What is predictable execution?

Brian Wichmann

August 24, 2006

1 Introduction

The concept of predictable execution is not as straightforward as it might seem. What is often meant is that the behaviour of the program was not as intended — but since we know that capturing the 'intent' in a concrete manner is hard, this line of reasoning is not very productive.

Consider an ordinary binary program, which, for simplicity, interacts with it environment in a minimal manner, but runs under some operating system. It is highly likely that the OS initialises the store and registers and hence the program may well behave predictably, *regardless of the nature of the machine code*.

One could claim that programs always act predictably, it is just that the reasoning can be rather complex in some circumstances. Some years ago, a Pascal program written to check the implementation of integer division ran inconsistently. Eventually, it was found that the processor chip had an error so that if an interrupt occurred when the integer division operation was being performed, the machine state was not recovered correctly - thus giving the inconsistency. The later processor chips corrected this error. In this paper we are not concerned with such implementation faults — we assume the processor and the execution environment (compiler, interpreter, loader, etc) does not have implementation faults.

However, deducing properties of the program just from the machine-code is very hard and not useful now that high level programming languages are almost universal.

2 High level language view

We need to be able to deduce the execution characteristics in a more convenient way based upon the following:

1. The relevant programming language definition.
2. The features defined for this specific implementation (which may in turn depend upon compiling options).

3. The source text.

This approach is quite seductive, but has a grave problem as formulated above. What if the language definition makes requirements not enforced by a compiler? Perhaps it is stated that there shall be no access to an uninitialised variable. Hence the first item needs to be restricted to the static semantics of the language which is required to be enforced by the compiler.

Consider now an array access in which the index value is not in range. In Java and Ada, this results in the raising of an exception, while for C the semantics is undefined. However, it must be assumed that this was a programming error and hence for all languages one requires that this situation does not arise. Hence one in forced to conclude that the concept of *predictable execution* requires that certain properties of a program are verified.

Hence our modified approach to predictable execution requires that one shows the absence of predefined exceptions (in Java and Ada) or the equivalent conditions which would otherwise result in an undefined execution.

2.1 Portability

Portability is not strictly necessary if the properties of the specific system are understood. However, strict portability makes in much easier to be sure that the execution properties apply in a specific case. Moreover, if a tool is used to detect for any non-predictable aspects, it is vital that the implementation defined characteristics are correctly set.

2.2 Language definition

The quality of these varies very substantially. For instance, Fortran does not define any dynamic properties (the concept of a run-time error is not defined). This was true in the 60s when Boyer and Moore did their classic work on proving programs correct [6]. They just had to write a definition of the language based upon the English text.

I believe that the definitions of Ada, Java and C are adequate to define the notion of predictable execution. I am uncertain about C++ in this regard which makes me nervous about the use of the language. (It seems that the problem with C++ is the size and complexity of the language, rather than inherent issues in the design.)

2.3 Language for code generation

Languages can be used to generate code from some other program specification. In many cases, problems which would otherwise arise may be avoided by using properties of the specification. In other words, predictable execution may not be clear from the generated code, but could be from the original specification.

3 Language problems

Execution order. If the execution order is not defined, then a combinatorial problem can arise in attempting to predict the execution characteristics of a program.

Side-effects in functions. This could be regarded as a special case of the execution order problem, but from the point of view of program analysis, banning side-effects is best.

Permitted optimizations. The C language introduces sequence points for this purpose, but causes some difficulties in establishing predictable execution.

Parameter passing. Fortran introduced special wording, which very few people understood to allow some flexibility in this area. Ada does something similar which can cause problems unless aliasing can be avoided. (In some situations, Ada structures can be passed by copy or reference.)

Aliasing. If an item of storage is accessible in more than one way, then the compiled code may depend upon how two different accesses are handled. Program proof has similar problems. Particularly troublesome with pointers.

Storage control. This is handled automatically with Java (but then gives problems with timing). Ada has an unsafe feature for reclaiming storage and hence does not require garbage collection.

Exceptions. The method which makes predictable execution easier to verify is to require that predefined exceptions are not raised. Many situations in C which result in unpredictable execution would raise an exception in Ada. In consequence an Ada coding with no exceptions being raised can be very similar to the C coding with no unpredictable execution.

Tasking. A very tricky area. This is not considerd here.

4 Ensuring predictable execution

No widely-used programming language makes achieving predictable execution easy. On the other hand, the great majority of execution of many programs are predictable and hence the problem is one of assurance and locating the sources of unpredictability.

The only language that I know that makes predictable execution its primarily aim is *Newspeak* [8].

Hence we have somehow to make do with existing languages, or at least subsets of them.

Conceptually, the language-based method of ensuring predictable execution should allow for tools to verify the property, or indicate statements which are not

predictable. Unfortunately, there are severe practical limits to this approach. Comprehensive languages like Ada 95 or C++ which are used without regard to static analysis makes checking many properties virtually intractable, such as ensuring no access to unset variables. (It is tempting to think that the speed of modern computers checking of code in the full language would be possible — this is not the case due to the exponential nature of the analysis required.)

The unset variable problem also indicates some of the issues in language design. Ada 95 and C are conventional in this area in that this condition for unpredictability exists. This implies that the number of paths through the code which need to be checked can grow exponentially with the program size. Moreover, programs can be 'correct' and yet be virtually impossible to check perhaps due to dynamic conditions which are unknown to any static analysis tool.

For Ada 95 and C a solution must be to adopt a coding style which makes the task of checking for no unset variables by a tool tractable. One must clearly avoid junk initialisations, since that just removes one issue to create another.

Java overcomes this problem by placing restrictions on local variables and requiring objects to be initialised. In the context of a very dynamic language like Java this is a very reasonable design choice.

One way round some of the static analysis problems is to use a subset of the full language. The adherence to the subset and the predictable execution properties could then be checked with manageable overheads.

4.1 Language subsets

One means of easing the problem of verifying predictable execution is to use a subset designed to avoid some of the static analysis issues.

At least in the area of high integrity safety systems, the use of subsets is well established.

To illustrate both the advantages and the problems of the use of subsets, we take two examples, SPARK [10, 9] and MISRA-C [11].

4.1.1 SPARK

SPARK is an Ada 95 subset in the sense that the production of code from SPARK uses an ordinary Ada 95 compiler. It is much more than an Ada subset due to the addition of annotations and the availability of the SPARK examiner which checks for adherence to the subset. The examiner is very fast and yet checks for no access to unset variables. It can achieve this because the annotations provide design information about the usage of variables which would not be present in conventional Ada code.

SPARK is really a design tool using the concept of *Correctness by Construction* [9]. This implies that converting full Ada 95 to SPARK is inappropriate — a redesign is needed.

In addition to the Examiner, a tool is available which will make a significant step to showing the code is exception-free. This tool exploits the Ada range

constraints, as would an Ada compiler. This tool can also exploit (optionally) additional annotations to prove key properties of a program. These properties could verify predictable execution, or go substantially further in verifying correctness of the code (a topic outside the scope of this paper).

There is an interesting alternative way of looking at predictable execution. The University of Toronto Research Group in defining the language Turing stated as their objective that a conforming implementation of the language should be *faithful* (see page 338 of [7]). By this term, they meant that a program which compiled would either fail to execute with an appropriate error message or the execution could be predicted exactly from the language definition. A note prepared by the author analyses SPARK from this point of view in 1998. Some small details are not correct for the current version of SPARK, but it does cover the main aspects of the issue.

4.1.2 MISRA-C

This subset is a complete contrast to SPARK, due to starting from a very different language and not attempting to define a design tool. The basic concept of MISRA-C is simple enough: to obtain predictable execution by removing any language features which are clearly problematic. For instance, `malloc` is not permitted.

The objectives of MISRA-C is not only predictability but also intelligibility which therefore includes some coding style requirements. Another major difference from SPARK is that there is no specific tool but rather an aim to encourage a market for tools which check for the subset.

MISRA-C gives the impression that predictable execution can be guaranteed by means of adhering to the subset which can be checked statically. This is clearly not the case, since, for example, an array index needs to be in range, which can only be checked (in general) dynamically.

4.1.3 Some comparisons

SPARK and MISRA-C are very different, even if some of the objectives are similar and both are being used in embedded safety systems.

SPARK is rather well defined, and a formal definition was produced a few years ago. In contrast, the definition of MISRA-C is is less precise which is to be expected since it handles intelligibility. MISRA-C is only stated to be appropriate for a Safety Integrity Level of 3 or less, while SPARK has been used for several Level 4 systems.

Over the years, the size of the SPARK subset has increased — an interesting recent development has been a verifiable tasking facility. MISRA-C has gone through one revision, but now appears to be static. SPARK is based upon the current Ada standard, while MISRA-C appears to be based upon the 1990 edition of the C standard rather than the current one of 1999.

5 Qualification of tools

Current languages effectively require to use of tools to demonstrate predictable execution. Since modern programming languages are relatively complex, showing that a tool is fit-for-purpose is difficult. There are three vital characteristics:

1. Showing that the tool reports any unpredictable aspects of the code;
2. Noting the extent to which a tool reports false positives, ie warns of insecure aspects which on further analysis turn out not to be insecure;
3. Aspects of usability of the tool.

It seems that the only feasible approach is to check these three aspects by running test cases. This is a very difficult and time-consuming task. In essence, it is similar approach used the validate compilers.

One development which could easy the construction of static analysis tools is standards like ASIS [1]. Here, the semantic information necessary to compile an (Ada) program is recorded in a data structure which can then be used by analysis tools. Apart from reducing the cost of developing static analysis tools, it reduces the risk that the tools diverge from the compiler (and hence the generated code).

The Pi Technology study of MISRA tools is excellent [13]. Unfortunately, from the point of view of establishing predictable execution, it has a number of drawbacks: the definition of MISRA-C has changed with the new edition [11]; the tools have also surely changed as well; compliance to the MISRA-C subset is somewhat different from establishing predictable execution. The study shows the difficulty in tool evaluation and the large potential difference between tools.

Two studies have been undertaken on static analysis to detect buffer overflow problems in (legacy) C code [12, 14]. These reports make depressing reading, since the detection is difficult and never 100% even though from the logical viewpoint, it is straightforward. The major questions arising from this work is the impact of using MISRA-C or more modern coding standards.

An alternative approach of using dynamic tests when a static test cannot be shown to be adequate. The is detailed in [14]. The problem with this technique is that when a dynamic test fails, the systems design must provide for an appropriate action. No such action is possible for some safety systems, such as fly-by-wire aircraft with unstable aerodynamics. For most systems, especially outside the safety area, reporting such a fault could lead to rapid fault correction and higher reliability than is currently achieved. The alternative to using a dynamic test is to undertake a review of the code — this could resolve the actual problems with the code, but is an expensive process.

6 Floating point

To establish predictable execution in a useful sense, numerical analysis needs to be applied, see [4, 3]. Hence this is not a language issue. If classical numerical

analysis has not been applied and an algorithm is unstable, then the numerical results are indeed unpredictable.

If one writes:

$$\textbf{if } 0.1 \times 10.0 = 1.0 \textbf{ then}$$

then the results could well be unpredictable. Some think that a coding standard prohibiting the use of equality with floating point solves that problem — it does not! For instance, suppose that a calculation requires the solution of a quadratic equation in which the physics ensures that the roots are real. Hence one might write: $\sqrt{b^2 - 4ac}$. The problem here is that rounding error might produce a (small) negative value. Hence it is necessary to check that the result is not negative, and that in the case of a negative value, it is small enough to be attributed to rounding error. Rounding error is unavoidable with floating point and it needs to be taken into account with all non-trivial calculations.

The IFIP book [4] has a lot of information about the relationship between programming languages and numerical computation.

Unfortunately, numerical analysis is rarely taught at the undergraduate level, which implies that there is a danger that this issue is missed in validating critical systems. This problem arose recently when a development team was unable to compute $\lfloor p \times q/r \rfloor$ where p, q and r are integers.

7 Conclusions

It is hard to justify any *unpredictable* execution in a high integrity application. On the other hand, establishing predictable execution in all circumstances is quite demanding.

It might seem that defensive programming is a solution to this problem, at least as far as checking for preconditions for language constructs. This can be done in many circumstances, such as for safety systems for which a safe state can be imposed. However, this then gives rise to another issue: how often will the system trigger the safe state response when the correct action is surely to correct the bug in the program.

Much more needs to be done to make predictable execution achievable for the widely used programming languages with industrial sized applications. Such activity might include:

1. Development of ASIS-style [1] standards and implementations for languages other than Ada.

2. Development of test facilities for static analysis tools.

3. Standards for annotations for languages to improve the effectiveness of static analysis tools.

4. Development of the proposed SC22 Guidelines [17].

7.1 Proposed wording for the ISO Guidelines

Definition Predictable execution: The property of the program such that all possible executions have results which can be predicted from the relevant programming language definition and any relevant language-defined implementation characteristics. The execution should not raise any exception and floating point calculations should be numerically stable.

References

[1] ISO/IEC 15291:1999, Information technology — Programming languages — Ada Semantic Interface Specification (ASIS)

[2] ISO/IEC TR 15942:2000, "Information technology – Programming languages – Guide for the use of the Ada programming language in high integrity systems" URL

[3] N J Higham. Accuracy and Stability of Numerical Algorithms. SIAM. 1996. ISBN 0-89871-355-2

[4] B Einarsson (Editor). Accuracy and Reliability in Scientific Computing. SIAM. 2005. ISBN 0-89871-584-9

[5] Entry for `strictfp`: URL

[6] Robert S. Boyer, Matt Kaufmann, and J Strother Moore. The Boyer-Moore Theorem Prover and Its Interactive Enhancement. *Computers and Mathematics with Applications*, 29(2), 1995, pp. 27-62. URL

[7] R C Holt and J N P Hume. Introduction to Computer Science using the Turing programming language. Reston, 1984.

[8] Currie, I. F., NewSpeak: an unexceptional language, *Software Engineering Journal*, pp. 170-176 (July 1986).

[9] Amey Peter. *Correctness by Construction: Better Can Also Be Cheaper.* Praxis Critical Systems Limited. CrossTalk Magazine, March 2002. URL

[10] J Barnes. High Integrity Software — the SPARK Approach to Safety and Security. Addison-Wesley. 2002.

[11] Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, 2004 (second edition).

[12] Misha Zitser, Richard Lippmann, Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, 2004.*

[13] A Comparison of MISRA C Testing Tools. October 2001. URL

[14] Thomas Plum, David M. Keaton. Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool. URL

[15] Kendra June Kratkiewicz. Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. URL

[16] B A Wichmann. Is SPARK faithful? October 1998. Available from the author.

[17] ISO/IEC Project 22.24772: Guidance for Avoiding Vulnerabilities through Language Selection and Use. URL

A Document details

1. First written March 2006.
2. Extensively revised, August 2006.
3. Minor revision to take account of comments from Clive Pygott, 8th August 2006.
4. Revised to add proposed wording for the Guidelines as a result of BSI meeting on 15th August.
5. Revised to include amendment resulting from email exchange, 24th August 2006.