Code Analysis of Safety-Critical and Real-Time Software Using ASIS

Mr. William Currie Colket
The Mitre Corporation
1820 Dolley Madison Boulevard
McLean, Virginia 22102-3481
+1 (703) 883-7381
Colket@mitre.org

1. ABSTRACT

The Ravenscar Profile is a restricted tasking profile that supports applications requiring separate threads of control yet would satisfy the certification requirements of high-integrity (safety-critical) real-time systems. If the Ravenscar Profile were to be used for systems having safety-critical and real-time requirements, it would be valuable to demonstrate that the application satisfies the restrictions. Code analysis is an important technique to support this demonstration. Ada Semantic Interface Specification (ASIS) based tools provide an excellent capability for the automatic identification of violations to that set of the Ravenscar Profile restrictions, which can be determined through static code analysis. All but one of these restrictions can be identified using static code analysis using ASIS. This paper provides an approach to building such an ASIS-based tool. This tool might promote the use of automatic tools for the analysis of the Ravenscar Profile and other tasking profiles to support safety-critical and real-time requirements. This paper should be viewed as work in progress.

SIGAda'99 10/99 Redondo Beach, CA, USA

1.1 Keywords
Ada Language, ASIS, Code Analysis, High Integrity, Ravenscar Profile, Real-Time, Safety-Critical, Tasking.

2. INTRODUCTION

The introduction provides a background on the Ravenscar Profile, Code Analysis, and ASIS.

2.1 The Ravenscar Profile

The Ravenscar Profile is an important product of the 8th International Real-Time Ada Workshop (IRTAW) held in Ravenscar, North Yorkshire, England from 8-10 April 1997[2]. The goals of this workshop covered a number of session areas, which addressed Tasking Profiles, Distributed Systems and Fault Tolerant Systems, Outstanding Language Issues, and Object-Oriented Programming and Real-Time. The Ravenscar Profile was a direct result of the Tasking Profiles session. The objective of this session was to identify one or more restricted tasking profiles that would satisfy the certification requirements of high-integrity (safety-critical) real-time systems. Such a profile would also be likely to offer improved performance.

Aspects of both preemptive and a non-preemptive "cooperative" tasking models were addressed. The preemptive tasking model has important benefits in both expressiveness and schedulability (very important justification for moving from a traditional cyclic execution model to an Ada tasking model). Even the non-preemptive tasking model has advantages over a cyclic execution model such as the ability to break up long tasks to improve schedulability without having to be concerned with code to explicitly save and restore the state of the task.

There is a significant part of the safety-critical community who are not yet ready to trust a full preemptive scheduling model, yet are ready to use a cooperative co-routine model of concurrent programming. The IRTAW participants reviewed each of the Ada tasking features and discussed the merits of possible restrictions with respect to both certification and performance. The result was a set of 17 restrictions named the Ravenscar Profile. The model is so named as the IRTAW was held in Ravenscar and the venue could be easily converted into the following acronym: Reliable Ada Verification Executive Needed for Scheduling Critical Applications in Real-time.

2.2 Code Analysis

Code analysis can be defined as the inspection of software source code to extract information about the software to predict system behavior and performance. Such information can pertain to individual software elements (e.g., standards compliance, test coverage), the element attributes (e.g., quality, correctness, size, metrics), and element relationships (e.g., complexity, dependencies, data usage, call trees); thus, code analysis can support documentation generation, code review, maintainability assessment, reverse engineering, and other software development activities. Boeing has identified the following promises of automated code analysis:[4]

- Promote ***discipline and consistency*** during development, increasing productivity and reducing unintended variation.
- Provide empirical evidence and ***metrics*** for process monitoring and improvement.
- Supplement code ***inspection and review***, diversifying beyond the limitations of testing or manual checking.
- Preserve ***architectural integrity*** in the software as compromises are made during development.
- Avoid violations of ***coding standards***, such as the use of inefficient language constructs.
- Increase the ***correctness and quality*** of delivered software, reducing defects via comprehensive assessment.
- Enhance ***safety and security*** by applying formal methods to verify assertions in program code.
- Expedite ***program comprehension*** during maintenance, for engineers new to the code.
- Support ***reengineering and reuse*** of legacy code, reducing costs.
- Result in ***reduced risk*** to budget and schedule.

Related to the safety and security aspects of code analysis, the ISO/IEC JTC1/SC22 WG9 Safety and Security Rapporteur Group (HRG) is developing an ISO/IEC report to serve as a guide on the use of the Ada language for building high integrity systems.[8] The report recognizes that high integrity systems must be shown to be fully predictable in operation and have all the properties required of them. It also states that this can only be achieved by analyzing the software. Analysis is identified as one of four approaches required by current standards and guidelines. The report describes ten analysis methods, which are required in different combinations by various standards. These are: control flow, data flow, information flow, symbolic execution, formal code verification, range checking, stack usage timing analysis, other memory usage, and object code analysis.

2.3 ASIS

The Ada Semantic Interface Specification (ASIS) was developed in response to code analysis requirements for the Ada Language. ASIS is an interface between an Ada environment as defined by ISO/IEC 8652 (the Ada 95 Reference Manual)[6] and any tool requiring information from this environment. An Ada environment includes valuable semantic and syntactic information useful for performing static code analysis. ASIS-based tools can support code analysis to assure high quality systems both during initial code development and especially for independent verification and validation.

The ASIS 83 interface was developed by the Association for Computing Machinery (ACM's) Special Interest Group on Ada (SIGAda) through its volunteer effort in the ASIS Working Group (ASISWG). ASISWG has continued this important work for Ada 95 in conjunction with the ISO/IEC JTC1/SC22 WG9 ASIS Rapporteur Group (ASISRG) to standardize ASIS as an international standard for Ada 95 [7]. ASIS is now available as an International Standard denoted as:

> ISO/IEC 15291:1999 Information technology
>
> — Programming languages
>
> — Ada Semantic Interface Specification (ASIS)

A variety of ASIS-based tools have been developed. These include: automated code monitors, browsers, call tree tools, code reformators, coding standards compliance tools, correctness verifiers, debuggers, dependency tree analysis tools, design tools, document generators, metrics tools, quality assessment tools, reverse engineering tools, re-engineering tools, style checkers, test tools, timing estimators, and translators. There is even an approach to translate an application using the Ada 95 Distributed System Annex (DSA) to the Common Object Request Broker Architecture (CORBA) Interface Definition Language (IDL).[10]

The WG9 Safety and Security Rapporteur Group (HRG) see the value of using ASIS to analyze conformance of safety critical applications to their safety and security guidelines [8]. There have already been several papers addressing static analysis of systems with high-integrity requirements using ASIS (See [9] and[11]).

The ASIS interface is rather mature. A test of one implementation has demonstrated the capability to reproduce source code which when compiled would produce the same object code as the original code. This was demonstrated using the Ada Compilation Verification

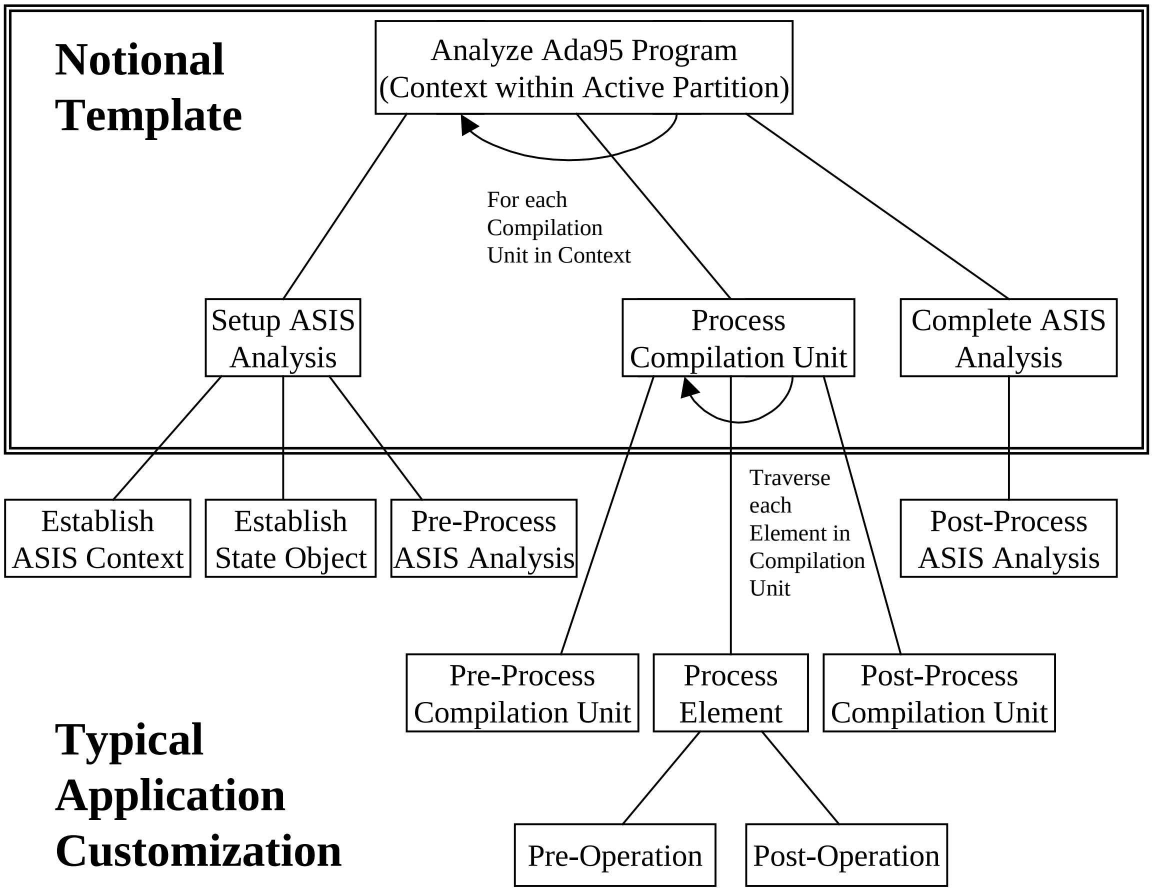

Figure 1 - Notional ASIS Application

Capacity (ACVC).[12] A powerful tool called the Interactive ASIS Interpreter (ASIStint) has been built to support the development of ASIS tools.[5]

3. Structure of ASIS Application

An ASIS application designed to support the code analysis for an Ada95 program might appear as shown in Figure 1. Inside the box, labeled Notional Template is the ASIS application code, which should be common to many code analysis applications. Externals to this box are those objects needed to customize the notional template; these are oriented to the specific needs of the ASIS code analysis. The ASIS application is divided into 3 major areas:

1. **Setup ASIS Analysis** – during the setup, the ASIS program initializes ASIS, establishes ASIS Context, establishes a State Object, associates the ASIS Context with this application, opens the ASIS Context, and performs any preprocessing needed to support the ASIS analysis.
2. **Process Compilation Units** – the main processing of most ASIS applications is to analyze each element in every compilation unit within the ASIS Context. The capability to perform pre and post compilation unit processing is provided.
3. **Complete ASIS Analysis** – the ASIS program should perform any post processing needed to support the ASIS analysis, close the ASIS Context, dissociate the ASIS Context, and finalize ASIS.

Black box testing on the context is typically performed at the Pre-Process ASIS Analysis level. Black box testing on each compilation unit is typically performed at the Pre-Process Compilation Unit processing. White box testing is typically performed at the Pre-Operation portion of the Process Element processing. Before discussing code analysis of the Ravenscar Profile, it is important to discuss several ASIS concepts.

3.1 ASIS Context

The ASIS Context is an important concept in understanding analysis of the Ravenscar Profile. The ASIS Context is the set of compilation units in the Ada Active partition, which will be analyzed by this ASIS application. Not to be confused with Ada 95 context clauses, the ASIS Context defines a set of compilation units and configuration pragmas processed by an ASIS application. ASIS provides any information from a context by treating this set as if its elements make up an environment declarative part by modeling some view (most likely one of the views of the underlying Ada implementation) on the environment.

3.2 The ASIS Environment

ASIS consists of a package Asis with a number of child packages. The major child packages include Errors, Compilation_Units, Ada_Environments, Implementation, Exceptions, Elements, Iterator, Declarations, Expressions, Clauses, Definitions, Statements, Text, and Ids. The ASIS architecture is described in detail in [3] and [7]. The ASIS Home Page provides a good discussion on each of these.[1]

3.3 ASIS Queries

ASIS has a powerful set of queries, which facilitates the analysis of a full range of syntactic and semantic information. Of the 367 queries, most support the analysis of Ada 95's syntax. The most valuable aspect of ASIS is the capability to support semantic analysis. Together, syntactic and semantic analysis provide for queries that can provide information on an element concerning its:

- Element Kind
- Component Elements
- Enclosing Elements
- Enclosing Compilation Unit
- Related Elements (e.g., Corresponding Type Declaration, Corresponding Name Definition, Corresponding Called Function, Corresponding Called Entity, Corresponding Type, Corresponding Body, Corresponding Entry)
- Text Span and Text Image (to print out original source code)

3.4 Element Traversal

The heart of any ASIS Analysis is the processing performed on an element during the element traversal in a logical syntax tree representing each compilation unit. The generic Asis.Iterator.Traverse_Element supports two generic procedures called procedure Pre_Operation and procedure Post_Operation. The Pre-Operation is evoked when Traverse_Element initially lands on an element, before child elements are processed. The Post_Operation is evoked when Traverse_Element has processed all child elements and is ready to return to the current element's parent. The Pre_Operation procedure contains the processing most people associate with ASIS code analysis. The Post_Operation is useful for providing statistics for an Element's children. Asis.Iterator.Traverse_Element contains a generic parameter called State_Information

3.5 State Information

During the process element traversal, the generic Asis.Iterator.Traverse_Element maintains an object passed each time the instantiated procedure is called. The type of this object, known as State_Information is limited private and can be instantiated to suit a variety of needs. This object is useful for maintaining things such as counts of objects found for metric analysis, counts of objects found for restriction checking, elements visited with processed information, and for a host of other purposes.

4. ASIS Analysis of Ravenscar Profile

The restrictions for the Ravenscar Profile are identified in Table 1. In the right column is an indication whether ASIS can support automatic code analysis of the restriction, and if so, whether it is done as Black Box testing at the Context level, Black Box testing at the Compilation Unit (CU) level, or White Box testing at the CU level. Only R13 will require dynamic analysis, not possible by ASIS.

4.1 Reporting of Violations to Ravenscar Profile Restrictions

Violations and Warnings of possible Ravenscar Profile violations are provided in the following two procedures using the number provided in column 1 as the Violation Number:

```
procedure Report_Violation
   (Violation_Number : in Wide_String;
    Violated_Element : in Asis.Element) is

begin

   Put("Violation of Ravenscar Profile: ");
   Put(Violation_Number);
   Put(" at line: ");
   Put(Asis.Text.Line_Number'Wide_Image
      (Asis.Text.First_Line_Number
       (Violated_Element)));
   New_Line;

end Report_Violation;

procedure Report_Warning
   (Violation_Number : in Wide_String;
    Violated_Element : in Asis.Element) is

begin

   Put("Warning of Possible Violation
        of Ravenscar Profile: ");
   Put(Violation_Number);
   Put(" at line: ");
   Put(Asis.Text.Line_Number'Wide_Image
      (Asis.Text.First_Line_Number
       (Violated_Element)));
   New_Line;

end Report_Warning;
```

	Forbidden Features	Detectable Using ASIS
R1	Task types and object declarations other than at the library level. Thus, there is no hierarchy of types.	White Box
R2	Unchecked deallocation of protected and task objects (and hence finalization). Dynamic allocation of such objects may be allowed, but only if the sequential part of the high-integrity language profile allows dynamic allocation of other objects.	White Box
R3	Requeue.	White Box
R4	Asynchronous Transfer of Control (ATC) via the *select then abort* statement.	White Box
R5	Abort.	White Box
R6	Task entries.	White Box
R7	Dynamic priorities.	Black Box at CU
R8	Calendar.	Black Box at CU
R9	Relative delays.	White Box
R10	Protected types other than at the library level.	White Box
R11	Protected types with more than one entry.	White Box
R12	Protected entities with barriers other than a single Boolean variable declared within the same protected type.	White Box
R13	Attempts to queue more than one task on a single protected entry.	Limited Support
R14	Locking policies other than *Ceiling locking.*	Black Box at Context
R15	Scheduling policies other *than FIFO within priorities*.	Black Box at Context
R16	All forms of select statement.	White Box
R17	User-defined task attributes.	Black Box at CU

Table 1 – Forbidden Tasking Features in the Ravenscar Profile

4.2 Black Box Analysis of Ravenscar Restrictions on ASIS Context

Table 1 identifies several restrictions, R14 and R15, which can be easily addressed using black box analysis at the ASIS Context Level. The appropriate location for this analysis is at the Pre-Process ASIS Analysis level as shown in Figure 1. These restrictions are associated with Configuration Pragmas for the entire ASIS Context.

```
procedure Pre_Process_ASIS_Analysis
    (My_Context : in     Asis.Context;
     My_State   : in out My_State_Information) is

begin

     Put_Line ("Beginning Asis Analysis");
     Check_Pragma_Restrictions(My_Context);

end Pre_Process_ASIS_Analysis;
```

The procedure Check_Pragma_Restrictions uses the function Configuration_Pragmas to get all of the configuration pragmas and then loops through each pragma, checking for the violation of the restriction.

```
procedure Check_Pragma_Restrictions
   (My_Context : in Asis.Context) is

   Pragma_List: constant
      Asis.Pragma_Element_List :=
      Asis.Elements.Configuration_Pragmas
         (My_Context);
   The_Pragma: Asis.Pragma_Element;

begin

   for I in Pragma_List'range loop

      The_Pragma := Pragma_List (I);
      Check_Each_Pragma(The_Pragma);

   end loop;

end Check_Pragma_Restrictions;
```

Procedure Check_Each_Pragma and function Is_Association_Equal are used to perform the pragma checking. A violation of R14 occurs when pragma Locking_Policy is not set to Ceiling_Locking. A violation of R15 occurs when pragma Queuing_Policy is not FIFO_Queuing or when pragma Task_Dispatching_Policy is not FIFO_Within_Priorities. Procedure Check_Each_Pragma checks for these pragmas and when found, queries for the pragma argument association. The function Is_Association_Equal tests to see if the named association is equivalent to the found association. As case may be an issue for the implementation's representation of the association, the associations of both parameters are converted to lower case for the comparison using Ada.Strings.Wide_Maps.Wide_Constants.

```
function Is_Association_Equal
   (Named_Association : in Wide_String;
    Found_Association : in Asis.Association)
    return Boolean is

   package ASWW renames
     Ada.Strings.Wide_Maps.Wide_Constants;

begin

   if (Ada.Strings.Wide_Fixed.Translate
      (Asis.Expressions.Name_Image
      (Asis.Expressions.Actual_Parameter
      (Found_Association)), ASWW.Lower_Case_Map))
      =
      (Ada.Strings.Wide_Fixed.Translate
      (Named_Association, ASWW.Lower_Case_Map))

   then return true;

   else return false;
   end if;

end Is_Association_Equal;


procedure Check_Each_Pragma
   (Elem : in Asis.Pragma_Element) is

   Pragma_Kind : Asis.Pragma_Kinds :=
      Asis.Elements.Pragma_Kind(Elem);
   The_Association : Asis.Association;

begin

  case Pragma_Kind is

    when Asis.A_Locking_Policy_Pragma =>
       The_Association :=
       Asis.Elements.Pragma_Argument_Associations
       (Elem)(1);
       if Is_Association_Equal
          ("Ceiling_Locking", The_Association)
       then
          Report_Violation("R14", Elem);
       end if;

    when Asis.A_Queuing_Policy_Pragma =>
       The_Association :=
       Asis.Elements.Pragma_Argument_Associations
       (Elem)(1);
       if Is_Association_Equal
          ("FIFO_Queuing", The_Association)
       then
          Report_Violation("R15", Elem);
       end if;

    when Asis.A_Task_Dispatching_Policy_Pragma =>
       The_Association :=
       Asis.Elements.Pragma_Argument_Associations
       (Elem)(1);
       if Is_Association_Equal
          ("FIFO_Within_Priorities ",
            The_Association)
       then
          Report_Violation("R15", Elem);
       end if;

   when others =>
       null;

  end case;

end Check_Each_Pragma;
```

4.3 Black Box Analysis of Ravenscar Restrictions on Compilation Unit

Table 1 identifies several restrictions, R7, R8, and R17, which can be easily addressed using black box analysis at the ASIS Compilation Unit level. The appropriate location for this analysis is at the Pre-Process Compilation Unit level as shown in Figure 1. These restrictions are associated with context clauses.

```
procedure Pre_Process_Compilation_Unit
   (CU        : in     Asis.Compilation_Unit;
    My_State  : in out My_State_Information) is

begin

   Put_Line ("Processing Unit:" &
      Asis.Unit_Kinds’Wide_Image
      (Asis.Compilation_Units.Unit_Kind(CU))
      & ": " & (
      Asis.Compilation_Units.Unit_Full_Name(CU)));
   New_Line;

   Check_Context_Clause_Restrictions(CU);

end Pre_Process_Compilation_Unit;
```

Procedure Check_Context_Clause_Restrictions obtains the list of Ada context clauses for the compilation unit and then loops through each context clause checking for violations.

```
procedure Check_Context_Clause_Restrictions
   (CU: in Asis.Compilation_Unit) is

   Context_Clause_List: constant
      Asis.Context_Clause_List :=
      Asis.Elements.Context_Clause_Elements(CU);

begin

   for I in Context_Clause_List’Range loop

      case ASIS.Elements.Clause_Kind
           (Context_Clause_List(I)) is

         when Asis.A_With_Clause =>
              Check_Each_Context_Clause
              (Context_Clause_List(I));

         when others =>
              null;
      end case;

   end loop;

end Check_Context_Clause_Restrictions;
```

The procedure Check_Each_Context_Clause and the function Is_Library_Unit_Equal are used to perform the context clause checking. Each context clause is tested to see if it contains the name of the package, which would constitute a violation, if used. The presence of Ada.Dynamic_Priorities violates R7; the presence of Ada.Calendar violates R8; and the presence of Ada.Task_Attributes violates R17. It is assumed that if the package is part of the context, it is used in violation of the restriction. This may, in fact, not be the case as

Ada.Calendar may be used for non-timing activities associated with reporting dates. Additional ASIS processing at the White Box level could be used to ascertain valid use from a true violation of the Ravenscar Profile restriction.

The function Is_Library_Unit_Equal tests to see if the named library unit is equivalent to the found library unit. As with the case for Is_Association_Equal, both parameters are converted to lower case for the comparison.

```
function Is_Library_Unit_Equal
   (Named_Library_Unit : in Wide_String;
    Found_Library_Unit : in Asis.Program_Text)
    return Boolean is

   package ASWW renames
     Ada.Strings.Wide_Maps.Wide_Constants;

begin

   if Ada.Strings.Wide_Fixed.Translate
      (Found_Library_Unit, ASWW.Lower_Case_Map)
      =
      Ada.Strings.Wide_Fixed.Translate
      (Named_Library_Unit, ASWW.Lower_Case_Map)

   then return true;
   else return false;
   end if;

end Is_Library_Unit_Equal;


procedure Check_Each_Context_Clause
   (The_Context_Clause: in Asis.Context_Clause) is

begin

  if Is_Library_Unit_Equal
     (Extract_Selected_Component
      (The_Context_Clause),
     "Ada.Dynamic_Priorities")
  then Report_Violation("R7", The_Context_Clause);
  end if;

  if Is_Library_Unit_Equal
     (Extract_Selected_Component
      (The_Context_Clause), "Ada.Calendar")
  then Report_Violation("R8", The_Context_Clause);
  end if;

  if Is_Library_Unit_Equal
     (Extract_Selected_Component
      (The_Context_Clause), "Ada.Task_Attributes")
  then Report_Violation("R17",The_Context_Clause);
  end if;

end Check_Each_Context_Clause;
```

The function Extract_Selected_Component should be viewed as a secondary layer function built on top of ASIS.

4.4 White Box Analysis of Ravenscar Restrictions on Compilation Unit

Table 1 identifies several restrictions, R1, R3, R4, R5, R9, R10, and R16, which can be addressed using white box analysis at the ASIS Compilation Unit level. The appropriate location for this analysis is at the Pre_Operation level to process elements as shown in Figure 1. These restrictions are associated with the remaining Ravenscar Profile restrictions identifiable by ASIS.

```
procedure Pre_Operation
   (An_Element : in     Asis.Element;
    Control    : in out Asis.Traverse_Control;
    My_State   : in out My_State_Information) is

begin

    Check_Statement_Restrictions(An_Element);
    -- for R3, R4, R5, R9, and R16

    Check_Library_Level(An_Element);
    -- for R1 and R10

    Check_Entry_Call_In_Task(An_Element);
    -- for R6

    Check_Unchecked_Deallocation(An_Element);
    -- for R2

    Check_Protected_Types_With_Multiple_Entries
    (An_Element);
    -- for R11

    Check_Barriers_Other_Than_Boolean(An_Element);
    -- for R12

end Pre_Operation;
```

Note: The last four checks for R2, R6, R11, R12 represent part of the work in progress and are not presented in this paper. The author believes these checks can easily be performed with static analysis using ASIS.

4.4.1 Checking for Restricted Statements

There are a number of restrictions, R3, R4, R5, R9, and R16, which can be easily checked through the analysis of the Statement_Kind. The presence of either a requeue statement or a requeue statement with abort is a violation of R3. The presence of an asynchronous select statement is a violation of R4, which has the select then abort syntax. The presence of an abort statement or a requeue statement with abort is a violation of R5. The presence of a delay relative statement is a violation of R9. The presence of a selective accept statement, or a timed entry call statement, or a conditional entry call statement, or an asynchronous select statement is a violation of R16. It should be noted that some statements might trigger multiple restriction reports. This is intentional as should a restriction be removed, the identification of the remaining restriction(s) will be unaffected.

```
procedure Check_Statement_Restrictions
          (Elem : in Asis.Element) is

   Stmt_Kind : Asis.Statement_Kinds :=
     Asis.Elements.Statement_Kind(Elem);

begin

  case Stmt_Kind is

    when Asis.A_Requeue_Statement =>
      Report_Violation("R3", Elem);

    when Asis.Ansynchronous_Select_Statement =>
      Report_Violation("R4", Elem);
      Report_Violation("R16", Elem);

    when Asis.An_Abort_Statement =>
      Report_Violation("R5", Elem);

    when Asis.Requeue_Statement_With_Abort =>
      Report_Violation("R3", Elem);
      Report_Violation("R5", Elem);

    when Asis.A_Delay_Relative_Statement =>
      Report_Violation("R9", Elem);

    when Asis.A_Selective_Accept_Statement |
         Asis.A_Timed_Entry_Call_Statement |
         Asis.A_Conditional_Entry_Call_Statement
         =>
      Report_Violation("R16", Elem);

    when others =>
      null;
  end case;

end Check_Statement_Restrictions;
```

4.4.2 Checking for Library Level Tasks and Objects

The checking for tasks and protected types at the library level are performed using the procedure Check_Library_Level and the function Is_Library_Level. Is_Library_Level is defined here as occurring when the enclosing compilation unit happens to be equal to the enclosing element. Other definitions are possible.

```
function Is_Library_Level
  (Elem : Asis.Element) return Boolean is

begin

   if Asis.Elements.Is_Equal
      (Asis.Elements.Enclosing_Element(Elem),
       Asis.Elements.Unit_Declaration(
       Asis.Elements.Enclosing_Compilation_Unit
         (Elem)))
   then
      return true;
   else
      return false;
   end if;

end Is_Library_Level;
```

The procedure Check_Library_Level checks the declaration kind to see if a task is present through a declaration of either a task type declaration or a single task declaration. If so, a violation of R1 is reported if Is_Library_Level returns false. The procedure Check_Library_Level performs a similar analysis for a protected type declaration and a single type declaration, reporting R12 violation when not at the library level. The check for restriction R1 is only partially complete. It should also include a check for task object declarations being at the library level. This extra check is work in progress.

```
procedure Check_Library_Level
      (Elem : Asis.Element) is

begin

  case Asis.Elements.Declaration_Kind (Elem) is

    when Asis.A_Task_Type_Declaration |
         Asis.A_Single_Task_Declaration =>
      if not Is_Library_Level(Elem) then
         Report_Violation("R1", Elem);
         -- Note: Does not catch task object
         -- declarations not at Library Level.
      end if;

    when Asis.A_Protected_Type_Declaration |
         Asis.A_Single_Protected_Declaration =>
      if not Is_Library_Level(Elem) then
         Report_Violation("R10", Elem);
      end if;

    when others =>
      null;
  end case;

end Check_Library_Level;
```

5. Notional Template

The notional template for the ASIS application to perform static code analysis on the Ravenscar Profile restrictions is included below. Details of this notional template may be found in either the ASIS Standard[7] or the ASIS Home Page[1] following the link to Tutorials. It is presented below as a convenience to the reader. It provides the context for instantiations of:

- Pre Process ASIS Analysis
- Pre-Process Compilation Unit
- Process Element
- Post Process Compilation Unit
- Post Process ASIS Analysis

Of interest, the procedure Process_Element is created through the instantiation of Asis.Iterator.Traverse_Element with entities residing in package ASIS_Customization. The notional example is bracketed by three sets of Asis calls to initialize/finalize the ASIS implementation, to Associate/Dissociate an ASIS Context, and to Open/Close that ASIS Context. A block statement is used to facilitate the creation of a list of compilation units in the ASIS context. Then a loop is established to process each compilation unit using a call to Process_Element. ASIS can raise a number of exceptions. The notional template contains a test for the raising of each exception and uses package Asis.Implementation to report both thee Diagnosis and the Status of the condition raising the exception.

```
Procedure ASIS_Analysis_Template is

   My_Context: Asis.Context;
   My_State:
      ASIS_Customization.My_State_Information;
   Control: Asis.Traverse_Control:= Asis.Continue;
   package AC renames ASIS_Customization;

   procedure Process_Element is new
      Asis.Iterator.Traverse_Element
       (ASIS_Customization.My_State_Information,
        ASIS_Customization.Pre_Operation,
        ASIS_Customization.Post_Operation);

begin

  Asis.Implemntation.Initialize;
  Asis.Ada_Environments.Associate(My_Context, "");
  Asis.Ada_Environments.Open (My_Context);

  ASIS_Customization.Pre_Process_ASIS_Analysis
     (My_Context, My_State);

  declare

    Unit_List: constant
      Asis.Compilation_Unit_List :=
         Asis.Compilation_Units.Compilation_Units
            (My_Context);
    CU: Asis.Compilation_Unit;

  begin

    for I in Unit_List'Range loop

      CU := Unit_List (I);

      case Asis.Compilation_Units.Unit_Origin(CU)
        is

          when Asis.An_Application_Unit =>
             AC.Pre_Process_Compilation_Unit
               (CU, My_State);
             Process_Element
             (Asis.Elements.Unit_Declaration
               (CU), Control, My_State);
             AC.Post_Process_Compilation_Unit
               (CU, My_State);

          when others => null;
       end case;

    end loop;

  ASIS_Customization.Post_Process_ASIS_Analysis
   (My_Context, My_State);

  Asis.Ada_Environments.Close (My_Context);
  Asis.Ada_Environments.Dissociate (My_Context);
  Asis.Implementation.Finalize;

exception

  when ASIS_Inappropriate_Context
    | ASIS_Inappropriate_Container
    | ASIS_Inappropriate_Compilation_Unit
    | ASIS_Inappropriate_Element
    | ASIS_Inappropriate_Line
    | ASIS_Inappropriate_Line_Number
    | Asis.Exceptions.ASIS_Failed
      => Put (Asis.Implementation.Diagnosis);
         New_Line;
         Put ("Status Value is ");
         Put (Asis.Errors.Error_Kinds'Wide_Image
                (Asis.Implementation.Status));
         New_Line;
   when others =>
        Put_Line ("Asis Application failed
           because of non-ASIS reasons");

end ASIS_Analysis_Template;
```

6. Summary

The approach to using ASIS-based code analysis to identify the restrictions in the Ravenscar Profile is sound. All but one of the Ravenscar Profile tasking restrictions can be addressed using static code analysis. Restriction R13, was the only restriction believed to require dynamic analysis. Even here, ASIS static analysis might be useful to identify potential queuing opportunities, which must be analyzed using dynamic analysis techniques. An approach was presented for about two thirds of the restrictions using either Black Box or White Box code analysis. The restrictions: R2, R6, R11, and R12 are believed to be analyzable using White Box ASIS analysis and remain part of the work in progress. The restriction R1 only is partially addressed, and the remaining portion is work in progress. The approach presented in this paper could be used to identify violations to other tasking profiles to support the analysis of safety-critical and real-time requirements.

7. ACKNOWLEDGMENTS

I thank the ACM SIGAda ASIS Working Group (ASISWG) and the ISO/IEC JTC1/SC22 WG9 ASIS Rapporteur Group (ASISRG) for their support in making ASIS a reality. I also want to specially thank Dr. William M. Thomas, Mr. Clyde Roby, and Ms. Maricarol Jacobi who reviewed this paper and provided excellent comments.

8. REFERENCES

[1] Association of Computing Machinery (ACM) Special Interest Group on Ada (SIGAda) ASIS Home Page. http://www.acm.org/sigada/WG/asiswg/

[2] Burns, A., Baker, T., Vardanrga, T. Proceedings of the 8th International Real-Time Ada Workshop, *Session Summary: Tasking Profiles*, ACM Ada Letters, September/October 1997, Vol. XVII, Number 5, pp. 5-7.

[3] Colket, C., Thomas, B., *et al*, *Architecture of ASIS, A Tool to Support Code Analysis of Complex Systems*,

ACM Ada Letters, January/February 1997, Vol. XVII, Number 1, pp. 35-40.

[4] Cooper, D., *ASIS-Based Code Analysis Automation*, Ada Letters, November/December 1997, Volume XVII, No. 6.

[5] Fofanov, V., Rybin, S., Strohmeier A., *ASIStint: An Interactive ASIS Interpreter*, Proceedings of the ACM SIGAda Annual International Conference (TRI-Ada'97), St. Louis, Missouri, pp. 205-209, November 1997.

[6] ISO/IEC 8652:1995 Information technology — Programming languages — Ada.

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

[8] ISO/IEC DTR 15942 — Programming Languages — Guide for the Use of the Ada Programming Language in High Integrity Systems, dated 11 May 1999.

[9] Michell, S., Saaltink, M., Wichmann, B., *Looking into Safety with the Safety and Security Rapporteur Group*, Proceedings of the ACM SIGAda Annual International Conference (SIGAda'98), 8-12 November 1998, Washington D.C., pp. 7-11.

[10] Pautet, L., Quinot, T., Tardieu, S., *CORBA & DSA: Divorce or Marriage*, 1999 Ada-Europe International Conference on Reliable Software Technologies Proceedings, Santander, Spain, LNCS no. 1622, Springer, pp. 211-225, June 1999.

[11] Pritchett IV, W., Riley, J., *An ASIS-Based Static Analysis Tool for High-Integrity Systems*, proceedings of the ACM SIGAda Annual International Conference (SIGAda'98), 8-12 November 1998. Washington D.C., pp. 12-17.

[12] Strohmeier, A., Fofanov, V., Rybin, S., Barbey, S., *Quality-for-ASIS: A Portable Testing Facility for ASIS*, 1998 Ada-Europe International Conference on Reliable Software Technologies Proceedings, Uppsala, Sweden, LNCS no. 1411, Springer, pp. 163-175, June 1998.