Experience using The Power of Ten coding rules

By Michael McDougall and Gerard J. Holzmann

Michael McDougall is a Senior Scientist at GrammaTech, which develops software analysis tools. Gerard J. Holzmann is a Fellow at the Jet Propulsion Laboratory and a Faculty Associate in Computer Science at the California Institute of Technology.

Introduction

Safety-critical software is software whose failure can lead to injury or death. It is present in planes, cars, medical devices, spacecraft, and other systems. The "Power of Ten" coding rules were introduced in 2006 as a minimal set of rules for writing safety-critical code in C. Unlike other guidelines, these rules were specifically designed to leverage the power of static analysis tools. The rules were developed at NASA's Jet Propulsion Laboratory (JPL), but they can fruitfully be applied to any safety-critical software written in C. They have been integrated into JPL's new institutional coding standard for flight software; for example, all flight software written for the upcoming Mars Science Laboratory Mission to Mars, which is due to launch in the Fall of 2011, will comply with this new coding standard.

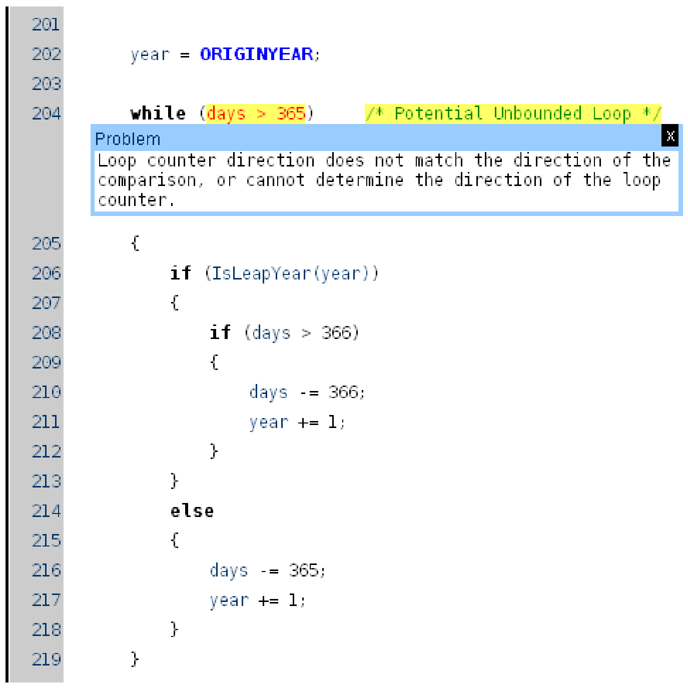
Summary of the Power of Ten Rules [as table]

1. *Restrict to simple control flow constructs.* For instance, do not use goto statements, setjmp or longjmp constructs, and do not use direct or indirect recursion.
2. *Give all loops a fixed upper-bound.* It must be possible for a checking tool to prove statically that a preset upper-bound on the number of iterations of a loop cannot be exceeded. If the loop-bound cannot be proven statically, the rule is considered violated.
3. *Do not use dynamic memory allocation after initialization.* This excludes the use of malloc, sbrk, alloca, and all variants, after thread or process initialization.
4. *Limit functions to no more than N lines of text.* A function should in principle not be longer than what can be printed on a single sheet of paper in a standard reference format with one line per statement and one line per declaration. Typically, this means no more than about 60 lines of code per function.
5. *Use minimally N assertions for every function of more than M lines.* Assertions are used to check for anomalous conditions that should never happen in an execution. Assertions must be *side-effect free* and are best defined as Boolean tests. For safety critical code, typical values for N and M are N=2 and M=20. For less critical applications, the value of M could be increased but to be meaningful it should always be smaller than the maximum function length (See Rule 4).
6. *Declare data objects at the smallest possible level of scope.* Data objects only used in one file should be declared file *static*. Data objects only used in one function should be declared local static.
7. *Check the return value of all non-void functions, and check the validity of all function parameters.* The return values of non-void functions must be checked by each calling function, and the validity of parameters must be checked inside each function.

8. *Limit the use of the preprocessor to file inclusion and simple macros.* The C preprocessor can have surprisingly complex behavior that is best avoided. Token pasting, variable argument lists (ellipses), and recursive macro calls are not permitted. All macros must expand into complete syntactic units. The use of conditional compilation directives should be restricted to the prevention of duplicate file inclusion in header files.
9. *Limit the use of pointers.* Use no more than N levels of dereferencing (star operators) per expression. A strict value for N is 1, but in some cases using N=2 can be justified. Pointer dereference operations may not be hidden in macro definitions or inside typedef declarations. The use of function pointers should be restricted to simple cases.
10. *Compile with all warnings enabled, in pedantic mode, and use one or more modern static source code analyzers.* All code must compile without warnings, and must be checked on each build with at least one, but preferably more than one, good static source code analyzer. It should pass the analyses with zero warnings.

The rules may at first sight seem overly strict, but recall that they are intended to guard the development of safety-critical systems. The rules can be compared to the use of seat-belts in cars—they are perhaps a little bit constraining at first, but easily justified by the reduction of the risk of death or injury that they bring.

The recent leap-year bug in the Zune30 MP3 player is a nice example where the violation of Power of Ten Rule 2 had significant consequences. The rule requires that all loops have a verifiable bound. This is particularly important in embedded systems, where runaway code can have dire consequences. The loop that the Zune 30 used to process dates did not properly handle the 366th day of a leap year. Nor did the loop did have a failsafe limit to catch infinite execution. As a result, Zunes were unusable for all of December 31st 2008. While the Zune is not a safety-critical device, the same failure in an airplane, car, or even a cell-phone (suddenly unable to call 911) could well be considered safety critical.



```
    year = ORIGINYEAR;

    while (days > 365)      /* Potential Unbounded Loop */
```

Problem
Loop counter direction does not match the direction of the comparison, or cannot determine the direction of the loop counter.

```
    {
        if (IsLeapYear(year))
        {
            if (days > 366)
            {
                days -= 366;
                year += 1;
            }
        }
        else
        {
            days -= 365;
            year += 1;
        }
    }
```

[Figure: “report.png”. Caption: “A report from the CodeSonar static analysis tool, warning that the loop in the Zune driver code is unbounded. The loop is unbounded in the case where the ‘days’ variable is equal to 366.”]

Why a New Coding Guideline?

Existing coding guidelines, like those by MISRA and the Joint Strike Fighter project, list hundreds of rules with voluminous supporting documentation. While such guidelines can be admirably comprehensive, many software teams will balk at having to learn and apply so many rules. In practice this means that most rules will be ignored. A project team could decide to monitor a small subset of the rules, but it can be difficult to determine which of the hundreds of rules offer the most bang for buck in terms of software quality improvements.

Another factor inhibiting adoption is a lack of generality. Guidelines often mix well-established best practices—which could apply to many projects—with stylistic rules (e.g., regulating the use of white-space) that are peculiar to a project or company. Rules may also deal with APIs or domains that are not broadly relevant—for example, rules about working with Windows DLLs are not relevant to a Unix project, while rules about managing page tables are irrelevant outside an OS kernel.

The “Power of Ten” rules were in part designed to overcome these problems. The rules are easy to memorize and apply when writing and reviewing code, yet they capture the essence of writing code in a safety-critical context. The rules do not define project-specific constraints on names or code layout, nor do they refer to particular APIs or platform-specific problems. Short, powerful, and widely applicable, the “Power of Ten” rules are a great starting point for embedded developers looking for a core set of coding rules.

Reaction to the Power of Ten

The Power of Ten rules were first published in IEEE Computer in 2006, with a brief summary posted at http://spinroot.com/p10. They have been discussed extensively on blogs, at workshops and tradeshows. The topic tends to generate much enthusiasm and some controversy. Frequently, the set of rules is strongly endorsed, but almost everyone will have one rule in the set of ten that they find unworkable. Curiously, there appears to be no consensus on which rule should be cast from the set: everyone will pick a different one. This probably means that the set is reasonably well defined. It is very hard to reach complete agreement on any attempt at standardization, and coding rules are of course no exception.

Using the Power of Ten Rules at JPL

Like any software development tool, the Power of Ten Rules needed to be tested in the field. The initial testing was carried out by a small JPL team developing a set of mission-critical flight software modules. The team used a combination of manual reviews and automatic checking using a prototype version of the CodeSonar tool developed by GrammaTech. These early experiments showed that it was readily possible to develop software under the constraints of the rules as well as the schedule constraints of a real flight project.

The relatively effortless adoption of the Power of Ten rules led to the creation of a new formal "JPL Institutional Coding Standard" that combines the Power of Ten rules with a small number of additional rules that are more specific to the spacecraft context. The JPL Coding Standard applies to all new flight software development at JPL. The next Mars rover will indeed be running software that complies with the Power of Ten rules. It has been a significant challenge to reach this point, because this next rover will also run more code than all previous missions to Mars combined (i.e., a few million lines of C).

Many of the additional rules in the JPL Standard constrain how tasks in a real-time system may interact; complex task interaction has been a source of many spacecraft bugs, including the one that almost led to the loss of the Mars Pathfinder lander in 2007. In some cases, the JPL Coding Standard puts additional restriction on what features of C can be used—preprocessor features are especially constrained. The ability to customize the rules (e.g., by defining the limit on function length, or the minimal number of assertions per functions of a given minimal size) allows for quick experimentation, which in turn makes it easy to find an acceptable and enforceable level of thoroughness.

Similar to the CMMI standard, the JPL Institutional Coding Standard divides its coding rules into six separate levels of compliance—different projects can adopt a different levels of compliance depending on how mission-critical the project is (highly critical code should be compliant at the level 6) and the cost of bringing any legacy code into full compliance.

The introduction of different levels of compliance was an important factor in getting critical support at JPL (among both management and developers) for the adoption of the new standard. Quality improvement is never an "all of nothing" proposition. It is possible to make significant improvements with a more gradual approach, easing the transition towards broader improvements of the software development process.

The JPL standard was developed over a period of approximately four years. The initial step was to ask all key developers for their opinion of the MISRA-C 2004 coding guidelines. We also asked the developers to identify what they believed to be the 10 most important rules from that set, as well as the 10 least important rules. This revealed considerable consensus among the developers, but curiously we found little or no correlation between the set of rules that most developers considered to be the most critical and the actual coding practices followed by those same developers in their own software development. Sometimes we need help sticking to what we know is the right thing to do. Until the Institutional Coding Standard was adopted, every project and mission defined its own standard, always slightly differently from other projects, and always with hundreds of rules but no verification that the code complied with those rules. JPL management strongly supported to move towards a single unified standard with verifiable rules, but the real support that counted was of course the support of the software developers themselves. Many of the key developers were part of the process that led to the adoption of the new coding standard, and virtually all have embraced it as a valuable tool in the effort to increase code quality and reduce risk. What is perhaps most unexpected is that the new Coding Standard has evoked a sense of enthusiasm among both managers and developers. For a standard that is meant to restrict what one can do, this is perhaps the best of all possible outcomes.

Using Static Analyzers to Enforce the Rules

A coding guideline has little value if it is not enforced, but manually reviewing potentially millions of lines of code to check compliance to the rules is a tedious, if not utterly infeasible. Automatic tools can check compliance quickly and thoroughly, while freeing software reviewers to focus on higher-level design issues. The Power of Ten rules were designed with automatic analysis in mind. The use of static analysis tools to check compliance with the Power of Ten rules is today a required part of the flight software development process at JPL.

In order to support automatic enforcement of the Power of Ten, GrammaTech worked with JPL's Laboratory for Reliable Software to define the rules precisely enough that they could be checked mechanically. GrammaTech then developed a prototype (based on the CodeSonar static analysis tool) for checking the rules. The extended CodeSonar prototype was deployed at JPL for use by engineers coding under the Power of Ten rules. As expected, early use revealed some cases where the tool's judgment deviated from what its users expected (for example, flagging loops as potentially unbounded when they used safe and commonly understood idioms). However, after some revisions, the prototype became the primary tool used to ensure compliance with the rules. CodeSonar is also one of a handful of static analysis tools looking for common errors, such as null-pointer dereferences and buffer overflows.

JPL uses a new code reviewing tool called Scrub to track compliance with the coding rules as an integral part of a peer code review process. Analysis tools perform automatic checks on every build of flight code, and the results are collated and presented to reviewers via Scrub. JPL expects to make the Scrub tool available to the public so that others can try the tool with their own static analysis tools. JPL also plans to release some of the non-commercial analysis scripts that feed results into Scrub, so users without access to the commercial analyzers will still be able to check for some undesirable patterns.

Evolution of the Rules

The public response to the Power of Ten rules and the experience of applying the rules to flight software development led to some clarifications and fine-tuning of the original rules. For example, initially Rule 9 forbade the use of function pointers under any circumstances, but this was too restrictive and provided a significant hurdle in the use of legacy code. Rule 9 has since been relaxed somewhat to allow function pointers so long as it is always tractable (if not by a human then at least by a static analyzer) which function or functions are actually pointed to. Rule 5 initially specified a minimum density of two assertions per function. However, this was too severe for small procedures that had minimal functionality, so it was generalized somewhat.

The rationale and discussion of the rules has also evolved. The Power of Ten website at http://spinroot.com/p10 now has an extended discussion of the intent and precise meaning of each rule, its rationale, and tips for applying. This content continues to evolve to address comments by others and to clarify the rules.

Adopting the Power of Ten in Other Contexts

Even though the Power of Ten rules were designed for safety-critical software written in C, their aim of reducing coding errors by making source code easier to understand and analyze is relevant to any project that cares about software quality.

If your project is written in a language other than C, many of the rules can still be applied with minimal modification. For example, Rule 2 (“Give all loops a fixed upper-bound”) can be applied to virtually all currently used programming languages.

If your project does not warrant a strict guideline like the Power of Ten, keep the general principles in mind. Perhaps limiting your functions to 60 lines is too constraining for your project, but the overall goal of keeping functions short so they are easy to understand should still apply. Complex pointer manipulation may be warranted in projects that are not safety-critical, but such projects can still benefit if they keep in mind the spirit of Rule 9: pointers are a common source of confusion, so limit levels of indirection when possible. In general, the idea that making source code easier to understand and analyze will yield better software is appropriate wherever software quality matters, whether or not you are working on safety-critical software.

Learning More

The Power of Ten rules were introduced ''The Power of Ten -- Rules for Developing Safety Critical Code,'' *IEEE Computer*, June 2006, pages 93-95, and are discussed in more detail at http://spinroot.com/p10/

An overview of commercial and free static analysis tools is available at http://spinroot.com/static/

The Motor Industry Software Reliability Association (MISRA) has coding guidelines for C and C++. They can be purchased at http://www.misra.org.uk/

The JSF coding standard for C++ is available at http://www.jsf.mil/downloads/documents/JSF_AV_C%2B%2B_Coding_Standards_Rev_C.doc

The JPL Institutional Coding Standard is maintained by JPL's Laboratory for Reliable Software: http://eis.jpl.nasa.gov/lars/ . A copy of the coding standard is available upon request.

Acknowledgement

The research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.