Citation for published version
Hatton, Leslie (2000) Balancing static and dynamic testing: some observations from measurement. UNSPECIFIED.

DOI

Link to record in KAR
https://kar.kent.ac.uk/21892/

Document Version
UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version. Users are advised to check http://kar.kent.ac.uk for the status of the paper. Users should always cite the published version of record.

Enquiries
For any further enquiries regarding the licence status of this document, please contact: researchsupport@kent.ac.uk
If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at http://kar.kent.ac.uk/contact.html
2000-
"Balancing static and dynamic testing: Some observations from measurement"

by

Les Hatton

Oakwood Computing, Surrey, U.K. and the Computing Laboratory, University of Kent
lesh@oakcomp.co.uk

Version 1.2: 09/Mar/2000

©Copyright, L.Hatton, 2000-
Overview

- Static v. dynamic testing
- Forensic work: patterns in failure
- Wallowing in data
Control Process feedback - the essence of engineering improvement

If you want to improve reliability, measure and analyse failures.
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
- A defect or bug is a generic term for either faults which fail or faults which do not
- Fault density is the number of faults divided by the number of lines of code
Preparing the ground

Note that the causal relationship between fault and failure differs in some standards:

- IEEE + other sources:
  \[
  \text{error} \rightarrow \text{fault} \rightarrow \text{failure}
  \]

- IEC 61508, (formerly IEC SC 65A):
  \[
  \text{fault} \rightarrow \text{error} \rightarrow \text{failure}
  \]
Preparing the ground

The basis of measurement is to define the dependent and independent variables

- Independent variables
  - LOC (line of code)
  - Time
  - Function points

- Dependent variables
  - Defect type
  - Defect severity
What is a line of code?

Correlation between two measures of line of code in systems written in C. The two measures are executable lines and total number of pre-processed lines, Hatton (1995).
Fault density is a function of time

Fault density depends on how much the system has been used, (c.f. HP)
Where and how do defects occur historically?

All faults

Those faults which fail
Mean time to fail in Adams (1984)
Cost v. detection point

Cost of fixing defects

Embedded systems tend to follow the high curve. Data from Boehm, (1981) and many others. Note that curve kicks only around coding stage.
Overview

- Static v. dynamic testing
- Forensic work: patterns in failure
- Wallowing in data
Patterns in failure

There are two complicating factors in the forensic analysis of software failure

- Exponentially increasing complexity
- Chaotic behaviour
Exponentially increasing complexity

The amount of software in consumer electronic products is currently doubling about every 18 months.

- Line-scan TVs have ~250,000 lines of C.
- There are around 200,000 lines of C in a car.
- Most consumer devices, washing-machines and so on have a few K of software.
- The Airbus A340 and Boeing 777 are totally dependent on software.
Chaotic behaviour

AT & T Jan, Jan 15, 1990:

- Single misplaced line of C in 3 million lines bypassed network error-recovery code
- For 9 hours, millions of long-distance callers just heard message “all circuits are busy”
- Reported $1.1 billion loss
Anatomy of a $1billion bug

... switch( message ) {
  case INCOMING_MESSAGE:
    if ( sending_switch == OUT_OF_SERVICE ) {
      if ( ring_write_buffer == EMPTY )
        send_in_service_to_smm(3B);
      else
        break; /* Whoops ! */
    }
    process_incoming_message(); /* skipped */
    break;

... do_optional_database_work();
...
Cars too ...:

- 22/July/1999. General Motors has to recall 3.5 million vehicles because of a software defect. Stopping distances were extended by 15-20 metres.

- Federal investigators received almost 11,000 complaints as well reports of 2,111 crashes and 293 injuries.

- Recall costs? (An exercise for the reader).
The PC picture ...

Mean Time Between Failures of various operating systems
Useful links

- On software failure:-
  - http://www.rvs.uni-bielefeld.de/publications, (aircraft)
  - http://www.bugnet.com/, (PC)
  - http://www.oakcomp.co.uk/TechPub.html, (general failure)
Overview

- Static v. dynamic testing
- Forensic work: patterns in failure
- Wallowing in data
Where and how do defects occur historically?

Looking for properties of defects

- Defects tend to cluster, (in one case 47% of defects in 4% of modules in IBM’s S/370 OS)
- The earlier you find them, the cheaper you find them
Where and how do defects occur historically?

Where you find one, you find more, (Pfleeger, (1998))

Defect clustering

Component

v. 1.2, 09/Mar/2000 , (slide 1 - 22). Not to be copied without permission from copyright holder.
Where and how do defects occur historically?

Where you find one, you find more. The effect is even more emphatic when you normalise against lines of code. (Hatton (1998), Pfleeger, (1998))

Defect density clustering

v. 1.2, 09/Mar/2000 , (slide 1 - 23). Not to be copied without permission from copyright holder.
Where and how do defects occur historically?

The following slides show distributions of faults and failures from a number of case studies, each with an introduction and a conclusion.
Where and how do defects occur historically?

Defect clustering in systems

Introduction:
The following data shows how defects cluster in systems as a function of module complexity.

Source:
Failures and component size, (new and changed)

Data from an OS study at Siemens (1993)
What happens for big components?

![Graph showing average size in statements over time for different components and predictions.](image)
Failure density and component size

Comparison of Ada and assembler, Hatton (1997)
Failure density and component size

Defect density v. C function size

Data from the GNU indent program, Swanton (1996)
The defect density $U$ curve

For Ada, various assembler, C, C++, Fortran, Pascal and PL/M systems:

Defects per KLOC

Average component complexity
What happens if you intervene at the top end?

There are two ways of restricting the appearance of complex components:-

- Design / Test intervention whereby test plans are required to evolve in parallel with the component
- Complexity metric limits
Complexity testing generally includes the following:-

- Measurement of complexity values such as lines of code (LOC), cyclomatic or path complexity
- Identification of the worst 10% of a population
- Using the known properties of the U curve to exclude this 10%
The defect density $U$ curve - invasive truncation

In those systems where excessive complexity has been restricted:-

![Graph showing the relationship between defects per KLOC and average component complexity](graph.png)
Complexity measures:-

- Cyclomatic complexity is a count of the number of decisions plus 1, (in an if else, don’t count the else. In a switch, don’t count the default).
- The path count is calculated by assuming that every decision is independent. Sequential blocks multiply and parallel blocks add.
Cyclomatic complexity distributions

Note the effectiveness of complexity limiting here (lower curve) in excluding the dangerous upper end in this experiment.
The same complexity limiting is equally successful at controlling path complexity, improving dynamic testability dramatically.
Complexity measurement limiting

Complexity limiting notes:-

– It doesn’t seem to matter which complexity metric you use to do this, they are currently very crude
– It should be used at either end because of the U-curve effect.
Where and how do defects occur historically?

Defect clustering in systems

Defects are not spread equally as a function of component size. They tend to cluster.

Conclusion:

- Use defect clustering to guide inspection and testing strategies
- Use complexity metric limits
Where and how do defects occur historically?

**Statically detectable fault**

**Introduction:**

The following slides show the distribution of statically detectable inconsistencies and widely-known faults in C and Fortran 77.

These were measured using purpose built tools exploiting the knowledge base of such behaviour.

**Source:**

Hatton (1995)
The logical argument

- We will establish the following chain of reasoning:-
  - Known fault modes exist in programming languages
  - They appear regularly in user’s code
  - These faults fail with a certain frequency
Sources of information

Sources of information on problematic behaviour in languages come from two sources:-

- The committee’s work, (formally identified problem areas). Approximately 300 items.
- Experience in the world at large through news groups, comp.lang.c, the Obfuscated C competition and so on, (informally identified problem areas). Approximately 400 items.
Problems with programming languages

The need for subsetting programming languages

Scope of Standard language
Extensions
Subset of well-defined features
Subset of allowed features
Formally identified problem areas

Let us consider C. The following areas of C are problematic:

- At standardisation in 1990 (197 items)
  - Unspecified behaviour
  - Undefined behaviour
  - Implementation-defined behaviour
  - Locale-specific behaviour

- Since standardisation (119 items)
  - Defect Reports
Examples reported by user community

- There are approximately 400 known. They are usually well-defined but misleading.

Examples:
- Returning the address of a local from a function.
- Assignment in a conditional
  ```c
  if ( a = b )
  ```
- Relational equality in an assignment
  ```c
  a == b;
  ```
- Spare semi-colons:
  ```c
  if ( a == b ); { ... }
  ```
Fault frequencies in C applications

Data like this is extractable using tools such as the Safer C Toolset, (http://www.oakcomp.co.uk)
Fault frequencies in Fortran 77 applications

Average of 12

Same application area one at 140 / KLOC and one at 0 / KLOC
Where and how do defects occur historically?

This study shows that statically detectable faults do in fact fail during the life-cycle of the software.
Where and how do defects occur historically?

Conclusions on safer subsetting:

– We can prove the following:
  
  - There is a class of defect in programming languages which to a significant extent is statically detectable, widely reported and entirely avoidable
  
  - This class of defect evades conventional testing to the extent of around 8 residual defects per 1000 lines of code
  
  - A significant percentage of this class of defect fails during the life-cycle of the code but we are not able to predict which faults fail, so we must remove them all.

– Engineer education and tool support is crucial to the control of this class of defect.
Do languages improve with time?

- Things get worse with time. The following areas of C are problematic because the committee could not agree:
  - At standardisation in 1990 (197 items)
  - At re-standardisation in 1999 (366 items)

- By comparison, C++99 contains the words:-
  - Undefined, 1825 times
  - Unspecified, 1259 times.
Why languages can’t improve

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.
Where and how do defects occur historically?

**Statically detectable fault**

Static analysis suffers from a noise problem

- When sometimes it's a fault and sometimes not, for example:
  
  ```
  if ( a = b )
  ```
  
  instead of
  
  ```
  if ( a == b )
  ```

- In this case, if we warn of all transgressions those statements which are OK will tend to hide those which are not from the programmer. The ‘signal’ is hidden by the noise.

- Some form of filtering is necessary, to maximise the likelihood of positive detection, for example a safer subset standard.
**Where and how do defects occur historically?**

**Statically detectable fault**

We do not know in advance which statically detectable faults will fail, but on average a significant percentage will.

**Conclusions:**

- Source code should not be released with any statically detectable fault
- Learn about the fault modes of your language
- Beware of the static noise problem
Conclusions

The view from data:-

- Static testing v. dynamic testing
  - Efficient static testing via inspections with semi-automated tool support has a dramatic beneficial effect on software reliability and production cost

- Tool support
  - Automation should and can support:-
    - The best static fault detection possible
    - Education of engineers on difficult language areas
    - Manual code inspections
    - Dynamic checking
    - Simple complexity control
More information ...

For more information on safer subsets, static testing, downloadable technical publications and tools and other links, you are invited to browse our site:-

http://www.oakcomp.co.uk/
Bibliography

- Gilb, T. & Graham D. (1993) Software Inspection, Addison-Wesley
Bibliography

Bibliography