# Quality-for-ASIS: A Portable Testing Facility for ASIS

Alfred Strohmeier[1], Vasiliy Fofanov[1], Sergey Rybin[2], and Stéphane Barbey[1]

[1]*Swiss Federal Institute of Technology*
*Department of Computer Science*
*Software Engineering Laboratory*
*1015 Lausanne, Switzerland*
*{alfred.strohmeier, stephane.barbey}@epfl.ch, fofanov@bigfoot.com*

[2]*Scientific Research Computer Center*
*Moscow State University*
*Vorob'evi Gori*
*Moscow 119899, Russia*
*rybin@gnat.com*

**Abstract**: This paper describes the project Quality-for-ASIS, aiming at the development of an extensive testing facility for ASIS implementations. First the specific problems and requirements are presented. After a section about the basic concepts of ASIS and after a short introduction to testing, the designs and implementations for testing important subsets of ASIS are described. Finally, adequacy coverage statistics for a test set based on the ACVC compiler validation suite are provided.

**Keywords**: Ada, ASIS, Testing, Black-Box Testing, Specification-Based Testing.

## 1. Introduction

The Ada Semantic Interface Specification (ASIS) [1] is an interface between an Ada environment, and any tool or application requiring statically-determinable information from this environment.

This paper reports about Quality-for-ASIS, a project aiming at the development of a portable testing facility for ASIS implementations. The facility is composed of test drivers, test sets and test coverage measurement tools. Quality-for-ASIS is developed in a joint project of Moscow University and the Swiss Federal Institute of Technology in Lausanne, within the ASIS-for-GNAT project [3, 4, 5, 7]. It is used for testing ASIS-for-GNAT, the ASIS implementation for the GNAT compiler [6].

Here are some of the specifics of the project:

- ASIS is a large library. ASIS is composed of 21 packages, two of which are optional. These packages define 349 queries.
- ASIS is a library and not a program. To exercise an ASIS query, a driver calling the query must first be implemented. In the case of ASIS, such a driver is called an ASIS application.

Paper accepted for publication in the proceedings of the 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, 1998.

- As usual testing cannot be exhaustive. Therefore coverage criteria must be defined, and the testing facility must be able to measure coverage in order to assess the quality of test sets.
- The input to an ASIS application is not just some simple data, but an Ada compilation unit, or a set of such units. So the test “data” are costly to develop. By using the ACVC test suite, we try to solve this problem.
- The results of many queries are not directly observable, because they are of some private type. Ways must therefore be find to observe indirectly the results of such queries, e.g. in order to compare them with expected results.
- The procedure Traverse_Element in the package Asis.Iterator is generic, which raises the problem of testing generic units.
- The testing facility should be as portable as possible.
- The test drivers should be self-checking, in order to provide for low-cost automatic regression testing.

The paper starts with a presentation of the basic ASIS concepts. A short introduction to testing follows. In some length, we then show how this testing “theory” can be applied to the generic procedure Traverse_Element. Some testing criteria for ASIS implementations are then defined. The designs of drivers and oracles for different classes of ASIS queries are then described. Finally, adequacy measurements obtained for the ACVC-based test set are reported for the ASIS structural queries.

## 2. Basic ASIS Concepts

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.

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;
```

*Element* is the term used by ASIS for a valid Ada syntax construct. The procedure body is an element. It can be decomposed by calling *structural queries*. The query Body_Declarative_Items will return the list of declarations, each declaration being an element in turn. The result of the query Body_Declarative_Items is therefore of type

Element_List. The query Body_Statements will return a list of elements, each being of the *kind* A_Statement, as defined in the enumeration type Element_Kinds. For any of these statements, it will be necessary to determine its so-called *subordinate kind* before continuing top-down parsing. For the first statement in the list, the call to the function Statement_Kind will yield the value An_Assignment_Statement. The statement can then be processed by calling an appropriate structural query, in this case Assignment_Variable_Name or Assignment_Expression for retrieving the left-hand and right-hand parts of the assignment. As soon as we know that the left-hand side of the assignment is a simple name, we can get its string image by calling Name_Image, which is one of the queries returning the image of a terminal. In order to find out about the type of the left-hand part of the assignment, it is possible to call the *semantic query* Corresponding_Expression_Type, which will return the element corresponding to the type declaration. What we said so far, can be summed up by this (correct, except for unconstrained array handling) piece of code of an ASIS application:

```
declare
   Unit: Asis.Compilation_Unit;
   Procedure_Body: Asis.Element;
   Declaration_List: Asis.Element_List;
   Stmt_List: Asis.Element_List;
   First_Stmt: Asis.Element;
   Declaration: Asis.Element;
begin
   -- initialize Unit
   Procedure_Body := Unit_Declaration(Unit);
   Declaration_List := Body_Declarative_Items(Procedure_Body);
   Stmt_List := Statement_List(Procedure_Body);
   -- Get the list of statements of the body.
   First_Stmt := Statement_List(1);
   -- Select the first statement.
   if Statement_Kind(First_Stmt = An_Assignment_Statement then
      Put(Name_Image(Assignment_Variable_Name(First_Stmt)));
      Declaration := Corresponding_Expression_Type
                        (Assignment_Variable_Name(First_Stmt));
   end if;
   -- etc.
end;
```

The generic iterator Traverse_Element is defined in the package ASIS.Iterator. It is a generic procedure used to traverse the syntax tree corresponding to an element. Here is its specification:

```
generic
   type State_Information is limited private;
   with procedure Pre_Operation
       (Element : in     Asis.Element;
        Control : in out Traverse_Control;
        State   : in out State_Information) is <>;
   with procedure Post_Operation
       (Element : in     Asis.Element;
        Control : in out Traverse_Control;
        State   : in out State_Information) is <>;
```

```
procedure Traverse_Element
   (Element : in     Asis.Element;
    Control : in out Traverse_Control;
    State   : in out State_Information);
```

Elements are traversed in top-to-bottom order, and at a given level from left-to-right following the textual order of the component elements within the parent element. For each element, the formal procedure Pre_Operation is called when first visiting the element. Each of that element's children are then visited recursively, and finally the formal procedure Post_Operation is called for the element.

## 3. Short Introduction to Testing

This section defines some terminology related to testing and provides a short overview of the theory behind testing. The interested reader may refer to [12] for a more comprehensive introduction.

Testing is the most popular verification technique. It aims at answering the following question:

> "Does a program *P* correctly implement a model *M*?"

In order to do so, it tries to find counterexamples showing that it does not [11]. This question is named the *verification requirement*.

A *test case* is an input on which the program under test is executed during testing. A *test set* is a set of test cases. After execution of a test case, the result generated by the program is compared with the one expected by the model. The means used to perform this comparision, be it a program or human inspection, is called an *oracle*.

The answer to the verification requirement is undecidable in the general case. Already in the mid-70s, Goodenough and Gerhart [13] pointed out that the central question was therefore to find a *test criterion*, i.e. a criterion that states what is an adequate test set. A great number of such critera have been proposed and investigated. The criterion is often defined by *sampling* the input domain of the program to exercise it with interesting cases only. Thus, the verification requirement becomes:

> "How well does a program *P* implement a model *M*
> according to a sampling criterion *C*?"

Sampling criteria are an essential part of any testing approach. There are two kinds of sampling criteria: correctness and fault hypotheses. The aim of a correctness criterion is to show that the program satisfies its model (that it is correct), whereas the aim of fault hypotheses is to show that particular classes of faults are not present. Test selection based on a correctness criterion should include test cases that represent the essential features of the model, and should establish that the test cases are feature-sufficient, i.e. that they are sufficiently representative for giving confidence that the program implements its model.

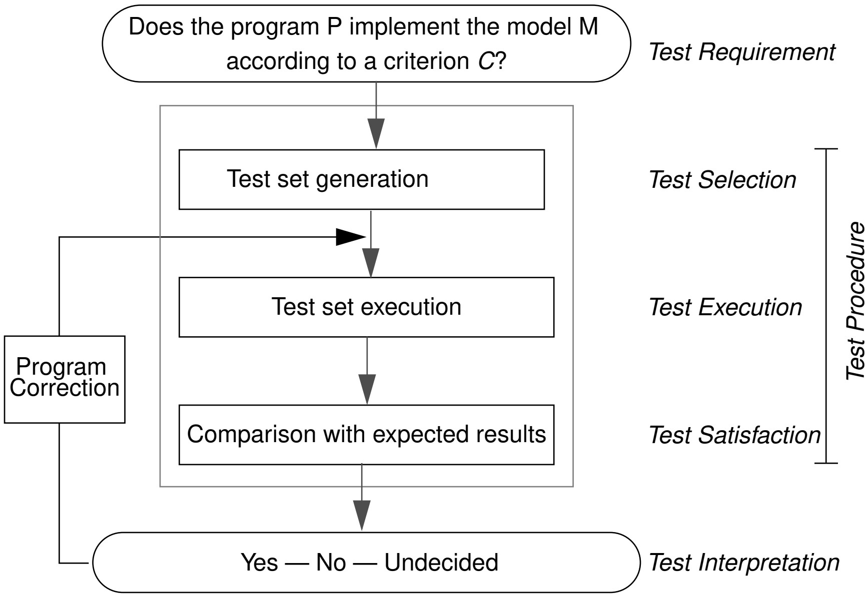


**Fig. 1.** Test process for specification-based testing

The quality of a test set is measured by its *adequacy*, which needs appropriate measurement techniques [9, 12]. Adequacy is usually a function of the coverage of the sampling criteria by a test set, but can also be a function of hypotheses set on the program [8]. The classical mutation analysis is an example of such a measurement [10]. In this approach, faults are injected in a program, and the quality of a test set is defined by the number of detected faults. For *program-based testing*, also called *white-box testing*, coverage measures include the percentage of called subprograms, executed statements, or followed simple paths. For s*pecification-based testing*, also called *black-box testing*, adequacy might be measured by coverage of axioms as found in the formal specification, or coverage of the input domain after partitioning in equivalence classes.

The test process (see figure 1) usually involves three steps:

- A test selection step, in which a test set is generated that will validate the test requirement, according to the previously chosen sampling criterion. The test selection procedure should also decide what the expected results of the test cases are.
- A test execution step, in which all the test cases in the set are executed.
- A test satisfaction step, in which the results obtained during the test execution phase are compared with the expected results. This last step is commonly performed by the help of an oracle.

The findings from this comparison are used to validate, reject or leave undecided the verification requirement. The last case happens when test execution does not return a result, or when the oracle is unable to perform the comparision.

Up to now, the test set adequacy criterion was used as a stopping rule; it is therefore called a test case selection criterion. But an adequacy criterion can also be used to decide whether a given test is adequate, or to measure the adequacy of a given test set. Such a rule or measure is usually referred to as a *test set adequacy criterion*.

In both approaches the fundamental underlying notion is that of test adequacy. As explained in [12], in many cases a criterion can be easily transformed from one form to the other.

## 4. Testing the Generic Procedure Traverse_Element

To make things more concrete, we will first explain how the outlined testing approach can be applied to the generic procedure Traverse_Element.

The adequacy criterion is to exercise Traverse_Element on every possible syntactical element of the Ada language at least once. ASIS itself specifies what is a syntactical element, and its definition will be used, because it is only slightly different from the one defined in the Ada language reference manual [2]. For the time being, elements corresponding to implementation-dependent Ada constructs, e.g. implementation-specific attributes and pragmas, will not be included in the adequacy criterion, because one of the goals is to develop a portable testing facility.

Traverse_Element is not a main program, so a test driver must be devised. Because Traverse_Element is a generic procedure, the test driver will have to instantiate it, and then call instantiations.

The only syntactical element that can be a direct input to an ASIS application, and therefore to the driver, is a compilation unit. For any other syntactical element, it is necessary to create an *execution context*. Such a context is formed by the sequence of enclosing elements, starting from the compilation unit. This approach provides a path from a test data input to a call to an instantiation of Traverse_Element on some actual syntactical element. To implement it, it is sufficient to call Traverse_Element iteratively for all the elements in the context, starting from the compilation unit.

However, traversal of a given element cannot be observed directly, and an observation context must therefore be created. The idea is to use traversal for reconstructing the source code of the traversed element. As a result, the driver will rewrite the source code.

Finally, it must be decided if the generated source code is equivalent to the original one. This is the task of the oracle. Requiring that the source codes must be the same would be too much. Indeed, extra blank characters might be suppressed, formal parameters of subprograms might be normalized: “X, Y: Integer” becoming “X: in

Integer; Y: in Integer", etc. Comparing the textual representations taking into account all these variations would be quite difficult. The idea is therefore to ensure their operational equivalence, by compiling the source codes and comparing the generated object codes. If they are the same, the oracle will decide that the test case passed successfully. Even though the comparison itself might be somewhat tricky, e.g. in the presence of a time stamp inside the object code, we think this approach can be adapted to many compilation systems. Its use with GNAT [6] is straightforward.

Let's put it all together. A test case is a compilation unit. Traverse_Element is instantiated with appropriate pre- and post-operations for writing out the source code of the element passed as a parameter. The test driver rewrites the source code of the test case by traversing it recursively by means of calls to the instantiation of Traverse_Element. The oracle finally checks if the two source codes are equivalent.

In the case of ASIS-for-GNAT, Traverse_Element is in fact implemented by recursive calls to itself. The source code of Traverse_Element itself can therefore be used for implementing the driver.

The rewriting instantiation of Traverse_Element was exercised on the ACVC test suite. This suite contains all possible syntactic elements (but no implementation-dependent constructs), and therefore satisfies the coverage criterion fully at 100%.

The implementation of Traverse_Element in ASIS-for-GNAT passed successfully this test.

## 5. Testing Criteria for ASIS Implementations

The goal is specification-based testing of an ASIS implementation. Following figure 1, the program P is an implementation of ASIS, e.g. ASIS-for-GNAT, and the model M is the specification of ASIS, as defined in [1]. "P implements the model M" means that for all legal Ada compilation units all the ASIS queries yield correct results as defined by their specifications. Because ASIS is a library, test drivers must be built.

Since the test unit is a library, sampling or adequacy criteria are to be expressed in terms of coverage of the library. Only correctness criteria will be proposed.

ASIS queries are completely defined in the sense that they *always* produce a result, either by returning a result value or by raising a well-defined exception.

**Criterion 1**

To meet the first criterion, any ASIS query must be called with a return result, at least once. Exception raising is therefore not covered by this criterion.

**Criterion 2**

Many ASIS queries have one of the following profiles:

Element -> Element

Element -> Element_List

Element $\times$ Boolean -> Element_List

Element -> some classification enumeration type

As already explained, elements are language constructs, and are classified, e.g. A_Declaration or A_Statement. There are 10 different element kinds, not including the so-called Not_An_Element. These different element kinds are in turn subdivided into a hierarchy of so-called subordinate kinds. On the overall, there are about 300 subordinate kinds. But usually, a query can validly be called only for a selection of subordinate kinds. Otherwise, an exception is raised.

To achieve criterion 2 coverage, any query must be called with all possible valid combinations of parameters and results.

**Criterion 3**

Sometimes a query cannot return a meaningful result. It then raises an exception, as defined by the ASIS definition. To fulfill criterion 3, any query must be exercised with all its possible exceptions raised.

Although coverage of criterion 3 is certainly desirable, Quality-for-ASIS does not yet take it into account.

## 6. Drivers and Oracles

The figure 2 shows the current state of the Quality-for-ASIS project, especially the availability of drivers and oracles for different sorts of queries.

**Structural queries and images of terminals**

For structural queries (except Enclosing_Element) and the queries yielding the image of a terminal, the test driver is the same as for the Traverse_Element generic procedure: it rewrites the source code of the Ada unit serving as an input, by parsing it top-down, and making calls to the structural queries for decomposing the encountered syntactical elements. The context of the call to a query is in some way top-down parsing of a compilation unit. The observation context is built by producing a textual representation of the element processed by the query. Finally, the same oracle as the one described for testing the Traverse_Element generic procedure is used for comparing the generated source code with the original one.

**Semantic queries**

The test driver for semantic queries also performs a traversal of the compilation unit submitted as an input. For any visited element, all applicable semantic queries are

| | | | | |
|---|---|---|---|---|
| Total ASIS queries | | | 349 | 100% |
| Tested | | | | |
| | Structural queries | 125 | | |
| | Images of terminals | 6 | | |
| | Traverse_Element | 1 | | |
| Subtotal | | | 132 | 38% |
| Implemented | | | | |
| | Semantic queries | 31 | | |
| Subtotal | | | 31 | 9% |
| Designed | | | | |
| | Classification queries | 35 | | |
| | Enclosing_Element | 1 | | |
| Subtotal | | | 36 | 10% |
| Done | | | 199 | 57% |
| To be done | | | 150 | 43% |

**Fig. 2** State of the Quality-for-ASIS project

called. The oracle works as follows: for each semantic query, it is first checked that the kind of the result is an allowed one. For queries having a complementary query, such as Corresponding_Body and Corresponding_Declaration, it is checked that composing the calls, e.g.

```
Corresponding_Body (Corresponding_Declaration (Element))
```

correctly returns the original Element. The check itself is performed by calling the ASIS query Is_Identical. Finally, for elements representing a named construct related to the same entity, a check is made that the names are the same.

The driver for this kind of queries is already implemented.

**Classification queries**

Once again, the test driver is based on top-down parsing of a compilation unit. For every visited element, the results returned by all classification queries are recorded sequentially in a text file. The conversion to a string is simply performed by applying the ‘Image attribute to the enumeration value returned by the query. The generated file must then be manually reviewed, but can further on be used as a reference file, e.g. for non-regression testing or for testing another ASIS implementation, because top-down parsing of a given compilation unit will almost always yield the same sequence of elements. Possible exceptions are due to implementation permissions granted by the ASIS specification.

The driver for this kind of queries is not yet implemented and reference files are not yet available.

**The Enclosing_Element query**

The Enclosing_Element query is in fact a structural query, but unlike all other structural queries used for top-down parsing, it performs a bottom-up step.

The implementation of a test driver, including an oracle, is straightforward. It is again based on top-down parsing of a compilation unit, implemented by instantiating Traverse_Element. At each step down in traversal, the pre-operation pushes the current element on the stack. Enclosing_Element is then called for all its children and the results are compared with the element at the top of the stack by calling the Is_Identical query. Finally, when coming back to an element, the post-operation will pop it off the stack.

## 7. Use of the ACVC Test Suite as a Test Set

The Ada Compiler Validation Capability, ACVC, now in version ACVC 2.1, provides an extensive set of Ada programs used to validate Ada compilers. This test suite is supposed to cover quite well the full language. It is therefore a good starting point for a test set. The so-called B tests must however be eliminated, because they correspond to illegal Ada units raising compilation errors; therefore they cannot be the input to an ASIS application. What remains are about 4'057 Ada compilation units.

At the time being, ASIS-for-GNAT is not able to deal with library-level instantiations of generic subprograms and with some kinds of static expressions that are transformed into values by the GNAT compiler. As a consequence, 61 compilation units belonging to the ACVC suite, i.e. about 1.5%, could not be included in the test set. In some sense, only partial correctness of ASIS-for-GNAT will be demonstrated by this test set.

## 8. Measurement of Adequacy

In ASIS-for-GNAT, the generic Traverse_Element procedure is a plain ASIS application, and its source code is publicly available. In order to gather statistics about test cases, its source code was modified by adding a logging mechanism: immediately before a query is called, its name is recorded. This generic procedure is called Traverse_Element_And_Log.

First, the Traverse_Element_And_Log procedure was instantiated with empty pre- and post-operations. This Gather_Query_Names procedure was then applied to the ACVC-based test set. The result is shown in figure 3. All structural queries are called except Enclosing_Element, with which we deal separately, and Body_Block_Statement, which is deprecated. The adequacy criterion 1 is therefore fully satisfied by the ACVC-based test set for the structural queries.

| | | | |
|---|---|---|---|
| Total structural queries | | 126 | 100% |
| (except Enclosing_Element) | Called | 125 | 99% |
| | Not called | 1 | 1% |

**Fig. 3** Adequacy criterion 1 for structural queries

Traverse_Element_And_Log was further extended to track parameter-result combinations, whenever a structural query is called. Results were then gathered in a file for the ACVC-based test set.

Manually, by inspection of the ASIS specification, the legal combinations were entered in a reference file. The two files were then compared. The results are shown in figure 4. The total number of legal combinations is 1212, 883 of which are exercised by the test set. For 82 structural queries, all combinations are exercised. Although all not called profiles correspond to syntactically correct constructs, most of them will never be found in “useful” Ada programs; e.g. a goto or raise statement does not make sense immediately within a loop statement (without being part of a compound statement). As a consequence, the ACVC suite does not contain these combinations, because it tries to avoid idiosyncratic constructs. However, the question remains open if an ASIS implementation should be tested on such combinations. Indeed, it would be perfectly possible, and perhaps even desirable, to have a tool implemented as an ASIS application, which searches for undesirable combinations, either for helping the programmer write better code, or for assessing code quality.

In conclusion, the ACVC-based test set fulfills criterion 2 at a level of 83% for structural queries.

If needed, the test set could be extended. For instance, the following test case:

```
package addon is
   type t is null record;
end addon;
```

can be used to check the Record_Definition query for a parameter of kind A_Record_Type_Definition and the result of kind A_Null_Record_Definition.

| | | | |
|---|---|---|---|
| Total number of profiles | | 1212 | 100% |
| | Called profiles | 883 | 83% |
| | Not called profiles | 329 | 27% |
| | Completely covered queries | 82 | 65% |

**Fig. 4** Adequacy criterion 2 for structural queries

The adequacy criterion statistics for the Enclosing_Element query are the same as for the set of the other structural querie, because it is in some way the inverse function.

For the semantic queries and the classification queries, we have not yet measured the adequacy of the ACVC-based test set.

## 9. Conclusions and Future Work

A testing approach, including adequacy criteria and a test set, was defined for an important part of ASIS. Testing facilities have been implemented for the structural queries, queries yielding the image of terminals, the semantic queries and the Traverse_Element query. Testing approaches were designed for the classification queries and the Enclosing_Element query. These 199 queries count for 57% of all the queries found in ASIS.

It was shown that the ACVC-based test set fulfils quite well the adequacy criteria for the structural queries, the queries yielding the image of terminals, and the Traverse_Element query. Adequacy for the semantic queries, the classification queries, and the Enclosing_Element query will be investigated in the future.

Work has still to be done for the other kinds of queries, e.g. those working with ASIS Contexts, Compilation_Units, etc.

However, structural queries together with Traverse_Element are used whenever a tool needs parsing of Ada source code. The current state of Quality-for-ASIS testing facility therefore comprises testing of a very important subset of ASIS.

## Acknowledgements

The first version of the source rewriting tool was implemented by Jean-Charles Marteau and Serge Reboul, then working for Sema Group in Grenoble.

The ASIS-for-GNAT project is partially funded by a grant from the Swiss Federal Government, grant no 7SUPJ048247.

## 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, Frankfurt - Germany, October 2-6 1995, LNCS no 1031, Springer, 1995, 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-Eu-

rope International Conference on Reliable Software Technologies Proceedings, LNCS no 1088, Springer, pp. 298-311, 1996.

[5] ASIS-for-GNAT is available electronically from LGL-EPFL by anonymous ftp: ftp://lglftp.epfl.ch/pub/ASIS.

[6] E. Schonberg, B. Banner; The GNAT Project: A GNU-Ada 9X Compiler; TRI-Ada'94 Proceedings, ACM Press, 1994; pp. 48-57. See also http://www.gnat.com.

[7] V. Fofanov, S. Rybin and A. Strohmeier, "ASIStint: An Interactive ASIS Interpreter", in Susan Carlson (Ed.), Proceedings of TRI-Ada'97, St. Louis, USA, ACM Press, 1997, pp. 205-209.

[8] Stéphane Barbey. *Test Selection for Specification-Based Testing of Object-Oriented Software Based on Formal Specifications*. PhD thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), December 1997. Ph.D. Thesis 1753.

[9] Robert V. Binder. Testing object-oriented software: a survey. *Journal of Testing, Verification and Reliability*, 6:125–252, September 1996.

[10] Muriel Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of 3rd International Symposium on Software Testing and Analysis (ISSTA-3)*, pages 158–171, San Diego, California, USA, January 1996.

[11] Glenford J. Myers. *The Art of Software Testing*. Business Data Processing: a Wiley Series. John Wiley & Sons, 1979.

[12] Hong Zhu, Patrick A. V. Hall, John H. R. May. Software Unit Test Coverage and Adequacy. In ACM Computing Surveys, Vol. 29, No. 4, Dec 1997. pp. 366-427.

[13] John B. Goodenough, S. L. Gerhart. Toward a theory of test data selection. In IEEE Trans. Softw. Eng. SE-3, June 1975.