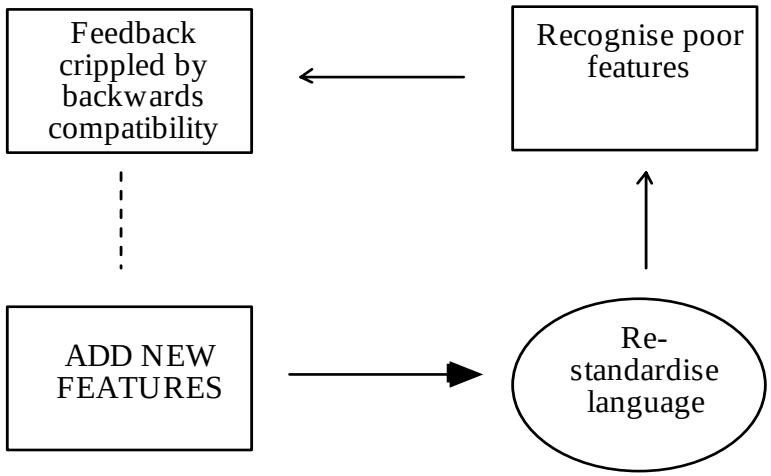
Linguistic decay

In my career, I have been forced to write programs in:-

- Focal
- Atlas Autocode
- Algol
- Assembler
- Fortran 66, 77
- C
- Pascal
- Ada (briefly)
- C++
- Java
- Various scripting languages, Perl, Tcl/Tk, Bash, Javascript, PHP
- C again, (this time from choice)

Why standard languages can't improve

Examples:
C90 -> C99, undefined behaviour doubles as does Standard.
C++99, 808 pages with word 'undefined' appearing 1825 times.
Ada83, 1200 interpretation requests



Using the model of control process feedback, we see that the feedback stage is crippled by the "shall not break old code" rule or "backwards compatibility" as it is more commonly known.

Even new languages struggle:-

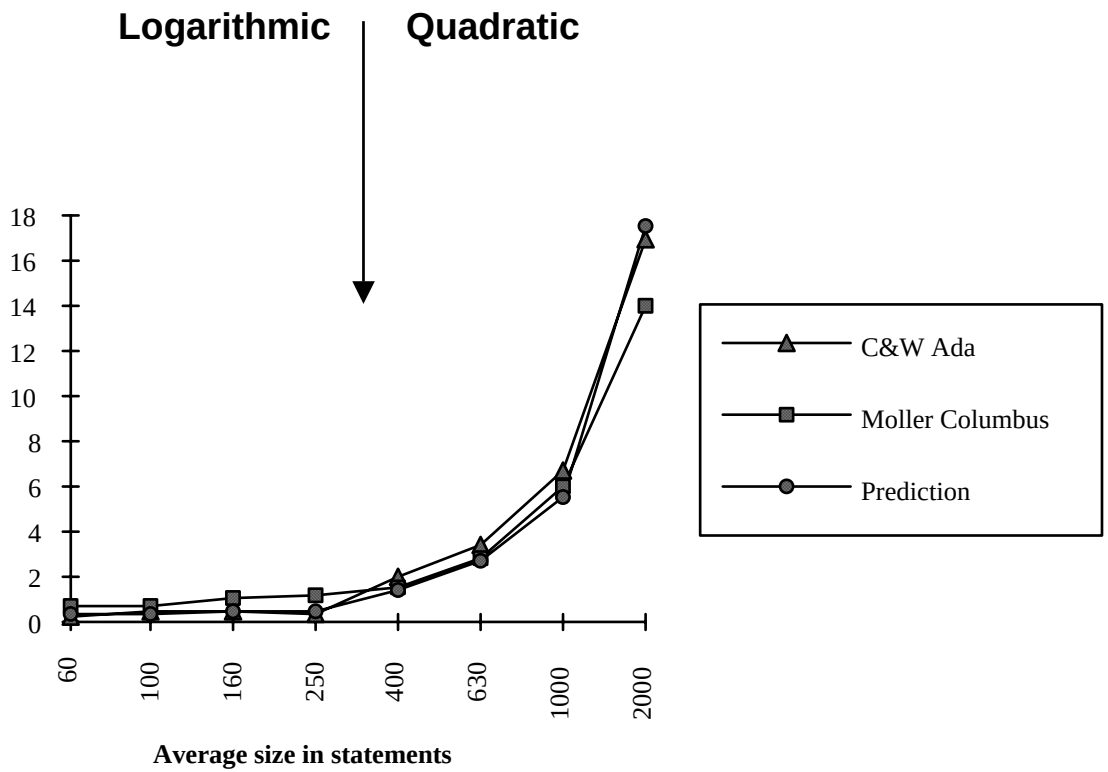
Non-standardised languages also suffer:-

- Javascript: Even precedence was not defined explicitly
- Java: Removed some defects, added some new ones
- Perl: 21 levels of precedence ...
- Excel: beware, -x^2 + 1 != 1 – x^2. (Stevens(2005)).

Standard languages repeat mistakes:

- IEC 1131 removed need to declare variables before use ‘for programmer flexibility’
- C added real variable iteration the year Fortran removed it
- The original C precedence table which contained known failure modes according to its authors is repeated in C++, Perl, PHP, Java, Tcl and so on
- Compiler run-time checking has effectively disappeared
- Compiler validation has effectively disappeared

Defect clustering



Note the non-linear growth. Why does it grow so slowly to start with ?

Necessary and unnecessary complexity

- **In the Knight-Leveson (1986) experiment:-**
  - 27 versions of the same algorithm were developed independently in Pascal
  - The smallest had around 300 lines and the largest was over 1000 lines.
  - The most reliable did not fail in 1 million trials, the least reliable failed nearly 10,000 times.
- **AT&T in the ‘70s and ‘80s:-**
  - it was frequently observed that rewriting the same algorithm 2 or 3 times reduced the size by about the same factor, e.g. diff.

Floating point arithmetic

- **The original *paranoia* was not easily modified for embedded control systems**
- ***Embedded system paranoia* (ESP) appeared in October 2003**
  - Unsupported parts of ISO C can be switched out
  - Single function message(char *string) to communicate with the host, (simply sends a stream of bytes to host)
  - Scores quality of arithmetic on a scale of 1 (excellent) to 6 (failed)
- **Freely downloadable from:-**

  http://www.leshatton.org/ESP_903.html
- **No embedded control system has passed without error yet**

Overall Summary

To conclude:

- On the negative side
  - We are ignoring systematic errors in our software *and* known ways of detecting them
  - Our languages seem to be degrading. They are also growing
  - Repetitive failure modes dominate software systems failure
  - Fluency in a programming language seems to be much more important than the language itself
  - There is little evidence of improvement
  - Defect injection variations between programmers appear to be much larger than defect inject variation in technologies
  - Complexity, deliberate and unnecessary is a major problem
- On the positive side
  - There are some exciting possibilities for improvement

Reference site

For more information, downloadable papers and software, see:-

http://www.leshatton.org/

l.hatton@kingston.ac.uk