

**ADL Translation System User's Guide:**

*Getting Started With ADLT*

*Document Release 2.0*

*For Use with ADLT System Release 2.0, February 1998*

## **COPYRIGHT AND LICENSE NOTICE**

Copyright © 1997-1998 The Open Group

Copyright © 1994 - 1996 Sun Microsystems Inc.

Copyright © 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,. 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 The Open Group, Sun Microsystems. 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. THE OPEN GROUP, 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.**

## *Trademarks*

Sun™, Sun Microsystems®, the Sun logo®, Solaris®, SunOS™ are trademarks or registered trademarks of Sun Microsystems, Inc.

UNIX® is a registered trademark in the USA and other countries licensed exclusively through X/Open™.

X/Open™ is a trademark of the X/Open Company Limited.



# Contents

---

Preface .....	vii
<i>Part 1 —A First Look at ADLT</i>	
<b>1. Overview .....</b>	<b>3</b>
What is ADLT? .....	3
Generating Tests with ADLT .....	6
Generating Documentation with ADLT .....	10
<b>2. Setting Up the Implementation .....</b>	<b>13</b>
The Default Implementation File Structure .....	13
<b>3. A First Test .....</b>	<b>15</b>
The Java-Language Module .....	15
The ADL File .....	17
Looking at the Natural-Language Specification (NLS) .....	23
Auxiliary Functions .....	24
The TDD File .....	25
Generating Tests .....	28

---

Factories and Relinquish Functions . . . . .	31
Making the Test Program . . . . .	32
Building the Test Program . . . . .	32
Running the Test Program . . . . .	32
<b>4. A First Look at Documentation . . . . .</b>	<b>33</b>
The Role of the Natural Language Dictionary . . . . .	33
<i>Part 2 —Effective Use of ADLT</i>	
<b>5. The Structure of the Test Suite . . . . .</b>	<b>39</b>
Overview of ADLT Files . . . . .	40
Importing and Referencing Specification Files . . . . .	46
Organizing the Test Suite . . . . .	58
<b>6. Using ADLT with TET . . . . .</b>	<b>63</b>
Mappings Between ADLT and TET . . . . .	63
Directory Structure and Required Files . . . . .	65
Environment/Configuration Variables . . . . .	65
Generating Test Programs for TET . . . . .	66
<b>7. Customizing the Test Program . . . . .</b>	<b>69</b>
Glossary . . . . .	73
Index . . . . .	IX-1

## *Preface*

---

ADLT (Assertion Definition Language Translator) is a test compiler. It accepts inputs in the form of specifications; it can generate a test of any procedure which can be called from a variety of languages, including Java, C++, and ANSI C.

The input specifications processed by ADLT are in three parts:

- Specifications of the behavior of the procedure being tested.
- Specifications of the required test data.
- Specifications of the testing process.

In addition, ADLT can produce natural-language translations of the formal specifications. These translations can be produced with no additional input beyond the two listed above. The user can improve the quality of the generated documents by furnishing a natural language dictionary (NLD) to guide the translation process.

The *ADL Translator User's Guide* is designed to give a quick start in the use of ADLT. It gives a concentrated look at the process of generating tests and documentation using ADLT.

### *Who Should Use This Book*

This manual is designed to be used by a senior software engineer who is interested in learning to use ADLT to produce automated tests and documents.

---

The manual assumes that the reader has the following technical background:

- several years of experience writing software using one of the supported languages
- extensive experience working in a UNIX software development environment
- familiarity with testing concepts such as white and black box testing, and equivalence partitioning
- familiarity with the Test Environment Toolkit (although this document provides adequate information to use ADL and TET without such familiarity)

## *How This Book Is Organized*

This manual is organized in two parts, each with several chapters, plus a glossary and five indices, as listed below:

### *Part 1 - A First Look at ADLT*

This part of the document provides an introduction to the ADLT system. It contains the following chapters:

**Chapter 1, “Overview,”** gives a very short summary of the steps in the ADLT testing process.

**Chapter 2, “Setting Up the Implementation,”** briefly describes the default file structure of the implementation to be tested using ADLT.

**Chapter 3, “A First Test,”** demonstrates the use of ADLT to specify and test a very simplified piece of Java code—a sample banking module.

**Chapter 4, “A First Look at Documentation,”** demonstrates the use of ADLT to produce documentation for the simple banking module introduced in Chapter 3.

### *Part 2 - Effective Use of ADLT*

This part of the document provides a more in-depth look at ADLT, offering information necessary to creating real-world test suites. It contains the following chapters:

---

**Chapter 5, “The Structure of the Test Suite,”** shows how to combine the elements of ADLT specifications and code into a test suite.

**Chapter 6, “Using ADLT with TET,”** describes how to execute the ADLT test program using the Test Environment Toolkit (TET) test harness.

**Chapter 7, “Customizing the Test Program,”** lists the ways the user may modify the default behavior of ADLT. It directs the reader to sections of the *ADL Translator Programmer’s Guide*, which gives detailed descriptions of customization features.

*Additional Information and Appendices*

**Glossary** - contains definitions of important ADLT terms.

## *Related Books*

More information about ADLT can be found in the following documents:

- *ADL Language Reference Manual for ANSI C Programmers* - Provides a formal description of the ADL language framework and its specialization as ADL for C.
- *ADL Language Reference Manual for C++ Programmers* - Provides a formal description of the ADL language framework and its specialization as ADL for C++.
- *ADL Language Reference Manual for Java Programmers* - Provides a formal description of the ADL language framework and its specialization as ADL for Java.
- *ADL Language Reference Manual for Java/IDL Programmers* - Provides a formal description of the ADL language framework and its specialization as ADL for Java when used in conjunction with the OMG’s Interface Definition Language.
- *ADLT Design Specification* - Describes the internal and external design of ADLT.

---

## Other Sources of Information

Users should see the file `RELEASE_NOTES`, supplied with the ADLT system distribution, for information about installation, environment variables, internationalization, and details about the implementation status of ADLT features.

## What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Examples
AaBbCc123	The names of commands, files, and directories; on-screen computer output; short code samples	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. system% You have mail. module sample { };
<b>AaBbCc123</b>	What you type, contrasted with on-screen computer output	system% <b>su</b> Password:
<i>AaBbCc123</i>	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
<i>AaBbCc123</i>	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be root to do this.

Syntax samples are enclosed in boxes and may display the following:

AaBbCc123	Reserved words and syntax samples	syntax sample
<i>AaBbCc12</i>	Terms that you expand or instantiate	module <i>module_identifier</i>

---

*Table P-1* Typographic Conventions

<b>Typeface or Symbol</b>	<b>Meaning</b>	<b>Examples</b>
[AaBbCc12]	Optional terms	[options]
AaBbCc12*	Terms that can be repeated as needed	repeat*
"AaBbCc12"	Syntax description characters that are part of the language	"[" bracketed_term "]"



*Part 1 — A First Look at ADLT*

---

---

## Overview



This chapter gives a high-level view of the ADLT system. It describes the steps involved in using ADLT to generate both tests and documentation.

### *What is ADLT?*

ADLT is a system that assists in the functional testing of software components. It is a compiler that generates test programs based upon formal functional and test-data specifications. ADLT-generated test programs call the components under test to determine if they meet their specifications. A component is defined to be any collection of routines callable; for example, a subroutine library or an operating system interface.

In addition to the specification inputs, the ADLT-generated test program depends upon the implementation of the components to be tested (either source or executable code), the ADLT library, and some user-written test support code. The support code is written using a framework generated by ADLT; the framework is derived from the user's input specifications.

ADLT's goal is to automate, as much as possible, current best practices in unit and subsystem testing. It is specifically designed with the following objectives:

- Testing interfaces to independently-invoked functions

ADLT tests functions. It is designed to explore all points in the function's input space that the test designer has specified as important or significant. Each invocation of the function with a set of input parameters constitutes an ADLT "test instance."

- Testing the relationship among functions

It is possible to describe the relationship among the elements of a component (e.g. a subsystem). ADLT will generate tests that exercise the elements of the component across each point in the input space that the test designer has specified.

- Black box testing

While ADLT does provide ways for the test program to have access to the internals of the function under test, ADLT is designed to test the interface to the function in a black-box manner. The goal is to specify the inputs with which the function is invoked, and the state of the system after the function completes execution.

- Functional testing

ADLT evaluates the functionality of the function under test against its specified behavior. ADLT is not specifically designed for stress testing, although the large number of test instances automatically generated by ADLT can place some measure of stress on the function under test.

- Providing a “test compiler”

ADLT provides a method whereby the test designer specifies the intended behavior of the function under test and the characteristics of important data inputs. ADLT’s role is then to compile those engineering notations into source code that can test the function. The ADLT-generated source code is linked with the implementation of the function under test—along with libraries supplied by ADLT, the system, and the user—to create the executable test program.

While ADLT focuses on automated test generation, the system also helps automate other parts of the software development process:

- Documentation

The ADL Translator generates documentation of the software components under test. It produces documents describing the generated tests, as well as natural-language versions of the functional specification.

- Development of new software components

ADLT helps in the implementation of new software by generating header files containing declarations of the elements in the specified component—the constituent functions, variables, constants, and type definitions. Since ADL specifications are not embedded in the components to be tested or documented, ADLT can be used with either new software or existing, already-compiled components.

### What ADLT Is Not

While ADLT is a powerful unit-testing tool, it cannot do all tasks associated with testing. ADLT was specifically designed *not* to be a test harness or debugger. ADLT relies upon the Test Environment Toolkit (TET) test harness to provide test management, building, execution, and clean-up structure.

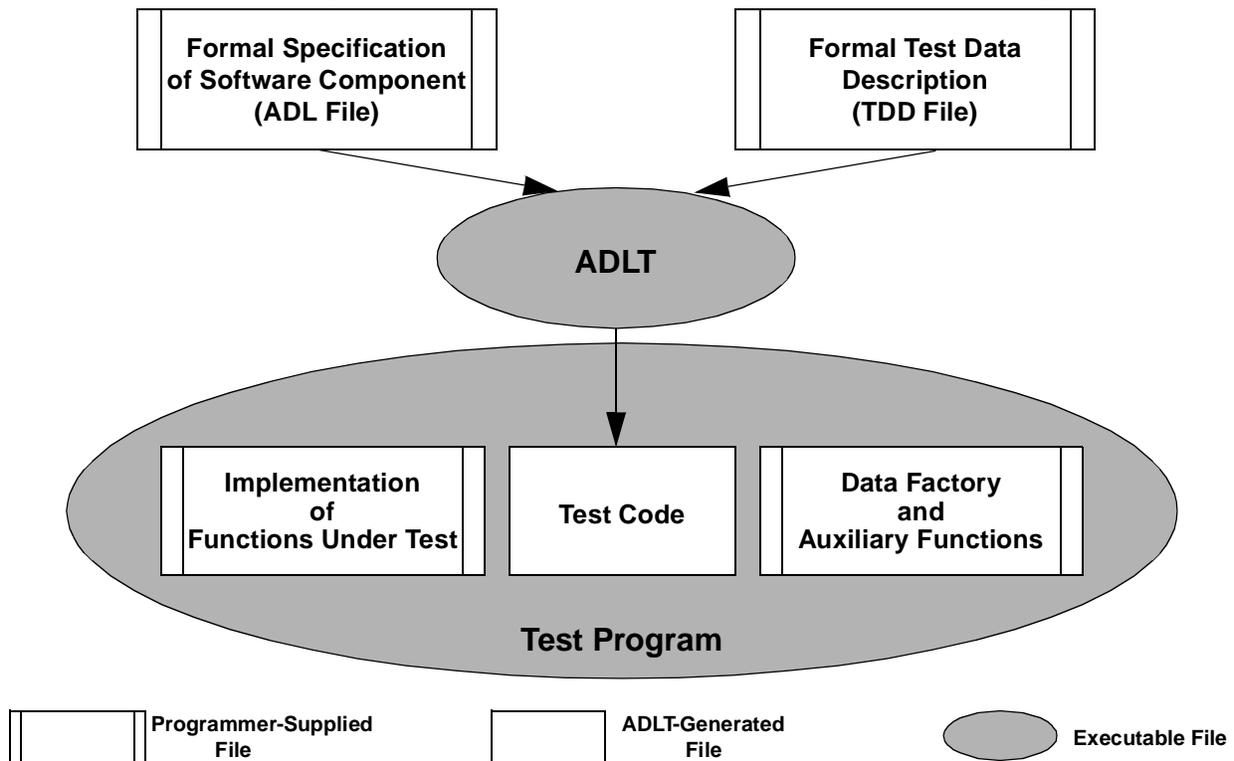


Figure 1-1 High-level view of ADLT test generation.

---

## Generating Tests with ADLT

The programmer specifies the behavior of the software component with the ADL file, and describes test data with the TDD file. Using the ADL and TDD files as inputs, ADLT generates test program code. The programmer then compiles the generated code, as well as programmer-supplied *test data factory* and any *auxiliary* functions, which are described below. To create the ADLT test program, the programmer links in the implementation of the function under test, along with libraries not shown in this high-level illustration. The linked component is the executable test program.

Here are the steps involved in using ADLT to generate tests:

### 1. Specify the component in the ADL file.

Using *Assertion Definition Language* (ADL) for the target programming language (e.g. *ADL for Java*), the engineer describes the functions that make up a software component to be tested. The specification includes not only the signatures of the constituent functions (along with the variables and constants they use), but also lists of *assertions* describing their intended behavior. In ADL, each element that makes up the software component should be grouped into a single specification file.

To check the specification, the engineer can now use ADLT to generate a natural-language version of the ADL module. See the next section, “Generating Documentation with ADLT,” for more information.

### 2. Write any auxiliary functions used in the ADL specification (or locate them in existing libraries).

An *auxiliary function* is any function that is used in an ADL specification to describe the workings of a function under test but which is implemented separately from the module being specified. An auxiliary function may be one written especially for testing purposes, or it may be one that already resides in a library. For example, a standard UNIX system call may appear in ADL assertions about a function; such a system call, because it is implemented separately from the module being specified, may be referenced in the ADL file.

Auxiliary functions may be implemented in-line, in the ADL specification described above. Alternately, the function implementations may reside in separate source or object files.

### 3. Describe test data and designate tests in the TDD file.

The engineer next describes the test data to be used in the ADLT-generated test. Test data may be specified both literally (by a list of actual values) and symbolically (by a list of abstract properties which take on symbolic values). The test data descriptions are placed in files called TDD files. TDD files also contain *test clauses*, which are TDD statements directing ADLT to generate a specific test on a specific function or collection of functions.

### 4. Run the ADLT compiler.

The ADLT compiler takes the ADL and TDD files as inputs. The program parses the inputs, performs a series of translations, and produces output files in the user's chosen programming language. The outputs include a compilable source files, makefiles, and files that help TET control the test execution. Many of these output files will eventually become part of the generated test program.

### 5. Write factory functions.

*Factory functions* are programmer-written functions that provide instances of data elements to be used in the test. Factory functions transform *datasets* (symbolic or literal descriptions of data collections) specified in TDD files into actual program values to be used in ADLT-generated tests.

*Relinquish functions* are also programmer-written functions. They relinquish any resources allocated by a corresponding factory function. If the factory function created a new instance of a class by calling `new`, for example, the relinquish function would normally return those resources to the system.

If the TDD specification included only literal descriptions of test variables (see Step 3), the user does not have to supply factory functions. The generated test program converts the literals into programming values of the specified type; those values are then used as inputs to the function under test. Therefore, if all test data are specified literally, the user may omit this step.

### 6. Write or locate the implementations of the functions under test.

If ADLT is being used to specify new software modules, now is the time to implement the functions. If the system is being used to test already-compiled functions, the user simply needs to locate the directory and/or library in which the functions reside.

**7. Make the test programs (one for each TDD test directive).**

The ADLT compiler (step 4, above) generates makefiles to help testers compile and link the various elements that comprise the ADLT test program. The test program elements include: the test source code generated by ADLT; implementations of the functions under test; implementations of factory, relinquish, and auxiliary functions; and libraries supplied by ADLT, TET, and the user.

### 8. Run the tests and examine the test result reports.

The user now runs the test program made in step 7, above. The test program gets data instances from the provide functions, passes them to the function under test, then evaluates the behavior of the function against its behavior specified in ADL assertions. Once the test run is complete, the test results can be reviewed either in summary or in detail using the `tetrep` program.

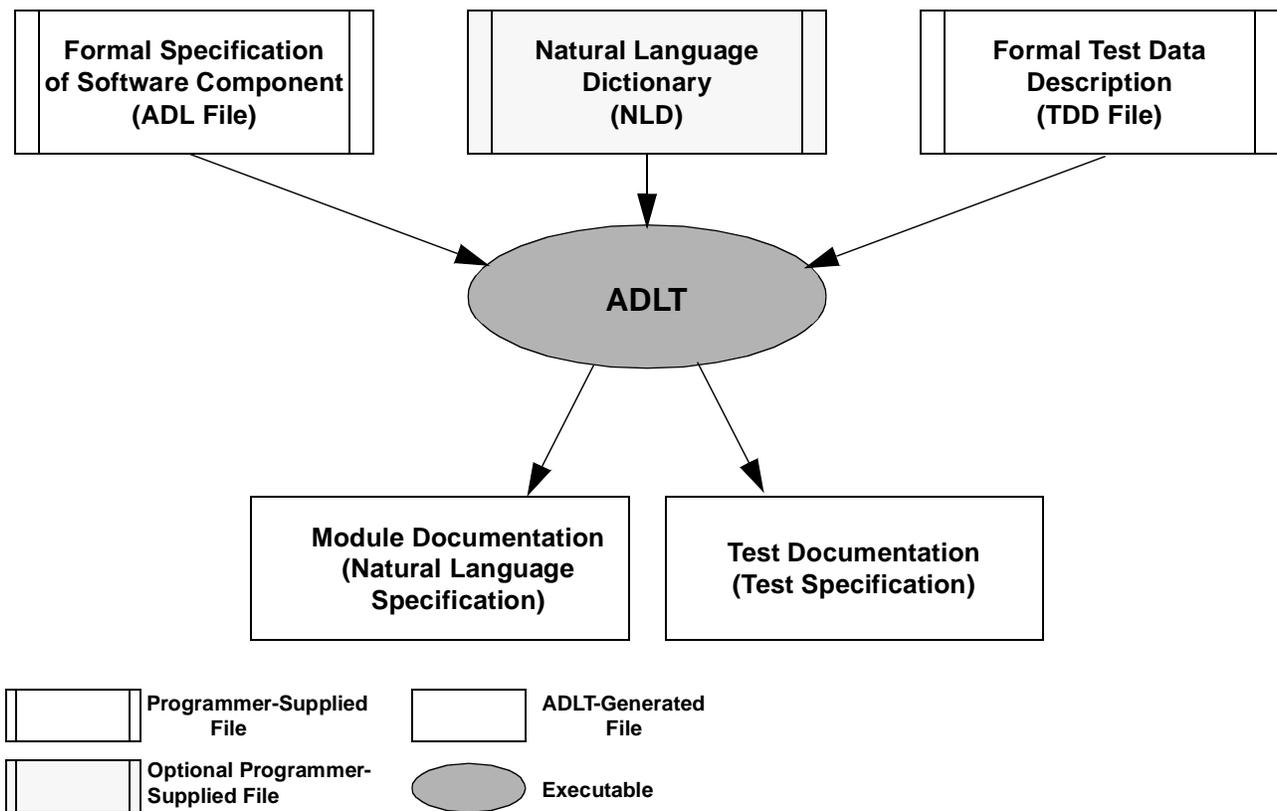


Figure 1-2 High-level view of ADLT documentation generation.

## *Generating Documentation with ADLT*

ADLT generates documentation of both the software components (as specified in ADL) and of the generated test. The system can generate all documentation from the ADL and TDD files. The user can improve the quality of the generated documents by supplying the Natural Language Dictionary (NLD).

ADLT can be used to generate two types of documentation:

- The *Natural Language Specification* (NLS)

The NLS is a natural-language description of each function under test. It contains a natural-language version of the ADL specification.

- The *Test Specification* (TS)

The TS is a natural-language description of the tests that can be generated by ADLT. It describes the functions under test, the assertions about their behavior, and the symbolic and/or literal values over which the test will be run.

Here are the steps involved in using ADLT to generate documentation:

**1. Specify the module in the ADL file.**

If this step has not already been performed for testing purposes (see the Section , “Generating Tests with ADLT,” on page 6), the engineer now creates the ADL specification of the functions to be documented.

**2. Describe test data and designate tests in the TDD file.**

If this step has not already been performed for testing purposes (see the Section , “Generating Tests with ADLT,” on page 6), the engineer creates the TDD file describing the data to be used in tests.

**3. Create the Natural Language Dictionary (optional).**

The *Natural Language Dictionary* (NLD) is a glossary of translations for the elements that make up ADL assertions. ADLT uses these translations to augment natural language descriptions that appear in the generated Natural Language Specifications and Test Specifications. ADLT can produce documentation without the input of the NLD, but the generated documents will be more mechanical and less “natural” than those produced with the assistance of the NLD. For more details, see Chapter 4, “A First Look at Documentation.”

#### **4. Run the ADLT compiler.**

The program can be run with a switch requesting only the NLS, only the TS, or both documents.

Notice that ADLT can produce documentation using only the inputs needed for testing purposes. With only the ADL file as input, ADLT can produce a default Natural Language Specification; with the input of both ADL and TDD files, ADLT can produce a default Test Specification. ADLT generates makefiles to assist in the generation of test programs and documents. See “Generating Tests with ADLT” on page 6.

With the use of the optional NLD, ADLT can be used as a document-generation system. ADLT natural-language specifications are derived from a formal language, which makes the generated documents more rigorous and less ambiguous than specifications created originally in a natural language such as English. The generated documents can therefore be viewed as the primary outputs of the ADLT system, or as documentation to be used in tandem with test generation.



## *Setting Up the Implementation*

---



ADLT can be used to specify and test already-compiled implementations of software components. The system does not mandate any specific structure for the implementation.

However, there is a default implementation file structure assumed by ADLT, which is reflected in some of the emitted files. If the implementation uses this default file structure, the process of introducing ADLT will be simplified; the user will not have to customize any of the emitted files.

### *The Default Implementation File Structure*

Here is how the default implementation file structure is derived from the specification:

- **The Module Name**

The *module* is the ADL unit of specification of a software component. The definition of the elements that comprise the component is up to the test designer. The module should consist of variables, constants, type definitions, and functions that are related for the purposes of specification, development, testing, and/or documentation. In general, the concept of the ADLT module is roughly equivalent to the contents of a C header file or a Java Class.

The designer groups the elements together into a specification unit and gives that unit some descriptive or already-existing name—the module identifier. The module identifier thenceforth determines several of the default file names expected by the ADLT system. These naming conventions are language dependent. The JavaLanguage, for example, does the following:

The ADL specification of a component named *module* must be placed in a file named *module.adl*. The default Java-language implementation of that module is a source file named *module.java* and a bytecode file named *module.class*.

- **A Source File Named *module.java***

ADLT expects to find the Java-language implementation of the module in a source file named *module.java* and a bytecode file named *module.class*. This source file should contain the Java implementations of the functions, variables, and constants that make up the software component under specification. This file implements the ADL specification.

This chapter provides an example of the ADLT testing process. It does not discuss all the features of ADLT. The goal is to give an overview of the inputs the engineer must provide to the ADLT system and a feel for the ADLT testing process.

The chapter uses a simple banking module—the “Hello, world!” example of ADLT testing in the Java programming language.

## *The Java-Language Module*

Here is the Java-language class to be specified and tested using ADLT. This class implements three methods:

- `deposit` - increases the balance in the account by the given amount
- `withdraw` - decreases the balance in the account by the given amount
- `bank` - a constructor that creates a new account

### *Java-Language Source File*

```
public class bank {  
  
    final public int MAX = 100;  
    protected long acct_number;  
    protected long balance;  
  
    public bank(long acct, long bal)
```

```
    {
        acct_number = acct;
        balance      = bal;
    }

public long get_account_number()
    {
        return acct_number;
    }

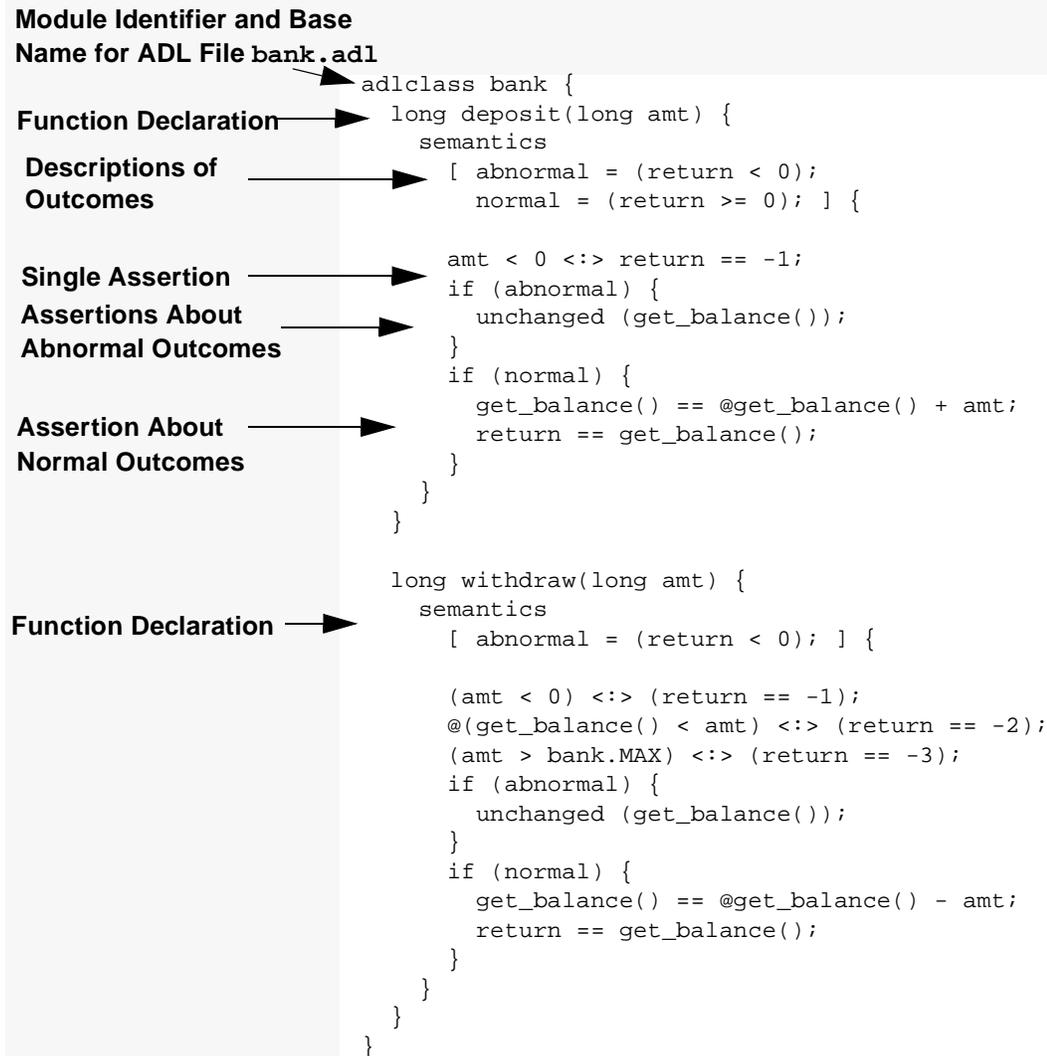
public long get_balance()
    {
        return balance;
    }

public long deposit(long amt)
    {
        if (amt < 0) {
            return -1;
        }
        balance += amt;
        return balance;
    }

public long withdraw(long amt)
    {
        if (amt < 0) {
            return -1;
        }
        if (balance - amt < 0) {
            return -2;
        }
        if (amt > MAX) {
            return -3;
        }
        balance -= amt;
        return balance;
    }
}
```

**Code Example 3-1** Implementation source file bank.java

## The ADL File



**Implication Operator** -->      **Call-State Operator** @      **Exception Operator** <:>

Figure 3-1 An annotated example of an ADL file: bank.adl

## *Illustrated Features of ADL*

In general, ADL follows the syntax of the target language (in this case Java). Experienced programmers should find it relatively easy to read and understand the meaning of the statements in the ADL file above.

However, the sample ADL file contains some differences between Java and ADL, as well as some of the new language elements and concepts introduced in ADL:

### **1. ADL specifications can be partial**

The ADL file does not have to specify every element in the Java-language implementation of the module. The sample ADL file does not mention the Java method `get_account_number`, for example, which is part of the implementation. Also, the assertions for `withdraw` are fairly extensive, whereas there is only a single assertion for `deposit`. It is up to the engineer to decide what should be specified and to what depth.

### **2. The ADL module**

The ADL unit of specification is the `module`. The `module` is any set of functions, variables, and constants which the engineer considers to be related for the purposes of specification and testing. Typically, a module is defined by one class file in Java, or one header file in C or C++.

There can be only one module per ADL file. ADLT requires that the base name of the ADL file be the same as the module name; the file suffix should be `.adl`. The name of the ADL file is therefore `module.adl`.

---

**Note** – ADLT requires that the implementation of the functions specified or referenced in the ADL file be in the CLASSPATH or the ADLT-specified `incpath`.

By default, ADLT also expects to find the Java-language implementation of the elements declared in `module.adl` in a source file named `module.java` or the bytecode file `module.class`.

---

### **3. ADL semantics**

ADL contains the reserved word `semantics`, which appears following the function signature. The reserved word introduces a block of bindings (see item 8, below) followed by a list of assertions about the function. In general, ADL assertions are boolean expressions that describe the state of the system immediately after the function under test completes execution; that is, in the

return state of the function (see item 6, below, for ways to reference values in the call state). For a function to pass an ADLT test, every assertion must evaluate to TRUE. Not all functions in the ADL file are required to have semantic descriptions; however, if they do, they are called *annotated functions*; that is, functions annotated by semantics.

#### 4. ADL expressions

Java, C++, and C-language statements are terminated by semi-colons. In ADL, statements in the semantics block are also terminated with semi-colons.

#### 5. ADL group expressions enclosed within braces

In ADL, expressions can be grouped within braces. Such groupings are called *group expressions*. In the sample ADL file, the statement following the `normal` reserved word is enclosed within the braces. The braces enclose a list of expressions. For the function `withdraw` in the sample file, the list consists of a single element.

#### 6. ADL call-state operator; the built-in function unchanged

In general, ADL assertions are boolean expressions that describe the state of the system immediately after the function under test completes execution (in the return state). However, the language also provide two means to refer to values in the call state, or state before the function under test is evaluated, of the function:

the *call-state operator* (`@`), a unary operator, which tells ADLT to evaluate the expression in the call state of the function

the built-in function `unchanged`, which returns TRUE if the value of its argument is the same in the call and return states of the function, else FALSE.

#### 7. ADL reserved word `return`

The reserved word `return` refers to the return value of the currently annotated function. This reserved word should not be confused with the Java-language statement. In ADL, `return` is not an imperative statement but a reference to a value.

#### 8. ADL Bindings

ADL uses the binding operator (`define`), which is not found in Java, C, nor C++. A binding is a mapping of an expression to an identifier. When that identifier is used in other expressions, it is replaced with the expression and the expression is evaluated in the context of the assertion. The user can

create any number of bindings. However, if any bindings are used within a group expression (that is, within any list of ADL expressions enclosed within braces), then the bindings must be the first statements in the list.

**9. Normal and Abnormal completion evaluation**

The reserved word `abnormal` should be bound to an expression that characterizes the way the function tells the caller that it has encountered some problem during execution—a file can't be opened, for example, or a communication line broke its transmission. The ADL reserved word `normal` should be bound to an expression that characterizes the normal workings of the function—the way the function tells the caller that no errors, anomalies, etc., have occurred. In UNIX system calls, for example, the usual binding for `normal` would be a return value of zero, and the binding for `abnormal` would be a -1 return value. Note that the ADL `abnormal` operator and ADL `normal` operator depend upon these bindings for their correct operation.

**10. ADL implication operators `-->`, `<--`, and `<-->`**

AD introduces three implication operators. The first, is the true implication operator `-->`, which may be pronounced as “left-hand side implies right-hand side.”

Second is the reverse implication operator `<--`, which simply reverses the order of implication (“right-hand side implies left-hand side”). Next is the logical equivalence operator `<-->`, which is used when the left- and right-hand sides should both be TRUE or both be FALSE simultaneously. For the formal definitions of these operators, language reference manual for your target language.

**11. ADL exception operator**

The *exception operator* (`<:>`) is used for assertions about exceptional function outcomes. The exception operator is used to specify conditions that should be TRUE when an exception occurs—the sorts of outcomes usually listed in the “Errors” section of a man page. For a formal definition of the exception operator, see the language reference manual for your target language.

**12. ADL `normal` operator**

The reserved word `normal` is a reserved word that evaluates to a boolean. It is used in the example above to conditionally evaluate assertions only when the function under test behaved normally. Similarly, the `abnormal` reserved word is used for the opposite purpose.

## *What to Place in ADL Assertions*

The ADL function declaration begins with the function signature, something available in any source file. The `semantics` section then augments information about the interface to the function. The assertions describe those aspects of the function which are important to the reader—anyone who is trying to use, understand, and/or test the function being specified.

The “important” aspects of the function which are candidates for ADL assertions include:

- constraints on input parameters
- constraints on parameters whose value is set by the function (output parameters)
- expected return values, including use of return values to characterize the normal and exception outcomes of the function
- significant conditions that should be true after the function has completed execution
- examples of outputs based on representative inputs

The last candidates for assertions—examples—do not necessarily create good tests; however, examples can be very useful in documenting and explaining the function interface to a human reader.

## *What Not to Place in ADL Assertions*

Ideally, assertions describe an interface to a function. A single ADL specification can potentially be used to specify and test multiple implementations. Therefore, assertions should not contain details about the internals of the function.

In general, do *not* write assertions containing:

- constraints on the method of implementation (unless such constraints are part of the interface)

Details about the inner workings of the implementation are best placed in auxiliary functions. In the sample ADL file, for example, any logic about how balances are stored is kept “hidden” behind the interface to the auxiliary function `balance`. If the implementation later changes the way account

balances are stored (in a database rather than in an array, for example), the ADL specification can survive the change. Only the auxiliary function needs to be rewritten to reflect the changes in the implementation.

### *Parsing the ADL File*

It is good practice to use the ADLT compiler to parse the ADL file before proceeding with the rest of the test.

To parse the sample ADL file, first ensure that there is a .class file for the sample bank implementation, then enter the command:

```
adlt bank.adl
```

The command tells ADLT to parse the ADL file. If the file parses correctly, ADLT generates all outputs dependent upon ADL input. The ADL-dependent output files are shown in the illustration below (in the case of Java, there is only one). (All output files are discussed later, in the section entitled “Program Inputs and Outputs” on page 29.)

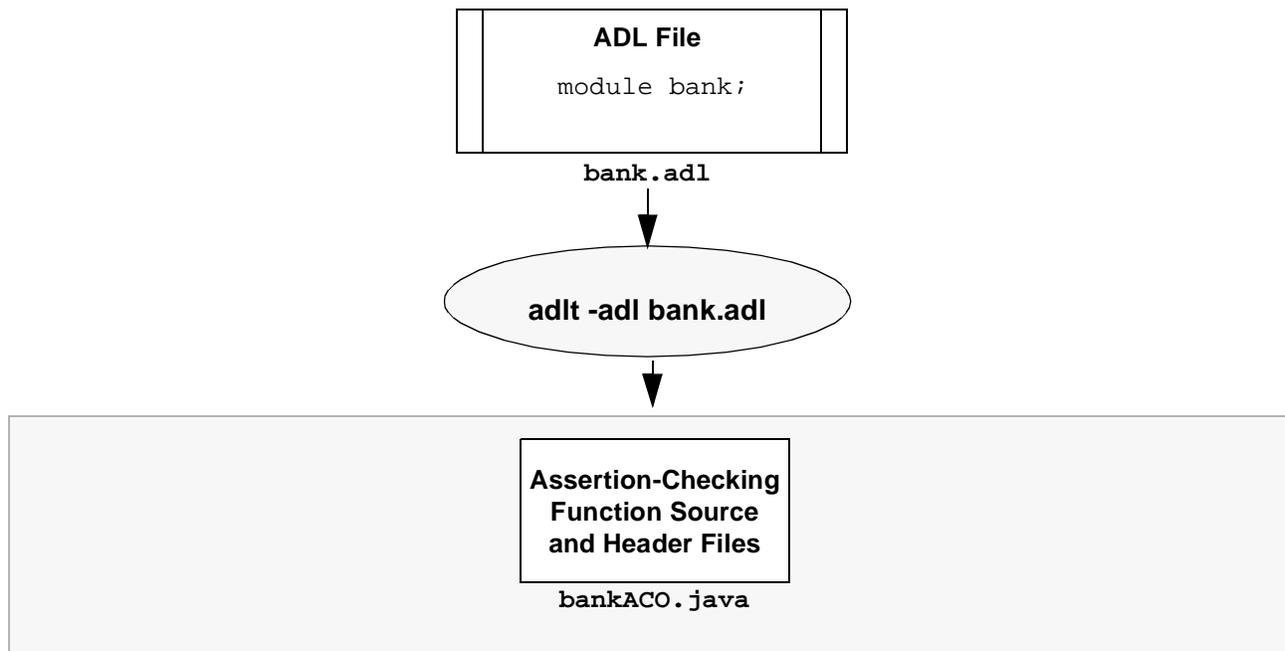


Figure 3-2 Outputs of ADLT dependent upon input of the ADL file.

### *Looking at the Natural-Language Specification (NLS)*

To verify that the functions in the ADL file are correctly specified, you may wish to use ADLT to generate the default natural-language specification file (NLS). The NLS contains a natural-language version of the ADL specification. (See Chapter 4, “A First Look at Documentation,” for more details.)

To generate the NLS, enter the following command:

```
adlt -nls bank.adl
```

## *Auxiliary Functions*

The next step is to supply an implementation of any auxiliary function or functions which were referenced in the ADL file and not defined in-line. In the case of Java, this can be done in any source file/bytecode file that is available in the CLASSPATH.

## The TDD File

```

public tddclass bankTest {
    dataset int ACCOUNTS      = { 1, 99 };
    dataset int BALANCES      = { 0, 100, 1001 };
    dataset int DEPOSITS      = { -1, 0, 3, 1000 };
    dataset int WITHDRAWS     = { -1, 0, 7 .. 9, 100, 101 };

    dataset bank BAL = make_account(ACCOUNTS, BALANCES);

    factory bank make_account (int account_number,
                              int initDeposit) {
        return new bank(account_number, initDeposit);
    }

    withdraw_test: test (bank b = BAL, int w = WITHDRAWS) {
        ADL(b).withdraw(w);
    }

    test (bank b = BAL, int d = DEPOSITS) {
        ADL(b).deposit(d);
    }

    full_test : test (bank b = BAL,
                    int d = DEPOSITS) {
        long bal = b.get_balance();
        ADL(b).deposit(d);
        ADL(b).withdraw(d);
        if (d <= b.MAX) {
            tdd_assert("bal == b.get_balance()", bal == b.get_balance());
        }
    }
}

```

**Literal Dataset Definitions** →

**Dataset where values factory generated** →

**Factory used by bank dataset** →

**Unit test - tests a single method** →

**Another unit test** →

**Subsystem test** →

Figure 3-3 An annotated example of TDD file named bankTest.tdd.

## *Illustrated Features of the TDD Language and Files*

Now that the module has been specified with ADL, the next step is describe the test data to be used as inputs to the functions under test. To do so, the engineer creates a TDD file.

Note the following illustrated features of the TDD file:

### **1. The `tddclass` statement**

Each TDD file must name its class. The class name is used to name generated files, and must be the same as the name of the tdd file.

### **2. Datasets**

In a TDD file, the test engineer defines collections of test data over which the test driver will iterate. These collections are known as *datasets*. A dataset can be defined as a collection of literal values, a collection of other datasets, or as a factory (see below).

### **3. Factories**

In addition to TDD datasets, the test engineer can define one or more test data factories. A factory is a method that returns an object of any type. The factory takes as arguments one or more datasets. At each iteration of the test driver, the factory method is called with the appropriate arguments. The factory is then responsible for establishing whatever conditions it wants and returning an appropriate value. The factory can also use methods such as `tdd_skip()` to control the execution of the tests.

### **4. Test directives**

The *test directive* directs ADLT to generate test code for the associated function or sequence of functions, using the specified datasets as descriptions of the parameters of the test function. The test name is taken from the label, if present, or else is generated by ADLT. An unlabelled test directive will result in a test name that is the identifier of the `tddclass`, the string “directive”, and an appended integer suffix. For example, the test name for the unlabelled test directive on `deposit` will be `D_bankTest_directive_1`.

## *Other TDD Features and Concepts*

In addition, here are other TDD features and concepts important to understanding how test data is specified in ADLT:

### 1. The TDD use statement

A TDD file can make visible the contents of any number of other TDD files via the `use` statement. For example, a TDD file containing the `use` statement below can “see” and reference all contents of the TDD file `balances.tdd`:

```
use "balances.tdd";
```

The effect of the `use` statement is transitive: a file “used” in a second file is imported transitively when that second file is later “used” by a third.

### 2. The ADLT Input Grid

Each test directive defines a set of points on an *input grid*. The input grid is the set of test data input points generated by ADLT for a specific test of a function.

The input grid is calculated as the cross product of all values of all datasets of all test variables involved in a given test directive. For example, the test directive labelled `withdraw_test` defines an input grid of 42 potential inputs: the 6 potential values for `b` times the 7 potential values for `w`. (We say that the inputs are “potential” because factories do not have to supply a value for each point for each test variable.)

## What to Place in TDD Files

When formulating the test variables and their datasets, keep in mind the following principles:

- The TDD file must contain at least one test variable with an appropriate dataset for each function parameter.

The user can create and re-use a single test variable, if that one test variable supplies a suitable description of each function parameter. A test variable has a “suitable” description of a function parameter if:

it has the same data type as the function parameter, or is compatible with it.<sup>1</sup>

it adequately characterizes the actual values the parameter should have during the test.

---

1. Type compatibility in TDD follows rules of the chosen language for assignment compatibility.

- Representation of the sample

The task in choosing properties is to select a representative sample from the possibly very large universe of parameter values. The problem is one of equivalence partitioning: the TDD author needs to identify the finite set of samples that provide a satisfactory representation of the universe of data values that can be assigned to a test variable.

- Combinations of independent properties

The TDD file factory mechanism is designed to work with properties that are independent of each other. During the running of the ADLT-generated tests, those properties will be concatenated into a list of adjectives describing the data element, and the associated factory will be asked to provide an appropriate value of that combination of properties.

## *Generating Tests*

Since the ADLT generator requires only two types of inputs—an ADL file and TDD files—the program can now emit test code.

### *The ADLT Command*

To generate a test for each directive in the test directive file, use the following command:

```
adlt -v bankTest.tdd
```

This command tells ADLT to generate all outputs related to test generation, and the `-v` argument tells the system to be verbose about it. As the outputs are emitted, their file names are listed.

Notice that the command gives ADLT the name of the TDD file containing the test directives currently of interest. If the TDD file containing the currently interesting test directives is in `myModule.tdd`, for example, the user gives the following command:

```
adlt -v myModule.tdd
```

The illustration below shows the inputs to ADLT and the generated outputs for the `bank` sample module.

### Program Inputs and Outputs

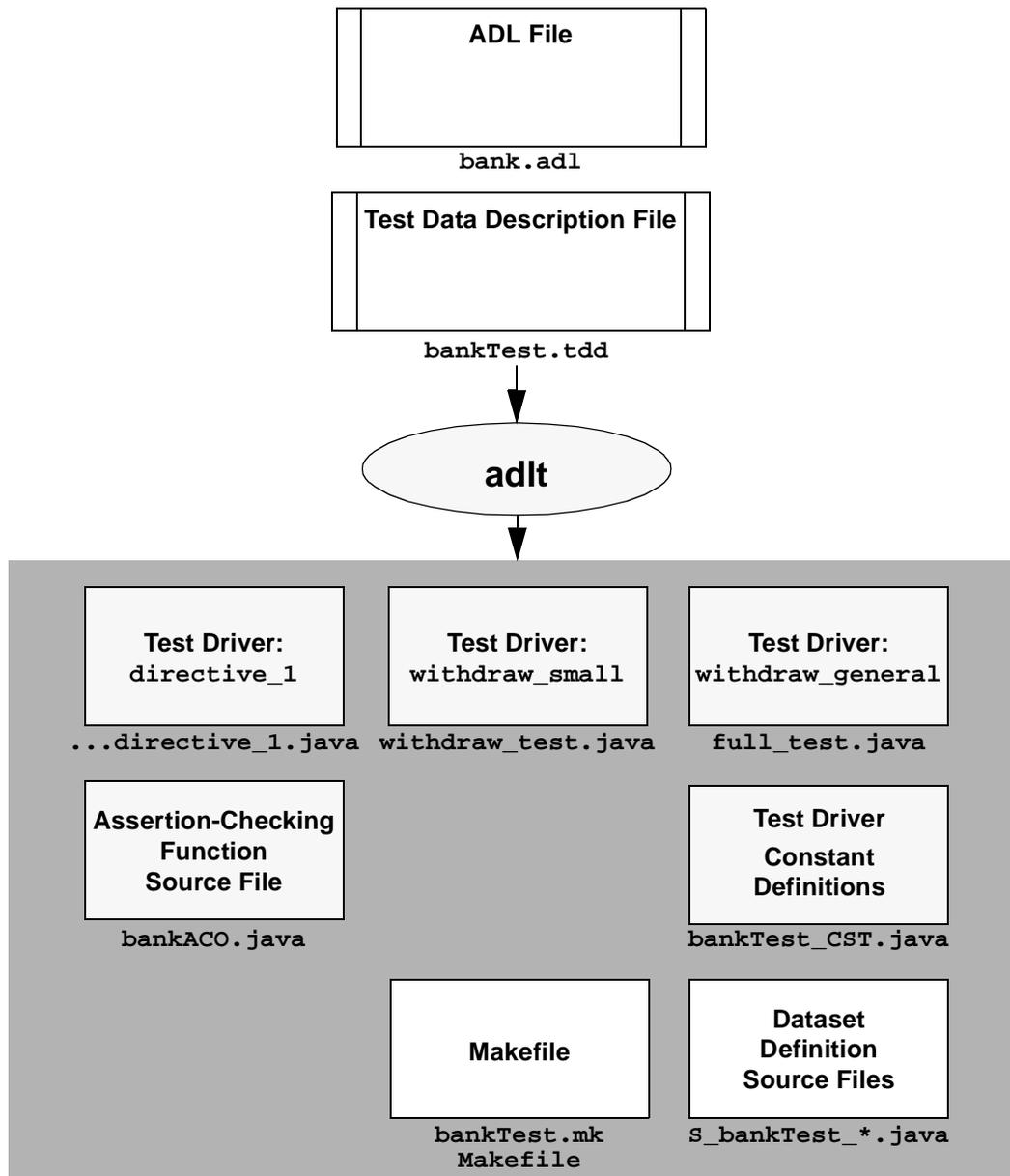


Figure 3-4 The inputs to ADLT and generated outputs for the bank module.  
Users Descriptions of Generated Files

ADLT emits the following types of files:

- Assertion-Checking Objects

ADLT translates the functional specifications in the ADL file into *assertion-checking objects*. Assertion-checking objects (ACOs) are functions that check for the behaviors specified by the ADL assertions. When compiled and linked into the ADLT test program, the assertion-checking functions compare the actual behavior of the functions under test to the behavior specified. ADLT creates a source code file containing one assertion-checking function for each annotated function in the ADL file.

Unless they wish to examine the generated code, users do not have to examine or manipulate the contents of this file.

- Test Drivers

ADLT creates one *test driver* for each test directive in the TDD file. When compiled and linked into the test program, the test driver iterates over the grid of test data inputs specified by the test variables in the TDD file. The driver calls factories and dataset functions to get actual values for each test variable, and it passes those values to the assertion-checking object(s) for the test function. The assertion-checking object, in turn, invokes the function under test, which executes using the provided test data values. The driver then conducts reporting functions.

Unless they wish to examine the generated code, users do not have to examine or manipulate the contents of this file.

- TDD Class Constants source file

For each `tddclass` declaration, ADLT generates a source file that defines a parent class for factories and datasets (in Java this is an interface definition). The dataset and factory source files (below) implement this interface definition, thereby inheriting any constant declarations that were defined in the `tddclass`.

- Factory and dataset source files

ADLT generates a source file for each factory. The class definition is taken from the implementation provided in the TDD file.

- Makefiles

The program also generates makefiles to help TET generate and maintain the test program and documentation.

- TET Scenario File

If an installation is using TET, ADLT can generate a scenario file. This generated file is not shown in the diagram.

## *Factories and Relinquish Functions*

The next step is to write the functions that will perform any setup necessary for the execution of unit or sub-system tests.

### *Role of Factories and Relinquish Functions*

As described above, factories are methods that are used to provide a value for use in testing. This value may be an isomorphic mapping from the input space, or it may be some more esoteric relationship between the values of several dataset-member values passed as parameters to the factory. Regardless of the purpose of a particular factory, the general purpose is simply to generate values of a specific type when those values (and associated side-effects) cannot be generated through simple literal dataset definitions.

When a factory has side-effects (e.g. allocating memory), the test engineer may choose to also write a relinquish function. The purpose of a relinquish function is to return resources to the system or otherwise clean up after a factory once the test has been performed.

Beyond these simple tasks, factories can evaluate system state or the combination of parameters passed to them. If, for example, a certain combination of parameters to a factory is not sensible for testing (e.g. a POSIX-compatible file name when the file system being tested doesn't handle POSIX file names), the factory can call the `tdd_skip()` method to indicate to the test driver that this test data instance should not be tested.

### *Factory/Relinquish File Naming Conventions*

ADLT generates a source file for each factory and relinquish function specified. That source file (in Java) defines a class that contains an ADLT-generated constructor, the user-provided factory and (optionally) the user-provided relinquish function. The source file is named for the tddclass in which the factory was defined and the name of the factory (e.g. `F_bankTest_make_account.java` in the bank example).

### *Making the Test Program*

Using the makefiles emitted by ADLT, the user processes the ADL and/or TDD inputs to create ADLT-generated files and test programs.

ADLT generates the necessary makefiles.

### *Building the Test Program*

Once the test program has been generated, it needs to be compiled. The simplest way to do this is to let TET handle it. Just enter the command:

```
tcc -b <test_dir>
```

TET will use the generated makefiles and the ADL-provided TET configuration files to call the appropriate compilers, generate the object files, and link them together into an executable test.

### *Running the Test Program*

To run the test program `withdraw_general`, enter the following command:

```
tcc -e <test_dir> withdraw_general
```

This command will run the test scenario `withdraw_general`

By default, the test program will attempt to execute all test instances. Users may also request individual test instances, a list of instances, a range, or only instances that produced errors in a prior run of the test program.

## *A First Look at Documentation*

---



This chapter describes the documentation that can be generated by ADLT. It shows generated documentation for the simplified `bank` module introduced in Chapter 3, “A First Test.” This chapter does not discuss all the documentation features of ADLT. The goal is to provide a feel for the ADLT system and its use in generating documents.

---

**Attention Reviewer!** – This section is not yet complete, as this part of the ADL Translation System is still under development.

---

### *The Role of the Natural Language Dictionary*

ADLT input files can be used, on their own, as documentation. An ADL file can be a means of documenting the interface to a module. And TDD files record the significant characteristics of test data.

However, users usually require more readable types of documentation. Each function in a module is normally described in a `man` page. And testers need a formal description of the specified tests. ADLT offers both types of documents:

- The Natural Language Specification (NLS)

The NLS is a natural-language description of each annotated function.

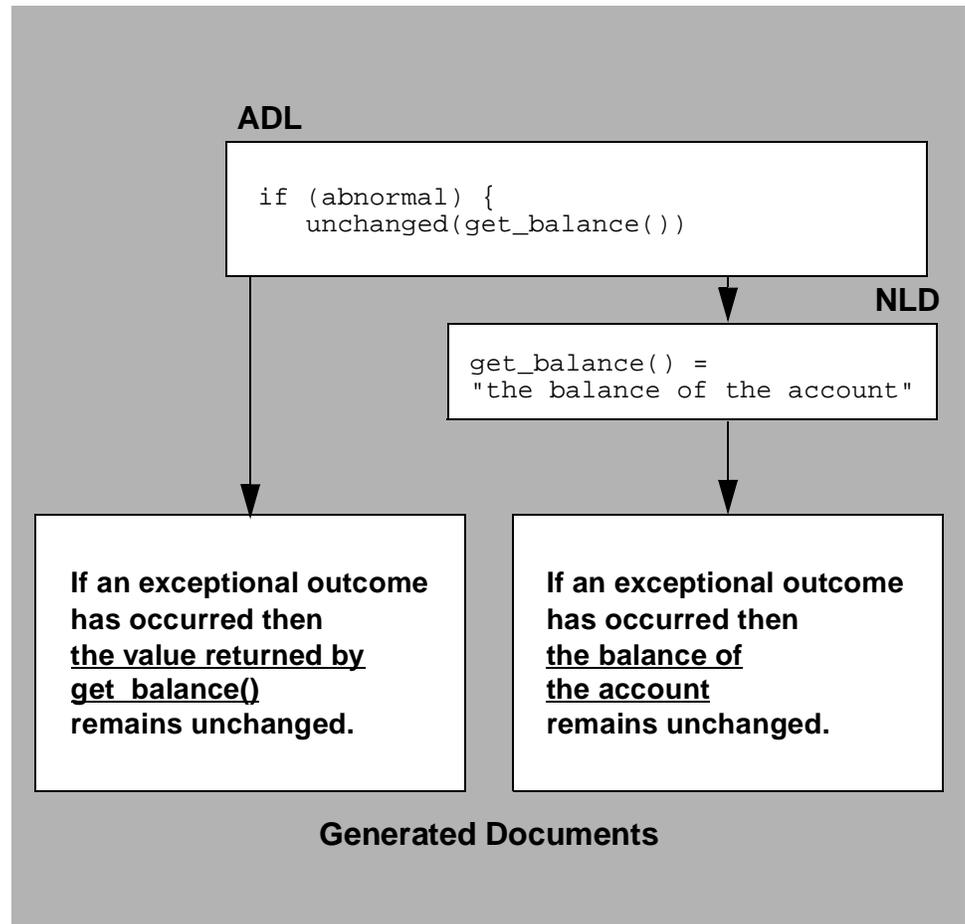
- The Test Specification (TS)

The TS is a natural-language description of the tests that can be generated by ADLT. It describes the functions under test, the assertions about their behavior, and the symbolic and/or literal values over which the test will be run.

ADLT can generate both documents from the ADL and TDD files written for testing purposes. In addition, the system offers a way for the user to provide explanations for identifiers used in the ADL specification. By supplying the optional Natural Language Dictionary (NLD), the user helps ADLT convey more meaning in generated natural-language translations.

The NLD is a glossary describing the elements that make up ADL assertions. These descriptions are used as translations. In generated documentation, the element names are translated into the given natural language descriptions. Via these translations, NLD entries augment the natural language renderings of ADL assertions as they appear in the generated NLS and TS.

The illustration below shows a sample NLD entry describing the function `balance` and the resulting improvement in the generated natural-language translation of an ADL assertion:



*Figure 4-1* Illustration of an ADL assertion, an NLD entry describing a function referenced in the assertion, and the resulting natural-language translation. The translation on the left was generated without the input of the NLD; the one on the right has been augmented by an NLD translation.

The use of NLD is optional. In practice, users will probably first generate documentation without NLD input, evaluate the quality of the generated document, then provide NLD entries for those areas that will benefit most from augmented descriptions.



## *Part 2 — Effective Use of ADLT*

---

---

This chapter discusses the organization of the files that make up an ADLT generated test suite. It gives an overview of the files involved in the test program and suggests various approaches to structuring ADLT specifications and organizing test suites. It contains the following major sections:

- Overview of ADLT Files (See page 40.)

Shows an overview of the files that make up the ADLT test program.

- Importing and Referencing Specification Files (See page 46.)

Discusses the use of multiple ADL and TDD files to structure specifications. Also discusses how ADL and TDD modules should be related.

- Organizing the Test Suite (See page 58)

Discusses approaches to organizing ADLT directories and files at the user's installation, with the goal of creating reusable libraries of ADLT specification and code files.

## *Overview of ADLT Files*

As illustrated on the next page, an ADLT test is composed of the following types of files:

**1. Specification Files**

The ADL, TDD, and NLD specification files.

**2. Implementation Files**

The file representing the implementations of the functions under test, along with the implementation's header file (in languages where this is appropriate).

**3. ADLT-Generated Files**

The language specific header files (when appropriate) and source files for test drivers and assertion-checking functions. In addition, to facilitate the making and running of the test program, ADLT generates makefiles and TET scenario files.

**4. Test Code Files**

Implementations of the factory and relinquish functions supplied by the user, any auxiliary functions used in the ADL specification, as well as any optional user-defined print, initiation, and termination routines.

**5. Libraries**

The ADLT runtime library and the TET library, as any user libraries used as auxiliary functions.

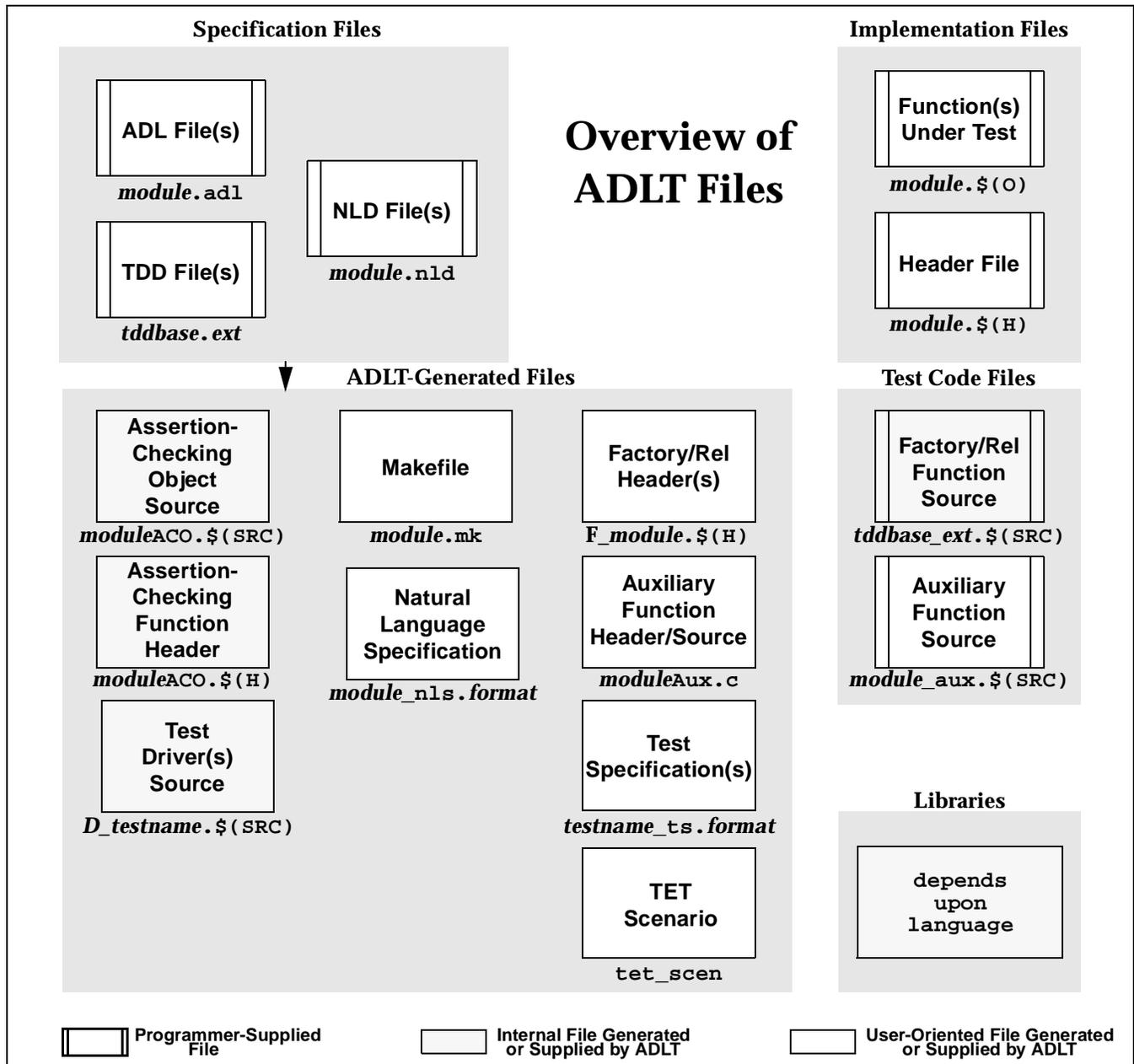


Figure 5-1 Overview of files involved in the ADLT test suite.

## File Naming Conventions

ADLT uses the following naming convention, where the suffixes vary depending upon the language binding being used:

File	Description of Contents
<b>Specification Files</b>	
<i>module.adl</i>	Top-level ADL file for the software module to be tested. Mandatory file name format.
<i>tddbase.tdd</i>	TDD file containing test directives for test code to be generated during the running of ADLT. Mandatory file name format.
<i>module.nld</i>	Optional NLD for the module. If present, file name format is mandatory. Note that the full path name may depend upon the locale.
<b>Implementation Files</b>	
<i>module.\$(O)</i>	Compiled functions to be tested. The file <i>module.\$(O)</i> should contain implementations of the constituent (non-auxiliary) functions declared in <i>module.adl</i> . The user's implementation file may have some other name; the user can change the name in the generated makefile <i>tddbase.mk</i> , or override it in an ADL configuration file.
<i>module.\$(H)</i>	Top-level header for all functions to be tested. Mandatory file name format.
<b>ADLT-Generated Files: User-Oriented</b>	
<i>tddbase.mk</i>	Makefile containing the make logic for each <i>testname</i> and the Test specification. Generated file name format.
<i>module_nls.format</i>	Natural language specification for the given module, in the chosen output format. Generated file name format.
<i>testname_ts.format</i>	Test Specification(s)—one per test directive—in the chosen output format. Generated file name format.

File	Description of Contents
<i>moduleAux.\$(H)</i>	Auxiliary function header. Also contains declarations of pointers for optional user-defined print functions and wrappers around the function under test. See the <i>ADL Translator Programmers Guide</i> for details. Mandatory file name format.
<i>moduleAux.\$(SRC)</i>	Source file containing any definitions of auxiliary functions placed in-line in the ADL file. Generated file name format.
<i>tddbase_CST.\$(H)</i>	Factory/relinquish header file(s)—one per <i>.tdd</i> file. Generated file name format.
<i>tet_scen</i>	Optional TET scenario file (for TET environments). Generated file name format.
<b>ADLT-Generated Files: Internal Files</b>	
<i>moduleACO.\$(H)</i>	ACF header file. Generated file name format.
<i>moduleACO.\$(SRC)</i>	ACF source code file. Generated file name format.
<i>D_testname.\$(SRC)</i>	Test driver source code file(s)—one per test directive. Generated file name format.
<b>User-Supplied Test Code Files</b>	
<i>moduleAux.\$(SRC)</i>	Source for implementations of functions referenced in factory, relinquish, or test functions that are not provided elsewhere in the implementation.  May also contain user implementations of any optional print-formatting functions, initiation and termination routines, and/or wrappers around the function under test. See the <i>ADL Translator Programmers Guide</i> for details. The user can work with other file names by changing the name in the generated makefile <i>tddbase.mk</i> .

Where:

*module*

is the name of the ADL module.

*tddbse*

is the base name of the TDD file at end of the “use” chain (usually the one containing the test directives currently of interest).

The TDD base name can be any name (subject to system file length restrictions). It can be the same as *module*, but it does not have to be.

*ext*

can be one of the TDD file extensions, `.tdd`, `.tdi`, or `.tdr`.

All suffixes identify TDD files; `.tdr` and `.tdi` identify restricted file formats. See “Information and Directive Files,” later in this chapter, for details about TDD restricted file formats.

*testname*

is the name of the ADLT-generated test program. The test name is constructed by using the test directive label or, if no label is present, by using the name of the function under test with an incremented integer suffix starting with 1.

*format*

is the documentation output format, either `man` (a file designed to be processed with `troff` and the `man` macros; the default), or `html` (hypertext markup language format).

*\$(SRC)*

is the source code suffix for the target language.

*\$(H)*

is the header file suffix for the target language (assuming it uses headers).

*\$(O)*

is the generated object code suffix for the target language.

### *Composition of the ADLT Test Program*

The ADLT test program is composed of ADLT-generated and programmer-supplied elements. These elements are compiled and linked with implementations of the functions under test and the ADLT library.

The source files for compilation are:

- the assertion-checking objects generated by ADLT. Assertion-checking objects determine whether or not the function under test behaves as specified in ADL.
- the test drivers generated by ADLT. There is one generated test driver for each test directive in the TDD. The test driver contains a loop that iterates over the test data inputs specified by the properties and values of each test variable.
- the programmer-supplied implementations of factory and relinquish functions. The factory function is the ADLT element that turns the symbolic description of data in the test variable into an actual programming value to use in the test. The relinquish function releases the resources allocated by the factory function.
- the programmer-supplied implementations of auxiliary functions (if any are declared in the ADL specification). Auxiliary functions provide access to functionality that is not exposed in the module but which is necessary for a complete specification of the intended behavior of a function. In the ADLT-generated test, the auxiliary function is called by the assertion-checking function in the course of evaluating the assertions about the function under test.
- the optional programmer-supplied functions to customize the test and the test report: test prologue and epilogue routines, wrappers around the function under test, user-defined print formatting functions. (See the *ADL Translator Programmers Guide* for details.)

The compilation of the elements above depends upon the following header files generated by ADLT or supplied by the user:

- the user-supplied implementation header file (*module.\$(H)*)
- the factory/relinquish function declaration header file or files
- the auxiliary constituent declaration header file
- the assertion-checking object declaration header file

The compiled elements, above, are linked with the following elements to produce the test program:

- the compiled implementations of the functions under test
- the ADLT library for the target language

- the TET library for the target language

ADLT generates two kinds of makefiles: one to process the elements relative to a TDD file, and one to control the building, execution, and cleaning of tests from TET. The former is in `tddbbase.mk`. The latter is in the file `Makefile`. This is dynamically updated `adlt` is executed in the directory. Variables in the generated `tddbbase.mk` file can be overridden via an ADL configuration makefile of variable definitions.

## *Importing and Referencing Specification Files*

In ADLT, the user has many options when structuring specification files. For a given module, there may be one ADL and one TDD file, or the specifications may be organized in multiple files that import and reference one another.

This section discusses approaches to using multiple specification files in the creation of ADLT tests. Specifically, it addresses:

- Importing ADL Files (See “Importing ADL Files” on page 46.)
- Referencing TDD Files (See “Referencing TDD Files” on page 51.)
- The Relationship Between ADL and TDD Modules (See “The Relationship Between ADL and TDD Modules” on page 53.)

### *Importing ADL Files*

ADLT uses the module as a way of organizing specification components; the module can be any collection of type definitions, variables, constants, and functions that are related for the purposes of specification, documentation, and/or testing. The constituents of an ADL module are approximately those that might appear together in a C/C++ header file or in a Java class definition.

It is up to the ADLT test designer to decide how the implementation’s header and code files should be represented in ADLT specification modules. The designer may create one ADL file for each header file, for example, or may create one large ADL file containing specifications of all related functions.

### *The Implementation Files*

For example, consider the following C header file and three source files:

```
/* balances.h */
typedef int account_balance;
typedef int transaction_amount;
typedef int account;

extern account_balance balance(account account_number);
extern account_balance deposit(account account_number,
                               transaction_amount amount);

extern account_balance withdraw(account account_number,
                                transaction_amount amount);
```

*Code Example 5-1* Implementation header file named bank.h

```
/* balance.c */
#include "balances.h"
account_balance balance(account account_number)
{
    /* function logic */
    return account_balance;
}
```

*Code Example 5-2* Implementation source file named balance.c

```
/* deposit.c */
#include "balances.h"
account_balance deposit(account account_number,
                       transaction_amount amount)
{
    /* function logic */
    return account_balance;
}
```

*Code Example 5-3* Implementation source file named deposit.c

```
/* withdraw.c */
#include "balances.h"
account_balance withdraw(account account_number,
                        transaction_amount amount)
{
    /* function logic */
    return account_balance;
}
```

*Code Example 5-4* Implementation source file named `withdraw.c`

### *Using a Single ADL Module*

Given the implementation files above, the test designer can create a single ADL module containing specifications of all three functions. The module definition, then, follows the organization of the header file `balances.h`, which is used by all three functions.

```
typedef int account_balance;
typedef int transaction_amount;
typedef int account;

account_balance balance(account account_number);
account_balance deposit(account account_number,
                       transaction_amount amount)
semantics{
    return == @balance(account_number) + amount
};

account_balance withdraw(account account_number,
                        transaction_amount amount)
semantics{
    return == @balance(account_number) - amount
};
```

*Code Example 5-5* ADL file named `bank.adl` containing specifications of three functions.

Given the organization of the ADL module above, the designer simply needs to create a header file named `bank.h`, which would `#include` the implementation's header `balances.h`.

## Using an ADL Module Hierarchy

As an alternative to the single-module organization shown above, the designer can create a hierarchy of ADL modules by using the ADL `imports` clause. An ADL file can contain the specification of only a single module. However, that one file can “see” the constituents of other modules through the use of the `imports` clause.

Consider the four modules below, each in a separate ADL file:

```
file balances.adl:
    typedef int account_balance;
    typedef int transaction_amount;
    typedef int account;

    account_balance balance(account account_number);
```

*Code Example 5-6* ADL file named `balances.adl`

```
file deposits.adl:

import balances;

    account_balance deposit(account account_number,
                            transaction_amount amount)
    semantics{
        return == @balance(account_number) + amount
    };
```

*Code Example 5-7* ADL file named `deposits.adl`

```
file withdrawals.adl:

import balances;

    account_balance withdraw(account account_number,
                              transaction_amount amount)
    semantics{
        return == @balance(account_number) - amount
    };
```

*Code Example 5-8* ADL file named `withdrawals.adl`

```
file bank.adl:  
import deposits, withdrawals {  
}
```

*Code Example 5-9* ADL file named `bank.adl`

In the example above, the modules `deposits` and `withdrawals` import the module `balances`, and the top-level module `bank` imports both `deposits` and `withdrawals`.

The effect of the `import` statement is to make visible the contents of the imported modules in the importing modules. Both `withdrawals` and `deposits` can “see” the constituents of `balances`, and they can refer to those constituents as if they had been declared inside their own respective modules. Further, importing is transitive. The module `bank` can see the constituents of `balances` as a consequence of importing either `deposits` or `withdrawals`.

For this hierarchical module organization, the designer must create four header files—one `module.h` for each module.

### *Reasons for Using Multiple ADL Files*

The hierarchical example, above, requires that the test designer create four header files instead of the one used in the single-module solution. However, the designer may decide that the additional headers are worth the trouble, if the type definitions and function declarations in the modules will be used by other modules representing code beyond that contained in `bank.c`.

For instance, there may be a source file specifically for automated teller transactions; that code might not perform deposits and withdrawals immediately (such transactions may be verified and processed overnight). However the automated teller module does need access to real-time account balances, and a module named `atm` might therefore import `balances`.

The use of multiple ADL files therefore facilitates the re-use of specification components. It is up to the designer to weigh the benefit of simplicity (a smaller set of specification files) against the longer-range goal of reusability. The decision involves balancing the goals of the software (ad hoc code versus the construction of a large system for long-term use) and the needs of the test (special-situation testing versus the construction of a test suite to be used over years).

## Referencing TDD Files

TDD files describe data to use in the test of functions described in ADL files. However, the organization of TDD files does not have to follow exactly that used by the related ADL files. The designer may decide to use and reference multiple TDD files; he or she can also employ TDD restricted-content information and directive files, as well as general TDD files.

### The TDD `use` statement

The `use` statement permits the TDD author to organize TDD files in hierarchies of files, similar to the way the ADL `imports` clause creates transitively visible modules.

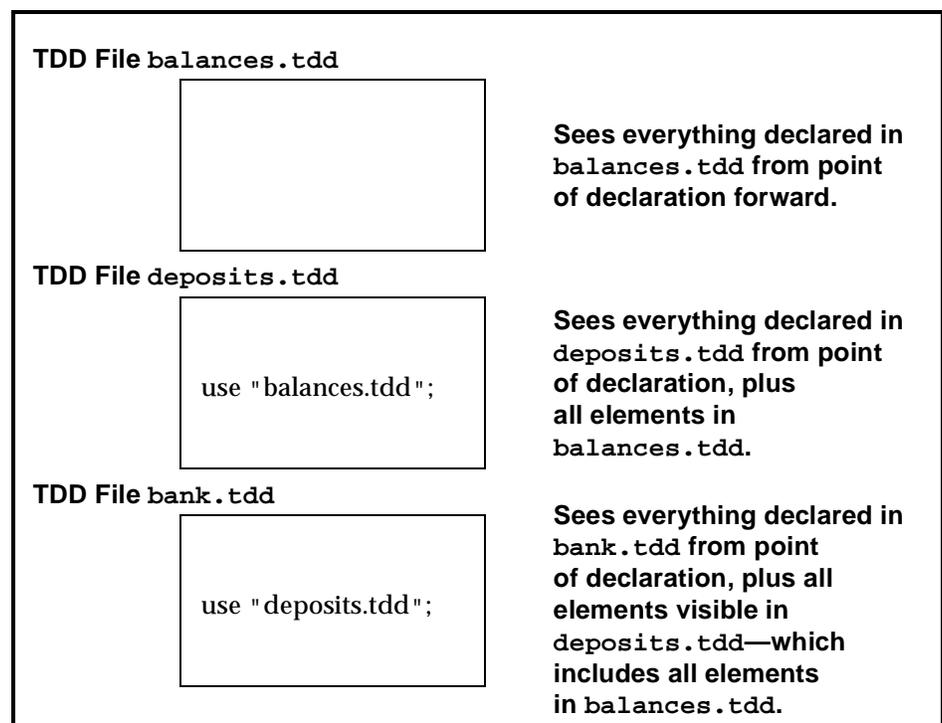
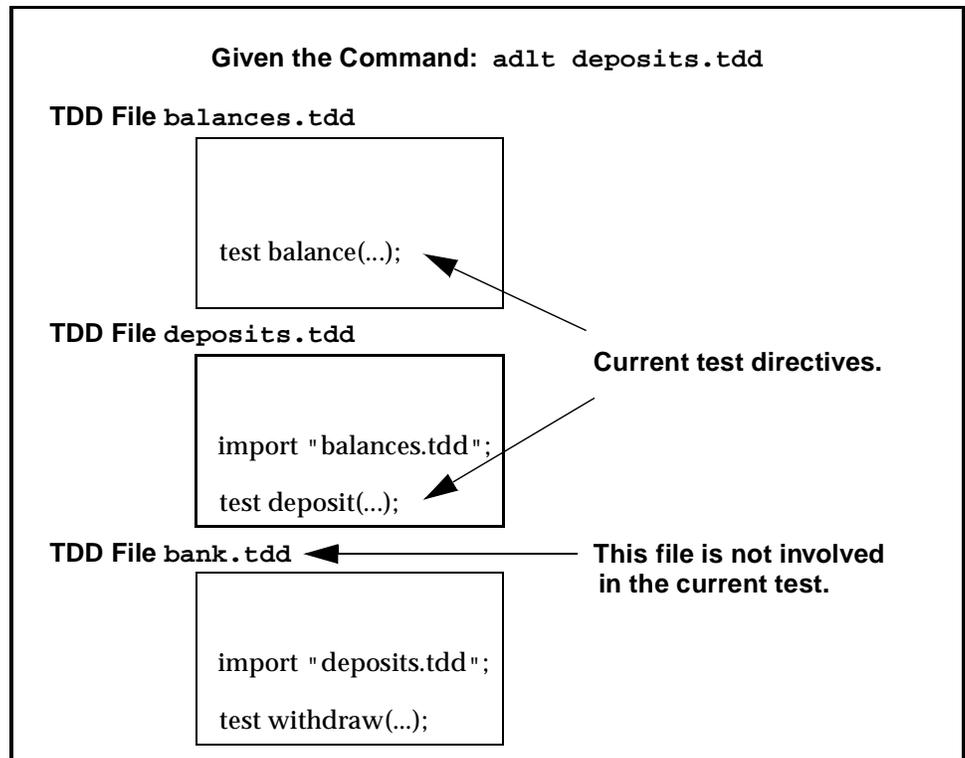


Figure 5-2 Illustration of transitive references among TDD files.

The effect of the TDD `use` statement is transitive. All the elements in the named file—properties, test variables, and test directives—along with any elements that were visible in the named file, are transitively visible in files containing the `use` statement.

### Strategies for Using Multiple TDD Files

When ADLT is run to process a TDD file, it is given the name of the TDD file containing the test directive or directives that will produce the tests currently of interest. That file is then the “innermost” TDD file in the current set—it is the one that can “see” all other files containing needed declarations but which is not itself referenced by any other files needed for the current test. The illustration below shows the relationship between the test directive and the file name given in the ADLT command:



The test designer can arrange TDD declarations in any import hierarchy that seems useful. The simplest is to use one TDD file per module under test. There are many other possible ways to structure TDD files. For instance, the designer could use multiple TDD files, all involved in the current test. Each project can decide upon the standards and conventions that best suit its needs. For a discussion of organizing principals, see “Organizing the Test Suite” on page 58.

### *Reasons for Using Multiple TDD Files*

The factors in deciding to use multiple TDD specifications are like those for ADL files: the balance of simplicity of the specification versus reusability of specification components. If a set of test variables will be useful in many testing situations (for example, for file names or integers used as incrementors, or any other common data type), the designer should probably place those TDD test variables in a separate file that will be “used” as necessary.

Some TDD files can define the architecture of the test, and provide general factories and associated relinquish functions. These then make up an archive of reusable testing code. Other TDD files can then define the individual tests in terms of these general factories. Over time, as users create libraries of reusable, referenced TDD files, they will also be creating libraries of factory and relinquish functions, thereby decreasing the effort involved in creating new ADLT test programs.

## *The Relationship Between ADL and TDD Modules*

Through the `import` statement, a TDD file gains visibility of all ADL type definitions and function names that are visible in the ADL files and target language source files thereby imported.

For example, given the following TDD statement in a file named `bank.tdd`:

```
import bank.*;
```

all ADL types and functions visible in package `bank` will also be visible in `bank.tdd`.

However, a TDD file cannot simply reference any arbitrary ADL module; the TDD file should describe data for one or more *related* ADL modules. Modules are related in ADL files through the ADL language `import` statement. The

ADL files therefore define a hierarchy of modules that are related for the purposes of documentation, specification, and testing. The purpose of the TDD file is to describe data for that same set of modules.

TDD references do not have to follow exactly the ADL hierarchy. However, the module *dependencies* in the TDD file should follow those in the ADL files. That is, the direction of the TDD `use` statement should be the same as the ADL `import`, as shown in the illustration below.

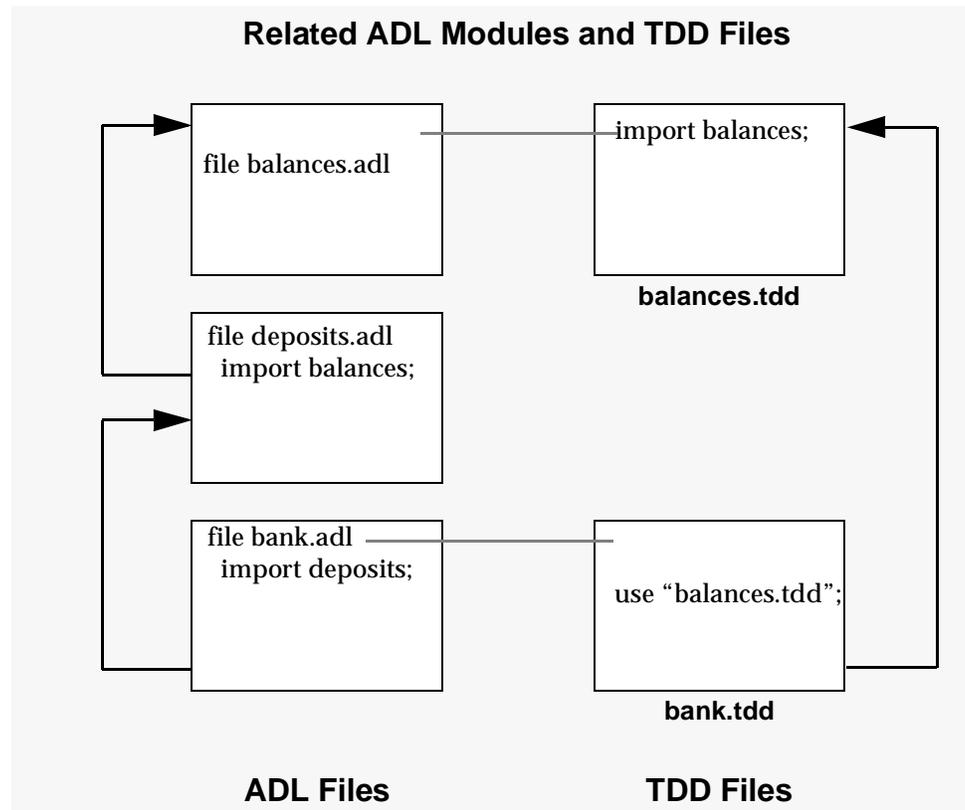


Figure 5-3 Sets of modules related in ADL and TDD files

In the illustration above, the use of the `use` statement in the TDD file is appropriate because it follows the direction of the dependencies in the ADL files. In the ADL files, the `bank` module imports the `balances` module, albeit indirectly. In the TDD files, the `bank` module also references the `balances` module. Because the TDD files follow the hierarchy used in the ADL files, the TDD `use` statement is legal. Note that the TDD hierarchy does not have to follow exactly the importing hierarchy in the ADL files. Only the direction of the dependencies among modules must be preserved across ADL and TDD files.

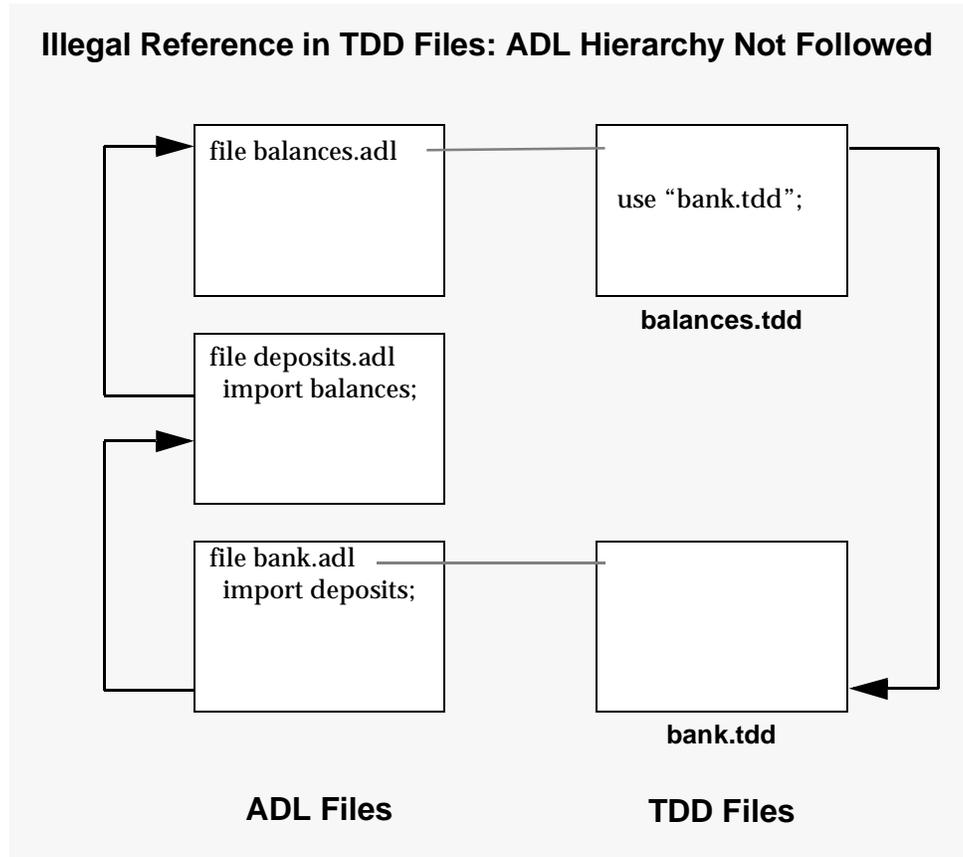
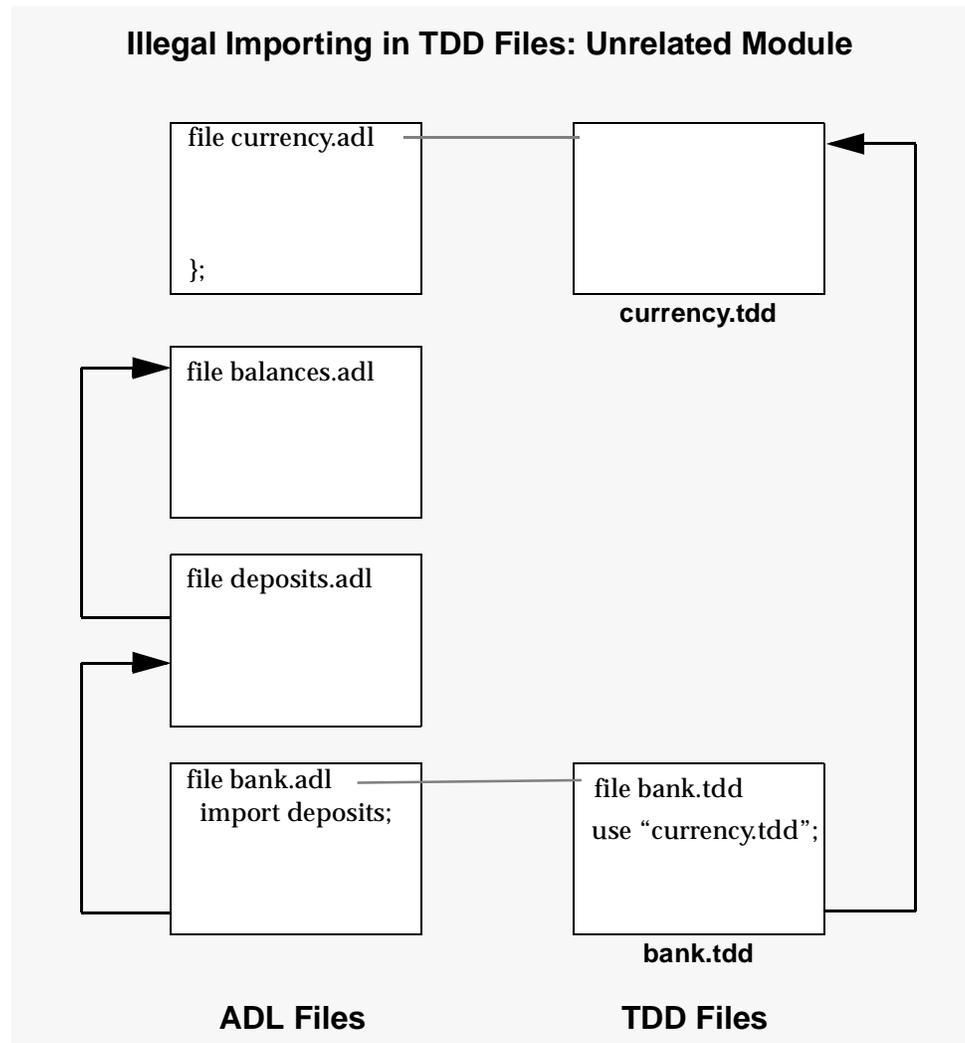


Figure 5-4 Illegal use statement in balances.tdd

The TDD use statement in the illustration above is not legal. Here, the TDD file balances.tdd tries to reference bank.tdd. While the bank and balances modules are indeed related in the ADL files, the TDD files do not follow the ADL import hierarchy. The file bank.tdd should use balances.tdd, not vice versa.



*Figure 5-5* Illegal import statement in `bank.tdd`. The module `currency` is not related to the bank module.

The illustration above shows another illegal use of the `use` statement in TDD files. Here, the file `bank.tdd` tries to reference a TDD file that describes an unrelated module. The module `currency` is not part of the ADL hierarchy of modules, and the TDD file may not `use` declarations that describe an unrelated

module. The rule prevents confusion; if module `currency` defines an identifier that is also defined by module `deposits`, the meaning of that identifier would be unclear in `bank.tdd`.

## *Organizing the Test Suite*

This section discusses approaches to organizing the elements of the ADLT test program. Its goal is to help the user create specifications and test code which meet the needs of specific testing situations and which also contribute, over time, to building a reusable library of testing components.

This sections discusses:

- Principals of Specification Structure (See page 58.)
- Recommended Directory Structures (See page 59.)
- The UNIX File System Example (See page 60.)
- Customizing the Test Suite Structure (See page 62.)

### *Principals of Specification Structure*

The overall goal is to write specifications and test code so that, over time, the user is constructing an archive of components that can be reused in multiple testing situations. As individual testing needs arise, the archived components—ADL and TDD specifications, accompanying auxiliary, factory, and relinquish functions—can be reused to create the currently needed test.

The following principals will support reusability:

- Data types used in more than one software component should be described in ADL and TDD files that are then imported and referenced as needed.
- It is often useful to define more than one TDD test variable for a given data type: one for normal values and one for abnormal/erroneous values.
- The user should balance the use of several test variables over the convenience of using refinements. The issue is whether to use a single test variable, one that contains a small amount of special case logic, and then to refine that single test variable; or to specify multiple test variables. If the factory function requires a great deal of special casing, multiple test variables are advisable.

- The user should consider reusability when defining auxiliary functions. The building of a library of auxiliary functions will shorten the effort needed to create new tests. Note that the definition of reusable auxiliary functions may require iteration, as the range of needed testing situations become clear.

Once specifications are designed for general, long-range reuse, they may be modified for specific test needs through the following ADLT facilities:

- Refinement of TDD test variables (in a separate TDD or TDR file)
- Use of test directives that use the refined test variables (in a separate TDD or TDR file)
- Addition, deletion, or modification of optional user-supplied code to customize the test program and test result report— test initiation and termination routines, wrappers around the function under test, user-defined print formatting functions. (See the *ADL Translator Programmers Guide*.)

### *Recommended Directory Structures*

To support the twin goals of reusability and per-test customizing, the following general directory structure is recommended:

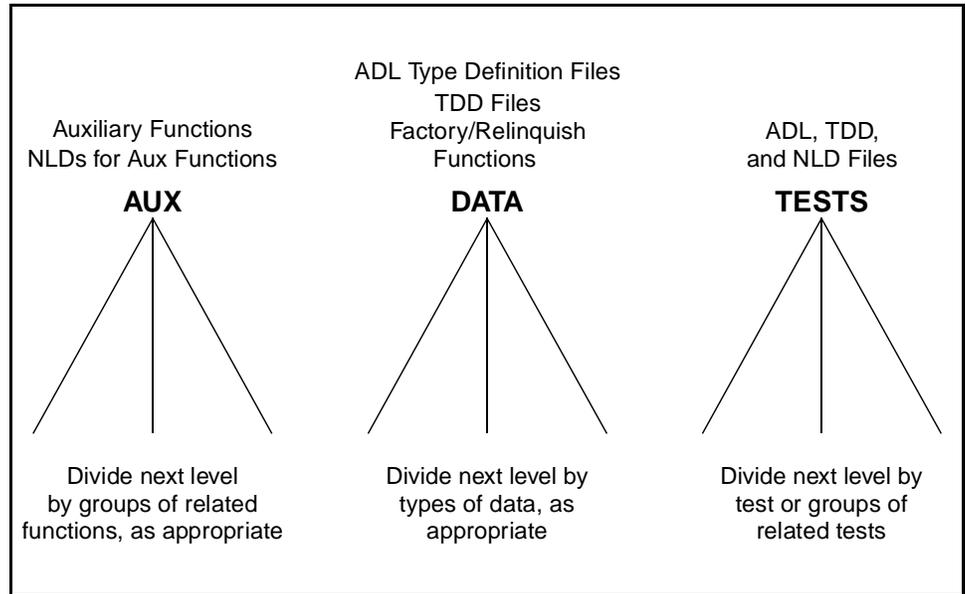


Figure 5-6 Recommended ADLT directory structure: AUX, DATA, and TESTS.

The purpose of the recommended structure is to separate components designed for re-use (type definitions, auxiliary functions, test variables and their accompanying factory/relinquish functions) from those elements used in particular tests (the ADL, TDD, and NLD files).

The **AUX** and **DATA** directories are kept separate due to the different organizing principles for the next level of directories. Auxiliary functions should probably be grouped by type of module they address (for instance, those related to file systems or to memory management); data is best organized by related types.

In addition, the **TESTS** directory can be further divided to facilitate re-use of ADL and NLD specifications. For example, one branch could be reserved for ADL modules imported by many groups of tests.

### *The UNIX File System Example*

The recommended directory structure is used in the sample, partial test suite for the UNIX file system. The example contains the outline of the suite that will test UNIX file system calls.

The UFS example is structured as shown below:

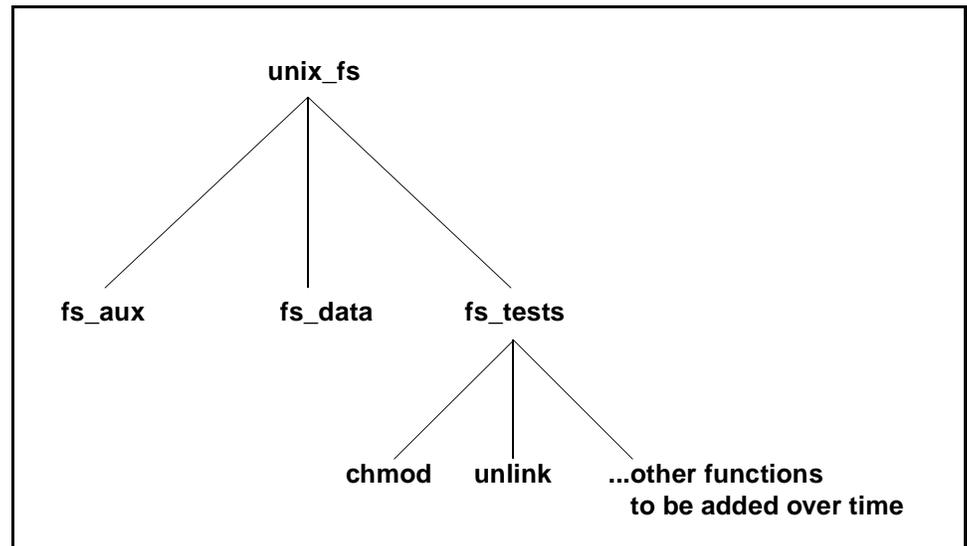


Figure 5-7 Directory structure for the UNIX file system example.

Beneath the root directory for the set of suites, `unix_fs`, the next level of directories follows the recommended structure:

- `fs_aux` holds all the auxiliary functions for the set of suites.
- `fs_data` holds the TDD files containing master test variables and their corresponding source files for factory and relinquish functions.
- `fs_tests` is a node which organizes the suites for individual system calls related to the UFS. The example provides tests for `chmod` and `unlink`. This structure permits other directories to be added as tests are developed for the additional UFS system calls.

Note that the `fs_data` directory also contains a program which sets up directories and files needed for the test suites. This program, `setup.c`, must run as root in order to set permissions; therefore, its logic could not be placed in an initialization routine.

## *Customizing the Test Suite Structure*

Obviously, the user may find other effective structures for their test suites besides the one recommended here. The makefile generated by ADLT “expects” to find files with given names in anticipated places. However, the user can modify the emitted makefile if and where user file names do not conform to ADLT defaults. Specifically, the makefile contains variables whose assignment may be changed by the user.

---

**Attention Reviewer!** – These variables are not yet fully defined.

---

*Code Example 5-10*

This chapter describes how to use the ADLT-generated tests via the Test Environment Toolkit. It provides a mapping of ADLT concepts to TET concepts and shows how to make the ADLT test program run with TET. The chapter also provides a description of test result report formats available for ADLT test programs running under TET.

### *Mappings Between ADLT and TET*

The paradigm used by ADLT testing is slightly different from that used by traditional conformance testing. Traditional testing involves specifying tests by their inputs and outputs. ADLT separates the specification of behavior—the ADL file—from the specification of the data and the test—the TDD file. Nonetheless, the ADLT and traditional testing models are similar enough to permit the use of ADLT for conformance testing under TET. This section summarizes how ADLT inputs, outputs, and tests, map to those used in conformance testing under TET.

#### *Test Units*

TET defines several levels of testing abstraction. The smallest level of abstraction, or unit of testing, is the test purpose. A test purpose in TET is a set of one or more assertions for which a single result will be reported. In ADLT-generated tests, each test instance maps to a TET test purpose. A test instance is defined as one invocation of the function(s) under test, using input data from one point on the ADLT input grid, followed by evaluation of all ADL

assertions about the behavior of the function(s). For each input gridpoint, the ADL generated test evaluates each assertion; then, when all assertions have been evaluated, it assigns a test result for the gridpoint.

The next level of testing abstract TET defines is the invocable component. TET calls this level “invocable” because it this level of abstraction can be explicitly invoked for testing from the command line. TET permits any number of test purposes to be encapsulated in an invocable component. ADL-generated tests map each test purpose to a single invocable component. This means that it is possible to select one or more specific test instances from the command line, and have the assertions of the function(s) under test evaluated against only those test instances.

### *Test Results*

As the ADL-generated test evaluates each assertion for each test instance, that assertion evaluation is assigned a result. Those results are recorded as interim results in the TET journal file. When all assertions for a test instance have been evaluated, TET uses the tiered relationship of test results to determine the aggregate result for that test instance.

In addition, the test author can set any desired result code by calling the functions *adlResult* or *tddResult*, which are part of the ADL library. These function may be called from a provide function, when it sets the user result for all assertions at that grid point, or from an auxiliary function, when it sets the user result for that assertion at that grid point, or even from within a test definition in a TDD file. The result code for an assertion is calculated by combining the automatic result code (calculated as above) and any user-set result codes.

### *Test Directives and Scenarios*

In ADLT, the test program is defined by the test directive. For each test directive in the TDD file, ADLT generates one test program.

In TET testing, each test suite is defined as a scenario in the TET scenario file, *tet\_scen*. The scenario file contains sequences of invocable components (ICs) that can be invoked under a scenario name. When ADLT is run under TET, the invocable component numbers, a range beginning with 1, are used to represent the ICs.

When ADLT is run against a TDD file, the program emits the TET scenario file, `tet_scen`. This file maps ADLT test directives to TET test suite scenarios. For each ADLT test directive, the defined scenario entry runs the test program over all invocable components. This name of the assertion is the same as the name assigned to the test clause in the TDD file.

For example, given the following labeled TDD test directive:

```
withdraw_general: test withdraw(account acct, int amount);
```

ADLT will generate the following scenario name:

```
withdraw_general
```

And, given the following unlabeled TDD test directive:

```
test withdraw(account acct, int amount);
```

ADLT will generate the following scenario name:

```
withdraw1
```

See the section “The Generated TET Scenario File” below, for a detailed discussion about the contents of the generated TET scenario file.

## *Directory Structure and Required Files*

Neither ADLT nor TET require the organization of ADLT files in any particular directory structure. However, the user may find it useful to separate shared, master ADL specification and TDD files from TDD files that define tests. For a full discussion of approaches to organizing ADLT files and directories, see Chapter 5, “The Structure of the Test Suite.”

In order to run ADLT test programs under TET, the scenario file `tet_scen`, and the files `tetbuild.cfg`, `tetclean.cfg`, and `tetexec.cfg` file must exist. Sample versions of these are available in the `ADL2 lib` directory. ADLT generates a default `tet_scen` file. For information on setting up a `tetexec.cfg` file, see the *Test Environment Toolkit Architectural, Functional, and Interface Specification*.

## *Environment/Configuration Variables*

When running ADLT test programs under TET, the following environment variables must be defined:

- `TET_ROOT` - An environment variable set to the main TET directory.

- ADL2HOME - An environment variable set to the directory in which the ADLT program was installed.

The following table contains a complete list of the environment variables that are used by the test program.

Name	Legal Values	Default Value	Description
ADL2HOME	a directory path name	none must be defined	Specifies the path of ADL system release directory

For information regarding TET configuration and communication variables, see the *Test Environment Toolkit Architectural, Functional, and Interface Specification*.

## *Generating Test Programs for TET*

Once the ADL specification and TDD files have been created, there are two main steps in generating an ADLT test program for use with the TET test harness:

- 1. Run ADLT**
- 2. Make the test program**

The scenario file is emitted by default, and is known to the generated makefile.

## *Running the Test Program under TET*

To run ADLT test programs under TET, use the following command:

```
tcc -e test_dir [scenario]
```

Where:

*test\_dir*

is the pathname of the directory that contains the ADLT test programs, given relative to the location of TET\_ROOT. If a pathname is not specified, the user cannot specify a scenario. In the

absence of a pathname, `tcc` assumes the TET scenario file, `tet_scen`, is in the current working directory and runs the scenario `all`.

*scenario*

is the name of a scenario in the TET scenario file, `tet_scen`. If the user wants to specify a scenario, he or she must provide a value for `test_dir`. If no scenario is specified, `tcc` runs the scenario named `all`.

ADLT generates a default `tet_scen` file for convenience. The contents of this scenario file and a description of how to customize it are provided in the section below.

### *The Generated TET Scenario File*

The TET scenario file contains scenarios composed of scenario names, comments, and command lines for the ADLT test program.

### *Modifying the TET Scenario File*

To add scenarios to the file, or modify existing ones, the user will need to supply the test program command in a format that is understood by ADLT.

Here is the general format for the ADLT command in the `tet_scen` file:

```
/ testname { range_list }
```

Where:

*testname*

is the name of the ADLT test program.

*range\_list*

specifies a range or list of invocable components to run.

Where *n* is the identifier of the invocable component to run, *range* can be specified as:

- 0           to run all invocable components in assertion-oriented mode
- all         to run all invocable components in data-oriented mode
- n* [ , *n*]\* to run individual invocable components in data-oriented mode
- n* - *n*     to run a range of invocable components in data-oriented mode

For example, to create a scenario to test the first 10 invocable components for the `withdraw` function in data-oriented mode, the user can create a new scenario called `withdraw_first_10` by adding the following lines to the scenario file:

```
withdraw_first_10
    "directive withdraw_general:test withdraw(acct, amount)"
    /withdraw_general_data {1-10}
```

To run this test, enter the following `tcc` command line:

```
tcc -e my_directory withdraw_first_10
```

## *Customizing the Test Program*

---



In many ways, ADLT is an open testing tool. Since the ADLT compiler produces source code which can be linked with user-supplied code, there are several points of interface between built-in and user-supplied functionality.

ADLT is a tool designed for programmers; it permits the programmer to customize the generated test through the addition of optional functions which operate at various points in the test-generation process and runtime test program. Such optional functions can control the environment surrounding the entire test or surrounding a single invocation of the function under test; can modify the content and format of generated test result reports; and can change the languages in which documentation is produced.

This chapter gives an overview of ADLT customization features. It introduces the mechanisms available to programmers, and contains references to more detailed information in the ADLT document set.

The programmer can modify the ADLT test and test report through the following means:

- **Initialization and Termination Routines**

It is sometimes necessary to establish elements in the environment of the runtime test: to create files, initialize libraries, set the values of global variables, or other factors in the test environment. To permit the user to establish this environment, ADLT contains a mechanism for user-defined initialization and termination functions. For details, see the language reference manual for the language binding you are using.

- **User-Defined Iterators for ADL Quantified Expressions**

The ADL language includes advanced features for existential and universal quantification; for example,

```
exists (int j : int_range(1, 10)) {
    P(j) };
```

which is an ADL expression using existential quantification over a compact set of integers; and

```
forall (deftype_t x : def_set()) {
    Q(x) };
```

which is an example of universal quantification over a user-defined set. The first example uses a set constructor defined in the predefined module `ADL_Standard`, which is implicitly available to all ADL modules. The second quantified expression uses a user-defined set constructor for a user-defined type. ADLT permits the user to define iterators for any type that is legal in the ADL specification. For details, see the language reference guide for the language you are using.

- **Controlling the Level of Expression-Evaluations on Reports**

By default, the ADLT test result report shows the evaluation, true or false, of each assertion. Alternately, the user may wish to see evaluations of lower-level expressions and subexpressions, down to the level of a single variable. To control the expression-evaluation level, users may set a command-line switch when running the test program, or may set the TET configuration variable `ADL_RPT_DETAIL`.

- **Inserting Messages into Test Result Reports**

Users may wish to place text in test result reports. To implement this facility, the ADLT library supplies several functions that can be called from any user-defined functions, such as auxiliary, provide/relinquish functions; initiation and termination routines, and/or other user-supplied code.

- **Setting the Test Result Code**

Users may set a test result code based upon some condition in the execution environment of the test program. The ADLT library contains functions to check configuration variables and set the test result.

- **Modification of the Generated Makefiles**

---

The ADLT-generated makefiles conform to some reasonably simple default for file names and directories. It is expected that only basic suites will be structured in accordance with the defaults. The makefile is constructed so that the user can easily identify and modify makefile variables to indicate whatever files and directory structures are most effective for the testing needs at hand.

For details, see “Recommended Directory Structures” on page 59 and “Customizing the Test Suite Structure” on page 62, both in Chapter 5, “The Structure of the Test Suite,” in this book

- **Modification of the Generated TET Scenario File**

When run with the TET option, ADLT generates a scenario file. This file contains scenarios for reports in two modes, with defaults for the invocable components to be run. The user can freely modify the scenario file in accordance with TET and ADLT conventions.

For details, see “Modifying the TET Scenario File” on page 67 of Chapter 6, “Using ADLT with TET,” in this book.



## *Glossary*

---

### **ADL**

See “Assertion Definition Language.”

### **ADLT**

Assertion Definition Language Translator. ADLT is a system that assists in the functional testing of software components. It is a compiler that generates test programs based upon the input of formal functional and test-data specifications. ADLT-generated test programs call the components under test to determine if they are compliant with their specification. The system also generates natural-language documentation of specification inputs and generated tests. ADLT’s goal is to automate, as much as possible, current best practices in unit testing.

### **Assertion**

An ADL assertion is a boolean expression that describes the normal and/or exceptional behavior of a function. ADLT evaluates each assertion after execution of the function. The function has executed properly if all of the assertions are true.

### **Assertion-Checking Functions**

Assertion-checking functions are C-code functions generated by ADLT. The functions have embedded in them the behaviors specified by the ADL semantic statements. When compiled and linked into the ADLT test program, the assertion-checking functions compare the actual behavior of the function under test to the behavior specified in the ADL assertions.

---

**Assertion Definition Language**

Assertion Definition Language is a formal grammar for declaring the constituents of a software module—its data types, variables, constants, and functions—and for specifying the intended behaviors of functions within the module.

**Auxiliary Constituents**

Auxiliary constituents are types, variables, constants, and functions that are used in the ADL specification to help describe the intended behavior of module functions. Auxiliary constituents are implemented separately from the module being specified. Declarations of auxiliary constituents are visible only within ADL semantic bindings and assertions.

**Auxiliary Functions**

Auxiliary functions are functions that are declared in the ADL specification but implemented separately from the module being specified. The functions typically examine some aspect of the implementation's execution state or surrounding environment. Auxiliary functions can be new functions defined especially for ADLT testing purposes, or they can be already-compiled functions, such as library and system calls.

**Constituent**

A constituent is an element of an ADL module. There are three types of constituents: type definitions; variables and constants; and functions.

**Environment Parameter**

An environment parameter is a TDD test variable used to set up a global state or condition that can affect the operation of the function under test.

**Input Grid**

The input grid is the set of test data input points generated by ADLT for a specific test of a function. The number of points on the grid is determined by the TDD file specification of the test variables in a given test directive. The test variables include those that describe function parameters, as well as any optional environment parameters. The input grid is calculated as the cross product of all values of all properties of all test variables.

**Literal Test Variables**

Literal test variables are TDD descriptions of test data defined with numeric and character constants. The ADLT test program uses the specified literal values as inputs to the function under test.

---

**Master Test Data Definitions**

Master test data definitions constitute the most complete description of TDD test variables or properties. Their purpose is to document the full set of important values that should be examined for a particular data element. Master definitions also determine the functionality of the provide function. The provide function must be ready to interpret the full set of values specified by the master definition of a test variable. (Also see “Test Data Refinements.”)

**Module**

An ADL module is a programmer-defined software component. The module may contain any variables, constants, type definitions, and functions that the programmer considers to be related for the purposes of specification, documentation, implementation, and/or testing.

**Natural Language Dictionary**

The Natural Language Dictionary is a glossary of translations for elements that make up ADL assertions. ADLT uses these translations to augment natural language descriptions that appear in generated Natural Language Specification and Test Specification documents.

**Natural Language Specification**

The Natural Language Specification is an ADLT-generated document that contains natural language descriptions of functions described by ADL semantics.

**NLD**

See “Natural Language Dictionary.”

**Property**

A property is a significant, independent aspect of a TDD test variable. Properties are defined as having sets of symbolic values—adjectives that characterize the actual values the test variable should take on during the ADLT test of a function.

**Provide Functions**

Provide functions are programmer-written functions that provide instances of data elements to be used in the test of module function. In arguments passed to the provide function, the ADLT test driver describes the combination of properties to be instantiated. The provide function creates or locates an instance of the data element with a value that agrees with the description in the properties. It then supplies that value to the driver to be used as part of the current ADLT test instance.

---

**Relinquish Functions**

Relinquish functions are programmer-written functions that relinquish any resources allocated by a corresponding provide function. If the provide function created a new instance of a data element, for example, the relinquish function would normally deallocate the element.

**Scenario**

A scenario is a TET test description.

**Semantics**

Semantics are ADL statements that describe the intended behavior of functions in the module. ADLT translates semantic statements into assertion-checking functions, which determine whether the function under test passes or fails the ADLT test program.

**Symbolic Value**

A symbolic value is an adjective that characterizes the actual values a TDD test variable should take on during the ADLT test of a function.

**TDD File**

See “Test Data Description File.”

**Test Data Description File**

The test data description file (TDD file) is a text file containing TDD-language descriptions of data to be used during ADLT tests. In the TDD file, data are described with either literal values or symbolic properties. The TDD file also contains test directives, which direct ADLT to generate a test for specific functions using specific test data descriptions. (Also see “Test Data Information File” and “Test Directive File.”)

**Test Data Information File**

A test data information file is a restricted form of a TDD file. It may contain only declarations of properties and test variables, as well as their refinements. It may not contain test directives. (Also see “Test Directive File.”)

**Test Data Refinements**

Test data refinements are definitions of TDD test variables and properties that specify a subset of the values described in master definitions. Refinements allow the TDD file author to limit the number of tests implied by the master definition of a test variable or property. Since particular testing situations can require more targeted test input values, refinements let the author specify a smaller set of values, as needed. (Also see “Master Test Data Definitions.”)

---

**Test Directive**

A test directive is a TDD-language statement that directs the ADL Translator to generate a test for the function given in the statement. The test directive also indicates the TDD test variables that should be used to generate test data inputs to the test.

**Test Directive File**

A test directive file is a restricted form of a TDD file. It may contain only refinements of test variables and properties, as well as test directives. It may not contain declarations of master properties or test variables. (Also see “Test Data Information File.”)

**Test Driver**

A test driver is a C-language source file emitted by ADLT. The program generates a test driver for each TDD file test directive statement. When compiled and linked into the test program, the test driver iterates over the grid of test data inputs implied by the test variable properties in the TDD file. The driver calls provide functions to get actual values for each test variable, and it passes those values to the assertion-checking function for the function under test.

**Test Instance**

A test instance is one invocation of the function under test with a discrete combination of input values; that is, at one point on the generated input grid. (Also see “Input Grid.”)

**Test Result**

A test result is the overall determination of whether a function passes or fails an ADLT test program. The function passes if all specified ADL assertions are true at all tested points on the input grid. (Also see “Input Grid.”)

**Test Specification**

The Test Specification is an ADLT-generated document containing natural language descriptions of an ADLT test.

**Test Variable**

A test variable is a TDD-language definition of a data element to be used in an ADLT-generated test. Test variables are defined with either literal values or symbolic properties.

**TET**

Test Environment Toolkit, a test harness in general use in UNIX environments.

---

**Translation**

An translation is a natural language phrase, sentence, or paragraph specified in an NLD. Each translation describes a programmer-defined element of an ADL assertion. When generating documentation, ADLT substitutes the given translation for each occurrence of the corresponding ADL assertion element.

**XPG**

X/Open Portability Guide. XPG is the specification produced by X/Open for a Common Application Environment (CAE).

# Index

---

## A

### ADL

- annotated functions, 19
- auxiliary functions, 6
- bindings
  - normal and exception, 19
- C-like syntax, 18
- group expression, 19
- imports statement, 49, 55
- multiple specification files, 50
- quantified expressions, 70
- reserved words, 19

### ADLT

- as a test compiler, 4
- compiler, generated outputs, 28
- customization features, 45, 59, 69
- documentation inputs and outputs, 10
- file naming conventions, 42
- generation of documentation, 10
- makefiles, 31, 32
- mappings to TET, 63
- test generation, 6
- test program
  - elements, 44
  - elements to be compiled, 44
  - elements to be linked, 45
  - functionality overview, 44

- test suites, 39

- annotated functions

  - as module constituents, 19

- assertion-checking functions, 30

- assertions

  - in ADL, 21

- auxiliary functions

  - in ADL, 6, 24

## B

- bindings

  - normal and exception, 20

## C

- call-state operator, 19

- customization features

  - declarations in the auxiliary header file, 43

  - in the generated makefiles, 62

  - initialization and termination

    - routines, 45

  - messages in test result reports, 70

  - overview, 69

  - print formatting functions, 45, 59

  - setting result codes, 70

  - wrappers around the function under test, 45, 59

---

## D

documentation  
and NLD input, 10, 34  
inputs and outputs, 10

## E

environment variables  
ADL\_OPTIONS, 70  
under TET, 65  
exception operator, 20  
expression evaluation level, 70

## F

file naming conventions  
for ADLT, 42

## G

group expressions, 19

## I

implication operator, 20  
imports statement  
transitivity among ADL files, 49, 50  
initialization and termination  
routines, 59, 69  
input grid, 27  
inputs  
to ADLT compiler, 28

## L

logical equivalence operator, 20

## M

makefile  
customization features, 70  
makefiles  
customization features, 62  
generated by ADLT, 8, 31, 32, 46  
messages

in test result reports, 70

module

constituents, 13  
organization, 46

multiple specification files, 46

ADL, 50

TDD, 51

## N

NLD

as a glossary of terms, 10, 34  
as optional input to ADLT, 11, 35

NLS file

generated by ADLT, 10, 23, 33

## O

operators

ADL call-state, 19

ADL exception, 20

ADL logical equivalence, 20

outputs

generated by ADLT compiler, 23, 28

## P

print formatting functions, 45, 59

## R

relinquish functions, 7

reserved words

in ADL, 19

result codes

user settings, 70

reverse implication operator, 20

## T

TDD

module statement, 26

multiple specification files, 51

use statement, 27, 51, 55

use statement, illegal, 56

---

- test directives, 7, 26, 64
- test driver, 30
- test generation, 6
- test instance, 3, 63
- test program
  - elements, 44
  - functionality overview, 44
- test result reports, 9
  - customization features, 70
- test suites
  - ADLT, 39
  - customizing the structure and files, 62
  - organization of, 58
  - recommended directory structures, 59, 60
  - TET, 65
  - the UNIX file system example, 60
- TET
  - scenario file, 71
  - test suites, 64
- TS
  - generated by ADLT, 10, 34

## U

- use statement
  - illegal, 56
  - in TDD, 27
  - transitivity among TDD files, 52

## W

- wrapper around the function under test, 45, 59

