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

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

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

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 their solutions. It also describes the current state of the implementation. This is a slightly revised version of the paper published in the proceedings of the Ada-Europe'95 conference.*

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 users 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-referencers, browsers, static code analyzers, 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 (e.g., subprogram, package, generic unit, specification, body), semantic dependencies between compilation units (e.g., those caused by with clauses), and navigation 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 subprogram declaration; 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 external storage (e.g., disk). Id can be used to implement persistent ASIS applications.

We use the terms "Library," "Compilation Unit," "Element, " and "Id" with the first letter capitalized to emphasize that we refer to the ASIS-defined notions.

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 Ada 95, the new Ada language standard [4], is in progress now; we will call it ASIS 95. There are plans to make ASIS an international standard. But already, the ASIS 95 draft [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. These queries form the second group of changes.

More than half of the ASIS 83 queries have been changed during the ASIS revision process; 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 83 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 implementor 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.E) contains the full definition of the revised ASIS Library model, which is better called an environment model to conform with 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 and semantic dependencies in the environment;

- The queries defined for the private type Compilation_Unit are revised in accordance with these changes. All queries reflecting the history of library creation and modifications (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;

- 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. 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 while GNAT is used as its compiler.

On the current stage of the project, we are building a partial prototype ASIS 95 implementation for GNAT; 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 limitations are:

1. it can open only one ASIS Library;
2. it can process only one Compilation Unit at a time; and
3. it provides only so-called structural queries (not semantic 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 a number of back-ends (code generators) for a variety of hardware platforms/software environments. GNAT is an 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 compilation process. It contains phases of syntax analysis, semantic analysis and code generation. These phases communicate through a common intermediate internal representation of Ada program units called 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 own purposes, GNAT does not need to produce, store, or retrieve any centralized or distributed "library information."

When a unit is successfully compiled, an Ada library information file is created together with the object file for the unit. Ada library information files are denoted with the .ali file-name extension. These files contain the list of names and time stamps of the source files of all the units upon which the given unit depends. As such, they can be used to determine dependencies between compilation units. Information contained in an .ali file can be retrieved from the corresponding AST. A compilation unit's Ada library information file must be conceptually considered as the part of the corresponding object file; the aim of its physical separation is to facilitate the binding of an Ada program.

It should be mentioned that 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 exists only as the internal structure of the GNAT front-end;

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.

2.3 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, it 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 application; the solution to 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 front-end 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 front-end. We modified the main GNAT driver (Gnat1drv) by replacing the call to the Back_End procedure with a call to an ASIS application. This application can make calls to our prototype, which in turn is built on top of the GNAT AST interfaces. Thus, the ASIS 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 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 functionality 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 us to reuse the trees already created during previous invocations of GNAT. When reading the AST from a disk file, all the GNAT internal structures related to the AST representation are initialized in the same way as when processing the corresponding sources by the GNAT front-end; therefore, all the GNAT AST interfaces can be used. 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 does not have 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 retrieving them from disk as needed, but the performance penalty may be too high.

3.3 Comparing the Two Approaches

The advantage of both of these approaches is that they 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 in 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. The GNAT procedure interfaces to this tree do not provide parameters to identify which tree to access. But this problem can hardly be solved without serious modification of the compiler itself.

As stated above, the GNAT source-based compilation model compiles a unit with all its direct and indirect supporters. Hence, the AST of a unit contains 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 calling GNAT again.

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.E, 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 applications:

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

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

For each of the ASIS packages making up the portable ASIS interface (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:

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

There is a minor technical change - these packages do not contain the Local_Renames packages. This has no impact on a portable ASIS-based application.

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

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 able to deal with only 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 application; these operations are defined in the packages Asis.Libraries, Asis.Compilation_Units, and Asis.Environment. The full implementation of these packages was postponed.

The implementation of Asis.Text and Asis.Ids packages was postponed as these are relatively independent parts.

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 ASIS structural queries based on the GNAT AST requires two main problems to be solved:

1. defining/implementing the mapping between AST nodes and ASIS Elements of various kinds, and
2. traversing the AST, with the purpose of yielding the decomposition as defined by ASIS using the operations provided by GNAT.

5.1 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 with its 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 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 a list of Elements). As a rule, this tree traversal is rather straightforward. However, to determine the result to be returned, we have to set its Internal_Kind field. In the remaining part of this subsection we will describe 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 different subordinate kinds (that is, statement kinds, declaration kinds). 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 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. Sometimes, a Node_Kinds value maps onto several Internal_Element_Kinds (one-to-many mapping):

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

Further, several Node_Kinds values can 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_Subprogram_Declaration or of kind N_Abstract_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.

First, 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, the AST N_Mod_Clause node must be traversed for record representation clauses; ASIS simply yields the expression part of this clause.

Second, 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, during compile-time optimization (e.g., to replace a static expression by its value), and during preparation 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 node flags (i.e., Rewrite_Sub and Rewrite_Ins) to identify the nodes which have been replaced by a subtree and detect the nodes inserted during semantic analysis.

Fortunately, GNAT keeps the "replaced" nodes of the original tree, and these can be retrieved by the function Original_Node. Unfortunately, the "original" node no longer contains a reference to the parent node in the tree. 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 Elements 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 of prime importance was: "Is enough information stored in the AST, retrievable by GNAT operations, to support a useful ASIS implementation?" 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 Compilation Unit. There is just one exception: GNAT does not care during tree creation whether or not a unit name, a statement name or an entry name is repeated after the last "end" (e.g., unit specification, unit body, control statement, accept statement); 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 resort, when 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. This reference could be used to jump from the node to the 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 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 (e.g., the defining occurrence can be retrieved for a direct name). This often means that we have to jump to another Compilation Unit. Implementation of the ASIS library model to support ASIS for GNAT is hence needed for semantic queries.

GNAT provides a simple "lightweight" Ada compilation model [3]. It sometimes looks very strange to Ada 83 programmers. It is completely in conformance with Ada 95 rules, but contains no compilation/recompilation order requirements 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 parameter.
- Searching 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 deleted from the Library, cannot be removed from the Library, and cannot be moved to another directory even within the ASIS Library. In addition, it's forbidden to create new .ads and .adb files in the ASIS Library.

7 Current State of the Prototype

In mid-November 1995, the prototype supported all of the ASIS 2.0.C structural queries except the tests for name repetition after the end of a declaration or a statement, the generic Traverse_Element procedure and the Enclosing_Element query.

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 an 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.

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 1.8 MBytes of self-documented Ada source code, including nearly 600 kBytes of the initial self-documented ASIS 2.0.E specification. It depends on most of the GNAT front-end components.

Systematic testing has not been performed so far. In order to avoid time-consuming rebuilding of the ASIS test applications when debugging, we implemented a debugging monitor which invokes ASIS queries interactively providing results 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, we found out that it opens several new avenues. In effect, the debugging monitor is an ASIS application accessing ASIS interfaces just as any software engineering tool might. Being a conventional ASIS application, it should be portable between various ASIS implementations. Consequently it can be a valuable teaching tool to those planning to use ASIS regardless of the ASIS implementation environment.

8 Conclusion

The project goal is to implement ASIS 95 for the GNAT Ada 95 compiler.

A partial prototype will be available at the end of 1995, with features as described in section 7.

The next step will be to extend the prototype, so it deals not only with the one compilation unit, but also with all of its supporters. Then we will implement semantic queries for explicit Elements. We hope to also have a first implementation of the Asis_Text package.

For Fall 1996, we hope to implement the full ASIS library model and all semantic queries, including those for implicit Elements. Then the Asis_Ids package will be the only missing part for completing ASIS-for-GNAT.

9 References

1. ASIS 1.1.1, ASIS 2.0.A - ASIS 2.0.E - ASIS documents are available electronically on the World Wide Web, URL: [http://www.acm.org/sigada/WG/asiswg/asiswg.html]. They are also available by anonymous ftp to the sw-eng.falls-church.va.us host in the /public/AdaIC/work-grp/asiswg directory.

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. 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.

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.

6. GNAT Compiler: can be obtained by anonymous ftp to the cs.nyu.edu host in the /pub/gnat directory.