An ASIS-Based Static Analysis Tool for High-Integrity Systems

William W. Pritchett IV
DCS Corporation
1330 Braddock Place
Alexandria, VA 22314
703-683-8430 x726
wpritche@dcscorp.com

John D. Riley
DCS Corporation
1330 Braddock Place
Alexandria, VA 22314
703-683-8430 x714
jriley@dcscorp.com

1. ABSTRACT

This paper presents the results of an analysis which determined how the Ada Semantic Interface Specification (ASIS) can be used to statically detect certain Ada 95 language features deemed to be unsuitable for use in safety-critical systems. This paper also offers the design of a tool utilizing ASIS to automatically detect these language features.

1.1 Keywords

Safety-Critical, Software Tool, Static Analysis, Ada 95.

2. BACKGROUND

In recent years, the public has become increasingly reliant upon computer software to ensure passenger safety, patient safety and confidentiality, information security and financial transaction safety and integrity. This increased reliance on software, however, comes with an increased risk of failure arising from inadequacies in the processes and tools with which software is developed. Examples of recent failures in high-profile software systems include the Ariane-5 rocket malfunction in which software was improperly reused from the Ariane-4, resulting in the loss of the rocket; and the Therac-25 patient dosage device, which, due to a software error, inadvertently gave several patients lethal doses of radiation [1].

These failures highlight the need to perform stringent analysis and verification on safety-critical systems. Verifying critical systems most often involves analyzing software to ensure that the software will perform as intended. Software can be analyzed either statically or dynamically. Static analysis can involve the use of formal proofs, the examination of control flow, information flow, or data use; or the measurement of properties shown to have a correlation with certain quality factors (metrics). Dynamic analysis involves the execution of complete or partial software systems. Dynamic analysis can be very expensive and time consuming as the testing is dependent on the inputs and initial conditions of the test, which can be very large for fielded systems. Conversely, static analysis does not depend on any input, is applied only once for any particular analysis, and, depending on the type of analysis can be readily automated with tools. This paper addresses the static analysis of critical systems.

2.1 Ada 95 and Safety-Critical Systems

For critical systems, software failures may result in loss of life or severe financial loss. Developers of safety critical systems must be assured with sufficient confidence that these systems will not fail and the key to this assurance is guaranteeing the software behaves in a predictable manner at all times. A recent report prepared for the Canadian Department of National Defense identified a framework containing four issues crucial to assuring critical software [2]. These areas include:

- *Predictability* - the provision of strong scientific and engineering evidence that a critical system will behave in an intended manner.
- *Analyzability* - the ability to determine whether a system satisfies certain properties.
- *Traceability* – the ability to follow all requirements throughout the development of a system.
- *Engineering* – the ability to “design in” certain features making the application more reliable.

Ultimately, the choice of programming language plays an important role in developing critical systems with respect to the previously mentioned framework. A programming language must provide a language definition that minimizes insecurities and facilitates independent verification and validation. One language that has built-in standard facilities to address these constraints is Ada 95 [3]. The Ada 95 Safety and Security Annex contains theses facilities and deals with the following issues:

- Validation and verification requires detailed knowledge of implementation and unspecified behavior.
- Validation needs to be performed at the object code level, and requires a way to associate object code with the corresponding source code.
- Tailored run-time systems may be required in certain applications, thus reducing the complexity of verification and validation.

An often-used technique aimed at simplifying the validation process is that of restricting the language constructs used. Ada 95 supports the ability to restrict the use of language features through the use of the *Restrictions* pragma. In the presence of this pragma, the compiler rejects the compilation unit if the restricted feature is used.

The *Restrictions* pragma, however, may not provide enough language restrictions for certain safety-critical applications. To better evaluate the suitability of these restrictions, the Canadian Department of National Defense commissioned a study to evaluate each feature of the Ada 95 language for its suitability for use in safety-critical applications [2]. The study concluded with a report providing guidance on the use of Ada 95 in high-integrity systems [4][5]. This guidance document examined every feature of Ada 95 with respect to a set of evaluation criteria they developed and published in [2]. The result was a set of language features they recommend be restricted, or used with caution, in safety-critical systems. Their recommended restrictions were divided into those covered and not covered by the *Restrictions* pragma.

Specific guidance in the use of Ada 95 for the development of safety critical systems was prepared for the Canadian Ministry of Defense [4][5] and is continuing to evolve through organizations such as the Safety and Security (Annex H) Rapporteur Group (ISO/IEC JTC 1/SC22 WG9/HRG). The Canadian study was intended to be a roadmap for the use of Ada 95 in the development of critical systems. The study analyzes every feature of Ada 95 and provides recommendations for their use in safety-critical applications. An update to the original report was recently completed by the HRG and circulated for comments [6]. This report provides a three-way classification scheme for Ada features. This classification is based on how easy it is to apply a particular verification technique to programs containing the language feature. The three categories are *included*, *allowed*, and *excluded*. That is, a language feature is *included* if it is amenable to the designated verification technique.

2.2 Static Analysis Using ASIS

The *Restrictions* pragma is a very useful construct in limiting Ada 95 language features for compilers who conform to that LRM annex. The Canadian study, however, identified several language features that are not covered by the *Restrictions* pragma. Additionally, the Safety and Security annex is optional and some compilers may choose not to implement that annex. These cases require additional analysis, either through peer reviews or automated static analysis of the source code.

Manually checking source code for the use of language features can be very time consuming and may be accomplished more efficiently using automated techniques. For Ada, the static analysis of source code can be accomplished in a straightforward manner using the Ada Semantic Interface Specification (ASIS) [7].

ASIS is a standard interface between an *Ada Environment* and tools requiring information from that environment. In Ada, "each compilation unit submitted to the compiler is compiled in the context of an *environment* declarative_part (or simply, an environment), which is a conceptual declarative_part that forms the outermost declarative region of the context of the compilation." [3]

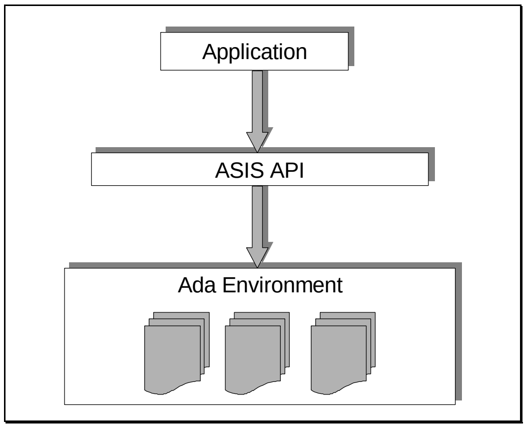

Figure 1. ASIS Reference Model

The mechanisms for creating and modifying this environment, however, are not specified by the language and are left to the implementation. To offset this implementation dependency, ASIS introduces the notion of a context that defines a set of compilation units and configuration pragmas to be processed by an ASIS application. A context may have one or more *Compilation_Units*. ASIS applications must query these *Compilation_Units* to perform useful analysis. The ASIS interface consists of a set of types, subtypes, and subprograms that provide the capability to query the Ada compilation environment for syntactic and semantic information. All ASIS subprogram interfaces are provided via Ada 95 child packages with the package ASIS at the root.

ASIS processes Ada source code via semantic and syntactic queries. Most ASIS queries process a specific construct with respect to the Ada Language Reference Manual (LRM) and take their names directly from the Ada LRM. The basic framework for all ASIS applications is the same and is shown in the code below.

```
   The_Current_Context : ASIS.Context;
begin
   -- Initialize and open the environment
   ASIS.Implementation.Initialize
      ( String'""));
   ASIS.Ada_Environments.Associate
     (The_Context => The_Current_Context,
      Name        => "ASIS Example" );
   ASIS.Ada_Environments.Open
      ( The_Current_Context );

   -- Get a compilation Unit Perform semantic
   -- queries on the unit as desired

   ASIS.Ada_Environments.Close
      ( The_Current_Context );
   ASIS.Ada_Environments.Dissociate
      ( The_Current_Context );
   ASIS.Implementation.Finalize;
end;
```

ASIS applications gather information about Ada compilation units through structural queries that provide top-down and bottom up decomposition of a compilation unit based on the unit's syntactic structure. ASIS structural queries are divided into black-box queries and white-box queries. Black-box queries produce information about compilation units as a whole and white-box queries produce information about the lexical elements within a compilation unit.

The basic lexical entity in ASIS is the element. Elements correspond to the nodes of a hierarchical tree representation of an Ada program. ASIS provides the ability to visit each element within the tree and query information about the element. The set of queries valid for a particular element depends on the specific kind of element being queried. Each of these subelements divided further into more specific elements. For example, A_Declaration can be one of the following elements:

- An_Ordinary_Type_Declaration,
- A_Subtype_Declaration,
- A_Constant_Declaration, etc.

Most of the names of the elements and the queries that operate on them are consistent with the terminology found in the Ada LRM. Classifying elements within an ASIS application is straightforward and can be accomplished as shown below.

```
case ASIS.Elements.Element_Kind
   ( Current_Element ) is
 when ASIS.A_Declaration =>
    case ASIS.Elements.Declaration_Kind
         ( Current_Element ) is
      when ASIS.A_Constant_Declaration =>
         -- Take appropriate action
      when others =>
         null;
     end case;
 when ASIS.A_Statement =>
   case ASIS.Elements.Statement_Kind
      ( Current_Element ) is
     when ASIS.An_Assignment_Statement =>
       -- Take appropriate action
     when others =>
            null;
   end case;
 when others =>
   null;
end case;
```

Once classified, the elements can be further processed as required by the application.

3. ASIS-BASED DETECTION

The focus of this project was to examine each of the restrictions proposed by [4][5] and determine which of the restrictions can be detected using the ASIS interface. The process in which the analysis was performed and the results of that analysis follows.

3.1 Analysis Process

Determining which of the proposed guidelines can be detected using ASIS requires a thorough understanding of the ASIS interface. To the credit of ASIS, the majority of the queries use the same terminology as that found in the Ada 95 LRM. Analyzing a particular restriction simply requires the Ada syntax corresponding to the restriction be broken down into its corresponding parts to form an element tree. For example, consider the restriction *No Task Allocators* and the following code example:

```
① task type Task_Type;
  type Task_Ptr is access
      Task_Type;
  My_Task : Task_Ptr
      := new Task_Type;
```

An element tree for line 3 is shown in Figure 2.

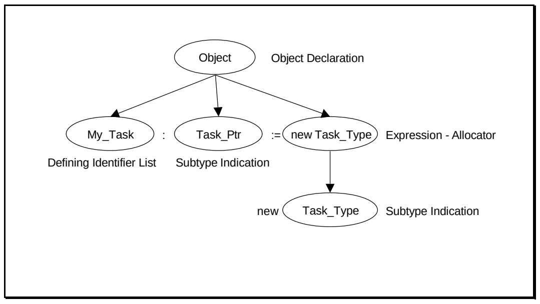

Figure 2. Example Element Tree

Once the element tree is determined, the next step is to map the elements in the element tree into the appropriate ASIS queries. Checking for the *No Task Allocators* restriction in this example ultimately requires that the application determine the type of *subtype indication* of the *allocator expression*. The progression of ASIS queries to arrive at that type is as follows:

① ASIS.Declarations.Initialization_Expression
→ Expression – Allocator (new Task_Type)
ASIS.Expressions.Allocator_Subtype_Indication
→ Subtype Indication (Task_Type)
ASIS.Definitions.Subtype_Mark
→ Subtype Mark (Task_Type)
ASIS.Elements.Corresponding_Name_Declaration
→ Type Declaration (task Type Task_Type)
ASIS.Elements.Type_Kind
→ A_Task_Type_Declaration

The first operation returns the initialization expression for the variable declaration. In this example, the initialization expression is an allocation from a subtype. The second operation returns the subtype indication for that allocation expression. The third operation returns the subtype mark, the name of the subtype, for the subtype indication. The fourth operation returns the element corresponding to the declaration for that subtype. And the final operation simply checks to see what kind of type is being declared, which in this case is a task type. This basic process was repeated for each of the proposed restrictions. The results of the analysis follow.

3.2 Analysis Results

The results of this Phase I research indicates that, using the operations specified in the ASIS interface, it is feasible to automatically check Ada 95 source code against the majority of the proposed HRG guidelines [6]. Specifically, as shown in Table 1, it is possible to detect 49 out of 53 restrictions using ASIS. Of the four restrictions not detectable, three involve run-time behavior that is beyond the scope of static-analysis, and the fourth one can possibly be detected using certain algorithms if all of the source code is available.

Table 1. Analysis Results

Restriction	Detectable Using ASIS
No Tasking	Yes
No Task Entries	Yes
No Asynchronous Select	Yes
No Protected Types	Yes
No Protected Entries	Yes
No Asynchronous Control	Yes
No Dynamic Priorities	Yes
No Implicit Heap Allocators	Yes
No Task Allocators	Yes
No Abort Statements	Yes
No Nested Finalization	Yes
No Task Hierarchy	Yes
No Allocators	Yes
No Local Allocators	Yes
No Unchecked Deallocation	Yes
No Exceptions	Yes
No Unchecked Conversion	Yes
No Access To Subprograms	Yes
No Unchecked Access	Yes
No IO	Yes
No Delay	Yes
No Recursion	Yes*
No Reentrancy	No
No Dispatching	Yes
Immediate Reclamation	No
No Floating Point	Yes
No Fixed Point	Yes
No Run-time Errors	No
No Uninitialized Scalers	Yes
No Discriminants	Yes
No Derivation Or Extension From Partial View	Yes
No Extension Of Tagged Types Outside Unit Of Declaration	Yes
No Discriminant Changes	Yes
No Enumeration Representation Clauses	Yes
No Types of Nonbinary Modulus	Yes
No_Fixed Point Conversion	Yes
No Representation Clauses On Aliased Objects	Yes

Restriction	Detectable Using ASIS
No Others	Yes
Enforce Parenthesis	Yes
No Conversions Between Different Representations	Yes
No Representation Claused Enumerations As Loop Iterators	Yes
No Gotos	Yes
No Access Parameters	Yes
No Default Expressions	Yes
No Overriding Predefined Operators	Yes
No Use Of Construct Before Ellaboration	No
No Non Local References In Body Sequence Of Statements	Yes
No Private Primitive Subprograms	Yes
No Ada.Finalization	Yes
No Aliasing	Yes
No Ada.Exceptions	Yes
No Generics	Yes
No Representation Items	Yes

Of the 53 restrictions, our analysis concluded that ASIS could directly detect 47 of the restrictions, can partially detect one restriction (recursion), and cannot detect 4 restrictions (use of construct before elaboration, no run-time errors, no reentrancy, and immediate reclamation).

With respect to recursion, ASIS can only detect direct recursion in which a subprogram calls itself as shown in the following example.

```
procedure Foo is
begin
   Foo;
end Foo;
```

ASIS cannot detect recursion in which subprograms may be aliased as in the case where access_to_subprogram types are used as shown in the following example.

```
type Proc_Ptr is access procedure;

My_Proc_Ptr : Proc_Ptr;
X : Integer;

procedure Bar is
begin
 …
end Bar;

procedure Foo is
begin

   if X = 1 then
       My_Proc_Ptr := Foo'access;
   else
       My_Proc_Ptr := Bar'access;
   end;
   My_Proc_Ptr.all;
end;
```

In this example, depending on the value of My_Proc_Ptr, this could be a recursive call. However, the exact determination cannot be made until run-time and cannot be statically detected.

Detection of the *use of construct before elaboration* restriction requires knowledge of the elaboration order of the program, which is beyond the scope of ASIS. An ASIS tool can suggest the use of elaboration pragmas to be of further assistance. Detection of *no run-time errors* obviously depends on program execution and is beyond the scope of ASIS. Detection of *no reentrancy* depends on the program design and is beyond the scope of ASIS. Detection of immediate reclamation is theoretically possible if the entire program is available. ASIS could possible check that the last reference of a variable is some sort of reclamation call, though visual inspection of the code would still be warranted.

4. TOOL OVERVIEW

Another goal of this effort was to design an ASIS-based tool that could automatically check Ada 95 source code to ensure the code meets the proposed guidelines. At a high-level, the tool was partitioned into a graphical user interface (GUI) and an analysis tool engine utilizing the ASIS interface as shown in Figure 3.

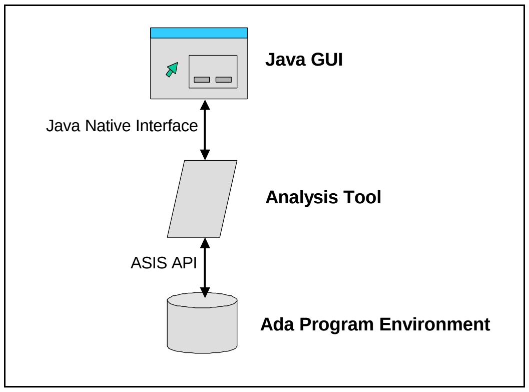

Figure 3. High-Level Design

The GUI was designed to execute on the Java Virtual Machine and interface with the analysis engine via the Java Native Interface (JNI). This facilitates portability among several popular software development environments and enables the tool to have one GUI implementation for all targeted platforms. Using the JNI, the GUI passes to the analysis tool information pertinent to the analysis. Typical information includes source file name(s) and the specific guidelines to enforce. The analysis engine, in turn, stores the results of the analysis in a string buffer which is then passed back to the GUI to be displayed to the user, saved to a file, or printed. It should be noted, however, that this is only a design and, although portions of this have been prototyped, the entire tool has not been implemented.

5. SUMMARY

The results of this Phase I research indicate that, using the operations specified in the ASIS interface, it is feasible to automatically check Ada 95 source code against the majority of the proposed HRG guidelines. The proposed static analysis tool greatly simplifies the process of checking Ada 95 source code against the HRG restrictions by providing the user an easy to use graphical user interface and informative output. In general, with the advent of Ada 95 and the increased attention on safety-critical systems, the ASIS interface coupled with the HRG guidelines fulfills an important need for systems designers. The results of this research indicate both the feasibility and the utility of a static analysis tool for safety-critical systems.

Future plans for this tool include completing the development of the analysis engine and the GUI. Once complete, the tool will undergo extensive testing using the Ada validation test suite as inputs. This should thoroughly exercise the tool’s capabilities to detect the proposed language restrictions.

6. ACKNOWLEDGEMENTS

This work was performed for the US Navy Space and Naval Warfare Command (SPAWAR) under Phase I Small Business Innovative Research (SBIR) contract N00039-98-C-0009. Our thanks to Mr. Duston Hayward, SPAWAR PMW-133, for his support and guidance throughout this project.

7. REFERENCES

[1] Peter G. Neumann, *Computer-Related Risks*, Addison-Wesley, Reading, MA, 1995.

[2] Dan Craigen, Mark Saaltink, and Steve Michell, *Ada 95 Trustworthiness Study: A Framework for Analysis*, TR-95-5499-02, 29 November 1995.

[3] *Ada 95 Reference Manual: Language and Standard Libraries: International Standard ISO/IEC 8652:1995*, S.T. Taft and R.A. Duff editors, Springer-Verlag, November 1997.

[4] Dan Craigen, Mark Saaltink, and Steve Michell, *Ada 95 Trustworthiness Study: Analysis of Ada 95 for Critical Systems*, Version 2.0, TR-97-5499-03a, 25 March 1997.

[5] Mark Saaltink and Steve Michell, *Ada 95 Trustworthiness Study: Guidance on the use of Ada 95 in the Development of High Integrity Systems*, Version 2.0, TR-97-5499-04a, 25 March 1997.

[6] B. A. Wichmann, et. al., "Guidance for the use of the Ada Programming Language in High Integrity Systems," *Ada Letters*, Vol. XVIII, No. 4, July/August 1998, pp. 47-94.

[7] *Ada Semantic Interface Specification*, ISO/IEC 15291 - Working Draft, 25 August 1997.

[8] William Pritchett and John Riley, *A Static Analysis Tool for High-Integrity Systems – Phase I Final Report*, DCS Corporation, Contract N00039-98-C-0009, 30 April 1998.