Foundational Spec-Based in Testing Tools and Techniques

I. **The tools and techniques in these notes are twenty years old.**

A. They are never-the-less quite valid today.

B. Production-quality tools for specification-based test generation lag very far behind the research.

1. There are a number or reasons for this, relating to economics and lack of wide-spread acceptance by software developers.
2. When the proper tools do finally catch on, there may well be a breakthrough in this area.
3. In the meantime, the promise of specification-based test case generation is not yet fulfilled.

Specification-Based Testing with ADL

II. **Introduction to ADL**

A. ADL is a C-based `Assertion Definition Language`.

B. ADL specs are predicative (1st order) just like the other model-based spec languages including JML

C. Two forms of *test conditions* can be derived from ADL specs:

1. *Call-state test conditions* are derived directly from *preconditions*
2. *return-state test conditions* are derived directly from *preconditions*

III. **Some terminology**

A. *ADLT* -- The ADL Translator that provides automated support for testing C programs

B. *Test driver* -- C program generated by ADLT given ADL program specification and ADL test data description

C. *SCT* -- specification coverage tool that derives test conditions from an ADL specification

D. *Coverage condition functions* -- C functions that determine whether derived test conditions are satisfied by some data.

E. *Function under test* -- the C function for which tests are generated and executed.

F. *Test program executable* -- the compiled and linked set of test driver, coverage functions, and function under test.

G. See Figure 1 on page 63 of the ADL paper.

IV. **Summary of ADL**

A. ADL specification consists of *modules* containing *constituents*

B. There are three types of constituents:

1. *types*
2. *objects*
3. *functions*

C. Functions contain *semantic descriptions* of two forms:

1. *Bindings* (aka, macros or "let" expressions)
2. *Assertions* (aka, pre and postconditions)

D. Two built-in bindings are *`exception`* and *`normal`*

1. The expression bound to `exception` defines the condition(s) under which the function fails

2. The expression bound to `normal` defines the condition(s) under which the function succeeds.
3. These are effectively the same as the "normal behavior" and "exceptional behavior" clauses of a JML spec.

E. Assertion expressions refer to two states:
1. The *call state* (expressions surrounded by the "@" operator)
2. The *return state*.

V. ADL compared to Java-based specification languages, such as JML

A. Here's a comparison table

ADL Construct	Java/JML Construct
module	.h file
type constituent	class (type) definition
object constituent	var or const declaration
function constituent	method declaration
binding	not available in JML; can be done with Java `const` declaration
assertion	and'd clause in postcondition
exception precondition	converted precondition (see below)
@ (call-state operator)	\old
--> *(implication)*	==>

B. On converted preconditions (in C/C++ notation)
1. Unconverted:

```
void Append(List* l, Elem* e)
/*
 * pre: !(l->Find(e))
 * post: l->Find(e)
 */
```

2. Converted

```
int Append(List* l, Elem* e)
/*
 * pre:
 * post: !(l->Find(e))
 *       ? (return == -1)
 *       : (l->Find(e) && (return == 1))
 */
```

C. The "@" notation in ADL
1. In ADL, the "@" operator surrounds an entire expression to indicate that it should be evaluated in the calling state, i.e, with input values of the variables in the expression;
2. In many publications on predicative specification, this effect is accomplished by using the prime ("'") notation, where unprimed variables have input values and primed variables have output values.
3. In JML, the `\old` operator is equivalent to ADL `@`

VI. Details of ADL test conditions

A. The main point of ADL tool is the automatic derivation of test conditions from ADL specifications.

B. A *call-state test condition* is evaluable in calling environment
   1. These conditions are surrounded by the "@" operator.
   2. They are expressions containing input-only values (no pointers, arrays, function calls or global vars).

C. ADL uses condition generation rules from the following (selectable?) strategies:
   1. Multi-condition (what's shown in ADL paper)
   2. Meaningful impact (an improved test selection strategy)
   3. Boundary-value (shown in ADL paper)
   4. Domain-specific (planned research that was never fully implemented in ADL)

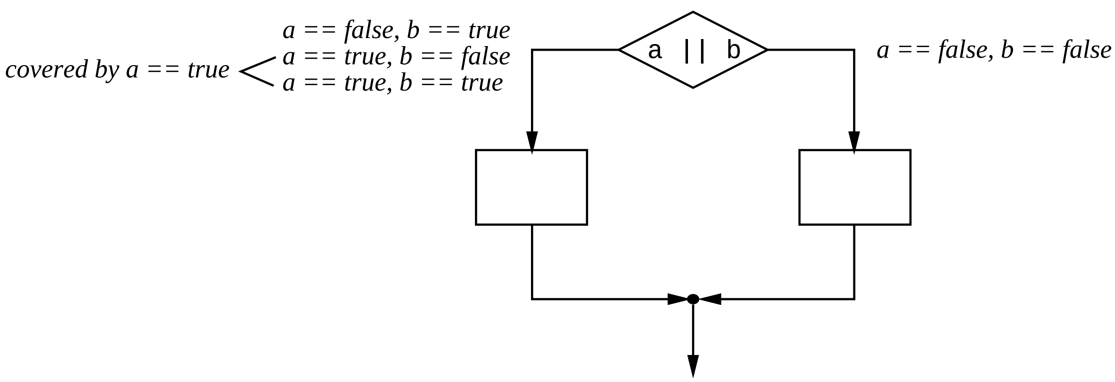
VII. **Details of multi-condition test condition generation**

A. Test conditions must be generated that exercise both branches of conditional tests, for all truth values of the conditional expressions.

B. Consider the conditional expression *a || b*.
   1. The truth table for this expression is

a	**b**	**a || b**
0	0	0
0	1	1
1	0	1
1	1	1

   2. An annotated flow graph involving this conditional is the following:



   3. Based on the truth table and flow graph, the multi-condition test cases for *a || b* are: {*a==false*, *b==true*}, {*a==true*}, and {*a==false*, *b==false*}.
   4. This information can be combined in a *truth and condition table*:

a	**b**	**a || b**	**test condition**
0	0	0	a==0, b==0
0	1	1	a==0, b==1
1	0	1	a==1
1	1	1	*covered by a==1*

C. Note that in the ADL paper, {*a==0*} is denoted {*!a*} and {*a==1*} is denoted {*a*} .

D. By similar analysis, the truth and condition tables for *a -> b* and *a && b* as follows:

a	**b**	**a -> b**	**test condition**
0	0	1	a==0
0	1	1	*covered by a==0*
1	0	0	a==1, b==0
1	1	1	a==1, b==1

a	**b**	**a && b**	**test condition**
0	0	0	a==0
0	1	0	*covered by a==0*
1	0	0	a==1, b==0
1	1	1	a==1, b==1

E. Note that in particular test generation contexts, we will constrain the value of expressions to be true or false.

1. In such contexts, only the conditions applicable to the constrained outcome must be generated.
2. E.g., if we constrained the value of *a* || *b* to be true, we would only need to generate only the two conditions {*a==0, b==1*} and {*a==1*}.
3. If *a* || *b* were constrained to be false, then only the single condition {*a==0, b==0*} would be generated.

VIII. **Details of boundary-value condition generation**

A. Consider the expression *(x < 0)* || *(x > 10)*.

B. Here is its truth and condition table

x < 0	x > 10	(x < 0) || (x > 10)	test condition	test data
0	0	0	!(x < 0) && !(x > 10)	x = 5
0	1	1	x > 10	x = 11
1	0	1	x < 0	x = -1
1	1	---- impossible ----		

C. In this example, the boundary value strategy picked a value just below the constant operand of the relational expression, and in the middle of the range expression.

IX. **Details of the ADL approach**

A. Parse the specs

B. Define a boolean-valued inherited attribute on each node that constrains the value of the subexpression below to be true or false, per the requirements of the test-condition generation strategy.

C. Traverse the parse trees to generate test conditions

1. Call-state conditions are generated only for subtrees that contain call-state evaluable expressions.
2. Return-state conditions are generated for all subtrees.

D. Consider examples on page 66 of ADL paper.

X. **Detailed walk-through of the ADL paper example**

A. Peruse page 3.

1. Note use of disjunctive normal forms (i.e., boolean expression clauses are or'd together).
2. E.g., second disjoin of then-clause of Assertion 3.

B. See parse tree notes on paper.

C. After parse subexpr parse trees are gen'd

1. Combine the precondition exprs with each of the 3 multi-condition-generated post-condition exprs, the obtain 3 basic test conds for assertion 3.

XI. **Some comments on the ADL methodology**

A. I think that pre- and post-conds are a little more intuitive to deal with than the "calling" an "returning" contexts ideas; the "@" notation seems particularly confusing compared to the more traditional "'" notation.

B. By defining preconditions explicitly, the potentially confusing notion of "call-state evaluable" goes away, since the set preconditions is exactly the set of call-state evaluable conditions.

XII. **Extending ADL to work with object-oriented constructs and quantifier logic**

A. This is the work of on-going research

B. It involves additions to the ADL C grammar, and updates to the test case generation algorithm.

The Meaningful Impact Strategy for Automatically Generating Test Data from a Boolean Specification

XIII. **Introduction**

A. Motivation for and intuition behind the strategy

1. In the multi-condition testing strategy employed by ADL and other comparable tools, the number of test cases is exponential on the number of input/output variables.
2. Specifically, for a function with *n* variables, there are 2^n test cases in an exhaustive specification-based test plan.
3. The point of the meaningful impact strategy is to reduce the number of test cases by considering the impact of specific variables in specific test cases
4. To be precise, a boolean term in a test case formula is said to have *meaningful impact* if changing the truth value of the term changes the value of the formula.

B. Weyuker et al. have built a tool that like ADL, automatically generates test cases from boolean specifications.

1. In their case, they employ the meaningful impact strategy rather than the multi-condition strategy.
2. They test the effectiveness of their approach, and show empirically good results.

C. How they demonstrate their results

1. They generate test data for the well-known, real-world specification of TCAS (the Traffic control and Collision Avoidance System).
2. They compare the size of their test plans to the size of exhaustive multi-condition test plans for the same spec
3. The evaluate the effectiveness of their specification using *mutation testing*.
   a. A program under test is first tested as written.
   b. Then the program is *mutated* by systematically introducing syntax errors that should change the output of the program.

c. If the generated test cases can distinguish the mutant output from the original output, then the test cases are successful.

d. Overall, the meaningful result strategy showed very favorable results when subjected to mutation analysis.

XIV. **Definitions**

A. Notation

1. Infix '+' means boolean **or**, e.g., $a + b$
2. *term concatenation* means **and**, e.g., ab
3. *Overbar* means not, e.g., $\overline{a}$

B. Definition: *Disjunctive normal form*

1. All terms of boolean expression and or'd together.
2. E.g., for the formula $a(b\overline{c}+d)$, the disjunctive normal form is $ab\overline{c}+a\overline{d}$.

C. Definition: *Canonical disjunctive normal form*

1. Each term in a disjunctive normal form formula contains all variables.
2. E.g., for the preceding formula, the canonical disjunctive normal form is

$$ab\overline{c}d + ab\overline{cd} + abc\overline{d} + a\overline{b}c\overline{d} + a\overline{bcd}$$

D. Definition: *Meaningful impact*

1. A literal in a boolean formula have meaningful impact if, everything else being the same, a different truth value assignment to that literal will result in a different value for the formula.
2. E.g., consider the formula *(ab + ac)* and the test case {a=0, b=1, c=0}.

   a. This test case causes the formula to evaluate to 0.

   b. Question: Does the value assigned to the first occurrence of *a*, i.e., a_1, have meaningful impact on the value 0 for the test case?

   c. Answer: Yes, since changing the assignment of a_1 to 1 will change the value of the formula for the test case to 1.

   d. On the other hand, the test case does not demonstrate that *b*, a_2, or *c* have meaningful input on the formula value of 0.

E. Definition: *True points*

1. The set of test cases that cause a formula to be true are called the *true points*.
2. The subset of true points that demonstrate meaningful impact are called *unique true points*.
3. Complementary definitions exist for *false points* and *unique false points*.

XV. **The basic strategy**

A. The intuition here is that if a term has no meaningful impact on a pre or postcondition, then it is likely to have no meaningful impact on the outcome of the function under test.

B. In circuit testing, the "stack-at-1" testing strategy is essentially the same as meaningful impact is here.

1. In hardware, there are theoretical and empirical data to validate the assumption that stuck-at assumption is reasonable.
2. Part of the contribution of this paper are empirical data that show this for software.

C. As a concrete example, of the basic strategy, see Table 1 on page 356 of the paper.

1. Note that this table is non-deterministic for some test cases, i.e., rows 1-4, 9, and 10.
2. The paper suggests strategies for simulating determinism in Section V.
3. Foster suggests a fully deterministic strategy that is not optimal.

D. Another way to eliminate the non-determinism is to convert all testing formulae to canonical DJF, as shown in Table 2 on page 357.

E. The problem with this is that it increases the number of test cases, without always obtaining more coverage.

XVI. **Assessment of the basic strategy**

A. A number of incorrect implementations are *guaranteed* to be detected by the meaningful impact strategy.

B. A number of incorrect implementations are *guaranteed not* to be detected by the meaningful impact strategy.

C. A number of incorrect implementations are *may or may not* be detected by the meaningful impact strategy.

D. Intuitively, meaningful impact does the following:

1. Divide the test data domain into subdomains that distinguish between meaningful and not meaningful data.
2. If an implementation fails for *all* of the points in a particular subdomain, then the failure will be detected.
3. If an implementation fails for *all* of the points in a particular subdomain, then the failure will be detected.
4. If an implementation fails for *some* of the points in a particular subdomain, and those points do not have meaningful impact, then the failure will go undetected.

E. The empirical evaluations in Section VI of the paper reveal that for a real-world specification, the number of incorrect

XVII. **Enhancing the basic strategy**

A. A family of algorithms has been devised based on the basic strategy

B. They differ by the strategies used to select test points where the basic strategy is non-deterministic.

XVIII. **Empirical results**

A. Specifications taken from TCAS II (Traffic alert and Collision Avoidance System II).

1. Thirteen of the larger specs were chosen, ranging in size from 5 to 14 variables.
2. Specs altered to account for variable dependencies that would cause infeasible test conditions.
3. See Figure 2 and Table III on page 360.

B. An assessment in terms of comparison with exhaustive multi-condition test case generation is quite favorable (this is Table III).

C. An assessment in terms of a thorough mutation analysis is also quite favorable.

1. See Tables IV thorough XII on pages 361 and 362.
2. Tables IV through VII show averages, including comparison to random and exhaustive testing strategies.
   a. The worst mutation score is 92.7 (out of 100).
   b. The averages are 97.9 - 99.7.
3. Tables VIII through XII show individual analysis for each of the following mutation operators
   a. *Variable Negation Fault:* Replace boolean variable by its negation.
   b. *Expression Negation Fault:* Replace boolean expression by its negation.
   c. *Variable Reference Fault:* Replace one occurrence of a variable by another.
   d. *Operator Reference Fault:* Replace one boolean operator with another.
   e. *Associative Shift Fault:* Change the associativity of terms in an expression.