

Automated Test Generation Technology

Assertion Definition Language Project

James deRaeve
Testing Business Unit Manager

Shane P. McCarron
Testing Research Manager

X/Open Company Ltd.

This paper presents the findings of a four year project conducting research in the area of a formal interface behavior description grammar and its translation into tests and documentation for the interfaces described. In this paper the research team presents a summary of the history of the project, some data about the project's progress, and summarizes the conclusions that the team has reached as the project concludes its initial phase.

Table of Contents

| | | | | | |
|-------|--|----|--------|--|----|
| 1. | Background and Objectives | 1 | 8. | Availability | 31 |
| 1.1 | Project Scope and Purpose | 1 | 9. | Conclusions | 32 |
| 1.2 | History | 2 | 10. | Future Plans | 32 |
| 1.3 | Research Performed | 2 | 10.1 | Deployment Plans | 32 |
| 1.4 | Research Participants | 4 | 10.1.1 | CORBA 2 Test Suite | 32 |
| 2. | Basic ADL Architecture | 6 | 10.1.2 | Common Desktop Environment Test Suite | 33 |
| 2.1 | Architectural Model | 6 | 10.1.3 | X/Open Specifications | 33 |
| 2.2 | ADL and the Test Environment Toolkit | 8 | 10.2 | Enhancement Plans | 33 |
| 3. | Assertion Definition Language | 9 | 10.2.1 | ADL for C++ | 33 |
| 4. | Assertion Checking Functions | 13 | 10.2.2 | ADL for IDL | 33 |
| 5. | Test Data Description Language | 15 | 10.2.3 | ADL for Java | 34 |
| 5.1 | Defining Tests in TDD | 16 | 10.2.4 | ADL Localization Packages | 34 |
| 5.2 | TDD and Test Data Generation | 17 | 10.3 | Follow-on Research | 34 |
| 6. | Natural Language Translation | 17 | 10.3.1 | Unit Testing | 34 |
| 6.1 | Objectives | 17 | 10.3.2 | System Testing | 34 |
| 6.2 | Design Decisions | 18 | 10.3.3 | Random Testing | 35 |
| 6.3 | Analysis | 20 | 10.3.4 | Stress Testing | 35 |
| 6.3.1 | The Translation Process | 20 | 10.3.5 | Interference Testing | 35 |
| 6.3.2 | Some Prolog Details | 21 | 10.3.6 | External Research | 35 |
| 6.4 | Lessons Learned | 23 | 11. | References | 35 |
| 7. | Evaluation of the Research | 24 | 12. | Papers and Presentations | 35 |
| 7.1 | Common Object Request Broker Test Suite | 24 | | | |
| 7.2 | Test Environment Toolkit Test Suite | 28 | | | |
| 7.3 | Changes to the Research | 31 | | | |

1. Background and Objectives

Since 1992 the ADL Project team has been working closely with the Information-technology Project Agency to further the goals of Open Systems through the research of Open Systems Testing Tools. The first major phase of this research is now nearly complete. The results of this research, freely available under an X Window System-like copyright and grant of rights, have been very promising. The purpose of this paper is to convey some of the key findings of this project. This paper presents:

- The basic architecture of the ADL system,
- The application of this system to two different examples of testing problems and the results therefrom,
- The applicability of this research to the generation of natural language specifications, and
- The plans of the research team for the future extension and application of this research.

1.1 Project Scope and Purpose

The Assertion Definition Language Project has defined a formal specification language and language translator that permit the specification of interface behavior. Once specified, the translator can transform the formal specification into a less formal “natural language” specification, a test specification, and tests for an implementation of the specification.

The purpose of this research has been to help ease the burden of the Open Systems industry in a number of important areas, as described below:

Improve overall test coverage

Historically, the coverage of test suites has been at best thorough (as defined by IEEE P1003.3.1-1991 [6]). The goal of this project was to sufficiently automate the generation of tests that it is possible to double the thoroughness of a test suite's

coverage of an interface specification while still providing tests of the same quality as those produced manually.

Reduce startup costs of test development

Traditional test suite development is a very expensive process. The translation of interface specifications to test specifications to test suite code currently takes a significant amount of skilled programming resource. Through ADL these expensive and scarce resources can be targeted at the difficult parts of test development, rather than the mundane aspects of it that are better handled by automation. This is accomplished by defining the specification itself in a formal language which minimizes the need for subsequent recasting of the specification for testing purposes.

Reduce the time to market of test suites

Currently the development of test suites takes a significant amount of time, and is often not accomplished until well after the interface specification is complete. Through this project the lag time between specification completion and test suite availability will be reduced dramatically. This is accomplished by reducing the trial and error nature of the existing practice which is characterized by a large number of time consuming iterations in defining the interface and test suite.

Increase code reusability

Traditional test suites contain a significant amount of repeated code, but this code is usually not reused. Through this technology the reuse of common code is easy and automatic.

Reduce lifecycle costs

The cost of maintaining a test suite as its underlying specification evolves or as the test suite is debugged is extremely high. By increasing the amount of code reuse and deriving the test strategies from the actual specification, this lifecycle cost can be reduced considerably.

Improve reliability of tests

Test Suites are traditionally only as reliable as the translation of the interface specification to assertions, and then of the subsequent translation of those assertions to test case code. By effectively automating the last two steps of this process, this project has increased the reliability of the resulting test suite (i.e. the number of programmer and translator introduced errors).

Increase the accuracy and completeness of the interface specification

Interface specifications are traditionally developed through an ill-defined and error-prone process that generates incomplete and inaccurate specifications. By providing a structured language in which the semantics and syntax of interfaces are defined, the accuracy and completeness of the interface specification can be significantly increased.

1.2 History

The Assertion Definition Language project arose out of the Open System community's desire to have tools that would speed the development of tests for interfaces defined by open interface specifications¹. The development of these tests has historically been slow and costly. X/Open, a leader in the development of these "conformance tests", recognized that the rapid expansion of the Open Systems community and resulting standards would soon far outdistance the ability of the industry to test such standards. X/Open felt that it was theoretically possible to describe the behavior of these standards in such a way that they could be automatically tested.

At about the same time IPA announced their Open Fundamental Software Technology Project and invited X/Open and UNIX International to submit a project proposal.

1. Open interfaces are ones controlled by an independent third party and subject to public review, as opposed to proprietary interfaces controlled by a single organization.

These organizations determined that the most important thing they could do for the industry jointly via this project was to sponsor research in the area of automated testing. This proposal was ultimately accepted by IPA, and the ADL Project was officially under way.

The first step of this project was to develop a set of comprehensive requirements for the research. In order to do this, X/Open and UNIX International invited a group of prominent conformance testing experts from across the Open Systems industry to participate. The resulting requirements document formed the basis of the research.

Following this work, X/Open and UNIX International solicited from the industry existing research that was towards the requirements, with the intent that they would partner with an existing research group and speed the project. This solicitation resulted in a number of potential starting points, each of which was carefully evaluated.

The research project selected is original, patented research by Sun Microsystem's Laboratories (SML). This original project, named PrimaVera, was targeted at automating the testing of interfaces described through a syntax much like that of the Object Management Group's (OMG) Interface Definition Language (IDL).

Since 1992, the ADL Project has worked to refine this basic research and deliver it in a form useful to the Open Systems community. The specification language has been extended and ported from IDL to ANSI C, natural language generation has been added, and test generation has been refined and extended.

1.3 Research Performed

This project has performed the following major research tasks over the last four years:

- Developed requirements for an automated test generation system.
- Evaluated existing industry research in this space and selected a base line from which to start.

- Designed and developed the Assertion Definition Language - a formal grammar for expressing assertions.
- Designed and developed the Test Data Description Language - a grammar for describing test data and tests in abstract terms.
- Designed and developed the Natural Language Dictionary language - a grammar for describing meaningful natural language translations of ADL assertion components.
- Evaluated various interim implementations of each of the above through a broad industry alpha and beta testing program.

The research has not been easily divided into the various fiscal years of the project. However, each fiscal year did have a number of specific deliverables:

Phase 1 Oct 92 - Mar 93

(1) Activities:

Initial Survey and research (Oct-Mar)
 Initial specification of ADL (Oct-Mar)
 Initial Design of ADL Translator (ADLT) (Oct-Mar)
 Design of necessary TET extensions (Jan-Mar)

(2) Deliverables:

Research Plan and results
 Draft ADL Specification
 Draft ADLT Design
 Draft TET extensions design

Phase 2 Apr 93 - Mar 94

(1) Activities:

Complete ADL specification (June-Jan)
 Complete design of ADL Translator (June-Dec)
 Develop prototype ADL Translator (Aug-Mar)
 Implement TET extensions (July-Jan)

Select beta sites (Oct-Mar)

(2) Deliverables:

ADL Language Ref manual, Version 1.0
 ADLT design spec, Version 1.0
 ADLT User's Guide, Version 0.6
 ADLT Programmer's Guide, Version 0.6
 dTET User's Guide, Revision 1.1
 dTET Programmer's Guide, Revision 1.0
 ADLT source code, Version 0.3
 TET extensions source code Version 2.2
 Beta site plans

Phase 3 Apr 94 - Mar 95

(1) Activities:

Support and maintain ADL Translator (Apr-Mar)
 Enhance ADLT Translator (Apr-Mar)
 Generate beta site test suites (2) (Apr-Dec)
 Plan internationalisation (Jul-Sep)

(2) Deliverables:

ADL Language Ref manual, Version 2.0
 ADLT Design Specification, Version 2.0
 ADLT User's Guide, Version 1.0
 ADLT Programmer's Guide, Version 1.0
 ADLT source code, Version 0.7
 Enhanced ADL Translator
 Prototype test suites (CORBA, TET)
 Beta site reports

Phase 4 Apr 95 - Mar 96

(1) Activities:

Support and maintain ADL Translator (Apr-Mar)
 Complete ADLT Translator (Apr-Dec)
 Localise ADL Translator (Apr-June)
 Complete evaluation of test suites (Apr-Jan)
 Generate test suite (File System) (Aug-Dec)

(2) Deliverables:

Enhanced/localised ADL Translator, Release 1.0

ADL Language Reference Manual, Release 1.0
 ADLT Design Specification, Release 1.0
 ADLT User's Guide, Release 1.0
 ADLT Programmer's Guide, Release 1.0
 Prototype test suite (File System)
 Beta site report
 Final project report

1.4 Research Participants

The success of the ADL project is the result of many people's contributions. First, credit must go to the Project Steering Board, who worked over several years to ensure that the project met the needs of a larger community (Table 1, "ADL Project Steering Board," on page 4).

Second, there were many people outside of the project who worked with us to help make the generated technology a mature piece of software (Table 2, "External Project Participants," on page 4). Third, the Japanese Information Technology Consortium hosted a multi-year workshop on ADL, providing valuable feedback about the quality and applicability of the research both in Japan and world-wide (Table 3, "ITC Workshop Participants," on page 5). Finally, there is the research team itself (Table 4, "Research Team Members," on page 5). All of these people, through their tireless efforts, helped to develop what we believe to be one of the most significant advances in testing technology ever.

| Company | Name | E-Mail Address |
|---|----------------|--------------------------|
| Information-technology Project Agency | Noboru Akima | akima@ipa.go.jp |
| US National Institute of Standards and Technology | Roger Martin | rmartin@ncsl.nist.gov |
| Open Software Foundation | John Morris | johno@eworld.com |
| Sun Microsystems Labs | Alberto Savoia | alberto.savoia@xopen.org |
| X/Open | Mike Lambert | m.lambert@xopen.org |

TABLE 1. ADL Project Steering Board

| Company | Name | E-Mail Address |
|-----------------------------------|-------------------|-----------------------|
| Applied Testing and Technology | Andy Silverman | andy@aptest.com |
| Pencom | Christine Kungl | cmk@pencom.com |
| UniSoft | Andrew Twigger | att@root.co.uk |
| | Tom Norris | tnn@root.co.uk |
| UK National Physical Laboratories | Nick North | nnp@nrl.gov.uk |
| Mount Bonnell, Inc. | Barry Book | brb@bonnell.com |
| | William King | wpk@bonnell.com |
| Kanri Kogaku | Masaharu Obayashi | obayashi@kthree.co.jp |
| | Hideyuki Okada | okada@kthree.co.jp |
| | Ryo Ojima | ojima@kthree.co.jp |

TABLE 2. External Project Participants

| Company | Name | E-Mail Address |
|---------------------|---------------------|------------------------------|
| Toshiba | Mitsukazu Uchiyama | uchi@ap.ilab.toshiba.co.jp |
| Matsushita | Yoshinori Itabashi | itabashi@trl.mei.co.jp |
| Sumisyo Information | Motoichiro Sakai | sakai@scs.co.jp |
| Toshiba Information | Yoshiyuki Fukano | nyanta@sohon.tjsys.co.jp |
| Kanri Kogaku | Masaharu Obayashi | obayashi@kthree.co.jp |
| | Hideyuki Okada | okada@kthree.co.jp |
| ITC | Tousaku Oishi | oishi@itc.co.jp |
| | Kazuo Nagashima | nagasima@itc.co.jp |
| | Hitoshi Iwamoto | iwamoto@itc.co.jp |
| | Taisuke Ueeda | ueeda@itc.co.jp |
| IPA | Hideyuki Takashima | hide@stc.ipa.go.jp |
| Fujitsu | Yuichi Takada | ytaka@zeus.yk.fujitsu.co.jp |
| Micro Soft Japan | Kaoru Okumura | kaoruo@microsoft.com |
| Nagoya University | Shinichiro Yamamoto | yamamoto@nuie.nagoya-u.ac.jp |

TABLE 3. ITC Workshop Participants

| Company | Name | E-mail Address |
|--------------------------|----------------|------------------------|
| X/Open Company Ltd. | James deRaeve | j.deraeve@xopen.org |
| | Chris French | c.french@xopen.org |
| | Shane McCarron | s.mccarron@xopen.org |
| | James Andrews | j.andrews@xopen.org |
| | Toru Yoneda | t.yoneda@xopen.org |
| | Yasushi Takada | y.takada@xopen.org |
| | Yasuhiro Izuno | y.izuno@xopen.org |
| Sun Microsystems Labs | Alberto Savoia | alberto.savoia@sun.com |
| | Matt Evans | matt.evans@sun.com |
| | Roger Hayes | roger.hayes@sun.com |
| | Bill Gogesch | bill.gogesch@sun.com |
| | Sriram Sankar | sriram.sankar@sun.com |
| | Roongko Doong | roongko.doong@sun.com |
| | Mark Hefner | merk.hefner@sun.com |
| | Jos Marlowe | jos.marlowe@sun.com |
| | Jon Gibbons | jon.gibbons@sun.com |
| | Rob Duncan | rob@aimnet.com |
| | Frank Nellis | frankn@netcom.com |
| Release Consulting Group | Mike Hatam | mike@rcgnet.com |

TABLE 4. Research Team Members

| Company | Name | E-mail Address |
|-----------|--------------|--------------------|
| | Kjell Olsson | kjell@rcgnet.com |
| Neologica | Ann Merrill | merrill@netcon.com |
| | Ellen Ullman | ullman@netcom.com |

TABLE 4. Research Team Members

2. Basic ADL 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^{®1}-like systems fit this description. The generated test programs are ANSI C. The System itself is written in C++ and ANSI C and will build and execute on any 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 4.0. It is provided in both source and PostScript[™] forms in the distribution. In addition to these requirements, a complete ADL Translation

1. UNIX is a registered trademark in the United States and other countries licensed exclusively by X/Open.

System platform may also have the Test Environment Toolkit installed as a test controller (see below).

The current ADL System distribution archive contains separate releases for both the ADL system software and the ADL documentation set. The complete system occupies approximately 30 megabytes of disk space. The System builds successfully and with no customization on many platforms, including generic System V Release 4.0 platforms with gcc version 2.6.0 or above installed.

2.1 Architectural Model

The ADL Translation System architecture is made up of ten basic components. 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 diagram in Figure 1. The elements of the diagram are described below.

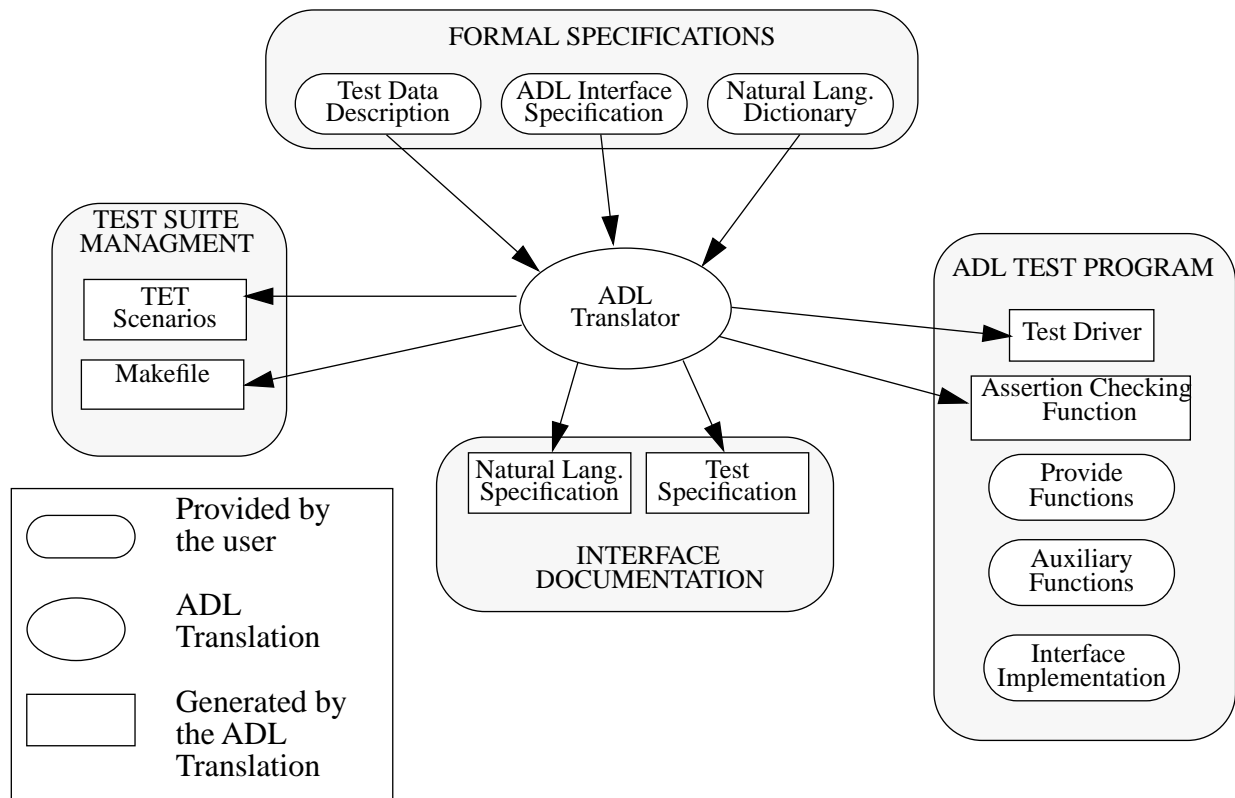


FIGURE 1 - Abstract ADL Architecture

Formal Specifications

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

ADL Interface Specification

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

Test Data Description Specification

The Test Data Description (TDD) specification is a file (or files) that define the test data and specify the test execution.

Natural Language Dictionary

The Natural Language Dictionary (NLD) is a reference file (or files) that defines the natural language translation for elements of

the ADL specification. 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 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 data definitions and test directives found in the TDD. This generated source code may be compiled and linked with user-written support code and with the ADLT library to create a test program. Each test program tests one function from the interface; several test programs can be created from one specification.

Test Driver

The ADL Translator generates source code for the Test Driver based on a test specification from the TDD file. 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 an implementation of the specified interface and checks the results of invocation against the semantic assertions specified in the ADL specification.

Auxiliary Functions

Auxiliary functions are user- or system-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 test directive, in either summary or detailed testing mode (see Section 2.2).

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 create the executable test program from the source (the specification, support code, and implementation). The Makefile will run the ADLT translator, C compiler, and linker as required. 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 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 parameterized 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 exist 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

```

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
      == INSUFFICIENT_FUNDS,
    exception -->
      unchanged (balance(acct)),
    normally {
      bank_errno ==
        TRANSACTION_OK,
      balance(acct) ==
        @balance(acct) - amount,
      return == balance(acct)
    }
  }
};

```

conditions that cause a system call to fail and test that the call has set the 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. |

TABLE 5. ADL Assertion Syntax

| ADL Assertion Syntax | Description |
|--|---|
| 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 <i>universally quantified expression</i> , indicated by the keyword forall or an <i>existentially quantified expression</i> , 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. |

TABLE 5. ADL Assertion Syntax

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.

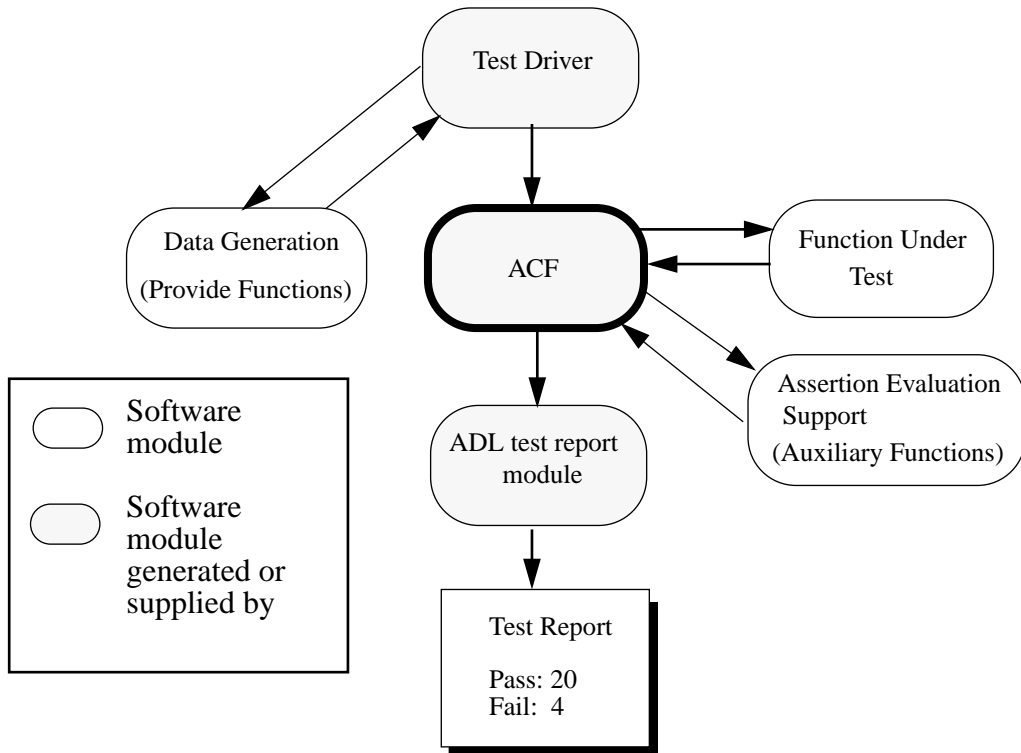


FIGURE 2 — ACF Flow of Control

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".

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 unusual test results UNDEFINED and AMBIGUOUS.

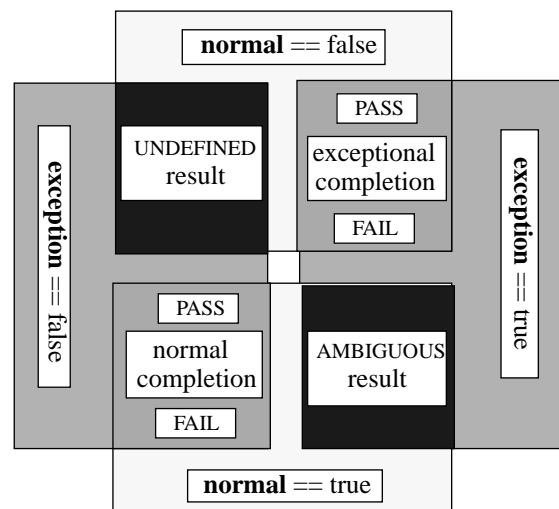


FIGURE 3 — Possible Reported Test Instance Results

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. 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 — Expanded 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.

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. Natural Language Translation

Documentation is an important aspect of software development. Accurate documentation is the product of good communication between development staff and documentation staff. ADLT natural language services help bridge the gap between developer and technical writer by rendering an accurate prose interpretation of the formal ADL specification at every stage of product development. In consequence, the technical writing staff will have a sound basis for documentation at the time of specification. This break from tradition is not only advantageous to the writing staff, but to the development staff, who can base their work on the ADLT-generated documentation.

Using ADLT moves two often neglected aspects of software development to the fore: testing and documentation.

6.1 Objectives

The challenge presented to the document generation component of ADLT is to render the formal assertions of an ADL specification into prose documents. Specifications written in ADL are first order logic assertions on ADL data objects - they are comprehensible mainly to a technical audience. The documents that ADLT generates should be useful to a broader audience.

While translation between natural languages is a task made difficult by context-dependent semantics and the consequent reliance on a knowledge-base and dictionaries for semantic disambiguation, ADL semantics are well-defined and context-free. Translation of ADL assertions into natural language is thus a much simpler task.

ADL is a specification language. It is used to describe the behavior of software systems implemented in some target programming

language. The three elements of the ADL assertions that undergo translation are ADL-defined operators, user-defined functions, and data objects.

ADL semantic specifications are made up of expressions, which in turn are made up of operators, data objects, and functions. The semantics of ADL operators are fixed: ADL users cannot rename or redefine the semantics of operators. ADL can be used to define data objects of various types. These data objects may be represented by user-defined identifiers, or by constant expressions. ADL also allows the user to define functions, whose semantics are defined by the user.

Converting ADL assertions to prose requires translating the semantics of each element into a natural language phrase. Our strategy is to combine those phrases in a context-independent manner; this is the principle simplification allowed by the simple structure of the ADL source of the translation. The three element classes described above require slightly different translation strategies.

The unambiguous semantics of ADL-defined operators makes them readily expressible in natural language. Descriptive prose can be substituted directly for each operator. That prose will generally be a verb phrase.

There is generally meaning associated with ADL data objects and functions that is not available from an ADL specification. The meaning intended by an identifier in ADL, the semantics of that data object, may or may not be reflected by the type or the name of the identifier. In translation, the meaning of an identifier is more likely the interesting aspect, that is to say, that the fact that an identifier refers to the balance of a bank account is the idea that must be captured in translation, not the fact that it is an integer. Like most programming languages, ADL relies upon the choice of meaningful names for data objects and functions in order to convey the intent of the specification writer, but there is always a trade-off between descriptiveness and convenience in the choice of a name. Therefore, the default translation of an

identifier should be its name, but it should be possible to substitute a more meaningful phrase.

In the case of data objects, noun phrases are used to convey meaning in translation. Functions are more complex. They are combinations of verb phrases for the function, describing what it does, and noun phrases describing how the function parameters participate in the actions of the function.

ADL assertions comprise combinations of data objects and operators, a mix of elements that are easily translated to prose and elements whose meanings cannot be found in an ADL specification. Useful translations of assertions almost always require some sort of exposition on an object or function in translation to prose.

There is a clear need for the expression of the intended meaning behind ADL objects and user-defined operators in order to produce good natural-language translation.

Summarizing, the problem facing the ADLT translation system is that of transforming the formal specifications of ADL into prose. The translation process requires information not necessarily available from the specification in order to produce useful prose. Furthermore, the system must be sufficiently flexible to produce prose for any locale.

6.2 Design Decisions

This section describes the design decisions made to address the natural-language translation challenges outlined in the previous section.

ADL partitions the task of natural-language translation into three categories:

- ADL-defined operators
- function identifiers
- data object identifiers

Because the semantics of ADL operators are well-defined, translation to a target natural language becomes the task of mapping an operator onto a prose expression of the operator's semantics.

The ADL semantics defined for objects are restricted to type information, which is of limited value in a natural-language translation. In fact, type information is a programming language detail that natural-language documentation seeks to abstract.

We designed ADLT to consult the Natural Language Dictionary (NLD) for the description of identifiers in an ADL specification. The NLD is the place to record the mapping of object identifiers and function identifiers into natural-language descriptions of those identifiers. It is a mechanism for explicating the full meaning of data objects and functions.

ADL objects and functions have scope defined by their declaration and use in an ADL specification. The translation system must observe the scope of a particular identifier and apply an NLD translation appropriate to that scope, and the NLD must be able to express the mapping of an identifier in the context of its ADL scope.

The syntax of the NLD is designed to be simple: expressions relating identifiers to their prose descriptions. There are no semantic constraints or grammatical constraints on the descriptions. That choice makes the NLD very easy to write. However, the unconstrained nature of the NLD creates problems for the natural-language translation process. How can the system ensure that NLD prose of arbitrary complexity is grammatically correct, much less sensible?

This introduced complexity is addressed by the current ADLT in a very direct and simple way. We use the three classes of mapping, ADL-defined operator, function and object, in a context-free text substitution process. Comprehension of user-supplied NLD prose by the translation system is thus obviated. This choice means that there is no mechanism to control the introduction of senseless mappings in the NLD. All NLD mappings that pass syntactic analysis will be propagated to the natural-language documents faithfully. NLD authors must craft their mappings with care.

Separation of components by purpose is a guiding principal during this phase of our research. ADL specifications describe the expected behavior of system under consideration, but are distinct from TDD descriptions of the test input data. Both ADL and TDD are separate from system implementation source. Likewise, the NLD is a separate entity that serves to clarify the translation of ADL assertions. As such, the NLD is not a required input. ADLT will produce documents containing default translations in the absence of an NLD. Adding a well-crafted NLD, however, results in better, more descriptive documents.

Another guiding principal of the ADL project is that partial specification must be possible. A system can be minimally specified at first, then, as development progresses, specifications can be extended to render better tests and better documents. In particular, ADLT may be run with no NLD. If the generated documents are found to be inadequate, an NLD can be introduced and ADLT can be rerun to produce better documents.

We chose Prolog as the translation engine because Prolog offers a concise expression of translation rules. Prolog rules represent a convenient localization quantity: each locale has its own set of Prolog rules describing natural-language translation for that locale. Moreover, Prolog offers rapid development and the power and flexibility to perform more complex translations with little perturbation to other components of ADLT as the document generation component evolves.

The C++ component of ADLT manages the user input, parses the ADL, TDD and NLD inputs and generates the test program. The C++ component acts as the coordinator of services that ADLT provides. It runs the Prolog engine as a child process and opens a UNIX pipe to the Prolog engine. Parent and child communicate bidirectionally over the pipe.

Having chosen this structure, we were faced with a choice of format for the ADL assertions presented to the Prolog engine. Because such a choice minimizes the amount of work done in

the C++ code and because Prolog is able to parse infix operator expressions, we chose to present ADL expressions directly as Prolog terms. However, ADL and Prolog have incompatible rules regarding the characters in an identifier. This was addressed by placing all ADL identifiers (and operators) in single quotes for the Prolog engine. In this form, ADL expressions comprise Prolog atoms and operators and are thus compatible with the Prolog parser.

The entire system, including the document generation facility, is to be internationalized early in its development. The first Japanese localization was completed in June of 1995, and augmented for Release 1.0.

Localization of the ADLT C++ code is performed in a manner compatible with X/Open specifications [3]. It requires separate messaging files for each supported locale. Similarly, the translation subsystem requires that a separate set of Prolog rules be written for each locale to describe the translation of ADL assertions into prose for that locale.

The ADLT document generator uses run-time LOCALE [3] environment information to select the Prolog rule files used to produce documents in a particular target language. Changing the target language of the documents is as simple as setting new LOCALE environment information and re-running ADLT.

6.3 Analysis

This section examines in greater detail the process of translating ADL specifications into prose.

6.3.1 The Translation Process

The principal task of the translation system is to take an ADL assertion, which might look like:

```
(a == b) --> (c == d)
```

and translate it into prose like:

```
if a is equal to b then c is equal
to d
```

Each locale has associated with it a static set of Prolog rules that describe translation for that locale. These are the core rules. For example, core rules for English translation define the translation of ADL operators into English prose. Core rules are sufficient to translate assertions in the absence of NLD input. ADLT is designed to look for additional translation information in an NLD file. That information is used to augment the default rules for that locale.

An example of an NLD file which might be used with the ADL file containing the above implication expression follows:

```
a = "the first leg of a triangle",
b = "the second leg of a triangle",
c = "the angle between the first leg
and the third leg",
d = "the angle between the second
leg and the third leg"
```

The use of this NLD would result in the following translation:

```
if the first leg of a triangle is
equal to the second leg of a
triangle then the angle between the
first leg and the third leg is equal
to the angle between the second leg
and the third leg.
```

That translation occurs as a result of the two cooperating components of the translation process. The C++ component of ADLT initiates a request for translations by sending a question to the Prolog engine. The Prolog engine answers the translation request with the translated text. The C++ component then packages the resulting translation in documents it is called upon to produce.

The following example is offered to illustrate translation of a simple but syntactically complete ADL module. Figure 4 represents an ADL module that makes assertions on some simple C functions that operate on triangles. One type of document generated by ADLT is the Natural Language Specification (NLS). Figure 5 shows the NLS output generated from

the ADL input in Figure 4 without an NLD. Figure 6 represents a possible NLD for the ADL in Figure 4. Figure 7 shows the NLS that ADLT produces from the ADL of Figure 4 with the NLD of Figure 6 as input. The generated NLS' in these examples have been abbreviated because of space limitations.

```

module triangle {
  boolean
  isosceles(int a, int b)
  semantics{
    (a == b) <-> return == TRUE
  };

  boolean
  equilateral(int a, int b, int c)
  semantics{
    isosceles(a,b)&&isosceles(b,c)
    <-> return == TRUE
  };
}

```

FIGURE 4 — Triangle ADL

SYNOPSIS:

```

boolean
isosceles(int a, int b)

```

GENERAL BEHAVIOR:

a is equal to b if and only if return is equal to TRUE.

SYNOPSIS:

```

equilateral(int a, int b, int c)

```

GENERAL BEHAVIOR:

The value returned by isosceles(a,b) and the value returned by isosceles(b,c) if and only if return is equal to TRUE.

FIGURE 5 — NLS without NLD

```

equilateral(p1,p2,p3){
  p1 = "the first leg of the
  triangle",
  p2 = "the second leg of the
  triangle",
  p3 = "the third leg of the
  triangle",
  $DESCRIPTION = "Verify that the
  triangle specified by its legs is an
  equilateral triangle."
}

Isosceles(p1,p2) = "test whether"
p1 "is equal to" p2{
  p1 = "leg one",
  p2 = "leg two",
  $DESCRIPTION = "Verify that the two
  legs given are equal."
}

```

FIGURE 6 — Triangle NLD

SYNOPSIS:

```

boolean
isosceles(int a, int b)

```

DESCRIPTION:

Verify that the two legs given are equal.

GENERAL BEHAVIOR:

Leg one is equal to leg two if and only if return is equal to TRUE.

SYNOPSIS:

```

equilateral(int a, int b, int c)

```

DESCRIPTION:

Verify that the triangle specified by its legs is an equilateral triangle.

GENERAL BEHAVIOR:

Test whether the first leg of the triangle is equal to the second leg of the triangle and test whether the second leg of the triangle is equal to the third leg of the triangle if and only if return is equal to TRUE.

FIGURE 7 — NLS with NLD

6.3.2 Some Prolog Details

When ADLT gets a request for documents it runs the Prolog engine. The Prolog engine is initialized with the core rules that produce the default translations of ADL assertions, unaided by NLD input. ADLT can produce documents

for each locale for which such a core set of rules is available. Core rules are stored as files in a hierarchy structured by locale. ADLT selects among the core rules based on its run-time locale information. The initialized Prolog engine runs in query mode connected to ADLT via a UNIX pipe.

ADLT makes translation requests in the form of Prolog questions. Those questions are formed with a term defined in the core rules, `xlate/2`. An ADL expression like:

```
a == b
```

is sent to the Prolog engine on the pipe as the following Prolog question:

```
xlate('a' '==' 'b', ctxt).
```

The *ctxt* term is a representation of the ADL scope of the ADL expression being translated, which is the first term of `xlate/2`.

The following predicates are from the core rules for the English locale:

```
natural(M '==' N,Ctxt,[En,is,equal,
to,En]) :-
    natural(M,Ctxt,Em),
    natural(N,Ctxt,En).
...
xlate(A, Ctxt) :-
    natural(A,Ctxt,Et),
    flatten(Et,S),
    unlist_write(S),
```

ADLT, in order to find the translation of `a == b`, writes `xlate('a' '==' 'b', ctxt)` on the pipe for the Prolog engine to read. Prolog reads that question and answers it by writing the translation “a is equal to b” on its end of the pipe. ADLT then reads that translation from the pipe and incorporates it in the current document.

When an NLD input is available, the Prolog engine is started as a child process and initialized just as when no NLD is present. ADLT then transforms NLD dictionary entries into Prolog facts, which are added to the core rules with the `asserta/1` predicate. As with other communications between ADLT and the

Prolog engine, those facts are sent over the pipe to the Prolog engine.

Recall the three classes of substitution translations performed by ADLT were operator, object and function. The NLD does not offer the ability to change the definitions of ADL operators. Thus, there are two classes of substitution affected by the NLD: object substitutions and function reference substitutions. Object substitutions are complicated by namespace rules of ADL for compound data types (structures, unions, enums) but they are nonetheless illustrated by the work ADLT performs on a simple object. An example of simple object substitution follows.

An NLD entry like:

```
a = "the phrase"
```

is transformed by ADLT into:

```
asserta(map('a',ctxt,'the phrase'))
```

where *ctxt* is a unique atom for the ADL scope in which the substitution should be performed. `asserta/1` is used to add the `map/3` fact to the core rules. Note that the `assert` occurs before any requests for translation are sent to the Prolog engine. Interpreting the NLD and sending the resultant facts to the Prolog engine are initialization steps the documentation system must take before any translations are rendered.

Object translation is defined within the context of ADL scopes. Each of the translation requests has a reference to the ADL scope over which it is valid. ADLT must assert some facts about the nesting structure of the ADL scopes. They are of the form:

```
asserta(nested(b,a)).
```

Where *b* is a scope that is nested within the scope *a*.

ADL is structured by modules. Modules can import other modules. Importation determines the visibility of identifiers. Facts relating to the module importation structure must also be asserted on the core rules. They are of the form:

```
asserta(imports(b,a)).
```

Where *b* is a module that imports the module *a*.

The following Prolog fragment is from the core rules for the English locale:

```
natural(X,Ctxt,X1) :-
    atomic(X),
    map(X,Ctxt,X1).

...

map(X,Ctxt,X1) :-
    nested(Ctxt,CE),
    map(X,CE,X1),
    not X = X1.

map(X,Ctxt,X1) :-
    imports(Ctxt,Imports),
    maplist(X,Imports,X1).

map(X,_ ,X).

maplist(X,[H|T],X1):-
    (map(X,H,X2),
     X = X2,
     maplist(X,T,X1));
    map(X,H,X1).
```

During translation, the predicate `map/3` is applied to the atomic nodes of ADL expression trees. Those facts `map` identifiers to their translations for the appropriate scope and module visibility. Where no applicable `map/3` facts have been asserted, the rule `map(X,_ ,X)` performs the identity translation.

NLD entries for functions are a bit more complicated. An NLD entry like:

```
a(x,y)="swap" y "for" x {
    x = "param one",
    y = "param two"
}
```

is transformed by ADLT into:

```
asserta(mapfn(a,['swap',2,'for',1]
))
asserta(map('x',ctxt,'param one'))
asserta(map('y',ctxt,'param two'))
```

`mapfn/2` is used in the core rules to match the function (*a* in this case) with its translation. The translation involves interpolating the

translation of the actual parameters of the function reference with the text from the NLD entry for that function, in the order specified by the NLD. The rules represent formal parameters as integers that correspond to the parameter's position in the NLD specification. Note that an ADL function expression is treated as a Prolog term and a function with no parameters in ADL is actually a term with one parameter, a blank in the following fragment from the core rules for English translation:

```
mapfun(X,C,Out) :-
    functor(X,F,A),
    A > 0,
    mapfn(F,M),
    xlatefn(X,C,M,Out).

mapfun(P,_ ,
[the,value,returned,by,P]) :-
    functor(P,_ ,Ar),
    Ar > 0.

mapfun(X,_ ,X):- !.

xlatefn(_ ,C,[],[]).

xlatefn(XX,C,[Mh|Mt],[Oh|Ot]) :-
    ((integer(Mh), arg(Mh,XX,P),
     natural(P,C,Oh));
     (not integer(Mh), map(Mh,C,Oh))),
     xlatefn(XX,C,Mt,Ot).
```

6.4 Lessons Learned

Though it makes no pretense at linguistic sophistication, the substitution-based translation used in ADLT represents a surprisingly useful level of translation service. It establishes a minimum level of translation service upon which future extensions might yield better prose. Even in its current form, the translation system has received positive reviews from early ADLT users.

Not only does the system deliver a useful level of translation, it does so with relatively small core rule files. There are about 370 lines in the core rules for English. About 300 of those must be replaced in order to create rules for other locales. In the case of Japanese localization, that task has been performed by someone with only a rudimentary knowledge of Prolog. Localization of ADLT is thus relatively easy,

even taking into account the automated document generation system.

Typically, ADLT requires little work beyond the creation of an ADL specification to produce accurate documentation. In fact, it requires no work if the default translations are found to be adequate. In the event that the generated prose is found lacking, the work required of the user is the creation of an optional NLD file and possibly some editorial work on the generated documents. System specifications can be structured to reuse NLD files to minimize the number of dictionary entries a user must create.

In common practice, document preparation is a very labor intensive, error-prone and expensive process. Synchronizing documents with specifications and implementations that evolve is a management nightmare. ADLT delivers a document generation system that is capable of producing draft quality documents from the formally specified system. If the system changes, the system documentation changes with it. ADLT-generated draft-quality documents will always be up to date with the system specification.

ADLT includes a Prolog-based document generation system that:

- is easy to localize
- produces draft-quality documents with little user input
- ensures documentation correctness

7. Evaluation of the Research

During the last two years of the project, the project conducted two experiments, applying an early version of the system to testing two very different styles of interfaces. The resulting tests and documentation were made available to the relevant communities. Members of those communities worked closely with the ADL Project to assist us in evaluating the value of the ADL-generated tests.

These experiments were conducted with an early version of the ADLT Translation system, so that the research team would have time to act on the feedback. It should be noted that the

team has now corrected many of the specific faults, and adapted the design to accommodate the desired mode of use. For example, the documentation has been completely re-written since the experiments, the test reporting has been completely revised to be more suitable for POSIX testing, and test suite creation has been greatly improved. Details of the experimental feedback can be found in Section 7.3, "Changes to the Research".

7.1 Common Object Request Broker Test Suite

The Object Management Group has developed a series of specifications relating to the server and client sides of communication with distributed objects. Central to these is the Common Object Request Broker Architecture specification (at the time of the experiment at version 1.2). This document describes the ways in which object interfaces are specified, how clients find and use objects, and (to a lesser extent) how the objects interface with the object system. The ADL Project Steering Board felt that this specification was a good candidate for testing because it was an emerging standard, require the use of external objects in the tests, and there were implementations available against which the tests could be applied.

The development team for this prototype was Applied Testing and Technology of Los Gatos, California. ApTest is an organization with substantial experience in conformance test development, and was involved in the original requirements generation for the ADL Project. This section of the paper is a summary of their report at the end of the experiment[4].

ADL as a Test Development Tool

In ApTest's experience with ADL its performance in this area was not in itself an improvement.

Developer productivity in producing the CORBA Test Suite was lower than would be expected from manual development. In part this is due to a very steep learning curve for ADL. However, after this initial learning phase was overcome, productivity

remained at below that for manual techniques.

It should be noted that very experienced software developers were involved in this project, more so than has traditionally been considered necessary for test development. ApTest expects an additional dip in productivity in using ADL with less seasoned personnel.

In measuring productivity ApTest considered the time required to complete test development for an interface manually and with ADL. This does not take into account the number of tests produced. This is much higher in the ADL case because of the technique of running a large number of different data points through each test method.

This increase in tests produced is certainly a benefit of ADL in terms of system stress and performance testing. While the large number of times ADL invokes an interface-under-test is also in some sense an increase in test coverage, ApTest found this was not necessarily a meaningful increase. It would only be meaningful if the test data set is carefully constructed.

In the CORBA test suite ApTest did not see an increase in the ability of the ADL generated tests to detect errors versus manually generated ones, despite an orders of magnitude increase in the number of tests. The point here is one made in the ADL documentation: test coverage with ADL is a matter of skilled design, not a necessarily a consequence of the technology. Thus while ADL does generate a lot of test executions, an increase in meaningful test coverage is not an inherent by-product.

In terms of test quality, ApTest again did not see a significant gain. CORBA Test Suite error rates were comparable to what is seen with test suites in general.

In the abstract, higher quality might be achieved through the generation of less

code by manual, and thus error prone, means. However, ApTest found the amount of code that had to be written using ADL to be only marginally less than what would be expected without it. Though ADL automates many mundane tasks in sequencing and executing tests, the body of the tests themselves must still be manually coded. As these aspects of the tests are much more error prone than those ADL automates, and by far the bulk of code necessary in ADL's early releases, ADL's impact on initial test error rates is limited.

In addition, ApTest found added complexity in debugging tests generated with ADL due to the nature of the code generated and the complex flow of control involved. This has a negative impact on quality. It is also an additional consideration in the higher level of experience necessary for a programmer working with ADL as discussed above.

Overall when compared with traditional development methods ApTest found ADL-based test generation to exhibit:

- roughly comparable test error rates
- roughly comparable meaningful test coverage
- lower programmer productivity, and
- higher requirements for the experience level of developers.

It should be noted that while ApTest believes improvements can be made in ADL that bear on these issues, they do not consider expectations for dramatic improvements in terms of programmer productivity, initial quality or test coverage through use of automated test development methods to be realistic.

Given the complexity of the task of software development and its historic resistance to automation (e.g. Artificial Intelligence research in Automatic Programming), a reasonable goal for ADL is that test development can be done within the envelope of performance that can be

expected with manual means. As initial development is a relatively small factor in the life-cycle cost of test suites, this limitation of the paradigm can be handsomely compensated for through improving specification quality and test suite maintenance.

In sum, ApTest feels that ADL entails a net increase in initial test development cost as compared to traditional methods but that this increase is sufficiently small as to be mitigated by its benefits in other areas.

ADL as a Specification Tool

The lack of compatibility between implementations based on written specifications is most often traceable to the very poor quality of the specifications themselves. These documents are often vague, imprecise and incomplete, leading to varied and incompatible implementations. From a testing perspective these same issues limit the ability to verify implementation compliance.

Improving the quality of specifications by expressing them in a formal language is the area where ADL can realistically be expected to make to the greatest contribution to the community, and was in fact the most significant positive in ApTest's experience with it.

In capturing the CORBA 1.2 specification in ADL many (over 100) areas of imprecision and ambiguity were found in the original specification. As the specification was considered complete before this project, this demonstrates a clear benefit of ADL to specification writers and test developers, one which it can only be wished was available when the original CORBA specification was drafted!

In terms of developing test specifications, ADL fell short of expectations. The then current ADL language primitives were limited as compared to the semantics of most programming languages. As a result

many aspects of tests which should have been exposed in the test specification had to be coded inside ADL provide or auxiliary functions, and are thus were visible within the specification. This issue was noted by many of reviewers of the CORBA Test Suite specification who complained that the test methods used were opaque from reading the specification.

In sum, ApTest feels ADL is very valuable as a specification development tool, though in need of improvement in natural language output and the richness of the ADL language.

ADL as a Maintenance Tool

As on-going maintenance is the largest cost-driver in software costs, and quick turn around for new versions of tests in response to specification changes is a strong requirement, ADL's performance in this area is an important consideration.

Conclusions regarding maintenance are preliminary, as developing the CORBA Test Suite has not provided hands on experience with test suite maintenance using ADL. However the process of developing it can provide insights into how this would work.

ADL has promise in this area through allowing changes to tests solely through modifications of the ADL version of the specification and facilitating easy reuse of existing code.

In practice, realizing these benefits was made difficult by the limitations of the ADL language discussed above. Since ADL specifications must currently be at a very high level, some changes will not automatically update the test suite appropriately and may require changes to or the development of new auxiliary or provide functions.

As well, the requirements for sophisticated programmers in order to utilize ADL discussed above must be taken into consideration. ApTest expects higher

experience levels to be required for ADL test suite support than have been required in traditional situations.

In sum, ADL shows promise in this area, however extensions to the ADL language will be required before meaningful benefits can be expected. Pending the availability of empirical data, ApTest also expects the level of expertise required for ADL maintenance to be a mitigating factor versus gains in productivity.

Product Review

ApTest found the current ADL product to correspond well to its specification.

In addition to those functional issues discussed above, two important deficiencies in the then current functionality were encountered:

- Applicability of the product to building large test suites is limited. Such suites have more complex source tree structures and build procedures than what ADL accommodated.
- ADL's provision for generating test suites under the Test Environment Toolkit (TET) is inadequate. Though some improvements have been made in recent ADL releases it was still not possible to generate the test output or test results expected of TET-based test suites.

User Interface

The user interface to ADL is adequate. Limited provisions for making the interface user friendly or facilitating productivity in a development environment have been made. The lack of a GUI-based front end for test developers and the large number of separate files which have to be manipulated to create ADL test suites contribute to the high level of expertise necessary to use it.

Portability

ADL portability was a problem when the CORBA Test Suite project was initiated but was substantially improved in subsequent releases.

By the end of the experiment the only outstanding portability problems are in the Prolog interpreter. This is incorporated into the distributed package. This part of ADL continued to be difficult to port and also exhibited problems in 64-bit environments.

Quality

The development of the CORBA Test Suite uncovered a number of errors in the implementation of ADL. The majority of these errors were corrected during the experiment, and additional regression testing was added to the release process resulting in acceptable quality of the last release reviewed (ADL 0.7).

Performance

Performance of ADL has declined somewhat over the term of the CORBA Test Suite project. Though it remains at an acceptable level for a test suite of this size, performance with a large scale test suite has yet to be evaluated.

Documentation

The documentation of ADL was very poor - the User and Programmer Manuals were effectively unusable. This is partly due to their not having been maintained as ADL evolved and partly to their having been mistargeted in the first place and thus of limited benefit to test developers.

Conclusions and Recommendations

In sum the experience of the CORBA Test Suite project is that ADL offers a valuable improvement in the production of test suites by facilitating much greater accuracy and completeness in interface specifications.

In terms of test development, ADL adds some cost in the initial development cycle. While there is potential for benefits in the maintenance cycle it is not yet clear how well these can be realized.

As a product ADL is reasonably complete and functional and offers acceptable performance.

In terms of the evolution, ApTest offered a number of recommendations for addressing the then current deficiencies in ADL and also for increasing its ability to improve the testing process.

- The ADL language should be extended to increase the degree to which test methods which can be defined in ADL rather than provide or auxiliary functions.
- The ADL interface to TET should be reworked to allow results to be returned by user written code, journal output to be created by user written code, report results in TET (assertion-by assertion) style rather than ADL (gridpoint-by-gridpoint style), and document assertions in natural language (rather than ADL).
- The interface specifications and test specifications produced by ADL need to be enhanced to be fully in troff format (this is simplistically the case for interface specifications now), allow the programmer to embed troff macros, format complex expressions in a readable fashion, and handle line and page formatting correctly.
- A substantial number of large test suites should be studied and ADL modified to accommodate existing practice in test suite structure and build procedures for such suites.
- The ADL documentation should be rewritten to focus on the experienced test suite developer who will use ADL for verifying real-world specifications.

- The necessity to create separate adl, nld and tdd files should be minimized and ideally eliminated or masked from the user. This complexity reduces productivity.
- The process of debugging ADL generated tests should be investigated with a focus on minimizing difficulty and providing tools that facilitate productivity.
- The specification generation and test development aspects of ADL should be further decoupled so that the ADL's benefits in specification work can be obtained by those who do not wish to use it for test development.
- The Prolog engine should be made reliably portable across Open Systems environments or replaced by something that is.
- The ADL user interface should be supplemented with friendly front end tools that allow less-sophisticated users to more easily utilize ADL.

7.2 Test Environment Toolkit Test Suite

The Test Environment Toolkit (TET) is a collection of tools and interfaces used by test developers. It was chosen for use in this experiment because it has a number of interesting properties that were expected to stretch ADL to its limits:

- TET is used by ADL itself, so the TET under test must NOT be the one that is in use by ADL.
- TET has both commands and interfaces, and both of these must be tested. Since ADL only supports exercising C language interfaces, the testing of the command must be encapsulated.
- TET's API is defined in a programming language-independent manner, and it has several programming language bindings (including C). All of these bindings must be tested, ideally by reusing the

assertion definitions or having a single set of assertion definitions that could be applied to all language bindings.

The TET Test Suite experiment was conducted by Mount Bonnell, Inc. of Austin, Texas. MBI is an organization with substantial conformance testing experience and intimate knowledge of the implementation of TET. Their report on the results of their experiment is summarized in this section[5].

Obtaining and Building the ADL Release

The source code and documentation as well as binaries for some systems are available via ftp. ADL is written largely in C++. Unfortunately the state of standardization and tools in the C++ is somewhat unstable. Fortunately ADL will build with recent versions of GCC. GCC source is available for most Open System platforms, although building it is time consuming.

Once a C++ environment is available, the next step is to build ADL,. Again this is fairly straight forward with the exception of the Prolog interpreter. The only supported configuration files shipped with Prolog were for SunOS 4.1 and Solaris 2.X. Other platforms might require some debugging.

Experiences Building ADL

During this experiment the ADL releases (0.5 and 0.6) were built on 3 different platforms:

- SunOS 4.1
- Linux 1.1.47
- NetBSD 1.0

The SunOS build went very well. At first there was some confusion about which version of the *make* utility to use. Many people using GCC tend to also use other GNU tools. Some earlier versions of ADL would only build with Sun's *make*. This problem was addressed in later versions.

The Linux build only presented a few easily fixed compile problems.

Unfortunately, the ADL translator core dumped when run. It appeared that there was a malloc and free problem in this version of Linux and after a few days of debugging the frees were commented out and the code worked.

Attempting to debug the ADL source code revealed an important issue. Some of the C++ code that makes up ADL is generated using a parser-generation tool called *pcct*. In these early releases the source used by *pcct* was not part of the distribution, so debugging required working on the generated code. Working on generated code is notoriously difficult, and doing it in ADL was no exception. This was addressed in subsequent releases - the sources used by *pcct* to generate the portions of ADL are now included in the distribution.

The NetBSD build also went very well except for building the Prolog interpreter. Fortunately, NetBSD is binary compatible with SunOS 4.1 and it was possible to use the distributed binaries on this platform. This worked mostly correctly on the NetBSD release.

All in all ADL appears to be very portable for a C++ program (assuming you have a robust C++ environment). The Prolog part is designed to be portable and as more system configuration files are added it will become easier to build. The one concern in this area is that some organizations may be nervous using a "free" test generator built with a "free" compiler. There must be a commitment to further support of the tool if it is to be successful.

Assertion Writing Using ADL

The first step to creating a test with ADL is creating the assertions. The team started by creating a set of natural language assertions from the TET documentation. The ADL interface was then written from these. The team felt that this is how ADL will be used to write tests for a specification that is already in use. One of the features of ADL is that it can create the natural language

specification directly from the ADL interface specification. Since there was already a specification the team did not use this feature. They did, however, use the generated test specification as a sanity check for the formal assertions against the human-generated ones. This was extremely useful.

The temptation in assertion writing is to just jump right in and do it. Unfortunately this will probably lead to rewriting many ADL interface specification files. The problem here is that in order to get many of the benefits of ADL you need to reuse your auxiliary functions. In order to do this you need to have an architecture in mind for the specification that can frequently reuse the same auxiliary functions. Without a clear architecture in mind, the user may find it difficult to get the ADL interface specification right the first time through. This becomes an iterative process that eventually results in a good set of reusable auxiliary functions, but it is time consuming.

Another problem encountered while writing assertions was the C-like syntax of ADL. In the evaluated versions of ADL, the language did not yet permit the use of all the C languages constructs. Since ADL is so C-like, this was very confusing. As a result of this input, ADL was extended to accept most C language constructs with the same semantics they have in C. This change should help alleviate the problem.

Creating Test Data

In order to create a working test the user creates a TDD file. This file contains the data points and test directives. The user must also write the provide and auxiliary functions. The team chose to create a TDD file with a single data point and use that while developing the auxiliary functions.

Creating the data files and the provide functions was the most convenient aspect of ADL. The ability to import (now called “use”) other TDD files enables the test

developer to define widely used test objects and incorporate them into a number of tests very easily. This feature saved a considerable amount of time in the TET experiment. Further, the ability to change the data set without modifying the test case is a significant advantage of ADL. It allows the user to create a test suite and then easily come back and make the tests more thorough by adding data points.

The one problem area here is there was no way for a provide function to indicate an error. This ability was added for the 0.9 release, and helped both provide and auxiliary functions become more robust.

Developing Auxiliary Functions

The writing of the auxiliary functions did not go nearly as well as the TDD/provide function writing in this experiment. Ideally auxiliary functions should be written once and used in multiple assertions testing multiple interfaces. Unfortunately, the nature of TET make it very difficult to craft these “generic” auxiliary functions. This is partly due to the fact that the TET tests are run in a subprocess and the auxiliary functions must use inter-process communication facilities in order to find out the state of the subprocess. This is a much more complicated model than most tests would use (and one of the reasons TET was chosen as a good experiment). It did, however, exaggerate a weakness in ADL – that of writing generic auxiliary functions. Without a comprehensive test suite architecture, it is nearly impossible to craft complicated reusable auxiliary functions.

Testing and Maintenance

The challenge with any tool that automates a process is in gaining the trust of the users of that tool. In the case of test suites, the users frequently feel the need to inspect the test code to ensure that it is working right, since they firmly believe that their implementation could not possibly be at fault. Inspecting the code generated by

ADL is nearly impossible. It is convoluted, and the generated identifier names are such that they are very hard to follow. While it was never the intent of the ADL Project team that the generated code be inspected, this does raise an interesting point. How can ADL gain the confidence of the test users?

Clearly, the answer is that it cannot do this over night. Test users must be able to inspect the code. However, just as a gcc user trusts it to generate correct assembly language, a test user also need to trust ADL to assemble the pieces of an interface test correctly. Those pieces must be available to the test user for inspection, to be sure. And users can inspect the generated source code if they like. But in the end, ADL, like any compiler must be trusted by its users.

7.3 Changes to the Research

From the experiment results reported above, it became clear that a variety of enhancements were required to the basic research. Most of those enhancements are now complete. Some of the most significant changes made as a result of the experiments are described in this list.

- The ADL language was extended to accommodate most C Language constructs. These extensions have greatly improved the ability of ADL users to fully express the behavior of interfaces.
- The syntax and semantics of the Natural Language Dictionary have been enhanced to make resulting translations more expressive.
- It is now possible to include NLD information directly in the ADL Interface Specification instead of in a separate file.
- The NLD translation engine has been changed so that the resulting output is structured more like a traditional manual page and is more readable.

- ADL has been extended overall to accommodate the construction and maintenance of large test suites. This includes the ability to organize a test suite hierarchically, to use libraries of auxiliary functions more readily, and to modify the behavior of the generated Makefiles without actually editing the makefiles themselves.
- The reporting mechanisms have been significantly enhanced. As a result, they are now able to generate reports that more closely resemble the way in which traditional assertion oriented testing is done. They also interact better with the Test Environment Toolkit. Finally, they can report a breakdown of assertion results so that it is possible to easily discern what parts of an assertion caused a failure.
- The Programmer and User documentation has been completely rewritten — targeted at experienced programmers rather than novices.

8. Availability

The ADL 1.0 implementation and documentation are freely available under a Copyright and grant of rights very much like that of the X Window System. The Copyright and grant of rights reads:

COPYRIGHT AND LICENSE NOTICE

Copyright (c) 1994-1996 Sun Microsystems Inc.

Copyright (c) 1994-1996 Information-technology Promotion Agency, Japan

This technology has been developed as part of a collaborative project among the Information-technology Promotion Agency, Japan (IPA), X/Open Company Ltd. and Sun Microsystems Laboratories.

Permission to use, copy, modify and distribute this software and documentation for any purpose and without fee is hereby granted in perpetuity, provided that this COPYRIGHT AND LICENSE NOTICE appears in its

entirety in all copies of the software and supporting documentation. Certain ideas and concepts contained in the software are protected by pending patents of Sun Microsystems, Inc. Sun hereby grants a limited license to use these patents, if any issued, only in this implementation of the software and documentation and in derivatives thereof prepared in accordance with the permission granted herein.

The names X/Open, Sun Microsystems, Inc. and Information-technology Promotion Agency, Japan (IPA) shall not be used in advertising or publicity pertaining to distribution of the software and documentation without specific, written prior permission.

ANY USE OF THE SOFTWARE AND DOCUMENTATION SHALL BE GOVERNED BY CALIFORNIA LAW. X/OPEN, SUN MICROSYSTEMS, INC. AND IPA MAKE NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE SOFTWARE OR DOCUMENTATION FOR ANY PURPOSE. THEY ARE PROVIDED "AS IS" WITHOUT EXPRESS OR IMPLIED WARRANTY OF ANY KIND. X/OPEN SUN MICROSYSTEMS, INC. AND IPA SEVERALLY AND INDIVIDUALLY DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE AND DOCUMENTATION, INCLUDING THE WARRANTIES OF MERCHANTABILITY, DESIGN, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL X/OPEN, SUN MICROSYSTEMS, INC. OR IPA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA, OR PROFITS, WHETHER IN ACTION ARISING OUT OF CONTRACT, NEGLIGENCE, PRODUCT LIABILITY, OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE OR DOCUMENTATION.

The implementation and documentation are available in a variety of formats and built for a variety of platforms. The ADL Project repository on the Internet is at <ftp.uu.net/vendor/adl/release>.

9. Conclusions

The ADL Project has been, and continues to be, a success. When the ADL Project team started this work, it was expected to realize a certain set of objectives. These objectives have been largely realized. The research results have been promising enough that X/Open and its members and Sun Microsystems Laboratories expect to continue evolving the basic research into the foreseeable future. While some of the initial hypotheses associated with the project have not proven correct, the overall result has been that a useful tool has been developed and presented to the industry for use.

10. Future Plans

Even though the initial major phase of the research is now complete, the ADL Project team feels that enough positive results have been achieved to warrant continued research and development in this area. In fact, the team is planning a variety of interesting projects based upon this research. Several of these are described in this section.

10.1 Deployment Plans

The team is working to encourage the broad deployment of ADL. The first tangible results of this work are the following large testing projects:

10.1.1 CORBA 2 Test Suite

The Object Management Group has recently completed work on revision 2 of their CORBA specification. This revision addresses many of the problems identified through the ADL Project's experimental application to CORBA 1.2. It also contains language bindings for C++ and Smalltalk, a definition of interoperability requirements, and some needed enhancements to the C Language Binding and the Interface Definition Language description.

X/Open and the Object Management Group are working together to develop tests for CORBA

2. These tests will be based upon the CORBA 1.2 testing work done last year. This project has already started with an initial testability analysis of the new CORBA 2 specification. Test Suite development is expected to start before the end of the year.

10.1.2 Common Desktop Environment Test Suite

Another important emerging standard is the Common Desktop Environment. This collection of specifications, published by X/Open, defines an enhanced graphical user interface based upon the X Window System and Motif.

The testing challenges presented by these specifications are significant. The specifications are not as tightly written as other, more mature standards. As a result, testing these specifications via ADL is expected to identify a number of ambiguities even as the implementations of the specifications are being completed. This should help speed the availability of consistent conforming products from a variety of Open Systems vendors.

X/Open is just now starting the process of developing the ADL interface specifications for CDE. Test suite development is expected to start in earnest by year end, with results in the second quarter of 1996.

10.1.3 X/Open Specifications

X/Open is the Open System industry's principle source of specifications and conformance test. As such, it incurs significant costs in the development of these specifications and tests, as well as in the maintenance of both. X/Open views the ADL research project as a potential way to reduce these costs and at the same time improve the quality of the specifications and the tests.

In order to determine the extent of these benefits, X/Open will start using ADL to develop complete specifications and their associated tests early next year. While the exact specification against which ADL will be applied has not yet been determined, X/Open has a general plan for its application:

- Train X/Open work group participants in ADL early in 1996.
- Have that work group develop their specification in ADL, including the production of the natural language dictionary.
- Work toward a mature draft of the specification in ADL by the third quarter.
- Start development of a test suite using the ADL interface specification in the fourth quarter, even as the base specification is being completed.
- Have input from the test developer assist in completing the specification.
- Publish a specification at the beginning of 1997.
- Have an ADL generated test suite available against the specification shortly thereafter.

Based upon the experience gained by this experiment, X/Open plans to start full deployment of ADL within its community at the beginning of 1997.

10.2 Enhancement Plans

Even as ADL Release 1.0 is being deployed, the research into extending and enhancing ADL will continue. Some of the expected research areas are described in this section.

10.2.1 ADL for C++

C++ is an emerging standard in the computing community. It is vitally important that ADL be able to be applied to the specification and testing of C++ interfaces and objects. X/Open and Sun Microsystems Labs are currently exploring the possibility of doing research in this space. While these plans are not finalized, we hope to begin research this calendar year.

10.2.2 ADL for IDL

While C++ is emerging as the programming language of choice for application developers, OMG's Interface Definition Language (IDL) is becoming the *de facto* standard for the

deployment of objects and object services. Since the ADL Project team believes that object-oriented application development will be commonplace in the near future, we are investigating the possibility of producing direct support for IDL within ADL. This project is not expected to start until late 1996, with results emerging in the second quarter of 1997.

10.2.3 ADL for Java

Sun Microsystems' Java Programming Language has become the most talked about innovation this decade. The ADL team believes that the Java community would greatly benefit from testing support via ADL. Consequently, the team is investigating supporting a Java binding for ADL in the coming two years.

10.2.4 ADL Localization Packages

One of the most significant advantages of ADL is its ability to produce quality specifications in any natural language. The support of these languages is not, however, automatic. A localization package, or set of files that describe the rules and default translations, for each language must be developed.

The ADL Project team is still considering which languages should be done, but we expect to develop packages for German and French at the very least. This work should be done in the second quarter of 1996.

10.3 Follow-on Research

Our intention is to make ADL the foundation for a family of related test and specification tools. In this way we make maximum use of the user's effort in creating the initial specification. By driving a family of tools from the same specification, we obtain several advantages:

- The investment in creating the specification is put to maximum use.
- Consistency is assured; the implementor and the various tests and documents are guaranteed to use the same specification.

- In many cases, the support software used for one test can be re-used in another test framework.

New testing capabilities can be added by both the ADL team (several efforts are underway) and by outside researchers (efforts are underway at the University of California at Irvine, the University of Texas at Austin, Oxford University, and others).

10.3.1 Unit Testing

ADLT is intended as a tool for unit testing. It is aimed specifically at the problem of conformance testing, checking each procedure in an interface separately to ensure that it meets its documented specification.

Each interface must be tested separately because each procedure may be used separately. The goal of unit testing is to test each element of the interface separately, to some standard of coverage that gives reasonable confidence that the element meets its specification.

10.3.2 System Testing

The purpose of integration, or system, testing is to ensure that the elements of an interface work correctly in various combinations. The number of possible combinations is astronomical, too large to attempt anything approaching complete coverage; the test criterion is to check some typical fragments that might be expected to occur in user code.

System testing provides a less exhaustive but more realistic check of an interface. It is not sufficient by itself, because the number of possible combinations makes it impossible to get good coverage; it is necessary as part of a complete testing program, because it may expose bugs that are not revealed by unit testing.

The existing ADLT tool can be extended to a tool for testing code fragments as well as individual procedure invocations. A design and a prototype for this extension have been completed.

10.3.3 Random Testing

Random testing is an alternate strategy for system testing. Rather than pre-selected code fragments, it executes a random sequence of calls from the interface. In previous experiments, we found that a pure random test sequence was usually not very interesting; it ends up exercising trivial error checking code over and over again. Since then, we have used some of the mechanisms from the present ADLT tests to create more interesting random tests. This work has advanced as far as a successful experimental prototype; we plan to continue this promising avenue of research.

10.3.4 Stress Testing

A stress test is one that subjects the software under test to uncommon stress, to determine that it fails gracefully when resources are exhausted. A stress test determines what the resource limits of an implementation are, and often exposes bugs in the implementation, because the error checking and catching code paths are seldom executed in normal circumstances.

An ADLT test is often, by its mechanical nature, a stress test for some aspects of the software; for example, a fault in resource reclamation will often be exposed by an ADLT test that cycles through thousands of instances of a resource. More specific stress testing is a possible topic for future investigation.

10.3.5 Interference Testing

Another task is interference testing, to check that simultaneous use of an interface by several clients does not lead to problems.

It is important to make a distinction between simultaneous independent use, in which the actions by one user are not supposed to affect the other user, and unrestricted concurrent use, in which the actions of one user may affect the other. In general, one user can affect another only if there are shared elements between the two users. For example, if two users are accessing different directories of a file system, the system should behave as if each one was the sole client; if they are accessing the same

directory, they should see the result of each others actions as well as their own.

Of the two testing problems, the one of checking for interference between independent clients is much simpler. It can be testing by the present ADLT test. We are investigating possible solutions for the harder problem of concurrent testing.

10.3.6 External Research

In addition to the ADLT team's research efforts mentioned above, there are collaborative research efforts at several universities and companies. We welcome such efforts and will support them to the best of our abilities. Part of the reason that the ADLT code is in the public domain is to enable experimentation into other test strategies.

11. References

- 1 Sriram Sankar and Roger Hayes, Sun Labs; *Specifying and Testing Software Components using ADL*; Sun Microsystems, 1993
- 2 Matt Evans, Sun Labs and Shane P. McCarron, X/Open; *Assertion Definition Language Translation System*, SRI Quality Week; May 1994
- 3 X/Open Company Limited, *Internationalisation Guide*
- 4 Andrew Silverman, *A Review of ADL*, Applied Testing and Technology, 1995
- 5 Barry Books, *ADL Report*, Mount Bonnell, Inc., 1995
- 6 IEEE P1003.3; *Test Methods for Measuring Conformance to POSIX*; Institute of Electrical and Electronics Engineers, Inc., April 17, 1991

12. Papers and Presentations

- 1 Sriram Sankar, Sun Labs; The ISO PIKS Standards Committee; Fall 1994

A presentation on the capabilities of ADL that resulted in the PICS committee

- choosing ADL as its formal specification mechanism.
- 2 Shane McCarron, X/Open; UniForum Monthly; December 1994
An article describing the capabilities of the ADL technology and the scope of the basic research.
 - 3 Matt Evans, James deRaeve, Shane McCarron, Assertion Definition Language Project Progress and Plans, IPA's 13th Annual Symposium, October 1994
An overview of the ADL research and a description of ADL's capabilities.
 - 4 Shane McCarron, X/Open; US National Institute of Standards and Technology Symposium on Automated Testing; May 1995
An presentation on ADL and its progress toward achieving its research goals.
 - 5 Shane McCarron, X/Open; IEEE's POSIX committees; April 1994 and October 1994
A presentation of the ADL project, its requirements, and some of its objectives as they pertained to the POSIX community.
 - 6 Shane McCarron, X/Open; ISO WG15's Rapporteur Group on Conformance Testing; July 1994
A presentation of a brief overview of the ADL project as it relates to the international standards community and its need for testing.
 - 7 Paul Tanner, X/Open; The European Community Manufacturer's Association's DEPLOY project; Fall 1994
A presentation of information about ADL and how ADL could be made to assist with the DEPLOY project - a project for automated testing of application's use of standard interfaces.
 - 8 Shane McCarron, X/Open; AT&T/NCR; Fall 1994
A presentation describing ADL as part of an overall testing framework that AT&T GIS could apply to its interoperability testing problem.
 - 9 Alberto Savoia, Sun Labs; Xerox PARC; Spring 1994
A presentation of information about ADL and how it could assist in testing object-oriented interfaces.
 - 10 Shane McCarron, X/Open; PLATINUM technologies ViaTech Labs; Fall 1994 and January 1996
A presentation of details of the ADL project and an overview of the technology, including how to use ADL to test C++ interfaces.
 - 11 Paul Tanner, X/Open; The UK Defense Research Institute; Fall 1994
A presentation of information about how ADL could help in doing static testing of applications via the DEPLOY project.
 - 12 Shane McCarron, X/Open; X/Open's Verification Strategy Work Group; May 1994
A presentaion of the details of the ADL research project and solicited requirements.
 - 13 James deRaeve, X/Open; X/Open's Technical Managers; June 1995
A presentation of X/Open's strategy for deploying ADL against its specification and testing problems.
 - 14 Shane McCarron, X/Open; X/Open Technical Managers; December 1995
A detailed presentation of ADL and its capabilities, including syntax and conventions, translation capabilities, and deployment strategies for X/Open.