Assertion Defintion Language Translation System

an Automated Test Generation Tool

Matt Evans (matt.evans@sun.com)

ADL Project Lead, Sun Microsystems Laboratories

Shane P. Mc Carron (s.mccarron@xopen.co.uk)

Testing Research Manager, X/Open Company Ltd.

1. Introduction

The Assertion Definition Language (ADL) Project is a four year effort on the part of a variety of organizations to develop a specification language that uses first order logic to describe the behavior of virtually any interface. The overriding goal of this effort is to develop a set of tools and a language specification that can be made freely available to the industry as a basis for future research in this area, but that is in itself complete and useful.

1.1 Project Overview

In 1992, the Information-technology Promotion Agency, Japan (IPA) a governmental agency under the Japanese Ministry of International Trade and Industry, entered into an agreement with X/Open Company Ltd., the open system community's primary specification and certification organization, for research into Automated Test Generation via an Assertion Definition Language. The purpose of this project was to be the research into and development of an Assertion Definition Language and ADL Translation System, with the intent of making the language and associated tools freely available under an X Window System-like copyright and grant of rights. These tools would further be fully internationalized, and localized versions of them would be made available as time and funding permitted.

After some requirements development and a review of relevant extant research, X/Open selected Sun Microsystems Laboratories (SML) and their PrimaVera project to be the basis of this research. Throughout 1993, SML and X/Open worked with many representatives of the open systems testing community to define the language, design the architecture of the translation system, and specify quality standards for the project. Drafts of all the project's specifications were made available to the community via a free ftp accessible repository on the Internet, and many public reviews were conducted via electronic mail and through project review meetings.

The research and development of the ADL Translation System was conducted by SML's Primavera group. The Primavera group has been working on applying formal testing techniques to software testing for the past eight years. The team members of the SML Primavera group include:

Alberto Savoia	-	Principal Investigator
Matt Evans	-	ADL Project Lead
Sriram Sankar	-	Project Researcher
Roger Hayes	-	Project Researcher
Roongko Doong	-	Project Researcher
Bill Gogesch	-	Project Researcher
Jos Marlowe	-	Project Researcher

Early in 1994, SML placed on the Uunet server the first early Alpha version of the working Translation System. Shortly thereafter, X/Open started related projects to test the utility of ADL using qualitative and quantitative techniques. Those related projects, as well as the basic ADL research and development project, are ongoing.

While the currently available version of the System is operational, there are many extensions and enhancements to be made before it is complete. Throughout 1994 X/Open and SML will work with the industry to perform Beta testing of the ADL Translation System via structured and unstructured projects. X/Open and SML currently plan to deliver a complete ADL Translation System, including localized system text, by the end of 1995.

1.2 Technical Overview

The Assertion Definition Language, as defined through its language reference manual, is a formal language that can be used to describe the behavior of programmatic interfaces. ADL consists of a small set of concepts designed to be portable to most programming languages. In its first

incarnation, ADL is used to describe interfaces written in the C Language (or interfaces that can be readily exercised via a C Language routine). The structure of the language allows the interface designer to describe function signatures and basic function behavior in a syntax that the ADL Translation System compiles into ANSI 'C' language Assertion Checking Functions and supporting routines. By linking the emitted code, some user supplied supporting routines, and the interface implementation, the resulting program can thoroughly test the conformance of the interface implementation against its ADL specification.

The ADL primitives are further extended by the definition of two supporting languages: The Test Data Description language, used by test designers for defining test data sets; and the Natural Language Dictionary, used by documentation authors to augment the ADL Translation System's inherent ability to translate ADL-based specifications into natural language documentation such as English or Japanese.

When used together, this collection of specification languages allows an interface designed using ADL to have its ADL-based specification remain useful throughout the life of the product in which the interface resides. Further, the ongoing maintenance of the ADL-based specification allows unit and system tests to be automatically revised as interface behavior changes. Finally, the extremely flexible nature of ADL allows low level support routines and behavioral descriptions to be reused both within a single interface specification and among any number of specifications.

The flexibility of the ADL Translation System, coupled with the ability to readily separate the three main aspects of development (design, documentation, and testing), makes ADL an ideal choice for developers just beginning projects. However, ADL is also useful for testing interfaces that have long been designed and implemented. The boolean nature of the ADL assertion primitives make it very easy to take a prose description of an interface and translate it into an ADL-based specification. This description can then be used not only to generate tests, but also to ensure that the original prose description is accurate and sufficient. In addition, the automatically generated test specification, derived from the ADL-based specification and the test data description, can be used to assure that adequate aspects of the interface's behavior are being exercised during testing. Should additional aspects need to be exercised, the test data description can be easily extended.

The ADL Translation System represents a significant weapon in the testing arsenal of any test development group. While ADL may not be a complete testing solution for every situation, it can be a positive boon in quality assurance during and after the development process.

1.3 Paper Overview

This paper describes the ADL project from a relatively high level. The first half of the paper is devoted to describing how ADL works and how the various components of the ADL Translation System work together to describe the behavior and test parameters of an interface. It then describes the outputs of the ADL Translator, and shows how these outputs can be used to do unit and system testing. Finally, this paper describes the ways in which the ADL Project is attempting to validate the ADL Translation System through the development of some prototype test suites, and describes how the reader can obtain early versions of the documentation and translation system so that they can participate in the validation process as well.

2. ADL Translation System Architecture

The first implementation of the ADL Translation System is designed to generate tests for any open system platform compliant with X/Open's Portability Guide version 4 (XPG4) — most modern UNIX-like systems fit this description. The System itself is written in C++ and ANSI C and will build and execute on an XPG4 compliant platform with a resident C++ compiler. The System builds well using the Free Software Foundation's gcc/g++ compiler. The documentation for the System was developed using Frame Technology Corporation's FrameMaker version 3.0. It is provided in both source and PostScript™ forms in the distribution. In addition to these requirements, a complete ADL Translation System platform may also have the Test Environment Toolkit installed as a test controller (see below).

The current ADL System distribution archive (release version 0.5) contains separate releases for both the ADL system software and the ADL documentation set. A Complete build of the ADL System requires approximately 30 megabytes of free disk space. The System builds successfully on a Sun SparcStation/Solaris 2.3 platform using gcc version 2.60. The resulting executable is approximately 4 megabytes stripped.

2.1 Architectural Model

The ADL Translation System architecture is made up of ten basic components. The bulk of this document describes each of these components in detail. This section provides an overview of how the various components relate to one another from a very high level. The components of the ADL Translation System are best illustrated through the following diagram:

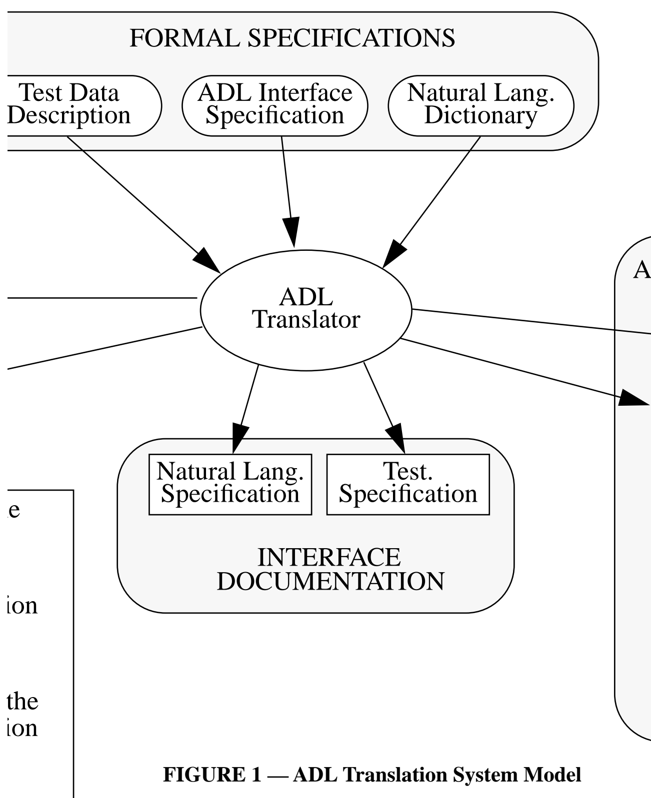

FIGURE 1 — ADL Translation System Model

Formal Specifications

Inputs to the ADLT are in the form of formal specification files. The ADL Translation System defines three specification languages; Assertion Definition Language (ADL), Test Data Description (TDD) and Natural Language Dictionary (NLD).

ADL Interface Specification

The ADL Interface specification is the file (or files) in which the signature and semantics of the interfaces are defined.

Test Data Description Specification

The Test Data Description (TDD) specification is a file (or files) that define the manner in which the function parameters described in an ADL Interface Specification are created during execution of the generated test case, and the order in which these tests are executed.

Natural Language Dictionary

The Natural Language Dictionary (NLD) is a user-provided reference file (or files) that define the way in which ADL Interface Specification elements are to be translated into a natural language (such as English). This

file is optional. If it is not provided, the ADL Translator will perform a rudimentary default translation.

Interface Documentation

One of the features of the ADL Translation System is the automated generation of interface documentation. The generated documentation is a natural language translation of the information in the ADL Interface Specification and the TDD specification.

Test Specification

A Test Specification is an ADL Translator-generated file that describes each function under test, the assertions for those functions, and the ways in which each function will be exercised during a test case run. This file is generated in a natural language (such as English) based upon information in the ADL Interface Specification, the Test Data Description, and the Natural Language Dictionary.

Natural Language Specification

A Natural Language Specification (NLS) is an ADL Translator-generated file that describes the syntax and semantics of the functions described in the ADL Interface Specification, based upon information in the ADL Interface Specification and the Natural Language Dictionary.

ADL Test Program

The ADL test program comprises the source code for an executable test. The ADL Translator generates appropriate source code based upon the ADL Interface Specification and the test directives found in the TDD. When run through an appropriate language compiler (e.g. an ANSI C compiler) and combined with the user created auxiliary and provide functions, the resulting ADL Test Program can be executed to test each function under test.

Test Driver

The ADL Translator generates source code for the Test Driver based on information translated from the TDD specification. The Test Driver is responsible for iterating over the requested list of test cases and managing the test execution.

Assertion Checking Function

The Assertion Checking Function is emitted source code that invokes the interface implementation and checks the results of invocation against the semantic assertions specified in the ADL specification.

Auxiliary Functions

Auxiliary functions are user-provided functions that assist the ADL-generated Assertion Checking Function in determining the veracity of assertions in the ADL Interface Specification. Auxiliary functions are referenced by using them in assertion statements.

Provide Functions

The provide functions are user-written functions that generate appropriate parameter values given the parameter properties defined in the TDD.

Interface Implementation

The Interface Implementation is the system or function being specified and tested by the ADL Program.

Test Suite Management

The ADL Translator can generate two files to aid in building and executing ADL Test Programs; a TET scenario file and a Makefile.

TET Scenarios

A TET Scenario is a file (or files) that describes the sequence of tests to be built, executed, and/or cleaned up by TET's Test Case Controller. The ADL Translator generates a TET Scenario file that describes a suite of tests comprised of all test directives in the TDD, as well as individual entries for each function under test in either summary or detailed testing mode (see below).

Makefile

The ADL Translator generates a Makefile to build the ADL Test Program. Once the Makefile is generated, the user can use the make facility to re-generate all outputs of the ADL Translation System. New versions of the ADL Test Program are easily compiled and linked with the generated Makefile.

2.2 ADL and the Test Environment Toolkit

The Test Environment Toolkit, or TET, is an open systems industry standard testing harness designed in 1989 by the Open Software Foundation, UNIX International, and X/Open. This free-ware harness provides a standard user interface for test users, and APIs in a number of programming languages for test developers. Through TET, the designers hoped to unify the open systems testing industry with the goal of allowing plug-and-play interoperability of test suites and test users. Within that small community, this goal has been largely achieved.

When X/Open first started the ADL Project, the intent was to develop a tool that would work in conjunction with TET to automate the generation of TET-based test suites. After considerable discussion with the team at Sun Microsystems Laboratories, the design goals of the project were changed to allow this interaction with TET, but not to require it. When ADL generated tests are used with TET, the interaction between TET and ADL generated tests suites is seamless. The ADL Translation System has the ability to generate complete test suites that can be built, executed, and cleaned-up by the TET test case controller.

In order to achieve this seamless integration, the System generates a TET scenario file for the generated test cases. A scenario file describes the sequence of tests to be executed, and may have subsets of tests defined within it. ADL generates a scenario file that describes a complete suite of tests for each test directive in the TDD.

Specifically, the ADL Translation System allows the generation of test cases that use the low-level TET primitives for reporting results and interim data. In addition, the System provides two modes of TET-style result reporting that are aligned with IEEE Std. 1003.3-1991 (the POSIX standard for assertion-based test methods). These modes allow the test case to record the results of assertion evaluation in a manner consistent with the IEEE “POSIX” standards of reporting a single result per assertion. This can be done either in summary mode (where the union of all evaluations of an assertion are considered the result for that assertion), or in detail mode (where each assertion evaluation reports a result).

Note that the ADL Translation System does not require the use of TET. Instead, it permits test engineers who are designing complete suites of interface tests to use this harness to simplify and automate the building, execution, and cleaning-up of test runs. This is particularly useful in system test environments, where a complete implementation of a set of interfaces must be tested prior to production release. It is also useful for conformance testing, where certifiable passing of a test suite is necessary to ensure conformance to industry standards.

3. Assertion Definition Language

The Assertion Definition Language was designed primarily for two purposes; to allow the easy and formal specification of the behavior of software interfaces and to facilitate the automated generation of conformance tests. ADL can be described as a meta-language, where a succinct list of software behavior concepts are represented by high-level syntax. The ultimate goal is to design specialized versions of the ADL meta-language syntax for the more popular programming languages in use today.

In the current implementation of the ADL Translation System, ADL has been specialized for specifying and testing software written in the ANSI C language. In this implementation, the structure of a C language interface is described using ANSI C type declarations. Interface behavior is specified using a combination of ADL specific constructs and ANSI C expression syntax.

ADL specifications describe the interface from a client as well as a testing point of view. The declaration of the interface provides all the information necessary to compile a client program that makes calls to the operations defined in the interface. However, this is only part of the information needed to make correct use of the interface. Knowing precisely what the operations of the interface will do under both normal and exceptional conditions is key to developing and ensuring error free programs that use the specified interface. A complete ADL specification provides an unambiguous formal description of the semantic behavior of the interface operations, providing enough information to develop a correct client program.

The following points describe some key features of the ADL language:

- ADL specifications describe operation (or function) behavior as a list of boolean expressions called *assertions*, where each individual assertion must evaluate to true upon termination of the operation. Thus, ADL specifications are post-conditional constraints on the program state after the operation is invoked.
- ADL provides language constructs to declare how an operation indicates to a client that it encountered an exceptional condition. Conversely, the language allows you to specify what the function should do under normal circumstances. An ADL assertion statement can be based on whether the function has indicated either a normal outcome, an exceptional outcome, or both.
- ADL has been designed to be translated into various forms that preserve the precise meaning of the original specification. An ADL specification can be translated into natural language documentation (such as a UNIX style “man page”) that reflects the intentions of the specification author as accurately as the target language will allow.
- ADL Assertions describing operation behavior can be easily translated into a client based testing oracle. In the ADL Translation System the generated test oracle, called the assertion-

checking function, invokes the function under test with paramertizable inputs and evaluates the translated ADL assertion expressions one-by-one to ensure each expression evaluates to true.

The following simple example illustrates the ADL specification concepts. This bank account example is used throughout this paper and the ADL Translation System documentation. It is an oversimplified description of a mythical banking software interface. It is useful for describing ADL, since banking operations are familiar to most people. The bank account ADL specification is presented from the outside in. Concepts are introduced as they appear. ADL keywords appear as bold text; all non-bold text appearing in the examples is legal ANSI C syntax.

```
module bank {
  ...
};
```

EXAMPLE 1 — ADL Bank Specification

The **module** keyword declares the name of the interface. It is an abstract encapsulation that contains the constituents that make up the interface. Module constituents can include declared data types, function declarations and functional semantic descriptions. Although not shown in this example, modules can import other modules. This has the effect of making the imported module constituents referable and programmatically visible in the base module. In this way ADL specifications can be aligned according to the component topology of the software system being specified.

```
module bank {
  const int TRANSACTION_OK=0;
  const int INSUFFICIENT_FUNDS =1;
  const int NEGATIVE_AMOUNT =2;
  int bank_errno;
  typedef int account;
  int withdraw (account acct, int amount);
  int deposit (account acct, int amount);
};
```

EXAMPLE 2 — ADL Bank Specification

The above additions to the bank example have not added any ADL specific syntax. ANSI C syntax is used to declare the constituents of the interface. In this case, the most important constituents are the function declarations of `withdraw` and `deposit`.In the following example the behavior of the withdraw function is specified.

```
module bank {
  const int TRANSACTION_OK=0;
  const int INSUFFICIENT_FUNDS =1;
  const int NEGATIVE_AMOUNT =2;
  int bank_errno;
  typedef int account;
  auxiliary {
     int balance (account acct);
  }
  int withdraw (account acct, int amount);
     semantics {
        exception := (return = -1),
        normal    := !exception;
     };
  int deposit (account acct, int amount);
};
```

EXAMPLE 3 — ADL Bank Specification

In Example 3 several ADL constructs are added. The specification now contains a curly-braced section denoted by the keyword **auxiliary.**The **auxiliary** section contains declared module constituents that are not visible to the client of the interface. They exists purely as an aid in augmenting the assertion statements that describe the operational behavior of the client visible functions defined in the specification. In terms of the generated tests, **auxiliary** constituents, especially **auxiliary** functions, are an important feature in creating complete and accurate conformance tests. **Auxiliary** functions are implemented or referenced by the test suite author. Their purpose is to provide program state information not normally accessible via the interface client. In the bank example, the declared **auxiliary** function `balance` will be called by the generated test oracle whenever `balance` is referenced in an assertion statement.

Additionally, The ADL specification in Example 3 starts to annotate the function `withdraw` with semantic constraints. In ADL, assertion statements are enclosed in a syntactical scope indicated by the keyword **semantics**. Statements in the **semantics** scope can be of two forms:

- Bindings
  Bindings are an association between a name and an expression. These names may be subsequently used as
  shorthand forms of the expressions they represent. Bindings take the form of `name := expression`**,** where '`:=`' is the ADL bind operator.
- Assertions
  As mentioned earlier, assertion statements are boolean expressions that must hold true when control has returned from the function. Assertion

statements combine ANSI C expression syntax and ADL logic operators and predefined functions.

In the **semantics** section of the withdraw function there are two special bindings labeled **normal** and **exception**. These two bindings describe how the function indicates to a client whether the function completed successfully or not. In this case the return value of withdraw (indicated by the ADL keyword **return**) is equal to -1 indicates that withdraw has encountered an exceptional or error conditional during its invocation. Correspondingly, a return value not equal to -1 is an indication that withdraw completed normally.

In Example 4 the functional behavior of the `withdraw` function has been fully specified. The specification consists of six assertion statements that describe how the `withdraw` function is supposed to behave given any input condition. The assertions also describe conditions of the program state after the function returns. The assertion statements are a mixture of ANSI C expression syntax and ADL syntax. The first two assertions in the `withdraw` semantics section use the *exception operator* '**<:>**'. The exception operator is used to indicate the conditions that may cause an exceptional return to occur and when it does, what additional constraints must hold true. It is no accident that the withdraw function is similar to a UNIX system call. It should be evident that when a test author writes ADL specification for UNIX–like system calls, the exception operator is used to declare the conditions that cause a system call to fail and test that the call has set the

```
module bank {
  const int TRANSACTION_OK =0;
  const int INSUFFICIENT_FUNDS =1;
  const int NEGATIVE_AMOUNT = 2;
  int bank_errno;
  typedef int account;
auxiliary {
  int balance (account acct_num);
}
int withdraw (account acct, int amount);
  semantics {
      exception := (return = -1),
      normal    := !exception;
      @(amount < 0) <:> bank_errno ==
         NEGATIVE_AMOUNT,
      @(amount > balance (acct)) <:>
         bank_errno == INSUFFICENT_FUNDS,
      exception -->
         unchanged (balance(acct)),
      normally {
         bank_errno == TRANSACTION_OK,
         balance(acct) ==
            @balance(acct) - amount,
         return == balance(acct)
      }
  };
  int deposit (account acct_num,
      int amount);
};
```

EXAMPLE 4 — Complete ADL Bank Specification

value of `errno` to the correct system defined error number. The ADL Translation System makes a special provision for the identifier `errno`. If `errno` is referred to in a ADL specification, the resulting generated tests will ensure that the value of `errno` is cached immediately after the return from the function under test.

The third assertion states that when the function indicates an exceptional return, it will not have altered the balance of the account. This is accomplished by using the ADL predefined function **unchanged**. The last three assertions have been enclosed in a *normally function*. A normally function takes an unlimited list of assertion statements and returns the collective result of evaluating each of the assertion statements in the list. Normally will only evaluate the expressions in the assertion list when the normal binding evaluates to true and the exception binding evaluates to false.

Below is a table defining the assertion related ADL specific operators, predefined functions, and key words:

ADL Assertion Syntax	Description
Keywords	
normal	Normal return binding identifier
exception	Exceptional return binding identifier
return	The value returned by the function upon completion.
Predefined Functions	
unchanged(P)	Returns the boolean result of whether the value of P before the function is called is equal to the value of P when the function completes. Equivalent to the assertion: @(P) == P
normally{assertion-list}	Evaluates each assertion in assertion-list when the function has indicated a normal completion. When an exceptional completion occurs, **normally** returns true without evaluating the assertions contained in the assertion-list.
forall (domain_spec) cond_expression **exists** (domain_spec) cond_expression	Indicates a quantified expression. A quantified expression allows specifying a range of situations over which a condition must hold true. A Quantified expression may be an *universally quantified expression*, indicated by the keyword **forall** or an *existentially quantified expression*, indicated by the keyword **exists**.
Operators	
name **:=** expression	The bind operator. Name is bound to expression.
@(P)	The call-state operator; equal to the value of P, before the function under test was invoked.
P **-->** Q	Logical implication; P implies Q.
P **<--** Q	Reverse logical implication; Q implies P.
P **<->** Q	Logical equivalence, P is equivalent to Q.
P **<:>** Q	Exception operator; declares when P is true, the function must have indicated an exceptional completion. Additionally it says that when both Q is true and the function indicates an exceptional return, then P must also be true. Equivalent to: P **--> exception** && Q && **exception** --> P
{assertion-list}	Indicates a group expression.The group expression construct allows the scoping of a list of assertion expressions. The result of a group expression is equal to the anded result of each individual assertion expression in the assertion-list. The **normally** clause is a special-case group expression.

4. Assertion Checking Functions

In any software testing scheme, there are two primary requirements; a means of invoking the system under test with a set of test-data and the ability to analyze the results of invocation against an expected outcome. In an ADL generated test program, the Assertion Checking Function (ACF) performs the task of invoking the function under test and analyzing whether the results of the function invocation correspond to the assertions made in the ADL specification. The Assertion Definition Language Translator (ADLT) generates the ACF by parsing the ADL interface specification. The ACF test results analysis algorithm is a direct translation of the semantic assertion statements in the originating ADL specification. The ACF is designed to do a single invocation of the function under test, evaluate each assertion check, and supply the results of the semantic evaluations to the ADL test report module. Figure 2 shows how the ACF fits into the overall architecture of the generated test program.

The ACF plays no part in the retrieval or generation of test data used to invoke the function under test. Instead, another ADLT generated component, the Test Driver, is responsible for calling the data generation functions and supplying the data to the ACF. A complete set of test-data that can be used in a single invocation of the function under test is called a test instance. In a complete execution of the ADL test program, the Test Driver cycles through all the requested test instances, repeatedly calling the ACF to invoke the function under test and report the results of the test outcome. Test data generation and specification are more fully explained in Section 5.2, "TDD and test data generation".

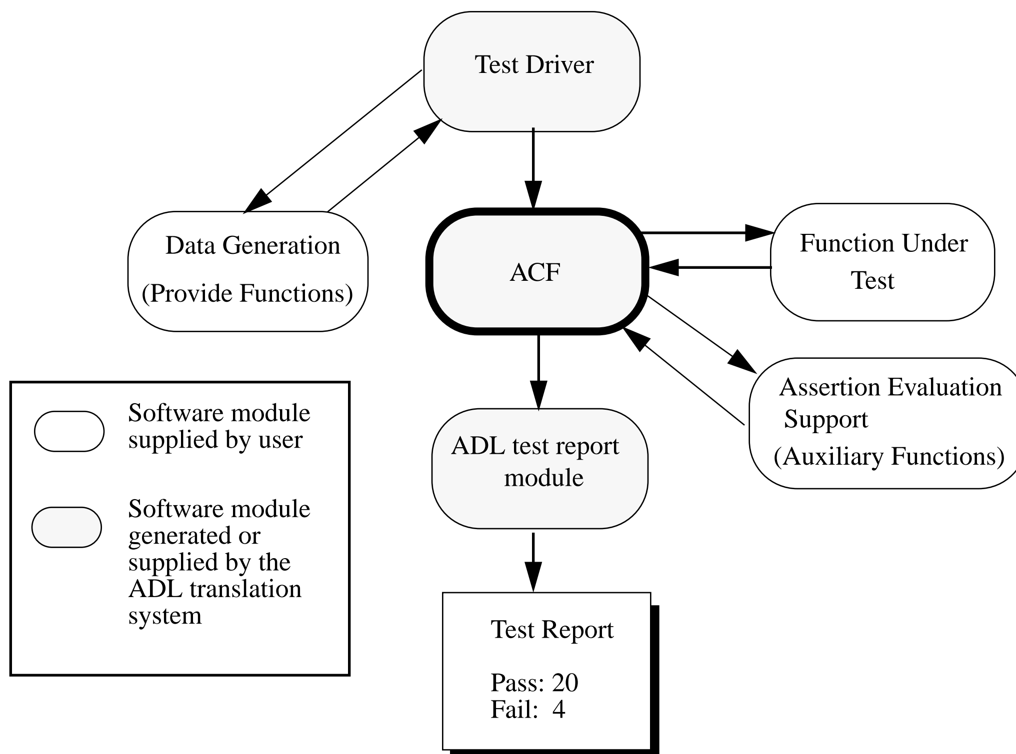

FIGURE 2 — The ACF and Test Program Architecture

The ACF can report the outcome of an individual test instance invocation in several ways. First is the traditional PASS and FAIL, where a PASS indicates that all of the semantic assertion checks held true, and a FAIL means one or more of the assertions evaluated to false. In ADL, the reported results of a test instance are extended to include the outcomes SKIPPED, ERROR, UNDEFINED and AMBIGUOUS. The result SKIPPED is not reported by the ACF at all; it originates from the Test Driver when a particular instance of test data could not be retrieved. A reported test instance of ERROR indicates that in the course of test instance execution an unexpected error resulted and prevented further processing. Finally, there are the novel test results UNDEFINED and AMBIGUOUS.

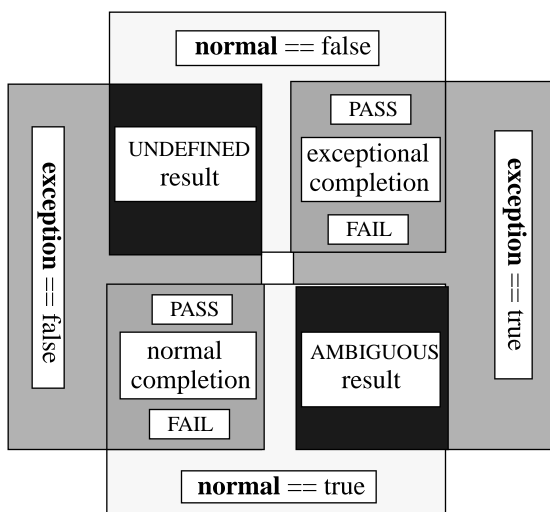

FIGURE 3 — Possible Reported Test Instance Results

Recall from Section 3, "Assertion Definition Language", that the completion of a specified function can be characterized by the bindings of **normal** or **exception**. **Normal** and **exception** are ADL reserved names bound to boolean expressions that indicate a successful or unsuccessful completion of the operation. Ideally, **normal** and **exception** should each evaluate to either true or false, clearly indicating a successful or unsuccessful completion of the function. However, certain situations may arise that will cause **normal** and **exception** to have equal values. In the two cases when they are equal, semantic evaluation of the assertions can not occur and a test instance result of PASS or FAIL would not make sense. When one of these two conditions arise, the ACF will report the more special-case results of AMBIGUOUS or UNDEFINED.

Figure 3, "Possible Reported Test Instance Results" gives a graphic depiction of the reported outcomes of a test instance. When both **normal** and **exception** are false, the test instance result is UNDEFINED. This is an indication that there is an unexpected value used by the function to indicate an error condition. Either the function is non-compliant or the evaluation range of **normal** and **exception** needs to be broader. When **normal** and **exception** are both true, the test result is AMBIGUOUS. Nearly always, this is a specification error and the expressions defining **normal** and **exception** should be examined closely.

The ACF reports the **normal** and **exception** evaluation separately from the evaluation of the specification assertion checks. (See Example 13, "ADL Test Program Standalone Test Result Report"). The value of **normal** and **exception** will influence whether some assertions are evaluated or not. In particular, when **exception** is true and **normal** is false, the ACF does not evaluate the *normally assertion* checks. The normally assertion checks are the translated assertion statements enclosed in the **normally** clause of the originating ADL specification.

5. Test Data Description Language

The TDD language is designed for expressing the structure of test data to be used in testing the functions from an ADL specification. A TDD specification is an abstract definition of test data, where symbolic names are used to express simple or complex data values instead of using actual numeric or character constants. By separating the characterization of data from actual data, the test specification can be reused for defining tests for different implementations that require data of the same type, but not necessarily of the same value. However, test data values are ultimately assigned by data generation functions written by the test developer. User written data generation functions are discussed later in this section. The TDD specification provides a framework through which the test designer characterizes and documents the kind and range of data that will test the function. Through the use of TDD, test data becomes the subject of a design process.

In a TDD specification, a test data type is represented by *a test variable* that defines the set of data instances that will be used as actual values of that type in a ADL generated test program. A test variable is defined in terms of one or more property definitions. A test variable property denoted by the keyword **prop**, contains a list of identifiers where each identifier symbolically characterizes a particular value of the data the test variable represents.

```
int amount (
  prop magnitude =
      {zero,small,medium,large,huge,max}
);
```

EXAMPLE 5 — Test Variable Declaration

Example 5 shows a declaration of the test variable `amount` of type `int`. The test variable contains a single property labeled `magnitude`. The property `magnitude` consists of 6 identifiers that represent 6 possible test values that would be ultimately generated in a test. Notice that the symbolic property values, or identifiers, are hints to such things as boundary conditions and equivalence classes. A main objective of the TDD language is to allow the test designer to codify the important qualities of the test input independently of the implementation being tested.

```
int amount (
  prop magnitude=
      {zero,small,medium,large,huge,max},
  prop sign=
      {positive,negative}
);
```

EXAMPLE 6 — Test Variable Declaration

In Example 6, another property named `sign` has been added to the test variable `amount`. This enhances the test variable in two ways; it signifies another important attribute of the data to be used in a test and raises the number of potentially generated test values for `amount` to 12. In TDD, the number of potential values represented by a test variable is the cross product of the property identifier sets it contains. In this case, `amount` contains the properties `magnitude` and `sign`, with `magnitude` containing 6 property values and `sign` containing 2. Thus, the number of potential generated values equals 12: (6 values of `magnitude` x 2 values of `sign`). A potential value of `amount` is now characterized by the property tuple of `magnitude` and `sign`. This example illustrates specifying a simple data type. The real power of TDD comes into play when describing a complex user data type or data that is simple programmatically, but has a complex meaning. For instance, a file descriptor in a 'C' program has a type of int, however it can represent a vast number of file name permutations, different file types and their initial state. TDD can be used quite effectively to describe the abstract values a file descriptor can represent.

5.1 Defining Tests in TDD

A complete TDD specification identifies the interface being tested, describes the test input, and describes the test instances to be performed on the functions that have been semantically specified in an ADL specification. A TDD specification defines a set of test instances with the *test directive* statement. The test directive statement specifies the function to be tested and the test variables to use for the parameters.

```
test atm_display (amount);
```

EXAMPLE 7 — TDD Test Directive

The test directive statement in Example 7 states that the function `atm_display` should be tested using the test variable amount. Using the test variable amount as declared in Example 7 would result in the function `atm_display` being tested with each of the 12 values represented by the test variable `amount` .

```
module bank;

int amount (
  prop magnitude=
      {zero,small,medium,large,huge,max},
  prop sign=
      {positive,negative}
);
account acct (
  prop type=
      {savings,checking,IRA,investment},
  prop balance=
      {zero,small,average,max}
);

test withdraw (amount,acct);
```

EXAMPLE 8 — TDD Specification for Bank Interface

Example 8 is a complete TDD specification for the bank interface specified in Example 4. In this TDD specification, the first line identifies the interface by its module name `bank`. Below that, the two test variables `amount` and `acct` are declared. The last line of the specification is a test directive that specifies the function `withdraw` is the subject of the test. Although not stated explicitly, the test directive identifies 192 possible test instances that can be performed on the function `withdraw`. Recall that the potential values represented by test variables are a function of the cross-product of the property value sets they contain. The cross-product rule is extended to the number of parameters a tested function contains as well. The `withdraw` function, which has two parameters, will be tested independently over the 12 potential values of `amount` and the 16 potential values of `acct` for a total of 192 test invocations. In the generated ADL test program, the emitted test driver iterates over the entire grid of potential test instances testing the function with each independent set of parameter values. One can imagine that the number of potential test instances for more complex functions can quickly become astronomical. TDD has language constructs called refinements, that allow the test engineer to manage this problem. Although too detailed for presentation in this paper, refinements and other TDD syntactic features are completely documented in the ADL Translation System documentation identified in Section 10, "Further Information".

5.2 TDD and test data generation

A TDD specification describes the test data at a symbolic level. These descriptions must be converted into real data at test case execution time. Test data used in the test program is supplied by *provide functions*. For every test variable declared in the TDD specification, a corresponding provide function is needed to translate the symbolic property values of the test variable into actual data values. These functions are written by the test engineer and are linked with the generated test program. The purpose of a provide function is to accept requests from the test driver to furnish data for a particular test instance specified in the TDD test directive. The test driver iterates over all of the test instances, calling each provide function to supply its data for every test instance. However, without a method of reclaiming the data, the test program could eventually consume the available system resources. Therefore, a provide function is paired with its counterpart, the *relinquish function*. The role of the relinquish function is to release system resources used in the initial construction of the data. After completion of each test instance, the test driver calls the available relinquish functions before setting up for the next test. For example, if a provide function uses the `malloc` function to allocate an instance of data, the relinquish function would be responsible for releasing the data with a call to `free`. In other situations the relinquish function may need to restore initial program state. The interaction of provide and relinquish functions are programmatic design decisions made by the test engineer.

6. ADL Natural Language Translation

One of the interesting features of the ADL Translation System is the automated generation of documentation. ADLT can generate two different documents; The Natural Language Specification (NLS) and the Test Specification (TS). In creating the documents, the adlt translates the formal ADL semantic assertions in the ADL specification and produces human readable natural language text. The ADL Translation System has been designed so that it can be localized for any language environment. In its current release the System is localized for English output translation.

The NLS appears much like a UNIX-style man page and is created from the information contained in the ADL specification. The Test Specification requires both the ADL specification and the TDD specification as input to the ADLT for generation. The information contained in the Test Specification is concerned with conveying the meaning of the tested assertions in the ADL specification as well as describing the test data and the functions under test. ADLT emits a single NLS document for the interface defined in the input ADL specification, and will emit a Test Specification for each test directive statement appearing in the TDD specification.

The default assertion translations in the generated documentation, although grammatically correct, can be quite formal and stilted. However, with some minor changes, the output is more acceptable and understandable. The ADL Translation System allows the user to improve the default natural language translations through the use of the Natural Language Dictionary (NLD). The NLD file is another input language file to the ADLT. It is a glossary of translations for identifiers that appear in an ADL specification. The purpose of the NLD is to describe, in short natural language phrases, the meaning of the identifiers that appear in the ADL specification. When element identifiers appear in assertions, ADLT translates the identifiers into the natural language phrases contained in the NLD file.

```
bank_errno = “transaction error number”;
balance(acct) = “the account balance of ”
acct {
  acct= “account number”
};
```

EXAMPLE 9 — NLD File Entries

Example 9 depicts sample entries from a NLD file used for the bank example presented earlier. The first line is a translation for the identifier `bank_errno`. With the NLD, the meaning of `bank_errno` is now defined as the “transaction error number”. Presumably, in the context of banking operations this is a more accurate description of the value represented by `bank_errno`. The second line is a description for the auxiliary function `balance`. In this translation, a description for the function `balance` and its single parameter `acct` is given.

```
...
int withdraw (account acct, int amount);
  semantics {
      ...
      normally {
          bank_errno == TRANSACTION_OK,
          balance(acct) ==
              @balance(acct) - amount,
      }
  };
};
```

EXAMPLE 10 — Partial ADL Bank Interface Specification

Example 10 is an excerpt from the ADL bank interface example presented earlier. Using the NLD translations in Example 9, the 2 assertion statements would appear in the generated documentation as follows:

- “if normal completion then transaction error number equals TRANSACTION_OK”
- “if normal completion then the account balance of account number equals the call-state value of the account balance of account number minus amount”

7. Building Automated Tests

Once all of the specification files have been constructed, the System is almost ready to generate the test cases. The final step before building test cases is constructing ‘C’ source files of functions that will assist the Assertion Checking Functions in evaluating the assertions and that will assist the Test Driver in generating test data. Once these are complete, the ADL Translation System can generate all of the relevant files.

7.1 Auxiliary Functions

Auxiliary functions, alluded to in Section 3, "Assertion Definition Language", assist the Assertion Checking Functions in evaluating assertions. While it is possible to develop assertions without referencing any auxiliary functions, such assertions could be extremely complex. Further, one of the significant productivity improvements to be gained by using the ADL Translation System is in the reuse of low-level objects. By referencing external, or auxiliary, objects in assertions, these objects can be reused as frequently as necessary when designing an interface and developing its ADL interface specification.

Recall Example 4, “Complete ADL Bank Specification". In that example, the auxiliary function `balance` is declared as:

```
auxiliary {
  int balance (account acct);
}
```

EXAMPLE 11 — Auxiliary Function Declaration

This auxiliary function is later referenced in several places in the specification. One assertion from that example is:

```
@(amount > balance (acct)) <:>
  bank_errno == INSUFFICENT_FUNDS
```

EXAMPLE 12 — Use of auxiliary function in an assertion

Because of the declaration and references, the System assumes that there is an external module that contains the definition of the function `balance`. Further, as this function was declared an `int` at the beginning of the example, the System created header file for the auxiliary functions contains a function prototype that is included and used by the Assertion Checking Functions.

The code to implement the `balance` function is beyond the scope of this paper. However, such a function would interrogate the “bank” implementation using some low-level primitives to determine the balance, feeding it back to the Assertion Checking Function for evaluation. In this case, the implementation of the auxiliary function is probably pretty simple. By creating “small” auxiliary functions such as this, it is possible to get a very high level of function reuse.

It is important to note that these auxiliary functions do not have to be provided by the user. In many cases, existing system functions can be considered auxiliary functions. These can easily augment the interface and test designer’s collection of assertion evaluation assisting functions. For example, when evaluating the UNIX system `chmod` function, some assertions will have to use other system primitives like `stat` and `access`. All that is needed is to declare the functions in the auxiliary clause in a manner consistent with the system function prototype and use it as any other auxiliary function. The System and the linker will do the rest.

7.2 Provide/Relinquish Functions

The Test Data Description files contain both test variable descriptions and test declarations. The purpose of these items was described in Section 5.2, "TDD and test data generation", but it is important to understand the mechanism fully.

The test variable descriptions usually describe a test variable in terms of sets of properties that the variable might have. Test variables, when used in conjunction with a provide function, generate an instance of that variable that is then used by the test driver as an argument to the function under test. Therefore, it is necessary that there is a provide function associated with each parameter of the function under test, along with any environmental variables that might need to vary during the evaluation of the function’s behavior. Because provide functions may set system state or allocate system resources, ADL also requires a user-supplied relinquish function. The purpose of relinquish functions is to release these resources at the discretion of the test designer. These provide and relinquish functions are expected to be in the same file. A header declaring the function prototypes and enumerated property sets is created by the System. It should be included by the user-provided functions, and is also included by the test driver. The System generated test driver calls these user-written provide and relinquish functions to create and destroy instances of the test data for each test case that is run.

7.3 Generating Test Cases with adlt

The System is capable of generating a complete test system against any given ADL interface specification and Test Data

Description. With various command line options, it is also possible to generate any subset of a test system. The description of the various adlt command line options is beyond the scope of this paper. When executed in its default mode with a single tdd file on the command line (e.g. `bank.tdd` in the case of the bank example), the System creates a makefile, a TET scenario file, and a number of C Language source files. These files, used in conjunction with the user provided auxiliary and provide/relinquish function sources, will generate a test case for each test directive in the associated TDD file. This can be done by using the Make program directly, or can be done using the build mode of TET.

Specifically, the adlt program emits a test driver source code file for each test directive statement in the TDD specification. This effectively creates a separate test program executable for each test directive. In addition, the translationof the TDD specification produces 'C' header files that declare the provide and relinquish function signatures. Included in this header file are 'C' enum types that contain constant identifiers that are a direct mapping of the test variable property names in the originating TDD specification. The test driver uses the enum values in the call to a provide function to identify the requested value needed for the test instance. The test designer is responsible for creating the source file that contains the provide and relinquish functions. The figure below shows the components of an ADL generated test program.

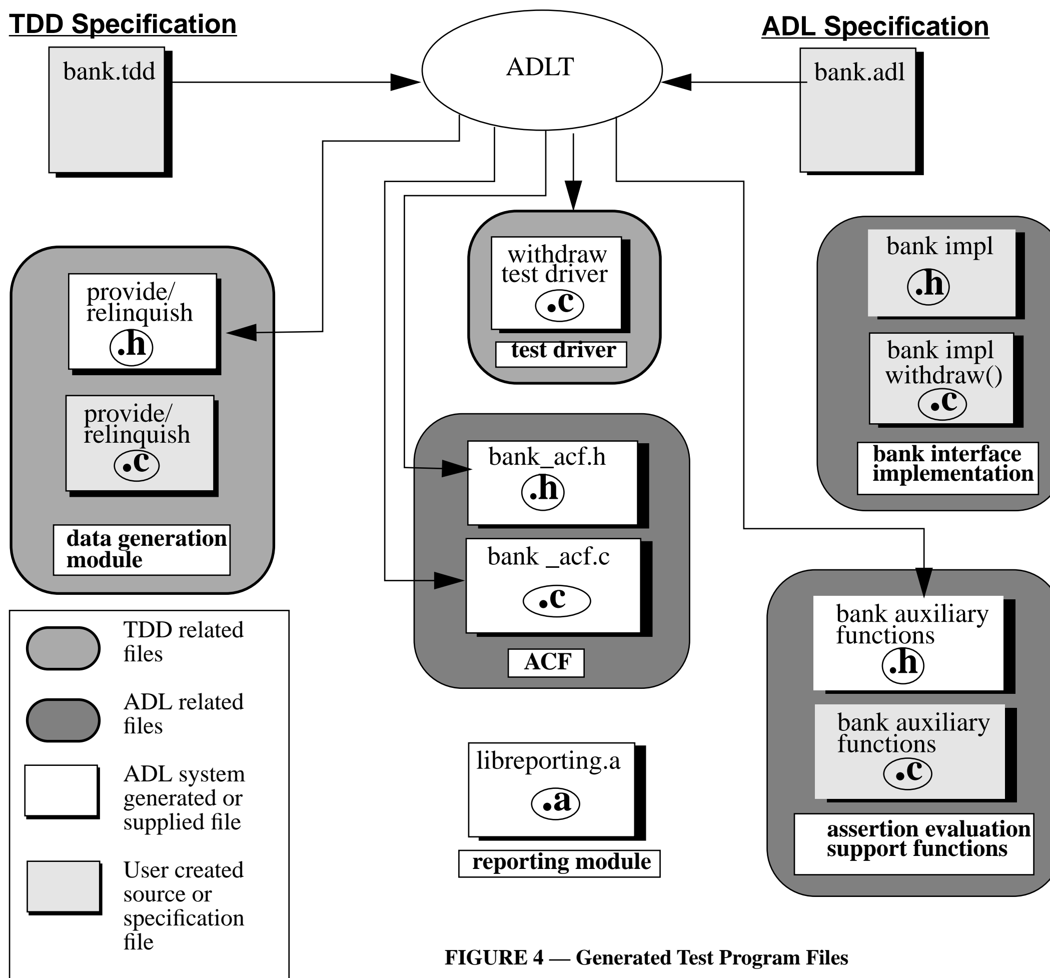

FIGURE 4 — Generated Test Program Files

8. ADL Test Program Reporting

Once the ADLT has emitted all of the relevant source files and the user has coded the required auxiliary and provide functions, they can be compiled and linked into a single executable. To properly create a test program, the environment must have access to an ANSI ‘C’ compiler and to the standard ‘make’ facility. For convenience, adlt automatically generate a makefile with all the required dependencies.The user executes ‘make’ to build the test program as indicated in the TDD specification. Once compiled, the resulting program is a self contained test suite that offers several formats of result output.

An ADL test program can be built for two modes of test reporting; stand-alone reporting, or TET mode reporting. By default, the emitted makefile will create an ADL test program with stand-alone reporting mode. To generate a TET compatible test program, the user needs to altering a few lines in the generated makefile to link special TET libraries and a TET compatible reporting module. In addition, the user must install the TET facility prior to compiling the ADL test program. The TET results reporting is different than reporting in stand-alone mode. This was done in order to accommodate POSIX-style test assertion reporting. The differences in the two reporting modes are described briefly in Section 2.2, "ADL and the Test Environment Toolkit". A detailed explanation is available in the ADL Translation System documentation. Example 13 shows a sample of reporting in stand-alone reporting mode.

Reporting of text execution results can be displayed in verbose, error or summary formats. The desired format is available through a command line switch argument. Additionally, the ADL test program has a command line switch that allows the user to select the range of test instances to be executed. Example 13 shows the verbose output of a single test instance of the bank interface example presented earlier in this paper.

```
Test result [1]: PASS

Outcomes:
  F exception binding
      exception := (return == -1)
  T normal binding
      normal := !exception

Assertions:
           T Assertion 1
               @(amount < 0) <:> bank_errno == NEGATIVE_AMOUNT
           T Assertion 2
               @(amount > balance(acct)) <:> bank_errno == INSUFFICENT_FUNDS
           T Assertion 3
               exception --> unchanged(balance(acct))
           normally (
               T Assertion 4.1
                    bank_errno == TRANSACTION_OK
               T Assertion 4.2
                    balance(acct) == @balance(acct) - amount
               T Assertion 4.3
                    return == balance(acct) - amount
           )

Test variables:
  acct:
      type = savings
      balance = zero
  amount:
      magnitude = zero
      sign = positive

Summary: withdraw: PASS
   passed:       1 (100% of total)
   skipped:      0 (  0% of total)
   failed:       0 (  0% of total)
   errors:       0 (  0% of total)
   undefined:    0 (  0% of total)
   ambiguous:    0 (  0% of total)
```

EXAMPLE 13 — ADL Test Program Standalone Test Result Report

9. Real World Usage

Part of the process of any research project is validation of its results. The ADL Project has several stages during its four year life when such validation is scheduled. During 1993 the project was dedicated to design of the system. In this phase, validation was done by conducting several industry reviews of the ADL design documents and incorporating industry feedback into the design. During 1994 the project is validating its results by conducting formal and semiformal beta testing of the early ADL Translation System implementation.

9.1 Formal Testing

In order to adequately evaluate the ADL Translation System as it evolves, and in order to ensure that the environment meets the needs of the test development community, it is essential that the ADL Project develop some prototype test suites using the ADL Translation System. During 1994 the project will develop two test suites, as described below.

These suites will be carefully evaluated by an independent group of experts in the area under test to ascertain the tests' value. In addition, the ADL Translation System will be evaluated by the test suite developers as a natural consequence of their using the environment to develop tests. The test suite developers will be expected to work closely with X/Open and Sun Microsystems Laboratories to ensure that any problems they encounter are quickly addressed in the evolving ADL Translation System. All of the work will be finished during calendar 1994 in order to give Sun Microsystems Laboratories adequate time to address issues raised before the end of the project year in March 1995, and in order to allow time to evaluate the results prior to the end of the project year.

CORBA

The first suite to be developed is a test suite against the Common Object Request Broker Specification, or CORBA, version 1.2. This specification, recently ratified by the Object Management Group, represents a significant interface in the community for which there is no test suite. The CORBA consists of seven major components, of which only some are to be tested by this project. Specifically, the ADL project will test the Dynamic Invocation Interface, the ORB Interface, and provide rudimentary testing of the IDL Syntax and Semantics and C Language Binding for the Syntax and Semantics (by implementation of objects). The test suite is being developed by Applied Testing and Technology of San Jose, California. The total number of function interfaces to be tested is on the order of 30.

In order to ensure that this test suite meets the needs of the CORBA community, X/Open has secured the cooperation of the Object Management Group and many of its members. They have assigned resources to work with the test developer to define the assertions, review the test specification, and evaluate the resulting test suite against their implementations. In order to ensure that this review is of the highest quality, there will be several review periods during which the CORBA community and the ADL project team will work together to resolve issues in the test suite.

However, the quality of the generated test suite is of secondary importance to the quality of the test of ADL. Consequently, the development plan has within it several milestones where the test suite developer is required to submit an evaluation report against each release of the ADL Translation System made in calendar 1994. In addition, the project team will establish a mailing list for the test suite developers and the team at Sun Microsystems Laboratories, so that these people can easily work together to resolve issues arising during test suite development. Finally, Sun Microsystems Laboratories has agreed to provide assistance to the test suite developers to speed their introduction into using ADL. All of these steps are designed to ensure that the process of developing this prototype test suite using the ADL Translation System results in the best possible test of that environment.

TET

The second test suite to be developed is against the interfaces defined in the Test Environment Toolkit, or TET, specification version 1.5. This specification defines a testing environment that is standard in the testing industry, and upon which the ADL Project is partially dependent. This environment is constantly evolving, and a quality test suite is essential to ensure that backward compatibility is not damaged during that evolution. The TET specification has four major components, only three of which are to be tested under this project. Specifically, the components to be tested are the C Language API, the Shell API, and a rudimentary test of the 'tcc' command line interface (through usage). This test suite is being developed by Mount Bonnell of Austin, Texas. The total number of interfaces to be tested is on the order of 20, with the Shell language API having identical semantics to the C Language API.

In order to ensure that this test suite meets the needs of the TET community, X/Open has identified a number of members of the TET Workshop that are assisting in the development of this test suite. They have committed resources to work with the test developer to define the assertions, review the test specification, and evaluate the resulting test suite against TET implementations. In order to ensure that this review is of the highest quality, we have scheduled several review periods during which the TET

community and the ADL project team will work together to resolve issues in the test suite.

In addition to the mechanisms for assisting in the interface between the test developer and Sun Microsystems Laboratories indicated above, the project is prepared to commission an independent audit of the generated test suite to ensure that it is of the highest quality. This audit will be conducted should the feedback from the TET Workshop *not* be of sufficient quantity and quality to persuade the ADL team that a thorough review of the deliverables has been completed. The audit will be done by an independent professional testing organization, and will be done after the completion of the test suite, but before the close of the project year. An audit report will be included in the project deliverables at the close of the project year.

9.2 Semiformal Testing

In addition to the formal tests described above, the entire testing community is being invited to participate in a formal beta test of the ADL Translation System. This beta test started on July 1, and continues throughout the project year. The purpose of this additional test is to ensure that the research results can be used effectively in a variety of work environments to test a variety of different types of interfaces.

The beta testers have been asked to perform an evaluation of each release of the ADL Translation System delivered during calendar 1994 against a specific set of evaluation criteria. In addition, the beta testers are expected to attempt to use the ADL Translation System to develop tests for some interfaces in their environment. Finally, the beta testers are expected to develop a year-end evaluation report summarizing their impression of the ADL Translation System and their recommendations for its improvement.

In exchange for this work, the ADL Project team has established a private mailing list for beta participants, assigned beta participants high priority in addressing their issues with ADL Translation System releases, and invited them to several ADL design meetings during the review period. In addition, the X/Open project manager is conducting monthly meetings with each beta participant, attempting to address any outstanding issues. Finally, all beta testers are to receive explicit credit in the ADL Translation System documentation for their contribution to the project.

9.3 The PIKS project

PIKS, or Programmers Imaging Kernel System, is a library of approximately 100 image processing functions whose API (interface) has been recently adopted as an ISO standard. A group was formed from members of the PIKS standards group whose mandate was to come up with a scheme to do conformance testing of the anticipated future implementations of PIKS. This group (the PIKS testing group) decided to use ADLT to facilitate this process.

Conformance testing is the process of exercising a software system on different inputs to determine whether or not the system behaves as described by its specifications. Conformance testing assumes the existence of both the *implementation* of the system as well as the *specification* of the system. The tasks of conformance testing is then to:

1. Determine the inputs on which to exercise the system;
2. Exercise the system on these inputs; and finally,
3. Determine that the system behaved as described by its specifications.

A collaborative effort was undertaken between the Primavera group and the PIKS testing group to come up with a conformance testing scheme for PIKS using ADLT. This process involves:

1. Rewriting the ISO Standard of the PIKS API in ADL;
2. Describing the inputs on which to test PIKS implementations in TDD;
3. Using ADLT to generate a conformance test-bed; and
4. Exercising PIKS implementations on this test-bed to validate them.

The manual effort involved in this project is in (1) and (2). Once this is done, the rest of the process, namely (3) and (4), can be done automatically.

PIKS PROJECT STRATEGY

The collaboration between the PrimaVera and the PIKS testing groups began around April 1994. A tutorial on ADLT was presented to the PIKS testing group which resulted in an initial understanding between the two groups. The PrimaVera group offered to perform the initial task of writing ADL and TDD specifications for five PIKS functions each representing a different category of PIKS functionality. The five functions are:

`allocate_image`:
: Allocates a reference to an image object and creates metric and color descriptor information in its attributes.

`get_pixel_array`:
: Returns a rectangular array of pixels to an application.

`colour_conversion_linear`:
: Performs linear inter-band color conversion of a trichromatic color image between color spaces.

`convolve_2d`:
: Performs two-dimensional convolution with a general impulse response function array for several image boundary conditions.

`rotate`:
: Performs two-dimensional or three-dimensional rotation of an image about a point.

During the first phase of this project which lasted till the end of May 1994, the PrimaVera group divided the task by assigning each function to a different person and each developed a prototype of the ADL and TDD specifications for the functions. The phase realized certain limitations of the ADLT tool-suite and its associated languages. The tool-suite and the languages were improved in various ways — to name a few, opaque types and quantifiers were added to the ADL language, and a more detailed scheme for reporting errors was added to the ADLT tool-suite.

In early June 1994, the two groups met again to coordinate efforts. It was agreed that ADLT had proved itself capable of being used as a conformance testing capability for the PIKS API. The next step was for the PrimaVera group to completely rewrite the specifications of the five functions in a more organized and modular fashion. This would set up an environment in which development of specifications for more functions would be made simpler by the availability of libraries of reusable concept specialized for use in PIKS specifications. While the PrimaVera group was achieving this goal, the members of the PIKS testing group decided to start working on specifying various functions themselves.

At this point in the project (the end of August 1994), the PrimaVera group has achieved its goals and the two groups will meet again shortly to make future plans, however there are two key risk factors — the process of ADL technology transition to the PIKS testing group, and financial funding to complete the project.

The results obtain so far from using ADLT as the test vehicle for this project are too inconclusive to report in detail. Due to the extremely preliminary implementation of the PIKS interface, we encountered numerous inconsistencies between the PIKS ADL specifications and results obtained from using the ADLT generated test cases on the preliminary PIKS implementation. More investigation of why the test case failures are occurring needs to be conducted. This situation pointed to an ADLT deficiency in the detail of information reported by the generated tests. More detailed reported information of why a particular test instance fails would have proven quite helpful in this early stage of the PIKS interface conformance testing. Hence, we are currently designing and implementing a much more detailed reporting system to be used by the generated tests. This new reporting enhancement will be available in version 0.6 of the ADL Translation System. Version 0.6 is scheduled for release Dec 23, 1994.

10. Further Information

The ADL Project is an ongoing research project open to all interested parties. The interim and final results of the research, including an implementation of the ADL Translation System, are freely available under an X Window System-like copyright and grant of rights. All project deliverables to date are readily available via the Internet. For more information on the project and its overall goals, please contact Shane McCarron (s.mccarron@xopen.co.uk). For access to project deliverables, mailing lists, and related information, see the following:

ADL Project Mailing List

A public electronic mailing list for ADL Project participants is sponsored by X/Open. This list is used to discuss research directions with the community, update the community on progress, and announce the availability of new releases of research deliverables. To join the mailing list, send mail to xopubadl-request@xopen.co.uk and ask to be added. Be sure to include your full name and company name, as these are placed in the X/Open database.

ADL Deliverable Repository

A repository of ADL project deliverables is maintained by X/Open on Uunet's public access ftp site. This repository contains the current and preceding release of ADL research deliverables, as well as a collection of background materials, and a number of documents that include:

- *ADL Language Reference Manual*
- *ADL Translator Design Specification*
- *ADL Translator Programmer's Guide*
- *ADL Translator User's Guide*

Deliverables are in source form, and delivered documents are also in PostScript™ form. To access the repository, connect to ftp.uu.net via ftp and change to directory /vendor/adl.

Test Environment Toolkit Workshop Mailing List

The Test Environment Toolkit (TET) is managed by a public group called the TET Workshop. X/Open sponsors a public electronic mailing list for the TET Workshop. To join this mailing list, send mail to tetworks-request@xopen.co.uk and

ask to be added. Be sure to include your full name and company name, as these are placed in the X/Open database. To formally join the TET Workshop, also send mail to r.martin@xopen.co.uk.

Test Environment Toolkit Repository

Current and past releases of the Test Environment Toolkit and related extensions are available on a number of repositories throughout the world. X/Open supports a repository on its ftp server. To access this repository, ftp to xopen.co.uk and change directory to /pub/TET. The README file in that directory contains more specific information.

11. Acknowledgments

The primary funding of the ADL Project has been provided by the Information-technology Promotion Agency of Japan. Without their sponsorship, the research into automated test generation systems would not have achieved the level completion and industry exposure it currently has under the ADL Project. Much of the information presented in this paper is a culmination of the research and development activities conducted by all members of the SML Primavera Group. We would like to thank each of them for their specific contributions to this paper and to the ADL Project. In addition we would like to thank Mark Hefner, the Primavera summer intern student, for his work on the PIKS testing project.

12. Biographies

Matt Evans is the ADL Project Lead for Sun Microsystems Laboratories Inc. He has worked for Sun Microsystems since 1990 in the capacity as Lead Test Engineer for Sun Connect’s Network Product Assurance Group. Later he transfer to SunSoft’s Corporate Software Engineering group with the primary responsibility of porting GridGen to the Distributed Object Everywhere (DOE) environment. GridGen was an early prototype of the ADL Translation System developed by SML’s PrimaVera group. He has been involved with the ADL Project since its inception and currently manages all aspects of the ADL project.

Shane P. McCarron is the Testing Research Manager for the open system community's primary specification and certification organization, X/Open. Since 1989, he has worked with the open systems software testing industry to improve the ways in which test suites are developed and distributed throughout the open systems community. His current projects include the Assertion Definition Language project, test suites for evaluating conformance to various open systems industry interface specifications, and the ongoing support and maintenance of the freeware Test Environment Toolkit.