Looking into Safety with the Safety and Security Rapporteur Group

Stephen Michell
Maurya Software Inc
29 Maurya Ct.
Ottawa, Ontario
steve@maurya.on.ca

Mark Saaltink
ORA Canada
1208-1 Nicholas St.
Ottawa, Ontario
mark@ora.on.ca

Brian Wichmann
National Physical Laboratories
Teddington, Middlesex,
United Kingdom
Brian.Wichmann@npl.co.uk

Abstract

The requirements of High Integrity (safety-critical, secure and mission-critical) Software force developers to use specialised development techniques. Often the choice of computer language and the constructs used are based upon static and dynamic analysis requirements, as well as regulatory requirements and project-specific requirements. To support the growing number of groups doing this kind of development and their different programming requirements, WG9 (the ISO Ada Working Group) and its Safety and Security Rapporteur Group (HRG) are forging ways to support these organizations. The HRG is developing an ISO Technical Document titled *Guidance on the Use of the Ada Programming Language in High Integrity Systems* to provide explicit guidance to groups using Ada in this arena. This document helps a project identify its analysis requirements, and determine which Ada language features are best-suited to support the analysis being done. This guidance, coupled with analysis based upon the Ada Semantic Interface Specification (ASIS), should help teams specify, support, and enforce the language features needed for their specialized requirements.

1 Introduction

The High Integrity Software Development Environment includes Safety-critical software, Secure software, and mission-critical software where a single failure could have catastrophic results. In order to ensure that even a single failure will not occur, this software is developed in a very constrained environment where endless testing and rigourous static analysis predominate. As software is being used more predominantly in critical situations, the number of people and projects that must consider the specialized techniques of this arena is expanding. If you are developing high integrity software in Ada95, the International Ada Working Group's Safety and Security Rapporteur Group (HRG) may be able to help you.

2 Background

There has been a definite move to a more careful engineering approach in the development of High Integrity Software in the past ten years. Formal analytical approaches, have been replacing the "prove safety by testing" approaches of the past.

High Integrity Software has been defined as software in which a failure results in a significant or catastrophic loss of life, or an unacceptable loss of resources. The shutdown of the Eastern US telephone system, the destruction of the Ariane 5 rocket, and the classic array overrun bug in "sendmail" and "finger" are all examples of catastrophic losses where software errors were the root cause [12, 11]. The looming year 2000 software crisis is another example of a catastrophy in the making due to preventable software design and coding errors.

Such catastrophes can result from any of a number of errors, including specification errors, design errors, coding errors, hardware faults, lapses in configuration management, poor user interface, or unanticipated operating conditions. The developers of high integrity systems thus need to take special care in all aspects of the development. Our focus in this paper is in one of the areas: developing safe and correct code.

Recently, software has been moving inexorably into every facet of human interaction with technology—airplanes, trains, automobiles, traffic control, communications, banking, environmental control, and medicine. As this occurs, professional societies are beginning to understand that they must take a stand on the ways that software is specified, designed, implemented, tested, and fielded. Software must be designed and implemented to a standard of "best practice", and be shown through early analysis that its use will not harm humans. If not the professional societies, then the courts through punative fines will deal with software developers that have taken a cavalier approach to the design and implementation of high integrity software.

After the Ada(95) language was standardized as ISO 8652-1995, a Safety and Security rapporteur group was created by ISO-IEC/JTC1/SC22/WG9 (Ada Working Group) called the HRG (after Annex H of the ARM). This group is chartered to develop guidance on the use of Ada in high integrity systems. This guidance will directly address the issues facing high integrity software developers who are using Ada in their projects.

3 Finding a path to safe code

The major challenge for the software engineer is to achieve the high levels of integrity demanded by critical systems. Furthermore, not only must the high integrity level be attained, often it must be possible to demonstrate this achievement to a regulator, and in the last resort it may be necessary to demonstrate the achievement in a court of law.

Engineers in other areas use a combination of testing and analysis to ensure the safety and quality of their constructs. Software, in contrast, is often only tested. Exhaustive testing is possible for systems with a small number of possible states, but is impractical for any system of realistic complexity. For complex systems, testing—even systematic testing—neither gives assurance that the probability of failure is acceptably low, nor that the code correctly implements the design, nor that the design satisfies the requirements.

The only currently practical approach to safety for most projects is a combination of testing and analysis, and especially static analysis. There are many kinds of static analysis such as data flow analysis, control flow analysis, range analysis, symbolic execution, and formal proof. Each has different merits and costs, and each can become intractible in certain cases. It is therefore important that users of these analysis techniques understand the strengths and limitations of them, and know how to write code that is amenable to the types of analysis they wish to perform.

4 The HRG and Ada95 Approach

For a number of years, Ada has been the language of choice for the development of large high integrity software systems, albeit in subset form. The strong type checking, modularity, and support for checkable separate compilation provided excellent support for team design and implementation. SPARK [7, 4], Alsys CSMART, AVA [16] and Penelope [9] are examples of systems used in this environment. Some of these subsets came with integrated proof tools that assisted the formal development and discharge of proof obligations, as long as the methodology and constraints were rigorously followed.

When Ada95 was being developed through the early 1990s, it was recognized that there was an opportunity to provide much better support for the High Integrity community. Ada95 explicitly added mechanisms to assist in restricting language usage, in building subsystems, in partitioning programs, and in supporting concurrency in a minimalistic way. These restrictions were mainly oriented at simplifying the runtime, but it was recognized at the time that many projects would have differing needs in this area, and that a single set of restrictions for high integrity would not succeed.

Since the standardization of Ada95, two major activities are combining to provide meaningful support to the developers of High Integrity systems. First, the work of the international Ada Working Group's Safety and Security Rapporteur Group (HRG) uses an analytical approach to consider the restriction of Ada95 language features [5, 13, 14]. Secondly, a standard (called the Ada Semantic Interface Specification, or ASIS) is being developed to provide projects and tool developers complete access to the Ada compiler's parsing and semantic analysis of a program to provide a unified, consistent view of the program being analysed. (This work is discussed below in the section titled "Understanding ASIS".) By using this interface, a project can verify constraints imposed by High Integrity requirements, industry-related constraints, or even by coding standards.

5 Understanding the HRG Approach

Since the HRG intends to support projects and business areas in defining their own coding standards based upon their specific needs, the first step was to set a framework for our analysis. We started with the premise that a project's language requirements were based upon the kinds of analysis that they would be doing on their code as it was being developed and tested.[1] To identify and categorize this analysis, we first developed a taxonomy of the techniques used in many high integrity software projects.

The taxonomy includes many forms of static analysis, such as various forms of flow analysis, symbolic execution, formal code verification, timing analysis, and stack usage analysis. It also includes a number of dynamic analysis techniques such as coverage analysis, requirements-based testing and structure testing.

Craigen, Saaltink and Michell [5] proposed and used a rating system for analysis of Ada features. Following a similar approach, the HRG developed nine categories based on grouping of analysis techniques with which to rate Ada language features:

- Flow Analysis (FA) includes control flow analysis, data flow analysis, and information flow analysis. These analyses are based on the program's statement structure and call graph, and can identify dead code, recursion, unintended data dependencies, and other anomolies.
- Symbolic Analysis (SA) includes symbolic execution and formal code verification. These analyses use algebraic or logical manupulations of code to derive information about its functionality and, in the case of formal verification, to check the functionality against a specification.
- Range Checking (RC) is used to detect arithmetic underflows and overflows or array indexing errors.
- Stack Usage Analysis (SU) is used to predict the maximum amount of stack space reqiuired in the execution of a program.
- Timing Analysis (TA) is used to establish temporal properties of the input/output dependencies of a program.
- Other Memory Usage (OMU) is an analysis of the use of any shared resources, such as the heap, I/O ports, or special purpose hardware.
- Object Code Analysis (OCA) is a validation of the object code produced by compiling a program, usually carried out by a manual inspection of the source code and object code.
- Requirements-based Testing (RT) includes several testing methods, such as boundary value testing, that are based on a program's requirements rather than on the code itself.

[1] The HRG acknowledges that in addition to the analytic-based categories, there are engineering-based categories which also affect how, or whether, a feature is used. Craigen *et al* [5] propose the consideration of a feature's robustness, security, and engineering support. The HRG felt that these categories are not objective enough for the rating system that was being used, and that they would be included as additional considerations when projects were defining their own subsets.

6.11.1 Evaluation

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
Unconstrained array types - including strings[1]	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Full access types	**Exc**[2]	**Exc**[2]	Inc	**Exc**[2]	**Exc**[2]	**Exc**[2]	Inc	Inc	Inc
Restricted storage pools[3]	**Alld**[4]	**Exc**[4]	Inc	Inc	Inc	Inc	Inc	Inc	Inc
General access types	**Alld**[4]	**Alld**[4]	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Access to subprogram	**Exc**[5]	**Exc**[5]	Inc	Inc	**Alld**[5]	Inc	**Alld**[5]	Inc	Inc
Controlled types including unrestricted storage pools	**Exc**[6]	**Exc**[6]	Inc	Inc	Inc	Inc	**Alld**[6]	Inc	**Exc**[6]
Indefinite objects	**Alld**[7]	**Alld**[7]	**Alld**[7]	**Exc**[7]	**Exc**[7]	**Exc**[7]	**Exc**[7]	Inc	**Exc**[7]
Non-static array objects	Inc	**Alld**[8]	**Alld**[8]	**Alld**[8]	**Alld**[8]	**Alld**[8]	Inc	**Alld**[8]	Inc

6.11.2 Notes

1. Note that the concatenate function returns a type of an unconstrained array. Refer to Section 6.6 for more information.
2. Full access types employ the run-time system to allocate from the heap and other memory areas, making memory use unpredictable, timing analysis problematic, heap exhaustion and fragmentation a significant risk. It can also create unbounded aliasing problems.
3. Pool-specific access types use memory similarly to array-based data types. However, they require careful implementation and use to ensure the algorithms are predictable.
4. Pools and general access types permit aliasing of data. See restrictions in Section 6.2.
5. Access to subprogram types disrupts control flow, and make it difficult to export analysis results of subprograms into calling subprograms. This exclusion can be enforced by the pragma Restrictions(No_Access_Subprograms). When used with static locations and linker tools, they can be used as a means of system reconfiguration.
6. Controlled types introduce hidden control flows due to user-defined initialisastion, assignment, and, especially, finilisation. These are hard to review, analyse or test, particularly in error conditions.
7. Indefinite objects consume time and memory in ways which are difficult, if not impossible, to predict. Their dependence on run-time values complicates analysis. Consequently, these objects should not be used in high integrity systems.
8. Arrays with bounds which are not static complicate analysis of resources used. Time and memory use depends on dynamic bounds. Analysis of data access based on array indexing is further complicated if the bounds are unknown until run-time.

6.11.3 Guidance

As noted in the introduction to this section, the use of dynamic mechanisms is to be minimised in high integrity systems. Appropriate enforcements can be provided by the use of pragma Restrictions(No_Implicit_Heap_Allocation), pragma Restrictions(N0_Allocators), and pragma Restrictions(No_Access_Subprograms).

Although the evaluation table has "Inc" against nearly all the testing based verification techniques, it should be noted thatthe effectiveness of these techniques may well be reduced. For example, a problem arising from the inappropriate use of aliasing may well be difficult to find during Requirements-based testing and Structure-based Testing. It is also true that code inspection techniques will be made more complex (and hence error-prone) by the use of these dynamic features.

Figure 1: Evaluation of types with dynamic attributes (from [10])

- Structure-based Testing (ST) includes several testing methods, such as statement coverage testing or decision coverage testing, that are based on the program code itself.

These categories are explained in detail in the HRG's report.

Along with the rating categories, a three-way scale is identified that could be applied to how an analysis category supported a language feature: Included; Allowed; and Excluded. An included rating for a feature with respect to an analysis technique means that a feature is directly amenable to that verification technique. A feature is allowed if the designated verification step is not straightforward, but is still achievable; or if the use of the feature is necessary and the use of the problematic verification technique can be effectively circumvented. A feature is excluded if there is no current cost effective way of undertaking the designated verification technique. Assurance of exclusion requires some form of verification that is amenable to the designated verification technique.

These ratings for each language feature are captured in tables that reflect a general grouping of features. The groupings chosen were Static Types; Declaration; Names, Scope and Visibility; Expressions; Statements; Subprograms; Packages; Arithmetic Types; Low level and Interfacing; Generics; Types with Dynamic Attributes; Exceptions; Tasking; and Distribution. As a typical example, Figure 1 shows the organization and notes for the table Dynamic Types and Access Types. Every feature that has an "allowed" or "excluded" against one of the analysis categories also has a note to explain the issues. Where appropriate, some features also have notes that explain restrictions or usage required to achieve the ratings given.

To show how the HRG document would be used, suppose that a project was considering the use of access types in the project. The table and notes in Figure 1 are from section 6.11 which deals with most of the issues of access types. A scan of this table tells us that full access types and access to subprogram preclude most categories of analyses. Pool-specific access types may be usable, if suitable care is taken. General access types provide a way to use pointers without dynamic memory allocation and are compatible with most analysis techniques providing that their use is restricted to support the analysis required; Section 6.2 of the HRG report describes the necessary restrictions.

A project following these guidelines would likely prohibit full access types, but would consider general access types with coding conventions. The most common restrictions are the exclusive use of objects of these types with aliased variables at the library level and restricting the full view to the private part or the body of the package.

When examining the table, or any other tables in [10], it is important to realize that the choice of features cannot be soley determined by examining one table for each feature. There are situations where features interact. For example, if one or more representation clauses is applied to an object, then storage analysis and timing analysis are affected and it is likely that additional runtime code (not represented in the source code) will be generated. Examination of section 6.9 on Low Level and Interfacing features considers the issues associated with representation issues.

6 ASIS

Since the Ada95 understanding of language restrictions or subsets is that a project or projects would define their own subsets based upon analytical criteria such as that defined by the HRG or project constraints, the important question becomes how will this subset be supported and enforced by compilers and tools? Traditionally, such tools have to have their own language parser and semantic analyser, as well as their own proprietary database. These factors severly limited the kinds of tools that could be built, and restricted the interoperability of such tools.

The Ada Semantic Interface Specification (ASIS) [3] is an ISO standard that provides tools and programs in the development environment a standard way of querying ada compilation systems about their knowledge of all compiled Ada units. The ASIS interface is being supported by almost every Ada compiler vendor.

The most significant contribution of ASIS is that it provides a common database (the compilation environment) for tool queries, so that analytic tools can be developed without forcing a project to become intimate with a single compiler vendor or to maintain their own parser and semantic analyser. This permits tools to be built which are portable across many compilation environments. ASIS also provides a standard way of querying implementation-dependent information, such as Integer'size or Ada.System.Max_Integer.

Because ASIS removes language-parsing issues from the tool, it lets tools work at the assembly of data into information level, not at data-identification and collection level. The information available through ASIS is very rich because of Ada's strong typing and modularity.

One ideal use of an ASIS tool is to support restrictions. An example of an ASIS-based restrictions checker is one to enforce the Ada Quality Style and Guide [15]. Another example is available from the ASIS web page,[2] and shows how ASIS an be used to enforce other kinds of restrictions.

7 A Challenge

A major issue for compilation environments and high integrity tools that interface with them is that it is highly desirable that the information determined by one is accessible by the others. With the increasing availability of ASIS, it is becoming easier for high integrity tools to access the information in the compilation environment, but currently there is no standard way for compilers to access information provided by tools, or for tools to provide information for each other.

An example of a use of tool knowledge by compilers is the scenario that a tool set may impose a set of restriction that does not correspond to the standard restrictions defined by the language. Nevertheless, a compiler may wish to take advantage of these restrictions to improve code generation, or to eliminate runtime routines.

An example of the aforementioned situation is the Ravenscar tasking model which could result in small, predictable runtimes for concurrent programs. To enforce the Ravenscar model, a compiler must be assured that the set of restrictions defined by the Ravenscar model are obeyed by the program. These restrictions do not exactly match the language-defined restrictions of Annex D and Annex H; for example there is no restriction to permit protected objects with exactly one entry or that entry barriers have only simple boolean expressions. Both conditions can be readily checked by a tool, but there is no way to transmit this knowledge to the compiler.

While it is likely that high integrity compilers will create implementation-defined pragmas for the Ravenscar Tasking

[2] http://www.acm.org/sigada/wg/asiswg/

model, they cannot handle other constraints as easily.

The requirement is therefore to develop a mechanism to improve communication between tools that use the Ada compilation database and ASIS. Since the compilation environment is essentially now an "open" database, mechanisms to populate this database with tool knowledge would be very useful. Three possible mechanisms to populate the compilation database with tool knowledge are;

1. Create ASIS calls to write to the compilation environment. The suggested mechanisms would be to add data to existing information to nodes which reflect legal Ada constructs such as type names, objects, or program units. Such additions could only add ancillary information and could not affect legality issues.

2. Extend the "pragma Restrictions" syntax to permit user-defined values and expressions. Standard restriction extensions could then be developed which compilers, as well as tools, could take advantage of.

3. Add a "pragma Annotate" to the language which would permit the population of the compilation environment with annotations on Ada lexical elements through the source code.

Each of the methods discussed have benefits, costs, and restrictions that must be discussed by the Ada community. The hope is, however, that this discussion begins soon to attempt to provide better integration of compilers and the tools that must augment them in these demanding environments.

8 Conclusions

With the increasing use of computer control in systems, and our increasing reliance on computerized systems, the need for high integrity software has never been greater. We expect more and more "mainstream" systems to be subjected to regulatory and commercial pressures to ensure their quality, trustworthiness, and safety. Software engineers and their managers will want to adopt the best practices for their systems development.

The HRG's guidance document helps developers in two ways: firstly by its discussion of the different possible code analyses, and secondly, by its tables that help to correlate a desired set of analyses and an allowable set of Ada language elements. Using this document, developers will be able to choose the most appropriate analyses and language features to use on their project.

The definition of ASIS provides an indirect benefit to these developers, by making it relatively easy to implement static analysers for Ada code. Several analysis tools have been written or are being written using ASIS, and the results are very promising.

Ada's pragma restrictions gives a programmer a way to inform the compiler that certain features, or combinations of feastures, are not used, and compilers are encouraged to take advantage of this declaration to improve their generated code. While not all of the restrictions identified in the HRG's document can be specified in the standard pragma, work is being done in the Ada95 community to augment these pragmas with ones that will support the HRG analysis.

For a number of years, Ada has been the language of choice for the development of large high integrity software systems. The new developments reported in this paper should add new weight in support of this choice.

References

[1] Intermetrics. *The Annotated Ada Reference Manual.* December 1994.

[2] ANSI/ISO/IEC 8652 International Standard. *Ada95 Reference Manual*, Intermetrics, January 1995.

[3] ISO-IEC/JTC1/SC22/WG9 ASIS Rapporteur Group. *Ada Semantic Interface Specification DRAFT 2.0 (ASIS).* Available by FTP from sw-eng.falls-church.va.us as file public/AdaIC/work-grp/asiswg/asis/v2.0/ASIS-2.0.N

[4] John Barnes. *High Integrity Ada: The Spark Approach.* Addison Wesley Longman, 1997.

[5] Dan Craigen, Mark Saaltink, and Steve Michell. *Ada95 Trustworthiness Study: A Framework for Analysis.* ORA Canada Report TR-95-5499-02, November 1995.

[6] *The Canadian Trusted Computer Product Evaluation Criteria.* Canadian System Security Centre, Communications Security Establishment, Government of Canada. Version 3.0e, January 1993.

[7] B. Carre and T. Jennings. *SPARK: The SPADE Ada Kernel.* Department of Electronics and Computer Science, University of Southampton, March 1988.

[8] *Software Considerations in Airborne Systems and Equipment Certification (DO-178B/-ED-12B).* RTCA Inc., Washington, D.C., December 1992.

[9] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal Verification of Ada Programs. *IEEE Transactions on Software Engineering*, vol. 16, no. 9, September 1990, pp. 1058–1075.

[10] ISO-IEC/JTC1/SC22/WG9 Safety and Security Rapporteur Group. *Guidance on the Use of the Ada Programming Language in High Integrity Systems*, Draft 3.5

[11] Nancy G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[12] Peter G. Neumann. *Computer Related Risks.* Addison-Wesley, 1995.

[13] Mark Saaltink and Steve Michell. *Ada95 Trustworthiness Study: Analysis of Ada95 for Critical Systems.* ORA Canada Report TR-95-5499-03, July 1996.

[14] Mark Saaltink and Stephen Michell. *Ada95 Trustworthiness Study: Guidance on the Use of Ada95 in the Development of High Integrity Systems.* ORA Canada Technical Report TR-97-5499-04a, 1997

[15] Software Productivity Consortium. *Ada 95 Quality and Style: Guidelines for Professional Programmers.* SPC-94093-CMC, October 1995.

[16] Michael Smith. *The AVA Reference Manual.* Technical Report 64, Computational Logic Inc., February 1992.

[17] *The Procurement of Safety Related Software in Defence Equipment (Parts 1 and 2).* U.K. Ministry of Defence, Standard 00-55 (Draft), 1995.