ASIStint: An Interactive ASIS Interpreter

Vasiliy Fofanov(1), Sergey Rybin(1), Alfred Strohmeier(2)

(1) Scientific Research Computer Center
Moscow State University
Vorob'evi Gori
Moscow 119899, Russia
e-mail: fofanov@such.srcc.msu.su, rybin@alex.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
http://lglwww.epfl.ch/

Abstract

ASIStint is an interactive ASIS interpreter with scripting facilities. It may be used for learning ASIS, i.e. the user may interactively try out the effects of the various ASIS queries. It might also be used as an assistant (sic!) when experimenting with ASIS queries, e.g. in order to discover a way of implementing a part of an ASIS application, or to correct it. Yet another use is debugging and testing an ASIS implementation. Input-output of a session may be recorded, and then be replayed.

Keywords: *Ada, ASIS, Interpreter.*

1 Introduction

The Ada Semantic Interface Specification (ASIS) [1] is an interface between an Ada environment, as defined by the Ada language reference manual [2], and any tool or application, called an ASIS application, requiring statically-determinable information from this environment. ASIS itself is defined as a set of self-documented Ada package specifications providing the types and operations, called ASIS queries, used to retrieve and to deal with this information.

ASIStint, a contraction of ASIS t[est] int[erpreter], pronounced like assistant, is an interactive ASIS interpreter with scripting facilities. ASIStint allows the user to call ASIS queries interactively. It may be used for learning ASIS, i.e. the user may interactively try out the effects of the various ASIS queries. It might also be used as an assistant (sic!) when experimenting with ASIS queries, e.g. in order to discover a way of implementing a part of an ASIS application, or to correct such an application. Yet another use is debugging and testing an ASIS implementation.

ASIStint commands may also be gathered together in a script. A script can be called both interactively and in batch mode. Both the input and the output of a session are recorded in files. The input commands can then be replayed in another session.

Although ASIStint was developed jointly with ASIS-for-GNAT, the ASIS implementation for the GNAT Ada 95 compilation system [3][4][5], it is just a regular ASIS application, and it can therefore be used with any ASIS implementation.

This article shows how to use ASIStint, describes its language, and gives some hints about its implementation. A full description of ASIStint is provided in its User Guide [5]. Its source code is part of the ASIS-for-GNAT distribution [5].

We will first show an example. Then we will briefly describe the ASIStint language, its execution environment, and how the interpreter is implemented. Finally, we will discuss possible uses and benefits of ASIStint.

Acknowledgment

We thank Clyde Roby for carefully reviewing this paper and for his many useful comments. We are also grateful to the anonymous reviewers for their helpful suggestions.

This work was supported by a grant from the Swiss National Science Foundation, no 7SUPJ048247, funding a project entitled "Development of ASIS for GNAT with industry quality".

2 Example of an ASIStint script

We will first give an example of a small Ada program, then show how to obtain information about it by an ASIS application, and finally use ASIStint for retrieving the same information.

Consider the following small Ada program:

```
with Text_IO;
procedure Demo is
   A : Integer := 1;
   B : Integer := 2;
   C : Integer;
begin
   C := A + B;
   Text_IO.Put_Line (Integer'Image (C));
end Demo;
```

Suppose we want to output the following elements:

- the name of the compilation unit, i.e. "Demo",

- the first name declared immediately within this unit, i.e. “A”,
- the call to the procedure `Put_Line`,
- the prefix of the actual parameter of the previous call, i.e. “Integer’Image”.

The expected output consists therefore of four lines of text:

```
Demo
A
   Text_IO.Put_Line (Integer'Image(C));
                     Integer'Image
```

The example will show some typical steps in an ASIS application:

- define an Ada context,
- process a compilation unit as a black-box,
- use structural queries to get syntactic information.

Let’s first show how an ASIS application can achieve the task:

```
with Text_IO;
with Asis;
with Asis.Implementation;
use Asis.Implementation;
with Asis.Ada_Environments;
use Asis.Ada_Environments;
with Asis.Compilation_Units;
use Asis.Compilation_Units;
with Asis.Declarations;
use Asis.Declarations;
with Asis.Elements;
use Asis.Elements;
with Asis.Expressions;
use Asis.Expressions;
with Asis.Statements;
use Asis.Statements;
with Asis.Text;
use Asis.Text;

procedure Demo_ASIS is

(1)  Ctx    : Asis.Context;
     Unit   : Asis.Compilation_Unit;
     Decl   : Asis.Element;

begin -- Demo_ASIS

(2)  Initialize (String(""));
(3)  Associate (Ctx, String(""), "");
(4)  Open (Ctx);
     -- Initialize the ASIS implementation (2),
     -- define the context Ctx to work with (3),
     -- and open it (4).

(5)  Unit := Compilation_Unit_Body("Demo", Ctx);
     -- Unit designates the compilation unit
     -- body "Demo" of the context Ctx.

(6)  Text_IO.Put_Line (Asis.Compilation_Units.
                       Unit_Full_Name (Unit));
     -- Print out the name of the compilation
     -- unit.

(7)  Decl := Unit_Declaration(Unit);
     -- Now we enter into Unit, in order to
     -- process its source code, called
     -- "white-box processing" in ASIS.
     -- Decl designates the library item part of
     -- Unit, which is in our case a procedure
     -- body declaration.

   declare -- block

(8)  Body_Decl : Asis.Element_List :=
       Body_Declarative_Items (Decl, True);
   -- Retrieve the elements of Decl, i.e. the
   -- declarations within the body. We
   -- need a block statement here, since
   -- Asis.Element_List is an unconstrained array
   -- type, and an initial value must therefore
   -- be provided when declaring the variable.

(9)  Decl_Names : Asis.Element_List :=
       Names (Body_Decl(1));
   -- Get the list of names declared in the first
   -- declaration. This is again an array
   -- variable!

(10) Body_Stmts : Asis.Element_List :=
       Body_Statements (Decl, True);
     -- Get the list of statements of the body.

(11) Call_Stmt : Asis.Element := Body_Stmts (2);
     -- Select the second statement to work
     -- with, which we know  is a call statement.

(12) Param_List : Asis.Element_List :=
       Call_Statement_Parameters
          (Call_Stmt, False);
     -- Get the list of the actual parameters of
     -- the call statement.

(13) First_Param : Asis.Element :=
       Actual_Parameter(Param_List(1));
     -- Select the first parameter.

(14) Pref : Asis.Element := Prefix(First_Param);
     -- Take the prefix of the first
     -- parameter.

   begin -- block

     -- All the decomposition has already been
     -- done, so we output the images we would
     -- like to see:

(15) Text_IO.Put_Line
       (Defining_Name_Image (Decl_Names(1)));
     -- Print out the first name in the list
     -- Decl_Names.
(16)  Text_IO.Put_Line
       (Element_Image (Call_Stmt));
     -- Print the call statement.
(17)  Text_IO.Put_Line (Element_Image (Pref));
     -- Print the prefix of the first actual
     -- parameter.

   end; -- block

end Demo_ASIS;
```

As all Ada programs, this so-called ASIS application `Demo_ASIS` must first be compiled. When run, it will produce the previously described output.

The solution in ASIStint (see below) parallels almost entirely the ASIS application. It is enough to issue commands corresponding to the statements and declarations bearing numbers in parentheses above. Since variables are declared dynamically and results are available immediately, they can be displayed immediately too. This is especially convenient when working with ASIS lists which, as we have seen, are unconstrained array types. Therefore, the calls to `Put_Line`, all gathered at the end of the `Demo_ASIS` program (15, 16, 17), can therefore be spread throughout the ASIStint program, e.g. (15) comes after (9), where they are replaced by the ASIStint `Print` command. The advantage to the user is evident: he can check immediately if what he gets is what he wants.

Instead of issuing the commands interactively, they can be stored in a script file.

```
(1)  set (Ctx)
     -- Create the variable Ctx, which is a
     -- context.

(2)  Initialize ("")
(3)  Associate (Ctx, "", "")
(4)  Open(Ctx)
     -- Initialize the ASIS implementation (2),
     -- define the context Ctx to work with (3),
     -- and open it (4).

(5)  set (Unit, Compilation_Unit_Body
          ("Demo", Ctx))
     -- Create the variable Unit, and make it
     -- designate the compilation unit body
     -- "Demo" of the context Ctx.

(6)  print (Unit_Full_Name(Unit))
     -- Print out the name of the compilation
     -- unit.

(7)  set(Decl, Unit_Declaration(Unit))
     -- Now we enter into Unit, in order to
     -- process its source code, called
     -- "white-box processing" in ASIS.
     -- Decl designates the library item part of
     -- Unit, which is in our case a procedure
     -- body declaration.

(8)  set (Body_Decl,
        Body_Declarative_Items (Decl, True))
     -- Retrieve the elements of Decl, i.e. the
     -- declarations within the body. Notice the
     -- difference with the Ada code: we don't
     -- need a block statement!

(9)  set (Decl_Names, Names(Body_Decl(1)))
     -- Get the list of names declared in the
     -- first declaration.

(15) print (Defining_Name_Image
            (Decl_Names(1)))
     -- Print out the first name in the list.

(10) set (Body_Stmts,
       Body_Statements (Decl, True))
     -- Get the list of statements of the
     -- body.

(11) set (Call_Stmt, Body_Stmts(2))
     -- Select the second statement to work
     -- with, which we know is a call statement.

(16) print (Element_Image(Call_Stmt))
     -- Print the call statement.

(12) set (Param_List, Call_Statement_Parameters
          (Call_Stmt, False))
     -- Get the list of the actual
     -- parameters of the call statement.

(13) set(First_Param, Actual_Parameter
          (Param_List(1)))
     -- Select the first parameter.

(14) set (Pref, Prefix (First_Param))
     -- Take the prefix of the first parameter.

(17) print (Element_Image (Pref))
     -- Print the prefix of the first actual
     -- parameter.
```

Although identifiers are not case-sensitive, in scripts and examples, we will use lowercase letters for identifiers of ASIStint-specific operations, i.e. `log`, `set`, and initial capital letters for ASIS queries, i.e. `Initialize`, `Associate`, `Open`, `Compilation_Unit_Body`, and for variable names, i.e. `Ctx`, `Call_Stmt`. In the text itself, we write all names with initial capital letters, e.g. `Log`, `Set`, `Open`, `Ctx`, etc.

The command `Set` is used to create a variable, to give it an initial value, or to change its value when the variable already exists. E.g. the command

```
set(Call_Stmt, Body_Stmts(2)) -- in (11)
```

creates a variable called `Call_Stmt` with the value provided by the second element of the list variable `Body_Stmts`.

In the other variant, the `Set` command has only one parameter; it is used for declaring an undefined context variable, e.g. `Set (Ctx)` in (1).

Most of the ASIS queries are functions yielding a result as expected by their name. ASIStint mimics exactly this behavior. E.g. the ASIStint function call `Prefix (First_Param)` in (14) calls the ASIS query `Prefix`.

3 ASIStint language description

Overview

The ASIStint language has a very simple structure; it is designed for interactive use and immediate interpretation. To some extent, the ASIStint command language is a functional language.

An ASIStint program is a sequence of function calls, also called commands, when at the level of the program. Function calls may take other function calls as arguments.

An ASIStint function call follows the Ada syntax for a function call, but only positional parameter association is allowed.

There are two kinds of functions: ASIS queries and ASIStint-specific functions, which are in turn divided into ASIStint utilities and ASIStint service functions.

ASIStint supports all queries defined by ASIS, except the generic procedure `Traverse_Element`, which does not fit consistently in the interpretation-oriented concept. The ASIS queries keep their names in ASIStint. To avoid ambiguous calls, there are two exceptions to this rule:

`Asis.Statements.Is_Name_Repeated` is named `Statements_Is_Name_Repeated`, and `Asis.Declarations.Is_Name_Repeated` is named `Declarations_Is_Name_Repeated`.

When a call to an ASIS query raises an exception, ASIStint intercepts it, and instead issues an error message.

Some ASIStint functions do not return any result, and are therefore like procedures. This holds for all ASIStint utilities, but also for some ASIS queries, e.g. `Initialize`, `Open`, etc.

ASIStint allows the user to define variables of the basic ASIS types and to pass them as parameters to ASIS queries.

Data types

ASIStint supports the following predefined Ada types: `Integer`, `Boolean`, and `String`. However only limited sets of operations are provided: `Add`, `Sub`, `Eq`, `Lt`, and `Gt` for `Integer`; `Not`, `And`, and `Or` for `Boolean`; `Concat`, `Eq`, `Lt`, and `Gt` for `String`. Literals for the predefined Ada types `Integer`, `Boolean` and `String` are supported.

All ASIS composite types are supported by ASIStint.

The values of ASIS enumeration types are mapped to string literals, in conformance to the `'Image` attribute.

The values of the `Ada.Calendar.Time` type are represented by Strings.

For variables of the ASIS list types, i.e. `Element_List`, etc., indexing is defined, e.g. `Param_List(1)` in command (13). Moreover, the function `Length` yields the length of a list.

Variables

ASIStint variables are used to store values returned from queries, in order to display them, and in order to pass them to subsequent calls.

ASIStint variables can be of any ASIStint type.

A variable name is an Ada-style identifier.

Variables are created and assigned by the ASIStint `Set` utility. There are no predefined variables at the start of a session.

ASIStint utilities

`Exit`	Exits the script and returns to the calling script or to interactive mode.
`Help [(name)]`	Displays a list and a brief description of ASIStint utilities, or the syntax of an ASIS query or ASIStint service function.
`If (bool_expr, ASIStint_command1 [,ASIStint_command2])`	Corresponds to a conditional statement. If `bool_expr` is true, then `ASIStint_command1` is interpreted next; otherwise, `ASIStint _command2` is interpreted next.
`Info`	Displays ASIS and ASIStint technical information.
`Log ("filename")`	Opens a file for session logging.
`Log`	Closes the current log file.
`Loglevel (level)`	Sets the log control level to the natural value level.
`Pause`	Pauses the current script, and switches to interactive mode.
`Print (expr)`	Displays the value of the expression `expr`.
`Quit [(exit-status)]`	Stops the program. If `exit-status` is present, ASIStint passes that value to the calling execution environment.
`Run ("filename")`	Launches the script in the file `"filename"`, reading further commands from it.
`Run`	Resumes the current script, which was previously paused.
`Set (ID, expr)`	Creates the variable `ID`, if it does not yet exist, and assigns the value of `expr` to `ID`.
`Set (ID)`	Creates an undefined context variable with name `ID`.

4 The ASIStint execution environment

When working with ASIStint, the user can input ASIStint commands from the keyboard. During an interactive session or within a script, it is possible to call another script with the command `Run` and to return to the callee by the command `Exit`. A script might also hand over control to the interactive session by the `Pause` command. The parameterless command `Run` then resumes the paused script.

ASIStint always records all commands in the file "input.log". Together with the input, output produced by ASIStint is always recorded in the file "session.log". The output consists of the results of the `Print` command, and of error messages and warnings issued by ASIStint.

It is also possible to have a filtered log of a session. The `Loglevel` command specifies the level of detail of filtering. The `Log` command provides a file for this kind of filtered logging, and starts, suspends, or resumes logging.

5 Implementation approach and limitations

To call ASIS queries, ASIStint uses look-up tables implemented as one-dimensional arrays indexed by enumeration types; the index values correspond to names of ASIS queries and the components are references to the queries themselves. ASIS queries are grouped into separate tables depending on their parameter and result profiles.

When a user submits a call to an ASIS query, ASIStint takes the query name as a string and passes it to the Value attribute in order to get the corresponding enumeration literal. For searching in the right table, ASIStint then determines the profile of the query. The enumeration literal is then used as an index into the table. In the case of overloaded queries (e.g. the `Debug_Image` query), multiple attempts are made to match the corresponding query.

ASIStint supports all queries defined by ASIS, except the generic `Traverse_Element` procedure. This generic procedure is not and will not be implemented in ASIStint. Indeed we do not see a simple solution for generic instantiation with user-defined procedures as actuals, within an interpretation-oriented environment.

ASIStint is a pure ASIS application, and does not depend in any way on the GNAT compiler. It is therefore fully portable, and can be used with any ASIS implementation.

6 Uses and benefits of ASIStint

6.1 Learning ASIS and experimenting with it

ASIS is a large piece of software, consisting of many packages and queries. To get acquainted with it, it is helpful to be able to play around with ASIS queries, to see how they should be used, and what information they return. It is really a pain to write, and then to compile and link, a full-blown ASIS application just to try out a few queries.

Since ASIStint is an interactive interpreter, it provides the user with a nice capability of "on-line" experimenting with ASIS. User errors, such as calling an ASIS query for an inappropriate element, do not lead to the termination of the interactive session. Indeed, all exceptions raised by ASIS are intercepted. ASIStint informs the user with an error message, and the user can then continue the session.

We are convinced that its interactive nature and its "error-forgiving" capability make ASIStint a good tool for learning ASIS and experimenting with it.

6.2 Testing an ASIS implementation

Testing a program means executing it to find errors in its implementation. A test set consists of data to be submitted to the program and the program's expected results.

Clearly, this view is limited to "main" programs, and does not apply to a set of software components, i.e. a binding or another kind of interface, like an ASIS implementation. To apply the approach, many "main" programs have to be written, and suitable test data must be devised. This approach was indeed used in the validation suite for the ASIS version for Ada 87. Each test in this suite consisted of three parts: an ASIS application, an Ada library to be processed by this application, and the expected results. It may be worthy to note that the suite had self-checking capabilities: special code fragments in the test program were checking the results against predefined values.

Unfortunately, this approach is extremely cumbersome. To say it again: When we want to test a query, we first have to write an Ada program (this is the data part of the test run) containing a construct to be retrieved by the query, then we have to write an ASIS application (also an Ada program) issuing the query. After having compiled the Ada program, and after having compiled and linked the ASIS application, we now can run the latter one with the Ada program as an input.

Clearly, both Ada programs may have errors, and therefore may not pass during their first compilation. Furthermore, even for a trivial ASIS application, compilation time may be lengthy and the size of the executable substantial, because of the complexity and size of the underlying ASIS implementation. The approach is therefore also time-consuming and resource-intensive.

Let's now see the possible benefits of using ASIStint for testing an ASIS implementation. The test cases can be input and debugged interactively, can be checked immediately by ASIStint, and their results can be inspected by the "tester". As we have seen, the input and results of such an interactive session are recorded. Further on, the corresponding script can be replayed, and the new results compared with the old one, e.g. when performing regression testing.

The development of ASIS-for-GNAT benefited greatly from ASIStint, since we used it as a debugging aid.

7 Conclusions

We have shown that ASIStint is a useful tool for learning ASIS, or experimenting with it, and for testing an ASIS implementation. In all these cases, the interpretation-based approach has major advantages over compilation, because ASIS is such a large software subsystem.

8 References

[1] ASIS documents are available electronically on the World Wide Web:
`http://www.acm.org/sigada/WG/asiswg/asiswg.html`
or by anonymous ftp:
`ftp://sw-eng.falls-church.va.us/public/AdaIC/work-grp/asiswg`

[2] 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

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

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

[5] ASIS-for-GNAT is available electronically from the ASIS-for-GNAT Home Page:
`http://such.srcc.msu.su/asis/`
or from LGL-EPFL by anonymous ftp:
`ftp://lglftp.epfl.ch/pub/ASIS`