Specifying and Testing Software Components using ADL

Sriram Sankar
Roger Hayes

SMLI TR-94-23 April 1994

Abstract:

This paper presents a novel approach to unit testing of software components. This approach uses the specification language ADL, that is particularly well-suited for testing, to formally document the intended behavior of software components. Another related language, TDD, is used to systematically describe the test-data on which the software components will be tested.

This paper gives a detailed overview of the ADL language, and a brief presentation of the TDD language. Some details of the actual test system are also presented, along with some significant results.

M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
sriram.sankar@eng.sun.com
roger.hayes@eng.sun.com

Specifying and Testing Software Components using ADL

Sriram Sankar *Roger Hayes*

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, California 94043

1 Introduction

At the start of the programming task, the programmer is supplied with a *specification* of the problem. The specification can range from being quite detailed, such as a document that describes the intended behavior of the program in all possible circumstances, to being quite informal, such as a prose description of what the program is supposed to do in a few situations.

In practice, the programmer has several sources of information available that comprise the specification. These may include a formal specification document, a working prototype, instances of program behavior, and *a priori* knowledge about similar software.

Working from this specification, the programmer develops the program. The *validation* of the program lies in the comparison of the program with the specification of intended behavior.

To automate the validation process, it is essential to formalize the specification of the program. For this purpose (among others), various specification languages and specification techniques have been designed—an overview of which can be found in [10]. Some examples of specification languages are Anna [11], Z [15], Larch [7], and VDM [2].

Our approach to program validation is through *testing*. In this approach, the program 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 that describes its behavior. Correct execution of the program on these test inputs increases the level of confidence in the program.

There are two parts to software testing. First, there is the problem of *test-data selection*. Since any test-data will necessarily be a very small sample of all the possible inputs, test-data should be selected in such a way that successful execution of a program on these test-data gives us a reasonable amount of confidence in the correctness of the program for all possible inputs. At the same time, the test-data should be such that redundancy in the testing process is minimized.

For this purpose, we have designed a high-level *test-data description* (TDD) language in which test-data for a program is described in a systematic manner.

Second, the program needs to be run on each set of test-data to determine if the program behaves as intended for this set of test-data. In most published work on testing, the details of this determination have not been dealt with. Rather, an *oracle* is assumed to exist. This oracle can judge for any specific set of test-data, whether or not the program behaves as intended. The idealization of the oracle has been essential for software testing. Lately, however, much work has been done to realize this oracle. In nearly all cases, this has been achieved by comparing the behavior of a program against its specifications that have been written in some formal specification language (e.g., see [5, 3, 14]).

We have designed a specification language called ADL (Assertion Definition Language). The design of ADL is well-suited for testing—generating a test oracle from an ADL specification is a straightforward process.

Pilot implementations for an ADL-based test generation system have been undertaken, and we have had significant results from these experiments. For example, we discovered an anomaly in an implementation of the `write` system call in the UNIX® operating system. The `write` function is specified to update 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. Since a write of 0 bytes does not change the file, either behavior is defensible, but the English specification is not clear on the point. 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.

Overview of this paper

This paper presents an overview of our testing methodology. The ADL and TDD languages are described, although more attention is paid to the ADL language. Other aspects (other than ADL) are covered in brief in this paper, but will be covered in detail in future papers.

The remainder of the introduction lists the high-level features of ADL and TDD, and then provides some background information on test-data selection and on earlier work of the PrimaVera group at Sun Microsystems Laboratories, Inc.

Section 2 describes the ADL language. Section 3 describes how *assertion-checking functions* (our oracle) are generated from ADL specifications, and provides a description of the TDD language. Finally, Section 4 concludes the paper with a discussion of ongoing work.

More information on the ADL testing methodology and environment is available in [16] and [17].

1.1 ADL

ADL is a language designed for formal specification of software components. It is well-suited for the purpose of testing. 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. This makes them suitable for specifying extant program libraries.
- 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.
- 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 TDD

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-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 and 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.

1.3 Background on Test-Data Generation

A large number of tools have been designed for test-data generation since the early 1970's. The emphasis has been to generate test-data that exercises as much of the program code as is practically possible. Some approaches have been to generate test-data that force every program statement to be executed (statement coverage), while others force every edge in the program's flowchart to be traversed (path coverage). A useful technique for test-data generation is symbolic execution of the program [3, 9]. Symbolic execution can be performed in a forward traversal or a backward traversal of the program paths. During these traversals, various constraints are established which are then used to generate the test-data. This falls under the general category of *white-box testing*. In white-box testing, the structure of the program is examined and test-data are derived from the program's logic. The other category is *black-box testing*. This is also known as *functional testing*. In this case, the internal structure and behavior of the program is not considered. The objective is solely to find out when the input-output behavior of the program does not agree with its specification. In this approach, test-data are constructed from the specification [8, 12].

Weyuker and Ostrand [18], building on the work of Goodenough and Gerhart [6], attempt to define a theoretically sound and practical definition of what constitutes an adequate test. The idea is to divide the test-data into a finite number of equivalence classes where testing on a representative of an equivalence class will, by induction, test the entire class. The equivalence classes are derived from both the program specification (called a problem partition of the input in [18]) and an examination of the program structure (called a path domain partition). TDD is very nearly an implementation of the concept of revealing subdomains from this work.

1.4 The PrimaVera Project

The PrimaVera group at Sun Microsystems Laboratories, Inc. has been working on applying formal specification techniques to software testing for eight years. Our emphasis is on good engineering solutions, minimizing the cost of adoption and training. We strive for systems that combine the benefits of formal methods with a low entry cost, and systems that allow incremental adoption with an early payback.

After several years of internal development and deployment, it was 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 Company Limited, 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).

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 particular 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., IDL [4], Rapide [1], and Larch [7].

The ADL language framework has been specialized for use with the C programming language, and with IDL—Object Management Group's interface definition language for the CORBA architecture specification. 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.

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.

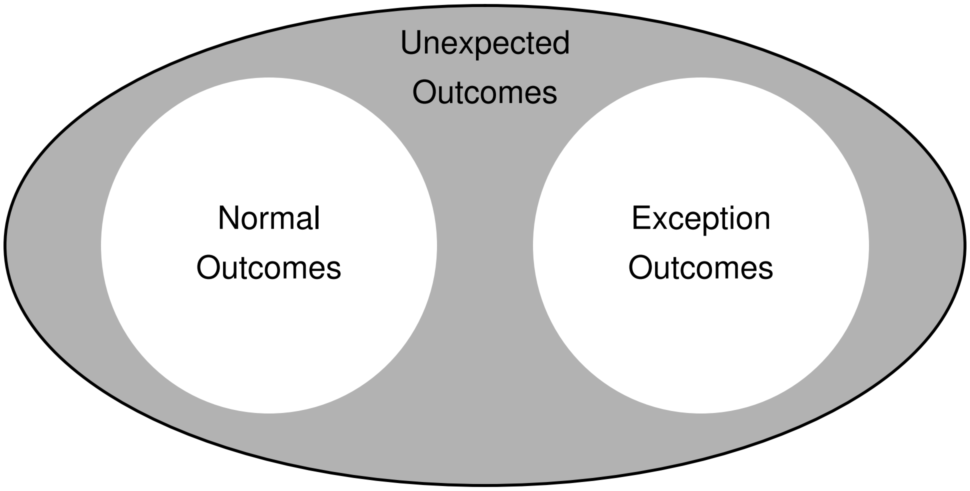

Figure 1. The Possible Outcomes of an Operation Evaluation

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* (or abnormal) 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 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*. 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. It's syntax is identical to the C `typedef` statement.

- *Object constituent*
  An object constituent introduces an object[1] and associates it with a type. It's 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. It's 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 it's 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 an 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 assertions 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).

---

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

- *Implication operators*
  ADL provides the standard logical implication and equivalence operators. It also provides an *exception operator* (`<:>`) that characterizes error situations by listing the conditions that cause the function to fail, 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 an 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`. The parameter is qualified to be of mode `in`, which indicates that only the input value of the parameter is relevant. `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

```
module bank {

   int errno;
   int NEG_AMT, INS_FUND;

   typedef int acct_no;

   int balance(in acct_no acct);

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

   int withdraw(in acct_no acct, in 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

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 fail 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 fail (i.e., `exception` will be true), and if the function fails 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 defined by the right operand, without prohibiting another independent exception.

This particular assertion states that if `amt` is less than 0 when `withdraw` is called, the function will fail (the function will return the value -1). Also, if the function fails 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 fail. Also, if the function fails 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` fails, 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.

```
module bank {

   . . .

auxiliary {
   int balance(in acct_no acct);
}

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

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

Figure 3. Auxiliary Definitions in ADL

3 Software Testing using ADL

ADL may be used in the testing of software in a variety of ways. In this section, we describe how ADL is used for the unit testing of C functions. A brief summary of other ways of testing being considered is presented in Section 4.

The following components are required for unit testing:

1. A function to be tested.
2. Test-data on which to execute the function. In this case, test-data is specified through the TDD file. This is described in Section 3.2.
3. A means of determining whether or not the function executed correctly. In this case, *assertion-checking functions* generated from the ADL specifications handle this task. This is described in Section 3.1.

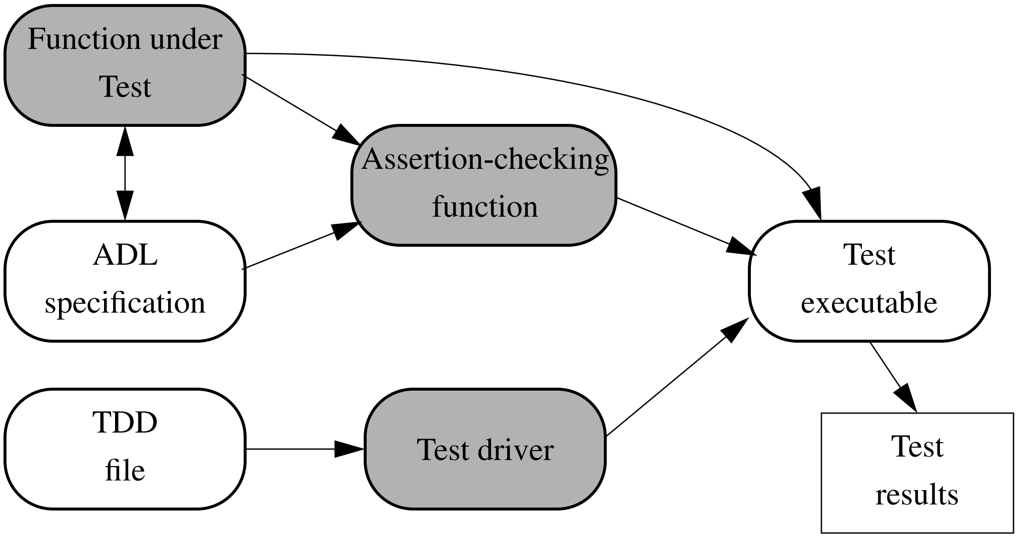

Figure 4. The Steps Involved in Unit Testing

Figure 4 illustrates the steps required to perform unit testing. The three components required for unit testing are shown on the left column. The double-edged arrow between the function under test and the ADL specification illustrates the binding between these two entities. From these two entities is constructed the assertion-checking function. Also, from the third component, the TDD file, is constructed the test driver that calls the function under test with different sets of test-data. The three shaded boxes are then linked and loaded together to obtain the test executable. When the text executable is run, it produces the test results that describe how the function performed on the given test-data.

The next two parts of this section present an overview of assertion-checking functions and TDD files.

3.1 Assertion-Checking Functions

An assertion-checking function is a wrapper around the function under test which, in addition to calling the function under test, also evaluates its ADL assertions to determine whether or not the function is behaving correctly. The assertion-checking function works roughly as follows:

1. Evaluate all sub-expressions qualified by the call-state operator, and save these results.
2. Call the function under test with the same parameters that were passed to it.
3. Evaluate all assertions after replacing the sub-expressions qualified by the call-state operator with their corresponding values as evaluated in the first step.
4. If any of the assertions evaluate to false, then report it as an error.

Advantages

The assertion-checking function isolates into a single function the task of checking that the function meets its specifications. Being a wrapper, assertion-checking functions can be used with any input. It could be used with exhaustive input which would not be feasible in any system that required human attention to each test case.

The assertion-checking function also isolates the task of checking from the preparation of test input and from function implementation.

Auxiliary definitions

If assertions refer to auxiliary code (such as functions), the assertion-checking functions will evaluate this auxiliary code. Hence, the implementations of auxiliary definitions have to be trusted. Care must be taken in choosing implementations of auxiliary definitions, for if they fail, they can jeopardize the testing process too.

3.2 TDD Files

The TDD file offers the test designer a formal and structured framework for describing test-data. The TDD file provides a structure through which the type of data used to test the function is characterized and documented. Through the use of the TDD syntax, test-data becomes the subject of a design process.

The testing process has both mechanical and creative aspects. Writing iterative loops to supply test-data to a function is rather mechanical work. However, deciding which test-data to supply requires some ingenuity. The possible universe of test-data for any one function parameter may be very large. The problem is breaking down that possibly very large universe into a finite set of test cases.

There are many analysis tools available for examining test results and relating them to the code under test, performing coverage analysis and error analysis of various sorts. TDD is not a tool in that category; instead, it is a design tool for test-data. It gives test designers a language for recording their insights in a systematic and easily communicated way. The design can then be used to help generate test programs.

The mathematical foundation for the language is the idea of dividing the possible test-data into equivalence classes and then selecting a representative for each class.

Equivalence and representatives

The basis for selecting a subset of all possible inputs is the notion of equivalence. Two test inputs are equivalent, with respect to a particular correctness check for a particular function under test, if they produce the same test result. The two inputs are equivalent if it is not possible to tell them apart by running the test.

Note that this definition of equivalence depends on the measure of correctness as well as on the function under test.

If a group of test inputs is equivalent, it is not necessary to run the test on more than one of the inputs; any one can be chosen. No additional information can be gathered by running the test on another input in the same equivalence class.

The art of the test designer is to define an equivalence partition; it requires insight into the implementation and into possible errors, as well as into the typical uses of the function under test and the structure of the data. There are tools to assist in the process of analysis. The TDD language is a tool to help record the results of the analysis, not to perform the analysis. In the terminology of [18], TDD is a tool to specify the problem partition.

The result of the analysis is a set of test cases that provide the desired coverage of the function under test. Mathematically, these are a set of representatives of the equivalence classes in the test input space. Operationally, they are the values on which to test the function.

Abstraction of test input

The TDD file describes the test-data symbolically. It is not simply a list of test inputs, but a set of characteristics of the test inputs. These characteristics are called *properties*. These symbolic properties serve as documentation of the test-data; they describe the intent of the data. In addition, by separating the characterization of the data from the actual data, the test specification can survive changes in the system environment.

Formally, a property is applied to a data type and notates a partition of that data type. Each datum in the data type is described by exactly one of the choices for the property. A data type is typically described by the cross-product of several properties. Each element in the data type is described by a tuple of choices from the cross-product of properties. The data type is hence partitioned into classes, each of which is an intersection of the classes determined by the individual properties of the data type.

Example

Consider testing a function that appends to the end of a file a transaction with a particular size and kind. In the TDD file, one would not list a set of numbers to use as the number of bytes, but instead describe the numbers symbolically:

```
prop trans_size =
   {negative, zero, small, large, huge}
```

Thus, the size of a transaction is characterized by one of the properties: `negative`, `zero`, `small`, `large`, or `huge`. Similarly, the kind of transaction is described symbolically:

```
prop trans_kind =
   {read, write, create, delete}
```

Anyone reading the TDD file has, at a glance, an idea of the tests that will be run on the function.

As a consequence, as the implementation is modified over time, the design of the test-data can survive. In testing a file system, the specific values for the size of a transaction will typically depend on the block size of the file system implementation. By using an abstract definition, the same test design can be used for any block-structured file system.

This symbolic description of the input, as well as being independent of the details of the data structures described, is also independent of the particular implementation of the function under test. While the TDD file should be written with insight into the possibilities of an implementation, it specifies a black box test. There is no direct input in terms of code coverage, for example, although feedback from such measurements can certainly be used to improve the quality of the input descriptions when developing and running tests.

Provide functions

The TDD file gives a symbolic description of the desired test input. That symbolic description is converted into actual data values during the execution of a test, by *provide functions*. These are functions written by the test engineer and linked with the generated test program; they encapsulate the task of generating test values. All too often, the complexity of the mechanism that produces test-data hides the intent of the data. Provide functions, in contrast, have a well-defined interface and perform a clearly defined task: to generate one data value with a specific set of characteristics described by a set of properties.

Isolating the data-generation code into the provide functions is an important benefit. The often-messy details of generating data values are encapsulated in the provide function; the property arguments to the provide function express the intent of the test variable. Provide functions are modular, reusable, and have a clearly defined task; the technique would be useful even without automatic test generation.

Relation to ADL and the test program

The purpose of the TDD language is to describe data values that will be used in tests. Those values are data in the host programming language. Values of test variables are described symbolically; the test designer writes a provide function to translate those symbolic descriptions to programming language data during the execution of the test. The ADL type of the test variable determines the signature of the provide function.

A test directive in the TDD file is translated into a test driver program; that test driver makes calls to the provide functions and to the assertion-checking functions.

A TDD file is written with respect to a fixed ADL file; the type names and function names that can be used in the TDD file are those declared in the ADL file. There can be more than one TDD file for a particular ADL file.

4 Conclusions and Future Work

In this paper, we presented an overview of a capability to test software components in a systematic and non-intrusive manner. We have not yet published any details of our experience in implementing and using a test generation system implementing these capabilities. We are currently constructing an implementation targeted for an ANSI C environment, which will be made widely available.

There is also scope to extend our capability in many ways. Keeping the ADL specifications and assertion-checking functions, we can replace the TDD file with many other test-data generating schemes. A straightforward approach is the generation of random data; another approach is the generation of large volumes of (possibly similar) data for stress-testing.

We are also considering more involved extensions of our testing capability. One extension is to test the behavior of combinations of operations (as opposed to unit testing). As a motivating example, there is very little that unit testing can do for a stack in this case; the push and pop operations have to be run in various combinations for comprehensive testing. Another extension is to test the behavior of an operation when multiple processes attempt to access it simultaneously. This happens frequently with system calls.

Research is currently underway to develop methods for translating ADL specifications into natural language documents. This will be the subject of a future paper.

Finally, we intend to apply the ADL specification methods to other languages such as C++ and Ada.

5 Acknowledgements

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

6 References

[1] Belz, Frank and Luckham, David C. “A New Approach to Prototyping Ada-based Hardware/Software Systems.” *Proceedings of the ACM Tri-Ada Conference* (December 1990).

[2] Bjorner, D. “VDM '87—A Formal Method at Work.” Vol. 252 of *Lecture Notes in Computer Science*. New York: Springer-Verlag, 1987.

[3] Boyer, R. S., B. Elspas, and K. N. Levitt. “SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution.” *Proceedings of the International Conference on Reliable Software* (April 1975): 234–245.

[4] Digital Equipment Corporation, Hewlett Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft, Inc. *The Common Object Request Broker: Architecture and Specification*. revision 1.1. OMG document number 91.12.1 edition. December 1991.

[5] Gannon, J., P. McMullin, and R. Hamlet. "Data-abstraction Implementation, Specification, and Testing." *ACM Transactions on Programming Languages and Systems* 3, no. 3 (July 1981): 211–223.

[6] Goodenough, J. B. and S. L. Gerhart. "Towards a Theory of Test-data Selection." *Proceedings of the International Conference on Reliable Software* (April 1975): 493–510.

[7] Guttag, J. V., J. J. Horning, and J. M. Wing. "The Larch family of Specification Languages." *IEEE Software* 2, no. 5 (September 1985): 24–36.

[8] Infotech International. *Infotech State of the Art Report. Software Testing Volume 1: Analysis and Bibliography.* 1979.

[9] King, J. C. "A New Approach to Program Testing." *Proceedings of the International Conference on Reliable Software* (April 1975): 228–233.

[10] Liskov, B. and S. Zilles. "Specification Techniques for Data Abstraction." *IEEE Transactions on Software Engineering* SE-1, no. 1 (March 1975): 7–19.

[11] Luckham, David C., Friedrich W. von Henke, Bernd Krieg-Bruckner, and Olaf Owe. "ANNA, A Language for Annotating Ada Programs." Vol. 260 of *Lecture Notes in Computer Science*. New York: Springer-Verlag, 1987.

[12] Meyers, G. J. *The Art of Software Testing*. New York: John Wiley & Sons, 1979.

[13] Richardson, D. J., S. L. Aha, and T. O. O'Malley. "Specification-based Test Oracles for Reactive Systems." *Proceedings of the Fourteenth International Conference on Software Engineering* (May 1992).

[14] Sankar, S. "Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs." Ph.D. thesis, Stanford University, 1989. Also Stanford University Department of Computer Science Technical Report No. STAN-CS-89-1282 and Computer Systems Laboratory Technical Report No. CSL-TR-89-391.

[15] Spivey, J. M. "Understanding Z, A Specification Language and its Formal Semantics." *Tracts in Theoretical Computer Science*, vol. 3. Cambridge University Press, 1988.

[16] Sun Microsystems, Inc., U. S. A. and Information-Technology Promotion Agency, Japan. *ADL Language Reference Manual*. document no. MITI/0002/D/0.1 edition. August 1993.

[17] Sun Microsystems, Inc., U. S. A. and Information-Technology Promotion Agency, Japan. *ADL Translator Design Specification*. document no. MITI/0001/D/0.1 edition. August 1993.

[18] Weyuker, Elaine J. and Thomas J. Ostrand. "Theories of Program Testing and the Application of Revealing Subdomains." *IEEE Transactions on Software Engineering* 6, no. 3 (May 1980): 236–246.

About the authors

Sriram Sankar is currently a member of the PrimaVera group at Sun Microsystems Laboratories, Inc. Previously, he was a research staff member at the Computer Systems Laboratory at Stanford University. His current research interests include programming methodologies, programming and specification languages, and the application of formal reasoning in software development and validation. His earlier work includes the development of compilers and environments for Ada and Anna. He received a BTech in Computer Science at the Indian Institute of Technology, Kanpur, and MS and Ph.D. degrees at Stanford University, California.

Roger Hayes received the BS degree in computer science from Portland State University in 1982, the MS degree in computer science from the University of Arizona in 1986, and the Ph.D. degree in computer science from the University of Arizona in 1989. Since 1989, he has worked at Sun Microsystems, Inc. in platform software and automated testing methods. He is currently a member of the PrimaVera group at Sun Microsystems Laboratories, Inc., Mountain View, California.