ADL — An Interface Definition Language for Specifying and Testing Software

Sriram Sankar* and **Roger Hayes**†

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue, UMTV29-112
Mountain View, California 94043-1100

Abstract

This paper describes an interface definition language called ADL which extends OMG's CORBA interface definition language with formal specification constructs. In addition to ADL's use in formal documentation, ADL's primary use is for testing software. ADL can be adapted for use with most programming languages.

This paper also presents an overview of a testing technology based on ADL and presents the highlights of a test-data description language (TDD) used to describe test-data.

1 Introduction

With the advent of large software systems, a lot of work has gone into designing software languages in which systems may be divided into many reasonably sized components, each of which can be dealt with more easily. This division requires a mechanism for the individual components to coexist and be developed simultaneously. Almost universally, this has been achieved by splitting each component into two parts:

1. its *interface* — a set of declarations of software entities (such as types and operations) which are provided by this component and which may be used by other components; and

2. its *implementation* — the implementation of the declarations in the interface.

Typically, the implementations are isolated from the rest of the software system — they may be modified without having to rebuild the rest of the system. Most common programming languages implement these concepts in some way or the other. C provides a rudimentary implementation of interfaces using header files, C++ provides classes, and Ada provides packages.

More recently, an effort has been undertaken by OMG called CORBA [2] (Common Object Request Broker Architecture) where the goal is to design and implement a distributed environment within which objects may coexist and interact with each other. These objects may be implemented in different programming languages. The interfaces of these objects are written out in a special language called IDL (Interface Definition Language). CORBA defines bindings from IDL to programming languages such as C and C++ — hence interfaces written in IDL may be implemented in C or C++. Bindings for many other programming languages are forthcoming.

The main problem with these interfaces is that they provide too little information about the components they describe. For example, the interface of a function usually only describes its parameter and result type profile. It would be nice to be able to describe other important properties of components. For this purpose (among others), a new family of languages called *specification languages* [5] have been designed. Specification languages may be used to describe the relevant aspects of the behavior of software components without having to overcommit an implementation. Examples of specification languages are Anna [6], Larch [3], and Z [7].

We, at the PrimaVera group at Sun Microsystems Laboratories, Inc., have been working on specification techniques for the past eight years. Our focus has been to apply formal specification techniques to software testing. We have concentrated our efforts to developing

*phone: (415)336-6230; email: sriram.sankar@sun.com

†phone: (415)336-4237; email: roger.hayes@sun.com

testing tools for use with IDL interfaces. Given that the IDL technology can be made to work with any programming language for which bindings have been defined, our testing technology can also be made to work with any of these programming languages.

Pilot implementations of our test system have been undertaken, and we have had significant results from these experiments. An example of an anomaly discovered by our testing tools was the way in which the `write` system call updated the last modification time of a file. On a particular version of the UNIX operating system, performing a write with a 0 byte data value changed the modification time on local files, but did not in the case of remote files. The anomaly had not been detected earlier, even though standard rigorous testing schemes had been applied on this system call. The reason we were able to detect this was due to the systematic nature of developing the specifications and test-data descriptions.

After several years of internal development and deployment, we decided to make our work externally available. The PrimaVera technology was submitted in response to a request for proposals for automated testing technology issued by X/Open, and was selected for a joint research project sponsored by a research grant from Information-Technology Promotion Agency, Japan (IPA), a governmental organization under Ministry of International Trade and Industry (MITI).

Content of this paper. This paper describes the specification language we have developed as part of our testing technology. This language, ADL (Assertion Definition Language), enhances IDL with constructs for behavior description of interface constituents. Given that programmers will usually be most comfortable writing specifications in a syntax familiar to them (*i.e.*, a syntax similar to the programming language they use), ADL has been designed as a language framework providing high-level specification concepts. These concepts may be *specialized* for any programming language by rendering them in a syntax similar to that programming language. In this paper, the ADL specialization for ANSI C is used in examples[1].

The highlights of ADL are listed in Section 1.1. A more detailed description with examples is provided in Section 2. Finally, Section 3 describes ongoing research activity in the language design.

In addition to ADL, this paper also presents an overview of our testing technology in Section 1.2. Full details of the testing technology is the subject of a future paper.

More information on the PrimaVera ADL project is available in [8] and [9].

1.1 ADL

ADL is a language designed for formal specification of software components. Its interface definition constructs are borrowed from IDL and provides mechanisms for defining types, objects, and operations. In addition, ADL defines a set of general-purpose specification concepts applicable for the specification of software written in most programming languages. Some of the key features of ADL are listed below:

- ADL is a language framework that provides a set of high-level specification concepts. These concepts may be specialized for use with a programming language by rendering them into a syntax similar to that of the programming language.
- All ADL specifications are post-conditions on operations (or functions) of software components. Therefore, ADL specifications are constraints on the program state at the time of termination of operation evaluations.
- ADL specifications are written as separate units — *i.e.*, they are not embedded in the program. The ADL specification writer defines a binding between the specifications and the functions in the program to provide the necessary association.
- ADL specifications may be partial. That is, the complete details of the function do not have to be written in ADL. Typically, ADL specifications are augmented with informal natural language documentation.
- ADL provides specialized constructs for the specification of errors. Most specifications written as natural language documents (such as UNIX `man` pages) describe error situations separately. ADL's error specification constructs allow a formal specification to be organized in a similar manner.
- ADL constructs are designed to allow translation of formal specifications into natural language documents. ADL's constructs are at a high-level of abstraction and permit a specification writer to write specifications very similar to the way they would do it in a natural language, hence the translation process is straightforward.

[1] Readers familiar with IDL may realize that we have made minor modifications to the underlying IDL syntax in the specialization process. However, we do maintain a mapping to IDL.

- ADL is well-suited for the purpose of testing software components. Difficult to evaluate constructs such as quantifiers have been excluded from the language for the time being.

1.2 ADL Testing Technology

To perform software testing, we need: *(1)* a piece of software to test; *(2)* a description of what this software does; and *(3)* a set of test-data on which to test the software.

In our case, the piece of software is any ANSI C program, and the description of what this software does is the ADL specification of this program.

For the third component, we have implemented one method for generating test-data. Test-data is described symbolically in a special ***test-data description*** language (TDD). This method is specifically for use with unit testing. We have future plans to develop similar schemes for other kinds of testing (such as sequence testing).

When using TDD, the program under test is run with many different test inputs in a systematic manner. Correct behavior is determined by examining the results of the program or function in terms of the specification describing its behavior. Correct execution of the program on these test inputs increases the level of confidence in the program.

The TDD language offers the test designer a formal and structured framework for describing test-data. TDD provides a structure for characterizing and documenting the data used in testing. Through the use of TDD's syntax, test-data becomes the subject of a design process. The important features of TDD are listed below:

- Data is characterized in an abstract and systematic way. By using a formal system for notating the description of test data, we focus on the test designer's intention rather than on the details of generating a particular instance of test input data.
- Test data is generated without prejudice. By isolating the description of test data from its realization, we explore what might otherwise have been blind spots. TDD encodes the designer's insight into formal descriptions, which are decompositions of the properties of the data. These descriptions are recombined into test cases. This ensures that all combinations of properties are tested; without the decomposition/recomposition process, it is very easy to omit an important test because it does not occur in an imagined scenario of use.
- Input is characterized independently of any particular implementation. TDD describes the input data from the point of view of a user of the tested software component. A TDD description may be constructed with insight into implementations, but its correctness does not depend on a particular implementation. This means that a TDD test suite is portable across implementations.
- Iteration over the test cases is systematic and thorough. The regularity of the process allows for better statistics. The regularity also helps reduce the incidence of errors missed due to oversight.
- Data manufacture is isolated. The messy task of generating actual test values is encapsulated in well-defined functions. These functions, that translate symbolic descriptions into actual values, can be used in manually written tests as well as in ADL-generated tests.

Complete details of the TDD language may be obtained from [9].

2 The ADL Language

ADL is a language framework designed for the formal specification and testing of software components. ADL defines a set of general-purpose specification concepts applicable for the specification of software written in most programming languages. ADL excludes specification concepts that, although useful, are difficult to implement using state of the art technology.

The concepts of the ADL language framework may be specialized for use with a programming language by rendering these concepts into a syntax similar to that of the programming language. This syntax may then be augmented with constructs from the programming language, such as its expression syntax. This approach of defining a language framework that may be specialized for use with any language has been used successfully in other projects, *e.g.*, Larch [3] and Rapide [1].

The ADL language framework has been specialized for use with the C programming language. We intend to specialize ADL for use with C++ and Ada shortly. For the purposes of this paper, ADL will be described through its specialization for the C programming language.

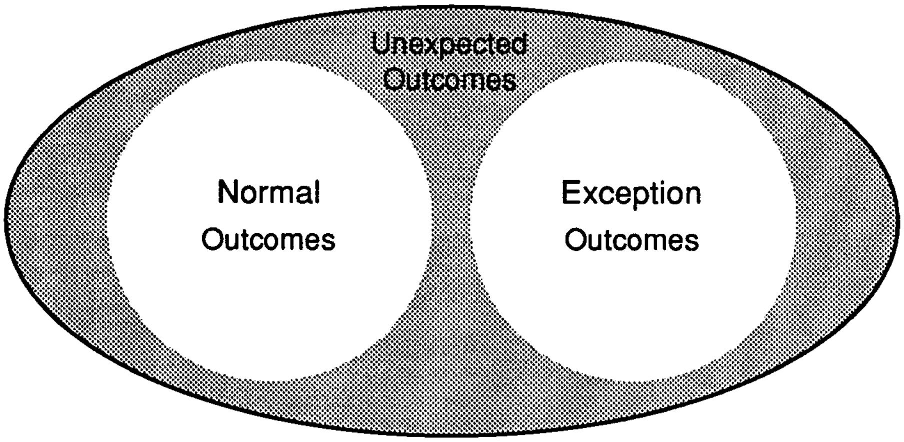

Figure 1: The Possible Outcomes of an Operation Evaluation

Other important aspects of ADL are described below.

- *Post-condition specifications.*
  All ADL specifications are post-conditions on operations (or functions) of software components. Therefore, an ADL specification is a constraint on the program state at the time of termination of operation evaluation. This constraint may be contingent on pre-operation program state by use of the *call-state* operator.

  This is an example of ADL's client orientation. An ADL specification does not give conditions under which the function must be called, but instead tells what will happen if it is called.

- *Non-intrusive.*
  ADL specifications are written as separate units — *i.e.*, they are not embedded in the program (*e.g.*, as in Anna). The ADL specification writer defines a binding between the specifications and the functions in the program to provide the necessary association. This binding provides sufficient information for the ADL testing tools to generate frameworks to test these functions.

  This approach is non-intrusive to the extent that the functions being specified need not be recompiled for testing purposes. Hence, the ADL testing technology may be applied to precompiled code, including code such as operating systems, that by their very nature cannot be recompiled and reloaded in a straightforward manner.

- *Constructs for specification of errors.*
  Figure 1 illustrates the possible outcomes of an operation evaluation. The outcomes can be divided into two categories — *expected* and *unexpected*. Expected outcomes (the white portion of Figure 1) are those that are included in the documented behavior of the function, while unexpected outcomes (the grey portion of Figure 1) are outcomes that are not supposed to happen (*e.g.*, (2+2) evaluating to 5).

  Our research has shown that it is convenient to divide the expected outcomes of an operation into *normal* and *exception* outcomes. This division is usually subjective, but some general guidelines may be laid down. For example the outcome of $(2+2)$ evaluating to 4 is usually considered a normal outcome, while the outcome of $(128+128)$ being an overflow is usually considered an exception outcome.

  ADL provides constructs to separate the handling of normal and exception outcomes of an operation, and to specify against unexpected outcomes.

- *Simple natural language mapping.*
  One of the mandates of the ADL project has been to develop a capability to transform a specification written in ADL into an equivalent natural language representation. This problem is, in general, untractable. However, ADL's constructs are at a high-level of abstraction and permit a specification writer to write specifications very similar to the way they would do it in a natural language. The error specification mechanisms discussed earlier are an example of these high-level

constructs. The task of translating ADL specifications into equivalent natural language documents (*e.g.*, UNIX **man** pages) becomes quite simple if the specification writer adheres to these high-level constructs while writing ADL specifications.

- *Enables testing.*
  The fact that the ADL design emphasizes applicability to testing of software components has already been mentioned. We reiterate this in the context of the other features such as the specification being non-intrusive, and being from a client's point of view. Furthermore, specification constructs such as quantifiers and algebraic specifications have been omitted from the current version of ADL. We have plans to explore the introduction of these constructs in the future.

2.1 ADL Constructs

The constructs provided by the ADL framework are described in this section. Some of these constructs are illustrated through examples in Section 2.2.

An ADL specification is made up of a set of *modules*[2]. Each module encapsulates a set of *constituents* that describe the entities in the C program that are being specified. Modules may also refer to each other's constituents by importing constituents from one module to another.

A constituent of a module may be one of the following:

- *Type constituent.*
  A type constituent defines a type and gives it a name. Its syntax is identical to C type definitions.

- *Object constituent.*
  An object constituent introduces an object[3] and associates it with a type. Its syntax is similar to that of a C object declaration. Objects introduced by object definitions are bound to C objects with the same type.

- *Function constituent.*
  A function constituent introduces a function and specifies its parameter and result types. Its syntax is similar to that of a C function declaration. Functions introduced by function definitions are bound to C functions that have the same parameter and result types.

Function constituents may contain *semantic descriptions.* A semantic description describes the behavior of the C function that is mapped to its function constituent. The semantic description constrains the program state at the end of calls to this C function. A semantic description has two components:

- *Bindings.*
  Bindings are associations between expressions and names. These names may be used subsequently as a short form for their associated expressions.

- *Assertions.*
  An assertion is a boolean expression that must be true whenever control returns from the function constrained by the semantic description.

It is often useful to make use of existing specification concepts while writing assertions. Sometimes these concepts may already exist as part of a module. However, there will be situations where these concepts are missing. In such situations, missing, but necessary, specification concepts can be declared as *auxiliary definitions.* Auxiliary definitions are simply ADL declarations that are visible only within bindings and assertions.

Predefined ADL operators and functions. Some of ADL's primitives for use in semantic descriptions are described below:

- *Call-state operator.*
  The call-state operator ("@") takes one argument and evaluates it at the time of call to the function being specified.

- `normal` *and* `exception`.
  `normal` and `exception` are predefined names that may be bound to boolean expressions that characterize the normal and exception outcomes respectively (see Figure 1).

- *Implication operators.*
  ADL provides the standard logical implication and equivalence operators. It also provides a *exception operator* (<:>), that characterizes error situations by listing the conditions that cause the function to

[2] We are describing the specialization of ADL to C in this paper. In this specialization, an interface in C which is a header file is modeled as a module in ADL. We have reserved the keyword "interface" for use with C++ classes in a forthcoming extension of this ADL specialization.

[3] The word "object" is used here in the same sense as in C.

have an exception outcome and relates them to the error conditions that take place. This operator is used to characterize the exception outcomes of the function being specified.

- **normally**.
  This is a function that characterizes the normal outcomes of the function being specified. It takes a list of boolean parameters that must all be true on any normal outcome.

2.2 ADL Examples

This section provides two examples of an ADL specification of a bank module. The first example defines three operations **balance**, **deposit**, and **withdraw** within a module. These functions map to C functions with similar names and signatures. The specifications written in ADL describe the intended behavior of these C functions. The ADL specification is shown in Figure 2.

The bank specification of Figure 2 contains 7 constituents:

1. **errno**: This is an object constituent. It is defined to be of type **int**. This maps to the standard C global variable **errno**.
2. **NEG_AMT**: This is also an object constituent. It describes a particular value that **errno** can take. This maps to a C integer constant.
3. **INS_FUND**: Just as **NEG_AMT**, this describes another value that **errno** can take and maps to a C integer constant.
4. **acct_no**: This is a type constituent. This defines **acct_no** as another name for **int**.
5. **balance**: This is a function constituent. It describes a function that takes a parameter **acct** of type **acct_no** and returns a value of type **int**. **balance** maps to a C function with the same name, and same parameter and return types.
6. **deposit**: This is another function constituent. It takes two parameters and has a semantic description associated with it. Just as in the case of **balance**, this too maps to a C function with the same name, and same parameter and return types.
7. **withdraw**: This is another function constituent with a more detailed semantic description.

Semantic Descriptions:

The semantic description of **deposit** contains two assertions. The first assertion contains the call-state operator "@". The call-state operator evaluates its argument (in this case **balance(acct)**) in the state at the time the function is called. This assertion states that the balance of account **acct** (*i.e.*, **balance(acct)**) after the call to **deposit** is equal to the sum of the balance before the call and the amount deposited (*i.e.*, **amt**).

The second assertion about **deposit** contains the reserved word **return**. This is used to refer to the value returned by the function (in this case **deposit**). This assertion states that the value returned by **deposit** is the new balance of account **acct**.

The semantic description of **withdraw** contains four bindings (the first four lines) and then a list of assertions. The bindings bind expressions to names. Use of these names in subsequent expressions then refer to the bound expressions.

The first two bindings bind expressions to the special names **exception** and **normal**. The bindings to these names together with their use in specifications characterize the normal and exception outcomes of **withdraw**. **exception** is bound to the expression (**return** == −**1**), while **normal** is bound to the expression **!exception**, (*i.e.*, **!(return** == −**1))**. In addition to providing a binding for **exception** and **normal**, these bindings also define the meanings of the exception operator <:> and the function **normally**. These are described in the following paragraphs.

The next two bindings provide short forms for the expressions (**errno** == **NEG_AMT**) and (**errno** == **INS_FUND**).

The first assertion about **withdraw** contains the exception operator <:>. The exception operator characterizes error situations by listing the conditions that cause the function to have an exception outcome and relating them to the error conditions that result. More specifically, the exception operator states that if its left operand is true, then the function will have an exception outcome (*i.e.*, **exception** will be true), and if the function has an exception outcome and the right operand is true, then the left operand must be true. The intent is that the left operand defines the only program state that can cause the particular exception outcome defined by the right operand, without prohibiting another independent exception outcome.

This particular assertion states that if **amt** is less than 0 when **withdraw** is called, the function will have an

```
module bank {

    int errno;
    int NEG_AMT, INS_FUND;

    typedef int acct_no;

    int balance(acct_no acct);

    int deposit(acct_no acct, int amt)
        semantics {
            balance(acct) == @balance(acct) + amt,
            return == balance(acct)
        };

    int withdraw(acct_no acct, int amt)
        semantics {
            exception := (return == -1),
            normal := !exception,
            negative_amount := (errno == NEG_AMT),
            insufficient_funds := (errno == INS_FUND),
            @(amt < 0) <:> negative_amount,
            @(amt > balance(acct)) <:> insufficient_funds,
            exception --> unchanged(balance(acct)),
            normally (
                balance(acct) == @balance(acct) - amt,
                return == balance(acct)
            )
        };
};
```

Figure 2: The Bank Module in ADL

```
module bank {

    ...

auxiliary {
    int balance(acct_no acct);
}

    int deposit(acct_no acct, int amt)
        ...;

    int withdraw(acct_no acct, int amt)
        ...;
};
```

Figure 3: Auxiliary Definitions in ADL

exception outcome (the function will return the value −1). Also if the function has an exception outcome and **negative_amount** is true (**errno == NEG_AMT**), then **amt** had to be less than 0 when **withdraw** was called.

Similarly, the second assertion about **withdraw** states that if **amt** is greater than the balance of account **acct** when **withdraw** is called, then the function will have an exception outcome. Also, if the function has an exception outcome and **insufficient_funds** is true, then **amt** has to be greater than the balance of account **acct** when **withdraw** was called.

The third assertion about **withdraw** contains a predefined function called **unchanged**. This function returns true if its argument has the same value after the call as before the call. This assertion therefore states that if **withdraw** has an exception outcome, the balance of account **acct** will not change.

The fourth assertion about **withdraw** uses the predefined function **normally**. **normally** takes an arbitrary number of boolean parameters and returns true if all its parameters are true whenever **normal** is true. Therefore, on a normal return from the function being specified, all the parameters of **normally** must be true.

This particular assertion states that on a normal return from **withdraw** (*i.e.*, the function does not return −1), the balance in account **acct** is decremented by **amt**, and the function returns the new account balance.

The second example illustrates the use of auxiliary definitions. Suppose the bank module did not define the function **balance**. Then it would not be possible to write assertions for **deposit** and **withdraw** in the style of the previous example, since these assertions make use of the notion of **balance**. In this case, **balance** may be introduced as an auxiliary definition. Figure 3 shows the bank module with the function **balance** introduced as an auxiliary definition. The auxiliary definition must be bound to a C *test function* with the same parameter and result types as **balance** for testing to be possible.

3 Conclusions and Future Work

We are currently revising the ADL language to make its definition as a language framework more formal. When this phase is over, ADL will be defined as a set of high-level specification concepts such as the call-state operator and the error specification constructs, with well-defined schemes for specializations to programming languages.

We are also refining the language definition to incorporate better and more state of the art specification features while keeping in mind that ADL has to remain simple for use by the typical programmer.

One example of a refinement we are considering is in the partial specification of types. Currently, ADL requires a specification to fully define the types it uses — therefore, in the specification of a stack, the type of the stack data structure has to be completely defined. A straightforward solution to this problem is to implement opaque types, such as Ada's private types. We are also considering the use of partial type specifications through subtyping very similar to the schemes used in Rapide [1, 4].

Another area of ongoing research is in applying ADL technology to other forms of testing than unit testing. Some approaches being considered are randomized testing, sequence testing, and concurrent testing. Each of these methods will have its own test specification language similar to TDD.

Finally, research is currently underway to develop methods for translating ADL specifications into natural language documents.

Acknowledgements

All members of the PrimaVera group has contributed to the work described in this paper. We have also received valuable comments and suggestions from our funders at X/Open Co Ltd and IPA, Japan and from other people within Sun Microsystems Laboratories.

The original work on ADL was done by Sun Microsystems Laboratories, Inc. This has now been extended in collaboration with X/Open Co Ltd with funding from the Information-Technology Promotion Agency (IPA), Japan. All results are being made publicly available and open industry review is invited.

References

[1] Frank Belz and David C. Luckham. A new approach to prototyping Ada–based hardware/software systems. In *Proceedings of the ACM Tri-Ada Conference*, Baltimore, December 1990. ACM Press.

[2] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft, Inc. *The*

Common Object Request Broker: Architecture and Specification, omg document number 91.12.1 edition, December 1991. Revision 1.1.

[3] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[4] Dinesh Katiyar and Sriram Sankar. Completely bounded quantification is decidable. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 68–77, San Francisco, California, June 1992.

[5] B. Liskov and S. Zilles. Specification techniques for data abstraction. *IEEE Transactions on Software Engineering*, SE-1(1):7–19, March 1975.

[6] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA, A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[7] J. M. Spivey. *Understanding Z, A Specification Language and its Formal Semantics.* Cambridge Unversity Press, 1988. Tracts in Theorectical Computer Science, Volume 3.

[8] Sun Microsystems Inc., U.S.A., and Information-Technology Promotion Agency, Japan. *ADL Language Reference Manual*, document number MITI/0002/D/0.1 edition, August 1993.

[9] Sun Microsystems Inc., U.S.A., and Information-Technology Promotion Agency, Japan. *ADL Translator Design Specification*, document number MITI/0001/D/0.1 edition, August 1993.