

SOFTWARE TECHNOLOGIES

The Power of 10: Rules for Developing Safety-Critical Code

Gerard J. Holzmann
NASA/JPL Laboratory for Reliable Software

Adhering to a set of 10 verifiable coding rules can make the analysis of critical software components more reliable.

Most serious software development projects use coding guidelines. These guidelines are meant to define the ground rules for the software to be written: how it should be structured and which language features should and should not be used. Curiously, there is little consensus on what a good coding standard is.

Among the many coding guidelines that have been written, there are remarkably few patterns to discern, except that each new document tends to be longer than the one before it. The result is that most existing guidelines contain well over 100 rules, sometimes with questionable justification. Some rules, especially those that try to stipulate the use of white space in programs, may have been introduced by personal preference; others are meant to prevent very specific and unlikely types of errors from earlier coding efforts within the same organization.

Not surprisingly, the existing coding guidelines tend to have little effect on what developers actually do when they write code. The most dooming aspect of many of the guidelines is that they rarely allow for comprehensive tool-based compliance checks. Tool-based checks are important because manually reviewing the hundreds of thousands of lines of code that are written for larger applications is often infeasible.

Existing coding guidelines therefore offer limited benefit, even for critical applications. A verifiable set of well-chosen coding rules could, however, assist in analyzing critical software components for properties that go well beyond compliance with the set of rules itself. To be effective, though, the set of rules must be small, and it must be clear enough that users can easily understand and remember it. In addition, the rules must be specific enough that users can check them thoroughly and mechanically.

To put an upper bound on the number of rules, I will argue that restricting the set to no more than 10 rules will provide an effective guideline. Although such a small set of rules cannot be all-encompassing, following it can achieve measurable effects on software reliability and verifiability.

To support strong checking, the rules I will propose are somewhat strict—some might even say draconian. The tradeoff, though, should be clear. When it really counts, especially in the development of safety-critical code, working within stricter limits can be worth the extra effort. In return, it should be possible to demonstrate more convincingly that critical software will work as intended.

SAFETY-CRITICAL CODING RULES

The choice of language for safety-critical code is in itself a key consideration. At many organizations, JPL included, developers write most code in C. With its long history, there is extensive tool support for this language, including strong source code analyzers, logic model extractors, metrics tools, debuggers, test-support tools, and a choice of mature, stable compilers. For this reason, C is also the target of the majority of existing coding guidelines. For fairly pragmatic reasons, then, the following 10 rules primarily target C and attempt to optimize the ability to more thoroughly check the reliability of critical applications written in C.

These rules might prove to be beneficial, especially if the small number means that developers will actually adhere to them.

Rule 1: Restrict all code to very simple control flow constructs—do not use goto statements, setjmp or longjmp constructs, or direct or indirect recursion.

Rationale: Simpler control flow translates into stronger capabilities for analysis and often results in improved code clarity. Banishing recursion is perhaps the biggest surprise here. Avoiding recursion results in having an acyclic function call graph, which code

June 2006 93

The Power of Ten

rules for safety critical coding

Gerard J. Holzmann

nimble research
oxnard ca 93035



software safety related standards

IEEE 1059	Guide for Software Verification and Validation Plans
IEEE 1228	Software Safety Plans
BS EN 50128:2001	Railway applications. Communications, signaling and processing systems Software for railway control and protection systems
ANSI/UL 1998	Software in Programmable Components, Second Edition
IEC 60880	Software for Computers in the Safety Systems of Nuclear Power Stations
IEC 60950-1	Safety of Information Technology Equipment
BSI IEC 61508-1-5	Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems
RTCA DO-178B/ED-12B	Software Considerations in Airborne Systems and Equipment Certification
IEE SEMSPLC 8	Guidelines: Safety-Related Application Software for Programmable Logic Controller
AECL CE-1001-STD REV.2	Standard for Software Engineering of Safety Critical Software
ANS ANSI/IEEE 7.4.3.2	Application Criteria for Programmable Digital Computer Systems in Safety Systems of Nuclear Power Generating Stations
ANSI/AAMI SW68	Medical device software - Software life cycle processes
DEF 00-56-1/2	Safety Management Requirements for Defense Systems
SAE AS9006	Aerospace Software Supplement for AS9100A
DoD MIL-STD-882D	Standard Practice for System Safety Program Requirements (Jan. 1996)
EIA SEB6-A	System Safety Engineering in Software Development
IEEE 1012	Software Verification and Validation Plans
IEEE 1074	Standard for Developing Software Life Cycle Processes
IEEE 16085	Software Life Cycle Processes - Risk Management
ISO 17894	Ships and marine technology - Computer applications - General principles for the development and use of programmable electronic systems in marine applications
ISO/IEC 12207-0,1,2	Information Technology-Software Life Cycle Processes (IEEE/EIA 12207, twice amended, 1995-1998)
ANS ANSI 10.4	Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry
FDA Guidance Doc. 4	General Principles of Software Validation - Final Guidance for Industry and FDA Staff
FDA Guidance Doc. 2	Guidance for Industry, FDA Reviewers and Compliance on Off-the-Shelf Software Use in Medical Devices
ASTM E 1578	Standard Guide for Laboratory Information Management Systems
IEC 61069-1	Industrial-Process Measurement and Control-Evaluation of System Assessment
BSI BS-7738	Specification for Information Systems Products Using Structured Systems Analysis and Design
DoD DI-SAFT-80100A	System Safety Program Plan (Jan 1993)
IEEE STD 982.2-1987	Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software.
NASA-GB-A302	Formal Methods Specification and Verification Guidebook for Software and Computer Systems.
NASA-GB-A201	NASA Software Assurance Guidebook, NASA Office of Safety and Mission Assurance, 1989.
NASA-GB-8719.13B	NASA Software Safety Standard, NASA Technical Standard, July. 2004.
MIL-STD-498	Military Standard Software Development and Documentation, DoD, Dec. 1994.
DOD-STD-2167A	Defense System Software Development, Feb. 1988.
DOD-STD-7935A	Automated Information Systems Documentation Standards, Oct. 1988
GSFC SEL-81-305	Recommended approach to software development, Rev. 3, June 1992

"the nice thing about standards is that there are so many of them to choose from" (Andy Tanenbaum)



coding standards

- when I joined nasa/jpl in 2003, there were as many coding standards as there were software projects
- each new mission defined their own new coding standard, based on the preferences of the *cog-e* (cognizant engineer)
- most contained hundreds of un-checkable, often dubious rules (e.g., regulating the use of white space), resulting in most of the rules being silently ignored
- *“most read like mediaeval guides to alchemy, based on folklore instead of measurable benefit”*



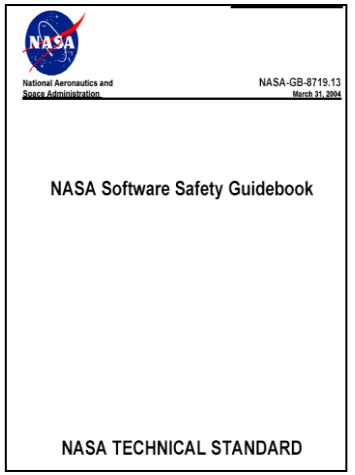
strong risk awareness

Safety is usually not on anyone's mind when they select a compiler, editor, or other tool, but it should be. **All software must be considered potentially flawed.** This isn't a cause for panic, however. Understanding how the software tool, library, operating system, or other element **could** fail is important in guarding against such a failure. Knowing that a potential failure could impact the safety of the system is the most important aspect. Don't become complacent when safety is involved!

NASA-GB-8719.13, 3/31/04 p.49

If the inherent reliability of software cannot be accurately measured or predicted, and **most software designs cannot be exhaustively tested**, the level of effort required to meet safety goals must be determined using other characteristics of the system.

NASA-GB-8719.13, 3/31/04 p.35


National Aeronautics and Space Administration

NASA-GB-8719.13
March 31, 2004

NASA Software Safety Guidebook

NASA TECHNICAL STANDARD

"Safety must be designed into software *from the beginning*."

D-560 Rev. C 3/9/1999

JPL Standard for Systems Safety


D-560, REV. C

JPL STANDARD FOR SYSTEMS SAFETY

* PRINTED COPIES OF THIS DOCUMENT MAY NOT BE CURRENT AND SHOULD NOT BE RELIED ON FOR OFFICIAL PURPOSES. THE CURRENT VERSION IS IN THE DMIE INFORMATION SYSTEM AT HTTP://DMIE *

Prepared by: James M. Lumadue 3/9/99

Reviewed by: List on File in Systems Safety Office

Approved by: Larry D. Condon, Manager Systems Safety Office 3/9/99

March, 1999

National Aeronautics and Space Administration

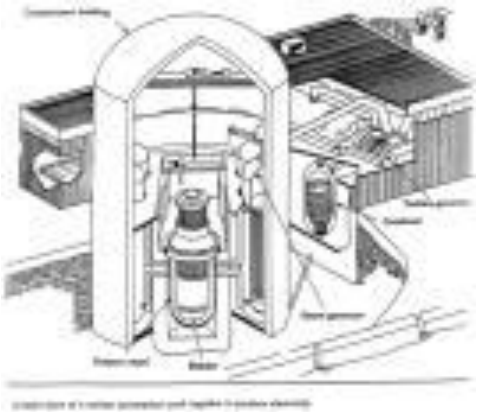
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL D-560, Rev. C

basics

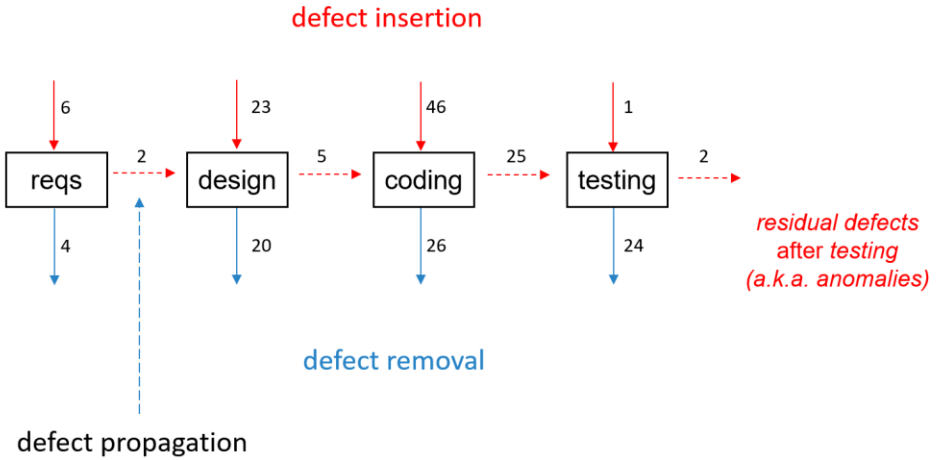
- all non-trivial software applications have residual defects
  - it is generally *impossible* to achieve 100% coverage in testing
  - it is *especially* hard to test multi-threaded software:
    - limited control (cannot control scheduling decisions or event orders)
    - limited observation (cannot observe all internal events of interest)
- the average *residual software defect ratio after testing* for non-critical software applications
  - 1-10 residual defects per KNCSL (1,000 lines of non-comment source)
- for *mission or safety critical* applications the target is lower
  - $\leq$ 0.1 residual defects per KNCSL (e.g., Space Shuttle)
  - with special scrutiny of potentially high-severity defects
    - risk analysis, mitigation and containment strategies
  - these objectives *cannot* be achieved through testing alone
- the effect of even *minor* software defects can be significant
  - misplaced statements, race conditions, synchronization errors
  - unintended *fault propagation* is a significant source of failure





the importance of collecting metrics

- requirements errors
  - missing or ambiguous requirements
- design errors
  - misinterpretation of requirements
  - a module misinterpreting the needs of or guarantees provided by other modules
- coding errors
  - misunderstood language concepts
  - misuse of language features
  - copy/paste errors
  - typos
- errors introduced in testing…
  - e.g., changing timeout values for a test
- propagation of errors
  - errors spreading beyond the module in which they occur
  - unwieldy dependency chains

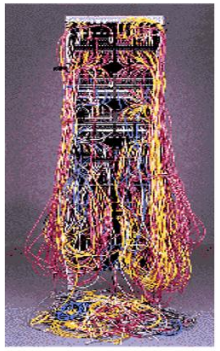


software reliability is not just about finding and fixing defects

every software defect also reveals a testing and a development process defect (quality gates)

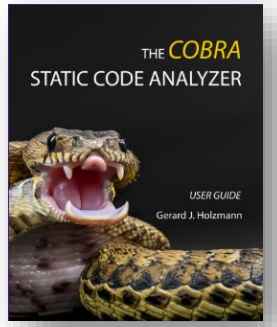
defect prevention strategies

- simplicity enhances verifiability & maintainability
  - bounds on software complexity
- language restrictions
  - avoiding language features that are frequently misunderstood or that contribute disproportionally to in-flight anomalies
- stronger guidelines for structuring code
  - modularity, encapsulation rules, data hiding, defensive coding, assertion density
  - standardized rules for thread interaction
  - robust interface definitions
- automation: use static code checkers, code review tools
  - a no brainer (*Cobra*, Coverity, Semmle, Codesonar)
  - use tools to check for defects and compliance with rules in all phases of development: during *coding, testing, code review, and in the build process*





Q: which of these systems would you rather fix when it breaks?



my favorite tool: codescrub.com

cataloguing software anomalies in space missions (2006)

	mission	basic coding	design (algorithmic)	memory use	thread use	code reuse	fault protection
1962	Mariner 1						
1963	Mercury						
1968	Apollo 8						
1969	Apollo 11						
1971	Eole 1						
1977	Voyager						
1981	Shuttle						
1982	Viking 1						
1988	Phobos						
1994	Clementine						
1996	Ariane 5						
1997	Pathfinder						
1999	MCO						
2000	MPL						
2004	MER						
2006	DART						
2006	MRO						

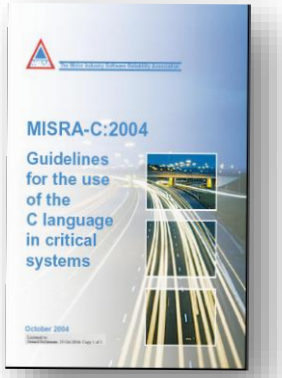
mission failure due to software problem

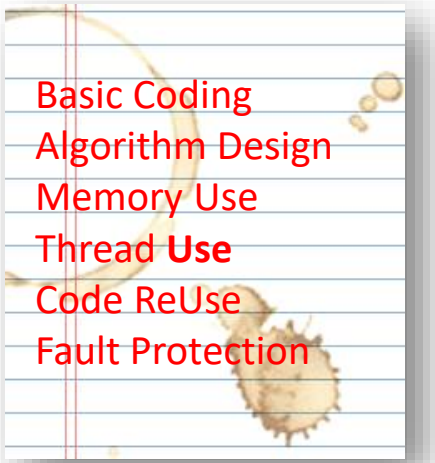
non-fatal software anomaly

observation: a small number of issues tends to cause a relatively large number of problems

example

- heap memory use
  - accidental use of freed memory
  - overrunning allocated blocks
  - running out of heap memory
  - quadratic performance of some allocators
- the use of dynamic memory allocation *after* initialization is prohibited by most standards for safety critical code
  - including in the MISRA-C Coding Guidelines for engine control software in cars
  - and in the Power of Ten ruleset





we can't prevent all errors,
but at least we can try to address these six issues

survey of responses to MISRA-C:2004 rules at JPL

Laboratory for Reliable Software

JPL Feedback on the MISRA 2004 Coding Guidelines (33 reviewers)

Collected by the NASA/JPL Laboratory for Reliable Software

121 required rules, 20 advisory rules

www.misra.org.uk

Very Popular	Most popular with reviewers.
Popular	Popular with reviewers.
Very Unpopular	Most unpopular with reviewers.
Unpopular	Unpopular with reviewers.
Controversial	Mixed positive and negative comments from reviewers.
Confusing	Reviewers were confused or had questions.
Most Comments	These rules received the most comments from reviewers.



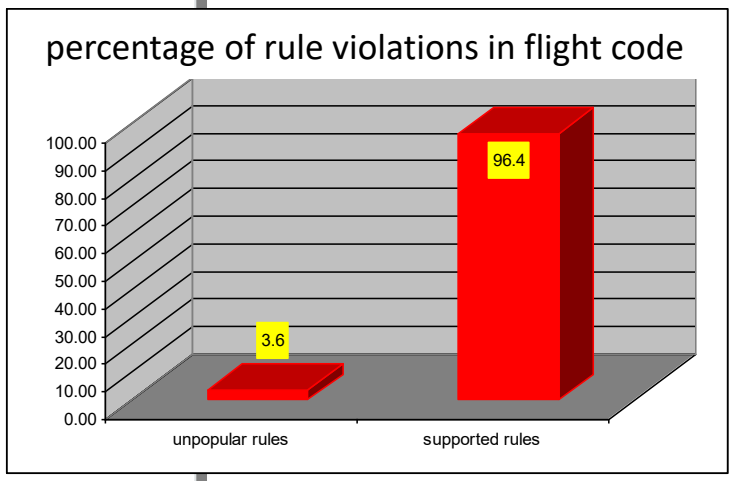
Rules with General Support Only (the 2 top violation categories are here)
7 of 15 rules checked from this category have zero violations (46%):

Rule	Description	Feedback Category
4.1	undefined escape sequence	Supported
4.2	using trigraph	Supported
5.1	**long identifier name**	**Supported**
(19.1)	**#include preceded by code**	**Supported**
(19.2)	non-standard char in header name	Supported
19.5	#define or #undef in block scope	Supported
19.7	function-like macro iso function	Supported (advisory rule)
19.8	macro call with insufficient args	Supported
19.9	macro arg matches directive name	Supported
19.10	formal param not enclosed in ()	Supported
19.11	undefined macro name	Supported
19.12	more than one ## in macro	Supported
19.13	use of ## in macro	Supported
19.15	include file processed twice	Popular
19.17	unbalanced #if..#endif directives	Popular

Unpopular Rules Only
9 of 20 rules checked from this category have zero violations (45%):

Rule	Description	Feedback Category
2.2	C++ style comment	Unpopular
2.3	nested /* comment	Controversial
7.1	octal constant	Controversial
14.4	uses goto	Controversial
14.5	uses continue	Controversial
16.1	function with var-length arg list	Unpopular
18.4	uses union	Very Unpopular
19.3	#include needing preprocessing	Controversial
19.4	possibly non-compliant macro type	Controversial
19.6	#undef	Controversial
19.14	malformed defined	Controversial
20.4	malloc.h	Very Unpopular
20.5	use of 'errno'	Unpopular
20.6	offsetof	Controversial
20.7	use of 'setjmp'	Controversial
20.8	signal.h	Unpopular
20.9	stdio.h	Unpopular
20.10	use of 'atof', 'atoi', 'atol'	Confusing
20.11	use of 'abort', 'exit'	Unpopular
20.12	time.h	Unpopular

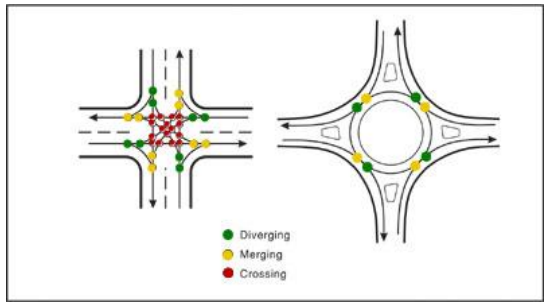
observation:
an unchecked rule is not followed,
even if its rationale is supported



to be effective, compliance with a coding rule must be *checkable*

not measurable

- there is a difference between
  - *when you get to an intersection: be very careful*



- and

measurable

1. stop at a red light
2. wear a seat belt
3. don't drive faster than 55 mph





rules are most useful if they are sparse, clear, are demonstrably correlate with risk, and are checkable with tools

"measure what is measurable and make measurable what is not so"

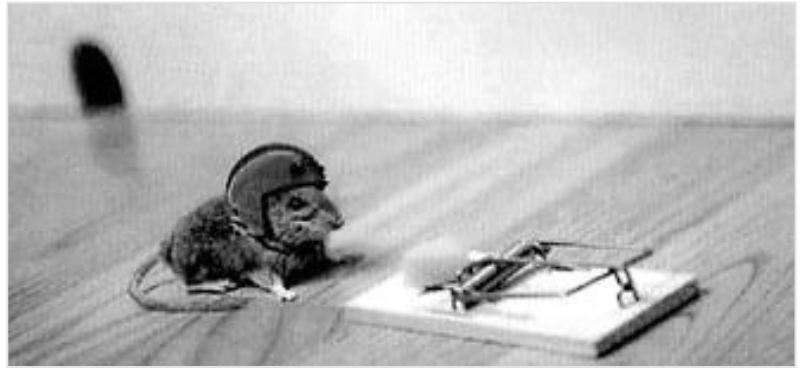


Galileo (1564-1642)

simple principles

protect yourself from doom
- use assertions
- practice defensive coding

use restraint
- small functions
- simple control flow
- bounded loops
- limited data scope (data hiding)
- limited reliance on preprocessing
- limited use of pointers
- predictable & measurable memory use





the power of ten ruleset (2006)

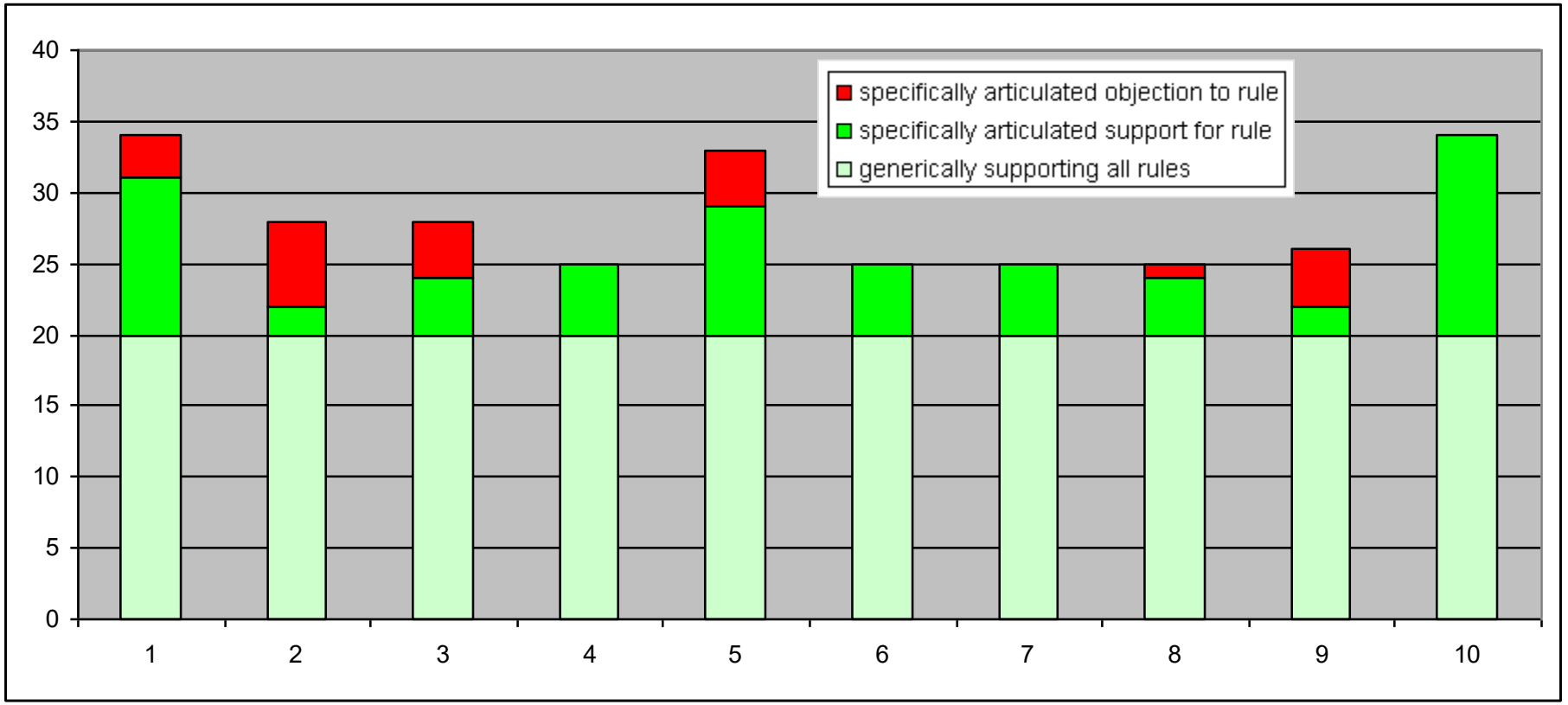

The Power of 10: Rules for Developing Safety-Critical Code

http://spinroot.com/p10/

1. simplify control flow: do not use gotos, setjmp, longjmp, nor direct or indirect recursion
2. provide and explicit iteration upper-bound for all loops
3. no dynamic memory allocation after initialization
4. use short functions – no longer than what can be viewed on one screen/page
5. *use assertions* – aim for an assertion density of 2% or higher
6. declare all data at the smallest possible level of scope
7. check the return value of all non-void functions, check the validity of function arguments
8. limit use of the preprocessor to file inclusion and simple macros – no token pasting, variable length argument lists – minimal use of conditional compilation directives
9. limit use of pointers – no more than one dereference, no function pointers
10. always *compile with all warnings enabled* at the compilers most pedantic setting – and use good static source code analyzers, minimally on every system build

initial responses per rule (28 specific comments)

weights: “yes”=2; “yes but”=1; “maybe”=0; “tough”=-1; “no”=-2;



1	2	3	4	5	6	7	8	9	10
simplify control flow	bounded loops	no malloc after init	short functions	use assertions	limited scope	check param and return values	limited use of cpp	limited use of ptrs	use static source code analyzers

external Blog posts discussing the rules

(> 50 responses, not all on topic…)

- you only need one rule
  - don't use C

- you only need two rules
  - use Scheme or Haskell,
  - use literate programming

- don't use C, instead use
  - Ada or SPARK/Ada (5 people)
  - ML, Erlang, Java, C++ (1 person each)

- the whole exercise is pointless
  - the correctness of the specification is the bigger problem
  - multi-threading / concurrency is the bigger problem
  - array bound checking is the bigger problem
  - this is like designing a "safe heating system" for the Titanic

https://yourdon.com/personal/blog/2006/06/20/ten-rules-for-developing-safety-critical-code/

https://talkaboutquality.wordpress.com/tag/tools/

https://lambda-the-ultimate.org/archive/2006/6/21

https://www.csci.csusb.edu/dick/blog.html#Holzmann06

assertion density

- ms office code (25MLOC) was reported in 2003 to have an assertion density of 1% (250K assertions)
- a general survey in 2005 added:

  proprietary projects: 3% average assertion density

  open source projects: 5% average assertion density

  projects written in the Eiffel language: 6.4% assertion density

  sources:

  C.A.R. Hoare, *Assertions: a personal perspective*, IEEE Annals of the History of Computing, 25 (2), 2003, pp. 14-25.

  P. Chalin, *Ensuring continued mainstream use of formal methods: an assessment*, Roadmap and Issues Group, D.S.R., Concordia University, 2005.

using assertions makes a measurable difference

Assessing the Relationship between Software Assertions and Code Quality: An Empirical Investigation

Gunnar Kudrjavets [1], Nachiappan Nagappan [2], Thomas Ball [2]
[1] *Microsoft Corporation, Redmond, WA 98052*
[2] *Microsoft Research, Redmond, WA 98052*
{gunnarku, nachin, tball} @microsoft.com

MSR-TR-2006-54

Abstract

The use of assertions in software development is thought to help produce quality software. Unfortunately, there is scant empirical evidence in commercial software systems for this argument to date. This paper presents an empirical case study of two commercial software components at Microsoft Corporation. The developers of these components systematically employed assertions, which allowed us to investigate the relationship between software assertions and code quality. We also compare the efficacy of assertions against that of popular bug finding techniques like source code static analysis tools. We observe from our case study that with an increase in the assertion density in a file there is a statistically significant decrease in fault density. Further, the usage of software assertions in these components found a large percentage of the faults in the bug database.

Keywords: Assertions, Faults, Bug database, Source control systems, Correlations.

1. Introduction

There is much literature that makes a case for the use of assertions and discusses the potential benefits of using assertions in software development. But to date, there have been limited studies in academia or in industry that empirically address the utility of assertions. Even when we talk to developers within Microsoft there are no unified opinions about the usefulness of assertions.

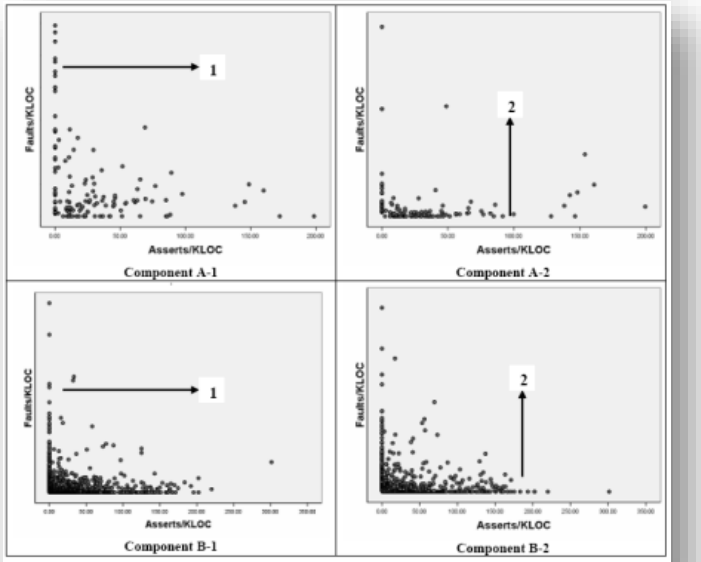


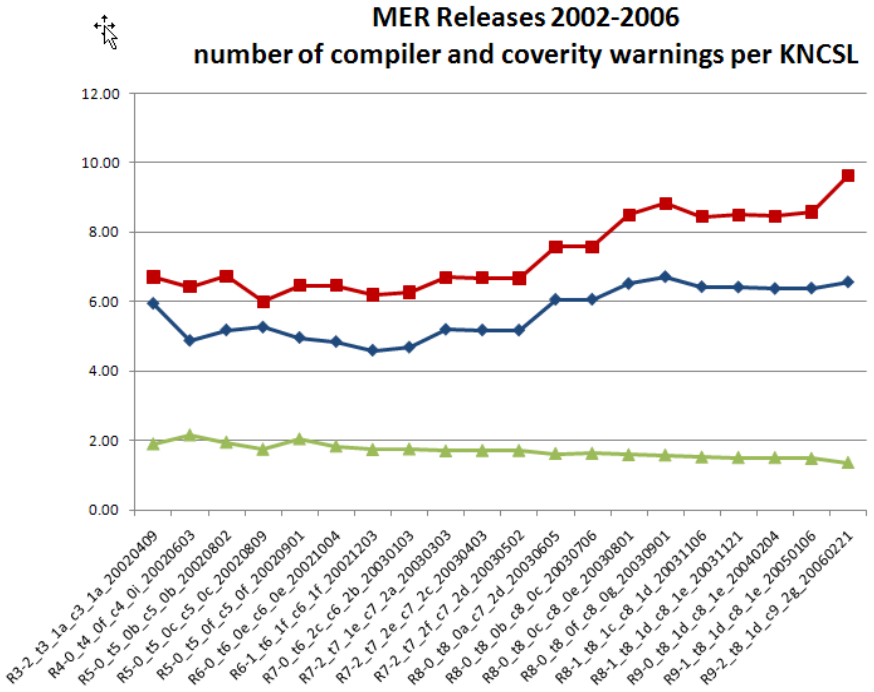
Figure 3: Scatter plots between assertion density and fault density for components A, B

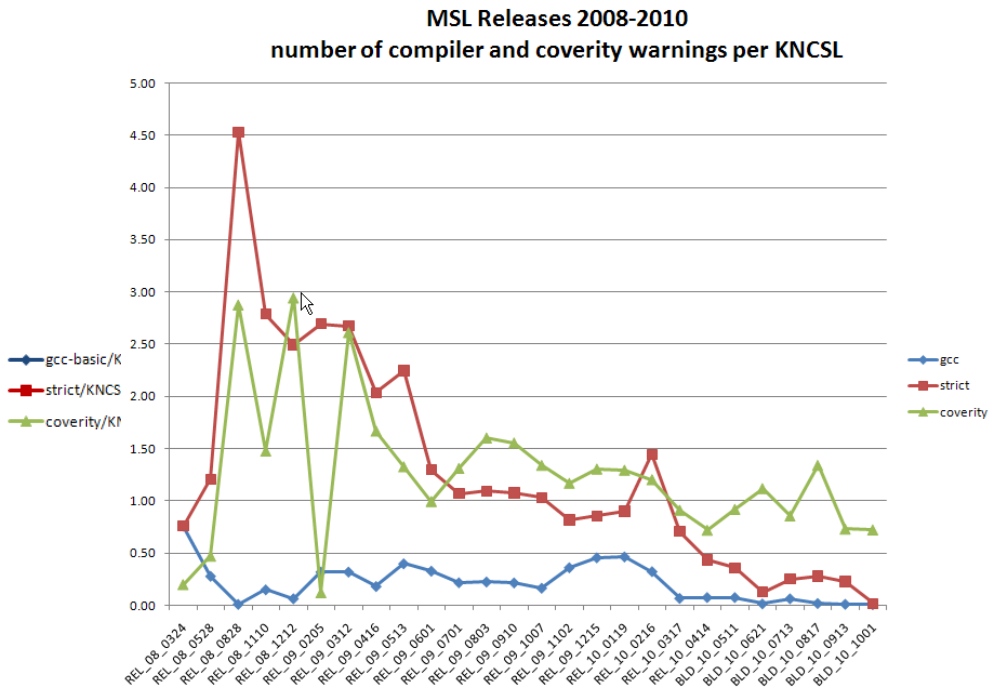
"with an increase in assertion density there is a statistically significant decrease in fault density"

compile & check with *all* warnings enabled
(and map warnings to errors that stop the build)









MER 3 years post-launch (2/06):

- 6.6 compiler warnings per KNCSL
- 9.6 compiler-strict warnings per KNCSL
- 1.4 static analyzer warnings per KNCSL

MSL 1 year pre-launch (10/10):

- no compiler warnings
- no compiler-strict warnings
- 0.7 static analyzer warnings per KNCSL

tool support: *automating checks*

(integration with the build and code review process)

Nightly Build → make → Build Log →

collect all compiler calls (thousands)

Defect Detection

Coding Rule Compliance

gcc

gcc strict

coverity

codesonar

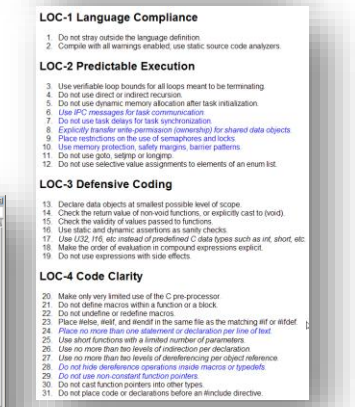
cobra

P10

JPL standard

MSL rules

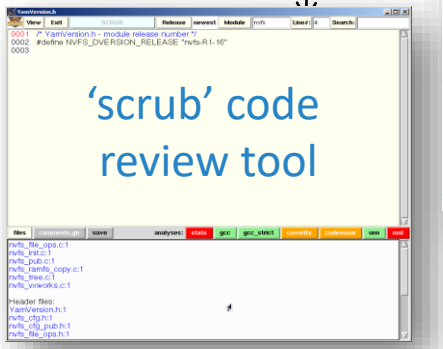




suppressed warnings

Global Analysis (~15hrs)
Module Analysis (~10min)

database →



consequences for breaking the build

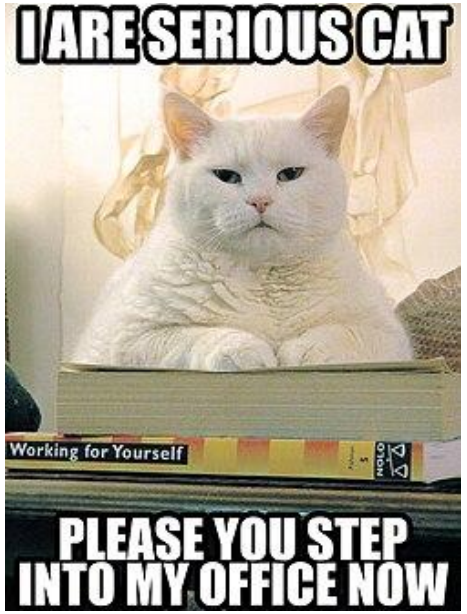


method 1 (spacex/lockheed):

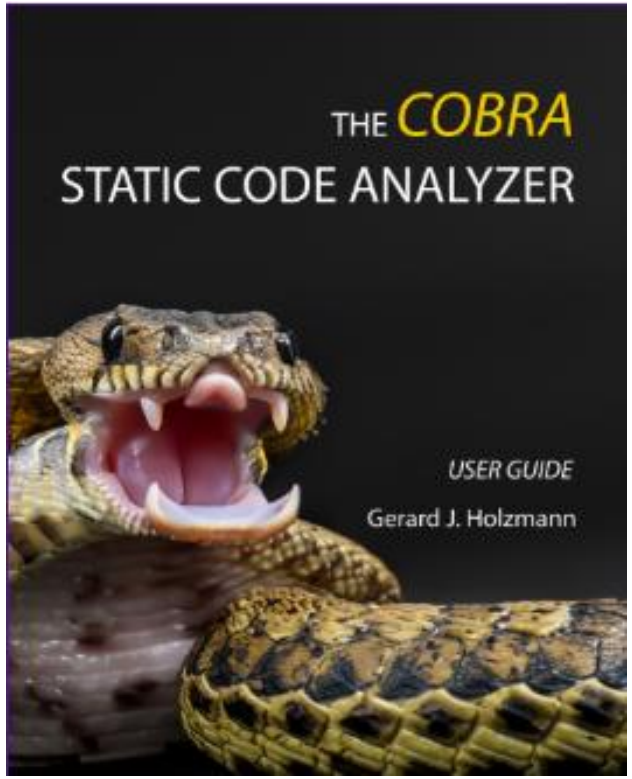
a life-size card-board cutout of Justin Bieber is placed in your office for the day

method 2 (jpl):

a lol-cat poster is taped on your office door for the day



tool support



https://codescrub.com

fast static source code analyzer, supporting

- interactive query commands
- token-based pattern matching
- inline programs
- builtin checks for standards (including P10)
- static and streaming input (runtime verification)
- language neutral and configurable
  builtin support for C, C++, Ada, Java, HTML,Python…
- open source since 2017 (github)

thank you!