WG14/N1155

High Integrity Coding Guidelines

A Proposed Rationale

Derek M. Jones
derek@knosof.co.uk

1 Introduction

The commercial production of software is an investment. Investors seek to maximise profit and minimise risk. An investment in a product or service containing software is not guaranteed to be a commercial success. Investors may be willing to trade-off an increase in costs if a product is successful (i.e., increased maintenance costs), against a decrease in exposure to risk (i.e., commercial failure) by reducing the initial investment.

Some products operate in an environment where there is the potential for software failure to cause significant harm to people (including death). In some cases regulatory authorities specify minimum requirements on the investment that needs to be made to show that software behaves in a predictable fashion. Meeting these requirements is a cost that forms part of the initial investment needed to produce a software program.

These guidelines are thus based on economic principles. There is an investment cost in ensuring some degree of predictable behavior and there is a perceived benefit acrued from reducing the probability that people will be harmed.

2 The spirit of high integrity coding guidelines

The following lists the basic aims of these coding guidelines recommendations:

- Ensure predictable behavior.
- Possible to measure adherence to the guidelines.
- Based on empirically derived information.
- Based on an economic analysis.
- Recognize that existing knowledge is incomplete.

For various economic reasons software may be written in a computer language that is different from the instruction set of the processor on which it executes (i.e., a high level language). These guidelines are intended to apply to high level languages.

2.1 Predictable behavior

It might be thought that given sufficient information the behavior of a program could be predicted to any desired level of accuracy. In practice there are a number of issues that can make accurate prediction very difficult. These issues include:

- Interaction between a source code and the language translator used. Computer language specifications do not always uniquely specify the behavior of every construct. It is possible for the same program to exhibit different behavior when translated by different translators, or even the same translator executed using different options.
- The resources needed to calculate the predicted behavior may not be economically practical.
- Mistakes may be made in calculating the predicted behavior.

Ensuring that a program behaves in the intended manner (i.e., it meets its specification) is outside the scope of these guidelines.

Testing is one method of increasing confidence that the behavior of software matches that predicted. However, testing is outside the scope of these guidelines.

2.1.1 Language translators

Language translators exist within particular economic and cultural environments.

The specification of a language (for some of the languages considered here this will be a published international standard) is driven by a number of factors. Some of these include the following:

- Multiple existing practices. Creating a standard is a political as well as a technical process. Reaching agreement sometimes involves allowing different existing behaviors to continue to be standards conforming.
- Efficient execution. The underlying instructions on to which language constructs have to be mapped vary between processors. The specification of a construct may support efficient execution by allow different mapping to be used. The efficiency ability of translators to generate efficient code can also be increased by not overly restricting the permitted behavior.

To be commercially viable a company selling language translators has to target as broad a customer base as possible. At the moment the high integrity market is not large enough to support the production of translators dedicated to its' requirements.

Given this economic reality it is considered unrealistic for these guidelines to require that any translators used during development handle a particular construct in a given way. It is accepted that software development projects will have to make use of the translators that are available to them. Therefore the approach taken by these guidelines is primarily aimed at developer behavior and sometimes require them to:

- Not use a construct.
- Not use a construct in such a way that differences in behavior might occur when different translators are used.

It is often possible for different executions of the same translator to generate, from the same source, machine code that has different behaviors. These changes in behavior are often the result of changes in the settings used for translator options. A translator is likely, but not required, to generate the same machine code from given source if the same set of options are used.

Requiring that source code be translated using the same set of translator options as those used during testing and assumed to be used by any program analysis tools helps ensure that predicted program behavior matches actual behavior.

The continuing desire for optimal code generation has led some researchers to investigate translators that use genetic algorithms. The extent to which genetic algorithms will be used by commercially available translators is not known. Use of genetic algorithms makes it likely that the executable program generated by a translator will vary between translations, even when the same options are given.

2.1.2 Resources required

The two primary resources used to predict the behavior of source code are software developers and automated tools. Each have their own strengths and weaknesses.

Automated tools are very a cost effective solution to verifying developers beliefs about source code, when they are available. However, there are a number of reasons why such tools might not be available, including:

- Economics. It is possible that nobody has invested in producing a tool, of the necessary quality, to perform a specific kind of code analysis. This economics of tool production is affected by the popularity of the computer language being analysed. For instance, the language ML is popular in some academic circles and many tools for analysing it have been produced by researchers, while the language C is popular in some commercial and academic circles and many tools for analysing it have been produced.
- An algorithm for reliably checking certain program properties and behaviors may not be known.
- The known algorithms require so much resources (e.g., time or memory) that nobody has thought it worthwhile implementing them.

The analysis performed by some algorithms can be sufficiently memory or cpu intensive that it limits their usefulness on real-life programs. In some cases the quantity of resources required is driven by the volume of source code that needs to be analysed, while in other cases it is driven by the kinds of constructs that appear in the source code.

The volume of source code is likely to be controlled by factors outside the scope of these guidelines.

One solution to the resource demands generated by the use of some kinds of source code constructs is to recommend against the use of such constructs, or limit their use to specific instances.

There are mathematical proofs showing that it is not possible to always solve some problems that occur in program analysis (i.e., they are undecidable). The solution to some program analysis problems is known to be NP-hard or simply P. Such mathematical proofs invariably show that there exists at least one program having these properties. Whether or not only one such program exists, or a few dozen, or most programs have this property (i.e., is the behavior likely to be encountered in practice) is of importance is deciding which simplifications might be recommended by these guidelines.

2.1.3 Errors made in predicting behavior

There are a number of sources of prediction error, including the following:

- Human fallibility. Software developers make mistakes. Reasons for these mistakes include: incomplete or inaccurate knowledge of the computer language used, cognitive limitations and common mode operating characteristics (e.g., making assumptions). Possible solutions include recommendations not to use constructs that experience has shown to be the root cause of developer error, and requiring that chunks of code do not exceed the cognitive capacity of the developers who are likely to read it.
- Unexpected translator behavior. In a few cases the specification of a language is driven by the behavior of its translator (e.g., Perl), but in most cases the specification is contained in a document that translators are expected to follow. Failure of a translator to follow the documented specification can result in it generating programs whose behavior is different from that predicted by the language specification. One way of increasing the confidence that a translator follows its language specification is through the use of validation suites.

2.2 Measuring adherence

If adherence to guidelines cannot be measured then:

- Conformance can be claimed without verifiable proof being available.
- Different interpretations of the requirements are likely to go undetected.

Automatic checking of guideline requirements means there is less likelihood of human error and to produce more consistent results.

2.3 Empirically based

It is possible to specify an infinite number of plausible coding guideline recommendations. Measurements of source code can provide a number of benefits, including the following:

- Evidence that the construct described in a guideline occurs in existing source code.
- Evidence that the behavior of a construct is not reliably predicted by developers.
- Evidence of the behavior of translators.

While existing source code can be measured, reliable information on the kinds of mistakes made by developers is scant.... Reliable information on the cost of defects is also scant...

2.4 Economic analysis

Predicting the behavior of programs is not an all or nothing process. In general the greater the investment of effort the greater the accuracy of the prediction. This suggests that at some point the cost of achieving a given prediction accuracy will be greater than the benefit of the information obtained. The following is some of the information that is needed to perform a cost/benefit analysis of a proposed guideline recommendation:

- How often does the construct covered by a guideline appear in source code?
- What is the cost associated with a construct covered by a guideline not being reliably predicted?
- Is the cost/benefit associated with using an alternative to the construct covered by a guideline more worthwhile than the cost/benefit of using the guideline construct?

2.5 Move forward cautiously

There are a number of reasons why it is necessary to take a cautious approach to creating coding guideline recommendations. These reasons include:

1. Little hard information is available on which guideline recommendations might be cost effective.
2. It is likely to be difficult to withdraw a guideline recommendation once it has been published.
3. Premature creation of a guideline recommendation can result in:
   - Unnecessary enforcement costs (i.e., if a given recommendation is later found to be unworthwhile).
   - Potentially unnecessary program development costs through having to specify and use alternative constructs during software development.
   - A reduction in developer confidence of the worthwhileness of these guidelines.

It is better to introduce guideline recommendations over time, as practical experience and experimental evidence is accumulated.