Ada and ASIS:
Justification of Differences in Terminology and Mechanisms

Sergey Rybin[1], Alfred Strohmeier[2]

[1] *Scientific Research Computer Center*
Moscow State University
Vorob'evi Gori
Moscow 119899, Russia
rybin@alex.srcc.msu.su

[2] *Swiss Fed Inst of Technology in Lausanne*
Software Engineering Lab
CH-1015 Lausanne, Switzerland
alfred.strohmeier@di.epfl.ch
http://lglwww.epfl.ch/

ABSTRACT. The Ada Semantic Interface Specification (ASIS) provides the basis for implementing portable tools aimed at analysing static properties of Ada source code. The main purpose of the Ada language definition is to specify the legality of compilation units and the behaviour of a program at run-time. While there is obviously a large common part in Ada and ASIS, the different goals also justify some specific terminology and mechanisms in ASIS. The article presents some of them.

ABSTRACT. Ada, ASIS, Compiler, Programming Environment, Software Engineering Environment.

1. Introduction

The Ada Semantic Interface Specification (ASIS) provides the basis for implementing portable tools aimed at analysing static properties of Ada source code. As stated in the ASIS Working Group's charter, the Ada Semantic Interface Specification (ASIS) [1] is an interface between an Ada environment, as defined by the Ada 95 Reference Manual (called RM95 below) [2], and any tool or application, called an ASIS application, requiring statically-determinable information from this environment. ASIS itself is defined as a set of self-documented Ada package specifications providing the types and operations, called ASIS queries, used to retrieve this kind of information and to deal with it.

In this paper we will describe and analyse some aspects, either important or quite subtle, of the current ASIS definition, which were discussed recently within the ASIS Working Group. We hope this discussion will show that the ASIS definition is nearing the maturity-level required for standardization.

In section 2, we will sketch the use of ASIS in order to give some concrete background to the following sections. Section 3 sums up the changes made to ASIS in response to the Ada language revision. In section 4, we then try to show that there are some fundamental differences between a language definition, i.e. the definition of Ada, and the interface definition of a tool, such as ASIS. The next three sections 5-7 discuss in detail the commonalities and differences of the Ada environment and ASIS context concepts, covering also the notions of compilation unit, configuration pragma, and inconsistency in an environment or context. Section 8 concludes this part by explaining how ASIS-for-GNAT implements ASIS contexts. We then proceed by giving examples of differences between the Ada language definition and ASIS, some well justified, others perhaps arguable. Sometimes a concept is missing as such in ASIS, but it can be rebuilt indirectly; e.g. implicit declarations of a declarative region can only be retrieved indirectly, as shown in section 9. To minimize the number of queries, ASIS sometimes unifies several concepts; examples are given in the three following sections 10-12, some well-justified, others more questionable. The paper finishes in section 13 with a short discussion of the current packaging structure of ASIS; the authors think that it could and should be improved.

2. Sketch of Use of ASIS

To give the reader some more concrete information about the use of ASIS, here is the sketch of some typical code:

```
-- Associate a Context with the external world,
-- and open it:
Asis.Ada_Environments.Associate(My_Context, ...);
Asis.Ada_Environments.Open(My_Context);

-- Now fetch the list of units in the context:
declare
   My_List_Of_Units: Asis.Compilation_Unit_List :=
      Asis.Compilation_Units.Compilation_Units
         (My_Context);
begin
   -- Now traverse the list of units
   -- for determining their kind, e.g.:
   for I in My_List_Of_Units'Range loop
      -- the function call
      --    Unit_Kind (My_List_Of_Units(I))
      -- will tell us if the unit is a package
      -- or a procedure or etc.
   end loop;
```

 Will be published in the Proceedings of TRI-Ada'96, December 3-7, 1996, Philadelphia, USA. Printed with permission.

3. From ASIS 87 to ASIS 95

ASIS for Ada 87 [5], known as ASIS version 1.1.1 or ASIS 87, was intended to be used as a de-facto standard.

Obviously, it was necessary to revise ASIS to take into account the changes in the Ada language definition. When starting the revision of ASIS, the following design goals were also agreed upon by the ASIS working group:

- Minimize cost of converting ASIS-based applications from ASIS 87 to ASIS 95.
- ASIS should be designed to be simple and uniform with minimal concepts and minimal operations.
- ASIS should allow for standardization at the international level.

The main changes in the ASIS definition are as follows:

- New queries were added for the new language features, such as child units and protected types, and old ones were revised to take into account language changes.
- Some queries of ASIS 87 were aggregated in order to minimize their total number.
- Whereas ASIS 87 was defined to be an interface to a program library, ASIS 95 is now an interface to an Ada environment, following the new ideas of Ada about the program structure and the compilation process.
- The set of package specifications defining ASIS was reorganized into hierarchies of child units.
- The terminology was revised and the comments in the ASIS package specifications were reworded to make them consistent with Ada 95.

4. About the Essence of Ada and ASIS

Clearly the Ada language definition, as described in RM95, and ASIS are different in nature and have different aims.

Ada defines the notion of a legal compilation unit, the notion of a legal complete partition and the dynamic semantics of a complete partition. It also defines some aspects of the process of compiling a compilation unit and some properties of the results of this process.

On the other hand, ASIS is more like the requirements definition for a tool which provide information about a set of Ada software components, which may or may not make up a complete partition. Moreover, as a tool, ASIS must provide management facilities which go beyond of what is needed in a language definition. E.g. ASIS needs names for designating the then-part and the else-part of an if statement.

Despite these differences in nature, the terminology used in Ada and ASIS should be the same whenever possible. Clearly, there should be no contradictions. Closeness in terminology eases understanding of ASIS by those who already know Ada, saves documentation effort, and is justified by maintenance, since every correction in Ada may impact ASIS.

Nevertheless, ASIS needs some new terms, even if they should be as few as possible. Here are some cases:

- To cut down the number of queries, ASIS sometimes needs to aggregate Ada-defined syntactic categories.
- To walk through a heterogeneous data structure, such as a list of compilation units and configuration pragmas, ASIS needs to introduce an unifying concept for item types in the structure.
- Finally, some concepts in Ada are vaguely defined compared to the needs of ASIS. It may then be better to introduce a new name rather than restrict the meaning of the Ada concept. The Ada environment and ASIS context concepts may serve as an example here, as we will see later.

5. Ada Environment and Compilation Units

One of the major points in the revision of ASIS was the replacement of the idea of an interface to an Ada program library in ASIS 1.1.1 by an interface to an Ada environment. This change has in fact some farther-reaching consequences than it may seem at a first look.

RM95 10.1.4(1) says: “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 any compilation. At run time, an environment forms the declarative_part of the body of the environment task of a partition”. And the next paragraph says: “The declarative_items of the environment are library_items appearing in an order such that there are no forward semantic dependencies.” RM95 10.1.4(5) then says: “When a compilation unit is compiled, all compilation units upon which it depends semantically shall already exist in the environment;...”

At a first look, there seems to be a contradiction between RM95 10.1.4(1,2) which defines an environment as a declarative_part and says that an environment contains library_items, but not compilation_units, and RM95 10.1.4(5) which speaks about the existence of compilation units in the environment.

But this apparent “contradiction” is resolved in RM95 10.1.1(9), which says: “The term *compilation unit* is used to refer to a compilation_unit. When the meaning is clear from context, the term is also used to refer to the library_item of a compilation_unit or to the proper_body of a subunit (that is, the compilation_unit without the context_clause and the **separate** (parent_unit_name)).”

This means that RM95 defines compilation_unit as a syntactical category, and at the same time it defines “compilation unit” (without underscore and typed in regular font) as another technical term, which may denote either one of the syntactical categories "compilation_unit", "library_item" or "proper_body".

We are not sure that ASIS is aware of this subtle difference and that it makes a consistent use of the term "compilation unit". It should be noticed that in ASIS "Compilation_Unit" is the name of a type. It seems also that objects of this type are always compilation_units (except if they are nonexistent, unknown, or null).

The purpose of the Ada concept of an environment is threefold. It allows Ada to define what is a legal compilation unit, what is a legal partition and what is the behaviour of a partition at run-time.

Strictly speaking, an Ada environment contains only library_items; indeed, context clauses do not make sense inside of a declarative_part, and subunits are enclosed in their parent unit in place of the corresponding stub.

6. ASIS Context and the Concept of Inconsistencies

ASIS has to deal with sets of compilation units. To avoid any confusion, it therefore introduces a new term, the concept of a context, which stands for a set of compilation_units maintained by an Ada compilation system. As we will see, a context has a state, consisting of the compilation units belonging to it, the configuration pragmas applying to them, but also depending on the history of compiler runs, or other actions taken by the compilation system. Thus, the ASIS concept of a context resembles closely an Ada environment when used as a dynamic concept like in RM95 10.1.4(3), which says: "The mechanisms for creating an environment and for adding and replacing compilation units within an environment are implementation defined."

It should be noticed that Ada does not deal with arbitrary sets of compilation units as a whole, but ASIS has to be able to do so. We will show this by an example.

Suppose we first compile a package P, then a procedure A withing P, then we modify P, recompile it, and finally compile the procedure B which too is withing P; we also suppose that all compiler runs are successful, i.e. do not detect compilation errors:

```
compile
   package P is ...

compile
   with P;
   procedure A ...

edit package P and change it to P'

compile
   package P is ...-- reference to P'

compile
   with P; -- reference to P'
   procedure B ...
```

As far as the legality of compilation units is the concern, we may distinguish the environments P alone, P together with A, P' alone, and P' together with B. It may well be that compiling A with P' would yield a compilation error. Nevertheless, all these environments contain only consistent unit sets following RM95 10.1.4 (5), which says: "When a compilation unit is compiled, all compilation units upon which it depends semantically shall already exist in the environment; the set of these compilation units shall be *consistent* in the sense that the new compilation unit shall not semantically depend (directly or indirectly) on two different versions of the same compilation unit, nor on an earlier version of itself."

Only when the results of these compilations are used to build a partition, the inconsistency of depending on two versions for P must be detected by the implementation following RM95 10.2 (27), which says: "The implementation shall ensure that all compilation units included in a partition are consistent with one another, and are legal according to the rules of the language."

As a conclusion, we can say that Ada does not deal with the above sequence of compiler runs as a whole, and especially does not bother about possible "inconsistencies".

On the contrary, it is clearly desirable for ASIS to deal with the previously described situation as a whole, so that tools can be built capable of analysing dependencies between compilation units and possible inconsistencies in the state of the "context". It is the idea that an ASIS context can refer to such a situation in some implementation-dependent way. As defined now, in no case ASIS will provide access to both versions of P, even indirectly by references from A and B. Depending on the ASIS implementation, the compilation unit A may or may not be part of a context. If it is, then ASIS must report that A is obsolete, e.g. by saying that A is inconsistent with P. If issued, queries about A may raise exceptions or produce wrong results. This does not preclude some ASIS implementation to provide correct answers to structural queries about A (which are about syntax only).

As we have just seen, ASIS introduces the idea of inconsistencies within a context. An inconsistency is always expressed in terms of pairs of compilation units, the second depending on the first: (A depends on P). An inconsistency will be reported if the latest compiler runs for A and P used different versions of P, or different versions of any other unit upon which P depends.

This is perhaps the place to speak about the concept of a version of a compilation unit. Even though Ada uses the term in RM95 10.1.4 (5), it does not define it. On the contrary, ASIS does not use it at all. ASIS provides queries for retrieving a library unit having a given name; it is stated that at most one unit can be retrieved, which means that library units, but also compilation units, are uniquely identified by their full expanded names, and that the concept of a version does not make sense in ASIS.

We hope that we have convinced the reader that the Ada environment and ASIS context concepts, though similar, are nevertheless different, and that the same holds for the concept of consistency. However, and even if technically perhaps not entirely correct, the idea of ASIS being an interface to an Ada environment reflects intuitively quite well the very nature of ASIS.

7. Configuration Pragmas

In Ada 95 the notion of a compilation is related to configuration pragmas, and as far as we know, this is its only use.

According to RM95 10.1.5 (8), the purpose of configuration pragmas is: "They are generally used to select a partition-wide or system-wide option." The same paragraph says about their placement that "[T]hey shall appear before the first compilation_unit of a compilation.", and later on about their "scope" that "The pragma applies to all compilation_units appearing in the compilation, unless there are none, in which case it applies to all future compilation_units compiled into the same environment".

ASIS does not use the notion of a compilation. The first reason for this is to keep ASIS as small as possible. The second justification is that compilers usually do not keep track of compilations, but only of their effects.

However, some ASIS-based applications may need information about configuration pragmas. ASIS therefore introduces a special kind of compilation unit, called A_Configuration_Compilation, containing only configuration pragmas. By retrieving the content of A_Configuration_Compilation, it is possible to know the configuration pragmas which are in effect for the next compiler run within this context.

It may be important to notice here that ASIS provides also a query to retrieve all the pragmas which were in effect when a given compilation unit was compiled, including both configuration pragmas and pragmas specific to the unit; clearly, some of the former may differ from those found currently in A_Configuration_Compilation.

8. Implementation of ASIS Contexts in ASIS-for-GNAT

The GNAT Ada 95 compiler is a part of the multi-lingual GCC compilation system. The main components of GNAT are: the GNAT compiler, the GNAT-specific binder and linker, and the gnatmake utility.

The GNAT compiler is known as a source-based system. Every time it is invoked, it processes not only the source file of the unit to be compiled, but also the source files of all the compilation units upon which the given unit depends, directly or indirectly. The compiler does not produce, store and reuse any "library information" for its own needs. As a result of a successful compilation, it can produce, among other results, an object file and an Ada library information (ALI) file containing the time stamps of the source files from which this object file was created.

On the contrary to the GNAT compiler, the GNAT binder is not a source-based system. The binder processes the object files created by the compiler, and uses the corresponding ALI files to check if these object files are consistent. Consistency means that all the object files needed to create the executable were generated from the same versions of the source files.

The most common use of gnatmake is to generate and execute a sequence of calls to the compiler, binder and linker needed to create an executable for a given set of source and object files, one being designated as the main subprogram. Gnatmake performs its own checks of consistency and completeness of the object files, and it also checks that all the object files correspond to the given versions of the source files. If it finds out about missing object files or inconsistencies, it calls the compiler to create or recreate the corresponding object files. If gnatmake completes successfully, the current set of object files corresponds to a complete partition and is consistent.

ASIS-for-GNAT, the ASIS implementation for GNAT, is described in [3,4]. In this paper we will only sketch how ASIS-for-GNAT is grafted onto GNAT.

During compilation, GNAT creates, in its internal data structures, a so-called Abstract Syntax Tree (AST) which contains a complete view of the Ada environment needed and used to compile the given unit. The AST also stores the pragmas which were applied to the given unit. The compiler can output the AST in a file, called a tree file. The main idea of the ASIS implementation is to rely only on these tree files. By reading them, ASIS-for-GNAT creates in its internal structures the same AST as the one handled by the compiler itself. Note that any Ada environment created by the compiler, as well as any view on such an environment stored in a tree file are always consistent, because the GNAT file name conventions and search path rules guarantee that only one version of each source file can be used by the compiler.

In ASIS-for-GNAT, the implementation of a context is simply a set of tree files, together with a special source file containing the configuration pragmas in effect. This source file is also used by GNAT and is recognized by a special naming convention.

This implementation decision for a context is not really restrictive, since it does not prevent an ASIS-based application to deal dynamically with compilation units. Indeed, after having detected that the source file of some Ada unit is accessible, but that the corresponding tree file does not exist, the application may call the GNAT compiler, which will create as a side-effect the tree file, and then the application can proceed with ASIS queries about the Ada unit.

9. Implicit Declarations

Ada distinguishes between explicit and implicit declarations (RM95 3.1 (5)): “A *declaration* is a language construct that associates a name with (a view of) an entity. A declaration may appear explicitly in the program text (an *explicit* declaration), or may be supposed to occur at a given place in the text as consequence of the semantics of another construct (an *implicit* declaration)” . In many ways, Ada deals in an uniform way with explicit and implicit declarations. We will see that this is not true for ASIS.

ASIS defines several queries returning lists of explicit declarations from different kinds of declarative regions, i.e. Body_Declarative_Items,Visisble_Part_Declarative_Items, Private_Part_Declarative_Items, and Block_Declarative_-Items.

Let’s now explain how ASIS deals with some kinds of implicit declarations.

A first case are type declarations. To retrieve the implicit declarations associated with a type, ASIS provides queries to get all the predefined and inherited subprograms associated with it, and also its implicit components, when it is a record.

A second case are statement_identifiers. Labels, block names and loop names are statement_identifiers. RM95 5.1(12) says that “For each statement_identifier, there is an implicit declaration [ ] at the end of the declarative_part of the innermost block_statement or body that encloses the statement_identifier.” ASIS does not provide a query listing the statement_identifiers implicitly declared at the end of a declarative_part. The only way to find out about statement_identifiers is to access the statements and then to retrieve the identifiers attached to them by means of the queries Label_Names and Statement_Identifier.

The two cases show that it would not be easy for an ASIS-based application to retrieve all implicit declarations of a given declarative region.

Nevertheless, the approach taken by ASIS is justifiable because compilers usually do not store the implicit statement_identifier declarations in their intermediate representations, and it would therefore be a major burden for an ASIS implementation to yield them as a result of a query.

10. Usage Names and Expressions

For usage names, Ada makes a difference between names which are and which are not expressions. ASIS considers all usage names as expressions.

For instance, the name of the procedure in a procedure call statement is considered to be an expression. Only when trying to apply to such an “expression” a query specific to an expression, yielding for example the type of the expression, ASIS will say that it is not an expression by returning a Nil_Element.

This approach allows ASIS to have a smaller element kinds classification. ASIS 87 already worked this way, and compatibility is another reason for keeping the approach.

11. Select Alternative

Let’s start by recalling the Ada syntax of a select_alternative (RM95 9.7.1 (4-7)):

```
select_alternative ::=
      accept_alternative
    | delay_alternative | terminate_alternative
accept_alternative ::=
    accept_statement [sequence_of_statements]
delay_alternative ::=
    delay_statement [sequence_of_statements]
terminate_alternative ::= terminate;
```

When analysing a selective_accept, ASIS sees the select_alternative just as a sequence_of_statements, and does not distinguish between its three possible forms. The advantage of this approach shows up when the ASIS-application programmer needs to deal with statements in a uniform way, for instance in a recursive descent. The price to pay is that she has to do her own analysis when the distinction is needed, by examining the first statement in the sequence.

12. Attribute Definition Clauses

In RM95 13.3 (2), Ada says that an attribute_definition_clause is defined as:

```
attribute_definition_clause ::=
    for local_name'attribute_designator use expression;
  | for local_name'attribute_designator use name;
```

In RM95 4.1, Ada says that a name may be an attribute_reference, which then is formally defined in RM95 4.1.4 (2) by:

```
attribute_reference ::= prefix'attribute_designator
```

ASIS uses this latter definition to simplify the attribute_definition_clause definition:

```
attribute_definition_clause ::=
    for attribute_reference use expression;
  | for attribute_reference use name;
```

From an attribute_definition_clause, ASIS therefore retrieves an attribute_reference, which can be further decomposed into the prefix and the attribute_designator parts.

Though the approach taken by ASIS is not wrong, we are not sure that it is worth to have this difference. At least in ASIS-for-GNAT, it would be much easier to implement queries yielding separately the local_name and attribute_designator parts of an attribute_definition_clause, because GNAT follows exactly the Ada 95 syntax.

13. Packaging Structure of ASIS

As already said, the set of package specifications defining ASIS was reorganized into a hierarchy by using child units. In its current state, ASIS is structured in seven root library units: ASIS, ASIS_Element_Kinds, ASIS_Vendor_Primitives, ASIS_Type_Primitives, ASIS_Unit_Kinds, ASIS_Errors and the optional ASIS_Portable_Transfer. An ASIS-based application needs access only to the members belonging to the hierarchy of which ASIS is the root library package. The purpose of the other root library packages is implementation.

ASIS is not empty, but is extensively used to make visible entities coming from "implementation" packages, by renaming and subtype declarations, e.g.:

```
package Numerics renames
   Asis_Vendor_Primitives.Numerics;
subtype Asis_Integer is
   Asis_Vendor_Primitives.Numerics.Asis_Integer;

subtype Compilation_Unit is
   Asis_Vendor_Primitives.Compilation_Unit;

subtype Asis_Time is
   Asis_Vendor_Primitives.Times.Asis_Time;
Nil_Asis_Time : Asis_Time renames
   Asis_Vendor_Primitives.Times.Nil_Asis_Time;
```

The authors of this paper think that the current packaging of ASIS is not ideal. First of all, there should be a single empty umbrella root library package named ASIS, like the Ada package in the predefined language environment. Then, since all entities of ASIS_Element_Kinds are made visible by the current ASIS package, this root library package can become a visible child of ASIS, named ASIS.Element_Kinds. If needed, an "implementation" package can be split into a public child of the ASIS package (or one of its descendants), its public part, and into a private child, its hidden part, e.g.:

```
package ASIS.Errors is ... -- the public part
private package ASIS.Implementation_Errors is ...
                                -- the private part
```

However, we think that ASIS being an interface, it should not need any "implementation" packages.

14. Conclusion

As we have seen, the Ada language definition and ASIS have different aims. When taking this into account, it can be said that ASIS follows, on the overall, as much as possible the syntax structure of Ada, that it uses Ada-defined terms consistently, and does not introduce unnecessarily new terms and concepts. Clearly, ASIS has a need for unifying concepts, e.g. for traversing heterogeneous lists, and it introduces them quite carefully and advisedly. Even though ASIS still needs some polishing, it is nearing maturity for standardization.

Considering that ASIS users and ASIS implementers are both interested in achieving stability of the ASIS 95 definition as soon as possible, and that ASIS has already proved its effectiveness for developing useful tools, the authors agree with the ASIS Working Group that the ASIS definition should be submitted as soon as possible to ISO IEC/JTC1/SC22/WG9 for standardization.

Acknowledgements

The authors would like to thank the members of the ASIS Working Group, especially Currie Colket, its chair, and Steve Blake, the main author of the current ASIS definition, for fruitful discussions and excellent remarks about a first version of this paper.

References

1. ASIS documents are available electronically on the World Wide Web:

http://www.acm.org/sigada/WG/asiswg/asiswg.html

or by anonymous ftp:

sw-eng.falls-church.va.us/public/AdaIC/work-grp/asiswg

2. International Organization for Standardization/ International Electrotechnical Committee 8652:1995 [ISO/IEC 8652:1995], Information technology - Programming language - Ada [Ada 95 Reference Manual], 15 February 1995.

3. S. Rybin, A. Strohmeier, E. Zueff: ASIS for GNAT: Goals, Problems and Implementation Strategy. In Marcel Toussaint (Ed.), Second International Eurospace - Ada-Europe Symposium Proceedings, LNCS no 1031, Springer, pp. 139-151, 1995.

4. S. Rybin, A. Strohmeier, A. Kuchumov, V. Fofanov: ASIS for GNAT: From the Prototype to the Full Implementation. In Alfred Strohmeier (Ed.), 1996 Ada-Europe International Conference on Reliable Software Technologies Proceedings, LNCS no 1088, Springer, pp. 298-311, 1996.

5. American National Standards Institute/Department of Defense Military Standard 1815A-1983 [ANSI/MIL-STD 1815A- 1983], Reference Manual for the Ada Programming Language, 17 February 1983 [Ada 83 Reference Manual]; identical to International Organization for Standardization 8652:1987 [ISO 8652:1987], Information technology - Programming language - Ada [Ada 87 Reference Manual], March 1987.