ISO/IEC JTC 1/SC 22/OWGV N0049

22 November 2006

Contribution from:
Brian Wichmann, Tool assurance for predictable execution, 3 November 2006

Tool assurance for predictable execution

Brian Wichmann

November 3, 2006

1 Introduction

The starting point of this paper is the concept of predictable execution as defined in [8]. Hence we are adopting a high level language view. Consider as an example, the high level language statement:

$$a[i] := a[i] + 1;$$

To be predictable we require:

1. i has been assigned.
2. i is in the index range of the array a.
3. $a[i]$ has been assigned.
4. $a[i]$ is not equal to the largest integer (for simplicity, we are assuming integers).

Hence it is clear that some degree of automation is required if reasonably sized programs are to be shown to be predictable.

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 prognosis of using static analysis to show predictable execution is not good since the comparatively small issue of buffer overflow has proved tricky, at least with the C programming language, see [2, 4].

However, we need to be able to assess static analysis tools and gain some assurance of their application to real code. We do this by analysis of the major issues in a language-independent fashion.

If full access to the source text is not available, then there is likely to be significant limitations to the assurance that can be provided. For those modules

for which the source is not available, information about the access and use of external data would be vital. Design information should be available in this case.

Assurance via static analysis will be very different for a hard real-time system with full source compared with a very large system using many components available only in binary form.

2 Language problems

We list here some of the issues which are likely to present problems in undertaking static analysis.

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.

Apart from expressions, for the chosen language, it is necessary to enumerate the language structures which can be evaluated in a different order such that an execution will not be predictable.

Example:
`printf("hello ")+printf("world")`

Computations *versus* update of state. Languages like Pascal and Ada separate functions and procedures so that the operations of computing a value and update of the state are syntactically separated. C/C++ and Java do not which can give rise to confusion is some cases. Program proof and static analysis is easy for languages making the separation. Note that Java is well-defined here by specifying the order of evaluation of expressions.

Example:
`f((*a)++, *b)`
where `a` and `b` point to the same entity.

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.

Example:
`f((*a)++, *b)`
where `a` and `b` are aliased.

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.

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

3 Qualification of tools

Since modern programming languages are relatively complex, showing that a tool is fit-for-purpose is difficult. There are four vital characteristics:

1. Claims made by the tool supplier.
2. Showing that the tool reports cases of unpredictable aspects of the code;
3. 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;
4. 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 to validate compilers.

We now consider the validation of a tool for our chosen language L. We do this by considering some of the language problem areas listed above:

Execution order. We assume our language L has already been analysed so that the features whose execution order can vary are known.

Each of the features could be allowed or prohibited in an application.

For each prohibited feature, we need to be assured that our chosen tool can check for its absence. This can be checked by a suitable test.

For each allowed feature whose execution order can vary, we need a suitable test case of its use. Applying the tool to these test cases can ensure that the tool detects these allowed instances of non-predictable execution.

Computations *versus* update of state. For languages in the Pascal/Ada style, side-effects in functions should be prohibited and checked by an appropriate tool. This is undertaken by the SPARK Examiner [1].

For language like C/C++ and Java the situation is more complex and harder to analyse.

Parameter passing. The analysis of the programming language semantics must determine if the parameter passing mechanism guarantees predictable execution or not.

If predictable execution is not guaranteed, then it is necessary to produce a test case whose execution could vary. This test needs to be applied to the chosen tool to demonstrate that this problem is detected.

Aliasing. Aliasing is a problem if program proof is being attempted. The paper [8] does not cover this issue, so we can ignore this and only address the issue of assurance of predictable execution. Note sequence points in C/C++ are introduced to allow some code optimization but can result in unpredictable execution via aliasing.

Hence the issue is whether the language semantics restricts aliasing in any way. For instance, program optimization might place x in a register. What happens if x is aliased to y and y is changed? The language C allows this between sequence points which therefore provides a potential source of non-predictable execution.

The analysis of the language must provide a list of such aliasing.

Test cases must be written to show that the chosen tool detects such situations.

Storage control. Languages such as Java and C# undertake automatic storage control and therefore the issue of validity of pointers does not arise at the language level (only at the implementation level).

There are several strategies that can be adopted here. Firstly, to prohibit all uses of constructs involving storage allocation (other than the stack); use a restricted set of facilities (typically allocating storage only during program initialisation) and lastly using the full facilities of the language.

The set test cases needed therefore varies according to the restrictions made. With no restrictions, many cases need to be considered in which storage control is misused. A typical case is to allocate and then deallocate storage, but then attempt to access the deallocated storage.

4 Conclusions

It seems feasible to provide guidance on the qualification to tools used the aid checking for predictable execution in a language independent fashion.

There is a significant issue which is not addressed here at all — the use of annotations, especially to give design information which can be checked by static analyis tools.

At this point, no attempt has been made to provide wording for the ISO guidelines.

References

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

[2] 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.

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

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

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

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

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

[8] B A Wichmann. What is predictable execution? Paper for meeting of SC22 group. August 2006.

A Document details

1. First written August 2006.
2. Very minor revision, 27th October 2006.
3. Significant revision, 3rd November 2006 to take into account comments from Les Hatton and those arising at the BSI meeting on the 31st October.