UC Irvine
ICS Technical Reports

Title
ADLscope : an automated specification-based unit testing tool

Permalink
https://escholarship.org/uc/item/8074g9vc

Authors
Chang, Juei
Richardson, Debra J.

Publication Date
1998-08-10

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

SL BAR
Z
699
C3
no. 98-26

ADLscope: an Automated Specification-based Unit Testing Tool

Juei Chang and Debra J. Richardson

Department of Information and Computer Science
University of California, Irvine, CA 92697

Technical Report 98-26

August 10, 1998

Abstract

Specification-based testing is important because it relates directly to what the program is supposed to do and can detect certain errors that are often not detected by traditional code-based testing techniques such as branch coverage and statement coverage. We have developed an automated testing tool, called ADLscope, that utilizes the formal specification of a program unit as the basis for test coverage measurement. A tester uses ADLscope to test Application Programmatic Interfaces (APIs) written in the C programming language. The API must be formally specified in the Assertion Definition Language (ADL), a language developed at Sun Microsystems Laboratories. The tester uses ADLscope to generate coverage conditions from a program's ADL specifications. When the API is tested, ADLscope automatically measures how many of the coverage conditions have been covered by the tests. An uncovered condition usually means that certain aspects of the specification have not been thoroughly exercised by the implementation. The tester uses this information to develop new test data that exercise the uncovered conditions. In this paper, we focus on the following aspects of ADLscope: the design and implementation of ADLscope and the specification-based coverage metrics used in ADLscope.

1 Introduction

This paper describes a tool called ADLscope that automates the process of measuring a program's test coverage with respect to its specification. We summarize ADLscope's contributions as follows:

- *ADLscope is a formal specification-based testing tool.* Specification-based testing relates directly to what the program is supposed to do and can detect defects in the software that are not detected by traditional code-based testing techniques. ADLscope guides the testing activity based on an ADL specification of the units to be tested. ADL is a formal specification language developed at Sun Microsystems Laboratories to describe behavior of functions in the C programming language.
- *ADLscope is a test coverage tool.* ADLscope measures coverage of program units with respect to their ADL specifications. Measuring test coverage provides testers an estimate of test adequacy. The advantage of ADLscope over other test coverage tools is that ADLscope measures specification coverage as opposed to just code coverage.
- *ADLscope facilitates test selection.* Any uncovered coverage condition reported to the tester usually means that certain aspects of the specification have not been tested adequately. This forces the tester to select additional test data to cover those aspects.
- *ADLscope facilitates unit testing.* Although system testing is important, alone it is insufficient. Moreover, in practice, it often involves just "poking-around." Unit testing is more rigorous and is also an essential in assuring quality of reusable software packages.
- *ADLscope facilitates API testing.* ADLscope is most useful for conformance testing of an API. Code-based coverage techniques are generally not applicable in conformance testing since higher code coverage does not imply that an implementation actually conforms to its specification. Moreover, many code-based coverage techniques require the source code, which is often not available for conformance testing because of proprietary issues, whereas ADLscope only requires the object code or compiled libraries.

In Section 2, we provide an overview of ADL and its companion tool, the ADL Translator (ADLT). In Section 3, we describe the design and implementation of ADLscope. In Section 4, we present the coverage metrics used in ADLscope. In Section 5, we discuss related work in this area of research. In Section 6, we summarize our contributions and discuss future work.

2 Overview of ADL and ADLT

2.1 ADL

In an ADL specification, a C function is accompanied by a set of assertions that describe the intended behavior of the function. The assertions are based on first order predicate logic and use the same type system as C. Sharing the same type system as C allows tools to easily map an entity in the implementation domain to the specification domain, and vice versa.

Figure 1 shows the C API of a stack abstract data type. The API is provided through a C header file. Figure 2 shows the ADL specification of the stack.

Here we will describe a few ADL constructs that will help the reader understand the stack specifi-

```
#ifndef STACK_H
#define STACK_H

#define MAX_STACK_SIZE 100
#define STACK_UNDERFLOW 1
#define STACK_OVERFLOW 2

typedef struct stack {
   int curr;
   float buf[MAX_STACK_SIZE];
} *stack;

extern int stack_errno;

extern stack new_stack();
extern int empty(stack s);
extern float top(stack s);
extern void push(stack s, float f);
extern void pop(stack s);

#endif
```

Figure 1: *stack.h*: API of the stack ADT

cation. Refer to the ADL Language Reference Manual [Sun96] for more information.

- The *auxiliary* section contains entities that are not part of the API but are needed by the specification. It usually contains declarations of temporary variables and utility functions. These utility functions are also called *auxiliary functions*.
- The *return* keyword refers to the return value of the function.
- The *call-state operator* "`@`" evaluates an expression before the function is called. All other expressions in the specification are evaluated immediately after the function returns.
- A *binding* (*name* := *expr*) binds an expression to a name. A binding is not an assignment. The name is simply a shorthand used to avoid rewriting a long expression.
- `normal` and `exception` are special bindings that the specifier can use to indicate whether an assertion describes the normal behavior of the function or some exceptional behavior of the function.

2.2 ADLT

ADLT, developed at Sun Microsystems Laboratories, is a tool that compiles an ADL specification and a high-level description of test data into a test program. The generated test program embeds an automated test oracle that reports to the tester whether an implementation conforms or violates its ADL specifications for a given test case. We provide a brief description of ADLT here. Refer to [SH94] for more information about ADLT.

ADLT is used in two stages: the compile stage and the run stage. In the compile stage, ADLT takes a function's ADL specification and its *Test Data Description* as input. The Test Data Description is a high-level specification of the test data. It allows the tester to specify the intention, the data types, and the enumeration of test data without supplying the actual values for the test data. The tester can provide the actual values for the test data in later stages of the testing process. ADLT generates *Assertion Checking Functions* as output. Assertion Checking Functions are C functions that evaluate the ADL specification to determine whether a test has conformed or violated the specification; they serve the purpose of the test oracle. The Assertion Checking Func-

```
module stack {
  typedef ... stack;   /* declare stack as an opaque type */
  const int MAX_STACK_SIZE;
  const int STACK_UNDERFLOW;
  const int STACK_OVERFLOW;
  int stack_errno;

  auxiliary {
    stack prev;
    int size(stack s);
    stack clone(stack s);
    float elementAt(stack s, int index);
  }

  stack new_stack()
  semantics { size(return) == 0 };

  boolean empty(stack s)
  semantics {
    prev := @clone(s),
    return == ((@size(s) == 0)? true: false),
    size(s) == @size(s),
    forall (int i: int_range(0, @size(s) - 1)) {
      elementAt(s, i) == elementAt(prev, i) }
  };

  float top(stack s)
  semantics {
    exception := @empty(s),
    normal := !exception,
    prev := @clone(s),

    exception --> (stack_errno == STACK_UNDERFLOW),
    size(s) == @size(s),
    forall (int i: int_range(0, @size(s) - 1)) {
      elementAt(s, i) == elementAt(prev, i) },

    normally {
      return == elementAt(s, size(s) - 1) }
  };

  void push(stack s, float f)
  semantics {
    exception := @(size(s) == MAX_STACK_SIZE),
    normal := !exception,
    prev := @clone(s),

    exception --> (stack_errno == STACK_OVERFLOW),
    exception --> (size(s) == @size(s)),

    forall (int i: int_range(0, @size(s) - 1)) {
      elementAt(s, i) == elementAt(prev, i) },

    normally {
      size(s) == @size(s) + 1,
      elementAt(s, @size(s)) == f }
  };

  void pop(stack s)
  semantics {
    exception := @empty(s),
    normal := !exception,
    prev := @clone(s),

    exception --> (stack_errno == STACK_UNDERFLOW),
    exception --> (size(s) == @size(s)),

    forall (int i: int_range(0, @size(s) - 2)) {
      elementAt(s, i) == elementAt(prev, i) },

    normally {
      size(s) == @size(s) - 1 }
  };
};
```

Figure 2: *stack.adl*: ADL specification of the stack ADT

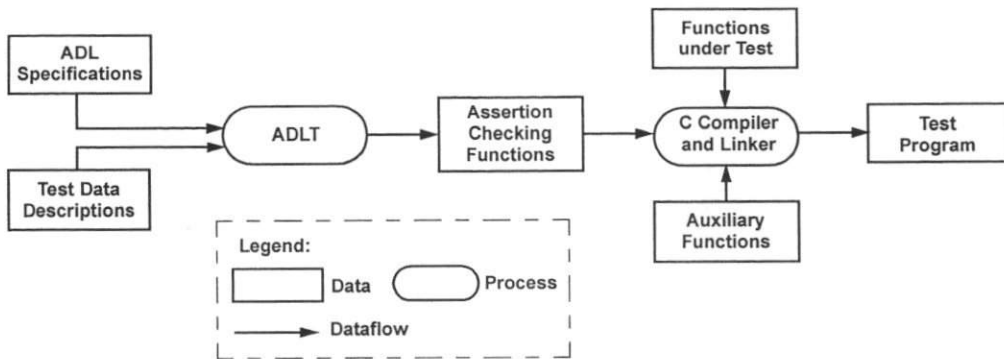

Figure 3: Dataflow of ADLT's compile stage

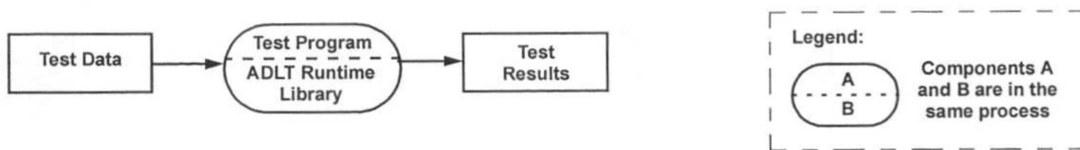

Figure 4: Dataflow of ADLT's run stage

tions, the functions under test, and all their auxiliary functions are compiled and linked into a test program. Figure 3 shows the dataflow of ADLT in the compile stage.

In the run stage, the user starts the test program to test the intended software. The test program utilizes the functionality provided by the ADLT Runtime Library to check assertions and report results to the tester. Figure 4 shows the dataflow in the run stage.

3 ADLscope Design and Implementation

We have developed specification-based coverage criteria for ADL. ADLscope is the collection of components that we have integrated with ADLT to support specification coverage measurement. This section describes the design and implementation of ADLscope[1].

3.1 Integration with ADLT

ADLscope is tightly integrated with ADLT. Figure 5 shows the dataflow of the integration of ADLscope and ADLT.

In the compile stage, ADLscope generates a set of *Coverage Checking Functions* which are C functions that evaluate coverage conditions in the run stage. The Coverage Checking Functions are linked into the test program. ADLscope also generates a *Static Coverage Data* file and stores it in the ADLscope database. The Static Coverage Data files store information about each coverage condition such as where it is derived in the source specification and to which function it belongs. It also contains the name of the ADL specification, the compilation time, and the path of

1. The ADLscope release (including ADLT) can be downloaded from www.ics.uci.edu/~softtest/adlscope. The release contains the source files as well as executables for Solaris and Linux. The original ADLT system and documentation can be downloaded from ftp://ftp.uu.net/vendor/adl.

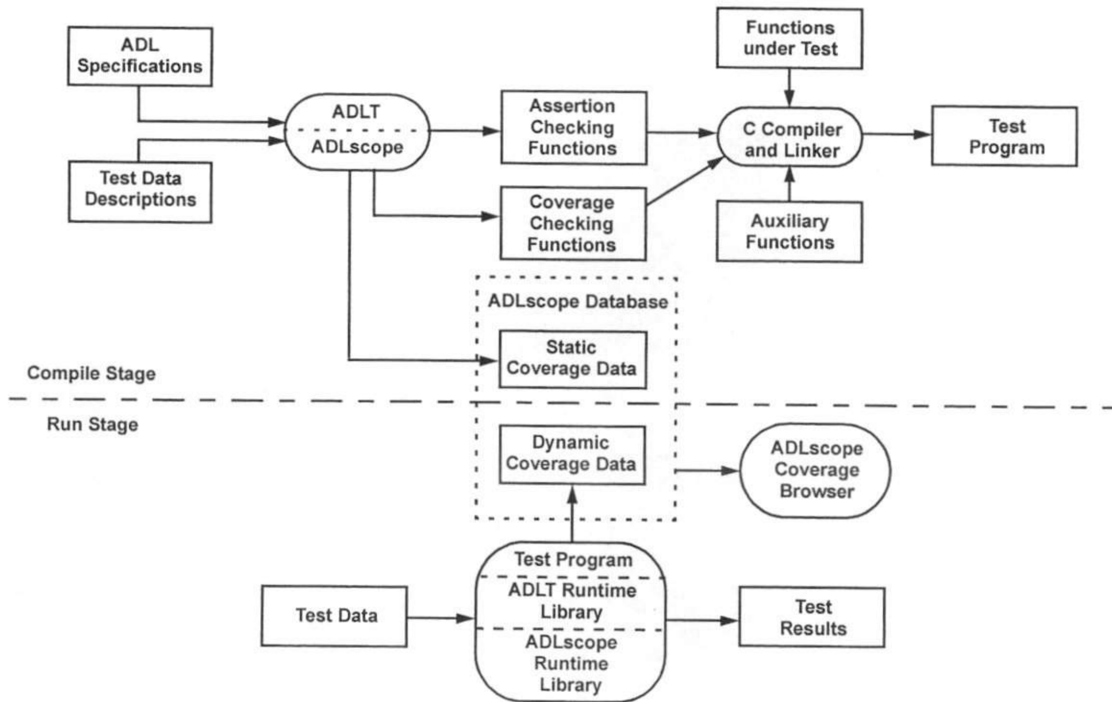

Figure 5: Dataflow of ADLscope

the ADL specification file. This information is later used by the ADLscope Coverage Browser or a report generator to display coverage results to the tester.

In the run stage, the test program calls the Coverage Checking Functions to determine which coverage conditions have been satisfied and utilizes the ADLscope Runtime Library to store this information into a *Dynamic Coverage Data* file. Each Dynamic Coverage Data File contains a count for each coverage condition. The count is the number of times a condition is covered. Each Dynamic Coverage Data file also contains the run time and the name of the test.

ADLscope requires no additional work from the user. The user needs only supply one additional flag `-cov` on the ADLT command line to enable coverage measurement.

The bulk of ADLscope (with the exception of the database API and the Coverage Browser, which are described in the following subsections) is developed in C and C++.

3.2 Database API

We have developed a Java API for reading both Static Coverage Data and Dynamic Coverage Data from the ADLscope database. This API is used by the ADLscope Coverage Browser to display coverage information through a graphical user interface. It can also be used by users to create customized coverage reports.

3.3 Coverage Browser

The ADLscope Coverage Browser is a hyper-linked browser. By clicking on a function or a coverage condition, the browser would display coverage information associated with that function or

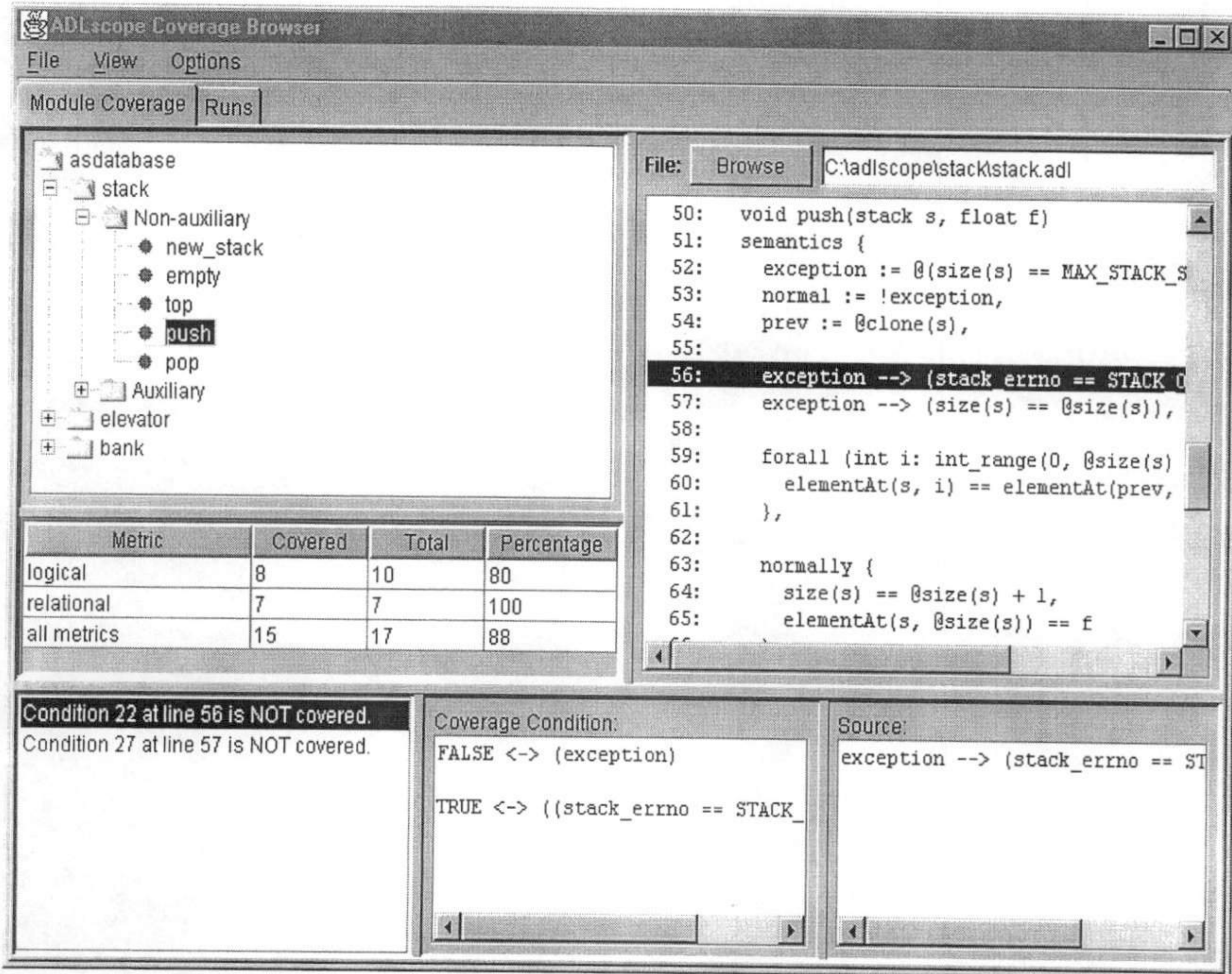

Figure 6: Screenshot of the ADLscope Coverage Browser

coverage condition as well as its location in the specification. This allows the tester to quickly focus on the part of the specification that requires more testing.

The ADLscope Coverage Browser is implemented in Java. Most of the GUI components are based on Java Foundation Classes. Figure 6 shows a screenshot of the ADLscope Coverage Browser.

4 Specification-based Coverage Metrics

This section describes the specification-based coverage metrics used by ADLscope. These coverage metrics are based on the expressions in the ADL language. Some of our metrics are based on existing test selection strategies. The multiple condition strategy [Mye79] is used on logical expressions, and we apply weak mutation testing [How82] to relational expressions. We have also developed strategies for constructs that are specific to ADL.

The strategies used in ADLscope are all linear techniques, that is, the number of required coverage conditions grows linearly with the size of the specification.

In the rest of this section, we describe the coverage conditions for ADL specifications. In Subsec-

tion 4.1, we first describe a technique that we use to remove some of the logically impossible coverage conditions. From Subsection 4.2 to Subsection 4.10, we present the coverage conditions required for each type of ADL expressions.

4.1 Constraint Propagation

Consider the following scenario: in specifying the *push* function, we may want to say that if the size of the stack has not changed, then there must have been a stack overflow error:

```
stack_errno == ((size(s) == @size(s))? STACK_OVERFLOW: 0)
```

Here it makes sense to require two cases: (1) the size of the stack changes after the *push* is called, and (2) the size of the stack does not change.

Now consider another example, in the stack ADL specification, the function *top* contains the following assertion:

```
size(s) == @size(s)
```

This assertion states that the size of the stack should not change. It would certainly make sense to require the case where the size of the stack indeed does not change. As a matter of fact, any correct implementation of *top* would certainly not change the size of the stack. However, it does not make sense to require a test case where the size does change since no correct implementation would ever satisfy this requirement.

When determining coverage conditions, ADLscope takes into account whether an expression's value is constrained to true or false by the enclosing assertion, and generates coverage conditions that are logically consistent with the constraint. The constraint also is propagated to each expression's sub-expressions so that the coverage conditions derived from sub-expressions are also logically consistent with the specification.

4.2 Logical Expressions

ADL has two kinds of logical expressions:[2]

a `&&` b

and

a `||` b

where a and b are sub-expressions. "`&&`" is the short-circuit AND operator. For each unconstrained short-circuit AND expression, ADLscope requires the following coverage conditions:

(1) {a `== true`, b `== true`}
(2) {a `== false`}
(3) {a `== true`, b `== false`}.

If a short-circuit AND expression is constrained to true, only (1) is required. If an AND expression is constrained to false, only (2) and (3) are required.

"`||`" is the short-circuit OR operator. For each unconstrained short-circuit OR expression, ADL-

2. There are no non-short-circuit AND and OR operators in ADL.

scope requires three conditions:

(1) {a == true}
(2) {a == false, b == true}
(3) {a == false, b == false}.

If a short-circuit OR expression is constrained to true, only (1) and (2) are required. If it is constrained to false, only (3) is required.

4.3 Conditional Expressions

There are two kinds of conditional expressions in ADL. The first kind is similar to conditional expressions in C and has the form:

a ? b : c.

For this kind of conditional expressions, ADLscope requires a to be true for some test and a to be false for some other test, regardless of the constraint on the expression:

(1) {a == true}
(2) {a == false}.

The second kind of conditional expressions has the following form:

if (a) a'
else if (b_1) b_1'
else if (b_2) b_2'
...
else if (b_n) b_n'
else c'.

For this, ADLscope requires (n+2) coverage conditions:

(1) {a == true}
(2) {a == false, b_1 == true}
(3) {a == false, b_1 == false, b_2 == true}
...
(n+1) {a == false, b_1 == false, b_2 == false, ..., b_n == true}
(n+2) {a == false, b_1 == false, $b2$ == false, ..., b_n == false}.

4.4 Implication Expressions

ADL has four kinds of implication expressions:

`a --> b`,
`a <-- b`,
`a <-> b`,

and

`a <:> b`.

"`-->`" is the standard logical implication operator. For each unconstrained implication expression,

ADLscope requires:

(1) `{a == false, b == false}`
(2) `{a == false, b == true}`
(3) `{a == true, b == true}`
(4) `{a == true, b == false}`.

If an implication expression is constrained to true, ADLscope requires (1), (2), and (3). If it is constrained to false, ADLscope requires only (4).

"`<--`" is the reverse implication operator. For each unconstrained reverse implication expression, ADLscope requires:

(1) `{a == false, b == false}`
(2) `{a == true, b == false}`
(3) `{a == true, b == true}`
(4) `{a == false, b == true}`.

If a reverse implication expression is constrained to true, ADLscope requires (1), (2), and (3). If it is constrained to false, ADLscope requires only (4).

"`<->`" is the logical equivalence operator. For each unconstrained equivalence expression, ADL-scope requires:

(1) `{a == false, b == false}`
(2) `{a == true, b == true}`
(3) `{a == false, b == true}`
(4) `{a == true, b == false}`.

If an equivalence expression is constrained to true, only (1) and (2) are required. If it is constrained to false, only (3) and (4) are required.

"`<:>`" is called the exception operator is used to describe an exceptional outcome of the function. *a* `<:>` *b* is equivalent to

```
(a --> exception) && ((exception && b) --> a).
```

For each unconstrained exception expression, ADLscope requires the following conditions:

(1) `{a == false, b == false, exception == false}`
(2) `{a == false, b == false, exception == true}`
(3) `{a == false, b == true, exception == false}`
(4) `{a == true, b == true, exception == true}`
(5) `{a == false, b == true, exception == true}`
(6) `{a == true, b == false, exception == false}`
(7) `{a == true, b == false, exception == true}`
(8) `{a == true, b == true, exception == false}`.

If an exception expression is constrained to true, ADLscope requires (1), (2), (3), and (4). If it is constrained to false, ADLscope requires (5), (6), (7), and (8).

4.5 Relational Expressions

ADL has the following relational expressions:

$a > b$,
$a < b$,
$a >= b$,

and

$a <= b$.

For each unconstrained GT (“>”) expression, ADLscope requires the following conditions (the conditions listed in parentheses below are for boundary testing of integer operands):

(1) $\{a > b\}$ (or $\{a == b + 1\}$)
(2) $\{a <= b\}$ (or $\{a == b\}$).

If a GT expression is constrained to true, only (1) is required. If it is constrained to false, only (2) is required.

For each unconstrained LT (“<“) expression, ADLscope requires

(1) $\{a < b\}$ (or $\{a == b - 1\}$)
(2) $\{a >= b\}$ (or $\{a == b\}$).

If an LT expression is constrained to true, only (1) is required. If it is constrained to false, only (2) is required.

For each unconstrained GE (“>=”) expression, ADLscope requires

(1) $\{a >= b\}$ (or $\{a == b\}$)
(2) $\{a < b\}$ (or $\{a == b - 1\}$).

If a GE expression is constrained to true, only (1) is required. If it is constrained to false, only (2) is required.

For each unconstrained LE (“<=”) expression, ADLscope requires

(1) $\{a <= b\}$ (or $\{a == b\}$)
(2) $\{a > b\}$ (or $\{a == b + 1\}$).

If an LE expression is constrained to true, only (1) is required. If it is constrained to false, only (2) is required.

4.6 Equality Expressions

ADL has two kinds of equality expressions:

$a == b$

and

$a != b$.

For each unconstrained EQ (“==”) expression, ADLscope requires

(1) `{`a `==` b`}`
(2) `{`a `!=` b`}`.

If an EQ expression is constrained to true, only (1) is required. If it is constrained to false, only (2) is required.

For each unconstrained NE ("`!=`") expression, ADLscope requires

(1) `{`a `==` b`}`
(2) `{`a `!=` b`}`.

If an NE expression is constrained to true, only (2) is required. If it is constrained to false, only (1) is required.

4.7 Normally Expressions

Normally expressions have the form

`normally` e

which is equivalent to

`normal?` e`: true.`

For each normally expression, ADLscope requires two coverage conditions:

(1) `{normal == true}`
(2) `{normal == false}`.

4.8 Group Expressions

Group expressions have the following form:

`{`e_1`,` e_2`, ...,` e_n`}`.

A group expression is true if all subexpressions e_1, e_2, ..., e_n evaluate to true.

For each unconstrained group expression, ADLscope requires:

(1) `{`e_1 `== true,` e_2 `== true, ...,` e_n `== true}`
(2) `{`e_1 `== false}`
(3) `{`e_2 `== false}`
...
(n+1) `{`e_n `== false}`.

If a group expression is constrained to true, only (1) is required. If it is constrained to false, ADLscope requires (2), (3), ..., (n+1).

4.9 Unchanged Expression

Unchanged expressions have the following form

`unchanged(`e_1`,` e_2`, ...,` e_n`)`

which is equivalent to the following group expression:

`{`e_1 `==` `@`e_1`,` e_2 `==` `@`e_2`, ...,` e_n `==` `@`e_n`}`.

For each unconstrained unchanged expression, ADLscope requires

(1) `{`e_1 `==` `@`e_1`,` e_2 `==` `@`e_2`, ...,` e_n `==` `@`e_n`}`

(2) `{`e_1 `!=` `@`e_1`}`

(3) `{`e_2 `!=` `@`e_2`}`

...

(n+1) `{`e_n `!=` `@`e_n`}`.

If an unchanged expression is constrained to true, only (1) is required. If it is constrained to false, ADLscope requires (2), (3), ..., and (n+1).

4.10 Quantified Expressions

ADL has both universally quantified expressions and existentially quantified expressions:

`forall (`*domain*`) {`e_1`,` e_2`, ...,` e_n`}`

and

`exists (`*domain*`) {`e_1`,` e_2`, ...,` e_n`}`.

domain is a function that returns an enumeration of values that the target expressions should hold. For each unconstrained universally quantified expression, ADLscope requires

(1) `{true == (forall (`*domain*`) {`e_1`}), ..., true == (forall (`*domain*`) {`e_n`})}`

(2) `{false == (forall (`*domain*`) {`e_1`})}`

(3) `{false == (forall (`*domain*`) {`e_2`})}`

...

(n+1) `{false == (forall (`*domain*`) {`e_n`})}`.

If the universally quantified expression is constrained to true, ADLscope requires only (1). If it is constrained to false, ADLscope requires (2), (3), ..., (n+1).

For each unconstrained existentially quantified expressions, ADLscope requires

(1) `{true == (exists (`*domain*`) {`e_1`}), ..., true == (exists (`*domain*`) {`e_n`})}`

(2) `{false == (exists (`*domain*`) {`e_1`})}`

(3) `{false == (exists (`*domain*`) {`e_2`})}`

...

(n+1) `{false == (exists (`*domain*`) {`e_n`})}`.

If the existentially quantified expression is constrained to true, ADLscope requires only (1). If it is constrained to false, ADLscope requires (2), (3), ..., (n+1).

4.11 Stack Example

Table 1 shows the coverage conditions that are required to achieve 100% specification coverage

for the *push* function of the ADL stack specification in Figure 2.

`{false == exception, false == (stack_errno == STACK_OVERFLOW)}`
`{false == exception, true == (stack_errno == STACK_OVERFLOW)}`
`{true == exception, true == (stack_errno == STACK_OVERFLOW)}`
`{stack_errno == STACK_OVERFLOW}`
`{stack_errno != STACK_OVERFLOW}`
`{false == exception, false == (size(s) == @size(s))}`
`{false == exception, true == (size(s) == @(size(s))}`
`{true == exception, true == (size(s) == @size(s))}`
`{size(s) == @size(s)}`
`{size(s) != @size(s)}`
`{true == forall(int i: int_range(0, @size(s) - 1)){elementAt(s, i) == elementAt(prev, i)}}`
`{forall(int i: int_range(0, @size(s) - 1)){elementAt(s, i) == elementAt(prev, i)}}`
`{true == normal}`
`{false == normal}`
`{true == (size(s) == @(size(s) + 1)), true == (elementAt(s, @size(s)) == f)}`
`{size(s) == @size(s) + 1}`
`{elementAt(s, @size(s)) == f}`

Table 1: Coverage Conditions for the *push* function

5 Related Work

The coverage metrics used in ADL are partially based on several existing, well-known test selection strategies. The multiple condition strategies [Mye79] is intended to exercise logical expressions in programs. We use the multiple condition strategy on ADL's short-circuit AND and OR expressions. The meaningful impact strategy [Fos84][Tai90][WGS94] is another strategy for selecting test cases from logical expressions. It is not currently used by ADLscope, and we are considering adapting this strategy for ADL and implementing it in ADLscope. For relational expressions, ADLscope uses weak mutation testing [How82].

Several papers have focused on formal specification-based test selection. Richardson and Clarke [RC85] propose using symbolic execution techniques to derive test cases from the specification. Richardson, O'Malley, and Tittle [ROT89] discuss several general approaches to specification-based test selection. ADLscope can be classified as a Specification/Error-based Testing technique under their categorization. Stocks and Carrington [SC93] advocate the use of a formal framework and a formal specification language Z for selecting tests. The framework itself, however, does not provide any automated techniques. Chang, Richardson, and Sankar [CRS96] focus on automated test selection based on ADL. We have realized that automated test selection requires more work from the tester and is more difficult to use and to implement. The coverage metrics used in ADL-scope are more intuitive and easier to use than an automated test selection tool.

6 Conclusion and Future Work

We have developed a fully automated specification-based coverage tool called ADLscope for testing APIs. In the past two decades, many papers have focused on selecting test cases from axiomatic specifications. However, we are not aware of any automated axiomatic specification-based

test selection tools. The main reasons that we identify for the lack of such tools are that:

- It is non-trivial to map entities from the specification domain to the implementation domain, and vice versa. This step usually cannot be automated and requires the user to manually maintain a list of mappings.
- Many test selection techniques require complex symbolic evaluation techniques that are difficult to implement and use.

We solve the first problem by using a specification language that shares the same type system as the implementation language and embedding ADLscope within an existing test execution tool. ADLscope requires no additional input from the user and changes the user's current process only slightly.

We address the second problem by using a simple and intuitive set of specification-based coverage metrics. Comparing ADLscope to symbolic evaluation techniques is analogous to comparing branch coverage with path coverage. While path coverage provides more thorough testing, it is computationally expensive and often not achievable in practice because of resource constraints. On the other hand, even though branch coverage is not as effective, it is usually more than adequate for most applications. Branch coverage, in general, is easy to learn and use.

We are about to embark on an empirical study to determine the effectiveness of ADLscope. The two questions that we intend to address are:

- Do ADLscope's specification-based coverage metrics provide a good estimate of the quality of the test data? As opposed to test selection tools where the objective is to detect errors, the main objective of any test coverage tool is to provide an estimate of the thoroughness of the test data.
- Can ADLscope detect errors? ADLscope does not detect errors directly. It detects errors indirectly by forcing the tester to cover the specification.

We plan to design experiments to address these questions.

Bibliography

[CRS96] J. Chang, D. J. Richardson, and S. Sankar. Structural specification-based testing with ADL. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA'96)*, pp. 62-70, San Diego, California, January 1996. ACM Press.

[Fos84] K. A. Foster. Sensitive test data for logic expressions. *ACM SIGSOFT Software Engineering Notes*, vol. 9, no. 2, pp. 120-126, April 1984.

[How82] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371-379, July 1982.

[Mye79] G. J. Myers. *The art of software testing*. New York: John Wiley and Sons, 1979.

[RC85] D. J. Richardson and L. A. Clarke. Partition analysis: a method combining testing and verification. *IEEE Transactions on Software Engineering*, SE-11(12):1477-1490, December 1985.

[ROT89] D. J. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based test-

ing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pp. 86-96, Key West, Florida, December 1989.

[SH94] S. Sankar and R. Hayes. Specifying and testing software components using ADL. Technical Report SMLI TR-94-23, Sun Microsystems Laboratories, Inc., Mountain View, California, April 1994.

[SC93] P. Stocks and D. Carrington. Test template framework: a specification-based testing case study. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA'93)*, pp. 11-18, Cambridge, Massachusetts, June 1993.

[Sun96] Sun Microsystems Inc., *ADL Language Reference Manual, Release 1.1*, December 1996.

[Tai90] K. C. Tai. Condition-based software testing strategies. In *Proceedings of the 14th Annual International Computer Software and Applications Conference*, 1990.

[WGS94] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, SE-20(5):353-363, May 1994.