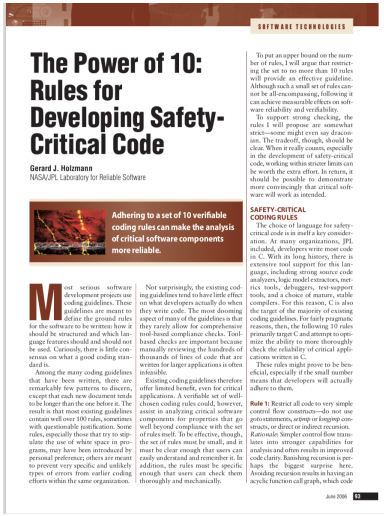
Lecture 7 — The “Power of Ten”

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

“Power Of Ten” — Overview

“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**.”



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 ode

June 2006 19

“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.**”

- **Author:** Gerald Holzmann, *NASA’s Jet Propulsion Laboratory*

Control-Flow Statements

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***"*

- Easier **analysis** and improved **code clarity**
- No recursion implies acyclic **call graph** ...
- ... making for easier **memory analysis** and **WCET**

Q) *What do we use recursion for?*

Bounded Loops

Rule 2

"Give all loops a fixed upper bound. It must be trivially possible for a checking tool to prove statically that the loop cannot exceed a preset upper bound on the number of iterations."

- Lack of recursion and bounded loops prevent **infinite loops**
- Some **exceptions apply**, such as for process scheduler.

```
for(int i = 0;i < arr.length; ++i) { arr[i] = 0; }
```

```
while(i >= 0 && i < arr.length) {
  if(arr[i] < 0) { i = i - 2; }
  else { i = i + 3; }
  arr[i] = 0;
}
```

Prohibit Dynamic Memory Allocation

Rule 3

"Do not use dynamic memory allocation after initialization."

- Appears in **most** safety-critical coding guidelines.
- Garbage collection / `malloc` has **unpredictable behaviour**.
- Many errors from **mishandled** memory allocation / deallocation.

```
for(Point p : points) { ... }
```

```
ArrayList<Point> ps = ...;
for(int i=0;i!=arr.length;++i) { ps.add(arr[i]); }
```

Function at most 60LOC

Rule 4

"No function should be longer than what can be printed on a **single sheet of paper** *in a standard format with one line per statement and one line per declaration. Typically, this means no more than about* **60 lines of code per function***."*

- Functions should be **logical units** of code.
- Functions should be easy to **understand** and **verify**.

Q) *Does this include comments?*

Minimal Assertion Density

Rule 5

"The code's assertion density should average to minimally **two assertions per function***. Assertions must be used to check for anomalous conditions that should never happen in reallife executions. Assertions must be* **side-effect free** *and should be defined as Boolean tests. When an assertion fails, an explicit* **recovery action** *must be taken such as returning an error condition to the caller of the function that executes the failing assertion"*

- Roughly expect **one defect** per 10 . . . 100 Lines of Code.
- Detecting defects **more likely** with more assertions.
- Can verify pre- and postconditions, loop- and class invariants.

Declare Objects with Smallest Scope

Rule 6

Declare all data objects at the smallest possible level of scope.

- Supports basic principle of **data hiding**.
- Cannot **corrupt** variables which are not in scope.

```
int i;
for(i = 0;i < arr.length; ++i) { ... }
```

```
class X { int i; void f(int[] arr) {
    for(i = 0;i < arr.length; ++i) { ... }
} }
```

```
int x = xs[0]; int y = xs[1]; xs[0] = y; xs[1] = x;
```

Must Check Returns

Rule 7

"Each calling function must check the return value of nonvoid functions, and each called function must check the validity of all parameters provided by the caller."

- Function's return value normally **exists for a reason**!
- Especially important if **error code** returned (e.g. `malloc`).
- Cast to **`void`** to signal really want to ignore it.

```
int max(int x, int y) { ... }

int[] create(int w, int h) {
  max(w,h); return new int[w];
}
```

Limited use of Preprocessor

Rule 8

"The use of the preprocessor must be limited to the inclusion of header files and simple macro definitions. Token pasting, **variable argument lists** *(ellipses), and recursive macro calls are not allowed."*

- C preprocessor a "powerful **obfuscation** tool"!!
- Code using macros can hard to decipher
- Use of **conditional compilation** not always avoidable.

Restricted Use of Pointers

Rule 9

"The use of pointers must be restricted. Specifically, no more than **one level** *of dereferencing should be used. ...* **Function pointers** *are not permitted."*

- Pointers **easily misused**, even by seasoned programmers.
- Hard to follow flow of data in a program.
- Pointers very difficult for **static analysis tools**.
- Use of function pointers can easily **hide recursion**.

Must Enable Compiler Warnings

Rule 10

"All code must be compiled, from the first day of development, with **all compiler warnings enabled** *at the most pedantic setting available.* **All code must compile without warnings**. *All code must also be checked daily with at least one, but preferably more than one, strong* **static source code analyzer** *and should pass all analyses with zero warnings."*

- **No excuse** for not employing such technology.
- Code **confusing** static analysis should be rewritten.
- Static analysers have **bad reputation** for producing false positives.