ASIS for GNAT:
Goals, Problems and Implementation Strategy

Sergey Rybin[1], Alfred Strohmeier[2], Eugene Zueff[1]

[1] *Scientific Research Computer Center*
Moscow State University
Vorob'evi Gori
Moscow 119899, Russia

[2] *Swiss Fed Inst of Technology in Lausanne*
Software Engineering Lab
CH-1015 Lausanne, Switzerland

e-mail: rybin@alex.srcc.msu.su,
zueff@such.srcc.msu.su

e-mail: alfred.strohmeier@di.epfl.ch

Abstract. This article describes the approach taken to implement the Ada Semantic Interface Specification (ASIS) for the GNAT Ada compiler. The paper discusses the main implementation problems and and their solution. It also describes the current state of the implementation.

Keywords. Ada, ASIS, Compilation, Software Engineering Environment.

1 Introduction

1.1 ASIS

The Ada Semantic Interface Specification (ASIS) [1] is an interface between an Ada environment, as defined by the Ada language definition [4], and any tool or application requiring information from this environment. ASIS is organized as a set of self-documented Ada package specifications providing abstractions and related queries which allow to obtain all statically-determinable information from any compilable unit contained in the Ada environment. Examples of tools which may benefit from ASIS include pretty-printers, cross-references, browsers, static code analyzers and derived measurement tools, etc.

ASIS is based on four main abstractions, represented by private types having the same names:

- Library - retrieval of state information about the Ada program library (for Ada 83) or the Ada compilation environment (for Ada 95);
- Compilation_Unit - retrieval of information related to the external view of a compilation unit such as its kind (subprogram, package, generic unit, specification or body, etc.) and of semantic dependencies between compilation units, including those defined by with clauses, and navigating between compilation units;
- Element - retrieval of information about syntactical components of a compilation unit, provided by its structural decomposition and the semantic links between related elements; as an example, the list of parameters can be retrieved for a sub-

program declaration, and then for each parameter specification, the name of the parameter, its type, and its default value, if any, can be obtained;

- Id - provides external images for items of type Element for storing to and retrieving from disk; can be used to implement persistent ASIS-based applications.

1.2 The ongoing revision of ASIS

The final ASIS specification defined for Ada 83 [5] is version 1.1.1 [1]; we will call it ASIS 83. The revision of ASIS for the new Ada language standard [4], sometimes called Ada 95 to avoid confusion, is in progress now; we will call it ASIS 95.There are plans to make ASIS an international standard. But already now, the ASIS 95 draft series [1] contains enough information to implement a working prototype for Ada 95.

The main changes in the new ASIS definition are changes in functionality and a new library/environment model.

1.2.1 Changes in functionality

Proposed changes in the functionality of ASIS can be subdivided in two groups.

The first group contains additions for new Ada 95 entities, such as child units and protected types.

With the aim of reducing the number of queries, many of the ASIS 83 queries have been aggregated in ASIS 95. Theses queries form the second group of changes.

For about half of the ASIS 83 queries, changes are proposed; therefore implementing ASIS 83 for an Ada 95 compiler would be a waste of resources.

1.2.2 The new library/environment model

The notion of the ASIS library, one of the fundamental ASIS abstractions, was revised to take into account the changes in the library model and compilation process definitions of Ada 95. Indeed, the new standard gives much more freedom to a compiler implementer than the old standard did. This issue is of special importance for the GNAT compiler which provides only a basic library facility.

The latest ASIS 95 draft (ASIS 2.0.C) contains the first version of the revised ASIS library model, which would be better called environment model to conform to Ada 95 terminology. The main points of this revision are:

- The notions of library, compilation/recompilation order and obsolete unit of Ada 83 are replaced by the notions of environment, semantic dependencies and consistency of a compilation unit of Ada 95; e.g., the consistency of a compilation unit as defined by ASIS 2.0.C is the consistency, in the sense of Ada 95, of the set of compilation units including the given unit and all other units on which it depends;
- The queries defined for the private type Compilation_Unit is revised in accordance with these changes. All queries reflecting the history of library creation and modifi-

cations (such as the test for a compilation unit to be obsolete or the query asking for the order of compilation/recompilation for a given set of units) are either removed or reformulated in terms of semantic dependencies and library units consistency;

- In the latest ASIS 95 documents, ASIS is defined as an interface to the Ada environment (not the Ada library), and the term ASIS library is for ASIS specific uses only; it really means an Ada environment for an ASIS application;
- ASIS 83 was really defined as being an interface to an Ada library linked to and managed by some Ada compilation system. But now for Ada 95, a stand-alone ASIS implementation, i.e. having no special connection to any Ada compiler and being able to process any consistent set of Ada sources, is considered to be very useful.

1.3 Implementing ASIS for GNAT

GNAT [6] is an Ada 95 compiler publicly available for many platforms under the copyright rules of the Free Software Foundation. Our project is aimed at the development of an ASIS 95 implementation for GNAT, called ASIS-for-GNAT. It is performed by the Scientific Research Computer Center of Moscow State University in cooperation with the Swiss Federal Institute of Technology in Lausanne. The project is supported by the Swiss EST funding programme.

The initial analysis and design efforts started with ASIS 83 in September 1994, but from the very beginning of the project (January 1995) we have been using the latest available ASIS 95 drafts as the basis for ASIS-for-GNAT. Ada 95 is used as an implementation language and GNAT itself is used as a compiler.

In a first stage we are building a partial prototype; the goal is to validate design and implementation decisions that can be used for further developments.

This prototype will be available at the end of 1995. Its main limitation will be that it can open only one ASIS library, that it processes only one compilation unit at a time, and that it provides only so-called structural queries.

2 Implementation strategy

2.1 The GNAT compiler

The GNAT compiler [2] is part of the multi-language GCC compilation system which includes front-end processors for several languages and code generators (back-ends) for various hardware platforms and software environments. GNAT is a Ada 95 compiler, consisting of a specific Ada 95 front-end, written mostly in Ada 95, combined with any standard GCC back-end.

GNAT uses the traditional scheme of the compilation process. It contains phases of syntax analysis, semantic analysis and code generation, which communicate through a common intermediate internal representation of Ada program units, the Abstract Syntax Tree (AST) [2]. The AST reflects the syntax structure of an Ada program unit (including symbol information) and also contains semantic attributes for all entities contained in the unit. GNAT has internal high-level procedure interfaces to this tree.

GNAT uses a so-called “source-based” compilation model requiring the source text of a program unit in the environment to be processed every time GNAT needs information about it [3]. For its operation, GNAT does not need to produce, store, or retrieve any centralized or distributed “library information”.

However, the contents of the so-called .ali files could be used to determine dependencies between compilation units. An .ali file is created together with the object file when a unit is successfully compiled, and must be considered as being conceptually part of the object file. Such an .ali file contains the list of names and time stamps of the source files of all the units upon which the given unit depends.

On the other side, an AST contains information not only about the represented program unit, but also about all its supporters, i.e. the specifications of the units on which it depends, directly or indirectly.

Thus, the AST decorated by various semantic annotations is the main and only data structure built and handled by the GNAT front-end. Conceptually, the AST contains all information needed to answer almost all ASIS queries. 2.2 The fundamental implementation decision

The general background for the ASIS implementation for the GNAT compiler is:

- there is no specific library information;
- the AST is the high-level data structure containing all statically determinable information about the unit having been compiled by GNAT;
- there is a high-level procedural interface to the AST;
- the AST is the only data structure produced and maintained by GNAT;
- the AST is only internal to GNAT.

The main implementation decision is to use the AST as the basis for the internal representations of ASIS abstractions: the ASIS Compilation_Unit type is basically represented by the top node of the AST, which is constructed by the front-end for the corresponding Ada unit, and the ASIS Element type is basically represented by the tree node which is created for the corresponding (explicit or implicit) construct. The direct consequence of this decision is to implement ASIS-for-GNAT on top of the GNAT AST interfaces.

2.2 Some implementation problems

The GNAT front-end is a procedure which builds and handles only one AST during its invocation. The AST is kept in hidden data structures and the functional interface to the AST is rigidly bound to these structures. On the contrary, the ASIS interface is provided as a set of packages, and its implementation must be able to provide information about multiple compilation units, and, consequently, must be able to handle multiple ASTs.

So the main implementation problems are:

- how to access an AST from within ASIS queries; this problem is of primary importance for the prototype;
- how to access several ASTs at the same time when running an ASIS-based application; the solution of this problem has been postponed, and the prototype will process only one compilation unit.

3 Architecturing the interaction with GNAT

There are two possible ways to solve the problem of accessing the AST from inside the ASIS implementation - embedding the prototype just in the context of the GNAT compiler or adding special ASIS-related capabilities to GNAT.

3.1 First approach: building a modified GNAT

The first approach is to completely embed the ASIS implementation into the GNAT compiler. We modified the main GNAT driver Gnat1drv by replacing the call to the Back_End procedure by a call to an ASIS-based application. This application can make calls to our prototype, which in turn is built on top of the GNAT AST interfaces. Thus, the ASIS-based application starts in the dynamic context created by invoking the GNAT Front_End procedure. Launching the modified GNAT compiler with some compilation unit as a parameter corresponds to launching the ASIS-based application for this compilation unit.

In this approach all the compiler's internal data structures can be naturally accessed by the AST interfaces, and therefore by ASIS queries, without any additional transformations.

3.2 Second approach: adding special capabilities to GNAT

The second approach uses the extra functionalities in GNAT for storing to and retrieving from files its internal data structures making up the AST. This feature has been implemented as a special compiler option which provides dumping of the AST into a disk file, and a special interface for reading such files. This interface allows to reuse the trees already created during previous invocations of GNAT. As a result, the external images of the AST-related data structures become independent objects and can be processed by the prototype. Therefore, as long as the external images exist, GNAT has not to be called again.

We performed some preliminary work on this approach, and plan to adopt it in the future.

This second approach could also provide a natural way to access several ASTs at a time by means of re-retrieving them from disk, but the performance penalty may be too high.

3.3 Comparing the two approaches

The advantage of both of these approaches is to conform with the current GNAT philosophy: a single set of sources is used to make both GNAT and additional tools (binder, cross-reference utility, etc.), and the same source modules are included into various programs.

It is impossible to adapt the first approach for dealing with several compilation units at a time. As already stated, the second approach would probably show poor performance when an ASIS-based application needs to process multiple compilation units. The real cause of this disadvantage is the nature of the GNAT AST interfaces - they are not parametrized by the tree to which they provide access. But this problem can hardly be solved without serious modification of the compiler itself.

As stated above, the GNAT source-based compilation model requires to compile with a unit all its direct and indirect supporters. The AST of a unit contains hence full information not only about the unit itself, but also about all its supporters. ASIS queries about these supporters can therefore be satisfied without building their own AST and without recalling GNAT.

4 Architecture and capabilities of the prototype

4.1 Packaging the prototype and accessing the internal ASIS data structures

ASIS 95, in its draft ASIS 2.0.C, defines the following Sample Implementation Structure:

The main ASIS private types (Library, Compilation_Unit, Element, Id) are declared in subpackages of the package Asis_Vendor_Primitives, which should not be "withed" by ASIS-based applications:

```
package Asis_Vendor_Primitives is
   package Elements is
      type Element is private;
      ...
   private
      type Element is ...; -- implementation-defined
      ...
   end Elements;
   subtype Expression is Asis.Element;
      subtype Statement is Asis.Element;
...
end Asis_Vendor_Primitives;
```

Then operations for these types are defined in various packages making up the portable ASIS interface, for example:

```
with Asis_Vendor_Primitives;
package Asis_Statements is
   package Local_Renames is
      package Asis renames Asis_Vendor_Primitives;
   end Local_Renames;
   use Local_Renames;
   function Assignment_Expression
      (Statement : in Asis.Statement)
         return Asis.Expression;
   ...
end Asis_Statements;
```

And finally, the umbrella package ASIS provides uniform renaming for all ASIS resources which are available, by definition of the ASIS specification, to portable ASIS-based applications:

```
with Asis_Statements;
with Asis_Vendor_Primitives;
   ...
   package Asis is
      package Statements renames Asis_Statements;
      subtype Element is Asis_Vendor_Primitives.Element;
      ...
end ASIS;
```

This structure cannot be implemented in a clean way (e.g. without using unchecked conversions), because the package bodies implementing operations for ASIS types cannot access the details of their representations, and the package Asis_Vendor_Primitives does not provide any "interface" to those types.

In ASIS-for-GNAT, the packaging structure was enhanced by using child units together with library level renaming, as shown below.

ASIS types are defined immediately within the Asis_Vendor_Primitives package and enclosed subpackages are removed:

```
package Asis_Vendor_Primitives is -- ASIS-for-GNAT
   type Element is private;
   subtype Expression is Asis.Element;
   subtype Statement is Asis.Element;
   ...
private
   type Element is record
      ...
   end record;
   ...
end Asis_Vendor_Primitives;
```

For each of the ASIS packages mentioned as making up the portable ASIS interface and defining operations for ASIS types a corresponding counterpart package is implemented as a child of Asis_Vendor_Primitives. All these child packages contain the same specifications as the corresponding ASIS packages (with only a minor technical change which has no impact on a portable ASIS-based application: they do not any longer contain the Local_Renames packages):

```
package Asis_Vendor_Primitives.Asis_Statements is
   function Assignment_Expression
      (Statement : in Asis.Statement)
         return Asis.Expression;
   ...
end Asis_Vendor_Primitives.Asis_Statements;
```

This new structure provides direct access to all implementation details of the ASIS private types in the bodies of the child units.

And finally, the child packages are renamed at the library level as the corresponding ASIS packages to form the required interface and to provide the top-level Asis package, yielding as a result the semantics of the original ASIS specification:

```
with Asis_Vendor_Primitives.Asis_Statements;
package Asis_Statements
   renames Asis_Vendor_Primitives.Asis_Statements;
```

The proposed packaging structure solves in a natural way the problem of accessing the implementations of the ASIS types in package bodies. It can be considered as an alternative to the Sample Implementation Structure.

4.2 Selecting a subset of ASIS to be implemented in the prototype

Besides the main limitation of the prototype, which is only able to deal with one compilation unit at a time, the following principles were used for subsetting ASIS:

- It was decided to implement the operations for initializing a single library and a single compilation unit for an ASIS-based application; these operations are defined in the packages:
- Asis.Libraries Asis.Compilation_Units Asis.Environment The full implementation of these package was postponed.
- The implementation of the Asis.Text and Asis.Ids packages was postponed as a relatively independent part;
- The implementation of the structural ASIS queries, i.e. queries providing step-by-step top-down decomposition of a compilation unit was selected as being of first priority.

5 Implementation of the structural queries

The ASIS structural queries provide the top-down decomposition of an ASIS Compilation Unit according to its syntax structure. As a rule, a structural query yields a child element (or a list of child elements) for its argument element.

Implementing the ASIS structural queries on the base of the GNAT AST thus requires two main problems to be solved: defining and implementing the mapping between AST nodes and ASIS elements of various kinds, and traversing the AST, with operations provided by GNAT, with the purpose of yielding the decomposition as defined by ASIS.

5.1 The representation of the ASIS Element type

The ASIS Element type is implemented by an undiscriminated record type. This record contains the reference to the AST node corresponding to the given Element in its Node field, and the reference to the position of this Element in the ASIS Element classification hierarchy in its Internal_Kind field (i.e. we use a "flat" image of the hierarchy, where each kind having the subordinate classification is replaced by its subordinate kinds).

5.2 Mapping between tree node kinds and Element kinds

To implement an ASIS structural query, we can use the Node field of its argument Element to jump into the AST context exactly to the place of the corresponding node. The AST operations provided by GNAT are used for traversing the tree in order to find the node corresponding to the child element to be returned by the query (or to find several nodes if the query returns list of Elements). As a rule, this tree traversal is rather straightforward, but to build the result, we have to set its Internal_Kind field. In the remaining part of this subsection we will describe what is the problem.

The kind of an element returned by an ASIS query can vary. Moreover, when a query returns a list of elements (e.g. a list of statements, a list of declarations), each element of such a list can belong to another kind. So, we have to map the kinds of nodes in the AST onto Element kinds. This mapping problem is of special interest among the technical implementation problems; it arises from differences between the two approaches taken by GNAT and ASIS to represent an Ada program unit.

For two thirds of the more than 200 Node_Kinds values, the mapping is a rather trivial one-to-one mapping between a Node_Kinds value and an Internal_Element_Kinds value, for example:

```
N_Defining_Character_Literal <=> A_Defining_Character_Literal
N_Case_Statement             <=> A_Case_Statement,
```

but for other Node_Kinds values, the mapping is not one-to-one, as shown below.

Sometimes, a Node_Kinds value maps onto several Internal_Element_Kinds depending on the context (one-to-many mapping):

```
N_Subprogram_Declaration => A_Procedure_Declaration
N_Subprogram_Declaration => A_Function_Declaration
```

Sometimes, several Node_Kinds values merge into one Internal_Element_Kinds value (many-to-one mapping). For example, A_Procedure_Declaration Element can be associated with a tree node either of kind N_Abstract_Subprogram_Declaration or of kind N_Subprogram_Declaration:

```
N_Subprogram_Declaration          => A_Procedure_Declaration
N_Abstract_Subprogram_Declaration => A_Procedure_Declaration
```

The last two examples also show that the same Node_Kind value, i.e. N_Subprogram_Declaration, may participate in a one-to-many and in a many-to-one mapping.

There are even holes in the mapping.

Firstly, there are some nodes in the AST having no Element counterpart. Sometimes they are for internal use for the compiler only and must be ignored, e.g. the node N_Freeze_Entity, but sometimes they are the hook of a subtree which must be traversed. For example, there is an AST N_Mod_Clause node for record representation clauses, whereas ASIS simply yields the expression part of this clause.

Secondly, some Element values do not have a node counterpart in the AST. For example, the tree contains no single node corresponding to An_Else_Path Element, representing the last structural part of a composite control statement. Instead, the tree stores in the node of the enclosing control statement (i.e. N_If_Statement, N_Selective_Accept or N_Conditional_Entry_Call) a pointer to the list of nodes corresponding to the statement sequence.

5.3 Reconstructing the original unit structure from the tree

GNAT makes various tree transformations during semantic analysis, when performing compile-time optimization (for example, to replace a static expression by its value) and when preparing the tree for the code generation phase.

During these transformations, some nodes are replaced by newly generated subtrees.

Code generation related transformations can be suppressed by invoking GNAT with the compile-only flag: -gnatc.

For detecting the other transformations, we have to examine the Rewrite_Sub and Rewrite_Ins node flags to recognize the nodes which have been replaced by a subtree and to detect the nodes inserted during semantic analysis.

Fortunately, GNAT keeps the “replaced” nodes of the original tree, and they can be retrieved by the function Original_Node. Unfortunately, the “original” node does no longer contain a reference to the parent node in the tree, and this reference must therefore be retrieved from the "replacing" node, i.e. the root of the replacing subtree.

Inserted nodes can be skipped for structural queries, but they may be of use for some semantic queries, especially for element representing implicit constructs.

5.4 Too many case statements?

The code of our ASIS implementation contains a lot of case statements, sometimes replaced by look-up tables. It's well known, that this style of "variant" programming is best replaced by "incremental" programming, i.e. an object-oriented approach. Unfortunately, neither GNAT, e.g. its tree retrieval operations, nor ASIS make use of such an approach.

5.5 Is there enough information in the AST?

From the very beginning of our project the question whether there was enough information stored in the AST that could be retrieved by GNAT-provided operations was of prime importance. Now we can say that the features provided by GNAT are sufficient for implementing all structural queries about ASIS Elements corresponding to explicit constructs in the program unit. There is just one exception: GNAT does not care during tree creation whether or not a unit name is repeated after the last "end" of the unit specification or body; it simply assumes that the name is always repeated.

We have to say, however, that we have not yet thoroughly tested the use of the Original_Node function for recreating the original tree, and further investigations must be performed as GNAT relies heavily on rewriting for generic units.

As a last escape, when some information is definitely missing in the AST, it would be possible to retrieve it from the source buffer, which is part of the GNAT data structures. Indeed, every node in the AST contains a reference to the position in the source buffer where it originates from, and this reference could be used to jump from the node to right place in the buffer.

6 Designing the ASIS library model for the GNAT environment

Structural queries cannot cross compilation unit boundaries, and therefore don't really need the ASIS library model to be implemented. On the contrary, semantic queries usually express the properties of an element in terms of other elements: for example, the defining occurrence can be retrieved for a direct name. As shown by this example, that often means we have to jump to another compilation unit. Implementation of the ASIS library model is hence needed for semantic queries.

GNAT provides a simple "lightweight" Ada compilation model [3]. It sometimes looks very strange to Ada 83 programmers, but it is completely in conformance with Ada 95 rules: it contains no compilation/recompilation order requirement and no explicit notion of a library.

Our design for the ASIS library model is based on the following points:

- The starting point for an ASIS library is a set of directories of the file system.
- An object of type ASIS.Library is initialized by calling the procedure Libraries.Associate with all the directories belonging to the library as an actual.
- Searching in these directories will be performed in the order of their appearance in the actual parameter.
- Only compilable .ads and .adb files, i.e. files containing sources of the specifications and bodies of Ada compilation units, in these directories are considered as belonging to the ASIS library. ASIS-for-GNAT will not require or assume anything about the presence or absence of any other files in the set of directories making up the ASIS library.
- The contents of all the directories making up the ASIS library must remain frozen between the initialization and finalization of the ASIS environment (otherwise the corresponding ASIS application could have erroneous behavior). “Frozen” means that .ads and .adb files cannot be changed, cannot be removed from the library, cannot be moved to another directory, and it’s forbidden to create new ones.

7 Current state of the prototype

In mid-September 1995, the prototype supported most of the ASIS 2.0.C structural queries. The most difficult queries that remain to be implemented are the Enclosing_Element query and the generic procedure Traverse_Element.

The Enclosing_Element query returns the parent Element for its argument. It can be implemented by using the reference to the parent node kept in tree nodes, but a lot of technical problems related to differences in the ways of traversing Ada code in ASIS and in the GNAT AST operations must be solved.

The Traverse_Element generic procedure provides the recursive top-down and left-to-right traversal of the element passed as an argument while performing some pre-operation and post-operation (provided by the corresponding generic parameters) for each subcomponent. It can be implemented on the base of all the other ASIS structural queries.

Let’s recall that the prototype can process only one library and only one compilation unit during its invocation.

When writing this article, the prototype was composed of nearly 1.7 MBytes self-documented Ada source code, including the 575 kBytes of the initial self-documented ASIS 2.0.C specification. It depends on most of the GNAT front-end components.

Systematic testing have not been performed so far. In order to avoid time-consuming rebuilding of the ASIS test applications when debugging, we implemented a debugging monitor allowing to call ASIS queries interactively and to view the result through debugging variables.

This monitor itself has become an additional branch of the project. Although it was originally seen as a mere secondary debugging and testing tool, it was found out that it opens several new avenues. First of all, being a conventional ASIS application, it appears to be portable between various ASIS implementations.

8 Conclusion

The project goal is to implement ASIS for the GNAT Ada 95 compiler; a prototype will be available at the end of 1995, with features as described in section 7. For fall 1996, we hope to implement the full ASIS library model and all semantic queries. The Ids and Text packages would then be the only missing parts for completing ASIS-for-GNAT.

9 References

1. ASIS 1.1.1, ASIS 2.0.A, etc. - ASIS documents are available electronically: on 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. E. Schonberg, B. Banner: The GNAT Project: A GNU-Ada 9X Compiler. Ada Europe News, No 20, pp. 10-19 (March 1995)
3. R. Dewar: The GNAT Compilation Model. Ada Europe News, No 20, pp. 20-23 (March 1995)
4. —: Ada 95 Reference Manual. Intermetrics, Inc., Feb. 15, 1995 (ISO/IEC 8652:1995)
5. —: Reference Manual of the Ada Programming Language. Washington, D.C.: Department of Defense, Ada Joint Program Office, Feb. 1983 (ISO/IEC 8652:1987)
6. GNAT Compiler: can be get by anonymous ftp: cs.nyu.edu/pub/gnat