ASIS-for-GNAT: A Report of Practical Experiences

Sergey Rybin[1], Alfred Strohmeier[2], Vasiliy Fofanov[1], Alexei Kuchumov[1]

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

[2]*Swiss Federal Institute of Technology in Lausanne*
Department of Computer Science
1015 Lausanne EPFL, Switzerland
email: alfred.strohmeier@epfl.ch

Abstract: What are the main difficulties met when implementing ASIS, what are the problems when using ASIS, are there any missing features in ASIS, e.g. when it comes to object-oriented programming, these are some of the topics dealt with in this experience report.

Keywords: ASIS, Ada, Programming Language, Programming Tool, Programming Environment.

1 Introduction

The Ada Semantic Interface Specification (ASIS) [1] [9] is an interface between an Ada environment [7] [20] and any tool or application requiring statically-determinable information from this environment. ASIS defines types: Context represents an Ada environment [17], Compilation_Unit represents an Ada compilation unit, and Element models a syntactic construct, e.g. a declaration, a statement, an expression, etc. Operations on these types and the results of their calls are called "queries" in ASIS terminology.

The authors of the paper have been involved in ASIS activities for more than five years. Most of the time was spent in developing and maintaining the ASIS implementation for the GNAT Ada 95 compilation system, called ASIS-for-GNAT [2], but we also took part in developing the ASIS ISO standard, teaching ASIS, providing technical support to ASIS-for-GNAT users and developing our own ASIS-based tools. We feel that this is a good moment for reporting our ASIS experience, hoping it will be useful to other people working with ASIS and will help further developments of the ASIS technology.

2 History

2.1. ASIS

The first version of ASIS was developed in the mid-eighties by Rational Software Corporation in response to the need for developing tools for supporting the various life cycle phases of an Ada program. Soon after, ASIS was implemented for other Ada 83 compilers, and the ACM SIGAda ASIS Working Group [3] was formed. ASIS version 1.1.1 is the de facto standard for Ada 83, and usually called ASIS 83.

Paper accepted for publication in the proceedings of the International Conference on Reliable Software Technologies - Ada-Europe'2000, Potsdam, Germany, June 26-30, 2000, Erhard Plödereder, Hubert B. Keller (Eds.), LNCS (Lecture Notes in Computer Science), No ??, Springer-Verlag, 2000, pp. ??.

The ASIS WG started adapting ASIS 83 to the revised definition of Ada some time before it became the new ISO Ada standard in 1995. The first version of the ASIS definition for Ada 95, called now ASIS 95, was presented in 1994, and several ASIS implementation projects, including ASIS-for-GNAT, were started. The ISO standard for ASIS 95 was adopted at the end of 1998 and officially published in the beginning of 1999.

At the time of writing, there are three ASIS 95 implementations respectively by Ada Core Technologies, Aonix and Rational Software Corporation, and a number of ASIS-based tools have been announced on the net and presented at Ada conferences [10] [11] [14]. In other words, ASIS is now a reality and an important part of the Ada 95 technology.

2.2. ASIS-for-GNAT

ASIS-for-GNAT was started at the end of 1994 as a joint research project of the Swiss Federal Institute of Technology in Lausanne and the Computer Center of the Moscow State University. In the middle of 1996, the development of ASIS-for-GNAT was handed over to Ada Core Technologies, the company providing support for the GNAT Ada 95 technology [4]. Soon after, ASIS-for-GNAT was announced as a fully supported product. Currently ASIS-for-GNAT is developed and maintained by ACT-Europe [5], the European GNAT company.

The general design decisions of ASIS-for-GNAT were described in [15] [16].

By now, a number of ASIS-based tools are available for ASIS-for-GNAT. Some of them are included in the GNAT tool kit.

3 Inside an ASIS implementation

3.1. Main implementation techniques and common implementation problems

ASIS is able to retrieve the complete statically-determinable syntax and semantics from an Ada environment for a tool requiring such information. The only way to achieve this without implementing an almost complete Ada syntax and semantics analyzer is to reuse the front-end of some Ada compiler. Therefore all the existing ASIS implementations are implemented as interfaces to existing Ada compilers, and not as stand-alone products.

The straightforward approach to implement ASIS is to use some intermediate representation, usually an abstract syntax tree, of an Ada program generated by the compiler. Such an intermediate representation contains the results of syntax and semantic analysis performed by the compiler. Conceptually, the implementation of objects of the ASIS types Compilation_Unit and Element are then just references to the component of the compiler's data structure representing the corresponding Ada compilation unit or syntax construct. The fact that the ASIS implementation actually extracts information from the compiler's internal data structure is at the origin of most implementation problems, which, we believe, are common to all ASIS implementations.

The main goals of the compiler are to check the legality of the Ada units being compiled and to generate object code, or to create some data structure from which the object code can be easily generated. Therefore, the compiler may (and most probably every compiler does) transform the original program source in order to simplify syntax and semantic checks, and make code generation easier. Some obvious examples are:

- computing static expressions;
- creating internal implicit types to represent anonymous subtypes with constraints, used e.g. as record or array component definitions;
- replacing some language constructs, such as select statements, with calls to run-time library routines.

As a consequence, the main job of an ASIS implementation is to convert the model used by the compiler to represent the compiled code into what is required by the ASIS definition. How difficult or easy this task is depends on how close the two models are.

The main GNAT internal data structure, the Abstract Syntax Tree (AST) [19], was designed with ASIS in mind. But dozens of small details and many more difficult issues had to be dealt with when the AST is used by the ASIS implementation. Most of the implementation problems arose in the following areas:

- implicit Elements;
- the predefined package Standard;
- accessing the contents of generic instantiations, called expanded generic code in ASIS;
- the Enclosing_Element query.

The difficulty of implementing Enclosing_Element results from the fact that when the compiler rewrites a node in a subtree and redirects the references in its successor nodes to this new node, the references to the original ancestor node are lost.

3.2. ASIS and the source-based compilation model

GNAT is an example of a source-based compiler. This means that GNAT does not create (and then reuse) any centralized library information, and when compiling a unit, it does not use the results of previous compilations of other units upon which the given unit depends. Instead, GNAT processes directly the source code of the given unit and all its supporters. As a result, in a source-based compilation system, we do not have any centralized data structure representing the whole Ada environment (or the whole partition), but we have independent data structures, each of which representing the semantics of some compilation unit, including all its supporting units. The task of the ASIS implementation is therefore to assemble these independent pieces to represent the whole Ada environment.

As already said, GNAT uses an Abstract Syntax Tree (AST) for intermediate code representation. When called with a special option, GNAT can output the AST into a file, called a tree file. ASIS-for-GNAT uses these tree files, and therefore gets access to the very same AST as used by GNAT for every successful compilation. However, whereas the compiler deals only with a single compilation unit, i.e. the root unit of the compilation, ASIS-for-GNAT deals with an ASIS Context, i.e. a set of compilation

units. As a consequence, ASIS-for-GNAT must access information in several tree files. A problem arises if the same compilation unit is present in several trees.

An example might help the reader understand the problem (Fig. 1).

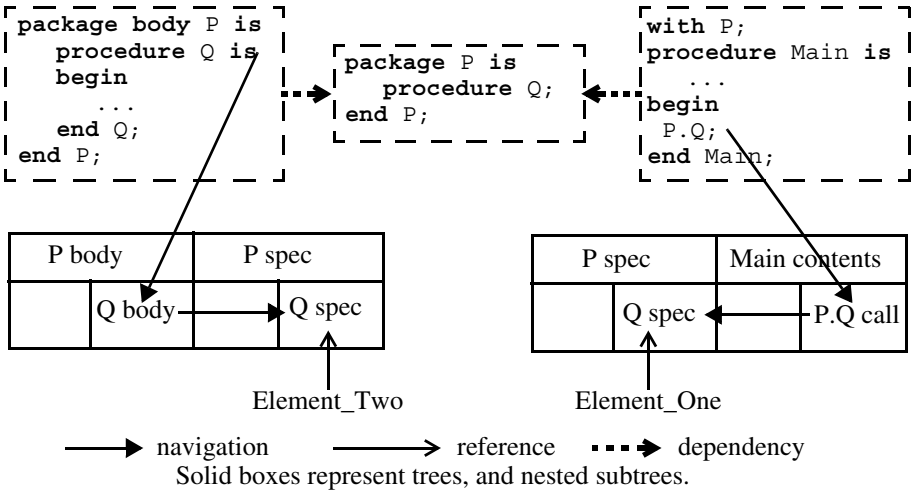

Fig. 1 Example ASTs containing a duplicated subtree

Let's suppose the ASIS Context contains the following compilation units: the procedure Main, and the specification and body of package P. ASIS-for-GNAT will use two trees, one for Main, and another for the body of P. Both these trees contain the specification of the package P. As a consequence, all information related to the package specification P is duplicated. However, ASIS defines features for deciding if two elements, i.e. Ada syntax constructs, are the same or not, and the ASIS implementation must therefore be able to make this decision. On the example, an ASIS-based tool might analyze the source of Main, find the call to P.Q, and from there navigate to the specification of the called unit, i.e. the declaration of Q in P, and name the result Element_One. The same tool might access the body of P, traverse all enclosed declarations, and navigate from the body of the procedure Q to its specification, naming the latter one Element_Two. Even though Element_One and Element_Two are not physically the same, they represent logically the same element, and the result of the call Is_Equal (Element_One, Element_Two) must therefore yield True. Otherwise stated, the implementation of ASIS has to deal with this identification problem, e.g. by using properties of the elements (syntax constructs) not varying with the tree.

An other problem arises if from Element_One, one of the two representations of the specification of procedure Q, we want to find the corresponding body. Indeed, the body is not part of the enclosing tree, and we have to switch in some way to the other tree. Note that getting the body related to Element_Two is straightforward.

The whole issue of dealing with elements (syntactic constructs) having multiple representations in multiple trees becomes even more complicated when it comes to library-level instantiations.

4 Testing an ASIS implementation

Testing is never an easy task, and testing an ASIS implementation is no exception. Moreover, there are several specific problems in the case of ASIS:

- ASIS is not a program but a library containing about 350 queries. To test an ASIS query, the corresponding test driver (or test ASIS application) needs to be written, and the input for an ASIS application is itself an Ada program. This requires additional efforts when preparing and running ASIS tests. But first of all, one has to write a test driver and test it, which by the very nature of ASIS is not a trivial task. Later on, when analyzing the test results, one has to be aware of the fact that a test may fail, either because there is a bug in the ASIS implementation, or in the test driver;
- The main ASIS abstractions (Context, Compilation_Unit, Element) are private types. This makes it difficult and very often impossible to create directly test data and to directly observe and analyze results for the query under test, and the only way to do it is to use other ASIS queries;
- The overall complexity of testing ASIS is of the same magnitude as testing an Ada compiler. Recall that ASIS deals with all the syntax and static semantics of the language, and a coverage criteria is therefore directly related to covering all these aspects of the Ada programming language.

Our approach for testing ASIS implementations, called Quality-for-ASIS [18], is based on two main components:

- The first component is a set of universal ASIS test drivers, corresponding oracles (analyzers of test results), and test set adequacy analyzers. The drivers perform extensive ASIS-based processing of Ada code, and the oracles use different approaches for indirect, and in some situations partial observation of the test results. For example, one of the drivers recreates the source of the Ada unit processed by ASIS by using only ASIS structural queries, and the corresponding oracle tests that the original and recreated sources are semantically the same. Even though such tests are "indirect" and "partial", their use has proved to be effective in practice. Altogether, these drivers provide a completely automated, portable test suite for about 2/3 of the ASIS queries, representing the most important part of ASIS, including structural and semantic queries, source text representation queries, traversal of source code and some other features.
- The second component is Asistant, an interactive interpreter of ASIS queries with log and script capabilities. Asistant itself is an ASIS application. It creates its own environment for evaluating sequences of ASIS queries, e.g. when developing an ASIS application, without having to write and compile the corresponding ASIS application as an Ada program. An early version of Asistant was described in [12]. Asistant is also useful for learning ASIS, e.g. when you are not quite sure about the effect of a sequence of queries, or for debugging "small" pieces of ASIS-based Ada code - you know that there is an error, but you are unable to locate it.

As an input test set for Quality-for-ASIS, we have chosen the ACVC test suite (without the illegal programs called B-tests, which clearly cannot be processed by ASIS). This is quite a natural choice, because ACVC provides a good coverage of the Ada syntax and semantics. We also use the ACVC test set on a daily basis to detect possible regressions in ASIS-for-GNAT.

5 The ASIS ISO Standard - a great success with some small flaws

Authors of this paper participated in the development of the ASIS definition for Ada 95 and in preparing the ISO ASIS standard. Now, that we have some real experience in providing technical support for ASIS users and in developing our own ASIS tools, it is possible to give an overview of what we consider to be shortcomings in the current definition of ASIS.

5.1. Element classification hierarchy versus flat Element classification

ASIS classifies Elements by repeating practically one-to-one the Ada syntax as defined in the RM 95. ASIS defines this classification as a hierarchy: at the top level we have basic notions such as A_Declaration, A_Statement, An_Expression, then we have more detailed classifications for these basic constructs, e.g. An_Ordinary_Type_Declaration, An_Assignment_Statement, and An_Attribute_-Reference, and so on, some positions in this classification requiring still another subordinate classification. Each level in this hierarchy is defined by an enumeration type, and ASIS has a classification query corresponding to each of these levels, e.g. Element_Kind at the very top, then Statement_Kind, Expression_Kind, etc. As a result, in order to know in the application code the exact kind of the element - which is a very common need, because most of the ASIS queries are element kind specific - several classification queries and nested if or case statements are needed.

We have found this classification scheme to be really cumbersome, including when implementing ASIS, because almost all ASIS queries have to detect the kind of the Element which passed as an actual. As a consequence, already at the very beginning of the ASIS-for-GNAT project, we introduced a flat ASIS Element classification scheme; roughly speaking, this classification is implemented by a single enumeration type which combines all the literals from all the classification types from the original ASIS classification hierarchy. Although this idea was thought to be an ad hoc implementation solution, very quickly we found it to be convenient when writing any ASIS-based code, because only one classification query was needed to detect the exact nature of an element, and a single case statement was enough to process an Element of an "unknown" kind. As a result, we provide now this flat classification scheme as an ASIS extension to ASIS-for-GNAT users.

5.2. Name collisions

The better is often the enemy of the good. The ASIS name space is heavily overloaded by synonyms for the Element type, like "subtype Declaration is Element". There are 58 such synonyms. They were thought to provide good mnemonics for the parameter profiles of ASIS queries. But the real consequence is, that if an ASIS application includes a use clause for the top-level ASIS package named Asis (where the main ASIS types are defined), it is impossible to declare names such as "Clause", "Declaration", "Definition", "Name", "Statement", etc.

While this is but a nuisance, the situation with the ASIS-defined name Subtype_Mark is bluntly dangerous. Subtype_Mark is defined as a subtype of Element in the package Asis, and there is a query Subtype_Mark declared in Asis.Definitions, with a single argument of the type Element, returning a result of the type Element.

To show the problem, let's consider the following code:

```
E1, E2: Asis.Element;

...

E1 := Subtype_Mark (E2);
```

In this piece of code, Subtype_Mark (E2) is either a type conversion (from the type Element to the subtype Subtype_Mark) or a call to the Asis.Definitions.-Subtype_Mark query, depending on the use clauses in effect! One of the authors had a very hard time finding an error due to this problem.

There are some other problems with naming.

The package Asis.Compilation_Units declares a query named Compilation_Units. It is therefore impossible to call this query by its simple name.

The set of subtypes of the type Element defined in the package Asis contains a subtype named Record_Component. At the same time, Record_Component is the name of the type implementing one of the main abstractions of the Asis.Data_Decomposition package. In the same program, it is therefore impossible to have use clauses for both Asis and Asis.Data_Decomposition (because type declarations are not overloadable), and the use of simple names is therefore quite restricted.

5.3. Duplicated and missing functionality

One of the primary goals when revising ASIS 83 for Ada 95 was to get rid of redundant queries, i.e. of queries which can be implemented as trivial combinations of other queries.

Many redundant queries were removed from ASIS, but at least one still exists: Asis.Expressions.Corresponding_Name_Declaration is a trivial combination of Elements.Enclosing_Element and Expressions.Corresponding_Name_Definition.

With regard to missing functionality, we would like to point out two things.

In Ada, there can be several occurrences of the same defining name corresponding to the same entity, e.g. the name of a subprogram in its specification and body, the name of formal parameters appearing both in the specification and body of the subprogram, etc. The ASIS definition allows the Corresponding_Name_Definition query to return a reference to any of the defining occurrences, and does not provide any selection mechanism. It is therefore difficult to implement an ASIS-based application maintaining name tables.

Another problem is the impossibility to go from a protected entry declaration to the corresponding entry body declaration, and vice versa; the ASIS standard states that the ASIS queries Corresponding_Declaration and Corresponding_Body are not supposed to work for protected entries. The underlying difficulty is that an entry declaration might be completed by a entry body when part of a protected type or might correspond to one or several accept statements when belonging to a task.

5.4. Do we have upward compatibility of ASIS 83 tools with ASIS 95?

Upward compatibility of ASIS 83 tools with ASIS 95 was one of the primary goals of the ASIS revision. In our opinion, this goal was not reached, and it cannot be. First,

there are too many differences between Ada 83 and Ada 95, and as a result too many new positions in the Element and Compilation_Unit classifications, which actually affect most of the ASIS queries. Second, this goal is in direct contradiction with two other important stated goals: removing redundant queries and reducing the number of queries by aggregating related queries into one single query, whenever possible.

[21] gives a detailed report of porting a set of ASIS 83 applications to ASIS-for-GNAT, and shows that porting ASIS 83 code to ASIS 95 is a major undertaking.

5.5. ASIS and object-oriented features of Ada

ASIS' view of a program, or a set of program units, is essentially based on its textual or syntactical structure (Fig. 2). Properties orthogonal to this structure are almost completely neglected. Even in Ada 83, it would have been interesting to get the primitive operations (as they are called now) of a type, or the list of all subprograms called in a sequence of statements.

When it comes to object-oriented programming, the ability to adopt another viewpoint than the textual one becomes essential. In an object-oriented mindset, the world and the program are viewed as a set of interacting objects and a hierarchy of classes. To the contrary of other object-oriented programming languages, like C++ or Java, where the class is the basic syntax and module construct, this view is not "wired" in the syntax of Ada. Tools able to provide such a view, e.g. a class browser, are therefore even more important.

```
package Pack1 is
   type T is tagged private;
   procedure P1 (Arg: in T);
   ...
end Pack1;
```

```
package Pack1.Pack2 is
   type T1 is new T with private;
   procedure P2 (Arg: in T1);
   type T2 is new T with private;
   ...
end Pack1.Pack2;
```

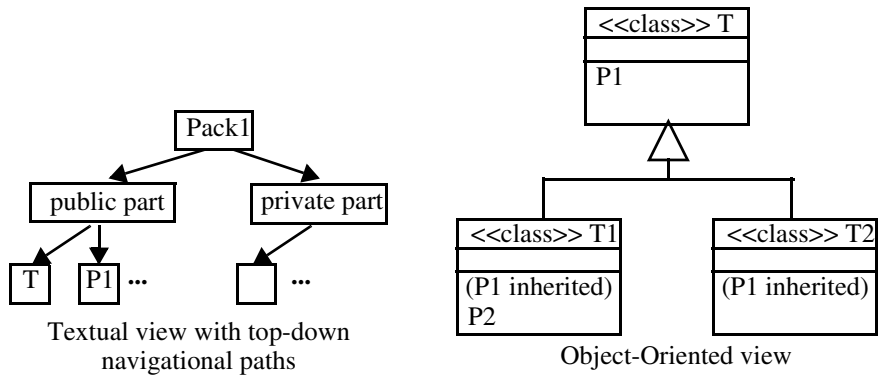

Fig. 2 Textual and object-oriented views of a program

Possible extensions to ASIS providing an object-oriented view are: find a derivation class, find the primitive operations of a tagged type, decide if a primitive operation is overridden, and for a dispatching call, find the list of possible implementations, etc. In section 8.2., we will present OASIS, a research prototype implementing such queries.

To conclude this section, and even though there might be some omissions in ASIS, we want to stress that the most important is that ASIS exists here and now, as a standard and implemented as additions to several compilers, and that it provides an excellent foundation for developing portable tools. A useful standard on time is better than a perfect standard too late, and problems described in this section could hardly be found before ASIS was intensively used.

6 Assisting ASIS users

In our ASIS tutorials one of the main statements is "ASIS is easy-to-start-with and easy-to-use". This is true and not true at the same time. We do believe that a half-day tutorial is enough to get an overview of ASIS and to learn how to write simple ASIS-based programs.

But using ASIS in a real project, i.e. developing a real tool which itself is supposed to process real Ada code, is another story. The main reason making the ASIS learning curve longer than one would expect is the large size of both Ada and ASIS.

ASIS contains 349 different queries, and 157 of them are structural and semantic queries working on more than 370 different kinds of ASIS Elements. Really, ASIS cannot be small and simple, because Ada itself is not a small and simple language. Therefore, typical questions of ASIS newcomers are: "What query should I use to go from an element of kind X to its component Y?"; "What are the kinds of the components of an element having the kind Z?"; "Can I get this semantic property of that element directly, or do I have to compute it myself? And if I have to, how to do it?". Most of these questions are very easy to answer for anybody who has several months of ASIS experience; even a newcomer can find the answers by browsing the ASIS standard. But such an approach takes time, and when there are too many of such questions, and when wrong guesses lead to compilation or run-time errors, then developing an ASIS-based application becomes tedious.

According to our experience in providing technical support for ASIS-for-GNAT during the last few years, quick answers for queries like the previous ones are really helpful for users during their first steps with ASIS.

The Asistant tool mentioned in section 4 can also be used when learning ASIS and as a quick interactive ASIS reference when working on an ASIS-based tool. In Asistant, you can "browse" the structure of Ada code without thinking about which query should be used to go from some element to its components, and while "browsing", you can ask for the kind of each visited element and get a list of applicable queries. In many situations, Asistant gives quick answers to questions like: "How to use ASIS in this situation for that element?".

Another starting help is to provide beginners with a set of easily extendable templates for ASIS-based tools. One example is the style checker program described in [13], which implements part of the Ada style guide [6].

7 Development of ASIS-based tools

We gained our experience as ASIS users by developing several ASIS-based tools: some are tools for testing ASIS implementations, or debugging them, i.e. Asistant, then there is a tool kit for pretty-printing Ada source code, an Ada-to-HTML converter, but also two tools included in the GNAT tool kit, i.e gnatstub which creates compilable body "stubs" for specifications of library packages, and gnatelim which detects unused subprograms.

Even though our experience is limited to about a dozen ASIS-based tools, we think that we can already summarize the first lessons learned from using ASIS.

The first observation is that many ASIS-based programs have a very similar general structure which actually maps quite well to the required sequence of calls to ASIS queries described in the ASIS standard:

(1) Defining the ASIS Context to be processed;

(2) Opening the Context and retrieving the compilation units to be processed;

(3) Iterating through this set of compilation units, and selecting some of them for further processing;

(4) Going into the internal structure of a compilation unit;

(5) Traversing the code of the compilation unit, usually performed by an appropriate instantiation of the generic procedure Asis.Iterator.Traverse_Element.

During this approach, the following points should be kept in mind:

- **Portability of ASIS tools**: One of the main ideas of ASIS is to provide portability of tools. At the same time, ASIS tools can not be fully portable, because the way an ASIS implementation interacts with the underlying Ada environment is implementation-specific, and as a result, defining and processing an ASIS Context is also implementation-specific. The best way to improve portability of ASIS tools is to separate the implementation-specific actions needed for defining and opening a Context from the detailed analysis of Ada code, which is implementation-independent. Coming back to the general structure of an ASIS tool given above, we may say that the result of step (1) is never portable, that there might be some implementation dependencies for the steps (2) and (3), and that the last two steps lead very often to completely portable code.
- **Code templates for ASIS tools**: The code implementing the steps (1) to (4) is very similar in many ASIS-based tools, and what is really different is the code of actual parameters used to instantiate Traverse_Element. The obvious idea is therefore to provide a set of reusable code templates. To build a new tool, a programmer then has only to write the code which is really specific to the tool. The latest version of ASIS-for-GNAT contains a set of such templates.

Another important observation is that in a very early stage of an ASIS-based tool project, a programmer is induced to design a set of so-called secondary ASIS queries. A secondary query is by definition implemented by combining basic ASIS queries and satisfies specific needs of a tool. A tool is indeed often built on top of ASIS and some secondary ASIS libraries. Of course, such a secondary library is mostly tool-specific, but some of the secondary queries may be of general interest. This leads to the idea of developing and standardizing general-purpose ASIS secondary libraries.

8 Extending the functionality of ASIS

8.1. Extending ASIS-for-GNAT

Currently, ASIS-for-GNAT provides about 30 queries extending the basic ASIS functionality. Some of them came from implementation needs, some are generalizations of secondary queries developed as parts of our ASIS tools, some resulted from customer requests.

There are different kinds of such queries, with regard to both their functionality and their implementation.

Some queries supplement the basic ASIS functionality, like the Is_Primitive_Operation and Is_Static boolean functions. It is impossible to implement these queries by combining primary queries, and they are therefore extensions, according to ASIS terminology. Other queries could be considered as secondary queries, but their implementation by combining primary queries is either difficult, or leads to an inefficient implementation, or both of them. Examples of this kind are the boolean queries Acts_As_Spec, Is_Renaming_As_Body, and Is_Completed, but also the query Components which returns the complete list of all the components of an Element, whatever its kind. Finally, some additional queries are GNAT-specific, like Is_Obsolete and Compilation_Dependencies, which work on compilation units and return GNAT-specific information about the validity of a compilation unit, respectively return dependencies between compilation units and source files.

We expect that the set of queries extending standard ASIS will grow. We also believe that every ASIS implementation will have its own set of such queries. It would be important to put this process under control, otherwise the portability of ASIS tools can no longer be guaranteed.

8.2. OASIS

OASIS, an acronym of Object-Oriented ASIS, is a research project aimed at developing an ASIS secondary library. The project is based on the public version of the ASIS-for-GNAT technology. The primary idea of OASIS is to provide a set of abstractions closely corresponding to the object-oriented model. The latest version of OASIS defines the following abstractions, partly illustrated in Fig. 2:

- A Derivation Class corresponds to the hierarchy of the types derived from some root tagged type;
- A Derivation Item corresponds to a node in the Derivation Class.
- A Component corresponds to a component of a tagged type or a type extension, either part of its declaration, or inherited from an ancestor;
- A Primitive corresponds to the notion of a primitive operation in Ada, be it inherited or not;
- A Class-Wide Operation reflects the Ada notion of a class-wide operation;
- A Dispatching Call reflects the Ada notion of a dispatching call.

A first set of queries in OASIS provides an object-oriented view of an ASIS context: it is for instance possible to retrieve all the roots of all the derivation classes in a given context. Then there is a set of queries for navigating in a derivation class: e.g find the

ancestor class, find the derived classes, and traverse a whole derivation class. Finally, there are queries to find the components, inherited or not, of a Derivation Item, its inherited and genuine primitive operations, and to decide if an operation is overridden.

8.3. Towards a unified set of ASIS-based libraries

Some time ago, the ACM SIGAda ASIS WG started discussions aimed at unifying the ASIS libraries extending the basic ASIS functionality. The main result of these discussions are the guidelines for developing such libraries provided by Steve Blake [8]. The main points of these guidelines are:

- ASIS secondary queries should be separated from ASIS extensions. ASIS secondary queries are implemented entirely on top of ASIS. Therefore, an ASIS secondary library can be ported from one ASIS implementation to another one. On the contrary, the implementation of an ASIS extension is based on internal compiler mechanisms, and therefore cannot be ported from one ASIS implementation to another without extending the ASIS implementation.
- A simple naming convention should be used to avoid name collisions between ASIS extensions and secondary queries provided by different implementations.

It seems ASIS will evolve like the Ada language itself: the ASIS Standard will be structured in a "core" part, and a set of secondary libraries and extension libraries defined in Special Needs Annexes.

9 Conclusions

At the time of writing, it is already evident that ASIS has become an important part of today's Ada technology. We expect that its use will grow, and that we will see more and more ASIS tools in a very near future. We hope that reporting our experience in working with ASIS might be useful to future users of ASIS, and become a valuable input to the next ASIS revision.

References

[1] Ada Semantic Interface Specification (ASIS); International Standard ISO/IEC 15291 1999 (E).

[2] ASIS-for-GNAT is available electronically as a part of the public version of the GNAT technology from http://www.gnat.com

[3] ASIS home page of SIGAda, the Special Interest Group on Ada of the Association of Computing Machinery (ACM): http://www.acm.org/sigada/WG/asiswg/

[4] Home page of ACT, Ada Core Technologies, http://www.gnat.com.

[5] Home page of ACT Europe, the European branch of Ada Core Technologies, http://www.act-europe.fr.

[6] Christine Ausnit-Hood, Kent A. Johnson, Robert G. Pettit, IV, Steven B. Opdahl (Eds.); Ada 95 Quality and Style: Guidelines for Professional Programmers; Lecture Notes in Computer Science, vol 1344; Springer-Verlag, 1995; ISBN 3-540-63823-7.

[7] John Barnes (Ed.); Ada 95 Rationale: The Language, The Standard Libraries; Lecture Notes in Computer Science, vol 1247; Springer-Verlag, 1997; ISBN 3-540-63143-7.

[8] Steve Blake; Message sent to the ASIS WG technical forum.

[9] Currie Colket et alii; Architecture of ASIS: A Tool to Support Code Analysis of Complex Systems; ACM Ada Letters, January 1997, vol. XVII, no. 1, 1997, pp. 35-40.

[10] Currie Colket; Code Analysis of Safety-Critical and Real-Time Software Using ASIS; Proceedings of the ACM SIGAda Annual International Conference (SIGAda'99), 17-21 October 1999, Redondo Beach, CA, USA, pp. 67-76.

[11] C. Daniel Cooper; ASIS-Based Code Analysis Automation; ACM Ada Letters, November/December 1997, Volume XVII, No.6, pp. 65-69.

[12] Vasiliy Fofanov, Sergey Rybin, Alfred Strohmeier; ASIStint: An Interactive ASIS Interpreter; Proceedings of TRI-Ada'97, St. Louis, MO, USA, November 11-13 1997, Susan Carlson (Ed.), ACM Press, 1997, pp. 205-209.

[13] Vitali Kaufman; The ASIS rule checking program of Vitali Kaufman is available from http://www.kolumbus.fi/vitali.kaufman/gch and as a link from http://www.acm.org/sigada/wg/asiswg.

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

[15] Sergey Rybin, Alfred Strohmeier, Eugene Zueff; ASIS for GNAT: Goals, Problems and Implementation Strategy; Proceedings of Ada-Europe'95, Toussaint (Ed.), LNCS (Lecture Notes in Computer Science) 1031, Springer, Frankfurt, Germany, October 2-6 1995, pp. 139-151.

[16] Sergey Rybin, Alfred Strohmeier, Alexei Kuchumov, Vasiliy Fofanov; ASIS for GNAT: From the Prototype to the Full Implementation; Reliable Software Technologies - Ada-Europe'96: Proceedings, Alfred Strohmeier (Ed.), LNCS (Lecture Notes in Computer Science), vol. 1088, Springer, Ada-Europe International Conference on Reliable Software Technologies Montreux, Switzerland, June 10-14, 1996, pp. 298-311.

[17] Sergey Rybin, Alfred Strohmeier; Ada and ASIS: Justification of Differences in Terminology and Mechanisms; Proceedings of TRI-Ada'96, Philadelphia, USA, December 3 - 7, 1996, pp. 249-254.

[18] Alfred Strohmeier, Vasiliy Fofanov, Sergey Rybin, Stéphane Barbey; Quality-for-ASIS: A Portable Testing Facility for ASIS; International Conference on Reliable Software Technologies - Ada-Europe'98, Uppsala, Sweden, June 2-8 1998, Lars Asplund (Ed.), LNCS (Lecture Notes in Computer Science), Springer-Verlag, 1998, pp. 163-175.

[19] E. Schonberg, B. Banner; The GNAT Project: A GNU-Ada 9X Compiler; TRI-Ada'94 Proceedings, ACM Press, 1994; pp. 48-57.

[20] S. Tucker Taft, Robert A. Duff (Eds.); Ada 95 Reference Manual: Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E); Lecture Notes in Computer Science, vol 1246; Springer-Verlag, 1997; ISBN 3-540-63144-5.

[21] Joseph Wisniewski; Transitioning an ASIS Application; Proceedings of the ACM SIGAda Annual International Conference (SIGAda'99), 17-21 October 1999, Redondo Beach, CA, USA, pp. 53-65.