2003-

“Five variations on the theme: Software failure: avoiding the avoidable and living with the rest“

Variation 2: “Measuring software and herding cats”

Prof. Les Hatton

Computing Laboratory, University of Kent, Canterbury

Version 1.1: 18/Nov/2003

Introduction to software failure

In the absence of any technology for guaranteeing the absence of defect, engineers are always left with two obligations:-

- When the system fails, it must have the least effect on the user relative to the benefit it provides
- The nature of the failure must be sufficiently well diagnosed that its cause can be found and corrected quickly so the failure does not re-occur.

These are both *Design* issues

Software defect measurement

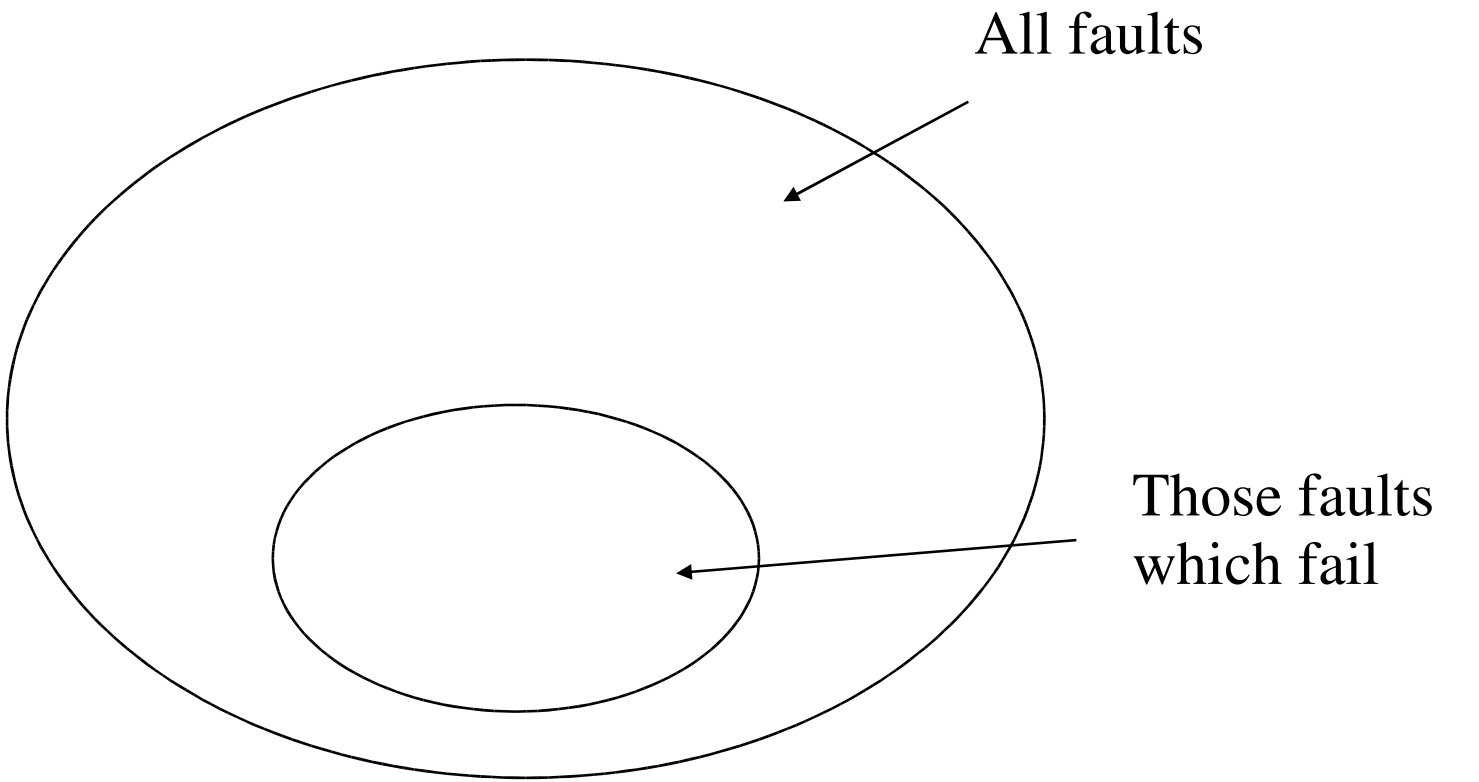
- Definitions of fault and failure
- Defect density and reliability
- The relationship between fault and failure
- Measuring complexity
- Measuring implementation quality

Definitions of fault, failure and defect

Let's define the following concepts:-

- Fault
  - Static property (identified before run-time)
  - Can be identified in either design or source code as a 'mistake'
  - Not all faults fail
- Failure
  - Dynamic property (identified at run-time)
  - Defined to be difference between actual and expected run-time behaviour.
  - Every failure is caused by at least one fault.
- Defect
  - A fault that fails in the life-cycle of the code.

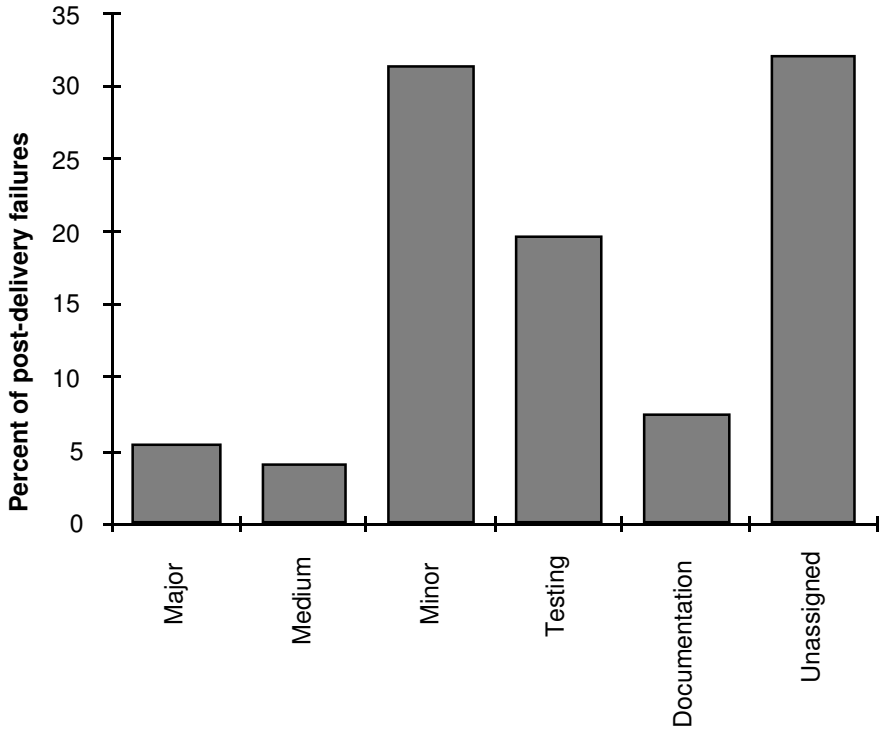
The relationship between fault and failure



Some notes

- When a system fails, it is essential to count the faults for *all* failures, not just the serious failures, (only 5-10% of the faults which fail are observed to lead to serious failure)
- We can't always find out why a system has failed.

Categories of post-delivery failures in CAA Study, 1997

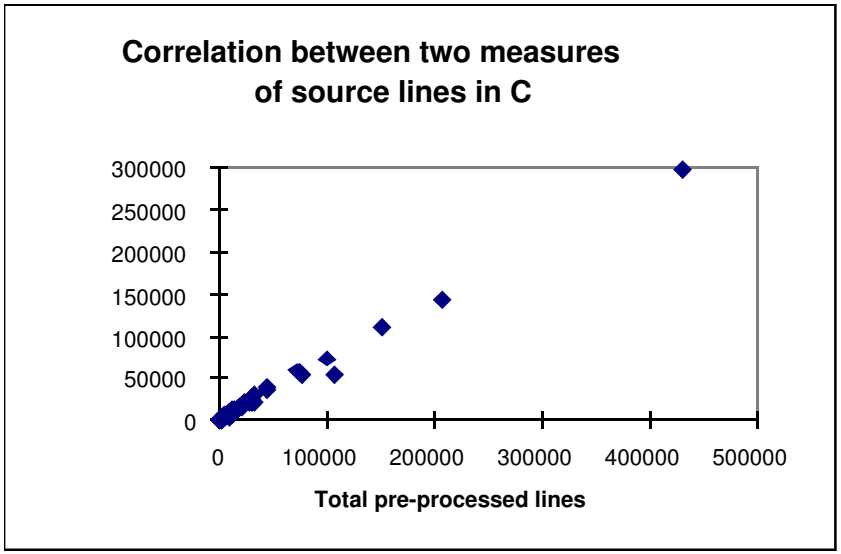


This data suggests that major / minor failure ratio is somewhere between 5% and 10%.

Measuring fault

- *Fault density* is the number of faults per thousand lines of code, (usually called KLOC or SKLOC).
- Definitions of a line of code can vary considerably. There is a range of 2:1 in C between:-
  - Total newlines
  - Non-blank pre-processed lines
  - Executable lines
- Definitions of time also vary. For mean time to fail, it is the *time of use* !, (c.f. HP).

Alternative line measures in C



Units of measurement

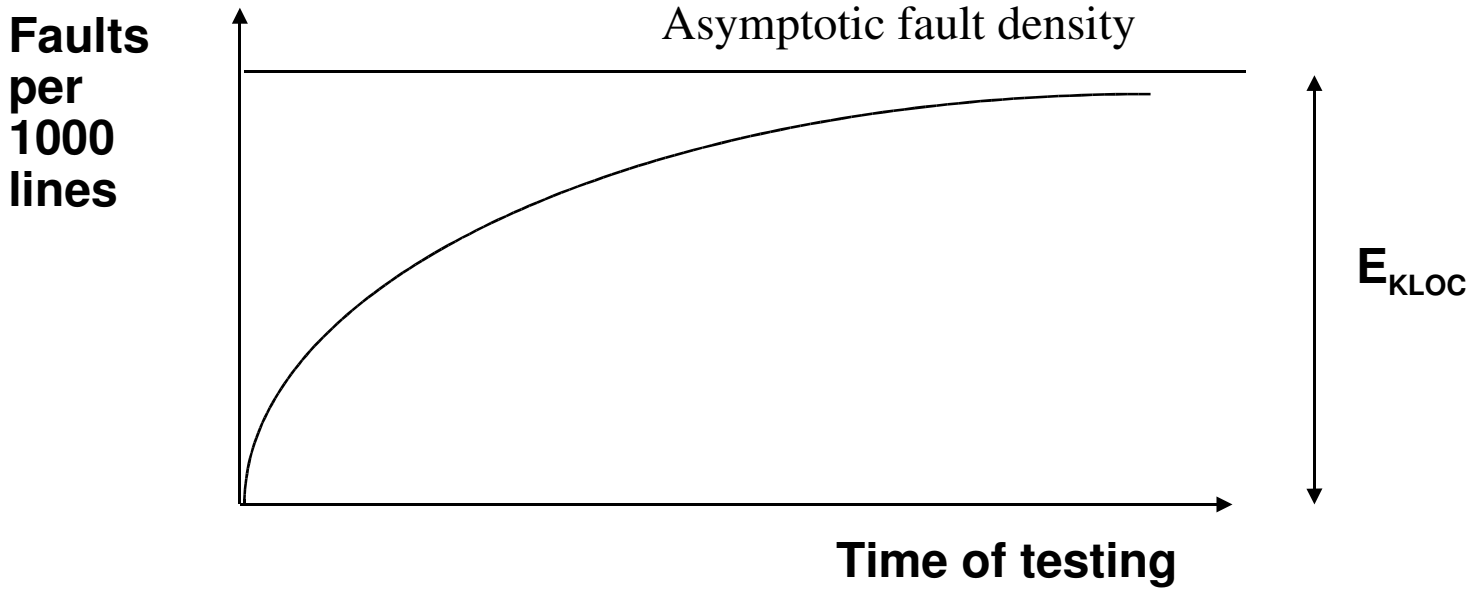
We can identify three different units

- Faults which haven't failed yet (found by static analysis)
  - Volumetric, number per unit 'volume' of code and not time-dependent in essence.
- Faults which have failed
  - Volumetric but time-dependent.

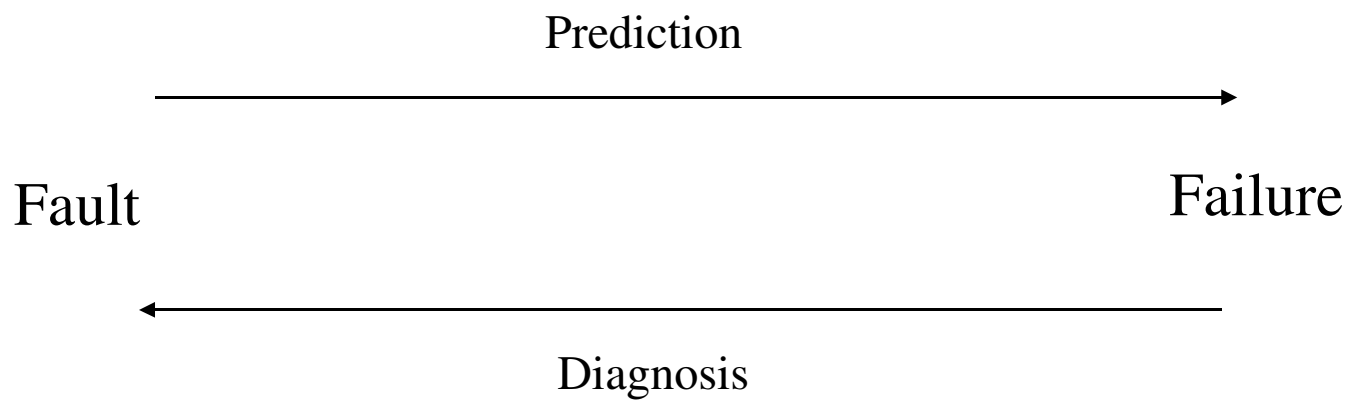
3. System failure
   - The traditional MTBF (Mean Time Between Failures), MTTF (Mean time to fail), PFOD (Probability of Failure on Demand).

Note: there is no known static methodology to predict whether a fault will fail and how serious the failure will be, (e.g. based on complexity).

"Faults that fail" density is a function of time



The prediction problem



Prediction is in its infancy and diagnosis is becoming more difficult

The current state of the art

Let us ask two questions:-

- How good is good ?
- Are we getting any better ?

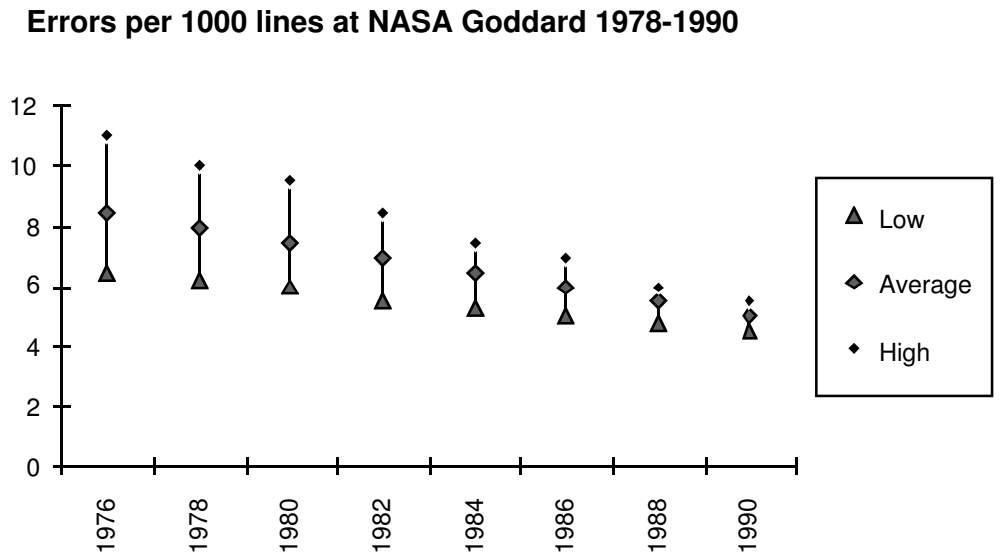
Consider the following data:-

The current state of the art

Part of a worldwide survey of defect densities:

Source	Language	Errors / KLOC	Formal methods used	Life-cycle
Siemens - operating systems	Assemblers	6-15	No	Release
IPL - language parser	C	20-100	No	Dev. only
NAG - scientific libraries	Fortran	3	No	Release
Praxis - Air-traffic control	C	1.25	Yes	Release
Lloyds - language parser	C	1.4	Yes	Release
IBM cleanroom	Various	3.4	Part	Release
IBM normal	Various	30	No	Release
Loral - IBM MVS	Various	0.5*	Part	Projected
Basili & Perricone (1984)	Fortran	6-16	No	Release
Compton & Withrow (1990)	Ada	2-9	No	Release

The current state of the art



The current state of the art

- **After release,**
  - The current state of the art appears to be about 1 defect / KLOC.
  - A reasonable commercial system can expect to have about 3-6 defects / KLOC.
  - A poor system is likely to have > 15 defects / KLOC.
  - Software defect density improving very slowly.
  - Language choice is not a major issue.

Some conclusions

To summarise:

- Given that defect density is roughly constant with time and the size of software systems is doubling every 18 months, then the number of software failures will **double** every 18 months.
- There is a large risk of failure for any new project.
- We are seeing significant evidence of software failure in consumer appliances already.
- Many existing failures could have been avoided.

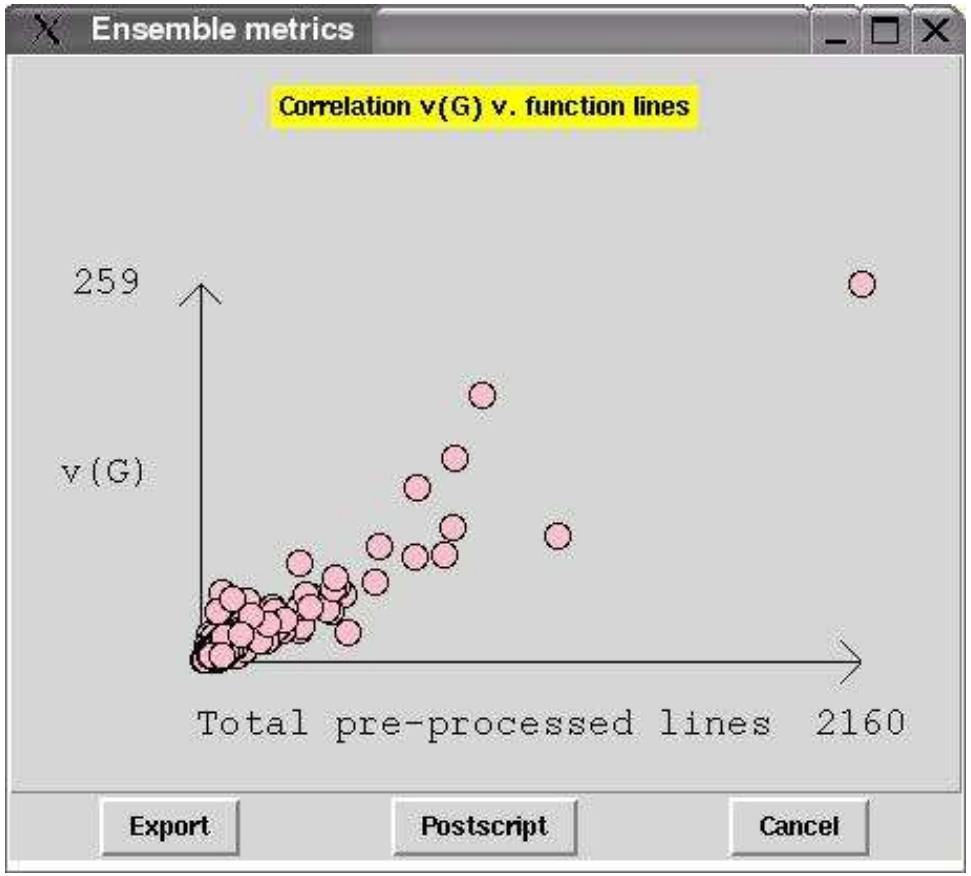
Measuring complexity

- The Holy Grail of static measurement is that we would find some statically measurable property of software which would allow us to predict how much and how severe software failure a system is likely to experience.

Early attempts

- Cyclomatic complexity (McCabe (1976)), (v(G))
  - Very influential. Essentially a count of the number of decisions plus 1. Values greater than 10 were stated to be closely associated with failure. In practice something of a disappointment and very over-hyped unfortunately.
  - Its close association with lines of code mean that lines of code is almost as useful and possibly more so because of the switch statement, (linear in v(G) but not linear in perceived complexity.

Distributions of cyclomatic complexity



All this graph really says is that there is very likely to be decision about every 8 lines

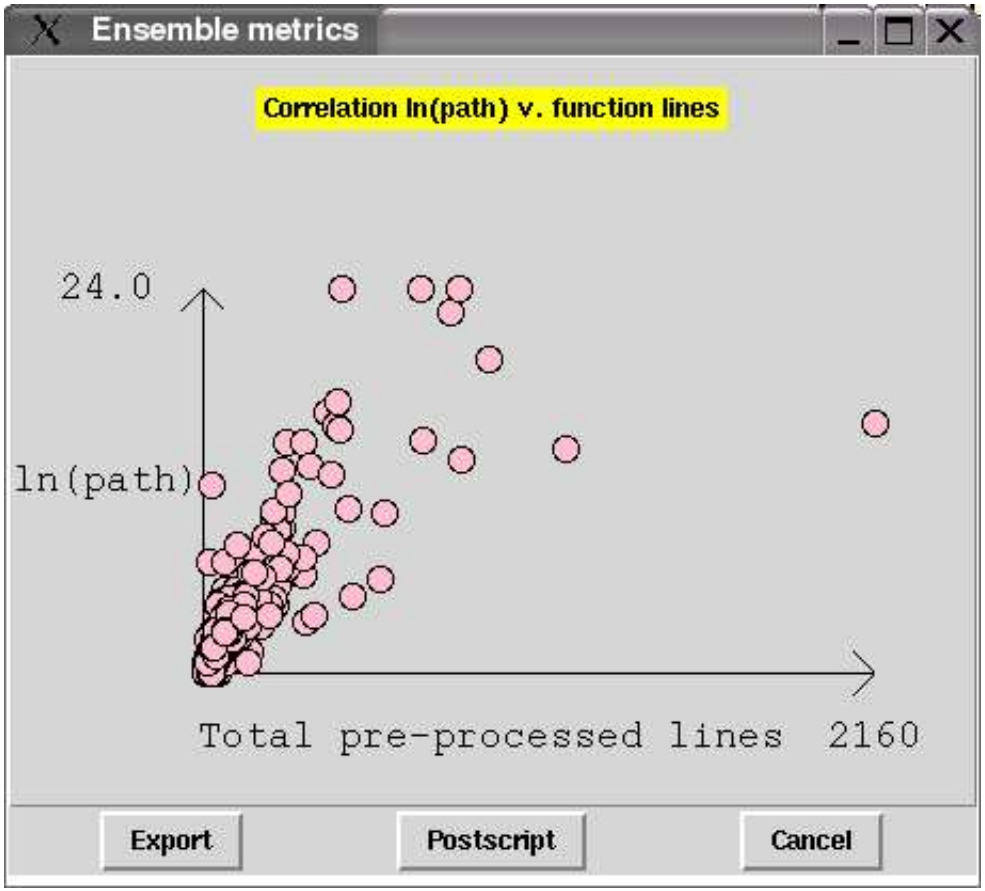
Early attempts

- Halstead ‘software science’, (1983)
  - Very influential and probabilistic arguments based on token occurrences in the language.
  - Case studies vary wildly in their conclusions suggesting that there is no systematic predictive ability.

Early attempts

- Log path count, (Nehmeh 1988, Hatton and Hopkins 1989)
  - Basically a count of the paths through a program assuming each decision is equally likely and then logged to get the numbers down to something reasonable.
  - Appears to be a better predictor of actual failures, certainly than cyclomatic complexity, (NAG library study, also see the study of the Heathrow traffic control system, (Pfleeger and Hatton (1997)).

Distributions of cyclomatic complexity



This has the same behaviour as real defects in that there is a clustering around small components

Measuring implementation quality

- To implement a software system, we must choose one or more programming languages. This means we are working with:-
  - Some kind of compiler or translator for the language
  - Computer hardware to run it on.
  - Some kind of operating system which acts as a go-between.

Measuring implementation quality – programming languages

- The following concepts are involved:-
  - Quality of language definition
    - Completeness (does it deal with all possible constructs)
    - Consistency (Is the logic self-consistent)
    - Agreement (how well were the committee able to agree)
    - Accuracy (are there are mistakes)
  - Quality of compiler implementation
    - Are there defects in the actual implementation from the language definition.
- Consider what happens for two commonly used standardised languages, C and Fortran …

Developments in standard C

- **Programmers should be aware of the following:**
  - 1990 Standardised as ISO C90, 9899: 1990
  - 1994 Normative Addendum 1.
  - 1995 Technical Corrigendum 1. This covers the first 59 Defect Reports in the original document. Some had relevance to end-users.
  - 1996 Technical Corrigendum 2. This covers a further 60 Defect Reports of interest only to compiler writers.
  - 2000. ISO C99 replaced ISO C90 in January, 2000.

Validation Suites (for the bits we agree on)

- **Two suites are used around the world for formal validation of C compilers:**
  - Perennial Suite, (FIPS 160). Around 1300 programs and 220 KLOC originally used by NIST in the US.
  - Plum-Hall Suite

  Note that validation effectively ceased around the world in 1998 !

The naughty parts: sources of information

- **Sources of information on problematic behaviour in C come from two sources:-**
  - The committee's work, (formally identified problem areas).
  - Experience in the world at large through news groups, comp.lang.c, the Obfuscated C competition and so on, (informally identified problem areas).

The effects of disagreement: ISO C90,<< and >>

Syntax:

shift-expression:

additive-expression

shift-expression << additive-expression

shift-expression >> additive-expression

Constraints

Each operand shall have integral type

Semantics

The integral promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width in bits of the promoted left operand, the behaviour is undefined

The result of E1<<E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. If E1 has an unsigned type, the value of the result is E1 multiplied by the quantity, 2^E2 mod ULONG_MAX+1 if E1 is unsigned long, UINT_MAX+1 otherwise. (The constants ULONG_MAX and UINT_MAX are defined in the header <limits.h>.)

The result of E1>>E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a non-negative value, the value of the result is the integral part of the quotient of E1 divided by the quantity, 2^E2. If E1 has a signed type and a negative value, the resulting value is implementation-defined.

The effects of disagreement: ISO C99, << and >>

Syntax:

shift-expression:

additive-expression

shift-expression << additive-expression

shift-expression >> additive-expression

Constraints

Each operand shall have integral type

Semantics

The integral promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width in bits of the promoted left operand, the behaviour is undefined

The result of E1<<E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. If E1 has an unsigned type, the value of the result is E1 *(2^E2 mod MAX representable+1). If E1 has a signed type and nonnegative value, and E1*(2^E2) is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

The result of E1>>E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a non-negative value, the value of the result is the integral part of the quotient of E1 divided by the quantity, 2^E2. If E1 has a signed type and a negative value, the resulting value is implementation-defined.

The full story: formally identified problem areas

- **The following areas of C are problematic:**
  - At standardisation in 1990
    - Unspecified behaviour
    - Undefined behaviour
    - Implementation-defined behaviour
    - Locale-specific behaviour
  - Since standardisation
    - Defect Reports

Unspecified behaviour

- **There are 22 items altogether.**
- **Examples:**
  - Evaluation order and side-effects

    ```
    a = b[i] + i++;
    func(i++,i++);
    ```

  - Representations of floating types
  - Order of # and ## in macro substitution

Undefined behaviour

- **There are 97 items altogether.**
- **Examples:**
  - The value of a void expression is used.

    file1.c: void func(blah)

    file2.c: a = func(blah)
  - An arithmetic operation is invalid.

    a = b / 0.;
  - The value of an uninitialised variable is used.

Implementation-defined behaviour

- **There are 76 items altogether, split into 14 sub-groups.**
- **Examples:**
  - The result of bit operations on signed integers.
  - The result of a right shift of a -ve signed integer.

    ```
    res = signed_i >> 6;
    ```
  - The result of casting a pointer to an integer.

Locale specific behaviour

There are 6 items altogether but as time goes by items move from implementation-defined behaviour to locale-specific behaviour.

- **Examples:**
  - The direction of printing.
  - The decimal-point character
  - The collating sequence of the execution character set.

Defect Reports 1

- **These were originally known as Interpretation Requests (ANSI term).**

  Their scope is as follows:
  - An item in the Standard is not understood.
  - An item is understood, but the wording is controversial.
  - The Standard does not say anything.
  - The Standard is not consistent.

Defect Reports 2

- **Examples:**
  - Pre-processor ambiguity X3J11/90-056

```
#define f(a,b) a+b
#if f(1,
        2)
#endif
```

Where does the #if end ? The committee decided that a line-splice character '\' must be used to allow the expression to be completed.

Defect Reports in other languages

Language	Pascal	Extended Pascal	C	Ada
Number of Interpretation Requests	1	~80	119	~1400

Informally identified problem areas

- **25 years of use of C has thrown up around 400 more problem code fragments.**

  Some examples follow …

Empirically determined misbehaviour 1

- **A category of well-defined but frequently abused issues.**

  Examples:

  – Returning the address of a local from a function.

  – Assignment in a conditional

    ```
    if ( a = b )
    ```

  – Relational equality in an assignment

    ```
    a == b;
    ```

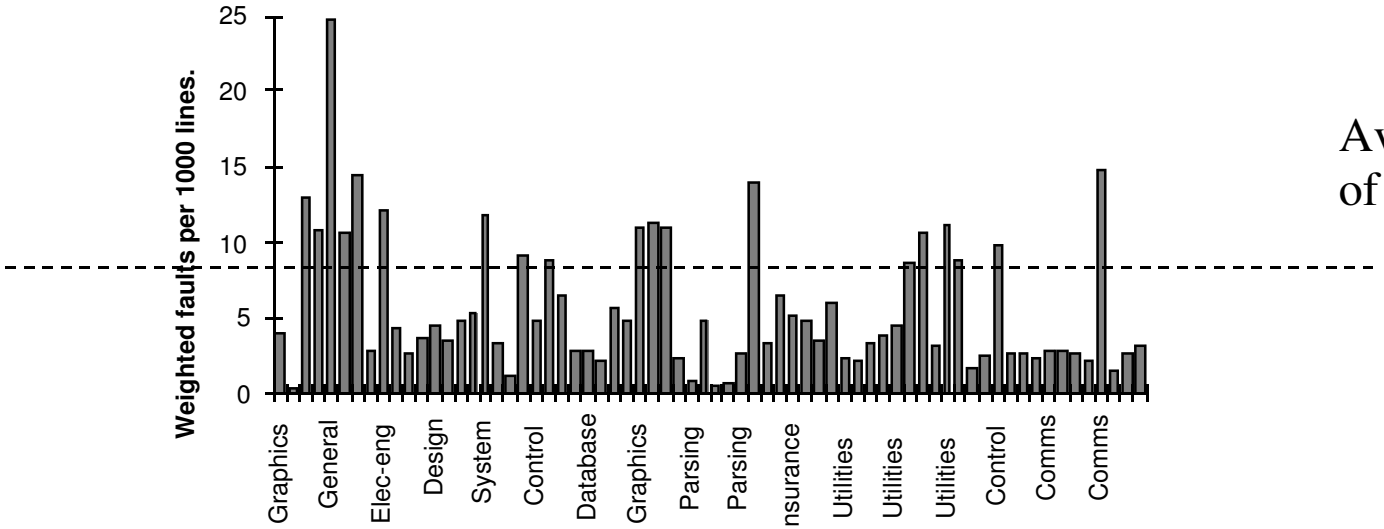
  – Spare semi-colons:

    ```
    if ( a == b );    { ... }
    ```

Empirically determined misbehaviour 2

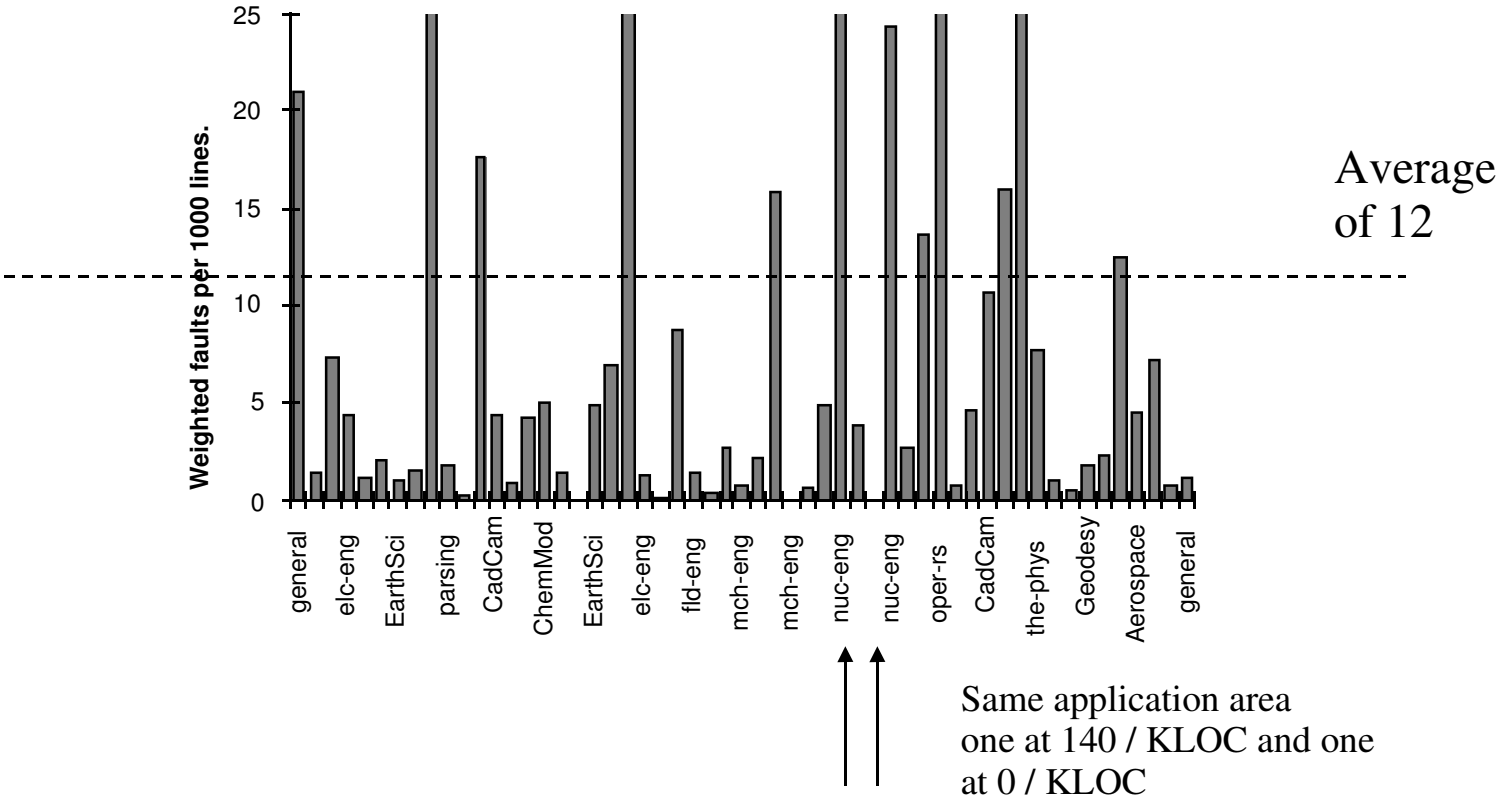
- y = x/*p /* what does this do ? */;
- int i,j = 0; /* i is not initialised */
- If (a == 0) return /* missing semi-colon */
  logrec = 1;
- Switch(b) {
  case 1: … break;
  defualt: … break;
  }
- #define abs(x) (x>0)?x:-x
  y = abs(a-b); /* ? */
- #define f (x) ((x)-1) /* ? */

Fault frequencies in C applications

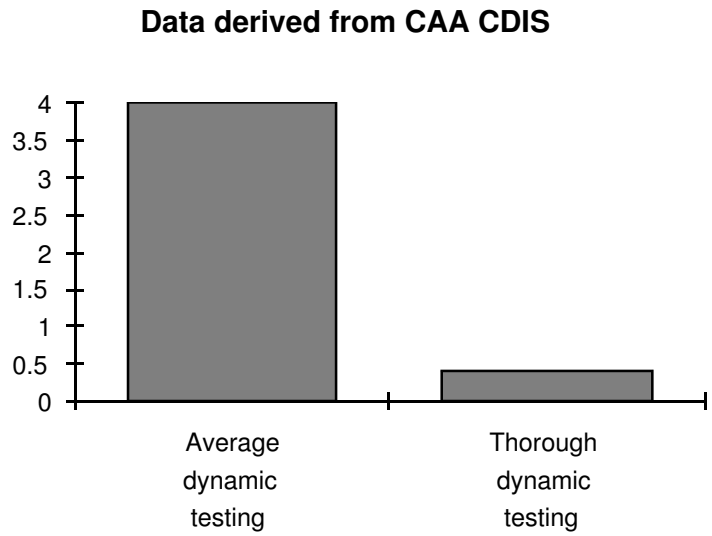


Average of 8

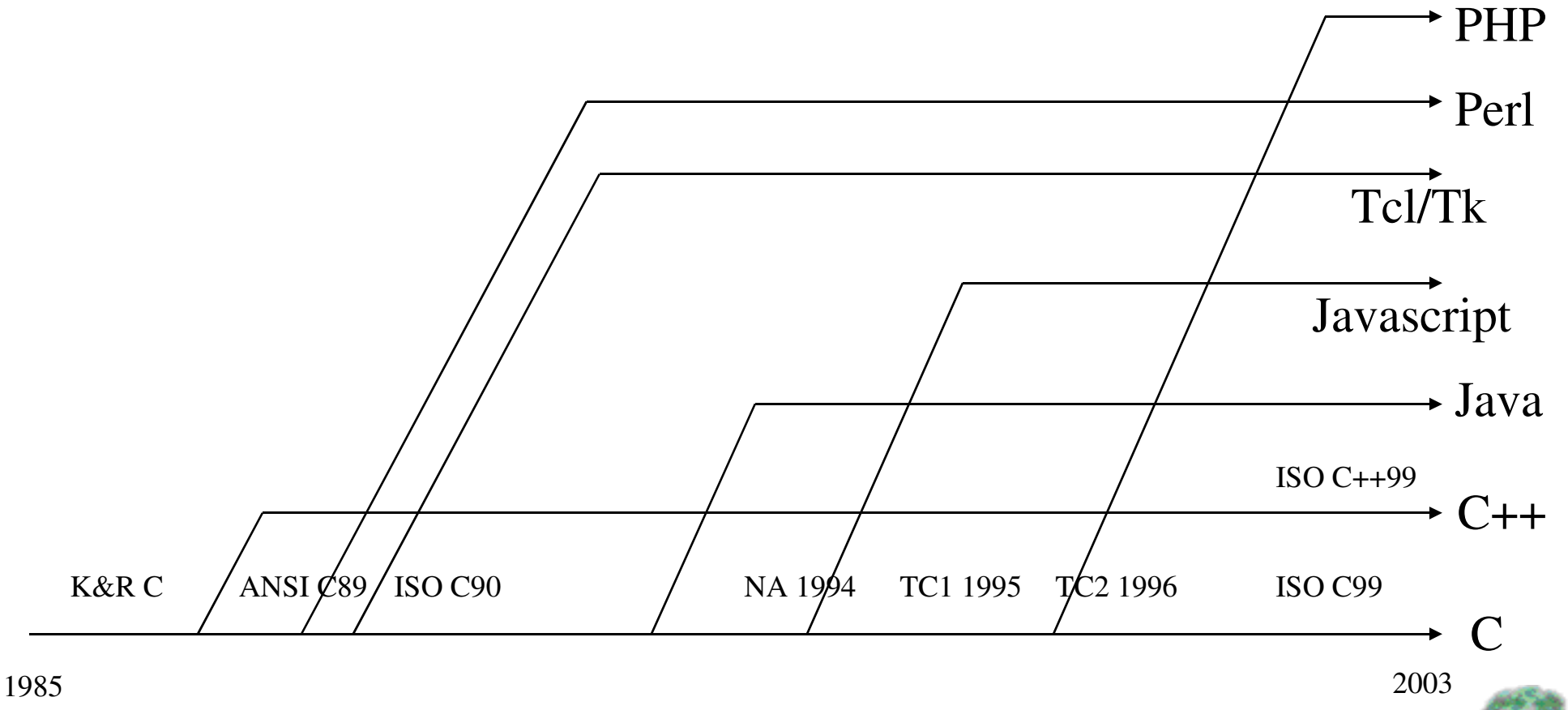
Fault frequencies in Fortran 77 applications



Do statically detectable faults fail ?



A slight digression: Inheritance and the C family tree



Measuring implementation quality – the arithmetic

- It would be nice to think after 50 years that computers could finally add up without too much trouble.

Floating Point validation

- **Arguably the best arithmetic validation suite of any is *paranoia*. This exercises:**
  - Adequacies of guard digits, rounding of arithmetic, underflow behaviour, representation of infinity, and consistency of comparisons.
  - Compatibility with IEEE 754 and 854
  - Four grades of problem: FAIL (worst), SERIOUS DEFECT, DEFECT and FLAW.

Paranoia example

- **Paranoia examples**
  - Sparc Classic, 1993. Full compliance with IEEE 754.
  - 266 MHz. Pentium II, Linux 6.0, 8/1999 gcc 2.7.2
    - FAIL: (1-U1)-1/2 < 1/2, where U1 is smallest fp separation
    - DEFECT: No guard digit in *, / and -
    - FLAW: Incorrect rounding on *, /
    - DEFECT: sqrt(X) not monotonic for X ~ 2.00000e+00
    - SERIOUS DEFECT: sqrt gets too many last digits wrong
    - DEFECT: Z^i error may invalidate financial calculations
    - and so on ...
  - PowerPC, gcc 2.7.2, Linux 5.0, one flaw.

Paranoia in embedded control systems (ESP)

- **The original paranoia was not easily run on embedded control systems**
- ***Embedded system paranoia* (ESP) appeared in October 2003, (freely downloadable from http://www.leshatton.org/ -> Benchmarking)**
  - 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)

ESP output examples

- **IAR MSP430, (simulator+fast math) (November 2003)**
  - 6 defects, 4 flaws: score:-

```
 Excellent
Very good
Good
Acceptable
Unacceptable
Broken
```

- **IAR MSP430, (October 2003)**
  - 6 failures, 7 serious defects, 12 defects, 1 flaw: score:-

```
 Excellent
Very good
Good
Acceptable
Unacceptable
Broken
```

ESP examples

Compiler / chip	Environ-ment	Date	Failures	Serious Defects	Defects	Flaws
TI-OMAP (ARM9)	Target	26-12-03	0	0	6	4
STAR-12	Target	19-09-03	6	7	12	1
IAR MSP430	sim.+ fast math	01-11-03	2	1	9	4
IAR MSP430	sim. + IEEE	01-11-03	0	0	6	1
IAR MSP430	sim.+ EC++	01-11-03	0	0	0	1

Measuring implementation quality – the OS environment

- With the exception of POSIX compliance, measurement of OS implementation quality is essentially absent.

Conclusions

- Software measurement is exceptionally immature and certainly does not satisfy the requirements of an engineering discipline.
  - We do not define the dependent variables consistently
  - We do not define the independent variables consistently
  - We do not perform calibrated or even repeatable experiments in most cases
  - In many fields of IT we do not measure anything at all.
- The formal verification of implementation quality has disappeared.

Useful links

- **On validation:-**
  - http://www.peren.com/, (Perennial Suite)
  - http://www.plumhall.com/, (Plum-Hall Suite)
  - http://www.nist.gov/ (NIST)